Fast Dissemination for CommentCast in Unstructured Peer-to-Peer Networks

Xu Han

Delft University of Technology

Fast Dissemination for CommentCast in Unstructured Peer-to-Peer Networks

Master’s Thesis in Computer Engineering

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

Xu Han

16th January 2011

Author
Xu Han (HughXHan@gmail.com)

Title
Fast Dissemination for CommentCast in Unstructured Peer-to-Peer Networks

MSc presentation
January 26, 2011

Graduation Committee

prof. dr. ir. H. J. Sips (chair)	Delft University of Technology
dr. ir. J. A. Pouwelse (Supervisior)	Delft University of Technology
dr. ir. F. A. Kuipers	Delft University of Technology

Abstract

Commenting is an important fundamental functionality offered with video streaming service. Users exchange commentaries and report problems about the video content through commenting. Tribler is, however, still lacking of the functionality of commenting. CommentCast is a fully distributed commenting system in Tribler based on the BuddyCast protocol stack.

In this thesis, we improve the original design of CommentCast by augmenting different protocols working cooperatively to supply a fast, bandwidth-efficient and reliable commenting service. Then, we elaborate the protocol focusing on fast dissemination. By comparing with gossip algorithm and flooding algorithm, we decide to use LightFlood [14], an algorithm combining pure flooding and spanning-tree broadcasting, to disseminate comments. Our experiments show that LightFlood is a very fast and cost-effective dissemination algorithm.

In order to study the knowledge of user commenting and evaluate our design, we collected the comment history of movie section of Verycd.com, a website providing P2P download resources. Since the movie section of Verycd.com has a similar context as Tribler, we take the collected comment history as a real workload for simulation of CommentCast.

Finally, we investigate the performance of the push protocol of CommentCast by simulating the algorithm working under the real-world workload. Our simulation shows that the CommentCast is able to spread comments rapidly to a large number of users. At the same time, the bandwidth consumption is also realistic based on today's network infrastructure.

Preface

I would like to thank my advisor Dr. Ir. Johan A. Pouwelse for his inspiration and guidance throughout my research. I appreciate his generous and kindness keeping me motivated. I would also like to thank Adele L. Jia and Boxun Zhang for their constructive suggestions and insightful feedbacks.

I would like to give my thanks to my parents and friends for their support and encouragement. I can not finish this project without them.

Last but not least, my thanks also goes to Prof. Dr. ir. Henk J. Sips and Dr. ir. Fernando A. Kuipers for their participation in my thesis committee, and their comments on my work.

Xu Han

Delft, The Netherlands
16th January 2011

Contents

Chapter 1

Introduction

1.1 Peer-to-Peer systems

Before the introduction of Peer-to-Peer(P2P) applications, the client-server(C/S) architecture dominated network applications, in which servers have a critical role in processing or routing etc. The C/S architecture provides very simple and intuitive solutions. Since the boom of the Internet, servers resources, however, become the bottleneck. Since the number of clients increased rapidly, the servers resources could not catch up with it. In contrast to C/S architecture, a P2P architecture seeks to remedy this problem by sharing client resources such as bandwidth, storage space or processing power. In recent years, P2P technology has become very popular for various applications such as file sharing or live streaming applications. P2P systems, or distributed systems usually distribute tasks equally on peers except for some noncore tasks like bootstrapping [32].

The key advantage of P2P systems is their scalability, because they make use of unused resources on client side. Under C/S model, new users deprive a part of resource from the existing users, which causes the system to only provide service to a certain number of users. However, as more and more new users join the system, the capacity of the system increases accordingly under P2P model. Take file sharing as example. Instead of utilizing only the bandwidth of the server, clients also upload parts of the file to other clients while downloading. This is the reason why P2P systems are able to provide services to large number of users compared with C/S model applications.

P2P systems can be classified as structured or unstructured according to the overlay being used [22]. The structured networks usually distribute peers and contents in the overlay according to consistent protocols to ensure that any peer or content are globally traceable, so that any search for content can be routed to the peers who have the content. The most widely-used structured network is Distributed Hash Table (DHT). It maps keys to the nodes of an overlay network and provides means of locating the current peer node responsible for a given key. An unstructured P2P network forms links arbitrarily between peers and peers can only contact directly-connected peers within the overlay. Peers in unstructured networks usually need to use flooding or gossiping to perceive the overlay in order to find desired resources. Structured overlays are convenient in locating desired resources. But peers' joining and leaving are expensive for structured overlay, since they cause a lot of overhead in maintaining the topology of overlay. On other hand, the advantage of an unstructured overlay is the low cost maintenance while not providing guarantees for locating desired resources.

1.2 Dissemination Algorithms in Unstructured Networks

In unstructured networks, overlay links between peers are established arbitrarily. Peers can not locate resources or peers by referring to a uniform function as in DHT based networks. Therefore, locating resources or routing is an important research area in unstructured networks. There are two widely accepted algorithms to do this in unstructured networks.

1.2.1 Epidemic algorithm

Epidemic algorithms refer to network algorithms allowing rapid dissemination or aggregation of information. The name is inspired by the spreading of viruses. It is also known as a gossip algorithm, because it is similar to spreading rumors in social networks where people randomly meet each other and exchange gossips. In epidemic algorithms, peers periodically exchange information with a random neighbor to achieve data dissemination or aggregation. Gossiping is becoming more and more popular in many research areas such as data aggregation and topology management. [30] provide a systematic survey of many of recent results on Gossip algorithms. [12] investigated the gossip algorithm for aggregating date under large dynamic networks. [18] provides a reliable version of gossiping under Byzantine/Altruistic/Rational (BAR model). A series of works on self-organizing topology management were introduced in [11] and [35]. A failure detection protocol is described in [29], which can be used to discover and leverage network topology. [1] introduces a mathematic model of gossip algorithm for arbitrary network graphs, then provides a fast gossip algorithm through theoretical analysis.

Here, we describe two kinds of gossiping, proactive gossiping and passive gossiping. Proactive gossiping, called push, means peers push their information to other peers. Passive gossiping, called pull, means peers extract information from other peers.

Proactive Gossiping

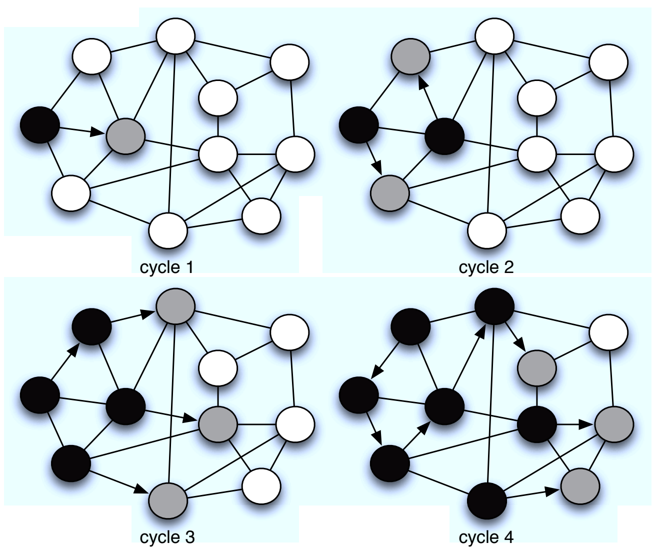


Figure 1.1: A Process of Basic Gossiping Algorithm. It shows 4 cycles of a basic proactive gossiping.

We illustrate a process of basic proactive gossiping where peers begin to send messages to a random neighbor in every cycle after they are infected. Proactive gossiping is usually called the push method. In the push method, the initiator of communication pushes information to the receiver, which means the receiver gets infected after being contacted by an infected peer. The process of a proactive gossiping is shown in Figure1.1. The black circles are the infected peers; white circles are the uninfected peers; gray circles are the peers who are newly infected in a particular cycle. The lines between peers are the links of the overlay and arrows represent initiating a communication.

In proactive gossiping, peers do not know how many peers are infected, that is they do not know when to stop gossiping. The most-common solution is appending a Time-To-Live(TTL) attribute to a message. If the TTL reaches 0, peers stop forwarding the message. The initiator of the message gives an initial TTL, and peers subtract 1 from TTL on every forward until it reaches 0. The initial TTL value takes an important role in overlay coverage. If the TTL is too large, peers keep sending redundant messages time and time again even when everybody has got the comment. On the other hand, if the TTL is not large enough, some peers would never receive the comment.

Passive Gossiping

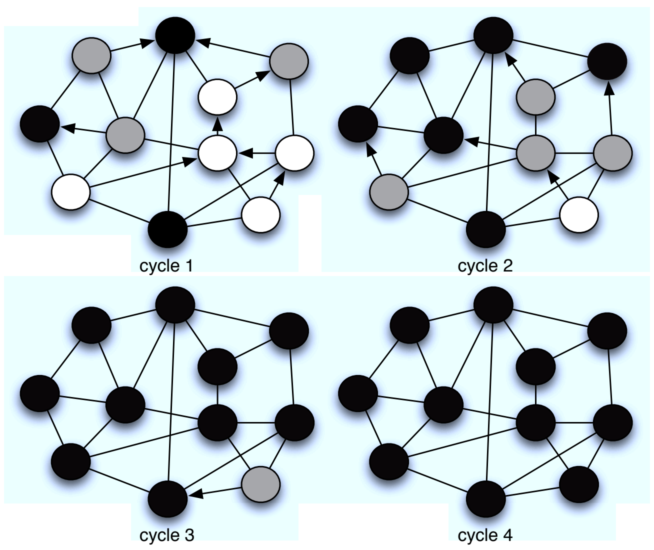


Figure 1.2: A Process of Basic Gossiping Algorithm. It shows 4 cycles of a basic proactive gossiping.

Figure 1.2 shows the process of passive gossiping. In passive gossiping, the initiator extracts information from the receiver, which means the initiator of communication is infected after communicating with an infected peer. At the beginning of the algorithm, there are 3 infected peers. In passive gossiping, the communication is initiated by the uninfected peer to extract information from other peers. The arrows in Figure 1.2 means extraction of information. So uninfected peers will be infected when they extract information from black peers. The gray peers are the infected peers in the current cycle.

At the beginning of passive gossiping, most peers fail in extracting information. Comparing with proactive gossiping, passive gossiping is more effective when there is a small fraction of uninfected peers. In real passive gossiping application, peers can not know when new information is injected. So usually, peers periodically contact a random peer to extract information.

Epidemic algorithms do not guarantee to achieve a particular goal within a given time period, but the most significant advantage is that it guarantees a maximum computational complexity and a bandwidth consumption. Peers will not dedicate too many resources in to one task in heterogeneous P2P network, and power of the peers will not be depleted unexpectedly in Mobile Ad Hoc Network (MANET).

1.2.2 Flooding Algorithm

Epidemic algorithms do not guarantee message arrival. Usually, reliability can be met by increasing the number of peers communicating with in each cycle, called fanout [34]. It also increases the dissemination speed of a message in a network. As the fanout is increased to the maximum value, the algorithm is usually called flooding. It means peers rebroadcast the message to all their neighbors when they receive a new message.

We introduce a flooding algorithm by increasing the fanout of gossiping, but this is actually different from gossiping. In a flooding algorithm, a peer usually forwards a received message to all of his neighbors, it then stops forwarding. In gossiping, peers keep forwarding a received message in every cycle until the TTL of the message drops to 0. Because of forwarding to all neighbors, a flooding algorithm guarantees arrival of a message. Gossiping, however, does not. Besides arrival of messages, flooding also guarantees shortest arrival time. Reliability and high speed are therefor the advantages of flooding algorithms. Flooding algorithms, however, lead to significant amount of retransmissions, which is the primary disadvantage. Figure 1.3 shows the process of a flooding algorithm. Compared with the basic

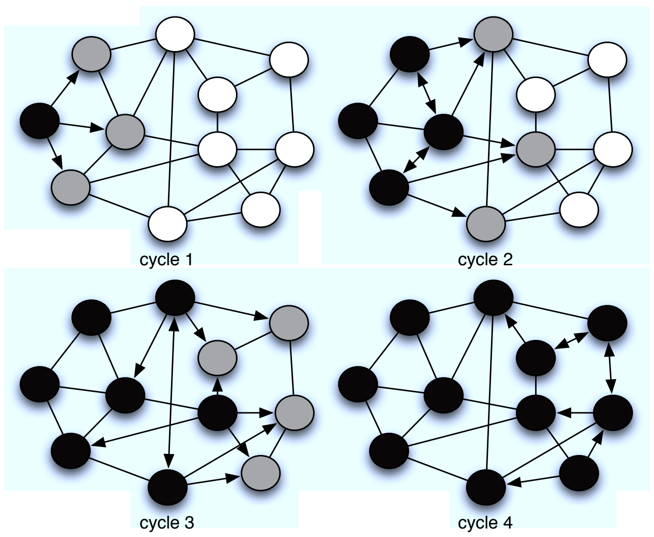


Figure 1.3: The Flooding Process. It shows 4 cycles of a flooding algorithm.The first difference between flooding and gossiping is that the fanout increases to the maximum value. The second one is that only gray peers being infected in the current cycle forward the message in the next cycle after being infected.

epidemic algorithm in Figure 1.1, flooding covers every peer in the overlay within 4 cycles. On other hand, flooding generates many more retransmissions than of the previous proactive gossiping.

1.3 Tribler

Tribler is a fully decentralized P2P file sharing system built on top of the BitTorrent protocol. By maintaining an unstructured social overlay based on user-taste, Tribler provides content discovery, content recommendation and downloading [26]. BitTorrent [5] is one of the most popular protocol for file downloading. It divides a large file into small pieces, creates hashes for each piece and encapsulates the results and tracker information into a torrent file. The tracker is responsible for keeping track of a list of peers who are downloading the file, so that the peers can contact them and exchange different pieces of the file. The BitTorrent protocol only focuses on transferring files and leaves file searching to other components, like Web sites. Tribler realize the fact that traditional P2P file-sharing systems neglect the power of the social phenomena and wants to exploit the effect of the social phenomena in content discovery, recommendation and file sharing based on BitTorrent protocol.

Tribler group has developed a series of protocols for taste-based overlay formation, distributed reputation management, collaborative downloading, video streaming etc. BuddyCast has been developed into a substrate of a complete epidemic protocol stack [27]. It selects peers to synchronize with and the higher layer protocols, like Barter Cast, SwarmCast, Friend Cast and CommentCast, do the synchronization of MegaCaches for different purposes.

1.4 CommentCast

The prior work we extend in this thesis is CommentCast, a fully distributed commenting system. It works without any central component. Commenting system has been developed over many years under C/S model. The best-known commenting system could be the bulletin board system (BBS), or Internet forum. BBS is already well studied and the algorithms for accessing database are mature. However, fully distributed commenting system is still under development. Distributed systems are designed for large scale applications, however algorithms for disseminating are usually not scalable enough for large networks. Data consistency is another problem. There are some algorithms focusing on the consensus issue, like Paxos algorithm [17]. Similarly, these algorithms are also not designed for large scale network. Last but not least, BBS is a kind of timely service. Users expect to receive updates as soon as possible while the distributed environment can not guarantee end-to-end relay. Here we would like to introduce a solution for a fully distributed commenting system called CommentCast which is protocol of BuddyCast protocol stack.

