Moobi: A Thin Server Management System Using BitTorrent

Chris McEniry – Sony Computer Entertainment America

ABSTRACT

We describe a tool that provides a method for running dataless caching clients – a hybrid combination of imaging systems with traditional diskless nodes. Unlike imaging systems, it is a single boot to get to a running system; unlike diskless systems, it is more robust and scalable as it does not continuously depend on central servers.

The tool, Moobi, uses the peer-to-peer protocol BitTorrent to provide efficient distribution of the image cache, and combines standard diskless tools to provide the basis for the running system. Moobi makes it possible to run large installations of "thin server" farms.

Introduction

Diskless clients are appealing for large clusters of similar systems performing similar functions since they limit the capability for local configurations and, more specifically, local misconfigurations on each system. This provides a more uniform environment and allows for the management of a large number of systems with fewer resources. The key for these systems is not so much that they are diskless in that they are dataless – there is nothing which is inherently tied to the nodes. In the desktop realm, these are typically referred to as thin clients. In the server realm, "throw away" and "field replaceable units(FRUs)" are some common terms.

The difference between diskless and dataless hinges upon drive cost, application robustness over drive failure rates, and the need for local working disk. The size of drives has increased significantly, especially when compared to the size of operating systems and applications bases. Pinheiro, et al. [1] demonstrated that annual hard drive failure rates were between 2% and 10% depending on age and usage; even with 2%, in a large population of machines, there is an inherent need for applications to account for downed nodes and thereby make drive failure moot. In addition, a local drive provides a convenient location for cached data or temporary work space, and so they are included in most clusters regardless of management methods. These reasons push commodity servers towards having local storage and treating it as a large cache or work disk, but not as a true data disk.

Since these systems have drives, they also have a unique local instance of the operating system. Whether this operating system is installed via an imaging system, or installed via an automated installation mechanism is largely irrelevant since the result is an individual system instance with a unique instance of every part of that system. This individual identity is necessary for certain items – hostname, ssh keys – but is undesirable for other items – /usr software. Any configuration management system for these setups must account and check for every piece individually to validate the consistency of the environment.

Truly diskless clients do not have this problem. Nothing is locally maintained, and a single instance, typically of an NFS server, is directly used for the environment. This provides the consistency, but loses in performance: diskless clients are very hard to scale to 100s of nodes. This limitation on scaling is due to their reliance on the large file servers that support the diskless nodes. An installed environment suffers a similar scaling issue whenever massive deployments have to happen in short order – initial spin ups, disaster recovery, or massive updates. Diskless client environments have a continuous scaling issue, while the installed environments have "impulse" scaling issues.

Moobi combines the consistency of diskless systems with the efficiency of imaged or installed systems. It achieves this by dividing the operating system image into a small configurable portion – the root or /etc – and a large fixed image – /usr – and by caching and sharing the large image via BitTorrent. The fixed image is maintained as a whole instance, so only one item needs to be validated against the source. Every time a node boots, or as often as is necessary for auditing purposes, it can scan the cache and downloads and updates the cache with any new image.

BitTorrent is used for validating and updating the image cache. Due to its nature, these two operations are inherently connected. In order to download and update, it must validate. It sees updating as the only logical followup after validating. In addition, BitTorrent allows for an update mechanism which does not require large monolithic file servers; thus it alleviates the bulk of the scaling issues for large installations. In addition, BitTorrent allows for multiple systems to distribute the image without much additional logic. This makes it easier to make the system most robust overall.

The mechanisms used in Moobi provide additional availability for applications running on Moobi

nodes. Not only does Moobi reduce the MTTR, but it also provides several hooks for system hardening. Periodic image checking provides a level of confidence in the sanctity of the software being used. In addition, the static file system images, such as the software image /usr, can be mounted with read-only permissions. Application stacks could be wrapped up in a similar manner to provide extra confidence in the stack.

Going to a hybrid solution also acts as a stepping stone. Most organizations grow organically from individual system instances to dataless systems. Hard drives tend to remain an item in new systems, and are not removed from repurposed systems.

