ORACLE FIELD SESSION

NETWORK BOOTABLE TEXT INSTALLER

Carl Blum
David Danford
Andrew Gracey
Chris Navrides

Abstract

For our field session project, we built a network bootable Text Installer (TI) for OpenSolaris. The TI is one method for installing the OpenSolaris operating system (OS) that involves multiple menu options and will end up only installing a non-graphical version of the OS. Last year, the text installer was developed by another CSM field session group.

Our project entailed adding the ability to boot the text based installer from a server, instead of a CD. This new capability makes it possible to install OpenSolaris relatively quickly onto multiple servers supporting a user's network. To do this using a CD would take a long time, and would require placing a CD in each machine, which would limit how fast a human can actively control multiple installs at once.

The network bootable text installer now allows the user to run the installs in parallel using only one CD or ISO file on the server. The difference between this installer and the Automated Installer (AI) is that our product is intended for use on servers, and other machines that do not have the capability of running GNOME.

We modified the existing AI net-boot scripts to determine if the current install was a text install. It checks this after it downloads the necessary files from the server. It then sets links, which make it appear to the TI that the files are located on the cdrom instead of in the hard disk. We also modified the GRUB menu creation scripts so that it would create the proper menu items automatically.

Introduction

Until its recent acquisition of Sun Microsystems, Oracle developed enterprise software. Now Oracle can deliver complete server stacks to satisfy business needs. Among the projects acquired with Sun Microsystems is the OpenSolaris project, and all of its teams. OpenSolaris is an open source unix operating system that is currently maintained by Oracle under the CDDL licence. This field session project required developing a solution for the install team.

Currently, three seperate technologies exist for installing OpenSolaris. The LiveCD allows the user to boot from CD into OpenSolaris, and install from the disc. The AI allows both network bootable and CD bootable options and installs a graphical version of OpenSolaris. The final technology is the text installer, which allows the user to modify the install options for OpenSolaris and installs a console based environment.

The text installer was prototyped by last year's field session group in C, and then converted by the Sun Microsystems install team to use python. Currently, the AI is implemented in C, though a long term goal of the install team is to convert all of the install technologies to python.

The install technologies currently support both the x86 architecture and the SPARC architecture. This project primarily focused on the x86 architecture, and if time had permitted we would have converted the installer to work on the SPARC architecture as well.

The main goal of this project was to allow a network boot of the text installer. If a large number of computers need to be imaged, but the graphics card does not support GNOME graphics, the network bootable text installer now allows the process to be completed in much less time using only one image, rather than requiring a CD for each computer. Further, the text installer could allow multiple remote machines to install slightly different package sets if a prompt were added to the new text installer, allowing the selection of packages to be added.

In the beginning we believed two distinct methods existed that could solve the primary goal of this project. The existing network interface provided by the AI could be extended to use the Text Installer's interface. We thought this solution would likely have required coding in C, rather than in python or bash. This approach would have require much less coding as the network functionality already exists, but would have require more research of surrounding technologies.

The other strategy we conceived was to extend the Text Installer to support both the media install and a network install, mirroring the current set up of the AI. This solution would have required many small changes in the existing files, and implementing the network functionality in python.

We discovered quickly that the project would require a method that combines the two methods. We took the existing netboot functionality from the AI, and pointed it to the text installer's interface. We then needed to modify the locations that the AI placed files to make it appear to the TI that the computer was booting from CD or USB.

Another requirement was to make the setup of the hosting server work properly and have the correct menu. The menu creation is what the client will see and needed to work if they had just an AI image or they had one that allowed both AI and TI. To accomplish this we added an extra hidden file when we created the image so that if it is there, the server menu creation will add the text-install option, otherwise it will just act like it normally does.

Finally, the Install Completion Task (ICT) was modified to add the newly installed boot sector to the GRUB menu. The AI already does this; however, it uses flags that would cause the Text Installer to fail. Specifically, the AI tried to load GNOME, which is unavailable in the text environment. If this wasn't changed, the boot would freeze on the splash screen for loading the GNOME login prompts.

Requirements

I. Functional requirements

The primary objective was to allow the text installer to boot from a network connection. This functionality now uses the previously existing functionality of the AI's network boot. When booted from a network connection, the user is now directed to a grub menu to select which bootable image to use. An option for the text installer has been added here.

