Stork: Package Management for Distributed VM Environments

Justin Cappos, Scott Baker, Jeremy Plichta, Duy Nyugen,
Jason Hardies, Matt Borgard, Jeffry Johnston, John H. Hartman
Department of Computer Science
University of Arizona
Tucson, AZ, 85721
stork@cs.arizona.edu

Abstract

In virtual machine environments each application is often run in its own virtual machine (VM), isolating it from other applications running on the same physical machine. Contention for memory, disk space, and network bandwidth among virtual machines, coupled with an inability to share due to the isolation virtual machines provide, leads to heavy resource utilization. Additionally, VMs increase management overhead as each is essentially a separate system.

Stork is a package management tool for virtual machine environments that is designed to alleviate these problems. Stork securely and efficiently downloads packages to physical machines and shares packages between VMs. Disk space and memory requirements are reduced because shared files, such as libraries and binaries, only require one persistent copy per physical machine. Experiments show that Stork reduces the disk space required to install additional copies of a package by over an order of magnitude, and memory by about 50%. Stork downloads each package once per physical machine no matter how many VMs install it. The transfer protocols used during download improve elapsed time by 7X and reduce repository traffic by an order of magnitude. Stork users can manage groups of VMs with the ease of managing a single machine – even groups that consist of machines distributed around the world. Stork is a real service that has run on PlanetLab for over 4 years and has managed thousands of VMs.

1 Introduction

The growing popularity of virtual machine (VM) environments such as Xen [3], VMWare [20], and Vservers [11, 12], has placed new demands on package management systems (e.g. apt [2], yum [25], RPM [19]). Traditionally, package management systems deal with installing and maintaining software on a single machine whether virtual or physical. There are no provisions for inter–VM sharing, so that multiple VMs on the same physical machine individually download and maintain separate copies of the same package. There are also no provisions for inter–machine package management, centralized administration of which packages should be installed on which machines, or allowing multiple machines to download the same package efficiently. Finally, current package management systems have relatively inflexible security mechanisms that are either based on implicit trust of the repository, or public/private key signatures on individual packages.

Stork is a package management system designed for distributed VM environments. Stork has several advantages over existing package management systems: it provides *secure* and *efficient* inter–VM package sharing on the same physical machine; it provides centralized package management that allows users to determine which packages should be installed on which VMs without configuring each VM individually; it allows multiple physical machines to download the same package efficiently; it ensures that package updates are propagated to the VMs in a timely fashion; and it provides a flexible security mechanism that allows users to specify which packages they trust as well as delegate that decision on a per-package basis to other (trusted) users.

Stork's inter–VM sharing facility is important for reducing resource consumption caused by package management in VM environments. VMs are excellent for isolation, but this very isolation can increase the disk, memory, and network bandwidth requirements of package management. It is very inefficient to have each VM install its own copy of each package's files. The same is true of memory: if each VM has its own copy of a package's files then it will have its own copy of the executable files in memory. Memory is often more of a limiting factor than disk, so Stork's ability to share package files between VMs is particularly important for increasing the number of VMs a single physical machine can support. In addition, Stork reduces network traffic by only downloading a package to a physical machine once, even if

multiple VMs on the physical machine install it.

Stork's inter–machine package management facility enables centralized package management and efficient, reliable, and timely package downloads. Stork provides package management utilities and configuration files that allow the user to specify which packages are to be installed on which VMs. Machines download packages using efficient transfer mechanisms such as BitTorrent [6] and CoBlitz [16], making downloads efficient and reducing the load on the repository. Stork uses fail–over mechanisms to improve the reliability of downloads, even if the underlying content distribution systems fail. Stork also makes use of publish/subscribe technology to ensure that VMs are notified of package updates in a timely fashion.

Stork provides all of these performance benefits without compromising security; in fact, Stork has additional security benefits over existing package management systems. First, Stork shares files securely between VMs. Although a VM can delete its link to a file, it cannot modify the file itself. Second, a user can securely specify which packages he or she trusts and may delegate this decision for a subset of packages to another user. Users may also trust other users to know which packages *not* to install, such as those with security holes. Each VM makes package installation decisions based on a user's trust assumptions and will not install packages that are not trusted. While this paper touches on the security aspects of the system that are necessary to understand the design, a more rigorous and detailed analysis of security is deferred to future work.

In addition, Stork is flexible and modular, allowing the same Stork code base to run on a desktop PC, a Vserver–based virtual environment, and a PlanetLab node. This is achieved via pluggable modules that isolate the platform–specific functionality. Stork accesses these modules through a well defined API. This approach makes it easy to port Stork to different environments and allows the flexibility of different implementations for common operations such as file retrieval.

Stork has managed many thousands of VMs and has been deployed on PlanetLab [17, 18] for over 4 years. Stork is currently running on over hundreds of PlanetLab nodes and its package repository receives a request roughly every ten seconds. Packages installed in multiple VMs by Stork typically use over an order of magnitude less space and 50% the memory of packages installed by other tools. Stork also reduces the repository load by over an order of magnitude compared to HTTP–based tools. Stork is also used in the Vserver[12] environment and can also be used in non-VM environments (such as on a home system) as an efficient and secure package installation system. The source code for Stork is available at `http://www.cs.arizona.edu/stork`

2 Stork

Stork provides both manual management of packages on individual VMs using command-line tools as well as centralized management of groups of VMs. This section describes an example involving package management, the configuration files needed to manage VMs with Stork, and the primary components of Stork.

2.1 An Example

Consider a system administrator that manages thousands of machines at several sites around the globe. The company's servers run VM software that allow different production groups more flexible use of the hardware resources. In addition, the company's employees have desktop machines that have different software installed depending on their use.

The system administrator has just finished testing a new security release for a fictional package `foobar` and she decides to have all of the desktop machines used for development update to the latest version along with any testing VMs that are used by the coding group. The administrator modifies a few files on her local machine, signs them using her private key, and uploads them to a repository. Within minutes all of the desired machines that are online have the updated `foobar` package installed. As offline machines come online or new VMs are created they automatically update their copies of `foobar` as instructed.

The subsequent sections describe the mechanisms Stork uses to provide this functionality to its users. Section 5 revisits this example and explains in detail how Stork provides the functionality described in this scenario.

2.2 File Types

Stork uses several types of files that contain different information and are protected in different ways (Table 2.2). The user creates a public/private key pair that authenticates the user to the VMs he or she controls. The *public key* is distributed to all of the VMs and the *private key* is used to sign the configuration files. In our previous example, the administrator's public key is distributed to all of the VMs under her control. When files signed by her private key were added to the repository, the authenticity of these files was independently verified by each VM using the public key.

The *master configuration file* is similar to those found in other package management tools and indicates things such as the transfer method, repository name, user name, etc. It also indicates the location of the public key that should be used to verify signatures.

The user's trusted packages file (*TP file*) indicates which packages the user considers valid. The TP file does not cause those packages to be installed, but instead

File Type	Repository	Client	Central Mgmt	Signed and Embedded
User Private Key	No	No	Yes	No
User Public Key	No†	Yes	Yes	No
Master Configuration File	No†	Yes	Yes	No
Trusted Packages (TP)	Yes	Yes	Yes	Yes
Pacman Packages	Yes	No	Yes	Yes
Pacman Groups	Yes	No	Yes	Yes
Packages (RPM, tar.gz)	Yes	Yes	Yes	Secure Hash
Package Metadata	Yes	Yes	Yes	No
Repository Metahash	Yes	Yes	No	Signed Only

Table 1: **Stork File Types:** This table shows the different types of files used by Stork. The repository column indicates whether or not the file is obtained from the repository by the clients. The client column indicates whether or not the file is used for installing packages or determining which packages should be installed locally based upon the files provided by the centralized management system. The centralized management column indicates if the files are created by the management tools. The signed/embed column indicates which files are signed and have a public key embedded in their name.

indicates trust that the packages have valid contents and are candidates for installation. For example, while the administrator was testing the latest release of `foobar` she could add it to her trusted packages file because she believes the file is valid.

