USENIX Association

Proceedings of the General Track: 2003 USENIX Annual Technical Conference

San Antonio, Texas, USA
June 9-14, 2003

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Fast, Scalable Disk Imaging with Frisbee

Mike Hibler   Leigh Stoller   Jay Lepreau   Robert Ricci   Chad Barb
School of Computing
University of Utah

Abstract

Both researchers and operators of production systems are frequently faced with the need to manipulate entire disk images. Convenient and fast tools for saving, transferring, and installing entire disk images make disaster recovery, operating system installation, and many other tasks significantly easier. In a research environment, making such tools available to users greatly encourages experimentation.

We present Frisbee, a system for saving, transferring, and installing entire disk images, whose goals are speed and scalability in a LAN environment. Among the techniques Frisbee uses are an appropriately-adapted method of filesystem-aware compression, a custom application-level reliable multicast protocol, and flexible application-level framing. This design results in a system which can rapidly and reliably distribute a disk image to many clients simultaneously. For example, Frisbee can write a total of 50 gigabytes of data to 80 disks in 34 seconds on commodity PC hardware. We describe Frisbee's design and implementation, review important design decisions, and evaluate its performance.

1 Introduction

While most computer use focuses on creating, storing and moving single files, many application domains exist where efficiently handling operations on entire disks is important. Classic system administration tasks, such as operating system installation, catastrophe recovery, and forensics, as well as new research such as mobile work environments [18] and computing utility farms [9, 14], benefit greatly from the ability to quickly read, transfer, and write entire disk partitions.

There are two basic disk-level distribution strategies. Differential update, represented by tools such as `rsync` [17] operates above the filesystem, and compares what is already on the disk with the desired contents, replacing only what is necessary. Disk imaging, used by tools such as Ghost [7], operates below the filesystem, unconditionally replacing the contents of a disk.

Differential techniques are very effective at synchronizing file hierarchies within a filesystem and are extremely bandwidth efficient. However, for distributing entire disks, disk imaging offers a number of important advantages:

Generality: Creating a disk image requires no knowledge of the filesystem being imaged. We show how limited knowledge can be beneficial in Section 3.1, but is not required. Synchronization above the filesystem, however, requires detailed understanding of the filesystem such as directory structure, file ownership, access controls, and times.

Robustness: Disk images have no dependence on the existing contents of the target disk; in contrast, file-based synchronization tools cannot, for example, be used on a corrupted filesystem.

Versatility: One filesystem type can easily be replaced with another using a disk image. This cannot be done with a file-based synchronizer.

Speed: Writing an entire disk image can be faster than determining which files need to be updated. Section 5.4 demonstrates that Frisbee runs much faster than `rsync` in our target environment.

Full-disk imaging does have drawbacks. It is less bandwidth-efficient than differential techniques—no matter how small the difference between the source and destination, the entire disk is copied. The client will also most likely have to be entirely dedicated to the task, since instead of updating files above the filesystem layer, raw disk contents are changed. The advantages outweigh the drawbacks, however, and we demonstrate that by taking advantage of characteristics of the target domain, disk imaging can used to implement an efficient and scalable disk loading system.

Adopting the strategy of complete disk reloading due to our application's requirements, we have designed, implemented and evolved Frisbee, a disk image generation and distribution system that is fast and highly scalable in a LAN environment. While designed for our network emulation testbed, Frisbee offers functionality and techniques useful in a variety of production and research environments.

Five aspects of Frisbee's design are key to its success:

This work was largely sponsored by NSF grants ANI-0082493 and ANI-0205702, Cisco Systems, and DARPA grant F30602-99-1-0503.

Author information: {mike,stoller,lepreau,ricci,barb}@cs.utah.edu, School of Computing, 50 S. Central Campus Drive, Rm. 3190, University of Utah, Salt Lake City, UT 84112–9205.
www.flux.utah.edu   www.emulab.net   www.netbed.org

- Domain-specific data compression. Although Frisbee's overall approach treats disk contents as opaque, we relax this requirement for several common filesystem types, building in enough knowledge of their formats to enable identification of free blocks and entirely avoid distributing or writing them.
- Two-level segmenting of the data, into randomly-accessed long segments composed of small blocks typically accessed sequentially. This approach meets the many competing needs of disk I/O, block decompression, network transmission, selective retransmission, and relative simplicity.
- A custom receiver-driven reliable multicast protocol, optimized for the LAN environment.
- Careful concurrency control in the receiver, using three-way multithreading to fully overlap I/O with decompression.
- Designing for this domain's particular pattern of resource availability: target machines entirely dedicated to the diskloading task, a shared server, and a secure high-bandwidth local network.

This paper makes the following contributions: (1) It shows that bulk disk imaging can be extremely fast and scalable, making it a practical approach to disk loading, and frequently a superior one. Furthermore, our performance results indicate that disk imaging is so fast that it can be applied in qualitatively new ways. (2) It presents the detailed design, implementation, and experimental evaluation of Frisbee and identifies the design aspects most important to its performance. (3) The disk imaging system it describes is, to our knowledge, the fastest such system extant. In addition, versions of the system have been proven in production use for over 18 months by hundreds of external users, and is available in open source form. (4) It discusses the design tradeoffs in disk imaging systems.

In the rest of this paper we first outline Netbed, the network testbed system that drove our need for disk imaging. We then outline the design tradeoffs in a disk imaging system, following with sections on Frisbee's design and implementation, performance evaluation and analysis, related and future work, and conclusion.

2 The Netbed Context

As Frisbee was developed primarily for use in the Emulab portion of Netbed, some background will be useful in understanding its motivation and target environment. Netbed [20] is a time- and space-shared facility for networking and distributed systems research and education. It has been in use by the community since April 2000. Emulab is one of Netbed's primary hardware resources—it consists of a cluster of commodity PC "nodes" with configurable network interconnectivity. Emulab is space-shared in the sense that it can be arbitrarily partitioned for use by multiple experimenters at the same time. Some resources in the system, such as nodes, can only be used in one experiment at a time; in this sense, Emulab is also time-shared. Experimenters are generally given full root access to nodes, meaning that they are free to reconfigure the host operating system in any way they wish, or even install their own.

In light of its time-shared nature, and the degree of control experimenters are given, it is critical that Emulab nodes be returned to a known state between experiments. Without this, experimenters have no guarantee that their results are not affected by configuration changes made by the previous user. Even worse, a malicious user could modify disk contents to facilitate compromise of the node once it has been allocated to another experimenter. To ensure a node is in a known state, its disk must be entirely reloaded.

Disk contents on Emulab are considered to be soft state—hard state, such as user accounts and information about network configuration, are kept off-node in a central database. This separation simplifies things by allowing the same disk image to be used on many nodes, with each node self-configuring from the central database at boot time. It also relieves users of the need to preserve configuration data. Thus, if an experimenter corrupts disk contents, say by introducing a kernel change that corrupts a filesystem, the disk can simply be reloaded, preserving any hard state and losing only soft state. This forgiving environment encourages aggressive experimentation.

Disk images can also be loaded automatically at experiment creation time. An experimenter who wishes to install their own custom operating system or make substantial changes to the default FreeBSD or Linux images provided by Emulab can create an image containing their customizations. They can then load this image on an arbitrary number of other nodes without manual intervention.

Since Emulab has a large number of nodes (currently, 170), and users have run experiments that use more than 120 nodes, speed and scaling are critical to enable these new uses of disk imaging. Waiting for scores of nodes to load serially would leave them unavailable for a long period of time, dramatically reducing the throughput of Emulab, so image distribution must be done in parallel. There are, however, distinct classes of nodes of differing speeds, so it is important that our disk imaging solution work well with heterogeneous clients.

