Enterprise Rollouts with JumpStart

Jason Heiss
Collective Technologies
jheiss@colltech.com

Abstract

JumpStart[tm] is Sun's solution for installing the Solaris operating system. The Custom JumpStart feature allows the installation process to be automated. However, configuring boot and NFS servers and the appropriate name services for JumpStart is a time-consuming and error-prone process. The scripts that Sun provides do not help this process much. This paper will talk about how to better automate the configuration steps needed to perform JumpStarts over the network, emphasizing speed and accuracy as well as simplicity from an operator's point of view. The infrastructure needed to perform large numbers of simultaneous JumpStarts is also discussed. By automating the actual rollout process, it is possible to JumpStart several hundred machines at once. Techniques for doing so are presented. The improvements presented in this paper allowed one company to improve the speed of their rollouts to 600 Sun workstations by more than an order of magnitude. This greatly decreased user downtime and saved the company hundreds of thousands of dollars. Lastly, possible future improvements to the process are discussed.

Introduction

JumpStart is Sun's solution for installing the Solaris operating system. JumpStart has three modes. The Interactive and Web Start modes are for interactive installs. While fine for installing Solaris on a single machine, this clearly doesn't scale to the enterprise level. The third mode is what Sun calls Custom JumpStart, which allows the install process to be automated. The information in this paper is based on the Sparc architecture. Solaris and JumpStart are available for the Intel architecture but the limitations of that architecture make large-scale rollouts difficult. Specifically, booting a PC off of the network is generally not possible. Instead, booting off of a floppy is usually required. This requires generating hundreds of boot floppies and visiting each machine to insert one. Configuring a Custom JumpStart is already well covered in Sun's book *Automating Solaris Installations* [Kas95], with Asim Zuberi's article "Jumpstart in a nutshell" [Zub99] presenting some useful updates, as well as Sun's documentation in the Solaris Advanced Installation Guide [SAIG]; thus those topics will not be covered here. Instead, this paper will talk about how to automate rollouts using JumpStart. While Custom JumpStart makes it possible to automate the process of installing Solaris (partitioning disks, installing packages, etc.), it is still a manual process to configure boot and NFS servers and name services so that the workstations can boot off of the network and start the Custom JumpStart. Automating that configuration process and performing the rollout itself are the topics of this paper. The infrastructure required for large-scale rollouts will also be discussed.

Interactive Installs

There are a number of problems with interactive installs (either Interactive or Web Start) on an enterprise scale. First, they are very laborious. Interactive installs take at least one hour per machine. While good for system administrator job security, this is expensive and tiresome. It also makes it difficult to get timely upgrades to the users. Second, with interactive installs it is tempting to customize the install for each user. This inconsistency in the installs makes future maintenance more difficult. A little more design and research work up front will usually allow an administrator to develop a standard installation that meets the needs of all users.

Custom JumpStart

Let's review how a Custom JumpStart works. Custom JumpStart makes it possible to automate the entire installation process. A Custom JumpStart begins with the client workstation booting off of the network in exactly the same way as a diskless workstation.

The first thing the client needs to do is get its IP address. This allows it to send IP packets onto the local subnet. The client gets its IP address using a protocol called RARP, or Reverse Address Resolution Protocol. The client knows its ethernet address but not its IP address. So it broadcasts a packet out on to the subnet that says, "My ethernet address is xx:xx:xx:xx:xx:xx, does anyone know my IP address?" With Solaris (and most other operating systems), you need to be running a special server to listen for and respond to RARP packets. This server is called `in.rarpd` in Solaris. The server listens for RARP packets, and when it receives one it attempts to lookup the ethernet address in `/etc/ethers` or the `ethers` NIS/NIS+ map (depending on the `ethers` setting in `/etc/nsswitch.conf`). If that is successful, the server gets the hostname for the machine. Next, the server attempts to lookup the IP address for that hostname. If that is successful, the server responds to the client with a packet saying "Hello xx:xx:xx:xx:xx:xx, your IP address is xxx.xxx.xxx.xxx." Note: the presence of a `/tftpboot` directory causes the standard Solaris initialization scripts to run `in.rarpd` and `rpc.bootparamd` (more on bootparams later).

