Blutopia: Stackable Storage for Cluster Management

Fábio Oliveira #1, Gorka Guardiola *2, Jay A. Patel ++3, Eric V. Hensbergen +4
#Department of Computer Science, Rutgers University
Piscataway, NJ 08854
[1]fabiool@cs.rutgers.edu

*Operating Systems and Networking Group, Rey Juan Carlos University
Madrid, Spain
[2]paurea@gmail.com

++Department of Computer Science, University of Illinois at Urbana-Champaign
Urbana, IL 61801
[3]jaypatel@cs.uiuc.edu

+Austin Research Lab, IBM Research
Austin, TX 78758
[4]ericvanhensbergen@us.ibm.com

***Abstract*— The complexity of today's computer systems poses a challenge to system administrators. Current systems comprise a multitude of inter-related software components running on different servers. In this paper, we propose the use of the *stackable storage* mechanism as the foundation of centralized systems management. At the management level, we show how this mechanism can be used to implement an infrastructure that allows administrators to perform typical tasks fast and effortlessly. In particular, we find that our prototype could have avoided 40% of the human mistakes observed experimentally by previous research. At the storage level, we identify three key characteristics of stackable storage that allow the definition of different policies with distinct performance and scalability behaviors. We quantitatively compare five storage policies under different workloads and conclude that stackable storage is a viable approach.**

I. Introduction

The computing infrastructure of data centers is becoming increasingly complex. Many such centers depend on software components distributed over a number of servers to run internal application suites or to provide on-line Internet services. These distributed components include database servers, email servers, identity management servers, application servers, Web servers, load balancers, and many others. This diversity of mutually dependent software complicates the fundamental tasks of cluster management, namely: initial installation, software upgrade, software/data migration, new hardware deployment, and cluster reprovisioning.

Depending on the nature of the service provided by the cluster, it is often the case that the administrator needs to deploy new hardware to expand the system capacity. In the case of multi-tiered configurations, sometimes it is necessary to rearrange the cluster — by exchanging roles of machines — in order to balance the workload across tiers.

Another recurring administrative task is software upgrade, which is typically motivated by the need for new functionality, important bug fixes, or security patches. Blindly trusting popular package and patch systems, such as apt-get, yum, and Windows installer, may not be appropriate since upgrades may fail due to unforeseen interactions with installed software components, not to mention unpredictable dependency problems.

All the aforementioned administrative tasks usually require bringing the service down partially or even entirely for software installation and cluster reconfiguration. Given that the cost of downtime can be enormous, these error-prone, tedious tasks have to be performed as quickly as possible to minimize the potential revenue losses or user dissatisfaction.

To make matters worse, humans invariably make mistakes, specially when under pressure. Recent studies have shown that human mistakes committed during system administration were a significant cause of online service disruptions and outages [1][2]. In particular, Oppenheimer and Patterson [1] studied three Internet services and found that human mistake was the largest cause of failures and the largest contributor to time to repair in two of the three services. In another study, Nagaraja *et al* [2] conducted experiments with 21 volunteer operators on a three-tiered online auction service. Analyzing 43 experiments in which the operators performed management tasks, the study detailed 42 mistakes committed by operators, even expert ones. Among other observations, the work found that 8 mistakes caused the service to be inaccessible, whereas 19 mistakes severely degraded the service throughput. Other reported effects of operator mistakes were security vulnerability, increased time to repair, reduced system capacity, incomplete component integration, and potential crash or inaccessibility of service components.

In this paper, we propose a cluster management infrastructure that facilitates the management tasks mentioned above, reducing both the risks of operator mistakes and the downtime required for maintenance. Our infrastructure is based on a centralized disk image management approach that relies on a mechanism which we call *stackable storage*. The name stack-

able storage stems from the idea of composing different views of the centralized storage system by stacking layers of disk image. In particular, each machine is given a logical volume that is made of layers of disk image. This mechanism allows for the segregation of machine-specific data and configuration from OS and application-specific images, thereby making it possible for multiple machines to share common, read-only layers. Typically, machines running the same OS version and requiring the same applications will share the corresponding layers, but will have their private, unique *personality* layers that rely on a copy-on-write mechanism.

Our cluster management model is motivated by four key observations. First, storage consolidation not only reduces the management costs by reducing the amount of storage devices that need to be managed, but it also reduces the energy consumed by the cluster infrastructure. Second, the stackable storage mechanism for building logical volumes is a flexible way of organizing the storage system that permits distinguishing between physical machines (the actual hardware of the servers) from logical machines (defined by the contents of their logical volumes). It is flexible enough to work either with or without virtual machine monitors. Third, our centralized management model guarantees that each machine is always assigned a consistent view of the storage system. Only after being tested are upgrades of applications released as new layers. To effectively upgrade the software running on a machine, a new disk volume is assigned to it. Since its layers have been previously tested together, the newly assigned disk volume is guaranteed to be free of any incompatibility that could result from a traditional upgrade process. Fourth, this model of management could leverage the existence of value-added resellers (VARs) who usually bundle operating system, middleware, and applications to provide a specialized solution to the end customers. With our infrastructure, the VARs could publish layers of disk image to be easily deployed/upgraded by the end customers or their local IT staff.

We have developed a prototype of the proposed infrastructure, which we called Blutopia. Besides automating the aforementioned administrative tasks, Blutopia provides version control of system changes, allowing the system administrators to checkpoint any logical volume state and rollback to any checkpoint previously taken. We used Blutopia to manage an IBM Blade Center with eight servers providing a prototype three-tiered online bookstore Internet service. Our experience with Blutopia shows that it is easy to use, and allows for the deployment of entire servers in as little time as it takes a server to reboot. It is equally easy and fast to augment the deployed infrastructure by creating new classes of servers (roles), assigning new roles to servers, exchanging roles between servers, upgrading existing roles, and rolling back software versions and configuration in the event of unexpected problems.

In order to quantify how effectively Blutopia can help system administrators, we thoroughly analyzed the logs of live operator experiments conducted and described by Nagaraja *et al* [2]. The architecture of the three-tier prototype auction service used in their work is similar to that of the bookstore service managed by Blutopia in our servers. We concluded that Blutopia would have prevented 40% of all human mistakes they observed. Furthermore, Blutopia would have prevented 57% of the mistakes that the technique they proposed — validation of operator actions — could not deal with.

