Networking 2013 1569702823

Open2Edit: a peer-to-peer platform for collaboration

Niels Zeilemaker, Mihai Capotă, and Johan Pouwelse
Delft University of Technology, The Netherlands
niels@zeilemaker.nl

***Abstract*—Peer-to-peer systems owe much of their success to user contributed resources like storage space and bandwidth. At the same time, popular collaborative systems like Wikipedia or StackExchange are built around user-contributed knowledge, judgement, and expertise. In this paper, we combine peer-to-peer and collaborative systems to create Open2Edit, a peer-to-peer platform for collaboration.**

Open2Edit provides the means for users to publish, modify, comment on, and categorize content. Content-centric collaboration in Open2Edit takes place within communities. A flexible permission system allows each community to customize the privileges users receive for interacting with content. Any user can create a new community, and Open2Edit facilitates their discovery. Furthermore, Open2Edit allows users to vote for communities, and keeps track of community popularity.

We deploy Open2Edit in Tribler, a BitTorrent-based peer-to-peer application, in order to create a YouTube-like media sharing system. Through Open2Edit, users in Tribler can collaboratively improve media metadata using a wiki-like approach. We present results from 7.5 months of usage that show Open2Edit is a viable, sustainable platform that leads to the emergence of collaboration.

I. INTRODUCTION

Peer-to-peer (P2P) systems such as BitTorrent have proved that it is possible to create a scalable, fault-tolerant file sharing system which depends only on infrastructure resources donated by users. Similarly, collaborative websites such as Wikipedia and StackExchange show that it is possible to build a repository of free knowledge and expertise using intellectual resources donated by world-wide communities of users. However, these collaborative websites, while providing a free service, require significant investments in infrastructure to keep running. We believe that if both types of user contributions are combined, a system will emerge which can provide users with access to free knowledge and expertise, while requiring little investment in infrastructure.

In this paper, we introduce Open2Edit, a platform capable of combining donated bandwidth, storage space, and CPU cycles with knowledge, judgement, and expertise. Open2Edit is a P2P system that allow users to create communities in which they can collaborate without relying on centralized servers. Furthermore, the flexibility of Open2Edit allows for a wide range of deployment scenarios, ranging from a closed traditional news website in which users can only comment on articles, to a community very similar to Wikipedia in which users can modify almost anything.

Within a community, the users of Open2Edit engage in content-centric collaboration: adding new content, editing, commenting, and categorizing. Any user can create a new community and grant other users a diverse range of permissions, including the ability to perform the content-centric actions mentioned above, but also the permission to grant permissions. Open2Edit allows users to vote on the quality of communities and aggregates the votes to promote the best communities.

Data synchronization among peers is an important issue in designing a P2P collaboration platform. In Open2Edit, we use the Dispersy data replication engine [1] to address this issue. Dispersy provides low-level functionality for synchronizing data between peers, a permission system, and NAT-traversal. In Dispersy, peers communicate through overlays and use Bloom filters to efficiently replicate peer data.

With extensive emulation, we show that Open2Edit can successfully synchronize data between 1000 peers. Furthermore, the experiment proves that Open2Edit load balances user contributed bandwidth, and requires a minimal overhead when community activity is low.

We show the maturity of the Open2Edit platform through our Tribler [2] deployment. Using Open2Edit we created a YouTube-like media sharing system based on Tribler, a BitTorrent-based P2P application. Besides file sharing, Tribler implements remote search, video-on-demand, and a reputation system. In addition to these features, using Open2Edit communities, we implement channels as a means for users to publish, modify, comment on, and categorize media items. Channel creators can make full use of the Open2Edit permission system to make their channels more or less open.

By monitoring Tribler, we can state that Open2Edit is a viable platform for collaboration proven to work in an Internet deployment. During 7.5 months of data collection, over 2000 channels were created by almost 90 000 users who cast over 100 000 votes expressing their preference for channels.

The rest of the paper is organized as follows. In Section II, we review related work. We present the motivation for building Open2Edit in Section III. In Section IV, we describe the Dispersy data replication engine. The design of Open2Edit is described in Section V. In Section VI, we present the Tribler deployment. The results of our experiments are discussed in Section VII.

II. RELATED WORK

P2P systems such as BitTorrent [3] have shown that a scalable and fault tolerant file sharing system can be created even by depending only on user contributed resources. In BitTorrent, a separate overlay is created for each file being shared. Within an overlay, peers use an incentive mechanism

called *tit-for-tat* which rewards user contributions with faster download speed.

The largest example of a knowledge-oriented collaboration platform is Wikipedia. Using Internet volunteers it has become an open alternative to traditionally edited encyclopedias. Wikipedia currently has more than 4 million articles written in English and over 40 other languages with more than 100 000 articles. When a user modifies an article, a new revision is saved. This new revision of the article supersedes the previous one, but no revisions are removed from the history. A small army of bots (automated scripts) monitor new revisions and detects vandalism – intentional errors introduced by users which corrupt articles. Wikipedia articles are text based, but users can upload images and movies to enrich articles. Furthermore, a special markup language is employed to allow users to create sections, lists, add references, and link to other articles. Wikipedia uses donations to pay for website hosting and its employees, and currently requires roughly $30 million per year [1].

Alternatively, Wikia allows users to create many wikis in order to address different topics. It is currently hosting over 280 000 wikis[2]. Every wiki is allowed to have its own look-and-feel, improving the sense of community. Furthermore, links between communities can be established. Finally, in contrast to Wikipedia, Wikia uses ads to generate the income necessary to pay for hosting.

