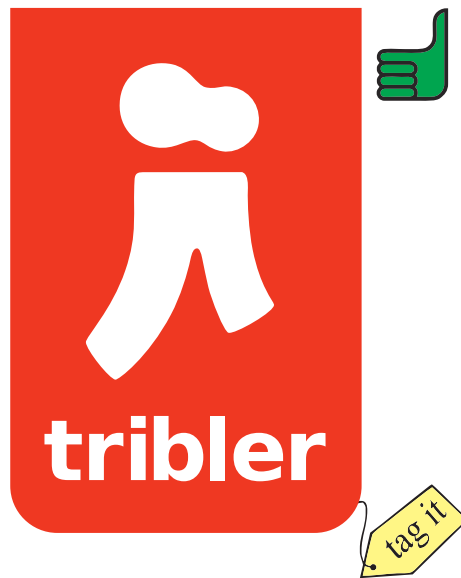


Metadata Infrastructure for Peer-to-Peer Video



Vincent Heinink



Delft University of Technology

Metadata Infrastructure for Peer-to-Peer Video

Master's Thesis in Computer Science

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

Vincent Heinink

2nd June 2008

Author

Vincent Heinink

Title

Metadata Infrastructure for Peer-to-Peer Video

MSc presentation

16th June 2008

Graduation Committee

prof. dr. ir. H. J. Sips (chair)	Delft University of Technology
ir. dr. D. H. J. Epema	Delft University of Technology
dr. ir. J. A. Pouwelse	Delft University of Technology
dr. P.H. Westendorp	Delft University of Technology

Abstract

Traditionally, the Internet has been used for text-transfer. Especially the World Wide Web was used for this purpose. However nowadays more and more rich content such as videos are shared using the Internet [6]. This data however is much harder to search through because of its lack of structure and much larger size. Metadata, extra data which is smaller and easier to navigate, can help in browsing and searching through the videos.

In order to provide good quality metadata, with a scalable and bandwidth efficient architecture, we present MODERATIONCAST. MODERATIONCAST uses a combination of Wikipedia-like editing and a peer-to-peer architecture. With this approach we hope to solve the quality issue using peer production [5]. Further we solve the lack of scalability that is imposed by centralized systems. In the design and implementation of MODERATIONCAST, we have taken into account; scalability, bandwidth efficiency and security. Further, we have taken countermeasures against pollution.

For evaluation of MODERATIONCAST, we have created a trace-based simulator. We have used this simulator to conduct large scale trace-based simulations of moderation-distribution in a 2000 peer Tribler network. From our results we can conclude that MODERATIONCAST has high scalability, bandwidth efficiency, availability and deals with pollution.

“Computer science is no more about computers than astronomy is about telescopes.” – prof. dr. Edsger W. Dijkstra

Preface

This document describes my MSc project on the subject of metadata infrastructure for peer-to-peer video. The research was performed at the Parallel and Distributed Systems Group of the Faculty of Electrical Engineering, Mathematics, and Computer Science of Delft University of Technology. This research is done in the context of the I-Share project, in which research is conducted on the sharing of resources in virtual communities for storage, communications, and processing of multimedia data.

First I would like to thank my supervisor Johan Pouwelse for his guidance and support during my thesis work. He was always passionate about Tribler and able to motivate people. Also I would like to thank Michel Meulpolder for giving feedback on my literature study. I have not met someone who can illustrate complicated issues as clearly as he can. Jie Yang, thank you for the insight you gave me in Buddycast and the use of your simulator. I admire your work and this work would not have been possible without you. Jelle Roozenburg, thank you for letting me use your Filelist.org traces. It must have been a mammoth task to gather them. Rameez Rahman and David Hales, thank you for the feedback on the drafts of my thesis. Jan David Mol thank you for the various discussions we had. Finally I would like to thank prof. dr. ir. H. J. Sips for chairing the examination committee and ir. dr. D. H. J. Epema for participating in the examination committee.

Vincent Heinink

Delft, The Netherlands
2nd June 2008

Contents

Preface	vii
1 Introduction	1
1.1 Peer-to-peer systems	2
1.2 BitTorrent	4
1.3 Tribler	5
1.4 Video metadata	6
1.5 Contributions	6
1.6 Thesis outline	6
2 Problem Description	7
2.1 Metadata for Video	7
2.2 Requirements	8
2.3 Related work	8
3 Design	11
3.1 Metadata element set	11
3.2 Direct spreading approach	12
3.3 Forwarder-based approach	15
3.4 On-demand downloading approach	16
4 Implementation	19
4.1 Technical design	19
4.2 Implementation issues	22
5 Trace based simulation	27
5.1 Simulation Setup	27
5.2 BitTorrent Trace	28
5.3 Buddycast simulator	29
5.4 Moderationcast simulator	31
6 Results	33
6.1 Policy experiments	33
6.2 Parameterize the number of forwarders	36

6.3	Parameterize the number of peers	41
6.4	Pollution removal	42
7	Conclusions and Future Work	47
7.1	Summary and Conclusions	47
7.2	Future Work	48

Chapter 1

Introduction

Traditionally, the Internet has been used for text-transfer. Especially the World Wide Web was used for this purpose. However nowadays more and more rich content such as videos are shared using the Internet [6]. This data however is much harder to search through because of its lack of structure and much larger size. Metadata, extra data which is smaller and easier to navigate, can help in browsing and searching through the videos.

Metadata has been used in central video systems such as YouTube [41]. The quality of this metadata however is rather low. This is mainly because only the uploader of a video can add metadata. A system in which everyone can contribute, is Wikipedia [39]. Although this is an online encyclopedia, it clearly demonstrates the potential of this “everyone can contribute” strategy. Within the first four years after its foundation, 3.7 million articles have been added to this encyclopedia [16].

However, because of their central nature such systems have limited scalability, availability and fault-tolerance. Peer-to-peer systems, in which consumers also share their resources with others, do not have these limitations [31]. One of the most used peer-to-peer systems is BitTorrent [18]. The Parallel and Distributed Systems group of the Faculty of Electrical Engineering, Mathematics and Computer Science has designed, implemented and deployed a special BitTorrent client called Tribler [35], which focuses on video.

The aim of this thesis is to design, implement and evaluate a peer-to-peer metadata infrastructure for video. To get good quality metadata we will use a similar strategy to what Wikipedia uses. This system will be integrated to Tribler as a proof of concept.

This Chapter provides an introduction to peer-to-peer systems and video metadata. Section 1.1 gives an explanation of peer-to-peer systems in general. Section 1.2 discusses in more detail the BitTorrent peer-to-peer system and Section 1.3 focuses on Tribler. Section 1.4 explains the concept of metadata. Section 1.5 then lists the contributions we make in this thesis and Section 1.6 provides the outline of this thesis.

1.1 Peer-to-peer systems

The traditional architecture on the Internet is client-server (see Figure 1.1(a)). In the client-server model, a server offers a service to clients. This approach is widely used as it is easy to update the service or content in one place. A disadvantage of this architecture is that it is inherently unscalable and has a single point of failure. As more clients request services simultaneously the server uses more resources and there will be a time at which the server runs out of capacity. When it does, the response time will be slower and/or some clients may not get any service at all. To solve this problem, often more than one server is used. These copy the content from the main server and then provide service to certain clients. Which server offers service to which client can be based on server load, geographical placement and other parameters.

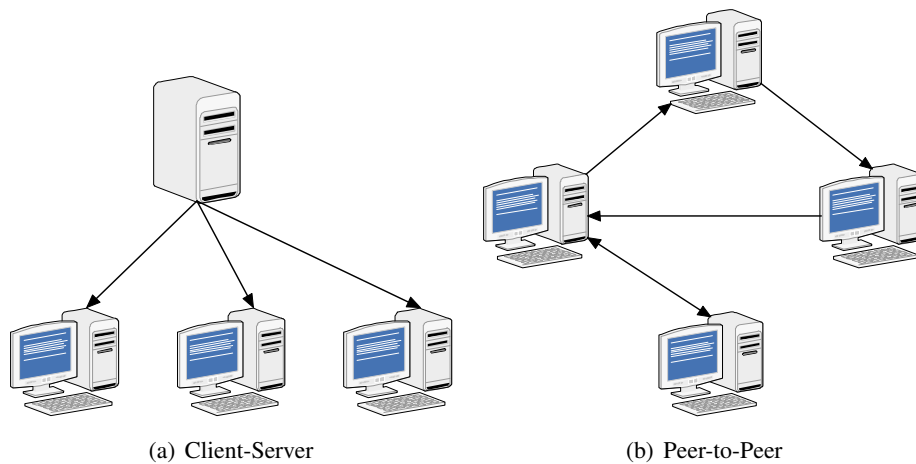


Figure 1.1: Network connections in a client-server architecture (a) and a peer-to-peer architecture (b).

Although using mirror sites may be a good policy for some services, other services are very resource intensive and may require an altogether different architecture.

For some services the cost of having extra servers, which are only used during peak-usage, may also be prohibitive. An architecture which is inherently more scalable and does not have a single point of failure is the peer-to-peer architecture (see Figure 1.1(b)). In this architecture, all the computers in the network are both servers and clients and are considered equals. This way, every time a client joins the network, it brings with it extra server-resources. This scales better than the client-server architecture. Because the resources are often geographically dispersed this also leads to better fault tolerance.

The peer-to-peer architecture however has its downside. Due to the non-centralized approach, there are often problems with trust and sharing resources. There are many examples of these problems, such as companies providing polluted content to fight copyrighted material from being spread and freeriding (contributing little or no resources while using the system). To solve these problems voting/pollution removal-measures and incentives to share need to be in place. This, together with distributing the design over all peers, makes the design of peer-to-peer systems much more challenging than client-server based systems.

Many systems with a (partial) peer-to-peer architecture exist today. Skype [33] for instance is very successful in using the peer-to-peer architecture for online telephony. Although peer-to-peer systems can be used for various purposes, file-sharing is the most dominant usage. Ipoque Internet Study 2007 [18] shows that between 49% and 83% of the Internet backbone is used by peer-to-peer systems, depending on the region (see Figure 1.2(a) for Germany for instance). Of this between 62% and 79% is used for video (see Figure 1.2(b) and note that the blue parts are all video). The most popular peer-to-peer filesharing system is BitTorrent.

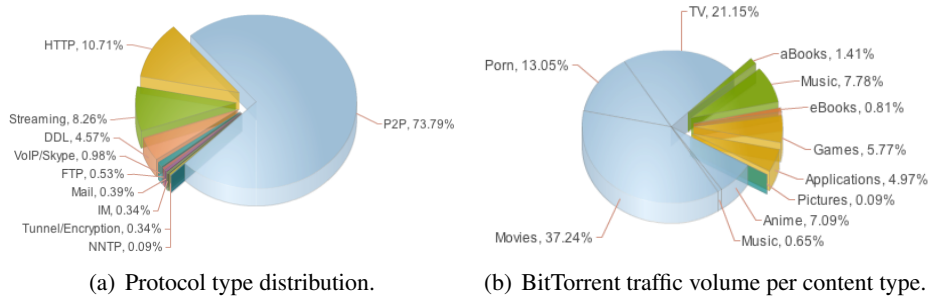


Figure 1.2: Ipoque Internet Study 2007, results for Germany [18].

1.2 BitTorrent

BitTorrent is a peer-to-peer filesharing system designed and implemented by Bram Cohen in 2003 [10]. BitTorrent groups peers downloading the same content in a swarm. Peers in a swarm can then exchange pieces of the content, a process called bartering. Because multiple pieces are bartered with multiple peers concurrently and these pieces are directly available to other peers, peers can quickly download a first piece and start bartering. This makes the protocol very efficient. Also the protocol is not hindered by peers with different available bandwidth, which would be the case if one peer could only download from one other peer concurrently.

The pieces which are bartered are selected using a “local rarest piece first” policy. This policy tries to maximize the availability of the most rare piece. This is very useful as this maximizes the availability of the content. To guarantee fairness, the peers with whom one barterers are selected using tit-for-tat. This policy uploads pieces to peers from whom we receive a lot of pieces. This hinders freeriding, because freeriders contribute very little and as a response will be last in queue to upload to.

To join the swarm in which certain content is being bartered, a new peer has to acquire TCP/IP-addresses of other peers in the swarm. To this end a central component, called a tracker is used. The tracker keeps track of which peers are online in a certain swarm. Whenever a peer requests, it will give a random set of TCP/IP-addresses of peers in the swarm. The peer can then connect to them and start bartering. Because of lack of scalability various attempts have been undertaken to substitute this central component with a distributed one [23, 21, 2, 1]. Most of these systems use a Distributed Hash Table (DHT). A DHT is a distributed system in which peers provide a lookup service similar to a hash Table. That is, given a key a value is given. Using this system a peer can request a set of peers given the infohash of the torrent.

To find the tracker and ID for certain content, torrent-files are used. These torrent-files also contain cryptographic hashes of all the pieces. This enables peers to detect alterations in pieces they have downloaded. If this occurs, the pieces can be re-downloaded from other peers. Finally to download the torrent file for certain content, there are websites, which host the torrent files together with a short description (which may also be part of the torrent file).

1.3 Tribler

After years of research on peer-to-peer networks and BitTorrent communities, the Parallel and Distributed Systems group of the Faculty of Electrical Engineering, Mathematics and Computer Science initiated the design, implementation and deployment of a new peer-to-peer client named Tribler [35, 27]. Tribler uses the BitTorrent file-sharing protocol, but adds a lot of extra features to the protocol.

