On High Performance Cloud Based File Synchronization with User Collaboration

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

Mounika Chillamcherla

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

Haiyang Wang

July 2016

Acknowledgements

I would like to take this opportunity to sincerely thank Dr. Haiyang Wang for his enormous support and guidance, without whom this thesis would not have been possible. I am greatly indebted to him for his vital suggestions throughout the work. He is very kind, encouraging, extremely supportive and always takes care of the well-being of all his students. I really feel blessed to have him as my thesis advisor.

I would like to thank Dr. Ted Pederson and Dr. Yang Li for serving on my thesis committee. A special thanks to Dr. Ted Pederson for teaching me the course Natural Language Processing and filling energy and enthusiasm in me, and for being my inspiration. Further, I would thank all the professors of CS Department, Lori Lucia, Clare Ford, Jim Luttinen and International Student Services for their timely help.

I would like to thank all my friends in the class for all the fun, knowledge, encouragement, and for making my stay a memorable one. Thanks to SaiKrishna Peachara for taking the time to proofread my thesis documentation and for his indispensable support. Lastly, I would like to thank my parents who stood by me and motivated me at all times.

Dedication

I would like to dedicate this thesis to my parents, Adinarayana Chillamcherla and Guru Lakshmi Chillamcherla and to my brother, Eshwar Sai Chillamcherla for their everlasting love and endless support. I am very blessed to have such a family that believes in me and supports the decisions I make in my life. I also would like to dedicate this thesis to each and every friend of mine who supported, motivated and inspired me in every walk of my life.

Abstract

Over the past few years, cloud-based file storage/synchronization systems like Dropbox, Gdrive and Skydrive, have achieved tremendous success among internet users. This new generation of service, beyond conventional client/server or peer-to-peer file hosting with storage only, provides reliable file storage and effective file synchronization for diverse user collaborations.

In this thesis, we take a close look to understand such cloud-based file synchronization and collaboration systems. Using Dropbox as a case study, our real-world measurement carefully decomposes its file synchronization protocol into different stages: *pre-processing*, *uploading*, *downloading*, and *post-processing*. We show that these series of computation and communication operations, which is far more complicated than those in conventional file hosting, is necessary for Dropbox-like services especially considering the cloud deployment. Such a design can significantly improve service reliability and avoid the possible task interference on cloud-based virtual machines (VMs). Unfortunately, these operations also lead to higher latency and cost. In particular, the variance of latency across different users increases with larger population, and thus individual users may face severe performance degradation when the system scale grows. Moreover, we also notice that Dropbox assumes that their users are not online at the same time. The files are therefore uploaded to a cloud storage server and then pushed to the destination. It is easy to see that such a design is inefficient when some of the Dropbox users are online at the same time. To address this problem, we propose an enhancement to let Dropbox detect user's online status and decide whether we can directly send them the file. We tested our prototype on *PlanetLab* and the evaluation indicates that the design can greatly reduces the file synchronization latency with minimal system overhead.

Contents

List of Tables

List of Figures

1 Introduction

Recent years have witnessed the rising popularity of cloud-based file storage systems. Such commercial products as Dropbox [6], Gdrive [8] and Skydrive [25] not only provide reliable file hosting but also enable effective file synchronization for user collaboration. It is known that this new generation of file synchronization is greatly benefited from virtualized cloud resources. For example, Dropbox is using Amazon's S3 service for file storage; Gdrive and Skydrive, on the other hand, utilize the cloud platforms by Google and Microsoft, respectively. The richer services as well as abundant computation, storage, and communication resources enabled by the clouds, way beyond those by conventional servers or CDNs, no doubtably contribute to the tremendous success of these systems.

To understand the detailed design of these systems, pioneer studies as [4] and [14] investigated the characterizations of such cloud-based file synchronization systems as Dropbox. Their measurements identified that the existing file synchronization systems have build-in chunk compression/processing and chunk delivery/transmission functions. The mix of such bandwidth-intensive tasks (e.g., chunk delivery) and computation-intensive tasks (e.g., chunk processing) enables seamless collaboration and file synchronization among multiple users. Unfortunately, it is known that the combination of bandwidth-intensive and computation-intensive tasks will also introduce certain interference and create a severe performance degradation on the cloud virtual machines (VMs) [24] [23]. It remains largely unknown if the most popular cloud-based file synchronization systems, such as Dropbox, are aware of this new challenge and provided related interference avoidance functions.

In this paper, we take a first step towards understanding the interference avoidance de-

sign in the cloud-based file synchronization systems. Using the popular Dropbox as a case study, we investigate their computation and data transmission flows in detail. Our measurement carefully reveals cascaded stages during Dropbox's file synchronization, namely, *pre-processing*, *uploading*, *downloading*, and *post-processing*. We show that this series of computation and communication operations, which is far complicated than those in conventional file hosting [18], is necessary for Dropbox-like services especially considering the cloud deployment. Based on the follow-up performance measurement, we found that the serial-based interference avoidance design can significantly improve service reliability and avoid the possible task interference on cloud-based VMs; yet it also leads to higher latency and cost. We also find that the mutual-interference between different users can also lead to higher performance variance. This sheds new light to our proposed system, where we use "user-user-cloud" pattern rather than using "user-cloud-user" pattern as used by Dropbox and many other cloud synchronization systems. Our proposed system considers user's online status and decides the best way to synchronize the file, thereby reducing the synchronization latency to a great extent when the users are online at the same time.

The rest of this thesis is structured as follows: In Chapter 2, we present the background and related works. Chapter 3 discusses how Dropbox is configured so that measurements can be performed on PlanetLab to evaluate its performance. Chapter 4 presents the measurement results and analysis of the same. In chapter 5, the design of proposed file synchronization system is demonstrated. Thereafter, its performance is evaluated by conducting experiments on PlanetLab. Some practical issues are also discussed. Chapter 6 concludes the paper and provide details about future work.

2 Background and Related work

File synchronization and data sharing have obtained enormous acclaim in the recent years. This chapter gives you an overview of existing ways of performing file synchronization and data sharing. This chapter discusses in detail about traditional client/server file delivery systems, Peer to peer based BitTorrent and Cloud-based file synchronization systems like dropbox.

2.1 Client-Server File Delivery

Any computer in the network acts as either a client or a server. A server shares its resources across the network, where as the client accesses the shared resources provided by the server. Client-Server relationship is also described as request-response relationship. As shown in the Figure 2.1, A client requests the information or resources from the server, which is referred as "Client request" and the server receives the request from client, processes the request and responds to the client by providing the requested information, resource etc., or by denying the client request based on the circumstances, which is referred as "Server response"

Multiple clients can be served by a common server, referred as Multi-user support. As shown in Figure 2.2, There are 3 clients in the network and a single server. The communication between a client and server is independent of the communication between other client-server i.e., the server processes request from each client independently and provides response to the corresponding client. Client-Server systems also provide an advantage of

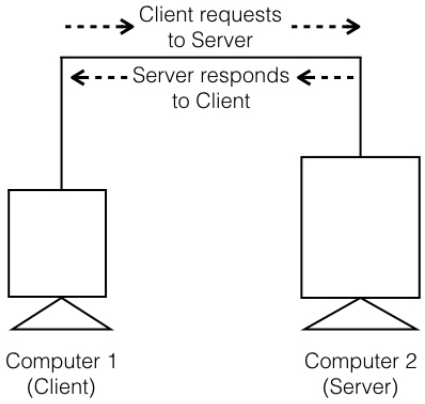

Figure 2.1: Client-Server Relationship

transparency of location i.e., the user or client need not be aware of the physical location of the server.

Application layer protocols such as File Transfer Protocol (FTP), Hypertext Transfer Protocol (HTTP), Simple Mail Transfer Protocol (SMTP) facilitate the communication between the clients and server.

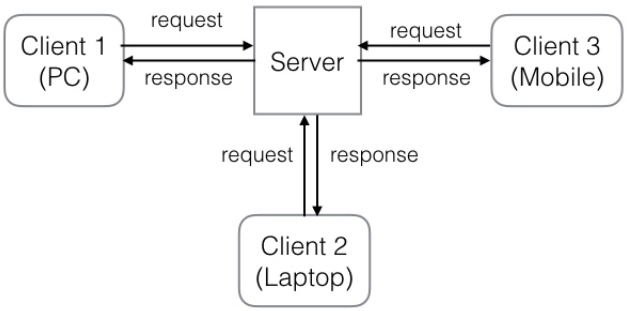

Figure 2.2: A server serving 3 clients(1Pc, 1 laptop, 1 mobile)

A good example of Client-Server systems is web browser-web server. Web browser acts as a client and Web server acts as a server. Web browser requests the information needed from the web server. Web server accepts the request, processes it and responds with either one of the following: (1) Responds with the information needed for the client (2) Denies to

provide the information,if the requested information is not found.

File Transfer Protocol (FTP):

The basic functionality of File Transfer Protocol (FTP) is to exchange files between client and server on a computer network. FTP uses TCP/IP protocols on port 21 to facilitate data transfer. FTP client downloads a file from server or uploads a file to the server. FTP server stores the files uploaded by FTP Client. FTP uses data connection and control connection in order to provide communication between client and server. The control connection allow the clients to connect to the server and send commands or requests to the server over the network. The data connection is used to send data or files over the network (from client to server or from server to client).

Hypertext Transfer Protocol (HTTP):

HTTP is a request-response protocol which allows the transfer of text along with multimedia files such as images, videos etc., between client and server. HTTP is implemented on TCP/IP on port 80 on the server by default. HTTP client initiates request via TCP connection to port 80 on the server. The server takes the request, processes it and returns with the status message "HTTP/1.1 200 OK" along with the body of data requested by user or an error message, if the information requested is not found in the server.

Simple Mail Transfer Protocol (SMTP):

SMTP protocol is used to send electronic mails (email) from client to server. SMTP client and SMTP server communicate using commands. SMTP is implemented on TCP on port 25. SMTP protocol is based on ASCII text-based, so it doesn't deal well with the images, videos etc.,

The main disadvantage of Client-Server File Delivery System is it's scalability. Server's workload grows linearly with the number of clients. This system is not scalable under flash crowd user arrival i.e., large number of users send requests to the server at the same time. Therefore, a scalable content delivery model is needed to satisfy the requests of fluctuating

user base.

2.2 Peer-to-Peer File Distribution

Peer-to-Peer file distribution system have gained huge popularity in the recent years. The basic difference between Client-Server Model and Peer-to-Peer model is that in Client-Server model, a computer in the network can act as either a client or a server, whereas in Peer-to-Peer model, peers can play the role of both client and server. Resources are shared across the network i.e., Any user can share resources on his computer with any other user's computer on the same network. The advantage of Peer-to-Peer system is that it scales based on the number of users trying to download the content and it is self-organized. Disadvantage of Peer-to-Peer model is Cross ISP traffic. BitTorrent is the most popular and most commonly used Peer-to-Peer File distribution system.

2.2.1 BitTorrent

Bram Cohen introduced BitTorrent in 2001, intended to distribute large files over the network. BitTorrent is better than client-server in terms of scalability, reducing the load on congested servers and accelerating the download rates for users. Companies like Facebook, Twitter use BitTorrent in order to push their data and updates to all their servers distributed across the globe, in a productive way. In BitTorrent, file is divided in to fixed-size partitions, knows as chucks or pieces.

2.2.2 BitTorrent Fundamentals

BitTorrent Swarm: Group of peers downloading the same content and exchanging chunks or pieces with each other.

.torrent file: A .torrent file contains the metadata of the file to be downloaded which is used to uniquely identify the BitTorrent swarm linked to that particular .torrent file and the IP addresses of all the trackers managing the peers in the swarm.

BitTorrent Server: Content providers upload the .torrent files to the BitTorrent server. BitTorrent clients, who wish to download particular content downloads the .torrent file and proceeds with content downloading.

BitTorrent Tracker: Tracker maintains the list of peers and the download progress of each peer participating in the swarm. When a new peer enters the swarm, the new peer communicates with the tracker and the tracker gives it the information about the peers in the swarm.

BitTorrent Client: BitTorrent client is a peer, which engages in the swarm by downloading chunks from other peers and/or by uploading chunks to other peers. BT Clients are classified in to Seeder and Leecher.

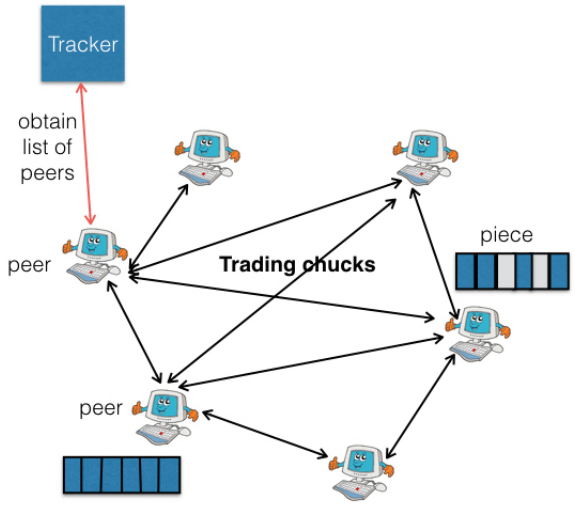

Figure 2.3: Peer-to-Peer relationship: BitTorrent

2.2.3 BitTorrent Algorithms

BitTorrent has 2 algorithms: 1. For Peer selection, BitTorrent uses Tit-for-Tat mechanism, also known as Choking/Unchocking Algorithm. 2. For Piece selection, BitTorrent uses Rarest Piece First algorithm.

Peer Selection: Chocking/Unchocking algorithm

When Peer A's neighbors are interested in downloading the content of A, this algorithm selects a node from A's neighbors, to which it transfers the requested chunk. All the connections from a node to its neighbors are chocked by default. Whenever a peer wants to transfer chunks to another peer, it unchokes it's connection with another peer. A node can upload it's content to other peers at a particular time is limited to 5 by default [2].

Tit-for-Tat strategy work as follows: A peer keeps track about the list of neighbors which uploaded content to it and also the upload rate with which they uploaded content to it. Based on this criteria, the peer(A) selects 4 peers which has the maximum upload rate to that node (A) i.e., a leecher helps those peers, which helped it in downloading chunks in the last 20 seconds. All the remaining peers were blocked or choked. The basic idea of Tit-for-Tat strategy for leechers is "If you help me in my downloading process, I will help you too in your downloading process." and Tit-for-Tat strategy for seeder is "I will unchoke those leechers, who were quick in downloading chunks from me."

Apart from Choking/Unchoking, BitTorrent has a mechanism called Optimistically Unchocking. In every 30 seconds, a leecher selects a neighbor randomly without considering any criteria and unchokes it, so that the unchoked peer can download chunks from the leecher. Optimistic Unchocking really helps the new peers in downloading the content in the swarm.

Piece Selection: Rarest Piece First

When a peer A is unchoked by an other peer B, peer A can request any block which peer B has. As every peer in the swarm knows about the list of chunks its neighbors have. Peer A chooses the rarest chunk among their neighbors. The reason behind this is: If a peer(XYZ) has the rarest block with it among their neighbors, that peer(XYZ) will get a lot of demand from their neighbors. As shown in the Figure 2.4, there are 4 peers(A,B,C,D), the node A is unchocked by node B. So, now node A can select a chunk from B to download. As chunk 3 is the rarest piece and it is only available at node B, it requests for chuck 3. This is how the "Rarest piece first" works.

