Delft University of Technology
Parallel and Distributed Systems Report Series

Robust Vote Sampling in a P2P Media Distribution System

Rameez Rahman, David Hales, Michel Meulpolder,
Maarten Clements, Vincent Heinink, Johan Pouwelse and Henk Sips
r.rahman@tudelft.nl

report number PDS-2008-004

ISSN 1387-2109

Published and produced by:
Parallel and Distributed Systems Section
Faculty of Information Technology and Systems Department of Technical Mathematics and Informatics
Delft University of Technology
Zuidplantsoen 4
2628 BZ Delft
The Netherlands

Information about Parallel and Distributed Systems Report Series:
reports@pds.twi.tudelft.nl

Information about Parallel and Distributed Systems Section:
http://pds.twi.tudelft.nl/

CONTENTS

Abstract

The explosion of freely available media content through BitTorrent file-sharing networks over the Internet means that users need guides or recommendations to find the right, high quality, content. Current systems rely on centralized servers to aggregate, rate and moderate metadata for this purpose. We present the design and simulations, using real BitTorrent traces, for a method combining fully decentralized metadata dissemination, vote sampling and ranking for deployment in the Tribler.org BitTorrent media client. Our design provides robustness to spam attacks, where metadata does not reflect the content it is attached to, by controlling metadata spreading and by vote sampling based on a collusion proof *experience* function. Our design is light-weight, fully decentralized and offers good performance and robustness under realistic conditions.

I. INTRODUCTION

THE BitTorrent protocol has transformed the distribution of large media files due to it's decentralized, scalable, efficient and robust peer-to-peer (P2P) architecture [1]. Peers share bandwidth to help distribute files of common interest. However, the protocol excludes mechanisms for searching, rating, and associating descriptive metadata to, content.

Consequently, to locate high quality content available for download, users often rely on web-based systems that provide both links to content (.torrent files) and associated metadata such as a text description, a thumbnail graphic, a URL to associated information on the web and other useful information. This allows users to search and browse available content before making a selection of what to download.

Web-based systems rely on user contributions of both .torrent files, that point to available content, and metadata that describes content. They are administered centrally so malicious content or incorrect metadata can be manually removed and the users who posted them excluded. Creating new identities in such systems involves some level of user cost since identities must be created on the web system prior to posting information. Hence such systems cannot entirely stop anti-social behavior but they make it costly because creating new user identities involves some time and effort.

A number of BitTorrent clients have, recently, integrated their own search and metadata systems such that users can locate and browse available content conveniently from within the client before downloading (e.g. Vuze[1] and Miro[2]). However, again, these rely on centrally administered servers to store and and serve metadata requiring the creation of user accounts.

In this paper we propose a design for a fully distributed metadata dissemination and rating system which provides similar functionality to centralized systems. Low quality metadata such as spam or incorrect information is combated through a distributed rating system based on the sampling of user votes (or ratings) in favor (or against) those peers who submit metadata (who we term moderators).

As with centralized systems we do not eliminate the possibility of anti-social behavior but we make it costly and difficult without the need for central servers or administration.

The target platform for our design is the Tribler[3] media client [13] but the design is generic enough that it should be applicable in other media sharing contexts where decentralized and robust metadata dissemination and rating are required. Tribler implements a non-spoofable distributed peer identity system using a public key infrastructure. This means that all communication between nodes is signed and bound to a known source identity, thus preventing forged or stolen identities. Also, the Tribler client provides local database services allowing state to be maintained over sessions.

The paper is structured as follows. In Section II we give an overview of the system. In Section III we briefly discuss the peer sampling service - which provides an essential service required by our design. In Section IV we describe our metadata dissemination mechanism. In Section V we present our vote sampling approach and in Section VI we present simulation results based on real BitTorrent peer traces. In Section VII we discuss some vulnerabilities and possible refinements of our design. In Section VIII we present a summary of related work. Finally we conclude with a summary of our contribution and possible future work.

[1] http://vuze.com
[2] http://www.getmiro.com/
[3] http://tribler.org

II. Design Overview

In this section we will describe the overall design of our system and briefly introduce all its building blocks.[4] We propose a method by which nodes in a P2P file-sharing system can submit, distribute and rank metadata by voting for those who introduce metadata into the system.

Any user in the system can inject metadata about any content item at any time. We term users who inject metadata into the system "moderators" and the metadata they submit as "moderations". Moderations are bound to a particular user identity (the moderator) and a particular media file (through association with a BitTorrent .torrent file). Moderations include a small thumbnail picture, a short descriptive free text field and a list of tags (or keywords). Hence moderations are small in size and a number of them can be exchanged in a single message between nodes.

All nodes have one of three evaluations on moderators - either a positive vote (+), a negative vote (-) or no vote (null). For a node to express a positive or negative vote the users must explicitly click a relevant button on their screens next to a displayed moderation from the associated moderator. The default setting for all moderators is initially null.

Over time, peers forward moderations from moderators they have voted positively for, to other nodes they encounter. Nodes discover each other via a peer sampling service (or PSS). The PSS is briefly discussed in Section III below. The moderation dissemination protocol is described in Section IV.