Stack Exchange represents a different type of collaboration platform, one upon which *Question & Answer* websites can be build. These Q&A websites allow users to pose a question to which other users can provide answers. Together, users vote for the best answer to a question. Moderators can modify, close, and delete posts and user profiles. Similar to Wikipedia, moderators are elected by users themselves. An example of a deployed Q&A website using the Stack Exchange platform is Stack Overflow. Mamykina et al. [4] state that in Stack Overflow more than 90% of questions asked are answered, and that the median answer time is only 11 minutes. Currently (January 2013), Stack Overflow has 4.2 million questions and 8.1 million answers[3].

In contrast to the centralized collaboration systems described above, there have been attempts at creating decentralized collaboration systems. These can be divided into two categories. The first focus on providing a decentralized wiki and try to automatically merge conflicting copies of an article. Examples of such systems are the XWiki Concerto [5], and its extensions Wooki [6] and Swooki [7]. These systems provide collaboration by storing modifications made by users as either *ins* or *del* operations. Operations made on a single article, can be merged at a later stage allowing for decentralized collaboration. However, no explicit details are given on how modifications of an article are transferred between peers besides stating that they use anti-entropy.

[1] https://blog.wikimedia.org/2012/01/02/wikimedia-fundraiser-concludes-with-record-breaking-donations/

[2] http://www.wikia.com/

[3] http://data.stackexchange.com/

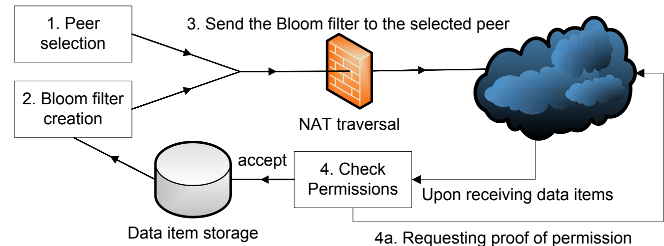

Fig. 1. Dispersy data replication procedure.

The second category of decentralized collaboration systems focus on distributing the load of a system like Wikipedia over a number of peers. These systems usually replicate articles in a DHT ring, and push articles to the peer closest to where the article should be stored in the ring. One example is Piki [8], which is build upon Pastry. However, the authors do not address the scalability issues which arise due to implementing search using the DHT, nor the potential problems caused by peer churn in the system.

III. Motivation

In this paper, we design and deploy a peer-to-peer platform for collaboration. Our aim is to provide the flexibility needed for a range of use cases. At the same time, we intend to create a uniform architecture that is content centric and supports fine-grained permissions.

As described in the related work section, there have been attempts at creating decentralized collaboration systems. However, most papers focus on creating a decentralized version of Wikipedia. Other types of collaboration websites, such as Stack Exchange, are not considered while building these systems. Moreover, the systems implementing a decentralized Wikipedia, such as XWiki Concerto, cannot be easily modified into a general collaboration platform as they only support the Wikipedia use case.

In a nutshell, we address the problem of building a P2P platform for collaboration by starting with the Dispersy data replication engine and adding the features necessary to accommodate user collaboration. The main feature we add is the possibility for users to create communities, in which collaboration takes place. In addition, we define permission sets that users can easily apply to their communities to customize their level of openness. Finally, we enable users to discover popular communities through a voting system.

IV. Dispersy

Dispersy is a general purpose P2P data replication engine, created as part of the Tribler project. In this section, we give an overview of Dispersy. For more details, please refer to our technical report [1].

A. Overview

Each peer which is replicating data in Dispersy is running a five step synchronization procedure, as shown in Figure 1: 1) Select a peer from the candidate list. This list contains all

locally known peers with which a peer can freely communicate (without being hindered by NAT-firewalls). 2) Create a compact hash representation (Bloom filter) containing the data the peer has stored in its local database. 3) Send the Bloom filter to the peer which was selected in step one. 4) Wait for a fixed interval and while waiting process the data items which are received from other peers. For each data item, a peer checks if the peer who created the item had the permissions to do so. If it was not known that a peer had this permission, the proof is requested from the peer who sent the data item. 5) Goto step 1 and repeat. In the following sections we will extend upon this brief description and highlight some key properties of Dispersy.

B. Data replication

Each Dispersy peer has a local database containing data items to be replicated among the other peers. A *data item* is a binary blob either created by a peer locally, or received from other peers. Data items are application dependent and have a variable length. In order to efficiently replicate data items between peers, Dispersy uses *Bloom filters*. Bloom filters were first described in 1970 by Bloom [9] and since then have become popular due to being a very space efficient method for membership testing. A Bloom filter contains a compact hash-representation of the data items in the local database of a peer. When receiving a Bloom filter, a peer uses its local database to detect data items missing from the filter. When missing data items are found, they are sent to the peer that generated the filter, in order to replicate them at that peer.

One problem associated with Bloom filters is caused by false positives. A false positive occurs when an item, which was not put into a Bloom filter, is considered present. Due to false positives, a peer is prevented from receiving some data items as they are incorrectly considered part of the filter. Dispersy solves this problem by introducing a random salt in each Bloom filter. This causes the data items which tested false positive in each Bloom filter to differ and hence have a minimal impact on the replication performance.

