SWITCH ASSISTED PEER TO PEER

by

Sriram Selvam

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

School of Computing

The University of Utah

August 2020

The University of Utah Graduate School

STATEMENT OF THESIS APPROVAL

The thesis of **Sriram Selvam**

has been approved by the following supervisory committee members:

Robert Preston Reikenberg Ricci	, Chair	**05/06/2020**
		Date Approved
Jacobus Erasmus Van Der Merwe	, Member	**05/06/2020**
		Date Approved
Sneha K Kasera	, Member	**05/06/2020**
		Date Approved

and by **Ross T Whitaker** , Chair/Dean of

the Department/College/School of **Computing**

and by David B. Kieda, Dean of The Graduate School.

ABSTRACT

Cloudlab provides a flexible testbed for research and education of future cloud computing. On receiving a request for a research environment, Cloudlab identifies the resources satisfying the constraints and assigns it to the user. Our thesis focuses on the crucial step of loading the user's choice of disk image to the machine.

Frisbee is the system that facilitates the image loading task. A Frisbee server process is spawned upon request for an image, Frisbee client running in target node downloads the image and installs it. The Frisbee server process can serve multiple clients simultaneously using multicast.

In this thesis, we study in detail the performance of the existing Frisbee system and its shortcomings. We propose a new Switch Assisted Peer to Peer (SWP2P) transfer system that addresses the shortcomings by significantly reducing the load on the Frisbee server and the traffic on trunk links in Cloudlab. SWP2P builds an image availability database on the switch to keep track of the availability among clients connected to it. The system then uses the database to redirect requests to other clients instead of the Frisbee server. We further evaluate the performance of the new and old system, where we show that SWP2P transfer reduces the server and trunk load by a minimum of 21% to a maximum of 96% depending on the level of concurrency among the clients.

Dedicated to my loving wife, Anneswa, for always motivating me to reach higher and to my mother, who has worked all her life to get me where I am.

TABLE OF CONTENTS

LIST OF TABLES

ACKNOWLEDGMENTS

I wish to express my most sincere gratitude to my advisor Dr. Robert Ricci who paved the way for me to obtain the master's degree with his continuous support. If it were not for his confidence in me and his brilliant guidance, this project's goals would not have been realized.

CHAPTER 1

INTRODUCTION

In Cloudlab [2,3,4], when initializing experiments, they are loaded with disk images of user's choice. Disk images generally contain the operating system and the user's data in some cases. Each node is formatted and installed with a fresh copy of the disk image before the user takes control of the experiment. As Cloudlab supports experiments of large scale, the disk loading system should be capable of high speed and scalability to guarantee low latency in provisioning experiments.

The Frisbee [1] system used by the Cloudlab for disk image distribution has a server and client component. The clients can request a specific piece of an image. The Frisbee system that facilitates the image transfer has the following features:

1. Multicast-based [7] server process to transfer the compressed image.
2. Decompressing the image on the client-side.
3. Writing the decompressed information to the disk.

When an experiment consists of multiple nodes, during the initial creation, all nodes request for the same image. As there is a single server node that is responsible for disk image distribution, the goal is to minimize the load on the server. The multicast-based transfer was adopted to achieve this goal. However, due to the nature of the multicast transfer, the clients that join the multicast group later do not receive the data

sent earlier. This scenario causes the multicast Frisbee server to send out multiple copies of the same chunk; the effects of the phenomenon mentioned above are studied in Chapter 3. In this thesis, we introduce a new transfer mechanism for the Frisbee system that overcomes the limitations of multicast and provides better performance.

The proposed Switch Assisted Peer to Peer (SWP2P) system leverages the concurrency among clients to reduce the load on the server. From our observation, the network transfer takes significantly less time than the disk image decompression and disk write operations. Longer duration of image decompression and disk write provide us a window of time, where each client has received the whole image and is waiting for the decompression and disk write to finish. Thus, the client is available for a long time after receiving the image to server the requests from peers.

In addition to the above, as the Frisbee client facilitates the installation of the required disk image, the user's experiment cannot be started before its completion; thus, all the system resources are available to the client. This allows the client to serve the image to other peers without any negative impact on its operations. The switch maintains an image availability database, to redirect the requests from clients to other peers having the same piece of the image. This task is accomplished by using a user-programmable switch that provides access to core switch functionalities through APIs and supports user programs.

1.1 SWP2P Working Principle

The SWP2P system works by allowing each client to act as a peer in a P2P fashion; the switch redirects the request messages towards clients that have the requested

piece of the image. A switch module running on user-programmable switch supports keeping track of the availability of image among different clients and redirecting the requests appropriately.

The process of keeping track of the image availability in the switch eliminates the need for building an overlay network that will be required by the traditional P2P systems. The strategic position of the switch in the topology allows it to have the total visibility of the image availability in all connected clients. The concept can also be extended to support clients connected across multiple switches by making the switch share a consolidated report for the availability among all clients connected to it.

- **Thesis Statement:** In the Frisbee disk loading system, Switch Assisted Peer-to-Peer image transfer can significantly lower load on the server while maintaining high performance.

We start by looking at related research work on Chapter 2 and then analyze the existing Frisbee system and areas that have scope for improvement in Chapter 3. We further discuss in detail the design and implementation of the proposed SWP2P system in Chapter 4. Finally, we evaluate SWP2P against multicast-based Frisbee system in Chapter 5 where we show that SWP2P significantly reduces the load on the server and trunk links by 21% to 96%.

CHAPTER 2

RELATED WORK

In this chapter, we discuss related works to SWP2P. We found various enhancements on multicast, peer to peer transfer, and content-based addressing schemes that were related to our work. We discuss in detail the ideas from other research articles and how SWP2P differs from them.

Atkinson *et al*. [5] studied the process of loading disk images in a multitenant datacenter and the need for optimizing their delivery. Their work describes in detail various aspects of disk image loading in Cloudlab and the need to optimize the delivery and reduce the load of the disk image server.

The Light-weight Multicast Services (LMS) [6] attempts to provide a scalable, efficient, and reliable multicast transfer mechanism. In LMS, the routers tag and steer the control packets to preselected endpoints and perform fine-grain multicast to guide responses to a subset of the group without transport-level processing.

Each router selects a single replier for each source in a multicast group. When a request comes from a nonreplier link, the router steers the request towards the replier link. If the replier has the requested information, then it sends the data. If the replier does not have the requested information, it sends a request to the switch. On receiving the request from replier-link, the switch forwards the request upstream. Such a router is

termed as a turning-point router, as it redirects the request packets from its links.

When forwarding the request upstream, the turning-point router adds its IP address to the request message. In this way, the request travels upstream, until it reaches a replier with the requested data or the multicast source. The node addressing the redirected requests from the turning-point router creates a multicast reply packet and encapsulates it in a unicast packet addressed to the turning-point router. This mechanism is termed as directed multicast.

Both SWP2P and LMS are trying to leverage the switch's location for its advantage. The difference is that LMS is still acting as an enhancement on top of the existing multicast protocol, whereas SWP2P introduces the concept of peer-to-peer serving and uses unicast transfer.

In the LMS method, only one link connected to the switch is selected as the replier. Thus, all the further requests from clients connected to the switch are addressed to the same replier. In SWP2P, the requests are redirected to all the clients having the requested data in a round-robin fashion.

In addition to the above, the LMS method is not exploiting the concurrency among clients in their overall runtime. LMS performs like multicast, in terms of leveraging only the transfer time concurrency. In the evaluation chapter, we show how SWP2P performs better by leveraging overall concurrency.

We further explored peer-to-peer systems that prefer closest peers to identify solutions that attempt to solve similar problems. In [17], the researchers show how choosing the neighboring peer can be done with a modest overhead in organizing and maintaining the overlay network. In [12], Wei Li *et al*. show how network tomography

can be used to understand and select closest peers in a P2P network. However, this method requires extensively probing the network from a tracing node and building network topology. It does not provide guarantees that if the data are available with a peer connected to a local switch, then the peer will be selected. SWP2P provides an alternate method to prefer the closest available peer by leveraging the switch's location in the topology and maintaining the availability database at the switch.

BitTorrent-based P2P [13] system is built for an environment where not all peers are necessarily cooperative. Thus, it uses tit-for-tat unchoking mechanisms to prefer cooperative peers. However, in our environment, we can assume all peers are cooperative as the peers are under our Cloudlab's control at the time of reloading. Thus, BitTorrent-based P2P implementation causes unnecessary overhead to peer selection when deployed in Cloudlab.

The Frisbee system follows Application-Level Framing [14] principles to achieve its fast and scalable performance. Frisbee system divides the image into chunks, where each chunk is useful by itself immediately after they are received. As each chunk can be individually decompressed and written to disk, the system does not have to wait until all the chunks in the image are received. This benefit will be lost when using BitTorrent-based P2P system as they do not follow Application-Level Framing principles, and the image will be split into pieces by the BitTorrent protocol. Our SWP2P design considers chunk boundaries in transfer, thus allowing the application to reap benefits from the existing Frisbee design.

In summary, SWP2P design can provide better performance than P2P transfer due to the following factors:

1. No requirement to maintain overlay networks like in P2P.
2. Ability to choose the nearest neighbor with minimal overhead by leveraging switch's location in topology and avoiding probing.
3. Takes advantage of Application Level Framing used by the Frisbee system.

As our method of redirecting the requests is based on the availability of the content (image data) among clients, we studied content-based addressing and routing [11] and content-centric networking [15, 16].

Content-centric networks (CCN) provide a nearby cache for popular content by using novel addressing schemes and that use name of the content known as ContentName [16]. A user requiring some specific content sends out an interest packet [16] on all available interfaces, including the ContentName. A router receiving the interest packets checks if the content is available in its buffer memory, called ContentStore [16]. If found, then the router delivers the information directly to the user. If not found, the router checks the CCN Forwarding Information Base (FIB) to see if any neighboring nodes have the requested content. If there is a hit on CCN FIB, then the request is forwarded to the neighbor. On a miss, in CCN FIB, the request is discarded.

The SWP2P design locates the closest available cache of the requested data by maintaining the availability among clients on the switch. This is possible as we know the topology in Cloudlab, and we can leverage switch's location in the topology. By using existing IP addressing, the SWP2P design eliminates the requirements for major changes in the overall routing mechanism. In the evaluation chapter, we show how SWP2P achieves significantly better results than multicast and P2P transfer while avoiding the cost of maintaining an overlay network using the existing IP addressing scheme.

CHAPTER 3

ANALYSIS OF THE EXISTING FRISBEE SYSTEM

To understand the bottlenecks in the existing Frisbee system, we analyzed the logs collected from the Frisbee server instances. The logs were collected from 1st January 2019 to 10th April 2019. We begin our analysis by looking at the data from a single server. Then we go through in detail about various terms present in the log and further proceed to analyze aggregate statistics collected over the total time.

This server instance is serving the Ubuntu-16 x86 image to four Frisbee clients whose client ID is mentioned in the Figure 3.1. The image UBUNTU16-64-X86.ndz:20 is of size 709 MB. Images are split to 1 MB chunks, and chunks are further divided into1024 KB blocks. Thus, the image being transferred has 709 Chunks, thus 709*1024 = 726016 Blocks.

In the log presented in Figure 3.1, concurrency among the clients is measured in the following manner. A client is considered as being concurrent if it has at least one other client running along with it. The Figure 3.2 further shows how the concurrency is measured.

In Figure 3.2, the example server has three clients, among which client 2 is 100% concurrent, as it has some other client running along its total lifetime. On the data presented in Figure 3.1, it is seen that there are four clients, and they are highly

concurrent (>99%). Since the clients are concurrent, the server should have sent only a single copy of each block in the image, though it can send a small number of duplicate copies of the block as few clients are not 100% concurrent. In the server report, we can see that the image has 726016 blocks, so the number of blocks sent on multicast should be close to this number.

However, the server has sent 1483776 blocks, which is 2.04 times the original number of blocks. Sending higher than the required number of blocks adds stress to both the Frisbee server and the trunk links that connect between switches. To understand the effect of this phenomenon, we introduce a new parameter called multicast transfer efficiency defined below:

Multicast Transfer Efficiency (MTE) = Blocks in Image/Blocks sent by the server.

For the server in Figure 3.1, the multicast transfer efficiency is calculated below:

MTE = 726016/1483776 = 0.489.

The optimal MTE value is 1.0, indicating only one copy of the image is sent out by the server. However, we can observe from our calculation that the multicast transfer efficiency is not optimal though the clients were running in parallel. It can be concluded that the multicast does not take complete advantage of concurrency among clients. To further understand why the multicast-based system is not able to take advantage of concurrency, let us begin by looking at things that contribute to client runtime. Client runtime is composed of two parts, as shown in Figure 3.3.