1.4.1 The Idea and Data Structure of CommentCast

The basic idea of CommentCast is similar to the other protocols, like BarterCast [21], of the BuddyCast protocol stack. In CommentCast, every peer of the system keeps a local database of comments. These comments are in the form of comment message stored in the local database. Peers collect as many comment messages as possible by exchanging comments with other peers. When a user is browsing a Channel, all corresponding comment messages are acquired from local database and then arranged in the Channel page according to their time stamps. A comment message includes Channel ID, Commenter ID, Commenter Nickname, Timestamp, Comment txt, signature. The detailed information are in Table 1.1.

Field	**Size**	**Description**
Channel ID	20 Bytes	ID of the channel
Commenter ID	20 Bytes	The PermID of the comment publisher
Nickname of Commenter	Max. 30 Bytes	The nickname of the commenter
Timestamp	4 Bytes	The time when the comment was post
Comment txt	Max. 280 Bytes	The content of the comment
Signature	67 Bytes	For security issue, generate from above four fields

Table 1.1: Comment message table entry structure in local cache

1.4.2 Specifications of CommentCast

CommentCast exchanges comment messages by sending CommentCast messages which are collections of comment messages. The detailed algorithm works as follow:

- Each CommentCast message contains 30 comment messages that are 10 my_recent_comments, 5 my_random_comments, 10 other_recent_comments and 5 other_random_comments.

- Peers periodically send out a CommentCast message to another peer. When and who to send depends on the period of BuddyCast.

- Upon receiving a CommentCast message, the recipient updates its local database with the new comments in the message.

The my_recent_comment are the comments that were posted by the peer itself most recently. My_random_comment are the comments picked randomly from my_comments. Likewise, other_recent_comments and other_random_comments are recent and randomly picked comments of other peers in the local database. The purpose recent comments is to spread the recent comments out with a high probability while the random comments will still cause eventual consistency.

Chapter 2

Problem Description

The most significant achievement of original CommentCast is that it supplies a possible solution for the fully distributed commenting system and the epidemic nature makes it very scalable in a large-scale network. But, the disadvantages of CommentCast are obvious as a commenting system. We can not ignore them.

2.1 Disadvantages of CommentCast

2.1.1 Low Dissemination Speed

The concept of CommentCast relies on the fact that every peer possesses a copy of all comments in the system, which means every single comment should be disseminated to all peers. CommentCast is not flexible. It exchanges data periodically and uses the period of BuddyCast, which is 4 hours per cycle. Our later experiment in section 5.2 of this thesis shows that such an epidemic algorithm takes more than 10 cycles to spread a comment to 500 peers. A single spreading cycle takes up to 4 hours. Spreading of a single message takes more than 40 hours. What we expect is that the new comments are spread to most peers within a shorter time, even enabling real-time communication through commenting.

2.1.2 Low Coverage Ratio

CommentCast can be classified as a technique of distributed-database synchronization schema. However, it is not a remarkable way of doing so. From the design we can see that recent comments have a higher probability being spread broadly while spreading of history comments is a random-based epidemic algorithm, which means the speed of spreading of a new comment decreases largely once it is classified as a history comment.

Multi-hop gossiping could not be achieved under such random-based epidemic algorithm. The statistics from [6] show that most peers, over 90%, only possess a fraction, less than 20%, of the records in BarterCast which uses a random-based epidemic algorithm focused on reputation management. Although we do not have similar statistics from CommentCast, we can expect that CommentCast will not achieve a much higher performance than this, because the design of CommentCast is similar to BarterCast.

2.1.3 Low Bandwidth Efficiency

A solution of distributed system should not use a large amount of bandwidth. Sending duplicated comments is a waste of bandwidth. Things get much worse when most peers are sending duplicated comments. The randomness of comment sending also has an impact on bandwidth usage. Based on the low data coverage, it is highly likely that most comments in CommentCast messages are duplicated comments for the receivers. Besides random picking, CommentCast also does a blind communication. Peers could send the same comments to the same recipient time and time again. We what a fast system with high communication efficiency.

2.1.4 Inefficient Usage of Local Hard-Disk

It is also debatable whether or not to store all comments from every channel in local database. Assuming peers have already possessed all comments, the original design definitely increases the speed of retrieving comments. Local hard-disk-usage becomes a negligible issue. Our experiences tell us that most users only have interest in a subset of channels and that subset may be only a very small portion of entire set of channels. Saving all comments seems to be an controversial solution.

2.2 Research questions

The problems of CommentCast have been analyzed. This thesis can only focus on a subset of those problems. This section will present a guide-line of the thesis.

Improve Speed of Dissemination

CommentCast uses a synchronization schema for data dissemination. The main advantage is that this approach is very reliable. On the other hand, the low spreading speed is not acceptable for a commenting system. Based on our experience, people sometimes chat on BBS nearly in real-time like using instant messengers. CommentCast can not provide service under such a constrain. As described in [7], the round trip delay is the index of primary interest in information retrieval requests and database queries. Even though [7] is based on traditional C/S model, this can also applied to distributed commenting system like CommentCast. The primary goal of this thesis is to improve the speed of dissemination.

Reduce the Cost of Dissemination

Considering data dissemination of a single comment, CommentCast is the same as the basic proactive epidemic algorithm presented in the first chapter. We've talked about the idea of improving the reliability and disseminating speed by increasing the gossiping fanout. But, as the fanout reaches the maximum value, the number of retransmissions also becomes very large. If spreading every comment is like a flooding process, the whole system will suffer from a extremely large amount of wasted bandwidth. Further more, CommentCast is not a main functionality of Tribler and it is definitely not acceptable to dedicate such a high cost in CommentCast. In order to provide a realistic design, reducing the cost of dissemination is significant as well. The second goal of thesis is reduce the cost while achieving high speed of dissemination.

Withstand the High Churn Rate

Churn is inevitable in real world systems. Distributed systems are not able to handle churn as easy as in client/server design. Furthermore, distributed systems are supposed to supply service to a much larger number of users, which means churn becomes even more serious. In order to design an applicable system which can be deployed in Tribler, churn should be taken into account and CommentCast should be able to handle the challenge of a distributed commenting system under a high churn rate.

Derive Real World Parameters for Design

The previous design of CommentCast only supplies a design idea of a fully distributed commenting system. It does not take real world parameters into account. Developing an applicable design should consider the real world workload and constrains. However, previous researches and measurements of commenting systems are not sufficient enough. It is necessary to collect relevant data from an exiting system.

Chapter 3

Structural Improvement and Possible Solutions for the New Structure

This chapter investigates possible solutions for improving CommentCast. First section of this chapter illustrates CommentCast from an architecture view and improves the design by enriching utilized protocols. Then, it elaborates every protocol in detail and provides possible solutions.

3.1 Architecture of CommentCast

Previous work presents the concept of a distributed commenting system and the working principle of CommentCast, but it doesn't provide detailed information needed for an implementation. Some details are still ambiguous. Here we would like to elaborate the architecture of the CommentCast.

3.1.1 Architectural Design for CommentCast

We derived an architectural design of CommentCast from previous work. CommentCast is responsible for retrieving new comments from the Tribler network and provides the corresponding comments when a user is browsing a channel. CommentCast does not interact with the user directly, so it works under control of the channel module in Tribler. Figure 3.1 shows the architecture of CommentCast. Comment messages are stored in a local database called MegaCache. There is a comment management unit (CMU) that works as a manager of the database, operating directly on the MegaCache. All comment operations including creating, retrieving and updating go through the CMU. The channel module works cooperatively between user and CMU. When a user is retrieving channel data, the channel module also sends a query with the channel ID to CMU besides retrieving channel information. CMU fetches comment messages for the channel module and replies

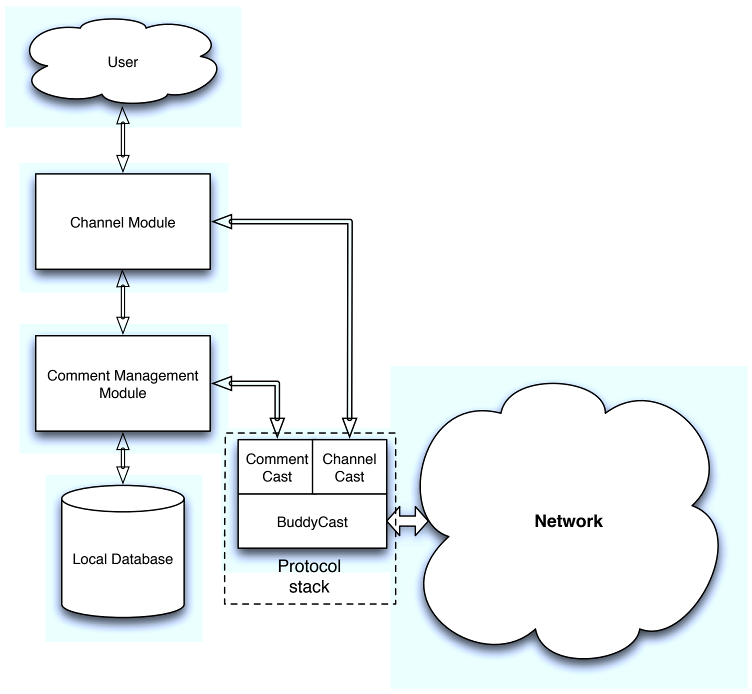


Figure 3.1: Architecture of CommentCast

with these comments messages. The channel module arranges the comments into a user-friendly format and puts them into a channel page which is then returned to the user. A user is also able to create a comment through the channel module. The channel module passes the comment text to CMU. CMU then creates a comment message containing the comment text, PermID, timestamp, etc. and stores them in local database. As a security measure, CMU also verifies if the comments are valid or not by checking peer signature using a Public Key Infrastructure (PKI).

The protocol stack consists of protocol(s) exchanging comments with other peers from the network. It only has one protocol in the original design, which is the epidemic exchanging protocol. We call it the synchronization protocol. The protocol stack does not perform any validation check or comment management and is only responsible for sending and receiving comments. When protocol stack receives a CommentCast message as introduced in section 1.4.2, it extracts comment messages and passes them to the CMU.

In the design, the CMU works as an agent with the local database. This is necessary because the distributed commenting system is still not perfect. The functionalities of a traditional commenting system can not be achieved completely under a distributed design. CMU could perform some complicated management instead of simply depositing and retrieving. For example, there are some inappropriate comments posted deliberately, like spams and curse words. Under CommentCast, those comments can not be removed even when the user doesn't want to see them again, because the protocol automatically retrieves absent comments from the network even if the user removed them. That is to say, the consistency of data is not maintainable under a distributed system, or formally called Byzantine generals problem. CommentCast translates the consensus problem of a thread into many consensus problems of individual comments, but those problems persist. There are no mature algorithms solving Byzantine generals problem which are scalable enough for large scale networks. So our design is that the CMU maintains a management field for each comment indicating if the comment is removed or not. This means the remove function does not really remove the comment, it hides the comment from user.

3.1.2 Architectural Improvement

The core of CommentCast is the protocol. It is a subprotocol based on the BuddyCast protocol stack. However, CommentCast is a comprehensive protocol. It could be divided into two subtasks. Including most recent comments in a CommentCast message can be considered push-based gossiping for disseminating new comments. The entire algorithm of CommentCast seems like a database synchronizing protocol. We would like to separate the CommentCast protocol into multiple protocols. The improved protocols are shown in Figure 3.2 which consists of a push protocol, a synchronization protocol and a search protocol. The protocol stack in Figure 3.2 does not show the other protocols of the BuddyCast protocol stack, since the structure is not the major concern here.

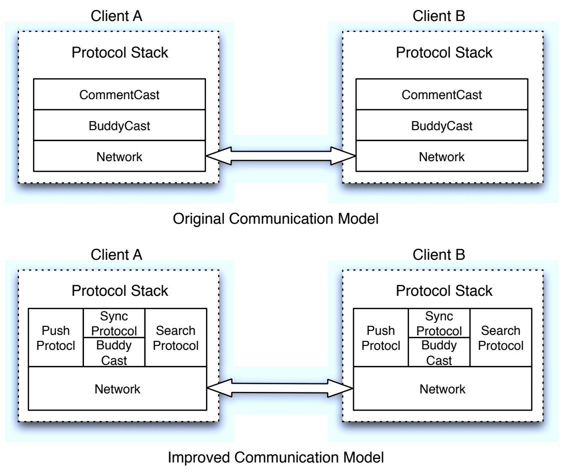


Figure 3.2: The improved design of protocol stack

The push protocol is supposed to achieve high speed of dissemination. Peers are trying to work cooperatively in spreading newly-injected comments as fast as possible, so that most channel browsers are able to see the new comments of a channel as soon as possible. It directly communicates with other peers through network layer.

The synchronization protocol tries to synchronize local databases of different peers to achieve high data coverage. The synchronization protocol is designed for three purposes. Firstly, it is used to keep consistency for online peers. Since the push protocol can not guarantee full coverage, peers periodically trigger the sync protocol to retrieve missed comments of push protocol, but the sync granularity is supposed to be sufficiently large such that it will not consume too much bandwidth. Secondly, it is used for churn repair. It is triggered when a regular user starts a Tribler client for pulling new comments during its offline time. Last but not least, bootstrapping also makes use of the the sync protocol. New Tribler users use it to initiate their local database efficiently. The sync protocol works by using the BuddyCast sublayer, instead of directly communicating with other peers.

The search protocol is based on a new concept that every peer only keeps the comments that are most likely to be read by a particular user. When a user is browsing the comments kept in his local database, it works as the same as the old idea. When user is browsing a channel without any comments in local database, CommentCast searches comments from other peers using the search protocol.

3.2 Possible Solutions for Protocols

The push protocol is the core of this thesis. Although the protocols work cooperatively in the distributed commenting system, the other protocols are out of the scope of this thesis. Therefore, we will elaborate possible algorithms for the push protocol first.

3.2.1 Push Protocol

LightFlood

The work of [14] has shown that roughly 70% messages are redundant in flooding with a TTL of 7 on Gnutella overlays. In order to reduce redundant messages generated by flooding, [14] introduces a fundamental cost-effective flooding operation in an unstructured P2P network. The fundamental idea of LightFlood is to construct an additional spanning tree on the overlay network and divide the dissemination process into two phases: first, do several hops of pure flooding; and then, the message will be only broadcasted using the spanning tree.

The flooding is a very reliable disseminating algorithm. It does not only guarantee the arrival of a message, it also guarantees the shortest end to end latency. But the number of redundant transmissions, or retransmissions, make it too expensive for a dissemination process. Therefore, flooding is used for topology discovering.