In addition to forwarding moderations based on their own individual vote, nodes ask other nodes that they encounter, again via the PSS, for their votes on moderators. Each node stores these votes in a local "BallotBox" cache which over time represents an increasingly accurate sample of the votes of other nodes in the entire population. Nodes use the sample stored in the BallotBox to rank moderators and their associated moderations. The BallotBox protocol is discussed in Section V-A.

It should be noted that providing the users with the ability to accurately rank moderators is one of the major motivations of this paper. Moderations can be used to improve the keyword search system in Tribler. Currently when a user initiates a search by entering a keyword, a set of matching .torrent files are returned in no particular order. With moderator ranking, the search list can be ranked based on the moderators' popularity as determined by the BallotBox.

In order to frustrate spam moderators gaining a large number of votes (by creating many identities and voting for themselves) we utilize a distributed mechanism based on an *experience function* which imposes a cost on new identities before their votes are accepted by other nodes.

In this way we create a kind of "two-tier" identity system. New nodes entering the system can do so with little cost (cheap identities) gaining all the benefits of full membership but being unable to pass votes to others until they have built-up a certain level of experience in the system. To build such experience implies a cost and faking experience is difficult and also costly. The experience function is presented in Section V-B.

The main contribution of this paper is to demonstrate how this scheme can be implemented in a fully distributed way without central servers or overheads for honest nodes.

[4]We apply a KISS design principle: *Keep It Simple Stupid!*. We provide what we consider to be the minimum functionality to affect an initial deployment.

III. PEER SAMPLING SERVICE

We assume each peer has access to a peer sampling service (PSS) which periodically returns a random peer from the entire population of online peers. This allows nodes to discover others and potentially exchange messages with them.

There are several ways to implement a PSS in a distributed and robust way. One approach uses *gossiping or epidemic protocols*. Such approaches maintain a random-like overlay network in which nodes regularly exchange their neighbor lists (or view) with others. Such PSS protocols have been shown to be robust, self-repairing, completely decentralized and scalable to tens of millions of nodes [8].

Our target system, the already deployed Tribler system, implements a variant of Newscast [6] called BuddyCast [13]. BuddyCast implements several refinements over pure random sampling of nodes but for the purposes of this paper the only relevant difference is that any particular online node is only returned once within a 4 hour time window. Put another way, the random sampling is *without replacement* within this time window. BuddyCast supplies a new random node, sampled from the online population, approximately every 15 seconds.

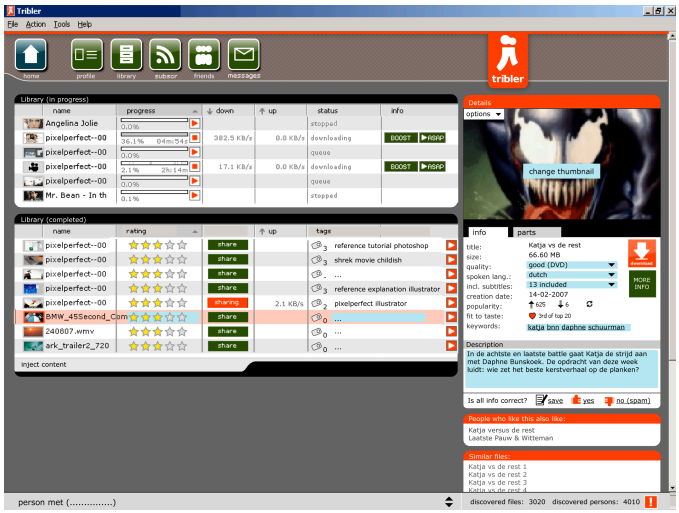

Fig. 1. Screenshot of the prototype Tribler (version 5.0) client allowing users to vote on moderators. Notice the ratings on the left and the thumbs-up and thumbs-down (vote) buttons in red on the right. Above the thumbs metadata for a content item is displayed.

```
do forever
  wait Δ
  j ← GetRandomNode()
  ml_i ← Extract(local_db)
  Send ml_i to j
  ml_j ← Receive(j)
  local_db ← Merge(local_db, ml_j)
```

(a) active thread

```
do forever
  ml_i ← receive(*)
  ml_j ← Extract(local_db)
  Send ml_j to i
  local_db ← Merge(local_db, ml_i)
```

(b) passive thread

Fig. 2. The push / pull gossip based metadata dissemination protocol. GetRandomNode is supplied by the PSS. The local_db stores received moderations. The Extract() function returns the moderations list (ml) sent to other nodes. The Merge() function inserts new moderations into the local_db. These operations take account of local node votes and moderation recency criteria.

IV. Metadata Dissemination

In this section, we will describe how metadata is disseminated in our system. The low level details of the metadata dissemination protocol, called ModerationCast, and extensive simulations, are given in [5]. Here we present the main features relevant to the voting mechanisms described later.

Moderations are disseminated in a gossip-like fashion to other peers by using the PSS. However, nodes only pass on metadata from those moderators they have approved. Approval involves the user explicitly selecting a thumbs-up icon displayed next to the metadata from the given moderator indicating a positive (+) vote for the moderator. Users may also disapprove of a moderator by selecting a thumbs-down indicating a negative (-) vote. Figure 1 shows a prototype screen from the Tribler client that allows users to vote in such a way.