Image transfer time is the sum of the following:

- Network transfer time
- Server file read time

- Frisbee protocol overhead (Request-Reply mechanism)
- Multicast protocol overhead (Start delay)

The design of the multicast protocols restricts the Frisbee server to take advantage of concurrency only in the image transfer phase of the client run time. As illustrated in Figure 3.3 and Figure 3.4, the Frisbee server must completely resend the image if two Frisbee clients arrive in the following manner.

Client 2 arrives after the image transfer of client 1 is complete, and when client 1 is writing the image to disk. Assuming both clients are connected to the same switch, client 2 requests image from Frisbee server, though the same image is completely available with client 1 on the same switch. This issue serves as the motivation to improve the performance of the Frisbee system by taking advantage of the complete client concurrency time.

3.1 Concurrency Among Past Frisbee Clients

In the above section, we have established the shortcomings present in the existing Frisbee system. In this section, we analyze the past log data to establish the amount of concurrency observed among Frisbee clients.

Figure 3.5 shows that out of the 18998 clients analyzed, 12737 clients have exhibited 90 to 100 % concurrency. This serves as evidence that Frisbee clients are highly concurrent. Considering the multicast's limitation to take advantage of only the image transfer time concurrency, data from Figure 3.5 further cements the argument for the need for a better transfer mechanism. The high number of nonconcurrent clients is due to the single node clients that are often seen in Cloudlab.

In Figure 3.6, we can see the multicast efficiency of clients from the past. Based on our analysis, servers that have single clients are mostly contributors to complete multicast efficiency. SWP2P will focus on reducing the number of servers having an efficiency of less than 1.0.

3.2 Retransmission Analysis

In the above section, we have concluded that there is a high amount of concurrency among Frisbee clients, despite that there is a high amount of retransmits, and thus low transfer efficiency. In this section, we study the reason behind retransmission.

In the Frisbee system, an image is divided as 1MB Chunks, and each chunk is further divided into 1KB Blocks. **Full Chunk Requests** are sent by client when it does not have any blocks of this chunk. This message is sent only one time per chunk. **Partial Chunk Requests** are sent by client when either some blocks of a chunk are missing, or the whole chunk that was requested earlier has not arrived yet.

Current Frisbee logs have information about partial requests and full chunk re-requests. So, complete chunk requests are sent only when the client did not see anyone requesting the same chunk, and it has not requested the same chunk earlier. The traffic loss on the client-side or network results in Partial Chunk Requests.

Figure 3.7 represents the split between servers with one client and more than one client. Figure 3.8 represents the split between servers with clients that have not missed any chunk fully and the ones that had clients that missed at least one chunk fully. The split in the Figure 3.8 resembles closely to the Figure 3.7; this gives us the impression that single client servers did not have any client that lost a whole chunk.

We also analyzed the average percentage of blocks missed that was due to the entire chunk being missed by the clients. Results are as follows.

From our analysis, it is also seen in Table 3.1 that 96% of the blocks that were repeated were due to clients not receiving the whole chunk. The same is illustrated in Figure 3.9. Since we have already established that the overall runtime concurrency is significantly high, increasing full chunk miss when there are many clients supports the assumption that the transfer time overlap among clients is minimal. This supports our plan to design a system that makes use of the entire overlap in client execution and not just the time taken to transfer the image via multicast.

```
Server Id                  : 55626
Serving Image              : /usr/testbed/images/UBUNTU16-64-
                             X86/UBUNTU16-64-X86.ndz:20
Total clients served       : 4
File read time             : 0.48
File size in MB            : 709.0
File repeated reads        : 775946240 Bytes
Total blocks on image      : 726016
Total blocks on multicast  : 1483776
Saving using multicast     : 1420288
    Client ID        : 1657444948
    Runtime          : 93.72
    Concurrency      : 100%
    Client ID        : 2058447516
    Runtime          : 93.705
    Concurrency      : 100%
    Client ID        : 1590336582
    Runtime          : 58.478
    Concurrency      : 99.18%
    Client ID        : 374246217
    Runtime          : 36.353
    Concurrency      : 99.02%
```