When selected, the text installer loads the necessary packages from a server on to the computer to be installed. The programming was entirely in the scripting languages python and bash. Our new implementation of the text installer works for the x86 architecture, but we did not have time to expand it to the SPARC architecture.

II. Non-Functional requirements

The system should be connected to a network that also contains a DHCP server, a TFTP server, and an IPS server. The DHCP server is necessary to direct the network boot option to where it can find files. The TFTP server provides the text installer to the system. Finally, the IPS server provides all the packages requested by the text installer for installation on the system.

III. Use Case

User wants to install OpenSolaris on a computer

Primary Flow

1. User connects the computer they wish to install OpenSolaris on to a network.

2. User turns on the computer.
3. User presses F12 to pull up the GRUB boot order menu
4. User selects the option to boot from the network
5. Network sends the available options to the user
6. User selects the Text Installer option
7. Computer pulls the necessary install material and begins the text install process
8. User goes through the text install process, selecting choices based on need such as language, time zones, and partition size.
9. With these prompts, the computer goes through the rest of the text install process automatically
10. The computer is now loaded with OpenSolaris

Alternate Flow

3a. User does nothing, computer tries to boot following the default boot order

4a. User can select other locations to attempt to boot from, such as hard drive or media.

6a. User can select other network install options, such as Automated Installer

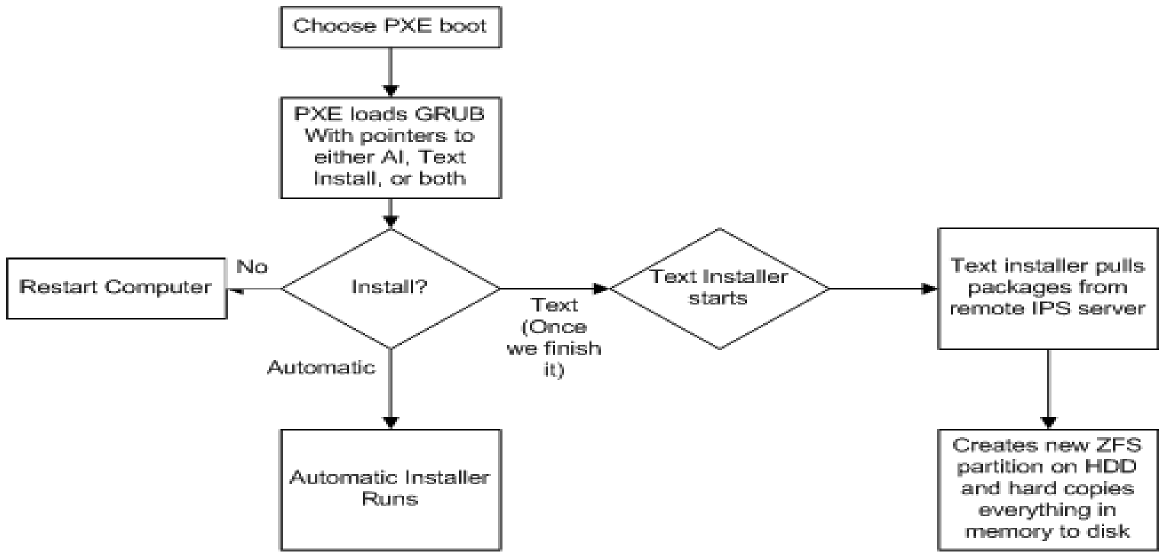
Previous System

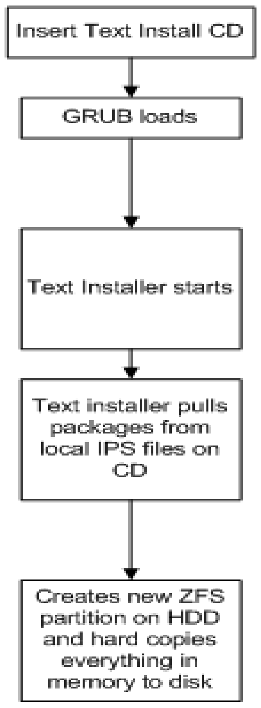
The new text installer had to interface with several systems. Two of which are owned by Oracle and the rest are supporting technologies that we had to learn about to allow us to complete our project. The Automated Installer (AI) and the Text Installer (TI) are two of the methods used to install OpenSolaris.

