

Node Abstraction Techniques for Linux Installation

Sean Dague

IBM Linux Technology Center, Poughkeepsie, NY

Abstract

There are over 150 Linux Distributions in the wild, each with their strengths and weaknesses, and each providing something that no other Linux Distribution provides. This allows Linux to reach into niches that no other operating system can touch, and build tailored solutions for individual problems. It is a great strength. It is also a great weakness. In the void before the Linux Standards Base, many Distributions have gone very different directions in structure and layout of their operating environment.

This paper will take a concrete example, network setup, and present a method for gracefully and natively dealing with these differences. It will also touch upon the newly forged System Installation Suite project, and how it will leverage the methodology presented in this paper to become a Linux Universal Installer.

1 What is a Computer?

What is a computer? Is it a beige colored box that sits on top of your desk at work? Is it a 2 inch thick pizza box that runs your web site? Is it the game console in front of your TV at home? Is it your palm pilot or cell phone? Is it the anti lock breaking system in your car? The simple answer is yes.

A computer is just a collection of electronics that does something useful. It starts with a central processing unit, which is nothing more than an abstraction layer above logic gates, as no one really wants to program logic gates. Great, so you have a bunch of logic gates with some slightly easier interface on top of them, this alone doesn't get you very far. Next you need some way of getting power to your CPU, and letting it communicate with other devices. Here enters the mainboard. The mainboard provides one or more communications buses. You can then add random access memory,

which gives you fast volatile storage for all the data going on and off the CPU bus. You generally add some sort of non-volatile storage (be it hard drives or ROM) as you never know when the State of California is going to hit you with another rolling blackout, and reentering the contents of RAM by hand is a pain in the butt. You may also add some sort of communications card (be it Ethernet, myrinet, serial, wireless, or something even more esoteric), which allows your CPU to communicate beyond the walls of the box that it exists in. You may also add other interfaces for input or output, like a keyboard, monitor, mouse, joystick, etc.

No matter what sort of “computer” you run, the picture looks pretty much the same from a hardware perspective. This rings true from everything from a desktop PC, to a large server, to a Playstation, to a cell phone. However, just having the hardware, doesn’t get you very far. Without software on top of it, you have just constructed a very expensive shiny new boat anchor.

As with hardware, software exists in layers. The first layer of software is generally considered the Operating System. This provides an abstraction layer over the hardware it is sitting on, so each application need not concern itself with what really needs to happen to write 5 bytes to an IDE tape drive. Running an Operating System is easy, getting it loaded into memory and started is a bit more interesting. This age old chicken and egg problem has been solved many times, many ways, but in each case one thing is true: the bootstrap layer must have intimate knowledge of both the hardware that it is running on, and the software it is trying to load.

Once the Operating System is loaded and running, you finally have an Application Layer of software. This software is generally what is employed to get real work done. Well written software at both the application level and the OS level consists of binaries which can be easily moved around from one location to another, and configuration data, which is generally in a human readable format (sendmail being the exception that proves the rule). A good example of this would be the Linux kernel. Although the same base binary may be running on many machines (for instance the Red Hat 6.2 2.2.14 kernel), they may be running in very different environments. Through the manipulation of `/etc/modules.conf`, different pieces of kernel code, and different options for those modules, may be active in memory on the different nodes. Because the software was built well, it didn’t need to be recompiled for every different node that it functions on.

2 Installing a Node

After careful examination of many installations of various computer operating systems, we realize that installation of Operating System and software applications on a node consists of a small number of discrete steps. These steps hold true regardless of Operating System or hardware.

1. Prepare Non-Volatile Storage for writing (Disk Partitioning and Formatting)
2. Apply Software (Install Operating System and Application Software)
3. Modify Software Configuration Data
4. Setup Bootstrapping

By looking at installation as these 4 steps, a few interesting observations arise. First, step 2 of this process consists of nothing more than writing bits to a hard drive. Other than the hard drive partition format (which has been abstracted in step 1), this is completely Operating System and hardware independent. This could be as simple as a straight copy of the software from one drive to another. It could even be some type of imaging software that ran under an operating system different from the one being installed. The only requirement is that the end result of the copying creates sectors on a disk that make sense to whatever Operating System will be running on the machine.