Another problem associated with Bloom filters is their limited capacity. For practical purposes, a filter transmitted over the network cannot grow indefinitely in size and thus can only accommodate a limited number of data items. In order to allow for an unbounded number data items to be replicated between peers, Dispersy includes only a subset of data items in the Bloom filter. When sending a replication message, indications on how the peer has selected this local subset are included such that the receiving peer can find missing data items in the same subset in its database. Subsets are defined by a heuristic which can be customized based on the needs of the application using Dispersy.

C. Overlays

Peers in Dispersy communicate through *overlays*, which are application-layer unstructured networks. For each overlay, a peer maintains a list of peers it can connect to, and periodically communicates with them in order to replicate data items.

Dispersy is designed to be modular and extensible by allowing applications to define new *overlay types* for specific goals. Defining an overlay type involves deciding what are the data items that should be replicated by Dispersy. Data items are actually a subset of the messages exchanged by Dispersy peers. The overlay definition includes message type definitions. Each message type definition includes a replication policy which designates messages of that type as data items to be replicated or as auxiliary protocol messages not to be stored in the database and replicated.

After defining an overlay type, any peer can initialize an instance of an overlay by creating a public/private key pair for it. The public key is used as the *identifier* of the overlay and must be known by other peers in order to join it. The private key is used to grant and revoke permissions to and from peers as we will show in Section IV-E. Dispersy overlays are completely separate from each other, however any peer can join multiple overlays. Data items are created by peers targeted to a specific overlay instance, and thus are not shared between overlay instances.

D. Peer discovery

After discovering the identifier of an overlay, peers attempt to connect to it by contacting peers already in the overlay. Each overlay has well-known peers that are used for bootstrapping. In order to maintain a good connectivity and achieve data replication, peers regularly, at a fixed interval, request a new peer from one of the already discovered peers in the overlay. The reply they receive contains the IP address and port of the new peer.

Dispersy also uses the peer discovery mechanism to puncture NAT-firewalls. This approach is similar the one described by Halkes et al. [10] and ensures reliable communication between unconnectable peers using help from intermediaries. In a nutshell, when unconnectable peer p, behind a NAT-firewall, wants to discover a new peer, also potentially unconnectable, it sends a request for a peer to intermediary peer i. As described above, i will reply with the IP address and port of a random peer n. At the same time, i instructs n to send a message to p, which will puncture n's NAT-firewall. Peer p can now send messages to n which will pass through n's NAT-firewall.

E. Permissions

The permission system is an important feature of Dispersy. Permissions are implemented at the overlay level and they refer to creating messages, revoking messages, and delegating to other peers the possibility to grant permissions. The private key of an overlay is used to grant and revoke permissions to or from peers in an overlay.

When defining the messages in a new overlay type, a developer can specify whether messages are protected by permissions or accessible to all peers. By default, all peers can create all messages in an overlay. However, when message creation is restricted, peers creating messages need to provide other peers a proof of being granted the permission to create messages of that type.

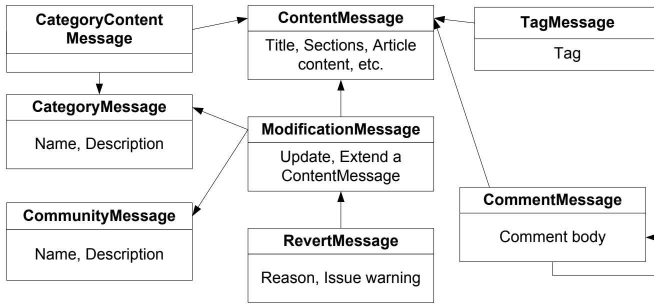

Fig. 2. Open2Edit CommunityOverlay messages.

These permission messages are also replicated by Dispersy as data items, which allows peers to detect when a permission is revoked. Granting and revoking permissions is based on time. As there is no real concept of time in a decentralized system, Dispersy uses an implementation of Lamport's logical clock [11]. The unique *identifier* of a message in Dispersy is the combination of the clock and the user id of the user which created the message. The identifier, together with a sequence number (messages restricted by a permission require all peers to include a sequence number in these messages) provides the platform with the tools to revoke a permission from a peer. A revoke message concerning a peer specifies the time and sequence number of the last valid message created by the peer. After this time, all messages created by the peer are dropped by peers who received the revoke message. If a peer receives a revoke message after already accepting an invalid message, then the incorrectly accepted messages are rolled back. If a peer attempts to spoof messages by re-using valid old time/sequence number combinations, all messages of this peer will be removed from the system, as messages with the same sequence number are not allowed to exist in an overlay.

In order to provide developers with some flexibility while defining their overlay types, Dispersy allows them to use dynamic permissions. This allows an overlay instance to switch between requiring permissions to create a message and not requiring them. By sending a *DynamicPermissionMessage*, an overlay creator can open up, or lock down functionalities in their overlay.

V. Open2Edit design

In this section, we describe the design of Open2Edit using the framework provided by Dispersy. Collaboration between peers takes place in a new type of overlay that we create, called CommunityOverlay, while community creation and discovery takes place in an overlay of type CommunityDiscoveryOverlay.

A. *CommunityOverlay*

An Open2Edit community is an instance of a Dispersy overlay type called CommunityOverlay. In order to implement the CommunityOverlay, we define eight message types, depicted in Figure 2.

A community has a name and description, which are set by a community moderator through a *CommunityMessage*. By default, the peer creating the CommunityOverlay instance is the only moderator, but can grant the moderator permissions to other peers to have a group of moderators controlling the community.

To create content, peers use the *ContentMessage*. This message is used to replicate an initial copy of the content, e.g., in a Wiki, it can be used to publish the first revision of an article; in a forum, it can be the opening-post of a thread; in a news-website it can be the news article, etc. There is no predefined structure in the ContentMessage, it is implemented as a dictionary (a list of key-value pairs).