3 Design Tradeoffs

There are three phases of a disk imaging system: image creation, image distribution, and image installation. Each phase has aspects which must be balanced to fulfill a desired goal. We consider each phase in turn.

3.1 Image Creation

In image creation, the goal is to create a consistent snapshot of a disk or partition in the most efficient way possible.

Source availability: While it is possible for the source of the snapshot to be active during the image creation process, it is more common that it be quiescent to ensure consistency. Quiescence may be achieved either by using a separate partition or disk for the image source or by running the image creation tool in a standalone environment which doesn't use the source partition. Whatever the technique, the time that the image source is "offline" may be a consideration. For example, an image creation tool which compresses the data as it reads it from the disk may take much longer than one that just reads the raw data and compresses later. However, the former will require much less space to store the initial image.

Degree of compression and data segmentation: Another factor is how much (if any) and what kind of compression is used when creating the image. While compression would seem to be an obvious optimization, there are trade-offs. As mentioned, the time and CPU resources required to create an image are greater when compressing. Compression also impacts the distribution and decompression process. If a disk image is compressed as a single unit and even a single byte is lost during distribution, the decompression process will stall until the byte is acquired successfully. Thus, depending on the distribution medium, images may need to be broken into smaller pieces, each of which is compressed independently. This can make image distribution more robust and image installation more efficient at the expense of sub-optimal compression.

Filesystem-aware compression: A stated advantage of disk imaging over techniques that operate at the file level is that imaging requires no knowledge of the contents or semantics of the data being imaged. This matches well with typical file compression tools and algorithms which are likewise ignorant of the data being compressed. However, most disk images contain filesystems and most filesystems have a large amount of available (free) space in them, space that will dutifully be compressed even though the contents are irrelevant. Thus, the trade-off for being able to handle any content is wasted time and space creating the image and wasted time decompressing the image. One common workaround is to zero all the free space in filesystems on the disk prior to imaging, for example, by creating and then deleting a large file full of zeros. This at least ensures maximum compressibility of the free space. A better solution is to perform *filesystem-aware* compression. A filesystem-aware compression tool understands the layout of a disk, identifying filesystems and differentiating the important, allocated blocks from the unimportant, free blocks. The allocated blocks are compressed while the free blocks are skipped. Of course, a disk imaging tool using filesystem-aware compression requires even more intimate knowledge of a filesystem than a file-level tool, but the imaging tool need not understand all filesystems it may encounter– it can always fall back on naive compression.

3.2 Image Distribution

Image distribution is concerned with getting a disk image from a "server" to one or more "clients." In our context it is assumed that the server and clients are different machines and not just different disks on the same machine. Furthermore, we restrict the discussion to distribution over a network.

Network bandwidth and latency: Perhaps the most important aspect of network distribution is bandwidth utilization. The availability of bandwidth affects how images are created (the degree of compression) as well as how many clients can be supported by a server (scaling). Bandwidth requirements are reduced significantly by using compression. Increased compression not only reduces the amount of data that needs to be transferred, it also slows the consumption rate of the client due to the need to decompress the data before writing it to disk. If image distribution is serialized, only one client at a time, then compression alone may be sufficient to achieve a target bandwidth. However, if the goal is to distribute an image to multiple clients simultaneously, then typical unicast protocols will need to be replaced with broadcast or multicast. Broadcast works well in environments where all clients in the broadcast domain are involved in the image distribution. If the network is shared, then multicast is more appropriate, ensuring that unaffiliated machines are not affected. Just as in all data transfer protocols, the delay-bandwidth product affects how much data needs to be en route in order to keep clients busy, and the bandwidth and latency influence the granularity of the error recovery protocol.

Network reliability: As alluded to earlier, the error rate of the network may affect how compression is performed. Smaller compression units may limit the effectiveness of the compression, but increase the ability of clients to remain busy in the face of lost packets. More generally, in lossy networks it is desirable to subdivide an image into "chunks" and include with each chunk additional information to make that chunk self-describing. In a highly reliable network, or if using a reliable transport protocol that provides in-order delivery beneath the image distribution protocol, this additional overhead would be unnecessary.

Network security: If the distribution network is not "secure," additional measures will need to be taken to ensure the integrity and privacy of image data. If the image contains sensitive data, then encryption can be used to protect it. This encryption can be done either in the network transport using, for example, SSL, or the image itself could be encrypted as part of the creation process. The latter requires more CPU resources when creating the image but provides privacy of the stored image and is compatible with existing multicast protocols. Ensuring that the image is not corrupted during distribution due to injection of forged data into the communication channel is also an issue. This requires that clients authenticate the source of the image. Again, many solutions exist in the unicast space, such as using an SSH tunnel to distribute images. For multicast, the problem is harder and the focus of much research [2]. Note that security is not just a wide-area network concern. Even in a LAN, untrusted parties may be able to snoop or spoof on traffic unless countermeasures are taken. However, in the LAN case, switch technologies such as virtual LANs can provide some or all of the necessary protection.

Receiver vs. sender-driven protocol: A final issue in image distribution is whether the protocol is server or receiver-driven [19]. A simple server-driven protocol might require that all clients synchronize their startup and operate in lock-step as the server doles out pieces of an image as it sees fit. Such a strategy would scale well in a highly reliable network with homogeneous client machines as little extraneous communication is required. However, if a client does miss a piece of the image for any reason, it might be forced to abort or wait until the entire image has been sent out and then request a resend. A client-driven protocol allows each client to join the distribution process at any time, requesting the chunks it needs to complete its copy, and then leaving. The process completes when all clients have left. The downside is more control traffic and the potential for significant redundant data transfers, either of which can affect scaling.

3.3 Image Installation

The final element of disk imaging is the installation of the transferred image on a client. As with image creation, the disk or partition involved must be quiescent with the image installer either operating on a second disk or partition or running standalone. Since image installation is typically concerned with installing or restoring the "primary" disk on a machine, we restrict the remaining discussion to the standalone case.

Resource utilization: In a standalone environment, the disk installation tool is in the enviable position of being able to consume every available local resource on the target machine. For example, it can use hundreds of megabytes of memory for caching image data incoming from the network or for decompressed data waiting to be written to disk. Likewise, it can spawn multiple threads to handle separate tasks and maximize overlap of CPU and I/O operations.

Overlapping computation and I/O: CPU is of particular interest since, on reasonably fast current processors, substantial computing can be performed while waiting for incoming network packets and disk write completions. The most obvious use of the cycles is to decompress data. However, on unreliable transports the time could also be used for computing checksums, CRCs, or forward-error-correction codes. On insecure transports, CPU resources may be needed for decrypting and authenticating incoming data.

Optimizing disk I/O: If disk I/O is the bottleneck when installing an image, the installation tool may be able to exploit client resources or characteristics of the disk image to minimize disk write operations. On machines with large physical memories, memory can be used to buffer disk writes allowing for fewer and larger sequential IO operations. If the image format uses a filesystem-aware compression strategy which distinguishes allocated and free blocks, the installation tool can seek over ranges of free blocks, thereby reducing the number of disk writes. Note that this method has security implications, since it has the potential to "leak" information from the previous disk user to the new user.

Optimizing network I/O: If network bandwidth is the bottleneck then it may be possible to take advantage of similarities between the old disk contents and the desired contents, as is done in the LBFS filesystem [15], designed for the wide-area. In this technique, acquiring a disk image would be a two-phase process. Whenever a client needs a block of data, it would first ask for a unique identifier, such as a collision-resistant hash [5], for that block which it could then compare to blocks on the local disk. If the block already exists on the local disk, it need only be copied to the correct location. Only if the block is not found, would the client request the actual block data. Such hashing techniques can place a heavy burden on the CPU as well as the disk, if local hashes must be computed at run time.

