# EMUSTORE: LARGE SCALE DISK IMAGE STORAGE AND DEPLOYMENT IN THE EMULAB NETWORK TESTBED

by

Raghuveer Pullakandam

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 2014

The University of Utah Graduate School

## STATEMENT OF THESIS APPROVAL

The thesis of **Raghuveer Pullakandam**

has been approved by the following supervisory committee members:

| | | |
|---|---|---|
| **Robert Ricci** | , Chair | **07/26/2012** Date Approved |
| **Ganesh Gopalakrishnan** | , Member | **08/31/2011** Date Approved |
| **Matthew Flatt** | , Member | **08/31/2011** Date Approved |

and by **Alan Davis**, Chair of

the School of **Computing**

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

# ABSTRACT

The Emulab network testbed deploys and installs disk images on client nodes upon request. A disk image is a custom representation of filesystem which typically corresponds to an operating system configuration. Making a large variety of disk images available to Emulab users greatly encourages heterogeneous experimentation. This requires a significant amount of disk storage space. Data deduplication has the potential to dramatically reduce the amount of disk storage space required to store disk images. Since most disk images in Emulab are derived by customizing a few golden disk images, there is a substantial amount of data redundancy within and among these disk images.

This work proposes a method of storing disk images in Emulab with maximizing storage utilization, minimal impact on performance, and nonintrusiveness as primary goals. We propose to design, implement, and evaluate ***EmuStore*** — a system built on top of a data deduplication infrastructure for efficiently storing disk images in the Emulab network testbed. The goal of this system is to take advantage of duplicate data in storing the disk images while remaining unobtrusive to other Emulab components.

# CONTENTS

# LIST OF FIGURES

# CHAPTER 1

# INTRODUCTION

Emulab [40] provides users with a network emulation environment. Emulab temporarily allocates appropriately configured physical machines to individual users for network experimentation. Disk loading is an important step in the process of node configuration in Emulab. Disk loading involves deploying operating system images, also known as disk images, to the target nodes. Emulab uses the Frisbee [23] disk image loading system for deploying disk images. Frisbee is a high performance disk imaging system optimized for LAN environments.

Emulab provides users with a standard set of disk images which the users customize. There are thousands of such customized disk images in Emulab and the number is growing. Storing these wide variety of standard and customized disk images requires significant storage resources. Considering that the size of a standard FreeBSD disk image in Emulab is 300MB, a collection of 3000 disk images may consume 1 TB of storage space.

Most of the disk images in Emulab are derived from a few disk images known as golden images. Hence, it is logical to assume that there exists a fair amount of duplicate data within and among disk images. Emulab currently stores a copy of every disk image on a network file system for easy access. Quite often, two disk images differ only in their kernel version or packages installed. For example, the only difference between a Ubuntu 10.10 disk image and a Ubuntu 10.10 disk image with zfs application installed is the presence of the ZFS filesystem on the latter disk image, which implies only a few data regions corresponding to the installed package are affected.

Hence storing a copy of every disk image leads to storing many redundant data regions even though they are identical to the data regions on the golden image from which the disk image is derived. The problem hence boils down to the efficient utilization of storage resources by avoiding storage of redundant data.

Our goal is to build a system that exploits the data redundancy prevailing within and among disk images in Emulab to maximize storage utilization while remaining unobtrusive to the other Emulab components and generating minimal impact on the overall disk loading performance.

## 1.1 Thesis Statement

Significant storage savings in the Emulab network testbed are achievable by exploiting data redundancy among disk images while remaining unobtrusive and imposing minimal impact on the overall performance of disk loading.

## 1.2 Focus

In summary, the focus of this thesis is twofold. First, to determine and quantify the amount of data deduplication [31, 12] existing within and among Emulab disk images and identify the optimum method to achieve significant storage savings and, second, to pipeline the overhead induced by this new approach of storing disk images with the other steps involved in disk loading thus creating only a minimal impact on the overall disk loading performance.

## 1.3 Approach

Our basic approach is to exploit the data deduplication existing within and among Emulab disk images. To realize the full benefits of data deduplication, we store Emulab disk images the way they are laid out on a filesystem as opposed to the existing approach of storing the disk images in a custom format convenient for fast deployment. To achieve the goal of significant storage savings and generating only a minimal impact on the overall performance, we explore the following techniques:

- Segmenting and storing regions of a disk image in a representation, consistent with the actual file system data layout.
- Experimenting to identify the optimum storage parameters which help us to achieve maximum storage space savings.
- Managing metadata that helps us reconstruct disk sector data in the client environment.
- Pipelining the process of regenerating parts of a disk image with other processes involved in the deployment phase thus achieving modest deployment latencies.

## 1.4 Trade-offs

Data deduplication is a technique used to achieve storage savings by discarding redundant data. Typically, data are stored as a collection of small data units known as data blocks. A data block is a sequence of bytes of a known length. The process of identifying and storing unique data blocks while discarding redundant data blocks is called data deduplication.

Though we strive to achieve maximum storage savings by exploiting duplicate data, it is, however, not always true that segmenting the file system into smaller regions provides maximum storage savings. Though there is a good chance that smaller storage segments duplicate well, there is a cost of storing associated metadata and the time taken to reconstruct portions of the disk image during deployment. Since, we use a data deduplication system which yields a 20 byte SHA-1 hash value for every data unit stored, it might not be a good idea to segment the disk image into very small regions for storing which might lead us to store more metadata than desired.

There is also a tradeoff between speed of reconstruction and the size of storage segments used for storing a disk image. Though the process of reconstruction can be pipelined with other processes involved in disk deployment, having very small storage segments increases the time to reconstruct desired parts of the disk image thus slowing down the entire process of disk loading. Contrarily, if the majority of the clients have very slow disks, having a very small storage segment might not affect the overall performance while providing significant storage savings on one hand.

## 1.5 Design Principles

We adhered to the following design principles through out our work. Leveraging a proven data deduplication system for storing content rather than building one from scratch. Therefore, our system had to operate within the constraints imposed by the framework we use. An important design choice we made was to adopt a request response model to interact with the data deduplication system. Since the data deduplication system we used forces our system to comply with its own threading model, ultimately leading to problems with portability, we had to use isolate it from other modules of the system. Secondly, we designed our system with a constraint of remaining unobtrusive to the client. In other words, we did not modify the client side of the existing system since modifying the client side of the protocol breaks the backward compatibility with already deployed client environments thus requiring the clients to redeploy the new system. As a result, we were under a strict mandate to ensure that a reconstructed chunk of a disk image can never exceed the size of 1 MB. In order to comply with the 1 MB size constraint, we had to come up with an algorithm to regenerate the exact chunk as the original during deployment phase. Finally, we designed our system in order to be capable of recreating a random out-of-order chunk which is crucial from the client perspective since at any point of time, multiple clients may request random chunks of a disk image. Therefore, reconstructing a disk image sequentially for every incoming request for a chunk is not practical and could prove to be a performance bottleneck thus requiring us to recreate a random chunk when requested.

## 1.6 Implementation Overview

As part of this work we implemented software components that come into picture during the process of Emulab disk loading. The following are the software artifacts implemented:

- A layer on top of the data deduplication system which enables other components to store and retrieve data blocks of variable size to and from the data deduplication system.
- A random chunk regeneration framework which uses the metadata information about the regions corresponding to the chunk and the layout information of the compressed part of the chunk to regenerate the precise chunk during deployment phase.
- Adding support to the Emulab's existing disk image deployment engine to interact with the chunk generator component and serve client requests for chunks by recreating the requested chunk using the data retrieved from the backend data deduplication system.

Another major contribution of this work is the quantification of the extent of possible storage savings by conducting large scale experiments with varying parameters and identifying the optimum parameters to achieve maximum storage savings.

## 1.7 Document Roadmap

Chapter 2 provides some background on the systems used in this work, namely, Emulab, Frisbee disk image loading system and Venti [34, 9, 8] archival [41, 16] storage system. Chapter 3 discusses various aspects of our design and concludes with a summary of how our large scale disk image deployment system works. Implementation of the system is discussed in Chapter 4. A comprehensive evaluation with statistics related to storage savings and justification for performance of our system is presented in Chapter 5. We discuss related work in Chapter 6 and conclude in Chapter 7.

# CHAPTER 2

# BACKGROUND

This chapter presents background on Emulab, Frisbee disk image loading system and the Venti archival storage system. This thesis work modifies the Frisbee disk image loading system to take advantage of our new storage infrastructure based on Venti archival storage system.

## 2.1 Emulab

Emulab is a network testbed, an experimentation environment which integrates simulation [30, 22], emulation [38] and live network experimentation [15] into a common framework with a goal of providing users with more realism [36] and control [35] in their experiment. To understand what this means, simulation, emulation and live networks are traditional environments used for network and distributed systems research. Each of these approaches have their pros and cons. Simulation loses accuracy due to abstraction. Live network environments such are MIT RON and PlanetLab achieve realism but are expensive to build and maintain. Emulators such as nse [28], DummyNet [38] provide greater control but require tedious manual configuration. Emulab brings the control and ease of use associated with simulation to emulation and live network experimentation without sacrificing realism.

Emulab provides a time- and space-shared environment for network researchers. Emulab behaves like a distributed operating system for network experimentation since it performs familiar operating system tasks such as resource allocation, management, synchronization and termination. An Emulab experiment is analogous to an operating system process. The life time of an Emulab experiment can vary from a few minutes to many weeks.

### 2.1.1 Experiment Creation

Emulab creates an experiment based on user parameters. Users specify various parameters such as number of nodes, desired topology, operating system and applications desired via a ns script. The fundamental units involved in a Emulab experiment are nodes and links. While specifying parameters, users can request specific hardware, operating systems, delay, bandwidth and packet loss. Emulab also supports events to change characteristics of a node or a link. The event system in Emulab is based on the Elvin [39] publish-subscribe system.

### 2.1.2 Experiment Swap-in

Experiment instantiation is the crucial step in Emulab. Experiment instantiation is known by the term “Swap-in.” The first step involved in swap-in is resource assignment. This NP-hard problem of finding and mapping the hardware that suits experimenter’s desired topology is solved by the assign [37] algorithm. After the resource allocation, experiment nodes contact Emulab’s master node to download and install the desired preconfigured operating system with applications known as disk images. The process of delivering and deploying the desired operating system environment on the target nodes is called “Disk Loading.”

### 2.1.3 Disk Loading

The main focus of this thesis work is on disk loading. Disk loading is a very important step in instantiating an Emulab experiment and thus needs to be very fast. Any delay in disk loading directly impacts the user as the user will have to wait to use his experiment until the disk loading is done. Hence, an indefinite delay in disk loading could be very annoying to the end user. In short, it is indispensible for disk loading in Emulab to be very fast. Therefore, disk loading in Emulab is performed using Frisbee [23]. Emulab’s disk image loading system is capable of deploying 50 gigabytes of data to 80 disks in 34 seconds.

## 2.2 Frisbee

Frisbee is a fast and scalable multicast based disk image loading system. It is one of the primary components of Emulab. Before introducing Frisbee, we need to understand disk imaging [13].

### 2.2.1 Disk Imaging

While transferring single files is the focus of most applications, many scenarios exist where transferring the entire disk contents efficiently is important. Operating system installation, disaster recovery [4, 14] and forensics are a few such scenarios. There are two different strategies for disk level distribution.

- Differential update used by directory syncing tools.

- Disk imaging used by Frisbee.

The following are the advantages of using disk imaging for distributing disk contents:

- Filesystem agnostic.
- Robust: Capable of transferring contents even if the filesystem is corrupt while file-based tools cannot.
- Versatility: Can easily replace a filesystem type by reloading a new disk while a file-based synchronizer cannot do this.
- Speed: Writing an entire disk image can be faster in a scalable environment.

### 2.2.2 Frisbee Design Principles

Frisbee disk loading system is based on some design principles which help make it fast and scalable in a LAN environment.

- Domain-specific data compression.
- Two-level data segmentation.
- Custom multicast protocol.
- High levels of concurrency in the client.

Although, Frisbee treats disk contents as opaque during disk image creation, it relaxes these requirements for a few common filesystem types. The information retrieved from the filesystem metadata is used to identify free blocks within the filesystem. During disk image creation, Frisbee skips free blocks and only compresses valid data regions thus drastically reducing the size of the resultant disk image. The action of skipping free blocks follows from the notion that free blocks do not add any content when deployed on the client and thus are not required to be transmitted over the network or written to disk.