Given the importance of stackable storage to our cluster management model, we investigated different approaches for implementing such a mechanism. In particular, we evaluated stackable storage across three dimensions: the *granularity* at which disk image layers can be stacked — block level or file system level —, the *location* of the stacking code — centralized storage server or clients —, and the type of *data transport protocol* used by the clients to access their logical volumes — file access protocol or block access protocol. We present a preliminary quantitative analysis of stackable storage across those three axes in order to unveil the design idiosyncrasies, performance behavior, and scalability of each possible combination. More important than our preliminary measurements, we view Blutopia not only as a cluster management infrastructure, but also as a testbed that allows its users to choose the stackable storage policy that best fit their environment.

In summary, our main contributions are:

- We propose the use of the stackable storage mechanism as a foundation for cluster management.
- We implement five variations of stackable storage, exploring the three dimensions we identified in the design space. We present a preliminary quantitative analysis of performance and scalability of stackable storage.
- We implement a prototype cluster management infrastructure based on stackable storage and quantitatively evaluate its ability to prevent system administrator mistakes.

II. Related Work

The abstraction of virtual disks provided by virtual machine monitors such as VMware [3] and Xen [4] allows for a management model that, like stackable storage, decouples physical servers from the functionality they offer. Unlike stackable storage, however, the composition and allocation of virtual disks are relatively static [5]. Ventana [5], a virtualization-aware file system, empowers virtual disks with support for fine-grained sharing, a functionality provided by Blutopia's stackable storage. Stackable storage is a generic mechanism that can be used in both virtualized and non-virtualized environments. In addition, Blutopia's infrastructure allows the users to choose the stackable storage policy that best fit their environment by selecting the right combination of stack granularity, stack location, and data transport protocol.

The emergence of large-scale server systems has led to a large number of network install and management systems. These range from disk imaging systems, such as Symantec Ghost [6], to more complicated solutions, such as Tivoli's Provision Manager for OS Deployment [7]. Blutopia differs from those systems by its simple, flexible management model, and higher coverage of management tasks.

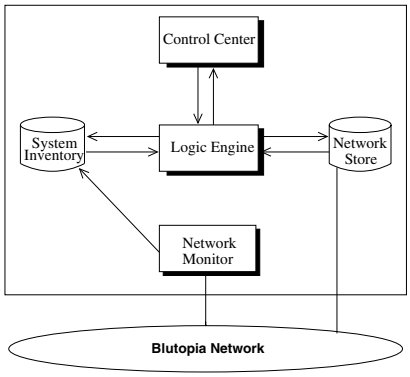

Fig. 1. *Blutopia architecture.*

Levanta [8] is a management appliance that simplifies the deployment of software in diskless machines and offers functionalities similar to Blutopia's by means of NFS and its own stackable file system called MapFS. Unlike Blutopia, it solely relies on a stackable file system; hence, it does not allow tuning the underlying storage policies. Along the lines of Blutopia's and Levanta's role-based management approach, the Collective [9][10] centralized management model based on virtual appliances permits users to logically migrate their working environment to different desktops; differently, virtual appliances are managed in a coarse-grained fashion, exhibiting less flexibility than stackable storage.

Another interesting aspect of our work is the evaluation of Blutopia's efficacy in helping system administration by potentially preventing some human mistakes. A similar study was conducted by Zheng *et al* [11] in the context of automatic configuration of Internet services. In fact, reducing the need for human intervention through automatic configuration [11] and validating operator actions before exposing them [2] are systems management approaches orthogonal to Blutopia's.

III. Management with Blutopia

Blutopia is a tool meant to manage a set of *services*. The services may range from online Internet services, such as a Web-based bookstore, to payroll and personnel management suites to be used internally by a business. A service comprises a set of *components* that work in conjunction, where each component is associated with a *role*. For example, a Web-based bookstore service might use five components: one component whose assigned role is *LVS (Linux Virtual Server) front-end load balancer*, three components whose roles are *Apache Web server*, and one component designated as *MySQL database server*. Blutopia also supports multiple *versions* of roles. For instance, the users might be able to choose among different versions of the role *Apache Web server*.

An important characteristic of Blutopia is extensibility. Users can create new roles, as well as new versions of a role (upgrades), making them available for future deployment. In a typical business setting, local IT staff would manage and deploy services, whereas contracted *publishers*, e.g. VARs, would supply new roles and upgrades.

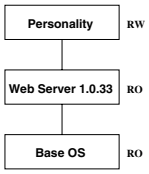

Fig. 2. *Layers of a Web Server logical volume.*

A. Architecture

Figure 1 depicts Blutopia's architecture. The Blutopia network embodies all machines running services managed by Blutopia, as well as the manager machine running Blutopia itself. As shown in Figure 1, Blutopia has three core components: Network Monitor, Control Center, and Logic Engine.

From a high-level perspective, users interact with Blutopia by means of a Web-based GUI, issuing commands to the Control Center which, in turn, invokes the functions exported by the Logic Engine. The Logic Engine effectively performs the actions needed to carry out the user commands. The commands available on the GUI are: install, upgrade, reprovision, checkpoint, rollback, and packager. The last command is used to upgrade an existing role or create a new one.

In the next few paragraphs, we briefly describe each core Blutopia component and how they interact with each other and with the Network Store and System Inventory (see Figure 1).

1) Network Monitor: The Network Monitor constantly monitors the network in order to detect new hardware, broken links, and service disruptions. When a new machine is detected, the Network Monitor updates the System Inventory (see Figure 1), a database that stores information about all machines belonging to the Blutopia Network. Whenever the Network Monitor detects a service disruption, it reports that the machines involved are unreachable. The connectivity status is displayed at all times on the user interface. The current prototype provides the Network Monitor functionality through scripts and programs that inspect the DHCP server log, as well as through the Ganglia monitoring infrastructure [12].