We have now made one task totally independent of operating environment, so we have three tasks left to tackle. None of them are very large in scope. Disk partitioning is tied to both hardware and software. The hardware may have requirements on what the partition table and disk labels on the disks look like. The software may only understand a few different partition types and formats. Hence, it can't be done in a vacuum. Bootstrapping is also tied to both hardware and software. It is the bridge by which the hardware hands off control to the Operating System, and must have intimate knowledge of each. Step 3, the software configuration, is obviously dependent on the software that was installed in step 2. As with the example of `/etc/modules.conf`, it may also have some dependency on the hardware, as well.

We have now defined the problem of installing any Operating System. If we become more specific, and take the case of installing a specific Operating System, Linux, it becomes fairly easy to build an abstraction layer above all three steps which are hardware and software dependent. You then have a tool which can install any Linux on any Architecture.

3 Motivation

Why is it important to abstract node installation for Linux?

The short answer: Red Hat != Linux && i386 != Computer.

The longer answer, different people have different needs.

In the `arch` directory of a stock 2.4.4 kernel there are 15 separate architecture subdirectories. Linux has been ported to all these platforms because someone, somewhere, found it really useful to run Linux on them. Some users need 64bit processing, some need faster floating point math than x86

provides, and some just need other features not found in x86 (like mean time between hardware failures at 30 years). If it wasn't useful to run Linux on non x86 platforms, people wouldn't have spent the considerable amount of time it takes to port Linux to them. Linux may have been born on x86, but it has grown quite a bit since those early days.

Talking Linux Distributions is like talking about politics. Everyone has their feelings about what is best. Usually it is some combination of whatever would actually install on their home machine, what comes with their favorite tools, and what didn't blow up on them at 2:30 in the morning taking out 3 weeks of valuable work as it went down. Irrespective of the reasons why people use different Linux Distributions, the fact is, they do. You either recognize it, and find a way to work for all Linuxes, or tie yourself to one vendor, and have the success of your product ride upon the success or failure of someone else's company or organization. I think this is a mistake many learned the hard way in the bad old days of closed source software.

Besides the pragmatic reason for making these abstractions, as we need them to unify installation, there is another solid reason to do it. It just isn't that hard once you have collected the right data.

When LUI 1.x was released its goals were to be platform and Distribution independent, however the development team had only used Red Hat 6.x on x86. The assumptions they made about what was core Linux, what was Red Hat specific, and what was x86 specific happened to be wrong. That really isn't a surprise. Any scientist will tell you that theories based on one data point are more akin to tarrot card readings than anything related to the scientific method. If you take a step back, look at a few more Linux Distributions, and a non x86 platform or two, you begin to see a bigger picture of what Linux is, and where Distributions and platforms differ. You are also greeted with a pleasant surprise... things aren't nearly as bad as they could be. Even though each Distribution could have done things totally differently than every other Distribution, they tend not to diverge that often, and when they do, it is for a real reason. This is one of the niceties of GPLed software, the energy to keep up a fork is much more than the energy to copy someone else's code, hence forks are rare.

Lets take our previous abstractions and apply them to a specific case, installing and configuring Linux just enough that it can boot up and attach to the network without human intervention. Once you have a Linux machine in that state, you can remotely access it to do any further updates that are required. With just these goals in mind, there are a small number of tasks which have to be accomplished:

- Disk Partitioning
- Detecting Hardware, and aliasing the proper kernel modules
- Creating Network Scripts
- Setting up the bootloader

We know that each one of these has the potential to be different depending on the Linux Distribution or hardware platform that we are running on. Lets take a more detailed look at network setup, to see how different Distributions solve this problem, and see if we can figure out a way to setup networking generically for all of them.

4 Network Setup

Bob Metcalf is famous for, among other things, Metcalf's law. Roughly stated, the usefulness of a network is proportional to the square of the number of devices on it. Without networking, a computer is nothing more than an expensive type writer. With networking, a computer can become a portal to the vast information stored on the internet, a device from which email and instant messages can put you in constant communication with folks on the other side of the world, or even a small piece of a very large super computer. From a management perspective, once a computer is on the network, and has a remote access daemon running, physical access to the machine is no longer required to accomplish useful work with it.

