

Splash: data synchronization in unmanaged, untrusted peer-to-peer networks

Giorgos Logiotatidis



Splash: data synchronization in unmanaged, untrusted 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

Giorgos Logiotatidis

August 18, 2010

Author

Giorgos Logiotatidis (glogiotatidis@sealabs.net)

Title

Splash: data synchronization in unmanaged, untrusted peer-to-peer networks

MSc presentation

August 18, 2010

Graduation Committee

prof. dr. ir. H. J. Sips (chair) Faculty EEMCS, Delft University of Technology

dr. ir. J. A. Pouwelse (advisor) Faculty EEMCS, Delft University of Technology

dr. ir. G. Gaydadjiev Faculty EEMCS, Delft University of Technology

dr. ir. G. P. Halkes Faculty EEMCS, Delft University of Technology

Abstract

Peer-to-peer networks rely on gossip algorithms to spread information about the peer activity and the network status. State-of-the-art gossip algorithms are not sufficient to spread the information widely, as the size and the complexity of the unmanaged networks grow. They suffer from high bandwidth utilization and lack mechanisms to verify the validity of the transferred information.

We set the following questions for research: How can we achieve high coverage ratio for all the peers in the network while using as little bandwidth as possible? How can such a mechanism scale to millions of nodes? How can we reduce the effects of high churn rates? And finally, how we secure that the data transmitted are genuine?

We develop Splash, a data synchronization algorithm with emphasis on high data coverage and efficient bandwidth usage. Our solution is based on Bloom filters and allows both partial and full synchronizations while avoiding the transmission of duplicate information.

We propose two schemes to synchronize current and historical data between peers, over unmanaged and untrusted peer-to-peer networks. We use a SyncLog to keep the state of the synchronization algorithm to a minimum. For scalability we categorize data based on their origin, as local or global, and we investigate the use of different synchronization policies. Finally we discuss mechanisms that validate data to prevent the spread to invalid information. Through a series of experiments we present the effectiveness of this solution in networks with different sizes and churn effects.

In simulation Splash achieves *95%* data coverage for all the peers in the network, which is *5 times more* compared to BarterCast. Additionally Splash uses *8.6 times less* bandwidth compared to BarterCast to achieve the same data coverage.

Preface

This document describes my thesis research as part of the Master of Science in Computer Engineering. The research was performed in the Parallel and Distributed Systems Group of the Faculty of Electrical Engineering, Mathematics and Computer Science of Delft University of Technology.

I would like to thank my advisor dr. ir. Johan Pouwelse for his support and inspiration throughout this project. His enthusiasm and vision got me involved with Tribler and motivated me during my work. I would also like to thank dr. ir. G.P. Halkes for reviewing my work and providing insightful feedback and the whole Tribler group for the nice working environment.

I am especially grateful to my parents and my sister for supporting all my choices. This international experience would not be possible without them. Last but not least I would like to thank Ioanna for 'being there' and all my friends in the Netherlands especially Dimitris, Antonis, Loucas and Effrosyni for the good times we had together.

Finally I would like to thank prof. dr. ir. H.J. Sips for chairing the examination committee and dr. ir. G. Gaydadjiev for participating in the examination committee.

Giorgos Logiotatidis

Delft, The Netherlands
August 18, 2010

Contents

Preface	v
1 Introduction	1
1.1 BitTorrent	2
1.1.1 BitTorrent Technology	2
1.1.2 BitTorrent Communities	3
1.2 Tribler	3
1.3 Contributions	4
1.4 Document Layout	4
2 Problem Description	7
2.1 Current Tribler Technology	7
2.2 Research Questions	9
3 Prior Work and Known Techniques	11
3.1 Bloom Filter	11
3.2 Set Reconciliation	13
3.3 Approximate Set Reconciliation	14
3.4 Optimized Union of Non-disjoint Distributed Data Sets	15
3.5 Efficient Reconciliation of Unordered Databases	16
4 Design and Implementation of Splash	17
4.1 Basic Bloom Filter Algorithm	17
4.2 SyncLog	20
4.3 Live and History Algorithms	22
4.4 Scalability	25
4.5 Security	25
4.6 Multi-Hop Gossiping	26
4.7 Compression	27
5 Experimental Setup	29
5.1 Simulator	29
5.2 Memory Optimized Simulator	31
5.3 Dataset	33

6	Experimental Results	35
6.1	Basic Experiments	35
6.2	Detailed Splash Experiments	37
6.3	Network Size Experiments	39
6.4	Churn Experiments	41
7	Conclusions and Future Work	47
7.1	Conclusions	47
7.2	Future work	48

Chapter 1

Introduction

A peer-to-peer architecture is any distributed network architecture composed of participants that share portion of their resources directly to the rest peers of the network, without the need for a central server. In a peer-to-peer network all participants are both clients and servers at the same time, as opposed to a client-server network where nodes have distinct roles. The shared resource can be either bandwidth, storage space or computing power in any combination.

Peer-to-peer networks often implement application level overlays that provide additional functionality such as message routing and resource searching. Overlays can be divided into two categories, *unstructured* and *structured*. In the former connections between peers are established in an arbitrary way, usually based on peer responsiveness and discovery through other peers. Peers use flooding to locate resources, namely they sent a message to all their neighbors that is sequentially forwarded to the neighbors of the neighbors and so on. Flooding is resource demanding and peers may fail to locate a resource.

On the other hand structured overlays use sophisticated indexing mechanisms, such as the distributed hash table (DHT) [1], to create connections with others and to route messages efficiently. Compared to the unstructured overlay this is more effective, less messages are transmitted and even rare resources can be located. However structured overlays are more vulnerable to churn and can be practically destroyed when they is no security mechanism, by injecting invalid information in the DHT.

Peer-to-peer networks are resource scalable by design. As nodes arrive and the demand for resources is increased, the total capacity of the network also increases. The system is more robust compared to a client-server network architecture because there is no centralized node that can turn into a single point of failure.

Today peer-to-peer networks are used for content delivery, for sharing super computing power during scientific experiments, for IP communications and most commonly for file sharing. Peer-to-peer file sharing was revolutionized in early 2000 by Napster [2] and now a large amount of file sharing protocols and networks exist, such as BitTorrent.

1.1 BitTorrent

BitTorrent is hybrid peer-to-peer and client-server protocol which is widely used to distribute files. The downloading and uploading of the actual data is done in peer-to-peer manner, whereas peer discovery is centralized and relies on a tracker. It was first introduced in 2001 by Bram Cohen [3]. BitTorrent is currently the most used peer-to-peer protocol for file exchange and it is estimated that it is responsible for about 50% of the Internet traffic [4]. Numerous compatible software clients exist, both proprietary and open source, that can be used to download millions of files.

1.1.1 BitTorrent Technology

BitTorrent is a simple protocol. A user who wants to share a file or a collection of files creates a single *.torrent* file using the appropriate software. The *.torrent* file contains information about the files to be shared, such as their size and their hash values. It additionally includes one or more user-defined IP addresses of trackers. The *tracker* is a server used by peers to locate other peers of the network. Once the *.torrent* is created the user has to distribute it and connect to the tracker as the first *seeder* of the torrent file. A peer is defined as a seeder when it owns 100% of the data of a *.torrent*.

Users usually post *.torrent* files to public *web portals*, especially designed to serve that type of file. Other users get *.torrent* files from the web portal and start downloading their contents from the peer-to-peer network using the appropriate software. Peers locate other peers through the tracker. The new peer is called *leecher* of a torrent until it completes its downloading and transforms to *seeder*.

The collection of *seeders* and *leechers* of the same torrent form a *swarm*. To encourage fair trading of information BitTorrent incorporates a *tit-for-tat* scheme, according to which peers favor the ones that have uploaded data to them. This policy improves the overall speed of the swarm, by creating incentives for users to cooperate. However it often develops suboptimal situations, for example when a peer first join the swarm and has no data to exchange. To counter these effects the BitTorrent protocol uses a mechanism called *optimistic unchoking* that commits peers to use part of their upload capacity to help random peers in the swarm.

Later developments in the BitTorrent protocol introduced more advanced technologies such as the *Distributed Hash Table* (DHT) [1] to provide decentralized tracking and the *Peer EXchange* (PEX) [5] protocol to exchange peer information.

1.1.2 BitTorrent Communities

A large number of websites provide indexing and searching for torrent files¹, a functionality which is absent from the BitTorrent protocol.

The users who search and download from specific websites formed *communities*. Several of these websites were enhanced to provide user management abilities to administrators, which lead to *private communities* also known as *Darknets* [6]. Users need special access authorization to join them, that can be obtained either by simple form registration, invite code or directly from the community administrators.

As BitTorrent gains popularity the websites continue to grow adding more and more features for the users, such as the ability to submit descriptions for the torrents they upload, or comments for the torrents uploaded by others. To ensure the quality of the provided service, elite members of the community obtain *moderation privileges* that allow them to remove or star torrents. Respectively the rest of the users obtain *voting rights* to rate torrents as good or bad and indirectly determine the reputation of the torrent uploader.

To achieve better member experience and higher speeds private communities augmented the user accounts with a metric called *sharing ratio*. The *sharing ratio* represents the cooperation of the user in the community and not only a specific swarm. It is defined as the percentage of the overall upload traffic over the overall download traffic of the user. Communities often require that the users have a sharing ratio of over 0.8. Users who fail to follow this rule receive warnings or penalties and eventually are forced out of the community, by denying tracker access.

Sharing ratio builds in an indirect way memory into the stateless tit-for-tat scheme and creates incentives for users to seed after they have downloaded the entire file and to share a file in the first instance. The use of this metric increases the overall download speeds in private communities as described in [7] and [8].

To summarize the community websites provide fundamental services of the BitTorrent network and create incentives for the users to join and expand the list of available files. In combination with the tracker software the overall client-server based communication related to BitTorrent is increasing, despite the latest progress in decentralizing the protocol.

1.2 Tribler

Tribler² is a peer-to-peer application designed for the next generation peer-to-peer networks. Tribler incorporates a number of technologies to provide a *fully decentralized* solution for file sharing and community management. It provides indexing and searching for torrents, fully distributed tracking, content recommendation

¹The Pirate Bay (<http://www.thepiratebay.org>) and isoHunt (<http://www.isohunt.com>) are two of the numerous available websites that serve as index and search engines of torrent files.

²Tribler's official website is located at <http://www.tribler.org>.

based on previous choices, file moderation, RSS feeds, user reputation which defines his sharing ratio as good or poor and other interesting features.

The Tribler team created a family of protocols that form an *overlay network* and provide the mentioned functionality: *BarterCast* is responsible for disseminating information on peer download and upload activity, *VoteCast* implements the moderation and voting system, *ChannelCast* for creating channels and so on. Each protocol transmits meta-data essential for its operation over the overlay network using a *gossip* protocol.

1.3 Contributions

In this thesis we present the design and implementation of a novel data synchronization mechanism, called *Splash*, which can be used to exchange meta-data in peer-to-peer networks. We focused on increasing peer coverage, namely the percentage of overall available information in the network that a peer knows, in an effective and bandwidth efficient way. *Splash* delivers high coverage results even with high peer churn rate and can be improved with a security mechanism to withstand fake information injection from malicious peers.

The contributions of this thesis are the following:

- We research data distribution and set reconciliation solutions and their design decisions.
- We propose a novel data synchronization mechanism that can deliver large amount of meta-data, utilizing partial or full synchronizations according to the needs of the application.
- We simulate and calibrate the parameters of the mechanism to fit the Tribler network.
- We suggest extensions of the proposed mechanism, to enhance the results of special conditions such as long absence from the network.

1.4 Document Layout

The remainder of this document is organized as follows: In Chapter 2 we describe the current data synchronization mechanism employed in Tribler, its performance, and the research questions that this work was based upon. Chapter 3 focuses on prior work and known data synchronization techniques. We select methods designed for database synchronization with focus on bandwidth utilization and a data structure named *Bloom Filter*, a well known and used method to compress your inventory of information.

The basic algorithm of *Splash* along with its improvements and expansions are described in Chapter 4. Chapter 5 contains information about the simulation of *Splash* and *BuddyCast* and the dataset used. The simulation results and comparison

graphs are included in Chapter 6 and finally in Chapter 7 we discuss the results of this work, we compare it against the current implementation and prospect the future possibilities.

Chapter 2

Problem Description

In this chapter we discuss in detail the gossip algorithm currently used in Tribler. We lay down the research questions to be answered by this thesis, after examining performance results and implementation limitations of the mechanism, as well as the goals of Tribler.

