The following paper was originally published in the
Proceedings of the Twelfth Systems Administration Conference (LISA '98)
Boston, Massachusetts, December 6-11, 1998

Bootstrapping an Infrastructure

Steve Traugott, Sterling Software and NASA Ames Research Center
Joel Huddleston, Level 3 Communications

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

Bootstrapping an Infrastructure

Steve Traugott – Sterling Software and NASA Ames Research Center
Joel Huddleston – Level 3 Communications

ABSTRACT

When deploying and administering systems infrastructures it is still common to think in terms of individual machines rather than view an entire infrastructure as a combined whole. This standard practice creates many problems, including labor-intensive administration, high cost of ownership, and limited generally available knowledge or code usable for administering large infrastructures.

The model we describe treats an infrastructure as a single large distributed virtual machine. We found that this model allowed us to approach the problems of large infrastructures more effectively. This model was developed during the course of four years of mission-critical rollouts and administration of global financial trading floors. The typical infrastructure size was 300-1000 machines, but the principles apply equally as well to much smaller environments. Added together these infrastructures totaled about 15,000 hosts. Further refinements have been added since then, based on experiences at NASA Ames.

The methodologies described here use UNIX and its variants as the example operating system. We have found that the principles apply equally well, and are as sorely needed, in managing infrastructures based on other operating systems.

This paper is a living document: Revisions and additions are expected and are available at www.infrastructures.org. We also maintain a mailing list for discussion of infrastructure design and implementation issues – details are available on the web site.

Introduction

There is relatively little prior art in print which addresses the problems of large infrastructures in any holistic sense. Thanks to the work of many dedicated people we now see extensive coverage of individual tools, techniques, and policies [nemeth] [frisch] [stern] [dns] [evard] [limoncelli] [anderson]. But it is difficult in practice to find a ‘‘how to put it all together’’ treatment which addresses groups of machines larger than a few dozen.

Since we could find little prior art, we set out to create it. Over the course of four years of deploying, reworking, and administering large mission-critical infrastructures, we developed a certain methodology and toolset. This development enabled thinking of an entire infrastructure as one large ‘‘virtual machine,’’ rather than as a collection of individual hosts. This change of perspective, and the decisions it invoked, made a world of difference in cost and ease of administration.

If an infrastructure is a virtual machine, then creating or reworking an infrastructure can be thought of as booting or rebooting that virtual machine. The concept of a boot sequence is a familiar thought pattern for sysadmins, and we found it to be a relatively easy one to adapt for this purpose.

We recognize that there really is no ‘‘standard’’ way to assemble or manage large infrastructures of UNIX machines. While the components that make up a typical infrastructure are generally well-known, professional infrastructure architects tend to use those components in radically different ways to accomplish the same ends. In the process, we usually write a great deal of code to glue those components together, duplicating each others’ work in incompatible ways.

Because infrastructures are usually ad hoc, setting up a new infrastructure or attempting to harness an existing unruly infrastructure can be bewildering for new sysadmins. The sequence of steps needed to develop a comprehensive infrastructure is relatively straightforward, but the discovery of that sequence can be time-consuming and fraught with error. Moreover, mistakes made in the early stages of setup or migration can be difficult to remove for the lifetime of the infrastructure.

We will discuss the sequence that we developed and offer a brief glimpse into a few of the many tools and techniques this perspective generated. If nothing else, we hope to provide a lightning rod for future discussion. We operate a web site (www.infrastructures.org) and mailing list for collaborative evolution of infrastructure designs. Many of the details missing from this paper should show up on the web site.

In our search for answers, we were heavily influenced by the MIT Athena project [athena], the OSF Distributed Computing Environment [dce], and by work done at Carnegie Mellon University [sup] [afs] and the National Institute of Standards and Technology [depot].

Infrastructure Thinking

We found that the single most useful thing a would-be infrastructure architect can do is develop a certain mindset: A good infrastructure, whether departmental, divisional, or enterprise-wide, is a single loosely-coupled virtual machine, with hundreds or thousands of hard drives and CPU's. It is there to provide a substrate for the enterprise to do its job. If it doesn't do that, then it costs the enterprise unnecessary resources compared to the benefit it provides. This extra cost is often reflected in the attitude the enterprise holds towards its systems administration staff. Providing capable, reliable infrastructures which grant easy access to applications makes users happier and tends to raise the sysadmin's quality of life. See the *Cost of Ownership* section.

This philosophy overlaps but differs from the ''dataless client'' philosophy in a subtle but important way: It discourages but does not preclude putting unique data on client hard disks, and provides ways to manage it if you do. See the *Network File Servers*, *Client File Access*, and *Client Application Management* sections.

The ''virtual machine'' concept simplified how we maintained individual hosts. Upon adopting this mindset, it immediately became clear that all nodes in a ''virtual machine'' infrastructure needed to be generic, each providing a commodity resource to the infrastructure. It became a relatively simple operation to add, delete, or replace any node. See the *Host Install Tools* section.

Likewise, catastrophic loss of any single node caused trivial impact to users. Catastrophic loss of an entire infrastructure was as easy to recover from as the loss of a single traditionally-maintained machine. See the *Disaster Recovery* section.

When we logged into a ''virtual machine,'' we expected to use the same userid and password no matter which node we logged into. Once authenticated, we were able to travel with impunity throughout the ''machine'' across other nodes without obstruction. This was true whether those nodes sat on a desktop or in a server room. In practice, this idea can be modified to include the idea of ''realms'' of security which define who can access certain protected areas of the virtual machine. You might want to implement a policy that disallows ordinary user logins on nodes of class ''NFS server,'' for instance. Note that this approach is markedly different from explicitly giving users logins on each individual machine. By classing machines, you ensure that when a new machine is added to a class, the correct users will already be able to log into it. See the *Authentication Servers* section.

Adds, moves, and changes consume a great deal of time in a traditional infrastructure because people's workstations have to be physically moved when the people move. Computing itself is enabling organizations to become more dynamic – meaning reorgs are becoming more prevalent. This makes free seating critical in modern infrastructures.

In a ''virtual machine'' infrastructure made up of commodity nodes, only the people need to move; they log off of their old workstation, walk over to their new desk, sit down, log in, and keep working. They see the same data and binaries, accessed via the same pathnames and directory structure, no matter which node they log into. This is well within the capabilities of modern automounters and NFS, particularly if you are willing to add some Perl glue and symbolic link farms. See the *Client File Access* and *Client Application Management* sections.

Traditionally, installing an application or patch means visiting each machine physically or over the net to install that package. In a ''virtual machine'' infrastructure, you ''install'' the package once by dropping it into a central repository and letting it propagate out from there to all of the hard disks. See the *File Replication Servers* and *Client OS Update Methods* sections.