Two-level segmentation of data, into randomly accessed chunks and sequentially accessed blocks within chunks, provides the flexibility of out of order request processing of chunks and parallelizing disk Input/Output (I/O) on the client side.

A custom multicast protocol helps Frisbee scale massively in a LAN environment. As we described earlier, multicasting reduces network congestion by avoiding transmitting the same packets again and again thus freeing up the network. Also, delivering same packets to multiple clients at once precludes the need for the clients to re-request missing segments.

Client side three-way multithreading to overlap disk I/O, zlib decompression and receiving chunk segments creates a high level of concurrency on the clients thus enhancing the speed of the protocol.

### 2.2.3 Frisbee Disk Image

The client's operating environment consists of an operating system, several applications and user data. Disk imaging distributes the entire contents of a disk to the target nodes. Since, disk imaging is filesystem agnostic, the contents transferred to the clients consists of the entire operating environment, including the operating system, applications and other data of the source disk.

A disk image is an encapsulation of the operating environment which includes the operating system, applications and other data. To put it simply, it is a block-by-block copy of the disk. When deployed on the client, a disk image metamorphosises into an exact clone of the source disk from which the disk image is created. In other words, disk image is the object of transfer in the process of disk imaging.

The process of creating disk image involves segmenting the disk data into smaller pieces known as "chunks." Assuming we have a disk with an operating system and a few applications installed on top of it, a disk image is created by storing the disk contents block-by-block. Such an approach as storing disk block-by-block would make a disk image filesystem agnostic. However, the filesystem metadata is used to identify free space that is discarded or ignored during the time of disk image creation. In short, only the valid data regions or sectors of a disk are part of a disk image.

As mentioned earlier, a disk image is composed of chunks. Each chunk packs the data corresponding to a few valid disk sectors. The information pertaining to the disk sectors comprised in the chunk, such as start offsets and sizes is stored in the header portion of the chunk. Hence, a chunk consists of a header and data. The data portion of the chunk is compressed using the zlib [21, 18] compression library. The compression of the chunk data region helps us reduce the overall size of the disk image, that is beneficial to reduce network congestion while transmitting over the network. The techniques of discarding free space and compressing chunk data part, during disk image creation typically limits the size of a disk image to one-third of the actual disk.

Since a chunk contains the data and metadata describing the disk sector offsets where the data belongs, chunks are considered independent entities. What this means is that a chunk can be processed independently on the client-side irrespective of other chunks. As a result, Frisbee clients are free to make out of order requests for desired or missing chunks and start processing the chunks as soon as they arrive. This flexibility of out of order processing guarantees concurrency on the client side thus speeding up deployment on the client. Figure 2.1 presents the composition of a Frisbee disk image. A chunk consists of regions that are described in the chunk header. A region is a collection of contiguous disk sectors on the disk. A region is represented using the start offset and the size of the region. When the Frisbee client receives a complete chunk from the server, it starts uncompressing the compressed data part of the chunk and writes the contents to the disk offsets denoted by the region start offsets in the chunk header. Hence, every chunk can be processed independent of other chunks.

### 2.2.4 Protocol

Frisbee is intended to be a fast and scalable disk image deployment system. The property of out of order processing on the client side makes Frisbee fast. However, it is the

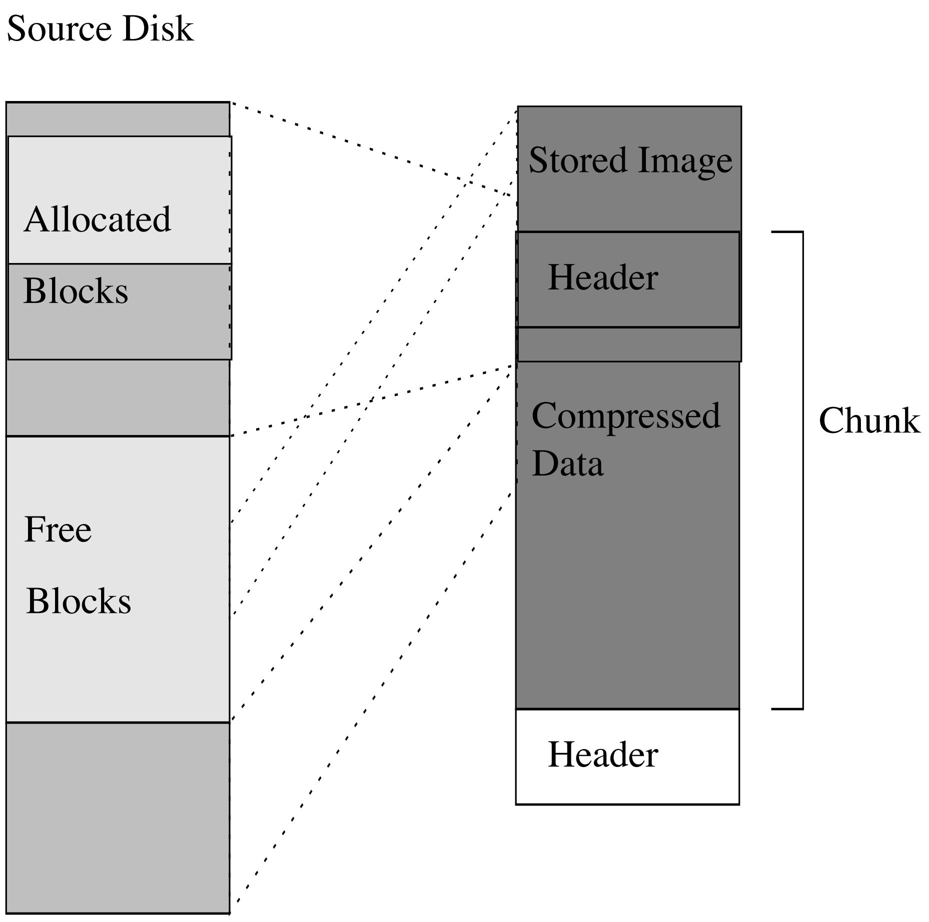

**Figure 2.1**: Frisbee Disk Image Composition. Based on [23]

protocol that Frisbee uses that makes in scale massively in a LAN environment. Frisbee is built on a client-driven protocol.

To understand the Frisbee protocol, a brief description of the way a chunk is transmitted over the network is essential. Frisbee clients request for a chunk from the server. Since a Frisbee chunk is an independent entity, clients overlap the operation of unravelling a chunk onto the disk with the process of requesting new chunks, thus, parallelizing the deployment process. However, a Frisbee client does not request the entire chunk in the event of a misdelivery. A Frisbee client only requests the missing parts of a chunk.

Before transmitting a chunk, Frisbee server divides a chunk into segments of 1 KB and transmits individual segments to the client. The Frisbee client tracks receives of these individual segments and reassembles them to obtain a valid chunk. In the event of lost segments, clients request only the missing segments of the chunk. The Frisbee server, having received requests for segments spanning across multiple chunks, treats requests to the same segment from multiple clients as one and multicasts the segments to all the respective clients. Quite often, multiple clients requesting the same chunk or parts of the same chunk prompts the server to multicast only one copy of the chunk or segment thus saving network bandwidth and scaling the system over a LAN very well.

Frisbee takes a constant 80 seconds to serve 25 nodes while Symantec Ghost [13] disk imaging system, which is a popular commercial disk imaging software, takes about 300 seconds to serve 25 clients.

## 2.3 Venti

Storing disk images in a data deduplicated storage system to achieve significant storage savings is the central requirement of this work. Instead of building our own efficient data deduplication system, we decided to rely on a proven data deduplication software and use it for our experimentation. The primary motivation to use a data deduplication system is to achieve storage savings over time. Quite a few commercial data deduplication softwares are available in the market. Popular ones include NetApp and Datadomain offerings which promise high storage savings. Another data deduplication system is Low Bandwidth Filesystem (LBFS) [32], that is a network filesystem that only transmits nonredundant data over the network.

We did not choose to use proprietary deduplication solutions due to various reasons. LBFS is a good option to serve content over the network but it operates at the file system level, whereas the contents that we transfer, i.e., disk images, operate at the block level.

Finally, Venti archival storage system was chosen for the following reasons:

- It is a block based storage system.
- It is used to store archival data such as e-mails accumulating over a period of time. It is designed to never delete the data it stores. Emulab disk images are cloned and modified by users as per their requirements. These modified disk images are stored in Emulab. However, we do not want to delete the old disk images since they can be reused later. We know the contents of the old disk images and they are verified secure. Hence, Venti fulfills our requirement as a disk image storage system.

Venti is an archival storage system built as a part of Bell Labs Plan9 [33] operating system. Venti is a highly efficient data deduplication system. We used the x86-64 Linux port [10] of the Venti software for our experimentation.

### 2.3.1 Key Properties

- Venti is a block level storage system. The interface for Venti enables clients to read and write variable sized blocks.
- Venti identifies data blocks by a hash of their contents. A hash of a data block is known as “fingerprint” since it is unique.
- Venti has a write-once behavior. Since, a block is identified using a fingerprint, a data block cannot be modified without changing its fingerprint.
- The most important property of Venti that provides the capability of data deduplication is the property of idempotent writes. Multiple writes of the same block can be coalesced and do not require additional space. This property of data deduplication is what eventually helps us achieve significant storage savings by storing disk images if disk images contain a lot of redundant data. In the event of high incidence of duplicate data within and among Emulab disk images, Venti stores only one copy of the redundant data blocks thus providing storage savings.
- Another benefit provided by Venti is the inherent data integrity. Both server and client can compute the hash of a data block served and compare them to determine the integrity of the data block served thus preventing data corruption.
- Venti uses SHA-1 cryptographic hash function as fingerprint.

- Venti provides the backend framework for complex applications to be built on top of it. Venti also provides data structures to realize complex systems. Several applications such as vac [6] — an archival utility; vacfs [7] — a filesystem; fossil [5] — another plan9 filesystem.

### 2.3.2 Configuration

Figure 2.2 presents the Venti configuration used in our system. The configuration consists multiple LVM [3] volumes configured on top of a RAID5 [11] disk.

### 2.3.3 Blob

Venti stores data in variable sized data units. We call them “blobs.” The maximum size of a blob is 56 kilobytes. In addition to storing blobs, Venti stores a hash value of every blob. We call them “fingerprints.” Venti identifies and retrieves a blob using its fingerprint. Venti achieves data deduplication by storing only one copy of identical blobs. In other words, only one copy of multiple blobs with the same fingerprint is stored. Obviously, smaller blobs increase the chance of data deduplication but smaller blobs means more fingerprints to keep track off, and more small reads and writes to the disk which affect disk

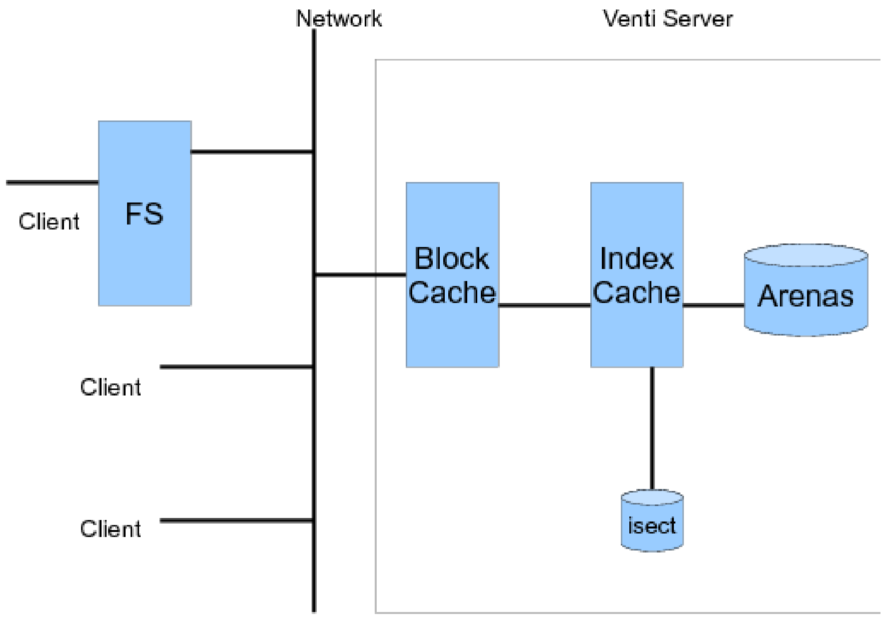

**Figure 2.2**: Venti Configuration. Based on [34]

I/O performance adversely. On the contrary, storing bigger blobs is faster but it leads to lesser data deduplication.