2.1 Current Tribler Technology

Tribler is moving the activity of the communities away from the web and into the peer-to-peer network itself. A collection of different components is used to distribute, collect and process information of the network, from which useful results can be extracted.

One of the main reasons private communities are so successful in the BitTorrent world is the sharing ratio mechanism. The mechanism creates incentives for peers to seed more, which results to higher download speeds for all. BarterCast is a fully distributed equivalent mechanism that is currently deployed in Tribler.

BarterCast is a protocol for *reputation management*. In BarterCast real time upload and download statistics are broadcasted to other peers with a gossip protocol. This information can be instantly used to calculate the reputation of a peer, i.e. how well or bad it has treated others in the same network. Since there is no central authority to verify that a peer is not lying, like for example a website, BarterCast employs the MaxFlow [9] algorithm to normalize received information. BarterCast creates incentives for the users improve their cooperation in the network, limits free riders and increases the overall download speeds.

For every download or upload that takes place between two Tribler peers a new BarterCast record is created and saved in their local database, called *MegaCache*. Each record contains the unique identifies for the peers called *PermIDs*, the size of the uploaded and downloaded data and finally the date and time that the transaction took place. Records from the MegaCache are used to construct BarterCast messages.

#	PermID	PermID	Data Uploaded	Data Downloaded	Timestamp
1	MFIwEze...JUNhs	MFIZIz...ddfRa	1495297	10507114	1245456594
2	MFIwEze...JUNhs	MFIZIz...e3rFs	478956	10507114	1245468791
⋮	⋮	⋮	⋮	⋮	⋮
20	MFIwEze...JUNhs	MFIZIz...koJHs	85805	1258	1245862188

Table 2.1: Example records in a BarterCast message

Peers transmit BarterCast messages to inform others about their network experience. A BarterCast message is sent periodically, every 15 seconds, or in response to a received BarterCast message. The receiver is selected either randomly or from a list of *taste buddies* which is generated using a *similarity function* [10]. To avoid contacting the same peer too often and therefore transmit duplicate data, a peer is contacted at most once per cycle of 4 hours.

Each message contains the last 10 transactions of the peer and 10 transactions with the largest download values. Peers include only information that are positive to be true, namely transactions that they were part of, due to the lack of a security mechanism to validate message authenticity. Table 2.1 contains example records contained in a BarterCast message.

BarterCast can be classified as a *gossip exchange* protocol since every peer replies to a BarterCast message with another message. A gossip exchange protocol effectively doubles the number of messages exchanged in the network. BarterCast is also a *stateless* protocol, meaning that peers do not hold knowledge about what the other peers know or which records were transmitted earlier.

These attributes of BarterCast create a network of peers that exchange a lot of information however most of it is likely duplicate. Duplicate records are not only useless but also consume a lot of bandwidth and CPU resources, to be transmitted and processed accordingly.

We observe, from analysis of the current Tribler network [11], that the big majority of the peers hold only a fraction of the available records, that is to say that almost 99% of the peer possess less than 20% of the overall records. Only 0.5% of the peers have a good view of the activity in the network, i.e. more than 80% coverage, and therefore can make good decisions. The average low coverage delivers a strong hit on the usefulness and accuracy of the *reputation* function and overall in the network operations.

We can additionally draw conclusions on the churn rate of the peers, in other words how much time a peer stays on-line. The vast majority of the peers stay on-line for a very short time, practically for as long as a download lasts, behavior that further throttles the data spreading.

The dataset used to extract this information was collected in the period of June to September 2009 by the Tribler team and it contains more than 1,3 million records and 9,000 unique Tribler PermIDs. The same dataset is used to create the graph of Figure 2.1.

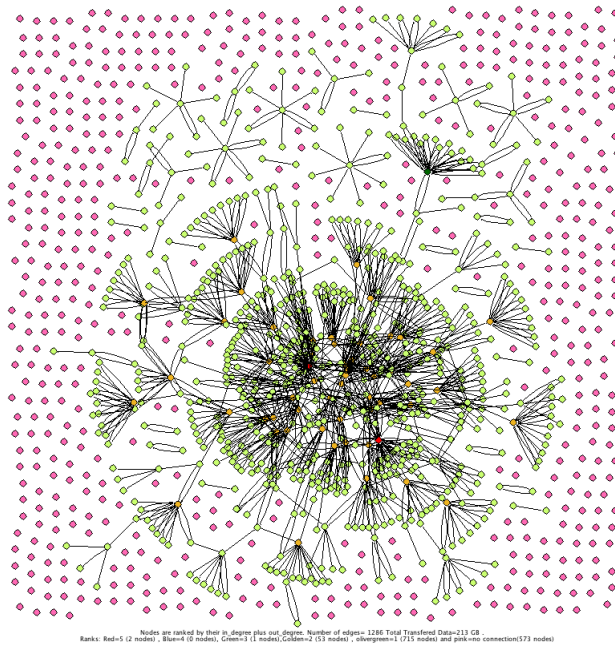


Figure 2.1: BarterCast graph: Representation of the network activity based on BarterCast records. Each node represents a peer and each edge a BarterCast transaction. Source: [11]

BarterCast protocol is not directly sending data to the network but instead its payload is attached to BuddyCast messages. Similarly other protocols such as the VoteCast and ChannelCast, piggyback on BuddyCast, thus equivalent dissemination results can be expected from all components of Tribler. BuddyCast [12] is a peer-to-peer epidemic protocol stack, responsible for spreading information in the Tribler network.

We use BarterCast as a case study throughout this thesis since it is such an important part of Tribler and it is responsible for a large part of the meta-data moved in the Tribler network.

2.2 Research Questions

Within the Tribler context we study a new data synchronization mechanism to overcome the problems and limitations of the current version. We investigate a flexible design that can introduce new possibilities and allow the further development of the Tribler network.

We look into a new mechanism that can answer the following research questions:

How can we achieve high coverage ratio for all the peers in the network while using as little bandwidth as possible? We want to develop a solution that will

move the majority of the peers from the 20% information coverage barrier up to 90% or more.

How can such a mechanism scale to millions of nodes? The advantage of peer-to-peer systems over the client-server model is their scalability. Tribler may have just a few thousand peers at the moment but the next generation network should be able to scale to millions. How can we deal with the ever increasing need for network bandwidth and disk space to move and store all the available network information?

How can we reduce the effects of high churn rates? Peers join and leave the network in high rates. This prevents network data synchronize and results in nodes with not enough knowledge.

How we secure that the data transmitted are genuine? The lack of central authority in peer-to-peer networks makes it easier for malicious peers to inject fake data into the network. Any solution must accommodate a set of measures to prevent spreading invalid data to other peers.

Chapter 3

Prior Work and Known Techniques

The synchronization of data in distributed environments has always been challenging research topic. Multiple algorithms have been proposed and implemented aiming to achieve minimum bandwidth usage, full data synchronization, minimum computational complexity or a combination of those. We investigate data structures which can be used to exchange information about the data sets and on set reconciliation techniques that efficiently combine databases, with focus on flexible designs with efficient bandwidth management.

3.1 Bloom Filter

The Bloom filter is a probabilistic data structure formulated by Burton Howard Bloom in the 1970's [13]. The structure is used to determine whether an element is a member of the set represented by the structure, in a fast and space efficient way. Space efficiency comes with the price of false answers. Bloom filters can trigger false positives, hence suggest that an element is member of the filter while it is not. On the other hand false negatives are not possible. In the original Bloom filter design it is possible only to append elements and impossible to delete. Each and every addition in the filter increases the rate of false positives.

A Bloom filter is an array of m bits, where k is the filter size. The filter bits are initialized to *zero*. A collection of k independent hash functions operate on $\frac{m}{k}$ positions of the array from which for every input one of them gets activated, with a uniform random distribution.

To *add* an element to the filter, first its value has to be hashed using the k functions and then the resulting array positions have to be set to *one*.

To *query* an element, first its value has to be hashed using the k functions and then all the resulting positions have to be checked against the array for their value. If all the positions have the value of *one* then the item belongs in the array, otherwise even if one position has the value of *zero* item does not belong in the array.

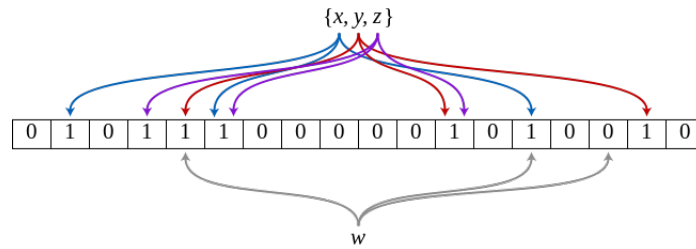


Figure 3.1: An example of a Bloom filter, representing the set x, y, z . The colored arrows show the positions in the bit array that each set element is mapped to. The element w is not in the set x, y, z , because it hashes to one bit-array position containing 0. For this figure, $m=18$ and $k=3$. Source [14]

The Bloom filter provides a compressed form of a set representation. Compared to other data structures for representing sets it uses less space to store the same number of elements. Moreover the use of a bit array and hash functions yield great speed both while adding and searching for elements, at the cost of calculating k hash functions and accessing k bits in an array.

On the contrary the small space requirement of the Bloom filter restricts the accuracy of the solution. Bloom filters are known to raise false positives, declaring that an element is a member of the set while it is not. Nevertheless the probability of a false positive, the *false positive rate* can be estimated precisely in a straightforward fashion, using the following formulas:

$$\left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k$$

which is almost equal to

$$\left(1 - e^{-\frac{kn}{m}}\right)^k$$

for k being the number of independent hash functions, m the size of the bit array and n the number of elements in the set. Knowing in advance the maximum number of elements that can be in the set and the number of hash function to be used, we can adjust the size m of the array to bound the error rate so that is acceptable for the application.

The attributes of the Bloom filter, namely the small size and the speed of addition and query, turns it into a good solution for a wide range of problems. The broad adaptation of the idea created vast amount of derivative works, which try either to solve the limitations of the original implementation or add new features.

The *Scalable Bloom Filters* by Almeida et al [15] overcome the problem of increasing error rate when adding more elements to a set, by introducing the use of a collection Bloom filters which adapts dynamically to the number of elements stored. Their key point is that they guarantee a maximum false positive rate.

The *Compressed Bloom Filters* by Mitzenmacher [16] introduce methods to reduce the number of bits required to communicate the Bloom filter among interested hosts, while maintaining the false positive rate sufficiently low. The reduced bandwidth usage comes at the cost of higher processing time for compression and decompression and more memory use at the hosts.

The *Counting Bloom Filters* [17] address the problem of element removal from the set. In the original Bloom Filter implementation is impossible to remove a single element from the filter by zero-ing the indexes produced by the hash function for this element, since the same indexes or part of them may represent the existence of other elements in the array as well. Proceeding with zero-ing indexes will introduce *false negative* errors and will eventually destroy the filter. Counting Bloom Filters suggest the replacement of bit array with an array of unsigned integers. On every addition the corresponding array element is increased by one and on every deletion is decreased by one, hence each position in the array represents how many items of the filter use it. Removal comes at the cost of larger space to store and transfer of the Bloom Filter due to the change of array type.

The above Bloom Filter variations are some of the literally tens available, since the Bloom Filters have been actively studied and extended for more than 40 years. We selected and studied the variations that have attributes which would help resolve our research questions.

The Bloom filters have been adopted to solve numerous problems in the database field and later in the networking field. It is known that the Bloom filters are used in Google's BigTable [18] implementation to reduce the disk look-ups for non-existent rows and columns, in Squid Web Proxy [17] software to improve the cache system, in hyphenation [13] and spelling systems [19].

Bloom filters have also been adopted lately in the field of networks, to build distributed cache system [17] and in peer-to-peer networks, such as the Vistabar [20] web browsing assistant.

As a result of the wide acceptance, a great number of Bloom filter implementations exists for many programming languages including C, C++, Java, Python and others.

3.2 Set Reconciliation

The Set Reconciliation scheme was developed by Minsky et al [21] as an algorithm to calculate the number of differences between datasets maintained by two hosts, with low computational and communication complexity. This scheme can synchronize hosts by sending one message in each direction of length $|A - B| + |B - A|$, hence essentially independent of $|A|$ and $|B|$. The messages exchanged contain a translation of the data in the set in a certain type of polynomial known as the *characteristic polynomial*.

Each host maintains a characteristic polynomial of the elements in its dataset and sampled values of it. In the event of a synchronization the two hosts exchange

the sampled values of their characteristic polynomials and sequentially they can calculate the set of differences by interpolating a rational function.

