STOIC: Streaming Operating Systems in the Cloud

Riley Collins, Teo Gelles, Benjamin Marks, Alex Simms, and Kevin C. Webb
Department of Computer Science, Swarthmore College, Swarthmore, PA
{rcollin4,tgelles1,bmarks1,asimms1,kwebb}@cs.swarthmore.edu

***Abstract*—As cloud computing continues to increase in performance, while simultaneously decreasing in cost, it comes as no surprise that traditional computing tasks are frequently being offloaded to cloud services. This paper presents STOIC, a service model for booting and streaming an operating system from public cloud infrastructure. Having booted with STOIC, users can perform regular activities with few noticeable differences between STOIC and more traditional methods of booting an OS.**

STOIC makes minimal assumptions about the hardware and software capabilities of the booting client and is compatible with many popular cloud storage providers. We show that STOIC's file streaming is responsive and demonstrate several use cases in which streaming an OS is beneficial when compared to alternative file distribution methods.

I. Introduction

The rise of public cloud computing is transforming the role of off-site infrastructure. Already, many public cloud providers [1][2] and {platform, software, etc.} as a service providers [3][4] promise infinite resources to anyone willing to pay, with prices that are competitive with traditional infrastructure. Despite this clear trend, cloud computing remains in its infancy, and many common computing tasks still lack online analogs.

This work presents STOIC, a service model that allows users to stream an operating system stored remotely in the cloud and persistently maintain system configuration and data between across reboots. A key insight making STOIC feasible is that at any given time, most of the OS and file system is not being used. Thus, STOIC transfers only a small amount of data to initially boot the system, transparently fetching more from the cloud as it is needed. With increases in consumer bandwidth and main memory availability, an analogous model already dominates the distribution of digital video (e.g., YouTube). Indeed, many users would likely balk at the idea of fully downloading large video files to the disk, and we believe that STOIC yields analogous benefits for OS distribution.

While we do not expect it to replace traditional disk-based OS storage for most tasks, streaming an OS with STOIC enables several interesting use cases:

OS service, with persistent storage: STOIC facilitates an *operating system as a service* model in which a provider maintains a selection of standardized system images. Such a service would allow end-host software maintenance to be outsourced to the cloud. Combined with a persistent cloud storage system, such a model decouples the OS from the hardware, permitting users to work from any machine, regardless of location, with their familiar files and operating environment.

Decoupling the OS and files from the hardware has the potential to improve data security. A stolen or misplaced laptop that exclusively relies on the cloud for its file system poses no threat of information leakage. Additionally, a user might choose to stream a privacy-preserving OS [5] (e.g., to avoid government censorship), without leaving a trace on her device.

Software demonstration: Many software packages require more environmental setup than can be provided by traditional "double click to install and run" installers. For example, the authors of Mininet [6] simplify its distribution by sharing a preconfigured virtual machine image. Other packages (e.g., Hadoop [7]) expect substantial configuration prior to executing, posing a barrier to rapid evaluation and deployment.

With STOIC, users can quickly boot their PC (or a VM) and stream an OS whose environment is preconfigured by the software maintainer. Rather than transferring an entire heavyweight VM image in advance, a STOIC user loads only the necessary data on demand.

Internet of Things (IoT) devices: As the number of always-connected devices increases, so too does the task of managing their software. The STOIC model allows IoT users to easily outsource software maintenance to the cloud, potentially reducing device cost by curtailing the need for onboard persistent storage. Further, users of prototyping devices based on open-source software (e.g., Raspberry Pi) would benefit from the flexibility of easily switching between operating systems and preconfigured environments to find the combination that best fits their prototype's needs.

OS installation and diagnostics: Booting a device over the Internet is convenient for infrequent tasks like OS installation, especially as fewer devices are shipping with removable media drives in favor of portability. We envision a publicly accessible repository of boot options that would allow users to stream Linux installers or diagnostic utilities.

This paper makes the following contributions:

- **STOIC service model:** We characterize the key components of the STOIC architecture, their responsibilities, and a cohesive vision for their adoption by cloud providers.
- **Prototype implementation:** We describe the design and implementation of a fully-functional STOIC prototype. The prototype introduces a few custom utilities that tie together standard infrastructure software and protocols, making it easily deployable on today's cloud systems.
- **Use-case evaluation:** We demonstrate the advantages and feasibility of STOIC by examining several realistic use cases and illustrating their performance.