Hopefully this chapter has provided the right background information about our work as we move now into the next chapter which discusses the design of our solution.

# CHAPTER 3

# DESIGN

In this chapter, we discuss the design choices we made in building Emustore. Our system involves storing Emulab disk images in a deduplicated storage system and reconstructing random pieces of the disk image on-the-fly during deployment. The design decisions we discuss pertain to both data storage and retrieval aspects of our system.

## 3.1 Data Storage

Storing disk images in a deduplicated storage system is the central idea of our approach and it provides significant storage savings. However, it is important to carefully consider how we format and align disk image data in the data store, since the way we store data could have significant impact on the storage savings achieved. Before delving into the storage schema, we need to understand the concept behind the "Data Shift Problem."

### 3.1.1 Data Shift Problem

A disk image is comprised of files, folders and free data regions. When users create a new disk image from a base image, they add content in the form of application data and user specific data to the vacant data regions. As a result, some free data regions in the original disk image are now occupied. Hence, it is reasonable to assume that while storing a derived disk image in the deduplicated storage system, only those regions that are different from the base image are stored. This is not always true as it depends on the way we segment and store disk image data.

Figure 3.1 compares the disk data layout of two disk images. Base image is a standard Emulab disk image while "Derived Image" is a disk image derived from the "Base image" by an Emulab user. The dark regions indicate free space in a disk image. As we might notice, the derived image has one less free space region than the base image since it now consists of user data. If we chose to segment and store only the valid data regions of a disk image, we might face a few issues with alignment. Let B1, B2, ... , Bn be the segmented data units known as blobs which are the fundamental units stored in the deduplication storage

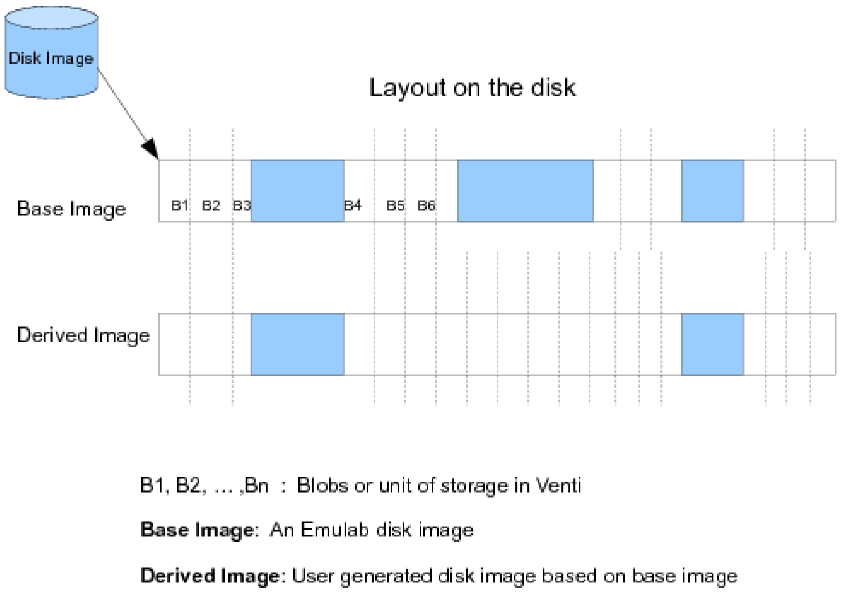
Disk Image
Layout on the disk
Base Image
B1 B2 B3
B4
B5 B6
Derived Image
B1, B2, ... ,Bn : Blobs or unit of storage in Venti
Base Image: An Emulab disk image
Derived Image: User generated disk image based on base image

**Figure 3.1**: Data Shift Problem

system. As we might observe, blobs B1 to B6 are identical in both the base image and the derived image. But due to the absence of an expected free data region in the derived image, new blobs are created. As part of this new blob creation, data from the newly filled areas are spilled into other blobs. In other words, the blob after B6 is a partial blob in the base image since the space between B6 and the next free block is less than the expected blob size. However, since the free data region is occupied in the derived image, the blob after B6 is now a full sized blob. As a result, all the blobs following B6 are now considered to be different. Consequently, this would create a ripple effect and lead to minute changes in all the blobs following. Even though the blobs are only aligned differently, they are considered different and are stored separately. This leads to storing a lot of blobs which is an undesired consequence. We refer to this problem as the data shift problem. The data shift problem played an important role in the design decisions we made and are explained in the following sections.

### 3.1.2 Storage Schema

There are different ways to store a disk image in a deduplicated storage system. The following are a few methods:

- Storing a disk image as it is.

- Storing raw data of a disk image chunk by chunk.
- Storing disk image raw data as a whole.
- Storing only the valid portions of a disk image raw data.

Storing a disk image in the deduplicated storage system without any modification is very convenient since it facilitates faster and predictable access to parts of the disk image during deployment.

Disk image is comprised of chunks which are independent data units consisting of compressed data and metadata. At finer granularity, however, a chunk consists of regions. Each region corresponds to a contiguous valid disk sector on the disk. The data within the disk sectors get altered as users make changes to the filesystem. This process of altering disk sector data has a twofold impact on the composition of a chunk. Firstly, even a slight modification to the data region alters the compressed version of the chunk in unpredictable ways. Secondly, the chunk header might get affected due to alteration in the data part. In other words, a minute alteration to the underlying data of a chunk changes the composition of the chunk in an unpredictable fashion and also affects all the subsequent chunks thus causing a ripple effect.

Another option for storing disk images is storing them in raw format. In other words, storing them uncompressed. A few benefits of this approach include:

1. No requirement for storing additional metadata.
2. Possibility of achieving higher rates of deduplication.

Before going further, it is important to understand what chunk reconstruction means. A disk image is composed of chunks that are transmitted over the network to the client in denominations of blocks. The process of aggregating blocks transmitted over the network into chunks on the client side is known as “chunk reconstruction.”

An obvious cost involved with storing disk images in raw format is high storage consumption. The uncompressed data are many times larger than the compressed data.

Yet another cost involved with storing disk images in raw format is overhead on “chunk reconstruction.” Chunk reconstruction process now involves identifying and compressing the raw data regions corresponding to the chunk on-the-fly, which is an expensive task. But this approach is worth pondering over, considering the possibility of hiding the latency induced by chunk reconstruction with other processes involved in disk deployment.

Storing disk images in a compressed format does not conform to the philosophy of data deduplication. Disk images derived from a common base disk image do not yield storage savings if stored in a compressed format as opposed to raw format. Early chunk changes (due to data compression) in a disk image can affect all subsequent chunks in an unpredictable way thus adversely affecting the rate of data deduplication. On the contrary, storing disk images in raw format exhibited good rate of data deduplication and significant storage savings. Hence, we decided that it would be better to store disk images in raw format. Having decided that storing raw data in the deduplicated storage system is a better approach rather than storing compressed data, there is still another design decision to make, that is, whether to process and store a disk image chunk by chunk or all at once. Both the approaches have their pros and cons.

The advantage of storing a disk image chunk by chunk is the ease of locating raw data corresponding to a chunk during chunk reconstruction; whereas, storing an image all at once requires us to keep track of the raw data belonging to every chunk since it will need that information during chunk reconstruction.

Firstly, the compressed part of a chunk is uncompressed to obtain the raw data. Further, the raw data generated are considered as a contiguous piece of memory, thus segmenting it into fixed data units and storing them in the deduplicated storage system. Segments which are smaller than the fixed data unit size are considered partial segments and padded out to the fixed size.

This approach, however, suffers from the ”data shift problem” explained in section 3.1.1. The data shift problem here occurs due to the padding of partial segments. As illustrated in Figure 3.1, padding of partial segments gives rise to two segments in place of one and also shifts the data contained in the following segments thus affecting all the following segments. One way to eliminate the data shift problem occuring here is by tracking partial segments and filling them up with data from the first segment of the next chunk, instead of padding the partial segment which is a very complicated and intricate approach.

Storing a disk image all at once suffers from the disadvantage of tracking the raw data corresponding to each chunk. However, it is not a big demerit when seen from a performance standpoint as all the book keeping is done offline and does not interfere with the chunk reconstruction process, which is critical to the overall performance of the system. It has a less complicated design and does not suffer from the data shift problem.

It is less complicated because it does not involve tracking partial segments and filling them up with data from the next chunk. Since the size of the disk partition and the size of the data segment are fixed, the segment boundaries never change. Only the data lying in the segments varies with the addition of application or user data. Even with the addition of new data, the data already present on the disk partition does not move.

We adopted a two step method to store a disk image in Venti. The first step involves installing the disk image on a physical disk. The second step involves reading this physical disk a chunk at a time and storing the corresponding raw data in Venti. This process is both time and space consuming in that uncompressing the disk image to a physical disk and reading every segment from the disk is time consuming while storing the entire raw disk data in Venti consumes more storage space. The issue of storage space consumption can be addressed by zeroing all the free blocks during installing the disk image to a physical disk. As a result, only one zero block is stored in Venti in place of all free blocks thus solving the problem of excessive storage space consumption. Also, since the process of storing disk images is done offline, the time intensive processes involved do not interfere with the performance of chunk reconstruction.

Another cost involved with storing a disk image raw data all at once is the need to track the raw data corresponding to a chunk. However, since the process of book keeping is a one time activity and is done offline, it does not affect the performance of the critical path. Also, since we store chunk headers persistently, the raw data pertaining to a chunk could be easily retrieved from the chunk header.

Having experimented with several approaches of storing disk images in the deduplicated storage system, we believe the method of storing disk image raw data all at once is the optimal design choice for the following reasons:

1. Its simplistic design.
2. All time intensive processing is done offline.
3. The raw data information corresponding to a chunk could be easily obtained.
4. The problem of more than desired storage consumption can be solved by zeroing free blocks.

## 3.2 Chunk Reconstruction

Chunk reconstruction is a crucial step in the process of disk image deployment. The disk image deployment process involves the Frisbee client requesting an arbitrary chunk (because the chunk order is randomized to avoid starvation of certain clients) from the

Frisbee server. In the traditional model, the Frisbee server directly accesses the chunk from the disk. However, in our system, serving a chunk is a much more intricate sequence of steps. The design decision to store disk images in a deduplicated storage system to achieve significant storage savings comes with the cost of chunk reconstruction during deployment phase.

### 3.2.1 Issues

Chunk reconstruction might seem intuitive and straightforward. In practice, however, chunk reconstruction comes with its own set of problems which are described below:

- Ensuring that the chunk size is less than 1 MB (as required by the Frisbee protocol).
- Reconstructing chunk headers for arbitrary chunks.
- Identifying and retrieving the raw data that goes into the chunk.

### 3.2.2 Design Decisions

Addressing problems with chunk reconstruction leads us to make the following design decisions.

The Frisbee protocol restricts the size of a disk image chunk to 1 MB to prevent the necessity of dealing with variable sized chunks on the Frisbee client. A 1 MB chunk is divided into 1 KB units each of which is multicasted over the network. The Frisbee client receives the 1 KB packets, requests missing 1 KB data units and uses these 1 KB packets to reassemble the chunk. This is what makes Frisbee protocol client-driven.

Our system has to comply with the chunk size constraint in order to remain unobtrusive to the clients, which is one of our design principles. The reason for remaining unobtrusive stems from the fact that any modification to the client side of the Frisbee protocol would require us to update a significant number of already deployed clients with the new protocol changes which is not a desirable proposition.

Since the size of the compressed data obtained by compressing a given data unit is unpredictable, it is difficult to ascertain if a reconstructed chunk exceeds 1 MB unless we compress the exact raw image data that a given chunk is derived from. Hence, the problem boils down to identifying the raw data needed from which a given chunk is synthesized. However, assuming we obtain the raw data corresponding to the chunk, it is not guaranteed that the reconstructed chunk size is always less than 1 MB. It is not just the data but also the method of compression that affects the size of a reconstructed chunk.