The algorithm was developed by Minsky et al as an answer to the high computation complexity required by prior work to calculate the set to differences. Other set reconciliation methods provide nearly optimal communication complexity, but the computational complexity is cubic in the number of differences. This version requires linear computational time while it maintains an acceptable communication complexity. Furthermore the algorithm provides exact set reconciliation and guaranties full synchronization of all entries between the communicating hosts.

On the other hand multiple rounds of communication are necessary for the algorithm to operate correctly. It is also demanded that the upper bound on the number of differences between the hosts is known. In the case of wrong upper bound estimation the number of rounds increases while the hosts calculate new values for it. Both these requirements can be addressed or may be of low importance in controlled networks but in an unmanaged peer-to-peer network, where nodes join and leave in unpredicted ways the number of rounds must be as low as possible. Moreover the nature and size of the network makes it really difficult easily calculate the upper bound the number of differences between hosts, thus increasing the number of rounds.

The algorithm is more efficient when the sets in question contain a large number of elements but only a few differences exist between them. If that is not the case Bloom filters and other synchronization methods can achieve faster results [22]. Finally the algorithm is based on complex mathematical functions, making it difficult to program, debug and maintain.

Two proof of concept implementations of Set Reconciliation exist, one from the original authors [23] and one from S. Agarwal et al [22][24][25]. The latter focused on implementing a real life application that synchronizes data between a PC and a PalmPilot.

3.3 Approximate Set Reconciliation

The *Fast Approximate Set Reconciliation* was designed by John Byers et al [26] as synchronization method for peer-to-peer networks. Primary focus of this algorithm is to synchronize as much data as possible between two hosts using only one message. Byers compromises the exact reconciliation provided by the algorithm presented in Section 3.2, with faster computation and less network communication.

To achieve the speed and efficiency Approximate Set Reconciliation utilizes all three of Bloom filters, Patricia tries and Merkle trees. Bloom filters are used to provide compact representations of the sets, Patricia tries for structured searching of subsets and Merkle trees to make comparisons of the subsets practical.

In more detail each host creates a Patricia trie its records. Each node of the trie holds a subset of the whole dataset. The construction of the the trie makes exact comparison of nodes in constant time impossible.

To provide constant time comparison, Merkle trees are build upon the Patricia tries. Merkle trees make use of hash functions to represent the nodes of the Patricia trie, in a way that they can be directly used in a set reconciliation algorithm. The advantage of using Merkle trees is that now we can perform constant time comparisons, with the risk of false positive due to hash collisions. Finally Bloom filters are used to compress the data and to provide fast transfer of Merkle trees between hosts.

Approximate Set Reconciliation utilizes Bloom filters, Patricia tries and Merkle trees, using each and every one of them for what they perform best. As a result this solution provides a fast way to determine the differences between two hosts. The use of only one message for the communication provides added value, because it is a key feature for an unmanaged peer-to-peer network with high churn rates.

However to host all the representations of the local set could be of a considerable size depending on its size .

On the other hand the adoption of many structures all in one algorithm for communicating differences between hosts can develop into a considerable issue independent of the application. The development of such application will be difficult to debug and maintain and therefore lead to insufficient implementations. Finally the use of hash functions and compact representation of the set may results in collisions which will prevent the full reconciliation of the hosts. Incomplete join of the sets may or may not be a complication depending on the application.

3.4 Optimized Union of Non-disjoint Distributed Data Sets

The Optimized Union of Non-disjoint Distributed Data Sets is the work of Dar et al [27] on server planning techniques. The goal of their research is to to develop efficient algorithms that avoid redundant data transmission of data while synchronizing from distributed databases. The common case in many distributed systems is that peers hold different sets of elements that partly overlap, therefore the same records are transmitted more than once, in the cost of network and time resources. This research provides algorithms for peers to co-operate and simultaneously transmit data to a receiver peer at minimal time.

The algorithm describes three functions, that are used from the peers acting as data providers to a receiver peer, with which peers communicate their datasets and decide the data flow towards the receiver. The algorithms works in a greedy manner, constantly checking the bandwidth usage of each peer and recalibrating it accordingly, to achieve optimal use of the download capacity of the receiver.

The algorithm was designed in the first place to avoid redundant data transmission and optimally exploit the network bandwidth capabilities. Theoretically one peer can get a full update on the element missing from the local set by performing only one action and at the top speed the network allows.

Server planning as described from this research will be more appropriate for a

network with trusted peers and low churn rates and not for an unmanaged network, where malicious peers may join and be part of the planning therefore easily injecting false data in the network or causing a performance downgrade or even a complete failure of the scheme.

Moreover the solution compromises on the accuracy to be bandwidth and time effective, which results in either hosts with incomplete databases or in duplicate transmission of records. As in other algorithms this is something to be questioned harmful or not depending on the application.

Finally to implement the suggested sophisticated ways of optimization prior knowledge of the bandwidth upload and download rates must be known and be guaranteed, which is not the case for the Tribler peer-to-peer network where many heterogeneous peers co-exist.

3.5 Efficient Reconciliation of Unordered Databases

The research paper “Efficient reconciliation of Unordered Databases” is a work of Trachtenberg and Minsky [28] who also worked together on “Set reconciliation” [21] described in Section 3.2. Although there are similarities between the two algorithms, they are independent.

The goal of this research is to determine the mutual difference of two databases with a minimum communication complexity. The proposed solution uses elementary symmetric polynomials to encode database records.

Two instances are analyzed by the authors, a client-server model and a more general peer-to-peer model. For the latter two simple divide and conquer reconciliation algorithms are provided.

The basic concept of the algorithm is that two peers recursively compare subsets of the datasets of the peers using hashes and communicate the elements that appear missing from each subset. Two solutions for peer-to-peer systems are provided by the authors which both follow an interactive divide and conquer design but differ in the method of splitting the data space.

The authors claim that their solution for database reconciliation is known to have practical computational complexity and almost optimal communication complexity. The peer-to-peer version described earlier is a simple algorithm which requires only one hash function to operate.

The algorithm it is designed for peer-to-peer systems nevertheless it is interactive and not single message. It involves multiple rounds, relevant to the number of differences between the databases of the peers, that for the peer-to-peer system we study is not a practical solution as we try to limit the rounds as much as possible. An additional disadvantage of the algorithm is that it targets networks where peers maintain databases that have at least half of the elements in common.

Chapter 4

Design and Implementation of Splash

Splash consists of a basic algorithm based on Bloom filters that can be used to transfer any kind of information. First we explain the basic implementation and then we extend and enhance it to limit the bandwidth usage, to enable partial synchronization and to provide scalability.

We discuss a security mechanism for record validation and consecutively, we introduce multi-hop gossiping which increases drastically the number of records for synchronization. Finally we explain the use of compression that further reduces the amount of transferred data.

4.1 Basic Bloom Filter Algorithm

The key point of the basic algorithm is to avoid duplicate information from being transmitted. Peers can maintain a database of previously synchronized data for each and every peer they meet with, however this is cannot be done a large scale peer-to-peer environment.

Instead we use Bloom filters (Section 3.1) to represent previously synchronized data. The filter is transferred first, at the start of each synchronization and then used to select records which need to be synchronized.

We choose Bloom filters because they have been studied and used for more than 40 years and we have solid knowledge about their workings and limitations. Multiple Bloom filter implementations exist, that cover a wide range of programming languages and offer highly optimized hash functions and data structure implementations. Using them we can design our data synchronization mechanism in a way that corresponds exactly to the needs of the project by taking advantage of the Bloom Filter flexibility.

Bloom filters offer a good balance between computational complexity and bandwidth usage. Prior work claims that Bloom filter implementations can be more efficient under specific circumstances [22], compared to more sophisticated solu-

tions, like Set Reconciliation (Section 3.2). Last but not least using Bloom filters we can construct a synchronization algorithm with low implementation complexity. It is always important to keep the code base clean and simple so it can be debugged and further extended, especially in a project like Tribler which escapes the academic environment and targets end users and open source developers.

The basic algorithm follows: Peer *B* creates a filter that contains a compressed representation of its database and gets transmitted to peer *A*. The latter checks which of its entries do not exist in *B*'s database and send these back. Peer *A*'s reply is targeted to Peer *B* and only contains information that it is not already known.

The basic algorithm can be inspected in Listing 1 and in the upper left graph of Figure 4.1.

Listing 1 Basic Bloom Filter Algorithm

1. Peer *B* sends Bloom filter to Peer *A*
 2. Peer *A* checks every record in its database against the received Bloom filter
 3. Peer *A* replies with the records that where not included in the Bloom filter
 4. Peer *B* saves the records and updates the Bloom filter
-

After a few synchronizations as shown in Figure 4.1 all peers hold the same information. Note that in all three synchronizations only the missing records are transferred. Especially in the third synchronization the return set is empty, since peer *C* already holds all the records of peer *A*, although this is the first time these two peers synchronize.

In this basic design every peer maintains a Bloom filter, that contains its entire database, which is transmitted at the start of every synchronization. For every new record in the database the Bloom filter has to be updated and at each sync it is transmitted to the other party.

Each record to be included in the Bloom filter must be unique to avoid false positives. Currently Tribler stores BarterCast records in the MegaCache, an sqlite database. A BarterCast record can be uniquely identified among the whole network by combining the PermIDs and the date and the time of the record creation. To include a record in the filter we hash a string containing all these values. Similar combinations can be found for other protocols as well, so this algorithm can be used to synchronize any type of data.

However the original Bloom filter implementation does not support deletion, therefore a variation like the Counting Bloom filters¹ must be used to delete expired records. Moreover because we do not know in advance the number of records the

¹Counting Bloom filters offer item deletion from the filter. An integer array is used to store the

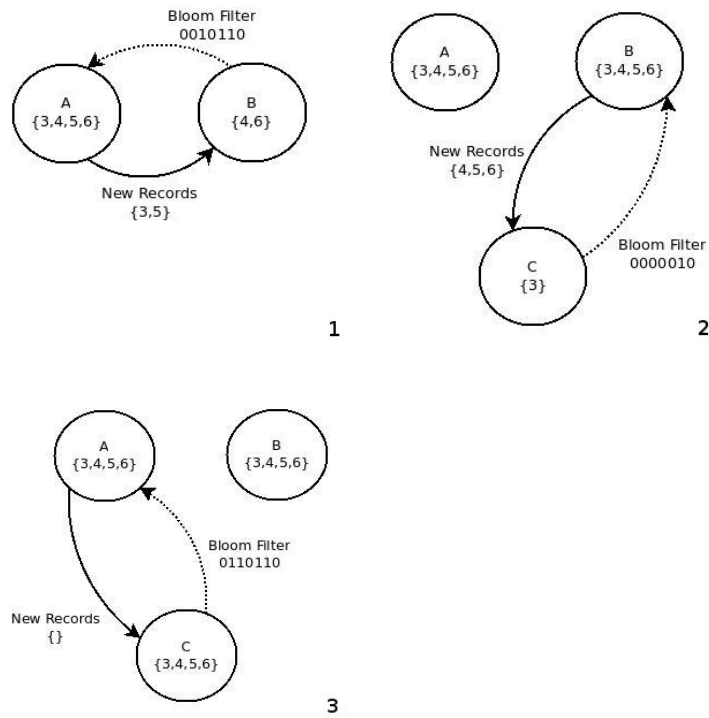


Figure 4.1: Example synchronizations using the basic Bloom filter algorithm. Only the missing records are transferred.

filter will hold, we risk to increase the false positive rate too much if we do not use a solution like Scalable Bloom Filters².

Note also that the Bloom filter represents the entire collection therefore it grows linearly along with the size of the collection and becomes expensive to transmit. The maintenance of one only filter imposes serious limitations and weaknesses that motivated us to work further on the design.

We extend the basic design and we suggest that peers create Bloom filters in *real-time*, during the synchronization with another peer. These filters will be for a single use and therefore there is no need for deletions. Also the number of elements to be included is known beforehand so we can control the error rate with high precision. Moreover if we intelligently select the records to be included we can produce even more efficient filters that are smaller in size and contain only information that is important for that specific transaction.

Real-time Bloom filters (Listing 2) come at the cost of higher CPU utilization,

filter instead of a bit array. For every insertion the value of array elements used is increased by one. The reverse procedure can be used for deletion. This extension comes at the cost of higher Bloom filter size. [17]

²Scalable Bloom Filters create a list of plain Bloom Filters which is expanding with new Bloom Filters when needed to limit the false positives to a predefined number. [15]

since we have to compute a new filter for each synchronization. We can limit this effect by using highly optimized hash functions and Bloom filter implementations, in combination with the limited number of elements in each filter.