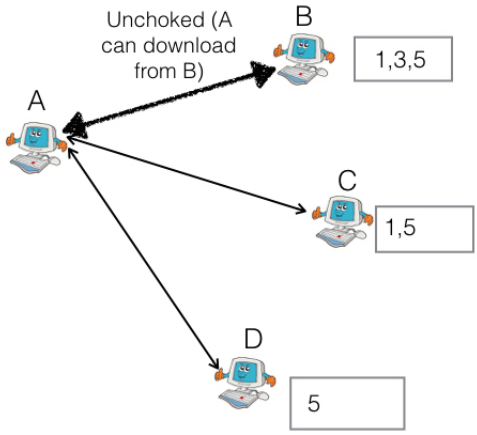

Figure 2.4: Piece Selection: Rarest Piece First

2.3 Cloud-Based File Synchronization (CBFS)

2.3.1 What is Cloud Computing

Traditionally Data centers were used in order to store data. The disadvantages of using data center are high maintenance, high power consumption, very difficult to scale, large

infrastructure, heating and cooling issues etc., In order to avoid these issues, companies have moved their data to cloud and enjoying the services provided by cloud providers. Cloud is the abstraction of data center. It gives the user the resources like storage, software, applications etc., but the user don't need to know where the cloud is located or how longer the resources are reliable and so on. Basically, cloud needs a datacenter, but the user can just use the resources and services provided by the cloud provider, without the need of dealing with their own datacenter and issues related with it. Cloud providers host multiple data centers in different geographic locations in order to provide data security during server crashes.

Cloud computing can be referred as the applications delivered as services over the internet along with hardware and systems software in the data centers that provide these services. The main advantages of cloud computing is 'pay as you go' feature. Elasticity of resources also play a major role in cloud computing i.e., The user can add the resources which he need in just few minutes and he can release them anytime after his use. So this feature gets rid of the burden to pre-plan the resources for the cloud users. Cloud computing also keeps the operational costs low for service providers. Users can use the services of the cloud at anytime and anywhere from all over the world, share data and store their data securely without any extra effort. [5] specifies the top 10 obstacles and opportunities by the cloud computing, which are concerned by the user while deciding transition from datacenter to cloud.

Cloud is distinguished as Public cloud and Private cloud. Public cloud is available to the public in the pay as you go manner. The popular public cloud providers are Amazon AWS, Microsoft Azure, Google App Engine and Rackspace Cloud Servers. Service sold by the public cloud providers are known as utility computing. Private cloud is referred to the internal databases of specific organizations. Each cloud provider offers services which vary widely in performance and cost, compared to others. For example, Amazon offers 17

instance types offering high I/O, high CPU, high memory etc., [1] systematically compares the performance and cost of cloud providers. The user will have the choice in selecting cloud provider, based on which cloud provider can satisfy the requirements of his application in best and in cost-efficient way.

Quality of Service (QoS) metrics such as high throughput, low response time and high service availability need to be assured by the cloud providers, to the users. Service Level Agreements are maintained by the users, with the service providers for the QoS properties. If the service providers fail to meet the QoS properties, then a large decrease in the user base occurs, which eventually leads to the loss of revenue. Acquiring resources from the cloud consists of few steps: Initially the application has to make a call to the cloud API, which starts the acquisition process. The machine will then be booted by a specified image, based on the requirements of the user and the application need to be started. Finally, the resources are given to the user. The set up time of an Instance or VM takes at least few minutes. So, the developer needs to be very attentive in choosing when to start an instance. If the user allocates the resources to the application on the forehand, over-provisioning or under-provisioning of resources might occur, which results in loss of revenue. On the other hand, if the user choose to allocate the resources spur at the moment, as it takes few minutes for the resource to be set up, the efficiency of the application will be decreased.

In cloud computing, a large Physical machine (PM) is divided into pieces and each piece is referred as a Virtual machine (VM). Each application resides in an operating system called Virtual machine, provided by Elastic Compute Cluster. Cloud providers like Amazon allocate the virtual instances for the users, upon request at variable prices for lease period. If the workload increases, the cloud provider allocates new virtual instances to the user and if the workload decreases, it just kills the unnecessary virtual instances. When the application running on a VM is attacked or if any crash occurs, the server just kills the attacked VM and starts a new one, providing security to the applications. Hypervisor is used to coordinate

the VM's. Each VM is isolated to each other and they divide CPU and the storage of it's host machine (Physical machine) among themselves. VM's with different processing power (CPU, memory, disc etc.,) are available. VM's takes less time to create, easy to add and kill them, but the processing power of VM is less than a PM. The relation between PM and VM is that the VM depends on the PM's memory, CPU and power. Xen, Kvm and VmWare provide virtualization. There are three types of virtualization: Para Virtualization, OS-level Virtualization and Application level virtualization. Each virtualization technique provides the services to the user, in a different way. So, it is better to choose a kind of virtualization based on the needs of the application.

2.3.2 Cloud-Based File Synchronization System(CBFS)

Dropbox, Google Drive and OneDrive are the best examples of CBFS. They provide real time storing and sharing of data, providing reliability and convenience to the user, so that they can use them from anywhere and at any time, following the scheme of "Store once, use at any time and from any location"

File synchronization is the vital function of cloud storage services, which allows user to add and modify files and map these changes made from local file systems into the cloud, with a series of network communication between client and cloud server.

Recent years have witnessed the rising popularity of cloud-based file storage systems. Such commercial products as Dropbox [6], Gdrive [8] and Skydrive [25] not only provide reliable file hosting but also enable effective file synchronization for user collaboration. It is known that this new generation of file synchronization is greatly benefited from virtualized cloud resources. For example, Dropbox is using Amazon's S3 service for file storage; Gdrive and Skydrive, on the other hand, utilize the cloud platforms by Google and Microsoft, respectively. The richer services as well as abundant computation, storage, and

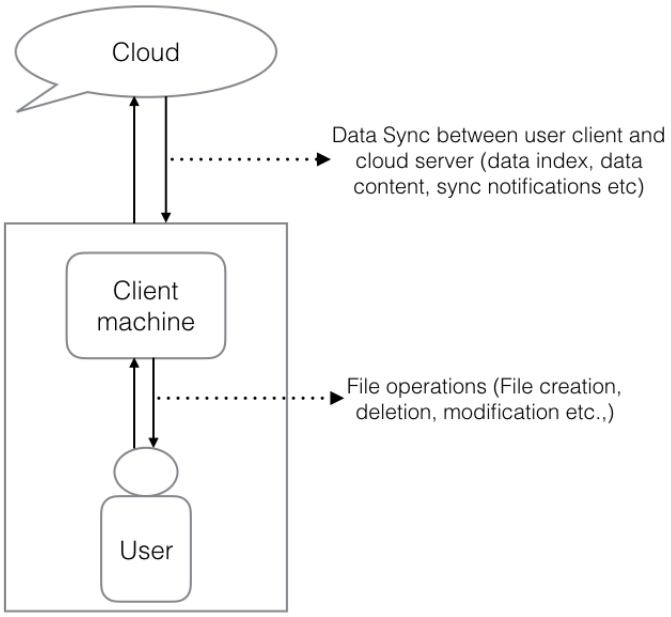

Figure 2.5: Data Synchronization principle.

communication resources enabled by the clouds, way beyond those by conventional servers or CDNs, no doubtably contribute to the tremendous success of these systems.

Data Synchronization principle is figuratively expressed in the Figure 2.5. When a user is using a cloud storage service, the fist step to do is to create a local folder, which is also called as "Sync folder". All the files to be stored in the cloud have to be initially stored in the sync folder on our local machine. When the user performs operations like file creation, file modification, file deletion etc., in the sync folder, these operations were observed and synchronized to the cloud. Data synchronization from local folder to the cloud includes sequence of Data Sync events such as data content, data index, sync status and notifications, sync acks and statistics [15]. The network traffic generated by data sync events is referred as "Data sync traffic".

2.3.3 Related Studies of Cloud-based File Synchronization services

Enormous number of experiments and vast research work has done to understand the behavior of cloud-based file synchronization systems like Dropbox, Google Drive etc., To understand the Dropbox Client behavior, Gil et al. [9] collected data about the dropbox usage through measurements. Three cloud-based file synchronization systems such as Box, Dropbox and SugarSync were analyzed in terms of performance aspects like upload speeds, download speeds, transfer rate etc., by Gracia-Tineda et al. [10]. Haiyang et al. [28] conducted a measurement to study the performance bottleneck of Dropbox. Based on their measurements, they found that Dropbox also relies on Amazon's EC2 instances for providing file synchronization services along with Amazon's EC2 servers for storage. Idilio et al. [5] explained the extensive peculiarity of Dropbox such as traffic patterns, typical usage and performance bottlenecks.

Quality of experience factor is measured by conducting experiments with a group of 52 users by Casas et al. [3] to measure the performance of personal cloud storage and file synchronization applications like Dropbox, Google Drive etc., A measurement study was proposed by Hu et al. [13] by comparing four commonly used cloud storage services like Dropbox, Mozy, CrashPlan and Carbonite on the measurement factors like backup and restore times, privacy, risks, faults etc.,

Zhenhua et al. [16] proposed a solution for traffic-overuse problem using Efficient Batched Synchronization mechanism. This mechanism batches updates from clients and send them to the server, reducing the session maintenance traffic. Sudharsan et al [26] explained the spam mail and phishing attack, which could possibly take place on Dropbox and proposed a system to avoid these kind of attacks and proposed security measures for Dropbox. Marshall et al [19] conducted real-time experiments using 106 people to know the user experience with the cloud file synchronization services like Google Drive and Dropbox.

The main advantages of Cloud-based file synchronization services are that they are easy to deploy and very scalable, but they still face problems with data privacy and efficiency. It is reported that the files created on one device are to synced to an other device in a corrupted way [20]. The other disadvantage is that with increasing scalability, user's interaction latency increases. Privacy concern is the biggest thing to worry about, while using third party file synchronization services.

As Dropbox is the most widely used cloud-based file synchronization system, we are going to measure the performance of Dropbox on various factors like Synchronization latency, Downloading rate, Uploading rate, CPU utilization and going to analyze the functionality of Dropbox. Later sections demonstrate the results of our measurement and also explains about the way we are trying to improve the system.

3 Measurement Configuration of Dropbox

3.1 Framework of Dropbox

Dropbox, mostly used Cloud-based File Synchronization system is initially released in September 2008. Dropbox has become one of the most popular cloud storage providers on the Internet. It has 500 million users and has stored more than 500 billion files with an increase of one billion files in every 48 hours [7]. Not only simple file hosting, Dropbox allows multiple users to effectively share, edit, and synchronize online files. It relies on Amazon S3 cloud for storage. Detection of changes and transmission of changes are the vital functions of Dropbox as a File Synchronization Service.

Detection of changes:

The dropbox client must be able to detect the changes both on our local machine and cloud. If any changes were made on our local system, Dropbox uses Linux's inotify service to detect those changes and send the changes to the cloud. When any changes were made to the file in the cloud by the shared user, dropbox uses push-based notifications in order to detect the changes and send those updates to the local machine. When the local machine is not connected to dropbox or, when the dropbox is offline, If a user makes changes in a file on his local machine, these changes were saved in the local database with metadata information about the file along with Time modified. When the dropbox client goes online,

it checks the time modified and gets to know whether the file has been modified or not, since the last update. If the file is found modified, it pushes the changes to the cloud.

Transmission of changes:

When a new file is added to the dropbox sync folder or local folder, it initially checks the size of the file. If the size of the file is greater than 4 MBytes, Dropbox splits each file into chunks of size 4 MBytes in size. Dropbox client calculates the hash values of all the chunks of file using SHA-256 algorithm. These hash values are used in avoiding the redundant uploading of chunks. Amazon S3 server is used for storing the files of Dropbox. Dropbox uses deduplication technique [30] while transferring updated files to and from the server. Deduplication technique works as follows: Before transmitting the file, hash values are calculated for each chunk in the file, these hash values were send to the server, if the server contains a chunk with the same hash value, that specific chunk is not transmitted again, If the server doesn't contain the chunk with the same hash value, that specific chunk is transmitted. In this way, only updated information is sent to and from server, thereby reducing the data sync traffic. Dropbox uses rsync function to transfer partially modified files. While moving files from source to destination, dropbox downloads file into a staging area, gathers them and then move to the destination in order to provide atomicity property.

We accordingly illustrate the Dropbox service framework in Figure 5.1, which consists of three major components on the server side (marked as grey boxes in the figure). The first component is 6 load-balancers that are deployed by Dropbox[1]. The second part is 360 EC2 instances which provide data uploading, downloading and file processing functions (referred to as *delivery servers* in the figure). Our packet level information shows that the Dropbox users will upload the files to these delivery servers during the file synchronization. The third part is the S3 server cluster that stores the uploaded files.

[1]Based on the packet level analysis, these servers are assigning different EC2 instances to the Dropbox users for file uploading. They also assign different instances to the same user when she/he sends different updates at a different time.

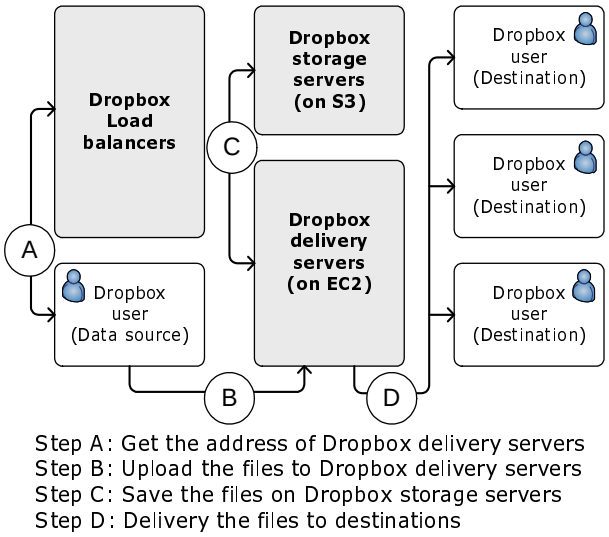

Figure 3.1: Dropbox Framework.

The following data flow facilitates a source user who wants to upload a file to her/his Dropbox folder:

1. The data source (a Dropbox user with files to upload) will send out a DNS request to query the IP address of the load-balancers of Dropbox. The DNS server will reply a whole list of six load-balancers to the data source. After that, the data source will randomly pick one load-balancer and send the related file information to this load-balancer, including the file size and type. The selected load-balancer will then assign one EC2 server to the data source (*Step A*).

2. The data source will upload the file to the EC2 server (*Step B*).

3. When the file is successfully uploaded, the EC2 server will forward this file to the user's S3 folders (*Step C*).

4. Meanwhile, another EC2 server will be used to deliver the file to the destinations that need to be synchronized (*Step D*).