Figure 3.1 Extracted information from Frisbee Logs for server 55626

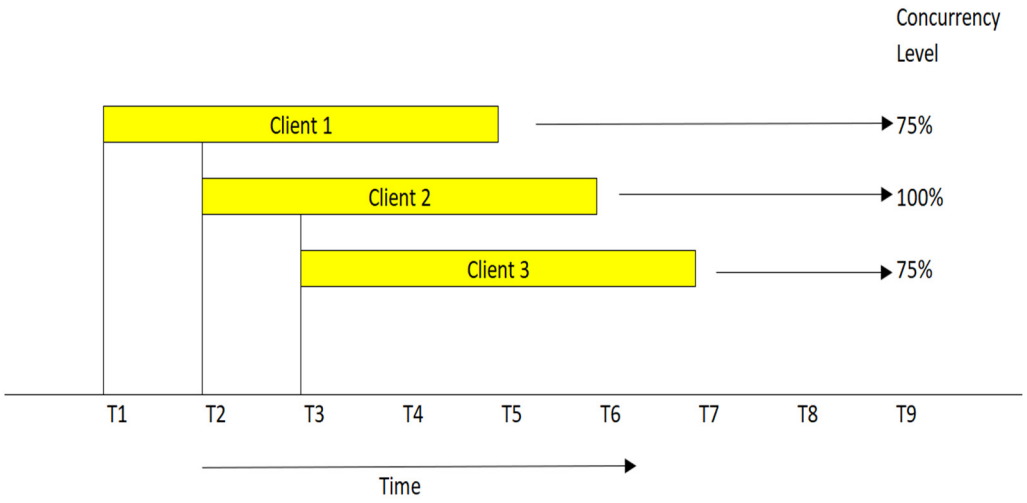

Figure 3.2 Concurrency measurement example

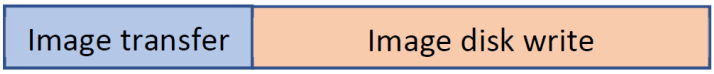

Figure 3.3 Split of client runtime

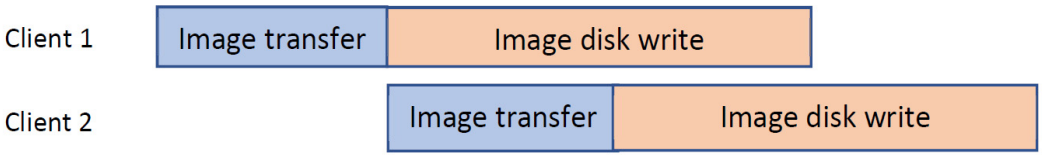

Figure 3.4 Concurrency example

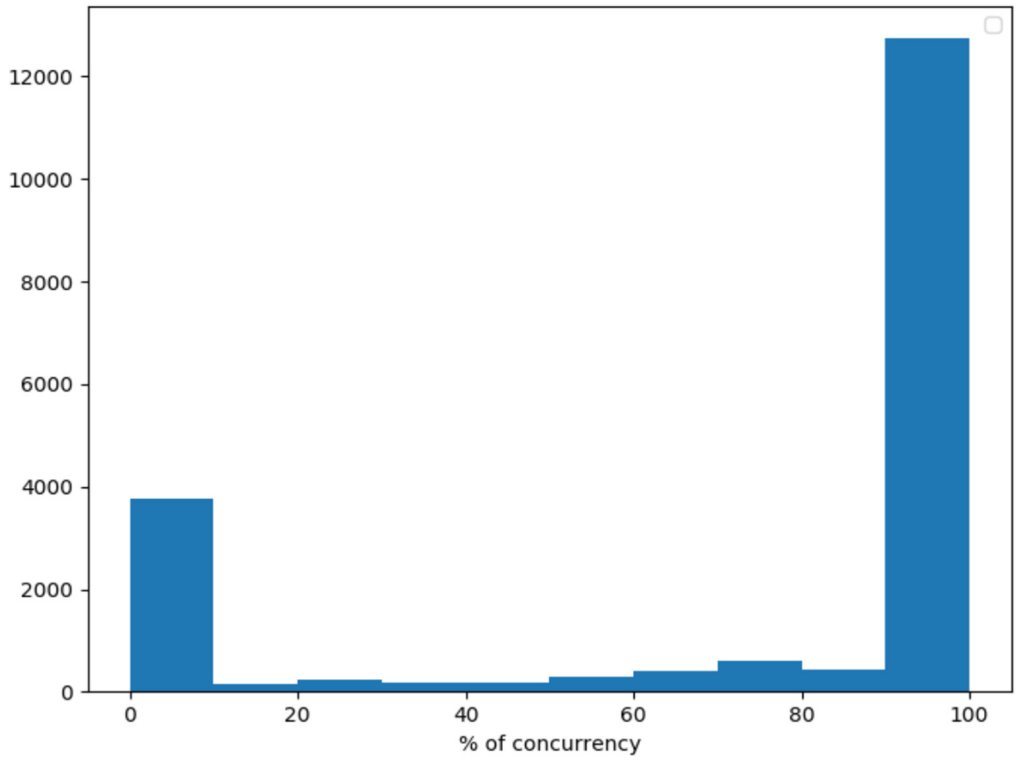

Figure 3.5 Histogram of concurrency

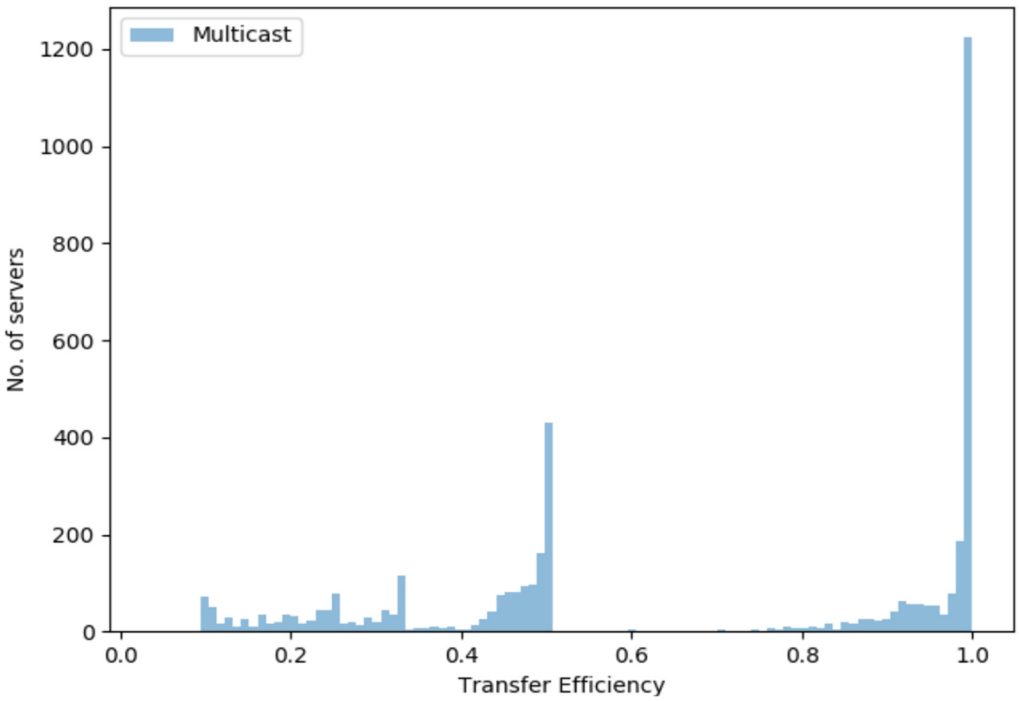

Figure 3.6 Multicast transfer efficiency histogram

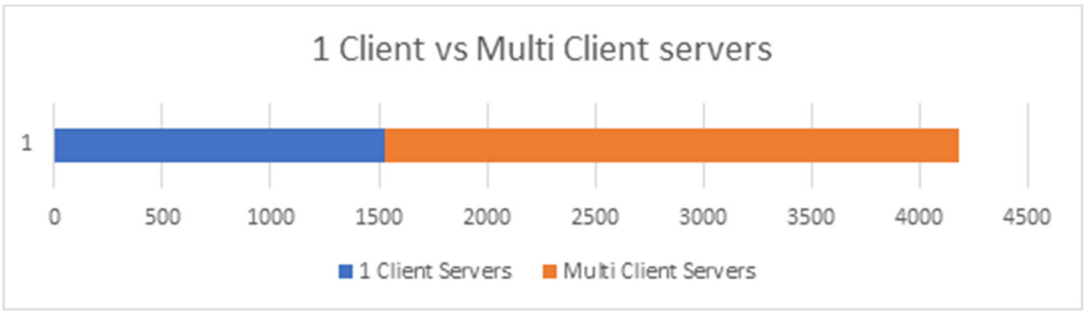

Figure 3.7 Ratio of 1 client versus multiclient experiments in Cloudlab

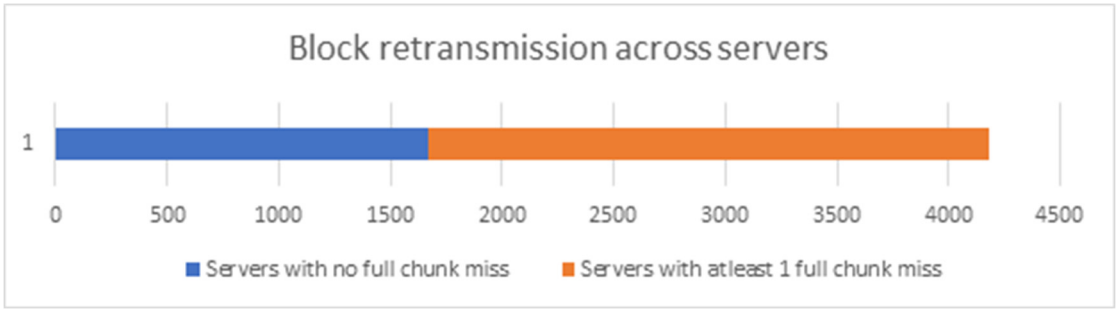

Figure 3.8 Ratio of servers with and without loss

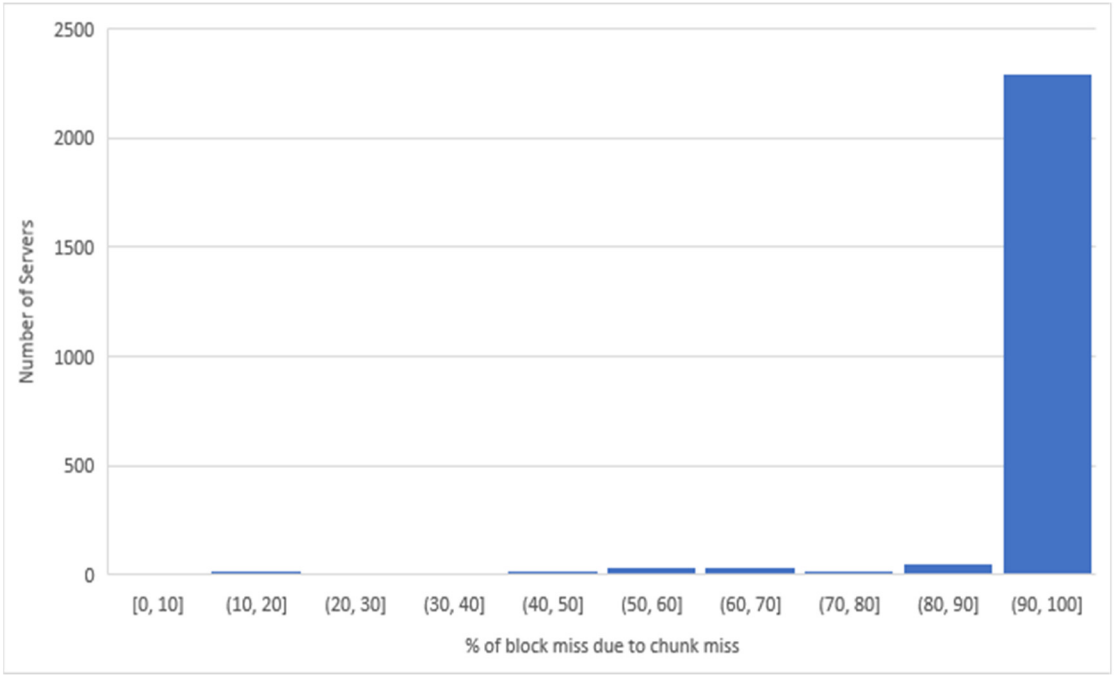

Figure 3.9 Histogram analyzing the chunk loss

Table 3.1 Analysis of block retransmission due to chunk miss

Server's clients	Average % of blocks retransmitted that can be attributed to full chunk miss
1	0.11
>1	91%

CHAPTER 4

SYSTEM DESIGN AND IMPLEMENTATION

SWP2P transfer works by preferring peers that are connected to the same switch to serving the request instead of the Frisbee server. The key operations in the SWP2P-based Frisbee system are described below:

1. Clients send Join message to the server, indicating the image of interest.
2. The server sends a Join-Reply message providing image information such as the number of chunks and blocks in an image.
3. Clients request chunks they do not have in random order using request message.
4. The request messages are redirected at the switch to appropriate peer or server.
5. On receiving the request, either the peer or server sends the requested chunk.
6. The client receives the requested chunk and sends a report message for every eight complete chunks it receives. In the report message, the client indicates a list of chunks it already has.
7. The switch uses the report message to build a chunk availability database.
8. Once the client finishes writing the image to the disk, it sends a leave message.
9. The leave messages are used by switch to update the chunk availability database.

Once the client receives the chunks, it stores them in memory until it sends the Leave message. As the images are compressed, they typically are of only a few GBs in

size; this makes holding them in memory feasible. Clients are available to serve chunks even after they finish receiving as decompressing, and writing it to disk takes a longer time, as mentioned in the previous chapter. The following example illustrates the working of the SWP2P system.

In Figure 4.1, we can see that client 1 is requesting Image 1's chunk 101. Upon receiving the request, the SWP2P module on switch checks for any entry in the chunk availability database for chunk 101. Since there is no entry, the request is forwarded to the server.

The scenario of two overlapping clients is shown in Figure 4.2. Client 2 requests for chunk 101. On receiving the request, the SWP2P module checks its database for available peers. As client one already received chunk 101, it has reported its availability, so the SWP2P module redirects the request to client 1. On receiving the request, client 1 sends the requested chunk to client 2. In this example, if client 2 had requested for any chunk other than 101, the request would have been forwarded to the server.

The existing Frisbee system uses the multicast protocol, in order to support the peer-to-peer data transfer among clients, and the unicast transfer has been implemented as part of this thesis. In addition to that, a new software module is also implemented in the switch to build a database of chunk availability among clients connected to it. In the following sections, we discuss in detail the changes in each component involved.

4.1 Switch

The user programmable switch DELL S4048-ON is utilized for the implementation. By default, the Dell switch arrives with FTOS Network Operating

System (NOS) installed in it. As the project depends on the programmability and access to open APIs in the NOS, OpenSwitch NOS OPX was selected.

OPX was selected after evaluating it against Open Network Linux's ONL. As OPX is supported by Dell Systems, it provided better compatibility on Dell S4048-ON switch. During the evaluation phase, we identified that OPX supports all our primary requirements listed below.

1. ONIE-based installation for easy operation.
2. ACL API support for installing custom filtering rules.
3. No rate limiting in lifting packets to control plane.

Figure 4.3 explains the OPX architecture [9] and the new SWP2P switch module's interactions.

The SWP2P module built on python3.5 interacts with the OPX using Control Plane Services APIs. The core responsibilities of the SWP2P module are listed below:

- Receiving all the Frisbee control packets from the clients.
- Processing the report and leave messages from clients to build a chunk availability database.
- Processing the request messages from clients and identifying the appropriate peer/server for the request.
- Redirecting the request to the peer/server.

4.1.1 Receiving Frisbee Control Packets

The Frisbee system uses various messages between clients and server to communicate. SWP2P module facilitates transferring these messages from the data-plane

to the control-plane of the switch by installing ACL rules. We will go through the ACL rules in detail shortly.

The messages that are supported by the Frisbee system are shown in Table 4.1 To support the transfer of all the Frisbee Control messages from the data-plane to control-plane, we create a dummy Ethernet interface with an IP address on the switch. All the clients are configured to send their messages to the switch's dummy ethernet IP address. The SWP2P module installs an ACL rule to uplift all the messages addressed to the dummy interface IP address; this is accomplished using the TRAP_TO_CPU option in the CPS ACL API.

ACL rule is needed as the packets, in general, are processed and forwarded by the data plane in the switch, which is primarily an ASIC (Application Specific Integrated Circuit). To make custom forwarding decisions of the request packet, we install ACL rules to lift it to the control plane of the switch, where the SWP2P module resides. The control plane, in general, consists of an x86 processor, Linux kernel, and support for user applications.

SWP2P module opens a socket with all the ethernet interfaces that have clients connected to it. By using the select system call, it then reads the messages coming from various clients. SWP2P module is also configured with the Frisbee server's IP address so that it can forward the messages when required. The Table 4.2 illustrates the response of the SWP2P module for each type of message.

4.1.2 Chunk Availability Database

The Chunk availability database is used to keep track of the availability of different chunks among different clients. It is updated each time a Report message is received about chunks available with a client. A Leave message is used to clear the availability of a client to serve chunks.

Figure 4.4 illustrates the design of the chunk availability database implemented in the SWP2P module residing in the switch. In this example, the database is created for the Ubuntu-16.0 image. The database implemented using python's dictionary has information about the availability of seven chunks. The database entry for the chunk one is expanded. Each entry in the database points to the Chunk Class object. Contents of the Chunk Class are explained in Figure 4.5.

If there has been no report from any client for the availability of the requested chunk, the chunk database entry points to NULL. In such cases, the request is redirected to the Frisbee server configured by the user. If the chunk class object is found in the database, it signifies the availability of clients connected to the switch that can serve these requests. The *serving_peer_list* contains the objects of client Class. Contents of the client Class are explained in Figure 4.6. To avoid overloading a client, the SWP2P module selects them in a round-robin manner from the *serving_peer_list.*

4.1.3 Maintaining the Chunk Availability Database

As explained in the previous section, the chunk availability database serves as the most important resource for redirecting the requests. To maintain the correct information in the database, the SWP2P module residing in switch depends on two types of messages

from the clients connected to it. In this section, we will study in detail about the following two messages and how they are processed at the switch.

1. PKTSUBTYPE_SWP2P_REPORT - referred as SWP2P Report message
2. PKTSUBTYPE_LEAVE - referred as Leave message

SWP2P Report message is a new message introduced for updating the chunk availability in the switch. This message is sent by the client to update the switch about the chunks that it can serve. SWP2P Report message contains the fields described in Figure 4.7.

The SWP2P Report message is filled with a bitmap where the respective bit is set as one if the chunk is available in the client. By setting this bit as 1, the client indicates that it is willing to serve this chunk to the peer clients. Upon receipt of this message, the SWP2P module in the switch updates the availability in its database.

A part of maintaining the database is to remove the client's availability when they leave. The client sends the Leave message as an indication that it has finished receiving the image and writing it to the disk. On receiving this message, the SWP2P module on switch removes the client from the *serving_peer_list* for all the chunks of the image.

4.2 Frisbee Client

The Frisbee client has been modified to accept the request from other peers and serve them. As part of this modification, a lot of changes has been introduced to the existing system, while still retaining the functionality of the existing Frisbee client implementation. The following are the main changes to the Frisbee client, and we will be discussing them in detail in the upcoming subsections.

- Transfer changed from multicast to Unicast
- Transport mechanism changed from UDP to TCP
- Receiving redirected request messages from the switch
- Sending requested chunks to the peers
- Updating the switch with report messages
- Moved to deterministic order for requesting chunks
- Request batching

4.2.1 Choosing Unicast Transfer Over Multicast

As we began working on this thesis, we selected unicast transfer over multicast to favor the simplicity of operation in the peer to peer transfer mechanism. We analyzed the existing code and understood that there are various mechanisms in place to take advantage of the multicast approach. Since they will not contribute anything to the unicast-based client, it has been disabled in our implementation. The important changes that were disabled are listed below.

- Request Suppression
- Dubious Chunk Support

Request suppression was achieved in the multicast system by allowing the client to send their chunk request to the multicast group. Thus, all client receiving the request will analyze and avoid rerequesting the same chunk again. In the multicast implementation, this prevented the server from receiving too many request messages at once. However, the request suppression is invalid in the unicast-based transfer system; thus, it has been disabled in the new client implementation. In the new client

implementation, the server's request processing load is reduced by the following three mechanisms.

- Peer to Peer transfer among clients
- Deterministic order of requesting chunks
- Request batching

Each of these three mechanisms is explained in detail in the further pages of this thesis.

4.2.2 Changing Transfer Protocol to TCP

In the UDP-based implementation, the client was responsible for managing the sending window. This was accomplished by encoding constants for burst management in the client code. The burst size and burst interval, regardless of their selection, do not promise lossless delivery. We observed significant packet losses with the default UDP Socket read, write size settings in the Linux kernel, and the default burst configurations on the client. These packet losses were observed in the receiving side, due to the socket buffer size.

To overcome this issue, we switched to using TCP sockets for the image data transmission. TCP provides an advantage through automatic buffer resizing and lossless delivery on the transport layer. Though we decided to utilize the TCP sockets for the data transfer, the UDP sockets are retained for the Frisbee protocol control packet transmission.