4 Design and Implementation

The previous section outlined a variety of issues and trade-offs in the design of a disk imaging system. In this section we describe our design choices, their rationale, and their mapping to implementation.

4.1 Overview

Creation and compression: To create an image, we first boot the source machine into a special memory file system-based version of Unix. This satisfies the need for disk quiescence, and allows us to create images without porting the image creator to run on all operating systems for which it can create images. Since image creation is much less frequent than image installation, we do not aggressively optimize the time spent creating images. To save on server disk space and bandwidth, the image is compressed on the client before being written to the server. Filesystem/OS-specific compression is used, including skipping swap partitions; generic `zlib`-based [4, 21] compression is used on allocated blocks. Partitions that contain unknown filesystem types are either compressed generically or, optionally, entirely skipped.

Multicast: Our distribution mechanism uses a custom application-level receiver-initiated multicast protocol with NAK-avoidance [10]. In turn, this protocol relies on IP multicast support in the network switches to provide one-to-many delivery at the link level. The number of control messages is kept under control by multicasting client repair requests (NAKs), so that other clients can suppress duplicate requests. In terms of the multicast design space put forth in RFC 2887 [8], our application design requires only scalability and total reliability. We do not require other constraints, in particular ordered data or server knowledge of which clients have received data.

Two-level segmentation: We now address the issue of the granularity of data segments. Since we will need to resend lost multicast packets, yet do not need to preserve ordering, it is clear that we want the client-side decompression routines to process data segments out of order. Therefore, each data segment must be self-describing and stand alone as a decompressable unit. Since compression routines optimize their dictionaries based on the distribution of their input data, they achieve better compression ratios when given longer input to sample. That argues for longer segments.

Since Frisbee's basic job is I/O—copying disks through memory over networks—the classic hardware and OS architecture reasons that favor sequential I/O for its speed and efficiency also favor long segments. However, to preserve network and machine resources, we want our multicast loss recovery algorithm to use selective retransmission, which requires relatively short segments. Finally, we want a small segment size that fits into the Ethernet MTU.

The fundamental problem is that we need to follow the principle of Application Level Framing (ALF) [3], yet have conflicting application requirements. We address these conflicting demands by using a two-level segmentation scheme. The unit of compression is the self-describing 1MB *chunk*, composed of 1024 1KB *blocks*. For the initial network transmission, the server multicasts an entire chunk, capping its rate by pausing every N (currently 16) blocks for a tunable period. Receivers selectively request missing blocks via partial request messages. In this way we achieve long segments that can efficiently be randomly-accessed, composed of small blocks that are typically, but not always, accessed sequentially. Subsets of blocks, as specified in receiver partial request messages, give us a flexible mechanism to request intermediate lengths, without undue complexity.

Specialization for resource availability: Since most modern Ethernet networks are switched, and the dedicated clients do not need bandwidth for other purposes, the main place that bandwidth must be saved is on the server and the server's network link, a situation for which multicast is ideally suited. The protocol is client-driven; this way, a server can be running at all times, but naturally falls idle when no clients are present. Additionally, this client-side control provides a high degree of robustness in the face of client failure and reduces server-side bookkeeping.

Receiver concurrency control: To install an image, we boot into a small, memory filesystem-based Unix system similar to the one used when creating the image. Using multiple threads, our disk loader client program takes care to overlap the computationally expensive decompression with the slow disk I/O. Using filesystem-specific compression turns out to give the biggest performance improvements at this stage—once compression is used to reduce the data transferred on the network, and maximal processor/IO overlap is achieved, the bottleneck in performance becomes disk writes. We thus obtain a huge savings by not having to write unnecessary disk blocks. The ability to write free areas of the disk is still available because, as discussed in Section 4.4, this may be needed for confidentiality reasons.

Security: Since users must register to use Netbed, our threat model does not encompass determined malicious local adversaries. We have not yet needed to ensure security in the face of malicious users. We do expect to provide more security eventually, perhaps by signing image data or with VLAN technology. In Emulab, images are distributed via the "control" network, a single switched virtual LAN connecting all nodes. Thus there is the potential for an experiment to observe data on, or inject data into, a Frisbee multicast stream for another experiment. Our focus is on preventing accidental interference between experiments, in particular ensuring the integrity of image data. We use a simple check of the source IP address of incoming block data, which, although maliciously spoofable, suggests that it comes from the approved host. We do not currently provide an option for ensuring privacy of distributed images. User-created images are protected via filesystem permissions while stored on the Frisbee server's disk.

4.2 Image Creation

In the Frisbee system, the *imagezip* application is responsible for creating images of either entire disks or single partitions. The images are compressed using both conventional and filesystem-aware techniques in a two-stage process. In the optional first phase, the partitions of interest are analyzed to determine if filesystem-aware compression can be done. Partitions are identified either explicitly by command line options or implicitly by reading the partition type field in the DOS partition table (on x86-based systems). Frisbee currently handles BSD FFS, Linux ext2fs and Windows NTFS filesystems as well as BSD and Linux swap partitions. If a partition is recognized, a filesystem-specific module is invoked to scan the filesystem free list and build up a list of free blocks in the partition. If a partition is not recognized, *imagezip* treats all blocks as allocated.

imagezip also has a limited ability to associate "relocation" information with data in a created image. This information allows it to create single partition images that can be loaded onto a disk that has a different partition layout. This facility is needed for filesystem types that contain absolute rather than partition-relative sector numbers. Notable examples of this are FreeBSD disklabels and LILO bootblocks.

In the second phase, the allocated blocks are read sequentially and compressed, producing 1MB chunks. Each chunk has a fixed-sized header with index information identifying the ranges of allocated blocks contained within it. Since the degree of compression is unpredictable, it is impossible to know exactly how much input data is required to fill the remaining space in a chunk. We counter with a simple algorithm that compresses smaller and smaller pieces as the chunk gets close to full; we then pad the chunk out to exactly 1MB. The padding typically runs around 20KB.

In summary, as shown in Figure 1, *imagezip* uses knowledge of filesystem types as well as conventional zlib compression to compress disk images. Images are segmented into self-describing 1MB chunks, each with independently compressed data.

4.3 Image Distribution

A compressed disk image in the Frisbee system is just a regular, albeit potentially very large, file and thus can be distributed in any number of ways, such as via `scp` or NFS, and then installed using the *imageunzip* command line program described in Section 4.4.

In a local area network environment, a more efficient and scalable way of image distribution is to use the Frisbee protocol as implemented in the *frisbeed* server and *frisbee* client. *Frisbeed* accepts request messages from multiple Frisbee clients and uses UDP on IP-multicast to transfer an image. Each *frisbee* client uses the multicast channel to request pieces of the desired image as needed until all pieces have been received, decompressed and written to the target disk. In the current implementation, each instance of *frisbeed* serves up a specific disk image using a unique multicast address. The information about what disk image and multicast address to use is communicated out-of-band to the server and clients. In Emulab's case, the client learns this from the central Netbed database.