2) Control Center: Upon receiving a command from the user, the Control Center calls the corresponding Logic Engine agent. It also displays notification messages corresponding to status information it receives from the Logic Engine, e.g., when the Network Monitor detects a new machine. Our prototype Control Center is implemented as a set of PHP scripts that interface with AJAX Javascript GUI interfaces.

3) Logic Engine: The Logic Engine comprises a set of administrative agents that interact with the System Inventory and Network Store to implement the commands exposed to the users. In particular, it uses the System Inventory to, among other things, keep a record of the disk images used by each machine. The actual disk images (layers) corresponding to operating system, *roles*, and private machine *personalities* are kept in the Network Store and are remotely accessed by the managed machines, which we also call Blutopia clients.

B. Stackable Storage

We define stackable storage as a mechanism to stack layers of disk image in order to compose different views of a

centralized storage system, where each stack defines a logical volume on which a particular client machine relies. A storage server (or storage appliance) running Blutopia to maintain all layers exported to client machines is the single point of management.

Our management model is based on three types of disk image layers. On the bottom of all stacks is a *base operating system layer* which can be shared by all client machines. It consists of the base file system needed by the operating system.

The second type of disk image layer is the *role-specific layer*. This one lies on top of the base operating system layer complementing it with the file system portion whose contents define the role of the machine. A typical Blutopia installation provides a number of predefined roles tailored according to the needs of a particular organization. For instance, in a traditional multi-tier online Internet service, the roles available for deployment could be *Load Balancer*, *Web Server*, *Application Server*, and *Database Server*.

Finally, each machine has its private *personality layer* on the top of its stack. The purpose of this layer is to store all changes made to the contents of the logical volume of a machine by means of a copy-on-write mechanism; as such, it is the only layer that gets modified throughout the lifetime of the logical volume. The contents of the role-specific and base operating system layers remain intact since they are supposed to be shared.

In our multi-tier Internet service example, the logical volume of a machine that has been assigned the role *Web Server* would comprise three layers, namely: a personality layer on top of the Web Server layer on top of the base operating system layer. This logical volume is illustrated in figure 2; on the right of each depicted layer is its status — read-only or read-write. The Web Server machine is oblivious to the fact that its logical volume is formed by three layers. Internally, however, the stackable storage mechanism guarantees that all changes made to the contents of the logical volume are confined in the personality layer, allowing the Web Server layer to be shared by all machines that have been assigned that role, and the base layer to be shared by all machines running that operating system. By merging the logical volume layers and taking into account the changes stored on the personality layer, the stackable storage mechanism gives the machine the illusion of a united file system on a conventional storage device.

Besides distinguishing and exporting multiple role-specific layers, Blutopia also associates versions with each layer.

C. Management Tasks

The abstraction of stackable storage, although conceptually simple, is a powerful feature that facilitates the sharing of common base and role-specific disk images, enables the storage server to obtain instantaneous snapshots of all logical volumes as well as to quickly rollback to a previously taken snapshot, and makes it trivial to perform fast cluster reprovisioning through simple role re-assignments. In the next paragraphs we describe how Blutopia uses stackable storage to handle typical cluster management tasks.

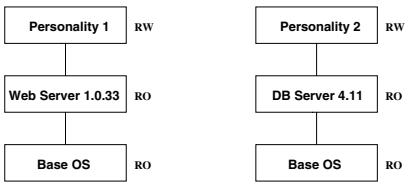

Fig. 3. *Web Server is transformed into a Database Server.*

1) Machine Installation: For the administrator, installing a machine is as easy as selecting a role and instructing Blutopia to assign it to the machine. As a result, the appropriate stack will be built and exported as a logical volume.

2) Reprovisioning: We refer to reprovisioning as changing the role of a machine. At the storage level, two steps are performed to carry out this action. First, the machine's current role-specific layer is replaced with the layer corresponding to the new role. Second, in order for the administrator to be able to rollback to the state before the reprovisioning, the current personality layer is saved as part of the repository of checkpoints and a new personality layer is assigned to the machine. By saving the current personality layer Blutopia can recreate the original stack if required by the administrator. Figure 3 depicts a Web Server being reprovisioned as a Database Server. On the left is the original storage stack that is replaced with the logical volume whose composition is shown on the right.

3) Checkpoint and Rollback: To create a checkpoint of a logical volume, Blutopia performs three actions, namely: (1) it saves the current top-most personality layer along with the stack configuration as the new checkpoint; (2) it transforms the current top-most personality layer into a read-only layer; and (3) it adds a new personality layer onto the stack. Actions (2) and (3) guarantee that changes made to the logical volume after the checkpoint was taken will be confined in a new personality layer, thereby ensuring that all checkpoints always remain valid.

In rolling back a logical volume to a previous snapshot, Blutopia first recreates the logical volume based on the stack configuration saved as part of the checkpoint. Next, it transforms the personality layer of the recreated stack into read-only and inserts a new personality layer on the top of the stack. Again, adding a new personality layer ensures that all checkpointed states remain valid at all times.

To illustrate the rollback operation, let us suppose that a certain machine has its logical volume configured as shown in figure 4(a), and that the system administrator wishes to rollback to a snapshot corresponding to the stack shown in figure 4(b). The logical volume given to the machine as a result of the rollback operation will have the layering configuration depicted by figure 4(c). Note the new personality layer on top of the stack and the change of the status of the previous personality.

4) Role Publishing: The idea of role publishing is to add new roles or create upgrades for existing ones so that the new roles and upgrades are made available for future deployment. At the storage level, each new role or new version of a

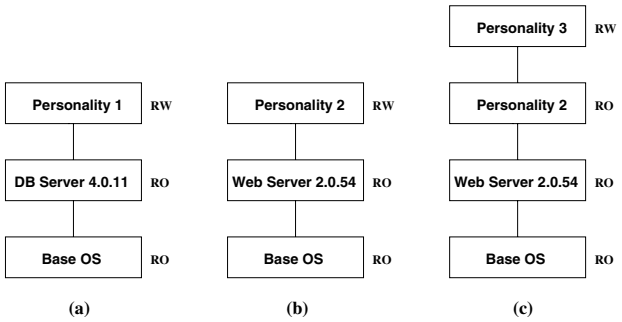

Fig. 4. *Rollback operation: (a) stack before rollback; (b) stack corresponding to the snapshot chosen for rollback; (c) new logical volume created as a result of rollback.*

role (upgrade) translates into a role-specific layer that can be selected to be part of logical volumes.