On the other hand, a spanning tree is a structure that minimizes the number of retransmission in dissemination. Peers have a parent peer and children peers in a spanning tree. They only forward a message to their parent and children peers. It means the dissemination in a spanning tree does not lead to any retransmission. However, the speed and the reliability are quite poor. Any leaving of peer and link failure can splits the tree into two parts, which decreases the coverage of dissemination. [14] shows that pure flooding takes 7 hops to cover 95% of peers in Gnutella's overlay while broadcasting on spanning tree takes more than 30 hops to cover the same amount of peers in the same overlay. Based on the analysis above, flooding and spanning trees are complements for each other. Therefore, LightFlood makes use of the high speed, reliability of pure flooding and the low redundancy of spanning tree broadcasting.

TTL is an important parameter in flooding. It directly affects the overlay coverage and redundancy of the flooding algorithm. [14] also shows that the redundancy is relatively low in first 4 hops while it increases dramatically in last 3 hops in order to cover 95% peers in the Gnutella's overlay network. So, in LightFlood, the first phase is M hops of pure flooding; the second phase is N hops of broadcasting on spanning tree. It's called (M, N) scheme. [14] also investigates the performance of LightFlood under different values of M and N. But the actual performance could be very different depending on overlay, because the redundancy and overlay coverage largely depend on the overlay structure.

Multipoint Relays

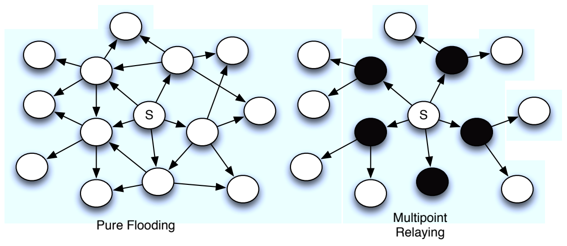


Figure 3.3: Broadcasting by pure flooding and broadcasting by multipoint relays

Multipoint relaying is a concept that reduces the number of duplicated retransmissions while forwarding a broadcast message [28]. This technique restricts the number of relaying peers forwarding a broadcast messages to a subset of its neighbor peers instead of all of its neighbors, like in pure flooding. This set is kept as small as possible by efficiently selecting neighbors which cover the same network overlay as the the complete set neighbors does. The small subset of neighbors is called multipoint relays of a given network. Figure 3.3 shows the concept of multipoint relay. Left side of the figure is a pure flooding and the right side is a multipoint relays broadcasting. The pure flooding usually generated a lot of retransmissions. In the example, the pure flooding leads to 10 retransmissions, while the multipoint relays technique avoids the retransmission.

An important aspect in multipoint relaying is the manner in which these relays are selected by each peer. Well-selected multipoint relays could eliminate retransmission completely, but it is also possible that they do not bring any improvement. [28] provides a heuristic to select the multipoint relays and evaluates the performance by theoretical analyses and simulations.

Although the multipoint relay performs well, it takes a lot of signaling messages to create the multipoint relay. Under a system with high churn rate, like Tribler, it may consume too much bandwidth for structure maintenance.

Catalogue-Gossip
[24] introduces a flash data dissemination in unstructured P2P networks based on a gossip algorithm, called Catalogue-Gossip. The objective of the Catalogue-Gossip is to distribute content of arbitrary format and size to all peers which are part of the network. Catalogue-Gossip relies on an underlying Membership Protocol [8] [13] for building up a consistent view of neighbors at every peer.

It divides the content into multiple chunks. Every peer maintains two structures for the Catalogue-Gossip. The first structure is a table containing all chunks that have been downloaded by a peer thus far. The table helps the node ascertain which chunks are still missing. To this end, the node can fetch missing parts from other peers. The second structure is a set of frequency counters whose objective is to offer an estimation on how frequent each specific chunk is in the entire system. Peers retrieve missing chunks in the first table based on Rarest-First or Random-First using the second table. Peers are also responsible for assembling the chunks into content after download is finished.

The advantage of Category-Gossip is that it is able to spread content with an arbitrary format and size. However, CommentCast is designed for spreading comments only. Furthermore, comments are usually very small in size. Dividing them into chunks is not necessary and leads to extra costs.

3.2.2 Synchronization Protocol

Chapter 1 has already presented the efficiency problem of proactive and passive gossiping in 1.2.1. Proactive gossiping is more efficient in terms of redundancy when most peers are not infected. Pushing the proactive gossiping to cover the entire network leads to a large amount of retransmissions, which is not efficient because of redundancy. Therefore, it seems to be reasonable that the push protocol covers the most peers in the network and the other uninfected peers passively pull the comments to reach the entire network, because passive gossiping is efficient in covering the last several percent of peers. In passive gossiping, the probability of a peer being infected increases as the overlay coverage increases. The synchronize protocol is out of the scope of this thesis, but we would only provide tentative suggestion for the protocol.

Splash, in[19], is a good solution for the synchronization protocol. It is a passive gossiping algorithm for database synchronization. It uses Bloom filters to reduce the bandwidth cost. [19] shows how Splash dramatically increases the average data coverage of peers to 95% coverage, which is 5 times more than BarterCast. Achieving the same data coverage as BarterCast only takes less than 1 percent of the traffic. CommentCast is quite similar to the BarterCast. Although our data shows the workload of CommentCast is less than BarterCast's, which leads to an increase in the performance, we believe that using Splash for synchronizing comments would provide a great performance.

The synchronization protocol is designed for 3 situations. The first one is for the users who stay online. The push protocol is responsible for spreading new comments to all online peers. However, for the sake of redundancy, the push protocol can hardly achieve 100% overlay coverage while churn is inevitable in real world systems, thus impacting the overlay coverage again. Therefor online peers are occasionally in the need of synchronizing their databases with each other as well.

The second situation is when regular users come online. Usually, nobody keeps their client open all the time even for loyal users. The comment database should be updated as soon as possible when a user logs in. When and for how long a user logs off is quite unpredictable. The synchronization algorithm should be flexible.

Last but not least, new users should also be considered. New users may come at any time, and attracting new users is very important to a software. The protocol should, therefor, be able to collect all history comments in a short time as well.

However, the false positive issue of Bloom filters could be a problem, because [19] shows that Bloom filters could waste a lot of traffic for synchronizing last 5% data. Furthermore, Bloom filters are computational-intensive scheme. They save communication cost, but consume local computational resources. Therefore, handling the Bloom filters is critical in designing the synchronization protocol.

3.2.3 Searching Protocol

The searching protocol is based on the idea that peers only keep a subset of all comments in the network. Most users usually don't browse the channels that they have no interest in. So keeping comments for all channels does not seem to be an efficient method, considering disk usage. Tribler already has a taste-based overlay in which peers are clustered according to their personal interest. We suggest that user to selectively keep comments according to their cluster. It's not like saving the browsing history, but more like predicting the channels that a user may browse in the future. Once, the user selects a channel for which can not comments can not be found, CommentCast triggers a searching process to get the corresponding comments. The searching protocol is a preliminary functionality of the idea. The idea doesn't only increase the disk usage efficiency, it also shrinks the range needed to be covered by the push protocol. From the results of our simulations, shrinking the overlay coverage would greatly increase the performance of push protocol.

Searching algorithm also uses flooding or gossiping in unstructured networks. The problem is how to get the results with a cost-effective manner. The simulations in chapter 5 are transferable. How to handle the information of user taste and form an appropriate semantic overlay can be a more important research problem. The entire design tries to balance performance and cost among different protocols. The search protocol is not a part of the scope of this thesis.

Chapter 4

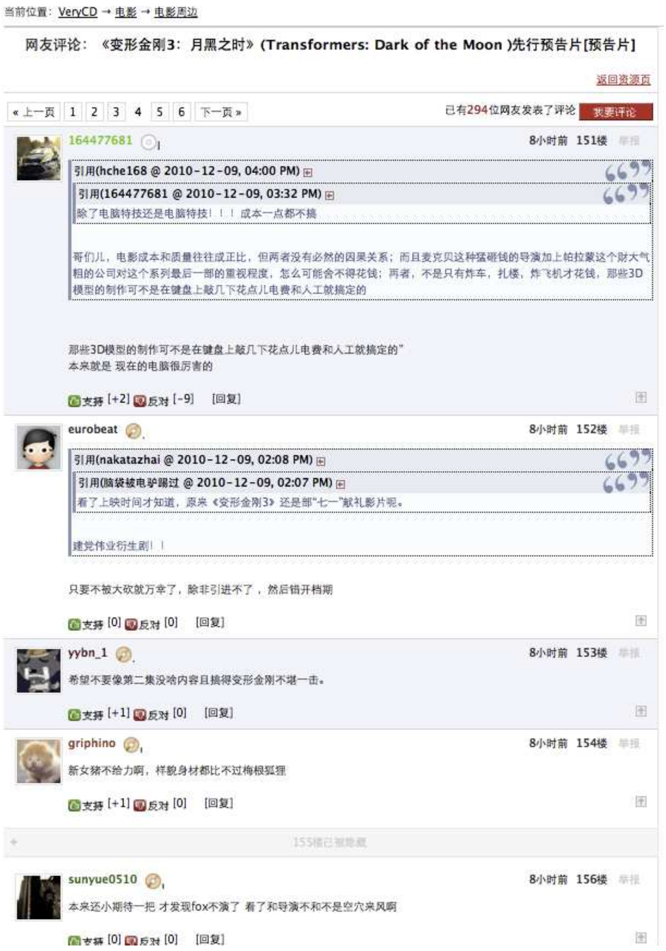
Data Collection and Analysis

The original design of CommentCast did not take real-world system workload and actual user behavior into account which are important in designing a system. Abundant measurement studies about social networks and P2P systems are available. [9] and [36] analyze system traffic and user behavior in video-on-demand systems. [3] focuses on social networks various network systems. There are also some important measurements in [25] [10] and [15] providing data of P2P systems. However, previous studies do not conform our context. We would like to collect some data from an existing system. The target system is preferred to have a similar context with Tribler, which is, firstly, providing a downloading or video streaming service; secondly, users are able to create threads, publish resources in the threads, or leave comments with regard to the contents in a thread; last but not least, the system is supposed to provide service for a large mount of users.

After considering various choices such as YouTube and EZTV, we have decided to focus on Verycd, http://www.verycd.com, a Chinese Web system providing resources for the eMule protocol [16]. Verycd conforms to our context, although it uses the eMule protocol instead of BitTorrent. The thread of Verycd is a good analogy to the channel in Tribler. Users can create threads and post multiple resources in one thread.

It also has an ideal number of users and comments for investigations. Our measurement show that 186,000 users have participated in the discussions of the movie section. According to its naming pattern, we estimate there are at least over a million registered users of Verycd. Furthermore, Verycd allows anonymous downloading, thus the actual the number of user is larger than the number of registered users. Although we can not know the actual number of users precisely, it is reasonable to believable that Verycd has enough users to be considered as a large scale network.

Additionally,Verycd is a commercialized Web system which existed for the last 6 years so the comments are well organized and free from advertisements. The quality of the data is quite good. This ensures the data won't have any bias due to spam or abnormal user behavior. We dont have to worry about availability and scalability of its server. Our measure won't be interrupted by any exceptions. Figure 4.1 shows a segment of comments.



当前位置：VeryCD → 电影 → 电影周边

网友评论：《变形金刚3：月黑之时》(Transformers: Dark of the Moon)先行预告片[预告片]

返回资源页

« 上一页 1 2 3 4 5 6 下一页 »　　已有294位网友发表了评论 我要评论

164477681　　8小时前 151楼 举报

引用(hche168 @ 2010-12-09, 04:00 PM)

引用(164477681 @ 2010-12-09, 03:32 PM)

除了电脑特技还是电脑特技！！！成本一点都不搞

哥们儿，电影成本和质量往往成正比，但两者没有必然的因果关系；而且麦克贝这种猛砸钱的导演加上帕拉蒙这个财大气粗的公司对这个系列最后一部的重视程度，怎么可能舍不得花钱；再者，不是只有炸车，扎楼，炸飞机才花钱，那些3D模型的制作可不是在键盘上敲几下花点儿电费和人工就搞定的

那些3D模型的制作可不是在键盘上敲几下花点儿电费和人工就搞定的"
本来就是 现在的电脑很厉害的

支持 [+2] 反对 [-9]　[回复]

eurobeat　　8小时前 152楼 举报

引用(nakatazhai @ 2010-12-09, 02:08 PM)

引用(脑袋被电驴踢过 @ 2010-12-09, 02:07 PM)

看了上映时间才知道，原来《变形金刚3》还是部"七一"献礼影片呢。

建党伟业衍生剧！！

只要不被大砍就万幸了，除非引进不了，然后错开档期

支持 [0] 反对 [0]　[回复]

yybn_1　　8小时前 153楼 举报

希望不要像第二集没啥内容且搞得变形金刚不堪一击。

支持 [+1] 反对 [0]　[回复]

griphino　　8小时前 154楼 举报

新女猪不给力啊，样貌身材都比不过梅根狐狸

支持 [+1] 反对 [0]　[回复]

155楼已被隐藏

sunyue0510　　8小时前 156楼 举报

本来还小期待一把 才发现fox不演了 看了和导演不和不是空穴来风啊

支持 [0] 反对 [0]　[回复]

Figure 4.1: A page of comments of a thread on Verycd.

This chapter will first illustrate how the crawler was deployed and gives a summary of the collected data in section 4.1. In section 4.2, we analyze the workload from Verycd, including the arrival rate of comments and threads, size of comments and number of comments per thread. Then we analyze the number of users and user behavior in the commenting system.

4.1 Overview of Data collection

Until 11.6.2010, Verycd has 9 categories of resources containing over 186 ,000 threads. Since channels in Tribler mainly offer movies to users, we decided to crawl all threads from the movie section. Different categories may lead to different user behavior in commenting. We suppose the comments from movie section are the most similar to the comments in our channel. Verycd supplies timestamps of threads and comments, which are critical to our measure. Thanks to this feature, we have access to entire comment history of a channel with a one-time visit.

Verycd has an archive list containing all resource which simplifies our crawling. In a normal run, the crawler started from the page of archive list with movies: http://www.verycd.com/archives/movie/. The crawler starts from the end of archive list and downloads all data until the first thread of movie section, which means the newest threads are downloaded first.It starts at the initial page, scan all links for archive pages and saves all page links in memory. Then the crawler looks for threads in every archive page. Usually, there are 100 threads in each archive page. It make a list storing those thread links. For every link, it downloads the thread page, parses the page and writes relevant information to a file. Since the comments of Verycd are dynamically loaded through an Ajax application, we need to go to every comment page to find comments. The flowchart of the crawler activity is as Figure 4.2.