Setting up a networking script in Linux can be as simple as:

```
ifconfig lo 127.0.0.1
ifconfig eth0 192.168.64.128 network 192.168.64.0 broadcast \
    192.168.64.255
route add default gw 192.168.64.1
```

This is actually very close to the way that Debian used to configure networking. All the function necessary is there and it works great to bring a machine onto the network. Unfortunately, as a user, you need to know that you have to add a route for every interface that you bring up, and you have to add it by hand. Also, if you wanted to drop a live interface, you would have to go through the man pages for `ifconfig` and `route` to find out what needs to be done with them. If you wanted to write a configuration tool to build this script automatically, you would have to know all the intricacies of `ifconfig` and `route`. This works ok for simple networking paradigms, but gets complicated fast when you add additional interfaces, or other advanced networking is attempted.

The raw networking solution is an ugly solution. Because of this fact, very few Linux Distributions still use it. Most of them have created at least one level of abstraction between `ifconfig` and `route` and the user. Lets look at three popular abstractions that are used by many Linux Distributions.

4.1 LinuxConf Networking Templates

Configuring Linux used to be really hard. You pretty much needed to know the intimate details of every piece of software on you machine to get any of

them to work. One of the things that Linux needed to expand into less hard core technical areas is a more gentle learning curve. One way to accomplish this was to provide GUI tools that helped a user configure most of the applications on their machine. LinuxConf was one of the first attempts at doing this. It wasn't (and still isn't) perfect, but it made a lot of people's lives a lot easier, and hence was adopted as a standard by many Linux distributors.

LinuxConf attacked the networking problem by creating a set of "Network Templates" which were located in `/etc/sysconfig`. These templates were nothing more than a set of shell variable declarations. This made them easy to parse in any language, easy to manipulate in Linux Conf. It also meant that the system initialization scripts could still be written in shell, as the scripts could just source the files to get access to their variables.

The crux of LinuxConf networking is the cooperation of a number of different files. First, there are a series of `ifcfg` scripts for every network interface on the machine, located in `/etc/sysconfig/network-scripts`. They look something like this:

```
/etc/sysconfig/network-scripts/ifcfg-eth0:  
DEVICE="eth0"  
BOOTPROTO="none"  
IPADDR="192.168.64.189"  
NETMASK="255.255.255.0"  
NETWORK="192.168.64.0"  
BROADCAST="192.168.64.255"  
ONBOOT="yes"
```

These variables provide all the information that `ifconfig` needs to run. The information is presented in a way that is a bit more human readable than a raw `/etc/init.d/network` file. Should you wish to specify that the interface be activated via `bootp` or `dhcp`, you merely need to specify that fact in the `BOOTPROTO` field. The scripts which source this file will handle `dhcp` or `bootp` client configuration if appropriate. Depending on the Distribution being used, differing `dhcp` clients may be used by default.

Secondly there is the network file. This specifies items about networking in general on the machine, which include what the default route should be. Any information not specific to an individual interface, but concerning all interfaces also goes in this file. The file format looks something like this:

```
/etc/sysconfig/network:  
NETWORKING="yes"  
FORWARD_IPV4="false"  
HOSTNAME="loki.sis.pvt"  
DOMAINNAME="sis.pvt"  
GATEWAY=192.168.64.1
```

Again, simple to parse, simple to use. So why doesn't everyone use this format? Well, in the early days of LinuxConf, the tool was more useful for data purging than anything else. Often LinuxConf would turn a perfectly good hosts file into something which looked a lot like rot13ed Cyrillic piped through gzip. Because of these early failures, many Distributions decided not to use LinuxConf.

Any Distribution that used Red Hat 5.0 or later as their base, generally still uses this type of networking format for their network scripts. Usually Distributions which start with Red Hat as a base, are mostly concerned about fixing the other issues with the Distribution, and don't really care about infrastructure items such as how networking is set up. (For instance, Mandrake was originally just Red Hat + KDE back at version 5.0).

4.2 Rc.Config

In the Perl world there is a saying "There is more than one way to do it". This mantra states the philosophy of the language, as well as a simple truth about the world: Given any problem that arises, there are generally many equally good solutions to the problem. Each solution has its advantages and disadvantages. The solution which is right for you depends on what criteria is most important, be that security, space, time, or pure aesthetics of the solution. Just because it is a different solution, doesn't make it less valid. As Fredrick Brooks said, "There is no silver bullet."

As LinuxConf was evolving, slowly, SuSE Linux decided they needed a solution to this problem as well, and solved it in a different way. One of the disadvantages to LinuxConf's configuration style, is that the information about a system's configuration is scattered through a myriad of different files located throughout a system. This can be a real pain in the butt if you would like to use a tool other than LinuxConf to report vital stats about a machine.