Related Work

Given Moobi's hybrid nature, similar work falls into either the imaging and installation category or into the diskless client category, yet neither category completely describes it. Unlike imaging systems, it is a single boot to get to a running system; unlike diskless systems, it is less fragile as it does not continuously depend on central servers.

Several commercial and noncommercial imaging systems exist to date. Norton Ghost [2] is the industry standard for commercial system imaging software. Several noncommercial systems provide similar features to Ghost, and have been able to address many of the distribution issues. Frisbee [3], in particular, has been shown to have very good image distribution performance [4]. Partition Image [5] provides a wide range of image support, but is not very efficient in its distribution.

Typical system imaging packages use one of two network mechanisms for the transporting of the golden image to the target machine: a unicast transfer, or a multicast or broadcast blast. Unicast transfers can be very expensive on the boot servers. Broadcast is very stressing on the network. Multicast requires advanced synchronization techniques and places additional constraints on how and when nodes can come online. In contrast, a BitTorrent swarm limits server load, limits the scope of network usage to the ports involved, and automatically handles synchronization – if a BitTorrent client is late to the swarm, its peers will catch it up.

All imaging systems require some mechanism to address host specific file overlays. Typically, this is done by just copying down the necessary files from a central system. Ghost in particular is very good at performing post imaging instance fix-ups to reduce unique identifier conflict without needing a central software repository. However, it does not address non-windows operating systems or other post installation fix-ups, and expects the environments to fix it either manually, or via a larger control system – Active Directory for instance.

By far, the prominent implementation of diskless linux is the Linux Terminal Server Project [6]. However, diskless nodes have a long history under other operating systems, and also have a long history of performance issues with a central server [7]. Traugott and Huddleston [8] highlight Sun Autoclient's CacheFS features to reduce this. Moobi reduces these issues to occurring only at boot time, and allows for redundant boot servers with little or no synchronization between them.

BitTorrent is heavily in use in distributing large software images to large audiences. Many linux distributions, live CDs, large software applications, and VMWare appliance images [9] are distributed using BitTorrent. There are discussions of integrating BitTorrent with linux distributions' installation or update systems, but this discussion has produced little.

Recently, SystemImager [10] of the System Installation Suite has incorporated BitTorrent as a distribution method for its imaging system. Moobi and SystemImager run into many similar obstacles, especially when handling image versus host specific overlays, and when attempting to achieve high performance and scalability. However, SystemImager still views the many systems as single instances and can fall into divergent configurations. In addition, SystemImager has a two boot imaging operation – one boot for the imaging, and another boot to start the imaged system for normal operation.

Moobi Design

Moobi Basics

Several aspects are assumed for the Moobi image building and deployment discussion: how Moobi leverages the linux boot process, how Moobi distributes its own software, how additional configuration information is passed to identical nodes, and host specific overlays.

First, Moobi intercepts the normal linux boot process to prepare a specialized environment prior to handing off to the standard init process.

Almesberger [11] describes the linux boot process in great detail. The relevant portions for this discussion are summarized here. With loadable modules being as prominent as they are, almost all recent linux boots require the use of the initrd. Typically, the initrd is a pseudo-root file system which contains the kernel module utilities as well as any modules necessary for the system be able to mount the true root file system. If this is a local disk, disk drivers and file system drives are loaded. If this is a network file system, the NFS modules are loaded. Once the kernel is done with its primary initialization, it mounts the initrd as a ramdisk and executes the linuxrc script contained therein. Typically, the linuxrc script is simplified to the point of just loading the modules necessary and then hands off execution to init. Each initrd is built during any kernel installation or upgrade – the build process tailors it for the system at hand.

Moobi, like LTSP and many other alternative boot systems, uses the initrd and linuxrc as its springboard. An initrd for a Moobi booted system ends up being the full root file system. The linuxrc detects the

hardware coarsely, and initializes the network interfaces. It then connects to the boot server and retrieves the necessary torrent file for its image, starts up BitTorrent, and joins the swarm in progress. After the image is complete, it downloads the host specific overlay, then cleans up after itself and hands off normal boot execution to init.