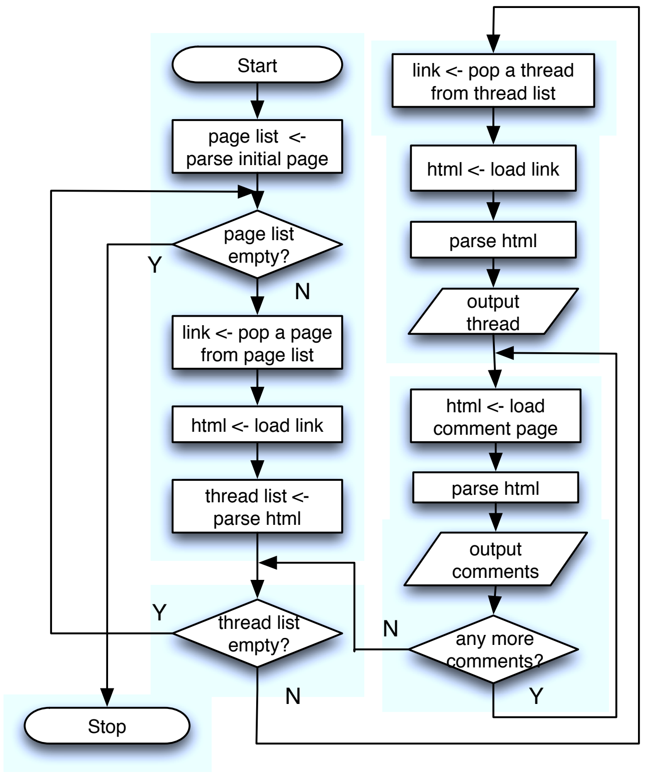

Start
page list <- parse initial page
page list empty?
Y
N
link <- pop a page from page list
html <- load link
thread list <- parse html
Y
thread list empty?
N
Stop
link <- pop a thread from thread list
html <- load link
parse html
output thread
html <- load comment page
parse html
output comments
N
any more comments?
Y

Figure 4.2: The flowchart of the Verycd crawler activity.

The crawler was started on June 6, 2010 abd downloaded information from over

28,000 of threads and 1 million comments, making over 60,000 of HTTP requests. It finished on July 13, 2010. The actual number of threads and comments are shown in Table 4.1. We collected the entire information of movie section of the Verycd which includes 28,259 threads and 1,189,669 comments, form 2003 until 2010. In a later analysis, we found that the first comment was injected on Oct. 6, 2006. As far as we know, Verycd did not provide a commenting service in its early stage. Thus the over a million comments were injected between Oct. 2006 and Jun. 2010.

Some threads and comments were not parsed due to incomplete or false data, but we successfully parsed over 95 percent of the downloaded data. We believe that these results are representative for the user behavior of Verycd.

	Thread	**Comment**
Download	28,589	1,189,669
Parsed	28,506	1,137,619

Table 4.1: Total number of threads and comments. Downloaded threads and comments are the numbers downloaded from server. However, not all threads and comments could not be parsed due to data errors. Last row is the total number of threads and comments that could be parsed.

4.2 System Workload of Commenting System

The system workload is critical to our measurement. The workload is an important factor, which can greatly affect decisions when designing a system.

4.2.1 Arrival Rate of Comments and Threads

First, we present how many comments per day are injected into the movie section of Verycd. Since Verycd did not provide a comment system several years ago, so we take the time of the first comment, Oct. 6, 2006, as the creation time of the comment system. Figure 4.3 shows the daily comment arrival rate from Oct. 6, 2006 until Jun. 6, 2010. Points in the figure represent the numbers of comment injected in a day.

It is clear from Figure 4.3 that comment arrival rate has an upward trend during the its entire life of the system. There were not so many people posting comments at the early stage. There were even several months when no comments were injected. However, the comment arrival rate increased during times. The first one began at the end of 2007 and ended in February of 2008 and the second one happened during the first 4 months of 2009. After an increase, comment arrival rate remains stable at the same level for a long time. During our collection, it fluctuates around 2000 comments per day. It is reasonable to believe the system load remains stable for a period.

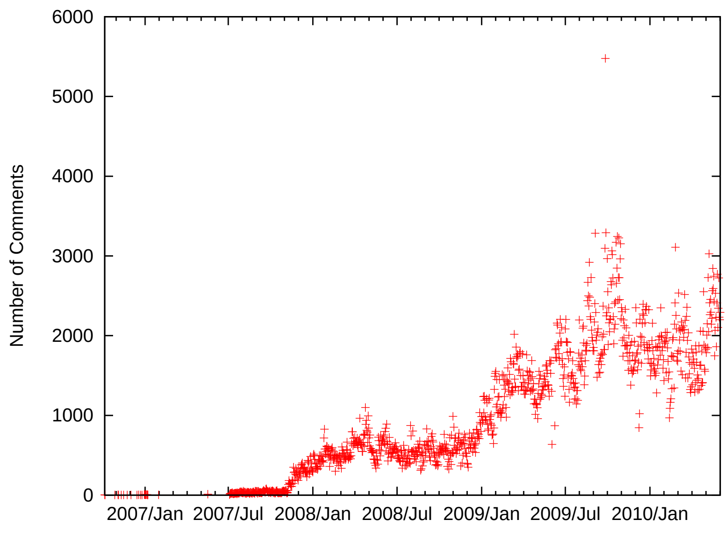


Figure 4.3: Daily comment arrival rate in Verycd. The figure shows the number of comments injected every day between Oct. 6, 2006 and Jun. 6, 2010. The horizontal-axis is dates and vertical-axis corresponds to the number of comments created in that day

The thread arrival rate is also a factor of system load, because comments rely on threads. Users do not regularly leave comments. They only leave comments for the contents of their own interest or participate in a controversial discussions in a thread. The more threads are created in the system, the more new comments will be injected. Comparing to the comment arrival rate, thread arrival rate increases more gradually and seems to be more irregular and hard to perceive. We present the monthly arrival rate of threads in Figure 4.4. Every point means the number of threads created within that month. Thread creation is a basic functionality of Verycd. It exists from the beginning of the system. Therefore, thread arrival rate starts from Sep. of 2003 while the comments arrival rate starts from Oct. 2006, which means the scale of the horizontal axis is different.

The thread arrival rate shows a different property from the comment arrival rate. It increases gradually. At the end of our data, it reaches around 650 threads per month. So there are a little bit more than 20 threads injected per day. The figure shows the thread arrival rate bottom at 0 in Jun. of 2004. It is highly likely that the system was down for more than a month. As it is beyond our scope, we will not try to find exact reason behind this.

4.2.2 Size of Comments

CommentCast keeps all comments in the local database, which means the the size of every comment directly affects how much local disk space CommentCast uses. It is important to estimate whether or not the CommentCast uses realistic amount of

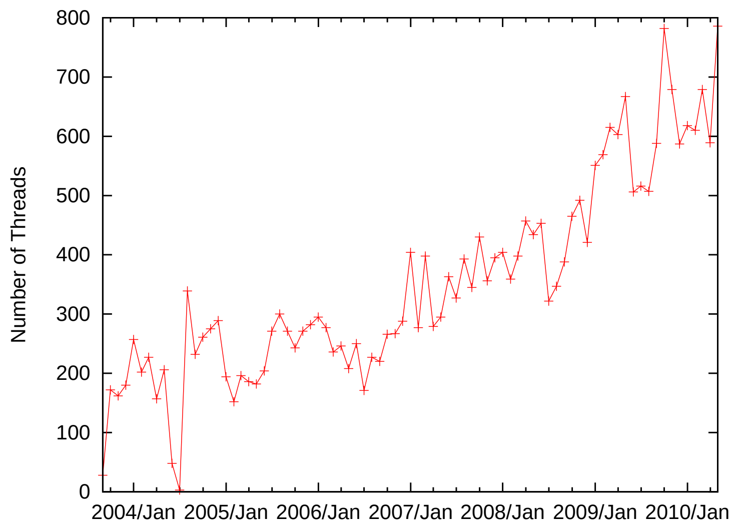


Figure 4.4: Monthly thread arrival rate in Verycd. The figure shows the number of threads injected every month Sep. 2006 and Jun. 2010. Horizontal-axis is months and vertical-axis is the corresponding number of threads created per month.

local disk space. In the design of CommentCast, size of comment text was limited to 280 Bytes. It is also important to know how many comments may exceed the the limitation, such that we can adjust our design.

There are two disadvantages of using our data in estimating the comment size. For one thing, comment size in Verycd could be very different form comment size on Tribler, since different language structures lead to different size of comments. For another, the size of a comment on Verycd is not very precise because Verycd uses UTF-8 in coding. UTF-8 is a variable-width encoding. Here we would like to take every character as 2 Bytes in estimating. Table 4.2 shows the total size of comments in terms of Bytes and lines with statistic values.

A comment in Verycd is more complex than our requirements in CommentCast. Its actually a piece of HTML code. Besides the text message, it can also include hyperlinks, pictures, Flash and eMule links. Since CommentCast is a text-based system, we do not need the extra information beyond text. We take the size of comments without any text as 0. There are 14,789 0-size comments in 1,137,237 comments, which accounts for 1.30% in total. Because pictures and links influence the size of a text message, we remove these 0-size comments in our statistical results.

The total size of comments is about 75Mbytes for 1 million comments. The average size is 65.9 Bytes per comment. A million comments were collected during almost four years. The size of comment is not overwhelming for computers nowadays. Another important fact is that more than 75% of the comments are less

	Bytes	**Lines**
Total	75,056,308	1,600,475
Mean	65.90	1.43
Max	123,500	2,516
99-percent quantile	572	6
95-percent quantile	182	3
85-percent quantile	82	1
3rd-quartile	56	1
1st-quartile	18	1

Table 4.2: The size of comments in Verycd in terms of Bytes and lines. First row is the size of all comments. Second row is the mean value of a comment size, followed by max value, 99-percent quantile, 95-percent quantile, 85-percent quantile, 3rd-quartile and then 1st-quartile

than the average size, 25% comments are less than 18 Bytes and 85% comments only have 1 line. Usually, a longer comment is more of interest to other users. On the other hand, a short comment can hardly carry any information. However, its difficult to tell what kind of messages is more useful without linguistic study.

4.2.3 Comments per Thread

We already have the approximate size of all comments in Verycd, but user behavior may change from time to time. For further analysis, we would like to find the average number of comments in threads. Figure 4.5 shows the cumulative distribution function (CDF) of the number of comments for every thread and some corresponding statistical values are in Table 4.3. Figure 4.5 shows that only a small portion of threads, less then 10%, has a large number of comments and a quarter of threads have no more than 6 comments. The mean value, 41.73 comments per threads, is another important value for estimating the disk space taken by CommentCast. Base on the result from the last subsection (65.9Bytes per comments), a thread takes on average 2.75Kbytes.

Max.	**99% quantile**	**90% quantile**	**3rd quartile**	**Median**	**Mean**	**1st quartile**
8031	454	79	42	18	41.73	6

Table 4.3: Some useful statistical values of Cumulative Distributed Function of comments per threads in Figure 4.5.

We divide threads into two groups, intensive threads and sparse threads, according to a quantile point in Table 4.3. Take the 90%quantile point as an example, threads with more than 79 comments are called intensive threads, and threads with no more than 79 comments are called sparse threads. For the 90% quantile point, intensive threads include 645214 comments, which accounts for 52.4% of com-

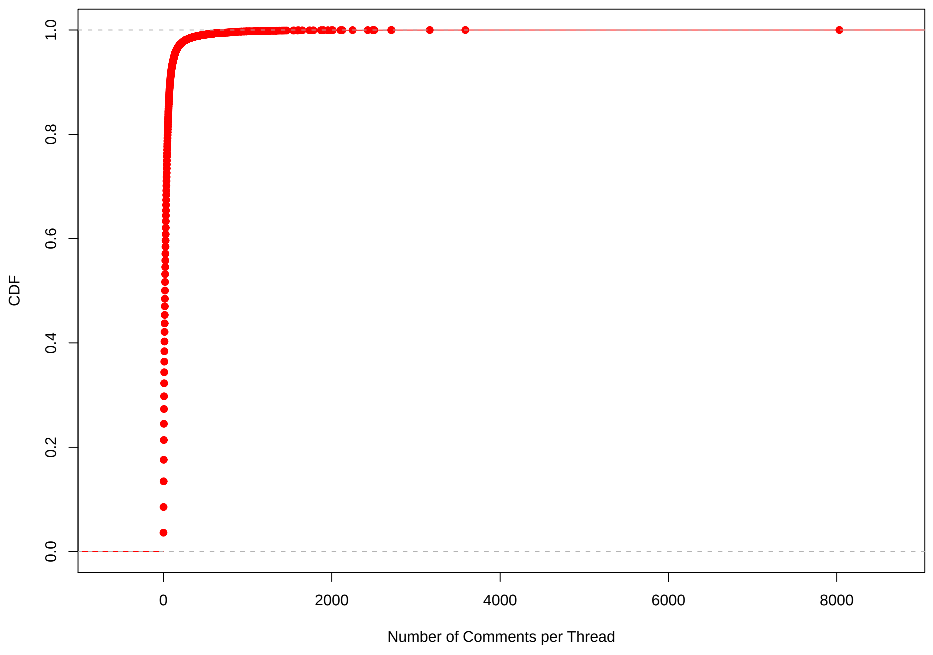


Figure 4.5: Cumulative Distributed Function of number of comments per thread.

ments. Moving to the 1st quartile, we find that the intensive threads include 91.7% of all comments. This means the top 10-percent most intensive threads contribute more than 50% comments and more than 25% of the threads, the sparse threads, contribute only 10% of comments.

4.3 Number of Users and User Behavior

Analysis of how many unique users behind the a million comments in Verycd is also important in designing CommentCast. The number of users largely affects the scale of a P2P system. It also affects the design of the disseminating algorithm as well.

Our data shows that there are 186,109 unique users that have left at least one comment in Verycd. But almost half of them only posted one comment. The number of users decrease rapidly as the number of comments that they post increases. This information is shown in Table 4.4. It means that most users only occasionally leave comments.

On the other hand, we also find that most comments are posted by a small amount of users. The top 10 most active users with the number of their comments are listed in Table 4.5. We calculate the number of comments posted by the most 10-percent

	number of users	**Percentage**
Unique users	186,109	100%
1-comment users	88,723	47.7%
2-comments users	30,167	16.2%
3-comments users	14,989	8.1%
4-comments users	9352	5.0%

Table 4.4: Unique users is the number of users who left at least one comment. X-comment users means the number of users who posted exactly X comments.

active users. The result is 732,958 which accounts for 77.6% of all comments. It means that the Pareto principle, saying that roughly 80% of the effects come from 20% of the causes, also applies in the context of user commenting behavior. The Pareto principle somehow contradicts the nature of fully decentralized system in which peers are exactly same [2].