There are two pacman files used for centralized management. The *groups.pacman* file allows VMs to be categorized into convenient groups. For example, the administrator could configure her pacman groups file to create separate groups for VMs that perform different tasks. VMs can belong in multiple groups such as ALPHA and ACCOUNTING for an alpha test version of accounting software. Any package management instructions for either the ALPHA group or the ACCOUNTING group would be followed by this VM.

The *packages.pacman* file specifies what actions should be done on a VM or a group of VMs. Packages can be installed, updated, or removed. Installation is different from updating in that installation will do nothing if there is a package that meets the criteria already installed while update ensures that the preferred version of the package is installed. For example, when asked to install `foobar`, if any version of the package is currently installed then no operation will occur. If asked to update `foobar`, Stork checks to see if the administrator's TP file specifies a different version of `foobar` and if so, replaces the current version with the new version.

The *packages* (for example, the `foobar` RPM itself) contain the software that is of interest to the user. The *package metadata* is extracted from packages and is published by the repository to describe the packages that are available. The *repository metahash* is a special file that is provided by the repository to indicate the current repository state.

2.3 Architecture

Stork consists of four main components:

- a *repository* that stores configuration files, packages, and associated metadata;
- a set of *client tools* that are used in each Stork client VM to manage its packages by interacting either directly with the repository or when available through the nest;
- a *nest* process that runs on physical machines and coordinates sharing between VMs as well as providing repository metadata updates to its client VMs and downloading packages;
- and *centralized management tools* that allows a user to control many VMs concurrently, create and sign packages, upload packages to the repository, etc.

The client tools consist of the *stork command-line tool* (referred to simply as `stork`), which allows users to install packages manually, and *pacman* which supports centralized administration and automated package installation and upgrade. While a client VM may communicate with the repository directly, it is far more efficient for client VMs to interact with their local nest process, who interacts with the repository on their behalf.

2.3.1 Repository

The Stork repository's main task is to serve files much like a normal web server. However, the repository is optimized to efficiently provide packages to Stork client VMs. First, the repository provides secure user upload of packages, trusted packages files, and pacman packages and groups files. Second, the repository pushes notifications of new content to interested VMs. Third, the repository makes packages available via different efficient transfer mechanisms such as BitTorrent.

Handling Uploaded Data The Stork repository allows multiple users to upload files while retaining security. TP, `groups.pacman`, and `packages.pacman` files must be signed by the user that uploads them. Every signed file has a timestamp for the signature embedded in the portion of the file protected by the signature. The public key of the user is embedded in the file name of the signed file (similar to self-certifying path names [13]). This avoids naming conflicts and allows the repository verify the signature of an uploaded file. The repository will only store a signed file with a valid signature that is newer than any existing signed file of the same name. This prevents replay attacks and allows tools to request the files that match a public key directly.

†In order to automatically deploy Stork on PlanetLab this restriction is relaxed. See Section 3 for more details.

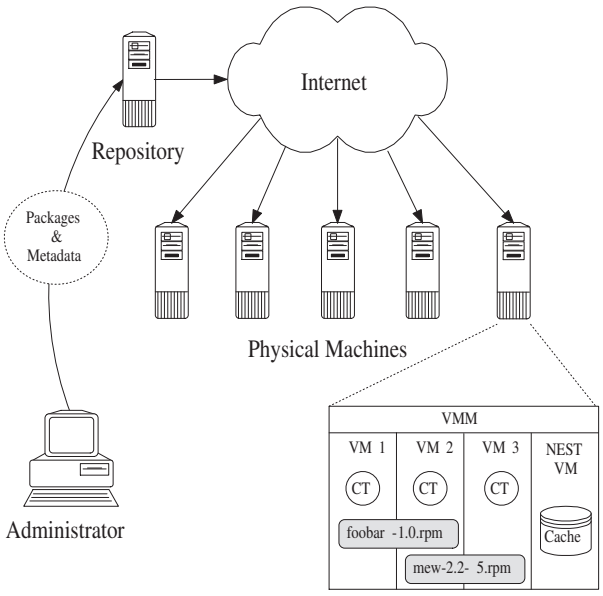

Figure 1: **Stork Overview.** Stork allows centralized administration and sharing of packages. The administrator publishes packages and metadata on the repository. Updates are propagated to VMs running on distributed physical machines. Each physical machine contains a single nest VM, and one or more client VMs that run the Stork client tools.

Packages and package metadata are treated differently than configuration files. These files are not signed, but instead incorporate a secure hash of their contents in their names. This prevents name collisions and allows users to request packages directly by secure hash. In all cases, the integrity of a file is verified by the recipient before it is used (either by checking the signature or the secure hash, as appropriate). The repository only performs these checks itself to prevent pollution of the repository and unnecessary downloads, rather than to ensure security on the clients.

Pushing Notifications The repository notifies interested clients when the repository contents have changed. The repository provides this functionality by pushing an updated repository metahash whenever data has been added to the repository. However, this does not address the important question of *what* data has been updated. This is especially difficult to address when VMs may miss messages or suffer other failures. One solution to this problem is for the repository to push out hashes of all files on the repository. As there are many thousands of metadata files on the repository, it is too costly to publish the individual hashes of all of them and have the client VMs download each metadata file separately. Instead the repository groups metadata files together in a tarball organized by type. For example, one tarball contains all of the trusted packages files, another with all of the pacman files, etc. The hashes of these tarballs are put into the repository metahash which is pushed to each interested client VM. No matter how many updates the client VM misses, it can examine the hash of the local tarballs and the hashes provided by the repository and determine what needs to be retrieved.

Efficient Transfers The repository makes all of its files available for download through HTTP. However, having each client download its files via separate HTTP connections is prohibitively expensive. The repository therefore supports different transfer mechanisms for better scalability, efficiency, and performance. Some transfer mechanisms are simple (like CoBlitz and Coral which require no special handling by the repository) and others (like BitTorrent) require special handling.

To support BitTorrent[6] downloads the repository runs a BitTorrent tracker and a modified version of the `btlaunchmany` daemon provided by BitTorrent. The `btlaunchmany` daemon monitors a directory for any new or updated files. When a new file is uploaded to the repository it is placed in the monitored directory. When the daemon notices the new file it creates a torrent file that is later seeded. Unique naming is achieved by appending the computed hash of the shared file to the name of the torrent. The torrent file is placed in a public location on the repository for subsequent download by the clients through HTTP.

2.3.2 Client Tools

The client tools are used to manage packages in a client VM and include the `stork`, `pacman`, and `stork_receive_update` commands. The `stork` tool uses command-line arguments to install, update, and remove packages. Its syntax is similar to `apt` [2] or `yum` [25]. The `stork` tool resolves dependencies and installs additional packages as necessary. It also upgrades and removes packages. The `stork` tool downloads the latest metadata from package repositories, verifies that packages are trusted by the user's TP file, and only installs trusted files.

