Towards Reliable User Collaboration over Cloud-based File Synchronization System: Dropbox as a Case Study

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Bharath Kumar Bommana

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Dr. Haiyang Wang

July 2016

Acknowledgements

First and foremost, I wish to express my sincere gratitude to my adviser, Dr. Haiyang Wang who supported and guided me throughout this thesis with his unmatched patience and immense knowledge. Without his continuous encouragement and effort, this thesis would not have been possible. I couldn't have asked for a better adviser and mentor for this thesis.

I would like to thank Professor Douglas Dunham and Dr. Xuan Li for serving on my thesis committee. Also, I would like to thank all the faculty members at University of Minnesota, Duluth, Computer Science department and fellow graduate students for their support during my of graduate study.

Lastly, I would like to thank my friends and family for their love and support throughout my life.

Dedication

I would like to dedicate this thesis to my parents, Prathap Reddy Bommana and Varalakshmi Bommana, and my brother, Sharath Kumar Reddy Bommana. Without their love and support, I would not have been able to achieve anything.

Abstract

Recent advances in cloud technology have turned the idea of Cloud Storage into a reality. Over the past few years, cloud-based file storage/synchronization systems such as Dropbox, Google Drive and One Drive, have gained tremendous success among Internet users. The cloud storage systems, most notably Dropbox, not only provide reliable file storage but also enable effective file synchronization with diverse user collaborations. It is thus interesting to explore if this cloud-based file storage system can provide reliable collaboration service for the users.

In this thesis, we, for the first time, collect samples of content shared on Dropbox and analyze it. We then take our first step to understand the reliable user collaboration over cloud-based file storage systems using the popular Dropbox as a case study. Our observations indicate that users' updates would be discarded without any warning in a classic collaborative file editing session on Dropbox like file storage systems. In particular, to ensure a reliable service, the users need to wait for over 40 seconds between two updates when their uploading capacity is lower than 100 KBytes/sec. This already exceeds the auto-saving interval of many real world applications and severely hinders the system applicability. We further investigate the root causes of this phenomenon and find that such a problem is due to the three-layer design(i.e. user client, cloud virtual machines and cloud storage) of cloud storage systems. To address this problem of losing updates in multi-user collaborative file editing session, we propose an enhancement to the system that keeps track of and saves all the user updates locally and provides the opportunity for the recovery of any historical updates. Our proposed enhancement ensures that none of the user updates will be lost during a file editing session irrespective of the time interval between the updates and thereby making Dropbox user collaboration more resilient and less prone to potential data loss.

Contents

List of Tables

List of Figures

1 Introduction

Over the past few years, cloud-based file storage/synchronization systems have achieved tremendous success among Internet users. This new generation of service has been proved to be more effective and robust compared to the conventional client-server or the self-scalable peer-to-peer file synchronization systems. The cloud-based file storage/synchronization systems such as Dropbox [10], Google Drive [16] and One Drive [33] provide not only reliable file storage but also file synchronization for diverse user collaborations. Different from dedicated collaboration applications such as SVN [35], the cloud-based file storage systems such as Dropbox do not require users to carefully monitor their local updates. In particular, the users are not required to take care of version control; the cloud storage will automatically keep their historical or even conflicted updates. Such a feature gives users the opportunity to recover any historical updates [20]. However, most cloud-based file storage systems are not dedicated for user collaboration, the reliability of their version recovery function remains largely unclear.

In this thesis, we study the content shared on Dropbox through a real-world measurement study. For our measurement analysis, we collected a number of Dropbox content links shared on Twitter over a week and analyzed the content. We find that majority of the files shared on Dropbox are relatively small in size and are mostly documents. We then investigate the reliable user collaboration on cloud-based file storage systems through a measurement analysis using Dropbox as a case study. Through our measurement study, we find that the Dropbox-like storage systems will discard users' updates without any warning in a collaborative file editing session. The drop rate is unfortunately related to the slow-

est (in terms of throughput) collaborator in the system, which severely hinders the system scalability. To ensure reliable file update, a user may have to wait a certain amount of time between each operation. For example, when user's uploading capacity is at 100 Kbytes/sec, he/she will have to wait for over 40 seconds between two updates even when he/she is the only user in the system. This already exceeds the auto-saving interval of many real-world applications. To make matters worse, our experiments further indicate that such a *safe interval* ramps up very quickly with increase in the number of users in the system under various network conditions.

It is worth noting that such a problem can be hardly observed in traditional peer-to-peer and Content Delivery Network (CDN) based storage systems. We therefore further investigate the root causes of this issue and reveal that the missing update is due to the cloud-based storage deployment. In particular, it is known that the user clients do not directly communicate with the Amazon S3 storage systems. The clients first upload their files to a set of Amazon EC2 VMs (Virtual Machines) and the EC2 VMs upload the files to other users and the S3 storage [9]. This three-layer design enables seamless connection between users and cloud storage. It however also introduces extra queuing costs on the EC2 VMs. In an extreme case, when we are synchronizing a large file from a slow client, the data transmission between EC2 and S3 will be prolonged by the limited uploading capacity on the user side. While EC2 is updating this file to S3, other conflicting operations (on this file) will be cached on the EC2 VMs and wait for future process. However, EC2 cannot provide unlimited queuing space for these conflicting updates. Random drop will happen when the queue is full on the EC2. Our follow-up discussion further investigates this design issue on the cloud file storage system and offers hints for practical improvement.

The rest of the thesis is structured as follows: In chapter 2, we present the background and related works. Chapter 3 discusses how measurements studies are configured to analyze the content shared on Dropbox and to investigate the Dropbox file synchronization

behavior in a collaborative file editing session on the PlanetLab testbed. Chapter 4 presents the measurement results and analysis of both the studies. In chapter 5, the details of the proposed system enhancement are presented. Chapter 6 concludes the thesis and provides details for future work.

2 Background

Over the recent years, large-scale data sharing and file synchronization have gained tremendous popularity. In this chapter, we will review the existing studies of content sharing and file synchronization. We start with the traditional Client-Server Model and then proceed through the self scalable Peer-to-Peer Model and then to the much more robust Cloud-based file storage/synchronization model.

2.1 Client Server Model

The Client Server Model is one of the earliest implementations of file sharing and content delivery between two systems over the Internet. Any computer in a network can act as either a client or a server. In a conventional Client/Server model, the server is a centralized component - providing data storage and content delivery to the various clients as shown in Figure 2.1. Clients make a request to the server for content and the server receives the request, processes it and responds to the client either by providing the client with the content requested or by denying the service to the client. The Client-Server relationship is often described as request-response relationship.

Each server can handle or service multiple clients. The communication between a client and the server is independent of the communication between the server and other clients. Also, the client need not be aware of the physical location of the server. Figure 2.2 gives an example of one such server that handles multiple clients, a computer, a tablet, a mobile and a laptop. The major reason for the popularity of Client/Server paradigm is its simplicity

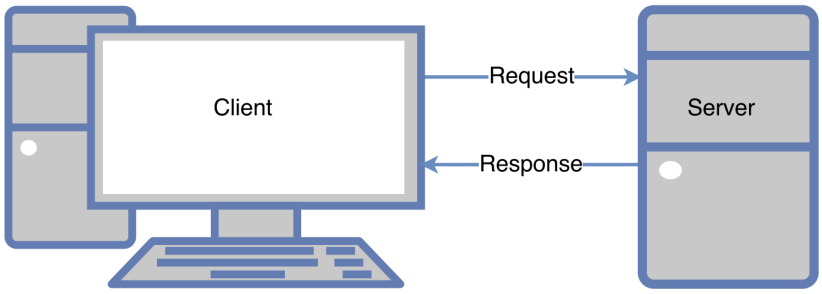

Figure 2.1: Client Server File Delivery System

and ease of deployment.

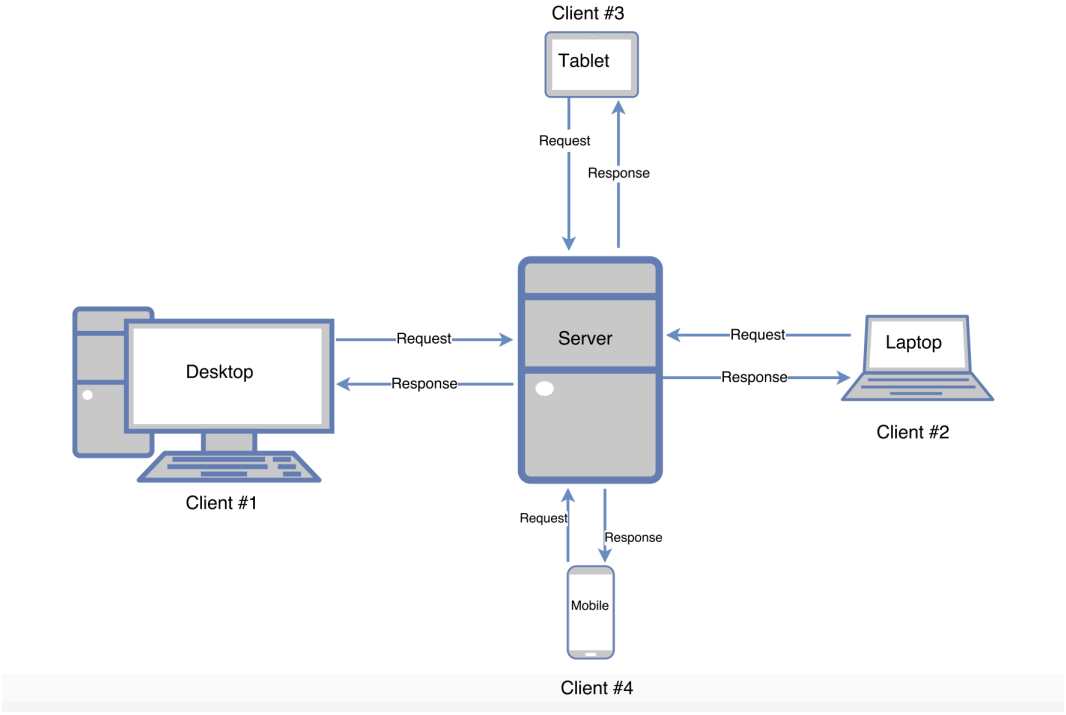

Figure 2.2: Client-Server Model with multiple clients

Content delivery and file sharing between the client and the server is achieved through the application layer protocols such as File Transfer Protocol (FTP), Hyper Text Transfer Protocol (HTTP) and Simple Mail Transfer Protocol (SMTP).

File Transfer Protocol:

FTP facilitates the file exchange between the server and a client over a computer network. FTP is based on client server architecture and uses TCP/IP network protocols to enable data transfer. An FTP connection is made on port 21 between a client and the server. FTP allows a client to either download a file from the server or to upload a file to the server. An FTP communication between a client and server is done in two connections- (1) the control connection and (2) the data connection. The control connection enables the client to connect to the server and send commands or requests whereas the data connection is used to send files over the network.

FTP connections are often made secure by using Secure Socket Layer (SSL) or Transport Layer Security (TLS). The users have to provide authentication using a sign in protocol- an username and a password for instance. However, FTP connections can be configured to allow anonymous login as well.

Hyper Text Transfer Protocol:

Hyper Text Transfer Protocol, or shortly, HTTP is the underlying protocol used in the World Wide Web (WWW). It is a request-response protocol and allows the transfer of text along with multimedia files like images, videos, etc. between client and server. HTTP is implemented on port 80 on the server by default and the client request for the content from the server by initializing a TCP connection request on port 80 of the server. The server responds to the client either with a "HTTP/1.1 200 OK" status message along with the body of the content requested or with an error message if the content is not found.

Simple Mail Transfer Protocol:

SMTP, short for Simple Mail Transfer Protocol is the protocol used for sending emails from a mail client to a mail server. SMTP is implemented on TCP on port 80. SMTP as the name suggests can only handle simple text and cannot handle graphics, fonts, attachments etc. SMTP uses a mechanism called "Store and Forward" to deliver the mail from the client

to the destination server. Each mail passes through a multitude of intermediate computers over the network since its inception at the client and delivery to the server. The mail is briefly stored at each of these computers before it is forwarded to the next one. SMTP is the basis of almost all email systems that facilitate exchange of emails over the Internet.

The major advantage of using a client server model is its simplicity and ease of deployment. However, the workload of the server increases with an increase in the number of clients. To handle the increased number of clients, one may have to deploy a powerful server leading to a greater infrastructure and maintenance cost. It is evident that the server capacity is a severe bottleneck in Client Server Delivery Model; scalability is also a major issue. It is quite obvious that a Client Server design cannot deliver as expected in the event of flash crowds- the scenario in which a large number of users send resource requests to the server, thereby creating high traffic on the network. The existing proposals such as using large server farms [15] or efficient load balancing infrastructures [30] or using content-caching proxies [4] are still not scalable in terms of maintenance costs especially during flash crowds. Hence, the need for a scalable content delivery model to fulfill the demands of large-scale systems.

2.2 Peer-to-Peer Model

Peer-to-Peer content distribution framework has gained enormous popularity over the recent years due to a variety of factors such as high scalability, robustness and good performance. The fundamental difference between Client Server Model and Peer-to-Peer (P2P) model is that a computer in client server model can act either as a server or as a client but not both whereas, in a P2P model, it can act both as a client and a server at the same time. Once a peer downloads the file, it is often made available for upload to other peers. This ability to use the uploading capacity of the peers makes peer-to-peer system highly scalable

and self-organizing; the peers that have downloaded the file will handle the new requests made for this file. In a P2P model, a file is usually divided into small segments called pieces to facilitate easy transmission and the peers trade these pieces to get the complete file. P2P Model can be viewed as a virtual overlay network of equal peers that are capable of communicating with each other and trading (file) pieces between them.

In this section, by using BitTorrent Protocol as a case study, we will attempt to understand the basics of P2P file sharing model.

2.2.1 Overview of BitTorrent

BitTorrent is a popular P2P content distribution protocol designed by Bram Cohen in 2001, to distribute large files over the Internet[8]. It is known to scale well with the number of participating clients, reduce the load on the congested server and increase download rates for the users. In BitTorrent, the peers interested in downloading the same content form an overlay network are usually referred to as 'Swarm'.

When a content publisher wants to distribute a (large) file over the Internet, he/she has to provide a metadata file called .torrent file associated with the content and then upload it to a BitTorrent Server. A .torrent file contains information about the file, its size, details of the chunks, hashing data and URL of the tracker. When a peer wants to download this file, he/she has to download the .torrent file from one of the BitTorrent Servers first. The tracker in .torrent file helps the peers to find each other. When a peer attempts to download the file, the tracker gives the list of all peers that are downloading the same file. Every file in BitTorrent will be divided into pieces of 0.25 MB each and by exchanging 'HAVE' messages every peer gets to know the information of pieces its neighbors hold. Every time a peer downloads a new piece, it notifies its neighbors by sending out a 'HAVE' message.