The Infrastructure Bootstrap Sequence

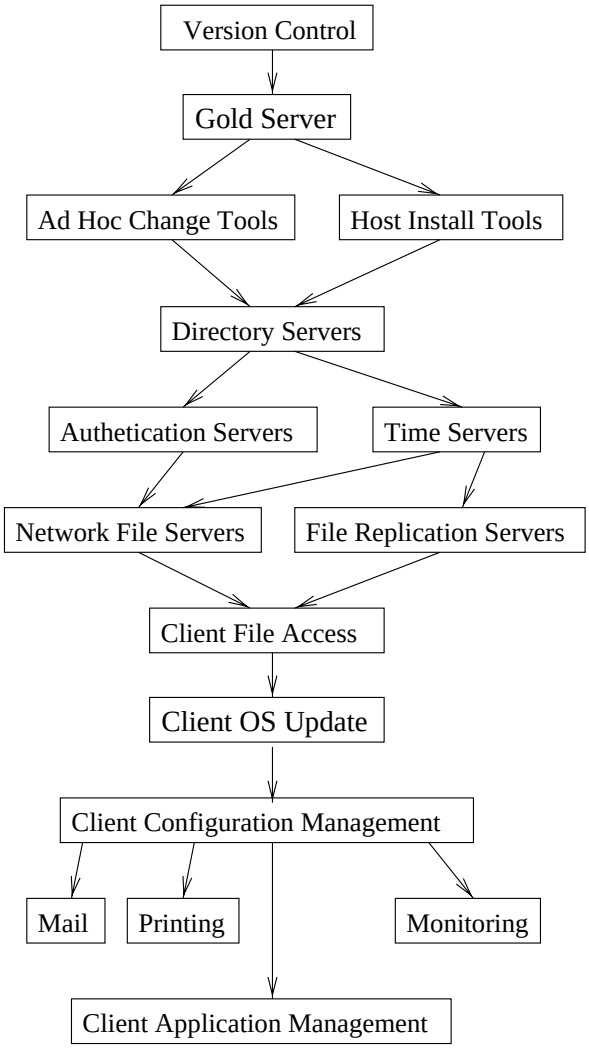
A certain sequence of events needs to occur while creating a virtual machine infrastructure. Most of these events are dependent on earlier events in the sequence. Mistakes in the sequence can cause non-obvious problems, and delaying an event usually causes a great deal of extra work to compensate for the missing functionality. These relationships are often not readily apparent in the ''heat of the moment'' of a rollout.

We found that keeping this sequence in mind was invaluable whether creating a new infrastructure from vanilla machines fresh out of the box, or migrating existing machines already in place into a more coherent infrastructure.

If you are creating a new infrastructure from scratch and do not have to migrate existing machines into it, then you can pretty much follow the bootstrap sequence as outlined below. If you have existing machines which need to be migrated, see the *Migrating From an Existing Infrastructure* section.

As mentioned earlier, the following model was developed during the course of four years of mission-critical rollouts and administration of global financial trading floors. The typical infrastructure size was 300-1000 machines, totaling about 15,000 hosts. Nothing precludes you from using this model in much smaller environments – we've used it for as few as three machines. This list was our bible and roadmap – while incomplete and possibly not in optimum order, it served its purpose. See Figure 1 for an idea of how these steps fit together.

The following sections describe these steps in more detail.



1. **Version Control** – CVS, track who made changes, backout
2. **Gold Server** – only require changes in one place
3. **Host Install Tools** – install hosts without human intervention
4. **Ad Hoc Change Tools** – 'expect', to recover from early or big problems
5. **Directory Servers** – DNS, NIS, LDAP
6. **Authentication Servers** – NIS, Kerberos
7. **Time Synchronization** – NTP
8. **Network File Servers** – NFS, AFS, SMB
9. **File Replication Servers** – SUP
10. **Client File Access** – automount, AMD, autolink
11. **Client OS Update** – rc.config, configure, make, cfengine
12. **Client Configuration Management** – cfengine, SUP, CVSup
13. **Client Application Management** – autosup, autolink
14. **Mail** – SMTP
15. **Printing** – Linux/SMB to serve both NT and UNIX
16. **Monitoring** – syslogd, paging

Figure 1: Infrastructure Bootstrap Sequence.

Step 1: Version Control

Prerequisites: none.

Lack of version control over your infrastructure leads to eventual confusion. We used version control for tracking OS configuration files, OS and application binaries and source code, and tools and administrative scripts. We managed independent evolution of several infrastructures, and were able to do rollbacks or rebuilds of damaged servers and other components.

It may seem strange to start with version control. Many sysadmins go through their entire careers without it. But infrastructure building is fundamentally a development process, and a great deal of shell, Perl, and other code tends to get generated. We found that once we got good at ''doing infrastructures,'' and started getting more work thrown at us, we had several distinct infrastructures at various stages of development at any given time. These infrastructures were often in different countries, and always varied slightly from each other. Managing code threatened to become a nightmare.

We found that CVS helped immensely in managing many different versions and branches of administrative code trees [cvs]. It took some careful thought and some tool building to be able to cram O/S configuration files and administrative code into a CVS repository and make it come back out okay on all the right machines. It was worth the effort. In later iterations, we began migrating the hundreds of megabytes of vendor-supplied O/S code itself into the CVS repositories, with some success. The latest version of CVS (1.10) has additional features which would have made this much easier, such as managing symbolic links natively.

Since all of our code was mastered from the CVS repository, we could actually destroy entire server farms and rebuild them with relative impunity during the course of development, moves, or disaster recovery. This also made it much easier to roll back from undesired changes.

In short, based on our experience, we'd strongly advise setting up and using CVS and associated tools

as the first step in an infrastructure development program.

We tried various vendor-supplied version control tools – everyone had their favorites. While many of these seemed to offer better features than CVS, none of them turned out to be flexible, robust, or WAN-centric enough to manage operating system code in-place on live machines scattered all over the world. Because of its Internet heritage and optimized use on far-flung projects [samba] [bsd], CVS was almost perfect for this. Where it wasn't, we were able to get under the hood and get what we needed in a way that we would never have been able to with a proprietary tool.

Step 2: Gold Server

Prerequisites: version control.

We used CVS to manage only one machine in each distinct infrastructure – the ''gold server.'' Changes to any other machine in the infrastructure had to propagate out from the gold server. This allowed us to make our changes reproducible, recoverable, traceable, and able to be ported and integrated into our other infrastructures. The results were rewarding: We were able to make a true migration from ''systems administrators'' to ''infrastructure engineers.'' We learned to abhor fixing the same thing twice, and got to spend our time working out fun, complex puzzles of infrastructure design (and then going home earlier).

We can't stress enough the fact that our gold server was passive. Understanding this concept is key to understanding our model. The gold server served files via NFS, SUP [sup], and CVS [cvs], and that's all. Client machines were responsible for periodically contacting the gold server to obtain updates. Neither the gold server nor any other mechanism ever ''pushed'' changes to clients asynchronously. See the *Push vs. Pull* section.

The gold server was an interesting machine; it usually was not part of the infrastructure, was usually the only one-off in the whole infrastructure, was not mission-critical in the sense that work stopped if it went down, but nevertheless the entire infrastructure grew from and was maintained by that one machine. It was the network install server, the patch server, the management and monitoring server, and was often the most protected machine from a security standpoint.

We developed a rule that worked very well in practice and saved us a lot of heartache: ''Never log into a machine to change anything on it. Always make the change on the gold server and let the change propagate out.''

We managed the gold server by maintaining it as an ordinary CVS sandbox, and then used SUP to replicate changes to client disks. It might make more sense today to use CVSup [polstra]. (See the *File Replication* section.)

We used one gold server for an entire infrastructure; this meant binaries had to be built on other platforms and transferred to the gold server's NFS or replication server trees. Other infrastructures we've seen use a different gold server for every hardware/OS combination.

Step 3: Host Install Tools

Prerequisites: Gold Server.

We managed all of our desktop machines identically, and we managed our server machines the same way we managed our desktop machines. We usually used the vendor-supplied OS install tool to place the initial disk image on new machines. The install methods we used, whether vendor-supplied or homebuilt, were usually automatic and unattended. Install images, patches, management scripts, and configuration files were always served from the gold server.

We managed desktops and servers together because it's much simpler that way. We generally found no need for separate install images, management methodologies, or backup paradigms for the two. Likewise, we had no need nor desire for separate ''workstation'' and ''server'' sysadmin groups, and the one instance this was thrust upon us for political reasons was an unqualified disaster.

The only difference between an NFS server and a user's desktop machine usually lay in whether it had external disks attached and had anything listed in /etc/exports. If more NFS daemons were needed, or a kernel tunable needed to be tweaked, then that was the job of our configuration scripts to provide for at reboot, after the machine was installed. This boot-time configuration was done on a reproducible basis, keyed by host name or class. (See the *Client OS Update* and *Client Configuration Management* sections.)