The main philosophy behind Tribler is that peers should be non-anonymous. To this end each peer creates a unique PermID, which remains the same even if the user goes offline or changes its TCP/IP-address. This enables the software and the people using it to build relationships with other peers. Using these relationships, extra services and better efficiency can be provided. Because people are now valuing peers based on their PermID, these PermIDs need to be non spoofable. This is done using public key encryption. Every time a peer connects to another peer, the connection is authenticated using a public key challenge response [26].

One of the core services using these non spoofable permanent identifiers is Buddycast [38]. Buddycast is an epidemic protocol. In an epidemic protocol (also called gossiping protocol), a peer periodically selects another peer, from all the peers in the network, to exchange information with [36]. Because the peers keep “infecting” each other with information, the speed at which information propagates through the network can be very high. If this is done methodically, n peers will all have the entire information that all the peers had combined, with only $\log_2 n$ exchanges. However, for this approach to work peers need to keep track of the addresses of peers which are part of the network. Keeping complete membership tables organized on each node is infeasible for large-scale, highly dynamic networks. Such an effort would impose tremendous synchronization problems, especially in systems with high churn, such as peer-to-peer systems [34]. Therefore a different approach is taken, in which peers only exchange information with a small subset of peers, the neighbor set. Also the information that is exchanged includes membership information. After every exchange the peers can use this membership information to change their neighbor set before exchanging information with the next peer. In doing so, peers can self-organize into topologies that better suit certain applications [36]. Buddycast uses the download history of the user to organize itself in a topology which is partially random and partially clustered with people that have a similar download history. Using this topology torrent files are exchanged based on their peer-relevance-ranking. In Chapter 3 we present the design of MODERATIONCAST, a metadata infrastructure for Video that uses Buddycast.

1.4 Video metadata

Metadata is data about data. The word meta comes from Greek, where it means “after” or “beyond” [40]. These two terms give two hints about what metadata is. It is data, describing certain other data, which is created after the described data. For a movie this includes; the title, its length and the actors. This metadata is useful if one needs to search through a collection of large data. One could try to search the metadata and would not have to search through the collection of relatively large data. In peer-to-peer systems that are used for sharing good quality video, this metadata can help a peer to find relevant video-files in a (large) collection of large video-files without downloading the complete collection.

1.5 Contributions

In this thesis, we make the following contributions:

- We present MODERATIONCAST, the first distributed Video metadata infrastructure in which people can contribute in a web 2.0 fashion. It is scalable, bandwidth efficient and deals with pollution.
- We develop a simulator for MODERATIONCAST, which we use to evaluate the design. This simulator uses real-world traces and copes with large scale simulations.

1.6 Thesis outline

The remainder of this thesis is organized as follows. Chapter 2 describes the problem of distributing metadata in a peer-to-peer architecture. It also defines design requirements for such an architecture and discusses systems and standards that are related to our work. In Chapter 3 we outline the design of our new metadata infrastructure for Video. Chapter 4 presents an overview of the implementation, focusing on the major issues. Chapter 5 presents traces gathered from [14], which together with a Buddycast simulator and a newly created simulator for MODERATIONCAST are used to evaluate the design of MODERATIONCAST. In Chapter 6 we evaluate our design using the experiments outlined in this Chapter and their results. We conclude and present future work in Chapter 7.

Chapter 2

Problem Description

This Chapter defines the problem that is at the heart of this thesis: the distribution of rich metadata, for Video in a peer-to-peer architecture. In Section 2.1 we describe the problem of providing Video metadata in a peer-to-peer architecture. In Section 2.2 we define design requirements for a scalable and pollution free metadata infrastructure for Video. In Section 2.3 we discuss systems and standards that are related to our work. Where applicable, we will review requirements that we define in Section 2.2, that these systems meet. We conclude that currently there is no metadata infrastructure, that meets the requirements defined in Section 2.2. Therefore we will design a new video metadata infrastructure in Chapter 3.

2.1 Metadata for Video

In peer-to-peer file sharing systems, a large number of videos are available. This gives the users lots of choice. However, determining whether or not to download certain videos becomes a problem. Downloading the wrong video results in extra bandwidth usage. In centralized systems, such as YouTube [41], finding the right video is less of a problem. The videos can often be watched on-demand, enabling the user to rate the video immediately. These sites also allow users to post comments. The person that uploads the movie can also add extra information to the video. This extra information, which is available prior to playback (metadata) can be very useful to determine if one wants to watch a certain video.

In peer-to-peer systems very little metadata is available. The reason for this is that in order to achieve higher availability, scalability and fault-tolerance than a centralized system, peer-to-peer systems introduce redundancy. Keeping the meta-

data, which can be added or changed, up-to-date has been shown to be a problem because of this redundancy [3]. Further, to adhere to peer-equality, peers should *a priori* have equal rights to add or change metadata. This allows peers to contribute to the system, a phenomenon described as peer-production [5]. However, as shown by Wikipedia, policies should be in place to keep pollution to a minimal level.

2.2 Requirements

We define four design requirements that have to be met in a good quality metadata infrastructure for peer-to-peer Video. Using these requirements, we will review related work. Further, we will use them as a motivation for the design of our infrastructure. These design requirements are:

Good quality metadata Without good quality metadata a metadata infrastructure is useless. Wikipedia has shown that *a priori* allowing all the users to contribute to the system can create valuable information. This way a lot of metadata can be generated. However, people should have a reason to create and distribute metadata. Without a reason, very few people will contribute to the system. Given the fact that pollution is a real issue on the Internet, policies to hinder pollution should be in place.

Scalability The infrastructure must scale to many thousands or even millions of users. This should be possible without adding extra dedicated resources, because putting them in place is costly and requires time. Recent studies have shown that flash crowds are very common on the Internet in general and peer-to-peer systems in particular [34].

Bandwidth efficiency Average users have relatively limited bandwidth. Because metadata is an extra, rather than a core feature for watching videos, the infrastructure for distribution should require little bandwidth overhead.

2.3 Related work

In this Section we describe work related to our thesis. There are various systems that have functionality that can be compared to that of our infrastructure. There are also standards for metadata and systems with a comparable architecture. For this related work we will indicate how it relates to our work, while focusing on how it differs from our work.

YouTube YouTube [41] is a website on which people can post their videos. It is considered a centralized system, as a fixed number of websites serve these videos. In centralized systems, it is easy to distribute metadata together with video. The metadata is hosted together with the video on the web servers. Changing the metadata in these systems is also relatively easy. However, the costs of operating such a system are very high, and the scalability is limited as resources are shared by all users. Also due to the centralized nature of these systems there are single points of failure. Our work differs from this system in that it follows a peer-to-peer architecture. Peer to peer systems do not have such limitations. However, distributing metadata in peer-to-peer systems is much more difficult as there is no single point where the metadata can be stored. This redundancy also implies that changing metadata becomes less straightforward.

Dublin Core Metadata Element Set The question what are useful and objective metadata-elements, has been studied by many and has lead to little consensus. A lot of standards exist today. One of the dominant metadata standards is the Dublin Core Metadata Element Set [11]. Specified in 1995, it consists of 15 elements which address the most basic descriptive, administrative and technical elements required to uniquely identify a digital resource [17]. However, this set of metadata-elements is very generic and although it can be extended, it would require a lot of work to be tailored to video. Therefore we have used more practical metadata-elements in our metadata-element set although we have several elements in common.

Wikipedia Wikipedia [39] is a website on which people can create articles. The articles on this site can then be viewed by every user and can also be edited, again by any user. This approach is so popular that by now Wikipedia is the most used and largest encyclopedia in the world. Although very controversial, because of the lack of editors responsible for the quality of contributions, it is generally agreed that on average the articles are accurate most of the time [16]. To further improve the accuracy of Wikipedia, the Wikimedia Foundation, has introduced policies to temporarily or permanently protect certain articles from being edited. Our metadata infrastructure for video will also allow for all the users to create and change the content. We hope that this will lead to the same results as Wikipedia. We will however also have similar problems with people that want to vandalize the content. Because of the peer-to-peer architecture, enforcing policies in our system is much more difficult.

Emule Emule [12] is a client for the eDonkey2000 and Kademia [23] peer-to-peer networks. Using the Emule client, users can search files and for given search results they can look at comments that other peers have entered. These comments are all from peers that have downloaded the file already. See Figure 2.1 for a screenshot. The comments that are generated, are found

using the Kademia DHT with the hashes of files as a key. The comments are simply listed in a user interface, without combining them in any way and we perceive the quality to be low.

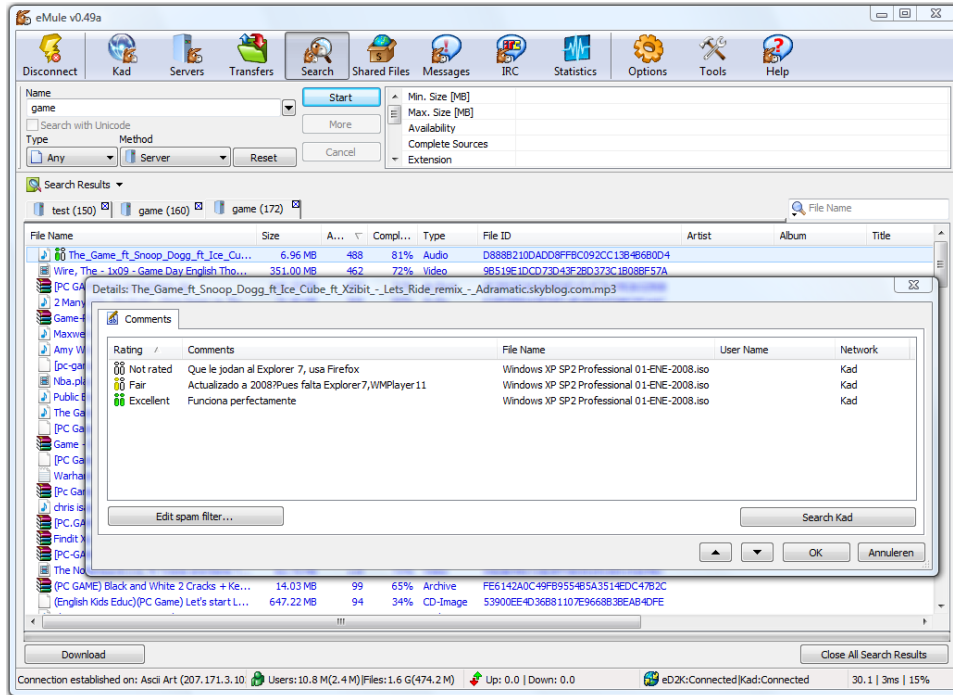


Figure 2.1: Emule client search window with comments window.

For existing systems described above we give a short overview of which requirements they do and do not meet. Note that Wikipedia is not actually used for Video metadata, but for encyclopedia articles.

System	Good quality metadata	Scalability	Bandwidth efficiency
YouTube	-	-	+
Wikipedia	+ (articles)	-	+
Emule	-	+	+

None of the existing systems meet all the requirements we specified in Section 2.2. The main issue seems to be how to combine good quality metadata and scalability. Wikipedia [39] uses a combination of a central website and allows people to contribute to the community such that the good quality requirement is met. However, the central nature of the system hinders scalability. On the other hand existing peer-to-peer systems, such as Emule, have the scalability but lack the good quality metadata.

Chapter 3

Design

The focus of this Chapter is on exploring the design alternatives for our infrastructure. First we present the metadata-elements that our architecture supports and then we outline the design in three steps. Every step adds more complexity to the design, but better fulfills the requirements. For every step we will explain which requirements are and are not met.

In Section 3.1 we present the metadata element set our architecture supports. In Section 3.2 we give design alternatives to distribute metadata and give the basic design using an epidemic protocol. Section 3.3 extends the basic design using forwarders, unleashing the power of peer-to-peer, whilst taking pollution into account. Finally Section 3.4 extends this design by including on-demand downloading, making the system more bandwidth efficient.

3.1 Metadata element set

We have chosen to use the following set of metadata for video. This set can be extended, however, using the extended metadata in old versions of Tribler is a problem because the user interface has to be updated to show the new metadata. From now on we will use the word moderation if we imply a subset of this metadata:

Description Text-based description of the video.

Tags List of words describing the video. (Useful for searching)

Thumbnail A picture to illustrate the video.

Language The spoken language of the video.

Subtitles Subtitles (in multiple languages) for the video.

We have chosen this set as we think it is a relatively straightforward set of elements that people can easily fill in. This is important as we count on the users to fill in the metadata. The elements should also be very unambiguous as ambiguity could easily lead to edit wars, where people keep overwriting each others contribution because they do not agree with the other user's opinion.

Within Tribler a new user-interface has been designed. This new user-interface design includes an interface for moderations. See Figure 3.1 for a screenshot of this design.

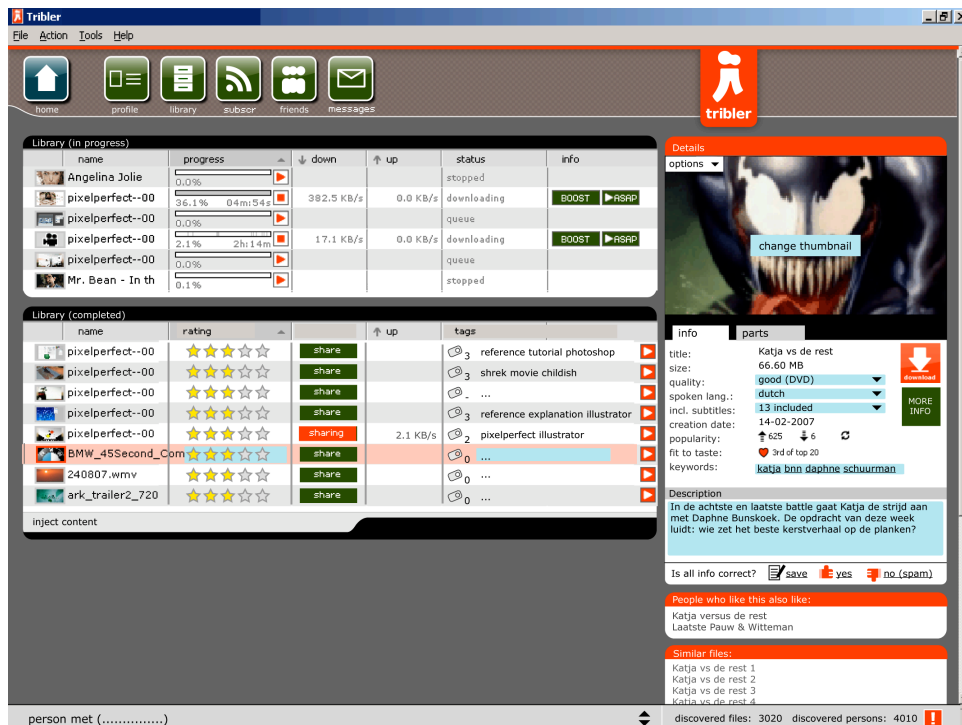


Figure 3.1: Moderation-interface design in Tribler.