BitTorrent (BT) uses Piece Selection algorithm to determine which piece to download

and choking algorithm to reward the uploading peers with better downloading rates. Rarest First Algorithm is used to determine which piece to download- the piece that is with the least number of peers in the swarm. The Choking algorithm uses Tit-for-Tat by unchoking the top four peers with high upload rates and to select the remaining one peer, BT uses optimistic unchoking(unchokes a random peer irrespective of its upload or download rates). This optimistic algorithm helps the newly joined peers to get started with a decent download rate [8].

Since the peers in P2P applications share a portion their resources such as processing power, network bandwidth with the other peers, the need for having a powerful centralized server is greatly reduced thereby reducing the cost of operation. This is what makes P2P content delivery model highly scalable. However, P2P application accounts for a great share of the present day Internet traffic and may result in unprecedented amounts of highly undesirable cross-ISP traffic.

2.2.2 Related Studies of P2P based File Synchronization

There have been numerous studies on the analysis, implementation and enhancement of P2P applications such as BitTorrent.

Bran Cohen gives an outline of the incentive based approach “Tit for Tat” used by BT to increase the robustness of the protocol. The peers that upload pieces to other peers will be rewarded by giving priority when it requests another peer for a piece[8]. However, Piatek et al., through real world measurements, demonstrates that a strategic peer can trick BT to improve its download performance without contributing the upload bandwidth[29]. The ubiquity, resilience and scalability provided by P2P systems makes them ideal for large scale distributed applications. The prospects of P2P design based live streaming [2] and P2P-Video On demand [23] have been greatly explored.

Most of the P2P applications employ an arbitrary peer selection policy that ignores the underlying Internet Topology and ISP link costs, establishing connections between random peers from around the world. Such a policy, while improving the performance of the P2P applications. also generates huge amounts of cross ISP traffic which is highly expensive. Choffnes et al. in [6] proposes that by employing a biased neighbor selection policy, the cross ISP traffic generated by a P2P application can be greatly reduced without sacrificing its performance or robustness.

P2P design as a file synchronization model is lately gaining popularity. P2P based applications such as BitTorrent Sync (BTSync) [3], Hive2Hive [19] and Storj [34] have seen remarkable growth in their user base. BTSync allows the users to sync files across devices using a P2P based model. It is a decentralized model, and the user need not upload the content on any third party storage (as is the case with cloud based file sync models). Farina et al. in [14] provides various functionalities that BTSync offers and studies the underlying technologies in BTSync and also provides its forensic analysis. BTSync uses Distributed Hash Table (DHT) that allows the completely decentralized discovery of peers associated with sharing a particular piece of content. Furthermore, the data is encrypted during the network transit and decrypted and reassembled when it arrives at the destination.

Hive2Hive [19], formerly Box2Box [24], is an open source library for secure, P2P based file synchronization and sharing. In addition to the functionalities provided by BT Sync, Box2Box also supports features like friend lists, recommendations and versioning. AeroFS [1] is another P2P based file synchronization application, unlike others it uses a centralized component and syncs files between the users without storing them online. It is mainly used by companies that need a secure file backup and synchronization service.

While the P2P system offers considerable advantages, it suffers from high node churn rate i.e. nodes arrive and depart any time they want, making the P2P system not completely reliable. The P2P file synchronization model requires the complete version of the file to be

residing on at least one of the participating peers. There is a chance that a malicious peer can attempt to corrupt, delete or deny access to the file pieces that are distributed between nodes.

2.3 Cloud Based File Synchronization Model

There is a tremendous growth in the popularity of cloud computing in the recent years both in industry and research. In a cloud-computing model, users can outsource their computation and storage to cloud service providers and accordingly pay for their services. In contrast to the conventional Client/Server model, cloud-computing model offers phenomenal advantages in terms of saving money that is traditionally spent on purchasing and maintaining the hardware and computational resources.

Cloud storage services such as Dropbox, Google Drive and SkyDrive have become extremely popular as tools that provide personal storage, data sharing, file synchronization and users with ubiquitous and reliable data storage. The users' data stored in cloud storage is automatically synchronized across multiple devices and can also be shared among a group of users.

In general, a cloud storage service is characterized by two components: a client application running on the user's device and a storage service that runs in the background and resides in the cloud. The user can manipulate (create, edit and delete) a file in a local folder called "sync folder" and these changes are synchronized to the cloud automatically by the client application. This synchronization process takes place in a series of events such as data index, data content, sync notification, sync status, sync statistics and sync acknowledgment [26]. All these data sync events generate a network traffic termed as "data sync traffic". Figure 2.3 gives an overview of the Cloud-based file synchronization principle.

In the next section, we provide an overview of Dropbox to understand the architecture

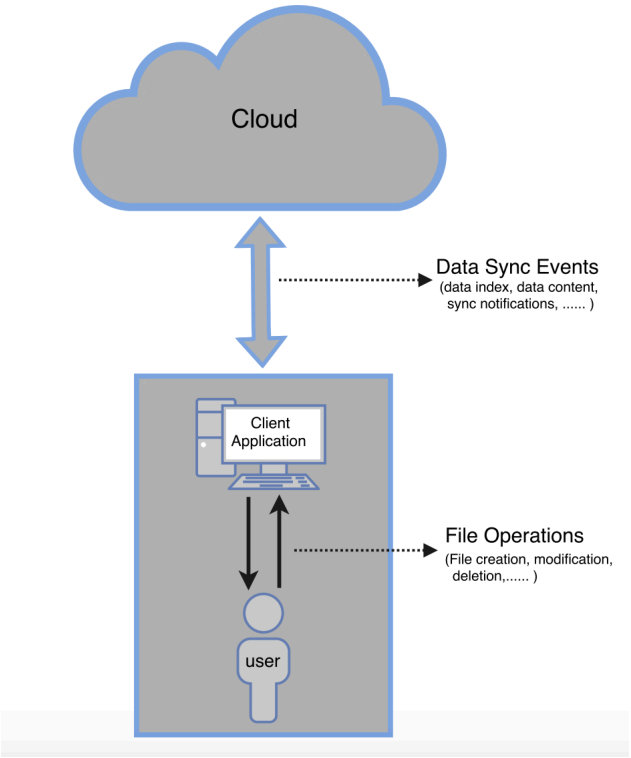

Figure 2.3: Data Synchronization Principle

of Cloud-based File Synchronization Model.

2.3.1 Overview of Dropbox

Among the cloud storage service providers, with more than 500 million registered users, Dropbox is the most popular one. Another fact that is worth mentioning here is that more than 1.2 billion files are synced using Dropbox every day [12]. In addition to online cloud storage, Dropbox provides functionalities to share, edit and synchronize files. The main operations in Dropbox file synchronization service are detection and transmission.

Detection

The client application running on the user's machine should be able to detect any changes made, both on the cloud and the local file system and synchronize these changes. Dropbox relies on Linux's inotify to detect any changes made on the local file system and sends these changes to the cloud. If changes are made in the remote cloud, Dropbox uses push-based notifications to synchronize these changes across all shared devices. If the changes are made when the Dropbox client is offline, such changes will be stored in the local databases along with the last modification time. When the client goes online (rebooted), these changes will be uploaded to the cloud based on the time stamp value.

Transmission

Dropbox splits a file into a number of chunks of size 4MB each. When a new file is added to a local Dropbox folder, the file is split into chunks and each chunk is assigned a hash value (calculated using SHA-256 algorithm) to prevent redundant file uploading. While sending the chunks to the servers, Dropbox uses a DE-duplication technique to reduce the network traffic being generated. It first sends the hash values of the chunks to the destination folder and if a chunk with a certain hash value is already present in the destination folder, that chunk will not be transmitted over the network, thus reducing the network traffic by avoiding redundant transmission. If there is only partial change, Dropbox relies on rsync utility to transmit only modified portions of each chunk. To accomplish atomicity at the client, Dropbox first downloads a chunk to a staging area, gathers them and then moves them to the destination location.

The architecture of Dropbox is composed of two servers: Control servers and Storage servers. Control Servers are managed by Dropbox Inc. whereas the Storage Servers are managed by storage provider- Amazon Elastic Compute Cloud (EC2) and Simple Storage

Service (S3). Dropbox relies on Amazon S3 systems for cloud storage.

Dropbox Framework

Dropbox framework comprises of three components on the server side: 1) Dropbox Load Balancers 2) Dropbox delivery servers (on EC2) 3) Dropbox Storage Servers (on S3). Figure 2.4 illustrates the framework of Dropbox and components.

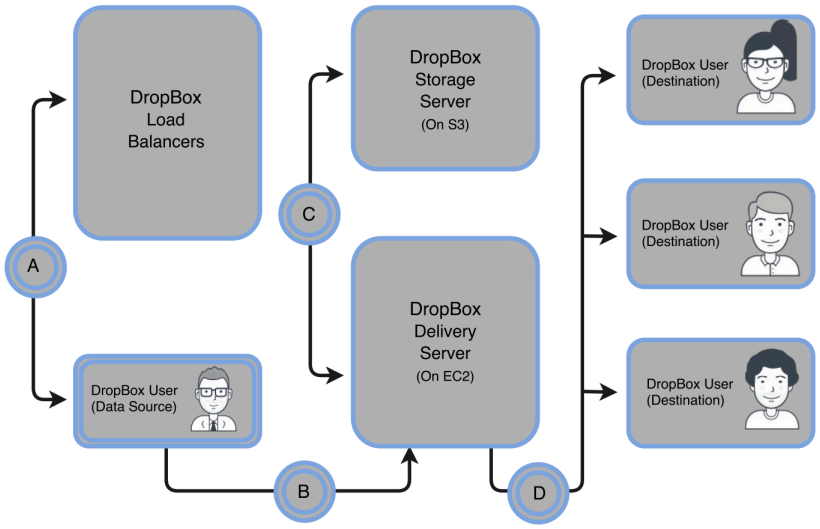

Figure 2.4: Dropbox Framework

The first component, Load Balancers are deployed by Dropbox and the second, Dropbox delivery Services that deploy thousands of Amazon EC2 instances for data uploading, downloading and file processing functions. The third component, Dropbox Storage Servers (cluster of Amazon S3 servers) is for storing the uploaded files.

The following steps are followed during a data flow from a source (user) to the destination Dropbox folder, when he/she uploads a file.

- **Step A**: The user (data source) sends a DNS request querying the IP addresses of the load-balancers of Dropbox. The DNS server sends out a list of available load-

balancers as the response. The Data Source randomly identifies a load balancer from the list and sends out meta information of the file to it. The selected load balancer assigns an EC2 server to the Data Source.

- **Step B**: The data source uploads the file to the designated EC2 server.
- **Step C**: Once the file is successfully uploaded, the EC2 server forwards this file to the Storage folders (S3).
- **Step D**: To synchronize these changes in all the destinations, another EC2 server is employed.

EC2 acts as a bridge between S3 storage servers and client applications. The De-duplication technique is applied in step A to avoid uploading redundant chunks and thereby reducing data sync traffic. This technique is referred to as a "Caching function".

2.3.2 Related Studies of Cloud-based File Synchronization Model

Pertaining to the popularity and enormous user base of Dropbox and other cloud based storage service providers, a lot of research has been done in understanding their file synchronization techniques.

Efforts have been made to understand the architecture of Dropbox file synchronization model. Goncalves et al. [18] presented an extensive study on Dropbox client behavior by collecting the data from three different campuses and analyzing the client processes. In another study, Drago et al. [9] presented a detailed architecture of Dropbox services and measurement studies that show Dropbox as the most widely used personal cloud storage service provider. They also presented the peculiarity of Dropbox while committing changes and discussed the possible performance bottlenecks in Dropbox. Through their extensive measurements they were able to identify scalability issues in Dropbox resulting from system

design choices. They also provided a comparison of system capabilities of various cloud based file synchronization systems. Wang et al. [37] presented a measurement study to understand the design and performance bottlenecks in the Dropbox system. Their study identifies its hybrid design with Amazon EC2 for computation and Amazon S3 for the storage. They found that this architecture helps in seamless file synchronization but induces a delay because of the Virtual Machines.

Quality of service provided by various cloud providers has been studied through measurement studies by a multitude of researchers. Li et al. [25] developed a tool that compares the services provided by various public cloud service providers such as Amazon AWS, Windows Azure, Google App Engine and Rackspace; Amazon EC2 was observed to be more suitable to handle large data objects rather than small data objects. Casas et al. [5] evaluate the performance of various personal cloud storage services like Dropbox, Google Drive with the help of a group of users and analyzing the Quality of experience. Hu et al [22] presented a comparison of performance attributes of four popular cloud storage services: Dropbox, CrashPlan, Mozy and Carbonite, to see if they live up to the users expectations. Their measurement study was focused on the backup and restore performance, data loss liability and data privacy.

Numerous studies have been made to study the effectiveness of the file synchronization functionality and ways to improve its performance. Li et al [27] studied the traffic overuse problem- the enormous amount of traffic generated by attempts to sync frequent, short updates to user data. They propose an update-batched delayed synchronization (UDS) to reduce the generated network traffic. UDS as the name suggests batches the updates from clients by acting as a middleware between users' file system and cloud storage system. However this approach results in an unsynchronized state among the machines connected to the account or with the destination on cloud storage.

While Cloud based storage models offer so many advantages compared to a traditional

storage model, the issues of data protection, security and reliability cannot be ignored. Since the cloud storage provider uses third party storage services, there has always been a concern about data protection and privacy. Zhang et al. [38] studied the possibility of corruption propagation in Dropbox–like file synchronization services and discussed the approaches to mitigate this problem. The Dropbox client may propagate undetected data corruption to the servers and thus pollute the copies on all the machines connected to the account.

Different from existing studies, our work focuses on the reliability of user collaboration over cloud file storage system using Dropbox as a case study, along with collecting and analyzing the content shared online by the Dropbox users. In the next chapter, we provide measurement configurations of both the studies.

3 Measurement Configurations

As mentioned earlier, this thesis is a two fold process. The first study is to understand the content shared on Dropbox through a real world measurement and the second is to investigate the reliability issue of cloud-based storage system using Dropbox as a case study, in a collaborative file editing session. In the subsequent sections, we present the details of our measurement configurations.

3.1 Analyzing Content Shared on Dropbox

In order to analyze the content shared on Dropbox, we ran a simple two-step experiment. The first step was collecting a number of Dropbox public shared links and the second was to download the content and analyze the files. The hardware configuration of the machine used is 3.10 GHz CPU with 4 cores, 8 GB memory and a bandwidth of 10 Mbps. Python is used as the Programming Language for writing related scripts.

Collecting Dropbox public shared links

If a Dropbox user wishes to share a file with a group of friends or colleagues, he can do so by placing the file into public folder in Dropbox and share the generated link via email or text message. People who click on the link will be able to preview the file and make changes if given necessary permissions.

We wanted to collect as many Dropbox public shared links as possible for this study. Our first idea was to search for links that contain the string *www.dropbox.com/s* on Google and collect the links. However, we could only get 280 links from Google, out of which only

Listing 3.1: Python code snippet to authorize the application

```
import tweepy
consumer_key = "*******"
consumer_token = "*******"
auth = tweepy.AppAuthHandler(consumer_key, consumer_secret)
api = tweepy.API(auth, wait_on_rate_limit=True,
                 wait_on_rate_limit_notify=True)
if(api):
    print "successfully connected"
    /*We can make the function call api.search(q='dropbox.com/s')
    and extract links from the tweets collected.
    call tweetSearch() here */
else:
    print "cannot authenticate"
```

105 were active. This was not very useful for our study. So we used Twitter Search API to get the desired Dropbox public links. The idea was to search for the tweets that contain the string *www.dropbox.com/s* and extract the complete Dropbox link from the tweet.