Modifications can be made to a ContentMessage, using the *ModificationMessage*. A ModificationMessage contains a reference to the ContentMessage, a key-value pair that modifies or extends the dictionary of the ContentMessage, and optionally, a reference to the previous ModificationMessage it is replacing.

Peers can post comments using the *CommentMessage*, which has a comment field, a reference to the ContentMessage it is a comment on, and two references to previous comments. The first reference is used for replying to another comment, while the second reference points to the latest received comment and is used for ordering comments.

Open2Edit also allows peers to post tags on a ContentMessage. Using the *TagMessage*, a peer can tag the message using free-form text. The tags from all users can be used, for example, to create tag clouds for each ContentMessage.

Community moderators can also create well-defined categories using the *CategoryMessage*. A category has a name, and a description. Any peer can assign a ContentMessage to a category using a *CategoryContentMessage*, but a ContentMessage can be part of multiple categories.

The CommentMessage and ModificationMessage can also be used for commenting on and modifying other types of messages, not only ContentMessages. If a CategoryMessage is referenced, peers can comment on, and modify a global category instead of a ContentMessage. Similarly, the CommunityMessage can be commented on, and modified.

Finally, in order to have some tools for moderation in a community, we have defined a *RevertMessage*. This message has a reference to another message (e.g., a ModificationMessage), a reason for reverting it, and, optionally, a warning. Issuing a warning, will allow all other peers in the community to detect that a peer has intentionally done something wrong.

B. *CommunityDiscoveryOverlay*

In order to discover CommunityOverlays, peers join a CommunityDiscoveryOverlay using a locally known identifier. When deploying Open2Edit, a developer must create a new instance of the CommunityDiscoveryOverlay and include its identifier in the source code of the deployment. If two peers join CommunityDiscoveryOverlays with different identifiers, they will never meet, and hence will never discover the same

CommunityOverlays. For the CommunityDiscoveryOverlay, we have defined three message types.

A *CommunityCastMessage* is used to send other peers information on the communities a peer knows. Every peer creates a CommunityCastMessage at regular intervals (15 seconds), and sends it to a random peer. The CommunityCastMessage contains a list of community identifiers divided into three sets: a) communities the peer has created; b) communities the peer is a part of; and c) communities we know the peer has interacted with. This allows new peers to quickly determine which communities are popular.

Furthermore, we have defined a *VoteMessage* which allows peers to cast a vote on communities. A vote can be either positive or negative. A peer can only cast one vote per community, i.e., a newer vote replaces an older one. Using the votes, peers can quickly discover popular communities.

After discovering a new community identifier, either through CommunityCastMessages or VoteMessages, a peer will request a snapshot of the community. A *SnapshotMessage* is sent to the peer which advertised the new community, requesting a small collection of data items from the community. A snapshot will allow a peer to get a general feel for the community. If, after having received the snapshot, a peer is interested in more content, it must join the community. Open2Edit will then start looking for other peers in the community and contact them to replicate the data in the community. It will also create a possitive VoteMessage, indicating to other peers that it appreciates the community.

C. Authentication

In order to prevent users from impersonating other users, we configure Dispersy to add a signature to each message created. Furthermore, each message contains the ID of the user who created it. The user ID is implemented as a SHA-1 hash of the public key. If the public key of a user is not locally known, Dispersy requests this public key from the peer who sent the message. As this peer has accepted the message, it should be able to also provide the public key. After receiving the public key, Dispersy can verify the signature and confirm the user ID of the message creator.

D. Flexibility

In total Open2Edit defines 11 different message types, some of them providing only minor functionalities. This however, is intentional as it allows the community-creator (the peer who created a community) to define which peers can send what messages. An example, if Open2Edit was used to create a distributed news-website, then articles (ContentMessages) must only be created by a limited set of peers (the journalists). Modifications must be disabled, and commenting can be enabled for all peers, mimicking a traditional news-website. However, if we would like to employ crowd-sourcing it could be useful to let other peers categorize the articles, but to not let them create categories. If so, then the community-creator can allow all peers to create CategoryContentMessages, but prevent them from creating CategoryMessages. Opening up, or closing down functionalities uses the DynamicPermissionMessage as described in the previous section.

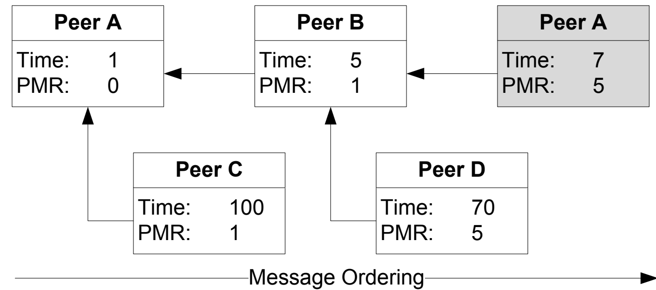

Fig. 3. Example of the global ordering of ModificationMessages using the prior modification reference (PMR).

E. Detecting conflicts

Conflicts are inherent to decentralized systems. In contrast to Xwiki concerto, Wooki and Swooki, Open2Edit does not implement conflict resolution algorithms. As Open2Edit is designed as a general collaboration platform, and not specifically as a decentralized Wiki, the additional complexity of such algorithms would hinder the flexibility as described in the previous section. However, we do implement a new method for providing a consistent ordering of ModificationMessages across peers.