3.2 Direct spreading approach

This system enables the user to add a moderation to a torrent. Such a user will be called a moderator from now on. These moderations are stored in a local database,

such that the user-interface can show the new metadata next to (or instead of) the metadata from the original torrent or file. This database is further used to distribute the moderations to other peers.

Various systems can be used to distribute metadata. These include: a central database, a DHT or an epidemic protocol. Using a central database in a peer-to-peer system hinders the scalability and reliability of the system. Therefore we will not consider it a feasible option. The DHT approach is useful, however it does have known weaknesses such as load balancing and security issues [8, 32]. Our system uses an epidemic protocol for information dissemination. Epidemic protocols offer, scalability, high fault tolerance, flexibility of deployment, adaptivity and a lack of central control [37].

We use the Buddycast epidemic protocol, as it is already included in Tribler and has proven to be robust enough to handle the real world scenario's such as high churn. Buddycast uses both random and similar peers to achieve clustering while keeping the network connected and reliable.

Every time a peer executing Buddycast connects to another peer, it will give the other peer some identifiers for which it has moderations. These identifiers consist of the infohash of the torrent they belong to and the timestamp of moderation creation. The other peer can then request some moderations, after which this peer may give them to the other peer. See Figure 3.2.

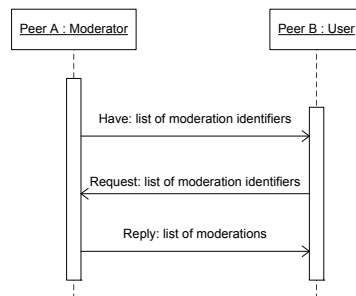


Figure 3.2: Sequence diagram of the direct spreading approach. (All the messages are also exchanged in the opposite direction.)

One might ask what will happen if two moderators moderate the same content. In this case a peer may receive two moderations for the same content. Therefore there is a need for a policy to choose or integrate moderations for the same torrents. The following policies can be used:

Use the first received moderation This naive policy is the easiest to implement. However this would imply that metadata for a torrent cannot change anymore

once a moderation has been received.

Keep latest moderation Keeping the latest moderation is a policy that is used by systems such as Wikipedia and is useful when people try to improve information. If we want our system to use this policy it requires a global notion of time [22]. If we further want it to potentially converge to a global consensus, it requires total ordering of moderations, per torrent.

Combine the two moderations This policy can be useful if the data from multiple moderations can be aggregated easily. This is the case for voting/tagging, etc. This policy can be used with the previous policy as a tiebreaker.

We have chosen not to enable combinations of moderations because for a lot of moderation-elements there is no suitable and generic way to combine the elements automatically. Taking this approach keeps things relatively simple. We have chosen to use the latest moderation, as it will decrease bandwidth usage (old moderations will not be propagated anymore by users that have both the new and the old moderation). To potentially reach global consensus we have chosen to create a total ordering of moderations per torrent using time with the permID of the moderator as a tie-breaker.

The direct spreading approach described above enables us to create and share moderations in a peer-to-peer system, without central authority. However it also has the following drawbacks, which we will try to address in the following Section:

No scalability Because the moderator is the only one that distributes the moderation that it has created, a moderation for a popular torrent is requested by almost every peer. This requires a lot of upload bandwidth from the moderator. It also requires a long time before the moderation is propagated to every peer. This is because the moderator must have direct contact with every other peer. For large peer-to-peer systems (which may have hundreds of thousands of users) this takes much too long and may not converge at all.

Moderator must be online for distribution Because the moderator is the only one distributing its moderations, the moderation dissemination stops when the moderator goes offline. Also if the moderator is connectable (behind firewall), the dissemination takes at least twice as long, because only passive Buddycast connections are possible.

Trust issue Because every peer can moderate as easily as any other peer, malicious peers can abuse this system to send unwanted moderations. Experience with email spam suggests that this is a real issue.

No system-wide incentive to share moderations Although the users may moderate in order to keep their libraries ordered, there is no incentive to share

these moderations with other peers. We can of course enforce this in the software, however there is no way that we can enforce this if someone creates a new client or alters our client. Although our system does not have such a system-wide incentive, we could integrate a system such as Tit-For-Tat, like in BitTorrent [10].

3.3 Forwarder-based approach

To make the direct spreading approach more scalable we need peers to be able to distribute other peers' moderations. This also solves the problem of the moderator going offline. However, how does a potential forwarder (a person that distributes moderations from other moderators) know which moderations are good enough to forward? Here there are two layers at which we can determine to forward or not: at the moderator level or at the moderation level.

We have chosen to okay moderators for forwarding instead of moderations. This makes it much easier for the forwarders, as they can okay bigger groups of moderations. It also reduces the time it takes for a forwarder to start forwarding a new moderation, as there is not user interaction required.

The trust issue is a big problem. Much effort has gone into the research and development of distributed trust systems [42, 20]. However no peer-to-peer system has been able to guarantee trust. We have chosen to enable peers to block moderations from certain moderators. This allows them to block moderators who send moderations that they do not like. This list is only kept locally. However the propagation speed of moderations from moderators that send bad moderations will be lower, as no honest user will forward for them.

Because moderators can now get blocked based on their moderations and forwarders are forwarding moderations for (other) moderators, there is need for authentication of moderations instead of only authenticating the connection on which the moderations are sent. This can be done by using public key signatures. This prevents alterations 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.

This results in the following steps being taken:

1. A moderator M creates a moderation m .
2. Other peers meet M (because of Buddycast). And receive m .

- Peers that have indicated that they want to forward for M and have received m will forward m to peers that they meet. But this will only be done if the moderation is signed with a valid digital signature matching M 's ID.

An example of this is given in Figure 3.3.

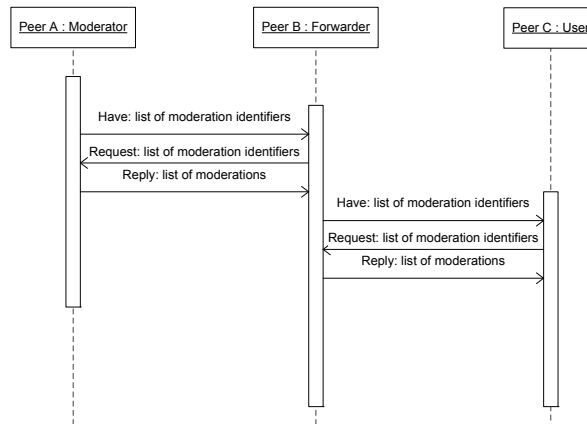


Figure 3.3: Sequence diagram of the forwarder-based approach.

This improved system is much more scalable because peers can assist other peers to distribute moderations. This alleviates the upload-bandwidth burden on the moderator. It also increases the speed at which the moderations propagate through the system. Because the good moderators can rely on forwarders to speedup their moderation dissemination and decrease their upload-bandwidth requirement, this system gives them an edge over moderators that distribute bad moderations. However this system still has a drawback. Although the moderations can now be distributed by more than one peer, there is a lot of wasted bandwidth. This is because in practice the largest items in the moderations will not be used by many peers. E.g. subtitles can take many kilobytes per language for every video. Some people may not even watch the video and some may choose to watch the video without subtitles, not needing the subtitles at all. Even if the user watches the video with a subtitle, he/she does not need the languages he/she does not use. Therefore it would be much more bandwidth efficient to download the subtitles on demand.

3.4 On-demand downloading approach

To make the previous approach more bandwidth efficient we need to create a scalable system for downloading bandwidth-intensive parts of metadata on demand

(when they are needed). There are two key requirements which we need to address:

Latency Some of the metadata elements may have a low latency requirement. For these elements there is probably a trade-off to be made between latency and bandwidth. Downloading metadata elements on demand will increase latency and will reduce bandwidth usage, because the elements are only downloaded when needed.

Scalability The system as a whole needs to be scalable. Therefore a centralized point for downloading large metadata, such as subtitles, would be unacceptable. It would be a bottleneck, single point of failure and single point of authority, none of which we would like to have in our system.

We have chosen to use the BitTorrent protocol itself to download the subtitles on demand. Although this protocol is often used for bigger files, we think it is an elegant solution to map the on-demand downloading of larger metadata elements back to the original protocol. To get rid of the central point used in BitTorrent (the tracker), and to further reduce the bandwidth consumption used by the subtitles in the gossiping protocol, we have chosen to use a trackerless torrent. This system uses a DHT to determine which peers have the subtitles. This has a significant increase in delay in comparison to a central tracker. A recent study has shown that DHT-lookup times can be in the order of several minutes [13]. However, this is still acceptable, in part because the download time for the small subtitles is very short. Using this system is far more robust and has better load balancing than using either a centralized point or letting the moderator or forwarder gossip the subtitles. Figure 3.4 shows the behavior of the three sorts of peers in this system.

This approach is more efficient than the previous, because it does not spread the larger metadata-parts to normal peers right away. This decreases the total amount of bandwidth used in the system. The on-demand metadata-parts are still requested by the forwarders, such that there is no loss in replication. This gives an equal amount of availability, compared to the previous system. The scalability and availability increase further because the downloading peers automatically seed the elements they download. The integrity and authenticity of the metadata-parts that are sent using this method are still guaranteed because of the hashes that are included in the part of the moderation that is gossiped. Drawbacks of this system include; an increase in latency, increased complexity and a need for an incentive for the normal peers to seed the metadata-parts they download. The needed incentive to share moderations is reduced further by this system because there is a decrease in upload-bandwidth.

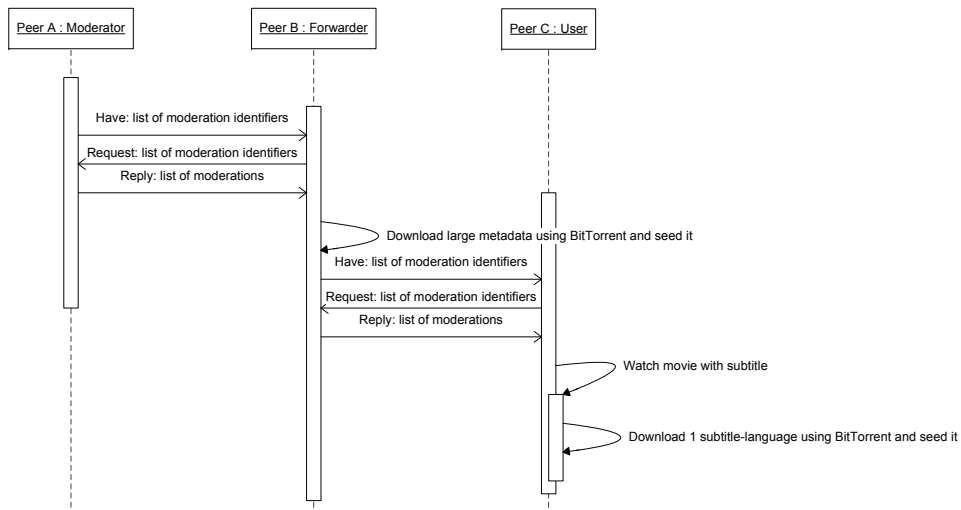


Figure 3.4: Sequence diagram of the on-demand downloading system.

In the next Chapter we will present MODERATIONCAST, the moderation system that we have implemented. This follows the design of the forwarder based approach with on-demand downloading.

Chapter 4

Implementation

After designing MODERATIONCAST in Chapter 3, we have implemented MODERATIONCAST as a proof of concept. We have done this using Tribler, to which we have added MODERATIONCAST. This Chapter outlines some of the elements that are implemented during this thesis. Section 4.1 gives an overview of the technical design, focusing on the class-relationships between MODERATIONCAST-classes and the rest of Tribler. Section 4.2 then goes into detail on the major implementation-issues, such as authentication and global consistency.

4.1 Technical design

We start off with an overview of our technical design, see Figure 4.1. This technical design shows the classes that comprise the MODERATIONCAST subsystem and how it integrates with the rest of Tribler. In the Tribler User Interface (TriblerUI) an extra window is created to show the moderations and moderators next to the torrents (see Figure 4.2). This window enables the user to create moderations for torrents. All moderations and moderators are stored in two databases, the ModerationDB and ModeratorDB. The storage and retrieval occurs trough a database-handler class. Because the TriblerUI now needs to show the moderation data, which can change at any moment, the Tribler User Interface is updated by the SynModerationDBHandler, which gives callbacks whenever there are updates on the moderators or moderations. This is a specialisation of the ModerationcastDB-Handler, which offers the same functionality, but does not callback on updates.

