QLectives – Socially Intelligent Systems for Quality
Project no. 231200

Instrument: Large-scale integrating project (IP)
Programme: FP7-ICT

Deliverable D.4.1.1

QLectives Platform v1 - Short report

Submission date: 2010-03-01

Start date of project: 2009-03-01 Duration: 48 months

Organisation name of lead contractor for this deliverable: TUDelft

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)		
Dissemination level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidental, only for members of the consortium (including the Commission Services)	

Document information

1.1 Author(s)

Author	Organisation	E-mail
Nazareno Andrade	TU Delft	N.FerreiradeAndrade@tudelft.nl
Tamás Vinkó	TU Delft	T.Vinko@tudelft.nl
Johan Pouwelse	TU Delft	J.A.Pouwelse@tudelft.nl

1.2 Other contributors

Name	Organisation	E-mail
Gertjan Halkes	TU Delft	G.P.Halkes@tudelft.nl
Alain van den Berg	TU Delft	A.M.Vandenberg@student.tudelft.nl

1.3 Document history

Version#	Date	Change
V0.1	4 January, 2010	Starting version, template
V0.2	1 February, 2010	Complete first draft
V0.3	11 Feburary, 2010	Complete revised version
V1.0	01 March, 2010	Approved version to be submitted to EU

1.4 Document data

Keywords	Peer-to-Peer, BitTorrent, Techno-Social Networking, NAT puncturing, P2P-Widgets
Editors address data	N.FerreiradeAndrade@tudelft.nl, T.Vinko@tudelft.nl
Delivery date	01 March, 2010

1.5 Distribution list

Date	Issue	E-mail
	Consortium members	QLECTIVES@list.surrey.ac.uk
	Project officer	Jose.FERNANDEZ-VILLACANAS@ec.europa.eu
	EC archive	INFSO-ICT-231200@ec.europa.eu

QLectives Consortium

This document is part of a research project funded by the ICT Programme of the Commission of the European Communities as grant number ICT-2009-231200.

University of Surrey (Coordinator)
Department of Sociology/Centre for Research in Social Simulation
Guildford GU2 7XH
Surrey
United Kingdom
Contact person: Prof. Nigel Gilbert
E-mail: n.gilbert@surrey.ac.uk

University of Fribourg
Department of Physics
Fribourg 1700
Switzerland
Contact person: Prof. Yi-Cheng Zhang
E-mail: yi-cheng.zhang@unifr.ch

Technical University of Delft
Department of Software Technology
Delft, 2628 CN
Netherlands
Contact Person: Dr Johan Pouwelse
E-mail: j.a.pouwelse@tudelft.nl

University of Warsaw
Faculty of Psychology
Warsaw 00927
Poland
Contact Person: Prof. Andrzej Nowak
E-mail: nowak@fau.edu

ETH Zurich
Chair of Sociology, in particular Modelling and Simulation
Zurich, CH-8092
Switzerland
Contact person: Prof. Dirk Helbing
E-mail: dhelbing@ethz.ch

Centre National de la Recherche Scientifique, CNRS
Paris 75006,
France
Contact person: Dr. Camille ROTH
E-mail: camille.roth@polytechnique.edu

University of Szeged
MTA-SZTE Research Group on Artificial Intelligence
Szeged 6720, Hungary
Contact person: Dr Mark Jelasity
E-mail: jelasity@inf.u-szeged.hu

Institut für Rundfunktechnik GmbH
Munich 80939
Germany
Contact person: Dr. Christoph Dosch
E-mail: dosch@irt.de

QLectives introduction

QLectives is a project bringing together top social modelers, peer-to-peer engineers and physicists to design and deploy next generation self-organising socially intelligent information systems. The project aims to combine three recent trends within information systems:

- **Social networks** - in which people link to others over the Internet to gain value and facilitate collaboration
- **Peer production** - in which people collectively produce informational products and experiences without traditional hierarchies or market incentives
- **Peer-to-Peer systems** - in which software clients running on user machines distribute media and other information without a central server or administrative control

QLectives aims to bring these together to form Quality Collectives, i.e. functional decentralised communities that self-organise and self-maintain for the benefit of the people who comprise them. We aim to generate theory at the social level, design algorithms and deploy prototypes targeted towards two application domains:

- **QMedia** - an interactive peer-to-peer media distribution system (including live streaming), providing fully distributed social filtering and recommendation for quality
- **QScience** - a distributed platform for scientists allowing them to locate or form new communities and quality reviewing mechanisms, which are transparent and promote quality

The approach of the QLectives project is unique in that it brings together a highly inter-disciplinary team applied to specific real world problems. The project applies a scientific approach to research by formulating theories, applying them to real systems and then performing detailed measurements of system and user behaviour to validate or modify our theories if necessary. The two applications will be based on two existing user communities comprising several thousand people - so-called ”Living labs”, media sharing community tribler.org; and the scientific collaboration forum EconoPhysics.

Executive summary

This report accompanies and documents the version 1.0 of the QLectives Platform software. The aim of the QLectives Platform is to combine social networking, facilitation of quality and scalable peer-to-peer (P2P) technology into a next-generation peer-production platform. It consists of several components which are continously refined using input from other workpackages. All the components of the QLectives Platform are (and will be) generic and re-usable as they can handle various content types (e.g. software, video, photo, text) and are not tied to a specific application domain.

The QLectives Platform is built on top of the already deployed and mature P2P tribler.org code-base, which provides most of the low-level P2P functionalities for the social networking and quality facilitation required. The software architecture of QLectives Platform is fully modular and its components can evolve separately.

This report gives an overview of the overall system architecture of the whole QLectives Platform. Most of the technical details on the components are supplied in the appendices as they essentially belong to the tribler.org code-base. Our software engineering methodology is evolving the components of the platform and adding new features to it. Built on the previously existed code-base, QLectives Platfrom version 1.0 provides the following functionalities:

- for community features and facilitation of quality it is critical that peers can communicate among themselve, without being prevented by NATs, firewalls, etc. Thus, a *NAT-puncturing module* has been developed which allows increased connectivity in a P2P system, circumventing prevalent issues in topology management; and

- the first version of the *P2P-Widgets module prototype* allows for code to be dynamically loaded from the network in a P2P application, enabling the

runtime addition of functionalities to the system.

Contents

Chapter 1

Introduction

This report accompanies and documents the version 1.0 of the QLectives Platform software. It is implemented in the context of the QLectives project which can serve as middleware to develop peer-to-peer (P2P) applications. Since the start of the project, the QLectives Platform is the core of the QMedia living lab software, which is called Tribler and described in detail in the deliverable D4.3.1. In the future, the QLectives project will consider porting the current QScience living lab software to the QLectives Platform.

The remaining of this document describes the first release of the QLectives Platform. Chapter 2 describes the necessary background for the subsequent chapter, overviews of the open-source code base and protocols that serve as starting point for the QLectives platform and presents the overall architecture of the platform. After the overview, the following chapters focus on contributions from the QLectives efforts that build on the code base to compose version 1.0 of the QLectives Platform. Chapters 3 and 4 highlight and describe the implementation associated with these contributions, namely:

NAT-puncturing module that allows for increased connectivity in a P2P system, circumventing prevalent issues in topology management.

P2P-Widgets module prototype that allows for code to be dynamically loaded from the network in a P2P application, enabling the runtime addition of functionalities to the system.

Chapter 2

System Architecture

This chapter first puts in context and describes the open-source code base that serves as the starting point of our implementation efforts, discussing Tribler and BitTorrent. Next, it overviews the architecture of the current release of the QLectives Platform.

2.1 Tribler Background

The QLectives Platform builds on the Tribler open-source software. Tribler is a media-sharing peer-to-peer client in active development since 2006 with code contributed by volunteers and research projects funded by the European Union and by Dutch funding agencies. The motivation for basing our implementation on this code base is twofold. First, Tribler was selected as the software for the QMedia Living Lab in the context of the QLectives project. It therefore makes sense to start the QLectives Platform from Tribler's code. Second, using a mature open-source project in active development as our code base allows us to share the burden of implementing basic software functionality with others and to focus on specific and innovative aspects of development.

2.2 BitTorrent Background

BitTorrent allows for scalable and efficient peer-to-peer data sharing. This protocol is used in all file-sharing in the QLectives Platform, and is therefore described here.

A peer wishing to download a particular file through BitTorrent first needs to obtain a *torrent* metafile for the file from, for example, a Web site or RSS news feed. The metafile gives the peer the address of a *tracker* for the file and checksums to verify downloaded parts of the file. The peer then contacts the tracker to obtain a list of peers currently involved in downloading the file, implying they have pieces of the file to share.

Next, the peer contacts a random peer to obtain a first piece of the file itself. With this piece in hand, the peer starts to contact other peers in the list to see if they will trade its piece for another part of the file. If so, the contacted peer sends a few blocks of the negotiated piece, and continues to do so as long as the other does the same. This tit-for-tat mechanism automatically locks out peers who are unwilling to upload themselves. By monitoring the download rate obtained from its current set of peers and randomly trying other peers to see if faster peers are available, a user can maximize its download rate. By always selecting a rare part of the file from the pieces on offer, a peer ensures it always has a piece of the file that other peers are interested in. These policies for piece selection and bandwidth trading lead to a balanced economy with suppliers meeting demand and achieving their own goal (fast download) at the same time. Once the peer has obtained the complete file it will become a *seeder* and provide pieces to other peers without any return. The set of all peers currently actively exchanging pieces of the file is called the file's *swarm*.

2.3 Architecture Overview

Given a description of Tribler and BitTorrent, it is possible now to discuss the architecture of the QLectives Platform. Recall that development has Tribler's code base as a starting point. This section describes the architecture of the QLectives Platform including all development done prior to the QLectives efforts (and hence Background Intellectual Property for the purposes of QLectives). The subsequent chapters highlight and detail the development done specifically in the context of the Qlectives project (Foreground Intellectual Property). The details for the remainder modules of the platform are presented in appendixes references throughout this section.

Figure 2.1 depicts the architecture of the version 1.0 of the QLectives Platform.

The bottom layer is formed by the BitTorrent socket layer that handles incoming TCP connections and UDP packets and hands them to the higher layers. Going from left to right, the `SecureOverlay`, `OverlayThreadingBridge` and `OverlayApps` classes implement the Overlay-Swarm extension, which handles strong authentication and message exchange between peers (see Appendix A.2). This functionality is used by a number of other QLectives Platform components collectively shown in the figure as the `Overlay Protocol Message Handlers`:

- Decentralized Recommendation Support module, also called BuddyCast and described in Appendix B;
- Remote Query module, detailed in Appendix C;
- Cooperative Downloading module, described in Appendix D;
- NAT/Firewall Detection module, described in Appendix E;
- Reputation System module, described in Appendix F;
- Social Networking module, described in Appendix G); and
- Channels module, described in Appendix H.

These components use a set of databases collectively known as the *MegaCaches*.

The `TorrentShare` component wraps all classes that deal with the download of a single torrent, and is described in detail below. The `Session` class manages the collection of currently active downloads and is controlled by the GUI. Most of its functions are delegated to the `LaunchMany` class that contains the thread responsible for handling network traffic. To make callbacks to the user of the API the `Session` uses a `UserCallbackHandler` and associated thread pool. The `DHT` class implements the mainline BitTorrent DHT functionality [10]. The internal tracker makes publishing content via the QLectives Platform easier. The `UPnPThread` helps in dealing with Network Address Translators (NATs) and firewalls, as described in Appendix E.

2.3.1 Overlay Protocol Message Handlers

Figure 2.2 shows the current set of overlay-protocol message handlers. There is one handler for each BitTorrent extension implemented in the Platform. We

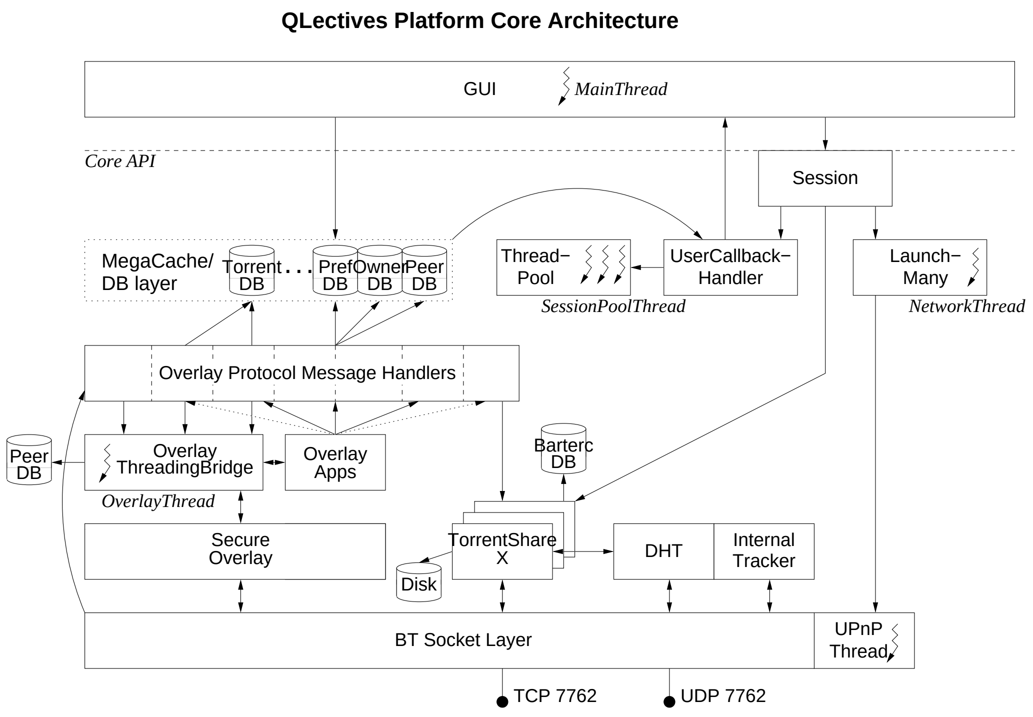

Figure 2.1: Current architecture of the platform. Boxes represent classes or components, cylinders represent databases or disk storage and lightning bolts represent threads.