Package management with the `stork` tool is a complex process involving multiple steps including *dependency resolution*, *trust verification*, *download*, and *installation*. For example, consider the installation of the `foobar` package. Assume `foobar` depends on a few other packages, such as `emacs` and `glibc`, before `foobar` itself can be installed. In order to perform the installation of `foobar`, the `stork` tool must determine whether `foobar`, `emacs`, and `glibc` are already installed on the client and if not, locate candidate versions that satisfy the dependencies. These steps are similar to those performed by other package managers[2, 25, 19]. Finally Stork ensures that those candidates satisfy the

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes" ?>
<TRUSTEDPACKAGES>

<!-- Trust some packages that the user specifically allows -->
<FILE PATTERN="emacs-2.2-5.i386.rpm"   HASH="aed4959915ad09a2b02f384d140c4
626b0eba732" ACTION="ALLOW"/>
<FILE PATTERN="foobar-1.01.i386.rpm" HASH="16b6d22332963d54e0a034c11376a
2066005c470" ACTION="ALLOW"/>
<FILE PATTERN="foobar-1.0.i386.rpm"  HASH="3945fd48567738a28374c3b238473
09634ee37fd" ACTION="ALLOW"/>
<FILE PATTERN="simple-1.0.tar.gz"    HASH="23434850ba2934c39485d293403e3
293510fd341" ACTION="ALLOW"/>

<!-- Allow access to the planetlab Fedora Core 4 packages -->
<USER PATTERN="*" USERNAME="planetlab-v4" PUBLICKEY="MFwwDQYJKoZIhvcNAQEB
BQADSwAwSAJBALtGteQPdLa0kYv+klFWTklH9Y7frYhl5JVlhgJa5PlGI3yK+R22UsD65_J4P
V92RUgVd_uJMuB8Q4bi1w4o6JMCAwEAAQ" ACTION="ALLOW"/>

<!-- Allowing the 'stork' user lets stork packages be installed -->
<USER PATTERN="stork*" USERNAME="stork" PUBLICKEY="MFwwDQYJKoZIhvcNAQEBBQADSwAw
SAJBAKgZCjfKDl9ISoclfBuZsQze6bXtu+QYF64TLQlI9fgEg2CDyGQVOsZ2CaX1ZEZ_O69AYZ
p8nj+YJLIJM3+W3DMCAwEAAQ" ACTION="ALLOW"/>

</TRUSTEDPACKAGES>
```

Figure 2: **Example TP File.** This file specifies what packages and users are trusted. Only packages allowed by a TP file may be installed. FILE actions are used to trust individual packages. USER actions allow hierarchical trust by specifying a user whose TP file is included. The signature, timestamp, and duration are not shown and are contained in an XML layer that encapsulates this file.

trust requirements that the user has specified.

Figure 2 shows a TP file example. This file specifically allows `emacs-2.2-5.i386.rpm`, several versions of `foobar`, and `customapp-1.0.tar.gz` to be installed. Each package listed in the TP file includes the hash of the package, and only packages that match the hashes may be installed. It trusts the `planetlab-v4` user to know the validity of any package it says (this user has a list of hashes of all of the Fedora Core 4 packages). It also trusts the `stork` user to know the validity of any packages that start with "stork".

Once satisfactory trusted candidates have been found, Stork downloads the packages from the repository and verifies that the packages it downloaded match the entries in the TP file, including the secure hashes. Finally, the packages themselves are installed.

Package removal is much less complex than installation. The `stork` command removes an installed package by deleting the package's files and runs the uninstall scripts for the package.

The `pacman` ("package manager") tool is the entity in a VM that locally enacts centralized administration decisions. The `pacman` tool invokes the appropriate `stork` commands based on two configuration files: `groups.pacman` (Figure 3) and `packages.pacman` (Figure 4). The `groups.pacman` file is optional and defines VM groups. VM groups are used by an administrator to collectively manage a set of VMs. The `groups.pacman` syntax supports basic set operations such as union, intersection, compliment, and difference. For example,

```
<GROUPS>
<GROUP NAME="ALPHA">
<INCLUDE NAME="planetlab1.arizona.net"/>
<INCLUDE NAME="planetlab2.arizona.net"/>
</GROUP>

<GROUP NAME="ACCOUNTING">
<INCLUDE NAME="ALPHA"/>
<INCLUDE NAME="pl1.unm.edu"/>
</GROUP>
</GROUPS>
```

Figure 3: **Example groups.pacman.** The "ALPHA" group consists of two machines in Arizona. The "ACCOUNTING" group also includes a machine at the University of New Mexico.