For this to work, the kernel image and the initrd must be available to the booting node. The bios's PXEBOOT is used to retrieve pxelinux [12] which in turn retrieves the kernel image and initrd. In this case, pxelinux is the boot loader Almesberger references, and is responsible for placing the initrd into the ramdisk and then handing off to the kernel.

Moobi maintains a self contained instance of its software on each node during boot. This self contained instance is kept under /tmp/bootbin and /tmp/bootlib. By making it self contained, Moobi is removed from most dependencies on distribution resources. However, there is still a minimal expectation on the remainder of the root file system, such as standard shell and bin utils. Moobi cleans up this instance after the preinit setup to avoid taking up precious resources during boot.

The ability to provide additional configuration or status information to the linuxrc during boot is very minimal. Basic identity – hostname and networking information – is provided by DHCP and DNS. Most other configuration aspects can be derived from those, or derived from hardware detection. However, some information must be passed in before those. For instance, unless they are autonegotiated, link speed and duplex must be known prior to network initialization; otherwise, none of it works.

Kernel variables are being used to pass in the additional configuration information needed for the bootstrapping step. MOOBI_NET is the parameter for the network link speed and duplex settings example above. MOOBI_NET can be set to AUTO, or 1000F, 100F, or other appropriate values. For the author, this was necessary due to some hardware and software compatibility issues related to network link autonegotiation. Additional parameters could be used if special parameters were needed to be passed for the storage subsystem startup, but a general hardware configuration mapping would be more useful. See Future Work for more discussion on this.

As is the case in most environments, the same image may be used in several slightly different contexts. For instance, the same software image may have different services enabled for it; or the same image may be used in multiple locations and needs to be customized for those locations – different network layer permissions such as firewall rules or tcpwrapper configs. While some of this can be handled via standard and optional extensions to DHCP, DHCP is not necessarily the best place to transfer file data. Almost every imaging system uses host overlays, and Moobi is no different. For minor configuration differences between nodes using the same image, Moobi maintains a simplified file transfer over HTTP to retrieve the appropriate overlays. Other mechanisms could easily be swapped in for this.

Before a node is available for Moobi, it must be prepared with a disk partition layout which works for the images in use. The partitioning scheme only need be done once per node for any family of images that use that partitioning scheme. The only requirement for the partitioning scheme is that an image cache partition must exist. More advanced logic in the linuxrc could be used to overcome this limitation of the current implementation.

Moobi Image Build

A Moobi ''image'' is actually comprised of several file system images, file system skeletons, and the kernel file. One of the true file system images is the root file system. Typically, the other true file system image is /usr. Given most installations, this is the largest and single most monolithic image that does not change. A file system skeleton is a simple text file which summarizes file and directory x ownership and permission properties for a file system. This is useful for the variable file systems such as /var and /tmp. Since these will either be local disk or ramdisk, a pure imaging technique provides no advantage over a simplified technique which just ensures the existence of the structure.

Images can be transfered in one of three ways: TFTP – primarily just for the root file system image and the kernel file; HTTP – for the skeleton images; and, the preferred, BitTorrent – for the large file system images. The root partition is transfered via TFTP; since TFTP is the least efficient and robust, any transfers using it should be minimized, and therefore the root partition should remain as small and streamlined as possible. Luckily, linux kernels at this point recognize gzip compressed initrds, so the root partition can be compressed. The skeleton files are transfered via HTTP since it provides a reasonable ability to recover from transient system or network performance failures, and due to the fact that multifile torrent support is limited in some BitTorrent client implementations. Given the average size of a skeleton is on the order of 10K, a swarm is not necessary. BitTorrent is used for large file system images since it provides both a robust and efficient transfer mechanism and a native file hash checker.