The client now has its IP address and the next thing it needs is the `inetboot` boot block. This is a very basic kernel that knows how to do enough IP networking to NFS mount a root directory and load the real Solaris kernel. The client downloads the `inetboot` file using a protocol called TFTP, or Trivial File Transfer Protocol. It has no authentication or security and very basic error handling. This makes it easy to implement TFTP in the limited memory of the Sun PROM but also makes the protocol very slow and a security risk. Speed, however, is not an issue with the size of the `inetboot` file (~150 kB). The security implications of TFTP will need to be handled in a way that complies with local security policy. In general, hosts don't run TFTP servers. Under Solaris, the TFTP server is started by `inetd`. By default, the entry in `/etc/inetd.conf` for `in.tftpd`, the Solaris TFTP server, is commented out.

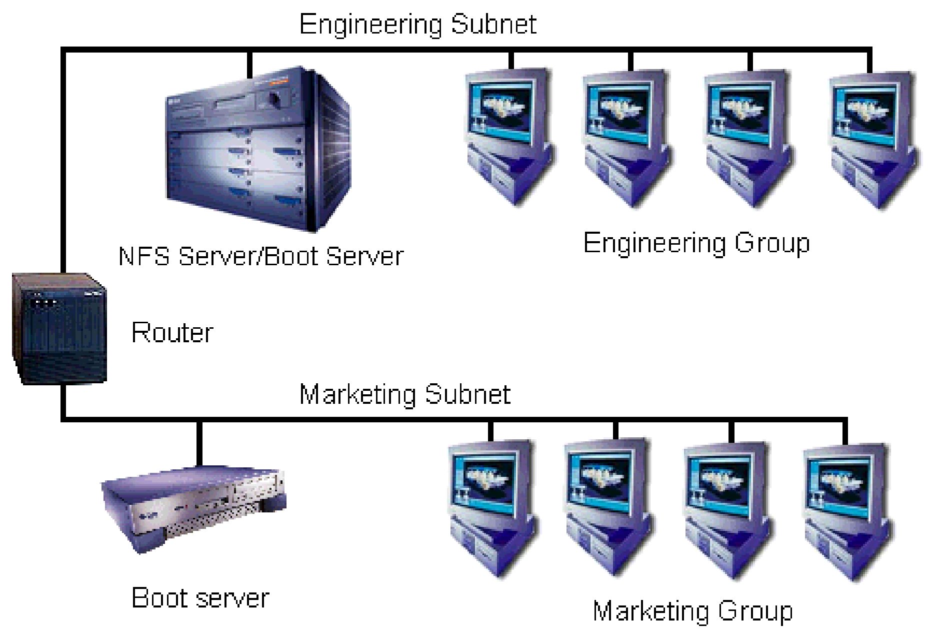
Once the `inetboot` file has been transferred, the client executes it. The first thing that the `inetboot` file does is to send out a bootparams "whoami" request. bootparams is a protocol that allows the transfer of key:value pairs from a server to the client. The client sends out a request for the value associated with a key and the server responds with that value. The bootparams server under Solaris is called `rpc.bootparamd`. The server listens for bootparams requests and when it receives one it tries to lookup the hostname corresponding to the IP address in the request. If that succeeds then the server tries to look for an entry matching that hostname in either `/etc/bootparams` or the `bootparams` NIS map (depending on the `bootparams` entry in `/etc/nsswitch.conf`). If that succeeds then the server sends a response packet back to the client with the value for the key that the client requested. A "whoami" request is special, however. The bootparams server responds to a "whoami" request with the client's hostname, NIS/NIS+ domainname and default router. The client next sends out an ICMP address mask request. Using the netmask from the ICMP request and the default router from the bootparams request, the client finishes configuring its network interface and routing.

The client then makes a bootparams request for the `root` key. The server returns the NFS path for the client's root directory and the client performs an NFS mount of that directory. At this point, the client loads the regular Solaris kernel. The JumpStart root directory and startup scripts are different from the standard Solaris configuration so the rest of this process differs from a normal diskless boot process. The startup scripts repeat all of the network configuration steps that the `inetboot` boot block performed. They then NFS mount the install directory, which is a copy of the Solaris media, at `/cdrom`. The path to the install directory is found via a bootparams request for the `install` key.