Rank	**Number of comment**	**User ID**
1	4857	@u7015659
2	2131	@u6896514
3	2020	@u5492518
4	1572	@u2672386
5	1495	@u1447113
6	1425	@u4842412
7	1076	@u5646300
8	1064	@u3440818
9	1063	@u5319687
10	1058	@u1621428

Table 4.5: A list of top-10 most active users. The middle column is the number of comments posted by the users. The last column is the Id of the user.

From Table 4.4, we can see the users who post less than 5 comments already account for roughly 77% percent of total users. As stated before, Verycd allows anonymous users to download and read comments, which means the registered users could be only a fraction. We can not tell the exact number of users reading the comments. What we know is that there must be an amount of anonymous users. Those anonymous users may have no interest in online discussion or even have no interest in receiving comments. Therefore, the cumulative number of users who post less than 5 comments is even larger than 77% percent and this number could be very large. Based on that fact, it is reasonable to let user to choose whether they use CommentCast or not.

Chapter 5

Exploration and Analysis of the Push Protocol

We discussed some of the possible solutions for the push protocol in previous chapter. We decided to focus on customizing LightFlood in Tribler's overlay. We have chosen LightFlood because our primary goal is high dissemination speed, thus the flooding-based algorithm is preferred. Additionally, LightFlood doesn't need complex signing messages or extra resources to implement. Last but not least, LightFlood fits in the context of Tribler's overlay due to strong connectivity.

This chapter explores the performance of LightFlood scheme in Tribler based on experimental results. We focus on the dissemination speed and the redundancy of dissemination. To facilitate comparison, we simulate the process of disseminating a single comment.

We simulate 3 algorithms. First one is a proactive gossiping. We generalize the original CommentCast as basic gossiping and make a simple improvement to speed up the dissemination without any extra retransmission. In the second simulation, we investigate the performance and redundancy of a flooding algorithm based on Tribler's overlay. Finally, we customize LightFlood for Tribler's overlay and simulate the dissemination of LightFlood. After comparison, we find that LightFlood is the best choice for the push protocol. Thus we investigate the performance of LightFlood under churn through another simulation.

5.1 Overall Description

Before our simulation, we will describe Tribler specifications, overlay construction, and experimental environment.

5.1.1 Tribler Specifications

Our experiments are based on Tribler's environment. It is also important describe specifications of Tribler. The complete specifications of Tribler can be found in

[33]. We only list some properties relevant to our experiments:

- Tribler's overlay is a taste-based-clustered graph. Peers are distributed in different clusters.
- Every peer has 10 fixed neighbors with similar taste and 10 random neighbors selected randomly the entire peer set. Fixed neighbors are always the peers who have most similar tastes with the peer itself. The random peers are not fixed and change over time.
- Peers are able to acquire random peers from the entire network.
- Every peer maintains TCP connections with their neighbors, which means peers are able detect logoff of their neighbors. Then they select new neighbors.

Our simulation does not use actual data from the Tribler overlay or peer tastes, since we do not have this information. Thus fixed neighbors are also selected randomly. The difference between fixed neighbors and random peers in our simulation is that fixed neighbors do not change during the simulation after selection while the random neighbors are select randomly at the beginning of every cycle of simulation.

5.1.2 Overlay Construction

Our experiment simulates data dissemination in Tribler's overlay which is a strongly connected network with clusters based on personal taste. Since we don't have data of Tribler's overlay, overlays are constructed in a simulation. There is a lot of existing work in creating a random graph or a random graph with an expected degree in [4] [23]. However, creating an overlay is not part of the scope of this thesis. We would like to apply the overlay in [19] which is used in a similar context as in our simulation.

We assume the cluster size is 50 [19]. Thus, the number of clusters C in a network, $C = Network\,Size/50$. Then we randomly distribute all peers into C clusters and every peer also keeps a reference of its cluster. This way, every cluster has around 50 peers, but not exactly 50 peers, which is an more realistic overlay. From a comparison experiment, we know that clustering causes a slight change of redundancy. Our observation is the number of redundant messages increased a little bit faster in first cycles, however, the total number of redundant messages is almost the same. The clustering effect is a complex problem, which is not of concern in this thesis. Therefore, we ignore the clustering effect in the following simulations.

Overlay construction is based on a fact that having a taste buddy, or a fixed neighbor in our simulation, is a mutual relationship, which means, based on a same

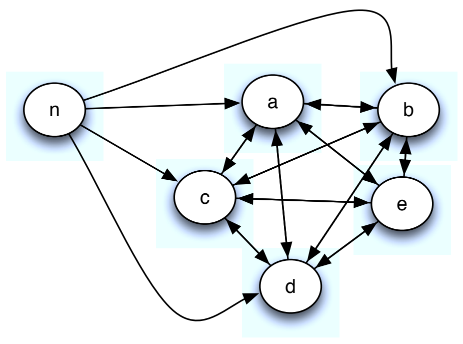


Figure 5.1: A diagram of peers that can not establish bidirectional connection.

similarity evaluation function that peer A is a taste buddy of peer B and B is also a taste buddy of peer A. Or we can say, links are bidirectional connections. When peer A selects peer B as its neighbor, peer B also adds peer A as its neighbor.

It is necessary to take the degree of peers into account when establishing connections in the simulation because another property of Tribler is that peers maintain 10 connections to their taste buddies. Under the fixed degree constrain, the mutual neighborhood introduces another problem that some peers may not find neighbors when the other peers in cluster have enough neighbors. An example is shown in Figure 5.1 where we take degree of 4 as an example. Peer a, b, c, d and e have already established bidirectional link with neighbors. The new peer n can not find any peer to connect with. So peer n connect a, b, c and d without bidirectional link. Our algorithm for these peers is repeat random picking form their cluster until the number of failures is greater than twice the size of the cluster. Then, they randomly establish unidirectional connections. The example in figure 5.1 is an extreme case, but it happened a lot during our simulation that bidirectional connections can not be established.

The flowchart of overlay construction is shown in Figure 5.2. The process of construction is on left hand side. N is the network size and C is the number of clusters which have been described already. Establishing overlay links is a sub-procedure which is on the right hand side. The sub-procedure is executed on every peer of the overlay.

5.1.3 Experimental Environment

The real world distributed systems are asynchronous systems. However, [19] shows that simulating a large scale network asynchronously is almost impossible based on normal computers. Therefore, we would like to simplify the process into a series of synchronous working processes. In order to ensure the reproducibility of our

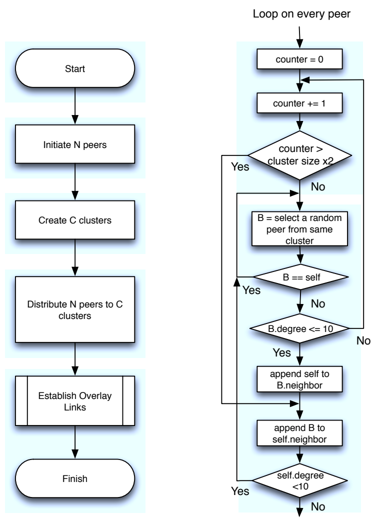


Figure 5.2: A flowchart showing the construction of the overlay. Entire process is on the left hand side. A sub-procedure of link establishment is on the right hand side. The sub-procedure is executed on every peer of the overlay.

simulations, it is necessary to make following assumptions:

1. Peers keep repeating communications cycles in which they communicate to each other in an arbitrary order.
2. Communications including sending and receiving takes one cycle to finish, which means the receiver can not forward a received message in the same cycle.
3. Peers can finish arbitrary local actions within one cycle. This means every peer can send its message to all neighbor peers with in one cycle.

In the real network environment, network communication are full of uncertainty. It's hard to define the duration of a communication cycle. We will compare the speed of different algorithms in terms of communication cycles.

5.1.4 Important Terms

We would like to distinguish two useful terms for our simulations, which are very important metrics.

A common term, coverage, is a personal view of global data, which means what fraction of all comments a peer has. I order to distinguish from the later term, we call it **data coverage**. For $peer_i$ data coverage could be represented as:

$$DataCoverage_i = \frac{Local\,Comments\,of\,Peer_i}{Global\,Comments\,of\,the\,Network}$$

For example, there are 100 comments in Tribler, $peer_i$ has 85 comments in its local database. Then for $peer_i$, the $DataCoverage_i = 85\%$.

The other term, called **overlay coverage**, means the fraction of infected peers in a network for a particular message. For a $comment_k$

$$OverlayCoverage_k = \frac{N_k}{Network\, Size}$$

The N_k is number of peers who are infected by $comment_k$ and the $Network\, Size$ is the total number of peers in Tribler. It means how many percents of peers have been infected by $comment_k$.

We usually consider an average data coverage of all peers. Thus two coverages are identical when there is only one message in the simulation. Data coverage is more of concern while overlay coverage is more intuitive in analyzing the disseminating speed of algorithms. Since this chapter only covers disseminating of one message, the coverage will refer to overlay coverage through out this chapter. The data coverage will not be mentioned until next chapter.

5.2 Simulation of Original CommentCast

This section presents the simulation of original CommentCast, then improves it with a very simple scheme to speed it up. The improved result is taken as a baseline for later simulations.

5.2.1 Simulation Specifications

Considering a process of spreading one comment, the original CommentCast works as follows: after receiving a message, every peer starts to relay the message to their neighbors from the next cycle. They stop sending until the comment is not a recent comment any more. The algorithm is similar to the proactive basic gossiping algorithm introduced in 1.2.1.

- A comment message is injected by a random peer of the network.
- Every cycle, peers who have received the comment take a random peer from the network to send the comment to.
- Receivers put the message in the receiving buffer when they receive a comment.
- The comment is not added to the comment list until all peers finish sending in the cycle.

- The simulation is done when the overlay coverage reaches 100%.
- There is a overall controller who is responsible for recording overlay coverage and total number of redundant messages in every cycle.

5.2.2 Simulation Result

Figure 5.3 shows the result of overlay coverage of a single comment in each cycle. In the figure, the vertical-axis is the overlay coverage as percentage. The horizontal-axis represents the communication cycles. We can see a clear trend in how the overlay coverage grow as the communication cycle increases.

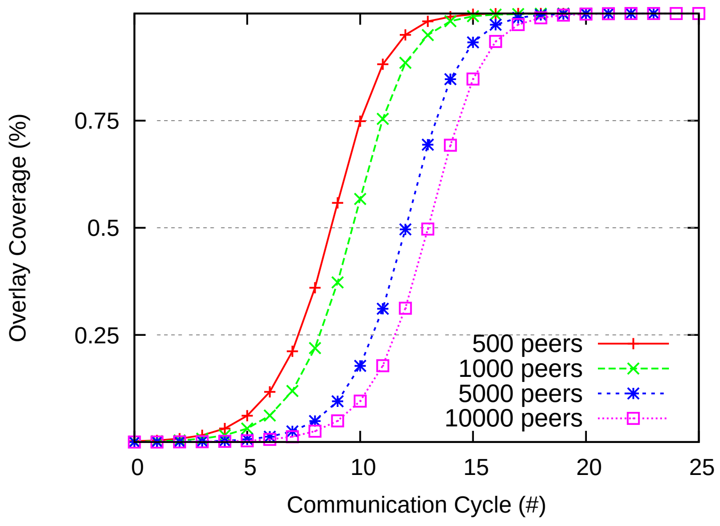


Figure 5.3: Disseminating Speed of the Basic Epidemic algorithm. Vertical-axis is the overlay coverage, horizontal-axis is the cycles of communication. The 4 lines stand for disseminating rate of the same algorithm for 4 different network size of 500 peers, 1000 peers, 5000 peers and 10000 peers

We did the same simulation on 4 different networks that are 500 peers, 1000 peers, 5000 peers and 10000 peers. Every simulation was repeated more than 30 times and the results were averaged to keep reproducibility of the figure. It is clear that the basic epidemic algorithm is scalable for for large scale networks.

5.2.3 Speed Up by Increasing of Initial Fanout

From the result of basic gossiping, we know that the overlay coverage grows fast during the coverage is between 0.25 to 0.75. However, it is not ideal both in first 25 percent and last 25 percent of coverage. In first 25 percent coverage, too few peers are infected. In last 25 percent of coverage, blind transmission mostly generates redundant transmissions.

When the initiator posts a comment, he is the only infected peer. Under the design assumption that every peer is expecting new comments, it is for sure that pushing a comment to all its neighbors does not generate any redundant messages. Thus, a simple improvement is that when a user initiate a comment, peers immediately push the comment to all his neighbors. Figure 5.4 shows a comparison between basic gossiping and improved gossiping with the push scheme. We call the improved gossiping as 'fast-push gossiping', or short as 'push'.

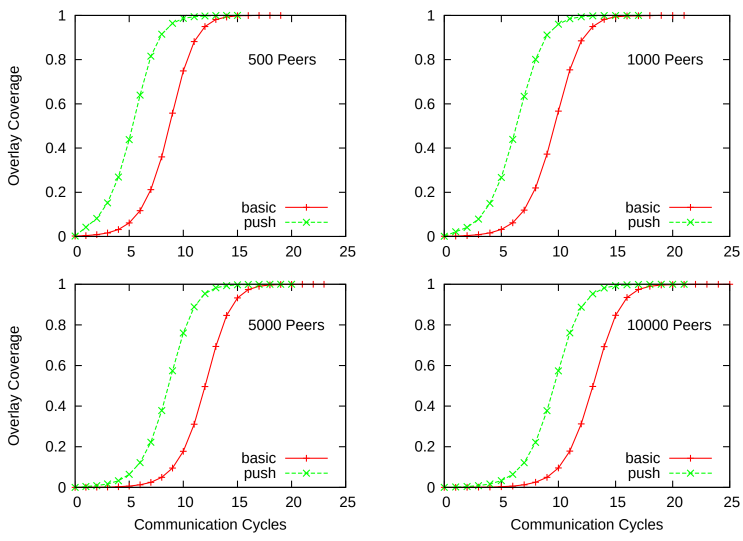


Figure 5.4: Comparison of overlay coverage between basic epidemic and fast-push gossiping. The vertical-axises are overlay coverages and horizontal-axises are communication cycles. Four results from different networks are shown, which are 500 peers at top left, 1000 peers at top right, 5000 peers at bottom left and 10000 peers at bottom right.

The fast-push schema increases the coverage growth of the basic gossiping. It utilizes available bandwidth to initial coverage of dissemination. The redundancy of fast-push gossiping is the same as the redundancy of basic gossiping. Thus the fast-push gossiping triumphs the basic gossiping in terms of speed and redundancy.

Redundancy of the algorithm is discussed in 5.4.3.

5.3 Simulation of Flooding

Flooding is a well known algorithm for unstructured networks. It is very reliable so that it is popular for topology updates or routing information discovering. In a flooding algorithm, the peer who received a message broadcasts the message to all its neighbors. The flooding algorithm used in the simulation was described in 1.2.2.