The AI requires very little user interaction and allows for a PXE netboot install along with a CD based install. The Text installer was a previous field session project and uses python to get user input on a variety of things including date and partition info. We had to learn a lot about both of these in order to finish our project.

We also needed to learn about how to get variables that are passed from the GRUB menu. Grub allows you to pass arguments from this menu to the operating system. This allowed us to specify what type of boot we are working with. GRUB is the standard boot loader for most Linux and Unix systems. There are others but they are not as common, and OpenSolaris currently uses GRUB.

We also needed to learn about the ZFS filesystem and how to work within it along with using the IPS package manager. ZFS allowed us to make copies of our partitions without actually copying the bits by keeping up with differences between the two copies. The IPS packaging system will be important since we will have to be pulling our packages from a known server as opposed to the CD which is what the Text Installer currently does. The figure below shows this system.


Choose PXE boot
PXE loads GRUB With pointers to either AI, Text Install, or both
Install?
No
Restart Computer
Automatic
Automatic Installer Runs
Text (Once we finish it)
Text Installer starts
Text installer pulls packages from remote IPS server
Creates new ZFS partition on HDD and hard copies everything in memory to disk


Insert Text Install CD
GRUB loads
Text Installer starts
Text installer pulls packages from local IPS files on CD
Creates new ZFS partition on HDD and hard copies everything in memory to disk

Risks

This project posed many risks. The main risk was choosing the wrong path and not realizing it until it was too late. There was also the technical risk that the hardware we were given would be too complicated. There was a slight risk that learning python would take too much of the project's time.

As previously mentioned, the main risk was choosing the wrong path. With this project we thought we had two possibilities; one was to modify the AI code to call the python Text Installer scripts and the second was to modify the TI code to allow for network boot ability.

We had initially chosen to modify the Text Installer to add the network boot; however, we determined that using the existing network capability of the AI would be a better choice. Unfortunately, we waited too long to change our decision, and were not able to implement many of the additional features Oracle hoped we could include.

The technical risk arose from not having access to the oracle campus every working day. This could have presented a major challenge in the technical department mostly because the version of OpenSolaris that we are working with is not publicly available. This implies that if we needed specific files or modules then we have to be within the oracle firewall, ie on the oracle campus.

The final risk involved the skills risk of learning python. Though some in our group were briefly exposed to python in the operating systems course, this was only a brief introduction to python. In order to achieve the results needed, more technically advanced python coding was necessary. This would involve using different network libraries that the group does not have experience with.

To mitigate these risks we used a couple of different strategies. The first and most important risk to manage was that of going down the wrong path. To help keep this as low as possible the group created a list of pros and cons of both methods. In addition to exploring the possibilities we also gave the option of which path to take to the OpenSolaris install team, as they knew far more about the potential challenges/pitfalls and could help guide us in the right direction. Unfortunately, we failed to mitigate this risk, and lost some time investigating the wrong path.

The technical risk was managed by creating a self contained server on a laptop. This will hold all of the packages and files needed without the need to even have internet access. If something had happened to our server installs, we would have had to wait to be on the Oracle campus, and would have had to redo all of that setup. The chance of the laptop failing was much preferable to only being able to work two days a week when we were on the Oracle campus.

The final perceived skills risk involved was our knowledge of python. However, when we changed our decision to extend the Text Installer scripts to add network bootable capability, most of this risk was alleviated. We still had to learn some about python, but it was much smaller than if we had tried to do all the networking in it.