Frisbee control packets include the request messages. Redirecting the request packets at the switch is not possible if these are TCP messages as they are part of the persistent session. Hence the UDP sockets are still retained. In practice, the control

packets are lost rarely; when lost, the client recovers by rerequesting the same chunk after the timeout. In the Figure 4.8, we represent in detail the various sockets used for communication among different components in the SWP2P system. In the Table 4.3, various connections and their purpose are shown.

Frisbee clients run a TCP server accepting incoming connections on TCP port 54321. This TCP socket is used to sending and receiving the data packets/Frisbee BLOCK type packets from server and peer clients. When used for communication between the client and server, the client waits for the server to initiate the communication. The server initiates the communication upon receiving the JOIN message from the client. When used for communication between the clients, the client that intends to send the message initiates the connection with the TCP server present in another client.

To segregate the peer-to-peer functionality from existing implementation, the redirected control messages (request and partial request) are sent to a new UDP port (54333) to the clients from the switch. This is accomplished by changing the UDP destination port in the request messages when they are redirected at the switch. On the client side, we run separate threads for processing messages from peers and the server. This design allows measuring the time taken by various entities in a discrete manner.

4.2.3 Processing Request Messages From Peers

Peer to Peer functionality is implemented in the clients by adding new threads for processing the requests from other clients. In this section, we will cover in detail the request queuing mechanism implemented in the clients; the various threads are illustrated

in Figure 4.9. The four threads are as follows:

1. Client Recv Thread (Modified).
2. Chunker Thread (Modified).
3. Request Recv Thread (New).
4. SWP2P Worker Thread (New).
5. Per Peer TCP Send Thread (New).

Client Recv Thread receives messages from the server and peers. It processes the incoming data in BLOCK messages. This thread is also responsible for updating the switch with the SWP2P_REPORT message. After completion of receiving each chunk of the image, the chunk is released from this thread to the Chunker Thread.

Chunker Thread is responsible for retrieving the completed chunks from the ChunkBuffer and writing them to the disk. Chunker Thread handles these data to the decompression thread, and once decompression is complete, it then writes the chunk to the disk. In the earlier Frisbee implementation, the chunks were discarded after they were written to the disk. Now we retain the written chunks also in the ChunkBuffer to serve requests from peers.

Request Recv Thread receives the requests from the peers and adds them to the Peer Request Queue. This thread parses the incoming request messages and adds the following information to the queue.

1. Requesting client's IP address
2. Type of request (Full/Partial)
3. Requested Chunk
4. Requested Block Bitmap

SWP2P Worker Thread processes each request from the Peer Request Queue. It retrieves the chunks from the Chunk Buffer and forms the Frisbee Response Packet. These packets are then queued to the Per Peer Queue. Each peer has its own TCP send queue in the client system; all the image blocks addressed to a client are added to its Per Peer Queue. Since requests can come in only for the chunks that are already complete, the Chunk Buffer is accessed without locks from the SWP2P worker thread to avoid delays.

Per Peer TCP Send Thread deques the Per Peer Queue and sends the message to the peer. The delays caused by the TCP send are faced only by this thread, thus isolating all other threads from such delays.

4.2.4 Changes to Request Mechanism

The image chunks were requested by clients in a random fashion in the existing implementation. In our analysis, we understood that requesting chunks in a dedicated order improves the efficiency of the SWP2P transfer system. In this section, we will be exploring that in detail.

Our idea is to allocate each client a dedicated range of chunks, which it should receive first. Let N be the total number of chunks in an image, and C be the number of clients in the experiment. We introduce another parameter Ci, the sequence number of the client in the total experiment. Let us take the following example, where N=1000, C=4. The first node in the experiment has Ci=1; the second node has Ci=2, and so on. Figure 4.10 shows the request order.

Requesting chunks in the order mentioned in Figure 4.10 creates a cascading

effect when clients are running in parallel. If they are not running in parallel, then this request mechanism does not cause any harm either.

Requesting chunks from clients in the specified order guarantees that the first *N* (1000) chunks served by the server are unique, constituting a complete copy of the image.

In addition to that, it also establishes approximate ownership of who is responsible for serving the chunk. The switch is still responsible for keeping track of chunk availability, but the approximate ownership allows clients to report chunks in a predefined order. This effectively acts as a hint mechanism to improve the efficiency of peer-to-peer serving. In the example on Figure 4.11, client 4 receives the 751-1000 chunks initially, and thus when the client 3 requests for the same chunks, client 4 can serve them. However, a key thing to observe is, because of the round robin-based peer selection in the switch, when the client 2 requests for the same chunks, Client 3 is chosen to serve instead of the client 4. The same algorithm for chunk request ordering is applied regardless of the number of chunks in the image or the number of clients starting in parallel.

The chunk request order is derived from the following simple algorithm.

```
int N = Total Number of Chunks in Image
int C = Total Number of clients in the Experiment
int Ci = Node number in the experiment


int window_size = N/C;
int copy_start = (Ci - 1) * window_size;
int idx = 0;
```

```
for (i = copy_start; i < N; i++)
{
  ChunkRequestList[idx] = i;
  idx ++;
}
for (i = 0; i < copy_start; i++)
{
  ChunkRequestList[idx] = i;
  idx ++;
}
```

To implement this algorithm, we added new command-line arguments to the Frisbee client to obtain the total number of nodes in the experiment and the current node number.

In addition to the above changes, we have also added a request batching feature to the existing Frisbee system. In the existing implementation, for each chunk, an explicit request was sent out. This mechanism was favorable earlier as the chunks were requested in random order. As each request needs to be processed by the switch to redirect them appropriately, the existing request mechanism added notable stress to the SWP2P module running in the Dell switch.

Since we have changed the Frisbee to request chunks in a sequential manner, we have added support in the Frisbee client, server, and SWP2P switch to support requesting *K* chunks at a time. The value of *K* can be set by a compile-time variable in all three components of the SWP2P system. In our experiments containing 32 nodes, the *K* value

has been set as 5, meaning only one request is sent for five chunks. When request batching is enabled, the number of request messages sent by the clients is lower; thus, the requests that must be processed by the switch also is reduced, thus having lower processing load on the switch.

4.3 Frisbee Server

Frisbee server has the least changes among the three components of the SWP2P system. The server process has the following modifications:

1. Transfer method changed from multicast to Unicast.
2. Transfer protocol changed from UDP to TCP.
3. Batching the incoming requests.

To accommodate the unicast requests, the servers in the work queue, now support remembering the clients that sent the request for the chunk. As all these changes have been explained in Section 4.2, we are going through the details in the subtopic.

4.4 Design and Implementation Summary

SWP2P system is implemented by making changes at the Frisbee server, client, and the switch module. Our primary design contributions are listed below:

- Modifying existing Frisbee system to support Unicast transfer.
- Installation of ACL rules in switch for lifting request packets.
- Maintenance of the chunk availability database at the switch.
- Redirecting request packets at the switch.
- Adding the serving capability to clients.

- Efficient multithreaded architecture on clients for low latency service.
- New sequential request mechanism coupled with request batching.
- Change of transfer protocol of blocks from UDP to TCP.