In our model of stackable storage, adding a new role can be done quite easily. To that end, the system administrator might use a machine whose logical volume is formed by only two layers: an empty personality layer on top of the base operating system layer. Then, the administrator effectively installs the software intended to be the new role. Once the installation is completed and tested, the administrator commands Blutopia to capture the personality layer of the logical volume on which the installation was performed and to save it as a new role-specific layer with a supplied name and version.

Although adding new roles is trivial, the creation of upgrades bears subtleties that are worth discussing. There are two ways of reasoning about upgrades, depending on the configuration of the logical volume from which they are derived. In its simplest form, a new version of a role (an upgrade) is derived in the same manner as described above. New roles and upgrades so derived depend only on the base layer; in other words, they will always be stacked on top of the base layer in logical volumes.

Due to convenience of installation (e.g., the existence of a small patch) or space efficiency considerations, upgrades might be derived directly from the role to be upgraded, as opposed to from the base layer. To clarify this issue, let us suppose that the administrator wants to create an upgrade for the Web Server role. Instead of installing the new version from scratch, i.e., from the base layer, the administrator might apply a patch to the current version of the role. To do so, a machine with a logical volume containing an empty personality layer on top of the layer corresponding to the current version of the Web Server role should be used. Once the patch has been applied and tested, the administrator instructs Blutopia to capture the personality layer of the logical volume and to save it as a new version of the Web Server role. Unlike the scenario described in the previous paragraph, upgrades derived in this fashion depend on the previous version; therefore, the layer corresponding to the newly created upgrade must always be stacked on top of the layer containing the upgraded version. Figure 5 depicts the creation of an upgrade for the Web Server role. The version 3.1.1 was derived from the version 2.0.54. All stacks where the layer Web Server 3.1.1 appears will have

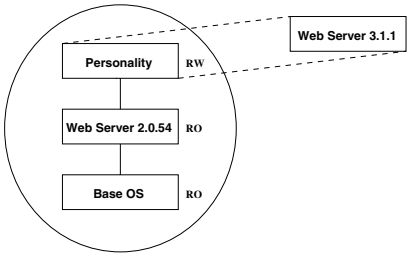

Fig. 5. *Creation of an upgrade for the role Web Server. The personality layer is captured and saved as Web Server version 3.1.1.*

the layer Web Server 2.0.54 under it. Needless to say, in order to ensure that the stacks satisfy these dependency rules, Blutopia records them.

5) Upgrade Deployment: Once an upgrade has been created and published as discussed above, it becomes available for deployment. Similar to upgrade creation, there are two different approaches to upgrade deployment. One possibility is to treat it as a reprovisioning, in which case the personality layer of the original logical volume is not kept in the resulting stack, as illustrated in figure 3. The assumption here is that the contents of the personality layer are not relevant in the context of the new stack configuration. Although this rationale makes sense when the role of a machine is changed, it may be unreasonable when the role remains the same. To accommodate the situations where it is desirable to maintain the current personality layer, the stack resulting from an upgrade deployment would comprise the base layer, the layer corresponding to the upgrade (observing the dependency rules), the original personality layer with read-only status, and a new writable personality layer at the top. An example of this scenario is shown in figure 6, where the upgrade created in figure 5 is effectively deployed. Note that the former personality layer is part of the resulting stack.

It should be clear that it is possible for a stack to have more than two personality layers. This happens, for instance, when an upgrade is deployed on a logical volume that has more than one personality layer. As a consequence of this multitude of personality layers, dependency rules should apply to them. For example, in figure 6, *Personality 2* depends on *Personality 1*.

6) Final Remarks: The tasks of checkpoint/rollback and upgrade deployment reveal a stackable storage issue: a stack can become large as checkpoints are taken and upgrades are deployed because a new personality layer is pushed onto the stack every time these tasks are performed. Another potential cause of large stacks are upgrades derived from a previous version of a role, as shown in figure 5. Although the current Blutopia prototype does not have the ability to coalesce adjacent layers, implementing such a feature should not pose major challenges. Besides, a previous work on UnionFS, a stackable file system for the Linux operating system, found that it is highly scalable [13]. We leverage UnionFS to implement stackable storage at the file system level, as described in Section IV.

There is another issue regarding upgrade deployment. Suppose that a certain machine is assigned the role *Web Server*.

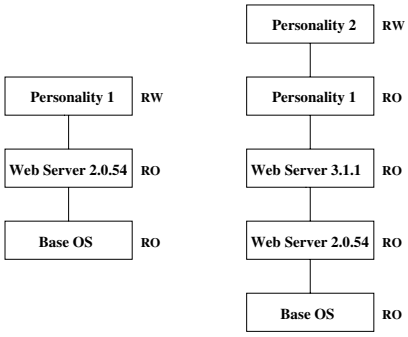

Fig. 6. *Role Web Server is upgraded from version 2.0.54 to 3.1.1.*

Later on, a configuration file is modified for performance tuning; therefore, such changes will be part of the personality layer of the machine. Some time later, the administrator decides to deploy a newly available version of the role *Web Server*. Two events might happen next. If the upgrade is deployed with the option of not keeping the personality layer, as commented in subsection III-C.5, the changes made to the configuration file will be lost. Alternatively, if the personality layer is kept, the changes to the original configuration file will be seen because a layer has higher precedence than any layer located under it in the stack. The former situation is acceptable if the format of the configuration file in the new version differs from that of the previous version, but it is arguably undesirable otherwise. Our current prototype is oblivious to this issue. Blutopia could be extended to associate a list of critical configuration files, provided by the *publisher*, with each role. Then, during an upgrade deployment, Blutopia could show the differences between the two versions of the affected files resulting from the upgrade, before actually carrying it out, in order to help the administrator fix any potential problem.