Assuming we obtain the raw data that goes into the chunk, one way of reconstructing the chunk is to perform a deflate [19] operation on the entire raw data. Our results showed that the chunk size exceeds 1 MB once in a while using this approach. We need to understand the structure of a chunk to understand the anomaly in the size of the reconstructed chunk.

The compressed part of a chunk is obtained by performing a zlib deflate operation on the associated raw data. The compressed part of the chunk is comprised of small, independent zlib deflate blocks. Each zlib deflate block is capable of being uncompressed separately. These independent zlib deflate blocks correspond to valid regions of the chunk. To further explain, a chunk is comprised of regions which correspond to a valid contiguous disk sector. When constructing a chunk, data contained in each of the valid regions is separately compressed using zlib deflate to produce a deflate block. The deflate blocks generated by compressing each individual data region are concatenated to produce a chunk. The chunk header describes the information about which regions are constituted in the chunk, their starting offsets and sizes. In fact, this is the same algorithm used by Frisbee to synthesize chunks.

Having understood the composition of a chunk, it makes more sense to comprehend why compressing the raw data associated with a chunk as a single zlib block produces a chunk which is sometimes greater than 1 MB. Clearly, compressing entire raw data in one go does not take into consideration the zlib block boundaries which denote the end of the current region and the start of the next valid region within a chunk.

Hence, to ensure the chunk size always complies with 1 MB size, we designed our chunk reconstruction solution to pay attention to the zlib block boundaries within the compressed part of the chunk. We record these boundaries and store them persistently in the metadata we maintain for the disk image. During the process of chunk reconstruction, we use this zlib block boundary information along with the raw data associated with the chunk to recreate the requested chunk.

An important design decision we made is retaining the mapping raw data to the chunk. In other words, we designed our solution such that raw data associated with a particular chunk does not change. Altering the raw data associated with a chunk would require us to recompute the region start and size information stored in the chunk header and would also make changes to the block header of the chunk. This process is tedious and error prone as there is a chance that the chunk size sometimes exceeds 1 MB. Hence, we made a choice to never modify raw data associated with a chunk, to achieve predictability.

Our decision to store chunk headers persistently has a two fold motivation. Firstly, the need to map fixed raw data to a chunk. Secondly, the ability to regenerate an arbitrary chunk at any given time without depending on other disk image chunks. Without readily available chunk headers, regenerating an arbitrary chunk would require us to regenerate all the preceding disk image chunks which could potentially slow down the entire process of disk image deployment. Since, the raw data associated with a chunk never changes, the region metadata stored in the chunk header does not need to change. The only chunk header attribute altered by the chunk reconstruction process is the size field in the block header. This happens since the reconstructed chunk, though less than 1 MB, is often not the same size as the original chunk thus requiring us to update the size field in the block header.

In this chapter we discussed the design decisions and the motivational factors in our decision-making process. The next chapter deals with the implementation details of our system.

# CHAPTER 4

# IMPLEMENTATION

In this chapter, we discuss the implementation details of our system.

## 4.1 System Overview

The system's implementation involves a revamp of the existing Frisbee server to enable it to serve disk images from the Venti archival storage system. This process needs to be transparent to maintain compatibility with Frisbee client code. The implementation of the system is a two-step process. The first step deals with storing the disk images in Venti so as to provide maximum data deduplication. The second step deals with reconstructing chunks from Venti, which are in turn served to the clients.

## 4.2 Components

Our system adopts a modular architecture clearly defining the responsibility of each component. The reason for embracing a modular architecture is to offer flexibility. For example, isolating the storage component from the rest would help us experiment with new storage technologies such as flash or solid state based storage systems in future while providing an unchanged interface to other components. Our system is comprised of the following main components.

- Frisbee server
- Venti server
- Chunkmaker

### 4.2.1 Frisbee server

Frisbee server serves an Emulab disk image to clients using a custom multicast protocol. Typically, Frisbee reads disk images from the file system. We modified Frisbee enabling it to serve disk images from the Venti storage system. This process involves retrieving the appropriate data from Venti and reconstructing the requested chunk on-the-fly. The changes to Frisbee protocol only modify the server. We do not change the Frisbee client.

### 4.2.2 Ventiserver

Ventiserver component is an interface to Venti put and get functionality. We store data blocks of a fixed size, known as blobs, in Venti. Storing a blob in Venti returns a SHA-1 hash that is used to uniquely identify the data blob. All requests for storing and retrieving blobs in our system go through Venti server. Since, the version of Venti storage system we use is a linux port of the original Plan9 operating system version, we had to deal with some plan9 conVentions. Ventiserver abstracts out these details and provides consistent interface to other components of our system.

### 4.2.3 Chunkmaker

Chunkmaker component is responsible for chunk reconstruction on-the-fly. It receives requests from Frisbee server to deliver a chunk. Chunkmaker interacts with Ventiserver to identify and retrieve appropriate data and recreates the chunk. Chunkmaker is also responsible for gathering metadata which shall be utilized during chunk reconstruction. However, the metadata computation and storage are done offline and thus do not impact the performance of chunk reconstruction.

## 4.3 Storage Management

In this section, we discuss the process of storing disk images in Venti. We also describe what metadata we store, how we store it and how is it used for reconstructing the chunk on-the-fly during deployment. Figure 4.1 describes the workflow of storing disk images in Venti.

### 4.3.1 Storing Disk Images

Our system stores Emulab disk images in raw or uncompressed format. This decision to store disk images in raw format is motivated by the "data shift problem" explained in section 3.1.1. A disk image is first laid out on a disk partition. We decide on the blob size used for storing the disk image in Venti since we chose to store data as fixed size blobs. The disk partition is read in segments of blob size and each segment is individually written to the Venti storage system. Upon writing a blob, Venti returns a SHA-1 hash that is used to uniquely identify the data blob. The SHA-1 hashes corresponding to a disk image are stored persistently to be used later during disk image deployment.

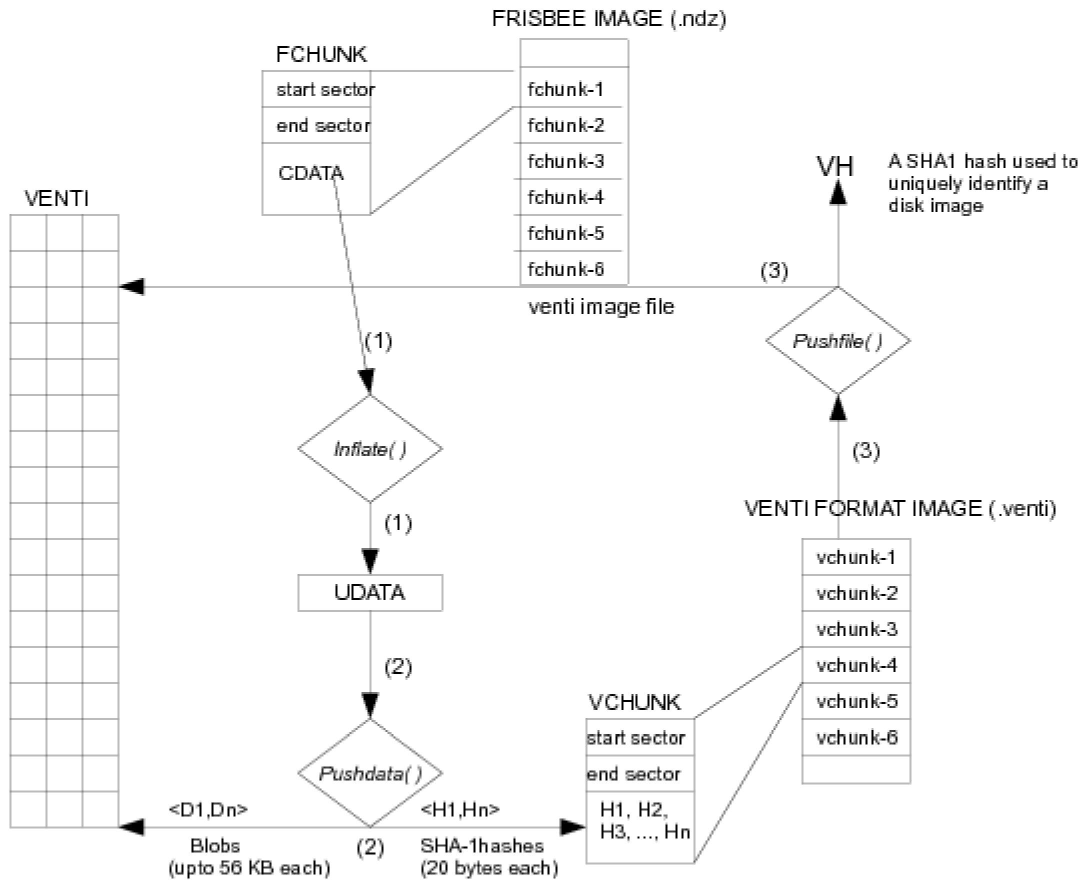

**Figure 4.1**: Store Disk Image In Venti Storage System

### 4.3.2 Storing Metadata

Storing metadata is important for enabling on-the-fly chunk reconstruction during deployment time. Hence, it does not have an impact on the chunk reconstruction performance. We store the following metadata.

- Chunk headers.
- Zlib block map.

Chunk headers are stored in their original format persistently. Since the reconstructed chunk consists of the same data as the original chunk, we do not need to change the chunk header information.

As discussed in section 3.2.1, there exists a potential issue of reconstructed chunk overflowing 1 MB thus leading to data corruption. Before we described how we addressed this issue, we need to explain zlib block boundaries. The data corresponding to a chunk is compressed using zlib compression library. Zlib takes a byte buffer as input and returns a compressed data buffer. We capture the block boundaries of a zlib stream and store them persistently as zlib block map. This zlib block map along with the chunk header is used to reconstruct the chunk on-the-fly during deployment.

## 4.4 Deployment

In this section, we discuss disk image deployment and how on-the-fly chunk reconstruction is done during a disk image deployment. Figure 4.2 explains the workflow of serving disk image chunks during disk image deployment.

### 4.4.1 Serving chunks from Venti

Frisbee serves chunks to clients upon request. The way we propose to store disk images in Venti complicates the process of serving chunks. The process of serving a chunk now involves identifying the blobs corresponding to the chunk, retrieving them from Venti, composing the chunk, and delivering it to the client. The blobs corresponding to the chunk are identified using the chunk header and zlib block map retrieved from the persistent metadata storage. The blobs identified are retrieved from the Venti storage system using corresponding SHA-1 hashes. The blobs retrieved from Venti are deflated using zlib to create the compressed part of the chunk and this in turn, along with the chunk header, is used to reconstruct the chunk, which is multicast to the clients.

In short, the process of serving chunks from Venti involves the following steps.

- Identifying the raw data corresponding to the chunk.
- Retrieving the raw data from the Venti storage system.
- Compressing the raw data using zlib.
- Coalescing chunk header with the compressed data.
- Deploying the chunk.

Identifying raw data belonging to a chunk could be done by examining the chunk header. Since we already stored the chunk header persistently, retrieving the start offsets and sizes of regions corresponding to a chunk from the chunk header helps us easily identify the raw data belonging to the chunk.