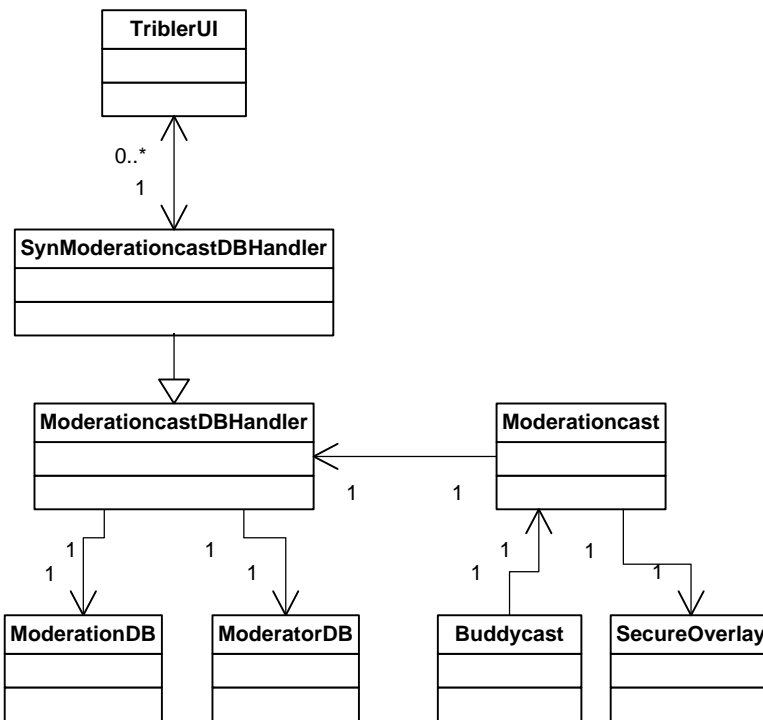


Figure 4.1: Technical design overview.

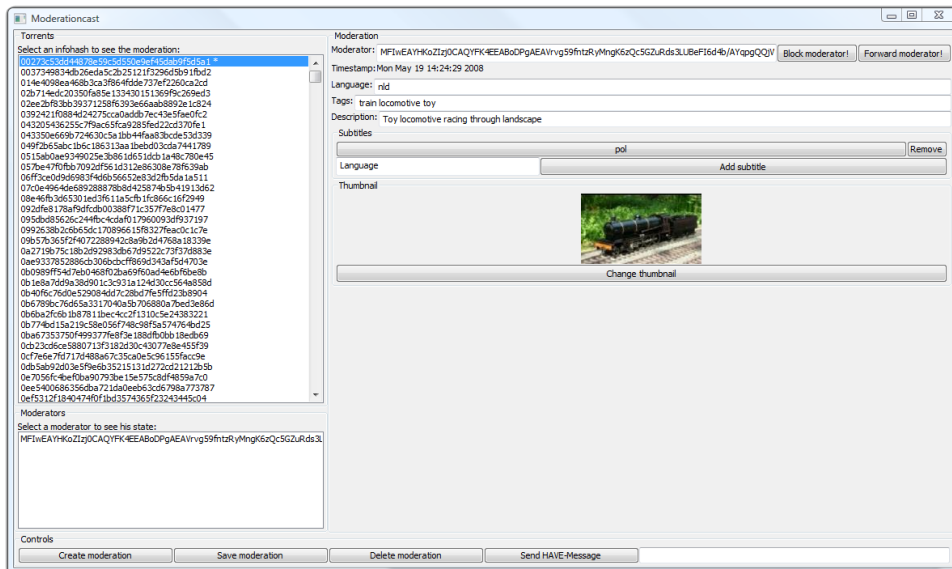


Figure 4.2: Screenshot of Moderationcast-interface.

The MODERATIONCAST system depends, in large part, on moderations from other peers. These moderations are entered into the databases by the Moderationcast class, using the ModerationcastDBHandler. The Moderationcast class receives a callback after each Buddycast-message sent by the Buddycast subsystem. This triggers the Moderationcast class to send a have-message (see Table 4.1) to the same peer that received a Buddycast-message using the SecureOverlay class. This results in the two peers sending each other have-messages after a Buddycast exchange. These have-messages are replied with request-messages, which indicate which moderations are requested. Then the reply-messages are sent, which contain the actual moderations.

Have			
Field	Description	Type	Size
infohash	torrent infohash	string	23 bytes
timestamp	seconds since epoch	int	12 bytes
size	moderation size in bytes	int	12 bytes
100 moderations* with a total of 4702 bytes			
*Determined by have message inclusion policy			
Request			
Field	Description	Type	Size
infohash	torrent infohash	string	23 bytes
max. 100 requested moderations with a total of max. 2302 bytes			
Reply			
Field	Description	Type	Size
infohash	torrent infohash	binary string	23 bytes
timestamp	seconds since epoch	int	12 bytes
description	text describing video	Unicode string	2 KB
subtitles	subtitles for several languages	{language encoding:torrent}	4 KB
thumbnail	picture	binary JPEG string	20 KB
tags	tags for video	list of Unicode strings	2 KB
spoken language	language encoding	ISO 639-3 string	5 bytes
moderator	PermID of moderator	binary string	124 bytes
signature	cryptographic signature	binary string	67 bytes
max. 100 moderations, 28 KB each			

Table 4.1: Message layout for have-, request- and reply-message. Bold fields are required, other are optional. Sizes are maximum sizes, using bencoding. The language encoding is a subset of ISO 639-3 [19].

The reason that there are two databases, the ModerationDB and the ModeratorDB is that the database technology used in Tribler, Oracle Berkeley DB, only allows for one index per database. Because we need two indexes, the infohash of the torrent and the PermID of the moderator, we need two databases, which we need to keep consistent. This is the responsibility of the ModerationcastDBHandler.

4.2 Implementation issues

The implementation of MODERATIONCAST had the following implementation issues (see Table 4.2). This Table also shows the number of lines of code that the actual implementation costs. As can be seen in the table the database related issues take up a major part of the implementation. This is because there are two databases which need to be kept consistent and the database(-handler) provides an interface to Tribler. The message handling implementation is quite complex as there are three sorts of messages which need to be created, sent and received. Because MODERATIONCAST is a peer-to-peer system in which we cannot trust other peers, thorough message-structure validation is needed, which requires a large number of lines of code. Finally the test interface takes up 750 lines of code, making it the largest individual part of the implementation. This is not so strange as a lot of user-events have to be implemented. We will now go into more detail of the individual implementation issues.

Implementation issue	LOC
Authentication	20
Total moderation ordering	20
Database	500
Dynamic updates	50
Bandwidth limiting	120
Message compression	30
Message handling	450
Message-structure validation	400
Test interface	750
Total	2340

Table 4.2: Implementation issues and the number of Lines Of Code used for these issues.

Authentication

Tribler uses PermIDs to establish non spoofable permanent identifiers for peers. These are used to authenticate a peer whenever one connects to it. This is done using a public key challenge-response system. This is what happens when peer *A* and peer *B* connect securely [26] (It should be kept in mind that PermID refers to the public key of the peer.):

1. Peer *A* generates a random number r_A and sends it to peer *B*.

2. Peer B receives r_A , generates a random number r_B and creates a signature S_B for (r_A, r_B) using its private key K_B . It then sends $(PermID_B, r_B, S_B)$ to peer A .
3. Peer A receives $(PermID_B, r_B, S_B)$, verifies S_B is the correct signature for (r_A, r_B) using $PermID_B$ and if and only if this is correct, A has successfully authenticated B and will create a signature S_A for (r_B, r_A) using its private key K_A . It then sends $(PermID_A, r_A, S_A)$ to peer B .
4. Peer B receives $(PermID_A, r_A, S_A)$, verifies S_A is the correct signature for (r_B, r_A) using $PermID_A$ and if and only if this is correct, B has successfully authenticated A .

However, because we are creating MODERATIONCAST, in which peers can forward other peers' moderations, we need to ensure authentication on a different level, namely of the moderations themselves. Only if we are able to determine if a moderation is truly created by a certain source, we can hold this source responsible and block, or start to forward for, him. Authenticating the source of a moderation is possible with public key encryption and is implemented in MODERATIONCAST in the following way:

1. When a peer A creates a moderation m , it creates a signature S_m for m with its private key and adds this to m , resulting in (m, S_m) .
2. When a peer requests m from A , it will receive and store (m, S_m) .
3. A forwarder B , forwarding for A , that has received and stored (m, S_m) replies to any requests for m with (m, S_m) .
4. A peer receiving (m, S_m) from B checks m with S_m and the PermID of the moderator, which is included in any moderation.

This approach ensures that any tampering with a moderation will be detected. Peers using MODERATIONCAST can therefore block moderators that provide bad moderations and forward for peers that provide good moderations.

Towards global consistency

In MODERATIONCAST the latest version of a moderation, for a given torrent, is used. However, in a distributed system it is sometimes impossible to determine the actual chronology of events [22]. Logical clocks, which assign numbers to

events, can be used to order events in a distributed system. However to get a causal ordering over the different versions in a distributed system, enabling us to identify the latest version, requires vector or matrix clocks [29]. These present considerable bandwidth overhead in a system with a large number of peers as they include a scalar logical clock per peer [9]. This extra bandwidth is prohibitive for our system. For this reason and because of the small chance of people actually moderating the same torrent during a small time-period, we have chosen a non-causal message ordering, based on the local system time of the peers. It works as follows;

1. Whenever a moderation is created, the moderator uses its local (GMT) time.
2. On receipt of a moderation, the included timestamp is checked to be less than 5 minutes into the future, else it will be discarded.

As long as the peers have a small clock-skew, which is relatively easy to guarantee using for instance the Network Time Protocol [25], moderations will not be discarded. Also, given that there is a small chance that people moderate the same torrent in the same five minute interval, the latest moderation will be shown.

As we are aiming at global consistency, we need every peer to choose the same moderation as latest. This is not guaranteed using the above mentioned ordering, as two peers can give equal timestamps to different moderation-versions. To this end we need a total ordering [22] of moderations. We have implemented this total ordering using the above method with the unique PermID of the moderator as a tie-breaker. Although this is a rather arbitrary tie-breaker, it asserts that any two peers will always choose the same moderation from a given set of moderations.

Dynamic updates

As MODERATIONCAST runs, moderations will be added to the ModerationDB continuously. Also there will be more and more moderators to keep track of in the ModeratorDB. Because this information may already be shown in the user interface, it would continuously have to poll the database. This is not a feasible solution. Therefore we have implemented a database-handler which informs the user interface of changes made to the database using the observer pattern [15]. It then only has to get the changed information from the database. See Table 4.3 for the methods and their callbacks.

This callback structure also works with cascading actions, for instance when a moderator is blocked, all the moderations for that moderator are removed from the

Method	Callback Event	Parameters
addModeration	add	infohash of torrent
deleteModeration	delete	infohash of torrent
addModerator	addModerator	PermID of moderator
blockModerator	blocked	PermID of moderator
forwardModerator	forward	PermID of moderator
deleteModerator	deleteModerator	PermID of moderator

Table 4.3: Methods and events for the Moderationcast database-handler which calls back on updates.

system. The database-handler automatically informs the user interface of all these removals using the delete-callback event.

Bandwidth limiting

To avoid hampering the performance of other applications, we have implemented bandwidth limiting in MODERATIONCAST. This allows the user to set a certain bandwidth (upload- and download bandwidth) that may be used per minute. We have done this using the three message-exchanges that take place in every Moderationcast interaction between peer *A* and peer *B*:

1. On receipt of a have-message from peer *A*, peer *B* only requests a subset of the moderations in this message. It selects moderations for which it has the torrent(s) and which have a later timestamp than any moderation it may have for those torrents. To further reduce download bandwidth, it will request an even smaller subset. The bandwidth usage is calculated using the moderation size information in the have-message. The resulting request is sent to *A* in a request-message.
2. On receipt of a request-message from peer *B*, peer *A* can reduce its upload-bandwidth requirement by sending a subset of the requested moderations.

To determine how much bandwidth should be reserved for every exchange, two instances of the SustainableBandwidthLimiter class are used. One for the upload-bandwidth and one for the download-bandwidth. This class keeps track of the timestamps and sizes of requests made within the last minute (default). Using this information, which it automatically updates, and a set bandwidth-limit, it gives the caller the size that can be used for a message-exchange. After actually using some size, the user calls another method of the SustainableBandwidthLimiter and it

updates the bandwidth-usage information. The size that can be used for a message-exchange, is calculated using the following formula:

$$size = \min(BW - \sum_{i \in R} i, \frac{BW}{|R| + 1}) \quad (4.1)$$

Here *size* is the size that can be used for a message-exchange, *BW* is the bandwidth limit per minute and *R* is the set of sizes of requests that have occurred during the last minute. This formula ensures that the bandwidth limit is abided while trying to give every request an equal share of the bandwidth.

To exchange messages, Tribler uses bencoding, which is less efficient than pure binary encoding [4]. To further reduce the bandwidth requirement, we have implemented message compression in MODERATIONCAST using zlib [43]. This reduces the overhead created by bencoding.

Chapter 5

Trace based simulation

After implementing the design of MODERATIONCAST, testing it to check if it meets the requirements set in Section 2.2 is non-trivial as the infrastructure is designed to be used in a distributed system with many thousands, or more, peers. Running the actual implementation for every peer would require too many resources and testing with so many users be impractical. Therefore we have chosen to test the system using a simulator and to use a trace gathered from a real BitTorrent community to supply user actions and states.

This Chapter gives a detailed description of the trace-file and the simulator used for the simulations we do in Chapter 6. First we give an overview of our simulation setup, followed by an explanation of the traces and the individual simulators used.