Twitter provides REST SEARCH API for searching tweets from up to as far back as 7 days. One has to register an application on *dev.twitter.com* to get authorization access and keys to download tweets using TWITTER SEARCH API. We used python and tweepy python library for accessing the twitter API. We used Application only Auth to access the public information on twitter. The code snippet Listing 3.1 authorizes the application to be able to use twitter search API. Since we are using Application only auth, we can practically get 45,000 tweets in a 15-minute window.

Once the application is successfully authorized, we can start collecting the tweets by calling the function tweetSearch(), a python function that collects the tweets containing the given string (results from api.search() call). The tweets obtained were in JSON format and we used JSON PICKLE to extract the links from the tweets. This function collects only unique Dropbox links and writes them into a file. The code snippet for the fuction tweetSearch() is shown in Listing 3.2.

Listing 3.2: Python code snippet to collect links from twitter

```
import tweepy, jsonpickle
def tweetSearch():
    searchQuery = "dropbox.com/s/"
    maxTweets = 10000000, tweetsPerQuery = 100
    fname = 'tweets.txt'
    tweets, links = ([]for i in range(2))
    sinceId = None, max_id = -1L
    tweetCount = 0
    with open(fname, 'w') as f:
        while tweetCount < maxTweets:
            try:
                if(max_id <=0):
                    if(not sinceId):
                        new_tweets = api.search(q=searchQuery,
                                        count=tweetsPerQuery)
                    else:
                        new_tweets = api.search(q=searchQuery,
                        count=tweetsPerQuery, since_id=sinceId)
                else:
                    if(not sinceId):
                        new_tweets = api.search(q=searchQuery,
                    count=tweetsPerQuery, max_id=str(max_id-1))
                    else:
                        new_tweets = api.search(q=searchQuery,
                count=tweetsPerQuery, max_id=str(max_id-1),
                since_id=sinceId)
                if not new_tweets:
                    break
                for tweet in new_tweets:
                    try:
                      for url in tweet._json['entities']['urls']:
                            link = url['expanded_url']
                            if "dropbox.com/s/" not in link:
                                pass
                            else:
                                links.append(link)
                    except KeyError as e:
                        pass
                tweetCount +=len (new_tweets)
                max_id = new_tweets[-1].id
            except tweepy.TweepError as e:
                break
    return set(links)
```

Download and analyze the content shared on Dropbox

The above script produces a file with all the unique Dropbox links collected. We used wget command to download content from all the links in the file as shown below. The content downloaded from Dropbox is analyzed and results are shown in Chapter 4.

```
wget -i links.txt
```

3.2 Reliability of User Collaboration

In this section, we present the measurement and configuration details of our second study. We focus mainly on PlanetLab [31] and related scripts. Python and bash were used for collecting the measurements. Python code was run on the local machine which triggered the bash script that is run on a remote planetLab node. The results from these experiments run on remote PlanetLab nodes were retrieved and stored on a local machine.

3.2.1 PlanetLab

PlanetLab is a network of computers set up across all the continents for research purposes; to test and develop new network services[7]. It serves as a test bed for users to perform large scale Internet studies. PlanetLab is a powerful tool for computer networking and distributed systems research as it runs over the common Internet routes and span nodes across the world. The underlying objective in conducting experiments on PlanetLab is to obtain real world data, which is far more realistic compared to results from a simulation.

Terms related to PlanetLab

Site: Site refers to the physical location where the PlanetLab nodes are located (e.g. University of Minnesota or HP Labs). There are currently 717 sites in PlanetLab. The details of your site can be seen under "My Site" tab in the PlanetLab account.

Node: A node is a dedicated server that runs components of PlanetLab services. These are the actual computers provided by the participating universities and organizations. In exchange for hosting a few nodes(servers), participants obtain access to a share of resources across the entire PlanetLab platform for deploying and evaluating planetary scale network services[28]. PlanetLab currently consists of 1353 nodes at 717 sites[31]. Figure 3.1 illustrates the geographical distribution of PlanetLab nodes. PlanetLab nodes run on Linux operating systems Fedora 8 and Fedora 14. These nodes can be managed only through OpenSSH protocol for security reasons.

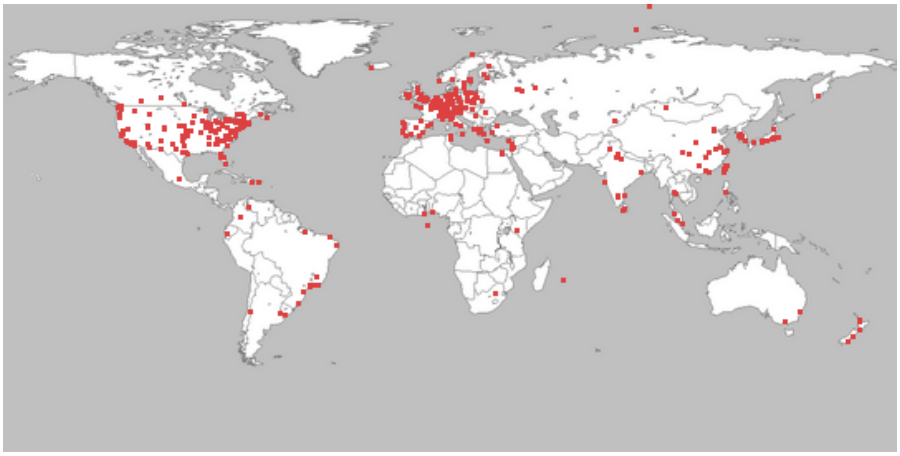

Figure 3.1: Geographical distribution of PlanetLab Nodes.

Slice: Peterson and Roscoe[28] identified *slice* as the centerpiece of PlanetLab architecture. Each slice is associated with some amount of CPU processing, memory, disc space and network bandwidth resources across a set of PlanetLab nodes. PlanetLab Principal Investigators are responsible for creating slices and assigning them to the users. Once a slice is assigned, the user can start adding nodes to her/his slice. After adding the nodes, virtual servers are created on each node for that slice. A slice may be viewed as a horizontal cut of global PlanetLab resources and each slice is completely isolated of other slices. It is a network of virtual machines, with a set of local resources bound to each virtual machine. Slices have a limited lifetime and must be renewed periodically, typically every two months.

Sliver: The set of allocated resources on a single node is called Sliver.

Virtualization: Instead of provisioning a physical node, PlanetLab assigns virtual machines to the users using virtualization technology. Each virtual machine runs on a single physical node and uses a fraction of its resources. Originally, Slivers were implemented as Linux-Vservers, which provide both performance and namespace isolation among the slivers on a node. Recently, PlanetLab has migrated to LinuxContainers(LXC), which is a fast and robust virtualization mechanism in the Linux Kernel. It is a container based, OS level virtualization and is natively supported by Linux[32].

The first and foremost thing to do to start accessing PlanetLab nodes is to register for a PlanetLab account and have a slice assigned to that account. To login to the slice, the user has to create an SSH key pair for RSA authentication. The user has to upload the RSA public key to the PlanetLab website under *keys* in *My Account* tab and safe-keep the private key. Once the key is uploaded, it takes 48 hours for the user to log into PlanetLab nodes via SSH protocol. The command used to connect the PlanetLab node *planetlab1.dtc.umn.edu* on the slice *umn_bharath01* is shown below. The corresponding private key needs to be supplied in the command.

```
ssh -l umn_bharath01 -i ~/.ssh/id_rsa planetlab1.dtc.umn.edu
```

Python Code for Connecting and Authenticating a PlanetLab user:

The PlanetLab Central API(PLCAPI) is the interface through with users access information about the nodes, sites and slices. The API should be accessed via XML-RPC over HTTPS. As shown in Listing 3.3, the python module authenticate() creates an auth structure using which the AutoCheck method of api_server validates the users' PlanetLab account. The example below shows *AuthMethod password* specification.

Retrieving information from PlanetLab nodes:

To retrieve the information about nodes, we can use GetNodes() method of api_server.

Listing 3.3: Python module to validate a user's PlanetLab account

```
import xmlrpclib

def authenticate():
    api_server = xmlrpclib.ServerProxy('https://www.planet-
        lab.org/PLCAPI/', allow_none=True)

 '''The first parameter to each XML-RPC call is an
    authentication structure. The code below shows how
    to set up this structure for password-based
    authentication.
 '''
    #Create an empty dictionary (XML-RPC struct)
    auth = {}
    # Specify password authentication
    auth['AuthMethod']='password'

    # Username and password
    auth['Username'] = 'sampleEmailAddress@host.com'
    auth['AuthString'] = '********'

'''Now we can verify this structure with the PlanetLab Central
    (PLC) API method AuthCheck(), which returns 1 if the
    authentication structure is valid.
'''
    authorized = api_server.AuthCheck(auth)

    if authorized:
        print 'You are authorized to use PlanetLab!'

    return (api_server, auth)
```

This method takes three arguments viz. an auth structure, a node_filter(optional) and a return_fields(optional). This method returns a list of nodes along with information about the nodes like node ids, host names, node status, etc. If the optional filters are not specified, as shown below, the method returns information about all nodes.

```
all_nodes = api_server.GetNodes(auth)
print all_nodes
```

The second argument(node_filter) is useful when the user wants to get information about a few selected nodes. The below code snippet in 3.4 returns information about the two nodes at University of Minnesota.

Listing 3.4: Python module to get information about few specified nodes

```
1 #Get information about two nodes at University of Minnesota.
2 minnesota_nodes = api_server.GetNodes(auth,
3       ['planetlab1.dtc.umn.edu' , 'planetlab2.dtc.umn.edu'])
4 print minnesota_nodes
```

As such, GetNodes() method returns a lot of information about each node. The third argument(return_fields) is useful to filter the information returned, based on the named fields. The code snippet shown in listing 3.5 returns only *node ids* and *hostnames* of each node with boot state as *boot*.

Listing 3.5: Python module to get node IDs and hostnames of nodes with boot state as boot

```
1 #Get node IDs and hostnames of nodes whose boot state is "boot"
2 boot_state_filter = {'boot_state': 'boot'}
3 named_fields = ['node_id', 'hostname']
4 nodes_with_boot_status = api_server.GetNodes(auth,
5   boot_state_filter, named_fields)
6 print nodes_with_boot_status
```

The code shown above lists out information about all the nodes. If the user wants to get only those nodes that are associated with a particular slice, GetSlices() method must be used. The code snippet in listing 3.6 illustrates how to get information about nodes associated with a slice.

Listing 3.6: Python code to print all the nodes associated with the slice

```
slice_name="sliceName"
#Get the node ids that are assigned to the slice
node_ids = api_server.GetSlices(auth, slice_name,
                    ['node_ids'])[0]['node_ids']

#Get the hostnames that are assigned to the slice
node_hostnames = [node['hostname'] for node in
               api_server.GetNodes(auth, node_ids, ['hostname'])]

#get the complete information of each node which
#is assigned to the slice
node_info = api_server.GetNodes(auth, node_hostnames)

#boot_nodelist has all the list of nodes with the boot status
for node in node_info:
    if node['boot_state'] == 'boot':
         print node['hostname']
```

3.2.2 Parallel Node Control via Vxargs

Using PlanetLab, we can run our experiments on multiple geographically distributed nodes to get accurate results. But running the same experiment on multiple nodes serially, can be tedious and time consuming. The solution is to use Vxargs[36], a python application, that is inspired by xargs and pssh. Vxargs is helpful in monitoring a large set of machines on a wide area network; allowing users to perform parallel execution of a set of commands on the nodes. Vxargs also provides visualization of the progress in a curse-based UI and also saves results to a specified folder. With the use of Vxargs, the execution of commands on multiple PlanetLab nodes is done in parallel, which otherwise require serial execution.

The commands that are mostly used in PlanetLab nodes are ssh, scp, rsync and using Vxargs, they can run all the bash commands making it perfect for the job. To use Vxargs, you need to first make a text file with the list of target nodes, represented either by IP address or hostname and the nodes must be separated by new lines. Vxargs takes at least

two arguments viz. the file with list of nodes and the command you want to run on those nodes. It works by launching a number of parallel threads(specified by -P option) and each thread runs the command on one node at a time. Once a thread finishes the job at a node, it pulls up a new node from the remaining nodes in the list and executes the command on it. The timeout option(-t) can be used to free a thread in the case of slow nodes and assign it to another node. The list of nodes is passed with the help of -a option. As mentioned earlier, Vxargs offers output redirection and the destination directory for the results is given using -o option.

For example, the command below is used to run a command(ls) on all the nodes given in listofNodes.txt file.

```
1 # python vxargs -t 40 -a listOfNodes.txt -P 10 -o Results
2                 ssh slice_name@{}  ls
```

Vxargs standard output:

Vxargs writes three output files in the Results folder for each node viz. hostname.out, hostname.err and hostname.status(hostname as appeared in the text file). The output of the remote node is written to .out file. If the command fails to execute on the node or generates an error, the error message will be stored in .err file and the exit status of the command executed by Vxargs on that node will be stored in .status file. This redirection of standard output/error of each individual job on all the nodes is very useful to gather the complete information from the experiment results.

3.2.3 PlanetLab-based Deployment of Dropbox

To perform experiments and collect measurements, Dropbox needs to be deployed on a PlanetLab node. The following set of steps ensure that a Dropbox is up and running on a node.

1. First and foremost, the user has to successfully login to the PlanetLab node via SSH protocol. The user can do so by using the RSA private key and login to any node that has been added to his/her slice using as shown below.

```
# ssh -i ../.ssh/id_rsa umn_bharath01@pl1.tailab.eu
```

2. If the PlanetLab node is currently active, user will be connected to the node, otherwise the connection will be refused. Upon successful login, user has to download Dropbox package to the home directory on the node. The command shown below downloads the 32-bit package of Dropbox.

```
# wget -O dropbox.tar.gz
    "http://www.dropbox.com/download/?plat=lnx.x86"
```

3. The downloaded package should then be extracted.

```
# tar -xvzf dropbox.tar.gz
```

4. Dropbox daemon is run from the newly created .dropbox-dist directory.

```
# ~/.dropbox-dist/dropboxd
```

5. Since we are using Dropbox on this node for the first time, we will be asked to copy and paste a link in a browser to add this server to an existing Dropbox account by providing login details.

6. Dropbox provides a python script(dropbox.py) to control Dropbox from command line. Upon running this script, user should be able to start/stop Dropbox daemon using start/stop commands respectively or check its status of using status command.

The above sequence of steps must be performed on all the PlanetLab nodes we want to run our experiments on, to ensure the proper installation and setup of Dropbox. The following steps have to be followed to perform experiments and collect the measurements on a PlanetLab node. Figure 3.2 shows the flowchart of the process.