```
<PACKAGES>
    <CONFIG SLICE="stork" GROUP="ACCOUNTING">
        <INSTALL PACKAGE="foobar" VERSION="2.2"/>
        <REMOVE PACKAGE="vi"/>
    </CONFIG>
    <CONFIG>
        <UPDATE PACKAGE = "firefox"/>
    </CONFIG>
</PACKAGES>
```

Figure 4: **Example packages.pacman.** VMs in the slice (a term used to mean a VM on PlanetLab) "stork" and in the group "ACCOUNTING" will have `foobar 2.2` installed and `vi` removed. All VMs in this user's control will have `firefox` installed and kept up-to-date with the newest version.

an administrator for a service may break their VMs into alpha VMs, beta VMs, and production VMs. This allows developers to test a new release on alpha VMs (where there are perhaps only internal users) before moving it to the beta VMs group (with beta testers) and finally the the production servers.

The `packages.pacman` file specifies which packages should be installed, updated, or removed in the current VM based on a combination of VM name, group, and physical machine. This makes it easy, for example, to specify that a particular package should be installed on all VMs on a physical machine, while another package should only be installed on alpha VMs, etc.

Although `pacman` can be run manually, typically it is run automatically via one of several mechanisms. First, `pacman` establishes a connection to the `stork_receive_update` daemon. This daemon receives the repository metahashes that are pushed by the repository whenever there is an update. Upon receiving this notification, `stork_receive_update` alerts `pacman` to the new information. A change to the repository metahash indicates that the repository contents have changed which in turn may change which packages are installed, etc. Second, when `stork_receive_update` is unavailable `pacman` wakes up every 5 minutes and polls the repository for the repository metahash. As before, if there is a discrepancy between the stored data and the described data, pacman

downloads the updated files. Third, `pacman` also runs when its configuration files change.

The `stork_receive_update` daemon runs in each client VM and keeps the repository's metahash up-to-date. Metadata is received from the repositories using both push and pull. Pushing is the preferred method because it reduces server load, and is accomplished using a multicast tree or publish/subscribe system such as PsEPR[4]. Heartbeats are pushed if no new metahash is available. If `stork_receive_update` doesn't receive a regular heartbeat it polls the repository and downloads new repository metahash if necessary. This download is accomplished using an efficient transfer mechanism from one of Stork's *transfer modules* (Section 4.1). This combination of push and pull provides an efficient, scalable, fault tolerant way of keeping repository information up-to-date in the VMs.

2.3.3 Nest

The Stork nest process enables secure file-sharing between VMs, prevents multiple downloads of the same content by different VMs, and maintains up-to-date repository metadata. The nest serves two important functions. First, it operates as a shared cache for its client VMs, allowing metadata and packages to be downloaded once and used by many VMs. Second, it performs package installation on behalf of the VMs, securely sharing read-only package files between multiple VMs that install the package (Section 4.2). The nest functionality is implemented by the `stork_nest` daemon.

The `stork_nest` daemon is responsible for maintaining connections with its client VMs and processing requests that arrive over those connections (typically via a socket, although this is configurable). A client must first authenticate itself to `stork_nest`. The authentication persists for as long as the connection is established. Once authenticated, the daemon then fields requests for file transfer and sharing. File transfer operations use the shared cache feature of the repository to provide cached copies of files to the clients. Sharing operations allow the clients to share the contents of packages using the *prepare interface* (Section 4.4).

Typically, the nest runs on each machine machine that runs Stork; however, there may be cases (such as in a desktop machine or a server that does not use VMs) where the nest is not run. In the case where no nest is running or the nest process fails, the client tools communicate directly with the repository.

2.3.4 Centralized Management

The centralized management tools allow Stork users to manage their VMs without needing to contact the VMs directly. In our example the administrator wanted to install `foobar` automatically on applicable systems under her control rather than logging into them individually. Unlike the client tools that are run in Stork client VMs, the centralized management tools are typically run on the user's desktop machine. They are used to create TP files, pacman packages and groups files, the master configuration file, public/private keypairs, etc. These files are used by the client tools to decide what actions to perform on the VM. In addition to managing these files, the centralized management tools also upload metadata and/or packages to the repository, and assist the user in building packages.

The main tool used for centralized management is `storkutil`, a command-line tool that has many different functions including creating public/private key pairs, signing files, extracting metadata from packages, and editing trusted packages, pacman packages and groups files. Administrators use this tool to create and modify the files that control the systems under their control. While files can be edited by other tools and then resigned, `storkutil` has the advantage of automatically resigning updated files. After updating these files they are then uploaded to the repository.

3 Stork on PlanetLab

Stork currently supports the Vserver environment, non-VM machines, and PlanetLab [17, 18]. The PlanetLab environment is significantly different from the other two, so several extensions to Stork have been provided to better support it.

3.1 PlanetLab Overview

PlanetLab consists of over 750 nodes spread around the world that are used for distributed system and network research. Each PlanetLab node runs a custom kernel that superficially resembles the Vserver [12] version of Linux. However there are many isolation, performance, and functionality differences.

The common management unit in PlanetLab is the *slice*, which is a collection of VMs on different nodes that allow the same user(s) to control them. A node typically contains many different VMs from many different slices, and slices typically span many different nodes. The common PlanetLab (mis)usage of the word "slice" means both the collection of similarly managed VMs and an individual VM.

Typical usage patterns on PlanetLab consist of an authorized user creating a new slice and then adding it to one or more nodes. Many slices are used for relatively short periods of time (a week or two) and then removed from nodes (which tears down the VMs on those nodes). It is not uncommon for a group that wants to run an experiment to create and delete a slice that spans hundreds of nodes in the same day. There are relatively loose restrictions as to the number of nodes slices may use and

the types of slices that a node may run so it is not uncommon for slices to span all PlanetLab nodes.

3.2 Bootstrapping Slices on PlanetLab

New slices on PlanetLab do not have the Stork client tools installed. Since slices are often short–lived and span many nodes, requiring the user to log in and install the Stork client tools on every node in a slice is impractical. Stork makes use of a special `initscript` to automatically install the Stork client tools in a slice. The `initscript` is run whenever the VMM software instantiates a VM for the slice on a node. The Stork `initscript` communicates with the nest on the node and asks the nest to share the Stork client tools with it. If the nest process is not working, the `initscript` instead retrieves the relevant RPMs securely from the Stork repository.

3.3 Centralized Management

Once the Stork client tools are running they need the master configuration file and public key for the slice. Unfortunately the ssh keys that are used by PlanetLab to control slice access are not visible within the slice, so Stork needs to obtain the keys through a different mechanism. Even if the PlanetLab keys were available it is difficult to know which key to use because many users may be able to access the same VM. Even worse, often a different user may want to take control of a slice that was previously managed by another user. Stork's solution is to store the public key and master configuration file on the Stork repository. The repository uses PlanetLab Central's API to validate that users have access to the slices they claim and stores the files in a area accessible by https. The client tools come with the certificate for the Stork repository which `pacman` and `stork` use to securely download the public key and master configuration file for the slice. This allows users to change the master configuration file or public key on all nodes by simply adding the appropriate file to the Stork repository.

4 Modularity

Stork is highly modular and uses several interfaces that allow its functionality to be extended to accommodate new protocols and package types:

Transfer A transfer module implements a transport protocol. It is responsible for retrieving a particular object given the identifier for that object. Transfer protocols currently supported by Stork include CoBlitz [15], BitTorrent [6], Coral [9], HTTP, and FTP.

Share A share module is used by the Stork nest to share files between VMs. It protects files from modification, maps content between slices, and authenticates client slices. Currently Stork supports PlanetLab and Linux VServers. Using an extensible interface allows Stork to be customized to support new VM environments.