Listing 2 Real-time Bloom Filter Algorithm

1. Peer A creates a specific Bloom filter for Peer B
 2. Peer A sends Bloom filter to Peer B
 3. Peer B checks every of record in its database against the received Bloom filter
 4. Peer B replies with the records that where not included in the Bloom filter
 5. Peer A saves the records and deletes the Bloom filter
-

4.2 SyncLog

We continue our work on the primary research question of reduced traffic by adding SyncLog to Splash. SyncLog is a database that contains PermID and timestamp pairs. Each pair is a *synchronization point*, namely it declares the last successful synchronization time with a specific PermID.

Using SyncLog we can intelligently select the records to be included in the real-time Bloom filter. Before generating the Bloom filter peer A will look-up when the last synchronization with peer B took place and include only records that where inserted in its local database after that timestamp. This timestamp is sent along with the Bloom filter to peer B so it can also constrain its activity accordingly.

All the records before the last synchronization point of two peers have been successfully synchronized in previous contacts. Therefore there is no need for peer A to include them in the Bloom filter nor for peer B to check them against the Bloom filter.

SyncLog bounds the number of records and therefore reduces the traffic required to transmit the filter and the processing time on both hosts. The algorithm is displayed in Listing 3.

Furthermore SyncLog enables *partial synchronization*. Partial synchronization means that two peers do not synchronize all their data in one communication, instead they set limits on the number of records synchronized. The limits relate either on the number of records, e.g. only 100 records per synchronization, or a time-frame, e.g. only records created in 2009.

Partial synchronization provides extra flexibility in the design which can prove useful in various network deployments. For example in the Tribler network where most of the peers suffer from limited upload bandwidth and high churn rates, an

upper bound on the number of records limits the use of the link and reduces the number of unfinished synchronizations due to peers going off-line.

Partial synchronization is implemented with SyncLog as following: Peer B lower bounds the return records with timestamp t_2 . Along with the records peer B sends a timestamp which represents the record creation date and time of the newest record in the returned set. Peer A saves in SyncLog the received timestamp instead of the current date and time. Later peer A can collect the rest of the records by using the SyncLog during a new synchronization with B .

Listing 3 SyncLog Algorithm using Real-time Bloom Filters

1. Peer A reads the last successful synchronization date and time with peer B from the SyncLog, t_1
 2. Peer A creates a Bloom filter with entries newer than t_1
 3. Peer A sends Bloom filter and timestamp t_1 to Peer B
 4. Peer B checks every of record with creation time newer than t_1 in its database against the received Bloom filter
 5. Peer B replies with (a fraction) the records that where not included in the Bloom filter and the timestamp t_2 of the record from the set with the newest local creation time.
 6. Peer A saves the records, deletes the Bloom filter and stores t_2 in SyncLog.
-

SyncLog records are stored in a database and can grow relatively large. The more new hosts one peer contacts and synchronizes with, the more entries are created in the SyncLog. This behavior can potentially create a large database table with PermID and timestamp pairs which will impact the access speed and consume a lot of disk space. We can prune records from the database to keep the size of the database within limits using a well selected deletion policy.

The use of the SyncLog does constrain the number of records under consideration in both sides of the line however it is not optimal for all situations. For example peer A synchronizes with peer B at t_1 . Between t_1 and t_2 where $t_2 > t_1$, both A and B synchronize large amounts of records with other peers. At t_2 peer A requests a synchronization from B and generates a Bloom filter with records newer than t_1 . The Bloom filter is relatively large because A obtained a lots of new records since the last synchronization and similarly the processing time for B is relatively long. However because both A and B have synchronized with many others it is highly probable that almost all records of B all already known by A . As a result both peers consume a lot of computing power and network bandwidth and the gain for A , if it exists, is small.

In a variation of the previous scenario, peer A synchronizes only a few new records between t_1 and t_2 . The Bloom filter is small and sent with low cost but B has to compare a large amount of its local records against the Bloom filter to find that only a few are included, thus consuming a lot of processing power for little bandwidth benefit.

Another scenario is the following: Peer A , new to network, uses partial synchronization as described in Listing 3 to bootstrap. It will start by receiving records with older timestamps from each and every peer it meets, until it reaches the point at which the other party returns records with recent timestamps. Since a synchronization point relates on one PermID, the total of the historical information of each peer must be traversed before A receives recent information from that peer, thus creating a long bootstrapping period.

We did consider the introduction of additional parameters such as a date and time range sent along with the Bloom filter to upper bound the number of records to be checked, however that resulted in the increase of the protocol complexity and offered small gain.

Special cases as the ones described are common in a unmanaged, open-to-join peer-to-peer networks like Tribler. To address these it is needed to further research and understand the type of data in the network, therefore we investigate the nature of information being exchanged.

4.3 Live and History Algorithms

We extend our research by examining the type of data we want to synchronize and the purpose of the network. We categorize data into *new* and *historical* according to their creation time to bootstrap a peer fast and keep it up to date, while raising its awareness about the past gradually. We suggest a algorithm based on the ones sketched earlier, that of two parts, the *live mode* and the *history mode*.

Tribler wants to be an efficient content delivery network. Users are more interested in recent content (e.g. today's news, latest music album) hence a every peer must be able to learn about new data³ fast. At the same time a peer must expand its knowledge about past events for the network to operate successfully.

This split of data enables peers to spread hot information faster and therefore to get access to the interesting content faster. Information about previous transactions is still valuable for the peer and it can be retrieved gradually from trusted peers.

When a peer joins the network it immediately enters the *live mode* which uses for all its lifetime to synchronize new data. The key point of live mode is that there is no SyncLog, peers agree that all the synchronization data exchanged refer to events of the past 24 hours⁴. Bloom filters are in use again to filter out shared

³Note that new data does not imply to new content but rather active content. For example the Debian Linux distribution swarm will be active and generating new data for around 2 years, until a new version of Debian will be released and a new torrent created.

⁴The 24 hour frame was selected taking into consideration that Tribler usage and churn is related

records.

Listing 4 Live Mode Synchronization

1. Peer *A* creates a Bloom filter with entries that have local creation time within the last 24 hours
 2. Peer *A* sends Bloom filter
 3. Peer *B* checks every of record with creation time within the last 24 hours against the received Bloom filter
 4. Peer *B* replies with *all* the records that where not included in the Bloom filter
 5. Peer *A* saves the records, deletes the Bloom filter
-

Algorithmically ‘live mode’ (Listing 4) is the same as the basic Bloom filter based method (Listing 1) except that all the peers agree to refer to the same time frame. This time frame is bonded to the present hence it creates a sliding window of updates.

Take note that ‘live mode’ implies no limitation on the number of records peer *B* returns. Although there is no theoretical upper limit on the number of records that can be created within a day, practically there is a limited amount of transactions between nodes and thus new records. Having no limitation causes the news to be spread fast while at the same time the 24 hour frame we suggest, keeps the number of records to reasonable numbers. There is however the possibility that peer *B* replies with less records than the maximum amount it can offer, which does not violate the concept of the algorithm but it may impact the *A*’s coverage, depending on the application. The specified time frame also keeps the size of the Bloom filter within acceptable bounds..

The second part of the algorithm is the history mode. *History mode* co-exists in parallel with the ‘live mode’. A peer can filter out peers that contain useful historical data for itself, by examining records already in the database and other parameters of the network. When a peer is located a transaction is made between them containing historical records.

The algorithm (Listing 5) uses a SyncLog but instead of moving towards the present, it moves towards the past. During historical synchronization between *A* and *B*, the former transmits and the last synchronization timestamp with the latter,

to human activity. Different timezones, thus different online times demand for an at least 24 hour frame, although the number can be easily configured to fit different applications.

Listing 5 History Mode Synchronization with SyncLog

1. Peer *A* reads the last successful synchronization date and time with peer *B* from the SyncLog t_1 and transmits it to *B*
 2. Peer *B* selects a record with timestamp t_2 where $t_2 < t_1$ and replies with t_2
 3. Peer *A* creates a Bloom filter with entries created between t_1 and t_2
 4. Peer *A* sends Bloom filter to Peer *B*
 5. Peer *B* checks every of record with creation time between t_1 and t_2 in its database against the received Bloom filter
 6. Peer *B* replies with *all* the records with timestamp between t_2 and t_1 that where not included in the Bloom filter.
 7. Peer *A* saves the records, deletes the Bloom filter and stores t_2 in SyncLog.
-

t_1 . In case there is no entry for *B* in the SyncLog, then the timestamp of the running date and time gets transmitted.

Peer *B* examines its database of records containing items with creation dates older than t_1 and selects one of them. The selection can be based either on the maximum number of records *B* is willing to transmit to *A* or on the maximum number of hours *B* wants to cover with only one communication. Peer *B* transmits the selected timestamp t_2 , where $t_2 < t_1$, back to *A*.

Peer *A* generates a Bloom filter with records from its local database between t_1 and t_2 and transmits it to *B* where the typical procedure of checking against the filter and returning rows will be performed. Finally *A*'s SyncLog will be updated with the new timestamp sent by *B* for later synchronizations.

If *B* does not have any older records than the received timestamp it informs *A* accordingly. Then *B* it is considered *fully synced* peer by *A* and it never gets contacted again regarding historical data. Its entry gets deleted from the SyncLog and its PermID gets hashed and included in a special Bloom filter which contains all the fully synced peers. The final historical related message transmitted from peer *B* is a local Bloom filter containing its fully synced peers. The received filter can be appended to *A*'s using a logical *AND* function. Using this technique *A* learns with which peers *B* is in full sync and therefore *A* is in full sync. Hence *A* avoids contact with peers that does not have more historical data to share. The SyncLog table does not grow indefinitely as opposed to the algorithm presented in Section 4.3. However note that the size and type of this Bloom filter must be chosen carefully to be able to store enough PermIDs. Last but not least peer *A* must trust *B* before adding its fully synced peer to the local Bloom filter.

The history mode can be deactivated or activated by the peers based on various policies. For example a peer may deactivate it when it is low on disk space or

alternatively activate when it wants to re-evaluate its similarity with others.

4.4 Scalability

Our second research question describes scalability. For scalability we aim to focus the synchronization efforts on the most interesting peers. With this enhancement peers keep track of the most relevant information and limit their traffic and storage requirements.

We performed data analysis on the current Tribler network and we concluded that, even for a relatively small sized network, transfer and storage of all the records is a major issue due to the large amounts of content generated. However Splash is designed to deliver the maximum amount of data as fast as possible, by randomly selecting peers for synchronization.

We suggest that peers do not select their synchronization hosts randomly but instead they select similar peers, i.e. peers with similar download activity. Only in rare cases a synchronization occurs with other peers.

Since a peer is more likely to contact similar peers to download content, we assume that the knowledge of meta-data about this peer and other similar peers, is more important than about the rest of the network. For example it is more important to be aware of the altruism of the similar peers compared to non-similar peers. Hence the focus in peers with relevant information shall not interfere with network experience of a peer.

Tribler incorporates a *similarity function* that was recently redesigned in [10] to provide more appropriate results faster. The similarity function tracks the patterns in download history, analyzes network activity and outputs a collection of similar peers to the local peer. This information is currently used to improve the keyword search for content in the network and can also be used to select peers for synchronization.

The occasional synchronization with non-similar peers provides records that can be processed again with the similarity function, to refine peer's collection of similar peers.

With this selection policy peers can transfer relevant information faster and cheaper and at the same time the disk space requirements are lowered as we practically limit the number of records to be saved.

4.5 Security

For the forth research question regarding security, a mechanism is required to ensure the validity of the data in transport. Although it is out of the scope of this thesis work to fully explore security, we propose a record validation mechanism based on *public key cryptography*.

Public key cryptography provides authentication for data communication. Every communicating party has a pair of private and public keys. The transmitting party

signs a message using its private key. The receiving party can verify the message if it holds the public key of the transmitter. Message verification guarantees that a message was created by the holder of the private key and that the message has not been altered in any way during transmission.

Tribler already includes Public Key Infrastructure (P.K.I.) which can be used to improve Splash. Using this, peers can sign each and every record they generate with their private key before sending it to others in the network. The record spreads through the network using Splash and now always carries along the signature of the original creator.

Taking advantage of P.K.I. peers can now identify and drop spoofed messages or messages not properly signed and therefore prevent the spread of invalid data. The public key required to verify each message can be retrieved using Splash itself from network-wide trusted peers. The latter can be located using the *Betweenness Centrality* [29] functions, currently under design by the Tribler team. These functions examine the network formation and locate the central nodes, namely the nodes connected to the most other peers which can be trusted with the task of public key spreading.