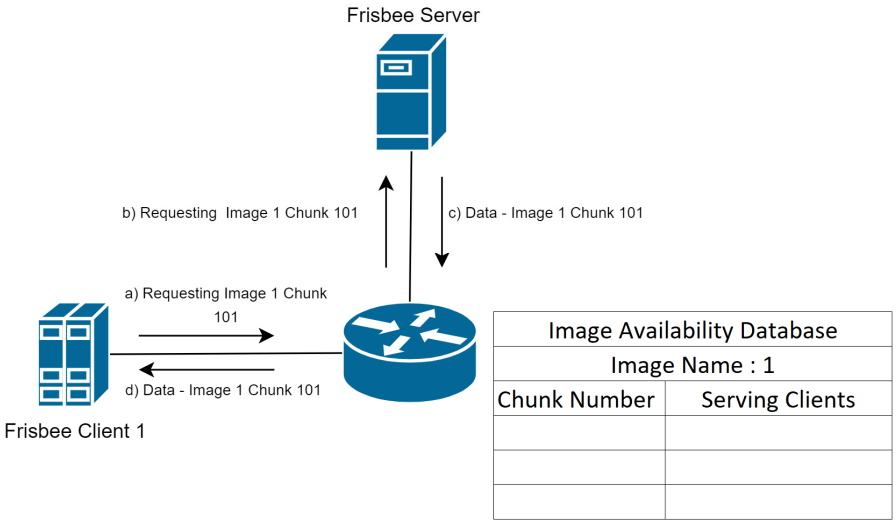

Figure 4.1 Example SWP2P transfer for one client

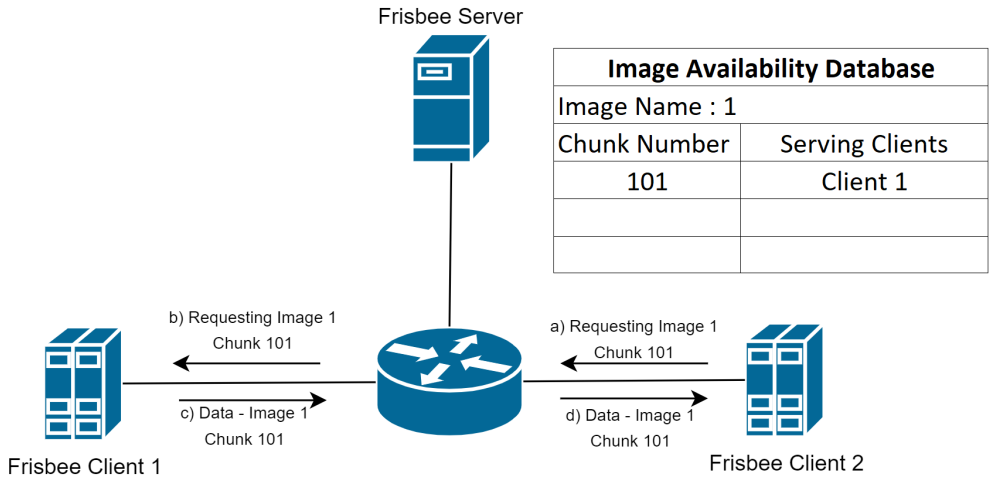

Figure 4.2 Example SWP2P transfer when there are two clients running in parallel

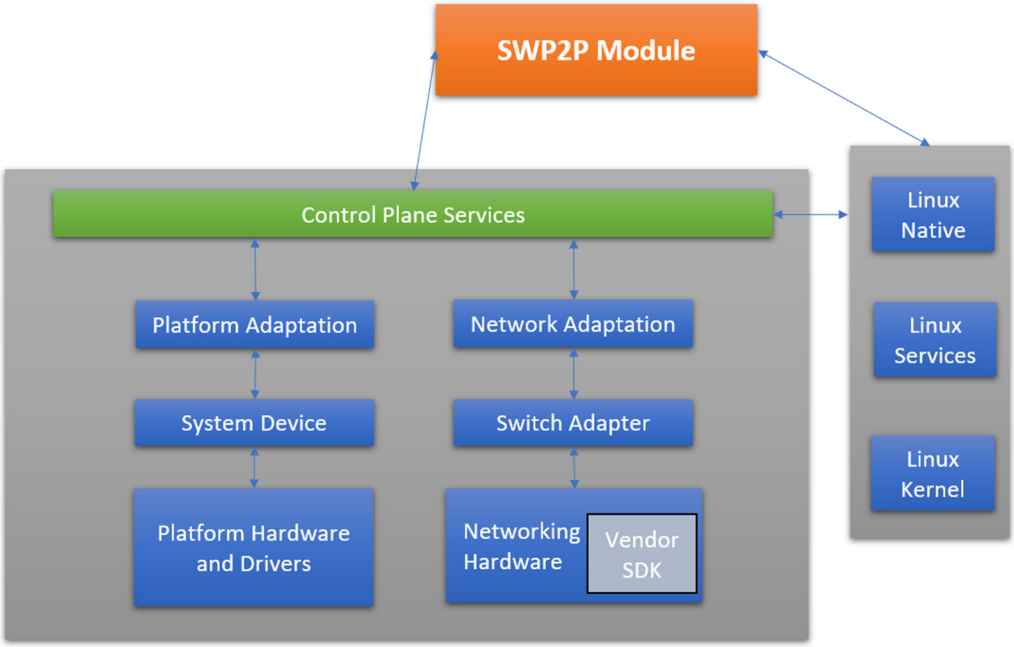

Figure 4.3 OPX Architecture and SWP2P module

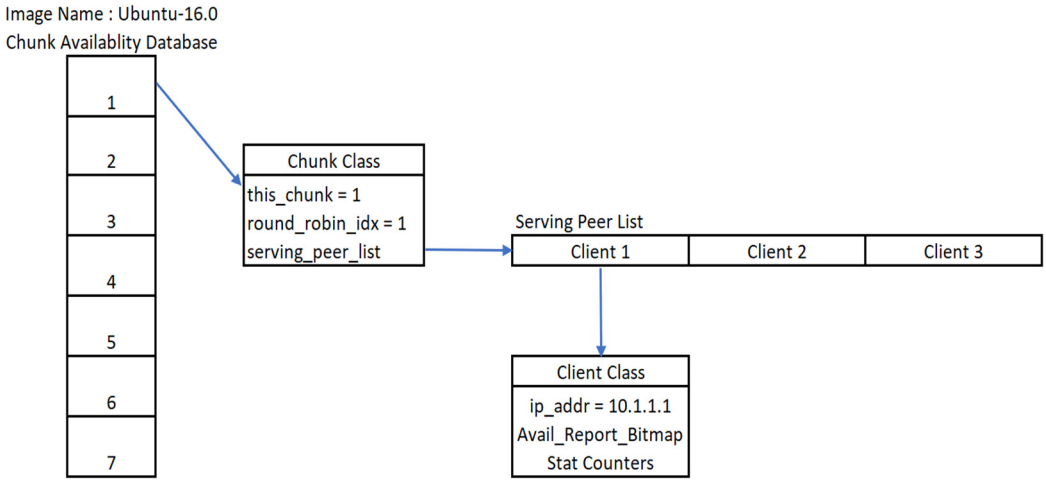

Figure 4.4 Chunk availability database design

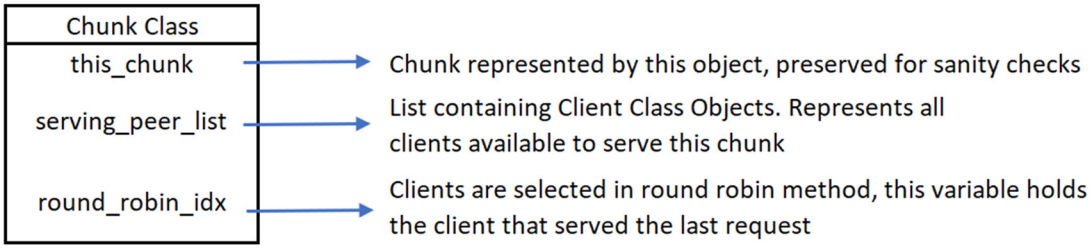

Figure 4.5 Chunk class object

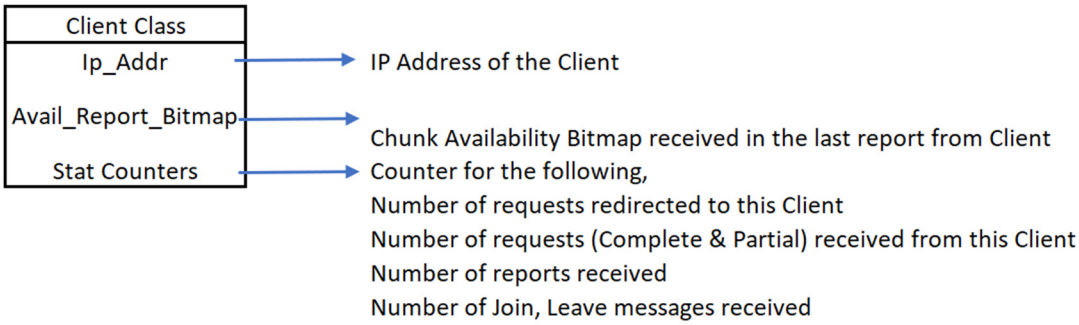

Figure 4.6 Client class object

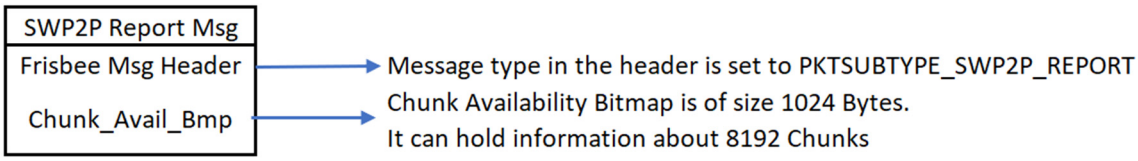

Figure 4.7 SWP2P Report message

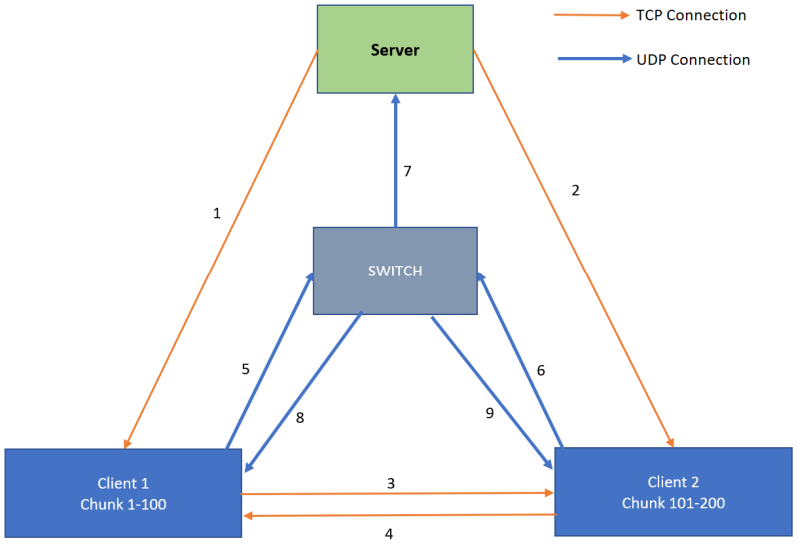

Figure 4.8 TCP and UDP Socket usage diagram

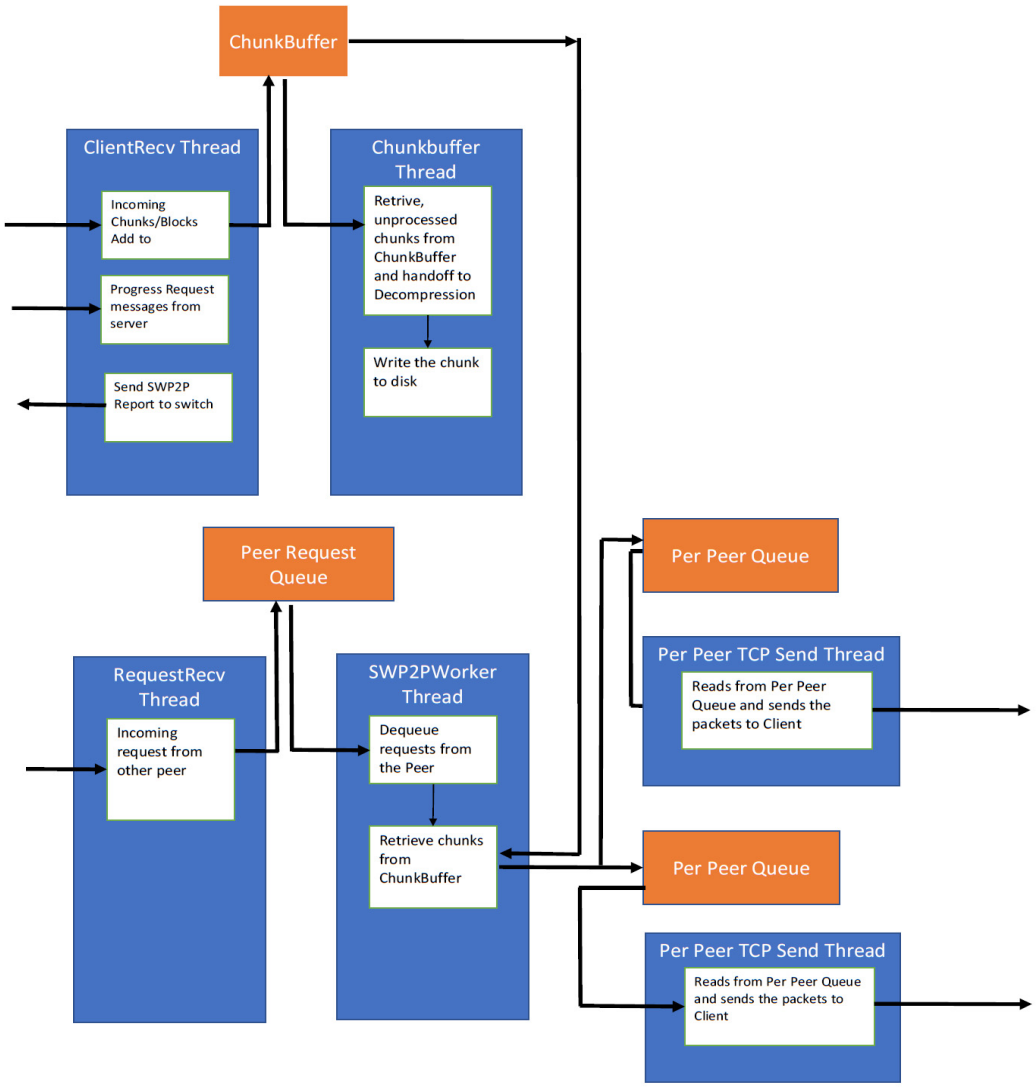

Figure 4.9 Multiple threads in the Client process

Node Number	Ci	Request Order
1	1	1-250, 251-1000
2	2	251-1000, 1-250
3	3	501-1000, 1-500
4	4	751-1000, 1-750

Figure 4.10 Chunk request order

Time	Client 1	Client 2	Client 3	Client 4
T1 (0 to 2 seconds)	1-250	251-500	501-750	751-1000
T2 (2 to 4 seconds)	251-500	501-750	751-1000	1-250
T3 (4 to 6 seconds)	501-750	751-1000	1-250	251-500
T4 (6 to 8 seconds)	751-1000	1-250	251-500	501-750

Figure 4.11 Chunk request order cascading effect

Table 4.1 Type of messages and their frequency

Message Type	**Description**	**Frequency per session**
PKTSUBTYPE_JOIN	Sent by clients to request image information from the server	Once
PKTSUBTYPE_LEAVE	Sent by clients when they have finished decompressing and writing the image to the disk	Once
PKTSUBTYPE_BLOCK	Sent by server or peers with blocks of image chunk	1024*Number of Chunks
PKTSUBTYPE_REQUEST	Sent by clients to request a whole chunk	1 * Number of Chunks
PKTSUBTYPE_PREQUEST	Sent by clients to re-request partial chunks	Based on loss in the network or receiving side
PKTSUBTYPE_PROGRESS	Sent by clients to server for centralized logging	Based on requests from server
PKTSUBTYPE_SWP2P_REPORT	Sent by clients to switch for updating chunks that are available with it	Once for every 8 chunks

Table 4.2 Message type versus action

Message Type	**SWP2P Module Action**	**Number of messages N = Image Size**
PKTSUBTYPE_JOIN	Forwarded to the server	O(1)
PKTSUBTYPE_LEAVE	Chunk availability database is updated as the client is no longer available to serve chunks	O(1)
PKTSUBTYPE_BLOCK	This message is not lifted	O(N)
PKTSUBTYPE_REQUEST	Redirected to another client/server based on availability in the database	O(N)
PKTSUBTYPE_PREQUEST	Redirected to another client/server based on availability in the database	O(N)
PKTSUBTYPE_PROGRESS	Forwarded to the server	O(N)
PKTSUBTYPE_SWP2P_REPORT	Used to update the chunk availability database for the client	O(N)

Table 4.3 Connection and their purpose

Connection Numbers	**Connection Purpose**
1,2,3,4	PKTSUBTYPE_BLOCK messages containing the image data to clients
5,6,7	PKTSUBTYPE_JOIN to server PKTSUBTYPE_REQUEST to switch PKTSUBTYPE_PREQUEST to switch PKTSUBTYPE_LEAVE to switch and server PKTSUBTYPE_PROGRESS to server PKTSUBTYPE_SWP2P_REPORT to switch
8,9	PKTSUBTYPE_JOIN Reply from server PKTSUBTYPE_REQUEST Redirected at switch to peer client PKTSUBTYPE_PREQUEST Redirected at switch to peer client PKTSUBTYPE_PROGRESS Request from server

CHAPTER 5

EVALUATION

We evaluate the SWP2P transfer by comparing it with the existing multicast-based Frisbee implementation. In addition to that, we also study in detail the effects of various design choices that were made during our implementation of SWP2P. The following are the core things that we will compare between multicast and SWP2P implementations:

1. Server load.
2. Load on trunk links.
3. Network Receive time in clients.

The server load is defined as the number of chunks sent out by the server to address the requests from the clients. As the server addresses more requests from the clients, it tries to merge requests if it already has a pending request in the queue. However, if there are no pending requests, then the server reads the file from disk to retrieve the requested chunks. This operation costs disk reads, server's CPU cycles, and network resources.

The trunk links are defined as the links that connect one switch to another in Cloudlab. Trunk links are a resource to be preserved as they carry traffic for multiple clients. By sending a smaller number of chunks from the server to the clients, we can

avoid using trunk links for carrying the chunks. SWP2P transfer aims to achieve this by preferring peers that are connected to the same switch, due to this the requests sent out from the switch is reduced, thus leading to less traffic over trunk links.