It is worth noting that in *Step A*, Dropbox will check whether an identical copy of this file (or some identical chunks) was previously uploaded by the user; if so, the source user

will not need to upload this file (or those identical chunks) again. We refer to this as the *caching function* of Dropbox. This reduces the data sync traffic.

3.2 PlanetLab-based Active Measurement

The design of our measurement approach i.e tools used, related scripting and analysis of code is discussed in this section. PlanetLab is the tool used to perform measurements. Scripting languages used for the measurements are Java, Python and Bash scripting languages. Python scripts are run on the local machine, which in turn triggers bash scripts on the selected remote PlanetLab node. The performance metrics considered for this measurement are Uploading and Downloading rates, Synchronization latency and CPU Utilization. After the measurement, all the results were retrieved from PlanetLab nodes and stored on our system to analyze the results. Java is used in implementing the proposed system.

3.2.1 PlanetLab

PlanetLab, recognized as Global Network Testbed [21], provides the users a platform to run distributed and network-based applications across multiple nodes, distributed all over the world. PlanetLab is a large set of systems located in all the continents and set up solely for the purpose of research. Real-time experiments were conducted by researchers on PlanetLab nodes and it is very easy to use. Any researcher who wants to conduct experiments on PlanetLab nodes has to create a PlanetLab account initially. Few universities across the world have dedicated their systems to the PlanetLab network, which are considered as nodes and these nodes were used to conduct experiments.

Terminology of PlanetLab:

Site: Site refers to the physical location of the PlanetLab node. Ex: University of Minnesota Duluth, University of California Los Angeles etc., The details of your site can

be known by looking at "My Site" tab in your PlanetLab account. There are 699 sites worldwide.

Node: Each site dedicates at least one server to the PlanetLab network. If a university has dedicated one or more servers to the PlanetLab network, it can be allowed to use all the PlanetLab nodes across the network, in order to conduct experiments. Based on this criteria, many universities dedicate servers to the PlanetLab network in order to strengthen their research work. 1353 nodes were allocated by 699 sites worldwide [21] as depicted in Figure 3.2. A slice is required to connect to PlanetLab nodes for any authorized user.

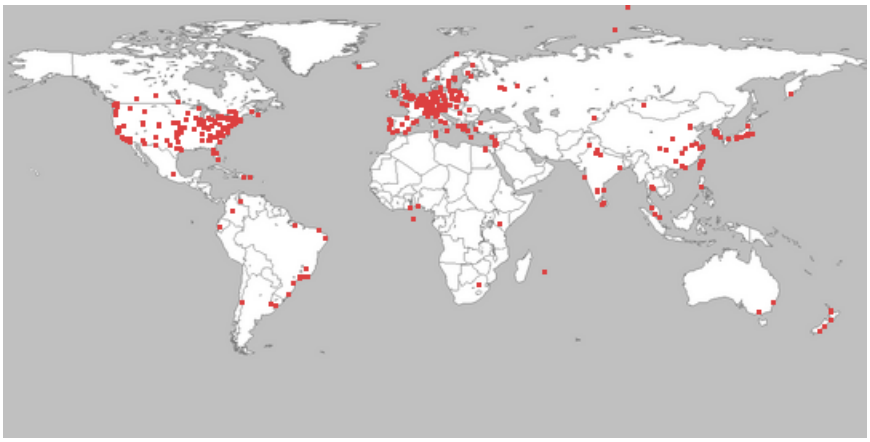

Figure 3.2: PlanetLab node locations.

Slice: Slice is used to access a PlanetLab node. A slice is a type of virtual machine, which is isolated from other slices. All the slices were provided few resources like CPU consumption, incoming and outgoing bandwidth, RAM memory, disc space and number of connections. User after creating PlanetLab account, has to request a slice from PlanetLab administrator. After getting assigned a slice to the account, the user can add any number of nodes to the slice. If a user wants to access the PlanetLab node (A), the node A has to be added to the slice in order to get connected, otherwise access would be denied. Slice has to be renewed periodically, otherwise it gets expired.

Silver: Collection of allocated resources on a single node is referred as Silver.

Virtualization: PlanetLab uses Virtualization technique and provide virtual machines to the user, rather than providing physical machines to the user. LinuxContainers (LXC), which is a container-based OS-level virtualization mechanism, is used to virtualize the physical machines. LXC provides complete isolation of one slice to the other slices. Now, LXC is the virtualization technique being used as it is the fast and most robust virtualization mechanism in the Linux kernel. PlanetLab nodes run on 2 operating system: Half of the nodes run on Fedora 14 (Laughlin) and the remaining half of the nodes run on Fedora 8 (Werewolf). Virtualization is the technique, which allows applications to share its physical resources and are more efficient in utilizing the resources.

PlanetLab's Acceptable Use Policy (AUP) specifies the limit of the resources being allocated to an application and to the slice. If a user doesn't follow the usage police or exceeds the regular usage policy on a specific node, PlanetLab kills the slice on that particular node and suspends the slice until the user provides a reasonable explanation to his/her behavior. If an application requires additional resources, they can be requested with a valid explanation from PlanetLab authority.

After the slice has been added to the account, the user has to generate a RSA key pair and add the public key to the PlanetLab account by going to the section "My Account", and click on the tab "Keys" and then "Upload new key". After a key has been added to the slice, it takes up to 48 hours for the user to connect to PlanetLab nodes through SSH protocol. The command used to connect to the PlanetLab node (planet3.cs.ucsb.edu) using my slice (umn_mounikac01) is shown below.

```
ssh -l umn_mounikac01 -i ~/.ssh/id_rsa planet3.cs.ucsb.edu
```

This command connects to the PlanetLab node planet3.cs.ucsb.edu, where the user can perform experiments. The user can disconnect anytime from the node by exiting.

Python Code for Connecting and Authenticating a PlanetLab user

Listing 3.1: 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 the 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)
```

XML-RPC is the programmatic interface provided by PlanetLab, which helps users to easily connect to PlanetLab nodes, perform measurements, manage applications and resources. The following python module authenticate() creates an object api_server, which has a method called AutoCheck(), which authenticates the user with username and password. The following code is provided by PlanetLab API in order to help users getting connected to PlanetLab nodes.

Retrieving information from PlanetLab nodes: In the previous step, we created an object called api_server, which has a method GetNodes(), which is used to get the information regarding PlanetLab nodes. This method takes 3 parameters. The first and main parameter is the authentication structure, which has the username and password information. The other two parameters were optional and they were used to filter the information. If only the first parameter is provided, GetNodes() method returns all the details of all the node structures, each node structure contains information about several named fields such as node ids, authority, host names, node status and other related information. The code section provided below, takes only the first parameter and returns the list of all the details of all the nodes.

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

If the user wants to get information only about a few nodes, we can do that by specifying the list of node IDs or hostnames in the second parameter of GetNodes method. The code is provided below.

```
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
```

If the user stills wants to filter down the details which he was looking for in the node information, he can do that by specifying the details in the third parameter of GetNodes() method. Therefore, the third parameter specifies which named fields to be returned. The following code only return node ids and hostnames of the nodes, which have boot state as "boot".

GetNodes() method gives the information about all the nodes in the PlanetLab network,

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

Listing 3.2: Python code that prints all the nodes associated with the slice with boot status

```
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 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']
```

where as GetSlices() method gives the information only about the nodes, which were associated to our slice. The code snippet Listing 3.2 gives details of the hostnames of the nodes, which are associated to our slice with boot state "boot".

The GetSlices() and GetNodes() methods are really important to know about many details of the node like node status, whether a node is active or not, hostnames, ids etc.,

3.2.2 Parallel Node Control via Vxargs

Network-based experiments need to be done on many machines for better and accurate. It is a very redundant and time consuming process to run the same set of commands on multiple PlanetLab nodes. The solution to this problem is using Vxargs command. Vxargs [27]

is influenced by xargs and a parallel methodology of open ssh tools such as pssh. Vxargs is a python command,through which users can perform execution of same commands on multiple PlanetLab nodes by providing Ip addresses of nodes or hostnames or both. The execution of the commands on multiple machines are done in parallel, rather than serially, which consumes a lot of time.

Commands that are most commonly used on PlanetLab nodes are ssh, scp, rsync [22]. Shell commands can be run on PlanetLab nodes using Vxargs. Vxargs needs at least two arguments to execute: a text file stating the list of PlanetLab nodes specifying either IpAddresses of nodes or host names and a command to run on these nodes. The concept of threads can also be used using Vxargs. We can use -P argument to specify the number of threads. A text file is passed using -a argument. Redirection is one of the best feature of Vxargs command, using which we can save the output/errors of each individual jobs using -o argument, so that the output file will be used for further analysis.

Consider the file "listOfNodes.txt", which contains Ip addresses of nodes, attached to the slice. If the user want to run a command (whoami) on all the nodes in the file "listOfNodes.txt", we can do it using Vxargs in the following way.

```
# python vxargs -t 40 -a listOfNodes.txt -P 15 -o Results
                ssh slice_name@{}   'whoami'
```

The variable in the above command is substituted with the IP address of the nodes present in the input text file "listOfNodes.txt". -P 15 stating that 15 threads can be run in parallel on different PlanetLab nodes. If the number of nodes in the input file are greater than the number of threads (15), then each thread after completing its job on one node, takes the job on other node given in the input file. This process continues until the command "whoami" is run on all the nodes given in the input file. Some of the nodes will be too slow and some does not work all the times. In order to skip these kind of defective nodes, we

can set a time out using -t argument. Time out is set to 40 seconds in the above command. If the node doesn't respond until 40 seconds, the thread releases the node and takes the job on other node. The output/error is generated for each node in a separate file, which is given the name of the PlanetLab node and stored in the Results folder.

Vxargs standard output: The standard output folder contains three files for each node, after Vxargs command is executed. The three output files for each node are: host.out, host.err and host.status where host is the host name of the node or Ip address as specified in the input file given to perform Vxargs command. Consider the following example., when Vxargs command is executed on the node planet3.cs.ucsb.edu, the output files generated will be planet3.cs.ucsb.edu.out, planet3.cs.ucsb.edu.err and planet3.cs.ucsb.edu.status. The output generated by executing the command on PlanetLab node will be stored in .out file. If an unexpected error was generated during the execution of the specified command, the details about the error are stored in .err file. Exit status value is stored in .status file. The output results folder is really helpful to see the output and analyze results and also to know the reason resulted in error on few nodes and so on., Vxargs also generates a file called "abnormal_list", which contains the list of nodes that have failed in executing the specified command.

3.2.3 PlanetLab-based Deployment of Dropbox

Dropbox has to be downloaded on PlanetLab nodes. We have conducted experiments on Dropbox on 50 PlanetLab nodes. Here are the steps to download Dropbox on a PlanetLab node:

1. Connect to the PlanetLab node (Example: pl1.tailab.eu) using the RSA key, which was added to your slice.

```
# ssh -i ../.ssh/id_rsa umn_mounikac01@pl1.tailab.eu
```

If the node is currently active, it connects to the specific PlanetLab node. Otherwise, it refuses the connection.

2. As all the PlanetLab nodes works on Linux servers, you will see the terminal like this:

Last login: Fri Oct 9 20:15:44 2015 from nomad196-123.d.umn.edu

umn_mounikac01@pl1.tailab.eu $

Enter cd so that it makes sure that you are in the home directory

cd

3. Download the stable 32-bit dropbox using the link below:

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

4. After downloading the dropbox, perform the sanity check, so that we are not going to clog the home directory.

tar -tzf dropbox.tar.gz

5. Extract the files from the downloaded dropbox folder

tar -xvzf dropbox.tar.gz

6. Run dropbox using the following command:

/.dropbox-dist/dropboxd

7. Once you run the command on step 6, you should the see the following:

"This client is not linked to any account... Please visit https://www.dropbox. com/-cli_link?host_id=7d44a67334d02c1 to link this machine."

8. Go to the link specified in the output using any web browser, where it asks you to enter the username and password of the dropbox. You are all set and the dropbox is linked to one of your accounts.

Download Dropbox on the PlanetLab nodes, on which we wish to conduct measurements. Figure 3.3 shows the flowchart about performing measurements on PlanetLab. In order to perform measurements on two PlanetLab nodes (Source and Destination), the fol-

lowing steps have to be follows.

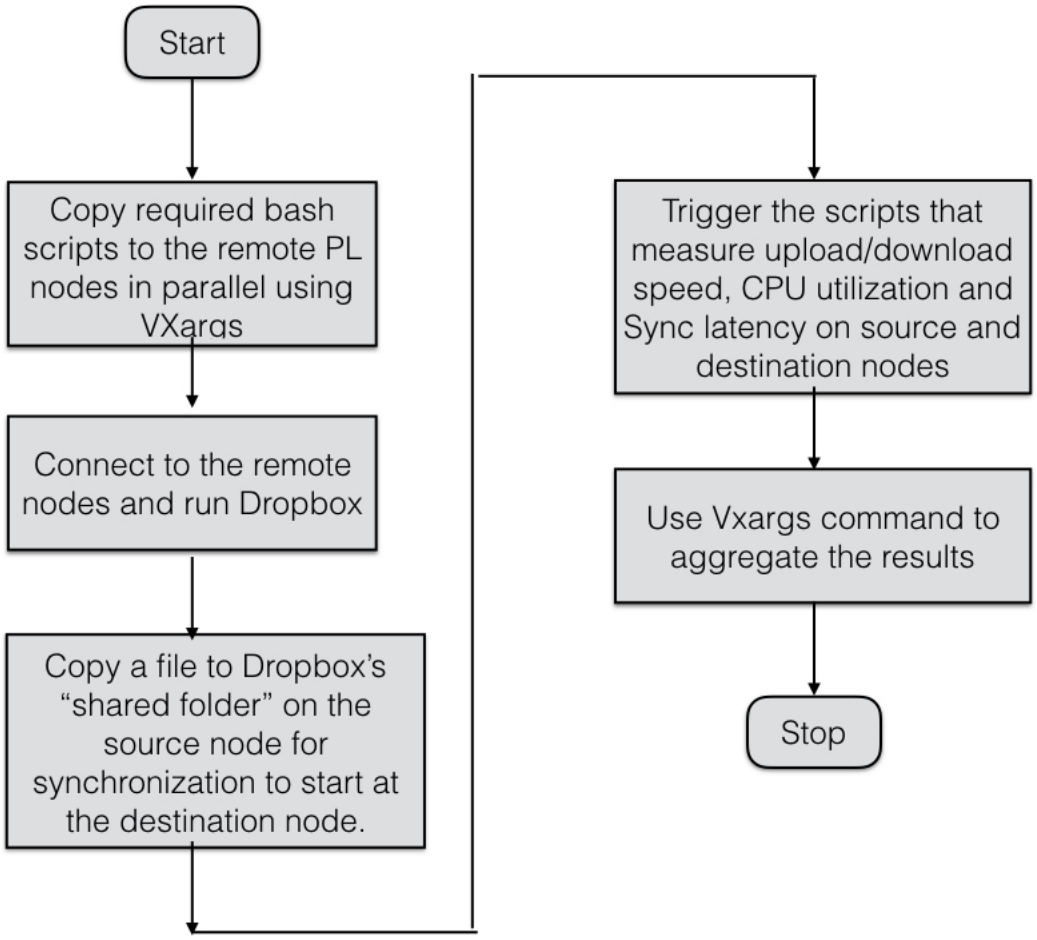

Figure 3.3: Flow chart about performing measurements on PlanetLab

1. Initially create a PlanetLab account and make it authenticated by PlanetLab Control (PLC) API. A slice is provided to each user, which has limited resources to be used by the user. Add few nodes to the slice, so that the user can perform measurements on these nodes.

2. Retrieve the nodes with "boot" status and collect the names of these PlanetLab nodes in a text file called "ListOfNodes.txt".

3. Copy the Dropbox related files and other related bash scripts to the nodes in "ListOfNodes.txt" file using Vxargs command.