Using the prior modification reference (PMR) in the ModificationMessages, we build a tree connecting all modifications. If a node in the tree is referenced by more than one child, a conflict occurred. When choosing between two children, we sort messages based on time (oldest first), and finally alphabetically based on the user IDs of the message creators. By default, Open2Edit uses the latest ModificationMessages to construct a representation of a ContentMessage. Figure 3 shows an example of an ordering by PMR. The message created by Peer A at time 7 is selected as the latest modification.

VI. Tribler deployment

We deploy Open2Edit in Tribler in order to create a YouTube-like media sharing system. In this section, we give an overview of Tribler and describe the features we add as a result of the Open2Edit deployment.

A. Tribler overview

Tribler is a BitTorrent-based file-sharing application that provides remote search, video-on-demand, live-streaming, and a reputation system, in addition to torrent downloads. Initiated in 2005, Tribler is designed to advance research in P2P technology. It has been downloaded more than one million times and it currently has a community of approximatively 3000 daily users.

One of the reasons BitTorrent became a success is the large number of communities that were created around it. Before the Open2Edit deployment, Tribler was lacking this social aspect, because it purposely removed the BitTorrent dependency on centralized websites by implementing decentralized search.

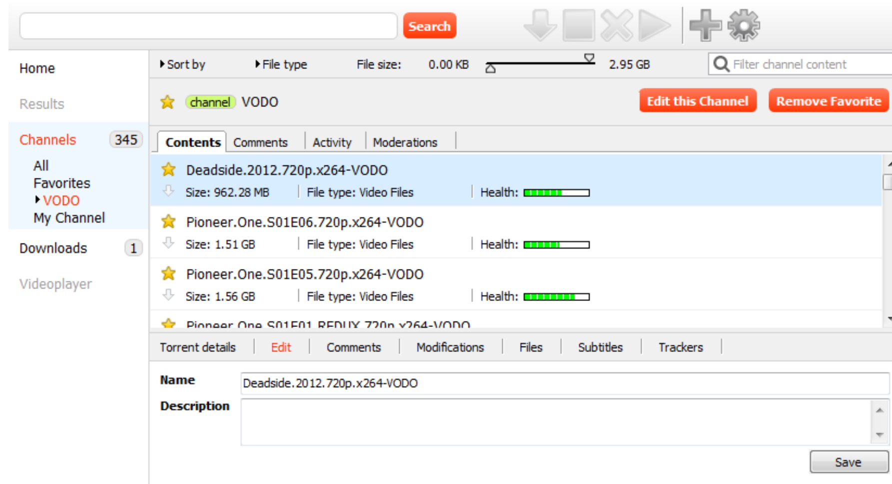

Fig. 4. Open2Edit deployed in Tribler, with a media item being modified in the edit tab, a Creative Commons licensed TV show from the VODO channel.

B. Implementation details

We add *channels* in Tribler as a means for user to collaborate in adding, modifying, commenting on, and categorizing media items like videos or photos. Channels are implemented as CommunityOverlays and media items are added to a channel using the ContentMessage. Media items, can then be improved by other users by modifying the name or description. Users can also create categories, and use them categorize the media items. Finally, users can comment on media items in a channel. Combined, all these features implement a decentralized YouTube-like system.

Furthermore, as we want to cater to the needs of different community creators, we implemented three levels of the openness of a channel: closed, semi-open, and open. Using the dynamic permissions of Dispersy we allow channel creators to switch between the different levels. When setting the level to open, all users can add, modify, comment-on, and categorize media items. The moderators can create categories, and remove comments and media items. The channel creator can grant moderator permissions to any user in the channel. The semi-open permission level for channels only allows users to comment on and categorize media items. Finally, in a closed channel only moderators can interact with the media items, while normal users can only download them.

C. User interface

While designing the user interface of Open2Edit in Tribler, we focused on exposing the channel activity to users. An open channel allows peers to perform almost all Open2Edit actions, and hence requires a method for users to monitor the community to prevent vandalism. By providing users with activity lists, similar to the recent changes in Wikipedia, users can monitor each other. This allows users to detect when vandalism occurs and easily revert it. This technique works well in Wikipedia, where 42% of damage to articles is reverted almost immediately [12]. A small sample of frequent Wikipedia users are actively monitoring the recent changes pages to prevent vandalism making an impact [13]. In Tribler, we also allow users to monitor the media items they have shown interest in by providing filters for the activity and moderation lists.

To encourage user participation, we implement methods to lower the barrier to entry for engaging in collaboration. This is very important as casual users can make significant contributions. Kanefsky et al. [14] study a NASA crowdsourcing website where volunteers identify craters on Mars and show that 37% of all craters were identified by users which only visited the website once.

The main user interface feature we implement to encourage collaboration is an edit mode for all editable entities, e.g., channels or item descriptions. Figure 4 shows the user interface as implemented and deployed in Tribler. The figure shows the editing of a media item called `Deadside.2012.720p.x246-VODO` (note the red highlighting for "Edit" in the lower part). The media item is part of a channel called `VODO`. Furthermore, the comments, activity, and moderations tabs for the channel are visible. These lists allow users to monitor the channel. Note also the star next to the channel name, signifying it is on the user's list of favourite channels. Marking a channel as favorite subscribes the user to the channel, thus initiating the replication process. Peers in Tribler do not automatically replicate all channels they discover, as that would result in significant overhead.

The user interface has other subtle features that help foster collaboration, like drag-and-drop categorization of media items and a "Thank you" button which allows users to quickly show their appreciation for people posting media items.