5.1 Simulation Setup

As said earlier, we use a trace gathered from a real BitTorrent community to run simulations. Because MODERATIONCAST uses Buddycast to establish connections between users, we need to simulate Buddycast and MODERATIONCAST. Buddycast already has a simulator we can use, which we can run on the input traces. Therefore there are two reasonable things we can do, either add the MODERATIONCAST simulator to the existing Buddycast simulator, or run the MODERATIONCAST simulator on output from the Buddycast simulator. We have chosen the later approach because the Buddycast simulator requires a lot of runtime and the process is the same for every run of our simulator. Every peer connects to the same peer at the same moment in every run of the Buddycast simulator, given the same trace.

For this approach to succeed we have changed the Buddycast simulator to output the connections made between peers together with the events in the original trace. Given this output trace, now called Buddycast trace, we run a newly created MODERATIONCAST simulator. See Figure 5.1. This simulator requires a simulation setup file indicating which peers moderate which torrents and which peers forward for whom. This simulator gives us a trace with performance- and bandwidth-usage-information.

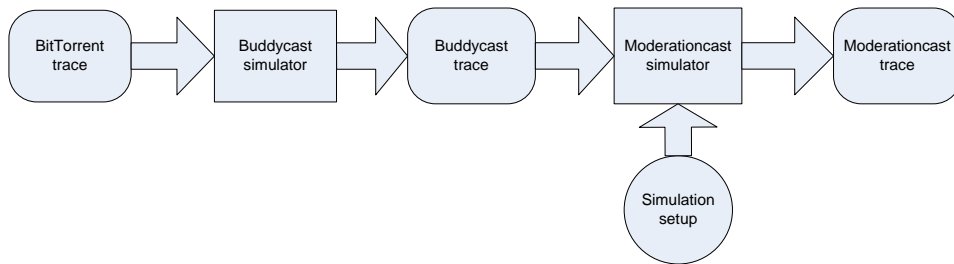


Figure 5.1: Global overview of the simulation setup.

5.2 BitTorrent Trace

For our simulations we use a trace gathered from a real BitTorrent community [14]. This trace was gathered earlier for [30]. This trace gives us the online- and offline times of peers, their connectability (connectable or behind NAT or firewall) and their download history during the trace period.

Because the Buddycast simulator uses a lot of memory, we can only simulate up to 2000 peers. Therefore we have selected 2000 peers from the original trace. We have chosen to use the top active peers because this corresponds with the users we have in Tribler. Figure 5.2 shows the number of online and connectable peers during the complete trace period. Figure 5.3 shows the distribution of torrents at the end of the complete trace.

Figure 5.2 clearly shows that during parts of the trace the number of peers drops significantly (to almost zero). This is due to some network failures and power outages that happened while gathering the trace. Therefore we have isolated a period of one week, during which no such thing happened. This week occurs after almost 9 weeks of tracing. We have used the online and offline times of the peers during this week, which results in Figure 5.4. However we have taken into account the download history of the peers during the past 9 weeks. This results in Figure 5.5.

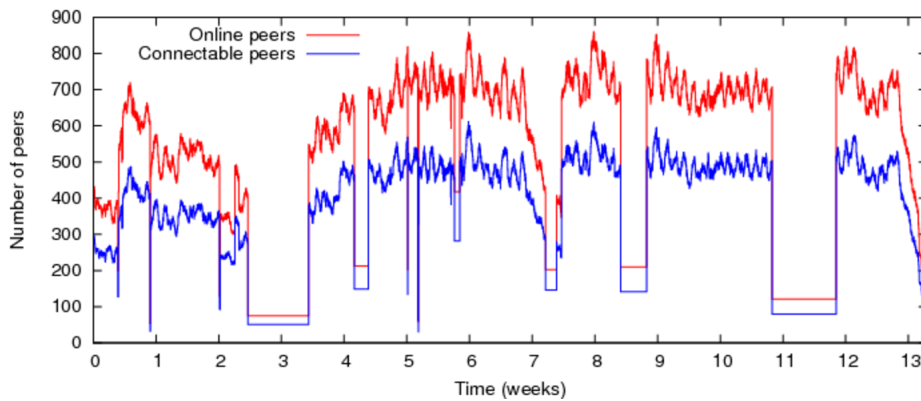


Figure 5.2: Number of online and connectable peers for the complete trace after filtering the top 2000 active peers.

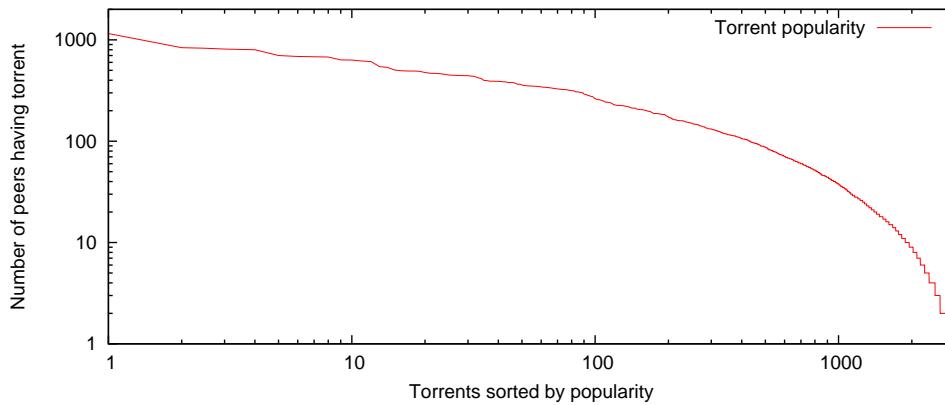


Figure 5.3: Torrent distribution at the end of the complete trace after filtering the top 2000 active peers.

5.3 Buddycast simulator

Using the BitTorrent community trace, we have run a trace based simulation using a Buddycast simulator. We have changed this simulator to log the times at which the information exchanges between peers happen. From the trace, the following data is used to set the behavior of the peers in the Buddycast simulation:

- **Connectability** Sets whether or not the peers can receive any incoming connections.
- **Online- and offline times** Sets the online- and offline times for the peers.

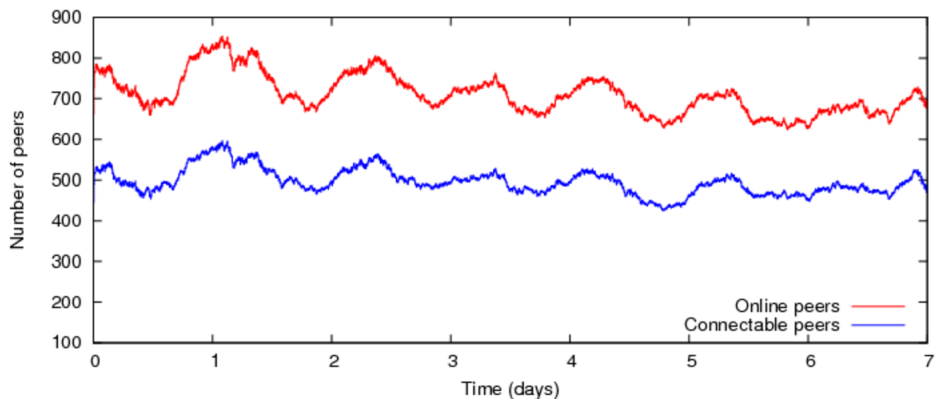


Figure 5.4: Number of online and connectable peers for the longest continuously monitored part of the trace after filtering the top 2000 active peers.

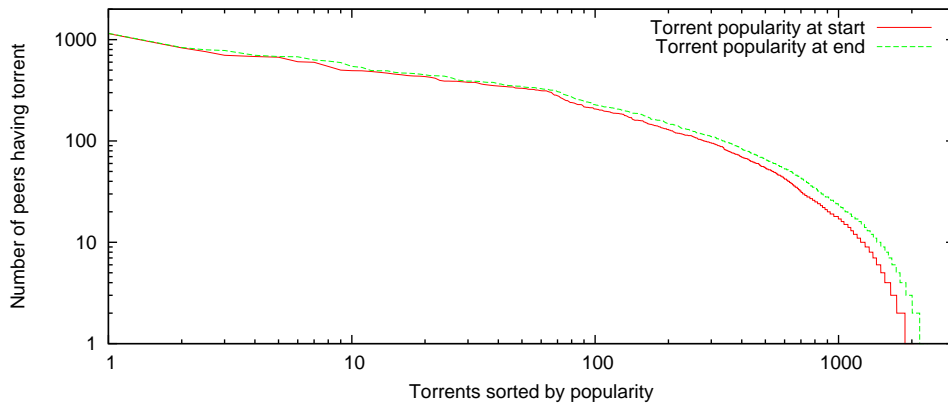


Figure 5.5: Torrent distribution at the start and end of the 7 day partial trace.

- **Torrent download actions** Sets when the peer adds the torrent to its download history (its preferences for Buddycast). For the download-history we do take the past 9 weeks into account.

The Buddycast simulator outputs the following data to be used by the MODERATIONCASTsimulator:

- **Online- and offline times** Same as in the Buddycast simulation log.
- **Timestamp of sending of Moderationcast Have message, its receiver and its size** Can be used to determine bandwidth usage.
- **Timestamp of sending of Moderationcast Request message, its receiver and its size** Can be used to determine bandwidth usage.

- **Timestamp sending of Moderationcast Reply message, the data send, its receiver and its size** Can be used to determine the propagation speed of moderations and the bandwidth usage.

5.4 Moderationcast simulator

Using the output from the Buddycast simulator, the Moderationcast simulator then runs a simulation for parameters:

- **Moderators** The peers that will be moderating torrents in the simulation. For each moderator a moderation policy can be specified. The moderators policy can be: Moderate a set of torrents or moderate N random torrents at a given rate per hour or moderate a set of moderations at given timestamps.
- **Forwarders** The peers that will be forwarding in the simulation. There are two options, either forward for all moderators or forward only for a given subset.
- **Have message policy** Policy to select moderations to show in have message. A maximum number can be specified for random-own moderations, recent-own moderations, random-forward moderations and recent-forward moderations.
- **Size of moderations** The size of a single moderation in bytes. This is used to determine the bandwidth usage of MODERATIONCAST. By default it results in 25 kilobytes (KB), a reasonable worst-case scenario in which every moderation has a 20KB thumbnail with 5KB of description, tags, spoken language and the hashes for subtitles.

The MODERATIONCAST simulator outputs the following data to be used for analysis of our protocol:

- **Sending of messages** The time, sender, receiver, sort and size of the following message-types: Buddycast, Have, Request and Reply. For the reply message also outputs the torrent-id, size, moderator and timestamp of moderation. This can be used to determine the propagation speed and bandwidth usage of our protocol.
- **Connectability of peers** The connectability of a peer during an online session.

- **Online- and offline times** The time a peer goes online and offline.
- **Torrent download actions** Sets when the peer adds the torrent to its download history (its preferences for Buddycast). Can be used to determine the number of peers that have a torrent at a certain moment. This is the propagation speed limit for a moderation concerning that torrent.

Using the simulation setup described above, we will experiment with parameters in the next Chapter.

Chapter 6

Results

This Chapter focuses on the scalability of our protocol and the way it copes with pollution. The evaluation is done using the simulator outlined in the previous Chapter. First we determine a good have-message inclusion policy in Section 6.1. This is an important parameter as it determines which moderations are shown to other peers. A wrong policy will show a lot of moderations that other peers already have and will thus hinder the speed of propagation. In Section 6.2 we determine how our protocol deals with different numbers of forwarders. More forwarders should give a higher propagation speed and should relieve the upload bandwidth requirement for the moderator. This makes the system more scalable. In Section 6.3 we determine the scalability of our protocol in terms of the number of peers. This is important as peer-to-peer systems may have a very high number of peers. Finally in Section 6.4 we use a scenario to show that pollution can be removed efficiently using our protocol, given some basic assumptions. This is important as pollution is a real issue on the Internet.

6.1 Policy experiments

The have-message inclusion policy, introduced in Section 3.2, determines which subset of a peers' moderations are shown to other peers it connects to. The reason why a subset is shown, is to reduce the bandwidth-usage compared to showing all moderations. The subset should be big enough to provide the other peer with moderations it does not have, if this is possible. To determine the performance of various policies we ran various simulations. During these simulations we keep track of the number of moderations being requested and the number of moderations shown in the have-message. The higher the number of requested moderations is,

compared to the number of shown moderations, the more effective the policy is. The lower the number of moderations shown in a have-message, the lower the size of this message and thus the less bandwidth-intensive the protocol becomes.

We have chosen to base our experiment on a system with 2000 users and to let one moderator create twelve moderations per hour. We have chosen twelve moderations per hour because we think this amount is reasonable for a set of moderators a forwarder forwards for. Because it does not matter if moderations are created by a moderator or only forwarded by it, we can deduce from this experiment how many moderations we need to show to other peers given that the moderators that we forward for, create 12 moderations per hour. We have chosen to test it this way, because it eliminates the possibility of moderating the same torrent twice, which would influence the results. We will now experiment with the following policies:

- *Random*, pick N uniform random moderations
- *Recent*, pick the N most recently created moderations
- *Random-Recent*, pick $N/2$ moderations using the random policy and $N/2$ moderations using the recent policy (without overlap).

For these three policies we vary the number of moderation-IDs in the moderators own have-message-part between 10 and 100. The results of this experiment are shown in Figure 6.1.

Figure 6.1 shows that the number of requested moderations increases when we increase the number of shown moderation-IDs. This makes sense, because a bigger sample results in a bigger chance of finding an unknown moderation. However this increase is sub-linear, indicating that the added benefit of increasing the number of moderation-IDs decreases. The random and random-recent policies have very similar behavior, where the performance of the random policy is slightly better at the higher number (from 60 onwards) of moderation-IDs. The recent policy performs less from 20 moderation-IDs onwards. A reason for this might be that peers that have gone offline are unable to obtain some moderations that were created when they were offline, because only the most recent moderation-IDs are shown. With the two other policies, this is not the case.