However with P.K.I. we cannot determine whether a message has been created using false values in the first place. Tribler already includes a function to normalize the trust of peers towards unknown peers based on the MaxFlow [9] algorithm. This function can not locate fake messages, but it can limit their effect and thus turn them worthless.

4.6 Multi-Hop Gossiping

We further work on coverage and churn questions as posed in the problem description chapter by introducing Multi-Hop Gossiping. Peers can use Splash to synchronize *all* records in their database, both the ones created locally and the ones collected from other peers. The intelligent selection of records done by Splash combined with a security mechanism on record level, as explained in Section 4.5, introduce the possibility of multi-hop gossiping without the drawbacks of network flooding or invalid information spreading.

The gossip mechanism currently deployed in Tribler does not allow multi-hop gossiping due to security issues and also, it cannot exploit its full potential due to the limited number of records in every message. On the other hand Splash can take full advantage of multi-hop gossiping and increase the number of records available for synchronization in orders of magnitude.

Multi-hop gossiping enables peers to receive more records with less synchronizations and helps effectively to reduce the effects of churn, since the spreading of records continues whether a peer is online or not.

4.7 Compression

The final optimization of the Splash is the *compression* of the returned records during a synchronization. Compression can further reduce the amount traffic required for synchronization.

Compression algorithms are known to work better on larger text than on smaller. Splash can send a large number of records on every synchronization, taking advantage of multi-hop gossiping and its limitless design. Therefore the combination of Splash with a compression algorithm results in savings in both traffic used and in time spend.

Compression in Splash is meaningful only for the returned records and not for the Bloom filters, which already contain data in a compressed form.

The addition of compression comes at the cost for higher need of computing power but the use of well known and optimized algorithms provide an acceptable trade-off to extreme traffic savings. We suggest the use the well known compression algorithms such as GZIP [30] or ZIP [31].

Chapter 5

Experimental Setup

Our experimental setup consists of a simplified event simulator with pluggable architecture. We developed plugins to simulate both BarterCast and Splash protocols. A second memory optimized simulator with Splash support was also developed, to simulate larger networks with more configuration options. With both simulators use a dataset created from real BarterCast records, crawled from the Tribler network. The dataset contains more than 1,3 million records and 9 thousand PermIDs.

5.1 Simulator

In order to evaluate our design and to understand the performance characteristics of Splash and BarterCast we created a simplified event simulator. We focus on simplicity to provide initial quantitative performance results. However the synchronization algorithms themselves are a full featured implementations instead of simulations.

The simulator executes a series of events in the order of their timestamp. We translate the actions of the network into series of steps. Each step contains a number of commands that normally are executed in real-time in the network, but the simulator executes them sequentially. Once a step is complete the simulator moves to the next time step. We choose to build a simplified event simulator to overcome the hardware requirements of a real deployment in 200, 400 and 600 systems.

The simulator was built using the Python programming language [32] and the Twisted Python asynchronous framework [33] and consists of two parts, the core and protocol. The core provides all the functionality that is essential for the simulator to work, whereas the protocol instructs the simulator how to complete the data exchange between two peers.

The core is responsible for the initializations and the management of the events. At start parses the input parameters and creates sqlite [34] databases in memory, each one representing the MegaCache of a peer. The total number of databases is equal to the total number of peers that are being simulated.

After the initializations of memory and of other essential parameters, the simulator starts parsing the *event file*. Event files contain a series of space separated fields defined as follows: `time`, `perm-id-from`, `perm-id-to`, `downloaded` and `uploaded`. The first field defines the exact time the BarterCast record appeared in the network and the following fields define the BarterCast record itself. Multiple lines in the *event file* can start with the same `time` field and are considered to be a *step* of the event simulator.

For every step, the simulator reads the actions to be taken from the *event file*. For every line in the event file it injects into one database a BarterCast record with the corresponding values. The database to receive the records is not chosen randomly but it is the database owned by the peer with PermID equal to the `perm-id-from` field of the line. This way we simulate the generation of BarterCast records in the real network, using real data.

Before executing to the next step the simulator calculates how much time has passed since the last synchronization of the peers. Note that since we are not using real time but rather virtual time steps, the time that passed is equal to the difference of the current action time stamp minus the last synchronization time stamp. If the time passed is larger or equal to the user configurable synchronization interval, then a synchronization round starts.

During a synchronization round the simulator instructs every peer sequentially, in random order, to synchronize with another peer selected randomly from the list. Once a peer receives the instruction to synchronize, it follows the steps defined in the protocol section of the simulator. In some cases, instructed by simulator configuration parameters, where peers commit multiple synchronizations per round the simulator ensures that all peers have completed one synchronization before moving to the next. As soon as the round is complete, simulator continues processing the next time event.

A synchronization round is a time point that the all the peers synchronize, however the simulator executes the synchronizations in sequence and not in parallel and chooses the pairs randomly thus faking the spread of synchronizations throughout the whole time span between the current and the previous round. This design decision for the simulator allows for better control of the number of the synchronizations and time of their execution and also leads to simple dataset structure. Experiments with the BarterCast protocol proved that the results of this simulator are similar to values collected from a real network.

The *synchronization interval* (flag `sync-interval`), namely how often the simulator enters a synchronization round, can be defined in seconds from the command line. The same holds for the number of synchronizations per round (flag `syncs-per-round`) which represents how many synchronizations will a be conduct during each round.

The simulation ends when the simulator completes with the execution of the actions defined in the event file. Before exiting the simulator prints detailed information about the number of records owned per peer and calculates the coverage percentage for each peer and the average value for the network. The report also

includes statistics about the total number of bytes transmitted and received by the peers during the synchronization as global sum and as per peer average.

The protocol defines the series of steps a peer must do to complete a synchronization. Both the ‘server’ and ‘client’ sides are programmed as Twisted Protocols, defined in the Twisted Framework¹. Each simulated synchronization does not differ from a real synchronization, as peers have to connect to each other through the network, initialize the procedure, exchange data and cleanly close the connection.

The simulator architecture allows for pluggable protocols thus, multiple different implementations can be tested using the same core. For our testing proposes we programmed two protocols, the original BarterCast and Splash. The former was created according to [35] and [12], whereas the latter according to the ‘live mode’ algorithm described in Section 4.3. We focused on implementing the ‘live mode’ algorithm to make fair comparisons with BarterCast which does not support exchange of historical data.

The core defines two Factories¹, one to be used as server and one as client during a synchronization. The factories are created at the start of the execution of the simulator and persist in memory until the end. We change the peer a factory represents by redefining the PermID and database it is connected before simulating a synchronization.

The use of Factories and Protocols, and the Twisted Framework in general, assist in the implementation of a full featured synchronization instead of a simulated synchronization. This design allows the researcher to take into consideration parameters as such the protocol overhead, link bandwidth and others.

The simulator was used to produce results for the initial tests of the Splash protocol, as well as to simulate the BarterCast protocol in terms of coverage and bandwidth usage. Due to the limitations imposed by its design, we constructed a memory optimized simulator (Chapter 5.2) to further investigate the attributes of Splash, in larger networks and considering other parameters such as churn.

5.2 Memory Optimized Simulator

To increase the accuracy of our simulations and to study Splash performance better we wanted to simulate networks with more than 200 peers. However the fully featured algorithm implementation and the use of separate databases per peer proved to be a bottleneck while increasing the number of simulated peers. Therefore we constructed a memory optimized simulator, especially designed for Splash, to conduct experiments in more depth.

By profiling the initial simulator we understood that having a unique sqlite database for each peer, although it can help to study the size and the performance

¹More information on the Twisted Framework exists on the official webpage <http://www.twistedmatrix.com>. Detailed description of the Protocols and the Factories can be found at <http://twistedmatrix.com/documents/current/core/howto/index.html>

of each peer individually, limits the number of peers to the number of available RAM. A sqlite database grows up to 50 MiB for 15.000 BarterCast records resulting in a demand for more than 10 GiB of RAM during simulation of 200 peers and more than 30 GiB for 600 peers. Moving the database storage to hard disk to overcome the RAM limitation issues, caused significant slowdown of the simulation. Moreover the overhead of real sockets throttled the performance.

For the memory optimized simulator we completely removed the Twisted Framework and the real sockets used in communication. We also removed the multiple database instances and replaced them with one database. This one database aggregates all the BarterCast records into one big table and the peer PermIDs into another table. The ownership of a BarterCast record is now defined as a *foreign key* relationship between a row in the PermID table and a row in the BarterCast table.

This version of the simulator requires a fraction of the memory compared to its predecessor. The linking of the records instead of copying increases dramatically the speed of the simulation. As a result the memory optimized version of the simulator allows the simulation of bigger datasets and larger networks in acceptable amount of time.

We further extended this version to support churn simulation. Through the command line interface we can define the `churn-middle` parameter. `Churn-middle` defines the number of hours per day that a peer is active in the network. On every day change, as instructed by the step timestamps and not the real clock, the simulator randomly chooses for each peer the part of the a day that it will be online. Both the time that a peer joins the network and the time that a peer stays connected are selected for each peer individually. The amount of time a peer stays online gets selected using a normal distribution with parameters $\mu = \text{churn-middle}$ and $\sigma = 1$.

The last extension of the simulator is the *semantic clustering* support, used to simulate the preference of a peer to synchronize similar peers instead random peers. Again using command line parameters, specifically the `semantic-groups` flag, we can define the number of semantic groups to be created from the simulator. Before the simulation begins peers get randomly distributed into groups, with one peer belonging to only one group. During the synchronization round the peer pairs to communicate are selected randomly but belong to the same group. The `semantic-random` flag instructs the number of synchronization for which the peers are selected randomly from the whole set and not only from the group, and can be used in combination with the `semantic-groups`.

To summarize, the experimental infrastructure consists of two simulators, with different design and capabilities, which can be used to simulate networks of 600 or more peers in a reasonable amount of time. We used the first simulator to measure BarterCast and Splash performance for small networks and datasets and the memory optimized version to further explore Splash properties. The simulation parameters include the time interval between two synchronization rounds, the number of synchronizations per peer per round, the amount of time a peer is online per day and finally the number of semantic groups including the percentage of group syn-

chronizations over the overall network synchronizations. Multiple combinations of all the parameter were used during the experiments presented in Chapter 6.

5.3 Dataset

An important part of the evaluation infrastructure is the dataset. It is a common mistake, when investigating the behavior of a network, to use a dataset that does not represent the actual activity of the network. For our experiments we used a dataset of BarterCast records collected from the Tribler network using a crawler, within the period June to September 2009. The dataset contains more than 1.3 million records that refer to more than 9 thousand PermIDs.

BitTorrent networks and therefore Tribler can be crawled to collect information about the network, its health and to identify possible failures. Tribler has integrated some additional support for crawling to augment the crawled information. Despite the Tribler crawler connecting to each and every one Tribler client it can locate and requesting records about its activity, none of the data collected contains personal information about the user, the files exchanged or anything else which could potentially evade his privacy.

Among other useful data the crawler collects the BarterCast records each peer stores in its MegaCache. Using these records we can visualize the knowledge of a peer about the rest of the network, as done for example in Figure 2.1. Furthermore if we process the timestamps of a BarterCast record and the timestamps generated when the crawler receives data we can calculate the moment a record first appeared in the network.

Each BarterCast record in our dataset contains the following fields: `peerid`, the PermID of the peer to which the record belongs, `peer-id-from` and `peer-id-to` which contain the PermIDs of the peers involved in the BarterCast transaction, `downloaded` and `uploaded` carry the number of bytes exchanged. Finally three timestamps `last-seen`, `remote-peer-time` and `crawler-time` correspond to the time the peer was last seen, the local time of the peer and the local time of the crawler accordingly. If we combine the `last-seen`, `remote-peer-time` and `crawler-time` with the following formula we can calculate the time record first pushed to the peer.

$$pushedtime = lastseen + crawlertime - remotepertime$$

If the `perm-id-from` is the same as the `peerid` then the *pushed time* variable represents the time a BarterCast record was created.

We extracted from the crawler database a list of all the BarterCast records. We filtered the records keeping only the ones where `peerid` is equal to `peer-id-from` and we calculated the time a record was originally created using the function above.

Subsequently we sorted the peers by network activity in descending order, namely how many records they generated in the period of 4 months and we produced three sets of 200, 400 and 600 peers accordingly.