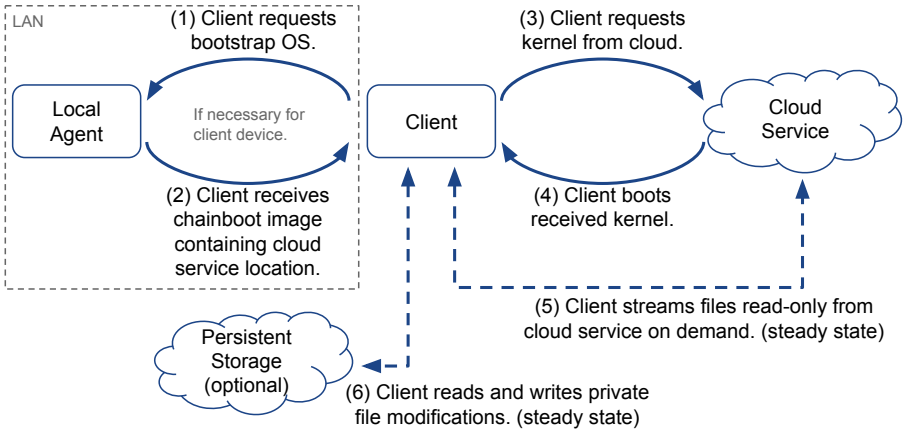


Fig. 1: The STOIC architecture and remote boot procedure.

II. Model and Vision

Clients following the STOIC model retrieve bootstrapping data, boot an OS from cloud-sourced files, and mount user file systems. The STOIC architecture divides the responsibility for the process into three components: a booting client, a local bootstrapping agent, and a cloud-based service offering bootable operating systems and (optionally) persistent file storage. Figure 1 summarizes STOIC's components and the boot procedure.

A. Local Bootstrapping

Many devices are equipped with vendor-supplied mechanisms for only *local* network booting and do not allow users to feasibly modify the boot process (e.g., by writing customized routines within the BIOS/EFI/firmware). Thus, the local agent is a pragmatic addition to the STOIC model to provide compatibility for such devices, which use it to acquire a small bootstrapping image, locally boot the device, and redirect its boot procedure to the proper wide-area cloud service. For home users, the role of local agent would likely be played by the ubiquitous home router/NAT/WiFi device. For enterprises, the local agent's functionality could easily be incorporated into standard DHCP infrastructure.

The local agent component is not fundamentally necessary to STOIC, and devices designed for wide-area bootstrapping (or those with freely available firmware sources) would not require one. However, eliminating the local agent for all devices would necessitate changes to the behavior of traditional PC BIOS/EFI beyond the scope of this project[1], although such work does exist for some PC hardware [9].

B. OS Selection

Having attained wide-area booting capabilities, the client contacts the cloud service to obtain a list of the operating systems that are available for streaming. The user selects an OS, retrieves from the cloud only the minimal amount of data necessary for booting it (the kernel and necessary hardware drivers), verifies the data's integrity, and subsequently boots the retrieved kernel.

C. File Systems

To ensure that all subsequently accessed data is streamed on demand, the client mounts its root file system directly from the cloud service. We envision a common use case in which many users wish to simultaneously boot from the same OS image (e.g., to run a public OS installer or load a policy-compliant corporate image). To facilitate such image sharing, STOIC's root file system defaults to being mounted read-only.

Users wishing to store persistent files can mount a writable file system overlay from an alternate location. To ease usability, utilities for mounting file systems from a variety of sources could easily be provided in the read-only root file system, prompting the user to select a cloud service and enter their account credentials.

D. Persistent Storage Service Models

In general, we expect that users will have many disparate use cases for STOIC, each of which may have contrasting persistent file storage needs. Thus, we make no attempt to prescribe a one-size-fits-all model for authorization, storage, and payment. Instead, we broadly classify the availability of an OS image and the user's persistent storage strategy as either *public* or *private* and describe how various combinations might be used by a different category of user.

Public OS images are those that any user can load from a well-known, publicly-accessible location. For public images, obtaining persistent storage would be up to the user (e.g., local disk or privately-funded cloud storage), although support for a particular provider could be integrated into the public image. In one model, a cloud service provider might attract users to their platform by providing a free, publicly-available OS that connects to the provider's paid storage service to host persistent file storage. Alternatively, images might be freely available, containing curated "live CDs" like the publicly-hosted operating system and software installation package mirrors that currently serve the open source community.