Figure 6.1 suggests that we should use either the random or the random-recent policy in order to have a high *request/show*-ratio, which improves the bandwidth-effectiveness of our protocol. From this Figure, we however cannot conclude which number of moderation-IDs is best to use. Increasing the number of moderation-IDs will increase the bandwidth usage, but will also increase the effectiveness of our protocol. In Figure 6.2 we have plotted the overhead imposed by the have- and

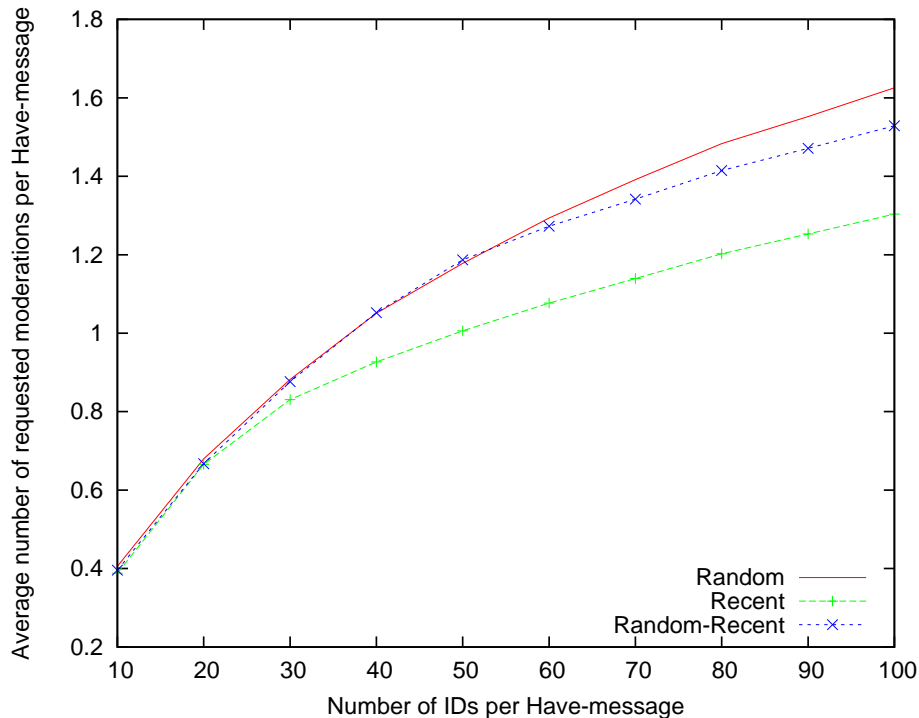


Figure 6.1: The average number of requests for three have-message inclusion policies and sizes.

request-messages per exchanged moderation for the above simulation. This overhead is calculated by summing up the average bandwidth usage for the have- and request-messages and dividing it by the average number of requested moderations. We do not add the reply-message bandwidth usage to the summation as it is not considered overhead. The Figure shows that the overhead per moderation will also increase for every increase in the number of moderation-IDs in a have-message. Which overhead is still acceptable depends highly on the size of the moderations. A couple of kilobytes of overhead will sound small compared to 25 KB moderations, however if the moderations only contain text, they might be smaller than the overhead presented by our system.

Given this bandwidth-propagation speed trade-off, we have chosen to use up to 100 moderation-IDs in every have-message. Of these 100, up to 50 are for our own moderations and the other 50 are for moderations from people we forward for. This number is high enough to ensure that a lot of moderations can be spread simultaneously. Although it may result in 100 moderations of 25 KB each being requested given one sent have-message, this potentially high excessive bandwidth usage can easily be stopped by only replying with a subset of the requested moderations. We

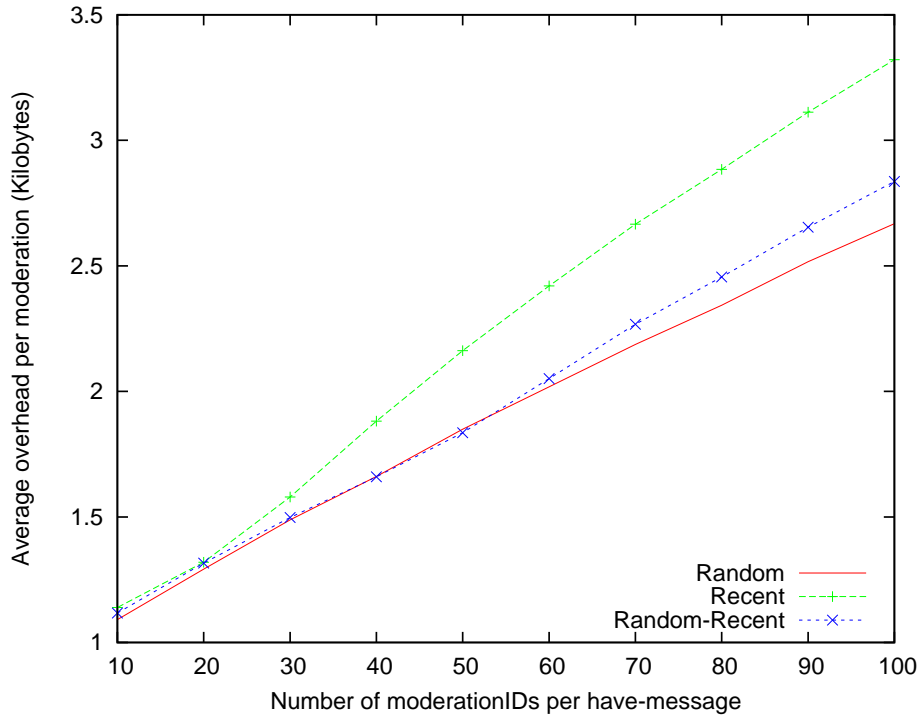


Figure 6.2: Average overhead per moderation.

have shown this in Section 4.2. Further, we have chosen the random-recent policy, as it has shown to work almost as well as the random policy and has a bias toward new moderations. This bias is useful because it will continue to work as more and more moderations are in a moderators or forwarders-database. Only selecting random moderations would result in performance degradation as very old moderations have as high a chance of being selected as new moderations. Only selecting recent is not an option as it would imply that offline users might actually not receive some moderations and will never receive them anymore. We will continue to use this policy in the coming experiments. For experiments with only one moderation, the policy does not matter.

6.2 Parameterize the number of forwarders

To determine the influence of the number of forwarders on the propagation speed of moderations and the upload-bandwidth usage for the moderator, we have run multiple simulations, increasing the number of forwarders. To make analysis and data visualization easier we focus on only one moderator and one moderation. To ensure a fair setup (no significant under- or overestimation of the propagation) we

have set the moderator and forwarders to be average peers. This is done by determining the median peers in the number of outgoing gossip (Buddycast) messages. We have run this simulation with the following parameters while keeping the total number of peers constant at 2000:

#Moderators	#Moderations	#Forwarders
1	1	0
1	1	2
1	1	4
1	1	8
1	1	16
1	1	32
1	1	64
1	1	128
1	1	256

The propagation of the moderation is depicted in Figures 6.3 and 6.4. In both Figures, gray areas show the offline times of the moderator. The solid red line in both Figures indicates the number of online peers, that have the torrent belonging to the moderation. The other lines show the number of online peers that already have the moderation. The reason that we only take into account the online peers is that these are the only peers that we can reach. Also taking into account the offline peers would give a very pessimistic view of the performance of our protocol. The closer the simulation-setup lines, labeled zero to sixteen forwarders, are to the red line, the better the performance of our protocol. The used upload-bandwidth by the moderator is shown in Figure 6.5. This Figure shows the cumulative upload-bandwidth usage for the moderator, given certain numbers of forwarders. In all three Figures, the grey area's show the offline-times of the moderator.

Figure 6.3 shows that after a short startup period in which the number of online peers having the moderation increases very rapidly, this number follows the number of online peers having the torrent. This is the case for all the simulation setups. However, when the moderator goes offline, the 0, 4 and 8 forwarder setups show a relatively high margin between these two numbers. From the 70th hour onward the 4 and 8 forwarders setups, which perform identically well, perform very similar to the 16 forwarder setup. Because the 16 forwarder setup already performs near optimal, we have chosen not to show the 32 and higher forwarder results in this Figure as they would hinder visibility without adding anything. From this Figure we can conclude that MODERATIONCAST is very effective and that with only 16 forwarders a moderation can be propagated fully, even with moderators that do not stay online.

Figure 6.4 gives a detailed view of the first 4 hours of the simulations. In the first couple of minutes the number of online peers having the torrent are increasing very

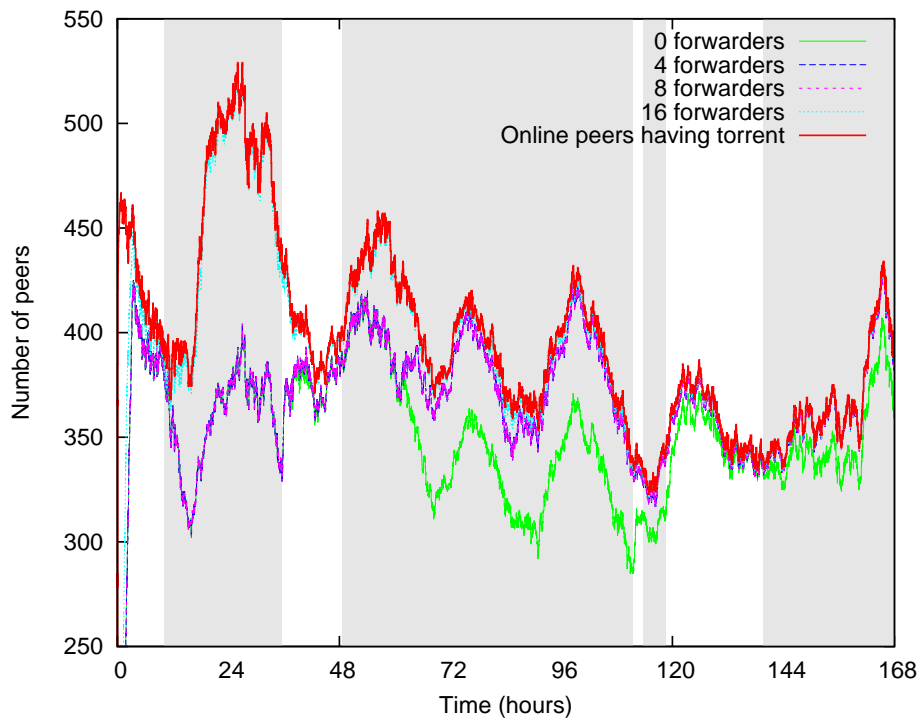


Figure 6.3: Propagation speed for a popular torrent and a moderation for that torrent. The gray areas show the offline times of the moderator.

rapidly from a very low number. This is because in the traces, the peers are coming back online. The torrent is hardly being downloaded during the simulations. In this Figure it is visible that increasing the number of forwarders results in reaching near-optimal propagation faster. The 128, 64 and 32 forwarder lines have a similar shape. The 32 forwarder setup gives near-optimal propagation within 100 minutes after moderation. For the 64 forwarder setup this is slightly faster and the 128 forwarder setup is the fastest, with a time of 40 minutes. The 0, 4 and 8 forwarder lines completely overlap in this Figure. This indicates that the forwarders are not helping at all. Possible reasons which explain that the 0, 4 and 8 forwarder setups perform equally well are:

Forwarder does not have torrent A peer may have indicated that it is willing to forward moderations from a moderator, however, it will not download moderations, for which it does not have the torrent. The reason why we did this is to ensure that bad moderations will be stopped. If a forwarder is forwarding moderations it does not see, it will not be in a position to see this bad behavior. For this particular torrent around 50% of the peers have the torrent.

Forwarder is not online A forwarder can be offline just like any other peer. When

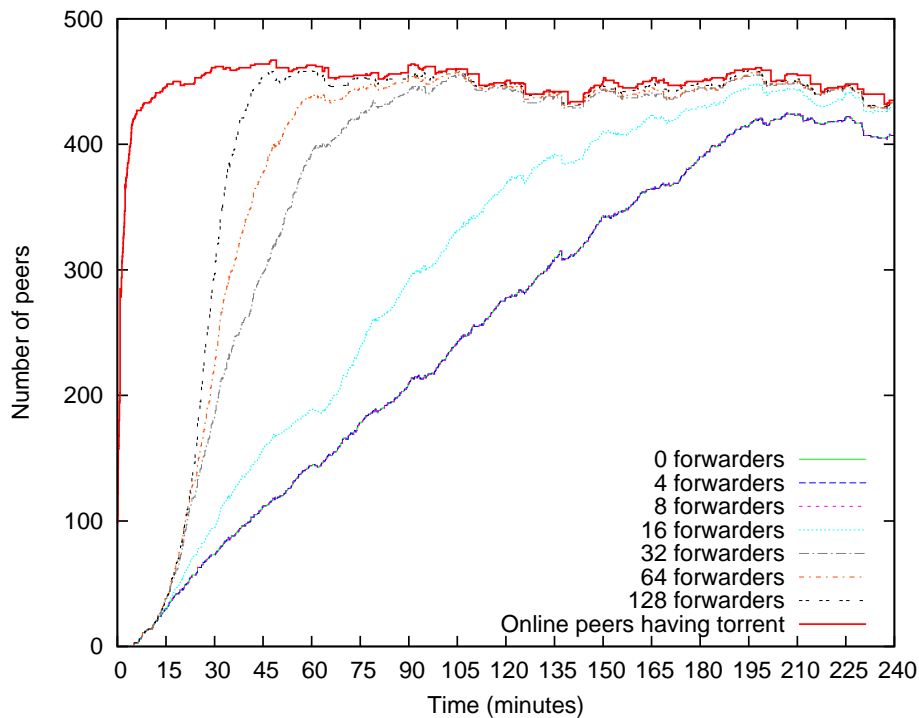


Figure 6.4: Detailed view of the first 4 hours of propagation for a popular torrent and a moderation for that torrent.

a forwarder is offline, it does not forwarding any moderations. Forwarders should be online as much as possible to support the moderator. In this trace between 32% and 42% of the peers are offline at any given time.