It is obvious that disseminating speed of flooding depends on the degree of peers and how peers are connected to each other. The Tribler overlay is strongly connected. [20] shows that most P2P overlays rely on high-degree super peers, thus the average degree is much lower, compared with Tribler. Thus we can expect the overlay coverage of flooding increases very fast in the Tribler's overlay. On the other hand, a highly clustered network, or we say a network with a high clustering coefficient, may lead to extra redundancy.

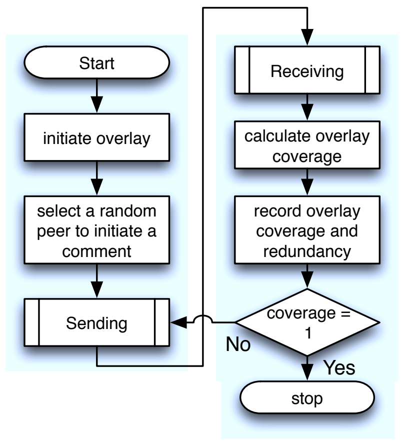

Start
initiate overlay
select a random peer to initiate a comment
Sending
Receiving
calculate overlay coverage
record overlay coverage and redundancy
coverage = 1
No
Yes
stop

Figure 5.5: Flow chart of flooding; the simulation repeats until the overlay coverage reach 100%. Sending and receiving are two loop procedures and are shown in figure 5.6

5.3.1 Simulation Specifications

The flooding algorithm was introduced in 1.2.2. The flow chart of the simulation is shown in Figure 5.5. The simulation does not stop until all peers are infected by the comment. During the simulation, overlay coverage and number of redundant messages is recorded for every cycle of communication. Every cycle of communication includes two sub-procedures, sending and receiving. Sending and receiving are loop-processes which are executed on every peer of the overlay. The receiving buffer and sending buffer are two stacks that can keep multiple comments which can be popped out for processing. The flow charts of sending and receiving are shown in Figure 5.6. The left one is for sending and the right one is for receiving.

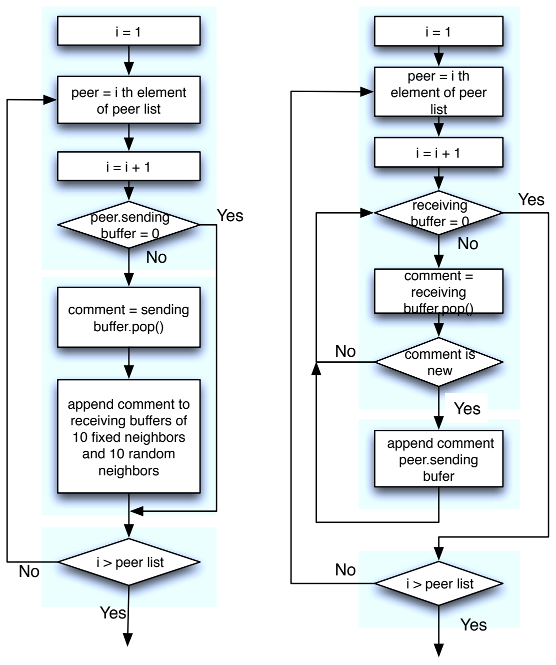

i = 1
peer = i th element of peer list
i = i + 1
peer.sending buffer = 0
Yes
No
comment = sending buffer.pop()
append comment to receiving buffers of 10 fixed neighbors and 10 random neighbors
i > peer list
No
Yes
i = 1
peer = i th element of peer list
i = i + 1
receiving buffer = 0
Yes
No
comment = receiving buffer.pop()
comment is new
No
Yes
append comment peer.sending bufer
i > peer list
No
Yes

Figure 5.6: Flow chart of the sending procedure(left) and flow chart of the receiving procedure(right). These two procedures are two components of flooding simulation.

5.3.2 Simulation Result

Figure 5.7 shows a graph which compares disseminating speed of flooding and fast-push gossiping. The horizontal-axis is adjusted for the flooding algorithm. Thus, some data of fast-push gossiping is ignored. The coverages of the first cycle are exactly the same in fast-push gossiping and flooding, because the two algorithms are identical in the first cycle. However, the coverage of flooding increases sharply after the first cycle even in the largest experiment which has 10 thousands peers. It only takes 4 cycles to cover entire network. It is also important to notice that the number of redundant messages is huge. It is too expensive for data dissemination. The gossiping algorithm is sometimes called the anti-entropy algorithm, and the major problem of flooding is the redundancy. There is no need to compare the result of the redundancy in the two algorithms. The redundancy comparison is shown in 5.4.3.

5.4 Simulation of LightFlood

Flooding in Tribler's overlay is a very fast algorithm. However, the redundancy is overwhelming. It is necessary to reduce the number of retransmissions in flooding. LightFlood, introduced by [14], is a smart scheme of flooding. It greatly reduces

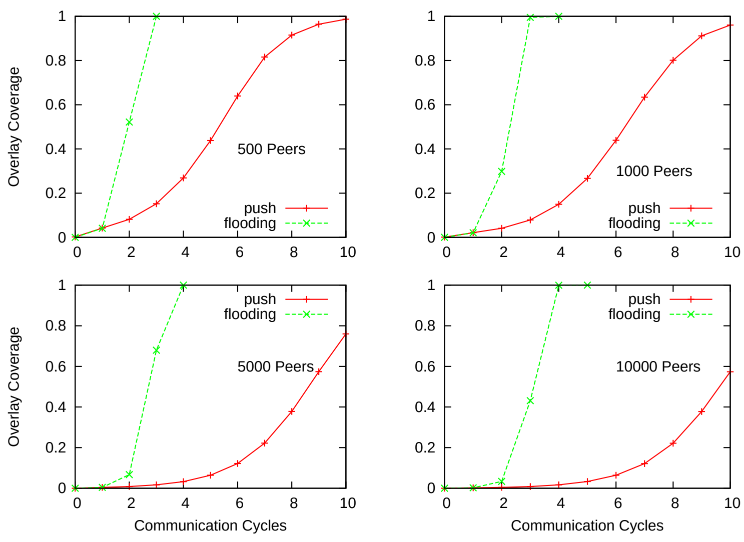


Figure 5.7: Comparison of overlay coverage between flooding and fast-push gossiping. The vertical-axises are overlay coverages and horizontal-axises are communication cycles. Four different networks are compared, which are 500 peers at top left, 1000 peers at top right, 5000 peers at bottom left and 10000 peers at bottom right.

redundant messages of flooding by constructing a distributed spanning tree and combine pure flooding and spanning tree broadcasting. We have already discussed LightFlood in 3.2.1. Here, we focus on building a spanning tree on the Tribler overlay, customizing LightFlood and evaluating the performance of LightFlood.

5.4.1 Constructing The Distributed Spanning Tree

We construct our spanning tree by using the PermIDs. Peers take the neighbor who has the smallest PermID as their parent peer. Every peer sends a HELLO message to his parent peer, so that the parent peers can know their children and keep them in a list for tree broadcasting.

There are several reasons of using PermID to construct spanning tree. First of all, PermIDs are one of the most accessible information. It is very cheap in terms communication, because Tribler has the PermIDs of all neighbors. No extra communication is required. Therefore, communication complexity of creating a distributed spanning tree is $C(N) = O(1)$ for a network with N peers. Secondly, PermIDs are unique identifiers of users. They are easy to compare and the result of comparison is transferable from peer to peer.

5.4.2 Simulation Specifications

The basic idea of LightFlood has been described in 3.2.1. We only would like to stress that received messages are forwarded only if they haven't received it before, like in flooding. The process of LightFlood is almost the same as flooding in Figure 5.5 and Figure 5.6. There is only one difference: The step append comment to receiving buffer of 10 fixed neighbors and 10 random neighbors', called the Flooding step, in Figure 5.6 is modified. The new process replacing the step is shown in 5.8. Threshold in 5.8 indicates whether to flood the message to all neighbors or broadcast the message on the spanning tree. Hop count is more easy to under-

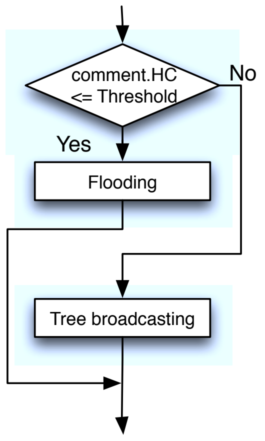


Figure 5.8: The new process for LightFlood replacing the flooding step in the process from Figure 5.6.

stand in LightFlood. We replace TTL with hop count (HC) which starts from 0 and increases by 1 when the message is forwarded. Therefore, in every cycle, the hop count of message received by infected peers is equal to the cycle. Peers do flooding when the hop count of the received comment is less or equal to the threshold, or do tree broadcasting if the hop count is greater than the threshold.

Thresholds are chosen based on the result of previous simulations of the Flooding algorithm. From the results in Figure 5.7, we can see the coverage reaches 0.5, 0.3 in cycle 2 respectively for 500 peers and 1000 peers, and 0.7, 0.4 respectively in cycle 3 for 5000 peers and 10000 peers. Our data also shows that the redundancy increase sharply from cycle 2 to cycle 3 for 500 peers and 1000 peers, from cycle 3 to cycle 4 for 5000 peers and 10000 peers.Optimal choice is obvious. The threshold will be 2 for 500 peers and 1000 peers, 3 for 5000 peers and 10000 peers. In the simulation, 2 and 3 were chosen as the thresholds.

Our target is to achieve 100 percent coverage and the simulations run until coverage reaches 100%. However, our results show that LightFlood can not reach 100% coverage without changing rule to only forward new messages. Therefore, the stop condition is modified so that the simulation stops when coverage doesn't increase for several cycles. We will discuss that coverage issue in 5.4.4.

5.4.3 Simulation Results

Redundancy

LightFlood targets reducing redundancy, so the redundancy is the main concern. In order to keep the redundancies of different networks comparable, we normalize the redundant messages to a redundancy index. The redundancy index equals the number of redundant messages divided by the network size. It represents the average number of redundant messages per peer. The redundancy indexes are shown in Table 5.1.

	500 peers	**1000 peers**	**5000 peers**	**10000 peers**
LightFlood	3.25	2.92	3.70	3.15
Flooding	18.77	18.77	18.76	18.76
Gossiping	5.94	6.90	8.90	8.70

Table 5.1: Redundancy Indexes for LightFlood, pure flooding and proactive gossiping.

We can see LightFlood effectively decreased redundancy of flooding in Tribler. On average, the redundancy of tree flooding is less than 10 percent of the redundancy of flooding. The redundancy is quite high in pure flooding, where every peer transmit 20 times every comment (1 + 19 retransmissions). In LightFlood, the redundancy is restricted to an acceptable range that is around 3 retransmissions for every comment.

What's more surprising is that the redundancy of Lightflood is even much less than the redundancy of gossiping. Gossiping is usually taken as an anti-entropy algorithm. But, we can not say the LightFlood it a better anti-entropy algorithm than gossiping, because usually no gossip is supposed to achieve 100% coverage, and reaching the last several percents of peers is very expensive for gossiping.

We also notice that the redundancies of pure flooding are quite stable as the network size changes. But the redundancy of spanning tree flooding varies largely in different networks. A more interesting phenomenon is that we can not find any pattern as the network size increases. The largest index comes from 5000 peers which is the second largest network while the largest network, with 10000 peers, has a very close index, second smallest, to the smallest network which has only 500 peers.

In order to find out the reason behind the irregular redundancy indexes, we extracted the messages generated during tree broadcasting, and then normalized them dividing by the network size. Table 5.2 shows the redundant indexes during tree broadcasting and the coverage when it switched from flooding to tree broadcasting.

	500 peers	**1000 peers**	**5000 peers**	**10000 peers**
Total	3.25	2.92	3.70	3.15
Tree part	2.93	2.80	3.02	2.90
Coverage	0.52	0.29	0.68	0.43

Table 5.2: Redundancy table. The first line is the network size. The second line is the redundancy indexes for entire LightFlood algorithm. The third line is the redundancy indexes for messages generated during broadcasting on the tree. The forth line is the coverage when it switches from flooding to tree broadcasting.

We can see that the largest redundancy index is 3.02 for 5000 peers and the largest networks size generates the second smallest index which is 2.90. Apparently, it is still not clear how does the network size affect the redundant messages generated by the algorithm. However, the redundancy index has an obvious rising trend as the initial coverage increases. Therefore, the conclusion is that the redundancy index doesn't depend on the network size but largely depends on the overlay coverage when it switches from flooding to broadcasting on tree. We will have a wide exploration of different LightFlood schemes in 6.1.1.

Dissemination Speed

Disseminating speed is also important to the algorithm. Figure 5.9 compares the dissemination in LightFlood, pure flooding and fast-push epidemic. Generally speaking, the disseminating speed of spanning tree flooding is very good. In most cases, LightFlood is almost as fast as pure flooding which is at the upper bound of disseminating.

However, in small networks, of 500 peers and 1000 peers, LightFlood is just a little faster than fast-push epidemic, even though the performance of spanning tree flooding is ideal for 90% coverage. A more significant problem is that spanning tree flooding can not reach 100% coverage. In order to improve the finial coverage of LightFlood, we refine the distributed spanning tree.

5.4.4 Refine Distributed Spanning Tree

We realize that our algorithm for constructing a spanning tree has a defect that it can not ensure the resulting spanning tree is connected. Figure 5.10 shows an example that the network will be portioned into two separated parts. Peer A has the

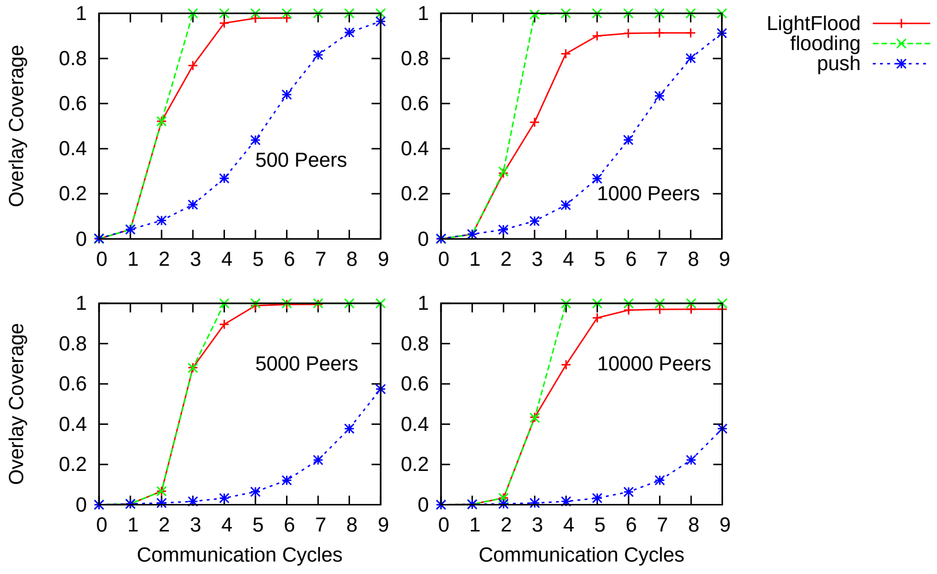


Figure 5.9: Disseminating speed of LightFlood, Flooding and Fast-push gossiping.

smallest PermID in the network. Peer B has the second smallest PermID in the network. White peers in the figure select A as their parent, black peers select B as their parent. Although the network is strongly connected, constructed spanning tree is partitioned in two subtrees.