We did not want to be in the business of manually editing /etc/* on every NFS server, let alone every machine – it's boring and there are better things for humans to do. Besides, nobody ever remembers all of those custom tweaks when the boot disk dies on a major NFS server. Database, NIS, DNS, and other servers are all only variations on this theme.

Ideally, the install server is the same machine as the gold server. For very large infrastructures, we had to set up distinct install servers to handle the load of a few hundred clients all requesting new installs at or near the same time.

We usually used the most vanilla O/S image we could, often straight off the vendor CD, with no patches installed and only two or three executables added. We then added a hook in /etc/rc.local or similar to contact the gold server on first boot.

The method we used to get the image onto the target hard disk was always via the network, and we preferred the vendor-supplied network install tool, if any. For SunOS we wrote our own. For one of our infrastructures we had a huge debate over whether to use an existing in-house tool for Solaris, or whether to use Jumpstart. We ended up using both, plus a simple

'dd' via 'rsh' when neither was available. This was not a satisfactory outcome. The various tools inevitably generated slightly different images and made subsequent management more difficult. We also got too aggressive and forgot our rule about ''no patches,'' and allowed not only patches but entire applications and massive configuration changes to be applied during install on a per-host basis, using our in-house tool. This, too, was unsatisfactory from a management standpoint; the variations in configuration required a guru to sort out.

Using absolutely identical images for all machines of a given hardware architecture works better for some O/S's than for others; it worked marvelously for AIX, for instance, since the AIX kernel is never rebuilt and all RS/6000 hardware variants use the same kernel. On SunOS and Solaris we simply had to take the different processor architectures into account when classing machines, and the image install tool had to include kernel rebuilds if tunables were mandatory.

It's important to note that our install tools generally required only that a new client be plugged in, turned on, and left unattended. The result was that a couple of people were able to power up an entire floor of hundreds of machines at the same time and then go to dinner while the machines installed themselves. This magic was usually courtesy of bootp entries on the install server pointing to diskless boot images which had an ''install me'' command of some sort in the NFS-mounted /etc/rc.local. This would format the client hard drive, 'dd' or 'cpio' the correct filesystems onto it, set the hostname, domain name, and any other unique attributes, and then reboot from the hard disk.

Step 4: Ad Hoc Change Tools

Prerequisites: installed hosts in broken state running rshd, sshd, or telnetd.

Push-based ad hoc change tools such as r-commands and expect scripts are detrimental to use on a regular basis. They generally cause the machines in your infrastructure to drift relative to each other. This makes your infrastructure more expensive to maintain and makes large-scale disaster recovery infeasible. There are few instances where these tools are appropriate to use at all.

Most sysadmins are far too familiar with ad hoc change, using rsh, rcp, and rdist. We briefly debated naming this paper ''rdist is not your friend.'' If we ever write a book about enterprise infrastructures, that will be the title of one of the chapters. Many will argue that using ad hoc tools to administer a small number of machines is still the cheapest and most efficient method. We disagree. Few small infrastructures stay small. Ad hoc tools don't scale. The habits and scripts you develop based on ad hoc tools will work against you every time you are presented with a larger problem to solve.

We found that the routine use of ad hoc change tools on a functioning infrastructure was the strongest contributor towards high total cost of ownership (TCO). This seemed to be true of every operating system we encountered, including non-UNIX operating systems such as Windows NT and MacOS.

Most of the cost of desktop ownership is labor [gartner], and using ad hoc change tools increases entropy in an infrastructure, requiring proportionally increased labor. If the increased labor is applied using ad hoc tools, this increases entropy further, and so on – it's a positive-feedback cycle. Carry on like this for a short time and all of your machines will soon be unique even if they started out identical. This makes development, deployment, and maintenance of applications and administrative code extremely difficult (and expensive).

Ordinarily, any use that we did make of ad hoc tools was simply to force a machine to contact the gold server, so any changes which did take place were still under the gold server's control.

After you have done the initial image install on 300 clients and they reboot, you often find they all have some critical piece missing that prevents them from contacting the gold server. You can fix the problem on the install image and re-install the machines again, but time constraints may prevent you from doing that. In this case, you may need to apply ad hoc tools.

For instance, we usually used entries in our machines' rc.local or crontab, calling one or two executables in /usr/local/bin, to trigger a contact with the gold server (via NFS or SUP) on every boot. If any of this was broken we had to have an ad hoc way to fix it or the machine would never get updates.

Since the ''critical piece missing'' on newly installed hosts could be something like /.rhosts or hosts.equiv, that means rcp, rsh, or ssh can't be counted on. For us, that meant 'expect' [libes] was the best tool.

We developed an expect script called 'rabbit' [rabbit] which allowed us to execute arbitrary commands on an ad hoc basis on a large number of machines. It worked by logging into each of them as an appropriate user, ftp'ing a small script into /tmp, and executing it automatically.

Rabbit was also useful for triggering a pull from the gold server when we needed to propagate a change right away to hundreds of machines. Without this, we might have to wait up to an hour for a crontab entry on all the client machines to trigger the pull instead.

Step 5: Directory Servers

Prerequisites: Host Install Tools.

You'll need to provide your client machines with hostname resolution, UID and GID mappings, automount maps, and possibly other items of data that are generally read-only (this does not include

authentication – see the *Authentication Servers* section). The servers you use for these functions should be part of your infrastructure rather than standalone. The master copies of the data they serve will need to be backed up somewhere easily accessible.

You'll probably want to use DNS for hostnames, and either use NIS or file replication for UID, GID, and automounter mapping.

Here are some things to consider while choosing directory services:

- Is the protocol available on every machine you are likely to use? Some protocols, most notably NIS+, have very limited availability.
- Does it work on every machine you are likely to use? A poor implementation of NIS often forced us to use file replication instead.
- Is it unnecessarily complicated? A full-featured database with roll-back and checkpoints to perform IP service name to number mapping is probably overkill.
- How much will you have to pay to train new administrators? An esoteric, in-house system may solve the problem, but what happens when the admin who wrote and understands it leaves?
- Is it ready for prime-time? We used one product for a while for authentication services that we wanted to abandon because we kept hearing ''Oh, that is available in the next release.''

DNS, NIS and the file replication tools described in the following sections eventually all became necessary components of most of our infrastructures. DNS provided hostname to IP address mapping, as it was easy to implement and allowed subdomain admins to maintain their hosts without appealing to a corporate registry. DNS is also the standard for the Internet – a fact often lost in the depths of some corporate environments. NIS provided only the authentication mechanism, as described in the next section. NIS may not be the best choice, and we often wanted to replace it because of the adverse affects NIS has on a host when the NIS servers are all unreachable.

We wanted our machines to be able to boot with no network present. This dictated that each of our clients be a NIS slave. Pulling the maps down on an hourly or six-minute cycle and keeping hundreds of 'ypserv' daemons sane required writing a good deal of management code which ran on each client. Other infrastructures we've seen also make all clients caching DNS servers.

We recommend that directory server hosts not be unique, standalone, hand-built machines. Use your host install tools to build and configure them in a repeatable way, so they can be easily maintained and your most junior sysadmin can quickly replace them when they fail. We found that it's easy to go overboard with this though: It's important to recognize the difference between mastering the server and mastering the data it's serving. Mastering the directory database contents from the gold server generally guarantees problems unless you always use the gold server (and the same mastering mechanism) to make modifications to the database, or if you enforce periodic and frequent dumps to the gold server from the live database. Other methods of managing native directory data we've seen include cases such as mastering DNS data from a SQL database.

We used hostname aliases in DNS, and in our scripts and configuration files, to denote which hosts were offering which services. This way, we wouldn't have to edit scripts when a service moved from one host to another. For example, we had CNAMEs of 'sup' for the SUP server, 'gold' for the gold server, and 'cvs' for the CVS repository server, even though these might all be the same machine.