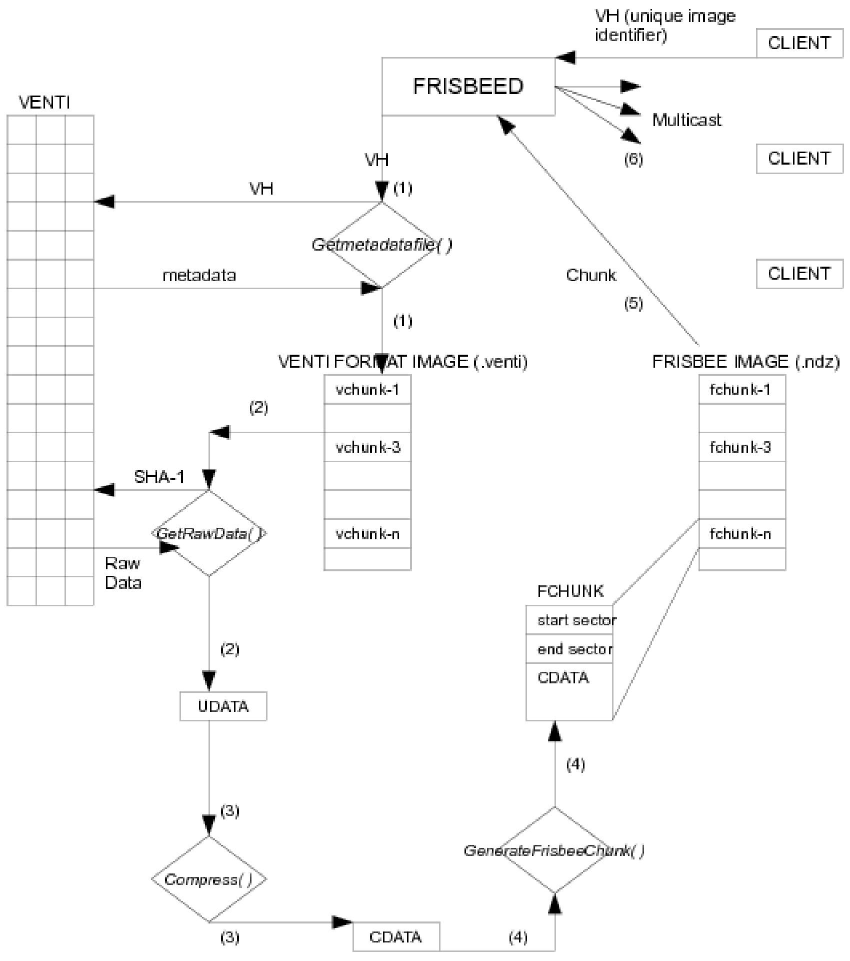

**Figure 4.2**: Serve Disk Image Chunks From Venti Storage System

Once we identify the raw data belonging to a chunk, we retrieve the SHA-1 hashes using the raw data that could be retrieve from Venti. The SHA-1 map stored persistently while storing disk images in Venti helps us identify this information. We load the SHA-1 map and other concerned metadata corresponding to the disk image being served into memory in order to avoid creating overhead on the process of chunk reconstruction.

We use the SHA-1 hashes to retrieve the raw data from the Venti storage system. This process involves requesting a Ventiserver component for data corresponding to a SHA-1 hash. Once we retrieve data blobs corresponding to all the SHA-1 hashes, we have the entire raw data corresponding to the chunk requested.

Compressing the raw data is a very important step and the most time consuming step of the entire chunk reconstruction process. The primary reason for compressing the raw data is ensuring compatibility with Frisbee clients. The Frisbee client expects to receive a chunk in a custom compressed format. Moreover, as explained in section 3.2.1, there exists a potential data corruption issue due to the chunk exceeding the expected size. To address this issue, we use the knowledge of zlib compression stream structure.

As described in section 4.3.2, knowledge of zlib stream structure is utilized to create zlib block map while constructing metadata. Each entry of a zlib block map indicates the amount of raw data belonging to a particular zlib stream. Utilizing knowledge of zlib compression stream helps us regenerate a chunk with a bounded size thus complying with Frisbee protocol.

The compressed data part generated using the chunk reconstruction process is coalesced with the original chunk header retrieved from the persistent metadata to recreate an identical chunk ready for deployment. This chunk is deployed to clients the way regular chunks are deployed thus remaining transparent to the client side of Frisbee protocol.

# CHAPTER 5

# EVALUATION

Our evaluation of EmuStore considers the following metrics:

- Rate of data deduplication
- Storage consumption
- Performance impact

## 5.1 Test Infrastructure

### 5.1.1 Test Setup

Our experiments use a central server which acts as both an archival storage system and a Frisbee server. In addition, we deployed 32 clients, each of them running a Frisbee client and requesting chunks from the Frisbee server. All machines have an 8-core 2.4 GHz 64-bit Xeon processor, 12 GB RAM, and two 232 GB, 10000 RPM SCSI disks. The server has two 1.5 TB, 10000 RPM SCSI disks aligned in a RAID-5 configuration thus effectively providing 2 TB storage space. All machines run standard Emulab Fedora Core 8 x64 operating system.

### 5.1.2 Test Data

The test data consist of 1020 Emulab disk images, the total size of which is 606 GB. Out of these 1020 images, 212 images, the total size of which is 115 GB, are standard Emulab images while 808 images, the total size of which is 491 GB, are user generated images.

## 5.2 Rate of Data Deduplication

We measure the rate of data deduplication as the ratio of the number of unique data blobs stored in Venti and the total number of data blobs stored in Venti. We are already aware of the property of Venti of discounting the incoming data blobs that are already stored in the system. This property enables us to achieve data deduplication. Each data blob stored in Venti corresponds to a contiguous data region on the filesystem from which the image is generated. Since, a large number of emulab disk images are derived from a few standard images, it is prudent to assume that disk images tend to share data blobs. A reasonable portion of a disk image is often unaffected by the addition of packages or user data which tend to deduplicate well.

In Figure 5.1 and Figure 5.2, we present the total number of data blobs across all images and the number of unique data blobs for different blob sizes. As one might notice, the rate of data deduplication increases as the blob size decreases. This is because of the narrowing of the window of comparison. We consider a slightly modified version of a blob as different. Larger blob sizes means more sensitivity to minute variations within the blob. To see what this means, first suppose that two blobs, each of size 32 KB and lying at the same offset when two different disk images are laid on the disk, are identical except for a contiguous region of 8 KB. In such a scenario, using a blob size of 32 KB would mean a loss of one out of one duplicate blob while using a blob size of 8 KB would mean a loss of only one duplicate blob out of four.

## 5.3 Storage Consumption

In this section, we present the disk storage utilization statistics for various scenarios. Firstly, it is imperative to understand the metrics of comparison. We base our results on

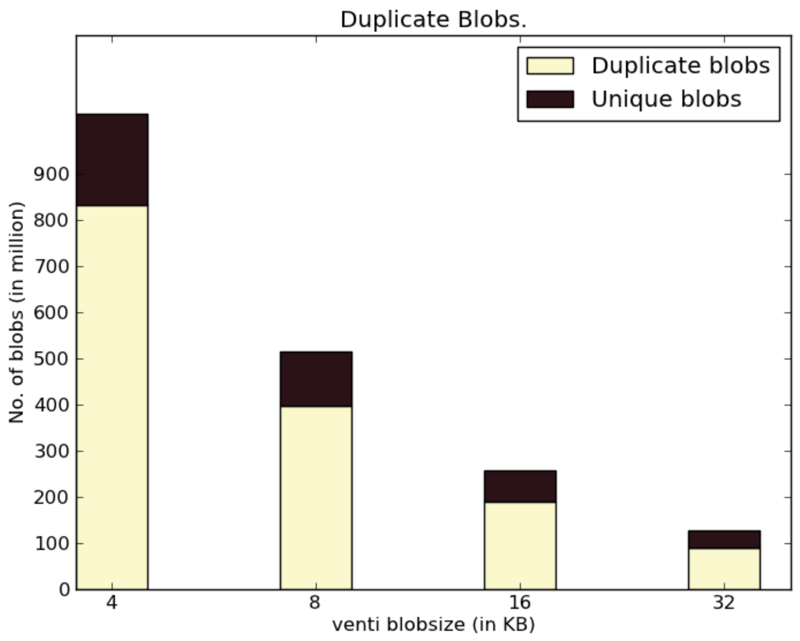

**Figure 5.1**: Number of Duplicate Blobs

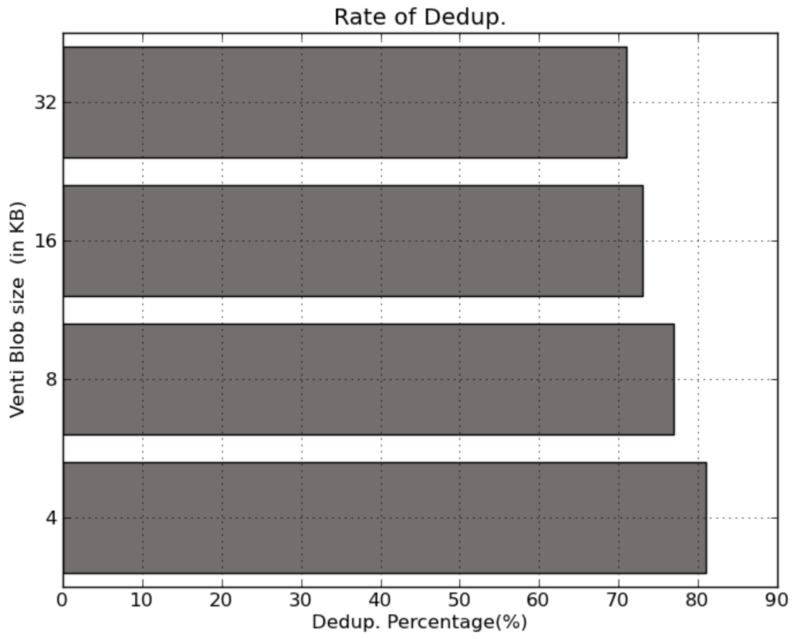

**Figure 5.2**: Rate of Data Deduplication

the comparative analysis of the actual physical storage resources utilized by the traditional approach and our approach of storing emulab disk image data. Figure 5.3 presents the storage utilization statistics for storing 600 GB of standard and user generated Emulab disk images. Figures 5.3a 5.3b 5.3c 5.3d show the fraction of storage used by our approach as compared to storing the disk images on the filesystem. The storage space consumed by our approach is not more than one-third of the storage space we would have consumed if we stored our disk images on a filesystem. As one might observe, the storage space consumed decreases with the decrease of Venti blob size. This result supports our assumption that larger blob sizes are more sensitive to minute changes and hence bring down the rate of data deduplication. Venti internally compresses the data blobs using a custom compression algorithm. Hence, the disk space utilized by Venti consists of compressed data. Venti decompressed the data while serving it to clients. However, choosing not to compress the data allows us to store the data in Venti in its original form. The benefits of storing uncompressed data include improved turn around time while serving a request for a data blob. However, the approach of storing raw data in place of compressed data comes with a cost of higher storage space consumption thus denting the overall storage space savings. Figures 5.4a 5.4b 5.4c 5.4d contrast the raw storage space consumed by our approach with the storage space consumed by the traditional approach of storing disk images on filesystem.

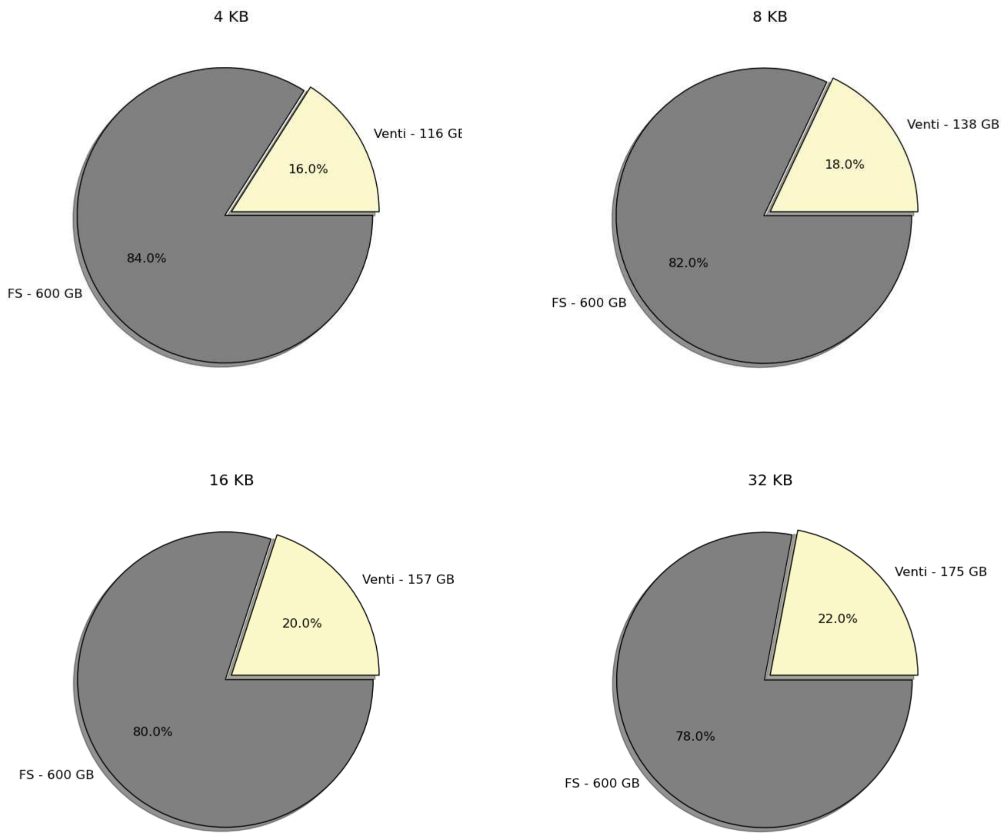

**Figure 5.3**: Storage Consumption: Compressed Disk Images vs. Venti for Several Blobsizes. a) Storage consumption for 4 KB Venti blobsize. b) Storage consumption for 8 KB Venti blobsize. c) Storage consumption for 16 KB Venti blobsize. d) Storage consumption for 32 KB Venti blobsize.