The client then NFS mounts the profile directory, found via bootparams using the `install_config` key. This directory contains the rules and profiles as well as the begin and finish scripts for a Custom JumpStart configuration. At this point the JumpStart scripts collect a few bits of information, like the local timezone and locale, from NIS or NIS+. Starting with Solaris 2.6, this information can also be configured through a local file. The rules file is then searched for a set of rules that matches the client's hardware. If a matching rule is found then a corresponding begin script, profile and finish script are selected. The begin script, which is run before anything on disk is changed, is typically used to save things like local calendar files or crontabs. The profile dictates how the disks in the client are partitioned and which Sun package cluster is installed. The clusters range from the Core cluster, which is the minimum software needed to boot Solaris, up to the Entire Distribution cluster. Typically the End User or Developer clusters are selected, depending on whether support for compiling programs will be needed. The packages associated with the selected cluster are installed once disk partitioning is completed. Then Maintenance Update patches are installed if available. Maintenance Update patches are what differentiate an FCS (First Customer Ship, i.e. the first release shipped to customers) release of Solaris from a later dated release of the same version. Once the Maintenance Update patches are installed, the finish script is executed. This is typically where site-specific customizations would be located. One very common finish script task is to install the latest Recommended Patch Cluster from Sun. Once the finish script exits, the client reboots and the JumpStart is complete.

The network booting and JumpStart processes require some network and system infrastructure in order to work. The network boot process primarily depends on a "boot" server. The clients need a server on their subnet to make requests to until they possess enough information to properly handle routing. As such, a boot server must be connected to each subnet. A boot server would run RARP, bootparams and TFTP servers as well as being a NIS slave in a NIS environment. It is possible to configure JumpStart to handle an environment without NIS+ replicas on each subnet. The other important piece of infrastructure is the NFS server. We will discuss how to size NFS servers for JumpStart later in this paper.

Figure 1 shows an example JumpStart environment. The NFS server, which serves as a boot server as well, is located on the engineering subnet. Marketing is on their own subnet so they need a separate boot server. They will then use the same NFS server as the engineering group.



The Old Way

When I first started working with JumpStart, the company I was working at had two employees assigned to performing Solaris updates on deployed workstations. The process at that time was manual and significantly error-prone. JumpStart configuration was done using the `add_install_client` script provided by Sun. On days when a JumpStart was scheduled, the two operators would attempt 10-20 JumpStarts during the lunch hour. On average, the JumpStarts failed on 30% of the machines, usually due to errors in the NIS maps or hardware that didn't match any of the JumpStart rules.

Given the high failure rate, the process generated much distrust and resentment in the user community. Managers knew that this was going to cost them downtime and did their best to avoid having their users' machines upgraded. The process was costing the company a lot of money. After the second day of watching this, I realized several things:

1. People make a large number of typos. Booting a machine off of the network requires a number of files and NIS/NIS+ maps to be configured exactly right. A single error will cause the entire JumpStart to fail in various, difficult to diagnose, ways.
2. Users are greatly annoyed when they come back from lunch and their machines are not functional. The 30% failure rate resulted in this happening to a number of users. Each failure typically took at least a few minutes to diagnose. Since the JumpStart itself took at least an hour, if anything went wrong the user's machine was down during part of their normal working hours. This frustrated the users and cost the company money.
3. The whole process was mind-numbingly slow. There was a lot of repetitive typing that went into configuring each machine using the `add_install_client` script. Additionally, there was quite a bit of manual error checking to do because `add_install_client` does only limited error checking. Further, visiting each machine to start the JumpStart took a long time because they were frequently spread out over a large area.
4. The mix of NIS maps and local name service files created a great deal of confusion and problems. `add_install_client` checks the appropriate NIS map for entries but if it doesn't find one then it

adds the entry to the local name service file on the boot server. This meant that the operator had to check both NIS and the local files when searching for missing or incorrect entries.

5. The process didn't get the OS upgrades to users in a timely fashion. When I started working with the two employees, they were just finishing up a rollout of an updated version of the JumpStart image. It had taken them close to 6 months to do this for the roughly 600 standard Sun workstations in use. If a serious bug had been found in the image, it would have taken another 6 months to roll out a new version. The process in use clearly was not scaling to the number of machines to which it was being applied.

So, I set about to devise a better mousetrap. My goals for the new process:

1. The process for configuring the JumpStart boot process should be quick and require minimal typing.
2. Far more complete error checking should be done in order to reduce or eliminate JumpStart failures.
3. The scripts developed should fully support the use of NIS or NIS+ and not create entries in local name service files.
4. The process should scale to hundreds or even thousands of machines.
5. A reliable method for kicking off the JumpStarts without physically touching the machine should be developed.