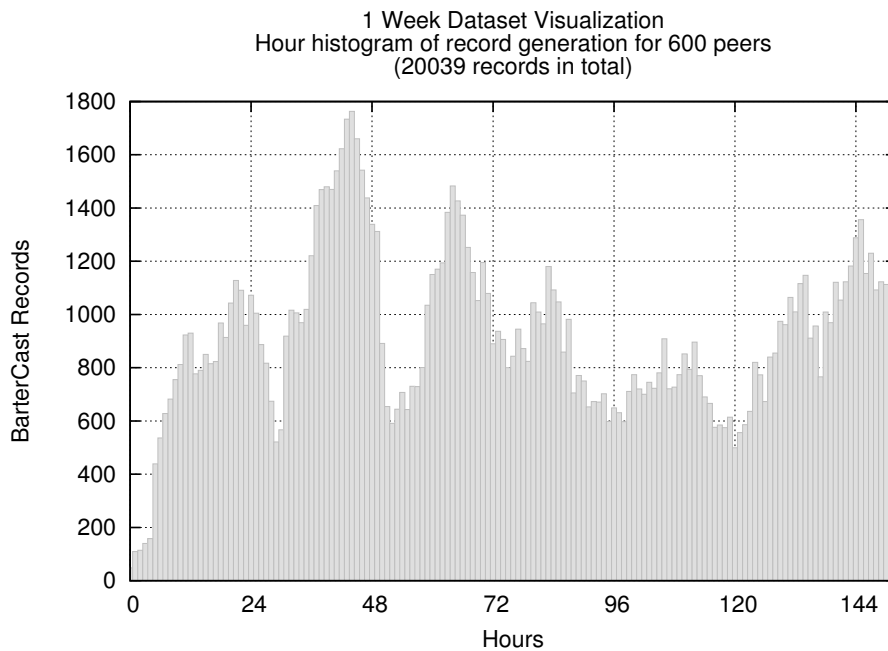


Figure 5.1: Visualization of the BarterCast dataset of one week (168 hours) with 600 peers and 20.000 records. The bars represent the number of records created within the last 24 hours, namely the number of records available for synchronization at each hour of the week using the ‘live mode’ algorithm.

For each of the three sets we generated three databases containing the BarterCast records in which the `peer-id-to` value belongs to the set. The three different sets and databases will be later used to simulate networks of different sizes. The distribution of the records in the datasets is visualized in Figure 5.1.

Chapter 6

Experimental Results

In this chapter we present multiple simulations we perform for both the BarterCast and Splash algorithms covering a wide range of cases. The BarterCast simulator is run foremost using the crawler collected records. We examine the bandwidth usage and data coverage of the BarterCast algorithm currently used in Tribler and we verify that our results are similar to the results measured in the real network.

Subsequently we confirm that the Bloom filters can be used successfully to synchronize data in a peer-to-peer networks by simulating small networks of 200 peers with 8 hours of BarterCast records. During these tests we calibrate the false positive rate of our Bloom filter to 5% which provides a good balance between the size of the Bloom filter and the false positive faults and ultimately we compare Splash against BarterCast.

By taking advantage of the simulator's configuration options, described in Section 5.2, we measure the coverage and traffic of Splash while varying the number of synchronizations per round, the round interval, the churn rate and the grouping of the peers. Finally we investigate the effects of churn on Splash by gradually decreasing the peer uptime.

Each network experiment is repeated 10 times to confirm that the results are always the same with minimal, less than 2%, standard deviation.

6.1 Basic Experiments

We used the first version of our simulator to get BarterCast simulation statistics related to coverage and traffic. The BarterCast column in Table 6.1 shows the results of coverage, inbound and outbound traffic and the number of synchronizations for a network of 200 nodes exchanging 8 hours of BarterCast records. The 'synchronizations' metric represents the number of BarterCast messages transmitted, either as a new message or as a reply to a received message.

BarterCast simulation results are in agreement with the image of the network as sketched in problem description (Chapter 2.1) using data collected from the crawler. The peers hold on average less than 20% of the total available information

in the network while each peer performs a total of 398 synchronizations.

Since BarterCast is a push protocol, meaning that peers forward records of the network, we expect the outbound traffic to be larger than the inbound traffic. The former number corresponds to the protocol overhead and the actual record transfer while the latter corresponds only to the protocol overhead.

We carry out the same experiment using Splash in ‘live mode’ with a time frame of 24 hours. We select the configuration of Splash to be three synchronization per 4 hours, a total of 6 synchronizations for the 8 hour dataset of this experiment. Respectively to BarterCast experiment, the ‘synchronizations’ metric counts the number of messages received containing records. We aim to produce similar coverage results to BarterCast so we can reasonably compare the bandwidth usage and number of synchronizations of the two implementations.

	BarterCast	Splash	Splash with Compression
Synchronizations (#)	398	6	6
Coverage (%)	18.92	23.34	24.14
Inbound Traffic (KiB)	2.66	30.90	5.16
Outbound Traffic (KiB)	46.55	0.58	0.55

Table 6.1: BarterCast and Splash simulation results for 200 peers for 8 hours of BarterCast records. Numbers represent the peer average in coverage and traffic and the total number of synchronizations per peer.

During these experiments we also measure the impact of data compression as explained in Section 4.7 . The inbound traffic of Splash stands for the protocol overhead and the size of the records received, compressed or uncompressed, while the outbound metric counts the transmission of the Bloom filters and the protocol overhead.

A total of 6 synchronizations per peer is enough for Splash to reach 24% average coverage which is close to the 18.9% of BarterCast that uses an average of 398 synchronizations. Since BarterCast is pushing records whereas Splash is pulling records from the network it is more fair to compare one’s inbound traffic statistics with the outbound statistics of the other and the other way around.

The uncompressed version of Splash consumes 1.5 times less traffic, only 30.9 KiB compared to the 46.55 KiB of BarterCast. This difference is due to BarterCast committing 67 times more synchronizations than Splash which translates in traffic consumption due to payload and protocol overhead.

Furthermore the compressed version uses 6 times or 9 times less bandwidth compared to the uncompressed Splash and BarterCast accordingly, demonstrating large gains from its usage. At the same time the comparison of the inbound traffic of BarterCast with the outbound traffic of Splash follows the same trend, with the latter requiring 4.5 times less resources. Adding up inbound and outbound

traffic and comparing again, results in Splash using 8.6 times less bandwidth than BarterCast.

It is clear that Splash outperforms BarterCast. We have to take into consideration that Splash is not superior only because it performs targeted data synchronization using Bloom filters but also because it does not limit the number of records transmitted on every synchronization as BarterCast does. Despite the lack of a limit Splash uses only a fraction of the traffic BarterCast uses, both inbound and outbound, and does not saturate the resources of the peers.

6.2 Detailed Splash Experiments

We examine the abilities of Splash in more detail by extending the size of the dataset to one week of 15.048 BarterCast records and the number of configurations to six covering 3, 6, 12, 15, 30 and 60 synchronizations per peer per day. These results are obtained from the memory-optimized simulator that has only Splash support.

During the first experiments we executed different simulations with a different combinations of how many peers are contacted per synchronization round and how often rounds occur. That results in a different number of synchronizations per day per peer for each experiment.

Some combinations yield the same number of synchronizations per day, for example when contacting one peer but repeat 24 rounds times a day and when contacting 24 peers all at the same round of a day. Experiments demonstrate that different configurations with the equal number of synchronizations per day produce the same coverage results¹. To better express the results and to avoid duplicates we selected only one configuration for each number of synchronizations per day favoring, when there was a choice, the configuration with the lowest time interval between the rounds to help the faster spreading of information.

The results of the detailed experiments are visualized in Figure 6.1 and Figure 6.2. According to Figure 6.1 Splash reaches the average coverage status of 95% for every peer using only 12 synchronizations per day. Increasing the number of synchronizations even further to 15, 30 or 60 per day results in coverage ratio that reaches 100%.

The larger the coverage the more the inbound traffic for every peer. Inbound traffic represents the size of the data received during synchronizations and include both the compressed records and the protocol overhead. We conclude from the left graph of Figure 6.2 that there is an trend of increasing traffic until we reach 12 synchronizations per day or about 95% coverage. Beyond that point although

¹Most of the configurations with the same number of synchronizations per day produced exactly the same coverage results. In rare cases the difference in the results was not larger than 1%.

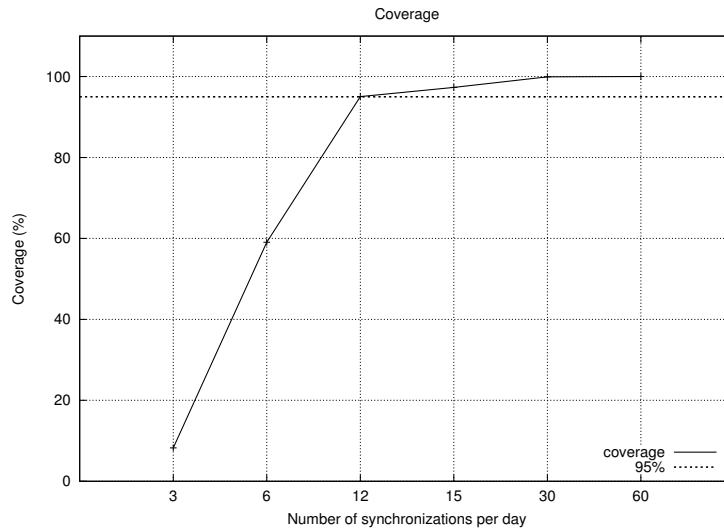


Figure 6.1: Simulation of peer data coverage for 200 peers using over 15.000 BarterCast records spread over a period of a week. The more synchronizations per peer per day the higher the coverage.

the number of synchronizations increases the algorithm does not consume significantly more bandwidth, except for the last configuration, since almost everything is already in the database of every peer. The increase of traffic for the last configuration is due to the protocol overhead which is 2-5 times more than in the other configurations. This graph provides a proof that Bloom filters can prevent duplicate information from being transmitted and therefore limit the bandwidth usage to the minimum.

On the contrary the outbound traffic, shown in the right graph of Figure 6.2, is increasing approximately linearly with the number of syncs per day, which is to be expected. This metric counts the bytes required to transmit the Bloom filter and the protocol overhead. The higher the coverage for each peer the bigger the Bloom filter is, therefore more bandwidth is used to transmit it.

It is important to note that although it may be bandwidth inexpensive to reach average coverage of 95%, it is very costly to reach 100% since it is needed for each peer to perform five times more synchronizations that translates to five times more transmissions of the Bloom filter. This 5% is related to the false positive error rate of Bloom filter that we configured to 5%. Lower error rate would result in larger Bloom filters that would increase the bandwidth use per synchronization but also the coverage. The choice of Bloom filter error rate and synchronizations per day can be configured to fit the application. For our experiments 95% fulfills our demands and therefore we choose to maintain Bloom filter size small.

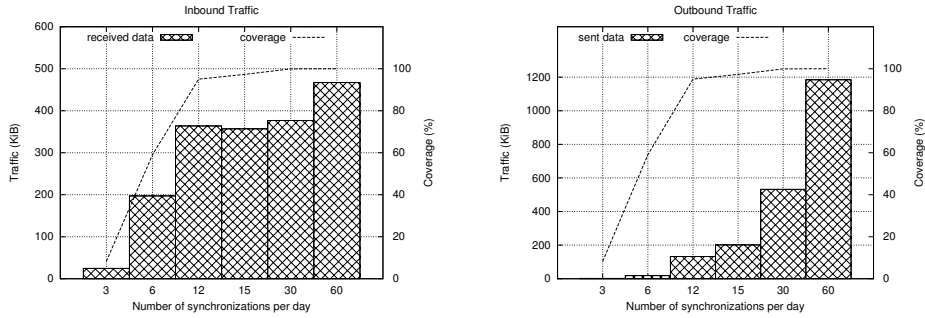


Figure 6.2: Inbound and outbound traffic during the simulation of 200 peers exchanging 15,000 BarterCast records spread over a period of a week. The dashed line represents the average peer data coverage and the bars represent the average data transferred per node.

6.3 Network Size Experiments

To evaluate the scalability of Splash we split the network into groups and we later increase the number of peers and the number of records gradually. We take advantage of the semantic clustering functionality of the advanced simulator that can split the 200 peer network into 4 groups of 50 randomly selected peers. Each and every peer belongs into only one group only and during the simulation synchronizes data only with others within the same group thus increasing its *local* group coverage. In rare cases, configured to happen once in every hundred synchronizations for this experiment, a peer randomly chooses another peer from the whole network to synchronize thus increasing its *global* coverage, meaning records that refer to peers outside the local group.

In Figure 6.3 the solid line represents the average local coverage, namely how many of the records regarding the cluster peers are known on average to every peer of the group. The dashed line represents the average number of records that where not created by peer of the same group, every peer holds.