[1]Apple's OS X Internet Recovery [8] feature demonstrates the feasibility of Internet booting directly from firmware, albeit in a far more restrictive capacity.

By contrast, private images require users to authenticate themselves prior to booting to protect against unauthorized access. The costs of hosting and transferring data are attributed to the individual or organization using the service. Being protected, the image could automatically mount persistent files at startup from a predetermined location, without user intervention, to realize an "operating system as a service" model akin to other services that have migrated to the cloud.

III. Prototype Implementation

This section describes our experiences in constructing a fully-functional prototype implementation of the STOIC model. The prototype's components closely match the architecture depicted in Figure 1, with a few exceptions as noted below. It represents a proof of concept for implementing STOIC with largely off-the-shelf software and is designed to serve traditional PC hardware (or PC-like VMs), which necessitates a local agent component.

A. Booting from the Local Agent

Much of the prototype's booting procedure is delegated to PXE [10], the Pre-boot eXecution Environment. PXE is a mature standard that is well-supported by most commodity PCs available today, enabling a client machine to boot from a server on its local network. STOIC clients connecting to their local PXE server receive an open-source bootstrapping image, iPXE [11], which enables booting from remote images.

Our prototype currently supports booting from two local agent platforms, each of which represents a distinct operating environment. The first corresponds to a home or small office, where the local agent is a Banana Pi R1, a Linux device akin to a Raspberry Pi with an embedded wireless Ethernet chipset and a 5-port hardware switch, giving it roughly the same form factor and processing capacity as a commodity home WiFi router. Our success with this platform demonstrates that a STOIC local agent could be easily incorporated into existing home router firmware, making STOIC immediately available to millions of users.

The second platform is a large-scale PXE installation used by the computer science department at Swarthmore College. Booting any of the approximately 100 PCs with STOIC required only minor modifications to existing PXE-related configuration files.

B. Cloud Backend

Having received the iPXE boot image from a local agent, the client contacts a web server running on Amazon's EC2 cloud service to fetch a list of available OS images. Our current implementation supports only Linux, although there is no technical barrier to booting other operating systems. Upon selecting one of the options, the client retrieves and boots a Linux kernel, initial RAM disk, and boot parameter string.

The initial RAM disk carries modular device drivers, and the parameter string contains contact information to mount a root file system via NFS. Like a "live CD", the root FS is mounted such that users can write to files in memory, but writes are not persisted to the cloud, whose NFS is exported read-only. Our prototype's NFS server runs on the same EC2 host as the web server. We selected NFS because it is well supported by Linux and because, by its nature as a network file system, it does not transfer file data until a file is accessed. While we recognize that WAN-oriented file system techniques [12][13][14] may improve performance in production, even NFS's relatively unsophisticated behavior satisfied the needs of our prototype implementation.

C. Persistent Storage

Users wishing to make persistent file changes must provide a writable device, which STOIC's OS images mount as a writable OverlayFS [15] transparent overlay on top of the root file system. The writable device may be local (e.g., local disk, USB flash drive, etc.) or remote. In the latter case, STOIC uses custom cloud storage *relay* software, which forwards I/O requests issued by the booted client to a cloud storage provider. The persistent storage overlay stores only the differences from the underlying read-only root file system.

Our prototype relay is implemented as a network block device (NBD) [16] server. NBD provides a simple interface akin to block devices in which a client may read/write fixed-size blocks, each of which is uniquely numbered. The relay receives block I/O requests from clients and forwards them to a supported cloud storage provider via public APIs. In contrast to Figure 1, our prototype relay software runs on the local agent, though it could be incorporated into the read-only OS image itself.

The relay is modular, supporting several cloud storage *backends*, with implementations for Google Drive, Dropbox, and Amazon S3. Most of our experiments with persistent storage have used the S3 backend, as the other two are free services that quickly rate limit data transfers. Backends may require parameters, which specify service locations and accept user authentication credentials. S3, for example, requires users to provide a bucket name and authentication key pair.