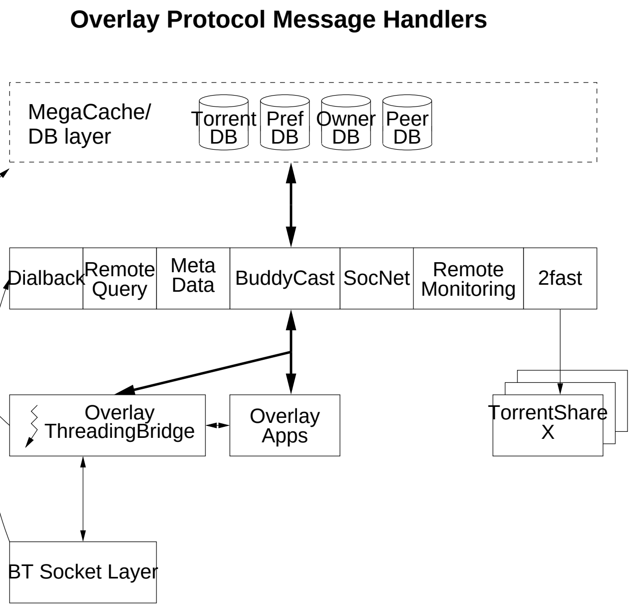

Figure 2.2: Architecture of the Overlay Message Handlers and MegaCache/DB Layer. Thick arrows represent a summary of all the point-to-point interactions between the components in the two layers.

refer to Appendix B and further for a complete description of the Dialback, Remote Query, BuddyCast+MetaData, Social Networking (SocNet) and cooperative downloader (2fast) extensions, and that of the Overlay-Swarm protocol itself.

2.3.2 TorrentShare Component

Going one level deeper, we arrive at the architecture of the TorrentShare component which handles a single ongoing download, shown in Figure 2.3. This part of the architecture is inherited from the original ABC BitTorrent client [15] on which the Tribler background IPR is based. To explain its architecture, this section describes the steps taken when downloading a torrent.

When an instance of the QLectives Platform starts it:

1. Creates a `RawServer` object which creates a `SocketHandler` object and schedules a task that checks for timeouts on the `SocketHandler`'s connections. A task is a method that the `RawServer` will execute once at a given time. Most tasks reschedule themselves, making themselves periodic.

2. Calls `RawServer` to find and bind to a TCP listen port and attempts to open

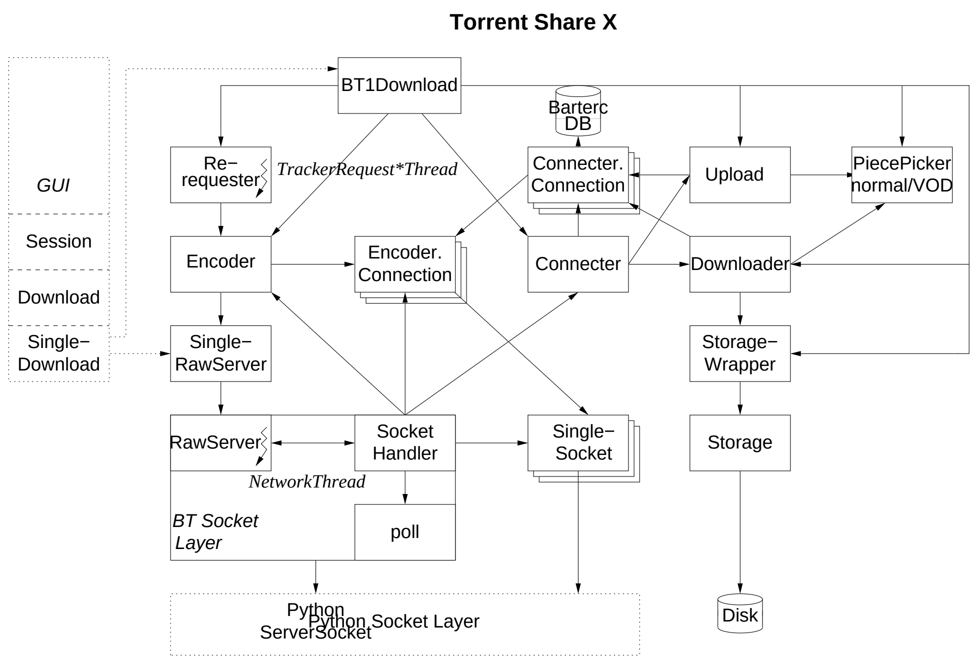

Figure 2.3: Architecture of the `TorrentShare` component.

this port on the firewall (if any) via UPnP.

3. When asked to download a torrent, the `Session` object (see above) via the `SingleDownload` object ultimately creates a `BT1Download`, shown at the top of the figure. This `BT1Download` object creates a `SingleRawServer` object to which all network traffic for this torrent is demultiplexed by the central `RawServer` just created.

4. The constructor of the `BTDownload1` object, in addition, creates a `PiecePicker(VOD)` object. This latter object controls which piece to download next. It also creates a `Choker` object that will execute the optimistic unchoking policy. To this extent it schedules a task with the central `RawServer`.

5. The `SingleDownload` then calls `BTDownload1.initFiles()`.

6. `initFiles` creates a `Storage` object, which maps all bytes in the files in a torrent into a single linear address space. All indices used in the BitTorrent protocol refer to this address space. So a piece of data identified by an index is mapped to a particular offset in a particular file by this object. Furthermore, it creates an empty file for each file in the torrent, and handles file locking.

7. `initFiles` creates a `StorageWrapper` object. Its main task is to execute the disk-allocation policy. Multiple policies are supported. E.g. "sparse" uses sparse files (i.e., the pieces are written on their correct place in a file, but only the actual data written is allocated on the file system). E.g. "pre-allocate" fills any gaps in the file with zeros. The default policy is "normal" which writes data consecutively into the files and moves the data in the right location in the background. In addition, the `StorageWrapper` schedules a task with `RawServer` that checks the integrity of any data already on disk in case of a restart. It also schedules a task that automatically synchronizes the data to disk.

8. The integrity check scheduled by the `StorageWrapper` object is run.

9. The `SingleDownload` object calls `BTDownload1.startEngine()`.

10. `startEngine()` initializes the `PiecePicker` with information about the pieces already on disk, if any. It creates `Measure` objects for up- and downloads. It creates a `RateLimiter` object. It creates a `Downloader` object that keeps track of the download side of the BitTorrent download (e.g. which pieces have been received, which pieces are available from peers, etc.). It creates a similar object for the upload side: `Upload`.

    It creates a `Connecter` object. It creates a `Encoder` object which schedules a task with `RawServer` to send keep alives on all connections. (In the BT protocol, messages of length zero are keep alives). Finally, it may create a `HTTPDownloader` object which is used by the 'httpseeds' extension to BitTornado that allows seeding of files directly from Web sites [9].

11. The `BTDownload1` object creates a `Rerequester` object which is the tracker client and which uses a separate thread.

12. When the `Rerequester` has obtained some peer addresses from the tracker it calls `Encoder.start_connections()`.

13. The `Encoder` calls `Rawserver.start_connection()` and creates a `Encode.Connection` object for the created connection.

14. The `Rawserver` calls `SocketHandler.start_connection()`.

15. The `SocketHandler` creates a Python socket and connects to the peer. It then registers the socket with a `poll` object from Python's `select` module, and creates a `SingleSocket` object for the connection.

16. The QLectives Platform's *Network Thread* finally calls `RawServer.listen_forever()`, the object's mainloop.

17. The mainloop calls `SocketHandler.do_poll()` which calls the TCP library's `poll()/select()` method for asynchronous single-threaded networking.

18. When data comes in on a connection, the `SocketHandler.handle_events()` method is called. This method reads the data from the socket and reports it to the `Encoder.Connection` for the connection via the `data_came_in()` method.

19. The `Encoder.Connection` calls the `next_func` function pointer to handle the data. In the handshake phase of a BT connection this will call methods in `Encoder.Connection` to handle the handshake. If the handshake is successful, the object calls the `Connecter`'s `connection_made()` method. All subsequently received messages are delivered to the `Connecter` via `got_message()`

20. The `Connecter connection_made()` method creates its own `Connecter/Connection` object which will handle all non-handshake BT messages. It also registers the connection with the `Downloader`, the `Upload` and the `Choker`. (Note that there are two classes called `Connection`, in the `Encoder` module and in the `Connecter` module.).

21. When messages come in, the `Connecter` will call the appropriate message handlers in these latter three objects and the `Connecter.Connection`. For example, when a `PIECE` message comes in, it calls the `Downloader`, which, in turn, calls the `StorageWrapper`.

22. To send a message, `Connecter.Connection` calls the `Encoder.Connection`, which, in turn, calls the `SingleSocket` object, which, in turn, writes the message to the Python socket.

23. When a new connection arrives on a listening socket, the `SocketHandler` calls the `Encoder` which creates a `Encoder.Connection` object for it, as with outgoing connections.

Chapter 3

NAT-puncturing module

NAT puncturing is a technique to increase the connectability among peers in a distributed application. This chapter describes the design, implementation and evaluation of a NAT puncturing module for the QLectives Platform.

In Peer-to-Peer (P2P) systems computers on the Internet connect with each other in a symmetrical fashion. This requires that arbitrary computers on the Internet are able to connect with each other. However, the deployment of firewalls and Network Address Translator (NAT) boxes creates obstacles. NAT puncturing circumvents some of these obstacles.

By their very nature, firewalls are meant to regulate what connections are permitted. Moreover, firewalls are frequently configured to allow only connections initiated by the computers inside its domain, based on the assumption of the client-server model of communication. Of course, in a P2P setting connections may also be incoming, often on non-standard ports, which these firewalls don't allow.

NAT boxes pose a separate but related problem. Although NAT by itself is not meant as a connection filtering technology, it does present an obstacle for setting up connections: the publicly visible communications endpoint (IP and port combination) is not visible for the computer behind the NAT box, and may even be different for each remote endpoint. To make matters worse, NAT technology is ofter combined with firewalling.

The techniques for dealing with NATs and firewalls are well known. For example, the STUN [13] protocol details how a computer can determine what kind of NAT and firewall is between itself and the public Internet. Connection setup can be done through connection brokering or rendez-vous [6]. It should be noted though that the connection setup techniques are most useful for UDP traffic. Setting up a connection for TCP traffic when both computers are behind a NAT/firewall requires non-standard use of TCP and IP mechanisms, and may rely on specific NAT/firewall behaviour to work [4].

3.1 Terminology

For the description of the NAT puncturing module in this document we use the terminology introduced by the BEHAVE working group [2]. Specifically we will use the following terms and their respective abbreviations:

Endpoint-Independent Mapping (EIM) The NAT reuses the port mapping for subsequent packets from the same IP address and port to *any* external IP address and port. So when sending packets to host A and B from the same internal IP address and port, hosts A and B will see the same external IP address and port.

Address-Dependent Mapping (ADM) The NAT reuses the port mapping for subsequent packet from the same IP address and port to *the same* external IP address, *regardless of the external port*. This means that host A will always see the same external IP address and port, regardless of the port on host A, but host B will see a different mapping.

Address and Port-Dependent Mapping (APDM) The NAT only reuses the port mapping for subsequent packets using the same internal and external IP addresses and ports, for the duration of the mapping. Even when communicating from the same internal IP address and port, host A will see two different external IP addresses and/or ports when different ports on host A are used.

Endpoint-Independent Filtering (EIF) The NAT or firewall allows packets from any external IP address and port, for a specific endpoint on the NAT or firewall.