Package A package module provides routines that the Stork client tools use to install, remove, and interact with packages. It understands several package formats (RPM, tar) and how to install them in the current system.

Prepare A prepare module prepares packages for sharing. Preparing a package typically involves extracting the files from the package. The Prepare interface differs from the Package interface in that package install scripts are not run and databases (such as the RPM database) are not updated. The nest process uses the prepare module to ready the package files for sharing.

4.1 Transfer Modules

Transfer modules are used to download files from the Stork repository. Transfer modules encapsulate the necessary functionality of a particular transfer protocol without having to involve the remainder of Stork with the details.

Each transfer module implements a `retrieve_files` function that takes several parameters including the name of the repository, source directory on the repository, a list of files, and a target directory to place the files in. The transfer module is responsible for opening and managing any connections that it requires to the repositories. A successful call to `retrieve_files` returns a list of the files that were successfully retrieved.

Transfer modules are specified to Stork via an ordered list in the main Stork configuration file. Stork always starts by trying the first transfer module in the list. If this transfer module should fail or return a file that is old, then Stork moves on to the next module in the list.

4.1.1 Content Retrieval Modules

CoBlitz uses a content distribution network (CDN) called CoDeeN[22] to support large files transfers without modifying the client or server. Each node in the CDN runs a service that is responsible for splitting large files into chunks and reassembling them. This approach not only reduces infrastructure and the need for resource provisioning between services, but can also improve reliability by leveraging the stability of the existing CDN. CoBlitz demonstrates that this approach can be implemented at low cost, and provides efficient transfers even under heavy load.

Similarly, the Coral module uses a peer-to-peer content distribution network that consists of volunteer sites that run CoralCDN. The CoralCDN sites automatically replicate content as a side effect of users accessing it. A file is retrieved via CoralCDN simply by making a small change to the hostname in an object's URL. Then a peer-to-peer DNS layer transparently redirects browsers

to nearby participating cache nodes, which in turn cooperate to minimize load on the origin web server. One of the system's key goals is to avoid creating hot spots. It achieves this through Coral [9], a latency-optimized hierarchical indexing infrastructure based on a novel abstraction called a *distributed sloppy hash table* (DSHT).

BitTorrent is a protocol for distributing files. It identifies content by URL and is designed to integrate seamlessly with the web. Its advantage over HTTP is that nodes that download the same file simultaneously also upload portions of the file to each other. This greatly reduces the load on the server and increases scalability. Nodes that uploads portions of a file are called *seeds*. BitTorrent employs a *tracker* process to track which portions each seed has and helps clients locate seeds with the portions they need. BitTorrent balances seed loads by having its clients preferentially retrieve unpopular portions, thus creating new seeds for those portions.

Stork also supports traditional protocols such as HTTP and FTP. These protocols contact the repository directly to retrieve the desired data object. It is preferable to use one of the content distribution networks instead of HTTP or FTP as it reduces the repository load.

Stork supports all of these transfer mechanisms and performance results are presented in Section 6. One key observation is that although these transfer methods are efficient, the uncertainties of the Internet make failure a common case. For this reason the transfer module tries a different transfer mechanism when one fails. For example, if a BitTorrent transfer fails, Stork will attempt CoBlitz, HTTP, or another mechanism until the transfer succeeds or gives up. This provides efficiency in the common case, and correct handling when there is an error.

4.1.2 Nest Transfer

In addition to the transfer modules listed above, Stork supports a nest transfer module. The nest transfer module provides an additional level of indirection so that the client asks the nest to perform the transfer on its behalf rather than performing the transfer directly. If the nest has a current copy of the requested item in its cache, then it can provide the item directly from the cache. Otherwise, the nest will invoke a transfer module (such as BitTorrent, HTTP, etc) to retrieve the item, which it will then provide to the client and cache for later use.

4.1.3 Push

Stork supports metadata distribution to the nests using a publish/subscribe system [8]. In a publish/subscribe system, subscribers register their interest in an event and are subsequently notified of events generated by publishers. One such publish/subscribe system is PsEPR [4]. The messaging infrastructure for PsEPR is built on a collection of off-the-shelf instant messaging servers running on PlanetLab. PsEPR publishes events (XML fragments) on channels to which clients subscribe. Behind the scenes PsEPR uses overlay routing to route events among subscribers.

The Stork repository pushes out metadata updates through PsEPR. It also pushes out the repository's `metahash` file that contains the hashes of the metadata files; this serves as a heartbeat that allows nodes to detect missed updates. In this manner nodes only receive metadata changes as necessary and there is no burden on the repository from unnecessary polling.

4.1.4 Directory Synchronization

In addition to pushing data, Stork also supports a mechanism for pulling the current state from a repository. There are several reasons why this might be necessary, with the most obvious being that the publish/subscribe system is unavailable or has not published data in a timely enough manner. Stork builds upon the transfer modules to create an interface that supports the synchronization of entire directories.

Directory synchronization mirrors a directory hierarchy from the repository to the client. It first downloads the repository's `metahash` file (the same file that the repository publishes periodically using PsEPR). This file contains a list of all files that comprise the repository's current state and the hashes for those files. Stork compares the hashes to the those of the most recent copies of these files that it has on disk. If a hash does not match, then the file must be re-downloaded using a transfer module.

4.2 Share Modules

Virtual machines are a double–edged sword: the isolation they provide comes at the expense of sharing between them. Sharing is used in conventional systems to provide performance and resource utilization improvements. One example is sharing common application programs and libraries. They are typically installed in a common directory and shared by all users. Only a single copy of each application and library exists on disk and in memory, greatly reducing the demand on these resources. Supporting different versions of the same software is an issue, however. Typically multiple versions cannot be installed in the same common directory without conflicts. Users may have to resort to installing their own private copies, increasing the amount of disk and memory used.

Stork enables sharing in a VM environment by weakening the isolation between VMs to allow file sharing under the control of the nest. Specifically, read-only files can be shared such that individual slices cannot modify the files, although they can be unlinked. This re-

duces disk and memory consumption. These benefits are gained by any slices that install the same version of a package. It also allows slices to install different package versions in the standard location in their file systems without conflict.

In Stork, sharing is provided through Share modules that hide the details of sharing on different VM platforms. This interface is used by the nest and provides five routines: `init_client`, `authenticate_client`, `share`, `protect`, and `copy`. `Init_client` is called when a client binds to the nest, and initializes the per-client state. `Authenticate_client` is used by the nest to authenticate the client that has sent a bind request. This is done by mapping a randomly named file into the client's filesystem and asking it to modify the file in a particular way. Only a legitimate client can modify its local file system, and therefore if the client succeeds in modifying the file the nest requested, the nest knows that it is talking to a legitimate client. The `share` routine shares (or unshares) a file or directory between the client and nest, `protect` protects (or unprotects) a file from modification by the client, and `copy` copies a file between the nest and a client.

The implementation of the Share module depends on the underlying platform. On PlanetLab the Share module communicates with a component of the VMM called Proper [14] to perform its operations. The nest runs in an unprivileged slice – all privileged operations, such as sharing, copying, and protecting files, are done via Proper.

On the Vserver platform the nest is run in the root context, which gives it full access to all VM file systems and allows it to do all of its operations directly. Hard links are used to share files between VMs. The immutable bits are used to protect shared files from modification. Directories are shared using `mount --bind`. Copying is easily done because the root context has access to all VM filesystems.