We observe in Figure 6.3 that peers can synchronize more important local information with fewer synchronizations, 6 synchronizations are required to reach 80% local coverage while for the same number of synchronizations without the semantic clustering peers hold 60% of the overall information (Figure 6.1). This is due to the peers synchronizing mostly within a group containing less peers than the whole network and the possibilities to contact the same peer in different periods to get new records is higher. The local coverage line trend is similar to the that of Figure 6.1.

The global coverage line is increasing linearly with the number of synchronizations per day, since peers randomly select a pair from the whole network every hundred synchronizations. Hence having more synchronizations will increase the number of times peer synchronize with the whole network and therefore increase

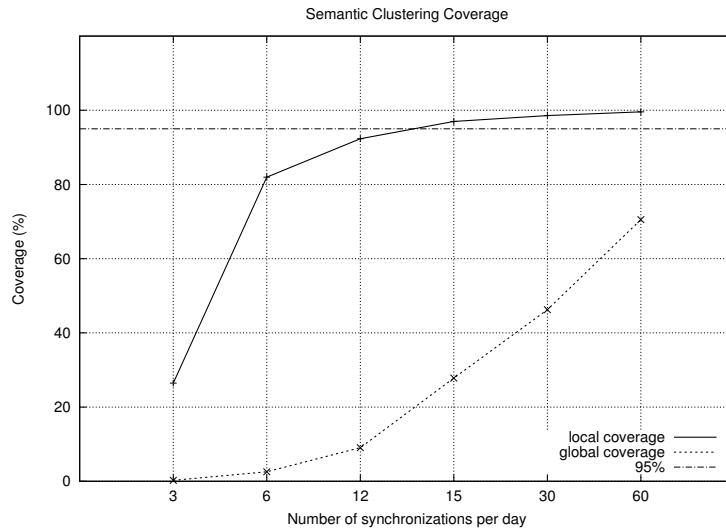


Figure 6.3: Simulation of 200 peers split over 4 groups of 50 peers each with a possibility of 1% for each peer to connect randomly to another peer not in the same group. The solid line represents the peer average local group data coverage and the dotted line represents the peer average global data coverage. There is a possibility one per one hundred synchronizations for each and every peer to contact a peer chosen randomly from the whole network and not only from the local group.

the global coverage. It is important to note that we don't aim for total local and global coverage of 100% since that would practically eliminate the use of semantic clusters. Instead we want to achieve good local coverage and coarse global coverage so peers hold all the important information while they are still aware about the rest of the network.

Twelve or fifteen synchronizations per day fulfill this requirement for the network and dataset under simulation, according to Figure 6.3, by providing around 95% local coverage and between 20% and 30% of global coverage. Note that the global coverage metric represents the number of records a peer holds in the network that do not originate from peers of the same group.

Semantic clustering was added in the design process of Splash in Section 4.4 to provide scalability for the algorithm in terms of bandwidth and storage requirements. The inbound and outbound graphs in Figure 6.4 show a significant drop in traffic especially for the configurations of 12 and 15 synchronizations per day. The inbound traffic drops from the 360 KiB and 355 KiB of the non-clustered experiment (Figure 6.2) down to 160 KiB to 180 KiB per peer accordingly. The same holds for the outbound traffic which drops from 130 KiB and 200 KiB down to 40 KiB and 60 KiB accordingly.

The next item in our experiment list is to study the behavior of Splash while

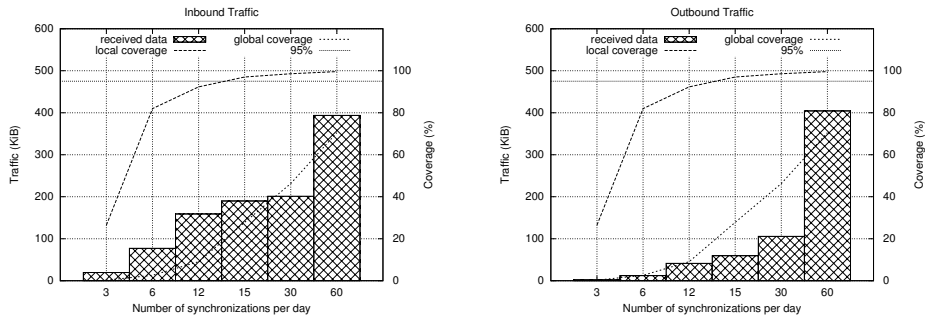


Figure 6.4: Inbound and outbound traffic during the simulation of 200 peers split over 4 semantic groups of 50 peers each, exchanging 15.000 BarterCast records spread over a period of a week. The dashed line represents the average local data coverage, the dotted line represents the average global data coverage and the bars represent the average data transferred per node.

increasing the overall number of peer in the system. We use datasets with 400 and 600 peers which contain 18.000 and 20.000 BarterCast records accordingly spread over the period of one week. The network gets split once more into groups of 50, thus 8 groups for the 400 peer network and 12 groups of the 600 peer network. We simulate these new datasets for 12 and 15 synchronizations per day, configurations that provided good results during the 200 peer network simulations.

In Figures 6.5 and 6.6 we observe that despite the increasing number of peers and records to be transmitted, local group coverage is preserved to the same levels. The constancy of the local coverage proves that the mechanism is not affected by the increase in number of records if we maintain a constant size of group.

However the percentage for global coverage is dropping, since the information is now spread over more groups and therefore every peer needs to complete more global synchronizations to collect the same percentage of records.

6.4 Churn Experiments

To further increase the accuracy of our results we execute a number of experiments simulating the churn effects that take place in a real peer-to-peer network. We perform three experiments where we gradually increase the time each and every peer stays off-line per day starting from 6 hours, to 12 hours and finally 18 hours in a network 200 peers. In Figure 6.7 we compare the average data coverage coverage with the results from the previous experiments without peer downtime (Figure 6.1). It is clear that downtime affects the coverage of the peers. The more downtime the worse the coverage for the same number of synchronizations.

Note that since peers are offline during a period of the day they do not execute the full number of synchronizations but only the synchronizations that happen during their online time. However the x-axis of the graphs in this section represents

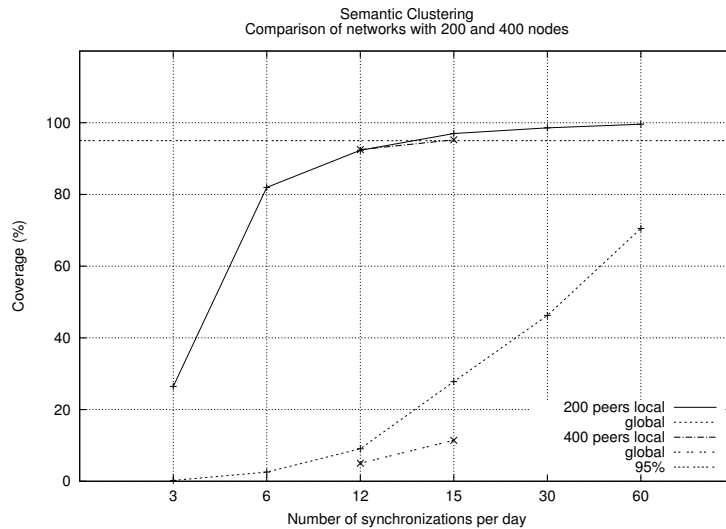


Figure 6.5: Simulation of 200 and 400 peers split over 4 and 8 semantic groups accordingly, of 50 peers each. Along with the size of the network the number of BarterCast records grows from 15.000 up to 18.000. The 400 peer network simulation was conducted for 12 and 15 synchronizations per day.

the total synchronizations a peer would have if it was online all the time so we can fairly compare the results against the previous experiment. For example in a simulation with 12 synchronizations per day per peer and peer downtime of 50% the latter will perform only 6 synchronizations per day and during the other 6 will be disconnected from the network.

We continue the experiments raising the number of synchronizations per day in an attempt to fight the churn of the peers. We use 8, 10, 30, 36, 72, 84, 96, 108 and 120 synchronizations per day per peer a total of 9 different configurations to locate the limitations of the system.

In the upper left graph of Figure 6.8 we observe that for a network with average 75% uptime for each peer the algorithm comes close to 90% local coverage with 10 synchronizations and exceeds 95% with 30 synchronizations, therefore it can fight the low churn rate effectively.

In the upper right graph of Figure 6.8 the peer uptime is reduced to 50%, namely to 12 hours per day. Although the coverage percentage drops, peers still hold 90% of the local records with 30 synchronizations per day and are able to reach 95% with a total of 72 or more synchronizations.

Finally in the bottom left graph the peer uptime is only 25% therefore around 6 hours per day. Despite the peers being mostly offline, Splash manages to transfer a considerable amount of information, more than 80% to every peer of the network by performing only 72 synchronizations. It is clear though that performing

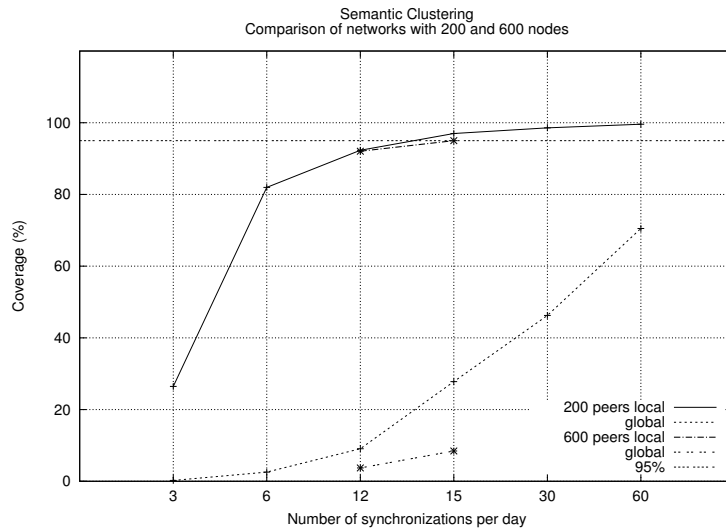


Figure 6.6: Simulation of 200 and 600 peers split over 4 and 12 semantic groups accordingly, of 50 peers each. Along with the size of the network the number of BarterCast records grows from 15.000 up to 20.000. The 600 peer network simulation was conducted for 12 and 15 synchronizations per day.

more than 72 synchronizations per day will not help the peer reach 95% coverage or more, since even 120 synchronizations do not overcome the 90% barrier. This behavior is expected because of the peers being offline a long time they miss the synchronization window of some records and therefore are unable to get these records no matter how many synchronizations they perform during online.

This limitation of the ‘live mode’ motivates the existence of the ‘history mode’ that peers can use to synchronize records that did not obtain while online.

We performed the same experiment in the 400 and 600 peer networks for 72 synchronizations and uptime of 6 hours per day per peer and we found similar results. For the 400 peer network the local coverage is 81% (83% for the 200 peer network) and the global coverage is 31% (32% for the 200 peer network). Similarly for the 600 peer network the local coverage is 82% and the global coverage is 30%.

In all cases Splash can deliver a good data coverage, above 80%, for each and every peer in spite of the churn rate, by configuring the number of synchronizations per day.

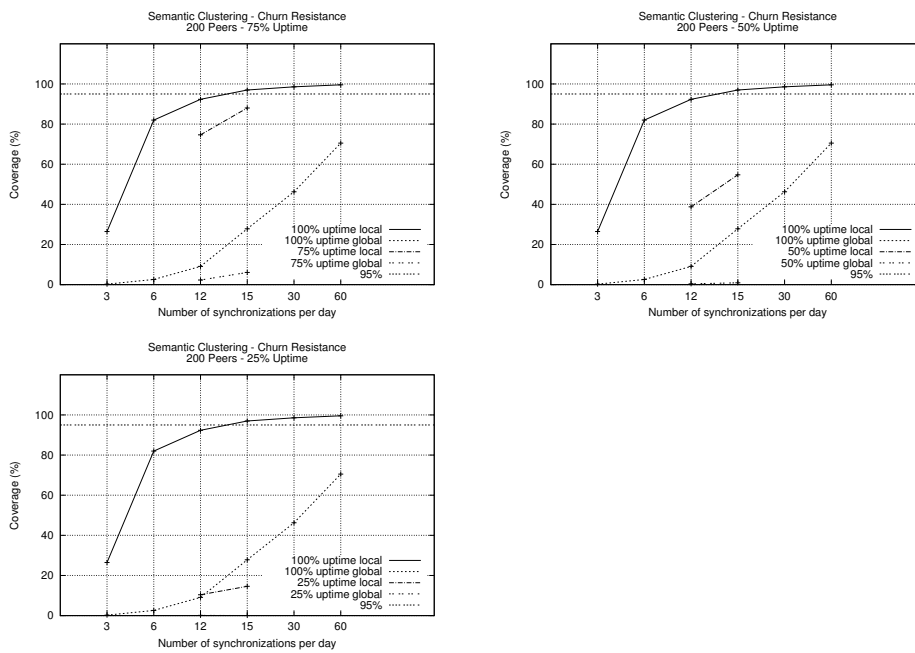


Figure 6.7: Simulation of data coverage for 200 peer network split over 4 semantic groups of 50 peers each, exchanging a more than 15.000 records. The solid and dotted lines represent the average local and global data coverage for each peer in the network when there is 100% uptime per day for each and every peer. The dashed and dotted-dashed lines represent the local and global data coverage accordingly for each peer with 75% uptime per day for each and every peer for the top left graph and 50% and 25% uptime for the top right and bottom left graphs accordingly. It is clear that downtime decreases both global and local coverage.