The ''imagebuilder'' script has been developed which aids building images from scratch. Imagebuilder takes many of the standard build system inputs: a description of the file system containers or image files, pre and post installation scripts, a package list, and a miniroot of static file assignments. In addition, Imagebuilder takes a series of configuration options for the script run itself: source, destination and

working directories, the kernel package to be used, and the image name. The imagebuilder.conf uses the .ini configuration file format. A typical file system configuration looks like:

```
 [partition:/]
name=root.img
size=128M
compress=1
skipmd5=1
 [partition:/usr]
name=usr.img
size=1024M
compress=0
skipmd5=1
 [partition:/var]
name=var.skel
skeleton=1
compress=0
skipmd5=1
```

Of note is the skeleton flag which signifies that the file system in question is a skeleton file system.

The imagebuilder process consists of:

1. Creating a temporary working directory to act as a installation root.
2. Running the pre-script to initialize the working directory.
3. Locating, via a search path, and installing all of the packages identified by the package list.
4. Copying over any additional stand alone files from a mini-root. The linuxrc is typically kept and installed from here.
5. Running the post-script on the working directory for any last fix-ups.
6. Building the image partition files, starting with the deepest path first and working up to the root.

For a true file system image, it creates the image file by:

- 6a. Generating an appropriately sized zeroed image file.
- 6b. Creating an ext2 or ext3 file system on the image file.
- 6c. Mounting the image file on a loopback device.
- 6d. Rsyncing the appropriate subdirectory onto the image file's mount point. Subdirectories for other images are explicitly excluded.
- 6e. Unmounting the image file and performing any additional operations on the image (generating a checksum hash, compressing it, etc.).

For skeleton images, it creates a text file by recursing through the appropriate subdirectory and collecting file name, ownership, and permission information.

This process does not need to be followed for a "valid" image to be built. Any generation system which creates valid image files containing valid file systems could be used. For instance, a linux live CD image could be used as a followup. However, for this to work, the linuxrc must be able to retrieve, place, and open the image files.

Moobi Image Deployment

Once the image is built, it is moved to the boot server, broken up and located as appropriate for the image transfer mechanism. The kernel image and initrd/root file system are place in the TFTP directory. The skeleton files are placed in the HTTP service's document directory. Large file system images are also located in the HTTP service's document directory, since a specific location is not necessary as the BitTorrent client will work any where. It also allows for singular fix-up transfers to happen over HTTP should the need arise.

The image shephard, or ishephard, process is responsible for maintaining all of the subsystems which are necessary for a complete Moobi boot: any network configurations, the dhcp daemon, the tftp daemon, the http daemon, and any BitTorrent seeders. The boot server is the primary for all of these, but additional boot servers could provide equivalent services. All function independent of each other, and typically, the first to respond would be the one to be used.

A typical Moobi node boot proceeds as shown in Figure 1.

The current boot server handles multiple VLANs, primarily to directly serve DHCP with network helpers. Since it already has an interface on each VLAN, it has a specific seed for that VLAN. This restricts network traffic two-fold. No BitTorrent traffic traverses routed interfaces, and this reduces the need for firewall holes. In most but not all cases, VLANs do represent separate network localities. Restricting network traffic to the VLAN should keep network stress localized. However, it would not require much additional configuration if an environment wants to allow routed interface traversing – does not want to provide a trunk to the boot server, does not have give VLAN distribution, etc. Each image distribution is given a specific port which is configurable and can be kept to a small range – so the firewall or network ACL changes could be kept minimal.

Experimental Data

Initial deployments have been able to boot over a hundred systems with 1.4 GB /usr images with an average deployment time of around 5-6 minutes on a 100 Mb/s ethernet network spanning multiple switches linked by multigig trunks. The boot failure rate with this high of a load has been unacceptably high at about 5-10% in the worst case which has been primarily in the initial tftp transfer.

A more rigorous set of tests where run for this paper. The environment consisted of 64 booting nodes, a DHCP/PXE boot server, and 1-4 BitTorrent seeds or one HTTP server. All systems were running on commodity hardware with a single 2.4 GHz P4, 4 GB of RAM, and a 1 Gb network connection. All devices

were evenly split across two Cisco 4948 switches with a 4 Gb trunk connecting them; logically, all devices lived on the same VLAN.