Forwarder is unconnectable A forwarder can be behind a firewall, when it is, the firewall can stop any incoming connection-attempts. This seriously hinders the underlying gossiping protocol and will result in less have-messages being sent and thus less propagation of the moderations. Actions can be taken to open ports for incoming connections, such that performance will not be affected by the firewall. In this trace around 30% of the peers are unconnectable.

Forwarder does not have the moderation A forwarder has no special priority in receiving the moderations. Therefore it might not receive a moderation for quite some time after the creation of the moderation. During this time it is unable to forward this moderation. The reason why we designed our system without such a priority is because there is no way to determine if a peer is a forwarder. If we were to give a higher priority to forwarders, it would be incentive for peers to lie about being a forwarder. Especially if the moderator goes offline before giving the moderation to a forwarder, this can slow down

propagation very drastically.

Figure 6.5 shows the upload bandwidth usage for the moderator. We have included have-, request-, and reply-messages in the calculation of the upload bandwidth usage. The Figure shows that in the initial hours after moderating the torrent, the used upload-bandwidth is increasing very rapidly. After this initial period the rate of increase in used upload-bandwidth decreases somewhat and then becomes 0. After this standstill (which lasts for about 1 day), the used-upload bandwidth increases again, but at a much lower rate than initially, except for the 0-8 forwarder simulations, which are still quite high. The rate of the upload-bandwidth increase after the standstill decreases for all the simulations. These standstills are coinciding with the grey areas, which indicate that the moderator is offline during this time. Increasing the number of forwarder results in a shorter period of intensive upload-bandwidth usage and a smaller total bandwidth usage. For the higher number of forwarders (8, 16, 32, 64 and 128), the upload-bandwidth decreases by about 50% for every doubling of moderators.

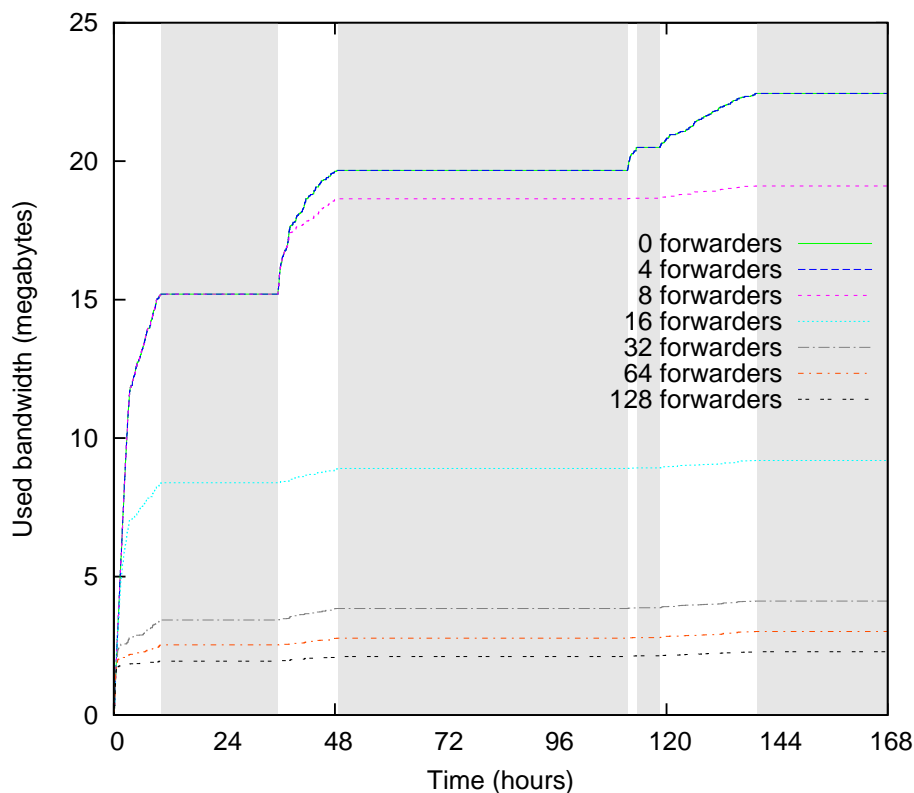


Figure 6.5: Upload bandwidth used by the moderator (for a moderation that 50% of the peers have).

6.3 Parameterize the number of peers

To determine the scalability of our protocol in terms of the number of peers, we have run multiple simulations, increasing the number of peers, while keeping the number of moderators, moderations and forwarders constant. Doing this allows us to determine the relation between the number of peers and the propagation time. We ran these simulations with the following parameters:

#Moderators	#Moderations	#Forwarders	#Peers
1	1	0	100
1	1	0	500
1	1	0	1000
1	1	0	2000

Figure 6.6 show the propagation of the created moderation for the four different number of peers. The vertical axis gives the percentage of peers that have the moderation, compared to the number of peers that are online and have the torrent belonging to this moderation. The horizontal axis denotes the time in hours and the grey areas again denote the offline times of the moderator.

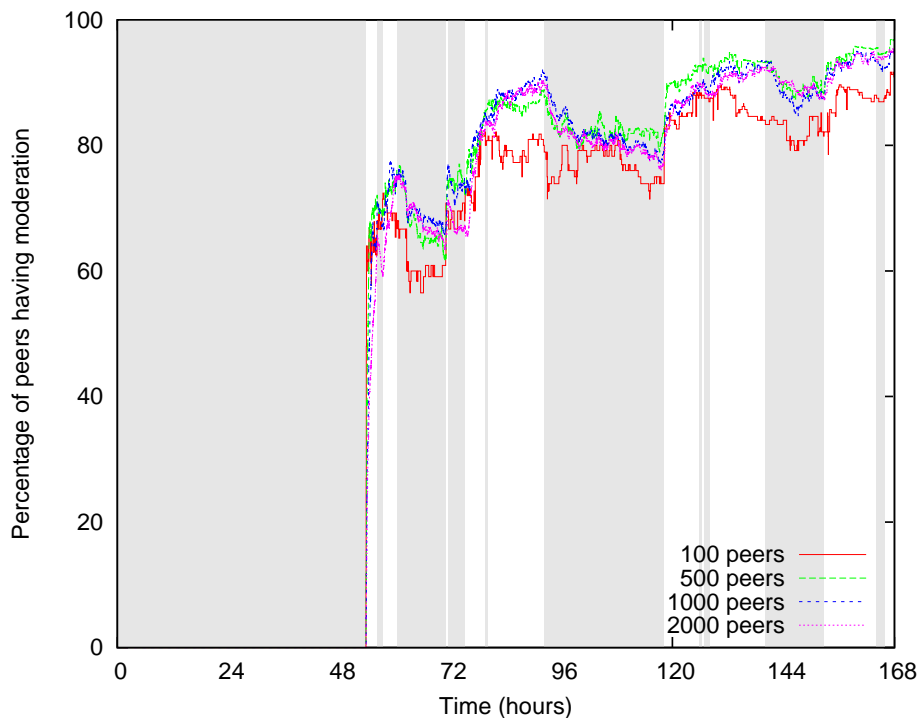


Figure 6.6: Percentage of online peers having a moderation, compared to the number of online peers having the torrent belonging to this moderation.

This Figure shows the moderator coming online after about 50 hours in the simulation. When it comes online it immediately creates a moderation for a single torrent, which then starts propagating through the network. Even though the moderator is offline for more than half the time, the moderation reaches more than 90% of the reachable peers within the remaining 5 days. Remember that this simulation does not include any forwarders and that as a result there is no one to gossip the moderation when the moderator is offline. This can also be seen in the decrease in propagation in the gray areas. The results for the different number of peers are very similar. Only the 100 peer simulation is a negative outlier. This is due to the fact that the Buddycast protocol will not reconnect to a peer within 4 hours of the previous connection with this peer. This is useful in a network with a lot of peers, but becomes a problem in a network where there are a very low number of peers. Because Buddycast connects to a peer every 15 seconds, this results in 240 outgoing connections per hour. Because every outgoing connection is received by a peer, there will be on average 240 incoming connections per hour. This results in $480 \text{ connections/hour} * 4 \text{ hours} = 1920 \text{ connections/4 hours}$. Therefore any lower number than this will be hampered in performance by this policy. The reason why this policy is implemented in Buddycast is to reduce bandwidth consumption while ensuring that peers have an equal chance of being served. For the 100 peer simulation this results in long periods of idleness in the network. From Figure 6.6 we can conclude that our protocol scales well as the number of peers grows.

6.4 Pollution removal

Pollution, which we define as inserting unwanted information into a system, is a big problem on the Internet, due to its open nature. Examples of this unwanted practice are e-mail spam and the encyclo-vandalism on Wikipedia. Because our system allows people to share information to a huge crowd, we should expect some pollution to take place in our system. We have taken the following measures to make pollution less tempting and less problematic:

Local block list If a peer finds some metadata in a moderation of poor quality, it can simply improve this part of the metadata. However, if the peer thinks that the moderator that created this moderation acted in non-productive way, such as polluting, it can choose to block the moderator permanently. This is done, based on the PermID of the moderator, which cannot be spoofed as the moderation is signed by a digital signature using public key encryption. This way, pollution is punished because the polluter will reach less peers the next time. Blocking a moderator will result in removing all moderations created by this moderator on the local system.

Forwarding Moderating costs upload-bandwidth. To relieve some of the upload-bandwidth burden, peers can indicate that they are willing to forward moderations for certain moderators. Because peers that are polluting will probably not have such help, they will have to do all the uploading themselves. In Section 6.2 we saw that help of forwarders will also increase the speed at which the moderation propagates through the network. This gives the good moderators, which contribute productively to the system, an edge over the bad moderators, which pollute the system.

To determine how well forwarding helps with the removal of pollution from the network, we ran the following experiment. We will assume that peers will not forward for peers that pollute the system. Therefore a polluting moderator must do all the work himself. In this experiment a bad moderator will moderate a torrent with polluted metadata. After 8 hours a good moderator will notice this polluted metadata and will re-moderate it with his own correct moderation. We ran the simulation with the following parameters, the total number of peers again is 2000:

# good forwarders	# bad forwarders
0	0
4	0
8	0
16	0

The results from these simulations are given in Figures 6.7 and 6.8. Figure 6.7 shows the propagation in percentage of all online peers that have the torrent, for the various numbers of forwarders. The red line is the propagation of the bad moderation and the blue line is the propagation of the good moderation. Further, the red bars show the offline times of the bad moderator and the blue bars show the offline times of the good moderator.

All the setups show similar propagation for both good and bad moderator (see Figure 6.7). All the setups also show that the bad moderation reaches more than 95% of the reachable peers within less than the first 8 hours. During these first 8 hours the bad moderation is the only moderation in the system. After the creation of a new moderation, for all the setups, the propagation decreases very rapidly after the creation of the new moderation. Within 8 hours of creation the new moderation is able to reach more than 80% of the reachable peers. This limits the propagation of the bad moderation to 20% of the reachable peers. This may be partially because the bad moderator has gone offline. As soon as the roles turn and the good moderator is offline when the bad moderator is online, the role of the forwarders becomes clear. Without forwarders, the propagation of bad moderations increases again and the propagation of good moderations decreases. This can be explained by peers

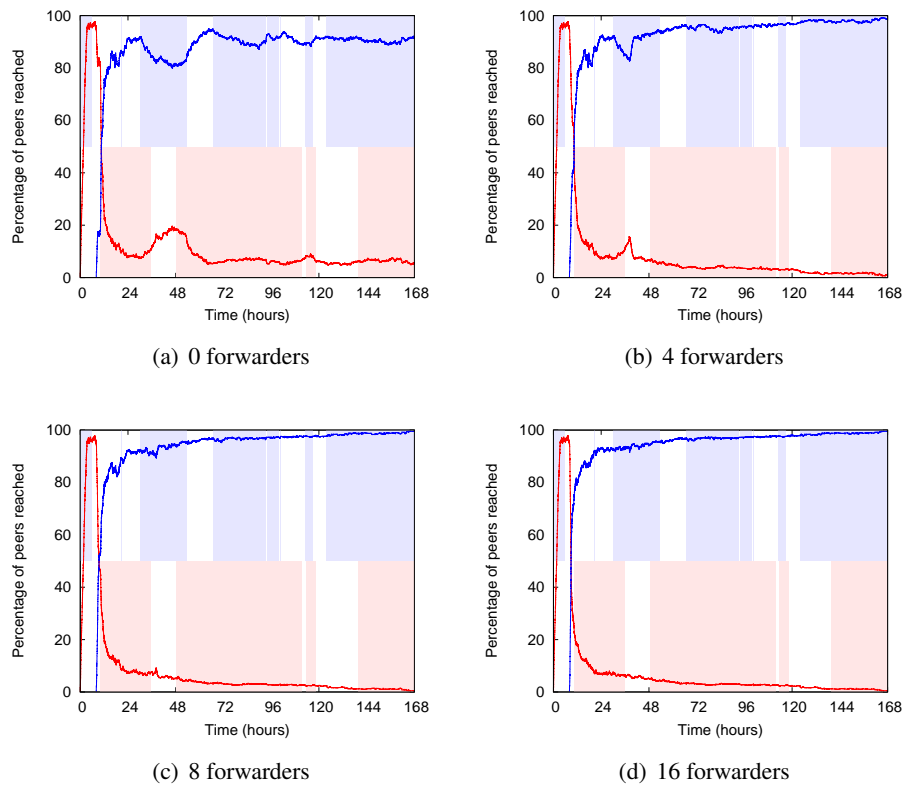


Figure 6.7: Propagation of one bad (red) and one good moderation (blue) in percentages. Propagation is the number of online peers that have the moderation, divided by the number of online peers that have the torrent. Red areas show the offline times of the good moderator and blue areas the offline times of the bad moderator.

without moderation coming online and receiving the bad moderation. However with a higher number of forwarders, this increase of bad moderation propagation can be stopped. This can be seen in the spike in the red line, just after 36 hours. This spike decreases from 0 to 4 forwarders, becomes almost unnoticeable for 8 forwarders and is not visible for 16 forwarders. This clearly illustrates the importance of forwarders. The good moderator has an advantage over the bad moderator as he is able to go offline, while the moderation is still spreading.