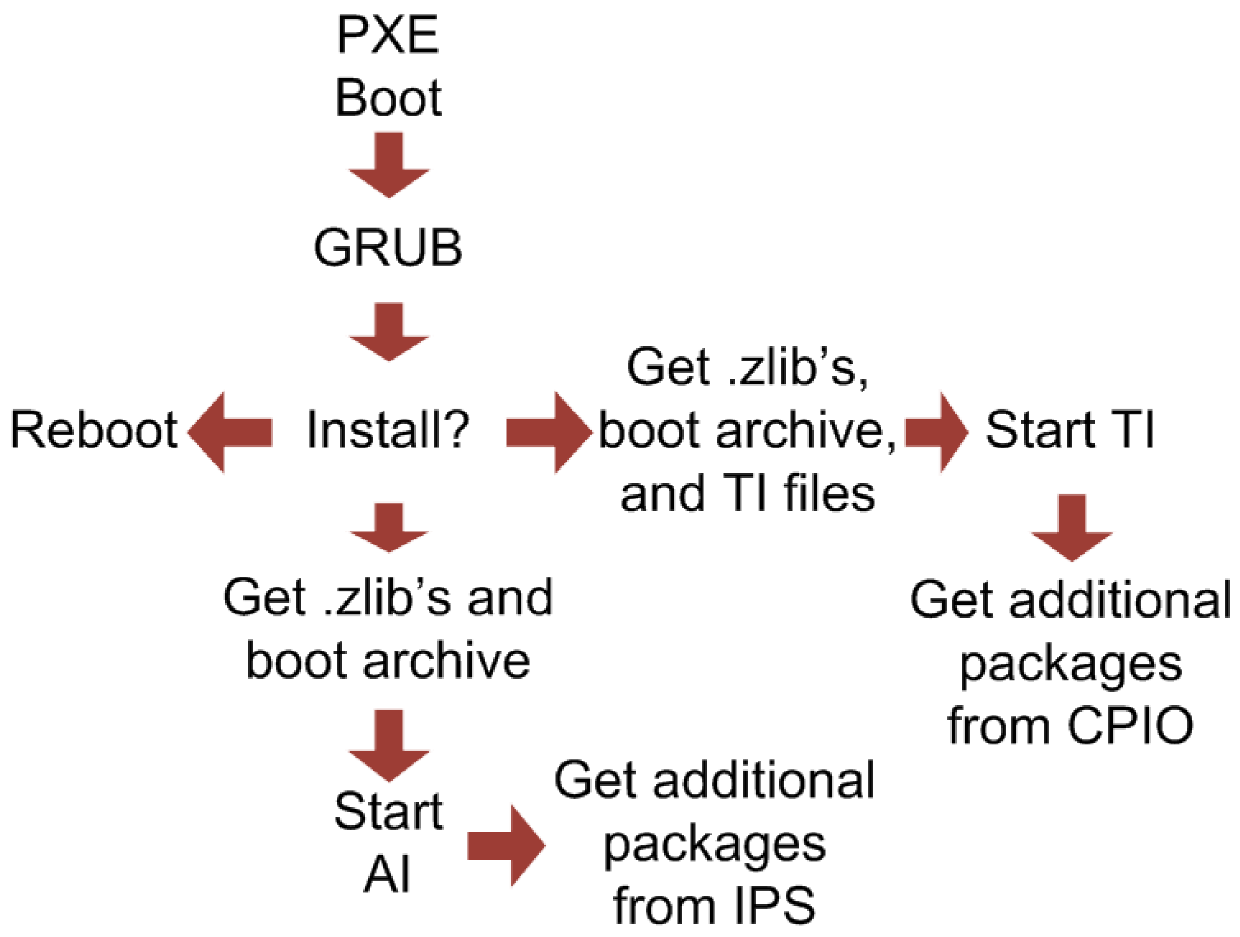
Design

I. System design

Our new Text Installer system consists of two main parts. The first part involved diverging the GRUB loader from calling the Automated Installer scripts and object files to instead call the Text Installer scripts. The second part is modifying the Text Installer scripts to load the information that they need from the network instead of the CD as it does currently.

To change the GRUB menu loader to instead load the Text Installer script we set a flag for the variable text, which represents that it is a text install, to true. The GRUB menu then calls the manifest which is the central file that then calls all of the Automated Installer scripts/object files. We then added tests to see if text was set to true. If it was the Text Installer scripts are called, if it hasn't been modified we will continue as normal.

To modify the Text Installer scripts we first had to check to see if the file we are looking for is available on the hard-coded location, currently the CD. If they are not on the hard-coded location we then change it to look at the network. This works because to get the scripts it either has to be a Text Installer CD or it has to be a network install. If it can find it on the CD we set a global variable of netboot to false and keep the same location. If it can not find the files on the CD then it will set netboot to true and change the file location to over the network.