```
# python vxargs -t 20 -P 10 -a ListOfNodes.txt scp -o
  stricthostkeychecking=no dropbox_scripts/* slice_name@{}
```

The about command produces 10 threads, which copies dropbox_scripts to all the nodes containing in the "ListOFNodes.txt" file and the time out period is given as 10 seconds.

4. After copying all the related files to the remote PlanetLab nodes, run the Dropbox application.

5. Move a file to Dropbox's "shared folder" on the source node, so that the file synchronization process begins at the destination node, following the Dropbox shared file mechanism.

6. In order to perform the measurements, as soon as the synchronization starts, trigger the bash scripts which measure upload speed, download speed, CPU utilization and synchronization latency.

```
# python vxargs -t 20 -P 10 -a ListOfNodes.txt -o Results
 ssh -o stricthostkeychecking=no -l slice_name -i
 ~/.ssh/id_rsa {} 'bash upSpeedCapture.sh'
```

The above command triggers the script "upSpeedCapture.sh" on each node and store the output in "Results" folder.

7. Use Vxargs command to aggregate the results on our local machine.

Measurements are conducted on multiple PlanetLab nodes, physically located at different location in order to have better analysis of results and come to a single-point conclusion. Figure 3.4 explains the flow chart of performing measurements on multiple nodes and aggregating the results. Initial four steps were same as in the flowchart described in Figure 3.3. Python random function is used to select the source and destination nodes randomly to perform measurements. So, there is one source node and multiple destination nodes. Source

node uploads a file in the "shared folder" and all the destination nodes have to download the file. On source node, Upload speed, CPU utilization and latency were recorded using bash scripts. On the destination nodes, download rates, CPU utilization and synchronization latency were measured and recorded using bash scripts. The results produced by each node were gathered on the local machine and further analyzed to better understand the behavior of Dropbox.

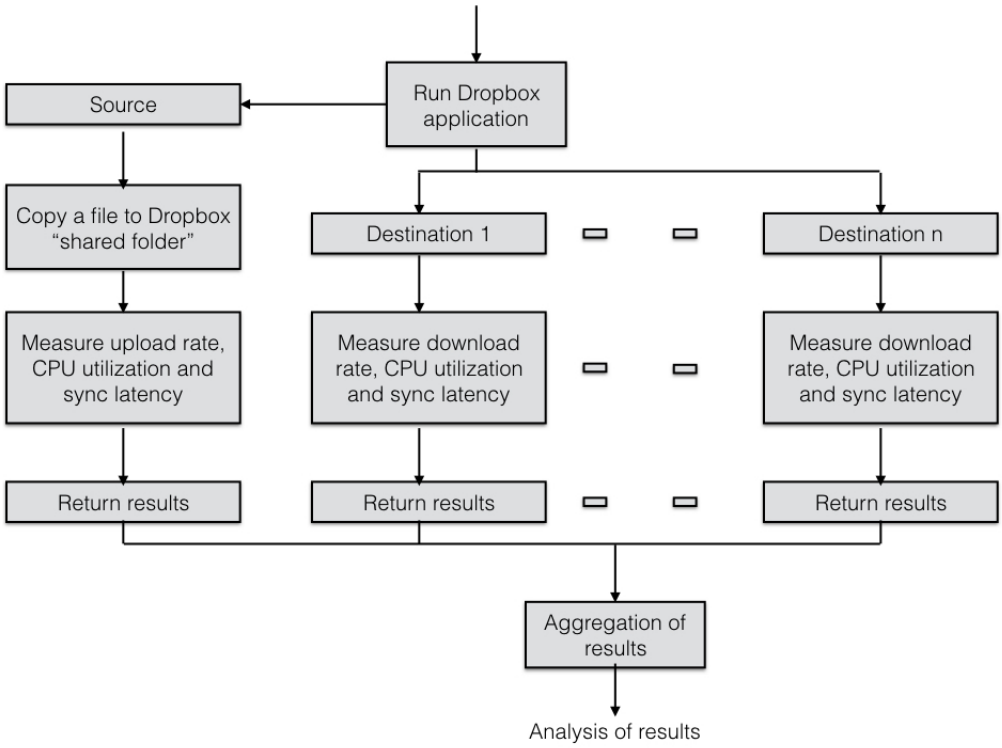

Figure 3.4: Flow chart of performing measurements on multiple nodes and aggregating the results.

3.3 Practical Issues

Few problems were encountered when using PlanetLab nodes. They are:

1. PlanetLab nodes have Linux-based Operating systems, half of the nodes run on Fe-

dora 8 version and the remaining half run on Fedora 14. The latest version is Fedora 23. So the PlanetLab nodes are running on very old versions, so it would have been really helpful and much easier, if PlanetLab nodes were updated to the new versions.

2. Each slice is given a limited resources like memory, bandwidth etc., sometimes the processes were killed by PlanetLab administrators because of exceeding the use of resources.

3. Some of the PlanetLab nodes were shut down due to maintenance. So, If the user did any work on a node, which later was shut down, the user would lose all the data stored on those nodes.

4 Measurement Results and Analysis

This chapter depicts the details of the experiments done to analyze the performance of Dropbox. PlanetLab is used to conduct experiments. The hardware specifications of every PlanetLab node is 2.67GHz CPU with 4 cores, 4 GB memory and a bandwidth of 15 MBps. Our study concentrates on understanding the attributes of Dropbox such as CPU utilization, synchronization latency, and downloading/uploading rates. The results obtained from the measurements were analyzed and represented in the form of graphs.

4.1 Decomposition of Cloud-based File Synchronization

Based on the existing measurements [12] [4] [14], Dropbox-like systems consist of basic chunk compression/processing and chunk delivery/transmission functions. The mix of such bandwidth-intensive tasks (e.g., chunk delivery) and computation-intensive tasks (e.g., chunk processing) in Dropbox enables seamless collaboration and file synchronization among multiple users. It is however known that such bandwidth-intensive and computation-intensive tasks will also introduce certain interference and create a severe performance degradation on the cloud virtual machines (VMs) [24] [23]. It is thus important to take a closer look into their system design and clarify if the latest cloud-based file synchronization protocols are aware of this new challenge and enabled related build-in functions.

We start from an experiment of two Dropbox users: one is the data source that uploads the file, and the other is the destination that needs to be synchronized. The data source is located in the University of British Columbia(UBC) and the destination is deployed in Si-

mon Fraser University (SFU). Both of them have similar hardware capacity with 1.7 GHz CPU, 4 GB memory and 1 Gbps Ethernet adaptor. Our access networks provide at least 20 MBytes/sec uploading and downloading capacity for these PCs, and the available throughput from our PCs to the Dropbox servers is around 15 MBytes/sec[1]. We link these two PCs with the same Dropbox account. Once we move a file to the Dropbox folder, this file will be synchronized to another PC. We are focusing on two metrics that are closely related to the synchronization latency: the CPU utilization, and the downloading/uploading rate on these two PCs.

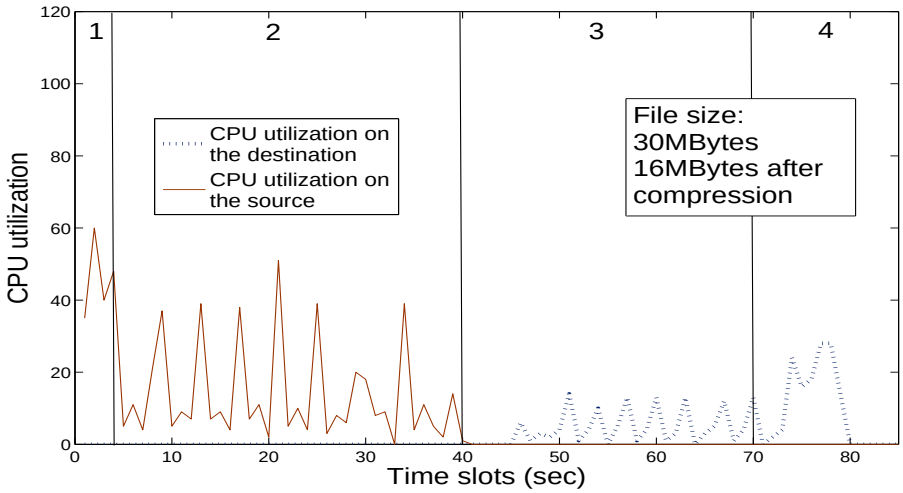

Figure 4.1: Client CPU utilization (file size 30MBytes)

We first synchronize a 500 MByte content between these two PCs. Figure 4.3 and Figure 4.4 present the CPU utilization and the downloading/uploading rate on these two Dropbox users, respectively. The solid lines refer to the data source, and the dotted lines refer to the destination. As we can see from Figure 4.3, the CPU utilization elevates sharply in the first 20 sec when we put the file into the Dropbox folder. During this time, there is no high-speed data transmission in Figure 4.4. A closer look shows that this elevating CPU

[1]This throughput is measured by *Initial Gap Increase and Packet Transmission Rate Tool* (IGI/PTR) [11]

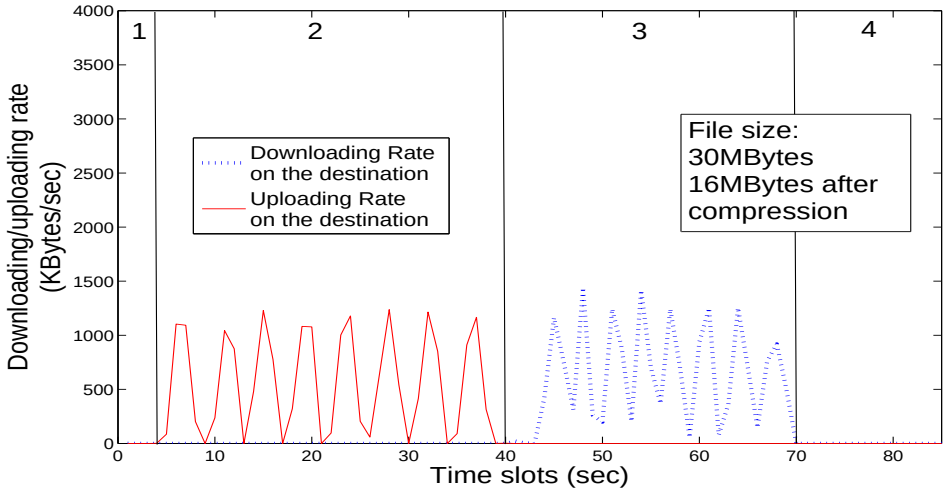

Figure 4.2: Client downloading/uploading rate (file size 30MBytes)

load corresponds to such file pre-processing as splitting the files into chunks and computing their hash values. We further check the packet level activities and find that the Dropbox client application (on the data source) is also communicating with the load-balancers to obtain the IP address of Dropbox delivery servers and sending chunk hashes to the delivery servers for comparison. This pre-processing stage is marked as *Stage 1* in the figures, which has remarkably elevated CPU usage without much data transmission.

After pre-processing, we can see that the uploading traffic on the data source starts to increase. In Figure 4.4, the uploading rate increases to around 1000 KBytes/sec. Meanwhile, the CPU utilization in Figure 4.3 decreases to around 50%. We mark this as *Stage 2*. It is worth noting that if we compare the 50% CPU utilization in Figure 4.3 (*Stage 2*) with the 1000 KBytes/sec uploading rate in Figure 4.4 (*Stage 2*), we can find that the uploading rate cannot cause such a high CPU utilization. An intuitive explanation is that the Dropbox client application is compressing the chunks while uploading. To verify this, we compute the total uploaded bytes and compare it with the original file size. We find that the total

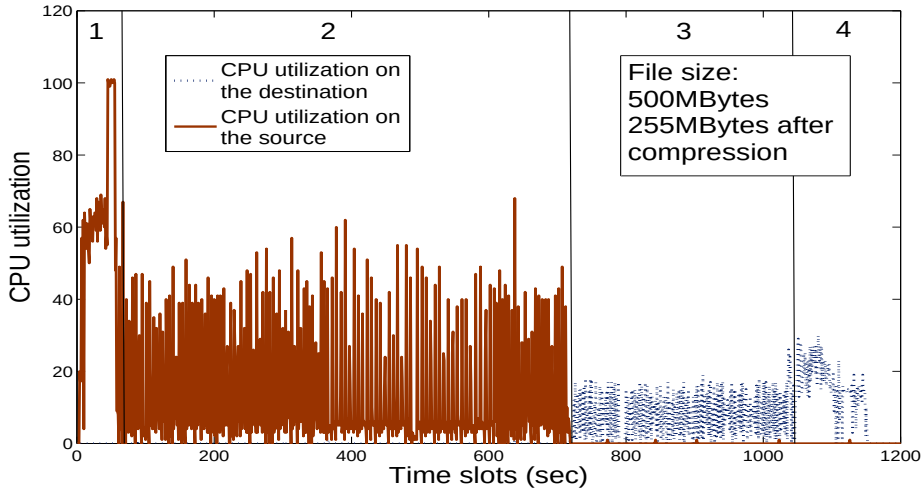

Figure 4.3: Client CPU utilization (file size 500MBytes)

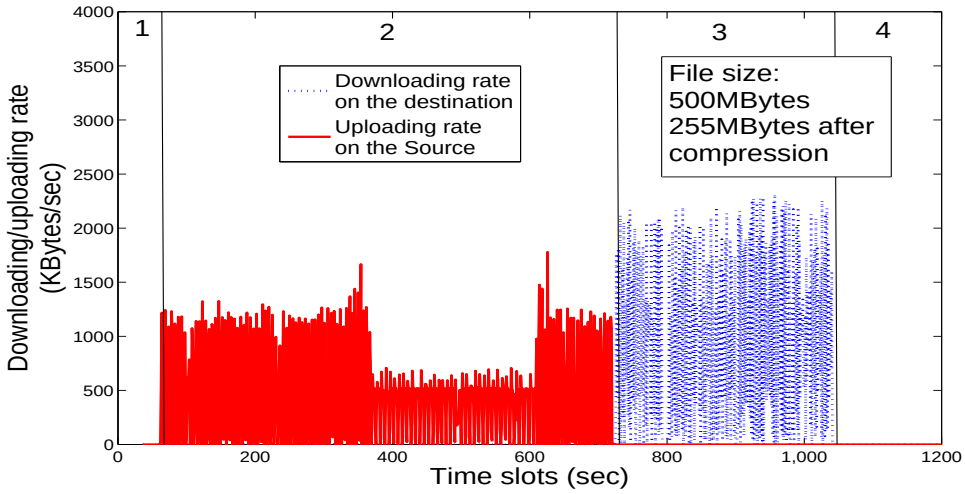

Figure 4.4: Client downloading/uploading rate (file size 500MBytes)

uploaded byte is around 255 MBytes which is much smaller compared to the original file size of 500 MBytes. It is easy to see that the file compression efficiently reduces the time cost in uploading.

As we can see from Figure 4.4, as soon as the data source finishes the uploading, the destination will start downloading (marked as *Stage 3* in the figures) from Dropbox servers.

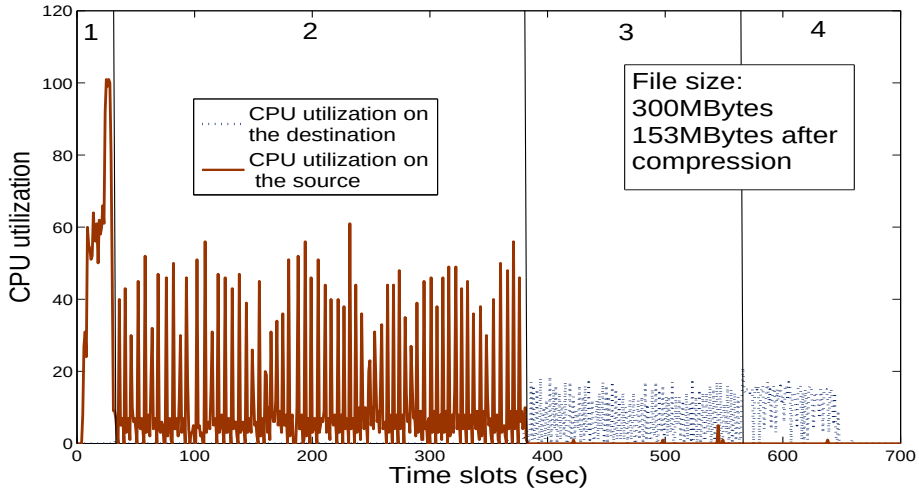

Figure 4.5: Client CPU utilization(file size 300MBytes)

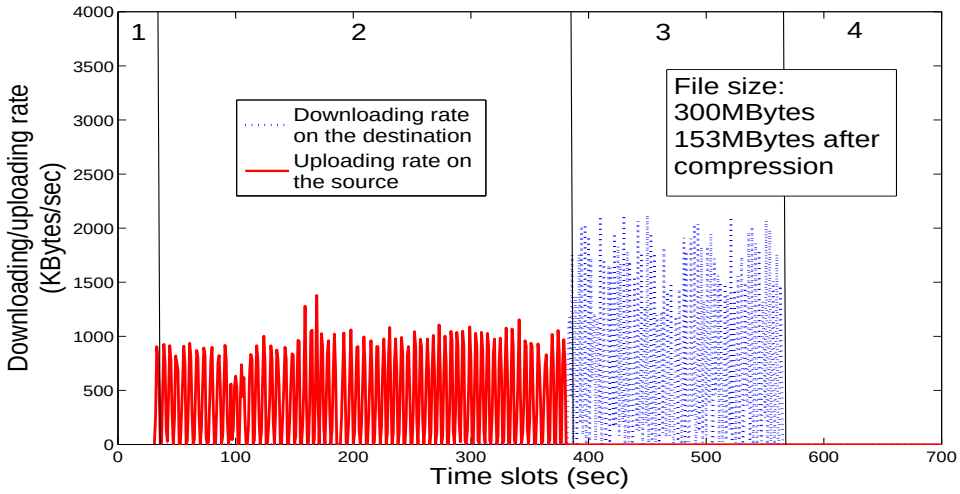

Figure 4.6: Client downloading/uploading rate (file size 300MBytes)

Figure 4.3 indicates that the CPU utilization (at the destination) during this stage is much lower compared to the uploading stage (at the data source). Since both servers have similar hardware configurations, we believe that this low CPU utilization is because the file decompressing process is much easier than the file compressing. Another noticeable feature is that the downloading will start only when the entire file has been successfully uploaded to the Dropbox servers. The reason is that the file is segmented and compressed on the

data source. The Dropbox servers are designed to merge the chunks together and then send the file to storage servers before further delivery. Although this design might not be the most efficient for minimizing synchronization latency, it largely avoids the possible errors during the file uploading and compressing. This distinguishes Dropbox from conventional file hosting systems that are often pipelined [18] [17].

When downloading finishes, the downloading rate at the destination drops to zero. However, Figure 4.3 shows that the CPU still keeps working for another 60 sec after the downloading, suggesting that there is a post-processing stage (marked as *Stage 4*). Since the files are segmented and compressed on the sender side, this CPU load is for decompressing the received chunks and merging them together.

In summary, our measurement shows that the Dropbox file synchronization can be decomposed into 4 cascaded stages: (1) *pre-processing*, (2) *uploading*, (3) *downloading*, and (4) *post-processing*. To generalize this observation and avoid possible bias, we have conducted a series of experiments and present representative results in Figures 4.5−4.1 and 4.6−4.2. We can see clear stages in all the experiments as marked in these figures. It is also worth noting that in Figure 4.5 and Figure 4.2, there is a very small delay (around 3 sec) between the end of uploading and the start of downloading stages. We believe that it is mainly due to such operation costs on the Dropbox servers as finding the right instances for delivery and sending the files to S3. This delay is relatively low (less than 5 sec in all cases), which can be largely ignored.