Finally, one might argue that different roles might require either different kernel versions or different kernel configurations. There is nothing preventing Blutopia from handling different base operating system layers.

IV. Stackable Storage Implementation

In conceiving our stackable storage abstraction, we identified three dimensions in the design space. One dimension is *stack granularity*, i.e., the granularity at which the disk image layers are created and stacked. The possible options here are to manage layers either at the file system level or at the block level.

Another important aspect to consider is *stack location*. Although our approach to disk image manipulation relies on data stored on a centralized server or storage appliance running Blutopia, the code that implements the stacks can run either on the server or on the Blutopia clients. That is what we mean by stack location. Also, recall that, in this context, we refer to clients as the machines managed by Blutopia.

Finally, *data transport protocol* is yet another facet of stackable storage, the options being to employ a file transport protocol (e.g., NFS) or a block transport protocol (e.g., iSCSI). To appreciate the importance of data transport protocol as a component of the design space, one should realize that *stack granularity* does not necessarily dictate a particular type of protocol. Clients may access data by means of a block transport protocol even if the layers are managed at the file system level; the converse is also true.

In this section, we discuss all possible combinations in the design space, grouping them by *stack granularity*. We first delve into stacking at the file system level and, in the sequel, describe block stacking.

A. File System Stacking

File systems that support the notion of namespace unification, such as UnionFS [13], can be used to implement stackable storage. In this work we leverage UnionFS not only due to its widespread use, but also because the level of abstraction it offers is well suitable for our conception of stackable storage.

UnionFS was designed for Linux kernel series 2.4 and 2.6. It is actually a piece of software lying between the Linux VFS (Virtual File System) and the lower-level file systems (e.g., ext2, ext3, or ReiserFS) that captures calls made to VFS and modifies the behavior of the corresponding operations in order to implement the functionality of namespace unification.

The main idea behind namespace unification is to merge different directory subtrees (possibly belonging to different file systems) into a single unified view. This can be done recursively for subdirectories (deep unification) or just for one level of directories. The latter is more efficient but it does not fit our needs. UnionFS does deep directory unification.

In UnionFS terminology, each merged subtree is called a *branch*, and the resulting merged file system is called a *union*. We treat each merged branch as a layer, and the resulting union as a stack. In Blutopia, the base operating system layer, the role-specific layers, and the personality layers are separate branches. When building a stack, Blutopia merges the appropriate layers using UnionFS in such a way that each layer has higher precedence than the one located under it. This is of paramount importance to ensure that the top-most personality layer — the only one that gets modified — is always the most influential in the logical volume contents perceived by the client. Hence, if a file gets deleted, a whiteout (a "negative file") is created on the personality layer indicating that the corresponding file should not appear on the logical volume; if a file gets modified, the version of the file stored on the personality layer is the one accessible from the logical volume; if a new file is created, it will be present only on the top-most personality layer and, therefore, will naturally show up on the logical volume.

In the realm of stackable storage based on file system stacking, there are three design options: server side stacking with file transport protocol; client side stacking with file transport protocol; and client side stacking with block transport protocol. The fourth combination, server side stacking with block transport protocol, is unfeasible. In the next paragraphs we describe each combination.

1) Sever Side Stacking and File Transport Protocol: In this configuration, Blutopia merges the layers on the server and exports the stacks to the clients. As previously mentioned, our

prototype uses UnionFS for namespace unification, and NFS is the file transport protocol we explored, due to its popularity. To make a given stack available to a particular client, the server builds it, mounts it on its local file system, and exports the mount point through NFS to the appropriate client.

2) Client Side Stacking and File Transport Protocol: This approach differs from the previous one in that the server does not export the stacks to the clients; instead, it exports the layers and the clients build their stacks by running UnionFS locally. It is up to the clients to mount through NFS all layers required to build their stacks.

Client side stacking enables two policies for managing the private personality layers. A client can either access its personality layer solely via NFS or keep a local copy. The latter has the potential to increase performance by providing an additional level of caching.

3) Client Side Stacking and Block Transport Protocol: In our prototype we used iSCSI as the block transport protocol. Like NFS, iSCSI enables the access to servers by means of IP-based networks. Differently, however, while an NFS server exports directories to clients, an iSCSI server makes a portion (or all) of its local disk(s) available to clients, which, in turn, operate on the remote disk(s) in exactly the same way as they would use a local disk. In particular, a Linux iSCSI server exports block devices to clients, where a block device might be a disk partition — e.g., `/dev/sda5`, or `/dev/sda6`.

Block transport protocol and file system stacking can coexist, provided that the stacking is done at the client side and the block devices exported by the storage server are not shared for writing. To satisfy the latter requirement, the server maintains one block device containing all shared layers (base operating system layers and all role-specific layers) and exports it to the clients. Since these layers represent the read-only portion of all stacks, the clients can safely import the block device and access it through iSCSI.

Regarding the personality layers, there are two options for managing them. Either the server provides one block device per personality layer, or the clients keep their personality layers locally. Local personality layers can potentially lead to overall better performance since they would provide an additional level of caching.

B. Block Stacking

An alternative to stacking directory subtrees by means of namespace unification is to build the stacks at the level of disk blocks. This variation of stackable storage is implemented at a lower level of abstraction, with no file system involvement.

In Linux, the target platform of our prototype, block-level stacking can be accomplished by treating block devices (e.g., `/dev/sda5`) as layers of the stacks. In this case, since two or more block devices are part of a stack, stacks are treated as logical block devices which, in turn, represent the logical volumes exported to the clients. This idea introduces the concept of *stackable block devices*.

Linux LVM (Logical Volume Manager) [14] could potentially be used to implement *stackable block devices*, as it supports a copy-on-write mechanism for taking snapshots of logical volumes. It was conceived as a backup mechanism that freezes the file system for a short period of time to backup the blocks as they are modified. However, implementing block stacking efficiently requires only writing to the appropriate layer of the stack and updating the metadata. To make matters worse, the metadata used by LVM for block addressing is not appropriate for situations where a large number of blocks are modified. We indeed confirmed that LVM's performance was below our expectations by measuring the performance of writing to a LVM snapshot using the Bonnie++ benchmark. We found that a stacked block device exhibited a performance degradation of an order of magnitude compared to a non-stacked block device.