SuSE solved this by creating a single configuration file for the entire system, located at `/etc/rc.config`. This file is a huge list of shell variable declarations, similar in many ways to the network templates LinuxConf uses. As manipulating this file by hand is a pain, and potentially dangerous (an erroneous entry could wipe out more than the section you were attempting to change), SuSE provided a tool to manipulate this called YaST, Yet Another Setup Tool. YaST provides a unified view of all the configuration changes that can be made on SuSE Linux. The section of `rc.config` which pertains to networking looks as follows:

```
NETCONFIG=" _0"  
...  
IPADDR_0="192.168.64.128"  
IPADDR_1=""  
IPADDR_2=""
```

```

IPADDR_3=""
...
NETDEV_0="eth0"
NETDEV_1=""
NETDEV_2=""
NETDEV_3=""
...
IFCONFIG_0="192.168.64.128 broadcast 192.168.64.255 \
          netmask 255.255.255.0"
IFCONFIG_1=""
IFCONFIG_2=""
IFCONFIG_3=""

```

You will note that in this file the actual `ifconfig` command line is stored as a string, versus storing the individual pieces and building the `ifconfig` string on the fly. If you wish to bring up an interface via `dhcp` or `bootp`, you need only replace the `ifconfig` line with the proper keyword. The `NETCONFIG` variable list which interfaces are to be activated at boot time. This is very similar to the `ONBOOT` flag in `LinuxConf` templates, which lets one store information for interfaces which will not get activated automatically. Default route is actually handled by a different file, `/etc/route.conf`. A single line of the format `'default 192.168.64.1'` will bring up the default route during network initialization.

No other Distributions that I am aware of use this type of networking. One of the main reasons is the fact that `YaST` is not true open source. It exists under a license that precludes its Distribution without permission from `SuSE`. This is unfortunate, as it means that `SuSE` is often considered a special case when it comes to Linux Distributions, even though their mind share is very high.

4.3 Interfaces

Shell scripts are fine for storing configuration data, but the format is clunky. If you define a large number of variables, you have to be exceptionally careful of name spacing issues. However, if you attempt to structure the data more than just a simple set of variable declarations, you no longer can use shell programs to interface with the data.

In the last Debian stable release (2.2, aka potato), a more structured format for networking was developed. The data is stored in `/etc/network/interfaces` in a structure as presented below:

```

iface eth0 inet static
    address 192.168.64.128
    netmask 255.255.255.0
    gateway 192.168.64.1

```

broadcast 192.168.64.255

If dynamic ip resolution is required, the static keyword is replaced by 'dhcp' and the rest of the information about the interface is not required.

Debian is able to use this more structured data format because the 'ifup' and 'ifdown' programs which run to activate and deactivate network interfaces are compiled C programs in Debian, whereas they are shell scripts in most other Linux Distributions. This means that systems administrators can't simply modify the main network ifup and ifdown behavior without getting the source and recompiling. This is not necessarily a bad thing. The common data format is also a big step forward from the raw network scripts used in Debian 2.1.

There are many Linux Distributions based off Debian. Because Debian is a community organization with no business model, it is seen by many as being more true to the Linux ideals than many of the Commercial vendors. Progeny, just to name one example, is a new corporation basing their business model around taking current Debian stable, adding a few configuration tools on top of it, and selling support contracts around it. Corel Linux is also based of Debian (though a much older version was used when the fork first happened).

5 Footprinting

The biggest sin that most people commit when writing code is testing for operating systems, and then make assumptions based on that. What is an Operating System anyway, especially when it comes to Linux? How many machines are really Red Hat 6.2, out of the box, no modifications made? Of these, how many have been taken over by a 12 year old kid in Canada, and are now running egg drop as their primary function?

Linux is a very fluid beast. Many people and companies are scared of it because it is such a moving target. All a Linux Distribution really is is a set of current software that was compiled with the same c compiler and c library, which happens to have passed a few integration tests. The administrator is free to add or upgrade software, either from packages or from source, as soon as they are done installing the nodes. Is Red Hat 6.2 with a 2.4 kernel still Red Hat 6.2?

Looking at the questions we realize that when most people make a test of the format `#ifdef REDHAT62`, what they really are trying to do is figure out what base libraries exist, and how the files are laid out, then make decisions based on them. Why not do that directly?