Figure 6.8 shows the upload bandwidth usage of both the moderators for the different number of forwarders. Keep in mind that the forwarders only forward for the good moderator.

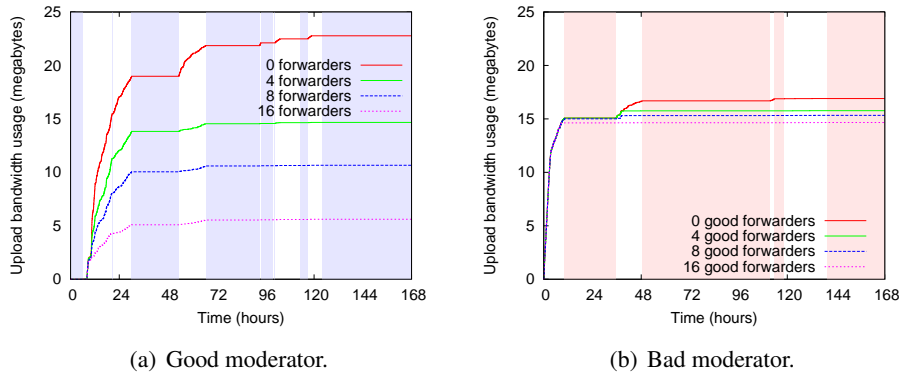


Figure 6.8: Upload bandwidth usage for one good (a) and one bad moderator (b).

For the bad moderator (see Figure 6.8(b)), the total upload bandwidth usage is between 15 and 17 megabytes. This depends very little on the number of good forwarders. The reason why it depends a little is because as the number of good forwarders increases, the good moderation spreads faster. This leaves the bad moderator with less peers to infect and thus reduces the upload bandwidth requirement for the bad moderator. One important thing to note is that although the upload bandwidth may not seem very large, the upload bandwidth is used mostly during the first hours. During these hours the upload bandwidth usage is around 1 kilobyte per second for this single moderation.

For the good moderator (see Figure 6.8(a)), the total upload bandwidth usage depends very much on the number of forwarders. For the 0 forwarder setup, the total upload bandwidth usage is around 24 megabytes. This is more than that of the bad moderator, because the good moderation propagates to more peers. Again we note that the upload bandwidth is used mostly during the first hours after moderation. If the number of forwarders is larger, both the total upload bandwidth usage and the length of the initial period of increased upload bandwidth usage decrease. For the 16 forwarder setup, the total upload bandwidth requirement is reduced to around 6 megabytes and the length of the initial period of increased upload bandwidth usage is about an hour. This shows that the good moderator has an edge over the bad moderator when he has some forwarders. This helps reduce his upload-bandwidth usage. This is less of an issue if the moderation is relatively small, which is the case if the moderation is text-only for instance.

Chapter 7

Conclusions and Future Work

In this Chapter we give a short summary of the problem we solved in this thesis and state our conclusions. Then we propose future research that can be conducted in the area of metadata distribution in peer-to-peer systems.

7.1 Summary and Conclusions

Metadata, extra information that is added to files, can help to search and browse through large collections of video. In peer-to-peer video systems this is useful as there is a large collection of videos and downloading the wrong video can cost a lot of time. This, because the files are large and bandwidth is limited. For central systems such as YouTube, metadata for video is already available. However, there are two problems with this approach; it is not scalable and the quality of the metadata is low. The scalability issue is due to the central system and the low metadata quality is because only the uploader of a video may add metadata.

In order to provide good quality metadata, with a scalable and bandwidth efficient architecture, we present MODERATIONCAST. MODERATIONCAST uses a combination of Wikipedia-like editing and a peer-to-peer architecture. With this approach we hope to solve the quality issue using peer production [5]. Further we solve the lack of scalability that is imposed by centralized systems. In the design and implementation of MODERATIONCAST, we have taken into account; scalability, bandwidth efficiency and security. Further, we have taken countermeasures against pollution by enabling peers to block malicious peers and remove bad metadata. Good moderators on the other hand can be rewarded by forwarding moderations for them. This reduces the bandwidth requirement for good moderators and in-

creases their moderation propagation speed through the network.

For evaluation of MODERATIONCAST, we have created a trace-based simulator. We have used this simulator to conduct large scale trace-based simulations of moderation-distribution in a 2000 peer Tribler network. From our results we can conclude that:

- **MODERATIONCAST is scalable.** For an increasing number of peers, from 100 to 2000, the propagation speed of moderations remains stable.
- **MODERATIONCAST deals with pollution.** Apart from the ability to block bad moderators, peers can also re-moderate bad moderations. With the help of forwarders, good moderators have a significant edge over bad moderators. Their upload-bandwidth requirement is much lower and they may go offline, while their moderations would still propagate through their forwarders.
- **MODERATIONCAST is bandwidth efficient.** Because of the use of forwarders, bandwidth requirements for the MODERATIONCAST epidemic protocol are relatively low. Given a 25KB moderation, which may include much larger, on-demand items, a 2000 peer network can easily be served with only 16 forwarders. Even for a popular torrent, this only results in an average of 0.4 KB per second upload-bandwidth requirement for the moderator.
- **MODERATIONCAST has high availability.** Although moderators may go offline, MODERATIONCAST can keep spreading their moderations. This because forwarders keep spreading their moderations even when the moderator is offline.

7.2 Future Work

During our thesis work, we have concluded that the following improvements should be made to MODERATIONCAST:

- **Incentivation** Currently there are no incentives for peers to share their moderations with other peers. Although adding metadata to ones downloaded files does not require altruistic behavior, sharing this data does. Therefore peers could use altered clients that do not share their moderations with other peers. Incorporating an incentive mechanism for sharing moderations would be a significant improvement as it would encourage peers to keep sharing their moderations. One possible solution would be to include MODERATIONCAST bandwidth counting to Bartercast [24].

- **More advanced pollution prevention measures.** Although Wikipedia has shown to work well, we would be very interested to evaluate pollution prevention measures other than local blacklisting and forwarders in MODERATIONCAST. Possibilities include non-local blacklists, whitelists, banning IP-addresses and voting. Robust voting, within the context of Tribler, is currently researched in [28].
- **Bloom filters** Currently in the have-messages random and recent moderations are selected from the ones a peer is willing to show to other peers. It is important that a have-message contains enough moderations that another peer does not yet have. Otherwise, bandwidth is used and no moderations are spread. A possible improvement would be to use a Bloom filter [7] instead of a list of moderation-IDs. This would increase the number of requested moderations for a given have-message size.

Further, we believe that the next step would be to integrate MODERATIONCAST with Tribler and to gather results from the system in a real world scenario with high usage. This is required because complex simulations which model real life nuances are difficult to setup as there are no traces or models for users' moderation behavior. Another problem with simulation is the amount of resources needed to simulate more peers than we have done so far. For our current simulations we already required more than 8 gigabytes of memory.

Bibliography

- [1] Azureus distributed database. http://www.azureuswiki.com/index.php/Distributed_hash_table, May 2008.
- [2] Azureus peer exchange. http://www.azureuswiki.com/index.php/Peer_Exchange, May 2008.
- [3] J. Barreto and P. Ferreira. *Euro-Par 2005 Parallel Processing*, volume 3648, chapter An Efficient and Fault-Tolerant Update Commitment Protocol for Weakly Connected Replicas, pages 1059–1068. Springer Berlin / Heidelberg, 2005.
- [4] Bencode. <http://en.wikipedia.org/wiki/Bencode>, May 2008.
- [5] Y. Benkler. *The Wealth of Networks: How Social Production Transforms Markets and Freedom*. Yale University Press, 2006.
- [6] T. Berners-Lee. The world wide web: Past, present and future. <http://www.w3.org/People/Berners-Lee/1996/ppf.html>, May 2008.
- [7] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
- [8] J. Byers, J. Considine, and M. Mitzenmacher. *Peer-to-Peer Systems II*, volume 2735, chapter Simple Load Balancing for Distributed Hash Tables, pages 80–87. Springer Berlin / Heidelberg, 2003.
- [9] B. Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39(1):11–16, 1991.
- [10] B. Cohen. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer Systems*, 2003.
- [11] Dublin core metadata element set, version 1.1. <http://www.dublincore.org/documents/dces/>, Januari 2008.
- [12] Emule. <http://www.emule-project.net>, May 2008.
- [13] J. Falkner, M. Piatek, J.P. John, A. Krishnamurthy, and T. Anderson. Profiling a million user dht. In *Proceedings of the ACM SIGCOMM conference on Internet measurement*, volume 7, pages 129–134, 2007.
- [14] Filelist. <http://www.filelist.org>, May 2008.
- [15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [16] J. Giles. Internet encyclopaedias go head to head. *Nature*, 438:900–901, December 2005.
- [17] S. Higgins. What are metadata standards? - metadata and digital curation. <http://www.dcc.ac.uk/resource/standards-watch/what-are-metadata-standards.pdf>, August 2006.
- [18] Ipoque internet study 2007. http://www.ipoque.com/news_&_events/internet_studies/internet_study_2007, May 2008.

- [19] Iso 639-3 language coding. <http://www.sil.org/iso639%2D3/>, May 2008.
- [20] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The eigentrust algorithm for reputation management in p2p networks. In *Proceedings of the 12th international conference on World Wide Web*, pages 640–651, 2003.
- [21] Khashmir. <http://khashmir.sourceforge.net>, May 2008.
- [22] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
- [23] P. Maymounkov and D. Mazires. *Peer-to-Peer Systems*, volume 2429, chapter Kademia: A Peer-to-peer Information System Based on the XOR Metric, pages 53–65. Springer Berlin / Heidelberg, 2002.
- [24] M. Meulpolder. Bartercast. <https://www.tribler.org/BarterCast>, May 2008.
- [25] Network time protocol. <http://www.ntp.org/>, May 2008.
- [26] Secure, permanent peer identifiers. <https://www.tribler.org/PermID>, May 2008.
- [27] J. A. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D. H. J. Epema, M. Reinders, M. R. van Steen, and H. J. Sips. Tribler: a social-based peer-to-peer system. *Concurrency and Computation: Practice and Experience*, 20(2):127–138, 2007.
- [28] R. Rahman, D. Hales, M. Meulpolder, M. Clements, V. Heinink, J. Pouwelse, and H. Sips. Robust vote sampling in a p2p media distribution system. PDS technical report PDS-2008-004, Technical University of Delft, 2008.
- [29] M. Raynal. About logical clocks for distributed systems. *ACM SIGOPS Operating Systems Review*, 26(1):41–48, 1992.
- [30] J. Roozenburg. Secure decentralized swarm discovery in tribler. MSc thesis, Delft University of Technology, November 2006.
- [31] D. Schoder, K. Fischbach, and C. Schmitt. *Peer to Peer Computing: The Evolution of a Disruptive Technology*, chapter Core Concepts in Peer-to-Peer Networking, pages 1–28. Idea Group Inc., 2004.
- [32] E. Sit and R. Morris. *Peer-to-Peer Systems*, volume 2429, chapter Security Considerations for Peer-to-Peer Distributed Hash Tables, pages 261–269. Springer Berlin / Heidelberg, 2002.
- [33] Skype. <http://www.skype.com>, May 2008.
- [34] D. Stutzbach and R. Rejaie. Understanding churn in peer-to-peer networks. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 189–202, New York, NY, USA, 2006. ACM.
- [35] Tribler. <http://www.tribler.org>, May 2008.
- [36] S. Voulgaris. *Epidemic Based Self-Organisation in Peer-to-Peer Systems*. PhD thesis, VU Amsterdam, De Boelelaan 1105, October 2006.
- [37] S. Voulgaris and M. van Steen. *Self-Managing Distributed Systems*, volume 2867, chapter An Epidemic Protocol for Managing Routing Tables in Very Large Peer-to-Peer Networks, pages 299–308. Springer Berlin / Heidelberg, 2003.
- [38] J. Wang, J. Pouwelse, J. Fokker, A.P. de Vries, and M.J.T. Reinders. Personalization on a peer-to-peer television system. *Multimedia Tools and Applications*, 36(1-2):89–113, January 2008.
- [39] Wikipedia. <http://www.wikipedia.org>, May 2008.
- [40] Metadata. <http://en.wikipedia.org/wiki/Metadata>, May 2008.
- [41] Youtube video posting site. <http://www.youtube.com>, May 2008.

- [42] B. Yu and M. P. Singh. *Cooperative Information Agents IV - The Future of Information Agents in Cyberspace*, volume 1860, chapter A Social Mechanism of Reputation Management in Electronic Communities, pages 154–165. Springer Berlin / Heidelberg, 2004.
- [43] Zlib compression library. <http://www.zlib.net>, May 2008.