For the reasons stated above, we implemented our own block stacking driver for Linux kernel 2.6 to realize the idea of *stackable block devices*. Our driver is used by the device-mapper [15], a component of the Linux kernel that creates logical block devices by means of published drivers called *targets*. Each logical block device created by the device-mapper by means of our driver is actually a stack of block devices that Blutopia exports as a logical volume.

Such a logical block device is seen as a unidimensional array of blocks. Internally, associated with each layer of the stack is a bitmap indicating which blocks are to be obtained from each layer. For instance, the bitmaps may indicate that block 10 is on the personality layer and block 90 is on the base operating system layer. A newly created logical volume has its personality layer empty; as the contents of blocks change, they are written to the personality layer and the bitmaps are updated accordingly. This scheme provides a one-to-one static block mapping which wastes space. There is a compromise between incurring the cost associated with sparse allocation and wasting space. We have left the sparse allocation out of the picture on purpose to isolate the variables of the study.

Stackable storage at the granularity of blocks lends itself to two combinations of *stack location* and *data transport protocol*, as described in the next few paragraphs.

1) Server Side Stacking and Block Transport Protocol: In this configuration, the server stacks the block devices and exports the resulting logical volumes to the clients. Our prototype, as mentioned before, adopts iSCSI as the block transport protocol. Therefore, the logical block devices created by the server are exported through iSCSI to the clients.

2) Client Side Stacking and Block Transport Protocol: In this variation of block stacking, the server exports through iSCSI the block devices to be stacked (the layers). The clients then import the block devices they need and build their stacks.

Stacking on the client side allows each client to keep the block device representing its personality layer locally, as opposed to having the server export it. This might improve the overall data access performance due to extra caching.

V. Evaluation

A. Human Mistakes Avoidance

One of the goals of Blutopia is to simplify system administration. One way of assessing its ability to do so is

to compare how many system administrator mistakes are committed with and without Blutopia. To that end, we studied the system administrator tasks and mistakes described extensively by Nagaraja *et al* [2]. In this work, the authors conducted experiments with human subjects, asking them to perform management tasks on a prototype three-tiered auction service modeled after Ebay, whose architecture is based on the typical Web (Apache), application (Tomcat servlet server), and database (MySQL) tiers. We analyzed the logs of user actions recorded and made available by the authors, as well as the mistakes they observed. We modeled their three-tiered service in terms of Blutopia abstractions and identified the steps the human subjects would have followed to perform the management tasks had they used Blutopia.

We consider all six tasks described in their work, three of which are the following scheduled maintenance tasks: to add an application server to the second tier; to upgrade the database machine; and to upgrade one Web server. The other three are diagnose-and-repair tasks that cover software misconfiguration, application crash/hang, and hardware fault.

For the task *upgrade the database machine*, all five mistakes made would have been avoided by Blutopia. Four of those mistakes involve misconfigurations that happened while the database contents were dumped from the old database machine to be later re-imported into the new one, and while a new instance of MySQL was installed and configured. With Blutopia, *upgrade the database machine* would simply entail to assign the role *MySQL Database Server* to the new machine, reusing the stack configuration of the old machine. The other mistake made during this task was restarting the wrong version of Apache on the Web servers while bringing the service back on once the upgrade has been finished. This mistake would have been prevented because two versions of a role would not coexist on a server managed by Blutopia. In the experiments conducted with human subjects, this task took 2 hours and 20 minutes on average due to the huge size of the database and the time it took the operators to fix the system after their own mistakes. With Blutopia, the duration of this task would be the amount of time it takes the new machine to boot up.

In the case of the task *upgrade one Web Server*, five out of seven committed mistakes would have been prevented by Blutopia. Those five mistakes have to do with the operators installing the new Apache version while keeping the previous one. As a result, some people forgot to update the parameters of a few scripts that were pointing to directories used only by the old Apache version. Since two versions of a role do not coexist in a single server, that problem would not have occurred. The two mistakes that Blutopia would have overlooked relate to configuration parameters that were either not present in the previous Apache version, or exhibited syntactical differences in the new one.

Seven other instances of the mistake of starting a wrong software version, causing a serious service disruption, occurred in the other tasks. This was the second most common mistake described by Nagaraja *et al* and would have been eliminated by Blutopia. Interestingly, Blutopia would have avoided eight out of fourteen mistakes that the technique of operator action validation (proposed by the authors of that work to deal with operator mistakes) could not detect. Those eight mistakes were a result of bad storage organization affecting upgrade tasks (hardware or software upgrades); hence, Blutopia would significantly enhance operator action validation by adding coverage to an important category of mistake.

Besides operator action validation, another approach that is orthogonal to Blutopia in simplifying systems management is automatic configuration of services [11].

B. Performance Evaluation

This section presents a preliminary experimental evaluation of the stackable storage flavors previously described. We compare them with respect to performance and scalability.

We conducted our experiments on nine machines, each one equipped with two dual-core 2.4 GHz AMD Opteron processors, with 1 MB of cache per core, 4 GB of RAM and a Broadcom NetXtreme BCM5704S Gigabit Ethernet card. We used one of the machines as the Blutopia server, and the remaining eight as clients. The server has two 73 GB ST973401LC SCSI disks, operating at 10,000 RPM with an average seek time of 4.1ms, as well as an LSI53C1010R Ultra160 SCSI controller.

The server runs the Linux Ubuntu Dapper distribution with kernel 2.6.17. All clients are diskless and boot from a separate server that provides them with NFS-roots based on the Red Hat Enterprise Linux distribution version 4 and kernel 2.6.17. All 9 machines of our testbed and the NFS-root server are connected to the same Gigabit Ethernet switch.

All experiments using NFS as the file transport protocol rely on NFSv3 mounted asynchronously. Regarding iSCSI, the server runs iSCSI Enterprise Target version 0.4.13 and the clients use Open-iSCSI version 1.0-485 as the iSCSI initiators. TCP is the transport protocol used for both NFS and iSCSI. For implementing the stackable file system, we use the CVS snapshot "20060616-1848" of UnionFS version 1.2. *Ext2* is the file system adopted to form the layers of our UnionFS-based stacks and to format the block devices used with iSCSI. One of the disks of the server is dedicated to our experiments, while the other is used only by the operating system.