The New Way

Over the course of the next several rollouts at this company, I was able to develop scripts and processes that allowed me to accomplish all of my goals. At the time I left we were able to JumpStart 200 machines in an hour. A complete rollout of a new image to all 600-650 machines could be completed in about 10 hours over the course of several evenings. This was a great improvement over the 6 months it used to take to perform a rollout.

Config

The first thing I did was to develop a replacement for `add_install_client`. I called my script `config`. Given the complexity of what I wanted to do, I didn't think the Bourne shell was the right tool for the job. So, I decided to write `config` in Perl.

The biggest advantage of `config` over `add_install_client` is that it checks for almost every problem that could possibly prevent a JumpStart from working. This includes errors that prevent the machine from correctly booting via the network and errors that would cause the Custom JumpStart to fail.

There are several things that can go wrong with the first step, the RARP request. First, the client's ethernet address must be correctly mapped to its hostname in the `ethers` map. There should only be one entry for each ethernet address. Each hostname needs to be in the `hosts` map correctly mapped to the IP address assigned to that hostname. Again, there should only be one entry for each hostname/IP address pair. Also, the boot server needs to be running a RARP server and no other host on that subnet should be running another RARP server. The reason for this is that multiple replies tend to confuse the client. Also, if there is more than one RARP server on a subnet, the other servers are typically ones that we don't control. People have a bad habit of leaving RARP servers running with incorrect or outdated information.

The client then tries to TFTP the `inetboot` file. For this to happen, the boot server must be configured to run a TFTP server and there needs to be a boot block in `/tftpboot` that matches the IP address of the client in hexadecimal. The way this is usually done is that one copy of the `inetboot` file, which is specific to each version of Solaris and each kernel architecture (sun4m, sun4u, etc.), is put into `/tftpboot` on the boot server and then symlinks for each client are made to the appropriate `inetboot` file. The TFTP server under Solaris is run by `inetd` when a TFTP request is received; so, the `tftpd` entry in `/etc/inetd.conf` must be uncommented. `inetd` must be sent a `HUP` signal if a change is made to `inetd.conf`. Otherwise, `inetd` won't re-read the configuration file. Here, again, we only want one server per subnet. If the machine running the RARP server doesn't respond quickly enough to the client then the client will broadcast any further requests. That means that if there is more than one TFTP server on that subnet with the appropriately named file, they will all respond. As with RARP, multiple TFTP replies tend to confuse the client and it will usually hang. Also, we want to avoid rogue servers with incorrect or outdated `inetboot` files.

The client then sends out a `bootparams` request. This, again, requires that the client's IP address be correctly associated with its hostname in the `hosts` map. Also, there must be an entry matching that hostname in the `bootparams` map. The boot server must be running a bootparams server (`rpc.bootparamd` in Solaris). Also, like with RARP and TFTP servers, there should only be one bootparams server on the subnet. Multiple replies don't cause as many problems as they do with RARP or TFTP; but, again, we want to avoid rogue servers with incorrect or outdated information.

The client then performs an NFS mount of its root directory. This requires that the root directory be located on the fileserver and that the directory is exported with the options of `ro,anon=0`. Exporting the directory read-only isn't required but it is a very good idea. (At various times, there have been bugs in Sun's JumpStart scripts such that things will break if the root directory is exported read-write. This provides even more reason to export it read-only.)

The client then performs NFS mounts of the CDROM image and the profile directory. This requires that the fileserver exports the CDROM image and the profile directories, with the same options as the root directory.

In an NIS environment, the client will then broadcast a request for an NIS server. This is the default behavior in an NIS+ environment as well, but it is possible to add an extra field to the client's bootparams entry to point it at an NIS+ server on another subnet. Thus, in an NIS environment it is necessary that there be an NIS server on the same subnet as the client.

The last thing that can go wrong is with the JumpStart itself. One of the first things that happens during a JumpStart is that the machine checks its hardware against what is called a `rules` file. The `rules` file defines various hardware configurations and what profiles go with them. The profiles dictate how the disks in the client are to be partitioned, as well as the Sun package cluster to be installed. In order for this to succeed, the machine's hardware must match at least one of the configurations in the rules file.

`config` requires `root` access to the NIS master in a NIS environment. This is most easily accomplished by running `config` as `root` on the NIS master, and that is what we did. However, site configuration and security policies might dictate using some other method. This access allows `config` to automatically make NIS map corrections and push the updated maps. Otherwise, the operator is responsible for making the map changes manually. Removing dependencies on the operator typing information correctly is one of the main reasons for developing `config`. `config` is very careful about how it makes changes and we never experienced any corrupted NIS maps due to its operation.