We have implemented three performance optimizations for the prototype relay. First, we avoid storing unused blocks on the underlying cloud storage service to reduce storage costs. Blocks are materialized at the cloud storage service on demand during their initial write. The STOIC relay maintains a small in-memory cache to record which blocks are currently allocated.

Second, the relay employs a small data cache containing the most recently read file system block(s). The cache defaults to a small size, just a single block, to avoid overcommitting memory-constrained platforms like in-home routers. Clearly the cache size represents a tradeoff between performance resource usage. We expect that larger deployments would dedicate more memory to caching than our prototype.

Finally, to reduce request latency, the relay is multi-threaded and asynchronous, allowing clients to issue multiple outstanding NBD block requests in parallel. In a naïve implementation, this might raise the possibility that relay worker threads could attempt to concurrently access the same remote blocks,

potentially executing requests out of order. STOIC avoids such races by hashing each requested block number when assigning requests to a thread. Thus, all requests for the same block will enter a queue for one thread, preserving the intended request ordering. Requests mapping to other threads are free to execute independently in parallel.

D. Security

The STOIC model assumes that in a commercial deployment, the OS image server and cloud backend would be secured against attack by the service provider, who is already incentivized to properly identify whether or not a client has proper authorization for billing purposes. Thus, we focus on ensuring that the following two security guarantees are implemented in STOIC: verifying the integrity of disk images downloaded through iPXE and secure communication when making reads and writes via the persistent storage relay.

The kernel and initial RAM disk files are transferred securely over HTTPS. To verify their integrity, STOIC provides cryptographic signatures alongside the OS images on the cloud backend server. iPXE has configuration options that mandate verification of disk images with signatures before booting, and we embed the necessary certificates in the local agent's iPXE response.

Communication between the STOIC relay and persistent cloud storage providers is encrypted for backends that supply a HTTPS API (e.g., Amazon S3). Our prototype currently does not encrypt all traffic served via NFS because it is assumed to be a read-only, publicly accessible root file system. Such traffic could easily be encrypted should protection be desired in a production setting.

IV. Evaluation

Our evaluation explores STOIC's ability to stream an operating system on demand from the cloud. We start by demonstrating novel use cases. Then, we characterize STOIC's performance and cost. In our experiments, clients contained 16-core AMD Opteron 6128 processors and 16 GB of memory. The STOIC cloud service backend was served from an Amazon EC2 m4.large instance, with two 2.4 GHz Intel Haswell-family processors and 8 GB of memory. The persistent storage relay ran on a Banana Pi R1 router development board and communicated directly with Amazon S3.

All tests were performed from Swarthmore College's computer science department network, which shares the college's 1 Gbps connection to the Internet with the rest of the campus. With our experiments, we aim to show that STOIC's approach is viable under good network conditions. Clearly connection performance matters a great deal in STOIC performance benchmarks. We typically see throughput rates of 35 to 60 Mbps from our AWS instance, which is reasonable for a home user in a metropolitan area.

A. Use Cases

The most important aspect of our evaluation is a qualitative illustration of the STOIC model's capabilities. While performance is likely to vary depending on network connectivity and congestion, these use cases demonstrate STOIC's novel scenarios. Here, we highlight three examples of STOIC's interesting use cases in order of increasing complexity.

OS installation: The simplest STOIC use case is booting an OS installation or recovery image. This example shows that STOIC is on par with existing Internet-based booting projects (e.g., Apple's Internet Recovery [8]). Here, users select an OS image to be used only once, with the intent of modifying their local disks for subsequent local boots. Our STOIC example serves the contents of an Ubuntu 15.04 installation disk. Upon booting, the user is presented with a graphical Linux environment, a transient in-memory file system, and several guided options for OS installation, recovery, and other administrative tasks.

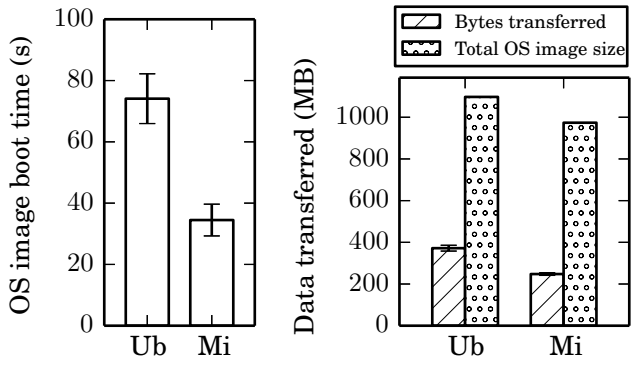
Software trial / demonstration: Deploying software in VM images and virtual "containers" is becoming an increasingly popular means of software distribution [17], [18]. Such containers are attractive to software maintainers because they allow complete control over the software's *environment* in addition to the software's behavior. For computational researchers, they make reproducing experiments easier, since the entire experimental environment is preserved [19]. One such research-focused software project shipping software in a VM is Mininet [6]. Mininet is a network emulator that simplifies testing and evaluating software on a virtual network.