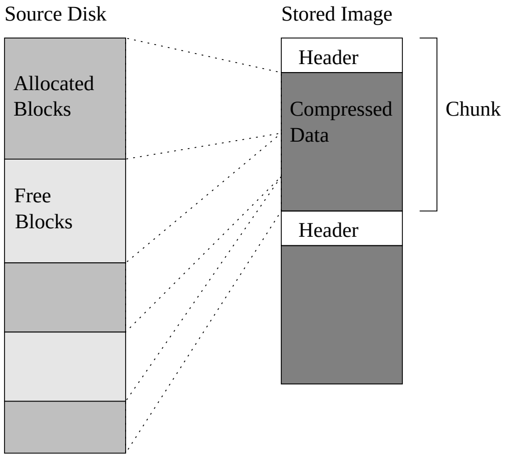

Figure 1: Image creation with *imagezip*

4.3.1 The *frisbeed* Server

The *frisbeed* server has two threads, one which receives incoming requests and one which processes those requests and multicasts image data to the network. The server receive thread fields three types of messages from clients. JOIN and LEAVE messages bracket a client's participation in a multicast session. The server's response to a JOIN message includes the number of blocks in the image.

Clients issue data REQUEST messages containing a chunk number and a block range; typically they request the entire chunk. The server receive thread places block requests on a FIFO work queue, after first merging with any already queued request that overlaps the requested data range.

The *frisbeed* transmit thread loops, pulling requests from the work queue, reading the indicated data from the compressed image file, and multicasting it to the network in Frisbee BLOCK messages. BLOCK messages contain a single 1KB block of data along with identifying chunk and block numbers. Since a request allows for multiple blocks to be specified, a single request from the work queue may generate multiple BLOCK messages.

In our current production system, the server's network bandwidth consumption is controlled by placing a simple cap on the maximum bandwidth used. Two parameters are used to implement the cap: a burst size and a burst gap. The burst size is the number of BLOCK messages that can be transmitted consecutively without pausing, while the burst gap is the duration of that pause. Ideally, just an inter-packet delay could be used to pace data to the network, but the resolution of UNIX sleep mechanisms is dictated by the resolution of the scheduling clock, which is typically too coarse (1-10ms). Our current values of burst size (16) and gap (2ms) were empirically tuned for our environment. Clearly, this capping mechanism is adequate only on a dedicated server machine in a switched LAN environment, as the server does not adjust its transmission rate in response to network load. The effect of this is shown, and an alternative mechanism discussed, in Section 5.3.

4.3.2 The *frisbee* Client

The *frisbee* client is structured as three threads in order to overlap network I/O, disk I/O, and decompression. The network thread, whose basic operation is shown in Figure 2, is responsible for retrieving BLOCK messages multicast by the server, accumulating the contained data blocks into complete chunks, and queuing those chunks for processing by the decompression thread. The network thread also ensures that data arrives in a timely fashion by issuing REQUEST messages for needed chunks and blocks. The decompression thread dequeues completed chunks, decompresses the data and, using the index information from the chunk header, queues variable-sized disk write requests. The disk thread dequeues those requests and performs the actual disk write operations. Once all chunks have been written to disk, the client exits. The remainder of this section focuses on the acquisition of data via the Frisbee protocol.

A *frisbee* client will of course receive not only blocks it has explicitly requested, but those that other clients have requested as well. Ideally, *frisbee* would be able to save all such blocks. However, since blocks for a given chunk must be kept until the entire 1MB chunk has been received, and a compressed image may be hundreds to thousands of megabytes, this is not practical. Thus, *frisbee* maintains a cache of chunks for which it has received one or more blocks, discarding incoming data for other blocks when the cache is full. Currently, the size of this cache (typically 64MB) is configured via a command line parameter and is fixed for the duration of the client run.

The client keeps a timestamp for each outstanding chunk in the image, recording when it last issued a request for the chunk or observed another client's request for it. The timestamp prevents the client from re-requesting data too soon. Before a partial or full chunk request is made, the client verifies that no client has requested the same chunk recently. *Frisbee* can track other clients' requests because all client-initiated messages (JOIN, LEAVE and REQUEST) are multicast.

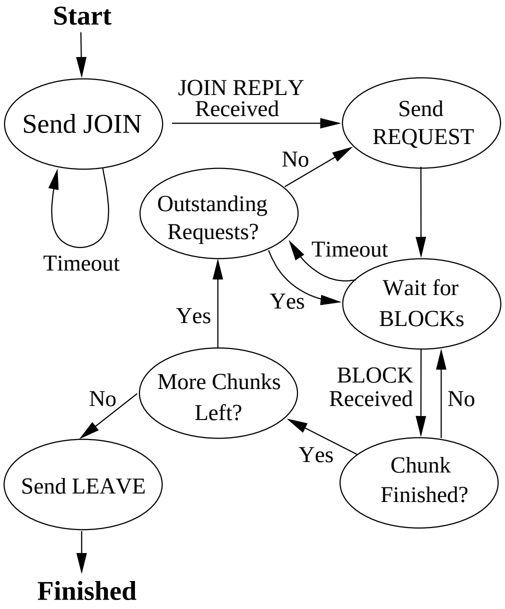

Figure 2: Basic operation of the *frisbee* client's network thread.

After a client joins a session, it sends one or more REQUEST messages to start the transfer process. Instead of having each client request chunks in sequential order, clients randomize their initial request list. This prevents the clients from synchronizing, requesting the same chunks at the same time, which would cause Frisbee's NAK-avoidance to perform less well. Each client is allowed to "request ahead" a fixed number of 1MB chunks.

Once a client has started and made its initial chunk requests, there are two situations in which it may make additional requests: when it has just completed a chunk and handed it off to the decompression thread or when it hasn't seen any packets (messages) for some period of time. The former represents the normal operation cycle: a client receives chunks, decompresses and writes them out, and makes further requests. When requesting new data following chunk completion, the first priority is completing any chunks for which some blocks have already been received. For each incomplete chunk currently in the client's cache, that chunk's timestamp is checked and, if it has been long enough since the chunk was last requested, the client issues a partial-chunk request to retrieve missing data for that chunk. Prioritizing partial-chunk requests over those for new chunks helps keep the decompression and disk threads busy and flushes data from the cache buffers sooner, making space available for new chunks. After handling partial-chunk requests, the client may also issue one or more full chunk requests to fill its request-ahead window.

If the client is initiating a request due to a receiver timeout, the request process is similar to the chunk-completed case: partial-chunk requests followed by request-ahead chunk requests. The difference is that, in the timeout case, the chunk timestamp is not consulted; the requests are made regardless of when the chunks were last requested. The reasoning is that a timeout indicates a significant packet loss event between this client and the server (and other clients), so that even recent requests are likely to have been lost. To prevent flooding the network with requests in the event of a prolonged disconnection from the server, for example a server crash, clients exponentially back-off on requests.

4.4 Image Installation

Images are installed on a disk by one of two client programs. One is the *frisbee* client discussed above; the other is a simple program called *imageunzip*. They differ only in how they obtain the image: *imageunzip* reads an image out of a file while *frisbee* uses the Frisbee protocol to obtain it from the network. Both clients share the code used to decompress the data and write it out to disk. This section describes the operation of that common code.

Since the disk image is broken into independent 1MB chunks, the decompression code is invoked repeatedly, once for each chunk. For each chunk, the header is read to obtain ranges of allocated blocks contained in the chunk. For each allocated range, the indicated amount of data is decompressed from the chunk and queued for the disk writer thread to write to the appropriate location. The separation of decompression and disk I/O allows a great deal of overlap since raw disk I/O in FreeBSD is blocking. For free areas between ranges, the client can either skip them or fill them with zeros. The former is the default behavior and speeds the installation process dramatically in images with a large proportion of free space. However, this method may be inappropriate in some environments since it can “leak” information from the previous disk image to the current. For example, in Emulab where machines are time shared between experiments, some users may wish to have all their data “wiped” from their machines when they are done. For these environments, the installation client can be directed to zero-fill free space.