`add_install_client` does not have a built-in option for bulk configuration. Automating bulk configuration with `add_install_client` is difficult because it requires knowledge of the client kernel architecture for each machine. With `config`, I built in support for bulk configuration. The script takes a list of hostnames and attempts to configure all of the machines on the list. `config` queries each machine for the information it needs, including the kernel architecture. This requires root login access on all of the clients. The operator then need only create the list of hostnames.

In combination with the scripts mentioned below, `config` tracks the status of each machine. This is especially useful when using the bulk configuration feature described above. The operator compiles a list of several hundred hostnames and runs them through `config`. The script will print a report at the end of its run listing which machines failed and the reason for the failure. The operator can then fix the problems and rerun `config` with the same list of hostnames. `config` knows which machines are already configured and skips over them. This saves hours of waiting when working with a large number of machines. This status tracking is accomplished through the use of a text file listing each machine and a numeric code indicating what state the machine is in.

Another advantage of `config` is that it automatically picks the boot and NFS servers for a machine when configuring it. This allows the operator to be blissfully ignorant of the network topology and the various servers. With `add_install_client`, the operator has to know which boot server is on the same subnet as the machine. While usually easy to figure out, it is one more mistake that can be made. The logic that `config` currently uses to pick the NFS server is simplistic, but could be improved to use `traceroute` to truly find the closest machine. Currently `config` looks for an NFS server on the same subnet as the client, if it doesn't find one then it picks a default server. This works well if most clients are on a subnet with an NFS server but would fail in an environment where this isn't the case.

`config` creates a centralized place to make JumpStart configurations. With `add_install_client` in a large environment, the operator must know which boot server to use for each machine and log into that server to perform JumpStart configuration. With `config`, the operator always logs into the same server and doesn't need to know which boot server to use. This also prevents uninformed operators from setting up more than one boot server on a subnet.

With `config`, a method for users and administrators to tag machines as non-standard is introduced. Machines are tagged as non-standard by creating `/usr/local/do_not_jumpstart`. These were usually machines that had been standard desktop clients but then additional software or other modifications were made to the machine. We didn't want to JumpStart these machines because that software would have been lost. Tagged machines are automatically skipped over, eliminating the need for a failure-prone, manual method of keeping track of these machines.

As you can see there are many advantages to replacing `add_install_client` with a more capable script. Hopefully Sun will realize this and improve `add_install_client`.

Infrastructure

The network and server infrastructure becomes important when JumpStarting many machines at once.

When installing 10 machines at a time, the CD image can be kept on a single drive on a Sparc 2 at the end of a shared 10 Mbit Ethernet connection. However, scaling up to JumpStarting several hundred machines at a time requires more bandwidth.

The first consideration is boot servers. The clients need a server on their subnet to make requests to until they possess enough information to properly handle routing. As such, a "boot" server must be connected to each subnet. Almost any sort of machine will work as a boot server since the total amount of data transferred from the boot server to each client is approximately 150 KB. Note that `add_install_client` requires the JumpStart root image to be located on each boot server. `config` does not require this and thus the bandwidth is limited to the `inetboot` file, which is approximately 150 KB. If you need or choose to be compatible with `add_install_client` then you will need to take into account the additional disk space and bandwidth requirements of the root image. This image is approximately 30 MB on Solaris 2.5.1 and earlier, it jumped to approximately 150 MB in Solaris 2.6 and later versions. If your network design already contains a server on each subnet with Sun clients then boot servers are not a concern. If not, then you will need to put something together. We used several Sparc 5s with quad Ethernet cards. Doing so allowed a single machine to serve up to 5 subnets (including the built-in interface in the Sparc 5).

There is one very subtle trick to using multi-homed boot servers. I mentioned earlier that the client gets its default router via a bootparams "whoami" request. The boot server, like most machines, would normally be configured with only one default router as that is all it typically needs. This default router is what the boot server would then send in response to a "whoami" request. The problem is that on a multi-homed boot server, that default router will be correct only for clients on the same subnet as that default router. The solution is to put multiple entries in `/etc/defaultrouter`, one for the default router on each subnet that the boot server is attached to. Then, when the boot server boots, it will create multiple default route entries in its routing table and will return the correct one, based on the subnet, in response to "whoami" requests. It is also generally a good idea to touch `/etc/notrouter` on multi-homed boot servers so that they don’t route packets between the various interfaces.