Step 6: Authentication Servers

Prerequisites: Directory Servers, so clients can find user info and authentication servers.

We wanted a single point of authentication for our users. We used NIS. The NIS domain name was always the same as the DNS domain name. It's possible we could have treed out NIS domains which were subsets of the DNS domain, but we didn't think we needed to.

We'd like to clarify how we differentiate between a simple directory service and an authentication service: A directory service supplies information through a one-way trust relationship – the client trusts the server to give accurate information. This trust typically comes from the fact that a local configuration file (resolv.conf, ypservers) tells the client which server to contact. This is part of an authentication service, but there is a fine distinction.

An authentication service supplies an interaction that develops a two-way trust. The client uses the service to prove itself trustworthy. The UNIX login process provides a good example of this interaction. The client (in this case, a person) enters a text string, the password. This is compared to a trusted valued by the server (the UNIX host.) If they do not match, no trust is developed. Login is denied. If they do match, the user is rewarded with control of a process operating under their name and running their shell. The whole point of an authentication service is that it allows the client to prove itself to be trustworthy, or at least to prove itself to be the same nefarious character it claims.

NIS, NIS+, Kerberos, and a raft of commercial products can be used to provide authentication services. We went through endless gyrations trying to find the ''perfect'' authentication service. We kept on ending up back at NIS, not because we liked it so much as because it was there.

It's useful to note that there are really only four elements to a user's account in UNIX – the encrypted password, the other info contained in /etc/passwd

(such as UID), the info contained in /etc/group, and the contents of the home directory. To make a user you have to create all of these. Likewise, to delete a user you have to delete all of these.

Of these four elements, the encrypted password is the most difficult to manage. The UID and GID mappings found in /etc/passwd and /etc/group can easily be distributed to clients via file replication (see the *File Replication* section). The home directory is usually best served via NFS and automounter (see the *Client File Access* section).

In shadow password implementations, the encrypted password is located in a separate database on the local host. In implementations such as NIS and Kerberos, the encrypted password and the mechanisms used to authenticate against it are moved totally off the local host onto a server machine.

We wanted to develop a single point of authentication for our users. This meant either replicating the same /etc/passwd, /etc/group, and /etc/shadow to all machines and requiring users to always change their password on a master machine, or using NIS, or installing something like Kerberos, or putting together an in-house solution.

It's interesting to note that even if we had used Kerberos we still would have needed to replicate /etc/passwd and /etc/group; Kerberos does not provide the information contained in these files.

What we usually ended up doing was using NIS and replicating /etc/passwd and /etc/group with minimal contents. This way we were able to overlay any local changes made to the files; we didn't want local users and groups proliferating.

In keeping with the ''virtual machine'' philosophy, we always retained a one-to-one mapping between the borders of the DNS and NIS domains. The NIS domain name was always the same as the DNS domain name. This gave us no leeway in terms of splitting our ''virtual machines'' up into security realms, but we found that we didn't want to; this kept things simple.

If you do want to split things up, you might try subclassing machines into different DNS subdomains, and then either use NIS+ subdomains, hack the way NIS generates and distributes its maps to create a subdomain-like behavior, or use different Kerberos realms in these DNS subdomains. Either way, these DNS subdomains would be children of a single parent DNS domain, all of which together would be the virtual machine, with only one gold server to tie them all together.

A note about username strings and keeping users happy: In today's wired world, people tend to have many login accounts, at home, at work, and with universities and professional organizations. It's helpful and will gain you many points if you allow users to pick their own login name, so they can keep all of their worlds synchronized. You don't have to look at and type that name every day – they do, over and over. They will think of you every time. You want that thought to be a positive one.

Step 7: Time Synchronization

Prerequisites: Directory Servers, so clients can find the time servers.

Without good file timestamps, backups don't work correctly and state engines such as 'make' get confused. It's important not to delay implementing time synchronization, probably by using NTP.

Many types of applications need accurate time too, including scientific, production control, and financial. It's possible for a financial trader to lose hundreds of thousands of dollars if he refers to a workstation clock which is set wrong. A $200 radio clock and NTP can be a wise investment.

Shy away from any tool which periodically pops machines into the correct time. This is the solution implemented on several PC based systems. They get their time when they connect to the server and then never update again. It works for these systems because they do not traditionally stay up for long periods of time. However, when designing for the infrastructure, it helps to think that every system will be up 24x7 for months, or even years, between reboots. Even if you put a ''time popper'' program in crontab, bizarre application behavior can still result if it resets the clock backwards a few seconds every night.

Eventually, you will implement NTP [ntp]. It is only a matter of time. NTP has become a standard for time services in the same way that DNS has become a standard for name services. The global NTP stratum hierarchy is rooted at the atomic clocks at the NIST and Woods Hole. You can't get much more authoritative. And NTP drifts clocks by slowing them down or speeding them up – no ''popping.''

If your network is connected to the Internet, your ISP may provide a good NTP time source.

Even if you need to run an isolated set of internal time servers, and sync to the outside world by radio clock or wristwatch, NTP is still the better choice because of the tools and generic knowledge pool available. But you may want to have only one stratum 1 server in this case; see the NTP docs for an explanation. You should also prefer a radio clock over the wristwatch method – see the trader example above.

For large infrastructures spread over many sites, you will want to pick two or three locations for your highest stratum NTP servers. Let these feed regional or local servers and then broadcast to the bottom tier.

Step 8: Network File Servers

Prerequisites: Directory Servers, so clients can find the file servers, Authentication Servers, so clients can verify users for file access, Time Servers, so clients agree on file timestamps.

We kept our file servers as generic and identical to each other as possible. There was little if any difference between a client and server install image. This enabled simple recovery. We generally used external hardware RAID arrays on our file servers. We often used High-Availability NFS servers [blide]. We preferred Samba [samba] when serving the same file shares to both UNIX and Windows NT clients. We were never happy with any ''corporate'' backup solutions – the only solution we've ever come close to being satisfied with on a regular basis is Amanda [amanda].

The networked UNIX filesystem has been reinvented a few times. In addition to NFS we have AFS, DFS, and CacheFS, to name a few. Of these, only NFS was available on all of our client platforms, so for us it was still the best choice. We might have been able to use AFS in most cases, but the expense, complexity, and unusual permissions structure of AFS were obstacles to its implementation. And if AFS is complex, then DFS is even more so.

One interesting network filesystem is Coda [coda]. Currently under development but already publicly available, this non-proprietary caching filesystem is freely available, and already ported to many operating systems, including Linux. It supports disconnected operation, replicated servers, and Kerberos authentication. These features when added together may make it worth the complexity of implementation.

An open-source implementation of CacheFS would also be good.

As mentioned before, we kept the disk image differences between a desktop client and an NFS server to a minimum. With few exceptions, the only differences between a desktop and server machine were whether it had external disks attached and the speed and number of processors. This made maintenance easy, and it also made disaster recovery simple.

Step 9: File Replication Servers

Prerequisites: Directory Servers, Time Synchronization.

Some configuration files will always have to be maintained on the client's local hard drive. These include much of /etc/*, and in our case, the entire /usr/local tree. How much you keep on your local disk is largely determined by how autonomous you want your machines to be. We periodically replicated changed files from the gold server to the local hard disks.

We needed a fast, incremental and mature file replication tool. We chose Carnegie Mellon's SUP (Software Upgrade Protocol) [sup]. We would have preferred a flexible, portable, open-source caching file system, but since none were available we opted for this ''poor man's caching'' instead. It worked very well.

Aside from the advantage of incremental updates, SUP offered a strict ''pull'' methodology. The client, not the server, chose the point in time when it would be updated. (See the *Push vs. Pull* section.)

Using this mechanism, we were able to synchronize the files in /etc on every client every six minutes, and the contents of /usr/local every hour. (This on a trading floor with over 800 clients.)