The runs were broken up into different load sizes (1, 4, 16, or 64 nodes), and into different load mechanisms or configurations. The configurations were labled:

SEED BitTorrent distribution using a single seed, and the booting node continued sharing after it finished booting.

NOSEED BitTorrent distribution using a single seed, but the booting node stopped sharing after it finished booting.

4SEED BitTorrent distribution using four seeds, and the booting node continued sharing after it finished booting.

4NOSEED BitTorrent distribution using four seeds, but the booting node stopped sharing after it finished booting.

HTTP Download of the fixed image using wget over http.

Each series of runs – load size and configuration – was performed five times. For the single node boot, only three configurations were used: SEED, 4SEED, and HTTP. The NOSEED and 4NOSEED configurations are identical to the SEED and 4SEEd configurations, respectively.

A run consisted of issuing a shutdown/restart to all of the nodes at the same time, and measuring all times relative to the shutdown issuance. Time measurements were taken from time of boot, time of the

Client Node	Server or Swarm
BIOS initialization	
DHCP request	
	DHCP response with network and followup up location for TFTP
TFTP request	
	TFTP response of pxelinux
TFTP request for kernel image and initrd	
	TFTP response of kernel and initrd
initrd loaded into ramdisk and kernel loaded	
kernel initialization	
kernel execs linuxrc	
linuxrc hardware detection	
network reinitialization DHCP request	
	DHCP response with network information
self identification DNS reverse lookup	
	DNS response
Mount cache file system	
HTTP request for image torrent	
	HTTP response of image torrent
BitTorrent Client startup image file checksum join swarm	
	Swarm sending and receiving image data
	Image file finish
Mount image file as appropriate	
HTTP request	
	HTTP response of skeleton files
buildSkeleton for each skeleton file	
Multiple HTTP requests	
	HTTP responses of overlay host files
Shutdown network and cleanup	
Hand off to init for normal boot	

Figure 1: Boot Times

fixed image transfer start, time of the fixed image transfer end, and time of the end of boot(last command in rc.local). From these the time for PXE transfer and fixed image transfer were calculated and reported; see Table 1.

Node Cnt.	Method	Total	PXE	D/L	Times(s) Failures
1	SEED/NOSEED	397.8	27.4	166.4	0.0
1	4SEED/NOSEED	390.8	27.2	163.0	0.0
1	HTTP	245.0	27.2	19.0	0.0
4	SEED	452.8	27.0	226.8	0.2
4	NOSEED	450.8	27.4	221.0	0.0
4	4SEED	418.6	30.0	188.8	0.0
4	4NOSEED	419.8	29.4	186.6	0.2
4	HTTP	277.0	28.7	66.3	0.0
16	SEED	542.8	36.0	315.0	0.0
16	NOSEED	490.2	36.2	258.8	0.0
16	4SEED	443.4	35.4	204.0	0.2
16	4NOSEED	456.8	35.6	225.6	0.2
16	HTTP	449.5	36.2	233.0	2.2
64	SEED	602.2	127.6	280.2	0.0
64	NOSEED	623.8	117.4	313.0	2.6
64	4SEED	569.0	122.6	247.6	0.2
64	4NOSEED	626.2	109.8	320.6	1.4
64	HTTP	1208.0	135.3	892.0	0.0

Table 1: Execution times.

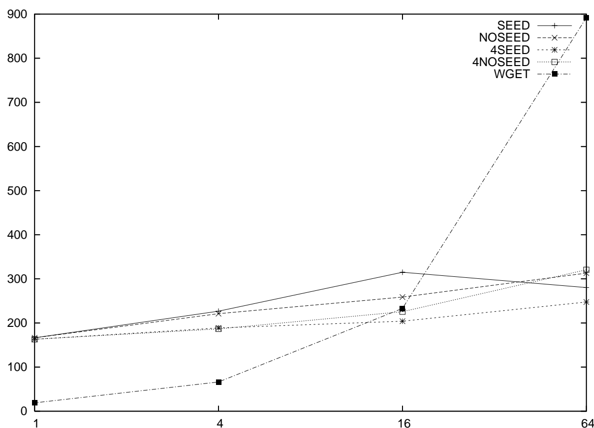

Figure 2: Boot times.