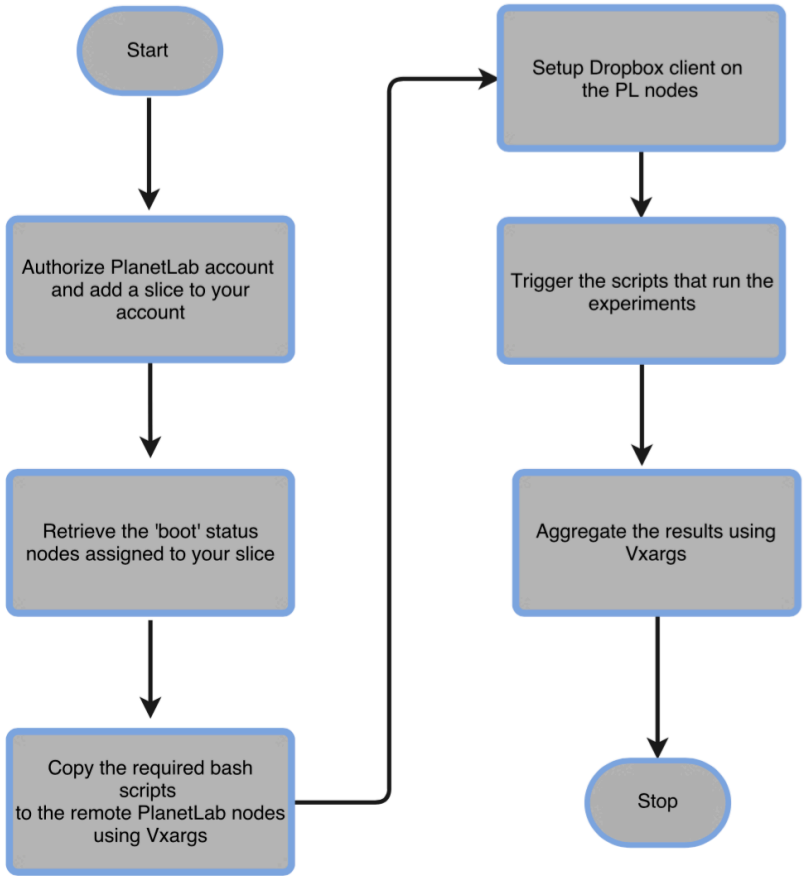

Figure 3.2: Flowchart for the framework to perform measurements on PlanetLab

1. As the first step in the flowchart suggests, user's PlanetLab account has to be authenticated via PLCAPI through username and password authentication. This enables the user to start using PlanetLab resources. Upon successful assignment of a slice to the user's account, he/she can start adding nodes to the slice.

2. Collect all those nodes associated with the slice, whose status is boot and are ready to be used for experiments. Write either IP address or host names of these nodes into a text file(listofNodes.txt).

3. Using Vxargs, copy all the required files into these nodes.

```
# python vxargs -t 30 -P 10 -a ListOfNodes.txt scp -o
  stricthostkeychecking=no dropboxScripts/* slice_name@{}
```

4. Set up Dropbox daemon on these nodes, as mentioned earlier. Also, check the status of the Dropbox daemon using *dropbox.py status* to ensure that the application is up and running.

5. Once the setup is ready, start the scripts that trigger experiments, as we discuss the details in Chapter 4.

6. The results produced on each node is aggregated on the local system using Vxargs.

Measurements are performed on multiple PlanetLab nodes to ensure that the results are unbiased and to precisely analyze the various performance bottlenecks in Dropbox file synchronization. The results aggregated from the nodes are accumulated on the local machine for analysis and safekeeping.

3.3 Practical Issues

Although PlanetLab provides a better platform for network based research, there have been numerous practical problems that we encountered while using PlanetLab.

1. PlanetLab nodes run on the Linux Operating System. A portion of them run on Fedora 8 and the rest run on Fedora 14. Fedora has recently released Version 24. These nodes

run on obsolete versions of the operating system and do not support some of the latest updates. It would have been easy to use these nodes, if they were running on latest versions of the Operating system.

2. Since PlanetLab nodes are shared among slices, there will be limitations on the usage of bandwidth and physical memory. The PlanetLab administrators would kill the process, at times, if these limitations were exceeded.

3. Using Vxargs allows multiple nodes to be controlled by triggering concurrent threads. But having these concurrent threads performing a bandwidth intensive task could easily make it exceed the system limitations thereby resulting in the killing of the process.

4 Measurement Results and Analysis

In this chapter, we provide details of the experiments we performed and the analysis of the results obtained. The experiment for understanding the content sharing on Dropbox was conducted on an Ubuntu Machine with the hardware configuration of 3.10 GHz CPU with 4 cores, 8 GB memory and a bandwidth of 10 Mbps. The experiment for investigating the reliability of Dropbox in a multi-user collaboration was conducted on PlanetLab, a testbed for networking and distributed systems experiments. The configuration of each node in PlanetLab is the same- 2.67 GHz CPU with 4 cores, 4 GB memory and a bandwidth of 10 Mbps.

4.1 Analyzing content shared on Dropbox

This measurement study focuses on analyzing the content publicly shared by Dropbox users. We have provided the details of our measurement configuration in Chapter 3. For this study, we collected 7115 of such links shared by the Dropbox users between 02/07/2016 and 02/15/2016 and then downloaded the files from the links. Of the 7115 links, we were able to download 5525 files and discovered that the rest of the links were inactive. We were particularly interested in analyzing the file size and file type (extension) of the content collected.

Figure 4.1 illustrates the distribution of number of files for each file extension type. In Figure 4.2, we plot a histogram representing the number of files for each file category(files with extensions .jpg, .png and .gif are categorized to Images and .pdf, .doc, .txt to Docu-

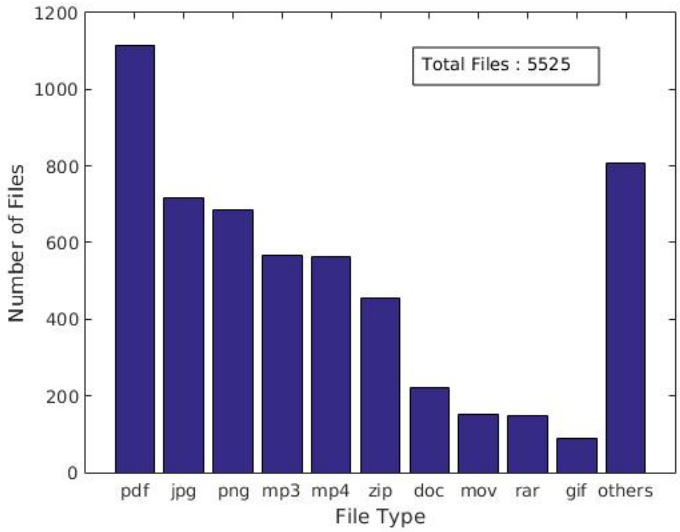

Figure 4.1: Number of files vs File Extension

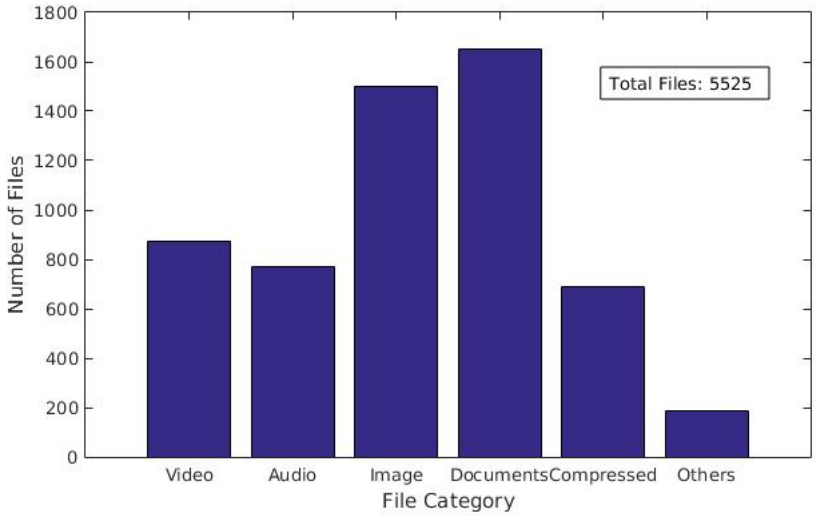

Figure 4.2: Number of files vs File Category

ments category and so on). In Figure 4.2, we can notice that almost 60% of the total files belong to either the Documents or the Images category indicating that these are the most popular files shared on Dropbox. Also in Figure 4.1, we notice that among individual file types, .pdf followed by .jpg followed by .png are the most popular type of files that users share on Dropbox. The results from Figures 4.1 and 4.2 loosely indicate that Dropbox users

mostly share files that are relatively small in size. Figures 4.3 and 4.4 supports this initial conclusion.

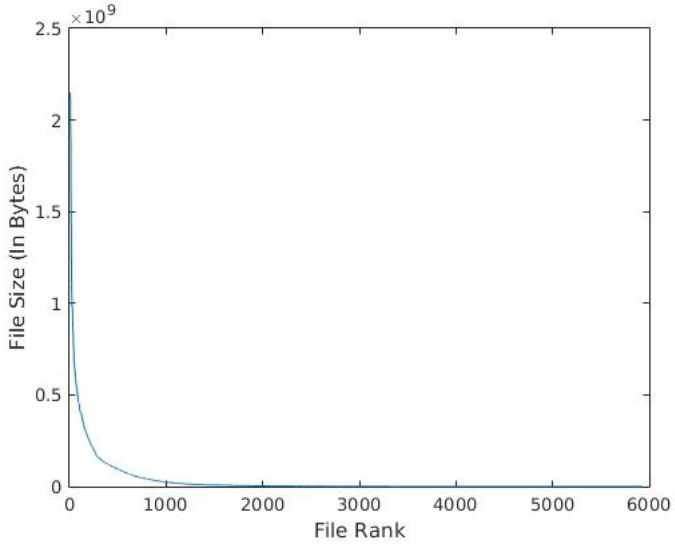

Figure 4.3: Distribution of file sizes

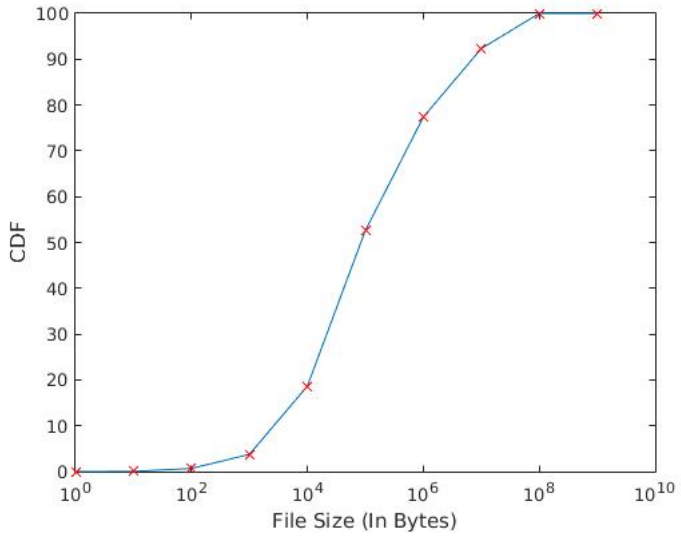

Figure 4.4: CDF of file sizes

We plot the distribution of the file sizes in Figure 4.3. In our sample space, the largest file shared on Dropbox was approximately 10 GB and the smallest file around 5KB. The

average file size was a little over 33 MB but it is heavily skewed by the presence of a few very large files. Figure 4.4 represents the cumulative distribution of files across file size. As we can see in Figure 4.4, around 88% of the files have sizes less than the average file size.

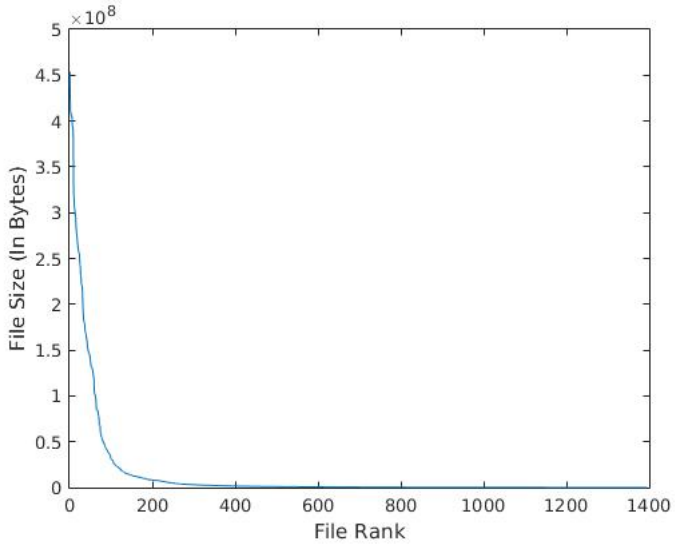

Figure 4.5: Distribution of sizes of files that belong to documents category

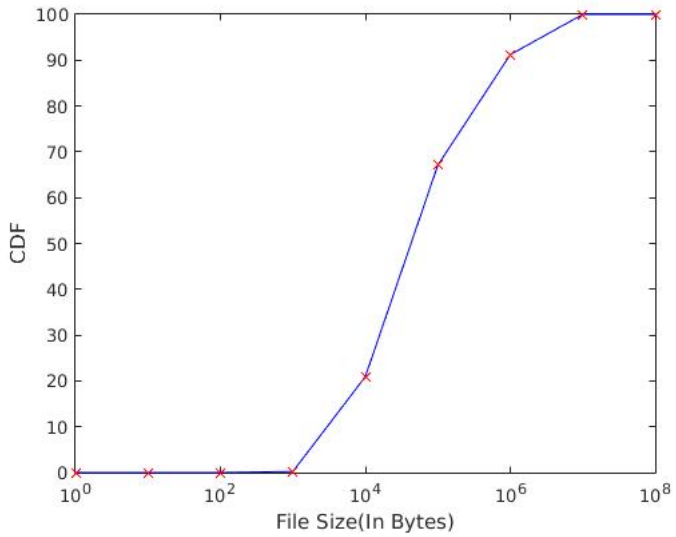

Figure 4.6: CDF of sizes of files that belong to documents category

Figure 4.5 represents the distribution of file sizes that belong to the Documents category(.pdf, .doc, .txt etc.) and Figure 4.6 represents the Cumulative Distributive Frequency respectively. Of the total files categorized into Documents, more than 94% are smaller than the average file size.