We also replicated selected applications from the NFS servers to the client hard disks. (See the *Client Application Management* section.)

We used SUP to replicate most of the files that NIS normally manages, like /etc/services and the automounter maps. We only used NIS to manage authentication – the passwd map.

A more recent development, familiar to many open source developers and users, is CVSup [polstra]. With ordinary SUP, we had to do a 'cvs update' in the replication source tree on the gold server to check the latest changes out of the CVS repository. We then used SUP jobs in crontab to pull the changes from there down to the client. Today it may make more sense to skip the intermediary step, and instead use CVSup to pull files and deltas directly from the CVS repository into the live locations on the client hard disks.

Step 10: Client File Access

Prerequisites: Network File Servers, File Replication Servers.

We wanted a uniform filesystem namespace across our entire virtual machine. We were able to move data from server to server without changing pathnames on the clients. We also were able to move binaries from servers to client disks or back without changing the pathnames the binaries were executed from. We used automounters and symlink farms extensively. We would have liked to see good open-source caching filesystems.

CacheFS was ruled out as a general solution because of its limited heterogeneity. We might have been able to use CacheFS on those clients that offered it, but that would have required significantly different management code on those clients, and time constraints prevented us from developing this further.

In keeping with the virtual machine concept, it is important that every process on every host see the exact same file namespace. This allows applications and users to always find their data and home directories in the same place, regardless of which host they're on. Likewise, users will always be able to find their applications at the same pathname regardless of hardware platform.

If some clients have an application installed locally, and others access the same application from a file server, they both should ''see'' the application in the same place in the directory tree of the virtual machine. We used symbolic link ''farms'' in the /apps directory that pointed to either /local/apps or

/remote/apps, depending on whether the application was installed locally or remotely. The /local/apps filesystem was on the client hard disk, while /remote/apps was composed of automounted filesystems from NFS servers. [mott]

One tiny clue to better understanding of our model is this: the directories served by an NFS server were always served from /local/apps on the server itself. Also, /usr/local was always a symlink to /local. One of our tenets was that all data unique to a machine and not part of the OS be stored in /local. This way we could usually grab all of the critical and irreplaceable uniqueness of a machine by grabbing the contents of /local. (OS-related uniqueness goes in /var, as always.)

The automounter has some pitfalls: Indirect mounts are more flexible than direct mounts, and are usually less buggy. If a vendor's application insists that it must live at /usr/appname and you want to keep that application on a central server, resist the temptation to simply mount or direct automount the directory to /usr/appname. UNIX provides the symbolic link to solve this problem. Point the /usr/appname symlink at an indirect mapped /remote/apps (or similar) directory. Similarly, a common data directory (perhaps, /data) managed by an indirect map could be defined for any shared data that must be writable by the clients.

Another serious danger is the use of /net. Automounters have the ability to make all exports from a server appear at /net/servername or something similar. This is very handy for trouble-shooting and quick maintenance hacks. It can, however, put an oppressive load on the server if the server is exporting a large number of filesystems – cd'ing to /net/scotty will generate a mount request for all of scotty's filesystems at once. Worse, it reduces the flexibility of your infrastructure, because host names become a part of the file name. This prevents you from moving a file to a new server without changing every script and configuration file which refers to it.

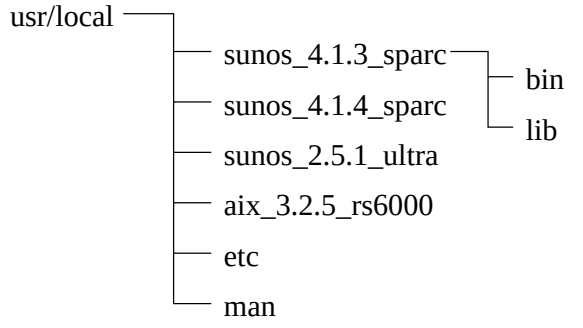


Figure 2: Example of a Heterogeneous /usr/local SUP Server Tree.

It was difficult for us to come up with a heterogeneous filesystem naming convention. We finally settled on installing a script (/usr/local/bin/platform) on every machine which, when run, spit out a formatted version of the output of 'uname -a'. The naming convention we used looked something like 'sunos_4.1.4_sparc', 'sunos_5.1.5_ultra', and 'aix_3.2.5_rs6000'. This script was called from everywhere; automounters, boot scripts, application startup scripts, and the makefile described below. We used this platform string in many places, including heterogeneous directory paths. See Figure 2. We made 'platform' a script, not a simple data file, to guard against the possibility that out-of-date information would cause errors.

Step 11: Client O/S Update

Prerequisites: Network File Servers, File Replication Servers.

Vendors are waking up to the need for decent, large scale operating systems upgrade tools. Unfortunately, due to the ''value added'' nature of such tools, and the lack of published standards, the various vendors are not sharing or cooperating with one another. It is risky to use these tools even if you think you will always have only one vendor to deal with. In today's business world of mergers and reorgs, single vendor networks become a hodge-podge of conflicting heterogeneous networks overnight.

We started our work on a homogeneous network of systems. Eventually we added a second, and then a third OS to that network. We took about five months adding the second OS. When the third came along, we found that adding it to our network was a simple matter of porting the tools – it took about a week. Our primary tool was a collection of scripts and binaries that we called Hostkeeper.

Hostkeeper depended on two basic mechanisms; boot time configuration and ongoing maintenance. At boot, the Hostkeeper client contacted the gold server to determine whether it had the latest patches and upgrades applied to its operating system image. This contact was via an NFS filesystem (/is/conf) mounted from the gold server.

We used 'make' for our state engine. Each client always ran 'make' on every reboot. Each OS/hardware platform had a makefile associated with it (/is/conf/bin/Makefile.{platform}). The targets in the makefile were tags that represented either our own internal revision levels or patches that made up the revision levels. We borrowed a term from the aerospace industry – ''block 00'' was a vanilla machine, ''block 10'' was with the first layer of patches installed, and so on. The Makefiles looked something like Listing 1. Note the 'touch' commands at the end of each patch stanza; this prevented 'make' from running the same stanza on the same machine ever again. (We ran 'make' in a local directory where these timestamp files were stored on each machine.)

We had mechanisms that allowed us to manage custom patches and configuration changes on selected machines. These were usually driven by environment variables set in /etc/environment or the equivalent.

The time required to write and debug a patch script and add it to the makefile was minimal compared to the time it would have taken to apply the same patch to over 200 clients by hand, then to all new machines after that. Even simple changes, such as configuring a client to use a multi-headed display, were scripted. This strict discipline allowed us to exactly recreate a machine in case of disaster.

For operating systems which provided a patch mechanism like 'pkgadd', these scripts were easy to write. For others we had our own methods. These days we would probably use RPM for the latter [rpm].

You may recognize many of the functions of 'cfengine' in the above description [burgess]. At the time we started on this project, 'cfengine' was in its early stages of development, though we were still tempted to use it. If we had this to do again it's likely 'cfengine' would have supplanted 'make'.

One tool that bears closer scrutiny is Sun Microsystems' Autoclient. The Autoclient model can best be described as a dataless client whose local files are a cached mirror of the server. The basic strategy of Autoclient is to provide the client with a local disk drive to hold the operating system, and to refresh that operating system (using Sun's CacheFS feature) from a central server. This is a big improvement over the old diskless client offering from Sun, which overloaded servers and networks with NFS traffic.

One downside of Autoclient is its dependence on Sun's proprietary CacheFS mechanism; another is its scalability. Eventually, the number of clients will exceed that which one server can support. This means adding a second server, then a third, and then the problem becomes one of keeping the servers in sync. Essentially, Autoclient does not solve the problem of system synchronization; it delays it. However, this delay may be exactly what the system administrator needs to get a grip on a chaotic infrastructure.

Step 12: Client Configuration Management

Prerequisites: Network File Servers, File Replication Servers.