Address-Dependent Filtering (ADF) The NAT or firewall allows packets destined to as specific endpoint on the NAT or firewall from a specific external

IP, after a computer behind the NAT of firewall has sent a single packet to the external IP.

Address and Port-Dependent Filtering (APDF) The NAT or firewall allows packets destined to a specific endpoint on the NAT or firewall from external endpoints only after a packet has been sent to that external endpoint.

For the purposes of our description there is no difference between ADM and APDM as only single endpoints on the different hosts are considered, which we will therefore combine into the abbreviation A(P)DM. Similarly we collapse the definitions of ADF and APDF into A(P)DF.

3.2 Implementation

We have implemented a UDP NAT/firewall puncturing module as part of the QLectives Platform. Although operational, the module is not yet used in user downloads at this stage. Because the evaluation of such a tool can only be made when deployed in users' machines, we opted to have the current module test its efficacy and report back its results before we move users' connection process to rely entirely on this module.

Currently, the NAT puncturing process works as follows. First, the puncturing module builds a swarm, much like the BitTorrent software. However, instead of having a separate tracker, it uses a peer at a pre-programmed address and port and employs a form of Peer EXchange (PEX) to find new peers to connect with. The PEX messages include the peer ID, the IP address and port at which the peer originating the PEX message communicates with the remote peer, and the NAT/firewall type of the remote peer. The latter is included so the receiving peer can determine whether it is useful to attempt to connect to the remote peer, as certain combinations of mapping and filtering are not able to connect to each other.

Our implementation tries to make minimal use of centralised components. Therefore the detection of the NAT and firewall type are not done using STUN. Instead, the peers rely on information reported by other peers and by checking if other peers can connect to it without previous communication in the reverse direction. Furthermore, peers use other peers as rendez-vous servers (see Fig-

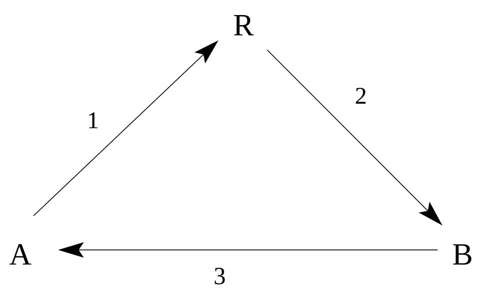
R
A
B
1
2
3
1. A sends a reverse connection request for B to R
2. R forwards the reverse connection request to B
3. B sends a connection request to A

Figure 3.1: Rendez-vous for connection setup.

ure 3.1). If a peer R is connected to two other peers A and B, it can serve as rendez-vous server for them.

In the following sections we will describe the tests used by peers to determine their NAT and firewall types.

3.2.1 NAT Type Detection

To allow determination of the NAT type that a peer is behind, all peers report the remote address and port they see when a connection is set up. So each peer will receive, from its communication partner, his own external address and port. If the reports from all communication partners are the same (or at least a large majority is the same), a peer will determine that there is either no NAT or the NAT has EIM behaviour. Note that the two cases (no NAT or EIM behaviour) are indistinguishable without a reliable determination of the local address. Furthermore, the difference is mostly irrelevant. If, however, the reported external IP address and/or port are constantly different, then the peer concludes that it is behind a A(P)DM NAT.

3.2.2 Filtering Behaviour Detection

The filtering behaviour of a NAT/firewall is detected by checking whether a direct connection request arrives before a reverse connection request arrives. To enable this to work, when a peer tries to set up a connection using rendez-vous, it will always first send a direct connection request to the remote peer. In most cases this direct connection request will arrive at the remote peer before the reverse

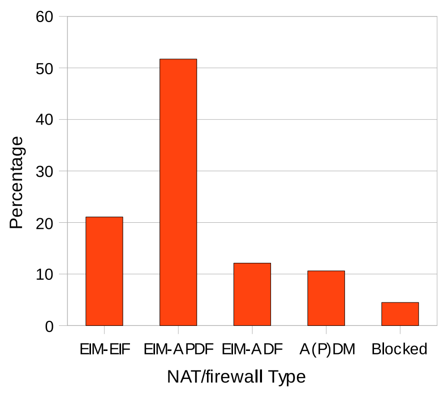

Figure 3.2: NAT/firewall type market share

connection request from the rendez-vous, unless the NAT/firewall behaviour is A(P)DF. So when for a significant fraction of incoming requests the direct connection request arrives before the reverse connection request, the peer concludes that the filtering behaviour is EIF. Otherwise it must conclude that the filtering type is A(P)DF.

In principle it would be possible to distinguish between ADF and APDF by deliberately sending a direct connection request to the wrong port. If the connection is subsequently successfully set up, the firewall is of the ADF type. However, to conclude that the firewall is of the APDF type requires several such experiments to ensure that packet loss is not causing a wrong conclusion. Furthermore, the added value is limited as the fraction of ADF firewalls is small (see Section 3.3) and the only extra connections it allows is between ADF and ADPM NAT/firewalled peers.

In the Section 3.3 we do make a distinction between ADF and APDF firewalls. This distinction was made from studying the logs. When a peer with A(P)DF filtering behaviour was able to connect to a peer with A(P)DM mapping behaviour it must be an ADF peer. This could also be implemented in the software, but the added value would be even more limited because to determine this we implemented the software such that A(P)DF peers try to connect to A(P)DM peers anyway, which results in the same connections being tried.

Table 3.1: Connection success rate per NAT/firewall type

From \ To	EIM-EIF	EIM-APDF	EIM-ADF	A(P)DM
EIM-EIF	88%	79%	79%	70%
EIM-APDF	83%	63%	76%	0%
EIM-ADF	79%	73%	84%	41%
A(P)DM	78%	0%	14%	0%

3.3 Evaluation Results

In this section we present the results of a trial of the NAT puncturing module with 826 random peers. The trial was conducted as part of a collaborative trial of QLective and P2P-Next (our collaborating project), where public broadcaster BBC provided us with video material This trial allowed us to deploy the NAT puncturing module on real users' machines.

Figure 3.2 shows the detected NAT and firewall types for 669 peers which were able to draw a conclusion from the connections they made. The most dominant type of NAT/firewall is EIM-APDF (52%). This includes both simple firewalls and EIM NATs. Theoretically these can connect to each other through a rendez-vous peer. The fraction of peers that are behind A(P)DM firewalls is only 11%. As these NAT/firewalls are the biggest obstacle, i.e. they can only connect to EIF filtered peers although connections with ADF filtered peers can also be set up if the ADF filtered peer initiates the connection.

3.3.1 Connection Success Rate

Next we look at the success rate in setting up a connection between two peers. We have split these out into the different NAT/firewall types we have distinguished (see Table 3.1). Note that these percentages refer to a single attempt, without retries. As expected the connections between EIF filtered peers are successful most often (88%). As already noted in the RFC describing the STUN protocol [13], even though two peers should be able to communicate given the classification determined here, subtleties in the actual implementation may prevent actual connection setup. This effect is already visible in the connections to EIF filtered peers, for which there should theoretically be no difference in success rate for different originating NAT/firewalls. However, there is a significant difference between EIM-EIF/EIM-EIF and A(P)DM/EIM-EIF connections.

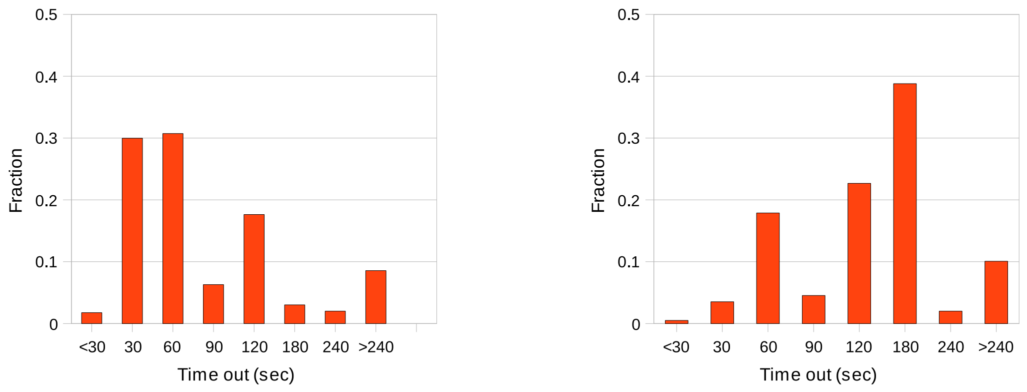

Figure 3.3: Mapping/firewall hole time out without (left) and with (right) handshake.

What should be noted is that the success rate for connecting to EIM-A(P)DF peers is expected to be lower, as that also depends on the success of the rendez-vous. As our implementation has no acknowledgements for the reverse connection request forwarding, the results should be scaled by the rendez-vous success rate (approx. 87%). However, even when taking the rendez-vous success rate into account, the EIM-APDF/EIM-APDF connection success rate is only 72%. This again indicates that the classification used does not capture all the subtleties of NAT/firewall operation.

Attempts to set up connections between EIM-APDF peers and A(P)DM peers always fail as expected, as do connections between A(P)DM peers. Interestingly, connections from A(P)DM peers to EIM-ADF peers do succeed for a small fraction of attempts. This may be due to an opened port from a previous attempt between the same peers, or due to mis-classification of an EIM-EIF peer as EIM-ADF.

3.3.2 Time out

A final measurement we made was the time out for the mappings/firewall holes. To measure the timeout, we sent UDP packets to a pre-programmed IP address and port, after which a reply was sent with a delay. To ensure a rapid measurement, several messages requesting replies at different delays were sent at the same time. This setup can give wrong results for NAT/firewalls which refresh the timeout for incoming traffic as well as for outgoing traffic. However, doing so is a security risk, so we expect few firewalls to behave this way.

The graph on the left in Figure 3.3 shows the results of this measurement. These results indicate that many firewalls employ a fairly short time out (1 minute or less) for the created mappings/firewall holes. However, some NAT/firewalls, particularly those based on the Linux kernel, use a longer time out if more than two packets have been sent on a particular mapping/firewall hole. We therefore also measured the time out where we include an initial handshake before requesting the delayed reply. The results are shown in the graph on the right in Figure 3.3. The results are markedly different. Most of the NAT/firewalls that have a 30 second time out without the handshake now use a much longer time out. The default value in the Linux kernel is 180 seconds, which explains the large fraction of peers with that time out when using an initial handshake.

In summary, our results show that connectable peers form a small minority (21%) on the Internet, and that most NATed and firewalled peers (63% of all peers) should theoretically be able to set up connections through a rendezvous mechanism. However, our results also show that even though connections should be possible, the connection success rate is significantly lower than for connections between non-filtered peers.

We intend to look into the effect of retries, which may increase the success rate, particularly for firewalled peers.

Chapter 4

P2P-Widgets prototype module

P2P Widgets are chunks or code that can be disseminated and dynamically deployed in a P2P network to create new services and applications on top of a P2P substrate. With an efficient method for widget discovery and deployment, users can choose to add new functionality to their P2P client on runtime, adding for example new group interaction mechanisms or quality assessment code based on his social interactions.

Our implementation of P2P Widgets has two dimensions. In this deliverable we describe the design and implementation of the widgets substrate, that allow any application developed on top of the QLectives Platform to discover, acquire and install widgets. In Deliverable 4.3.1 we describe a user interface implemented in Tribler to expose this functionality to the user and some example widgets.

4.1 High Level Design

4.1.1 Runtime Environment

A Widget Runtime Environment or Widget Engine is the part of the system that takes care of executing and managing the widgets. For a successful Runtime Environment, it is necessary to determine the widget language and file format, the features that are available to the widget (API), and any security measures for execution of malicious widgets.

QLectives Platform is written in Python, thus the Runtime Environment has to be written in either Python or a language that is easily integrated with Python, such as C. When we choose a language and file format for the widget, it is important that the Runtime is able to parse the files and execute the language. There are several interpreters and compilers written in Python. None of them, however, are far enough developed to be used without any problems. Therefore, we chose to use the Python interpreter itself to execute the widgets. This means the widget must at least contain one or more Python files. Other files could be metadata files (which contain for example the author, a Widget ID, version number) and resource files such as images.

In the current version of our P2P Widgets prototype we chose to not support metadata and resource files and only support widgets that consist of one Python file.

The widgets, which are essentially Python modules, have access to all the Python libraries and all the code of QLectives Platform. This means the Widget API is already abundant and the developer can implement a lot of features, when he knows about them, but can also misuse a lot of features. Thus, on the one hand there should be enough de velopers resources to be able to create interesting widgets, but we should also take care of security issues. Implemented security measures in QLectives Platform are as follows

Code signing means that the author of the widget and the widget integrity can be verified. This verification allows one to run only widgets from trusted sources and discourages the publishing of malicious widgets.