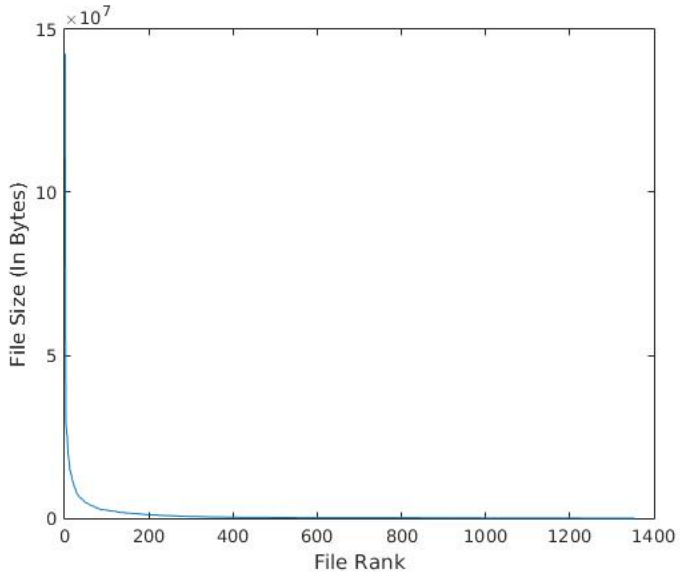

Figure 4.7: Distribution of sizes of files that belong to Images category

Figures 4.7 and 4.8 illustrates the distribution of file sizes and Cumulative Distributive Frequency respectively, of the files that belong to the Images category. We notice that almost all the image files are less than 33 MB.

Figures 4.9 through 4.12 represent similar illustrations of the files belonging to Audio/Video and Compressed categories.

To summarize, our experimental results from this study indicate that documents and images are among the most popular files shared by users on Dropbox and that theusers generally share small size files rather than very large files. It is thus interesting to investigate the performance issues and quality of service of file synchronization on Dropbox for small size documents, particularly in a multi-user collaborative file editing session.

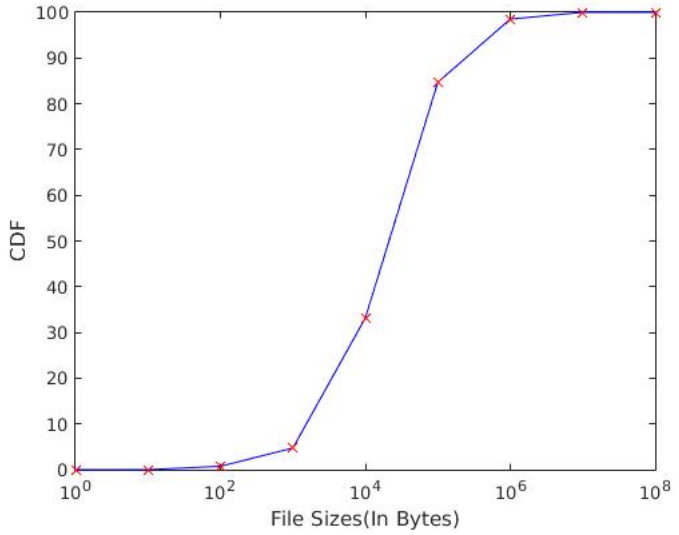

Figure 4.8: CDF of sizes of files that belong to Images category

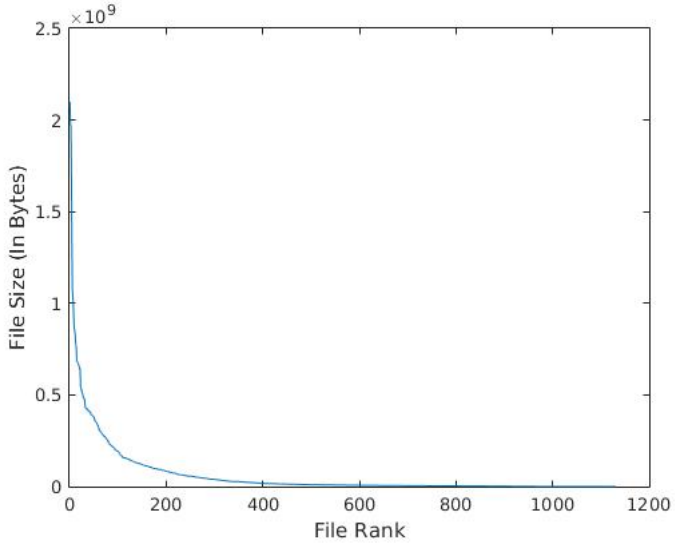

Figure 4.9: Distribution of sizes of files that belong to Video/Audio category

4.2 Reliability of User Collaboration

A necessary and fundamental requirement of a reliable user collaboration is that the system should not discard users' updates without any warning. To ensure that, Dropbox provides two key functions to manage users' updates: keep conflict versions and record file

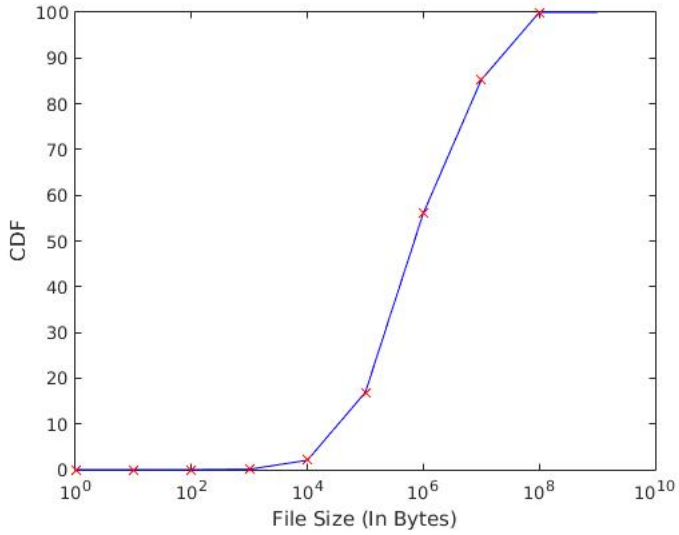

Figure 4.10: CDF of sizes of files that belong to Video/Audio category

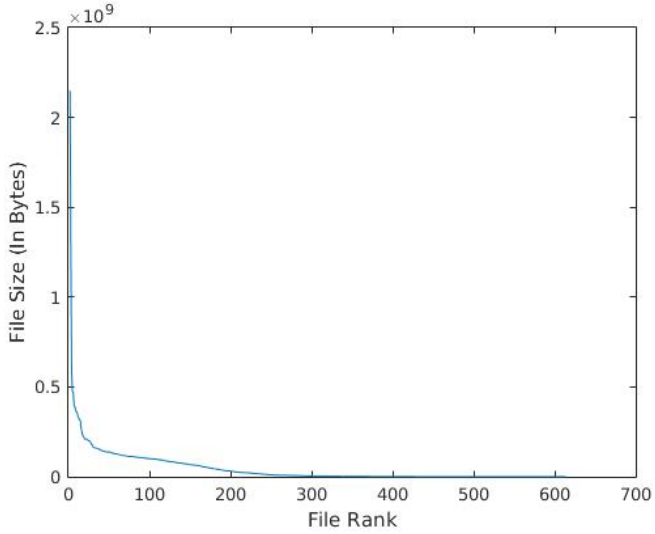

Figure 4.11: Distribution of sizes of files that belong to compressed category

history[21]. When multiple users are editing a file simultaneously, only one copy of the file will be saved as the original copy and other updates will generate new copies of the file as 'conflict version'. It is an intuitive design to avoid possible loss of users' updates but is known to pose certain problems. For example, the synchronization of conflict files creates more conflict files and can overwrite some users' updates. To avoid this potential

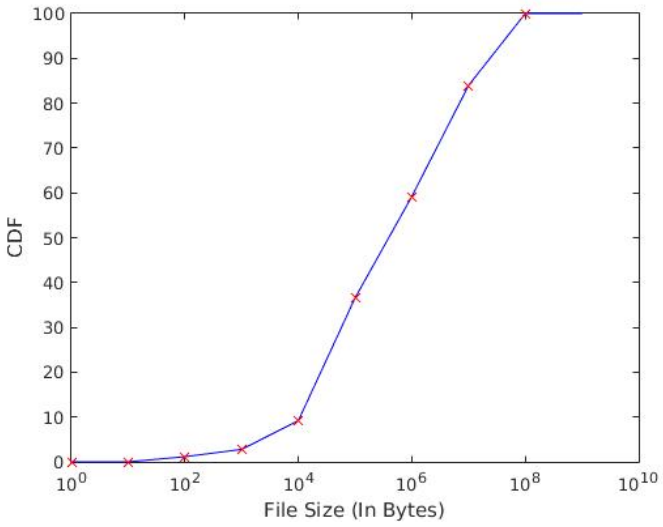

Figure 4.12: CDF of sizes of files that belong to compressed category

problem of losing users' updates, Dropbox provides the file history function that records the historical updates on the file. For free users, this function keeps track of users updates for 30 days and for paid users, historical updates are saved for unlimited time[11].

To evaluate the efficiency of these functions, we conducted two simple experiments on PlanetLab nodes- a single user case and a multiple user case. The details of the experiments are provided in the next section. For these experiments, Dropbox is deployed on PlanetLab nodes and the user(s) edits/updates a file in the Dropbox folder.

4.2.1 Single User Case

In this experiment, file is updated by a single user. In an ideal scenario, all the updates from the user should be captured by Dropbox as historical updates, either in the original file or as conflict versions. Through a series of experiments, we intend to check if this holds true, even when the user is slow in terms of latency.

In our first experiment, the user adds a small file of size 0.1 KB to Dropbox and updates it continuously with 10 edits to the file. The time interval between consecutive updates is

varied from 1 to 15 seconds and check the total number of updates added to the file history. The Round Trip time(RTT) between the user and the server is 80ms. As mentioned earlier, ideally, Dropbox should keep all 10 updates as historical files. Our experiment results, as illustrated in Figure 4.13, shows the number of updates dropped plotted against the time interval between the updates. We can observe that when the time interval is one second, Dropbox drops 7 out of the 10 updates and the number of dropped updates decreases with increase in time interval. When the time interval is 12 seconds, Dropbox was able to keep all the updates from the user. These results indicate that the user should wait 12 seconds between updates in order to ensure that Dropbox does not discard any of the updates. We refer to such a time interval as *safe interval*.

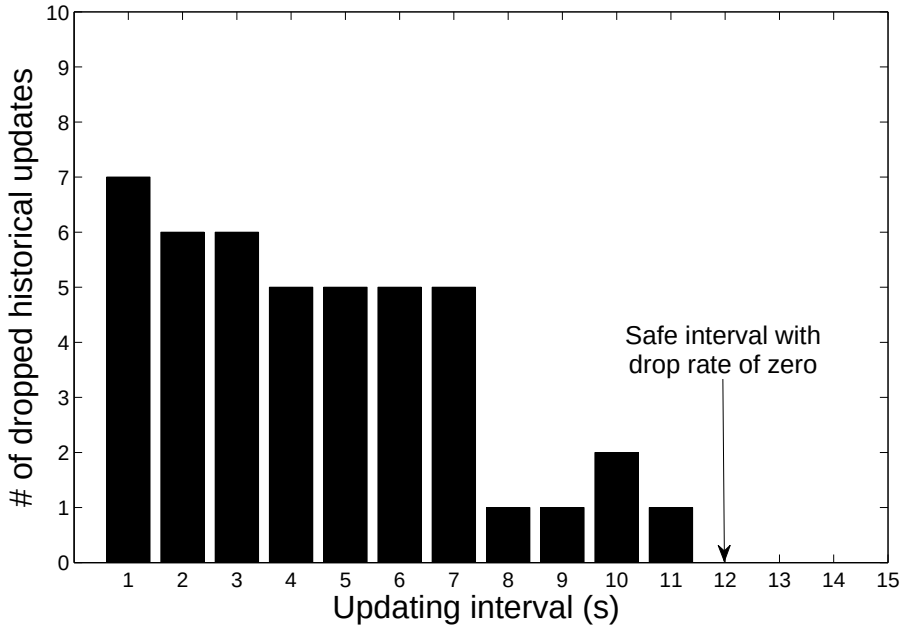

Figure 4.13: Number of dropped updates with different upload intervals (File size 0.1 KB)

To further investigate the problem, we study the impact of the file size and RTT on safe interval time. Figure 4.14 shows the relationship between file size and safe interval time. We can observe that safe interval time increases with an increase in file size.

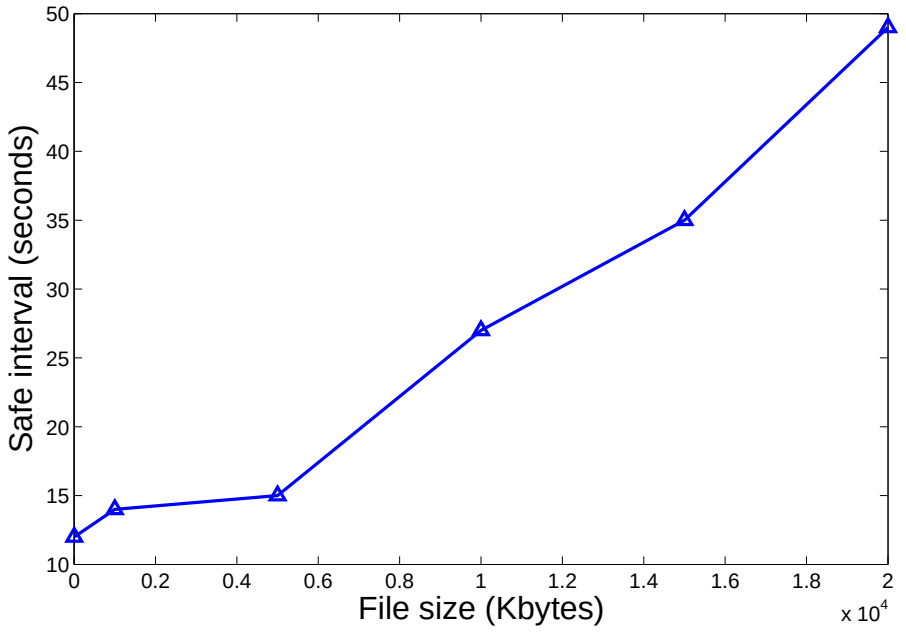

Figure 4.14: Safe intervals under different file sizes

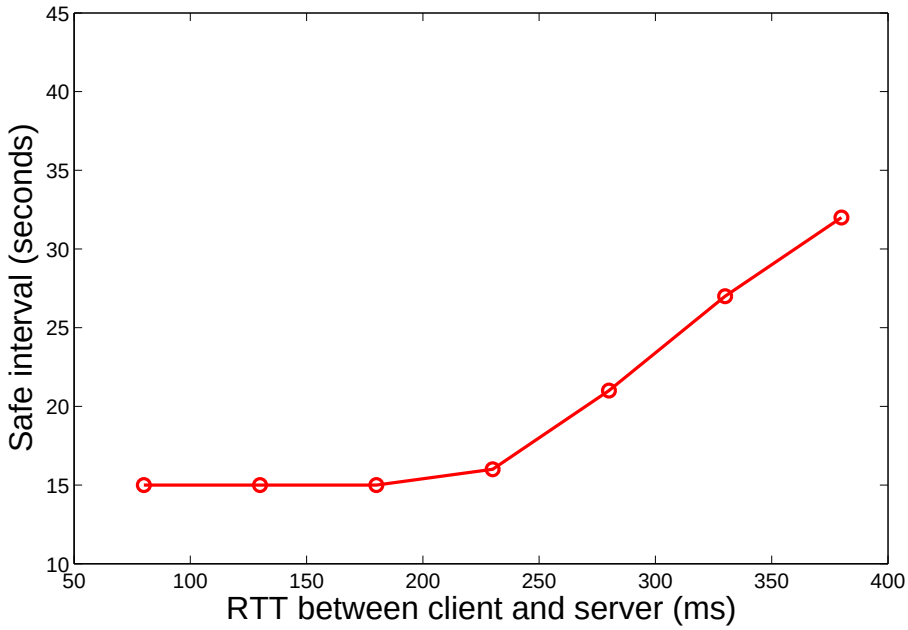

Figure 4.15: Safe intervals under different RTTs (File size 5 MB)

The relationship between safe interval time and RTT, for a file of size 5 MB, is depicted in Figure 4.15. We can notice that safe interval increases with increase in client-server latency. The safe interval at RTT 380ms is twice that of at 200ms. We also notice that there is a steep rise in safe interval time once the RTT crosses 200ms owing to the reduced uploading capacity of the client. This observation confirms that the users with high latency e.g. mobile users are more likely to lose their updates.