While these serial operations look natural, they may apparently less efficient than pipelined operations as in the traditional file storage systems, e.g., [18] [17]. Besides simplicity and better reliability in handling data, the uniqueness of cloud virtualization would be a key reason towards adopting the serial operations. It has been found that, for a virtual machine, traffic load can largely slow down the running of computation-intensive tasks (such as compressing/decompressing) and create a severe bottleneck in cloud-based systems [24]. This

is because the control and data paths in a virtualized network interface controller (NIC) are much longer than that of a non-virtualized counterpart; e.g., in EC2, the paravirtualized NIC involves 3 CPU interrupts and 3 data copies for receiving one packet, as contrast to 1 interrupt and 1 in a bare-metal NIC. The impact to multi-core CPUs is even higher with extra switches across CPUs to handle the interrupts. To avoid potential interference, the bandwidth-intensive and computation-intensive tasks shall be interleaved without overlap, as Dropbox does.

4.2 Analysis of Synchronization Latency

It is easy to see that the latest serial operations in Dropbox may apparently less efficient than pipelined operations. To understand the real-world performance of Dropbox, we will now investigate the synchronization latency between different Dropbox users. We still start from the synchronization between two users and then increase the user population to examine the system scalability.

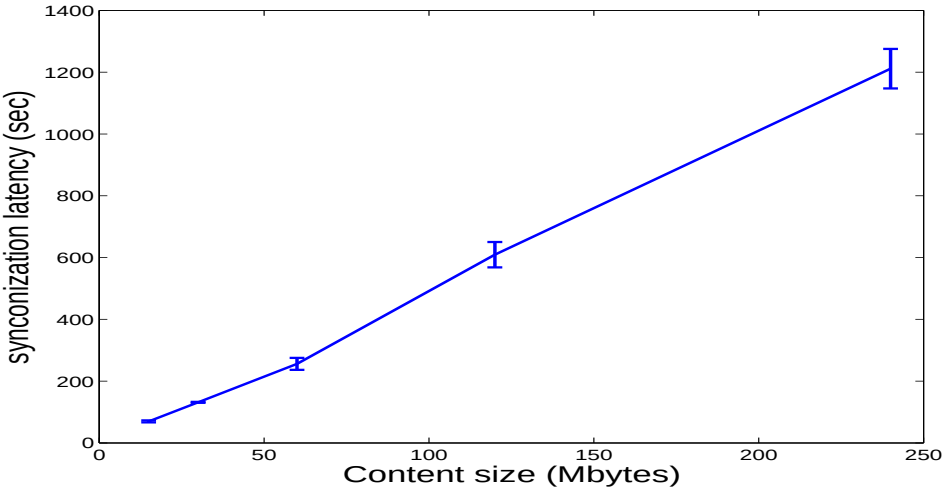

Figure 4.7: Synchronization latency of Dropbox

In the first experiment, the configuration is similar to that in the previous section, but

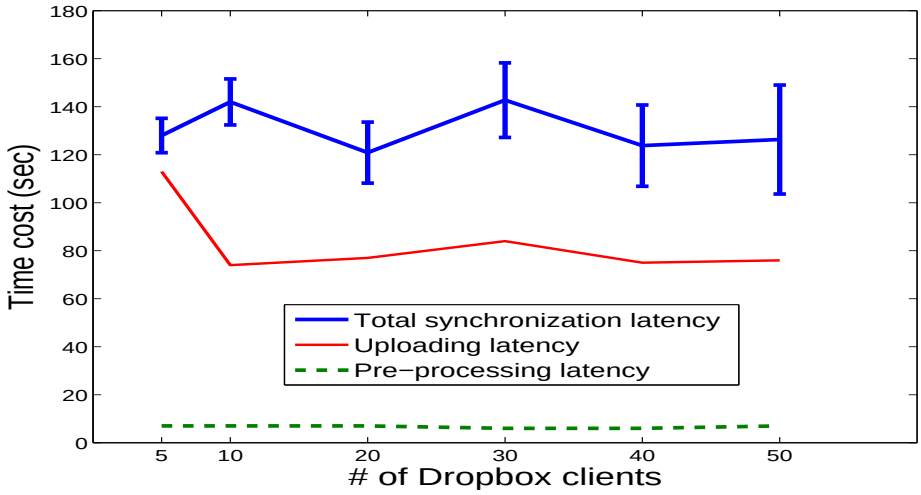

Figure 4.8: Time cost on the data source (uploader side)

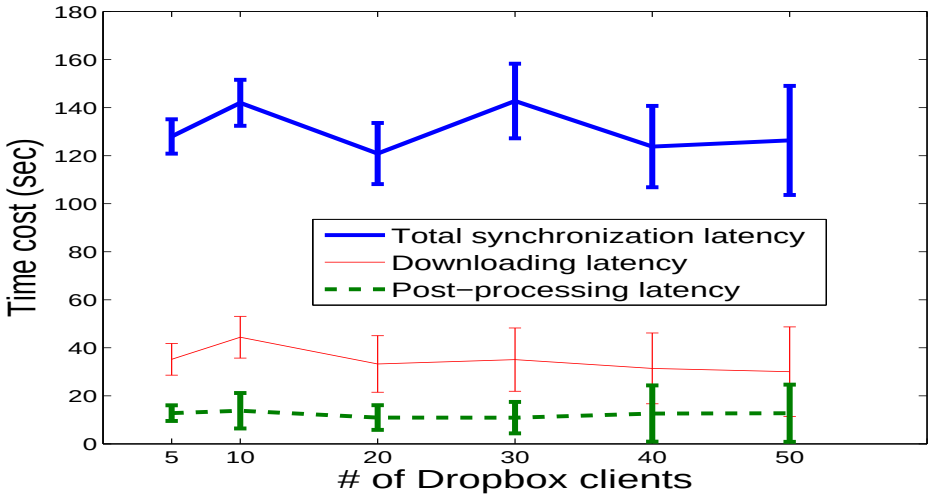

Figure 4.9: Time cost on the destination (downloader side)

the file size varies. To avoid triggering the Dropbox's caching functions (as mentioned in Section 3), we randomly generate the content of files in each experiment and test the file size of 15 MBytes, 30 MBytes, 60 MBytes, 120 MBytes and 240 MBytes, respectively. We run each experiment 4 times and present the average synchronization latency in Figure 4.7. It is easy to see that the synchronization latency increases roughly linearly with the file size. The standard deviation also slightly increases with larger files. This result shows that

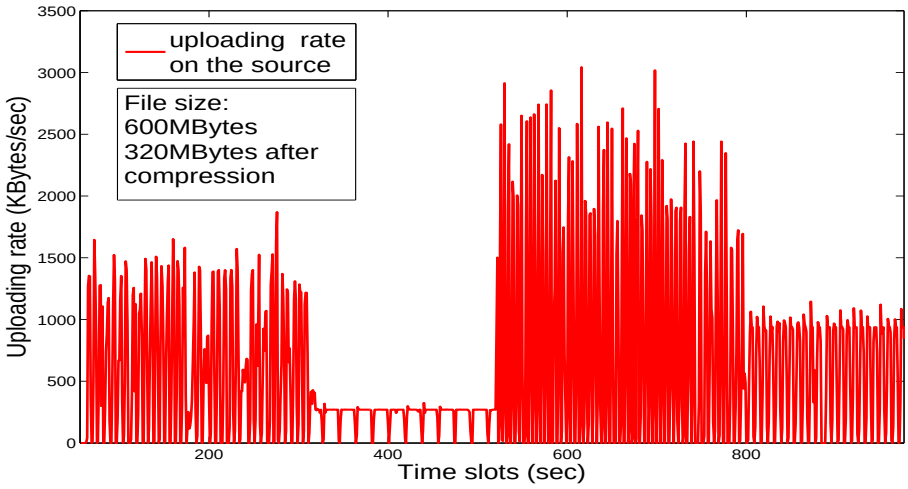

Figure 4.10: Uploading rate of the Dropbox client (file size: 600MBytes)

Dropbox can scale quite well with file size. The detailed statistics can be found in Table I.

Table 4.1: Synchronization latency with different file sizes

FILE SIZE	AVG	STD	MAX	MIN
15 MBytes	69.6 sec	3.2	72.0 sec	66.0 sec
30 MBytes	131.3 sec	1.5	133.0 sec	130.0 sec
60 MBytes	256.0 sec	19.2	278.0 sec	242.0 sec
120 MBytes	609.3 sec	41.0	644.0 sec	564.0 sec
240 MBytes	1211.3 sec	64.0	1275.0 sec	1147.0 sec

To examine the synchronization latency across more Dropbox users, we carry out a Planet-lab based experiment across 51 Dropbox users (one data source and 50 destinations) using a file size of 30 MBytes. We use this scale because the advertised capacity of one Dropbox account is currently 40 (based on Dropbox's official documents). Moreover, it is also hard to assume that a user will use one single Dropbox account to synchronize the files across over 50 computers in real-world[2].

Figures 4.8 and 4.9 present the time cost of different stages (the four stages that we

[2]This could be a reasonable assumption for some enterprise users. We will further explore this issue in our future works.

have mentioned in the last section) at the data source and destinations, respectively. We can see that the average time costs of these four stages are not sensitive to the increasing number Dropbox users. However, the variance of latency across different users increases with population, and thus individual users may face severe performance degradation when the system expands.

We also find that the time cost of file uploading is always more expensive than that of downloading. The pre-processing and the post-processing, on the other hand, are generally quite fast especially compared to data transmission. In particular, the pre-processing of a 500 MBytes file (before compressing) is around 60 sec, and the post-processing costs only 80 sec. The pre-/post-processing of smaller files (less than 1 MBytes), will cost even less time (around 10 sec). While this time cost is not significantly high, it can be further optimized via more efficient compressing/decompressing algorithms.

4.3 Further Discussions

Our experimental results indicate that the serial-based interference avoidance design scales well with user population in terms of average synchronization latency. The variance of the latency however increases with user population, implying that the great amount of users will also introduce mutual-interference, potentially leading to higher performance variance. In our experiments, some users can finish the synchronization of a 300 MBytes contents within 10 minutes while others may have to wait for more than 30 minutes. Such unfairness has also been complained by Skydrive and Gdrive users. While this would be acceptable for free services, it can severely hinder the commercialization of such systems for paid users. Our results also suggest that this problem is getting worse when the system scale expands. Providing both fast and fair services, or better yet guaranteed services, to all subscribers remains a challenging task for Dropbox-like systems.

In addition, from Figure 4.4 and Figure 4.6, the file uploading from Dropbox client application to the EC2-based Dropbox servers contribute to almost 60% of the synchronization latency. The uploading rates of Dropbox are mostly around 1 to 1.5 MBytes/sec whereas the downloading rates can achieve more than 2 MBytes/sec[3]. We also find that the uploading rates are quite unstable over time (for example, in Figure 4.10), which is not the case in the downloading stage. Since our experimental platform is dedicated for the measurements, we believe that this is due to the arriving loads of other Dropbox users and the follow-up mutual-interference.

To verify this, we have deployed a set of Xen-based[4] local VMs to examine the overhead of both incoming and outgoing traffic. Our experiment shows that the receiving of TCP traffic is more expensive than sending. In particular, the receiving rate of 200 MBytes/sec will cost more than 40% CPU on a virtual machine that has 7.5 GB memory and 2 virtual cores. Meanwhile, the sending rate of 200 MBytes/sec will only cost around 20% CPU on this VM. For example, when more users upload their files to the Dropbox servers, such an increasing receiving traffic will greatly slow down the servers and unavoidably prolong the file processing as well as message forwarding. Such interference potentially creates a severe bottleneck in the system.

Based on the above measurements, it is easy to see that the serial-based interference avoidance protocol archives good performance and scalability in Dropbox system. Although such interference is successfully avoided for individual users, we can see that the large amount of users will also introduce mutual-interference, potentially leading to higher performance variance and higher overheads while uploading local contents to the cloud VMS.

[3]Note that this is not due to the limited uploading capacity of Dropbox users because our dedicated servers have more than 20 MBytes/sec uploading capacity and 15 MBytes/sec throughput to the Dropbox servers.

[4]Amazon EC2 uses Xen-based virtualization [29]

5 Peer-to-Peer Enhancement for Cloud-based Synchronization

5.1 Framework Design

In the existing implementation of Dropbox, the file is processed, compressed and then transferred from source to the cloud (Amazon EC2 servers). After gathering all the chunks of the file, the file is transferred from Amazon EC2 server to Amazon S3 server for the purpose of storage. Now the file is transferred from cloud to the destination. If both the source and destination are online at the same time, transferring the file from source to cloud and then from cloud to destination increases synchronization latency.

In our proposed system, If both the source and destination are online at the same time, we transfer the file from source to destination using socket programming and then transfer the file from destination to the cloud and store it in Amazon S3 servers. In this way, the synchronization latency decreases to a great extent. We can check whether the destination system is online or not by a ping command. If the destination computer replies to our ping message with in 5 seconds, it means that both source and destination are online at the same time. If they are online, transfer the file from source to destination using socket programming and then transfer the file from destination computer to the cloud. Thereby we are replacing the "user -> cloud -> user" strategy with "user -> user -> cloud" strategy. If the destination computer goes offline in the middle of the file transfer, the system

follows the approach of Dropbox to transfer the file. By this method, we can completely eliminate the latency occurred by pre-processing, compressing, calculating hash values etc., thereby decreasing the synchronization latency. As the socket programming is much faster, our proposed system greatly reduces the file synchronization latency with minimal system overhead.

If the destination is not online, the file is transferred through the approach of Dropbox. i.e., the file is uploaded to cloud storage server and is pulled by the destination machine, when the system goes online.

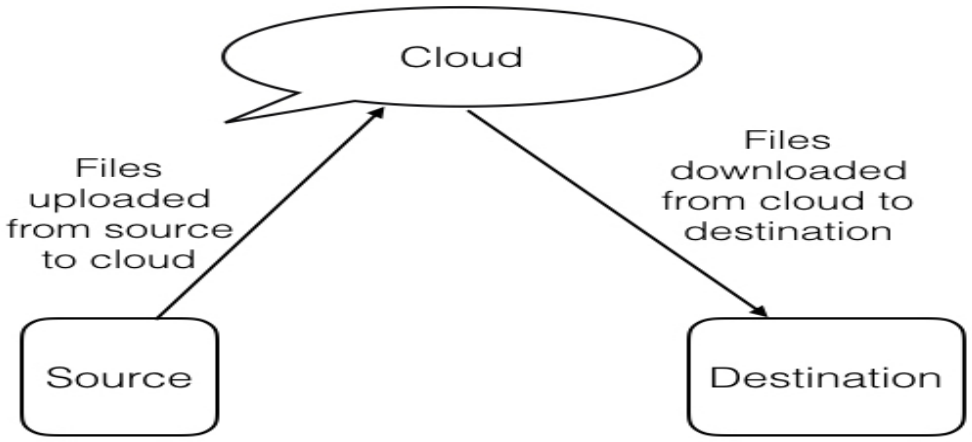

Figure 5.1: Dropbox functionality

The code to check if a system is online or not is given below. Source machine checks to see if the destination machine is online or not. If there are multiple destination machines, we need to check the online status of each of the destination and determine the best way to synchronize the file.