Whitelisting can be done by certificates. Certain peers are designated to be initially trusted and they may create whitelist certificates to whitelist a widget author. We can then distinguish between trusted widget authors and untrusted widget authors.

A rating system is implemented in the Widget Market (described in section 4.1.4), where users can rate and review widgets. These reviews and ratings can be used by other users to evaluate widgets before acquiring them. However, a rating system alone would not suffice. The reasons are as follows. First, there may be false ratings and comments. Second, users have to try the widget to rate or review it, which means they must first install the malicious widget before they notice it is malicious. But then it may be already too late. However, the three security measures combined should take care of most of the abuse.

4.1.2 Discovery and Download

The Widget Discovery and Download subsystem is a key element of the Widget System and collaborates closely with the Widget Market, which is used to find, browse, rate and review the widgets. The Discovery and Download subsystem should take care of the discovery and download of widgets using the P2P System in a scalable and decentralised manner. With the design of the subsystem, the browsing requirement of the widget repository should be taken into account.

The discovery and download strategies are the most important design choices. BuddyCast, which disseminates infohashes and torrents, is used for the discovery and download strategy. When a torrent is created for each widget and its seeding gets started, BuddyCast automatically disseminates the torrent files in the network. We only need to distinguish widget torrents from others and on receipt of a widget torrent, notify the Widget Market. However, BuddyCast only maintains 5000 torrents in the local database. But with a few modifications in BuddyCast and TorrentCollecting, we could make a bias towards widget torrents, such that they are disseminated faster. Because we now use torrents, the BitTorrent download protocol can be used easily for downloading.

We distinguish two extremes in download strategies, namely the Download-On-Install and the Download-On-Discovery strategies. The Download-On-Install strategy takes the least space on disk, but might take a lot of patience from the user when he wants to install a widget. The Download-On-Discovery strategy takes the most disk space and the most bandwidth, because it will eventually have most of the widgets on disk, but instalment will take almost no time. We chose for a strategy that is most similar to the Download-On-Discovery. However, we will only download one widget at the time, to be able to control the bandwidth usage. We select the most popular widgets to collect first, as this will decrease download time on installation in most of the cases.

4.1.3 Widget Communication and Storage

The Widget Communication and Storage subsystem takes care of the local storage per widget, finding intra-widget communication peers in the P2P network and handling the intra-widget communication messages. The local storage and communication are combined in one subsystem, because they are so closely related. Both the local storage and Communication API can be designed such that there is a lot of freedom for the widget or they can be very strict. For example, we might design the local storage to support 5 key-value pairs per widget, or to support a full database where tables can be created and queried as one would like.

The Communication API might support functions for retrieving communication partners, direct communication and broadcast. This way, there is little control over the messages and bandwidth to be communicated. It could also support gossiping in a way that the system controls the bandwidth. This can be done by letting the widget implement functions for handling received messages and for the creation of a gossip message. The system then chooses the communication partners and is able to control the bandwidth by checking the message size and choosing the message interval.

We have chosen to support the gossip system [1], to have more control over the widget communication. The Widget Communication and Storage subsystem takes care of selecting the gossip partners and controls the bandwidth. The message structure is left to the widget itself. Because widgets are essentially small programs with one theme, it is enough to support one database table per widget. The table structure can be defined in the widget such that its data is completely customised. With an easy API for inserting, deleting, selecting and updating the storage table, a widget that communicates should be easily created.

Finding communication partners for a widget can be supported in two obvious ways. First, BuddyCast data contains information on who is seeding which torrent. Since widgets also have a torrent which it seeds when the widget is installed, this information is automatically propagated by BuddyCast. Another way is by hooking into the swarm: retrieving peers from the tracker, DHT or PEX messages. The peers that are seeding are likely to have installed the widget. Of course, the swarm seems the most active and up to date, thus this will be the primary source for widget communication partners.

4.1.4 Widget Market

The Widget Market is the part of the system that allows users to browse, install, rate and review widgets. It collaborates closely with the Discovery and Download subsystem. After the widget to be installed is downloaded, the Widget Runtime Environment is called to install the widget.

In our implementation the Widget Market is a widget itself, and use the Widget Communication and Storage features to disseminate and store ratings and reviews. The reasons are as follows. First, the Widget Communication will only exchange information with peers that have the same widget. This implies that the bandwidth of other peers is not wasted and every exchange of information is useful for the receiver. Communication is thus a lot more effective than for example using BuddyCast, which would exchange ratings and reviews for millions of torrents, most of which the receivers of those ratings and comments will not look at. The second reason is that using this setup we can easily show the effectiveness of the Widget Communication and Storage subsystem.

4.2 Technical Design

In this section, we give the details of our design. We start by presenting the architecture of the system and then discuss each subsystem in detail. In Figure 4.1, the class diagram of the major components are visualised with their connections to the associated Tribler components. BuddyCast, MetadataHandler, TorrentCollecting and the OverlayBridge are original components of Tribler, which are used by or modified for our system.

4.2.1 Runtime Environment

The WidgetPanel, WidgetCore and WidgetDBHandler together form the Widget Runtime Environment. The WidgetPanel is the panel where the widgets are shown on. The WidgetCore is used for monitoring widget downloads for installation, reading the widget files and installation of the widgets. The WidgetDBHandler is the database handler for both the Widget and the WidgetInstance tables. Apart from the usual insert, update and delete functions, it has more advanced functions such as calculating a suitable free position on the WidgetPanel for a widget using other widget locations and sizes, and selecting a popular widget for downloading.

When a torrent is received from the MetadataHandler, various torrent information is stored in the database. Widget torrents are enriched with specific widget data and this is stored in the Widget database table. The fields it includes

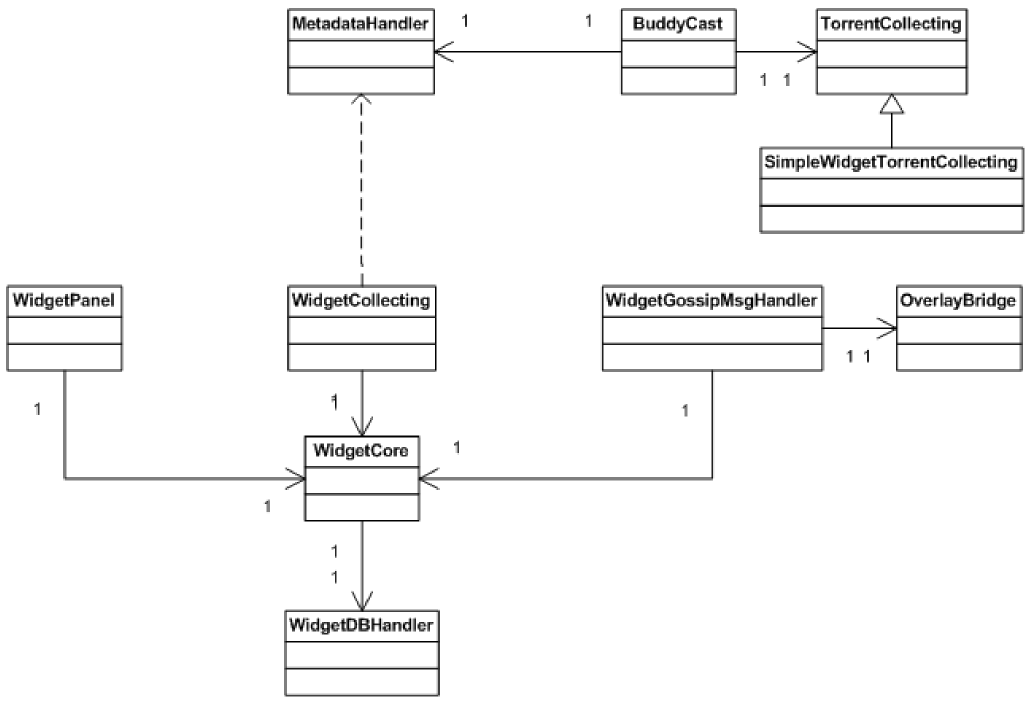

Figure 4.1: Class Diagram of the Widget System in relation to QLectives Platform

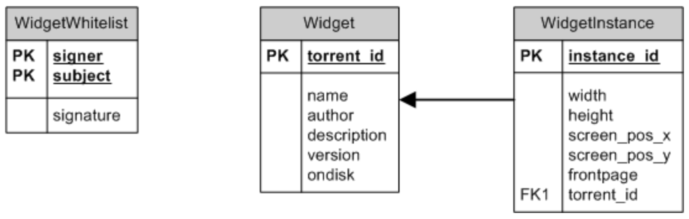

Figure 4.2: Database design of the Widget System.

can be viewed in Figure 4.2. This information is primarily used by the Widget Market.

Upon installation of a widget, the widget filename is retrieved from the database and the file is read. The widget Python module is imported and a widget class instance is made. A suitable position on the WidgetPanel is calculated and when the widget instance, including size and position information, is inserted into the database, it receives its unique instance ID. The widget is then encapsulated inside a widget wrapper, adding a title bar with close button to the widget and for security reasons: it is harder to get to the WidgetPanel from the widget wrapper, because the WidgetPanel is not the parent of the widget.

Now that the widget is installed, the WidgetCore initialises the local storage and gossip features of the widget, by creating a database table and local storage database handler, and notifying the WidgetGossipMsgHandler of this widget. It is now possible to use the whole QLectives Platform API and Python libraries, without intervention of the Runtime Environment. Also, the WidgetCore starts seeding the widget file, for finding gossip communication partners and to allow fast downloads for new users of the widget.

Upon removal of the widget instance, its instance is removed from the WidgetPanel and WidgetGossipMsgHandler, and removed from the WidgetInstance table. The seeding of the widget file is stopped and the user is removed from that swarm, but the Widget file itself is kept on disk. This enables fast re-installation without the need for downloading the widget again. Upon re-installation, the local storage is lost because that table is dropped when the widget instance is removed.

4.2.2 Widget Format

Widgets consist of one Python file, thus one module. However, the widgets Python file should conform to a specific format. First, it is necessary put the following metadata in the widget module: name, author, version, description. Second, the frontpage and menuitem options can be set. They default to a frontpage widget without menuitem. For frontpage widgets, the width and height must also be set. Further, there must be a widget class in the file, that extends the tribler widget class, which is basically a panel with several function stubs, such as OnClose, OnPostInit, OnCreateGossipMessage, OnHandleGossipMessage, and OnMenu. In the init function, the user interface must be created. In this widget class, the gossip option can be enabled, and the local storage structure must be set. Further, the widget developer is left free to put whatever he wants in the Python file.

4.2.3 Security

As discussed, we choose to implement three security measures, namely code signing, whitelisting and a rating system.

Code signing uses the facilities provided by the QLectives Platform to sign a torrent with the users PermID. When the widget is inserted into the Platform, a torrent is created and the torrent is automatically signed with the publishers

PermID. Using this signature it is possible to verify the publisher of the widget. Further, the infohash in the torrent is used to check the integrity of the widget code.

Whitelisting is done much like public key infrastructures. At first, every widget is untrusted. There are a few initially trusted peers, that can create certificates to whitelist widget authors. In a whitelist certificate, the public key (PermID) of the widget author is put and it is then signed by the initially trusted peer. Signing is done by first adding the signers PermID and the date to the certificate, creating a signature of this certificate and adding the signature to the certificate. Validating a certificate can be done by first removing the signature from the certificate and then verifying the certificate, signer PermID and signature. Initially, only whitelisted widgets are shown in the Widget Market, but there is an option to show every widget, including the untrusted ones.

Finally, the rating system makes use of direct feedback from users to provide an initial body of information for users to asses the usefulness and security of a widget.

4.2.4 Discovery and Download

Widget discovery is done by first creating torrents for the widgets and seeding them in the QLectives Platform. BuddyCast then automatically disseminates the widget torrents just like the other torrents. A BuddyCast message includes infohashes of the torrents it is seeding and the infohashes it has collected most recently. Each receipt of a BuddyCast message triggers the TorrentCollecting module to select one random torrent to download from the other peer. By extending the lists of infohashes with the type of the torrent (video, music, document, widget, etc.), we can differentiate the widgets from other file types. Therefore, TorrentCollecting can be more effective and specialised. For example, when the user is primarily interested in movies, it can create a bias by selecting more movie torrents to collect than other torrents. For the current release we implemented SimpleWidgetTorrentCollecting, an extension of the TorrentCollecting module that selects either a widget torrent or another type of torrent. The widget torrent that will be selected is the most popular widget locally known; the other types of torrents are still randomly selected. The calculation of the popularity of torrents is discussed in the next subsection.

The MetadataHandler is the QLectives Platform component that handles the downloading of torrent files. When a torrent is downloaded, the extra widget information that is stored in the widget torrent (name, author, version, description) is added to the database and will be available to the Widget Market from then on. The MetadataHandler then informs the WidgetCollecting module that there is a new widget torrent available.