In addition to the above parameters, we will also discuss in detail the effects of processing packets in the switch's control plane. The data plane is made of ASIC that is capable of handling and transferring packets at the line rate (10Gbps). However, the control plane is in general of low processing capability. In our experiments, we used the Dell S4048-ON switch, which uses Intel(R) Atom(TM) CPU @ 1.74GHz, and it supports only two threads for parallel execution. Due to the low processing power, when a large number of clients start concurrently, they suffer from delays in request rerouting. We explore alternate ideas on how to overcome such delays in SWP2P implementation in the following sections.

5.1 Client Runtime Components

Client runtime represents the total lifetime of the Frisbee clients process. Frisbee clients are responsible for receiving, decompressing, and writing the image to the disk.

Figure 5.1 shows two components contributing to the client runtime:

1. Network Receive Time.
2. Decompress, and disk write time.

The data represented are based on the runtimes, when one client receives an image from the server. Figure 5.1 also shows how these components change when the image is of different sizes and has a different compression ratio. A common factor in Figure 5.1 is that the network receive time is significantly shorter when compared to the

time taken by the Frisbee client to decompress and write the image to the disk. In the upcoming sections, we show how SWP2P implementations can take advantage of the longer decompression and disk write operations.

5.2 Start Delay

The start delay is the time difference in the start of two clients. In this thesis, we refer to the start delay as the time delay between the start of two consecutive clients. Figure 5.2 shows how clients start when the start delay parameter is set as 1 second.

In this thesis, we use start delay as a parameter to simulate various degrees of concurrency among clients, and we then study the effect of such concurrency on various other parameters like client runtime and chunks sent by server.

5.3 Experiment Setup

Our experiment setup is based on the Cloudlab profile, which allocates 32 nodes and one user-programmable switch. Switch details are already provided at the start of this chapter. All 32 nodes are identical and have the configuration mentioned in Table 5.1. All the nodes are connected directly to the switch. Among the 32 nodes, the first node is selected as the Frisbee server, and this node hosts the Frisbee server process.

5.4 Effect of Start Delay on Server Load

SWP2P is designed to take advantage of the concurrency in the overall client runtime. Multicast lacks this advantage as, by design, it can only leverage the concurrency in the network receive time of the client's total runtime.

The data in Figure 5.3 are collected by running 30 clients with varying degrees of start delay among them. The image used for this experiment consists of 730 Chunks, which accounts for 730 MB. In this experiment, each client receives a 730 MB image, decompresses it, and writes it to the SSD disk.

In order to achieve our goal of reduced load on server and trunk links, the chunks sent out by the server have to be close to 730, as that indicates that the server had to send out only one copy of the image. The average runtime of a client when only one client is receiving the data from the server is around 10 to 11 Seconds. This time includes the Frisbee protocol delay, time taken by the client to receive the image, time taken for decompressing, and writing the image to the disk.

5.4.1 Analysis for Start Delay Between 1 to 9 Seconds

From Figure 5.3, it can be inferred that the SWP2P performed as well as the ideal scenario when the start delay is between 1 to 9 seconds. This can be explained by the components contributing to the client runtime shown in Table 5.2.

Even if the start delay is 9 seconds, the SWP2P implementation can take advantage of the client that is running in parallel to it. This is illustrated in Figure 5.4.

As the start delay increases, the client runtime overlaps, or the concurrency occurs when client 1 is in component 2 of its runtime. In such a scenario, with SWP2P transfer, client 1 can process requests from client 2, in turn avoiding sending the request to the server.

It can also be inferred that the multicast transfer-based Frisbee system performs incrementally worse as the start delay increases, this can also be attributed to the overlap occurring in component 2 of its runtime.

5.4.2 Analysis for Zero Second Start Delay

The start delay of zero seconds implies that the clients were started in parallel. This is the ideal scenario for both SWP2P and multicast transfer. However, from Figure 5.5, we can observe that both the transfer did not meet the ideal limits.

The multicast transfer did not achieve the ideal limits due to subsecond delays in multicast join mechanism and other Frisbee protocol overhead. SWP2P transfer did not achieve the ideal limits due to minor time mismatches that occur between the reports and the requests. These mismatches occur, when a client requests for a piece of the image, before the client responsible for that piece reports its availability to the switch. In our analysis, we found the time difference between such reports and requests in the subsecond range.

5.4.3 Analysis for Start Delay Greater than 12 Seconds

When the start delay is greater than 12 seconds, both the SWP2P and multicast transfer performs equally poor. This is because there is no overlap in the client runtime. As there is no overlap, there is no client that can be leveraged for performance by SWP2P. The same is illustrated in Figure 5.6.

It should be noted that the maximum start delay until which SWP2P can leverage concurrency, depends on the size of the image. Thus, for a larger image, the start delay can be greater than 12 seconds, and SWP2P can still perform better than multicast.

5.5 Effect of Processing Time on the Switch

As we mentioned at the start of the chapter, the switch has limited processing ability. The following are the major time-consuming tasks in the switch:

1. Report messages – Updating the image chunk database by adding new client availability.
2. Request messages – Redirecting the requests to the appropriate client or server.
3. Leave messages – Updating the image chunk database by removing the client availability.

The processing time represented in Figure 5.7 is the sum of all the three components listed above. The processing time increases linearly with the number of clients actively running in the experiment. This posed a serious threat when the request batching feature was not in place.

Without request batching, the clients needed to send one request for each chunk. This resulted in the requests being queued at the switch, thus leading to delay in the image delivery. All the results that are discussed in this section are collected with request batching enabled in switch, client, and server.

5.5.1 Effect of Parallel Clients on Runtime

In our experiment containing 30 clients and one server, we observed that as more clients start at the same time, their runtime linearly increases. This is due to the increased processing time at switch, and the request is queued.

In Figure 5.8, the runtime represented is the average runtime of the client. As the multicast-based Frisbee system does not require the switch to make any forwarding decision in the control plane, the average client run time remains consistent across the increasing number of clients starting at the same time.

However, we can observe that the average client runtime increases linearly in the SWP2P implementation. It should also be noted that the top of the rack switches are naturally limited to 48 ports. We scale the experiments to 30 clients and address all the concerns about the linear increase of processing time in client and switch. In the next section, we show how this effect can be mitigated by using a small start delay among clients.

5.6 Using Start Delay to Mitigate Effects of Switch Processing Time

When many clients start at the same time, they all send requests and report messages at the same time. As the switch can only process the requests and report messages in a sequential fashion, the requests received later are queued for processing at the switch. This effectively introduces a delay in receiving the chunks, thus slowing down the whole process. The delay introduced in the network receive component of the client runtime causes no issues if it is less than the time taken by decompression and disk write thread at client.

A non-zero start delay effectively avoids request queuing at the switch. As the clients start with an offset in time, they do not send the request and report messages at the same time. Thus, by using the start delay and the request batching feature, we can mitigate the problems that arise due to lower processing power at the switch's control plane. The results of various start delays are shown in Figure 5.9.

5.7 MCAST Versus SWP2P in Server's Perspective

In this section, we show the results from our experiment, where we simulated a various number of clients attempting to download the image from the server at the same time. In the previous sections, we have illustrated the impact of clients starting at the same time, the effect of it on the switch, and how it can be mitigated. In addition to that, in Section 5.4, we discussed how SWP2P performs better than multicast in the server's perspective. In this section, we further cement that opinion by showing how SWP2P consistently outperforms multicast by having the least load on the server and trunk links.

From Figure 5.10, it can be observed that SWP2P consistently has a lower load on the server node. This results in the server sending fewer packets over the trunk link, thus reducing the usage of trunk links.

We chose to showcase this scenario as multicast has its maximum efficiency when clients start at the same time. However, due to minor start delays introduced by the multicast join mechanism, it still does not perform as good as SWP2P. This is because in the SWP2P-based transfer, even when the clients start with a delay, they leverage their concurrency on overall runtime.

5.8 Effectiveness of New Request Mechanism

As part of the SWP2P implementation, we have made the two following changes to the request mechanism of the Frisbee system:

- Ordered request with dedicated range for each client (Section 4.2.4).
- Request batching (Section 4.2.4).

Both enhancements are introduced to reduce the processing load of the switch. Figure 5.11 illustrates the difference introduced by these features in average client runtime. When the chunks are requested in random order, the request batching feature cannot be implemented at the clients and server. Due to this, the clients need to send a request for each chunk. This significantly increases the number of requests that must be redirected at the switch.

As more requests are sent to switch at the same time, the requests get queued at the switch. This introduces a delay in the chunks being served; thus, the overall runtime of the client increases significantly. The request batching feature, when enabled, allows the clients to send only one request message for requesting 10 chunks.

5.9 Evaluation Summary

In summary, SWP2P outperforms multicast-based Frisbee implementation in all the scenarios we tested. Our SWP2P implementation shows that it is possible to take advantage of the concurrency among clients and thus having a lower load on the server machine and the trunk links.

The only hiccup we faced in the SWP2P implementation was the low processing power at the switch's control plane. In this thesis, we also show how such hiccups can be

successfully overcome using the start delay parameter. From Figure 5.3, it can be observed that there are significant advantages when the start delay is less than the client's runtime. For example, with a 3-second start delay, the SWP2P sends only 730 chunks, whereas the multicast implementation sends 14 times of that.

Thus, we show that SWP2P transfer mechanism significantly reduces the load on the server and trunk links when compared to the multicast-based Frisbee system.