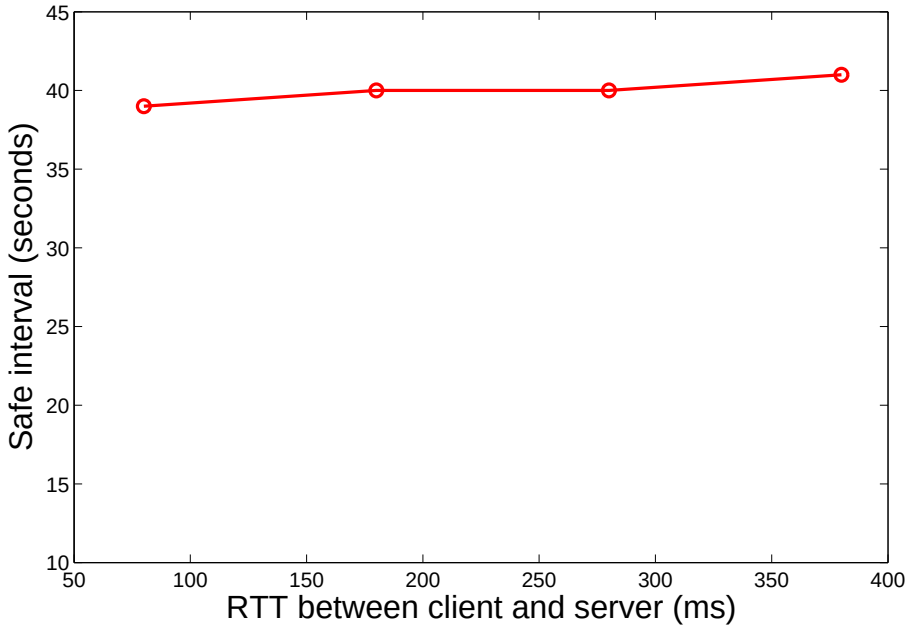

Figure 4.16: Safe interval when uploading rate is limited to 100 KBytes/sec (File size 5 MB)

The impact of safe interval with change in uploading rate is illustrated in Figures 4.16 through 4.18. As we can see in Figure 4.16, the safe interval was as high as 40 seconds when the uploading rate is limited to 100 KBytes/sec. If we compare the results of Figure 4.16 and Figure 4.17, the user with uploading rate 200 KBytes/sec have much shorter safe intervals compared to the user with uploading rate 100 KBytes/sec. Figure 4.18 also confirms the observation that safe intervals for users with better uploading rate are much shorter than

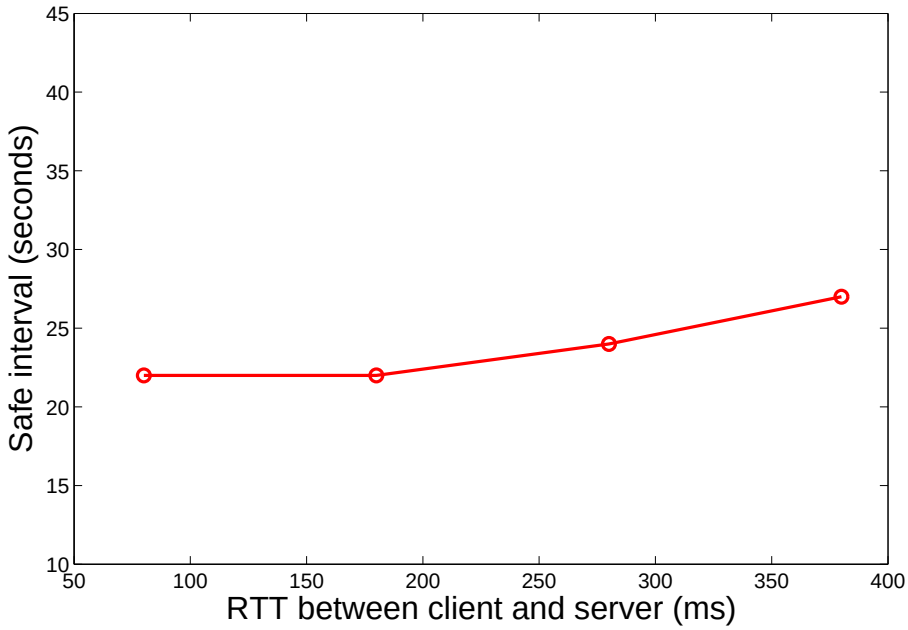

Figure 4.17: Safe interval when uploading rate is limited to 200 KBytes/sec (File size 5 MB)

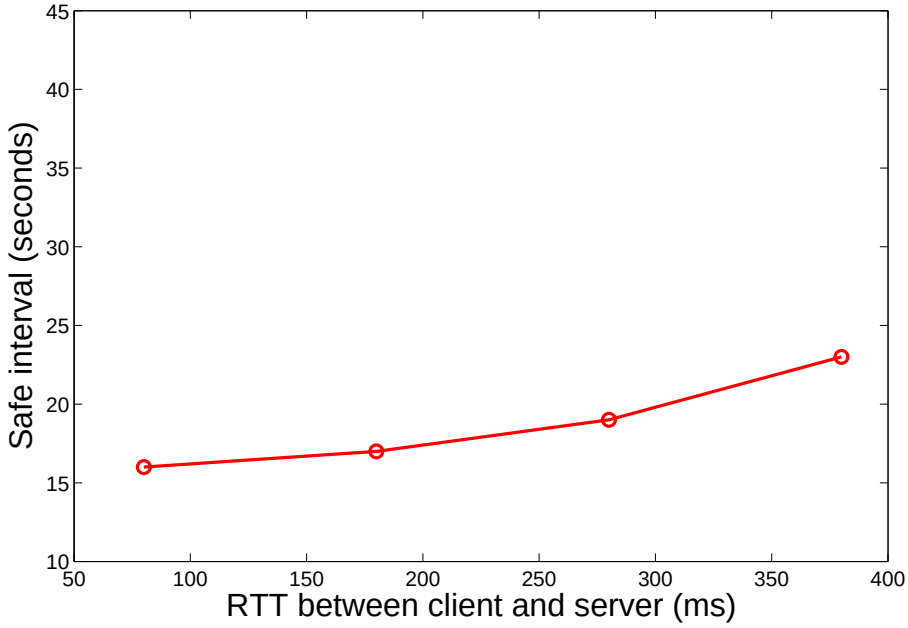

Figure 4.18: Safe interval when uploading rate is limited to 300 KBytes/sec (File size 5 MB)

Table 4.1: Safe interval under different file sizes with uploading speed of 100 KBytes/sec

File size (MB)	0.0001	1	2	3	4	5	10
Safe interval (s)	12	14	18	23	31	39	80

those with lower uploading rate.

We also present the safe intervals for different file sizes with a limited uploading rate of 100 KBytes/sec in Table 4.1. As expected, safe interval increases with file size, for any given uploading rate, albeit without any general trend in the rate of increase.

4.2.2 Multiple User Case

The results from the previous experiment shows that updates from a slow user (in terms of uploading rate and latency) are more likely to get dropped. In this section, we present our experimental results for a multiple user case, where updates come from more than one Dropbox user. Our underlying objective is to understand how a multi-user collaboration is impacted with the presence of slow collaborators i.e, users with higher RTT.

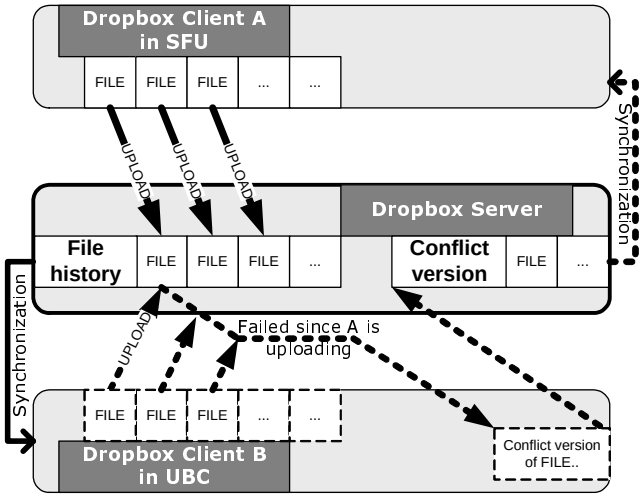

Figure 4.19: Experiment across multiple users

When two Dropbox users are editing a file simultaneously, in order to have a reliable

user collaboration, the Dropbox server should be able to record all the updates from both the users. Figure 4.19 represents a collaboration with two Dropbox users - User A deployed on a PlanetLab node at Simon Fraser University(SFU) and User B, deployed at University of British Columbia(UBC). Both the users edit the same file 10 times simultaneously, with constant time intervals between every two updates. Client-server latency for both the users is same (80 ms) and the size of the file is around 0.1 KB. Figure 4.19 represents an example of the ideal scenario where the all updates from both the users are recorded successfully. All the updates from user A, are recorded in original file history as normal updates whereas all the updates from user B are recorded as conflict versions. The Dropbox server should record a total of 20 copies- 10 from user A as normal updates (represented as solid lines in the figure) and 10 from user B as conflict versions (represented as dotted lines in the figure). If any of these updates are missing, the users' collaboration is affected, which is unfortunately the case in real world. The results from our experiments are presented in the figures below.

Figure 4.20 shows that Dropbox drops some of the updates from the users. As the Figure 4.20 shows, Dropbox failed to record 80% of the updates when the time interval between the updates is two seconds. We can also observe that the Dropbox server manages to successfully record all the updates only when the interval is 17 seconds or more. Recall that from Figure 4.13, the safe interval for a single user case was 12 seconds and it can no longer guarantee a reliable multi-user collaboration.

Figure 4.21 and Figure 4.22 illustrate saved updates from the users along with the details of whose updates are saved as normal updates and whose updates generated conflict versions. As Figure 4.21 shows that when the time interval is eight seconds, a total of 8 updates are saved as normal updates, 5 from SFU user and 3 from the UBC user. Figure 4.22 shows that, when the update interval is 8 seconds, 5 conflict versions are generated, 1 from SFU user and 4 from UBC user. To summarize, the Dropbox system has dropped 4 out of 10 up-

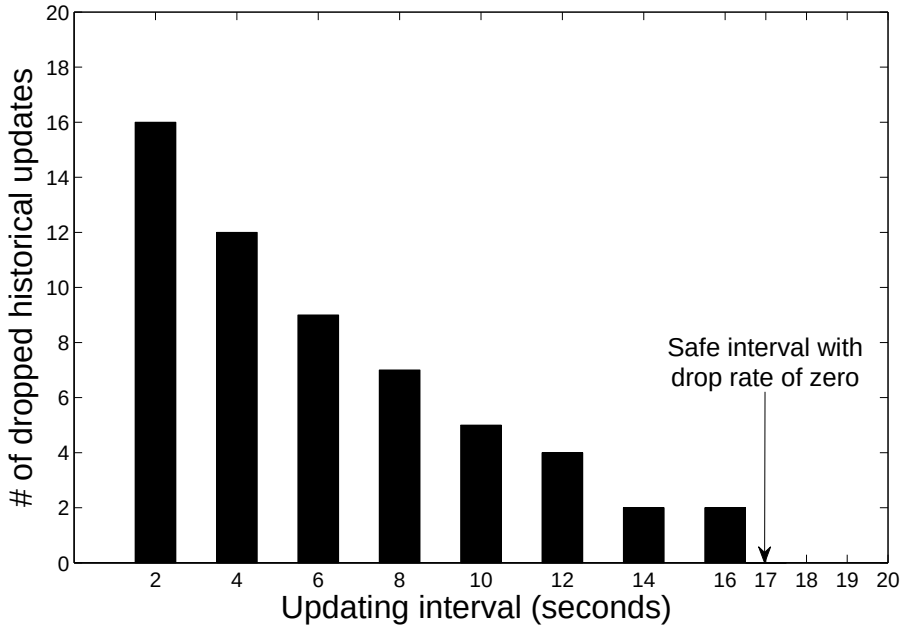

Figure 4.20: Number of dropped updates with different upload intervals (File size 0.1 KB and 2 users)

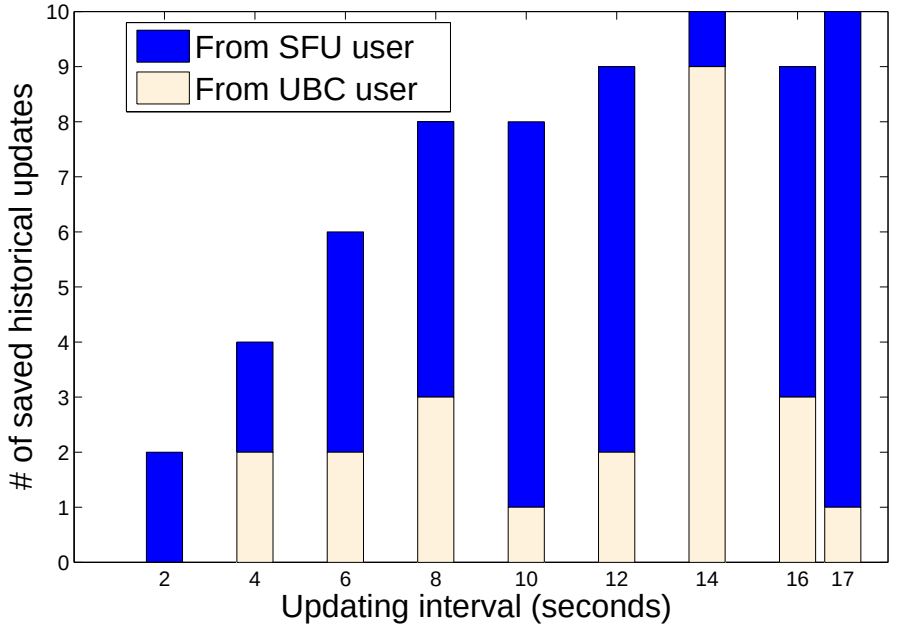

Figure 4.21: Number of normal updates from different users

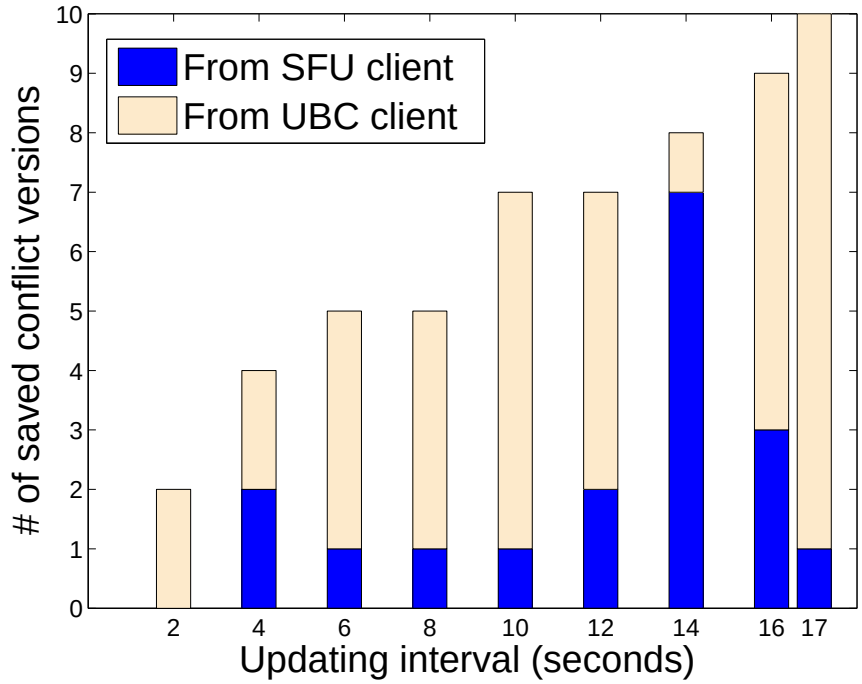

Figure 4.22: Number of conflict versions from different users

dates from the SFU user and 3 from the UBC user. The Dropbox system drops these updates randomly with no determined pattern. This unpredictability and randomness in dropping the updates can pose serious problems in establishing a reliable user collaboration.