5 Evaluation

In this section, we evaluate the performance of our disk imaging and loading system, testing the speed of individual parts, as well as the entire system, with a variety of disk image properties, client counts and network conditions. Furthermore, we compare the performance of our system with a similar popular commercial offering and with a differential update program.

Image	FS	Data	Pct.	Comp.	Compressed	
	Size	Size	Free	Type	Size	Time
Small	3067	624	79%	Naive	678	627
				FS-Aware	180	146
Savings from FS-awareness					74%	77%
Large	3067	1776	42%	Naive	944	685
				FS-Aware	655	416
Savings from FS-awareness					31%	39%
XP	4094	1894	64%	Naive	1688	968
				FS-Aware	575	282
Savings from FS-awareness					66%	70%

Table 1: Performance of *imagezip* on three evaluation filesystems using both naive and filesystem-aware compression. Sizes are in (1024x1024) megabytes and times are in seconds.

For our tests, we use one or more of a standard set of three test images. Our “small” image is a typical clean installation of FreeBSD on the FFS filesystem, which uses 642MB (21%) of a 3067MB filesystem. Our “large” image is a similar installation of FreeBSD that contains additional files typically found on a desktop workstation, such as several large source trees, compressed source archives, build trees, and additional binary packages; this image uses 1776MB (58%) of the available filesystem space. For comparison with Symantec Ghost, which performs best with NTFS filesystems, we used our “XP” image, which is a typical clean installation of Microsoft Windows XP Professional Edition. It uses 990MB of data with a 384MB swap file and 520MB “hibernation” file for a total of 1894MB (46%) of disk space in a 4094MB filesystem. All tests were performed on Emulab.[1]

5.1 Image Creation with *imagezip*

To characterize the performance of reading and compressing disk image files, we ran *imagezip* on our large and small images. The output file was discarded, rather than written, to isolate the image creation time from time spent writing the created image to a remote filesystem or local disk. We used both filesystem-aware and naive compression. Results are shown in Table 1. As expected, the savings obtained by using filesystem-aware compression are roughly proportional to the amount of free space on the disk. Compression speed is more than adequate

[1] In this evaluation, the clients are 850MHz Pentium IIIs, with an Intel 440BX motherboard chipset, 512MB RAM, and a 100MHz system bus. Their disks are 40GB IBM 60GXP 7200RPM IDE drives running at ATA/33, with 2MB buffers. The measured sustainable write speed to the region of the disks used in our tests is 21.4MB/second. The server is a 1.5GHz Pentium IV with 256 MB of PC133 RAM and an ATA/100 IDE disk. The clients are connected at 100Mbps and the server at 1000Mbps to a single switched LAN on a Cisco Catalyst 6509.

Target	Small FS, naive compress.	Small FS, FS-aware compress.	Large FS FS-aware compress.
null	98	21	65
disk	155	33	86
null (1 thread)	96	21	65
disk (1 thread)	242	50	145

Table 2: Time in seconds to decompress and install images from memory with both single- and multi-threaded *imageunzip*. The large naively compressed image was too large to fit into memory on our test nodes, and thus was not tested.

for our application, where images are usually generated once and used many times. Additional optimization is likely possible by multithreading the disk read and compression tasks, and eliminating internal data copies.

5.2 Image Installation with *imageunzip*

To characterize the performance of decompressing and writing disk images (independent of network distribution), we ran *imageunzip* on our large and small image files, reading the images from a memory-based filesystem. *imageunzip* uses the same decompression and disk writing code as the Frisbee client. For each test, the image file was read from the memory filesystem, decompressed, and written to disk. A second set of tests isolated decompression performance by discarding the decompressed image rather than writing it to disk. To measure the effectiveness of overlapping decompression and disk writing we repeated the tests, disabling multithreading in *imageunzip* so that a single thread both decompresses and writes the data. Table 2 contains the results.

By comparing the first two columns, we see significant savings from using filesystem-aware compression: the small image, with 80% free space, sees a time savings of 78% over the naively compressed image. From the last two rows, where there is only a single thread and thus no overlap of decompression and disk writing, we see that disk write speed is the limiting factor. Writing to disk accounts for 55–60% of the total time. Since disk writes are synchronous and the majority of the time is spent waiting, decompressing in parallel effectively hides much of its cost. This is demonstrated in the difference between the single- and multi-threaded results in which the multithreaded case is up to 40% faster.

5.3 Image Distribution with Frisbee

Scaling: To show Frisbee's speed and scalability, we ran a number of tests, reloading sets of clients ranging in number from 1 to 80. During these tests, all clients began loading at the same time. Figure 3 shows the average client runtime for the small and large images using both naive and filesystem-aware compression. The minimum and maximum times are indicated with error bars, but the variance is so low that they are not identifiable at the default magnification. Frisbee is fast and scalable: it loads the small image onto one node in 33.8 seconds, and onto 80 nodes in 33.6 seconds. It loads the naively-compressed images in nearly constant time. For the filesystem-aware large image, the runtime does increase slowly: Frisbee loads 1 node in 94 seconds and 80 nodes in 102 seconds. The reason for this difference remains to be explored; we suspect that the fraction of partial requests may be increasing, or the clients' chunk buffers may be filling up.

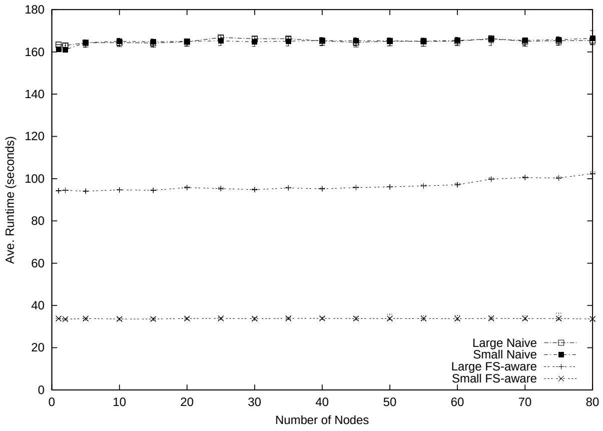

Figure 3: Frisbee client scaling from 1 to 80 nodes.

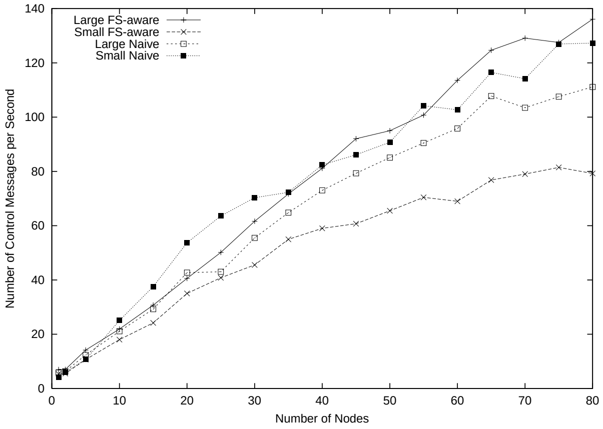

Figure 4: Average number of messages per second received by the *frisbeed* server for the scaling experiment in Figure 3.

Across all runs, Frisbee's network efficiency is very good; the number of duplicate blocks transmitted due to packet loss or duplicate requests did not exceed 8% of the total blocks sent. Note the nearly identical runtimes for the two naively compressed images, despite the nearly 50% difference in their compressed sizes. Since both must write a full 3GB of data to disk, this demonstrates that the disk is indeed the bottleneck on these machines.