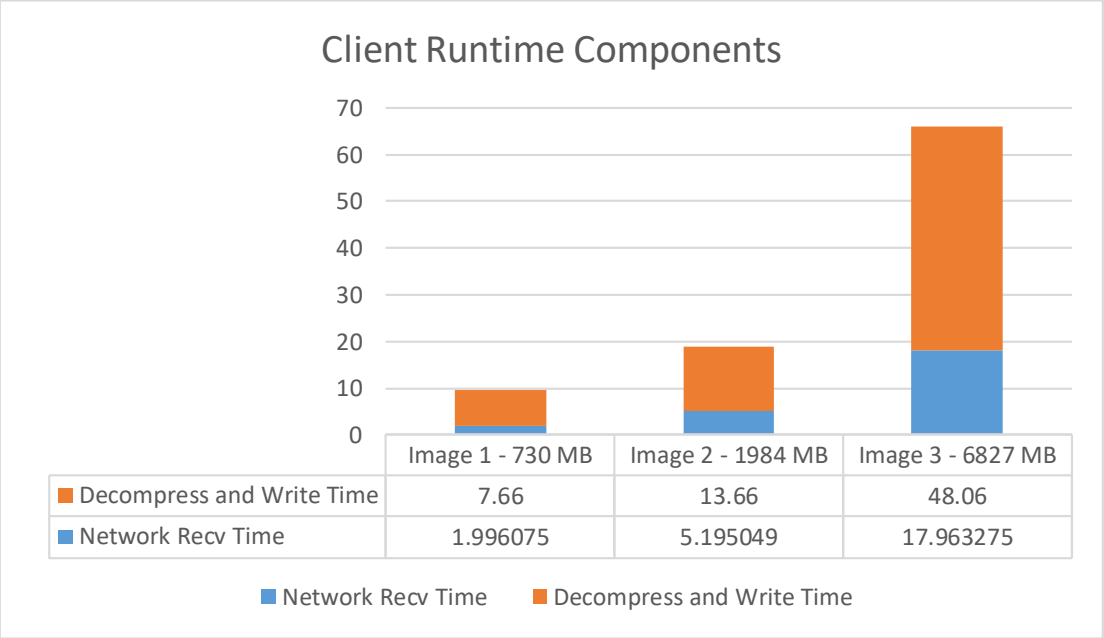

Figure 5.1 Analysis of client runtime

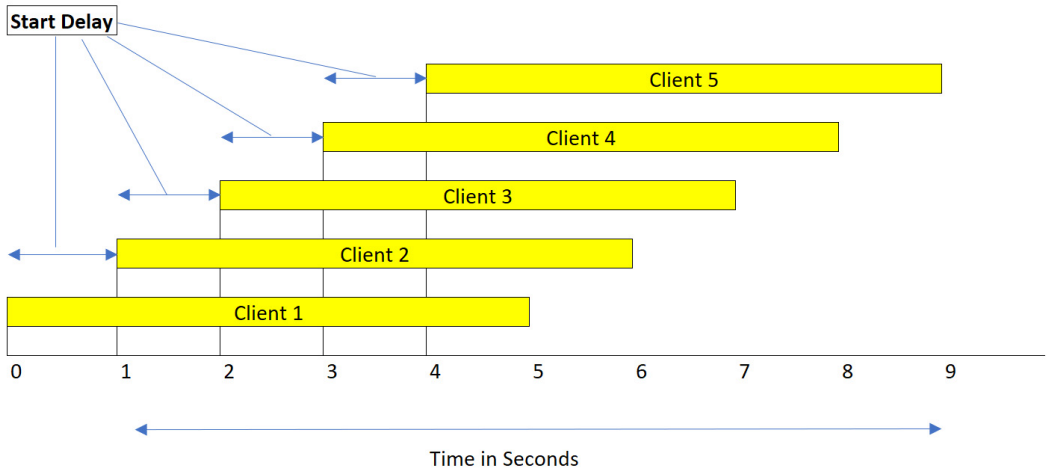

Figure 5.2 One-second start delay among five clients

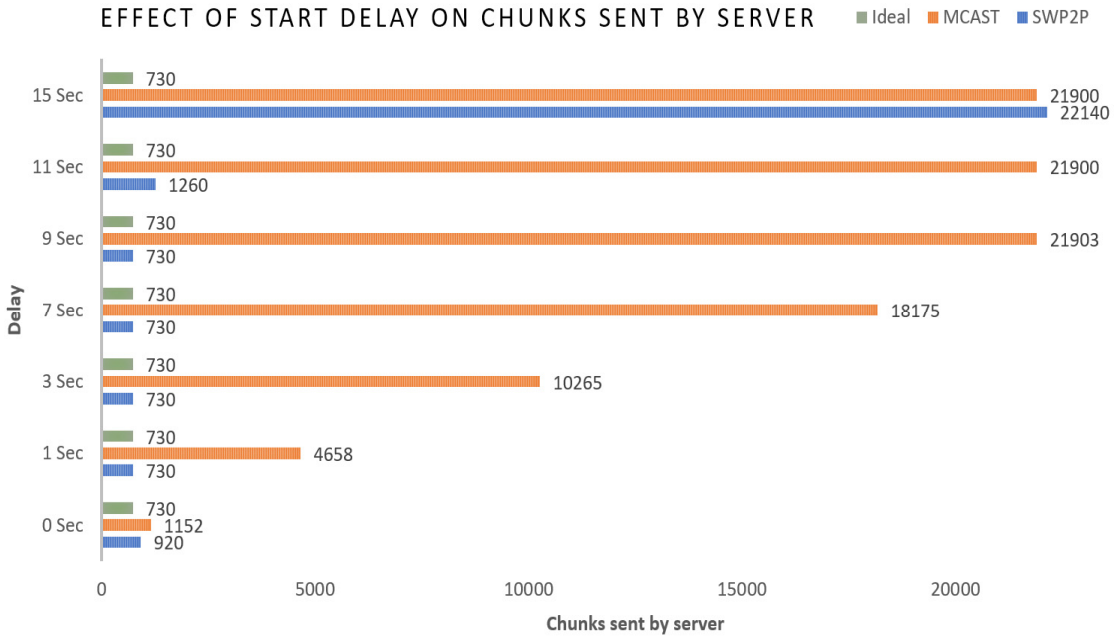

Figure 5.3 Start delay's effect on server load

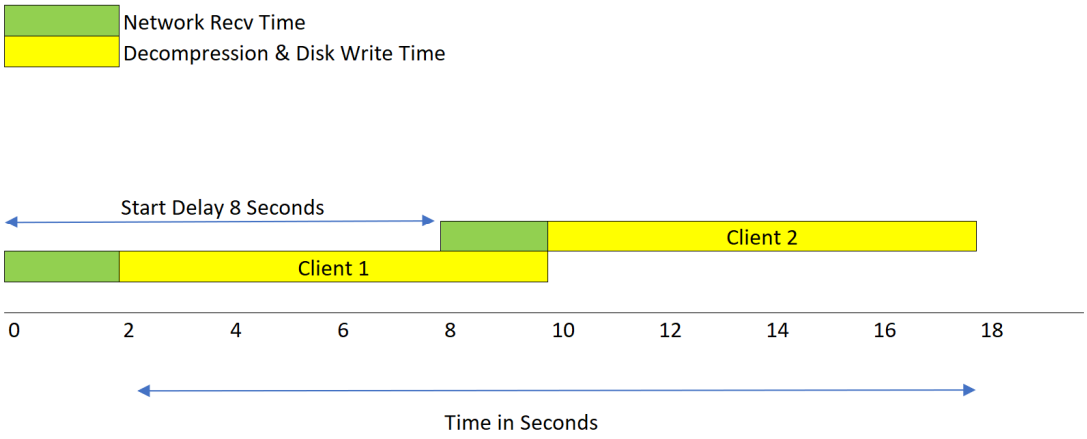

Figure 5.4 Client runtime overlap on component 2

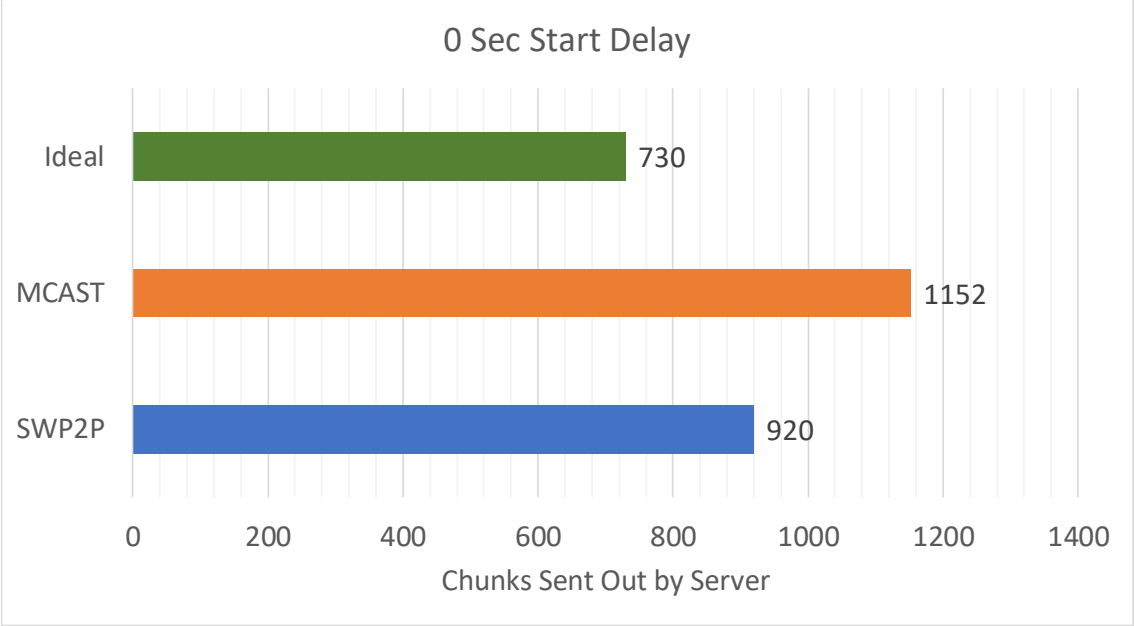

Figure 5.5 Detailed analysis of zero-second start delay

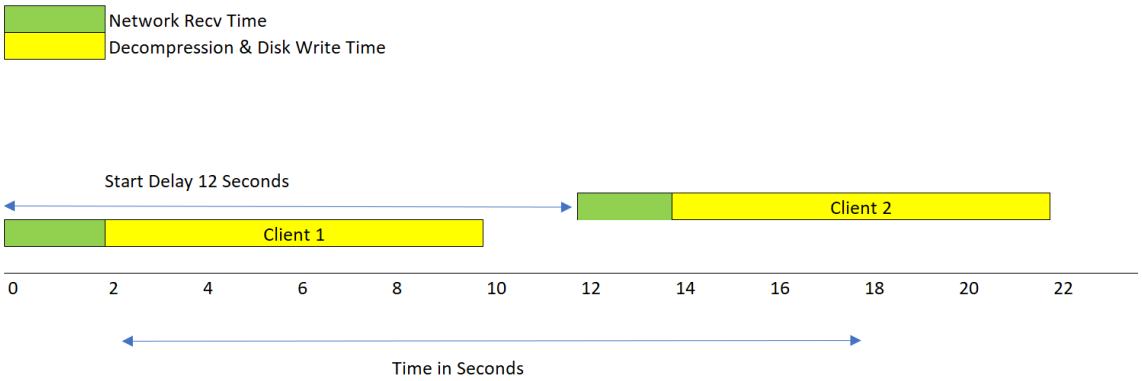

Figure 5.6 No overlap illustration

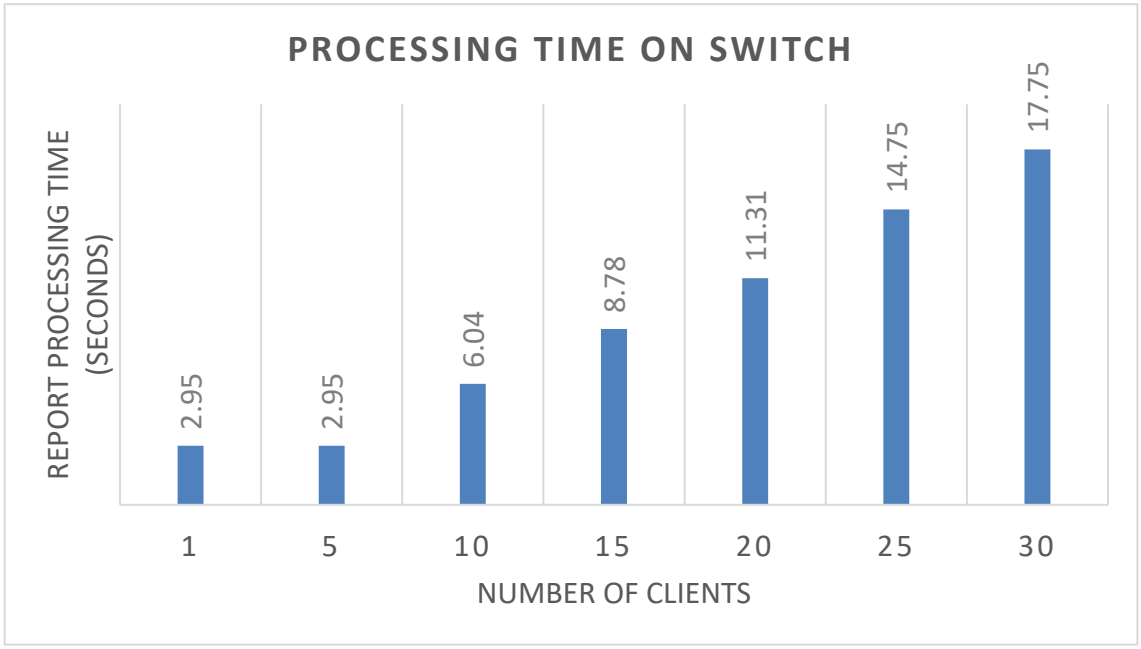

Figure 5.7 Processing time on the switch

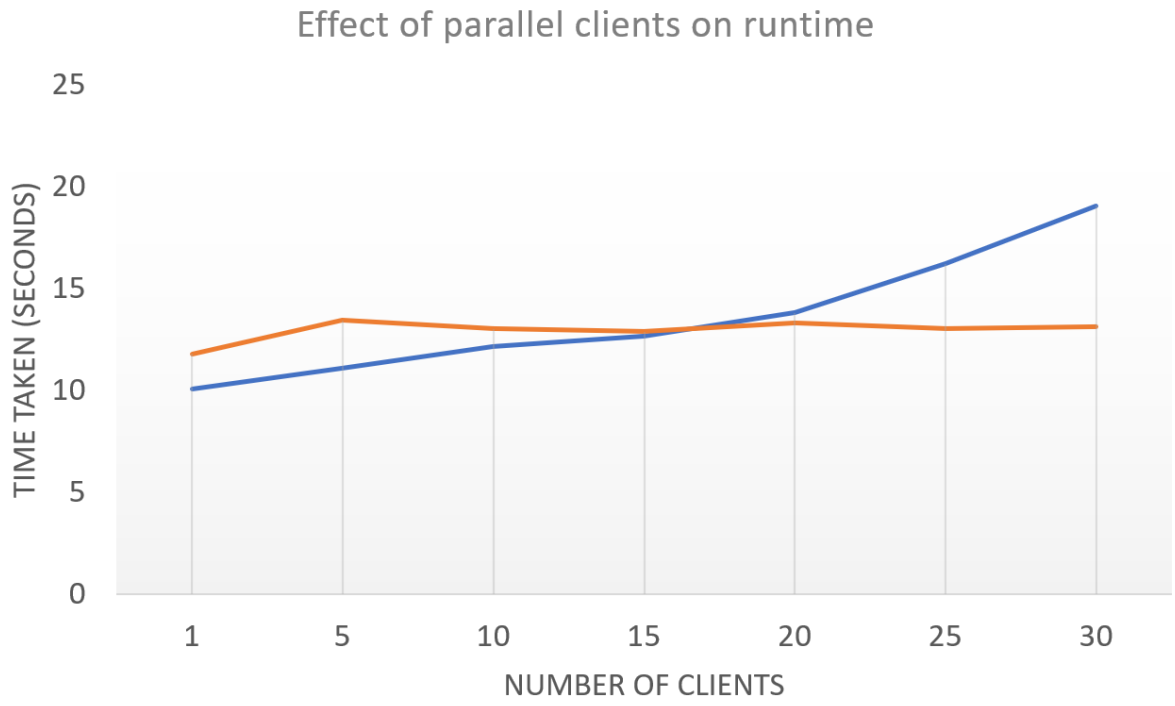

Figure 5.8 Linear increase in client runtime when clients start at the same time

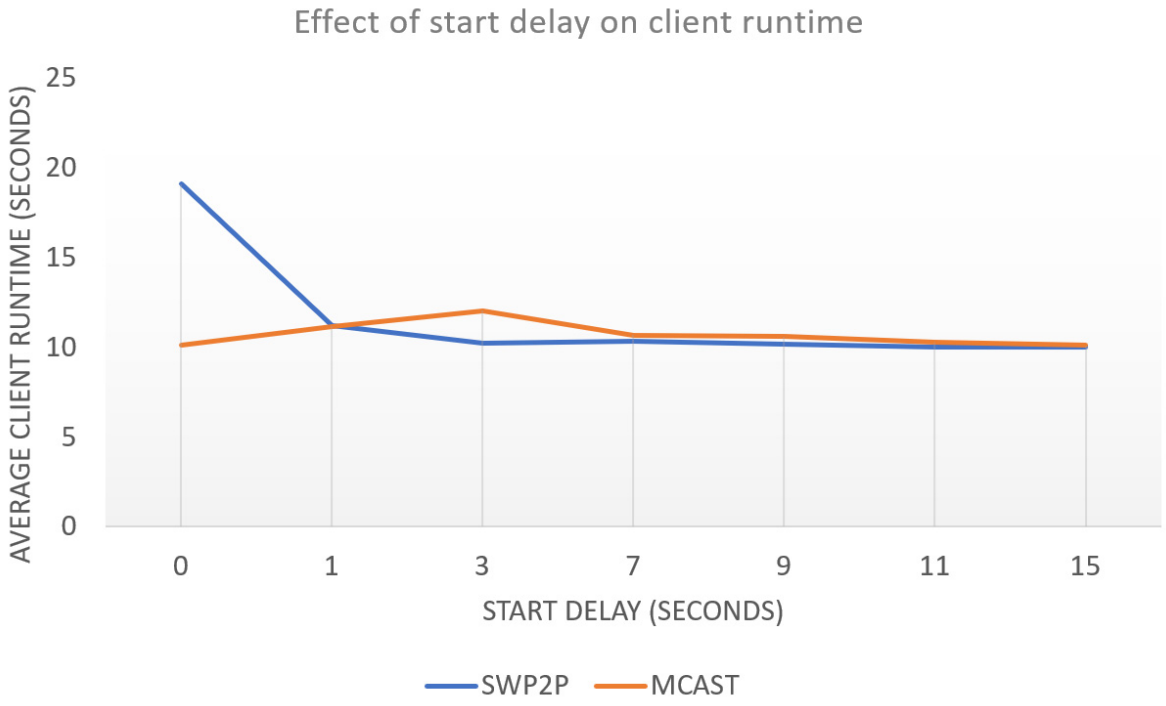

Figure 5.9 Start delay's effect on average client runtime

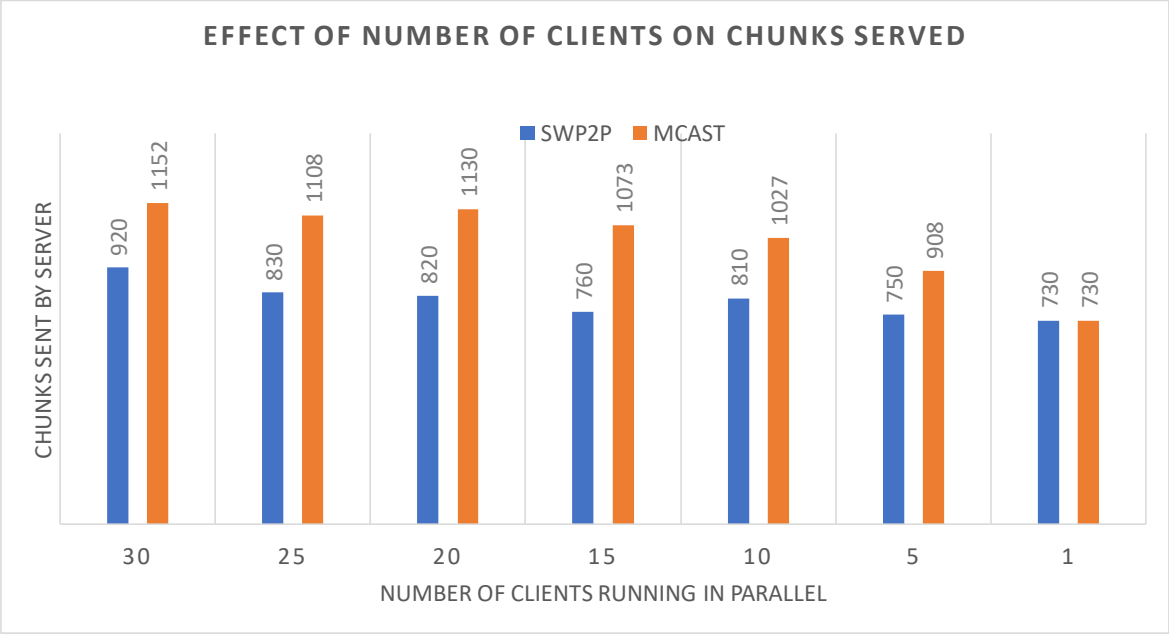

Figure 5.10 Consistent performance of SWP2P

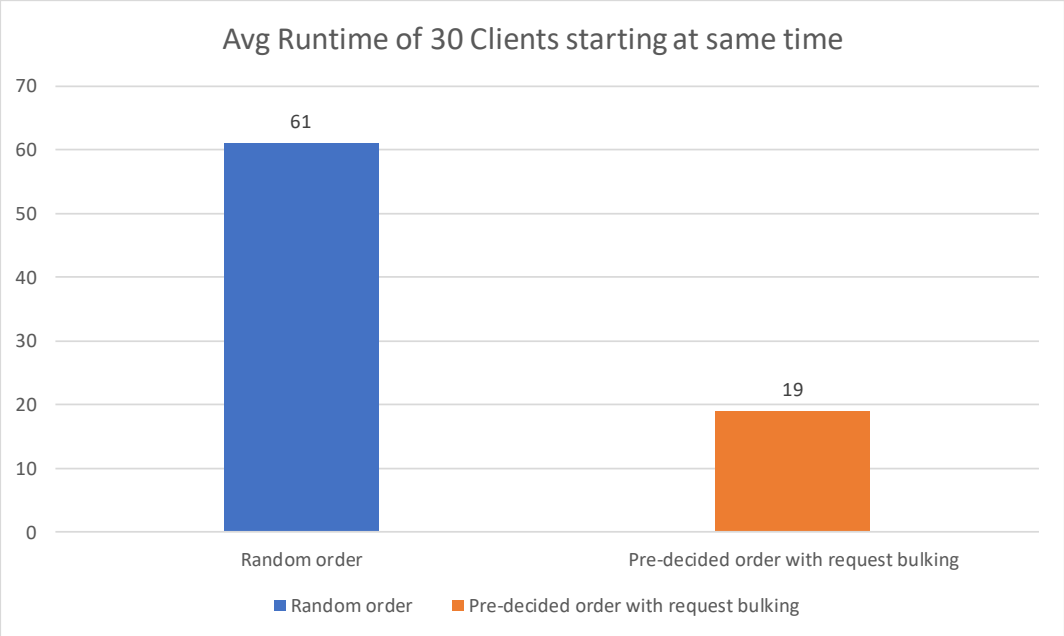

Figure 5.11 Request mechanism effectiveness

Table 5.1 Node configuration

Processor	Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz
Memory	64 GB
Network Speed	10000Mb/s

Table 5.2 Client receive time

Component 1	Network Receive Time	2 to 3 Seconds
Component 2	Decompression and Disk Write Time	8 to 9 Seconds

CHAPTER 6

CONCLUSION

Our motivation for this thesis was the high load on the server and trunk links when the multicast-based Frisbee protocol was deployed in Cloudlab. We introduced Switch Assisted Peer-to-Peer transfer mechanism to overcome this issue. In the evaluation chapter, we addressed various challenges that were overcome by the SWP2P method and how the concurrency among clients can be leveraged to attain a lower load on the server and the trunk links. We also showed how the longer decompression and disk write times help achieve our goal. The SWP2P method provides the advantages of a peer-to-peer transfer system by only maintaining the availability database on the switch, thus eliminating the need to maintain an overlay network. In our evaluation, SWP2P reduces the server load from 21% to 96%, depending on the start delay parameter. SWP2P also performed better than multicast transfer in all the scenarios we tested.

Thus, we proved that in the Frisbee disk loading system, Switch Assisted Peer-to-Peer transfer could significantly lower the load on the server while maintaining high performance.

In the future, the SWP2P system can be extended to be used across switches by exchanging a consolidated report message among switches. Such an extension will allow

the benefits of the SWP2P system to be scaled across the network by further lowering the load on server and trunk links.

REFERENCES

[1] M. Hibler, L. Stoller, J. Lepreau, R. Ricci, and C. Barb, "Fast, scalable disk imaging with Frisbee," in *USENIX 2003 Annual Technical Conference*, San Antonia, TX, USA, 2003.