Such a problem becomes even worse with the introduction of a slow user in the mix. Figure 4.23 illustrates that the safe interval increases with the increase in RTT of the users. We can observe that safe interval goes up to 40 seconds when the RTT is 330ms, indicating that the users have to wait for at least 40 seconds between their updates to make sure that their updates are not lost. The results from these experiments show that the slowest user in the collaboration limits the overall performance of the system.

The impact of the file size on safe interval is depicted in Figure 4.24 for two users. Notice that the safe interval increases with file size and is considerably larger than the safe interval time for the same file size in a single user case. Figure 4.25 indicates that the safe

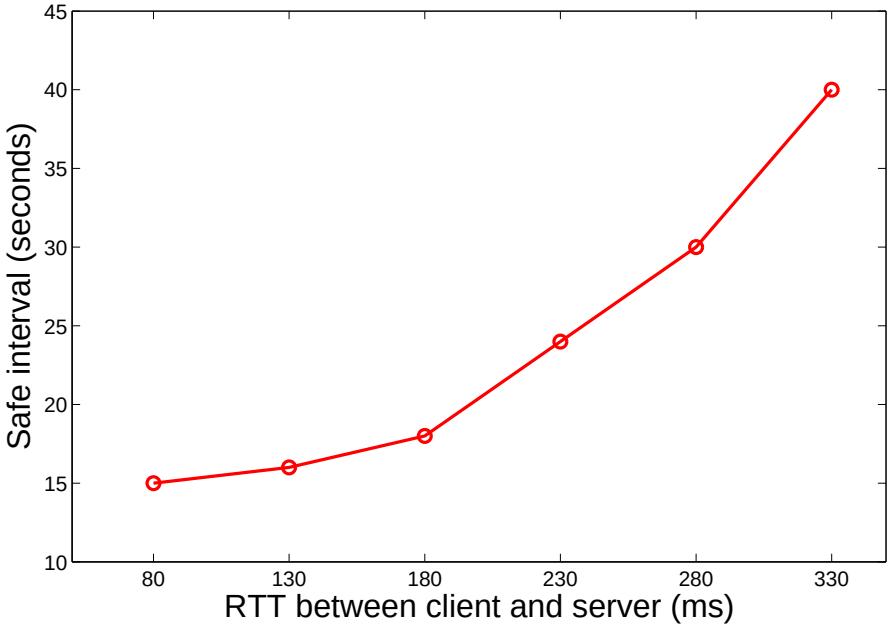

Figure 4.23: Safe interval under different RTTs with file size of 5 MBytes (2 users)

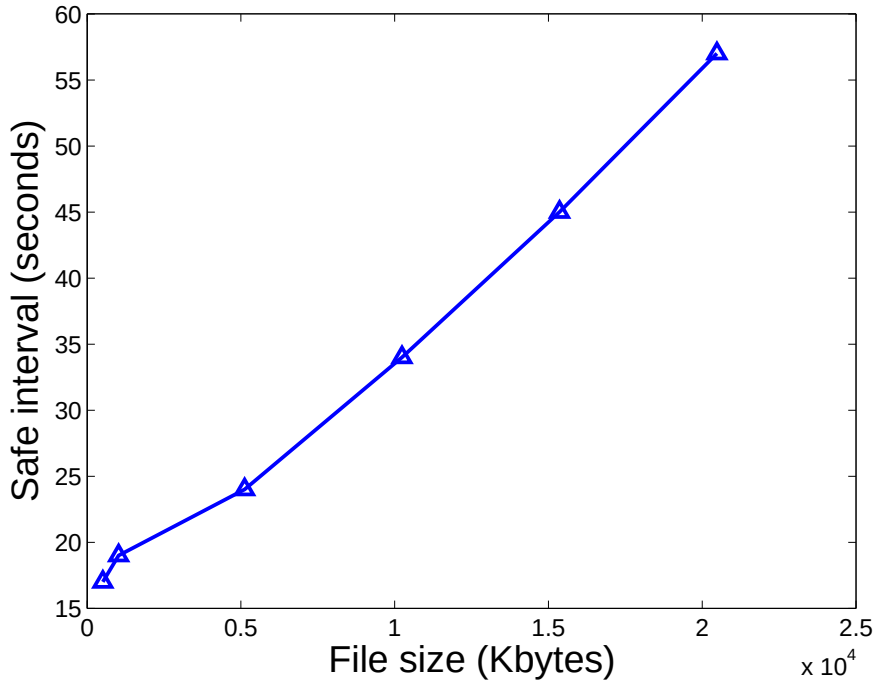

Figure 4.24: Safe interval under different file sizes (2 users)

interval increases as the number of users in the collaboration increases. The safe interval increases from less than 25 seconds for 5 users to over 50 seconds for 10 users. It is evident that matters become worse when there are some high latency users in a large scale collaboration.

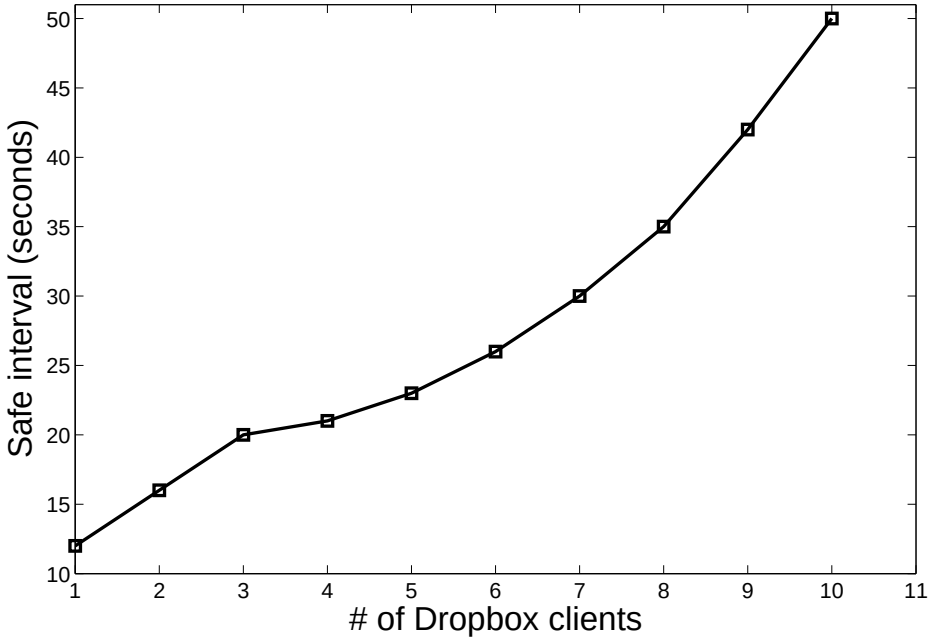

Figure 4.25: Safe interval with more users

4.2.3 Discussions

The unpredictability and randomness in dropping the users' updates is undesirable. The intuitive explanation for losing the updates is that the users' updates are coming in too fast that they are exceeding the service capacity of Dropbox. One way of handling this issue is by enforcing the safe interval between user updates. This enforcement may help ease the cost of maintenance for Dropbox but it does not provide any improvement in the quality of service with respect to the users. Moreover, it does not explain the reason for the possible

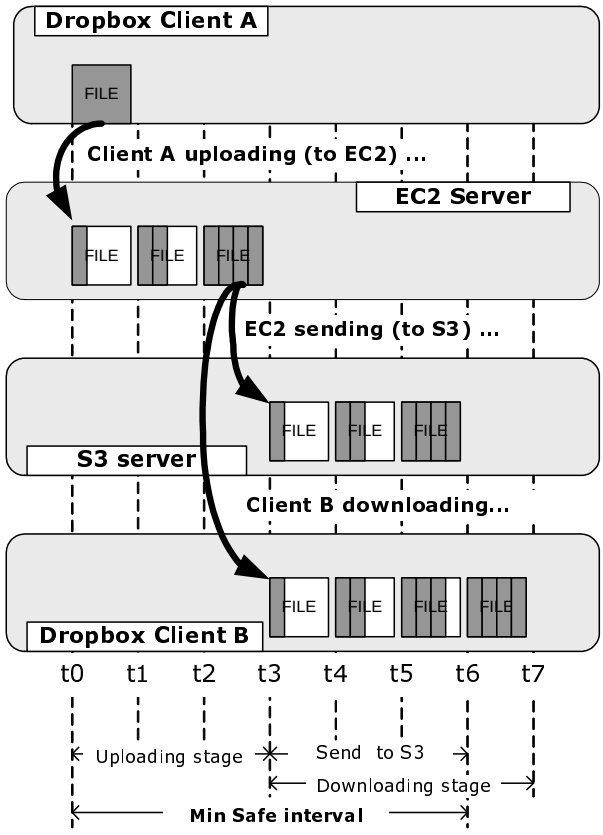

Figure 4.26: Analysis of safe interval

data loss due the dropping of users' updates. We make an attempt to find the root causes for the updates getting dropped using Dropbox Framework design.

Recall from Figure 2.4, that there are two types of communications in Dropbox file synchronization step, viz. communication between Dropbox users (both data source and destinations) and EC2 servers and communication between EC2 servers and S3 storage servers. By comparing the file size with the total amount of traffic, we ensured the communication between the users and EC2 servers. Hence, it is reasonable to believe that the problem is due to the communication between EC2 and S3 servers. We use Figure 4.26 as an example to provide an explanation.

As we can see in Figure 4.26, the four components viz. Dropbox client A, client B, EC2

server and S3 server are represented as large boxes and the small black boxes represents the pieces of files(chunks) that are transmitted during file synchronization. The flow of chunks is represented using solid lines. Dotted lines represents the time slots t_0 through t_7. As shown in the figure, Dropbox client A starts synchronizing the file at t_0 seconds and the file is completely uploaded to EC2 server at time t_3 seconds. The complete file reaches the destination client B at t_7 seconds. We assume that the communication between EC2 and S3 servers starts at t_3 seconds since we are not aware of the protocols between them. If another Dropbox user (say client C) tries to upload the same file to EC2 servers between t_0 and t_3 seconds, that will trigger conflict versions of the file since EC2 is receiving that file from client A. When these conflict versions (or normal updates) are synchronized between t_3 and t_7, they are very likely to be dropped. This also explains why safe interval is hugely impacted by file size, RTT and the number users in the collaboration.

In the next chapter, we propose an enhancement to the system, that handles the potential data loss issue due to the performance bottlenecks.

5 Proposed Enhancement for Reliable User Collaboration

Based on the measurement analysis, it is quite evident that Dropbox drops some users' updates randomly, without any warning, resulting in data loss. The safe interval increases with increase in latency of the user and also with the file size. In case of a multi-user collaboration, the safe interval jumps up with the increase in the number of users and in particular, when there are slow users in the collaboration.

To avoid losing the updates, the users have no other choice but to wait out the safe interval time before making new updates. This not only hinders the scalability of the system but also makes the system prone to data loss. In this thesis, we make an effort to propose an enhancement to Dropbox system to mitigate the problem of losing updates there by avoiding any data loss. The set of specifications and requirements we considered in providing this system enhancement are provided below.

- The system must be fairly resilient to changes in uploading rate, latency issues as well as the file size.
- System must be able to capture all the user updates irrespective of number of users in the collaboration or the presence of slow collaborators in the system.
- The system must provide the users with an option to revert to the previous versions of the file.

- Currently, updates made when the Dropbox client is offline, do not appear in the version history of the file. For this, Dropbox suggests the users make edits only when the device is online, in case there is a chance that user would need to access a previous version of the file later [13]. If possible, the proposed system must also provide version history on the offline updates.

Our main idea is to save the updates locally, if and when changes are made and provide the user with an option of recovery of the historical updates. Our enhancement is based on the implementation provided in [17]. This local file version control in conjunction with Dropbox online file version history, satisfy most of the specified system guidelines.

5.1 Framework Design

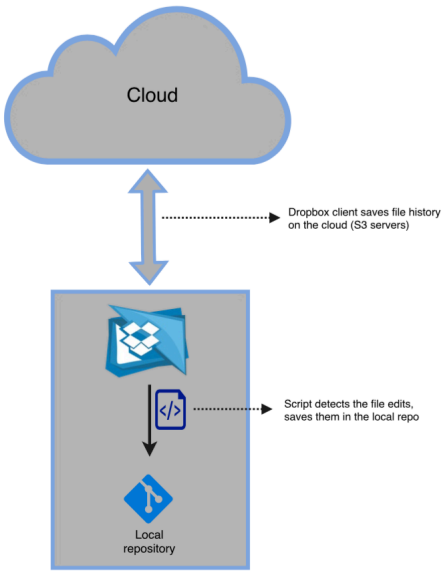

Figure 5.1: Proposed combination of Dropbox and local file version control

The proposed system comprises the Dropbox client and a script that watches the Dropbox client folder. When the user edits a file in the Dropbox client, the script captures the

changes and saves them into a local file repository. Figure 5.1 presents the framework design of the proposed system.