WidgetCollecting is the component that is dedicated to downloading the most popular widgets on discovery, one at a time. Widgets can also be downloaded when the user selects a widget to be installed which is not yet collected. When WidgetCollecting is initiated, it first checks whether there are any widgets being downloaded. These downloads are checked whether they are being downloaded for installation or not. When there is a widget that is not being downloaded for installation, WidgetCollecting continues to monitor that widget. If there is no widget currently being collected, it queries the database for the most popular widget locally known that is not yet collected, and starts to download and monitor that widget. When there are no widgets locally known that are not yet collected, it waits for the MetadataHandler to send a notification of a new widget torrent and starts to download that widget.

When the widget is collected within the time limit, a value that can be adjusted but is set at a 5 minutes default, the procedure to find a new widget to collect is restarted. But when the widget download takes too long, it tries to find another widget to collect. If another widget is found, the current widget is marked as being a slow download, and starts downloading the other widget instead. If not, it will reset the time limit and proceed with the slow download. Bandwidth limitations for widget collecting are currently not implemented, as widgets are so small they can be downloaded without notice with current Internet connection speeds. However, because WidgetCollecting uses the standard BitTorrent Download API of the QLectives Platform, WidgetCollecting could easily be extended with bandwidth limitations, when necessary. The widget collecting process is illustrated by the state machine in Figure 4.3.

4.2.5 Widget Popularity

The Widget TorrentCollecting and WidgetCollecting both make use of the popularity of widgets. A natural value for the popularity would be the swarm size

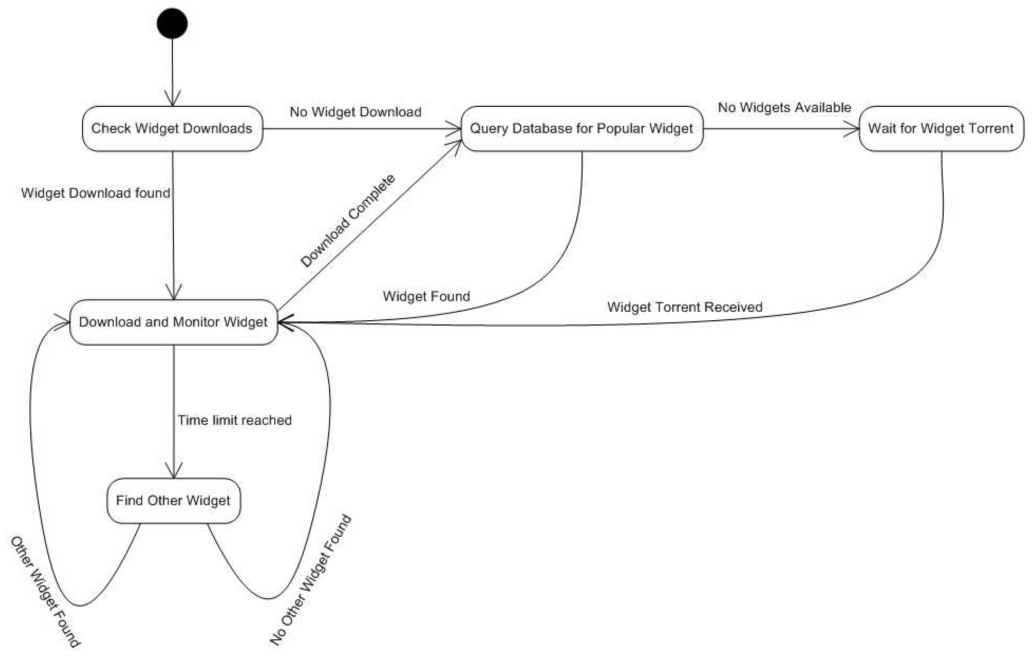

Figure 4.3: State machine of the WidgetCollecting module.

of the torrent, or a derivative of this. Since some trackers and peers lie about the number of seeders and leechers, giving false popularitys, it is necessary to verify this information. In order to do so, BuddyCast was extended such that peers exchange their latest information about the swarm. This swarm size information contains the number of seeders, leechers and the number of locally met QLectives Platform peers who are seeding this torrent. The swarmsize information from all encountered peers is stored with timestamps in the Popularity table.

For ranking torrents by popularity, we average the latest number of peers, the number of QLectives Platform seeds seen by this peer, and the number of locally known QLectives Platform seeds. The number of locally known QLectives Platform seeds is added to reduce the effect of lying peers. The value for each torrent is saved per torrent in QLectives Platforms Torrent table, and updated when new popularity information is received.

4.2.6 Widget Communication and Storage

The Widget Communication and Storage subsystem encompasses two things. First, it must take care of the initialisation of the communication and local storage. Second, it must allow the widgets to communicate with each other by finding the communication candidates and dispatching messages.

The first task is done by the WidgetCore upon installation of the widget. The widget specifies the local storage structure within the Python file as a list of tuples. The first entry of the tuple is the column name and the second entry the column type. The primary key of the table can be specified by placing the keyword “key” behind the column type of the primary key columns. Valid column types are “integer”, “text” and “numeric”. The WidgetCore validates the local storage structure and dispatches the creation of the database table to the WidgetDBHandler. When the local storage table is initialised, the widget gets a database handler variable, which can be used to manipulate the table. The widget must also specify whether it would like to use the gossip features. When this is the case, the WidgetCore notifies the WidgetGossipMsgHandler of this new widget.

The WidgetGossipMsgHandler takes care of all the intra-widget communication. It keeps a list of all the widget instances and a queue of their connection candidates. Every gossip round, it calls the function of the widgets that create and return their gossip message, validates the message, selects a gossip candidate per widget and tries to send the message. Before the message is sent, the message is prepended with the WIDGET GOSSIP type and the widget infohash. Similar to BuddyCast, it keeps a Send Block List and Receive Block List per widget. When a message is sent to/received from a particular peer, the peer is kept in the Send/Receive Block List for the block interval, respectively. When the peer is removed from the list, it is added to the end of the connection candidate list. After selecting a gossip communication partner from the connection candidate list, we shuffle the list to randomise peer selection.

Upon receipt of a WIDGET GOSSIP message, the widget infohash is taken from the message and it is checked whether the widget is installed. If not, a WIDGET NOT INSTALLED message will be replied. If the widget is installed, the appropriate widget message handler is called. After handling the message, the peer is added to the Receive Block List. When a WIDGET NOT INSTALLED is received, the peer is removed from that widgets connection candidate list and the Tribler Preferences table is updated. When a message could not be sent, the peers number of tries is increased and the peer is added to the end of the list. If the number of tries is greater than three, the peer is removed from the connection candidates.

Connection candidates are added from BuddyCast messages and Swarm connections. Upon receipt of a BuddyCast message, it is checked whether the other peer has installed widgets we have installed too. If this is the case, we add the peer as connection candidate for the widget. When a swarm connection is made, we also check whether this is a widget swarm or not. If it is, the connection partner is added. Because the swarm connection is already established, we do not have to create an additional connection and thus keep the number of connections to a minimum. Further, we know from the peer that he is online because we just communicated with him. When the QLectives Platform is started, initial connection candidates are retrieved from the Preference table of Tribler. This table stores information on who is seeding which torrents.

4.2.7 Widget Market

To accomplish the goals of creating a Widget Market, we created a management widget. A widget is ideal for this, because it requires a lot of user interface code and the Widget Communication and Storage features for the widgets are ideal to disseminate reviews and ratings.

We must define the local storage structure to store the reviews per widget, and choose a gossip message format. We define a review as a comment and a rating on a widget. The local storage structure can be seen in Table 4.1. Because we limit each user to rate and review each widget only one time, we must use the PermIDs of the users and the infohashes of the widgets as primary key. Further, we added a human readable name. The rating, comment and date fields are self-explanatory. The clock integer is used to create an order in the reviews, such that a review with a reply to another user is shown beneath that user. When a rating is inserted, its clock value must be higher than the maximum clock value in the table for the corresponding infohash.

The gossip message consists of the 5 most recent reviews and 5 random reviews. Each review has the same format as the local storage structure.

Field	Description	Type
infohash	infohash of the widget	text
permid	permid of rating/comment author	text
name	name of the rating/comment author	text
comment	the comment	text
rating	the rating	integer
date	local date when this rating/comment was added	numeric

Table 4.1: Local Storage Table structure for storing the Widget Markets widget reviews.

Chapter 5

Summary and Further Research Questions

This report documents the first release of the QLectives Platform, a set of modules that can be used to base the development of different P2P applications. For that, this report describes the code base which the QLectives Platform is part of, a code base that is shared with Tribler and QMedia. The overall architecture and background of the different modules that compose the architecture were detailed and discussed. Given this description, the remainder of the document presents the contributions of the QLectives project to the common code base that were necessary to meet the QLectives requirements for version 1.0 of the QLectives Platform.

This version of the QLectives Platform is part of the QMedia release that will be deployed in the first quarter of 2010. The experience with this deployment will inform the forthcoming development of the platform in conjunction with the research plan of QLectives. The release of QLectives Platform version 2.0 is scheduled for month 24 of the project.

Bibliography

[1] A.-M. Kermarrec, M. van Steen, Eds. ACM SIGOPS Operating Systems Review 41, Special Issue on Gossip-Based Networking. 2007.

[2] F. Audet and C. Jennings. Network Address Translation (NAT) Behavioral Requirements for Unicast UDP. RFC 4787 (Best Current Practice), January 2007.

[3] E. Barker, Barker. W., W. Burr, W. Polk, and M. Smid. Recommendation for Key Management - Part 1: General. Special Publication 800-57, National Institute of Standards and Technology, Gaithersburg, MD, USA, August 2005.

[4] Andrew Biggadike, Daniel Ferullo, Geoffrey Wilson, and Adrian Perrig. NATBLASTER: Establishing TCP connections between hosts behind NATs. In *SIGCOMM Asia Workshop*, April 2005.

[5] J. Breese, D. Heckerman, and C. Kadie. Empirical Analysis of Predictive Algorithms for Collaborative Filtering. Technical Report MSR-TR-98-12, Microsoft Research, Redmond, WA, USA, May 1998.

[6] Bryan Ford, Pyda Srisuresh, and Dan Kegel. Peer-to-peer communication across network address translators. In *USENIX 2005*, April 2005.

[7] D. Harrison. Assigned Numbers. BEP 4, BitTorrent Community Forum, `www.bittorrent.org`, January 2008.

[8] G. Hazel and A. Nordberg. Extension for Peers to Send Metadata Files. BEP 9, BitTorrent Community Forum, `www.bittorrent.org`, January 2008.

[9] J. Hoffman and DeHackEd. HTTP Seeding. BEP 17, BitTorrent Community Forum, `www.bittorrent.org`, February 2008.

[10] A. Loewenstern. DHT Protocol. BEP 5, BitTorrent Community Forum, `www.bittorrent.org`, January 2008.

[11] A. Nordberg, L. Strigeus, and G. Hazel. Extension Protocol. BEP 10, BitTorrent Community Forum, `www.bittorrent.org`, January 2008.

[12] R. Rahman, D. Hales, M. Meulpolder, M. Clements, V. Heinink, J. Pouwelse, and H. Sips. Robust Vote Sampling in a P2P Media Distribution System. Technical Report PDS-2008-004, Delft University of Technology, Delft, The Netherlands, May 2008.

[13] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing. Session Traversal Utilities for NAT (STUN). RFC 5389 (Proposed Standard), October 2008.

[14] B Schneier. *Applied Cryptography*. John Wiley & Sons, New York, NY, USA, 2nd edition, 1996.

[15] Tim Tucker. ABC [Yet Another Bittorrent Client]. `http://pingpong-abc.sourceforge.net/`, May 2009.

[16] S. Voulgaris and M. van Steen. Epidemic-style Management of Semantic Overlays for Content-Based Searching. In *Proceedings 11th International Euro-Par Conference*, pages 1143–1152, Lisbon, Portugal, August 2005.

Appendix A

Overlay-Swarm module

BitTorrent does not require strong authentication of peers, as peer-to-peer interactions are transient and shortlived and security stems from the digests in the trusted torrent file. In the context of the QLectives Platform, however, we want to establish longer term relationships between peers and introduce a number of privileged operations which should only be available to friends. We therefore extend the protocol with secure, permanent peer identifiers called *PermIDs*. We assume a PermID maps to a single IP address and port number and is initially also used to identify users. The mapping of PermID to IP address is controlled by the owner of the PermID (a user). Initially we used PermIDs for authentication of friends in cooperative downloads.

The idea is to use public-key cryptography and give each peer a public/private keypair, where the public key will act as the PermID. We intend to use Elliptic Curve-based public key cryptography [14] because it provides stronger protection using small keys than e.g. RSA-based algorithms [3]. Having small PermIDs is useful to allow caching of large numbers of (PermID,IP) pairs, as discussed next.