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

Essentially then, the idea is that, "good" moderators, as judged by the approval of others, will spread their metadata quickly but "bad" moderators, obtaining low numbers of approvals and / or disapproval, will only be able to spread their metadata slowly. However, it is important to note that even bad moderators can spread their data to others through direct interactions with nodes that have not already indicated disapproval. Figure 3 shows a schematic diagram showing how a moderation spreads in the population based on approvals and disprovals by other nodes.

Because moderators can be blocked based on their moderations and forwarders are forwarding moderations for (other) moderators, there is need for authentication of moderations. This is done using public key signatures. This

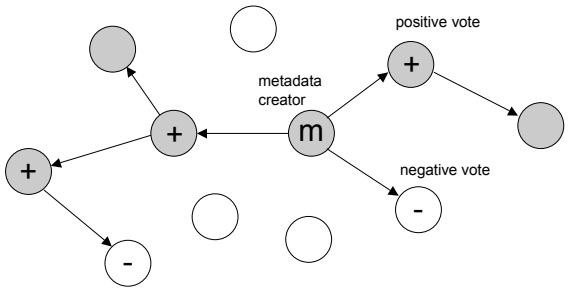

Fig. 3. Diagram illustrating how moderations spread from a moderator node that creates the metadata (m) and nodes that give the moderator positive votes (+). Shaded nodes have received and stored the metadata in their local_db. Nodes that give negative votes (-) or no votes (null) do not pass on the medadata.

prevents alteration of moderations, which could lead to bad moderations spreading very well and good moderators being blocked because of malicious peers forwarding altered or fabricated moderations on behalf of the good moderator. Figure 2 shows outline pseudocode for the gossip based metadata dissemination protocol.

V. VOTE SAMPLING

In this section we describe how votes are sampled and collected by peers in the network. We designed two related protocols: BallotBox and VoxPopuli and an associated *experience function*.

These protocols support a two-tier identity system because the population is partitioned into an experienced core and an inexperienced periphery. Figure 5 illustrates this view of the system schematically.

Figure 4 shows outline pseudocode for the two protocols. We describe each in turn below.

A. BallotBox protocol

As previously stated votes are generated by users registering approval (a positive vote) or disapproval (a negative vote) against moderators. Each peer node stores a list of the votes the local user has made in a structure we term the *local vote list*. Each entry in the local vote list contains a pair mapping a unique moderator ID to a vote (either positive or negative) plus a time stamp indicating when the user made the vote. Moderators may only appear once in the list since a user is only allowed to make one vote against each unique moderator. Moderators with null votes do not appear in the list. The length of the list therefore indicates the total number of votes cast by the local user. It is a record of their local voting pattern. It can be thought of as a ballot paper that the users fill in as they make votes.

Periodically each node i selects another node j from the population randomly (using the PSS). Each node then applies an experience function E to the other to determine if to request the local vote list of the other (Section V-B below discusses the E function in detail). Requested nodes communicate their local vote list to the other. Nodes then merge received vote lists into a structure we term the *local ballot box*. The local ballot box is a list in which each entry contains four items: mapping a unique moderator ID to a vote, a time stamp and a unique peer ID. The local ballot box is similar in format to the local vote list except that moderators may appear several times in the list, recording votes for the same moderator received from different peers. Also the time stamp records when the vote was received, by the local node, rather than made by the remote node.

Essentially then, each peer individually conducts its own poll by asking other randomly selected peers directly to supply their local vote list. Hence pairs of peers meet randomly and exchange votes, building, over time, a sample of the votes of the population in their local ballot boxes.

Nodes do not forward or share the accumulated information in their local ballot box with other peers. This precludes certain kinds of malicious vote manipulation where a node could lie about the votes received from others. But this means that each peer can only accumulate a sample of the population votes, based on its direct experience, not a globally accurate total count. Faster and more accurate epidemic-style aggregation protocols have been proposed but they are highly vulnerable to lying behaviour [7].

The local ballot box has a maximum size of B_{max} sets of votes from unique peers - beyond which new votes replace the oldest votes. Hence BallotBox determines voting statistics from a maximum sample of B_{max} other peers. Assuming the PSS produces random samples and B_{max} is large enough then we can expect the local cache to converge to a reasonable accuracy.

The BallotBox protocol, therefore, is similar to an "opinion poll" as carried out by polling organisations when attempting to determine the opinion of an entire population on some matter of interest. In general such polls ask individuals directly their own opinion but not what they believe others opinions are. The BallotBox turns every peer into a pollster enquiring on moderators.

The local ballot box is not deemed to be sufficiently populated until in contains at least B_{min} unique peers votes. After this threshold is reached it can be used to calculate a ranking of moderators.

In order to take the raw votes from the local ballot box and produce a ranking of moderators we order moderators by their scores. Scores are calculated by summing all votes in the local ballot box for each moderator - where a positive vote contributes +1 and a negative vote -1. Hence for each moderator M we produce a value by summing all +1 votes for M and adding the sum of all negative votes for M. We then order the list of moderators by this value.