The script requires *inotifywait* and uses *git* as the local repository. inotifywait is available as part of *inotify-tools*, a set of command line programs based on *inotify*. inotify is a feature of Linux kernel to monitor the filesystem events. inotifywait command blocks for inotify events on any set of files/directories recursively. The script blocks for changes in the file/directory(Dropbox client) and auto-commits the changes to git every two seconds. The two seconds wait is added to avoid any race conditions that may occur while watching for changes in directory.

5.2 Performance Evaluation

We have implemented the proposed system on PlanetLab nodes to check whether it satisfies the system guidelines. The proposed system manages to capture all the user updates without failing and preserves the historical evolution of the file. Table 5.1 presents the comparison of total number of updates captured by Dropbox and by the proposed system, in a single user case(when only one user is making edits to the file). Notice that the propsed system manages to save all the user updates successfully on the local machine. Table 5.2 presents similar comparison for a multiple user case, when file edits are coming from two users. The updates saved on both machines are combined and presented in Table 5.2. We can notice that the proposed system records more than 20 updates on both devices together, when the users made only 20 updates. This is because of the synchronization process that happens in the background. The local version control system captures the changes occured during synchronization and captures the historical updates. Figure 5.2 shows sample log of the historical updates saved on the local machine. Using git, the user should be able to revert to any of those historical versions, as needed.

Table 5.1: Comparison of number of updates recorded by Dropbox and proposed system-Single User Case

Time Intervals in seconds	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Updates made by users	10	10	10	10	10	10	10	10	10	10	10	10
Updates recorded by Dropbox	3	4	4	5	5	5	5	9	9	8	9	10
Updates recorded locally	5	10	10	10	10	10	10	10	10	10	10	10

Table 5.2: Comparison of number of updates recorded by Dropbox and proposed system-Multiple Users Case

Time Intervals in seconds	**2**	**4**	**6**	**8**	**10**	**12**	**14**	**16**	**17**	**18**	**19**	**20**
Updates made by users	20	20	20	20	20	20	20	20	20	20	20	20
Updates recorded by Dropbox	4	8	11	13	15	16	18	18	20	20	20	20
Updates recorded locally	24	28	31	33	35	36	38	38	40	40	40	40

Our evaluation summary is presented below.

- The proposed system captures all the updates on the local machine. Some of these updates otherwise would have been dropped by Dropbox during the file synchronization at Cloud storage. It is evident that the proposed system would not drop any user updates due to latency or uploading rate of the users or the number of users in the system.

- Since the proposed system uses Git to keep track of the updates and version control, it provides an easy option for users to revert to the previous versions of the file if necessary. The version history of the file for updates made when the Dropbox client was offline will be available in Git.

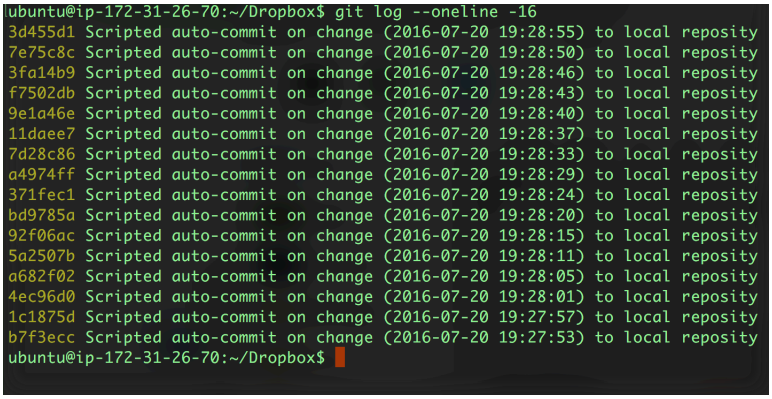

Figure 5.2: Sample log of historical updates saved on local machine

5.3 Further Discussions

There are potential limitations in implementing the proposed solution. Even though the proposed system manages to capture all the users' updates without failing, the user has to verify both the Dropbox file version history and the version history saved on the local machine. That is, if the user wants to revert to a previous version of the file, he has to first find out where the previous version is saved, which can severely affect the ease of use and there by quality of experience of the user. Moreover, if the file is of fairly large size, the user has to dedicate disc space on the local machine to keep the version history whilst paying for Dropbox services. Furthermore, the proposed enhancement has been implemented and tested on computers and its applicability to mobile users is unknown. In general, mobile devices do not have a very large system memory and so the proposed enhancement may not be an ideal implementation for mobile users.

6 Conclusions and Future Work

In this thesis, we make an attempt to understand the type of content shared on Dropbox. We then investigate the performance of cloud-based file synchronization systems. Using Dropbox as a case study, we find that Dropbox like storage systems discard some of the users' updates without warning. Our measurements indicated that the drop rate is hugely impacted by the file size and the latency of the users. In particular, the quality of service drops with the presence of slow users in a multi-user collaboration. We make an attempt to investigate the root cause for the dropping of updates. To overcome the problem of data loss due to discarded updates, we proposed a system that saves file history on the local machine as well as in the cloud.

This thesis presents our initial attempts in understand the performance bottlenecks in cloud-based file synchronization and also our attempts in improving the quality of service. We expect our insights to be the starting step towards establishing a reliable multi-user collaboration on cloud-based storage systems. There are many possible avenues for future work. We are interested in exploring the efficiency of our proposed system with mobile devices. Future works include the possibility of using the same mechanisms to save the local version history as Dropbox uses to save historical updates and check whether we can synchronize the historical updates between local machine and the cloud. There have been reports that Dropbox is moving some of their servers out of Amazon EC2 to operate them closer to the users. It would be interesting to see how this system upgrade impacts our findings.

Bibliography

[1] *AeroFS Syncs Files Between Computers without Storing Them Online*. URL: http://lifehacker.com/aerofs-syncs-files-between-computers-without-storing-th-1509941226/ (cit. on p. 10).

[2] S. Ali, A. Mathur, and H. Zhang. ``Measurement of commercial peer-to-peer live video streaming". In: *Proc. of Workshop in Recent Advances in Peer-to-Peer Streaming*. Citeseer. 2006, pp. 1–12 (cit. on p. 9).

[3] *BitTorrent Sync*. URL: https://www.getsync.com/ (cit. on p. 10).

[4] K. S. Candan, W.-S. Li, Q. Luo, W.-P. Hsiung, and D. Agrawal. ``Enabling Dynamic Content Caching for Database-driven Web Sites". In: *SIGMOD Rec.* 30.2 (May 2001), pp. 532–543. ISSN: 0163-5808. DOI: 10.1145/376284.375736. URL: http://doi.acm.org/10.1145/376284.375736 (cit. on p. 7).

[5] P. Casas, H. R. Fischer, S. Suette, and R. Schatz. ``A first look at quality of experience in Personal Cloud Storage services". In: *2013 IEEE International Conference on Communications Workshops (ICC)*. June 2013, pp. 733–737. DOI: 10.1109/ICCW.2013.6649330 (cit. on p. 16).

[6] D. R. Choffnes and F. E. Bustamante. ``Taming the Torrent: A Practical Approach to Reducing Cross-isp Traffic in Peer-to-peer Systems". In: *SIGCOMM Comput. Commun. Rev.* 38.4 (Aug. 2008), pp. 363–374. ISSN: 0146-4833. DOI: 10.1145/

1402946.1403000. URL: http://doi.acm.org/10.1145/1402946.1403000 (cit. on p. 10).

[7] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. ``PlanetLab: An Overlay Testbed for Broad-coverage Services". In: *SIGCOMM Comput. Commun. Rev.* 33.3 (July 2003), pp. 3–12. ISSN: 0146-4833. DOI: 10.1145/956993.956995. URL: http://doi.acm.org/10.1145/956993.956995 (cit. on p. 21).

[8] B. Cohen. *Incentives Build Robustness in BitTorrent*. 2003 (cit. on pp. 8, 9).

[9] I. Drago, M. Mellia, M. M. Munafo, A. Sperotto, R. Sadre, and A. Pras. ``Inside Dropbox: Understanding Personal Cloud Storage Services". In: *Proc. ACM/USENIX IMC, 2012* (cit. on pp. 2, 15).

[10] *Dropbox*. URL: https://www.dropbox.com/ (cit. on p. 1).

[11] *Dropbox File History*. URL: https://www.dropbox.com/help/11/en/ (cit. on p. 39).

[12] *Dropbox: Global growth*. URL: https://www.dropbox.com/news/company-info (cit. on p. 12).

[13] *Dropbox: Version history FAQs*. URL: https://www.dropbox.com/en/help/9114 (cit. on p. 53).

[14] J. Farina, M. Scanlon, and M. T. Kechadi. ``BitTorrent Sync: First Impressions and Digital Forensic Implications". In: *CoRR* abs/1409.8174 (2014). URL: http://arxiv.org/abs/1409.8174 (cit. on p. 10).

[15] A. Gandhi, M. Harchol-Balter, and I. Adan. ``Server Farms with Setup Costs". In: *Perform. Eval.* 67.11 (Nov. 2010), pp. 1123–1138. ISSN: 0166-5316. DOI: 10.

1016/j.peva.2010.07.004. URL: http://dx.doi.org/10.1016/j.peva.2010.07.004 (cit. on p. 7).

[16] *Gdrive*. URL: https://drive.google.com/ (cit. on p. 1).

[17] *gitwatch*. URL: https://github.com/nevik/gitwatch (cit. on p. 53).

[18] G. Gonçalves, I. Drago, A. P. C. d. Silva, A. B. Vieira, and J. M. Almeida. ``Modeling the Dropbox client behavior''. In: *2014 IEEE International Conference on Communications (ICC)*. June 2014, pp. 1332–1337. DOI: 10.1109/ICC.2014.6883506 (cit. on p. 15).

[19] *hive2hive, An Open-Source Library for P2P-based File Synchronization and Sharing*. URL: http://hive2hive.com/ (cit. on p. 10).

[20] *How do I recover old versions of files?* URL: https://www.dropbox.com/help/11/en/ (cit. on p. 1).

[21] *How do I recover old versions of files?* URL: https://www.dropbox.com/help/11/en/ (cit. on p. 38).

[22] W. Hu, T. Yang, and J. N. Matthews. ``The Good, the Bad and the Ugly of Consumer Cloud Storage''. In: *SIGOPS Oper. Syst. Rev.* 44.3 (Aug. 2010), pp. 110–115. ISSN: 0163-5980. DOI: 10.1145/1842733.1842751. URL: http://doi.acm.org/10.1145/1842733.1842751 (cit. on p. 16).

[23] Y. Huang, T. Z. Fu, D.-M. Chiu, J. C. Lui, and C. Huang. ``Challenges, Design and Analysis of a Large-scale P2P-vod System''. In: *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*. SIGCOMM '08. Seattle, WA, USA: ACM, 2008, pp. 375–388. ISBN: 978-1-60558-175-0. DOI: 10.1145/1402958.1403001. URL: http://doi.acm.org/10.1145/1402958.1403001 (cit. on p. 9).

[24] A. Lareida, T. Bocek, S. Golaszewski, C. Lüthold, and M. Weber. ``Box2Box - A P2P-based file-sharing and synchronization application". In: *IEEE P2P 2013 Proceedings*. Sept. 2013, pp. 1–2. DOI: 10.1109/P2P.2013.6688736 (cit. on p. 10).

[25] A. Li, X. Yang, S. Kandula, and M. Zhang. ``CloudCmp: Comparing Public Cloud Providers". In: *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*. IMC '10. Melbourne, Australia: ACM, 2010, pp. 1–14. ISBN: 978-1-4503-0483-2. DOI: 10.1145/1879141.1879143. URL: http://doi.acm.org/10.1145/1879141.1879143 (cit. on p. 16).

[26] Z. Li, C. Jin, T. Xu, C. Wilson, Y. Liu, L. Cheng, Y. Liu, Y. Dai, and Z.-L. Zhang. ``Towards Network-level Efficiency for Cloud Storage Services". In: *Proceedings of the 2014 Conference on Internet Measurement Conference*. IMC '14. Vancouver, BC, Canada: ACM, 2014, pp. 115–128. ISBN: 978-1-4503-3213-2. DOI: 10.1145/2663716.2663747. URL: http://doi.acm.org/10.1145/2663716.2663747 (cit. on p. 11).

[27] Z. Li, C. Wilson, Z. Jiang, Y. Liu, B. Y. Zhao, C. Jin, Z.-L. Zhang, and Y. Dai. ``Efficient Batched Synchronization in Dropbox-Like Cloud Storage Services". In: *Middleware 2013: ACM/IFIP/USENIX 14th International Middleware Conference, Beijing, China, December 9-13, 2013, Proceedings*. Ed. by D. Eyers and K. Schwan. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 307–327. ISBN: 978-3-642-45065-5. DOI: 10.1007/978-3-642-45065-5_16. URL: http://dx.doi.org/10.1007/978-3-642-45065-5_16 (cit. on p. 16).

[28] L. Peterson and T. Roscoe. ``The Design Principles of PlanetLab". In: *SIGOPS Oper. Syst. Rev.* 40.1 (Jan. 2006), pp. 11–16. ISSN: 0163-5980. DOI: 10.1145/1113361.1113367. URL: http://doi.acm.org/10.1145/1113361.1113367 (cit. on p. 22).

[29] M. Piatek, T. Isdal, T. Anderson, A. Krishnamurthy, and A. Venkataramani. ``Do Incentives Build Robustness in Bit Torrent''. In: *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*. NSDI'07. Cambridge, MA: USENIX Association, 2007, pp. 1–1. URL: http://dl.acm.org/citation.cfm?id=1973430.1973431 (cit. on p. 9).

[30] E. Pinheiro, R. Bianchini, E. V. Carrera, and T. Heath. *Load Balancing and Unbalancing for Power and Performance in Cluster-Based Systems*. 2001 (cit. on p. 7).

[31] *PlanetLab*. URL: https://www.planet-lab.org/ (cit. on pp. 21, 22).

[32] *PlanetLab Migrating to LXC*. URL: https://www.planet-lab.org/node/263 (cit. on p. 23).

[33] *Skydrive*. URL: http://windows.microsoft.com/skydrive/home (cit. on p. 1).

[34] *Storj, a decentralized cloud storage platform that implements end-to-end encryption*. URL: https://storj.io/ (cit. on p. 10).

[35] *SVN*. URL: http://subversion.apache.org/ (cit. on p. 1).

[36] *Vxargs Homepage*. http://vxargs.sourceforge.net/. Accessed: 2015-07-20 (cit. on p. 26).

[37] H. Wang, R. Shea, F. Wang, and J. Liu. ``On the impact of virtualization on Dropbox-like cloud file storage/synchronization services''. In: *Quality of Service (IWQoS), 2012 IEEE 20th International Workshop on*. June 2012, pp. 1–9. DOI: 10.1109/IWQoS.2012.6245967 (cit. on p. 16).

[38] Y. Zhang, C. Dragga, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. ``*Boxx: Towards Reliability and Consistency in Dropbox-like File Synchronization Services''. In: *Proceedings of the 5th USENIX Conference on Hot Topics in Storage and File Systems*.

HotStorage'13. San Jose, CA: USENIX Association, 2013, pp. 2–2. URL: http://dl.acm.org/citation.cfm?id=2534861.2534863 (cit. on p. 17).