CPU and network usage for the serving hosts were watched during the course of these runs. Of note, only two resources appeared to be stressed:

1. the HTTP server's network interface during the fixed image transfer – expected since it is the only source for upwards of 128GB of data to transfer.
2. the DHCP/PXE boot server's load during tftp transfers – it spawns separate processes for each tftp connection.

Beyond those, CPU utilization never passed 35% and network utilization never passed 10%.

A kickstart was also performed as an additional comparison. The approximate time for the kickstart through to an available system was on the order of 1200 seconds. Given the distribution method for the kickstart(single source HTTP), the expected time would increase as the HTTP methods from above.

Observation 1: As expected, a peer-to-peer distribution system works exceedingly well in scaling to many nodes.

Observation 2: The transfer time for the PXE portion grows as a single source. This leads to boot failures which require manual intervention.

Observation 3: There is only a minor improvement when providing additional seed nodes.

Observation 4: The time for a seed image distribution compared to an installed mechanism was approximately 1:4.

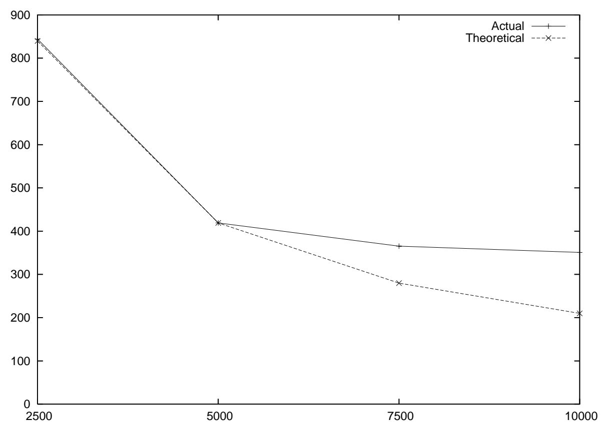

Figure 3: Boot Times with a Rate Limited BitTorrent.

Observation 5: The largest influencer of distribution time is the initial seed's transfer rate. Even for a large environment – 64 nodes – the download time was very close to the case where one node would download at the transfer rate.

Transfer Rate Limit	Actual Time			Theoretical Time
	Total	PXE	D/L	
2500	1165.6	122.8	844.2	839.7
5000	740.7	122.3	419.0	419.0
7500	682.4	123.4	365.4	279.9
10000	665.5	120.5	350.8	209.9

Table 2: Boot Times with a Rate Limited BitTorrent.

Summary

In general, Moobi performs very well for large installations. It appears to have a break point on the order of 10s of nodes at which it is equivalent to installed or imaged systems from a single node. In addition, it does not show significant performance degradation as the number of clients increases. Given its efficient scaling, it allow for large clusters of systems which are image updated on a very regular basis, and this allows for a shift in the way those systems are managed.

Critique

Moobi is reliant upon the node's bios's pxe implementation, either system or NIC, for proper behavior. Several pxe implementations which the author has worked with are not very robust. They either are unable to recover from an overloaded boot server to hand out a response in a given unconfigurable timeframe, or they are unable to recover from a slow tftp transfer – once the tftp has slowed down, the client will not receive data any faster then the lowest previous data send even if the server is able to recover and begin sending more. Additional boot servers, a more efficient tftp server, or better bios support could overcome this.

Several pieces of the current Moobi implementation are specific to RedHat-like distributions. This is just a limitation of this implementation and not a limitation of the technique. RPMs could be replaced with DEBs. The python BitTorrent client and related hostfiles tools could be replaced with compiled or other scripting languages better suited for the desired distribution.