The final ranked list can be used as input to relevance ranking of content on the users screen after a keyword search operation. The raw result of a search operation is an unordered set of matching .torrent files which point to content items. Where these items are associated with metadata produced by ranked moderators they are ordered based on moderator rank.

Another possible use for the vote sample information is to display a screen listing the top-K moderators themselves along with their estimated percentage of the popular vote and other associated information. We believe such a screen

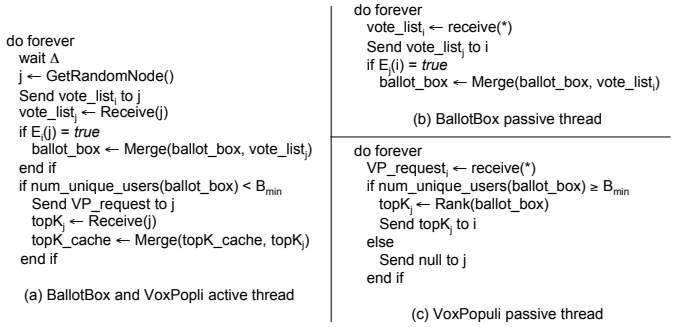

Fig. 4. The BallotBox and Voxpuli protocols comprise one active (a) and two passive threads (b and c). Notice the active thread always executes the BallotBox protocol but only conditionally executes the VoxPopuli protocol - when the number of unique votes in the ballot box is below some threshold B_{min}. The passive thread (c) only responds with a top-K list when that node j is itself not executing VoxPopuli. This prevents nodes unwittingly passing potentially malicious top-K lists received from others.

could psychologically incentivise moderators to produce good moderations since they can see themselves rise in the ranks of listed moderators as others vote for them.

B. Experience function

Since new identities are cheap, immediate voting power would enable and encourage Sybil [3] and collusive flash crowd attacks (where a large number of new peers join the system with the explicit aim of promoting a moderator by voting for them for nefarious purposes). We therefore enforce that new nodes joining the system can not register votes with others until they are considered "experienced" by the receiving nodes. In other words, any node should only take a vote from another node into account when this other node is experienced.

We define the general *experience function* as a binary function E that determines whether or not a node is considered to be experienced. Any candidate E function must be implementable in a fully distributed way and be robust to attempts to fake experience (i.e. to immediately and cheaply generate experienced identities). While the overall design of our voting system does not assume a particular definition of experience, in the implementation that we will present in this paper we consider a node experienced when it has contributed a certain amount of data to the peer population by sharing files.

In order to determine the contribution of a node in a safe way we use the BarterCast protocol [11], which is deployed in the Tribler system. In essence, by using BarterCast, any node in the system can estimate the contribution of any other node (that it knows via the PSS) based on up- and download statistics that are exchanged among nodes in a reliable way. First, nodes record statistics of their own BitTorrent file-transfers. Second, nodes exchange their own direct statistics with other peers they encounter. Based on these combined statistics each peer can build a graph of the network with directed edges that denote the amount of MBs transferred from one node to another node. The protocol then applies a maxflow algorithm to derive peer contributions. The maxflow approach highly reduces the effect of (collaborations of) nodes that provide false information in order to fake experience. This approach and its properties are described in detail in [4] and [11].

In the remainder of this paper, we assume that node i can compute a reliable *contribution value* for a node j. We denote the contribution of j to i by $f_{i \to j}$. For the purposes of implementing our experience function E, we apply a simple threshold value T over the contribution function $f_{j \to i}$. Hence node i considers node j to be experienced where:

$$E_i(j) = \begin{cases} true & \text{iff } f_{j \to i} \geq T; \\ false & \text{otherwise.} \end{cases}$$

In our current design we determined an appropriate T value via simulations based on real BitTorrent traces (see Section VI). However, as we discuss later, T could be adapted dynamically to adjust to differing conditions.

C. VoxPopuli protocol

The BallotBox protocol involves nodes constructing a sample of votes based on direct interaction with individual peers, who are considered experienced. Therefore, new nodes entering the system need time to build a reasonably

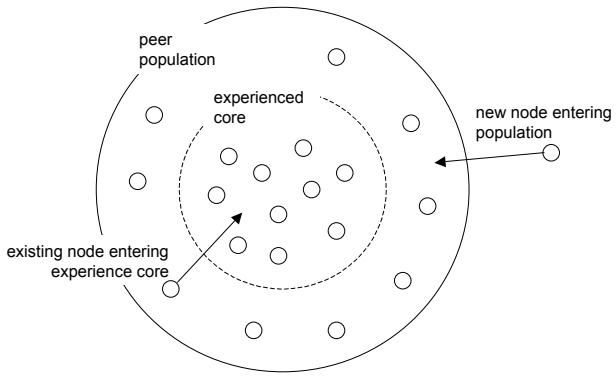

Fig. 5. Diagram illustrating the concept of a peer population comprising an experienced core of nodes and new nodes entering the system. Over time new nodes will enter the experienced core as other nodes recognise them as experienced.

sized sample before they can extract meaningful statistics. This may take some time because new nodes by definition will not be considered as experienced by existing nodes in the population until they have uploaded a sufficient amount during BitTorrent interactions with others. A minimum sample size of B_{min} votes from other peers is required before any statistics are used. During this bootstapping process nodes use the VoxPopuli protocol - a less accurate but speedier protocol which dispenses with the need for individual vote counting or the application of an experience function.