Startup	Runtime (s)		Client	Dup
Scenario	Ave	Range	msgs	Data
Small Image				
Simultaneous	33.6	32.9–34.7	2753	3.2%
Clustered	35.6	33.2–40.3	4561	46%
Uniform	40.0	34.5–51.0	7875	59%
Large Image				
Simultaneous	100.2	100–101	12772	7.3%
Clustered	113.3	106–126	17266	26%
Uniform	132.4	120–147	23842	37%

Table 3: Effect of skewed client start times on Frisbee load of the small and large images, with 80 clients under three scenarios.

Figure 4 shows the average number of control messages (JOIN, REQUEST, and LEAVE messages) received by the server per second. Since the control messages are at most 152 bytes, the peak number of messages per second shown in this graph, 127, represents at most 154Kbps of upstream traffic to the server. If the linear scaling shown in this graph holds for larger client counts, control message traffic should not run into packet rate or bandwidth limitations on a 100Mbps LAN until we reach tens of thousands of clients.

One thing to note in these graphs is that the maximum node runtime remains flat even as the control message traffic rises. This is because the *frisbeed* server merges redundant requests in its work queue. For example, in the worst case at 127 messages per second, over 93% of the REQUEST messages were at least partially redundant. This indicates that there is considerable opportunity for improvement in the NAK avoidance strategy, a topic discussed later.

Another important result is that load times with Frisbee are very similar to the load times reported in Table 2 for *imageunzip*: Frisbee is able to keep the disk-writing thread supplied with data at a high enough rate that network transfer rate is not the bottleneck. With respect to supplying the disk writer, Frisbee's multicast distribution provides nearly the same level of performance as reading from local RAM on the client, and maintains this performance for a large number of clients.

Skewed Starting Times: We examined Frisbee's performance when client nodes are not started simultaneously. In practice, this can occur when clients are not rebooted simultaneously, when their boot durations vary, or when they are rebooted in groups. In this test, shown in Table 3, we loaded the small and large images on 80 clients under three different scenarios. In the first, all 80 clients start loading simultaneously, as in the scaling tests of the previous section. This is the idealized Emulab large experiment creation situation. In the second, clients are started in four groups of 20 at 10 second intervals.

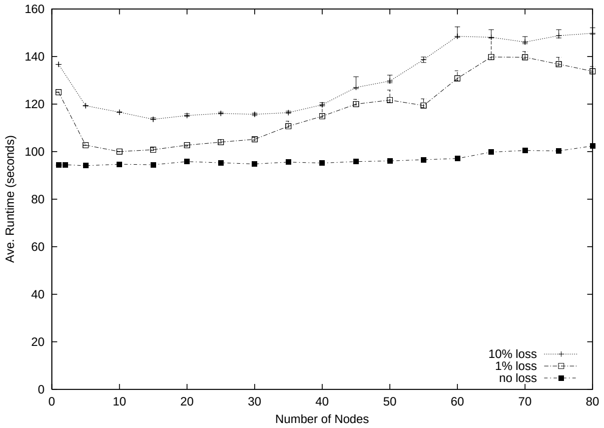

Figure 5: Frisbee client scaling from 1 to 80 nodes with packet loss. Error bars show the minimum and maximum times.

This is a realistic representation of Emulab's current behavior, where node reloads are effectively clustered by staggering reboots in groups, to avoid boot time scaling problems related to PXE, DHCP and TFTP. Finally, we uniformly distribute start times of the 80 clients over a 30 second interval, the same interval over which the clients were started in the cluster test.

As one might expect, skewing requests results in redundant block transfers. Late joining clients miss the blocks requested by earlier clients and thus request them again. This, in turn, stalls the early joining clients when the server sends redundant data. As a result, late joining clients tend to finish significantly faster. The Frisbee client could be made more fair by making its request behavior less aggressive. Currently, the client issues its own requests even if it is constantly receiving sufficient data to keep it busy. Making the client more passive, requesting data only when not making forward progress, would restore fairness. That change risks causing a client's runtime to further increase, if it doesn't quickly enough transition to making its own requests, and falls idle. However, even in the current state, we consider the ability to introduce clients into a Frisbee run at any time to be well worth the increases in client runtime and resource consumption.

Packet Loss: In general, we do not expect Frisbee to have to contend with packet loss, since its target environment is a switched LAN in which the receivers are dedicated clients. However, packet loss can still occur if the server or switch is overloaded. To investigate Frisbee's behavior in the presence of packet loss, we loaded the large image on 1 to 80 clients with packet loss rates of 0%, 1% and 10%. Packet drops were done at the server; since Frisbee clients are running only Frisbee, there will be no contention for their links. Figure 5 summarizes the results. Packet loss makes Frisbee more sensitive to the

# of Clients		Ave. Runtime (s)		Client	Dup
pc600	pc2000	pc600	pc2000	msgs	Data
Server at 70Mbit/sec					
0	4	–	94.3	895	0.0%
1	3	96.7	94.0	918	0.6%
3	1	95.9	93.9	885	0.2%
4	0	95.5	–	877	0.2%
Server at 90Mbit/sec					
0	4	–	72.3	667	0.1%
1	3	105.0	81.6	986	24.0%
3	1	106.7	93.7	1222	30.8%
4	0	106.5	–	1186	28.7%

Table 4: Effect of combinations of heterogeneous clients on Frisbee load of large image with two different server bandwidths.

number of clients, and there is definitely room for improvement, since with a large number of nodes, the 1% packet loss case performs similarly to the 10% case. Still, performance is clearly acceptable for what we expect to be a rare occurrence. It is interesting to note that a single client performs worse with loss than do multiple clients. When there is a single client, and a REQUEST message it sends to the server is lost, a timeout must pass before it will ask again. During that time, the client is idle. When there are multiple clients, blocks sent as the result of their requests will enable the first client to make progress until its timeout period expires.

Heterogeneous Clients: Thus far we have run all tests on clients of the same type. This reflects the current node base of Emulab, in which the majority of nodes are of the same type. However, given the pace of technology, it is typical for a large collection of machines gathered over time to be much more diverse. To gain a feel for how Frisbee would perform in such an environment, we performed a small-scale experiment using combinations of machines of two widely different types loading the large image. A *pc600* is a 600MHz processor with 100MHz SDRAM and an ATA/33 hard drive while a *pc2000* is a 2GHz processor with 400MHz RDRAM and an ATA/100 hard drive. Both have 100Mbit ethernet interfaces. The large difference in CPU and memory speed enables the pc2000 to decompress data much faster. The higher frequency disk interface, coupled with a newer-generation hard drive, also allows it to write much faster (38.8 MB/sec vs. 21.4 MB/sec). The hypothesis is that the pc2000s will request blocks at a much higher rate than the pc600s, causing the latter to miss blocks and make many more re-requests. These re-requests will in turn slow the effective data rate to the pc2000s. Results are shown in Table 4.

The top half of the table shows runs using the default server network bandwidth of 70Mbps, a value tuned to efficiently support the 600-850Mhz class of machines in Emulab. Here we see that runtimes are very similar for all combinations. However, the lack of improvement by the pc2000s is because they are throttled by the network bandwidth, not by the presence of slower machines. This is illustrated in the lower half of the table, where the server bandwidth was increased to 90Mbps. At this rate, a set of four pc2000s is able to load the image 23% faster, while a set of four pc600s takes 12% longer. In this configuration, we do see an effect when combining the two types. Combining a single pc600 with three pc2000s slows the faster machines, increasing their runtime to 81.6 seconds, while the slower machine runtime remains unchanged. With three pc600s and a single pc2000, the latter is further slowed to 93.7 seconds, with little change for the pc600s. This slowdown is directly attributable to the increase in duplicate data caused by the slower machines' re-request messages. While not shown in the table, the duplicate data rate tops out at 35% with eight pc600s. At this rate, the pc2000 continues to run faster than the pc600s, taking 102 seconds versus 112 for the the slower machines.