PXE Boot
GRUB
Install?
Reboot
Get .zlib's, boot archive, and TI files
Start TI
Get additional packages from CPIO
Get .zlib's and boot archive
Start AI
Get additional packages from IPS

II. System testing

As the project progressed we had to regression test each new piece of code that we wrote. Few to none of the tests were able to be automated as there was no testing suite available and there was no need to write our own. Each time we boot up the first test is whether or not the bootloader works. After the system actually boots we are able to watch for output from our scripts to know that the scripts caught the text install flag that is passed from the boot-loader.

We were able to write output to a log file. We used this to add debugging outputs that we searched for when something went wrong. Each time something went wrong the installer dropped the user into a command line interface. In this interface we could read through the logging to find out how far the installer got. There was also a command built in that we used to find why certain services failed. Since each of our scripts are actually services that run during a boot environment similar to the run levels in Linux knowing which one failed told us a lot about what happened.

Since the project was very procedural we could test everything by running the system and finding how far we made it without errors. Obviously we needed to keep track of past tests to make sure that we were making forward progress, but since everything is so modular we didn't have to worry about that. If a section failed we could still manually jump to the next section to test it. This allowed us to work on several sections in parallel. To cut down time we had two team members work on a current section and two researching the next section so that we weren't wasting too much time on compiling.

We tested the finished product by installing it using both methods to check that we did not screw anything up. This involved running the Automated Installer scripts then rebooting and running the Text Installer scripts. After each installation we should be able to boot from the local hard disk and have a working version of OpenSolaris.

Process

Most of our time was spent doing research into how the existing software works so we can interface with it effectively. This allowed us to not duplicate any code and use what was already written by the Oracle team. We had several issues come up while coding. Sometimes the documentation was missing or incorrect.

The first option that we had was to modify the TI to allow network bootable capabilities. The second option was to modify the existing AI network bootable capabilities to call the TI scripts instead of the AI ones. The Oracle team had mixed opinions on this however. To determine which path to go down we created pros and cons and sent it to the manager of the install team. She gave us the freedom to decide which one to use.

We originally started down the path of trying to get the TI to have network bootable capabilities. This proved quite challenging however, and we quickly abandoned this. We changed the direction of the project to instead modify the existing AI code. This had another benefit that both AI and TI were now supported via netboot from one disk. This created new challenges too however because we then needed to modify the menu and menu creator to support this new option

To start we imported the TI code into the AI image and turned off AI so it would give us a console to use. This was done so we could run TI from the shell. This allowed us to test changes that could be made without needing to build a new image saving us time. All we had to do now was change the environment at this time to look like the CD image.

Since the filesystem at this point is read-only with the exception of the /tmp directory we had to make all of our changes using symlinks. Finding all of the files that were needed for TI was time consuming due to the large existing codebase that we needed to read through.

We also were working on getting an extra menu item in the GRUB menu. We decided that the user would need to include a "text=true" flag in the bootargs. These flags were found by using the `prtconf` command. We could use this command in a script to decide which set of install scripts should be run. First, we decided to put this decision in one of the AI scripts which only slightly worked. It made everything glitchy. We finally moved the logic into the script that sets up the filesystem.

Once we had these steps done we now needed to make it easy for a user to install the system. This included changing the installadm tool to include support for our image. We did this by including a text-install file to act as a flag for our changes. When the file was there installadm would know that the image has the option of performing a text based install and includes the option in the grub menu. When there is no such file the tool does not include the option.

The last change that we made was to one of the ICT scripts that set up the boot menu for the installed system. All we did was remove the AI flag and include a TI flag and it worked as expected. Once we had this done we had a working system on reboot.

Final Implementation Details

Most of our coding was done in python, as the Text Installer is currently written entirely in python. We also modified some shell scripts, and other files associated with the Automated Installer. The locations of the files that we modified were all scripts that were called before the actual install scripts. This was because to modify the actual install scripts would have been time consuming, and probably would have caused errors due to dependencies and the complexity of the scripts.

The path we took to solve this problem was to modify the AI net-boot scripts. Looking at the scripts we found that the net-fs-root was the logical place to make modifications to. The net-fs-root is the script that is used to call all of the AI scripts and object files in order to get the installs to work. The net-fs-root downloads all of the files that are needed such as the .zlib's, which are compressed files. It then unpacks these files and sets up the very basic network.