In a nutshell, client configuration is localization. This includes everything that makes a host unique, or that makes a host a participant of a particular group or domain. For example, hostname and IP addresses must be different on every host. The contents of /etc/resolv.conf should be similar, if not identical, on hosts that occupy the same subnet. Automount maps which deliver users' home directories must be the same for every host in an authentication domain. The entries in client crontabs need to be mastered from the gold server.

Fortunately, if you have followed the roadmap above, most of this will fall into place nicely. If you fully implemented file replication and O/S update, these same mechanisms can be used to perform client configuration management. If not, do something now. You must be able to maintain /etc/* without manually logging into machines, or you will soon be spending all of your time pushing out ad hoc changes.

Earlier, we mentioned the Carnegie Mellon Software Update Protocol (SUP). SUP replicated files for us. These files included the /etc/services file, automount maps, many other maps that are normally served by NIS, and the typical suite of gnu tools and other open-source utilities usually found in /usr/local on UNIX systems. In each case, we generalized what we could so every client had identical files. Where this was not practical (clients running cron jobs, clients acting as DNS secondaries, etc.), we applied a simple rule: send a configuration file and a script to massage it into place on the client's hard disk. SUP provided this ''replicate then execute'' mechanism for us so we had little need to add custom code.

```
block00: localize
block10: block00 14235-43 xdm_fix01
14235-43 xdm_fix01:
         /is/conf/patches/$(PLATFORM)/$@/install_patch
         touch $@
localize:
         /is/conf/bin/localize
         touch $@
```

Listing 1: Hostkeeper makefile example.

```
root:all:1 2 * * * [-x /usr/sbin/rtc] && /usr/sbin/rtc -c > /dev/null 2>&1
root:all:0 2 * * 0,4 /etc/cron.d/logchecker
root:all:5 4 * * 6   /usr/lib/newsyslog
root:scotty:0 4 * * * find . -fstype nfs -prune -o -print >/var/spool/lsR
stevegt:skywalker:10 0-7,19-23 * * * /etc/reset_tiv
[...]
```

Listing 2: Crontab.master file.

In most cases we ran SUP from either a cron job or a daemon script started from /etc/inittab. This generally triggered replications every few minutes for frequently-changed files, or every hour for infrequently changed files.

The tool we used for managing client crontabs was something we wrote called 'crontabber' [crontabber]. It worked by looking in /etc/crontab.master (which was SUPed to all client machines) for crontab entries keyed by username and hostname. The script was executed on each client by SUP, and execution was triggered by an update of crontab.master itself. The crontab.master file looked something similar to Listing 2.

Step 13: Client Application Management

Prerequisites: Client Configuration Management.

Everything up to this point has been substrate for applications to run on – and we need to remember that applications are the only reason the infrastructure exists in the first place. This is where we make or break our infrastructure's perception in the eyes of our users.

We wanted location transparency for every application running on any host in our ''virtual machine.'' We wanted the apparent location and directory structure to be identical whether the application was installed on the local disk or on a remote file server. To accomplish this, we used SUP to maintain identical installations of selected applications on local disks, automounted application directories for NFS-served apps, and Perl-managed symbolic link farms to glue it all together [mott].

A heterogeneous and readily available caching filesystem would have been much simpler to understand, and as mentioned before we originally considered AFS.

We made all applications available for execution on all hosts, regardless of where the application binaries physically resided. At first, it may seem strange that a secretary might have the ability to run a CAD program, but an ASIC engineer will certainly appreciate the fact that, when their own workstation fails, the secretary's machine can do the job (see the *Disaster Recovery* section).

We executed our apps from /apps/application_name. We had the automounter deliver these binaries, not to /apps, but to /remote/apps/ application_name. We then created a symbolic link farm in /apps. The link farm simply pointed to the /remote/apps directories of the same name.

To support the extra speed we needed for some applications, we used SUP to replicate the application from the NFS server into the /local/apps/application_name directory on the client hard disk. The Perl code which drove SUP referred to a flat file (/etc/autosup.map) which listed applications to be replicated on particular machines. We inspiringly dubbed this code 'autosup' [autosup]. The autosup.map file looked something like:

```
scotty: elm wingz escapade metrics
luna: elm wingz
[...]
```

After 'autosup' updated the local copies of applications, possibly adding or deleting entire apps, another Perl script, 'autolink', updated the symbolic link farm to select the ''best'' destination for each /apps symlink. The selection of the best destination was made by simply ordering the autolink targets (in /etc/autolink.map) so that more preferential locations overrode less preferential locations. The autolink.map file usually looked something like Listing 3. The trivial example in Listing 4 shows how the symbolic links in /apps would look with a CAD package installed locally, and TeX installed on a file server.

The 'autosup' script was usually triggered by a nightly crontab which SUPed down the new autosup.map, and 'autolink' was usually triggered by 'autosup'.

It is important to note that part of application management is developer management. At first, many

```
# create          from
#
/apps           /remote/apps
/apps           /local/apps
/apps/pub       /remote/apps/pub
#
/prd/sw          /net/${HOMESERVER}/export/apps${HOE}/prd/sw
/prd/sw          /local/apps1/prd/sw
[...]
```

Listing 3: autolink.map file.

```
/apps/CAD-----> /local/apps/CAD    /remote/apps/CAD (ignored)
/apps/TeX            ----->        /remote/apps/TeX
```

Listing 4: /apps link farm examples.

of our application developers loved to have their programs write files in the directory tree that contained their program, and they tended to hardcode pathnames to other binaries. We consider this a bad thing. For our in-house developers we managed to convince them to refer to environment variables for where data and binaries lived. For external applications we had to do tricks with symlinks.

Step 14: Mail

Prerequisites: Client Configuration Management.

Now that you have a way of managing sendmail.cf on client hard disks you can set up mail. Avoid like the plague any attempts to use non-SMTP mail solutions – the world has gone SMTP, and there are now many fine GUI SMTP mail readers available. Proprietary solutions are no longer necessary for user-friendliness. We used NFS-mounted mail spools: POP or IMAP would probably be the better choice today.

Step 15: Printing

Prerequisites: Client Configuration Management.

During the first few days after any new infrastructure went live, we usually spent about 80% of our time fixing unforeseen printing problems. Printers will eat your lunch. Assuming you can pick your printers, use high-quality postscript printers exclusively.

The best print infrastructure we've seen by far is the one a major router vendor uses internally – 90 Linux print servers worldwide spooling to 2000 printers, seamlessly and reliably providing print service to thousands of UNIX and NT clients via Samba [samba]. The details of this infrastructure have not been released to the public as of the time this paper goes to press – check www.infrastructures.org for an update.

Step 16: Monitoring

Prerequisites: Client Application Management.

When all of the above was done, we found very little monitoring was needed – the machines pretty much took care of themselves. We never got around to setting up a central syslogd server, but we should have. We only had paging working spottily at best. These days, with most alpha paging vendors providing free e-mail gateways, this should be much easier. Otherwise, you may want to take a look at the Network Paging Protocol (SNPP) support in HylaFAX. [hylafax]

Migrating From an Existing Infrastructure

Think of a migration as booting a new virtual machine, and migrating your old hardware into the new virtual machine.

The first infrastructure we used to develop this model was in fact one that had started chaotically, as four desktop machines that were administered by the application developers who sat in front of them. As the internal application they developed became successful, the infrastructure grew rapidly, and soon consisted of 300 machines scattered worldwide. At the time we embarked on this effort, these 300 machines were each unique, standalone hosts – not even DNS or NIS were turned on. This state of affairs is probably all too typical in both large and small organizations.

If you are migrating existing machines from an old infrastructure (or no infrastructure) into a new infrastructure, you will want to set up the infrastructure-wide services (like NIS, DNS, and NFS) first. Then, for each desktop host:

1. Create a replacement host using your chosen "Host Install" tool as described in this paper.
2. Have the user log off.
3. Migrate their data from their old workstation to an NFS server.
4. Add the new NFS-served directory to the automounter maps so the new host can find it.
5. Drop the new client on the user's desk.