4.3 Package Modules

Stork supports the popular package formats RPM and tar. In the future, other package formats such as Debian may be added. Each type of package is encapsulated in a package module. Each package module implements the following interfaces:

`is_package_understood`. Returns true if this package module understands the specified package type. Stork uses this function to query each package module until a suitable match is found.

`get_package_provides`. Returns a list of dependencies that are provided by a package. This function is used to generate the metadata that is then used to resolve dependencies when installing packages.

`get_packages_requires`. Returns a list of packages that this package requires. This function is used along with `get_package_provides` to generate the package metadata.

`get_package_files`. Returns a list of the files that are contained in a package. This function is also used when generating package metadata.

`get_package_info`. Returns the name, version, release, and size of a package. This information allows the user to install a specific version of a package.

`get_installed_versions`. Given the name of a package, return a list of the versions of the package that are installed. This function is used to determine when a package is already installed, so that an installation can be aborted, or an upgrade can be performed if the user has requested upgrades.

`execute_transactions`. Stork uses a transaction-based interface to perform package installation, upgrade, and removal. A transaction list is an ordered list of package actions. Each action consists of a type (`install`, `upgrade`, `remove`) and a package name.

4.3.1 Supported Package Types

`stork_rpm`. Stork currently supports RPM and tar packages. The RPM database is maintained internally by the `rpm` command-line tool, and Stork's RPM package module uses this tool to query the database and to execute the `install`, `update`, and `remove` operations,

`stork_tar`. Tar packages are treated differently because Linux does not maintain a database of installed tar packages, nor is there a provision in tar packages for executing install and uninstall scripts. Stork allows users to bundle four scripts, `.preinstall`, `.postinstall`, `.preremove`, `.postremove` that will be executed by Stork at the appropriate times during package installation and removal. Stork does not currently support dependency resolution for tar packages, but this would be a straightforward addition. Stork maintains a database that contains the names and versions of tar packages that are installed that mimics the RPM database provided by the `rpm` tool.

4.3.2 Nest Package Installation

A special package manager, `stork_nest_rpm`, is responsible for performing shared installation of RPM packages. Shared installation of tar packages is not supported at this time. Performing a share operation is a three-phase process.

In the first phase, `stork_nest_rpm` calls `stork_rpm` to perform a private installation of the package. This allows the package to be installed atomically using the protections provided by RPM,

including executing any install scripts. In the second phase, stork_nest_rpm contacts the Stork nest and asks it to prepare the package for sharing. The prepare module is discussed in the following section. Finally, in the third phase stork_nest_rpm contacts the nest and instructs it to share the prepared package. The nest uses the applicable share module to perform the sharing. The private versions of files that were installed by stork_rpm are replaced by shared versions. Stork does not attempt to share configuration files because these files are often changed by the client installation. Stork also examines files to make sure they are identical prior to replacing a private copy with a shared copy.

Removal of packages that were installed using stork_nest_rpm requires no special processing. stork_nest_rpm merely submits the appropriate remove actions to stork_rpm. The stork_rpm module uses the rpm tool to uninstall the package, which unlinks the package's files. The link count of the shared files is decremented, but is still nonzero. The shared files persist on the nest and in any other clients that are linked to them.

4.4 Prepare Modules

Prepare modules are used by the nest to prepare a package for sharing. In order to share a package, the nest must extract the files in the package. This extraction differs from package installation in that no installation scripts are run, no databases are updated, and the files are not moved to their proper locations. Instead, files are extracted to a sharing directory.

Prepare modules only implement one interface, the prepare function. This function takes the name of a package and the destination directory where the extracted files should be placed.

RPM is the only package format that Stork currently shares. The first step of the stork_rpm_prepare module is to see if the package has already been prepared. If it has, then nothing needs to be done. If the package has not been prepared, then stork_rpm_prepare uses rpm2cpio to convert the RPM package into a cpio archive which is then extracted. stork_rpm_prepare queries the rpm tool to determine which files are configuration files and moves the configuration files to a special location so they will not be shared. Finally, stork_rpm_prepare sets the appropriate permissions on the files that it has extracted.

5 Stork Walkthrough

This section illustrates how the Stork components work together to manage packages using the example from Section 2.1 in which an administrator installs an updated version of the foobar package on the VMs the company uses for testing and on the non-VM desktop machines used by the company's developers.

1. The administrator uses storkutil to add the new version of the foobar package to her TP file (if she hasn't done so already).
2. She uses storkutil to add the groups Devel and Test to her groups.pacman file, representing the developer's end systems and the testing VMs, respectively. Since groups can be reused, this step most likely would have been done already.
3. The administrator uses storkutil to add a line to her packages.pacman file instructing the Test group to update foobar. She does the same for the Devel group.
4. Storkutil automatically signed these files with her private key. She now uploads these files to a Stork repository. If the new version of the foobar package is not already on the repository she uploads this as well.
5. The repository treats the TP and pacman files similarly. The signatures are verified using the administrator's public key that is embedded in the file name. The new files replace the old if their signatures are valid and their timestamps newer. The foobar package is stored in a directory whose name is its secure hash. The package metadata is extracted and made available for download.
6. The repository uses the publish/subscribe system PsEPR to push out a new repository metahash to the VMs.
7. The VMs are running stork_receive_update and obtain the new repository metahash. The stork_receive_update daemon wakes up the pacman daemon.
8. The pacman daemon updates its metadata. On non-VM platforms, the files are downloaded efficiently using whatever transfer method is listed in the Stork configuration file. On VM platforms, pacman retrieves the files through the nest (which means the files are downloaded only once per physical machine).
9. Pacman processes its metadata and if the current VM is in either the Test or Devel groups it calls stork to update the foobar package.
10. The stork tool verifies that it has the current metadata and configuration files. This is useful because it is not uncommon for several files to be uploaded in short succession. If this is not the case it retrieves the updated files in the same manner as pacman.

11. Stork verifies that the specified version of foobar is not already installed; if it is, Stork simply exits.

12. Stork searches the package metadata for the specified package. If no candidate is found then it exits with an error message that the package cannot be found. Multiple candidates may be returned if the metadata database contains several versions of foobar.

13. Stork verifies that the user trusts the candidate versions of foobar. It does this by applying the rules from the user's TP file one at a time until a rule is found that matches each candidate. If the rule is a DENY rule, then the candidate is rejected. If the rule is an ACCEPT rule, then the candidate is deemed trustworthy. The result of trust verification is an ordered list of package candidates.

14. Stork now has one or more possible candidates for foobar. However, if foobar depends on other packages stork repeats steps 11 - 14 for the dependencies to determine if those dependencies can be satisfied.

15. Stork now has a list of packages that are to be updated, including foobar and its missing dependencies. Stork uses a transfer module to retrieve foobar and dependant packages. The highest priority transfer method is to contact the repository, which is via the nest in VM environments.

16. In a VM environment the nest receives the requests for foobar and its dependencies from the client VM. If these files are already cached on the nest, then the nest provides those local copies. If not, then the nest invokes the transfer modules (BitTorrent, CoBlitz, etc) to retrieve the files. When retrieval is complete, the nest shares the package with the client VM.

17. Stork now has local copies of foobar and its dependant packages. The client queries the package modules to find one that can install the package. In non–VM environments the stork_rpm module installs the packages using RPM and returns to stork which exits. In VM environments the stork_nest_rpm module is tried first (stork will fail over and use stork_rpm if this module fails). Because foobar is an RPM package, stork_nest_rpm can process it. Stork builds a transaction list and passes it to the execute_transactions function of stork_nest_rpm