We added to this to first check and see if a flag was set, which is only set if TI was selected at the GRUB menu by the client. If the flag is set it will call a file called *"setup.sh"* that downloads the extra files that the text install needed such as *".live-cdrom-content"* and *".volsetid"*. After doing this it will set up links so that it appears that the files are located in .cdrom while not wasting the memory to copy them to that specific location.

After setup.sh is finished net-fs-root will disable *"manifest locator"* and the console login. The manifest locator is what would start the AI scripts to run and the console login needs to be disabled so that the TI scripts can run without problems. After these are disabled the TI smf script, which is what runs the TI scripts, is enabled.

At this point the environment is set up exactly as it would be if the TI was run from a CD. All of the links are up to allow those files to appear as if they were on the cdrom and the zlibs are in the proper place.

One of the first scripts in the TI that is called is *"ti_install.py."* In this file we made it create a file called *".textinstall"* that needs to be in the root in order for the ICT to create the proper menu. The ICT checks to see if a .textinstall file is in the root, and if it is, it will make the build boot into a text mode. If .autoinstall were in there, it would boot into a graphical version of OpenSolaris.

We then proceeded to modify where in the *"installadm-common.sh."* This is where the menu list on the server is created. To make sure that this was backwards compatible when we made the image of the install we put in a file called .textinstall. The installadm-common.sh looks to see if that file is on the ISO that it is making the server for. If it is there then it will add an extra option of text install where the flag to let net-fs-root be set to true.

In order to get the proper files onto the image we created a script called *"csm.py."* This script copies over all of the files that we modified into the proper place before it is packed up by the distro-const method which actually creates the image. To call csm.py we added a checkpoint to the dc manifest xml which we renamed *"csm_image.xml."* This checkpoint is added in after the original files have been placed so that they end up just copying over those files.

- ti_install.py - added a .textinstall file that is then used to create the proper GRUB boot menu on the client
- .livecd-cdrom-content - Added to boot image, required element for the text install to fully complete
- .volsetid - Added to boot image, required element for the text install to fully complete
- csm_image.xml - used to create actual image, calls csm.py. Also modifes boot_archive to include all files located in text and auto installers boot_archives
- csm.py - copies all other modified files and replaces them in the zlib, so after they are unpacked, text installer has all neccessary files as well as needed logic to ensure it is a text install
- net-fs-root - Where "TEXT_FLAG == true" is tested. If true, using svcadm, net-fs-root will disable the Auto Install SMF script, disable console login, enable the Text Install SMF script, then uses svcadm refresh to apply these changes
- menu.lst - added Text Install option to the initial GRUB menu. Future changes include modification via svcadm so that it automatically includes text install option which is the same as auto install except it also sets text equal to true.
- setup.sh - this script is called before the text install SMF script executes, sets up all the links
- .packages - list of packages
- installadm-common.sh - This is the installamd script that sets up the GRUB menu that is available when the computer boots from the network. If the .text-install file exists on the installer image, then the Text Install option is created along with the Auto Install option. Modification allows backwards compatibility with purely Auto Install image
- .text-install - Created to let installadm to know to create the GRUB menu to include the Text Install option

Results

Our image is slightly larger than the original Automated Installer image. The original Automated Installer image was 289 megabytes, and our new image is 333 megabytes. However, this is still an improvement over the original text installer image, which was 436 megabytes. This is partly because the image contains less packages to be installed. Future releases will include support for selecting packages to install from a network location. This will eliminate the need for large disk images.

The other main criteria that Oracle was interested in was the time it takes for the end user to do a complete install. Since there are two main parts that can be timed with a middle part that is user dependent in between there will be two timed sections to compare; from startup to first user input and from the time the install starts to it's completion.

The time taken from the startup to the first time user input is required is fairly close for both methods. To do this from the CD takes one minute and thirty seconds. Doing this over our simulated network, which is ten mega-bytes, only takes fifty-three seconds. This time is used to set up the environment and get all the necessary files onto the computer.

The next time interval had a much larger difference to it. This time interval was timed from when the actual install started, after all the options have been selected by the user, to the completion of the install. This step takes six minutes twenty-five seconds when installing from the CD. Compared to the two minutes fourty-seven seconds it took when doing it over the network.