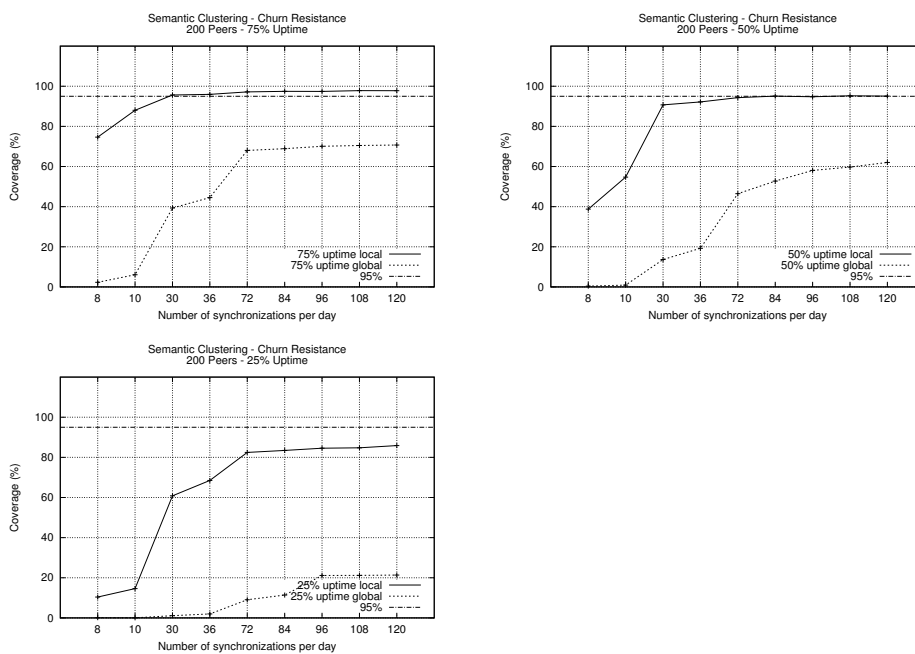


Figure 6.8: Simulation of data coverage for 200 peer network split over 4 semantic groups of 50 peers each, exchanging a more than 15.000 records. The graphs represent 9 different configurations of the simulator with varying synchronizations per day ranging from 8 up to 120.

Chapter 7

Conclusions and Future Work

In this chapter we revisit the research questions as posed in Chapter 2, we summarize the results of our experiments and we present the strengths and weaknesses of using our implementation compared to BarterCast. Finally we discuss the future development of Splash.

7.1 Conclusions

During this thesis work we designed and implemented a new data synchronization algorithm with powerful features. The gossiping algorithm currently used by Tribler has proved incapable to spread information widely and efficiently, both during simulation and from real network measurements. This functional weakness in combination with its design inflexibility motivated us to design new ways to synchronize data throughout the network.

We set four research questions regarding data coverage per peer, network scalability, churn and security and we designed Splash to resolve them. Splash uses Bloom filters to intelligently choose which records to synchronize with each peer and selects the records depending on their existence in other peer's databases, contrary to BarterCast which selects the latest records or the records with the highest values. This way all records have equal probability to be transmitted over the network resulting in a high data-coverage ratio per peer. According to the simulation experiments of Chapter 6, Splash can achieve on average more than 95% coverage per peer with only a few synchronizations per day, which is 5 times more than BarterCast. Additionally Splash uses 8.6 times less bandwidth compared to BarterCast to achieve the same data coverage.

Splash can scale along with the network and dataset and provide high data coverage with acceptable bandwidth and storage usage by taking advantage of its partial data synchronization capabilities. Peers do not have to have the same set to synchronize, nor have to synchronize all their entries. Using Splash in combination peer similarity in Tribler, data synchronization can be restricted on records with specific origin or be completely unrestricted on demand.

Despite the heavy churn rates a peer-to-peer network may suffer Splash can effectively keep the high average data coverage per peer by letting peers pull as much information as offered in a few synchronizations that take place while they are online. The lack of a limit for the number of records synchronized per transaction, combined with the use of an effective compression scheme enables each peer to synchronize hundreds of records in every transaction in a bandwidth-efficient manner.

For peers that have been offline for a long period of time and lost the ‘live’ window of opportunity to synchronize some records, a ‘history’ algorithm is described which can be used to synchronize records that were created in the past efficiently once again with the use of Bloom filters.

Last but not least we discuss a security mechanism on record level that can enhance Splash performance by increasing the number of records available for synchronization per peer. The described security mechanism which is based on the public key infrastructure, that is already available in Tribler, enables peers to verify record validity and effectively reduce the spreading of invalid or malicious data.

Splash is definitely of higher complexity than the current algorithm in use to spread BarterCast records. The latter constructs simple self-contained messages with a number of records selected using simple criteria and instructs peers to forward them to each and every of their connections. On the contrary Splash requires a round of communication, and not just a message, to select the records to forward to each peer individually.

To summarize, Splash has proved to be a practical solution to synchronize data in untrusted and unmanaged networks and its flexible design provides adaptability to number of different networks having different requirements, including Tribler.

7.2 Future work

During the design and development of Splash the following the following extensions and modifications of the algorithm have been left for future work:

- Implementation and integration of a security mechanism operating on the record level, to validate data and prevent fake information from spreading. A possible implementation was described in Section 4.5 using public key cryptography that is known to provide the needed functionality using strong cryptography.
- Simulation of the ‘history’ mode. This mode is based on the same principle of operation as the ‘live’ mode which was implemented and tested with this work but it is designed to synchronize older records.
- Deployment of Splash in a network of thousands of peers to rate its performance.
- Design of a policy regarding the synchronization interval and the number of synchronizations per day. A sophisticated algorithm can be used by peers to

determine the amount of data available in the network and therefore increase or decrease the number of synchronizations accordingly.

- Integration and deployment of Splash into Tribler by gradually using it to serve the needs of protocols like ChannelCast, VoteCast, BarterCast and others. Ultimately Splash can replace the whole BuddyCast layer and be responsible for the meta data transport within the Tribler network.

Bibliography

- [1] Maymounkov Petar and Mazières David. Kademlia: A peer-to-peer information system based on the xor metric. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 53–65, London, UK, 2002. Springer-Verlag.
- [2] Shawn Fanning. Napster, 1999. <http://en.wikipedia.org/wiki/Napster>.
- [3] Bram Cohen. Bittorrent - a new p2p app, 2001. <http://finance.groups.yahoo.com/group/decentralization/message/3160>.
- [4] Ipoque. Internet study 2008/2009, 2010. http://www.ipoque.com/resources/internet-studies/internet-study-2008_2009.
- [5] Bittorrent peer exchange conventions, June 2010. <http://wiki.theory.org/BitTorrentPeerExchangeConventions>.
- [6] Peter Biddle, Paul England, Marcus Peinado, and Bryan Willman. The dark-net and the future of content distribution. Levine's Working Paper Archive 61889700000000636, David K. Levine, Oct. 2003.
- [7] David Hales, Rameez Rahman, Boxun Zhang, Michel Meulpolder, and Johan Pouwelse. Bittorrent or bitcrunch: Evidence of a credit squeeze in bittorrent? In *Proceedings Wetice 2009*, pages 99–104. IEEE CS Press, 2009.
- [8] Andrade Nazareno, Mowbray Miranda, Lima Aliandro, Wagner Gustavo, and Rippeanu Matei. Influences on cooperation in bittorrent communities. In *P2PECON '05: Proceedings of the 2005 ACM SIGCOMM workshop on Economics of peer-to-peer systems*, pages 111–115, New York, NY, USA, 2005. ACM.
- [9] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*, pages 651–664. MIT Press and McGraw-Hill, second edition, 2001.
- [10] Niels Zeilemaker. Improving p2p keyword search by combining .torrent metadata and user preference in a semantic overlay, 2010.
- [11] Rahim Delaviz. Swarm-based reputation consensus, 2010. <https://www.tribler.org/trac/wiki/SwarmBasedReputationConsensus>.
- [12] Johan A. Pouwelse, Jie Yang, Michel Meulpolder, Dick H.J. Epema, and Henk J. Sips. Buddycast: an operational peer-to-peer epidemic protocol stack. In *Proc. of the 14th Annual Conf. of the Advanced School for Computing and Imaging*, pages 200–205. ASCI, 2008.
- [13] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.
- [14] Wikipedia. Bloom filter. http://en.wikipedia.org/wiki/Bloom_filter.
- [15] Almeida Paulo Sérgio, Baquero Carlos, Prego Nuno, and Hutchison David. Scalable bloom filters. *Inf. Process. Lett.*, 101(6):255–261, 2007.

- [16] Mitzenmacher Michael. Compressed bloom filters. *IEEE/ACM Trans. Netw.*, 10(5):604–612, 2002.
- [17] Fan Li, Cao Pei, Almeida Jussara, and Broder Andrei Z. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, 2000.
- [18] Chang Fay, Dean Jeffrey, Ghemawat Sanjay, Hsieh Wilson C., Wallach Deborah A., Burrows Mike, Chandra Tushar, Fikes Andrew, and Gruber Robert E. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):1–26, 2008.
- [19] J. K. Mullin and D. J. Margoliash. A tale of three spelling checkers. *Software: Practice and Experience*, 20(6):625–630, 1990.
- [20] Marais Hannes and Bharat Krishna. Supporting cooperative and personal surfing with a desktop assistant. In *UIST '97: Proceedings of the 10th annual ACM symposium on User interface software and technology*, pages 129–138, New York, NY, USA, 1997. ACM.
- [21] Minsky Yaron, Trachtenberg Ari, and Zippel Richard. Set reconciliation with nearly optimal communication complexity. Technical report, Ithaca, NY, USA, 2000.
- [22] Starobinski David, Trachtenberg Ari, and Agarwal Sachin. Efficient pda synchronization. *IEEE Transactions on Mobile Computing*, 2(1):40–51, 2003.
- [23] Sachin Agarwal and Ari Trachtenberg. Practical set reconciliation implementation, 2004. <http://ipsit.bu.edu/programs/reconcile/>.
- [24] Ari Trachtenberg, David Starobinski, and Sachin Agarwal. Fast pda synchronization using characteristic polynomial interpolation. In *Proc. INFOCOM*, pages 1510151–9, 2002.
- [25] Cpisync, characteristic polynomial interpolation for fast data synchronization., <http://nislalab.bu.edu/CPISync.html>.
- [26] John Byers, Jeffrey Considine, and Michael Mitzenmacher. Fast approximate reconciliation of set differences. In *BU Computer Science TR*, pages 2002–19, 2002.
- [27] Dar Itay, Milo Tova, and Verbin Elad. Optimized union of non-disjoint distributed data sets. In *EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology*, pages 12–23, New York, NY, USA, 2009. ACM.
- [28] Minsky Yaron and Trachtenberg Ari. Efficient reconciliation of unordered databases. Technical report, Ithaca, NY, USA, 1999.
- [29] Barthelemy Marc. Betweenness centrality in large complex networks, May 2004.
- [30] Gnuzip. <http://en.wikipedia.org/wiki/Gzip>.
- [31] Pkzip application note. <http://www.pkware.com/documents/casestudies/APPNOTE.TXT>.
- [32] Python. <http://www.python.org>.
- [33] Twisted python. <http://twistedmatrix.com/>.
- [34] Sqlite. <http://www.sqlite.org>.
- [35] Michel Meulpolder, Johan A. Pouwelse, Dick H.J. Epema, and Henk J. Sips. Bartercast: A practical approach to prevent lazy freeriding in p2p networks. In State University of New York Yuanyuan Yang, editor, *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, Los Alamitos, USA, May 2009. IEEE Computer Society.