This may sound impossible if each of your desktop hosts have unique filesystem layouts, or still have a need to retain unique data on their own hard disk. But we were able to accommodate some of these variations with some thought, and get rid of the rest. Some of the ways we did this are described in the sections above.

We found it to be much easier and more effective in the long run to roll through an existing infrastructure replacing and rebuilding hosts, rather than trying to converge a few files at a time on the existing hosts. We tried both. Where we replaced hosts, a 100-host infrastructure could be fully converted to the new world order in under three months, with one sysadmin working at it half-time. User impact was limited to the time it took to swap a host. Where we instead tried to bring order out of chaos by changing one file at a time on all hosts in an infrastructure, we were still converging a year later. User impact in this case was in the form of ongoing and frustrating changes to their world, and prolonged waits for promised functionality.

Disaster Recovery

The fewer unique bytes you have on any host's hard drive, the better – always think about how you would be able to quickly (and with the least skilled person in your group) recreate that hard drive if it were to fail.

The test we used when designing infrastructures was "Can I grab a random machine and throw it out the tenth-floor window without adversely impacting users for more than 10 minutes?" If the answer to this was "yes," then we knew we were doing things right.

Likewise, if the entire infrastructure, our "virtual machine," were to fail, due to power outage or terrorist bomb (this was New York, right?), then we should expect replacement of the whole infrastructure to be

no more time-consuming than replacement of a conventionally-managed UNIX host.

We originally started with two independent infrastructures – developers, who we used as beta testers for infrastructure code; and traders, who were in a separate production floor infrastructure, in another building, on a different power grid and PBX switch. This gave us the unexpected side benefit of having two nearly duplicate infrastructures – we were able to very successfully use the development infrastructure as the disaster-recovery site for the trading floor.

In tests we were able to recover the entire production floor – including servers – in under two hours. We did this by co-opting our development infrastructure. This gave us full recovery of applications, business data, and even the contents of traders' home directories and their desktop color settings. This was done with no hardware shared between the two infrastructures, and with no ''standby'' hardware collecting dust, other than the disk space needed to periodically replicate the production data and applications into a protected space on the development servers. We don't have space here to detail how the failover was done, but you can deduce much of it by thinking of the two infrastructures as two single machines – how would you allow one to take on the duties and identity of the other in a crisis? With an entire infrastructure managed as one virtual machine you can have this kind of flexibility. Change the name and reboot...

If you recall, in our model the DNS domain name was the name of the ''virtual machine.'' You may also recall that we normally used meaningful CNAMES for server hosts – gold.mydom.com, sup.mydom.com, and so on. Both of these facts were integral to the failover scenario mentioned in the previous paragraph, and should give you more clues as to how we did it.

Push vs. Pull

We swear by a pull methodology for maintaining infrastructures, using a tool like SUP, CVSup, or 'cfengine'. Rather than push changes out to clients, each individual client machine needs to be responsible for polling the gold server at boot, and periodically afterwards, to maintain its own rev level.

Before adopting this viewpoint, we developed extensive push-based scripts based on rsh, rcp, and rdist.

The problem we found with the r-commands was this: When you run an r-command based script to push a change out to your target machines, odds are that if you have more than 30 target hosts one of them will be down at any given time. Maintaining the list of commissioned machines becomes a nightmare.

In the course of writing code to correct for this, you will end up with elaborate wrapper code to deal with: timeouts from dead hosts; logging and retrying dead hosts; forking and running parallel jobs to try to hit many hosts in a reasonable amount of time; and finally detecting and preventing the case of using up all available TCP sockets on the source machine with all of the outbound rsh sessions.

Then you still have the problem of getting whatever you just did into the install images for all new hosts to be installed in the future, as well as repeating it for any hosts that die and have to be rebuilt tomorrow.

After the trouble we went through to implement r-command based replication, we found it's just not worth it. We don't plan on managing an infrastructure with r-commands again, or with any other push mechanism for that matter. They don't scale as well as pull-based methods.

Cost of Ownership

Cost of ownership is priced not only in dollars but in lives. A career in Systems Administration is all too often a life of late nights, poor health, long weekends, and broken homes.

We as an industry need to raise the bar for acceptable cost of administration of large numbers of machines. Most of the cost of systems administration is labor [gartner]. We were able to reduce this cost enough that, while the number of machines we were administering grew exponentially, our group only grew linearly. And we all got to spend more nights and weekends at home.

While we were unable to isolate any hard numbers, to us it appears that, by using the techniques described in this paper, systems administration costs can be reduced by as much as an order of magnitude, while at the same time providing higher levels of service to users and reducing the load on the systems administrators themselves.

SysAdmin or Infrastructure Architect?

There's a career slant to all of this.

Infrastructure architects typically develop themselves via a systems administration career track. That creates a dilemma. A systems administration background is crucial for the development of a good infrastructure architect, but we have found that the skillset, project time horizon, and coding habits needed by an infrastructure architect are often orthogonal to those of a systems administrator – an architect is not the same animal as a senior sysadmin.

We have found, in the roles of both manager and contractor, that this causes no end of confusion and expense when it comes to recruiting, interviewing, hiring, writing and reading resumes, and trying to market yourself. Recruiters generally don't even know what an ''infrastructure architect'' is, and far too often assume that ''senior sysadmin'' means you know how to flip tapes faster. Most of us at one time or another

have been restricted from improving a broken infrastructure, simply because it didn't fit within our job description.

In order to improve this situation, we might suggest that ''infrastructure architect'' be added to the SAGE job descriptions, and USENIX and affiliate organizations help promulgate this ''new'' career path. We'd like to see more discussion of this though. Is an IA an advanced version of a sysadmin, or are they divergent?

There seems to us to be a mindset – more than skillset – difference between a sysadmin and an architect.

Some of the most capable systems administrators we've known are not interested in coding (though they may be skilled at it). When given a choice they will still spend most of their time manually changing things by logging into machines, and don't mind repetitive work. They tend to prefer this direct approach. They can be indispensable in terms of maintaining existing systems.

As mentioned before, infrastructure architects tend to spend most of their time writing code. They are motivated by challenges and impatience – they hate doing the same thing twice. When allowed to form a vision of a better future and run with it they, too, can be indispensable. They provide directed progress in infrastructures which would otherwise grow chaotically.

While most people fall somewhere between these two extremes, this difference in interests is there – it may not be fair or correct to assume that one is a more advanced version of the other. Resolving this question will be key to improving the state of the art of enterprise infrastructures.

Conclusion

There are many other ways this work could have been done, and many inconsistencies in the way we did things. One fact that astute readers will spot, for instance, is the way we used both file replication and a makefile to enact changes on client disks. While this rarely caused problems in practice, the most appropriate use of these two functions could stand to be more clearly defined. We welcome any and all feedback.

This is the paper we wish we could have read many years ago. We hope that by passing along this information we've aided someone, somewhere, years in the future. If you are interested in providing feedback on this paper and helping improve the state of the art, we'd like to welcome you to our web site: Updates to this paper as well as code and contributions from others will be available at www.infrastructures.org.

Acknowledgments

Karen Collins has probably read this paper more times than any human alive. The good wording is hers – the rest is ours. Her tireless attention to details and grammar were crucial, and the errors which remain are due to our procrastination rather than her proofreading ability. We'd hire her as a technical writer any time. Next time we'll start a month earlier, Karen.

Many thanks go to Pamela Huddleston for her support and patience, and for providing the time of her husband.

We were extremely fortunate in getting Eric Anderson as our LISA ''shepherd'' – many substantial refinements are his. Rob Kolstad was extremely patient with our follies. Celeste Stokely encouraged one of us to go into systems administration, once upon a time.