A.1 PermIDs

In the QLectives Platform, each client creates a public/private key pair based on Elliptic Curve Cryptography. The (currently uncertified) public key acts as the PermID for the user. Users distribute this PermID to their friends out-of-band to establish trusted friend relationships. When two peers connect as part of a download, the QLectives Platform client checks whether the peer supports our PermID extension. If so, it will also setup a overlay-swarm connection to the peer. To successfully set up an overlay-swarm connection both peers need to authenticate themselves using the standard ISO/IEC 9798-3 challenge/response identification protocol.

If the peer is successfully authenticated but not a friend of the user (i.e., does not appear in the list of friends' PermIDs), the QLectives Platform will allow it to request non-privileged operations, such as exchanging file preferences (see Section B.1). If the peer is a friend, it may request privileged operations such as coordinating a friends-assisted download (see Section D).

A.2 The Overlay Swarm

The recommendation and cooperative download feature both require new BitTorrent protocol messages. We require a clean method for extending the protocol because our aim is to include more features in the future. Another requirement is being the least invasive in existing implementations. Furthermore, the current BitTorrent protocol does not allow communication outside the context of a swarm, that is, clients can only communicate with clients that are downloading the same file. For our extensions, we must be able to communicate outside the context of a single file swarm.

We therefore propose to create a new virtual swarm that encompasses all peers that are using the system, called the *overlay swarm* for high-level communication between peers. The swarm to which a peer connection belongs is defined by the infohash field during the initial BitTorrent handshake. This infohash normally contains the SHA1 hash of the contents of the torrent file. In case of the overlay swarm, the infohash must contain a value of all zeros. The overlay swarm has no central BitTorrent tracker. A peer that wants, for example, to exchange preference lists with another peer must use the overlay swarm. The peer connects to the other peer's listen socket and uses the zero infohash value in the handshake. If the handshake is successful both parties know that new extension messages can be exchanged. After connecting to a peer on the overlay swarm the peers must run the challenge/response protocol from Section A to exchange and validate their PermIDs before any other communication.

By using this non-valid infohash value we remain fully backwards compatible and also are minimally invasive to the BitTorrent protocol. It also does not require extra TCP listen ports. The latter implies that no extra configuration of firewalls or Network Address Translators (NATs) is required by the user. This overlay can be extended in the future with new messages for secure gossip, sharing ratio enforcement, social networking, voting/moderation, reputation management, etc .

A.3 Protocol Versioning

As we expect the overlay-swarm protocol to change frequently as new features are added or improved, we have to allow for many different versions of the protocol. Traditionally, the BitTorrent protocol has been versioned using the 64 reserved bits in the BitTorrent header.

More accurately phrased, the client's features are expressed using the reserved bits. See [7] for the current allocation of the reserved bits. More recently, various vendors have started using the (now) official BT extension protocol [11]. We do the same as it has a larger identifier space and thus doesn't require coordination with other BitTorrent vendors to prevent clashes.

A.3.1 Basic Protocol Versioning

We currently have one change to the basic protocol, namely, Merkle torrents. For this feature, we used a reserved bit. At the time, there was only one officially reserved bit, the right-most bit. This is used by BitTorrent 4 to indicated DHT-based tracker support. The "de facto" standards were as follows:

Azureus The left-most bit indicates support for the Azureus Messaging Protocol.

BitComet The first 2 bytes spell 'ex'.

Therefore, to prevent clashes we used bits in the middle of the 64-bit sequence. To indicate Merkle-torrent support, a client must set bit 42 (where the left-most bit is bit 0 and the right-most is 63).

A.3.2 Overlay-swarm Protocol Versioning

We expect considerable protocol evolution in the swarm protocol, so it will be versioned differently. An important requirement is to allow backward compatibility. That is, if the protocol has been upgraded to V2, but a V2 client can still talk to a V1 client and vice versa, this should be possible.

To indicate support for any version of the overlay-swarm protocol, a client must use the BT extension protocol. In particular, it must define the extension `Tr_OVERLAYSWARM` in the EXTEND handshake message 'm' field. Preferably the extension should have message ID 253.

Which versions of the protocol are spoken is currently communicated via the peer-ID field, as we do not use the peer ID in the overlay swarm. The versioning information will be encoded into the 20-byte peer ID field of the header as follows:

Bytes 0–15 Used as before.

Bytes 16+17 16-bits big endian unsigned integer indicating the lowest protocol version this client supports.

Bytes 18+19 16-bits big endian unsigned integer indicating the current protocol version of this client.

In general, version negotiation works as follows. Peer A initiates an overlay-swarm connection to B encoding the lowest and current versions in A's peer ID. B checks if it supports a protocol in A's range. If not, it closes the connection. If so, it does not close the connection and sends its own handshake if it did not already do so (B may send it before the check). Upon receipt of B's info A does the same check. If it does not fail, A and B will choose the highest support protocol version.

Normally, the lowest protocol field should be set to the lowest version supported. Alternatively, a client may set it lower. Consider the following example: Assume there are 3 versions of the protocol: 3,4 and 5. Protocol 3 and 5 are good, but protocol 4 is broken. Client A wants to support all the good protocols, but not broken protocol 4. A then sets the oldest protocol it can support to 3 instead of 4 and the current one to 5.

To prevent using protocol 4, A then acts as follows. Assume B supports at most protocol 4 and at least protocol 2. A initiates an overlay swarm connection with B. Via the normal procedure the candidate protocol would be 4, which is unacceptable to A. To fix this situation A closes the current connection, and then reconnects to B with the oldest protocol and current protocol set to 3 (i.e., the only good protocol that both A and B can speak).

A.3.3 Protocol History

v1 Used only internally.

v2 First public release, QLectives Platform$>= 3.3.4$

v3 Second public release, QLectives Platform$>= 3.6.0$, Dialback, BuddyCast 2.

v4 Third public release, QLectives Platform$>= 3.7.0$, BuddyCast 3.

v5 Fourth public release, QLectives Platform$>= 4.0.0$, Social Networking (SOCIAL_OVERLAP message).

v6 Fifth public release, QLectives Platform$>= 4.1.0$, Remote query, extra BC fields.

v7 Sixth public release, QLectives Platform$>= 4.5.0$, Remote monitoring and friendship making support.

Appendix B

Decentralized Recommendation Support Module

The list of content a person downloads via BitTorrent can be considered the taste of the user. There are well-known centralized techniques for using your list of downloads and those of other users to discover new content that you will want to download. An example of such a recommendation technique is *user-based collaborative filtering* [5]. We have developed a decentralized version of this algorithm that will allow software built on theQLectives Platform to make such personal recommendations.

B.1 BuddyCast Protocol

Through its downloads the user builds up a *preference list* of content. The preference list contains by default all downloaded files from which the user can add or remove entries. These preference lists are exchanged freely amongst peers using the *Buddycast* algorithm. Using this algorithm the user builds a collection of a few hundred or more of such preference lists. This collection is called the *Preference Cache*.

The recommender component uses the Preference Cache to calculate both similarity between peers and to recommend certain content the user is predicted to like, using a special collaborative filtering algorithm. When a certain peer has a preference list with high similarity to the user's they have the same download taste. We call such similar peers *taste buddies*.

Epidemic Protocol

The Buddycast algorithm is based on an epidemic protocol and roughly works as follows. Each peer maintains two lists of peers: (1) a list of its top-N taste buddies along with their current preference lists, and (2) a list of random peers. Periodically, a peer selects an entry from one of the lists and sends it its preference list, taste-buddy list and a selection of random peers. The receiving peer stores the preference list and uses the taste buddy and random peer info to update its own lists.

Furthermore, if the sending peer has downloaded some content which is of interested to the receiving peer (according to the collaborative filtering algorithm), the receiving peer may request the associated torrent file for the content from the sender, using a `GET_METADATA` message. It will also download the torrent files from some randomly selected content, to improve the spread of information through the network. The whole process is referred to as *torrent collecting*. We alternatingly select a random peer and a taste buddy to exchange with. The exact protocol is described below.

BuddyCast takes into consideration the *connectivity* of the peers. A peer is connectible if it can be reached by another peer from the Internet. An unconnectible peer can only talk to other peers if it itself initiates a connection. We found that many peers are, unfortunately, unconnectible due to extensive firewalls and the dynamic property of peer-to-peer networks. To counter this problem, BuddyCast keeps open connections with a number of peers and only uses the addresses of the peers it is currently connect to fill the outgoing message, so all the peers broadcast by a peer are online.

Initially, the links to similar peers created by the BuddyCast algorithm were used just for recommending new content in a passive way. Later, we started using the links to answer active keyword searches from the user for particular content. The rationale is still that the similar peers are more likely to have the content the user is searching for and thus keeping links to them gives a higher hit rate. To improve the keyword search capability we extended the protocol. A peer now sends a list of recently collected torrents, that is, torrents he recently retrieved from other peer or obtains from an RSS feed. This list ensures that when two random peers meet, they both can discover and exchange a fresh torrent file.

B.2 Detailed Algorithm

A peer running BuddyCast 2 keeps the following data structures in memory:

Connection List C - A list of peers to whom we keep open TCP connection.

C consists of the following 3 sublists.

- *Connectible Connected Taste Buddy List* C_T: A list of connectible peers to whom we keep a connection and which have similar tastes as us. The maximum number of peers in this list is 10.
- *Connectible Connected Random Peer List* C_R: A list of connectible peers who established connection with us most recently and are not in C_T. The maximum number of peers in this list is 10.
- *Unconnectible Connected Peer List* C_U: A list of unconnectible peers who connected to us. The maximum number of peers in this list is 10.

Connection Candidates List C_C - A list of peers which we can select as the target for a BUDDYCAST message. The maximum number of peers in this list is 100.

Block List B - It contains a number of peers which you should not contact in a period (4 hours). It includes a Send Block List B_S (do not send message to any peer in this list) and a Receive Block List B_R (discard messages received from any peer in this list).

In addition to the in-core lists, every QLectives Platform client has several database to store the information of peers, torrents and preferences it discovered in the network. We call these database the *megacaches*. Using the megacaches, a QLectives Platform client can calculate similarity between peers and recommend torrents to download.

B.2.1 Pseudo Code

When QLectives Platform starts it executes the algorithm shown in Figure B.1, and sends out `BUDDYCAST` messages periodically. When it receives a `BUDDYCAST` message, the client executes the algorithm of Figures B.2, updating its in-core lists and the megacaches. Both the send and receive algorithms use the

Table B.1: Functions in BuddyCast

Function	Description
blockPeer(Q, $block\_list$, $time$)	Add the peer Q into $block\_list$ for a period of $time$
fillPeers($message$)	Put the addresses from the indicated list in the $message$
connectPeer(Q)	Connect to peer Q
getSimilarity(Q)	Get the similarity between peer Q and myself

`createBuddycastMsg`, `addConnectedPeer` and miscellaneous methods shown in Figure B.3, Figure B.4, and Table B.1, respectively.

B.2.2 Valid Peers and Bootstrapping

The `BUDDYCAST` message requires that each client knows other online peers. After the software is installed the client needs to obtain an initial online peer. We call this process bootstrapping and use well known superpeers to solve it. The addresses of the super peers are preloaded in the client's Peer Cache.

After installation the QLectives Platform client will :

- Select a superpeer randomly from the Peer Cache.
- Connect to this superpeer via the overlay swarm.
- Send a `BUDDYCAST` message with an empty preference list.
- Receive a `BUDDYCAST` message from the superpeer with filled in random-peers list..
- Initiate Buddycast using the superpeer's random peers.

When a superpeer is sent a `BUDDYCAST` message, this superpeer will respond with random-peers list. It may also have a filled in taste buddies list and preference lists. The latter can be used to promote certain important content and bootstrap a taste network around it.