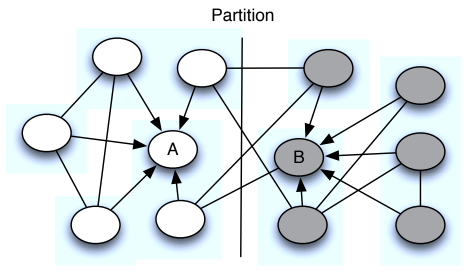


Figure 5.10: Tree partition. An example of tree partition. Peer A has the smallest id in the network, peer B has the second smallest id in the network, but they are not connected directly. Lines and arrows both are the overlay links. Arrows also shows the spanning tree links. Pointed peers are parents.

Multiple subtrees exists in the network. We expect merging partitioned trees could improve overlay coverage. In the simulation, it is easy to tell which peer is the root of the fully connected spanning tree, called the complete tree. However, in an unstructured distributed environment, finding the root of a tree is quite expensive. The merge algorithm should be iterated finite times. Here we consider the number of iterations be 3 and we compare performance of the tree produced by 3 iterations and the complete tree.

The pseudo code of merge algorithm is in Algorithm 1. The merge algorithm works as follows: whenever a peer, peer A, doesn't have parent, it will randomly pick a peer and keep requesting the parent of the selected peer until peer A find the root of the tree containing the selected peer. We call the root as peer B. Then it tries to combine it self, peer A and peer B. Combining principles: 1: If the PermID of peer B is smaller than the PermID of peer A, peer A will take peer B as its parent and break out to finish the merge algorithm. 2: If the PermID of peer B is larger than the PermID of peer A, peer B will take peer A as its parent, then it begins a new iteration. The merge algorithm requires consistent control procedures in a real distributed system, but it is not the concern of this thesis, we do not cover this issue.

Algorithm 1 Tree Partition Merging Algorithem

```
if self.parent = NULL then
  repeat
    p = random peer
    while p.parent is not null do
      p = p.parent
    end while
    if self.id $<$ p.id then
      p.parent = self
    else
      if self.id $>$ p.id then
        self.parent = p
        BREAK
      end if
    end if
    $i = i + 1$
  until $i \geq iteration$
end if
```

The results show the merge algorithm increases the overlay coverage and also slightly increase the speed of achieving final coverage. The number of peers covered by original tree, the tree after 3 iterations of merge, and the complete tree are shown in Table 5.3. We repeat the simulation 20 times to get the average values. We can see that the tree after 3 iterations of merge and the completed tree almost give the same results, especially in small networks.

The complete tree still can not ensure that LightFlood covers 100% peers. In order to find the reason behind this, we modified the simulation in which peers retransmit any message during the tree broadcasting part no matter if they were infected before. After the modification, LightFlood finally reaches 100% coverage.

	500 peers	1000 peers	5000 peers	10000 peers
Original	489.9	914.3	4943.8	9706.2
3 iterations	499.6	999.2	4992.1	9987.1
Complete	499.6	999.3	4993.6	9990.4

Table 5.3: An average overlay coverage of LightFlood with different spanning trees in terms of number of peers. The first row are the peers covered by LightFlood with the original spanning tree. The second row are peers covered by LightFlood with a tree after 3 iterations of the merge algorithm. The third row are peers covered by LightFlood with a complete tree.

But the the number of redundant messages became too large to accept, and it increased rapidly as the network size increased. Because push protocol doesn't have to reach 100% coverage, the tree after 3 iterations of merge is good enough for the push protocol.

5.4.5 Influence of Churn

We have already seen that the spanning tree is very fast at data disseminating and it is very cost-effective while achieving the high disseminating speed. It is also important to know if it is stable under churn. Churn is the effect of many users leaving or joining the system at the same time [31]. Churn doesn't lead to big problems under client/server model especially for peers leaving, because the sever takes the key role in communication and the rest of the peers work well. However, churn is a very disturbing problem persisting in Peer-to-Peer systems. In structured P2P system, frequent joining and leaving cause huge system overhead. The unstructured P2P systems usually handle joining issue elegantly. However, large amount of peers leaving may break down of the system. Here, we focus on the structure problem caused by peers leaving.

The spanning tree itself is an very fragile structure when it comes to churn. Any leaving peer causes partitioning of the spanning tree, or we say the node connectivity of spanning tree is 1.

LightFlood is not fragile to churn since it utilizes the randomness of flooding. The algorithm is duplicating the source message abundantly before broadcasting on the spanning tree. Figure 5.11 shows a diagram of the duplicated source message. The message initiator randomly makes a lot copies of the message, which is the pure flooding part of the algorithm. When the hop count is large enough to stop random duplicating, multiple messages are broadcasted on the spanning tree. The more message sources exist in spanning tree the more reliable the algorithm is. It is the reason behind the reliability of the algorithm despite the fragile spanning tree.

Another advantage of LightFlood is that building spanning tree almost doesn't require any extra cost. It only needs local information. Furthermore, the Tribler

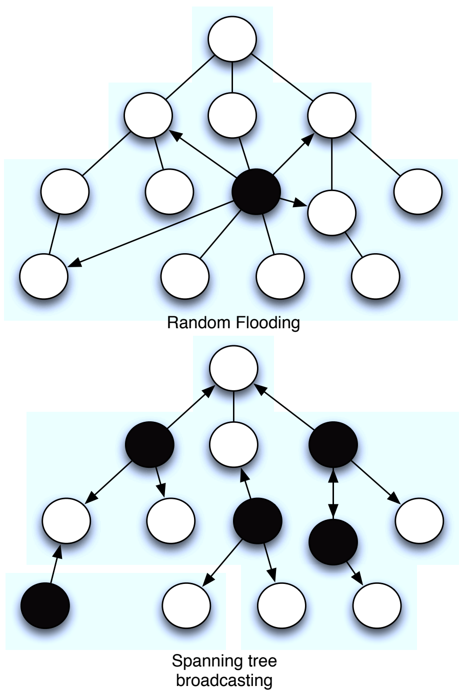


Figure 5.11: A diagram of duplicated messages in LightFlood. The black nodes are the peers who received the message during random flooding. They broadcast the message along the spanning tree.

client is able to detect the leaving of neighbors, because the clients maintain TCP connections with their neighbors. When they detect their parent is gone, clients can react rapidly by selecting another parent with local information and recover the spanning tree structure. The partition merging algorithm is not that fast, taking several communication cycles. The merging algorithm is not taken into account.

We will focus on the simulation of the system with 10,000 of peers in churn, since the LightFlood does not depend on the network size and large scale networks are more of a concern for Tribler. After initiating an overlay with 10000 peers as the overlay in LightFlood with partition merging, a number of peers are randomly removed. Then, the same spanning tree flooding algorithm will be carried out on the system without any tree recovery. The number of removed peers are decided by a given churn rate which increases from 0.05 to 0.5. The churn rate means what percent of peers in the network are removed. For example, a churn rate of 0.1 means 10 percent of the peers are removed. We are interested in the impact of churn on the final coverage of LightFlood for the rest of the peers, because the influence in each step is trivial and the final coverage is a cumulative result of the churn effect. The figure 5.12 shows the results of the simulation.

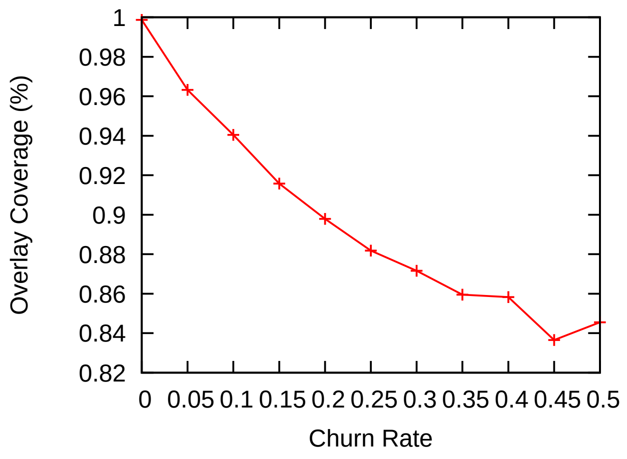


Figure 5.12: The results of churn simulation. The horizontal-axis shows the churn rate which is the percentage of peers are removed. The vertical-axis shows the final coverage for the remaining peers using spanning tree algorithm.

From the graph we can see the overlay coverage falls elegantly as the churn rate increases. Even when the churn rate reaches 0.5, which means 50% of peers are removed, the spanning tree flooding can still cover over 80% of the remaining peers. Under such an extremely high churn rate, LightFlood without tree-recovery remains very stable. Based on the discussion above, we believe the LightFlood will have an excellent performance in the real Tribler client.

Chapter 6

Simulation of LightFlood under Real-World Workload

In the previous chapter, we have explored the theoretical performance of different disseminating algorithms. The results show that LightFlood is the best choice among them which reaches high coverage with a relatively low redundancy. We investigate the performance of LightFlood in a more realistic context which is based on a real-world workload. The simulation focuses on network traffic and data coverage under a real workload.

6.1 Simulation Setup

The size of a network greatly affects the performance of a disseminating algorithm. Estimating of the size of the target network is very important in a simulation. However, the size of a network keeps changing all the time. It is hard to give a precise estimation. Tribler is devoted to provide service to over a million users. Considering the processing power of our machine, simulating such a huge network is not realistic. We will take 10,000 peers as network size of our simulation.

The term coverage refers to the overlay coverage in previous simulations. Since the simulations in this chapter involve multiple comments, Any further usage of coverage will refer to data coverage in this chapter. Data coverage and overlay coverage are described in 5.1.4.

In order to provide appropriate workload to our simulation, we crawled the entire history of comments of the movie section on Verycd.com, which is a commercial website providing download resources for the eMule protocol. It has a very similar context to CommentCast. The detailed description of the data is in Chapter 4. Figure 4.3 in Chapter 4 shows that comment arrival rate keeps increasing after the release of the commenting system of Verycd. It begins to fluctuate around 2000 comments per day from September of 2009 until the end of our crawling. Therefore, we will use the comments between Sep. 1, 2009 and Jun. 6, 2010.

We select comments based on hourly workload, because the hour is an appropriate granularity. Workload can change largely with in a day, and users usually stay online only for several hours. We randomly pick a moment within the period, then find all comments injected in the following hour after that moment. In order to eliminate bias of using different hours, we randomly choose 100 hours and import the comments posted in those 100 hours into a file as our workload. Based on the random picking, we get 7,806 comments and every comment has its own size. The total size is 466 Kbytes for the 7,806 comments.

There are 5,088 actual users behind those comments where the most active user posed 32 comments and 3,775 users only posed 1 comment. Our simulator needs 10000 user IDs to initiate the network. We still need some extra users besides the actual 5,088 users and these extra users will not post any comments during the simulation. The extra users will randomly use vacant IDs between the largest id and smallest id of the actual users.

We continue to use the overlay and the spanning tree from the previous simulations in Chapter 5. Disseminating each comment is taken as separated processes, which means the next comment is not injected until dissemination of the previous comment is finished. The simulator works as follows: 1. Select the comments for the random 100 hours and put them into a file 2. Read a comment from the file 3. Do LightFlood, if the read comment is not null. The flowchart of the simulation is in Figure 6.1.

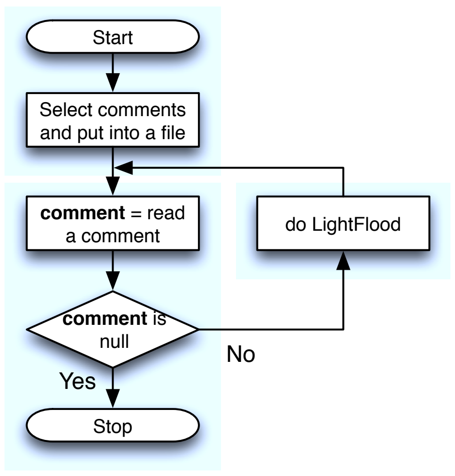


Figure 6.1: The flowchart of the simulation

6.1.1 Setting Parameters of LightFlood

The overlay coverage and redundancy of LightFlood are affected directly by the parameters, m and n, which are hops of pure flooding and hops of tree broadcasting. In the previous simulations, we take the most intuitive value of m where the coverage reaches around 50% while the redundancy doesn't increase sharply, then we keep disseminating by broadcasting on spanning tree, which means increase the value of n, until the message is disseminated to all peers. However, real deployment needs fixed values of m and n. Therefore, we will investigate the performance of different LightFlood schemes with different pairs of the m and n values.

We use different values of m and n, ranging from 1 to 4 and 0 to 4 respectively, to disseminate a hundred comments through the process in Figure 6.1. The average results of data coverage and redundancy are shown in Table 6.1 and 6.2. Four hops of pure flooding reach almost 100% coverage, thus (4, 1), (4, 2), (4, 3) and (4, 4) are not necessary.

	m=1	**m=2**	**m=3**	**m=4**
n=0	0.002	0.035	0.452	0.999
n=1	0.008	0.128	0.872	NA
n=2	0.042	0.487	0.993	NA
n=3	0.207	0.900	0.998	NA
n=4	0.595	0.999	0.999	NA

Table 6.1: The coverage of different LightFlood schemes.

	m=1	**m=2**	**m=3**	**m=4**
n=0	0	72	2470	79176
n=1	14	362	13784	NA
n=2	92	2574	28434	NA
n=3	622	12665	32169	NA
n=4	4008	26743	32450	NA

Table 6.2: The redundancy of different LightFlood schemes.

In the improved design of CommentCast, push protocol doesn't have to cover the entire overlay, because it works with a synchronize protocol based on a passive gossiping. Therefore we choose 85% as our target coverage. There are 7 pairs of parameters achieving higher than 85% coverage. After considering redundancy, (3, 1) and (2, 3) are preferred, because their redundancies are only a little bit higher than 1 redundant message per peer.

Churn resilience is also an important factor. We illustrate the reason of why the LightFlood is resilient to churn in 5.4.5 and conducted a simulation of LightFlood with high churn rate. Here we will redo the simulation to compare the churn resilience of (3, 1) and (2, 3) schemes.