As one might observe, even storing raw data provides significant storage savings when the blob sizes are smaller. However, the experiments conducted reveal that decompressing data before serving it is not a bottleneck in the turn around time.

### 5.3.1 Metadata Storage

Though the storage consumption reduces with decrease in Venti blob size, there is a cost involved with storing the metadata. Storing images in Venti with a blob size of 8 KB involves storing up to four times the amount of metadata as opposed to using a Venti blob size of 32 KB. Hence, it is essential to identify the ideal blob size.

Figure 5.5 shows the disk space consumed by storing metadata and the aggregate storage utilization for each blob size.

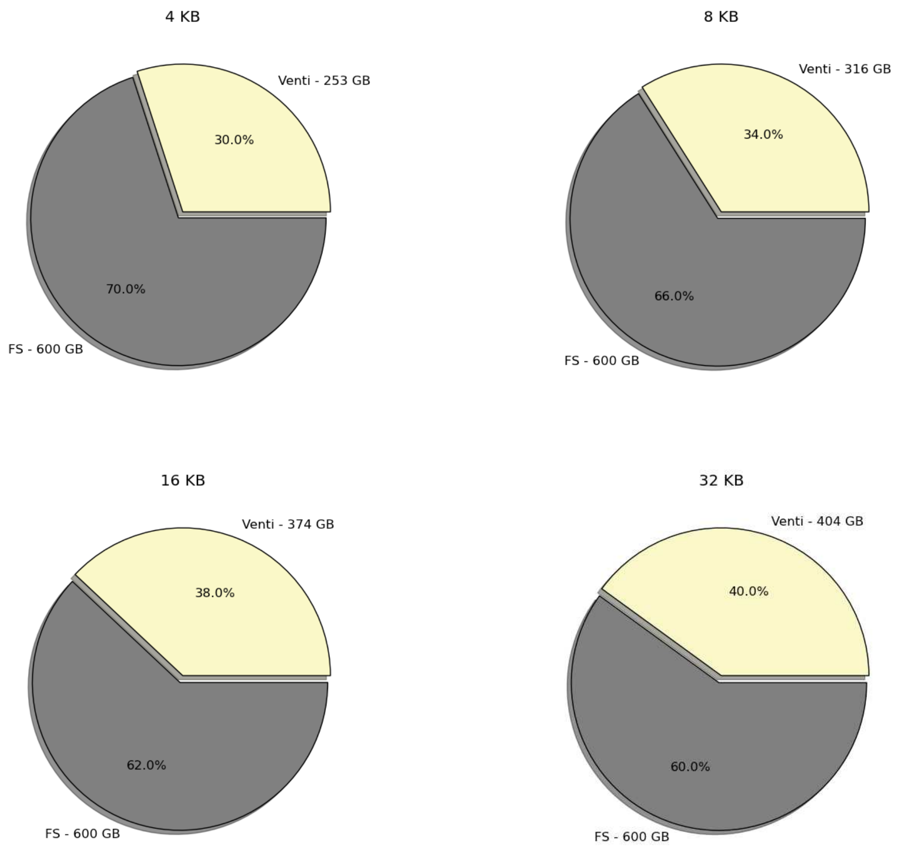

**Figure 5.4**: Storage consumption: Compressed Disk Images vs. Venti (raw data + metadata) for several blobsizes. a) Storage consumption for 4 KB Venti blobsize. b) Storage consumption for 8 KB Venti blobsize. c) Storage consumption for 16 KB Venti blobsize. d) Storage consumption for 32 KB Venti blobsize.

### 5.3.2 Aggregate Storage Savings

Figure 5.6 presents the aggregate storage consumption statistics while storing disk images using varying blob size parameters. As one might observe, the storage space consumed for storing data reduces while the storage space consumed for storing metadata increases as we reduce the blob size. However, it is imperative to identify the optimum blob size in order to achieve maximum storage space savings. From the statistics, it appears that a blob size of 4 KB and 8 KB provide optimum storage savings.

The results obtained indicate about 70 - 80 % storage savings achieved by storing Emulab disk images in Venti deduplicated storage system which is consistent with our thesis hypothesis.

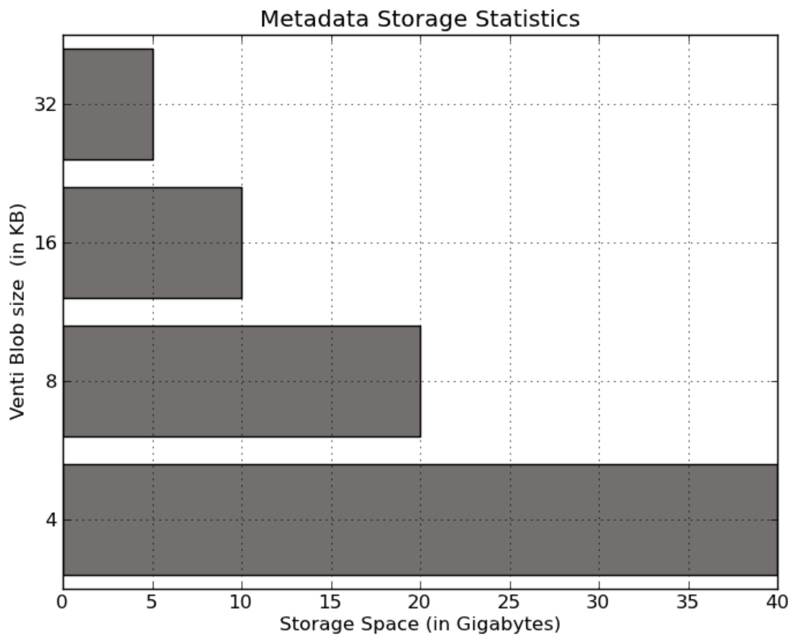

**Figure 5.5**: Metadata storage consumption for various Venti blob sizes.

## 5.4 Performance Impact

As discussed in section 2.1.3, the performance of disk loading is critical to the overall response time of an Emulab experiment instantiation. Due to the modifications we made to the disk loading, chunk reconstruction is now a critical step in the process of disk loading. In other words, the Frisbee server has to reconstruct the requested chunk before sending it to the clients. The performance impact induced can be quantified by conducting image reconstruction and end-to-end Frisbee deployment experiments.

### 5.4.1 Image Reconstruction

Figure 5.7 presents the time duration for reconstructing a full disk image from a backend deduplicated storage system. Blob size denotes the data block sizes used for storing disk images in the deduplicated storage system. As one might observe, the time to reconstruct an image varies with the blob size used for storing the disk image. This is reasonable since reconstructing a disk image that is stored in blocks of 4 KB would mean retrieving eight times the number of data blocks to be retrieved for a blob size of 32 KB. However, no sharp decline in the performance is observed due to the following reason:

1. Though using a blob size of 4 KB instead of 16 KB would mean retrieving four times the number of blobs, it is not required to retrieve as many blobs due to the presence

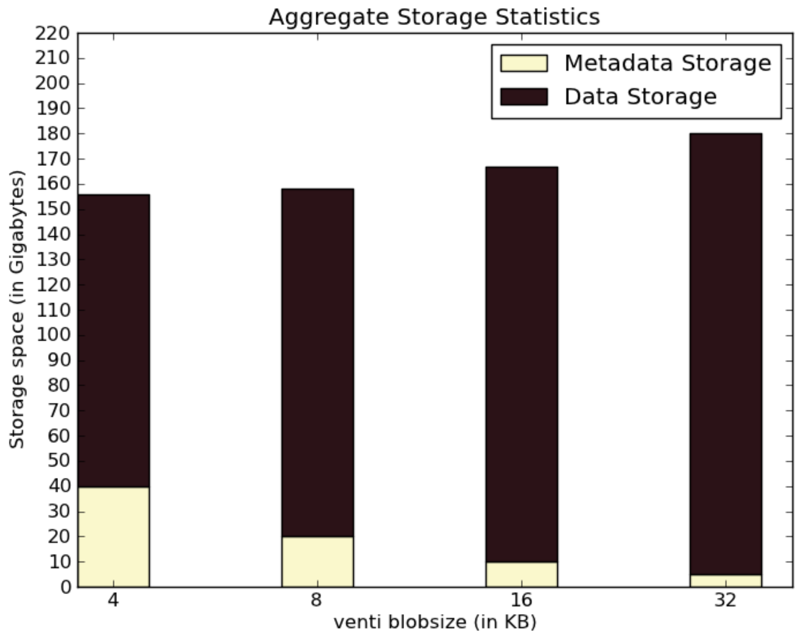

**Figure 5.6**: Aggregate Storage Statistics

of duplicate blobs. Every blob retrieved from Venti is cached in a block cache and all the subsequent accesses to the blob are served from the block cache. This alleviates the performance penalty.

### 5.4.2 End-to-end Frisbee Performance

End-to-end Frisbee performance testing helps us determine how the system works in a production environment. The tests are carried out for various blob sizes and a varying number of clients. Figure 5.8 and Figure 5.9 present the disk image deployment time statistics for serving a disk image from a back end deduplication storage system with blob size 32 KB and 16 KB, respectively. These results are compared to the performance of baseline Frisbee with regard to deploying the same disk image. Figure 5.10 presents the disk deployment comparison statistics for baseline Frisbee and Frisbee serving images from Venti storage system with blobsizes 16 KB and 32 KB. The performance overhead generated when disk images are served from Venti varies about 25% when compared to the baseline Frisbee, which induces a latency of a few seconds while deploying disk image to the clients. Also noticeable is the way the performance overhead remains constant with increase in the number of clients. This phenomena is observed due to the following reasons:

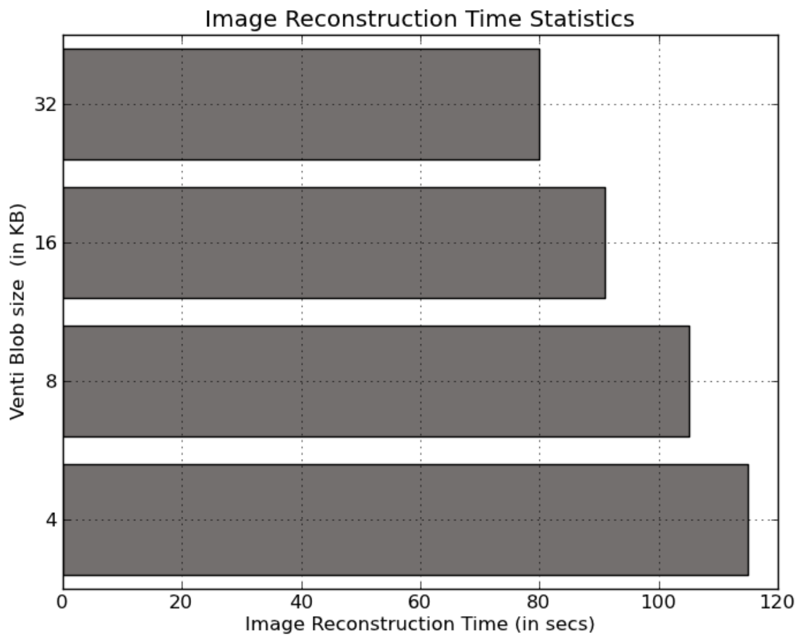

**Figure 5.7**: Image Reconstruction Time

- Pipelining the process of chunk reconstruction with the highly parallel nature of Frisbee clients.
- There are other portions of the disk loading process that are much slower than the process of chunk reconstruction. Network transmission delay and client side disk I/O are slower operations since the network bandwidth is shared and the clients typically have slow disks. Hence, the slower operations involved in the disk loading process hide the latency induced by the chunk reconstruction process thus exhibiting only minimal performance impact on the overall performance of disk loading.