We'd like to thank NASA Ames Research Center and Sterling Software for the supportive environment and funding they provided for finishing this work – it would have been impossible otherwise.

Most of all, we'd like to thank George Sherman for his vision, skill, compassion, and tenacity over the years. He championed the work that this paper discusses. We're still working at it, George.

This paper discusses work performed and influenced by Spencer Westwood, Matthew Buller, Richard Hudson, Jon Sober, J. P. Altier, William Meenagh, George Ott, Arash Jahangir, Robert Burton, Jerzy Baranowski, Joseph Gaddy, Rush Taggart, David Ardley, Jason Boud, Lloyd Salmon, George Villanueva, Glenn Augenstein, Gary Merinstein, Guillermo Gomez, Chris York, Robert Ryan, Julie Collinge, Antonio DiCaro, Victoria Sadoff, James McMichael, Mark Blackmore, Matt Martin, Nils Eliassen, Richard Benzell, Matt Forsdyke, and many, many others over the course of four years, in three cities, spanning two continents. Best wishes to you all. If we missed your name we owe you dinner at Maggie's someday.

Author Information

Steve Traugott taught himself BASIC while standing up in Radio Shack in front of a TRS-80 Model I. At Paradyne he was a modem internals technician when modems were more expensive than cars, and dabbled in COBOL on an IBM/370 clone he built behind his desk. He decided to stop all of that when Challenger exploded within view of his office, and became an F-15 and AC-130 gunship crew chief in the U.S. Air Force to gain a better understanding of the aerospace industry. After realizing that aerospace needs better computing, he returned to civilian life at IBM to port OSF/1 to their mainframe family, worked on the last releases of System V UNIX at AT&T in 1993, and experienced DCE at Digital Equipment Corporation. He became a senior architect, then Vice President of trading floor infrastructure engineering for Chemical and Chase Manhattan banks, then escaped from New York for a contract at Cisco Systems. He has now found a home in the supercomput-

ing branch of NASA Ames Research Center, in Silicon Valley.

Joel Huddleston also taught himself BASIC while standing up in Radio Shack in front of a TRS-80 Model I. He began his computer career at Geophysical Services, Inc. operating a TIMAP II computer that still used punched cards for job entry. After a distinguished and lengthy career as a college student and part-time programmer/systems administrator at Texas A&M University, Mr. Huddleston entered the ''Real World'' when he discovered that students cannot be tenured. As a computer consultant for SprintParanet, Mr. Huddleston worked for such diverse firms as CompUSA, Motel 6, and Chase Manhattan Bank on projects ranging from a 100 seat Banyan/Netware migration to the design of 800+ seat Solaris trading floors.

References

[afs] John H. Howard, Carnegie Mellon University, ''On Overview of the Andrew File System,'' *USENIX Conference Proceedings*, Winter, 1988.

[amanda] *Advanced Maryland Automatic Network Disk Archiver,* http://www.cs.umd.edu/projects/amanda/index.html .

[anderson] Paul Anderson, ''Towards a High-Level Machine Configuration System,'' *USENIX Proceedings: Eighth Systems Administration Conference (LISA '94)*.

[athena] G. Winfield Treese, ''Berkeley UNIX on 1000 Workstations: Athena Changes to 4.3BSD,'' *USENIX Conference Proceedings*, Winter, 1988.

[autosup] *Automatic Application Replication Tool*, available from http://www.infrastructures.org .

[blide] A. Bhide, E. N. Elnozahy, and S. P. Morgan. ''A Highly Available Network File Server,'' *Proceedings of the 1991 USENIX Winter Conference,* 1991.

[bsd] *BSDlite 4.4, FreeBSD, NetBSD, OpenBSD Distribution Repository,* http://www.freebsd.org/cgi/cvsweb.cgi .

[burgess] Mark Burgess, *GNU Cfengine,* http://www.iu.hioslo.no/~mark/cfengine/ .

[coda] *Coda Distributed Filesystem,* http://www.coda.cs.cmu.edu .

[crontabber] *Client Crontab Management Script,* available from http://www.infrastructures.org .

[cvs] *Concurrent Versions System,* http://www.cyclic.com/cvs/info.html .

[dce] *OSF Distributed Computing Environment,* http://www.opengroup.org/dce/ .

[depot] Ken Manheimer, Barry Warsaw, Steve Clark, Walter Rowe, ''The Depot: A Framework for Sharing Software Installation Across Organizational and UNIX Platform Boundaries,'' *USENIX LISA IV Conference Proceedings,* October 24-25, 1991.

[dns] Paul Albitz & Cricket Liu, *DNS and BIND, 2nd Edition,* O'Reilly & Associates, 1996.

[evard] Rémy Evard, ''An Analysis of UNIX System Configuration,'' *USENIX Proceedings: Eleventh Systems Administration Conference (LISA '97)*, October 26-31, 1997.

[hylafax] *Using HylaFAX as an SNPP Server,* http://www.vix.com/hylafax/ixotap.html .

[libes] Don Libes, *Exploring Expect,* ISBN 1-56592-090-2, O'Reilly & Associates, 1994.

[frisch] AEleen Frisch, *Essential System Administration, 2nd Edition,* O'Reilly & Associates, 1995.

[gartner] Cappuccio, D., Keyworth, B., and Kirwin, W., *Total Cost of Ownership: The Impact of System Management Tools,* Gartner Group, 1996.

[hagemark] Bent Hagemark, Kenneth Zadeck, ''Site: A Language and System for Configuring Many Computers as One Computing Site,'' *USENIX Proceedings: Large Installation Systems Administration III Workshop Proceedings,* September 7-8, 1989.

[limoncelli] Tom Limoncelli, ''Turning the Corner: Upgrading Yourself from 'System Clerk' to 'System Advocate','' *USENIX Proceedings: Eleventh Systems Administration Conference (LISA '97)*, October 26-31, 1997.

[lirov] Yuval Lirov, *Mission-Critical Systems Management,* Prentice Hall, 1997.

[mott] Arch Mott, ''Link Globally, Act Locally: A Centrally Maintained Database of Symlinks,'' *USENIX LISA V Conference Proceedings,* September 30-October 3, 1991.

[nemeth] Evi Nemeth, Garth Snyder, Scott Seebass, Trent R. Hein, *UNIX System Administration Handbook, second edition,* Prentice Hall, 1995.

[ntp] *Network Time Protocol Home Page,* http://www.eecis.udel.edu/~ntp/ .

[polstra] John D. Polstra, *CVSup Home Page and FAQ,* http://www.polstra.com/projects/freeware/CVSup/ .

[rabbit] *'rabbit' expect script for ad hoc changes,* available from http://www.infrastructures.org .

[rpm] *Red Hat Package Manager – open source package tool,* http://www.rpm.org .

[rudorfer] Gottfried Rudorfer, ''Managing PC Operating Systems with a Revision Control System,'' *USENIX Proceedings: Eleventh Systems Administration Conference (LISA '97),* October 26-31, 1997.

[samba] *CVS Access to Samba Source Tree,* http://samba.anu.edu.au/cvs.html .

[shafer] Steven Shafer and Mary Thompson, *The SUP Software Upgrade Protocol,* http://www.cs.cmu.edu/afs/cs.cmu.edu/project/mach/public/sup/sup.ps, 8 September, 1989.

[stern] Hal Stern, *Managing NFS and NIS,* O'Reilly & Associates, 1991.

[stokely] Extensive UNIX resources at Stokely Consulting – thanks Celeste!, http://www.stokely.com/ .

[sup] *Carnegie Mellon Software Upgrade Protocol,* http://www.cs.cmu.edu/afs/cs.cmu.edu/project/mach/public/sup/sup.tar.gz, ftp://sunsite.unc.edu/pub/Linux/system/network/management/sup.tar.gz .