In terms of stack composition, our evaluation takes place on a three-layer configuration comprising the base operating system image, a role-specific layer (Apache Web Server image), and the personality layer. All disk read operations are satisfied with data residing on the personality layer so that we can isolate and properly compare the overhead of layer lookup of both file system and block stacking. Therefore, our numbers do not account for the time it takes to copy a file from a read-only layer to the personality; this is done by the file system stacking approach (using UnionFS) whenever a file not present in the personality layer is modified. Wright *et al* [13] conducted a detailed performance analysis of UnionFS.

In order to evaluate the performance and scalability of our different stackable storage configurations, we chose two widely used benchmarks: Postmark and Bonnie++. Since the

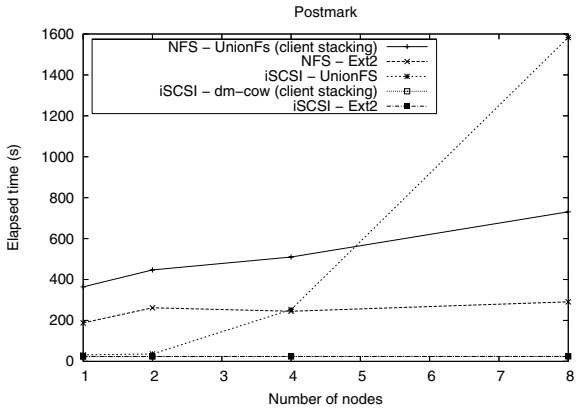

Fig. 7. *Postmark results for client-side stacking.*

former is metadata-intensive and the latter is data-intensive, our preliminary evaluation covers these two fundamentally different cases. In the sequel, we present the results of our experimental evaluation.

1) Postmark Results: The Postmark benchmark first creates an initial pool of files of varying sizes. After creating the pool, it enters the so-called transaction phase, during which it randomly chooses one of two types of file operations to perform: (1) create or delete a file; (2) read a file or append data to a file. In our Postmark runs, we configured it to generate a pool of 20,000 files, whose sizes range from 500 to 1,024 bytes, and to perform a total of 200,000 operations. The two types (and subtypes) of file operations were configured to be selected with equal probability. This workload mimics the use of multiple small short-lived files, which is common in applications such as online web-based e-commerce and electronic mail.

Figure 7 shows the runtime of Postmark for three stackable storage configurations: block stacking with block transport protocol (iSCSI-dm-cow), file system stacking with block transport protocol (iSCSI-UnionFS), and file system stacking with file transport protocol (NFS-UnionFS). In all cases, the stack is built at the client side. The graph also shows the baseline for iSCSI (iSCSI-Ext2) and NFS (NFS-Ext2). The plotted data points represent runs with 1, 2, 4, and 8 client machines.

Comparing the baseline results for the two data transport protocols, we see that iSCSI performs roughly an order of magnitude better than NFS. This is the result of NFS sending all metadata operations over the network to be resolved by the file system executing on the server, making the server disk operate at 75% of its maximum number of transfers per second, even in the single-client scenario. By contrast, in the iSCSI case, file system operations are resolved locally — primarily from the system's cache. Furthermore, since iSCSI volumes are implicitly exclusive, no coherence traffic is required, allowing for very aggressive caching of both metadata and file contents. A previous work presented a detailed comparative analysis of NFS and iSCSI [16].

UnionFS combined with NFS doubles the Postmark runtime as compared to the baseline, chiefly because it increases the number of metadata operations due to additional lookups in the multiple stack layers, thereby exceeding the maximum number of transfers per second the disk can handle. Differently, UnionFS combined with iSCSI behaves much better for one and two clients, approximating the performance of the iSCSI base case. However, starting at 4 clients, the additional metadata operations issued by UnionFS begin to disrupt the iSCSI cache resulting in a dramatic increase in overhead. This problem is more pronounced in the 8-client case in which UnionFS on iSCSI performs worse than UnionFS on NFS. Therefore, we conclude that UnionFS on NFS scales better than UnionFS on iSCSI for metadata-intensive workloads.

When block stacking is used with iSCSI, there is no noticeable overhead: the curves labeled iSCSI-Ext2 and iSCSI-dm-cow coincide. This is explained by the relatively small footprint of Postmark, which allows the entire bitmap for the block stack to be cached in memory, and by the natural match between iSCSI and block stacking. Combining all of that with iSCSI's aggressive caching yields excellent results for metadata-intensive workloads.

For server side stacking configurations, our results are similar to those shown in figure 7, exhibiting the same trends. Note that server side stacking is not applicable to iSCSI-UnionFS.

The behavior of the configuration NFS-UnionFS is not noticeably affected by stack location when up to 8 clients are used. To further investigate the effect of stack location, we conducted Postmark runs with 16 and 32 clients. Since we had only 8 client machines, we ran 2 and 4 instances of Postmark per machine to emulate 16 and 32 clients, respectively. The results of these experiments are shown in figure 8. Matching our intuition, offloading the file system stacking code to the clients is beneficial when a large number of clients run metadata-intensive workloads.

We were not able to run the large-scale experiments with iSCSI as they would require 16 and 32 different block devices for the personality layers, i.e., 16 and 32 partitions. Unfortunately, Linux does not support more than 15 partitions per SCSI disk, and we did not have room in our server for an extra SCSI disk. Having multiple instances of Postmark running on the same node share a mount point for the personality layer is not an option either, since local caching effects would render the results incomparable to the previous ones.

2) Bonnie++ Results: The Bonnie++ benchmark creates a file of a specified size and performs a number of character-based and block-based operations on it, emulating a database type of workload. In our experiments, we explored the block-based operations. Bonnie++ has three distinct phases: (1) file creation through a sequential write; (2) file rewrite, during which each block is read, dirtied, and rewritten, requiring a seek operation; and (3) file read, when the whole file is sequentially read. In order for the results to be valid, the size of the file is recommended to be twice the size of the memory.