B.2.3 Rate Control

A vital part of any epidemic protocol such as Buddycast is controlling the bandwidth it uses. Within a single minute it is possible to exchange preferences with

```
1: loop
2:   wait($\Delta T$ time units) {15 seconds in current implementation}
3:   remove any peer from $B_S$ and $B_R$ if its block time was expired.
4:   keep connection with all peers in $C_T$, $C_R$ and $C_U$
5:   if $idle\_loops > 0$ then
6:     $idle\_loops \leftarrow idle\_loops - 1$ {skip this loop for rate control}
7:   else
8:     if $C_C$ is empty then
9:       $C_C \leftarrow$ select 5 peers recently seen from Mega Cache
10:    end if
11:    $Q \leftarrow$ select a most similar taste buddy or a most likely online random
       peer from $C_C$
12:    connectPeer($Q$)
13:    blockPeer($Q$, $B_S$, 4hours)
14:    remove $Q$ from $C_C$
15:    if $Q$ is connected successfully then
16:      buddycast_msg_send $\leftarrow$ createBuddycastMsg()
17:      send buddycast_msg_send to $Q$
18:      receive buddycast_msg_recv from $Q$
19:      $C_C \leftarrow$ fillPeers(buddycast_msg_recv)
20:      addConnectedPeer($Q$) {add $Q$ into $C_T$, $C_R$ or $C_U$ according to its sim-
         ilarity}
21:      blockPeer($Q$, $B_R$, 4hours)
22:    end if
23:  end if
24: end loop
```

Figure B.1: The protocol of an active peer.

```
1: loop
2:   receive buddycast_msg_recv from $Q$
3:   $C_C \leftarrow$ fillPeers(buddycast_msg_recv)
4:   addConnectedPeer($Q$)
5:   blockPeer($Q$, $B_R$, 4hours)
6:   buddycast_msg_send $\leftarrow$ createBuddycastMsg()
7:   send buddycast_msg_send to $Q$
8:   blockPeer($Q$, $B_S$, 4hours)
9:   remove $Q$ from $C_C$
10:  $idle\_loops \leftarrow idle\_loops + 1$ {idle for a loop for rate control}
11: end loop
```

Figure B.2: The protocol of a passive peer.

```
function createBuddycastMsg()
$My\_Preferences \leftarrow$ the most recently 50 preferences of the active peer
$Taste\_Buddies \leftarrow$ all peers from $C_T$
$Random\_Peers \leftarrow$ all peers from $C_R$
$buddycast\_msg\_send \leftarrow$ create an empty message
$buddycast\_msg\_send$ attaches the active peer's address and $My\_Preferences$
$buddycast\_msg\_send$ attaches addresses of $Taste\_Buddies$
$buddycast\_msg\_send$ attaches at most 10 preferences of each peer in $Taste\_Buddies$
$buddycast\_msg\_send$ attaches addresses of $Random\_Peers$
```

Figure B.3: The function of creating a BUDDYCAST message

```
function addConnectedPeer($Q$)
if $Q$ is connectable then
  $Sim_Q \leftarrow$ getSimilarity($Q$) {similarity between $Q$ and the active peer}
  $Min_{Sim} \leftarrow$ similarity of the least similar peer in $C_T$
  if $Sim_Q \geq Min_{Sim}$ or ($C_T$ is not full and $Sim_Q > 0$) then
    $C_T \leftarrow C_T + Q$
    move the least similar peer to $C_R$ if $C_T$ overloads
  else
    $C_R \leftarrow C_R + Q$
    remove the oldest peer to $C_R$ if $C_R$ overloads
  end if
else
  $C_U \leftarrow C_U + Q$
end if
```

Figure B.4: The function of adding a peer into C_T or C_R

many peers. When file downloads take days to complete it is important that no excessive amount of bandwidth is consumed by Buddycast. However, discovery of new files and new peers means that some amount of bandwidth needs to be spent.

Currently we use a simple policy to control rate: When starting for the very first time, we contact a peer every second for the first 2 minutes. For subsequent starts we contact a peer every 5 seconds for the first 30 minutes. From 30 minutes till 24 hours that we contact a peer every 15 seconds, after that once every minute. If we exchanged preference with a peer in the last 4 hours, we will not contact it again (but that peer can still connect you since it may have changed its preference).

B.3 Wire Format

The preference and taste buddy lists of a client are exchanged via the overlay swarm (see Section A.2), using a new `BUDDYCAST` message. The payload of this message contains 50 recent entries from your preference list, as well as the address information of your 10 most similar taste buddies and 10 random peers from your Peer Cache.

The exact format of the `BUDDYCAST` message is as follows. Its message ID is 249 and its payload consists of a bencoded dictionary with the following keys. All character string values are UTF-8 encoded.

'connectable' Whether I am directly reachable from the Internet (Boolean)

'ndls' My total number of downloads (integer)

'nfiles' Total number of torrents I collected (integer)

'npeers' Total number of peers discovered (integer)

'name' My name as a string.

'ip' My current IP address (string encoding, in dotted quad format).

'port' My listen port number (integer encoding).

'preferences' List of infohashes, one per preferred file (byte string). The minimum length of this list is 0, the maximum length is 50.

'taste_buddies' A list of taste-buddy records. The minimum length of this list is 0, the maximum length is 10. A taste-buddy record is a dictionary containing the following keys:

- **'preferences'** List of infohashes, one per preferred file (byte string). The maximum length is 0 (i.e., unused, see below)
- **'PermID'** Public key of the taste buddy (string encoding).
- **'ip'** The last known IP address of the taste buddy (string encoding, in dotted quad format).
- **'port'** The port number that the taste buddy is listening on (integer encoding).
- **'similarity'** Similarity between me and this taste buddy as an integer, higher meaning more similar.
- **'connect_time'** When I established a connection with this taste buddy as an integer.
- **'oversion'** The overlay-protocol version (see Sec. A.2 spoken by this taste buddy.
- **'nfiles'** Number of torrent files this taste buddy has collected as an integer (used to select peers to send remote keyword searches to).

'random_peers' List of peer addresses. The minimum length of this list is 0, the maximum length is 10. A peer address is a dictionary with the following keys:

- **'PermID'** Public key of the random peer (string encoding).
- **'ip'** The last known IP address of the random peer (string encoding, in dotted quad format).
- **'port'** The port number that the random peer is listening on (integer encoding).
- **'similarity'** Similarity between me and this random peer as an integer, higher meaning more similar.
- **'connect_time'** When I established a connection with this random peer as an integer.

'oversion' The overlay-protocol version (see Sec. A.2 spoken by this random peer.

'nfiles' Number of torrent files this random peer has collected as an integer (used to select peers to send remote keyword searches to).

'collected torrents' List of infohashes, one per recently collected file (byte string). The minimum length of this list is 0, the maximum length is 50.

After receiving a `BUDDYCAST` message a peer must directly send its own message in reply, filled with the peer's own preferences and taste buddies. After sending the reply, the peer updates its *Preference Cache* and its *Peer Cache*. We do not yet take security into account and thus simply overwrite existing entries if the age the obtained preference list is superior then any possible previous entry. If a peer encounters unknown infohashes in the preference lists it may send a `GET_METADATA` message to obtain the metadata (i.e., the torrent file) of this new content, as described below.

To keep the overlay-swarm connections to taste buddies and random peers open, (see Figure B.1) an active peer may send `KEEP_ALIVE` messages periodically. The message ID of a `KEEP_ALIVE` message is 240 and it has no body.

B.3.1 History and Open Issues

The above specification describes the BUDDYCAST message as it is in version 6+7 of the overlay-swarm protocol. Previous versions have the following changes:

v3
- Added 'connectable' field.
- Removed 'age' field from per taste-buddy/random-peer dict.

v4
- Added 'collected torrents' field.
- Added 'similarity' field to per taste-buddy/random-peer dict.
- Deprecated 'preferences' field in per taste-buddy/random-peer dict, now always empty.

v6
- Added 'npeers', 'ndls' and 'nfiles' field.
- Added 'oversion' field to per taste-buddy/random-peer dict.
- Added 'nfiles' field to per taste-buddy/random-peer dict.

B.4 Obtaining Metadata

After a preference exchange the `GET_METADATA` message is used to obtain information on a unknown infohash. The response is a `METADATA` message containing the torrent file for the given infohash. Since this mechanism was added to QLectives Platform in 2005, an official BitTorrent extension has been proposed for obtaining the torrent file of a swarm. See [8].

The exact format of the `GET_METADATA` message is as follows. Its message ID is 248 and its payload consists of a bencoded infohash.

The exact format of the `METADATA` message is as follows. Its message ID is 247 and its payload consists of a bencoded metadata record. A metadata record is a dictionary with the following keys:

'torrent_hash' The infohash of the torrent (byte string)

'metadata' The torrent (byte string)

'last_check_time' Time of last check at the torrents tracker for how many peers are in the swarm, UTC in seconds as integer.

'status' String indicating the status of the checks:

- **'good'** The tracker is responding.
- **'dead'** The tracker has not responded on a number of tries.
- **'unknown'** The tracker is flaky.

'leecher' Number of downloaders at last check.

'seeder' Number of seeders at last check.

The check fields were added in version 4 of the overlay-swarm protocol.

Appendix C

Remote Query Module

The BuddyCast protocol establishes connections between a user and its taste buddies, see Sec. B.1. Initially, these connections were established to allow meaningful recommendation of new content to users following the principle of collaborative filtering. This principle says that if two people, A and B, have a similar taste in content, then any new content that A downloads is also likely to be appealing to B and vice versa. At the moment, taste buddies periodically exchange information about new content which then results in a list of recommended items for the user. As a next step we now use these connections to taste buddies to implement an efficient search mechanism for content. If a user wants to watch some new content that he heard about, but it is not on his list of recommended items yet, he can now do an explicit search of the databases of his taste buddies to see if they have already found this content. This mechanism, known as semantic-overlay search, has been shown to yield high hit rates [16].

The remote search mechanism queries the megacaches of the peers you are currently connected to (as a result of the BuddyCast protocol). To query a peer's database, the client sends an overlay-swarm `QUERY` message to the peer, containing a query ID and a query specification. The receiving peer checks if the sender has not exceeded the quota for `QUERY` messages. For senders who are marked as friends by the receiver's user, the quota is unlimited. For unknown senders there is a 100 query quota. If the quota has not been exceeded, the receiver parses the query and executes it on its megacache. The results are then sent back in an overlay-swarm `QUERY_REPLY` message that carries the same query ID and a set of answers. At the moment, queries are limited to simple keyword searches in the receiver's torrent database, but it can be extended to full-fledged SQL-like queries in the future.

When the query results come in, they are displayed to the user. When the user decides to download one of the found torrents, the user clicks on the result, and its client then sends a `GET_METADATA` message to the peer that returned the result. The peer, if still online, will then return the desired .torrent file in a `METADATA` message. This is the same mechanism for obtaining a torrent file from a peer as used in the Cooperative Download feature, see Sec. D.

Appendix D

Cooperative Downloading

When there are few seeders in BitTorrent, your download speed is equal to your upload speed. As most people have an asymmetrical network connection with a maximum download speed that is larger than the upload speed, the downlink is often not fully exploited. If you have a group of friends whose connections are idle, they can be used to fill the downlink. For example, the friends can each download a piece of the file that you not yet have using standard BitTorrent. Once the piece is received, they then send it to you over your underutilized downlink without expecting any data in return. So by doing barter-free downloads with your friends you can utilize your asymmetric network link to its fullest.

Protocol

Peers from a social group that decide to participate in a cooperative download take one of two roles: they are either *coordinators* or *helpers*. A coordinator is the peer that is interested in obtaining a complete copy of a particular file, and a helper is a peer that is recruited by a coordinator to assist in downloading that file. Both coordinator and helpers start downloading the file using the classical BitTorrent tit-for-tat and cooperative download extensions. Before downloading, a helper asks the coordinator what piece it should download. After downloading a file piece, the helper sends the piece to the coordinator without requesting anything in return. In addition to receiving file pieces from its helpers, the coordinator also optimizes its download performance by dynamically selecting the best available data source from the set of helpers and other peers in the BitTorrent network. Helpers give priority to coordinator requests and are therefore preferred as data sources.

The Protocol in Detail

To invoke the help of a friend, the coordinator opens an overlay-swarm connection (see Section A.2) to the helper and sends a `DOWNLOAD_HELP` request. When the `DOWNLOAD_HELP` message is received, and the helper is willing to help, it obtains the torrent file to use from the coordinator using a `GET_METADATA` message. After receiving the corresponding `METADATA` message, the helper establishes two connections with the coordinator: a *control connection* and a *data-exchange connection*.

The control connection is used by the helper for claiming pieces at the coordinator. Control connections are blocking; the helper can block waiting for the response from the coordinator. Only the control messages `RESERVE_PIECES` and `PIECES_RESERVED` are sent over the control connections. The pieces of the file the helper downloaded on behalf of the coordinator are transferred over the data-exchange connection. The control connection was introduced to make sure the helper immediately knows whether a particular piece was claimed elsewhere or not.

Whenever helper H wants to download a piece P, H first contacts the coordinator and tries to reserve (claim) P using a `RESERVE_PIECE` message. If P was not claimed by anybody else, the coordinator sends a `PIECE_RESERVED` message in return, and H, in turn, sends requests for piece P to the offering peer in the swarm. Otherwise, H checks its download bandwidth utilization. If the amount of unused download bandwidth is above a certain threshold, H requests P (although P was claimed by some other helper).

In order to decrease the number of messages exchanged between the coordinator and its helpers, the coordinator from time to time appends a list of all pieces that have been already claimed by others to the `PIECE_RESERVED` reply to a helper. This optimization greatly improves the performance in the later stages of the download when most of the pieces have already been claimed, and only a few still have to be downloaded. With this list, the helper can then determine locally which pieces have been already obtained without asking coordinator for the status of each piece separately.