This test was very biased, however, mainly due to the fact that not all of the packets are being installed when net-booting. This is not a bad thing however, because Oracle wishes to further modify the TI so that it gets the most basic install, like ours currently is, and uses IPS to get the rest of the packages. This will then allow for further customization by the user.

Scope

The main focus and primary goal of our team was to generate a net boot text installer that will install OpenSolaris on an x86 architecture computer from a server. The Text Installer now emulates the AI network bootable process as closely as possible, such as appearing in the GRUB boot menu in the same place and, ensuring that no part of the Text Installer is omitted.

Coding all the requirements to meet this goal in python and bash was preferable as Sun and Oracle would need to translate anything we write into these scripting languages further into the development cycle anyway. However, with our knowledge of the language and all the requirements pressed on us, it was not feasible for us to write network boot ability in a scripting language.

Instead, we used already existing code for the network boot ability, and did most of our work in the python files of the text install, and some of the bash files associated with the network boot. This method was acceptable and is definitely a valid solution to the presented problem, and it doesn't place a larger burden on the client down the road when it would be time for them to take our code and clean it up. Instead, they only have to change code that already existed and already needed changed.

Conclusions and Future Directions

This project has been a great learning experience for what the industry will mostly be like in the future. We learned that doing backups and using versioning is extremely important, so it is possible to regress. Making poorly documented changes can make it extremely difficult to get back to where you were. If you've made several changes in a system, and didn't document their previous state or back it up, tracking down which change made the code stop working can be extremely difficult. Using a repository, such as Subversion or Mercurial, to store lifetime changes to the code would ultimately save time and energy when dealing with a faulty build.

We also learned being flexible is extremely important. Making a wrong decision can put you in a difficult situation, but clinging to that decision can make the situation even worse. We had initially decided to add the network bootable function to the python scripts of the text installer; however, we quickly realized the amount of work involved with adding network functionality in python was beyond our current abilities. While we could have pursued this option further, we decided that modifying the Automated Installer would be more within our abilities and time constraints.

Had time permitted, extending this process to the SPARC architecture would have been our next step in the project. This would require further research into the differences between the two architectures and what we would have to do differently to ensure the text installer worked on both systems with no input from the user.

The SPARC architecture was originally developed by Sun Microsystems, and Oracle continues to produce it in their server systems today. Therefore, it is important for OpenSolaris to continue supporting the SPARC architecture. It is specifically appropriate to this project, since it targets servers which commonly don't have graphics cards that could support GNOME.

We were also hoping to change the text installer from using CPIO to using the IPS server installed on our server machine. Down the road, this change would have allowed more customization of each system at the installation. Individual packets, and their dependencies, could be added and removed before install. This would take place on an additional curses screen, which could either list all available packages, or commonly installed packages.

The end goal of the install project is a single image that contains the Automated Installer and the Text Installer. The image should be able to network boot, as well as install from a CD or USB image. Use of the IPS server to acquire packages is an integral step in keeping the size of this image small enough for a single image to be feasible.

The install image we created will currently network boot both the Text Installer and the Automated Installer, and will boot the Automated Installer from a CD or USB. While making the text install work from CD or USB should not be difficult, we ran out of time and did not have a chance to implement it. All in all, our project was a step in the right direction, and should prove helpful to the install team.

Glossary

- AI - Automated Installer, a set of object files and scripts that will automatically install the default settings for OpenSolaris.
- CPIO - A binary file archiver and its resultant format, used in the text install to get packages
- DHCP - Dynamic Host Configuration Protocol, Dynamically assigns IP addresses
- GNOME - GNU Object Model Environment, an open source graphical environment
- GRUB - Grand Unified Bootloader which allows multiple operating systems on the computer
- ICT - Install Completion Task, this task adds the final install to the GRUB menu of the system
- IPS - Image Packaging Service, contains all available packages for install
- OpenSolaris - An open source Unix operating system created by Sun and now maintained by Oracle
- PXE - Netboot firmware implemented in the x86 architecture

- TI - Text Installer, a set of python scripts that allows for customization of OpenSolaris install.
- TFTP - Trivial File Transfer Protocol, used to provide the microroot and zlib files
- ZFS - Filesystem that OpenSolaris uses.

References

OpenSolaris Bible; Solter, N., Jelinek, G., Miner, D.; Wiley Publishing 2000