The next consideration is network bandwidth. As a rough estimate, each workstation will transfer about 500 MB in the course of the JumpStart. The client’s network connection is of little concern as 10 Mbit Ethernet can easily handle that much data in an hour. However, the bandwidth of the NFS servers’ network connections must be considered. It is easy to figure out the required network bandwidth. As an example, say you want to JumpStart 200 machines in 1 hour. 200 times 500 MB divided by 1 hour is 100 GB/hr. Network bandwidth is usually measured in Mb/sec. By a simple conversion (1 GB/hr =~ 2.2 Mb/sec) we see the required bandwidth is about 220 Mb/sec. Switched 100-Base-T is fairly common these days and, as the name implies, runs at 100 Mb/sec. If you can spread the load evenly over three switched 100-Base-T interfaces or three servers with 100-Base-T interfaces then there will be sufficient bandwidth. Keep in mind that the network traffic of JumpStarting many machines will cause a high collision rate on shared Ethernet, so avoid it if possible. Server interfaces on shared Ethernet will only produce about 10% (at best) of the rated bandwidth during JumpStarts.

The last issue is NFS server capacity. The most common measure of server capacity for NFS servers is the SPEC SFS benchmark. SFS93 was also known as LADDIS, the current version of SFS is SFS97. The results of the test are expressed as the number of NFS operations (NFSops) per second the server can handle. Results from SFS93 and SFS97 are not comparable. Brian Wong, of Sun Microsystems, estimates that each client requires 50-70 NFSops/sec (as measured by SFS93) during a JumpStart [Won97]. This matches very closely with the performance we observed as well. SPEC provides SFS93 and SFS97 benchmark results for a wide variety of NFS servers on their website at http://www.specbench.org/osg/sfs97/. Sun also provides results from the older LADDIS benchmark on their website. For example, a Sun Enterprise 3500 with four CPUs can handle approximately 8900 SFS93 NFSops/sec according to Sun. At 60 NFSops/sec/client, that server could serve approximately 150 clients. To achieve this level of performance, the JumpStart data should be striped over a number of disks using RAID 0 or 5.

Start

The last step was to develop a script to start the JumpStart process on a large number of systems simultaneously. The simplest form of such a script would loop through and `rsh` to each machine in order to execute `reboot net - install`. Unfortunately, it takes approximately 15 seconds to make the connection and initiate the reboot for each machine. When attempting to start 250 machines at the same time, it will take more than an hour just to get all of the machines started. Additionally, a few things should be checked on the machine first, such as whether there are any users logged on. So now, the naive implementation of a kickoff script will take a couple of hours just to get all of the machines started. The `start` script I developed forks into a number of processes, each of which is assigned a group of hosts to work on. This allows the operator to get all of the machines started within a few minutes of each other. `start` also references the status file updated by `config` to ensure that it only starts machines that were successfully configured. The script checks to see if any users are logged in. It also checks to see if the machine has been tagged as non-standard. Although `config` checks for this as well, it is possible that someone might tag the machine in the day or two after `config` is run but before the rollout is performed. `start` also checks to see if the `root` password is the standard `root` password. The clients will typically all have a common `root` password, but sometimes when someone customizes a machine he will change the `root` password. This was put in as a last-ditch effort to catch non-standard machines even if they had not been tagged as non-standard.

Once `start` has started the install on a machine, it continues to attempt to `rsh` in periodically and check on the progress of the JumpStart. This progress is recorded in the status file. This requires a minor change to `/sbin/sysconfig` in the JumpStart root image on the NFS servers to start `inetd`, and thus allow `rshd` to start, during the JumpStart.

During a rollout we would have several people on hand to deal with any problems that arose during the JumpStart. To help everyone spot problems as quickly as possible, I wrote a CGI script that references the same status file used by `config` and `start`. The script generates web pages with the status of all of the machines indicated by red, yellow or green dots and a status message. The pages automatically refresh in order to display up-to-date information. As machines encounter problems, they are flagged on the web page with yellow or red dots depending on the severity of the problem. The people on hand then read the status field and determine if the problem needs intervention. Most common were users who remained logged in during the scheduled rollout. In those cases, someone would call the user or visit the machine and ask the user to log out. Other problems were similarly handled in order to keep the rollout moving. This allowed the client to scale up to performing several hundred simultaneous JumpStarts. We had several people on hand to deal with the rare problems that cropped up. They would watch the web pages for troubled machines, figure out what the problem was and attempt to fix it. Our record was just over 250 machines in an hour and a half.