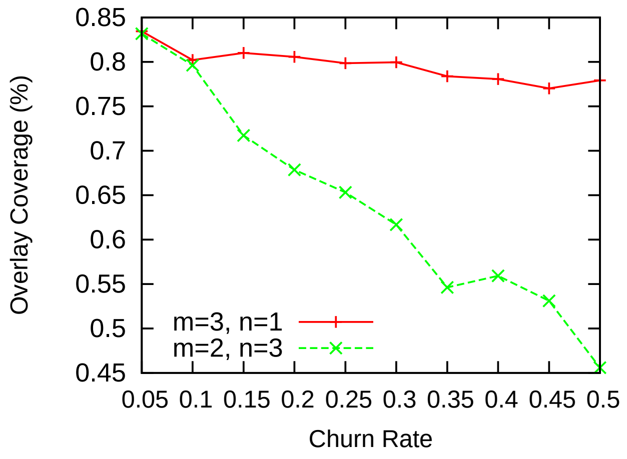


Figure 6.2: The coverage of (3, 1) and (2, 3) LightFlood under churn. The horizontal axis is the churn rate ranging from 0.05 to 0.5. It presents the percent of the peers leaving the system. The vertical axis is the overlay coverage of the two schemes under the churn rate.

Figure 6.2 shows the overlay coverage of the two schemes under churn rate from 0.05 to 0.5. The churn rate represents the percent of peers leaving the system. We can see that the (3, 1) scheme is much better than (2, 3) under high churn rate. Although (2, 3) is a little better in terms of the coverage and redundancy, it is not reliable in a real deployment.

Based on the analysis above, (3, 1) is chosen as our target system. At the same time, we are also interested in (3,4) LightFlood, which is a full coverage scheme, and (2, 2) LightFlood which covers 50% of the peers with a low amount of redundancy.

6.2 Simulation Results

The results of the simulation is shown in table 6.3. We only calculated the upload traffic. From the view of network, one peer uploading means downloading of another peer, so the total download traffic equals the total upload traffic. The speed of the algorithm is not given, because the speed of dissemination depends on the scheme of LightFlood. The speed equals the sum of m and n. (3, 1) means the the dissemination will be finished after 4 communication cycles. The speed is measured only in communication cycles, because the precise time of each TCP transmission is hard to be estimated.

LightFlood Scheme	(3, 1)	(2, 2)	(3, 4)
Number of comments	7,806	7,806	7,806
Sum of comment size (Kbyte)	466	466	466
Average data coverage	90.36%	47.96%	99.42%
Total traffic (MByte)	9,798	3,317	17,545
Valid traffic	41.97%	65.79%	25.79%
Redundant messages	97,791,820	19,549,558	223,308,976
Redundant traffic	58.03%	34.36%	74.20%

Table 6.3: Results of the simulation under real workload for 10 hours.

(3, 1) LightFlood achieves 90% data coverage on average for 10,000 peers with 9,798MBytes, in which 58% traffic is wasted in retransmission. It means every peer uploads 100 KByte and downloads 100 KBytes in one hour. The bandwidth efficiency is reasonable for a fully distributed comment system on average.

LightFlood (3, 4) is also an acceptable scheme which consumes twice the bandwidth of the (3, 1) scheme. It can achieve nearly full data coverage. However, the redundant traffic takes 70% of total traffic. On the other hand, (2, 2) LightFlood achieved 48% data coverage. The 70% of bandwidth is consumed in valid transmissions.

Spreading traffic evenly among peers is also very important to a system. Figure 6.3 shows the CDF of the network traffic of each peer with some important statistical values in Table 6.4.

Min.	**1st Qu.**	**Median**	**Mean**	**3rd Qu.**	**Max.**
3.90	24.40	39.03	100.33	85.56	3352.07

Table 6.4: Statistical values of the network traffic of each peer for one hour.

We can see that the network traffic is negligible for most peers, but some peers consume much more bandwidth than others. The max individual traffic went up to 3,352 KBytes in one hour. Given the recent network infrastructures, 3M Bytes per hour is still acceptable. There are two reasons behind the uneven workload. The first reason is the tree structure of LightFlood. Comparing with leaf peers, root peers forward more messages. The second reason is user behavior. We have described the Pareto principle of user commenting in 4.3, which means some users are very talkative in online discussions. We are only interested in individual traffic. Balancing the network traffic is not part of this thesis.

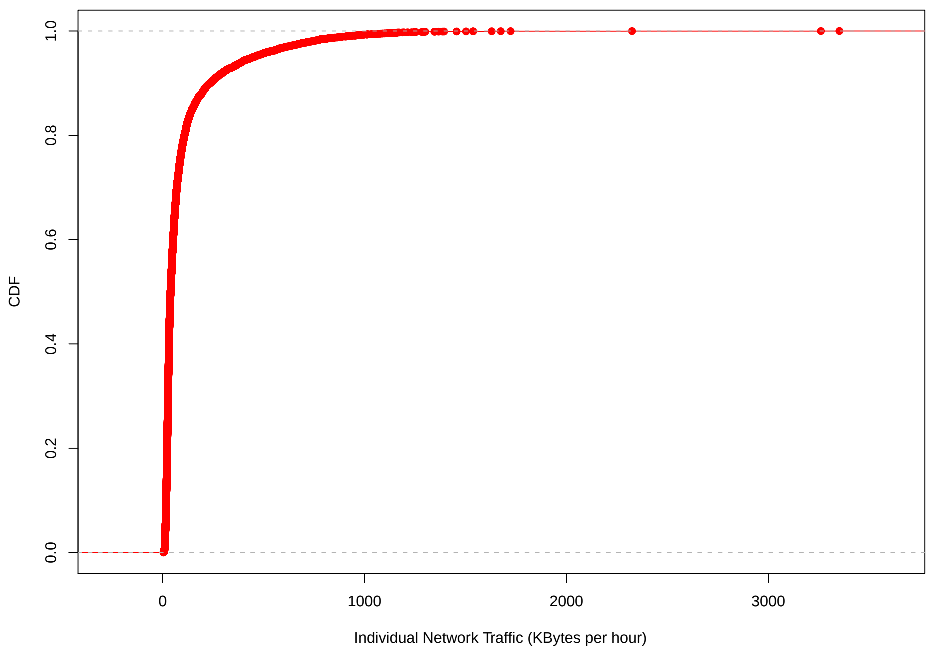


Figure 6.3: Cumulative distributed function of network traffic of each peer for 1 hour. The horizontal axis is the network traffic in KBytes.

Chapter 7

Conclusions and Future Work

This chapter revisits the research questions in Chapter 2 and summarizes our empirical study and the simulation under real-world workload. With the results and conclusions, we go back to the improved design of CommentCast and discuss the future work on CommentCast.

Conclusions

This thesis focuses on the improvement of CommentCast. As a commenting system, the original dissemination speed of CommentCast is too low because of the inflexible structure of CommentCast. Exchanging of CommentCast message which contains 30 comments is executed once every 4 hours. If we shorten the cycle of CommentCast to speed up disseminating, it will lead to a a huge amount of retransmissions, because our data shows there is about one comment injected in the system per one minute. Therefore, we propose a new structure that divides the functionality of original CommentCast into data dissemination and data synchronization. Data dissemination is achieved by a push protocol with a very high speed. It is supposed to cover not 100% of the peers, but most of them in the network. Furthermore, a push protocol can not take care of offline peers. The synchronization protocol is used to extract missed comments in the push protocol and retrieve new comment when offline peers log on.

After the structure improvements, we studied the performance of basic proactive gossiping and flooding as dissemination speed and redundancy, using experiments under Tribler environment. We also applied LightFlood to Tribler and evaluated the performance of LightFlood by comparing it with gossiping and flooding. The results show that LightFlood has a comparable speed with flooding, which is much faster than gossiping, especially in large scale networks. As the same time, the redundancy generated by LightFlood is much less than the one generated by flooding. LightFlood is even more cost effective than the basic proactive gossiping in achieving a high overlay coverage.

Finally, we conducted an intensive study on the performance of different schemes of LightFlood. We found LightFlood (3, 1) to be the best choice for our target system. Based on the real-world workload crawled from Verycd.com, LightFlood achieved 90.36% average data coverage in a network with 10,000 peers. The redundant traffic takes around 50% of the total traffic compared with a centralized commenting system. The data is disseminated within 4 cycles. On average, every peer spends 100 Kbyte per hour on CommentCast. Our simulation also shows that LightFlood is very resilient to churn. Under extreme churn, LightFlood retains its high performance.

Future Work

LightFlood is a very fast, cost-effective and reliable algorithm. The bandwidth consumption is realistic, such that it can be deployed on real system. However, it also has its defects. The most significant defect is that workload is not well balanced among peers even though the overall bandwidth consumption is small. Figure 6.3 shows the busiest peer should spend a around 3 MBytes per hour on CommentCast. We should prevent the situation that some busy peers spend too much bandwidth on CommentCast. A technique to balance the workload across peers is required.

In this improved design, the synchronization protocol is also plays a critical part. It is responsible for retrieving missing comments, churn repair for offline peers when they log on and bootstrapping for new users. Coordinating the balance of overlay coverage of the push protocol and synchronization protocol also needs a lot of work. Because the speed and cost of the push protocol is affected by the overlay coverage directly, we suppose finding the optimal balance is important to CommentCast.

Last but not least, we also suggested the idea of a searching protocol and the concept of taste-based local database of comments in 3.2.3. The peers don't have to collect and keep all comments in the system. Focusing on a subset based on user taste would be enough for a user. This would take a lot work of data mining and clustering, but we can expect a significant improvement from implementing the relevant design.

Bibliography

[1] S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah. Randomized gossip algorithms. *IEEE Transactions on Information Theory*, Volume 52 Issue 6, June 2006.

[2] Y.S. Chen, P.P. Chong, and M.Y. Tong. Mathematical and computer modeling of the pareto principle, November 1993.

[3] X. Cheng, C. Dale, and J.C. Liu. Statistics and social network of youtube videos, June 2006.

[4] F. Chung and L.Y. Lu. Connected components in random graphs with given expected degree sequence, August 2002. Annals of Combinatorics.

[5] B. Cohen. The bittorrent protocol specification, Jan 2008. http://www.bittorrent.org/beps/bep_0003.html.

[6] R. Delaviz. Swarm-based reputation consensus, 2010. https://www. tribler.org/trac/wiki/SwarmBasedReputationConsensus.

[7] D. Ferrari. Client requirements for real-time communication services. *IEEE Communications Magazine*, pages page 65–72, November 1990.

[8] A. J. Ganesh, A. M. Kermarrec, and L. Massoulie. Peer-to-peer membership management for gossip-based protocols. *IEEE Transactions on Computers*, Volume 52 Issue 2, February 2003.

[9] P. Gill, M. Arlitt, Z.P. Li, and A. Mahanti. Youtube traffic characterization: A view from the edge, 2007. Proceedings of the 7th ACM SIGCOMM conference on Internet measurement (IMC ’07).

[10] X.J. Hei, C. Liang, J. Liang, Y. Liu, and K.W. Ross. A measurement study of a large-scale p2p iptv system, Dec. 2007.

[11] M. Jelasity and O. Babaoglu. T-man: Gossip-based overlay topology management. *Engineering Self-Organising Systems Third International Workshop*, Volume 3910, 2006.

[12] M. Jelasity, A. Montersor, and O. Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Transactions on Computer Systems*, volume 23(No. 3):pages 219–252, August 2005.

[13] M. Jelasity, S. Voulgaris, R. Guerraoui, A. M. Kermarrec, and M. van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems (TOCS)*, Volume 25 Issue 3, August 2007.

[14] S. Jiang, L. Guo, and X.D. Zhang. Lightflood: an efficient flooding scheme for file search in unstructured peer-topeer system. In *International Conference on Parallel Processing (ICPP’03)*, 2003.

[15] T. Karagiannis, A. Broido, N. Brownlee, K.C. Claffy, and M. Faloutsos. Is p2p dying or just hiding?, January 2005.

[16] Y. Kulbak and D. Bickson. The emule protocol specication, January 2005.

[17] L. Lamport. Paxos made simple, November 2001.

[18] Harry C. Li, A. Clement, E. L. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin, editors. *BAR gossip*. OSDI '06 Proceedings of the 7th symposium on Operating systems design and implementation, USENIX Association Berkeley, 2006.

[19] G. Logiotatidis. Splash: data synchronization in unmanaged, untrusted peer-to-peer networks. MSc thesis, Delft University of Technology, August 2010.

[20] E. K. Lua, J. Crowcrof, and M. Pias. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Magazine*, volume7(No.2):page 72–93, 2005.

[21] M. Meulpolder, J.A. Pouwelse, D.H.J. Epema, and H.J. Sips. Bartercast: Fully distributed sharing-ratio enforcement in bittorrent. Technical Report PDS-2008-002, Delft University of Technology, 2008.

[22] P. V. Mieghem. *Data Communications Networking*, chapter Peer-to-peer network, page 343. Techne Press, 2006.

[23] M. E. J. Newma. Random graphs as models of networks, Febuary 2010.

[24] A. Papadimitriou and A. Delis, editors. *Flash Data Dissemination in Unstructured Peer-to-Peer Networks*. ICPP '08 Proceedings of the 2008 37th International Conference on Parallel Processing, 2008.

[25] J.A. Pouwelse, P. Garbacki, D.H.J. Epema, and H.J. Sips. The bittorrent p2p file-sharing system: Measurements and analysis, June 2005.

[26] J. A. Pouwelse, P. Garbacki, A. Bakker, A. Iosup, D. H. J. Epema, M. Reinders, and H. J. Sips. Tribler: a social-based peer-to-peer system, January 2007.

[27] J. A. Pouwelse, J. Yang, M. Meulpolder, D. H. J. Epema, and H. J. Sips. Buddycast: An operational peer-to-peer epidemic protocol stack. Technical Report PDS-2008-005, Delft University of Technology, 2008.

[28] A. Qayyum, L. Viennot, and A. Laouiti, editors. *Multipoint Relaying for Flooding Broadcast Messages in Mobile Wireless Networks*. 35th Hawaii International Conference on System Sciences, 2002.

[29] R. Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service, 1998. the IFIP International Conference on Distributed Systems Platforms and Open Distributed.

[30] D. Shah. Gossip algorithms. *Foundations and Trends in Networking*, Volume 3(1), 2008.

[31] D. Stutzbach and R. Rejaie. Understanding churn in peer-to-peer networks, 2006. IMC '06 Proceedings of the 6th ACM SIGCOMM conference on Internet measurement.

[32] S. A. Theotokis and D. Spinellis. A survey survey of peer-to-peer content distribution technologies. *ACM Computing Surveys (CSUR)*, volume 36(Issue 4), December 2004.

[33] Tribler protocol specification, January 2009.

[34] S. Verm and W. T. Ooi. Controlling gossip protocol infection pattern using adaptive fanout, June 2005. Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE International Conference.

[35] S. Voulgaris and M. V. Steen, editors. *Epidemic-Style Management of Semantic Overlays for Content-Based Searching*. Euro-Par 2005 Parallel Processing, USENIX Association Berkeley, 2006.

[36] H.L. Yu, D.D. Zheng, B.Y. Zhao, and W.M. Zheng. Understanding user behavior in large-scale video-on-demand systems, 2006. Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006.