```
1 //Get the Inet address of the destination machine
2 InetAddress inet = InetAddress.getByName(ipAddress);
3 System.out.println("Sending Ping Request to " + ipAddress);
4 boolean isConnected = false;
```

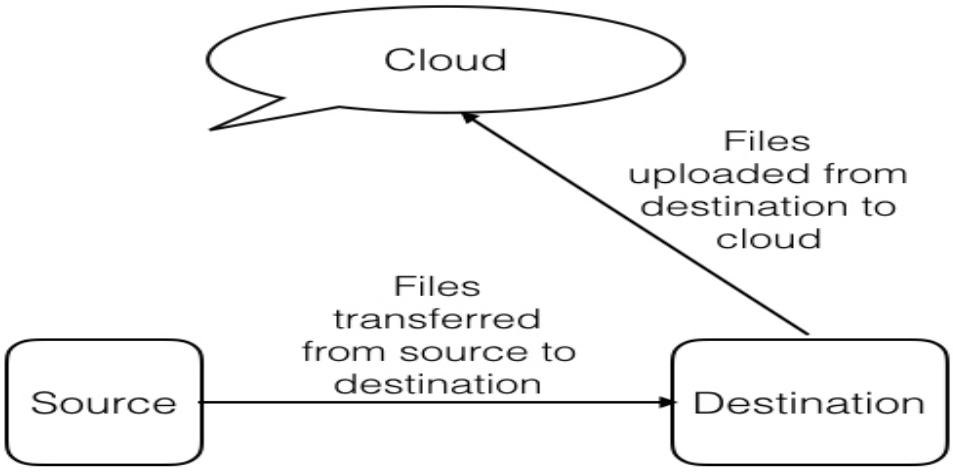

Figure 5.2: Improved system functionality when both source and destination are online

```
5  //Sending the ping request to see if destination
6  //is reachable or not
7  if(inet.isReachable(1000)){
8       isConnected = true;
9     }
10   System.out.println(isConnected == true ?
11       "Host " + ipAddress + " is reachable" :
12        "Host " + ipAddress + " is NOT reachable");
```

If the destination machine is online, i.e., if isConnected == true, then transfer the file using socket programming to the destination machine and then to cloud. If the destination machine is not online i.e., if isConnected == false, copy the file to Dropbox folder on the source machine, so that the synchronization takes place using classic Dropbox model. In our system, we also consider multiple destination machines, so we are using Multi threaded socket programming. Each thread handles a single connection between source and destination.