NAK Avoidance: We ran Frisbee with its NAK avoidance features, snooping on control messages and time-limiting of re-requests, disabled. With 80 clients, the message received rate at the server increased dramatically, from 85 per second to 264 for the small image, and from 146 per second to 639 for the large image.

As noted earlier, the NAK avoidance features still seem to allow a large number of spurious control messages, which are then ignored by the server. These messages are the result of using a static time limit (one second) for re-requests. When the limit is changed to two seconds, the request rate is reduced to 47 per second for the small image and 84 for the large image. However, blindly increasing the static value can result in increased client runtime when small numbers of nodes are involved and messages are truly lost. Ideally, we need to take into account the transfer rate of the server and the length of the server's work queue (which varies with the number of active clients), both of which affect the latency of an individual request. A dynamic time limit could be implemented by having the server piggyback current bandwidth and queue length information on BLOCK messages. Clients would use that information to calculate a more appropriate re-request rate.

Server Load: Although we have demonstrated that the Frisbee client performs well in a variety of situations, another important consideration is how the *frisbeed* server performs. In this section we consider the CPU, disk and network resources required for a single server instance, as well as for multiple instances running on the same host.

As *frisbeed* essentially just moves data from the disk

Startup Scenario	Server Runtime	Client msgs	Data xfer Rate (MB/s)	CPU use (%)
Small Image				
At once	34.7	2753	5.36	9.9
Clustered	55.1	4561	6.02	12.0
Uniform	65.7	7875	6.67	12.3
Large Image				
At once	101.0	12772	6.99	14.5
Clustered	126.5	17266	7.05	14.5
Uniform	150.1	23842	6.95	14.2

Table 5: Server load observed during skewed client startup tests.

Servers x Clients	Ave. Srv. Runtime (s)	Data xfer rate (MB/s)	Total CPU use (%)
1 x 80	34.7	5.36	9.9
2 x 40	35.2	11.3	32.0
4 x 20	56.5	23.0	51.6
8 x 10	58.1	31.3	72.0

Table 6: Server load with multiple, concurrent *frisbeed* servers loading the small image. The CPU time used by CPU and network monitors is not included—at 8 servers, the CPU is saturated.

to the network, we would expect the use of all three resources to increase with the number of BLOCK messages processed. As seen in the client performance measurements, increased requests most commonly occur when client startup is skewed or there is significant packet loss, causing the server to resend data. Table 5 details the run time, CPU use and amount of data transferred from disk to network for the skewed client experiment reported in Table 3. The rate of CPU and disk use is bounded by the network send rate which, as mentioned in Section 4.3.1, is controlled by a simple static bandwidth cap. The value of 70Mbits/sec used in our evaluation, which includes all network overhead, translates to 7.7MB per second of image data. In the table we can see that as the data transfer rate approaches this value, CPU us does not exceed 15%.

More problematic is the multiple server scenario. With no provision for dynamically altering bandwidth consumption, resource use is additive in the number of running servers. Table 6 demonstrates this effect as we run from one to eight *Frisbeed* instances to load 80 client nodes. Even with a 1000Mbps link from the server, at two *Frisbeed*s we are near the 100Mbps limit of the client links and the switch begins to drop packets. By eight *Frisbeed*s, the CPU is saturated. Moreover, not shown in this table is the lack of fairness between servers. For example, in the four-server case one finished in 35 seconds while the other three took longer than 60 seconds.

While we can tolerate this behavior in the current Emulab, where server and switch resources are plentiful, a better solution is needed. Recently we have prototyped a rate-based pacing mechanism so that *frisbeed* will adapt to network load. We use a simple additive-increase multiplicative-decrease algorithm which dynamically adjusts the burst size based on the number of lost blocks. The key to calculating the latter is that the server can treat any partial chunk request as indicating a lost packet. Results for this version of *Frisbeed* are mixed, with two servers quickly adapting to each take half the 100Mbps bandwidth, but with four and eight servers wildly oscillating. We believe the latter is merely a consequence of our simplistic rate-equation and not a reflection on the Frisbee protocol and its ability to detect loss events.

5.4 Comparison to `rsync`

To get some idea of the speed of our disk imaging approach compared to differential file-based approaches, we ran `rsync` on the three filesystems in our small image, with essentially no changes between source and target machines. We configured `rsync` to identify changed files but not to update any. This is a best-case test for `rsync`, since its runtime strongly depends on the amount of data it must copy.

We found that, when identifying changed files based solely on timestamps, `rsync` is approximately three times faster than Frisbee—it took 12 seconds to compare two machines vs. Frisbee's 34 seconds to blindly write the same image. Security and robustness concerns, however, prevent us from using timestamps as an accurate way of comparing files, since they are not reliable when experimenters have full root access. When `rsync` performs MD4 hashes on all files to find differences, its runtime increases to 170 seconds, five times longer than Frisbee. Given our static disk distribution needs, some domain-specific optimizations to `rsync` should be possible. For example, while it must always hash the target disk's files, the server can cache the hashes of its unchanging source disk. In the above test, server-side hashing accounted for approximately 60 seconds of `rsync`'s runtime and is serialized with client-side processing. Therefore, this optimization should reduce `rsync`'s runtime to three times Frisbee's.

However, these small tests still demonstrate that, on a fast distribution network where bandwidth is not the major bottleneck, and with disk contents such as ours, it is unnecessary to spend time identifying changed files. It is faster simply to copy the entire disk.

5.5 Comparison to Ghost

We compared Frisbee to one of the most popular commercial disk imaging packages, Symantec Ghost. Ghost has a similar feature set, including filesystem-specific compression and multicast distribution. Ghost's "high"

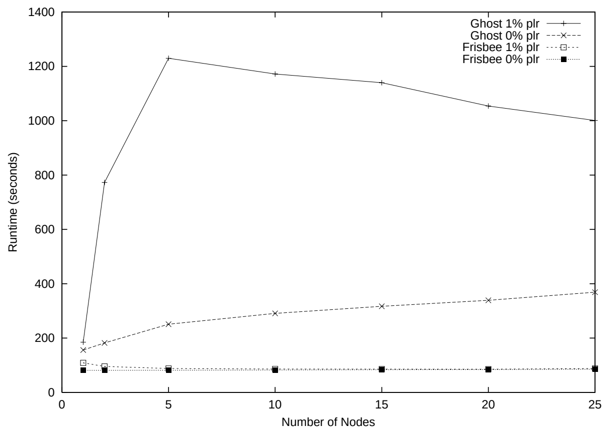

Figure 6: Scaling of Frisbee and Ghost with and without 1% packet loss.

compression setting (level four out of nine) appears substantially similar to *imagezip*'s compression (using zlib level four). We used the Windows XP disk image for this comparison, which *imagezip* compressed to 575MB and Ghost compressed to 594MB. Both Ghost and Frisbee have the ability to skip the swap and hibernation files, whose contents do not need to be preserved.

Figure 6 shows Frisbee and Ghost load times on 1 to 25 clients, with no packet loss and with a 1% loss rate. Since Ghost is a commercial product with per-client licensing, the maximum number of clients we tested was limited by licensing costs. Still, clear trends are visible: Ghost's base (one-client) time of 156 seconds is nearly twice as high as Frisbee's 81 seconds, and it increases with the number of clients to 369 seconds, while Frisbee's grows only 5% to 85 seconds. Frisbee exhibits excellent tolerance to 1% packet loss. The extremely poor behavior of Ghost in the presence of packet loss is remarkable, and bears further investigation.