VII. Experiments

We run two experiments outlining the performance of Open2Edit. The first experiment is an emulation of Open2Edit on the DAS-4 supercomputer. The second concerns user activity in Tribler during the Internet deployment of Open2Edit.

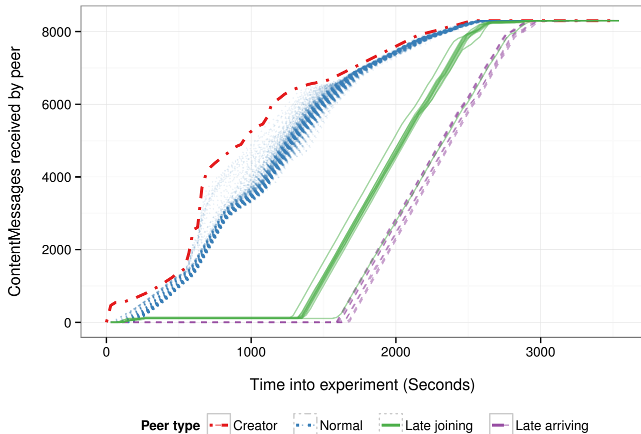

Fig. 5. Community experiment, number of ContentMessages received by peers.

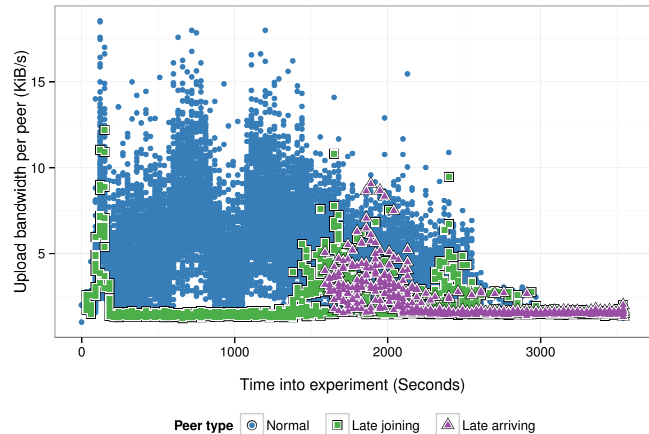

Fig. 6. Community experiment, upload bandwidth used by peers.

A. *DAS-4 emulation*

Using the DAS-4 supercomputer[4], we deploy 1000 peers onto 20 computing nodes. Peers run Open2Edit without any modifications. Although, we would like to perform an emulation of Open2Edit with more than 1000 peers, the nodes on the DAS-4 cannot cope with running more than 50 Open2Edit processes simultaneously. The experiment uses an activity trace from the Tribler Internet deployment. We assign one of four different roles to each peer. One peer is the community creator; 980 *normal peers* are online at the start of the experiment and join the community immediately; ten peers are *late-joining peers* that are online at the start of the experiment, but only join the community after 1250 seconds; and finally nine *late-arriving peers* come online and join the community after 1500 seconds.

Using an actual trace from a channel created by a user in Tribler, we assign the community creator with the task of creating ContentMessages. Each second in the experiment translates to approximatively 2 hours in the trace.

Discovering the CommunityOverlay instance can be achieved using two methods. The first method involves the CommunityCastMessage. A peer that has discovered a community can advertise it using CommunityCastMessages sent to other peers. The second method involves VoteMessages. All peers in the CommunityDiscoveryOverlay are replicating VoteMessages. After receiving a VoteMessage for an unknown community, a peer request a snapshot and thus discovers the community.

In this instance of Open2Edit, we configure peers to return at most 25 KiB worth of data in response to a replication request. This causes to maximum upload rate per peer to be 5 KiB/s as we are using a 5 second replication interval. Bloom filters were configured to accommodate at most 1000 data items.

[4]http://www.cs.vu.nl/das4/

Figures 5 and 6 depict the results of the emulation experiment. Note that we use different line types to differentiate between the four user roles. In Figure 5, the community creator, that is also creating the data items, always has more data items than the other peers. As time passes, the other peers synchronize their local databases and receive more and more data items until they have all of them, around 3000 seconds into the experiment.

Between 500 and 1000 seconds into the experiment, the community creator is creating new data items at a rate higher than 5 KiB/s. This causes the normal peers to not be able to keep up due to their configured maximum upload rate. After 1000 seconds, they are starting to catch up, and at 1500 seconds most data items are again replicated to all peers in the community at that time. In practice, the upload rate will not affect the replication speed, as in this experiment 1 second represents 2 hours in real life. Furthermore, both the replication interval and the upload limit are configurable when deploying Open2Edit.

The late-joining and late-arriving peers are showing the 25 KiB upload limit as well. Their slope is defined by this limit. Furthermore, from these two groups of peers we can see that the subset selection process is working very well. The peers were configured to create Bloom filters which can only accommodate 1000 data items. In this experiment, we have more than 8000 data items, hence the peers are selecting subsets of data items in order to replicate. The difference between late-joining and late-arriving peers can be seen in the figure as well, as the late-joining peers replicate a small number (100) of data items while online before joining. This is due to the snapshots they collect. The late-arriving peers do not collect snapshots, and hence do not have any data items.