When executing the VoxPopuli, nodes request from others (encountered via the PSS) a rank list of the top-K moderators. No experience function is applied. Only those nodes which are *not* themselves executing VoxPopuli respond based on their current local ballot box statistics - by producing a ranked list of moderators truncated to a maximum size of K and sending this to the requesting node.

Each node executing VoxPopuli maintains a local cache of the last V_{max} top-K lists received and performs a merge operation to produce its own top-K list which is used for relevance ranking. There is no minimum sample size required so after the first VoxPopuli exchange a node can rank moderators.

Any rank merging method could be used. We apply simple averaging of the rank of each moderator over all stored top-K lists. Where a moderator does not appear in a list they are assumed to have rank K+1 for that list. Rank ties are resolved based on the total number of times a moderator appears in all lists or, if this does not resolve the tie, arbitrary ranking based on moderator ID is used.

VoxPopuli is therefore quick (in the order of a few seconds) but vulnerable to inexperienced and colluding malicious peers - if a sufficient number enter the system such that they form a large enough proportion of the total peer population and they promote the same spam moderators.

VI. Simulation results

In order to test our proposed design we performed simulations based on real traces from the Filelist.org BitTorrent tracker obtained in January 2006. It is possible to trace the behavior of unique peers over multiple sessions and multiple swarms with this tracker. This is not possible with public trackers such as mininova.org. The traces record the size of the files that are shared in each swarm and the connectability of peers (i.e. if they are behind a firewall or freely connectable).

The traces capture the realistic high churn rates found in deployed P2P systems. On average only 50% of the total population of nodes are online at any given time[5]. For our simulations we used a dataset of 10 unique traces of 7 day duration monitoring 100 unique peers. Each trace records approximately 23,000 unique events making a total of 2.3×10^4 events[6].

Our simulations operate at the BitTorrent file piece level. This means we simulate every action that a BitTorrent client would need to take, down to the exchange of file chunks, peer choking and piece selection. We simulate nodes that are predisposed to seed files altruistically after they have downloaded and those which free-ride by leaving swarms as soon as they have downloaded their file. Such detailed simulations are non-trival to implement and require significant computational power to run.

In the following sections we perform simulation experiments that demonstrate:

- The generation of an experienced core - Section VI-A
- The performance of vote sampling protocols - Section VI-B
- The performance of the system under spam attacks by a massive number of colluders who attempt to promote spam metadata - Section VI-C

A. Experience formation

We used trace based simulations to determine how quickly our system would produce an experienced core for given threshold values T for the experience function E (as described in Section V-B). In order to gain a general picture of the formation of experience between nodes we calculated a time series for a value we term the Collective Experience Value (CEV). Since E is a binary and non-symetrical function that can be applied to any pair of ordered nodes we calculated the average of E applied to all ordered pairs of nodes i, j where $E_i(j) = true$ contributes a value of 1 and $E_i(j) = false$ contributes a value of zero, hence:

$$e_i(j) = \begin{cases} 1 & \text{iff } E_i(j) = true; \\ 0 & \text{otherwise.} \end{cases}$$

$$CEV = \frac{1}{N} \sum_{i \in N} \sum_{j \neq i} \frac{e_i(j)}{N-1}$$

where N is the known total population size[7].

The CEV is therefore a form of directed graph density value in which edges are defined between nodes based on the experience function E. It characterises the amount of experience between nodes. Figure 6 shows simulation results based on a typical trace from the dataset for various threshold values T. We conducted these series of experiments in order to ascertain the time required for the generation of a sizeable experienced core for different T values.

Based on the results shown in figure 6 we selected the lower value of $T = 5$ MB for use in the further simulation experiments discussed below. This value was chosen pragmatically. The results indicate that approximately 20% of ordered node pairs produce experience within 12 hours. This indicates the early formation of a core but implies a time and upload cost is necessary before a node can enter it. This frustrates Sybil or flash crowd attacks while placing only modest costs on honest new peers entering the system.

Note, however, that even after seven days (168 hours) some nodes are still not part of the core. This is due to freeriding behaviour in which nodes download but upload little[8] and the fact that some nodes are rarely present in

[5]By total population we mean all nodes that enter the system - for however long - over the seven day period of the traces.

[6]The datasets used, and associated documentation, are publicly available at: http://davidhales.com/tom-data.zip

[7]The CEV value is therefore a measurement requiring global information and could not be calculated by individual nodes easily. We use our global knowledge of the traces to calculate the CEV value for our own experimental purposes. CEV plays no part in the protocols running in the nodes.