STOIC serves the Mininet VM's contents as one of its streamable OS images, making it easy for users wishing to experiment with Mininet (or potentially any other containerized software) to evaluate it without committing any permanent storage. A STOIC user can easily boot a machine directly into the environment created by Mininet's authors in approximately 30-40 seconds. Furthermore, because STOIC streams only the data necessary to serve user requests, much of the rarely-used environmental data is not transferred, offering substantial savings, which we quantify below.

For users not taking advantage of STOIC's persistent storage options, the trial image's data simply disappears when the user reboots. Alternatively, users can easily store all permanent writes to Amazon S3 using the file system overlay described in Section III. Thus, users wishing to return to the experimental environment can modify any file, and only the file differences need be safely stored by STOIC in S3. By booting the same image and overlaying the same remotely-backed file system, a user can restore earlier sessions with all file modifications preserved.

An OS for a cluster, as a service: Beyond software trials, one might wish to outsource the maintenance and configuration of an OS to a third party, who offers the operating system as a service. Our final STOIC use case is an example of a cluster environment in which we have crafted a pair of OS images, based on Ubuntu's 15.04 live CD, that boot a preconfigured MapReduce [20] cluster using Apache Hadoop [7]. STOIC allows a user with a rack of machines to boot into a fully-configured Hadoop cluster in approximately five minutes.

The first image in the pair represents the master node, and the user begins by selecting one node to boot using this



(a) Time to boot two example OS images.

(b) The bytes transferred booting each OS compared to their total image sizes.

Fig. 2: The time (a) and bytes transferred (b) during ten boots of the Ubuntu (Ub) and Mininet (Mi) OS images. Error bars represent standard deviations. Each image was tested on a different day, and transfer rates varied due to transient network conditions.

image. Once the master has booted, it displays instructions to the user directing them to run an included script, which listens for connections from worker nodes. To eliminate user interaction on each worker node, STOIC embeds the master's IP address in the worker OS image's boot parameter string. Other rendezvous possibilities, such as assigning the master node a resolvable DNS name, could work as well.

With the master online, the user continues by booting any number of machines with the worker image. Each worker generates a unique host name and ssh key pair at start up, which it sends to the master without any manual intervention. When all the workers have been booted, the user informs the script on the master, which disseminates the set of host names and key pairs to every worker, allowing each node to execute remote commands on the others via ssh.[2] Finally, the master node formats a new Hadoop file system and starts the Hadoop daemons. At that point, the cluster is fully functional, and the user can submit Hadoop jobs to the master node.

B. Boot Performance

An important aspect of STOIC's performance is the time it takes to boot an OS. To quantify boot performance, we measured the boot time and number of bytes transferred while booting the Ubuntu live CD and Mininet VM images described above. Figure 2 shows the averages of these two metrics across ten boots. The measurements in Figure 2a demonstrate that STOIC boots quickly, with averages of 74 seconds and 35 seconds for Ubuntu and Mininet, respectively.

Related to boot time, STOIC would ideally transfer as few bytes as possible during the boot process. Any remaining data should be streamed on demand to allow the user to begin working as quickly as possible. Figure 2b shows the data transferred during the boot process for the same ten experimental runs. It additionally compares those values with the total size of each image. A streaming strategy is clearly effective: a user could boot the Mininet image four times before copying the same number of bytes as the total image.

[2]This is a common remote execution model for Hadoop and other widely-used cluster software.

TABLE I: A comparison of the theoretical and measured costs of transferring files with STOIC's persistent file storage.