The coordinator decides to download a missing piece from either one of its helpers or any other peer in the swarm using the standard BT peer selection mechanism. A helper which is getting a `REQUEST` message for a piece from the coordinator puts this request in front of its sending queue, consequently giving them the highest priority. The connections between helper and coordinator are never choked.

Appendix E

NAT/Firewall Detection module

Many of the clients run on a machine with or behind a firewall or Network Address Translator (NAT). This poses several problems:

1. Unless the QLectives Platform listening port (7762 by default) is opened on the firewall, other peers cannot connect to it.

2. The QLectives Platform client can no longer obtain the IP address via which it is reachable on the Internet from the operating system. As a result, it is not able to provide this address to others.

To solve these problems we have extended QLectives Platform with a facility for detecting a firewall, and discovering a client's external IP address. In particular, we added two messages to the overlay-swarm protocol called the *dialback messages*. These messages are used as follows. At client startup, 7 peers are selected from the database of encountered peers. This database is initially filled with the addresses of the 8 QLectives Platform superpeers. The client attempts to send a `DIALBACK_REQUEST` to each of the 7 peers using the overlay swarm.

When a peer B receives a `DIALBACK_REQUEST` it closes the existing overlay-swarm connection. It then tries to connect back to the initiating peer A. In particular, it will try to connect back to the IP address X that initiated the previous connection and the listen port that peer A specified in the BitTorrent handshake message (see Sec. A.2). If the connection succeeds, B sends an `DIALBACK_REPLY` message containing the IP address X it used to connect. Peer B thus informs the initiating client A that (1) the client is reachable from the Internet and (2) what its external IP address is (which is X).

To protect against malicious peers, the client will record the external IP addresses returned by the 7 peers and select the address the majority agrees on as being its real address. If there was no majority either because not enough peers replied or they disagreed, the client start the process over again with 7 other peers after 30 seconds. The client will retry 5 times, so contact at most 35 peers.

In order to improve reachability of peers the client warns the user. The user interface clearly indicates when QLectives Platform is not reachable and that port forwarding should be turned on on its firewall for full performance.

Appendix F

Reputation System module

The current reputation system in the QLectives Platform is based on three parts: ModerationCast, VoteCast and Metada dissemination. In conjunction, these three parts allow attaching a reputation to media objects shared in the system.

F.1 ModerationCast

ModerationCast deals with spreading/dissemination of the metadata of torrents that users add. ModerationCast has three kinds of messages:

MODERATION_HAVE message which signal to other nodes a set of available moderators stored in the megacache that can be sent if required.

MODERATION_REQUEST messages which are a reply to a `MODERATION_HAVE` message listing which moderations are required.

MODERATION_REPLY messages which contain the actual metadata for the requested moderations as previously listed in the `MODERATION_REQUEST` message.

When a peer A connects to another peer B, `MODERATION_HAVE` message is sent, along with BuddyCast (if there are any moderations). B checks whether it has that moderation or not and if so, whether it is newer than the existing moderation in which case B sends a `MODERATION_REQUEST` message. A sends the moderation, in `MODERATION_REPLY` message, which would either insert the new record or overwrite the older one in the receiving peer.

F.2 VoteCast

VoteCast builds on BuddyCast and ModerationCast. Through ModerationCast, nodes may inject and propagate metadata bound to a hash of the torrent. Users may vote on a moderator in either a +ve (real) or -ve (fake) way or not at all. ModerationCast uses these votes to determine whether to receive or pass on moderations (i.e. metadata associated with a moderator).

The task of VoteCast is to allow nodes to collect these votes from other peers they encounter (via BuddyCast) in a local BallotBox. This allows peers to determine how popular or unpopular a moderator is by counting votes. This can then be used for relevance ranking after keyword search.

F.3 Metadata Dissemination

Moderations are disseminated in a gossip-like fashion to other peers. However, nodes only pass on metadata from those moderators they have approved. Approval involves the user explicitly selecting a thumbs-up icon displayed next to the metadata from the given moderator indicating a positive (+) vote for the moderator. Users may also disapprove of a moderator by selecting a thumbs-down indicating a negative (-) vote.

Over time as nodes encounter others, through gossiping, they will receive new moderations either directly from the moderator, if they encounter them, or from those nodes which have approved the moderator. Received moderations are stored in a local database. Hence highly approved moderators will tend to spread their metadata more quickly than moderators that are not highly approved. If no other node approves a moderator then the only way that its metadata can spread is through direct contact with other nodes. Nodes that disapprove a moderator remove all associated moderations from their local database and refuse any new moderations from that moderator.

Essentially then, the idea is that, “good” moderators, as judged by the approval of others, will spread their metadata quickly but “bad” moderators, obtaining low numbers of approvals and / or disapproval, will only be able to spread their metadata slowly. However, it is important to note that even bad moderators can spread their data to others through direct interactions with nodes

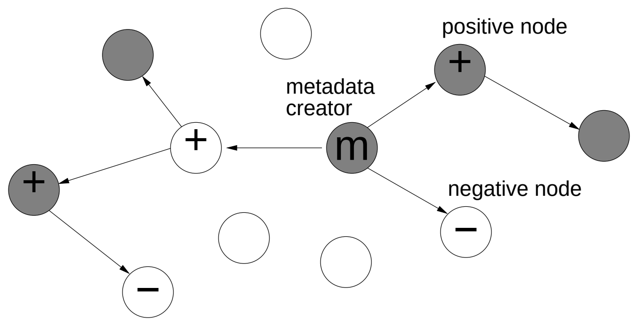

Figure F.1: Spreading of moderations based on approvals and disapprovals.

that have not already indicated disapproval. Figure F.1 shows a schematic diagram showing how a moderation spreads in the population based on approvals and disapprovals by other nodes.

Because moderators can be blocked based on their moderations and forwarders are forwarding moderations for (other) moderators, there is need for authentication of moderations. This is done using public key signatures. This prevents alteration of moderations, which could lead to bad moderations spreading very well and good moderators being blocked because of malicious peers forwarding altered or fabricated moderations on behalf of the good moderator.

In order to discourage spam voting through the creation of many cheap identities VoteCast only counts votes from peers with a suitable reputation level which is supplied by our BarterCast extension. Hence, although identities are cheap, votes are not. This mechanism is fully covered in [12].

Appendix G

Social Networking module

The social networking features of the QLectives Platform consist of a message for exchanging nickname and avatar pictures and a new mechanism for friendship establishment. This mechanism does not require the two parties involved to be online at the same time. It also does not require a central component to achieve this asynchronicity.

G.1 Functionalities

Nickname and Picture Exchange

A user's public name is the name chosen by the user by which he will be publicly known. As it is chosen by the user, it is not necessarily system-wide unique. A user's public picture is a picture chosen by the user that will be shown to other users.

Friendship Making

The peer who initiates a friendship request to another peer is known as the source peer, and the peer for whom this request is intended is known as the target peer. The functionalities provided to the users are the following:

Adding new friends In order to build a social circle, a peer can request other peers discovered by the underlying peer sampling service (PSS), which are potential candidates for being friends, to become their friends. The target peer has to reply to the friendship request sent by the source peer, and if the reply is positive, both the peers become friends.

Removing friends The source peer removes the target peer from its friends list. Also, it requests the target peer to remove it from its list.

Maintaining status of friends The system must keep peers up-to-date about the online status of their friends.

G.2 Basic request-reply protocol

For establishing a friendship link, the mechanism follows the request-reply notion. The source peer initiates it by sending a friendship request to the target peer. The target peer then takes its decision by accepting or rejecting the friendship request, and send its reply back to the source peer. If the reply is positive, both the source and the target peer become friends.

G.3 Unavailability of the peers

In order to deal with the unavailability of both the source and the target peer, we have designed two mechanisms, which work for both friendship requests and friendship replies, which we discuss below.

Retry

If the target peer is not online, the source peer will retry to connect to it after every five minutes, in case the target peer comes online. Similarly for receiving the reply from the target peer, if the source peer is not online, or unconnectable for some reason, the same retry mechanism is adopted by the target peer to dispatch its reply to the source peer. The initial retry time interval of minutes is increased to 24 hours, after one day has passed since the friendship request/reply was initiated. After a week of unsuccessful delivery of requests or replies, all pending friendship messages (requests and replies) are dropped from the source and the target peers. In order to increase the chances of contacting the other peer, both the source and the target peers save messages that could not yet be successfully delivered, i.e., the pending messages (requests and replies), in case they are going offline. In their next session, both of them read these messages and then dispatch them.

Helpers

To increase the chances of establishing a friendship link between the source peer and the target peer, we have introduced the concept of helpers. Helpers are online friends and taste buddies of the source peer, in case of friendship requests. And in the case of friendship reply, they are online friends and taste buddies of the target peer. When the source peer is unable to connect to the target peer for requesting friendship link establishment, it dispatches its friendship request to these helpers. Helpers then also try to contact the target peer every five minutes. Helpers also used by the target peer for forwarding its friendship reply to the source peer, in case it is unable to contact it. Helpers, just like the source and the target peer, also save the unsuccessful friendship requests/replies locally when they are going offline. On their next startup, they try to deliver them to the intended peer.

G.4 Scenarios of establishing friendship links

Depending upon the availability of the source and the target peer, we distinguish different scenarios for establishing a friendship link between them. Note that these scenarios only show the friendship request part. The reply part follows the same scenarios. The possible scenarios of friendship link establishment between the source and the target peer are the following:

- Scenario 1: Both the source and the target peers are online. The source peer directly sends the friendship request to the target peer. Depending upon the target peers response, it is added to the source peers friends list.
- Scenario 2: The source peer is online, but the target peer is not. The source peer after an unsuccessful attempt to connect to the target peer, employs the retry mechanism mentioned above, involving both itself and the helpers.
- Scenario 3: The source peer has gone offline after initiating the friendship request, but the target peer is online. Since the source peer can not connect

to the target peer, it dispatches the friendship request to its helpers. The helpers then connect to the target peer and forward the friendship request to it.

G.5 Possible attacks prevention

In the current design, it is possible for malicious peers to target and subvert the system. There are two main possible attacks, which are Distributed Denial of Service (DDOS) and a special DDOS, man in the middle attack. In order to thwart such potential attacks, we have established certain safeguards which we shall detail below along with an explanation of the attacks. DDOS is a type of attack where a peer is asked by a huge number of other peers for some service. The motive behind this attack is to overload a peer so that even legitimate peers are unable to access it and get its service. For the DDOS attack, we restrict a user, who is running the client from a binary, to make at most 10 friends per day. We cant, of course, overcome this problem if a user has modified our source code. This restriction can be accomplished fairly easily as all the friendship requests will be recorded by the system. In the man in the middle attack, a helper tries to overload a peer, or a group of peers, with a huge number of illegitimate friendship requests. To counter this, we have devised a solution by incorporating the use of signed requests: the peer who initiates the request (source peer) first signs it with its private key. This would allow the receiver (target peer) to determine that it is indeed from the source peer. Since only one instance of a Tribler client can run on a single machine, no malicious peer can fake or develop multiple instances, and thus multiple identities.

Appendix H

Channels module

The original BitTorrent protocol excludes mechanisms for searching, rating, and associating descriptive metadata with, content. The TUDelft research group, within the collaborating project P2P-NEXT, has already proposed a design for fully distributed metadata dissemination and rating system which allows users to locate and browse available content conveniently from within the client before downloading (see the Appendix G for details). This idea has been redesigned around the concept of a channel. Low quality metadata such as spam or incorrect information is combated through a distributed rating system based on the sampling of user votes (or ratings) in favor (or against) those peers who submit metadata, who we now term channels (also referred to as moderators interchangeably), and the metadata they submit as moderations.

ChannelCast message is used in gossiping the moderations of channels, both own and subscribed. Each ChannelCast message consists of bunch of recent and random moderations of these channels. When a peer sends a ChannelCast message, the receiver checks its own database to see if each received moderation is present. If a moderation is not found, the receiver adds this record to the database and subsequently requests (to the sender) for torrent file corresponding to this moderation. In this way, both metadata and content of channels are spread in the system. ChannelCast message is sent along with BuddyCast message; hence, the load and frequency of dissemination are dependent on BuddyCast interval.

Inspired by ModerationCast (see Appendix G), ChannelCast extends its idea (as well as replaces it), but differs in following features. First, instead of three-message protocol of ModerationCast (Have, Request and Reply), ChannelCast uses only one message. Second, unlike in ModerationCast where moderations alone are sent, moderations are transferred along with torrents (if not present at receiver end). Third, along with ‘infohash’, ’torrenthash’ (hash of bdecoded dictionary of the whole torrent) is also a part of moderation record; this solves the problem of a fake-tracker attack, where the tracker field(s) of the torrent (i.e., ’announce’ / ’announce-list’) are changed to fake address which corrupts the torrent, despite its infohash being unaltered.