```
/* If the destination is reachable, accept the connections from
 destination machines (clients) and start transferring the file
 using socket programming */
if(isConnected){
    new ServerThread(serverSocket.accept(),FILE_TO_SEND).start();
}else{
//If the destination is not reachable, copy the file to Dropbox
    File source = new File(FILE_TO_SEND);
    File destination = new File("/Users/Dropbox/abc.zip");
    InputStream is = null;
    OutputStream os = null;
    try {
        is = new FileInputStream(source);
        os = new FileOutputStream(destination);
        byte[] buffer = new byte[1024];
        int length;
        while ((length = is.read(buffer)) > 0) {
          os.write(buffer, 0, length);
        }
      }
      finally {
         is.close();
         os.close();
      }
    System.out.println("File copied to dropbox");
```

We have implemented this system on PlanetLab and conducted measurements to calculate synchronization latency, upload speed and CPU utilization. The results show that our proposed system greatly reduces the synchronization latency when both the source and destination are online.

5.2 Performance Evaluation

5.2.1 Measuring Synchronization Latency

We performed an experiment to find the synchronization latency of the proposed system between two PlanetLab nodes with different file sizes ranging from 1 MBps to 400 MBps. This experiment is conducted multiple times on different PlanetLab nodes in order to obtain better accuracy. We start from the synchronization between two users and then increase the user population to examine the system scalability. The average synchronization latency between 1 source and 1 destination for different file sizes is presented in the Figure 5.3

As we can see in the Figure, the synchronization latency increases with file size. This indicates that the synchronization latency using proposed system increases linearly with the file size. It also indicates that our system is scalable for large files. The synchronization latency for 100 MB file is 11 seconds, which is much less compared to Dropbox synchronization which takes approximately 8 minutes to synchronize 100 MB file.

We increased the number of destination machines to which the file is synchronized with and checked to see if the system is scalable or not with multiple number of users. Our results show that as the number of Clients (destination machines) increases, synchronization latency increases too. The average synchronization latency between 1 source and 2 destination machines for different file sizes is presented in the Figure 5.4.

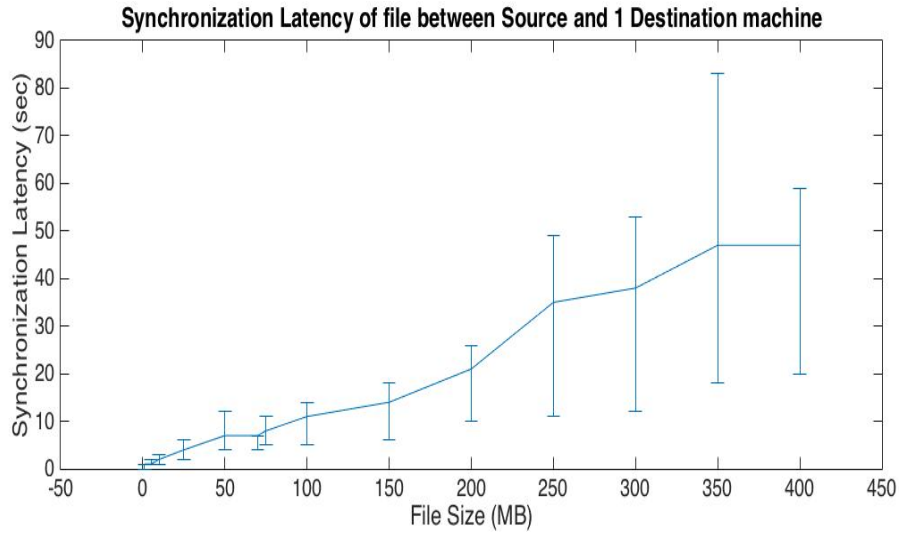

Figure 5.3: ErrorBar plot of Synchronization Latency between source and 1 destination for different file sizes.

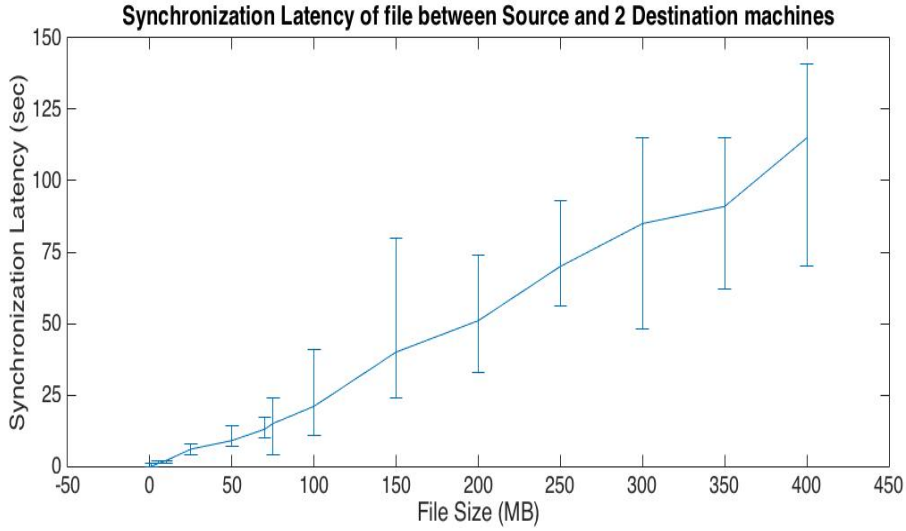

Figure 5.4: ErrorBar plot of Synchronization Latency between source and 2 destination machines for different file sizes.

If we compare Figure 5.3 and Figure 5.4, we can see that for file size of 400MB, it took on average of 47 seconds if there is one destination and it took 115 seconds if there were 2 destination machines. This indicates that as the number of clients(destinations) increases, downloading speed on destination machines decreases, thereby increasing the synchronization latency.

Figure 5.5 depicts the synchronization latency for different file sizes, when a file is synchronized from source to 3 destination machines. The synchronization latency is further increased compared to synchronizing with 2 destination machines in Figure 5.4

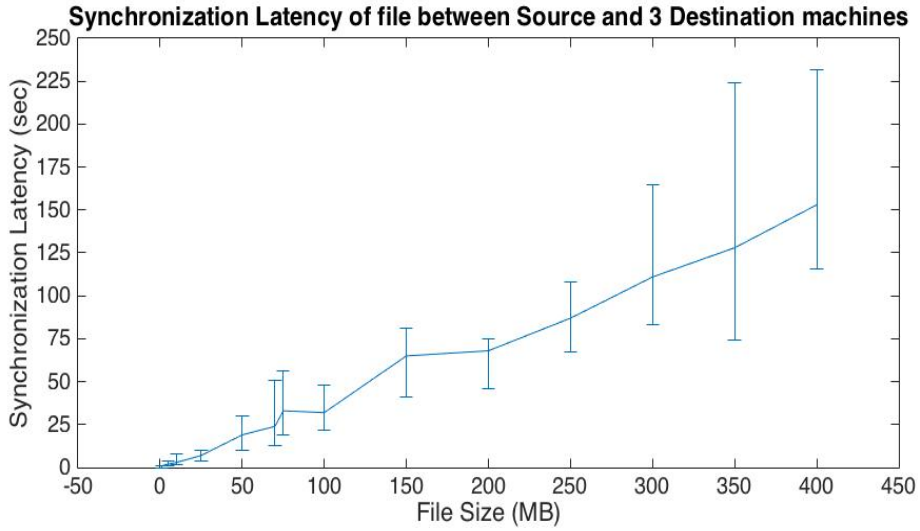

Figure 5.5: ErrorBar plot of Synchronization Latency between source and 3 destination machines for different file sizes.

Figure 5.6 depicts the synchronization latency for different file sizes, when a file is synchronized from source to 5 destination machines. If we compare Figure 5.5 and Figure 5.6, we can see that for file size of 400MB, it took on average of 153 seconds if there were 3 destination machines and it took an average of 240 seconds if there were 5 destination machines. By analyzing these figures, we can say that synchronization latency is directly proportional to number of clients (destination machines).

Figure 5.7 shows the synchronization latency for different file sizes, when a file is synchronized from source to 10 destination machines. In our experiments, we have choose the maximum number of clients (destination machines) to which the file is synchronized as 10. As it is highly unlikely that a group of more than 10 people are online at the same time, we believe that experimenting with a maximum of 10 destination machines is sufficient for a good analysis. As each machine has limited uploading bandwidth, sending the file

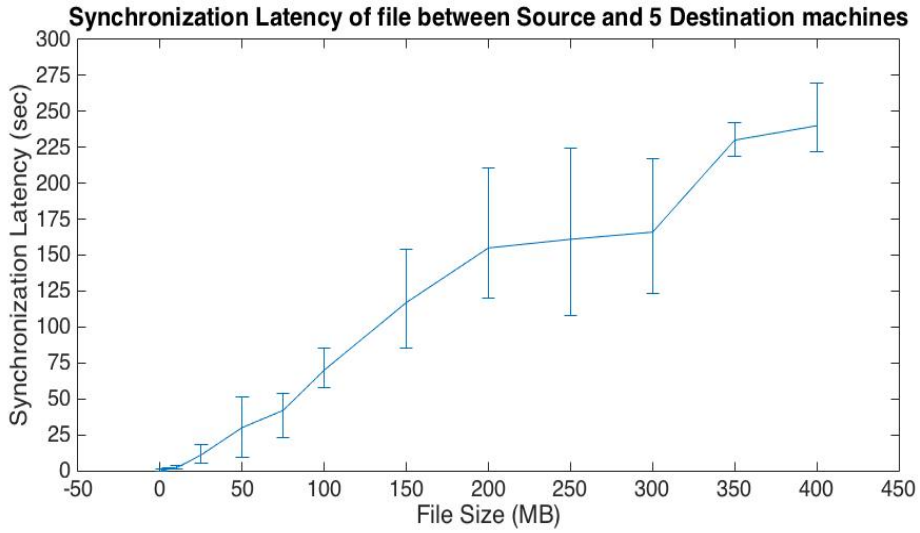

Figure 5.6: Errorbar plot of Synchronization Latency between source and 5 destination machines for different file sizes.

to 10 destination machines almost saturates the entire bandwidth, thereby increasing the synchronization latency.

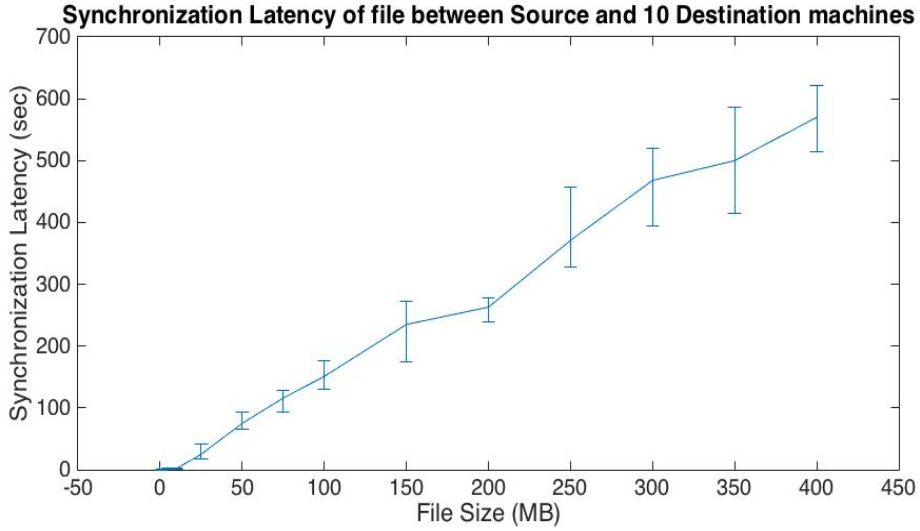

Figure 5.7: Errorbar plot of Synchronization Latency between source and 10 destination machines for different file sizes.

In order to examine how synchronization latency varies with the number of clients (destination machines) with which the file has to be synchronized, we represent the synchronization latency for a 25MB file with varying number of users. For 1 client, it took 4

seconds to synchronize and for 10 clients, it took 25 seconds to synchronize a 25MB file. More results are depicted in the Figure 5.8. This figure indicates that the synchronization latency increases linearly with the number of destination machines, to which the file has to be synchronized.

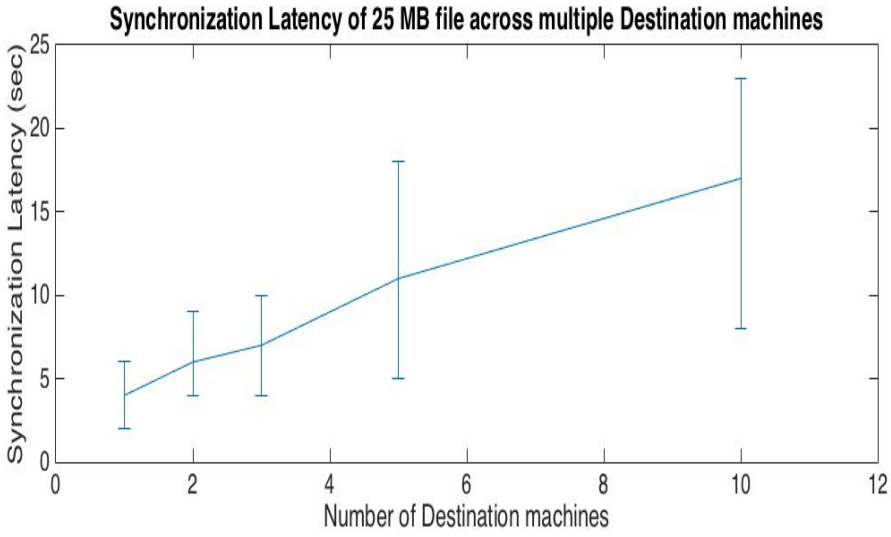

Figure 5.8: Errorbar plot of Synchronization Latency of 25MB file for different number of destination machines

We also represented Synchronization latency for 100MB file and 300MB file for different number of destination machines in the Figure 5.9 and Figure 5.10 respectively. According to Figure 5.9 and Figure 5.10, we can agree that the synchronization latency is directly proportional to the number of destination machines. i.e., as the number of destination machines increase, the synchronization latency increases too.

We have represented CDF graph for Synchronization latency between source and 1 destination machine for different file sizes. By inspecting the Cumulative Distribution Function graph (Figure 5.11), we can see that 50% of the destination machines can download the file in less than 35 seconds and 90% of the clients (destination machines) can download the file in less than 45 seconds. This indicates that the synchronization latency is very low and our proposed sytem works much faster than the classic Dropbox model.

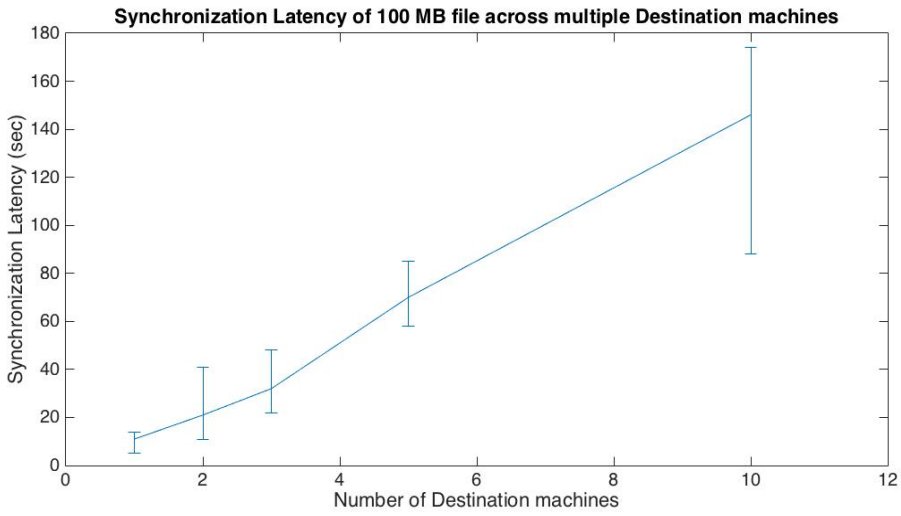

Figure 5.9: Errorbar plot of Synchronization Latency of 100MB file for different number of destination machines

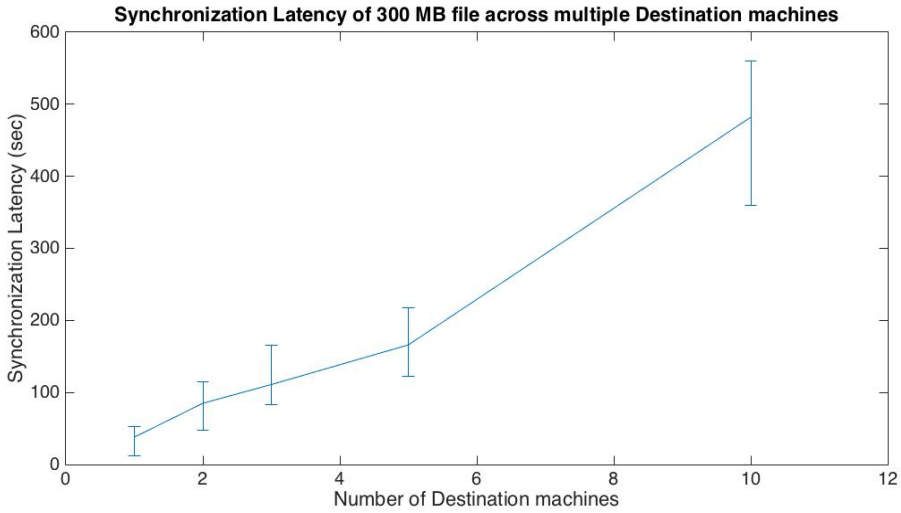

Figure 5.10: Errorbar plot of Synchronization Latency of 300MB file for different number of destination machines

Figure 5.12 compares the synchronization latency of classic Dropbox system with our proposed system. Dotted line refers to the synchronization latency of existing system and the thick line represents the synchronization latency of the proposed system. The detailed statistics can be found in the Table below. These results indicate that the proposed system performs much better with very less synchronization latency compared to classic Dropbox

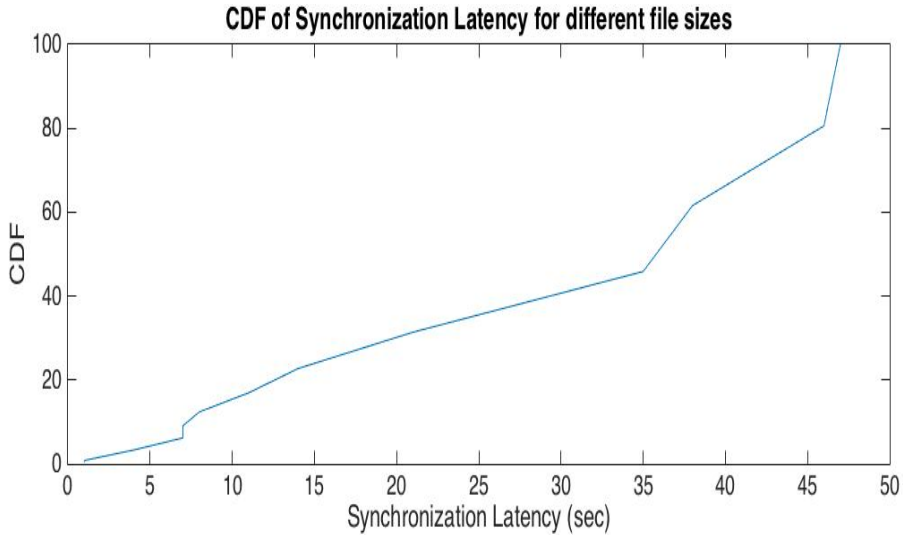

Figure 5.11: CDF of Synchronization Latency for different file sizes

model, when the users are online at the same time. As shown in the table, synchronization latency of 240 MB file using Dropbox model takes 1211.3 seconds, where as it only takes 34 seconds using our proposed model, which shows that the performance of the system can be greatly improved using our proposed system. Our results also show that our proposed system is very scalable with increase in file size and also with increase in number of destination machines, to which the file has to be synchronized.

Table 5.1: Synchronization latency comparision

FILE SIZE	Latency of Dropbox	Latency of proposed system
15 MBytes	69.6 sec	3 sec
30 MBytes	131.3 sec	5 sec
60 MBytes	256.0 sec	7 sec
120 MBytes	609.3 sec	12 sec
240 MBytes	1211.3 sec	34 sec

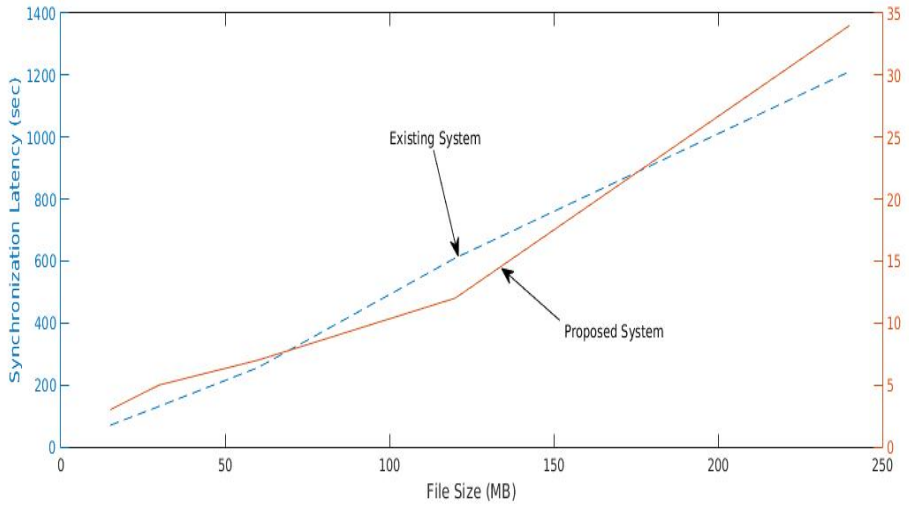

Figure 5.12: Comparision of Synchronization latency between Dropbox and our proposed system

5.2.2 Measuring Upload Speed and CPU Utilization