The end-to-end Frisbee disk image deployment statistics indicate that our system induces 25% performance degradation, which amounts to a few seconds compared to the baseline system in an experiment consisting of up to 20 clients. We believe achieving 70 - 80 % storage savings while inducing the latency of a few seconds for deploying Emulab disk images is a very reasonable cost to pay.

### 5.4.3 Performance Pipeline

To understand the reason behind low performance impact of Venti on the overall process of a disk image deployment, we need to analyze the individual steps involved in a disk image

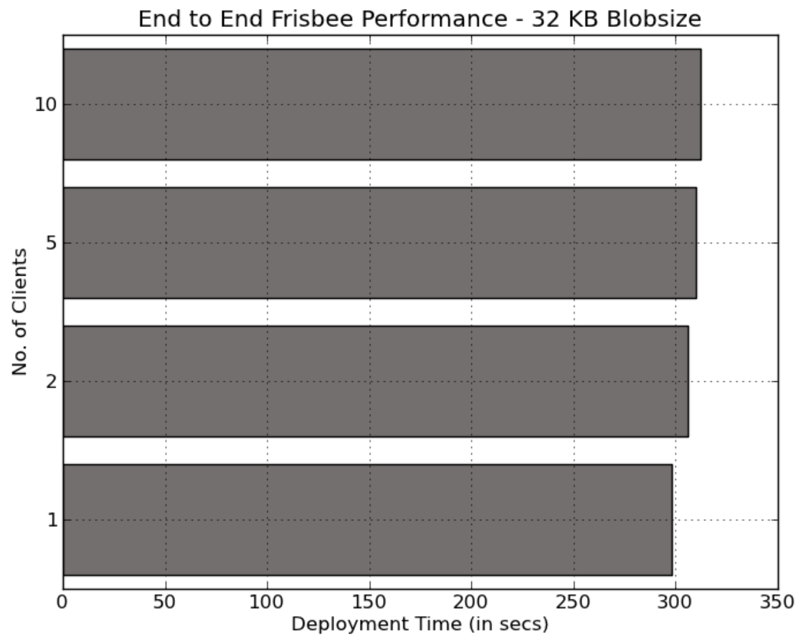

**Figure 5.8**: End-to-End Frisbee Test — 32 KB Blob size

deployment.

Figure 5.11 presents the timeline of deploying arbitrary chunks during disk image deployment. The figure compares the timeline for baseline Frisbee and Frisbee with the capability to serve disk images from Venti. The metrics used in Figure 5.11 and their meanings are as follows.

- IDLE TIME - Indicates the timestamp with reference to unix real time clock when a chunk deployment starts.
- RECONS TIME - Time taken to reconstruct a chunk from Venti. This time is zero in case of normal Frisbee deployment.
- PROP DELAY - Propogation delay induced by the network.
- CLIENT DELAY - Latency induced by the client before starting disk I/O.
- CLIENT IO - Time taken to uncompress the chunk and write it to the client disk.

Observation of Figure 5.11 leads to the following conclusions.

- The only noticeable difference between the normal Frisbee and Frisbee with Venti integration is the latency induced by chunk reconstruction i.e. RECONS TIME.

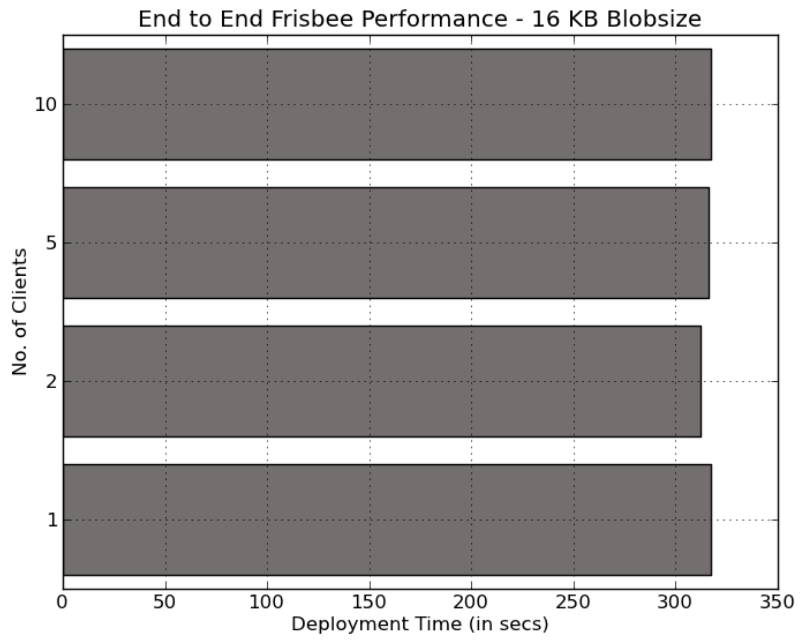

**Figure 5.9**: End-to-End Frisbee Test — 16 KB Blob size

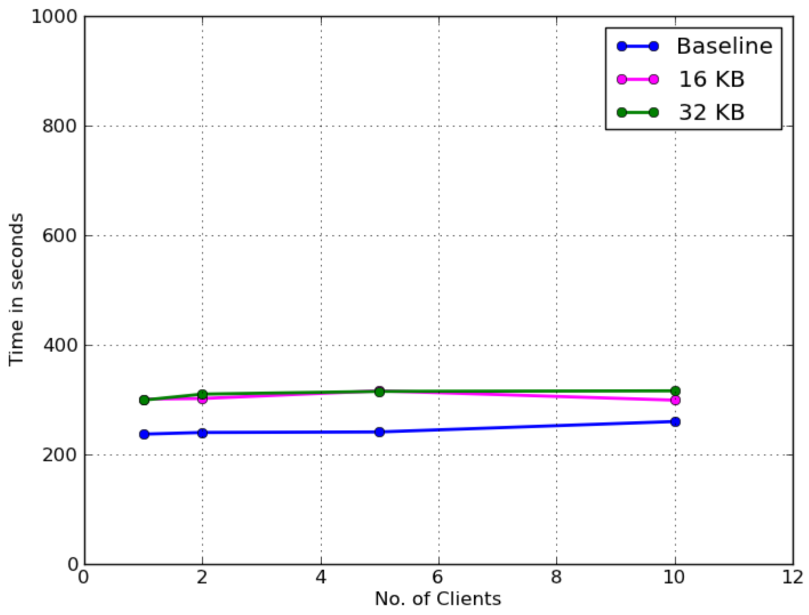

**Figure 5.10**: End-to-End Frisbee Disk Image Deployment Comparison

The results clearly indicate that during disk image deployment, the process of chunk reconstruction can be overlapped with client side delay thus creating only a minimal impact on the overall performance.

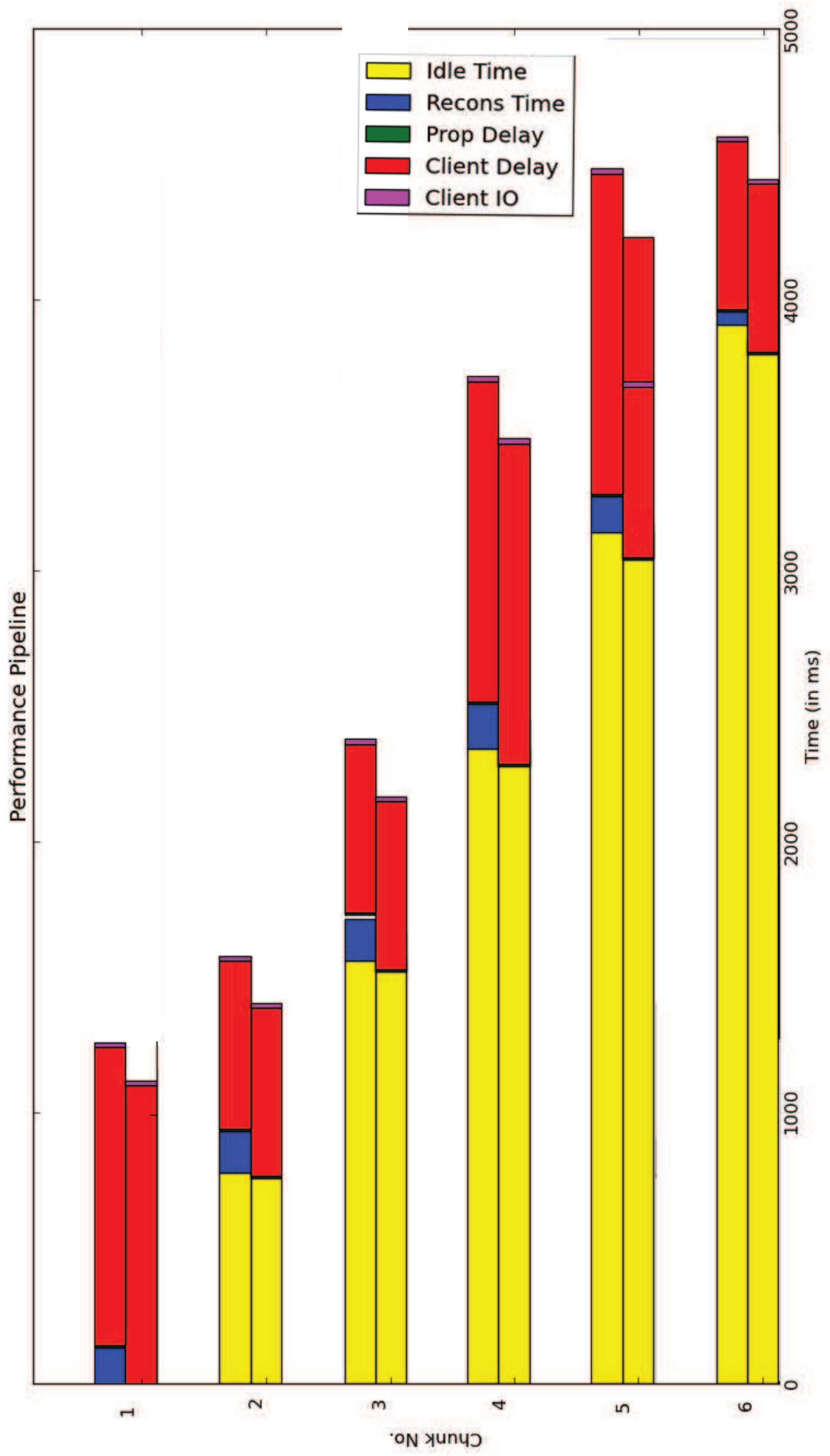

**Figure 5.11**: Performance Pipeline — Frisbee Disk Image Deployment

# CHAPTER 6

# RELATED WORK

We classify the related work into three categories: (a) Applications of data deduplication that experiment and quantify its benefits, (b) systems provisioning resources, such as virtual machines in a scalable manner, (c) miscellaneous.

## 6.1 Data Deduplication

### 6.1.1 Options

Data deduplication is very popular in the industry. Many commercial software such as NetApp WAFL [29] and Acronis [12] support deduplication. These systems are used in every day production environments which underscores the benefits achieved and utilized by commercial entities. Moreover, there are very interesting studies in the research community about the benefits of data deduplication in storing diverse content.

### 6.1.2 Virtual Machine Data

Jin and Miller [24] talked about the effectiveness of deduplication on virtual machine disk images. Their work only discusses storage and not distribution of the disk images. As observed by them, the use of data deduplication can save 80% or more storage space and also disk images corresponding to a single operating system lineage deduplicate well. However, our work, while corroborating the effectiveness of data deduplication on our physical disk image data set, integrates the benefits achieved by the use of data deduplication into the realtime environment of a live network testbed and demonstrates that performance need not be compromised for storing massive data using data deduplication techniques.

### 6.1.3 Practical Deduplication

Meyer and Bolosky [31] contrast the differences between block based deduplication and whole file deduplication over a large data set of user data. The data set used for their experiment is live system data collected from 850 Windows desktop computers that are in everyday use. As they observed, their whole file deduplication delivered a close performance at 72 % storage savings with block based deduplication, which achieved 83 %

storage savings. They, however, claim that the performance impact induced by block based deduplication over whole file deduplication convinced them that whole file deduplication is a better alternative.