Operation	Cost
PUT 1,000 times	$0.005
GET 10,000 times	$0.004
Theoretical cost to upload a 100-MB file	$0.004
Theoretical cost to download a 100-MB file	$0.00031
Measured cost to upload 100-MB file	$0.0041
Measured cost to download 100-MB file	$0.00035

C. Persistent Storage

To be of practical use, STOIC's persistent file storage mechanism must be both inexpensive and reasonably fast when paired with commercial cloud storage providers. This section examines the cost of transferring a large file from STOIC's file relay backed by Amazons S3 storage service. We measured the time and monetary cost of writing a 100 MB file to S3 in 128 KB chunks and subsequently reading that file back. In order to remove confounding effects of caching, we flushed OS caches between the write and read phases. Thus, this test effectively represents the worst-case I/O performance scenario where there are no cache hits.

Table I quantifies the theoretical and observed costs associated with streaming a 100-MB file, using a 4-GB file system overlay, through a STOIC relay. The values above the divider show the Amazon pricing model and theoretical costs of moving a 100-MB file using that model. Below that, we show the averaged results of five experimental runs. The transfer time was consistent, ranging from 41 to 49 seconds for the 100-MB file, and the measured costs closely reflect the theoretical minimums.

V. Related Work

Thin clients and VDI. STOIC shares the goals of many prior systems aimed at enabling resource-constrained "thin" clients, namely remote storage and on-demand data streaming. One of the objectives stated for the Multics project [21] was to provide a computation service to users similar to electric and telephone utilities. Another system, Sprite [22], supported diskless workstations, which booted from a remote file system. These and other systems laid the groundwork for STOIC, but they existed under local network conditions that differ significantly from today's cloud computing model.

Efforts like SLIM [23] and Lai & Nieh's wide-area thin client performance analysis [24] describe an environment that more closely resembles STOIC's, and similar to our experience with STOIC, their analyses show that streaming core system data over the wide area can yield a high-quality, interactive user experience. Both projects focus on outsourcing computation in addition to data storage, whereas STOIC assumes that clients are typical commodity PCs (i.e., fully capable processors) that wish to compute their own results and render their own graphics.

The Collective [25] proposes an operational vision that is most similar to that of STOIC. In the Collective, PCs serve as simple virtual appliance transceivers (VATs) that download and cache remotely-managed virtual appliance images. Remotely stored user profiles track which users are permitted to access each virtual appliance. Like STOIC, the Collective outsources the management of system and application data to an external party, leading to comparable benefits. In contrast to STOIC, PCs in the collective are still assumed to boot from a local storage device, and virtual applications are cached at on the VAT's local disk. STOIC makes no assumptions about the availability or contents of local storage. Further, in the Collective, each application is managed separately, as an independent VM container, whereas STOIC maintains an OS image at the granularity of an entire file system.

More recently, Virtual Desktop Infrastructure (VDI) has been gaining popularity in industry with services such as VMWare's Horizon [26] and Amazon's Workspaces [27]. Like STOIC, such services are easily accessible to users via public cloud providers. VDI differs from STOIC primarily in the location of computation. With STOIC, client machines perform their own processing, with the cloud representing storage and OS data. On the other hand, VDI clients offload computation to the cloud. We believe VDI and STOIC to be complementary, with benefits and use cases for both models.

Remote OS Installation and recovery. Several projects and commercial vendors have recently begun supporting Internet-based OS installation and maintenance tasks. The most notable example is Apple's OSX Internet Recovery [8], which serves as an OS data source for Apple hardware without removable media [28]. Open source projects like Cloudboot [29] and Netboot.xyz [30] provide similar functionality, via iPXE, to commodity PCs. These systems differ from STOIC in that they transfer an entire OS image prior to booting rather than streaming files on demand. Furthermore, their goal is to write an OS to a local disk and then disappear; they do not provide persistent storage options.

Cloud operating systems. Many projects have recently begun touting "cloud operating systems", which mainly concentrate on web-based applications. Though their details vary considerably, they relate to STOIC in that each is centered around a notion of streaming remotely-accessible data over a wide area. One such system is Google's Chrome OS [31], which provides a simple browser-based interface for web-based software as a service applications. Others like ZeroPC [32] aim to launch a persistent, desktop-like environment from within a user's web browser tab. We believe that these efforts, while interesting, are orthogonal to STOIC due to their focus on browser-based data distribution.

VI. Conclusion

In the STOIC model, operating systems are distributed as a service, and users to can optionally maintain persist files over multiple sessions using their preferred storage service. STOIC is efficient and readily compatible with existing cloud service provider infrastructure.