[8]Analysis of the traces showed that approximately 25% of peers uploaded little to others

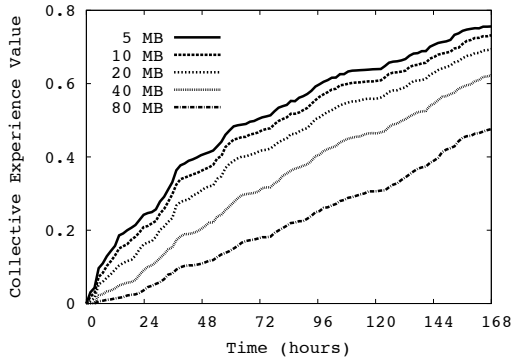

Fig. 6. Chart showing the Collective Experience Value (CEV) over time for a typical trace of BitTorrent activity. The CEV indicates the proportion of all ordered pairs of nodes (i, j) for which i considers j to be experienced. Hence when CEV=1 all nodes consider all others to be experienced. Results using five different threshold (T) values for the experience function E are shown.

the trace - they may enter and quickly leave the system spending most of the time offline. As stated previously, analysis of the traces indicated that churn rates are high with approximately 50% of nodes offline at any given time. This indicates the traces reflect actual BitTorrent client filesharing dynamics. It is important to note that the results in figure 6 show results for all peer nodes that comprise the total population over the seven day period not just those who are online at any given time. It therefore gives a global picture of the traced population rather than online snapshots.

B. Vote sampling

We tested the vote sampling protocols by running simulations in which a small number of moderators spread metadata and are allocated votes from other nodes when they receive this metadata. We set the first three nodes (M1, M2 and M3) entering the system to be moderators and to spread a moderation related to a .torrent file. We selected 10% of the population at random to provide a positive vote for M1 and 10% to provide a negative vote for M3. M2 gets no votes. Hence the correct ordering, based on the popular vote, should be $M1 > M2 > M3$.

For the BallotBox we set $B_{min} = 5$ and $B_{max} = 100$. The VoxPopuli cache size was set to $V_{max} = 10$ and for the top-K moderator lists $K = 3$.

The results for a typical run are shown in Figure 7. Notice that at approximately 12 hours there is a sharp rise. This is result of the bootstrapping properties of the VoxPopuli protocol. Recall that VoxPopuli allows nodes which have not yet received at least B_{min} votes to determine an ordering of moderators by randomly sampling the population requesting the top-K moderators from others who have received at least B_{min} votes. The steep rise indicates that a number of nodes have obtained these thresholds and can thus spread the top-K moderators to others. Hence nodes who have attained the required BallotBox sample size, since they have received enough votes from core nodes, share their rankings with new nodes entering the system - who cannot distinguish core nodes from other new nodes.

C. Spam attack

A major threat to our system is an attack on new normal nodes - those which have not yet received at least B_{min} votes from core nodes but act honestly - by a flash crowd of new nodes promoting a spam moderator. Such a flash crowd could be comprised of colluding nodes or the result of a Sybil [3] attack in which a single node creates and maintains multiple identities. In either case it is clear that if the size of the spam crowd is significantly larger than the online experienced core then the spam attack will be successful against newly entering normal nodes. However, over time, new nodes will recover from the attack when they have obtained at least $B_{min} = 5$ votes in their local ballot box. So correctly functioning new nodes will only be vulnerable to such an attack for some initial period of time. Figure 8 illustrates this kind of attack diagrammatically.

Figure 9 gives results from simulations based on the 10 traces. However, here we fixed 30 nodes to be part of the experienced core. We set the flash crowd size to 30 and 60 in size. As can be seen when the flash crowd is

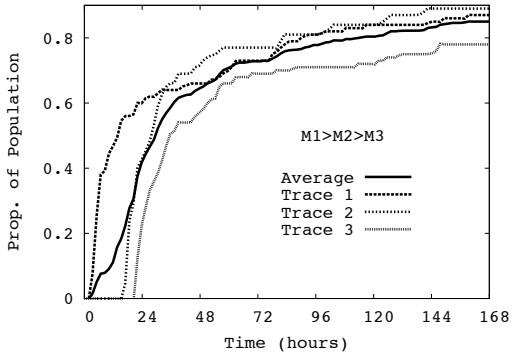

Fig. 7. Chart showing the effectiveness of the vote sampling system over time. The correct ordering is $M1 > M2 > M3$ based on votes. Voting nodes do not vote until they receive the appropriate moderations via the moderation dissemination protocol. Three typical runs are shown for different independent traces. The average is over 10 independent runs.

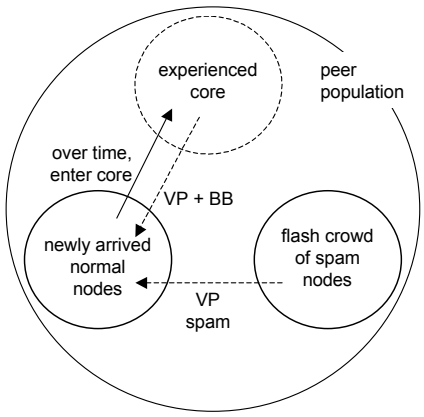

Fig. 8. Diagram illustrating a flash crowd attack on a population comprising an experienced core plus other newly arrived nodes. Dotted arrows show the flow of protocol messages between subpopulations: VoxPopuli (VP) and BallotBox (BB). Notice that the flash crowd can not influence the experienced core because core nodes do not not considered new nodes to be experienced. Over time newly entered normal nodes will eventually become experienced and enter the core.

