Enterprise Rollouts with JumpStart^{TM}

Jason Heiss
Collective Technologies
jheiss@colltech.com

Abstract

JumpStart 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, 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. 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 important updates, 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

I want to quickly mention the problems with interactive installs. First, they are very labor intensive.

While good for job security, this is expensive and tiresome. It also makes it difficult to get timely updates 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.

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. The two operators would attempt 10-20 JumpStarts a day during the lunch hour. On average, the JumpStarts failed on 30% of the machines.

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 JumpStarted. 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 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. And then 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 process didn't get the OS updates 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.

So, I set about to devise a better mousetrap.

The New Way

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.

`config`'s biggest advantage 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. Specifically, `config` checks and corrects the appropriate NIS maps and the `inetboot` boot block in `/tftpboot` on the appropriate boot server. It checks to make sure the RARP, bootparams and TFTP servers are enabled or running on the appropriate boot server. `config` checks for rogue bootparams servers and will eventually check for rogue RARP servers. Also, `config` ensures that the CD-ROM image is available and exported on the appropriate NFS server. Lastly, the script checks the hardware of the target machine to make sure it matches one of the hardware rules for the custom JumpStart. (Note to reader: This will all be filled in with more detail in the final paper. In fact, most of it is already written and just needs to be pasted in here.)

`config` requires root access to the NIS master. This allows it to automatically make NIS map corrections and push the updated maps. Otherwise the operator is responsible for making the map changes. Removing dependencies on the operator typing information correctly is one of the main reasons for developing `config`.

`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 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 architecture. The operator 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.

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 may be made.

`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 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. 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 JumpStarting 10 machines at a time, the CD image can live 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 protocols that Sun uses to boot via the network are not routable. 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 for each machine is approximately 150KB. 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.

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-Base-T 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. (Note to reader: I have an example calculation that will be inserted here in the final paper.) Keep in mind that the network traffic of JumpStarting many machines will overload shared ethernet; so avoid it if possible. Server interfaces on shared ethernet will only produce about 10% of the rated bandwidth during JumpStarts.

The last issue, and possibly the hardest to judge, is disk bandwidth. While the bandwidth of the various forms of the SCSI bus are well known, it is more difficult to find the realistic bandwidth of individual drives or RAID volumes. We found that a single drive in a SPARCstation could keep up with a 10-Base-T connection and that a 4 disk RAID 0 stripe in an Auspex fileserver was able to keep up with three FDDI (100 Mbit) connections. Note that the Auspex caches much more aggressively than a regular Sun fileserver and JumpStarts are perfect for caching. (Note to reader: I'll research this a little more and get some real numbers.)

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 will take a couple of hours to get all of the machines started. The `start` script I developed forks into a number of processes that are each 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.

My `start` script also generates web pages with the status of all of the machines indicated by red, yellow or green dots and a status message. The script runs in a loop and checks on the status of the machines periodically as the JumpStarts progress. This allowed the company previously mentioned 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.

The Future

It has now been a year since I worked on the project. In the interim I've thought of some ways to improve the scripts and the process in general. 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 used 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 company after I left and made the rollouts proceed more smoothly.

Earlier I mentioned that we check for rogue bootparams servers but not rogue RARP servers. This is because the `rpcinfo` program provides a feature that makes it easy to check for bootparams servers on a subnet. If 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 servers. Eventually, I will write a program to do this.

The greatest improvement that I have planned is to switch to multicasting to distribute the disk images to the workstations. Each machine of a given architecture gets an identical image, thus this is a prime candidate for multicasting. (Note to reader: This will be one of my major research areas in preparing the final paper. I hope to include detailed instructions on using multicasting for this purpose or details on why it is not currently possible.)

References

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

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