An important difference between Frisbee and Ghost is that Frisbee allows new clients to connect while other clients are in the process of receiving an image. Ghost, on the other hand, requires all clients to start simultaneously. This substantially impacts the latency of the system, as all clients must wait for the slowest to begin, and clients that wish to join after a session has been started must first wait for the ongoing session to finish. One can work around this restriction by starting a new Ghost session for the same image, with the downside of unnecessarily increasing network traffic.

6 Related Work

Partition Image [16] is an open-source program for creating and restoring disk partition images. Like Frisbee, it uses filesystem-aware compression in conjunction with conventional compression to reduce the size of the image and accelerate image distribution and installation. Partition Image currently supports a larger set of recognized filesystem types. Unlike Frisbee, images are compressed as a single unit and thus the image must be decompressed sequentially. Partition Image also does not support creating complete disk images with multiple partitions. Partition Image uses a stream-oriented unicast protocol with optional encryption. Thus it will not scale as well as Frisbee's multicast protocol, but will work unchanged in a wide-area network environment. The Partition Image client can both save and restore images over the network where Frisbee currently has no built-in mechanism for saving images across the network.

HCP [18] is a hybrid technique for synchronizing disks, using a form of differential updating, but below the file level. HCP is one method used in Stanford's Collective project to copy virtual disks ("capsules") between machines. In HCP, a cryptographic hash is used to identify blocks in the client and server disks. To synchronize a block between the two, the client first requests the hash for the desired block and compares that to the hash for all blocks in all local virtual disks. If any local block matches the hash, that block is used to provide the data, otherwise the actual block data is obtained from the server. HCP takes advantage of the high degree of similarity between the multiple virtual disks that could reside on any client and the fact that the same virtual disk will tend to migrate back and forth between a small set of machines. Still, as noted by the authors, HCP is only appropriate in environments where the network is the bottleneck due to increased disk seek activity on the client.

Numerous other multicast protocols for bulk data transfer have been proposed, such as SRM [6] and RMTP [11]. Frisbee's target environment, high-speed, low packet loss, low-latency LANs, allows a much simpler protocol, which can be optimized for very high throughput. In the taxonomy of known multicast protocols presented in [10], the Frisbee protocol is considered a RINA (Receiver Initiated with NAK-Avoidance) protocol.

7 Future Work

Extending the Frisbee system from a LAN environment into the wide area presents an interesting challenge. In addition to its Emulab cluster, Netbed manages a number of nodes at sites around the world. Currently, images compressed by `imagezip` are distributed via unicast, and installed with `imageunzip`, but this will clearly not scale for a large number of nodes or frequent image distribution. Extending diskloading to the wide area will assuredly raise issues that are not present in our LAN environment. Some of these issues, such as differing client bandwidths and TCP-friendliness, have been the subjects of extensive research and we will undoubtedly be able to leverage this work. For example, techniques such as those employed by Digital Fountain [1] or WEBRC [12] may be useful. Digital Fountain uses a multicast protocol based on erasure codes [13] to create a large-scale software distribution system. WEBRC obtains an estimate of multicast RTT for flow control and TCP friendliness, and uses multiple multicast streams and a fluid model to serve clients of differing bandwidths.

When sending data in the wide area, security is also a concern—while it is acceptable to send images unencrypted and unauthenticated on a tightly-controlled LAN, care will have to be taken in the wide area to ensure that eavesdroppers cannot obtain a copy of sensitive data on the image, or alter disk contents.

8 Conclusion

We have presented Frisbee, a fast and scalable system for disk image generation, distribution in local area networks, and installation. We summarized our target application domain and have shown how aspects of that domain governed our choices in designing the system. As well as discussing our use of established techniques, we have explained our methods of filesystem-aware compression and two-level segmentation, and how they are particularly well-suited to our multicast file transfer protocol. Finally, we have shown that this system exceeds our performance requirements and scales remarkably well to a large number of clients.

Acknowledgments

Many thanks to Kirk Webb for gathering important performance results, to Russ Christensen for implementing NTFS compression, to Dave Andersen for implementing an early unicast disk imager in the OSKit, and to the anonymous reviewers for their useful feedback.

References

[1] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A Digital Fountain Approach to Reliable Distribution of Bulk Data. In *Proc. of ACM SIGCOMM '98*, pages 56–67, Vancouver, BC, 1998.

[2] R. Canetti, J. Garay, G. Itkis, D. Micciancio, M. Naor, and B. Pinkas. Multicast Security: A Taxonomy and Some Efficient Constructions. In *Proc. of INFOCOM '99*, pages 708–716, Mar. 1999.

[3] D. D. Clark and D. L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *Proc. of ACM SIGCOMM '90*, pages 200–208, Sept. 1990.

[4] P. L. Deutsch and J.-L. Gailly. ZLIB Compressed Data Format Specification version 3. Internet Request for Comments 1950, IETF, May 1996.

[5] FIPS 180-1. *Secure Hash Standard*. U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, VA, Apr. 1995.

[6] 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 Transactions on Networking*, 5(6):783–803, Dec. 1997.

[7] Symantec Ghost. http://www.symantec.com/sabu/ghost/.

[8] M. Handley et al. The Reliable Multicast Design Space for Bulk Data Transfer. Internet Request For Comments 2887, IETF, Aug. 2000.

[9] IBM Corp. The Océano Project. http://www.research.-ibm.com/oceanoproject/.

[10] B. N. Levine and J. Garia-Luna-Aceves. A Comparison of Known Classes of Reliable Multicast Protocols. In *Proc. of IEEE ICNP '96*, pages 112–123, Oct. 1996.

[11] J. C. Lin and S. Paul. RMTP: A Reliable Multicast Transport Protocol. In *Proc. of INFOCOM '96*, pages 1414–1424, San Francisco, CA, Mar. 1996.

[12] M. Luby, V. K. Goyal, S. Skaria, and G. B. Horn. Wave and Equation Based Rate Control Using Multicast Round Trip Time. In *Proc. of ACM SIGCOMM '02*, pages 191–204, Aug. 2002.

[13] A. J. McAuley. Reliable Broadband Communication Using a Burst Erasure Correcting Code. In *Proc. of ACM SIGCOMM '90*, pages 297–306, Philadelphia, PA, Sept. 1990.

[14] J. Moore and J. Chase. Cluster On Demand. Technical Report CS-2002-07, Duke University, Dept. of Computer Science, May 2002.

[15] A. Muthitacharoen, B. Chen, and D. Mazières. A Low-bandwidth Network File System. In *Proc. of 18th ACM SOSP*, pages 174–187, Banff, AB, Canada, Oct. 2001.

[16] Partition Image for Linux. http://www.partimage.org/.

[17] rsync. http://rsync.samba.org/.

[18] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the Migration of Virtual Computers. In *Proc. of OSDI '02*, pages 377–390, Boston, MA, Dec. 2002.

[19] D. Towsley, J. Kurose, and S. Pingali. A Comparison of Sender-Initated and Receiver-Initiated Reliable Multicast Protocols. *IEEE Journal on Selected Areas in Communications*, 13(3), April 1997.

[20] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of OSDI '02*, pages 255–270, Boston, MA, Dec. 2002.

[21] zlib: A Massively Spiffy Yet Delicately Unobtrusive Compression Library. http://www.gzip.org/zlib/.