18. In a VM environment the stork_nest_rpm module passes the transaction list to stork_rpm in order to install a private non-shared copy of the foobar package.

19. In a VM environment the stork_nest_rpm module then contacts the nest and issues a request to prepare and share foobar. The nest uses the appropriate prepare module to extract the files contained in foobar. The nest uses the appropriate share module to share the extracted files with the client VM. Sharing overwrites the private versions of the files in the client's VM with shared versions from the foobar package.

In some cases there will be systems that do not receive the PsEPR update. This could occur because PsEPR has failed to deliver the message or perhaps because the system is down. If PsEPR failed then pacman will start every 5 minutes to check for updates. If the system was down then when it restarts pacman will run. Either way pacman will start and obtain a new repository metahash and the system will continue the process from Step 8.

If nest or module failures happen, stork fails over to other modules that might be able to service the request. For example, if the packages cannot be downloaded by BitTorrent, the tool will instead try another transfer method like CoBlitz as specified in the master configuration file.

6 Results

Stork was evaluated via several experiments on PlanetLab. The first experiment measures the effectiveness of Stork in conserving disk space when installing packages in VM environments. The second experiment looks at the memory savings Stork provides to packages installed in multiple VMs. The final set of experiments look at the impact Stork has on package downloads both in performance and in repository load.

6.1 Disk Usage

The first experiment measured the amount of disk space saved by installing packages using Stork versus installing them in client slices individually (Figure 5). These measurements were collected using the 10 most popular packages on a sample of 11 PlanetLab nodes. Some applications consist of two packages: one containing the application and one containing a library used exclusively by the application. For the purpose of this experiment they are treated as a single package.

For all but one package, Stork reduced the per-client disk space required to install a package by over 90%. It should be noted that the nest stores an entire copy of the package to which the clients link; Stork's total space savings is therefore a function of the total number of clients

Rank	Package Name	Disk Space (KB) Standard	Stork	Percent Savings
1	scriptroute	8644	600	93%
2	undns	13240	652	95%
3	chord	64972	1216	98%
4	j2re	61876	34280	45%
5	stork	320	32	90%
6	bind	6884	200	97%
7	file	1288	36	97%
8	make	808	32	96%
9	cpp	3220	44	99%
10	binutils	6892	60	99%

Figure 5: **Disk Used by Popular Packages.** This table shows the disk space required to install the 10 most popular packages installed by the slices on a sampling of PlanetLab nodes. The *Standard* column shows how much per-slice space the package consumes if nothing is shared. The *Stork* column shows how much per-slice space the package requires when installed by Stork.

Rank	Package Name	Application Name	Memory (MB) Standard	Stork	Percent Savings
1	scriptroute	srinterpreter	5.8	3.2	45%
2	undns	undns_decode	4.2	2.0	53%
3	chord	adbd	7.6	2.3	70%
3	chord	lsd	7.5	1.1	86%
4	j2re	java	206.8	169.5	18%
5	stork	stork	3.4	1.2	64%
6	bind	named	36.7	32.1	12%
7	file	file	2.6	1.3	50%
8	make	make	2.5	1.1	54%
9	cpp	cpp	2.5	1.2	52%
10	binutils	objdump	3.3	1.4	59%
10	binutils	strip	2.9	1.0	65%
10	binutils	strings	3.4	1.7	50%

Figure 6: **Memory Used by Popular Packages.** Packages installed by Stork allow slices to share process memory. The *Standard* column shows how much memory is consumed by each process when nothing is shared. With Stork the first process will consume the same amount as the *Standard* column, but additional processes only require the amount shown in the *Stork* column.

sharing a package.

One package, `j2re`, had savings of only 45%. This was because many of the files within the package were themselves inside of archives. The post-install scripts extract these files from the archives. Since the post-install scripts are run by the client, the nest cannot share the extracted files between slices. By repackaging the files so that the extracted files are part of the package, this issue can be avoided.

6.2 Memory Usage

Stork also allows processes running in different slices to share memory because they share the underlying executables and libraries (Figure 6). The primary application was run from each package and how much of its memory footprint was shared and how much was not recorded. It was not possible to get memory sharing numbers directly from the Linux kernel running on the PlanetLab nodes. Instead, the `pmap` command was used to dump the processes' address spaces and from this which portions are shared and which are not determined. The results are only approximate, however, because the amount of address space shared does not directly correspond to the amount of memory shared. Portions of the address space may not be backed by a page in memory. More accurate measurements require changes to the Linux kernel that are not currently feasible.

Another difficulty in measuring memory use is that it changes as the program runs. Daemon programs were simply started and measured. Applications that process input files (such as `java` and `make`) were started with a minimal file that goes into an infinite loop. The remaining applications printed their usage information and were measured before they exited.

The resulting measurements show that Stork typically reduces the memory required by additional processes by 50% to 60%. There are two notable exceptions: `named` and `java`. These programs allocate huge data areas that are much larger than their text segments and libraries. Data segments are private, so this shadows any benefits Stork provides in sharing text and libraries.

Transfer Protocol	Effective Client Bandwidth (Kbps) 25%	Median	Mean	75%	Nodes Completed	Server MB Sent
HTTP	413.9	380.6	321.2	338.1	280/286	3080.3
Coral	651.3	468.9	253.6	234.1	259/281	424.7
CoBlitz	1703.5	737.2	381.1	234.0	255/292	77.9
BitTorrent	2011.8	1482.2	1066.9	1044.0	270/284	255.8

Figure 7: **Package Download Performance.** This table shows the results of downloading a 10 MB file to 300 nodes. Each result is the average of three tests. The client bandwidth is measured with respect to the amount of file data received, and the mean, median, 25th percentile, and 75th percentile results given. The *Nodes Completed* column shows the number of nodes that started and finished the transfer. The *Server MB Sent* is the amount of network traffic sent to the clients, including protocol headers and retransmissions.

6.3 Package Retrieval

Stork downloads packages to the nest efficiently, in terms of the amount of network bandwidth required, server load, and elapsed time. This was measured by retrieving a 10MB package simultaneously from 300 nodes (Figure 7), simulating what happens when a new package is stored on the repository. Obviously faulty nodes were not included in the experiments, and a new randomly–generated 10MB file was used for each test. Each test was run three times and the results averaged. It proved impossible to get all 300 nodes to complete the tests successfully; in some cases the nodes never even started the

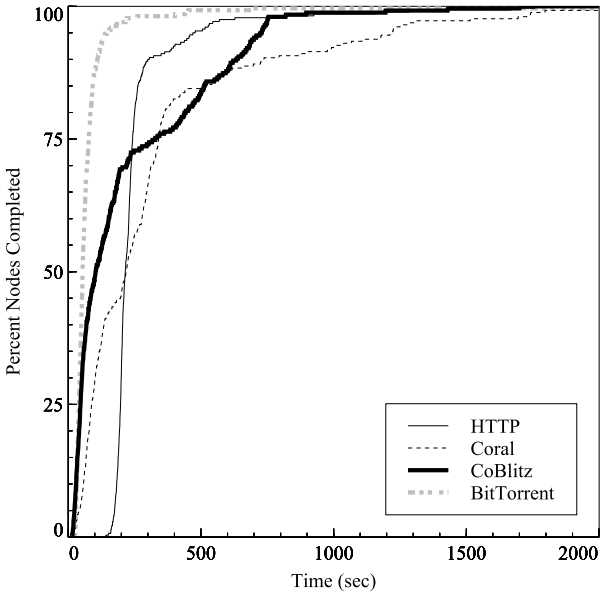

Figure 8: **Elapsed Time.** This graph shows the cumulative distribution of nodes that complete at a given time. Only nodes that successfully completed are included.

test. Faulty and unresponsive nodes are not unusual on PlanetLab. This is dealt with by simply reporting the number of nodes that started and completed each test.

Repository load is important to system scalability, represented as the total amount of network traffic generated by the repository. This includes retransmissions, protocol headers, and any other data. For BitTorrent, this includes the traffic for both the tracker and the initial seed as they were run on the same node; running them on different nodes made negligible difference. At a minimum the repository must send 10MB, since the clients are downloading a 10MB file. CoBlitz generated the least network traffic, sending 7.8 times the minimum. BitTorrent sent 3.3 times as much data as CoBlitz and Coral sent 5.5 times as much as CoBlitz. HTTP was by far the worst, sending 39.5 times more than CoBlitz. In fact, HTTP exceeded the product of the number of clients and the file size because of protocol headers and retransmissions.

For each test the amount of useful bandwidth each client received (file data exclusive of network protocol headers) is reported, including both the median and mean, as well as the 25th and 75th percentiles. BitTorrent's mean bandwidth is 2.8 times that of CoBlitz, 3.3 times that of HTTP, and 4.2 times that of Coral. HTTP does surprisingly well, which is a result of a relatively high-speed connection from the repository to the PlanetLab nodes.

Figure 8 shows the cumulative distribution of client completion times. More than 78% of the nodes completed the transfer within 90 seconds using BitTorrent, compared to only 40% of the CoBlitz and 23% of the Coral nodes. None of the HTTP nodes finished within 90 seconds.

The distribution of client completion times also varied greatly among the protocols. The time of HTTP varied little between the nodes: there is only an 18% difference between the completion time of the 25th and 75th percentiles. The BitTorrent clients in the 25th percentile finished in 48% the time of clients in the 75th percentile, while Coral clients differed by 64%. CoBlitz had the highest variance, so that the clients in the 25th percentile finished in 14% of the time of the clients in the 75% percentile, meaning that the slowest nodes took 7.3 times as long to download the file as the fastest.