2 x core size (60) then most new nodes are defeated (polluted by a spam moderator - M0) for approximately 24 hours. When the flash crowed is 1 x core size then only a minority of nodes are defeated initially.

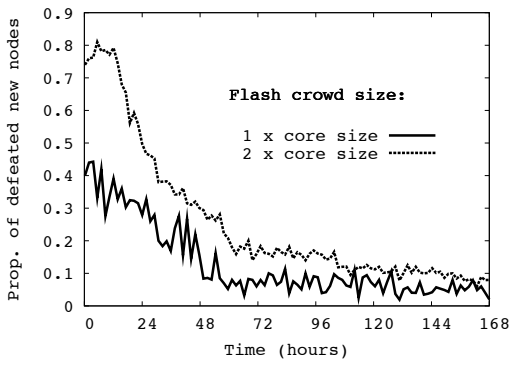

Fig. 9. Results of a spam attack. The population comprises an experienced core, a collusive flash crowd plus other newly arrived normal nodes. At the start of the run the entire core is converged on a top moderator M1. The flash crowd promote M0 as top moderator (representing a spam moderator). Lines show the proportion of newly arrived nodes ranking M0 top for different sizes of the flash crowd relative to the core size. For attack sizes smaller than 1 x core size, zero pollution is obtain within the first hour. Averages of 10 trace runs are shown.

VII. Discussion

The claim that the experience mechanism can discourage spam rests on two assumptions: 1) that the experienced core, over time, will become large and be populated by non-spam (or non-spam voting) nodes and 2) that spam nodes will not be able or prepared to pay the upload cost to join the experienced core.

Our first assumption is based on the idea that a population only becomes desirable for spamming if it is already large and that any small population that is already infected by spam nodes will not become large since users will go elsewhere and join other more productive communities. Hence we believe it is not unrealistic to assume that the founders (or elders) of any successful peer community are unlikely to be malicious. To put this another way, no successful community can form if the founders are malicious.

Our second assumption relies on the fact that to gain enough experienced identities to influence the popular vote the spam nodes would need to pay a high price in time and upload bandwidth - a price that would be too high to make an attack worthwhile. The larger the size of the core the higher the cost of an attack since more spam identities are needed to influence the vote. As we have demonstrated, even if the number of new malicious identities are equal in size to the existing core their voting influence is limited. Here we benefit from scaling - the larger the core the better the defence against attack.

However, if a sufficiently large set of colluding identities are willing to pay the cost, then our system can be defeated.

We selected a T value using global information and traces of past activity. A more robust approach would adapt the T value endogenously based on experience. We aim to explore adaptive T values in future work.

Another potential flaw in our approach is that *it is possible to fake experience* by clever collusion within the BarterCast protocol but this is difficult and again costly. A detailed analysis of possible attacks is given in [11]

Although such an attack would be complex and costly to mount it would appear that the incentives to commit it would increase as the core population size increased. This is because a large population is more desirable for spamming purposes but is no more robust than than a small to such attacks. Here we note the reverse scaling issue because as the core becomes large the incentives to attack increase but the cost of attack remain constant.

VIII. Related Work

System models, analysis, and performance evaluation of unstructured self-managing rating systems (UMR) and structured supervising rating (SSR) systems are presented in [15]. UMR is implemented in unstructured networks where every peer stores its rating of other peers on its superpeer. While, in SSR, each peer has a supervisor peer which is responsible for storing other peers' rating of that particular peer. Each supervisor node is located using a DHT [14] based algorithm.

Our approach of data dissemination and storage is based on a push-pull gossip approach which differs from both these approaches. Furthermore, like most work on ratings, their approach assumes that all peers will issue ratings after each transaction with other peers. In the real world, of our media distribution scenario, this is an unrealistic assumption since user rating activity is often much lower - most users do not take the time to rate or vote on content.

Additionally [15] gives a detailed account of a "personalized credibility sub-system"' which bears some resemblance to our experience function. Each peer, i, maintains a personalized credibility set (PCS) that contains the peers who vote honestly on i. However, unlike our system, ratings from outside the credibility set are also accepted albeit with lower weights. This approach also differs from ours in how it deals with new peers. While we bootstrap the new peers with randomly sampled top-K ratings (see Section V-C), no method for bootstrapping new peers is presented.

EigenTrust [9] uses an algorithm which is similar to the PageRank algorithm by Google [12]. It provides globally consistent trust values for peers in a network using DHT techniques. However, it is computationally expensive and doesn't perform well in networks with high churn. [2] presents a system in which peers ask for ratings on particular peers by broadcasting poll messages. This is basically a UMR system. In the PeerTrust system [10], ratings on a peer are maintained by its supervising peer. This makes it an SSR system in the terminology of [15]. Such "supervisory" systems are often impractical and not suitable for real world implementation and deployment.