We have conducted another experiment to measure the Upload speed on Server using a tool named Wireshark. Wireshark is used to collect packets transferred between source and destination and it is a very convenient tool to perform packet level analysis. Figure 5.13 represents the upload speed on server, when a file is synchronized from source to 3 destination machines. As we inspect the figure, we can analyze that the upload speed rate increased with the size of file and it became constant at 15 MBps, when it reached maximum upload speed capacity.

The Cumulative Distribution Function graph of CPU utilization for a 30 MB file is presented in the Figure 5.14. From the figure, we observe that CPU utilization of 60% of the destination machines is less than 30% when synchronizing a 30 MB file and the CPU utilization of 90% of destination machines is less than 33% which is quite reasonable. This indicates that our system has very less system overhead.

The CDF graph of CPU utilization of a 100 MB file is shown in the Figure 5.15

Figure 5.16 depicts the CDF of CPU utilization on server for different file sizes ranging

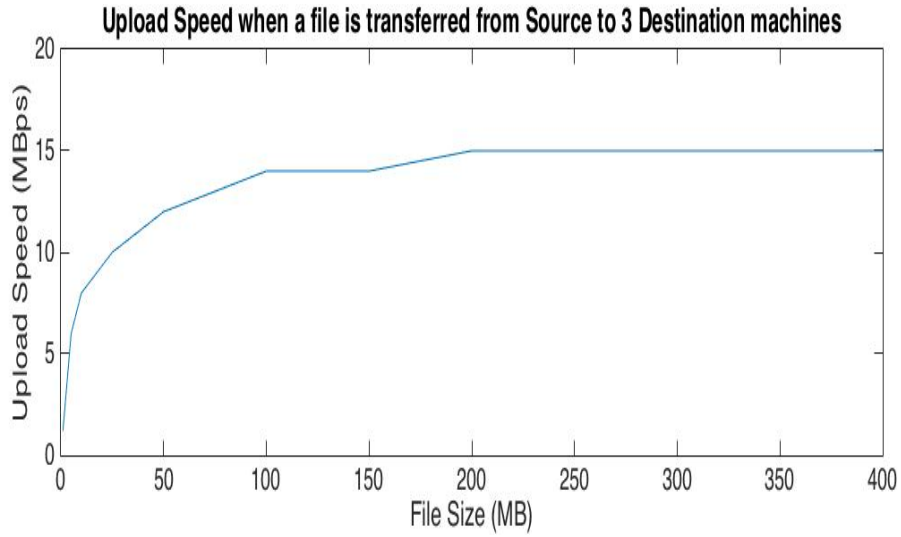

Figure 5.13: Uploading rate between source and 3 destination machines for different file sizes.

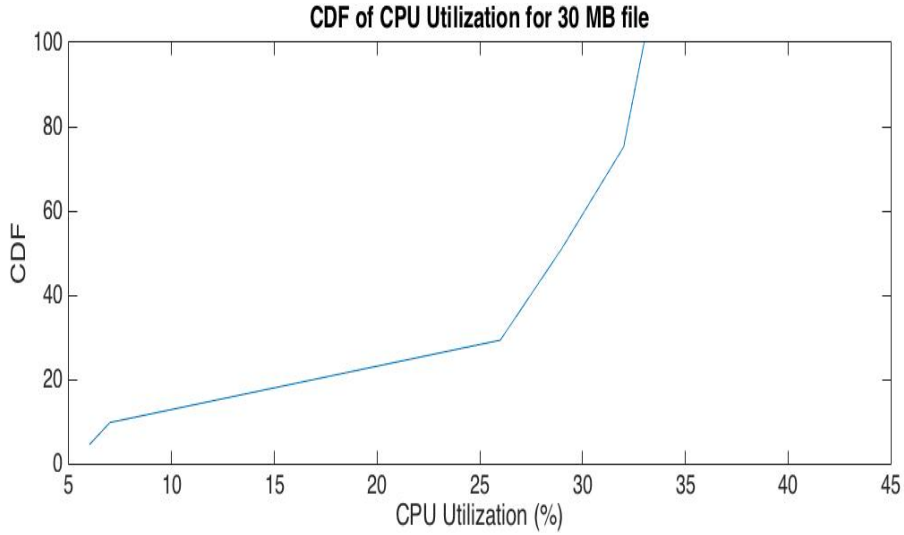

Figure 5.14: CDF of CPU Utilization of 30MB file

from 1 MB to 400 MB. From the figure, we can analyze that 90% of the destination machines utilize less than 35% of the CPU resources when synchronizing a file of different sizes. This indicates that our system is scalable and also has very less system overhead.

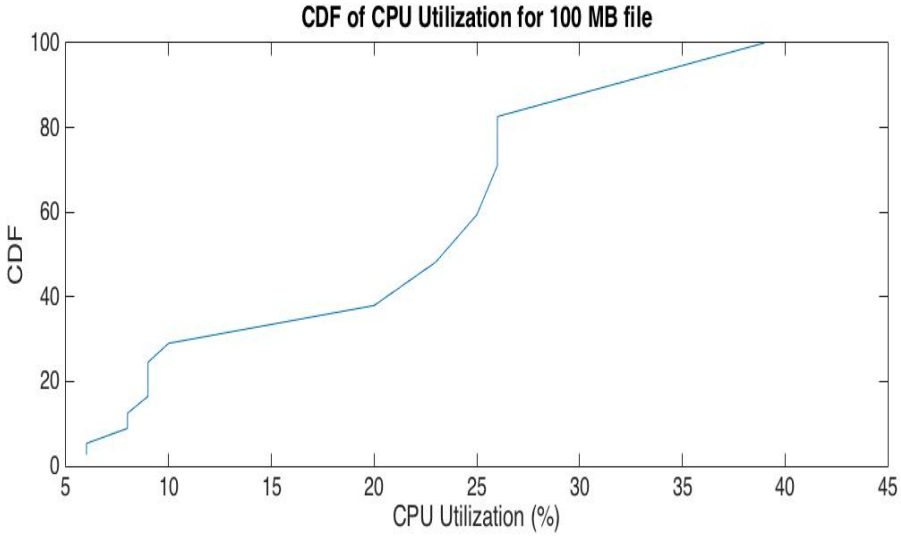

Figure 5.15: CDF of CPU Utilization of 100MB file

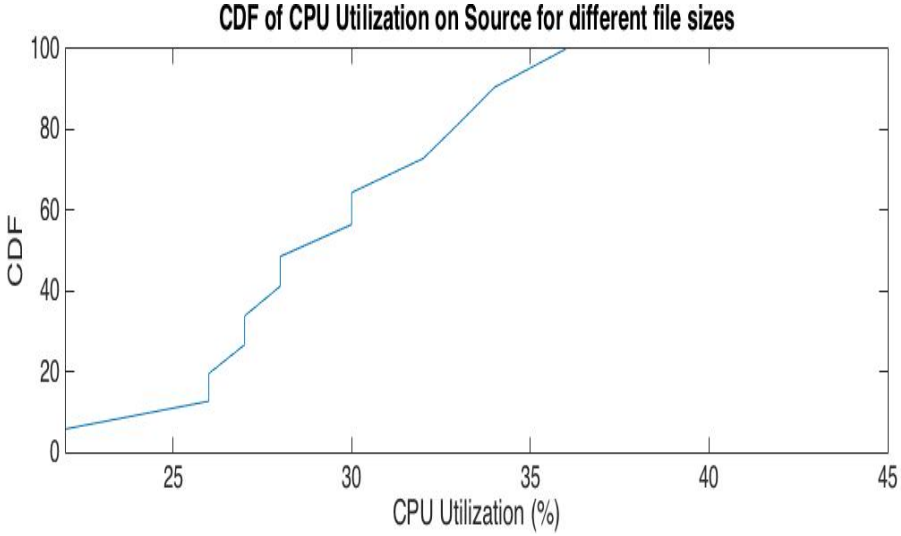

Figure 5.16: CDF of CPU Utilization for different file sizes.

6 Conclusions and Future Work

In this thesis, we took a close look to understand Dropbox-like cloud-based file synchronization and collaboration systems. We identified the cascaded computation and communication operations in Dropbox and showed that such a design can acheive reasonable performance and scalability. Meanwhile, we found that such a design is also introducing higher latency and cost. To address this problem, we proposed a system which considers user's online status and decides the way to synchronize the file with lower synchronization latency. The evaluation results showed that our proposed system greatly reduces the synchronization latency with minimal system overhead when some Dropbox users are online at the same time.

This thesis represents an initial attempt towards understanding the design of Dropbox-like cloud file synchronization systems. We expect that our findings help with optimizing these systems as well as with migrating more Internet services to cloud platforms. There are many possible future directions to explore. We are particularly interested in efficient and scalable collaboration/interaction among the users with decentralized cloud deployment, as well as examining other similar applications to further generalize our findings.

Some of the future work includes the following: As the number of destination machines increase, our proposed system is going a have a lot of system overhead. For example: if there are 100 destinations for which the file has to be synchronized with, the source has to ping all the destination machines to check if they are online. Therefore, we can try to use something like ICMP Sweep in order of determine, if IP addresses are online at the same time. If there is a firewall that's blocking the socket programming traffic, we can get the

port which is used by Dropbox and use that port for transferring the file from source to destination using socket programming. We can also consider some cases, where we can send the file from source to cloud and also from source to destination at the same time, when both the source and destination are online at the same time.

Bibliography

[1] S. Androutsellis-Theotokis and D. Spinellis. ``A survey of peer-to-peer content distribution technologies". In: *ACM Computing Surveys (CSUR)* 36.4 (2004), pp. 335–371 (cit. on p. 11).

[2] R. Bindal, P. Cao, W. Chan, J. Medved, G. Suwala, T. Bates, and A. Zhang. ``Improving traffic locality in BitTorrent via biased neighbor selection". In: *Distributed Computing Systems, 2006. ICDCS 2006. 26th IEEE International Conference on*. IEEE. 2006, pp. 66–66 (cit. on p. 8).

[3] P. Casas, H. R. Fischer, S. Suette, and R. Schatz. ``A first look at quality of experience in personal cloud storage services". In: *Communications Workshops (ICC), 2013 IEEE International Conference on*. IEEE. 2013, pp. 733–737 (cit. on p. 14).

[4] 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. 1, 32).

[5] I. Drago, M. Mellia, M. M Munafo, A. Sperotto, R. Sadre, and A. Pras. ``Inside dropbox: understanding personal cloud storage services". In: *Proceedings of the 2012 ACM conference on Internet measurement conference*. ACM. 2012, pp. 481–494 (cit. on p. 14).

[6] *Dropbox*. URL: https://www.dropbox.com/ (cit. on pp. 1, 12).

[7] *Dropbox Users Save 1 Billion Files Every 48 Hours*. URL: https://www.dropbox.com/news/ (cit. on p. 16).

[8] *Gdrive*. URL: https://drive.google.com/ (cit. on pp. 1, 12).

[9] G. Goncalves, I. Drago, A. P. Couto da Silva, A. Borges Vieira, and J. M. Almeida. ``Modeling the dropbox client behavior''. In: *Communications (ICC), 2014 IEEE International Conference on*. IEEE. 2014, pp. 1332–1337 (cit. on p. 14).

[10] R. Gracia-Tinedo, M. Sanchez Artigas, A. Moreno-Martinez, C. Cotes, and P. Garcia Lopez. ``Actively measuring personal cloud storage''. In: *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*. IEEE. 2013, pp. 301–308 (cit. on p. 14).

[11] N. Hu and P. Steenkiste. ``Evaluation and Characterization of Available Bandwidth Probing Techniques''. In: (cit. on p. 33).

[12] W. Hu, T. Yang, and J. N. Matthews. ``The Good, The Bad And The Ugly of Consumer Cloud Storage''. In: *ACM SIGOPS Operating Systems Review, 44(3):110-115, 2010*. ACM. 2010 (cit. on p. 32).

[13] W. Hu, T. Yang, and J. N. Matthews. ``The good, the bad and the ugly of consumer cloud storage''. In: *ACM SIGOPS Operating Systems Review* 44.3 (2010), pp. 110–115 (cit. on p. 14).

[14] Z. Li, C. Wilson, Z. Jiang, Y. Liu, B. Zhao, C. Jin, Z. Zhang, and Y. Dai. ``Efficient Batched Synchronization in Dropbox-likeCloud Storage Services''. In: *Proc. ACM/IFIP/USENIX Middleware , 2013* (cit. on pp. 1, 32).

[15] 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. ACM. 2014, pp. 115–128 (cit. on p. 13).

[16] 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: (cit. on p. 14).

[17] F. Liu, Y. Sun, B. Li, and X. zhang. ``FS2You: Peer-Assisted Semi-Persistent Online Hosting at a Large Scale". In: *IEEE Transactions on Parallel and Distributed Systems, 21(10) 1442-1457, 2010* () (cit. on p. 37).

[18] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. ``On the Scalability of Data Synchronization Protocols for PDAs and Mobile Devices". In: *IEEE Network, 16(4) 22-28, 2002* () (cit. on pp. 2, 37).

[19] C. Marshall and J. C.Tang. ``That syncing feeling: Early user experiences with cloud". In: *Proceedings of the 2012 ACM conference*. ACM. 2012 (cit. on p. 14).

[20] *Microsoft OneDrive file sync problems*. https://www.kuppingercole.com/blog/small/microsoft-onedrive-file-sync-problems. Accessed: 2015-07-20 (cit. on p. 15).

[21] *PlanetLab Homepage*. http://planet-lab.org/. Accessed: 2015-07-20 (cit. on pp. 19, 20).

[22] *Rsync Homepage*. https://rsync.samba.org/. Accessed: 2015-07-21 (cit. on p. 25).

[23] R. Shea and J. Liu. `` Network Interface Virtualization: Challenges and Solutions". In: *IEEE Network, 26(5):28-34, 2012* () (cit. on pp. 1, 32).

[24] R. Shea and J. Liu. ``Understanding the Impact of Denial of Service Attacks on Virtual Machines". In: *Proc. IEEE/ACM International Workshop on Quality of Service (IWQoS), 2012* (cit. on pp. 1, 32, 37).

[25] *Skydrive*. URL: http://windows.microsoft.com/skydrive/home (cit. on pp. 1, 12).

[26] N. Sudharsan and D. K. Latha. ``Improvising Seeker Satisfaction in Cloud Community Portal: Dropbox". In: *Proceedings of the 2013 IEEE international workshop*. IEEE Press. 2013 (cit. on p. 14).

[27] *Vxargs Homepage*. http://vxargs.sourceforge.net/. Accessed: 2015-07-20 (cit. on p. 24).

[28] H. Wang, R. Shea, F. Wang, and J. Liu. ``On the impact of virtualization on dropbox-like cloud file storage/synchronization services". In: *Proceedings of the 2012 IEEE 20th international workshop on quality of service*. IEEE Press. 2012, p. 11 (cit. on p. 14).

[29] *Xen-based Amazon EC2*. URL: http://en.wikipedia.org/wiki/Amazon_Elastic_Compute_Cloud/ (cit. on p. 42).

[30] Y. Zhang, C. Dragga, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. ``*-Box: 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*. USENIX Association. 2013, pp. 2–2 (cit. on p. 17).