QuickJump

As mentioned previously, a standard Custom JumpStart consists of a begin script, disk partitioning, Solaris package installation, possible Maintenance Update patch installation and then typically additional patch installation during the finish script. Package and patch installation on older machines, like a Sparc 5, can be quite slow. We found that installing the Recommended Patch Cluster for Solaris 2.5.1 would take 8 hours or longer on a Sparc 5. This in turn makes the JumpStart take a long time. Mike Diaz of Sun Professional Services developed a system for this client we called QuickJump to speed up the JumpStart process. Rodney Rutherford and Kurt Hayes of Collective Technologies further refined this process.

QuickJump consists of performing a standard Custom JumpStart once on a machine in the test lab. Once the JumpStart completes, each filesystem on the client is dumped using `ufsdump` to the NFS file server. Then we create a separate Custom JumpStart configuration for use in the rollout. In that configuration we select the Core cluster of Solaris packages, which is quite small, in the JumpStart profile. We also disable Maintenance Update patches if the Solaris version in use has them [SI15834]. Then we modify the finish script in the JumpStart configuration to restore the `ufsdump` images onto the client. This overwrites everything on disk, namely the Core cluster of Solaris packages. For even greater speed, it is possible to disable the installation of a package cluster altogether, however this requires somewhat extensive modifications to the Sun JumpStart scripts. The Core cluster takes only a few minutes to install so we decided that it wasn't worth the effort to make those modifications. The finish script then merely has to change the hostname and IP address as appropriate and it is done. This vastly speeds up the JumpStart process on older hardware, bringing it down to 45 minutes or so for a Sparc 5. The difference becomes negligible with machines based on the UltraSPARC[tm] processor, so QuickJump may not be necessary once the older hardware is replaced.

The Future

It has now been a year since I worked on the project. Although the client is still successfully using these scripts and processes largely unchanged, I've thought of some ways to improve the scripts and the processes. Some of these suggestions are minor. Using multicast, however, would change the process significantly.

The main difficulty we had with the scripts I wrote was the status file that the various scripts use to keep track of the status of all the machines. Similar to the problems that large webservers encounter with their log files, this file turned out to be a bottleneck in the middle of JumpStarting a large number of machines. The multiple forks of the `start` script were in contention with each other in trying to update the status file. In the future, I will split the status file so that there is a separate file for each machine. This will eliminate the contention problems within `start`. It will also allow multiple operators to configure machines at the same time and not have to wait for access to the monolithic status file.

Another change that will improve the system is to split the `start` script into a true start script and a script that keeps track of the status of the JumpStart on each machine. These two scripts will run simultaneously during a rollout. On the other hand, helpdesk personnel or an operator could use just the basic `start` script to remotely JumpStart a single machine. This was done at the client after I left and made the rollouts proceed more smoothly.

The current method of tracking the progress of the JumpStart, using `rsh` to log into the machine and attempt to figure out what is going on, is somewhat error-prone. A nice addition would be to be able to view a real-time version of the log that is displayed on the client's screen while the JumpStart is progressing. This would allow the operator to see exactly what is going on and any errors that are displayed, without having to visit the machine.

Earlier I mentioned that we check for rogue bootparams servers but not rogue RARP or TFTP servers. Checking for rogue bootparams servers is simple because the `rpcinfo` program provides a feature that makes it easy to check for all bootparams servers on a subnet. Given an RPC service number and the `-b` flag, `rpcinfo` will broadcast a request on the subnet for that service and report all of the servers that respond. Any servers other than our designated boot server can be flagged as needing to be disabled. Unfortunately there isn't a similar program to check for rogue RARP or TFTP servers. Eventually I will write programs to do this.

In the long term I would like to investigate the possibility of using multicasting to distribute the disk images to the workstations. Each machine of a given kernel architecture gets an identical image, thus this is a prime candidate for multicasting. This would drastically reduce the NFS server bandwidth and capacity requirements as the NFS server would only need to serve out a couple of copies of the disk images (one for each kernel architecture) independent of the number of clients. This technology will be one to watch and keep in mind as it develops.