Figure 6 shows the upload bandwidth usage per peer. We left out the community creator as this peer is sending each ContentMessage it created to 10 neighbors, and thus is incurring a much higher bandwidth usage. From the figure, we can see that normal peers have bandwidth spikes at the 600

TABLE I
INTERNET DEPLOYMENT DATA SUMMARY

Collection period	2011-12-09 to 2012-07-23
Users	89 819
Channels	2 038
Votes	103 189
Media items (total)	503 461
Media items (mean per channel)	247

second and 1200 second marks. This is when the community-creator is creating many new ContentMessages, and hence you would expect an increase in bandwidth usage. The late-joining and late-arriving peers only start participating in the replication process after they join the community (at 1250, and 1500 seconds respectively). This is clearly visible from their upload bandwidth.

After 3000 seconds, the community-creator has stopped creating ContentMessages and all other peers replicated all the data items. The remaining bandwidth usage is the overhead Dispersy generates while trying to replicate two communities. Peers are part of both the CommunityDiscoveryOverlay and the CommunityOverlay and are trying to discover data items to replicate in both. At 3200 seconds, the mean bandwidth usage per community for each peer is 0.56 KiB/s.

To conclude, we can observe that Open2Edit achieves a good load distribution, as no single peer is consuming more than 20 KiB/s. Moreover, we note that Open2Edit has a low bandwidth overhead at the end of the experiment when the load is low.

B. Internet deployment

In order to monitor the deployment of Open2Edit in Tribler we modified one client to, in contrast to normal peers, automatically subscribe to all channels it discovers. This client was started on a server and collected data over a 7.5 month period. During this period, we discovered over 2000 channels and observed that more than 100 000 votes were cast by almost 90 000 peers. The data collected is shown in Table I.

After releasing a new version, many new users install Tribler and use Open2Edit to discover and join channels. In Figure 7 we show the effect of new votes per channel being cast by users after such a successful launch. For clarity, we only show channels which received at least 100 votes during this period. Also, note the logarithmic vertical scale. The most successful channel received more than 7000 votes in this launch window.

After this period, because the new Tribler users have already discovered their favorite channels, voting for new channels is less frequent. Collaboration now moves away from the CommunityDiscoveryOverlay into the specific CommunityOverlay instances.

Figure 8 shows a trace of all activity within a single channel. In this channel, 66 peers collaborated in improving the metadata of 280 media items. Tagging media items is the most popular method for collaboration, which is expected as it requires less effort than commenting on, modifying, or categorizing media items. Initially, we can observe that

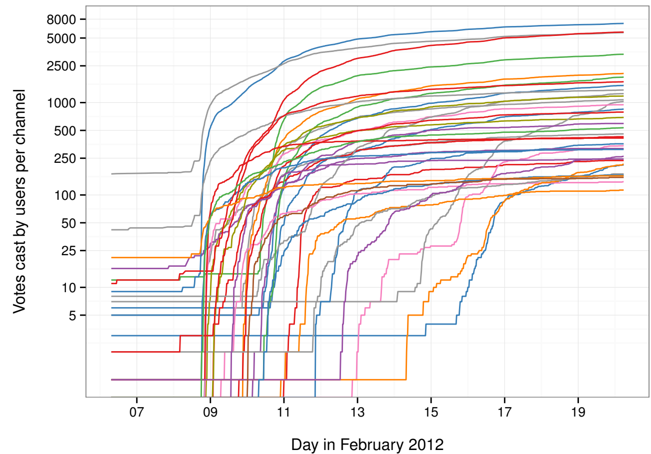

Fig. 7. Votes cast by users for the 37 channels with more than 100 votes in a two-week period around a Tribler release.

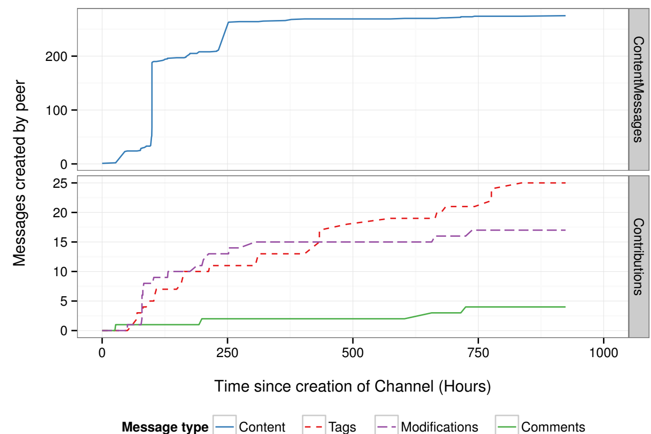

Fig. 8. History of activity in a channel.

one peer is inserting new content into the channel. However, thereafter other peers continue improving the metadata of these content items by tagging and modifying them.

To conclude, we can observe that Open2Edit is able to serve a sizeable group of users for a long time. Furthermore, it keeps functioning under stress, during a successful launch window, when channels are quickly discovered and voted upon by many new users.

VIII. DISCUSSION

During the deployment of Open2Edit in Tribler we have shown that Dispersy can synchronize over 100 000 data items. While this is sufficient for smaller communities, such as those found in Wikia, it is not sufficient if Open2Edit is used to implement a community of a similar size as Wikipedia. In such a community every peer will attempt to synchronize millions or tens-of-millions of data items in order to get all revisions of all articles, their comments etc. We propose two possible solutions which will allow for larger decentralized

communities in the future. The first is based on the concept of *decay*, which means specifying when a data item is no longer relevant and can be removed from a community. In Open2Edit, decay can be used to remove older revisions of an article, thus significantly reducing the number of data items in a community. A revision is only considered to be old if it is both old time wise, and has newer revisions. The second solution is based on *replication management*, which defines a subset of items to be stored at each peer. Replication management is a well researched topic, however most papers only consider a system in which backup is the primary goal and item availability latency is expressed in hours. For our use case, latency should be a few minutes at most, therefore we first need to find a new approach to replication management.