[2] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, and L. Landweber, "The design and operation of cloudlab," in *USENIX Conference on Usenix Annual Technical Conference*, Renton, WA, USA, 2019.

[3] R. Ricci, "CloudLab," 22 05 2020. [Online]. Available: https://cloudlab.us/. [Accessed 22 05 2020].

[4] R. Ricci and E. Eide, "Introducing loudLab: Scientific infrastructure for advancing cloud architectures and applications," *Login USENIX Mag.,* vol. 39, no. 6, 2014.

[5] K. Atkinson, G. Wong, and R. Ricci, "Operational experiences with disk imaging," in *11th USENIX Symposium on Networked Systems Design and Implementation*, Seattle, WA, USA, 2014.

[6] C. Papadopoulos, G. Parulkar, and G. Varghese, "Light-weight multicast services (LMS): a router-assisted scheme for reliable multicast," *IEEE/ACM Trans. on Netw.* vol. 12, no. 3, pp. 456-468, 2004.

[7] P. Savola, "Overview of the internet multicast routing architecture," 2008. [Online]. Available: https://tools.ietf.org/html/rfc5110 [Accessed 22 05 2020]

[8] "OPX (OpenSwitch)," [Online]. Available: https://www.openswitch.net/. [Accessed 22 05 2020].

[9] "OpenSwitch OPX Developers Guide Release 3.0.0," 2018. [Online]. Available: https://archive.openswitch.net/docs/3.0.0/openswitch_opx_300_dev_guide.pdf. [Accessed 22 05 2020].

[10] Big Switch Networks , "Open Network Linux," [Online]. Available: http://opennetlinux.org/. [Accessed 22 05 2020].

[11] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Content-based addressing and routing: A general model and its application.," *Technical Report CU-CS-902-00,* 2000.

[12] W. Li, S. Chen, and T. Yu, "UTAPS: An underlying topology-aware peer selection algorithm in BitTorrent," in *22nd International Conference on Advanced Information Networking and Applications*, IEEE, 2008, pp. 539-545.

[13] B. Cohen, "The BitTorrent Protocol Specification," 4 02 2017 . [Online]. Available: https://www.bittorrent.org/beps/bep_0003.html. [Accessed 22 05 2020].

[14] S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang, "A reliable multicast framework for light-weight sessions and application level framing," *IEEE/ACM Trans. on Netw.,* vol. 5, no. 6, pp. 784-803, 1997.

[15] V. Jacobson, M. Mosko, D. Smetters, and J. Garcia-Luna-Aceves, "Content-centric networking," *Whitepaper, Palo Alto Research Center,* pp. 2-4, 2007.

[16] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking named content," in *5th International Conference on Emerging Networking Experiments and Technologies*, Rome, Italy, 2009.

[17] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron, "Topology-aware routing in structured peer-to-peer overlay networks," in *Future Directions in Distributed Computing: Research and Position Papers*, Springer Berlin Heidelberg, 2003, pp. 103-107.

[18] X. Lin, M. Hibler, E. Eide, and R. Ricci, "Using deduplicating storage for efficient disk image deployment," in *10th International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TRIDENTCOM)*, Vancouver, Canada, 2015.