Currently, the linuxrc used within Moobi is very specific to the structure of the image in use. This is a feature and failing of Moobi since it requires in depth knowledge of the process to be able to boot systems. This could be made to be more robust and accept different image structures as an output of the image building process.

Future Work

Advanced hardware detection. Currently, this is only variable to three possible hardware configurations and so the detection mechanism is very minimal – mainly observing the hard drive(IDE versus SCSI) and the ethernet interfaces. This fails to scale as is, but two approaches can be attempted for this. The first is a relatively small hardware lookup table. Since many large organizations use a limited set of vendors and system configurations, a small table will most likely be sufficient. This could be passed in as a kernel command line parameter (MOOBI_HARDWARE_ID or so), or a similar detection of base hardware assets.

Image overlays. The image overlay system is very immature. It currently just lists files to be transfered with the correct properties. This would be an ideal place to drop in high level configuration management systems, such as Cfengine or Puppet, for a more manageable approach.

Partial image updates. The current usage of Moobi treats each new image as a completely different item. This means that all updates involve transferring the entire new image, and Moobi has certainly been optimized for this. However, an additional incremental improvement can be achieved when acknowledging that most of the time the primary image is not very different. Typically, only a small fraction of the image description changes – a new software package or such – and that corresponds to only a small fraction of the fixed image file changing. Given BitTorrent's blocking scheme, it would pick up that only the blocks with changes need to be sent, and could easily reduce the image transfer.

BitTorrent locality information. BitTorrent is very efficient, yet it does not know anything about its local world. All clients are equivalent to it. Since many large cluster nodes are grouped by switch, it would be convenient to leverage this so as to not saturate any switch to switch links.

Author Biography

Chris McEniry left the Massachusetts Institute of Technology with a BS-EECS in 2000, and has spent over ten years as an official and unofficial systems administrator. He can most recently be found in Southern California, helping to keep the lights on with Sony Computer Entertainment America (aka Playstation). Reach him electronically at cmceniry@mit.edu .

Bibiliography

[1] Pinheiro, Eduardo, Wolf-Dietrich Weber, and Luiz Andre Barroso, "Failure Trends in a Large Disk Drive Population," *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, pp. 17-28, 2007.

[2] *Norton Ghost*, http://www.symantec.com/home_homeoffice/products/overview.jsp?pcid=br&pvid=ghost12 .

[3] Hibler, Mark, Leigh Stoller, Jay Lepreau, Robert Ricci, and Chad Barb, "Fast, Scalable Disk Imaging with Frisbee," *USENIX Annual Technical Conference Proceedings*, pp. 283-296, 2003.

[4] White, Brian, et al., "An Integrated Experimental Environment for Distributed Systems and Networks," *Proceedings of the 5th Symposium on Operating Systems Design & Implementation*, pp. 255-270, 2000.

[5] *Partimage*, http://www.partimage.org .

[6] McQuillan, J., "The Linux Terminal Server Project: Thin Clients and Linux," *Proceedings of the 4th Annual Linux Showcase and Conference (ALS2000)*, Usenix Association, 2000.

[7] Gusella, R., "A Measurement Study of Diskless Workstation Traffic on an ethernet," *IEEE Transactions on Communications*, Vol. 39, Num. 9, pp. 1557-1568, 1990.

[8] Traugott, Steve and Joel Huddleston, "Bootstrapping an Infrastructure," *Proceedings of the 12th Systems Administration Conference (LISA '98)*, pp. 118-196, 1998.

[9] *BitTorrent download info*, http://torrent.vmware.com:6969/ .

[10] Finley, Brian, "VA SystemImager," *USENIX Annual Linux Showcase and Conference Proceedings*, pp. 181-186, 2000.

[11] Almesberger, Wener, "Booting Linux: The History and the Future," *Proceedings of Ottawa Linux Symposium*, 2000.
[12] *PXELINUX – SYSLINUX for network boot*, http://syslinux.zytor.com/pxe.php .