These results reflect how the different protocols download the file. All the nodes begin retrieving the file at the same time. Clients in BitTorrent favor downloading rare portions of the file first, which leads to most of the nodes downloading from each other, rather than from the repository. The CoBlitz and Coral CDN nodes download pieces of the file sequentially. This causes the clients to progress lock-step through the file, all waiting for the CDN node with the next piece of the file. This places the current CDN node under a heavy load while the other CDN nodes are idle.

Based on these results Stork uses BitTorrent as its first choice when performing package retrievals, switching to other protocols if it fails. BitTorrent decreased the transfer time by 70% over over HTTP and reduces the amount of data that the repository needs to send by 92%.

7 Related Work

Popular package management systems [2, 7, 19, 24, 25]. typically retrieve packages via HTTP or FTP, resolve dependencies, and manage packages on the local system. They do not manage packages across multiple machines. This leads to inefficiencies in a distributed VM environment because a service spans multiple physical machines, and each physical machine has multiple VMs. The package management system must span nodes and VMs, otherwise VMs will individually download and install packages, consuming excessive network bandwidth and disk space.

Most VMMs focus on providing isolation between VMs, not sharing. VMMs such as Xen [3] and Denali [23] do not provide mechanisms for sharing files or memory between VMs. There are exceptions: Disco [5] implements copy-on-write memory sharing between VMs. This allows the buffer cache to be shared, for example, so that storing software on a shared disk results in a single copy both on disk and in the buffer cache. Similar benefits can be had by sharing software via NFS.

Disco shares the message buffer pages between the client and server VMs, resulting in zero-copy file access. This doesn't solve the problem of supporting conflicting package versions in different VMs, however.

Several VMMs [11, 12] share package files that have already been installed. They use unification programs that search the VM filesystems for common packages and then link read-only package files together. This unification happens after the package has been installed; each VM must download and install the package, only to have its copies of the files subsequently replaced with links. Stork avoids this overhead and complexity by linking the files in the first place.

Stork allows VMs to share the memory used by shared applications and libraries. VMware ESX Server [21] also allows VMs to share memory, but does so based on page content. A background process scans memory looking for multiple copies of the same page. Any redundant copies are eliminated by replacing them with a single copy-on-write page. This allows for more potential sharing than Stork, as any identical pages can be shared, but at the cost of having processes create multiple identical pages only to have them culled.

Stork uses content distribution mechanisms to download packages to nodes. Alternatively, a distributed file system could be used. There are numerous distributed files systems: Shark [1] and SFS-RO [10] are two that have been promoted as a way to distribute software. Clients can either mount applications and libraries directly, or use the file system to access packages that are installed locally. The former has performance, reliability, and conflict issues; the latter only uses the distributed file system to download packages, which may not be superior to using a content distribution mechanism.

Most package management systems have support for security. In general, however, the repository is trusted to contain valid packages. RPM and Debian packages can be signed by the developer and the signature is verified before the package is installed. This requires the user to have the keys of all developers. In many cases package signatures are not checked by default because of this difficulty. The TP file mechanism in Stork allows multiple signatures per package so that users require fewer keys.

The SFS-RO file system [10] uses *self-certifying paths* to alleviate the key problem. The name of a file contains the public key used to sign it. If you know the name, then you know the key. Stork uses a similar technique to sign the `trustedpackages` file and the configuration files for `pacman`. Package names include their hash, instead of the key because a package can have multiple signatures, and so that packages can be refused. For example, Stork has a security user who explicitly rejects packages that are known to have security problems. SFS-RO has no way to trust a user's signature for only a subset of packages.

8 Conclusion

Stork provides both efficient inter–VM package sharing and centralized inter–machine package management. When sharing packages between VMs it typically provides over an order of magnitude in disk savings, and about 50% of the memory costs. Additionally, each node needs only download a package once no matter how many VMs install it. This reduces the package transfer time by 70% and reduces the repository load by 92%.

Stork allows groups of VMs to be centrally administered. The `pacman` tool and its configuration files allow administrators to define groups of VMs and specify which packages are to be installed on which groups. Changes are pushed to the VMs in a timely fashion, and packages are downloaded to the VMs efficiently. Stork has been in use on PlanetLab for over 4 years and has managed thousands of virtual machines. The source code for Stork may be downloaded from `http://www.cs.arizona.edu/stork`

9 Acknowledgments

First and foremost we would like to thank all of the undergraduates who were not coauthors but assisted with the development of Stork including Mario Gonzalez, Thomas Harris, Seth Hollyman, Petr Moravsky, Peter Peterson, Justin Samuel, and Byung Suk Yang. We would also like to thank all of the Stork users. A special thanks goes out to the developers of the services we use including Vivek Pai, KyoungSoo Park, Sean Rhea, Ryan Huebsch, and Robert Adams for their efforts in answering our countless questions. We would especially like to thank Steve Muir at PlanetLab Central for his efforts on our behalf throughout the development of Stork and Proper.

References

[1] ANNAPUREDDY, S., FREEDMAN, M. J., AND MAZIÈRES, D. Shark: Scaling File Servers via Cooperative Caching. In *Proc. 2nd NSDI* (Boston, MA, May 2005).

[2] Debian APT tool ported to RedHat Linux. `http://www.apt-get.org/`.

[3] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the Art of Virtualization. In *Proc. 19th SOSP* (Lake George, NY, Oct 2003).

[4] BRETT, P., KNAUERHASE, R., BOWMAN, M., ADAMS, R., NATARAJ, A., SEDAYAO, J., AND SPINDEL, M. A shared global event propagation system to enable next generation distributed services. In *Proc. of the 1st Workshop on Real, Large Distributed Systems* (San Francisco, CA, Dec 2004).

[5] BUGNION, E., DEVINE, S., GOVIL, K., AND ROSENBLUM, M. Disco: running commodity operating systems

on scalable multiprocessors. *ACM Transactions on Computer Systems 15*, 4 (Nov 1997), 412–447.

[6] COHEN, B. Incentives Build Robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer Systems* (2003).

[7] Debian – dpkg. `http://packages.debian.org/stable/base/dpkg`.

[8] EUGSTER, P. T., FELBER, P., GUERRAOUI, R., AND KERMARREC, A.-M. The Many Faces of Publish/Subscribe. *ACM Computing Surveys 35*, 2 (Jun 2003), 114–131.

[9] FREEDMAN, M. J., FREUDENTHAL, E., AND MAZIÈRES, D. Democratizing content publication with coral. In *Proc. 1st NSDI* (San Francisco, CA, Mar. 2004).

[10] FU, K., KAASHOEK, M. F., AND MAZIÈRES, D. The Click Modular Router. *ACM Transactions on Computer Systems 20*, 1 (Feb 2002), 1–24.

[11] KAMP, P.-H., AND WATSON, R. N. M. Jails: Confining the Omnipotent Root. In *Proc. 2nd Int. SANE Conf.* (Maastricht, The Netherlands, May 2000).

[12] LINUX VSERVERS PROJECT. `http://linux-vserver.org/`.

[13] MAZIÈRES, D., KAMINSKY, M., KAASHOEK, M. F., AND WITCHEL, E. Separating key management from file system security. In *Proc. 17th SOSP* (Kiawah Island Resort, SC, Dec 1999), pp. 124–139.

[14] MUIR, S., PETERSON, L., FIUCZYNSKI, M., CAPPOS, J., AND HARTMAN, J. Proper: Privileged Operations in a Virtualised System Environment. In *Proc. USENIX '05* (Anaheim, CA, Apr 2005).

[15] PARK, K., AND PAI, V. S. Deploying Large File Transfer on an HTTP Content Distribution Network. In *Proc. of the 1st Workshop on Real, Large Distributed Systems* (San Francisco, CA, Dec 2004).

[16] PARK, K., AND PAI, V. S. Scale and Performance in the CoBlitz Large-File Distribution Service. In *Proc. 3rd NSDI* (San Jose, CA, May 2005).

[17] PETERSON, L., ANDERSON, T., CULLER, D., AND ROSCOE, T. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proc. HotNets–I* (Princeton, NJ, Oct 2002).

[18] PlanetLab. `http://www.planet-lab.org`.

[19] RPM Package Manager. `http://www.rpm.org/`.

[20] VMWare Workstation. `http://www.vmware.com/`.

[21] WALDSPURGER, C. A. Memory resource management in VMware ESX server. *Operating Systems Review 36* (2002), 181–194.

[22] WANG, L., PARK, K., PANG, R., PAI, V., AND PETERSON, L. Reliability and Security in the CoDeeN Content Distribution Network. In *Proc. USENIX '02* (San Francisco, CA, Aug 2002).

[23] WHITAKER, A., SHAW, M., AND GRIBBLE, S. D. Scale and Performance in the Denali Isolation Kernel. In *Proc. 5th OSDI* (Boston, MA, December 2002), pp. 195–209.

[24] Windows Update. `http://update.windows.com/`.

[25] Yum: Yellow Dog Updater Modified. `http://linux.duke.edu/projects/yum/`.