However, our experiments with block based deduplication, in contrast to claims in the work mentioned previously, show that you can in fact get very good performance by storing data at block level while achieving significant storage savings at the same time. These contrasting conclusions, presumably, could be attributed to the nature of the data being stored. The data set in the work mentioned earlier, primarily consists of arbitrary files with no defined structure. While our data set comprises of mostly disk images that have a defined structure. The structured nature of our data set provides us the flexibility to perform certain optimizations while storing the data. One such optimization is to store only those blocks that seem to contain valid data while discarding the remaining blocks.

### 6.1.4 Performance

Though we demonstrated that storing and serving disk images in a deduplication storage system does not impact the overall performance of an Emulab experiment instantiation, performance of our system is important as Emulab experiments grow in size and scale. Currently, our system works effectively with 20 clients without creating a significant impact on the performance. However, a 20-fold increase in the number of clients might stress the backend deduplication storage system from which the data is served. Hence, it is imperative to come up with solutions to scale our system. We could leverage already existing systems in conjunction with our system to achieve the scalability we desire. We introduce a few such systems, illustrate their interaction with our system and explain how they complement each other.

Lukkien [27] presents memventi, an alternative implementation of the Venti data deduplication system protocol which utilizes part of main memory for fast lookups. However, this implementation does not scale for massive data. An improved version of memventi, that scales over massive data will be very useful in boosting the performance of our backend deduplicated storage system.

Memcached [20] is a highly scalable distributed caching system. Memcached is known for its ease of scalability. Memcached caches data items up to a size of 1 MB and identifies a data item using a unique id. Memcached can help reduce the number of disk I/O operations performed in our current Venti configuration. Caching chunks of frequently accessed disk images in a distributed memcached based caching system ensures instant access to the desired chunks without the need of reconstructing them on the fly. By means of parallelising reconstructing and caching chunks, we could attain near baseline Frisbee performance for deploying a disk image.

Chunkstash [17] builds an inline data deduplication system based on flash memory and claims to outperform traditional deduplication systems based on raw disks by 7x-60x in speed and also reducing RAM usage by 90% while offering similar data deduplication rates as traditional approaches. It organizes metadata on flash memory and uses an in-memory hash structure to index them thus reducing the dependence on RAM. This work motivates the need for our system to use newer technologies such as flash based memory to achieve good data deduplication rates while achieving better performance which could help our system scale much better.

## 6.2 Provisioning

Provisioning resources is a fundamental activity of cloud computing environments. Popular cloud computing infrastructures such as Amazon EC2 [1] deploy standard disk images [2] to the target nodes as part of provisioning a resource. However, addressing storage management of the disk images helps reduce storage consumption costs in these cloud computing environments. Our work essentially demonstrates the effects of provisioning resources in a distributed testbed environment utilizing data deduplication techniques in the backend storage system. We believe similar optimizations could be performed in a cloud computing environment in order to realize the benefits of data deduplication.

### 6.2.1 Snow Flock

Snow Flock [26] is a rapid virtual machine cloning framework with a focus on fast creation and termination of a virtual machine environment. Snow Flock delivers parallel execution for resource-intensive applications by running them on virtual machine clones in parallel. When tied with a similar infrastructure as ours, Snow Flock can be even more powerful with the capability to clone a large variety of virtual machine configurations. Our work is complementary to Snow Flock. The use of a backend deduplicated storage system helps Snow Flock clone virtual machines with diverse configurations thus providing the users with more powerful set of features.

### 6.2.2 VMPlants

VMPlants [25] deals with managing user created virtual machine execution environments that are cloned for use at a later point of time. This is precisely the use case we explored in our work although for physical machine disk configurations. VMPlants is complementary to our work. VMPlants could be combined with a backend deduplicated storage system, like in our experiment, to store ad hoc virtual machine execution environments thus providing storage savings and the capability to store a much wider variety of virtual machine execution environments. This property ultimately makes VMPlants more powerful and capable of providing and cloning from a more diverse set of virtual machines.

# CHAPTER 7

# CONCLUSION AND FUTURE WORK

We presented the design and implementation of a large scale disk image storage and deployment system for the Emulab network testbed. We explored techniques to achieve optimal utilization of available storage resources while addressing the issues involved in making the system have minimal impact on the overall performance of an Emulab experiment swap-in.

Our work leveraged the Venti archival storage system for applying data deduplication techniques and achieving significant storage savings. Towards achieving storage savings with a minimal impact on the disk loading performace, we explored several techniques:

- Optimum techniques for achieving maximum storage savings while storing Emulab disk images.
- Application of file system knowledge to further trim storage utilization.
- Reconstructing portions of a disk image, known as chunks, on-the-fly during disk image deployment.
- Utilizing zlib knowledge to reconstruct the compressed part of a chunk with minimal information about the disk data layout.
- Pipelining the process of chunk reconstruction with deployment to achieve bounded disk image deployment latencies.

Finally, we implemented our system to remain unobtrusive to other components of Emulab especially the Frisbee protocol. In other words, our system does not require changing the Frisbee protocol and is thus able to work out-of-the-box with already existing Frisbee clients. However, the Frisbee clients now have an opportunity to request a disk image from a massive collection of eclectic disk images thus encouraging large scale heterogeneous experimentation within Emulab.

Several new ideas could be explored to further extend this system and make it more scalable and robust. A few of them are described.

- Exploring novel storage techniques such as flash storage and utilizing it as backend storage. Implementing better data deduplication techniques for such types of storage.
- Scaling Frisbee protocol with ten times the number of clients and observe the effects it has on our system and use the observations to improve our system.
- Analyze user disk image request patterns and provide suggestions to rearrange disk image data. In other words, decide which disk images need to be cached or stored in a faster memory and which disk images need to be stored in the slower backend storage devices.

# REFERENCES

[1] Amazon Ec2. http://aws.amazon.com/ec2/.

[2] Amazon Machine Image. http://aws.amazon.com/amis.

[3] Linux Volume Manager Manual. http://tldp.org/HOWTO/LVM-HOWTO/.

[4] Meeting The Challenges Of Disaster Recovery. http://m.softchoice.com/files/pdf/brands/acronis/Acronis_Meeting_Challenge_of_Disaster_Recovery_White_Paper.pdf.

[5] Plan 9 From Bell Labs. Fossil Manual Page. Fossil(4). http://man.cat-v.org/plan_9/4/fossil.

[6] Plan 9 From Bell Labs. Vac Manual Page. Vac(1). http://swtch.com/plan9port/man/man1/vac.html.

[7] Plan 9 From Bell Labs. Vacfs Manual Page. Vacfs(4). http://swtch.com/plan9port/man/man4/vacfs.html.

[8] Plan 9 From User Space. Venti Administration Guide. Venti(8). http://swtch.com/plan9port/man/man8/venti.html.

[9] Plan 9 From User Space. Venti Overview Manual Page. Venti(7). http://swtch.com/plan9port/man/man7/venti.html.

[10] Plan9 Gnu/Linux Port. http://www.swtch.com/plan9port/.

[11] Raid5. http://tldp.org/HOWTO/Software-RAID-HOWTO.html.

[12] Storage Savings With Acronis Deduplication. http://m.softchoice.com/files/pdf/brands/acronis/Storage_Savings_with_Acronis_Deduplication_white_paper.pdf.

[13] Symantec Ghost Disk Imaging Software. http://www.symantec.com/sabu/ghost/.

[14] Veritas Volume Replicator. http://eval.symantec.com/mktginfo/products/White_Papers/Storage_Server_Management/sf_vvr_wp.pdf.

[15] Chun, B., Culler, D., Roscoe, T., Bavier, A., Peterson, L., Wawrzoniak, M., and Bowman, M. Planetlab: An Overlay Testbed For Broad-coverage Services. *ACM SIGCOMM Computer Communication Review 33*, 3 (2003), 3–12.

[16] Crespo, A., and Garcia-Molina, H. Archival Storage For Digital Libraries. In *Proceedings of the Third ACM Conference on Digital Libraries* (1998), ACM, pp. 69–78.

[17] Debnath, B., Sengupta, S., and Li, J. Chunkstash: Speeding Up Inline Storage Deduplication Using Flash Memory. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (2010), USENIX Association, pp. 16–16.

[18] Deutsch, L., and Gailly, J. Zlib Compressed Data Format Specification. *rfc1950 etc.* (1996).

[19] Deutsch, P., and Gailly, J.-L. Zlib compressed data format specification version 3.3, 1996.

[20] Fitzpatrick, B. Distributed Caching With Memcached. *Linux Journal 2004*, 124 (2004), 5.

[21] Gailly, J., and Adler, M. Zlib Home Page. http://zlib.net.

[22] Henderson, T., Lacage, M., Riley, G., Dowell, C., and Kopena, J. Network Simulations With The ns-3 Simulator. *SIGCOMM Demonstration* (2008).

[23] Hibler, M., Stoller, L., Lepreau, J., Ricci, R., and Barb, C. Fast, Scalable Disk Imaging With Frisbee. In *Proc. of the 2003 USENIX Annual Technical Conf.* (San Antonio, TX, June 2003), USENIX Association, pp. 283–296.

[24] Jin, K., and Miller, E. The Effectiveness Of Deduplication On Virtual Machine Disk Images. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference* (2009), ACM, p. 7.

[25] Krsul, I., Ganguly, A., Zhang, J., Fortes, J., and Figueiredo, R. Vmplants: Providing And Managing Virtual Machine Execution Environments For Grid Computing. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing* (2004), IEEE Computer Society, p. 7.

[26] Lagar-Cavilla, H., Whitney, J., Scannell, A., Patchin, P., Rumble, S., De Lara, E., Brudno, M., and Satyanarayanan, M. Snowflock: Rapid Virtual Machine Cloning For Cloud Computing. In *Proceedings of the 4th ACM European Conference on Computer Systems* (2009), ACM, pp. 1–12.

[27] Lukkien, M. Venti Analysis And Memventi Implementation. Tech. rep., Technical Report 694, University of Twente, 2007.

[28] Mahrenholz, D., and Ivanov, S. Real-time Network Emulation With ns-2. *DS-RT* (2004), 29–36.

[29] Marks, H. Data De-Dupe Hits Storage. *Network Computing 18*, 10 (2007), 21–21.

[30] McCanne, S., Floyd, S., Fall, K., Varadhan, K., et al. Network Simulator ns-2, 1997.

[31] Meyer, D., and Bolosky, W. A Study Of Practical Deduplication. In *FAST'11: Proceedings of the 9th Conference on File and Storage Technologies* (2011).

[32] Muthitacharoen, A., Chen, B., and Mazieres, D. A Low-bandwidth Network File System. *ACM SIGOPS Operating Systems Review 35*, 5 (2001), 174–187.

[33] Pike, R., Presotto, D., Thompson, K., and Trickey, H. Plan 9 From Bell Labs. In *Proceedings of the Summer 1990 UKUUG Conference* (1990), Citeseer, pp. 1–9.

[34] Quinlan, S., and Dorward, S. Venti: A New Approach To Archival Storage. In *Proceedings Of The Conference On File And Storage Technologies* (2002), pp. 89–101.

[35] Rakotoarivelo, T., Ott, M., Jourjon, G., and Seskar, I. Omf: A Control And Management Framework For Networking Testbeds. *ACM SIGOPS Operating Systems Review 43*, 4 (2010), 54–59.

[36] Ricci, R. *Enhancing Realism And Scalability In Network Testbeds.* PhD thesis, Citeseer, 2010.

[37] Ricci, R., Alfeld, C., and Lepreau, J. A Solver For The Network Testbed Mapping Problem. *ACM SIGCOMM Computer Communication Review 33*, 2 (2003), 65–81.

[38] Rizzo, L. Dummynet: A Simple Approach To The Evaluation Of Network Protocols. *ACM SIGCOMM Computer Communication Review 27*, 1 (1997), 31–41.

[39] Segall, B., and Arnold, D. Elvin Has Left The Building: A Publish/Subscribe Notification Service With Quenching. In *Proceedings of AUUG97* (1997), Brisbane, Australia, pp. 3–5.

[40] White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. An Integrated Experimental Environment For Distributed Systems And Networks. *OSDI02 0*, 0 (Dec 2002), 255–270.

[41] You, L. L., Pollack, K. T., and Long, D. D. E. Deep store: An archival storage system architecture. In *Proceedings of the 21st International Conference on Data Engineering* (Washington, DC, USA, 2005), ICDE '05, IEEE Computer Society, pp. 804–8015.