With a straightforward prototype implementation built from common, well-supported protocols, we have demonstrated several practical STOIC use cases for end users, system administrators, and software distributors. Further, we believe that the steady decline in cloud computing costs will make STOIC more attractive over time.

References

[1] Amazon, "Elastic Compute Cloud (EC2)," http://aws.amazon.com/ec2.
[2] Microsoft, "Windows Azure," http://www.windowsazure.com.
[3] Heroku, https://www.heroku.com.
[4] Salesforce.com, https://www.salesforce.com.
[5] Klint Finley, "Out in the Open: Inside the Operating System Edward Snowden used to Evade the NSA," Wired, 2014, http://www.wired.com/2014/04/tails.
[6] B. Lantz, B. Heller, and N. McKeown, "A Network in a Laptop: Rapid Prototyping for Software-Defined Networks," in *HotNets*, 2010.
[7] The Apache Software Foundation, "Hadoop Project," http://hadoop.apache.org.
[8] Apple, "About OS X Recovery," https://support.apple.com/en-us/HT201314.
[9] A. Borisov, "Coreboot at Your Service!" *Linux Journal*, vol. 2009, no. 186, October 2009.
[10] Intel Corporation, "Preboot Execution Environment (PXE) Specification," 1999.
[11] iPXE, http://ipxe.org.
[12] Y. Dong, H. Zhu, J. Peng, F. Wang, M. P. Mesnier, D. Wang, and S. C. Chan, "RFS: A Network File System for Mobile Devices and the Cloud," *ACM SIGOPS OS Rev.*, vol. 45, no. 1, Feb. 2011.
[13] J. Liang, A. Bohra, H. Zhang, S. Ganguly, and R. Izmailov, "Minimizing Metadata Access Latency in Wide Area Networked File Systems," in *IEEE HiPC*, 2006.
[14] A. Muthitacharoen, B. Chen, and D. Mazières, "A Low-bandwidth Network File System," in *ACM SOSP*, 2001.
[15] Neil Brown, "Overlay Filesystem," Linux Kernel Documentation, https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt.
[16] P. T. Breuer and A. Martn Lopez and Arturo Garca Ares, "The Network Block Device," *Linux Journal*, vol. 2000, no. 73es, May 2000.
[17] C. Sun, L. He, Q. Wang, and R. Willenborg, "Simplifying Service Deployment with Virtual Appliances," in *IEEE SCC*, 2008.
[18] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," *Linux Journal*, vol. 2014, no. 239, March 2014.
[19] C. Boettiger, "An Introduction to Docker for Reproducible Research," *SIGOPS Operating Systems Review*, vol. 49, no. 1, January 2015.
[20] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *USENIX OSDI*, 2004.
[21] F. J. Corbató, J. H. Saltzer, and C. T. Clingen, "Multics: The First Seven Years," in *Proceedings of the May 16-18, 1972, Spring Joint Computer Conference*, 1972.
[22] M. Nelson, B. Welch, and J. Osterhout, "Caching in the Sprite Network File System," *ACM TOCS*, vol. 6, February 1988.
[23] B. Schmidt, M. Lam, and J. D. Northcutt, "The Interactive Performance of SLIM: a Stateless, Thin-Client Architecture," in *ACM SOSP*, 1999.
[24] A. Lai and J. Nieh, "On the Performance of Wide-Area Thin-Client Computing," *ACM TOCS*, vol. 24, May 2006.
[25] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M. S. Lam, "The Collective: A Cache-Based System Management Architecture," in *USENIX NSDI*, 2005.
[26] VMWare, "Horizon," http://www.vmware.com/products/horizon.html.
[27] Amazon, "Workspaces," https://aws.amazon.com/workspaces.
[28] D. Frakes, "Hands on with Mountain Lion's OS X Recovery and Internet Recovery," http://www.macworld.com/article/1167870, July 2012.
[29] Cloudboot, http://cloudboot.org.
[30] Netboot, http://netboot.xyz.
[31] S. Pichai and L. Upson, "Introducing the Google Chrome OS," Google Official Blog, 2009, https://googleblog.blogspot.com/2009/07/introducing-google-chrome-os.html.
[32] ZeroPC, https://www.zeropc.com.