Finally, we want to elaborate on our experience deploying Open2Edit into Tribler. For us, a big lesson learned is that implementing Open2Edit is only half the challenge. The other half is creating a user interface which is easy to use, does not hide the activity in a community, and entices users to start modifying the content within a community. Furthermore, as we were deploying Open2Edit into Tribler we had a difficult time explaining the concept of communities to our users as its not common to be able to create your own community in a file-sharing application.

IX. Conclusion

The main idea behind Open2Edit is combining user contributed resources to create a self-sustaining P2P collaboration platform. We use Dispersy as a sound technical basis for our work and implement communities on top of it. At the same time, we preserve the flexibility of the system, allowing for a wide range of deployments, from wikis to forums and news sites.

We deploy Open2Edit in Tribler to create a media sharing system with YouTube-like functionality. The thousands of Tribler users can now create their own channels or join others' channels to publish, modify, comment on, and categorize media items. The robust permission system allows channel creators to choose their preferred level of openness and designate other users as moderators.

To test Open2Edit we create an emulation environment on the DAS-4 supercomputer using a trace of real-world Tribler usage as input. We observe that Open2Edit distributes the load in the system in a balanced way, which is a key feature for a P2P system. Conversely, when the load is low, Open2Edit has a very low overhead for the peers.

While the emulation experiment proves that Open2Edit functions in a controlled environment, only an Internet deployment can prove that the system also works in an uncontrolled possibly-hostile environment. The data we collected from Tribler unequivocally shows that Open2Edit can serve Internet-scale user communities. Almost 90 000 users actively collaborated in over 2000 channels during the 7.5 months of monitoring.

References

[1] N. Zeilemaker, B. Schoon, and J. Pouwelse, "Dispersy bundle synchronization," TU Delft, Tech. Rep., 2013, [Online] http://www.pds.ewi.tudelft.nl/fileadmin/pds/reports/2013/PDS-2013-002.pdf.

[2] "Tribler." [Online]. Available: http://tribler.org

[3] B. Cohen, "Incentives build robustness in BitTorrent," in *Workshop on Economics of Peer-to-Peer systems*, vol. 6, 2003, pp. 68–72.

[4] L. Mamykina, B. Manoim, M. Mittal, G. Hripcsak, and B. Hartmann, "Design lessons from the fastest Q&A site in the west," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '11. New York, NY, USA: ACM, 2011, pp. 2857–2866. [Online]. Available: http://doi.acm.org/10.1145/1978942.1979366

[5] G. Canals, P. Molli, J. Maire, S. Laurire, E. Pacitti, and M. Tlili, "XWiki Concerto: A P2P Wiki System Supporting Disconnected Work," in *Cooperative Design, Visualization, and Engineering*, ser. Lecture Notes in Computer Science, Y. Luo, Ed. Springer Berlin Heidelberg, 2008, vol. 5220, pp. 98–106. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-88011-0_13

[6] S. Weiss, P. Urso, and P. Molli, "Wooki: A P2P Wiki-Based Collaborative Writing Tool," in *Web Information Systems Engineering WISE 2007*, ser. Lecture Notes in Computer Science, B. Benatallah, F. Casati, D. Georgakopoulos, C. Bartolini, W. Sadiq, and C. Godart, Eds. Springer Berlin Heidelberg, 2007, vol. 4831, pp. 503–512. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-76993-4_42

[7] C. Rahhal, H. Skaf-Molli, and P. Molli, "SWOOKI: A Peer-to-peer Semantic Wiki," INRIA, Rapport de recherche RR-6468, 2008. [Online]. Available: http://hal.inria.fr/inria-00262050

[8] P. Mukherjee, C. Leng, and A. Schurr, "Piki - a peer-to-peer based wiki engine," in *Peer-to-Peer Computing , 2008. P2P '08. Eighth International Conference on*, sept. 2008, pp. 185 –186.

[9] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970. [Online]. Available: http://portal.acm.org/citation.cfm?doid=362686.362692

[10] G. Halkes and J. Pouwelse, "UDP NAT and firewall puncturing in the wild," in *Proceedings of the 10th international IFIP TC 6 conference on Networking - Volume Part II*, ser. NETWORKING'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 1–12. [Online]. Available: http://dl.acm.org/citation.cfm?id=2008826.2008828

[11] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978. [Online]. Available: http://dx.doi.org/10.1145/359545.359563

[12] R. Priedhorsky, J. Chen, S. T. K. Lam, K. Panciera, L. Terveen, and J. Riedl, "Creating, destroying, and restoring value in Wikipedia," *Proceedings of the 2007 international ACM conference on Conference on supporting group work - GROUP '07*, p. 259, 2007. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1316624.1316663

[13] F. Viegas, M. Wattenberg, J. Kriss, and F. Ham, "Talk Before You Type: Coordination in Wikipedia," *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*, pp. 78–78, Jan. 2007. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4076527

[14] B. Kanefsky, N. Barlow, and V. Gulick, "Can Distributed Volunteers Accomplish Massive Data Analysis Tasks?" in *Lunar and Planetary Institute Science Conference Abstracts*, vol. 32, 2001, p. 1272. [Online]. Available: http://adsabs.harvard.edu/abs/2001LPI....32.1272K