Making It Work For You

Hopefully by now I've interested some of you in improving your own JumpStart environments. I encourage you to take what I've done, modify it to fit the specifics of your site and then improve on things. I would welcome feedback on what worked, what didn't work and what you've done differently. Here are the major steps to implementing a system such as what I've described.

First, you'll want to build up your infrastructure. This includes placing boot servers on each subnet and ensuring sufficient NFS server capacity and bandwidth. If your clients are spread over multiple subnets, it would be preferable to have an NFS server on each subnet. This reduces the amount of traffic that needs to cross a router. If your NFS servers have sufficient capacity, they can be attached to multiple subnets using several network interfaces.

The other infrastructure consideration is on the client end. Client workstations need to have standardized hardware. In addition, your environment should be designed in such a way that no data is stored on the clients. This is usually easy to accomplish and it makes the JumpStart process much simpler.

I would also highly recommend that you use some sort of distributed name service. The two most common are NIS and NIS+. The reason for this is that it ensures uniformity across all machines and provides a centralized location for making changes to the name service maps. This makes it easier and more reliable for the `config` script to check the accuracy of the name service maps and correct any errors. My scripts were written to support NIS but it should be fairly easy to convert them to an NIS+ environment.

The server that hosts the `config` and `start` scripts needs to have remote `root` login privileges on the clients, the boot servers and the NIS master (if applicable). This can be accomplished very insecurely using `rsh` and `/.rhosts` files or more securely using Secure Shell (SSH) and RSA keys. Login privileges are needed on the clients in order to gather information like kernel architecture, Ethernet address, etc. `root` privileges are needed to gather some of the required information. `root` login privileges are needed on the boot servers in order to make modifications to the links in `/tftpboot` and to start the RARP, bootparams and TFTP servers if necessary. `root` privileges are needed on the NIS master to edit the NIS maps and initiate map pushes.

Once all of these items are in place then you can work on integrating my scripts into your environment. As I mentioned previously, the scripts are written in Perl. Roughly 7000 lines of it, in fact. So you'll need to have Perl 5 available or lots of time to port the scripts to some other language. The scripts may be had from my web page at http://ofb.net/~jheiss/js/ I have released them under the GNU Public License.

The last recommendation I have is that you set up a test lab to test everything in. There are a lot of pieces to getting this all working and attempting a rollout without testing would be, well, crazy. I would recommend an isolated subnet with at least one machine of each model that exists in your environment. Thus if you have Sparc 5s, Sparc 20s and Ultra 1s as desktop machines you would have at least one of each in your test lab. This is especially important if you plan on using something like QuickJump as it is tricky to get the device trees right on each platform after restoring the dump files.

References

[Kas95] Kasper, Paul Anthony and Alan L. McClellan. *Automating Solaris Installations*. Prentice Hall, April 1995. ISBN: 013312505X.

> This book provides detailed instructions on how to configure a Custom JumpStart. It is essential reading and reference for administrators new to configuring JumpStart. However, it does not discuss how to perform a rollout, especially on a large scale.

[Zub99] Zuberi, Asim. "Jumpstart in a Nutshell." *Inside Solaris* February 1999, pp 7-10.

> A quick article with the steps to configure and use JumpStart with Solaris 2.6. As of Solaris 2.6, Sun rearranged the Solaris CD somewhat and this article shows the new paths you'll need to know. This makes a good companion to the book when working with newer versions of Solaris.

[SAIG] Solaris Advanced Installation Guide

> The official Sun documentation for JumpStart. Very good information and example scripts for configuring Custom JumpStart but again, it does not discuss how to perform a rollout.

[Won97] Wong, Brian L. *Configuration and Capacity Planning for Solaris Servers*. Prentice Hall, February 1997. ISBN: 0133499529.

> This book provides excellent, detailed information on capacity planning for Solaris servers. The section relevant to sizing NFS servers for JumpStart begins on page 33.

[SI15834] Sun Infodoc 15834

> Instructions for disabling the installation of Maintenance Update patches.

About the author

Jason Heiss graduated from the California Institute of Technology in 1997 with a BS in Biology. He works for Collective Technologies as a systems management consultant, specializing in Solaris and Linux system administration, security and networking. He can be reached via email at jheiss@colltech.com. He can be reached via U.S. Mail at Collective Technologies; 9433 Bee Caves Road; Building III, Suite 100; Austin, TX 78733.