We comment only on Bonnie++ results for the single-client case inasmuch as we observed that multiple running instances

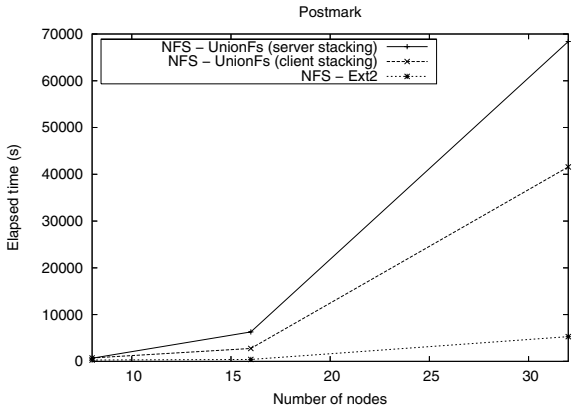

Fig. 8. *Benefit of client-side stacking.*

of Bonnie++ caused the disk seek time to dominate the results, as the disk head moved to satisfy requests for blocks belonging to distinct (and huge) Bonnie++ files.

Unlike the Postmark case, our Bonnie++ experiments did not reveal any significant difference between the baselines iSCSI-Ext2 and NFS-Ext2. The data-intensive nature of Bonnie++ outweighed the overhead of NFS metadata operations. For the same reason, the extra overhead of UnionFS combined with NFS was only 18% on average, which is much less than the performance penalty of 93% that UnionFS inflicted on NFS in the single-client Postmark run.

Regarding the iSCSI configurations, block stacking again did not cause any noticeable overhead, whereas UnionFS decreased the performance by 11% on average as compared to the baseline iSCSI-Ext2.

Due to space constraints, we do not present the results of the experiments with Bonnie++ in any further details.

3) Final Remarks: A previous study has quantified the overhead that large UnionFS stacks impose to the local file system and concluded that it scales well [13]. We used Postmark and a few micro-benchmarks to perform a similar study with NFS-UnionFS, varying the stack size up to 16 layers, and found that network latency was the dominating factor.

One problem with pure file system stacking is the potential performance degradation caused by copying huge files to the personality layer. With UnionFS, even a single block change on a file located below the personality layer causes the entire file to be copied. This problem did not manifest in our experiments because of the way we placed the data on the stacks. This performance penalty is unacceptable for systems like database servers, where large files are predominant.

Concerning internal storage manageability, one can argue that file system stacking is more flexible because the individual layers can be easily manipulated with conventional file tools. In block stacking, on the other hand, each layer depends on the layer immediately under it; therefore, the layers cannot be mounted independently. However, our experience with Plan 9's Venti [17] indicates that with appropriate tools and minimal metadata dealing with multiple levels of disk blocks can be made as simple as manipulating individual layers of a file system stack. Exploring a hybrid stackable storage that combines the flexibility of stackable file systems with the performance of stackable block devices is an interesting avenue for future research.

VI. Conclusions

In this paper, we proposed the mechanism of stackable storage as the basis for a centralized cluster management model. Using our prototype we performed a comparative analysis of five different stackable storage policies in terms of performance and scalability under metadata and data-intensive workloads. We also assessed the efficacy of our infrastructure in simplifying cluster management. Our evaluation showed that: (1) our system can potentially eliminate some types of human operator mistakes and speed up the execution of management tasks; (2) the stackable storage performance behavior is not an impediment to its use in real systems.

References

[1] D. Oppenheimer, A. Ganapathi, and D. Patterson, "Why do Internet Services Fail, and What Can Be Done About It," in *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS'03)*, Mar. 2003.

[2] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen, "Understanding and Dealing with Operator Mistakes in Internet Services," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*, Dec. 2004.

[3] G. Venkitachalam and B. Lim, "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor," in *Proceedings of the USENIX Annual Technical Conference (USENIX'01)*, 2001.

[4] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer, "Xen and the Art of Virtualization," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'03)*, Oct. 2003.

[5] B. Pfaff, T. Garfinkel, and M. Rosenblum, "Virtualization Aware File Systems: Getting Beyond the Limitations of Virtual Disks," in *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI'06)*, 2006.

[6] Symantec, "Put Imaging to Work for You," http://sea.symantec.com/content/article.cfm?aid=99, 2004.

[7] Tivoli, "Tivoli Provisioning Manager for OS Deployment," http://www.rembo.com/index.html, 2006.

[8] "Levanta Intrepid M," http://www.levanta.com/, 2005.

[9] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M. S. Lam, "The Collective: A Cache-Based System Management Architecture," in *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI'05)*, May 2005.

[10] C. Sapuntzakis, D. Brumley, R. Chandra, N. Zeldovich, J. Chow, M. S. Lam, and M. Rosenblum, "Virtual Appliances for Deploying and Maintaining Software," in *Proceedings of the Large Installation Systems Administration Conference (LISA'03)*, Oct. 2003.

[11] W. Zheng, R. Bianchini, and T. Nguyen, "Automatic Configuration of Internet Services," in *Proceedings of Eurosys 2007*, Mar. 2007.

[12] M. Massie, B. Chun, and D. Culler, "Ganglia Distributed Monitoring System: Design, Implementation, and Experience," *Parallel Computing*, vol. 30, no. 5-6, June 2004.

[13] C. P. Wright, J. Dave, P. Gupta, H. Krishnan, D. P. Quigley, E. Zadok, and M. N. Zubair, "Versatility and Unix Semantics in Namespace Unification," *ACM Transactions on Storage (TOS)*, vol. 1, no. 4, November 2005.

[14] "Logical volume manager," http://sources.redhat.com/lvm2/.

[15] "Linux device-mapper," http://sourceware.org/dm/.

[16] P. Radkov, L. Yin, P. Goyal, P. Sarkar, and P. Shenoy, "A Performance Comparison of NFS and iSCSI for IP-networked Storage," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'04)*, Mar. 2004.

[17] S. Quinlan and S. Dorward, "Venti: A New Approach to Archival Storage," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'02)*, 2002.