IX. Conclusion

We have presented the design and simulations of a metadata dissemination and ranking system applicable to the deployed Tribler P2P media client. Three protocols were introduced: ModerationCast, BallotBox and VoxPopuli. They work together to spread and rate matadata robustly.

We aimed to produce a light-weight, self-organising and robust design that is realistic enough to be deployed and requires zero server support.

We have built on existing deployed protocols BuddyCast and BarterCast to provide a peer sampling service (PSS) and a robust experience function.

However, we consider our design to be generic enough that other PSS protocols and experience functions could be used. Specifically any distributed trust metric that can rate nodes in a robust way could be adapted as basis for an experience function and hence can be used to support a robust distributed vote sampling model.

We have discussed the vulnerabilities in our approach and indicated why we consider these to be tolerable in realistic environments. We also indicated some future lines of research that may address some open issues.

Our design approach has been to consider the social dynamics that emerge from the interaction of peer nodes over time. We have focused on group formation and boundary recognition as key factors in understanding and engineering productive social interactions at several scales. We aim to develop this approach in future research towards what we term a *Tribal Programming* paradigm.

To our knowledge Tribler is currently the only deployed P2P file-sharing system using a fully distributed PSS to augment the BitTorrent protocol to provide proactive media discovery and seeding incentives. To this we add a fully distributed metadata dissemination and rating system.

In future work (the next major release of Tribler) we will deploy our proposed mechanism and obtain measurements directly from the live system.

Acknowledgment

The authors would like to thank Jelle Roozenberg and Jie Yang for collecting and processing the data traces we used as the basis for our simulations and Maarten ten Brinke for the prototype Tribler v5.0 GUI allowing users to vote on moderations. We also thank Prof. Dick Epema and Dr Tamas Vinko for their valuable comments on earlier drafts of this paper. This work was partially supported by the European Commission within the FP7 Programme - Project: P2P-Next[9].

References

[1] B. Cohen. Incentives build robustness in bittorrent. In Workshop on Economics of Peer-to-Peer Systems, Berkley, USA, May 2003.
[2] Damiani, E., di Vimercati, D., Paraboschi, S. and Samarati, P. (2003) A reputation-based approach for choosing reliable resources in peer-to-peer networks. *Proc. CCS 02*, Washington, DC, November 18-22, pp. 207-216. ACM Press, New York, NY.
[3] John R. Douceur. The sybil attack. In *Proc. of the 1st International workshop on Peer-to-Peer Systems (IPTPS02) Workshop*, Cambridge, MA (USA), March 2002.
[4] M. Feldman, K. Lai, I. Stoica, and J. Chuang. Robust incentives techniques for peer-to-peer networks. In *Proc. of 5th ACM conference on Electronic Commerce (EC'04)*, pp. 102-111, May 2004.
[5] Vincent Heinink. *Metadata Infrastructure for Video on Demand*. Technical University of Delft, PDS technical report PDS-2008-003, 2008.
[6] M. Jelasity, W. Kowalczyk, and M. vanSteen. Newscast computing. Technical Report IR-CS-006, Vrije Universiteit Amsterdam, Department of Computer Science, Amsterdam, The Netherlands, Nov. 2003.
[7] M. Jelasity, A. Montresor, O. Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst.* 23(3): pp. 219-252, 2005.
[8] Jelasity, M., Voulgaris, S., Guerraoui, R., Kermarrec, A., and van Steen, M. 2007. Gossip-based peer sampling. ACM Trans. Comput. Syst. 25, 3, Aug. 2007.
[9] S.D Kamvar, M.T. Schlosser, and H. Garcia-Molina. The eigentrust algorithm for reputation management in p2p networks. In *12th WWW conference*, New York, 2003.
[10] Xiong, L. and Liu, L. (2004) Peertrust: supporting reputation-based trust for peer-to-peer electronic communities. *IEEE Trans. Data Knowl. Eng*., 16, 843-857
[11] M. Meulpolder, J.A. Pouwelse, D.H.J. Epema, H.J. Sips. Fully distributed sharing-ratio enforcement in bittorrent. Technical University of Delft, PDS technical report PDS-2008-002, 2008. http://www.pds.ewi.tudelft.nl/~meulpold/papers/bartercast_techreport.pdf
[12] Page, Lawrence; Brin, Sergey; Motwani, Rajeev; Winograd, Terry *The pagerank citation ranking: bringing order to the web.*. SIDL-WP-1999-0120, Stanford University. Standford, CA. 1999.
[13] 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*, Vol. 20, 127-138, 2008.
[14] Stoica, I., Morris, R., Liben-Nowell, D., Karger, D. R., Kaashoek, M. F., Dabek, F. and Balakrishnan, H. Chord: a scalable peer-to-peer lookup service for internet applications. Proc. SIGCOMM 01, San Diego, CA, August 27 31, 149 160. ACM Press, New York, NY. 2001.
[15] Ye Tian, Di Wu and Kam-Wing Ng. On Distributed Rating Systems for Peer-to-Peer Networks. The Computer Journal Vol. 51 No. 2, 2008

[9] http://www.p2p-next.org/