If we look specifically at the example of the three different network types we could write a piece of code which does the following:

```
if(has_LinuxConf_templates) {
    setup_LinuxConf_templates;
```

```

}
if(has_rcconfig_file) {
    setup_rcconfig_file;
}
if(has_etcnetwork_directory) {
    setup_etc_network_interfaces;
}
}

```

Here we are testing for capabilities, and doing the right thing depending on what type of footprint we find on the machine. If a new Distribution comes along, and we don't have any knowledge of it, if it conforms to the way that one of the other Distributions does for networking, it will work out of the box. If not, it is easy to add an additional test and setup routines for the new type of networking.

Just because networking is handled differently, doesn't mean that other aspects of setup (like boot loading and kernel modules setup) will be different. This takes potentially a very large problem, configuring and installing all Linux Distributions, and turns it into a bunch of little problems that are easy to solve.

These same strategies can be applied to setting up bootloader and kernel modules as well. If a new need arose, like setting up network time services on a machine, this could also be handled under the same strategy: test for one of many configurations, and do the right thing when you found the configuration.

New setup aspects can also be easily added based on capabilities. An example might be the addition of network time synchronization. There is no guarantee which network time programs exist on any machine. You could test for them all in a specific order, and do the right then as soon as you found one that worked.

Testing for capabilities makes you Distribution inclusive, rather than Distribution exclusive. It means that at a minimum you support the number of Distributions stated in your README. Odds are, you actually support many more than that.

6 System Configurator and System Installation Suite

Theory is great, but “a line of code is worth a thousand words.”¹ We have taken the theories laid out above, and created a utility called System Configurator. System Configurator is a non interactive application which provides a unified API for Networking and Bootstrapping across Distributions and architectures. For the sake of making Bootstrapping and Networking much easier to get right, it also provides a modest hardware detection layer for Network Interface Cards and SCSI block devices. System Configurator is

¹Linux Kernel Mail List FAQ

designed to be a library that can be used by any installer. It is written in Perl to make it as platform independent as possible.

The goals for System Configurator v1.0 will support the following:

- At least 20 Linux Distributions, including Red Hat, SuSE, Debian (2.2 and 2.1), Mandrake, Connectiva, MSC.Linux and others.
- 3 Hardware Platforms x86, PPC, Alpha. Itanium support may go into v1.0 depending on availability of hardware for testing.

The first major user of System Configurator is VA System Imager. This is part of a foundry of projects called System Installation Suite. This foundry is a joint community effort being spear headed by VA Linux and IBM's Linux Technology Center. It is based on the work that the IBM LTC did on LUI (Linux Utility for Cluster Installation), and the ongoing work of VA Linux on VA System Imager. It consists of three main programs:

- SystemInstaller - a generic installation program that creates master images for installation currently in beta. This team is headed by Michael Chase-Salerno of IBM's LTC.
- SystemImager - a program developed by Brian Finley at VA Linux which uses rsync to replicate master images to client nodes.
- System Configurator - a postinstall configuration program whose job is to smooth over all the differences between Distributions and architectures for node setup. This team is headed by Sean Dague of IBM's LTC.

The System Installation Suite's goals are grand: install and configure a collection of machines with any Linux Distribution, on any Hardware, using the same interface.

SIS introduces a new model for Linux Installation. Instead of doing installs machine by machine, SystemInstaller generates one software image for a group of machines, SystemImager replicates this image out to many machines at once, then System Configurator automatically configures those items which may be different between the machines.

SIS is not tied to any platform or Distribution, so anyone using Linux can use SIS to install and manage their machines. Although making Linux cluster installation simple is a primary goal of SIS, it is not the only goal. A corporation might have a fleet of laptops that all need to be kept up to date, SIS will work great for that. A computer manufacturer might have a standard installation that they want to go out on all there computers, SIS will do that wonderfully. Any time you have a group of machines running Linux which all look very similar from a software perspective, SIS is the right tool to use.

7 Conclusion

The Linux Standards Base is coming, but even when it arrives, it will not solve all the differences between Linux Distributions. Bootloaders will always be hardware dependent, there will always be multiple package managers, and Systems Administrators will always install some software from source. Distributions will rise and fall like the city states of Ancient Greece, their popularity waxing and waning based on sun spots and phases of the moon. If one uses the techniques described here when writing any sort of software for Linux, that software will truly be generalized Linux software.