Tuxconfig.

A Linux device configuration tool.

Robert Brew.

Abstract

Despite it's small memory footprint and install size, Linux is way behind windows when it comes to choices of operating systems, rarely bundled with a computer bought from a major supplier. One reason for this is the problem of configuring devices using a terminal console, forcing the user to enter obscure configuration commands. Ubuntu has this covered in it's restricted driver application, but the devices listed here are few and far between compared to the vast array of devices supported for windows.

Can we find, programmatically, a more effective means of installing obscure hardware, removing the need to type in commands to configure a device in place of a graphical tool to do so?

I have :). I present a client application and server back-end to do this.

This is not only an application to install third party drivers. It is a platform from which knowledgeable Linux users can submit code to a centralised database for standard users to download, compile and install hardware drivers

Acknowledgements

I would like to thank:

Kent university.
My parents for their support.
My supervisor, Ian Utting, for his insight.
The god of coffee.

Content Page

Structure of dissertation:

- Requirements document.
- Architectural design.
- Design influences
- Design
- Testing.
- Conclusion.
- References.
- Appendices.

Introduction

This dissertation answers the question, can we safely and programmatically implement a piece of software to enable the installation and configuration of devices in Linux using a graphical interface.

Currently in Ubuntu there is a graphical means of achieving this using the restricted drivers program. However this is limited to closed source devices as configured by the Ubuntu development team. Can this be improved on? Can any device install be verified not to be malicious? Can this install process be made safe? Can a platform to submit device configurations be made available? Can the success or failure of an install be logged, in order to recommend successful installs to future users?

The nearest point of reference to this project is the Ubuntu restricted drivers program, the source code for which I have not analysed. This dissertation builds on this available program by creating a means for any developer to submit the software needed for a device to work to a central server from which those device configurations can be made available to other users.

The program is called Tuxconfig and consists of a client front end written in C++ with a server / database written in Java and Mysql. Contributions to a device install are made available by referencing repositories as stored on the GitHub.com website. GitHub.com is a service which stores source and binary code interacted with using the git protocol.

This document explains the architecture for the proposed system from a high level of abstraction. It is intended for people with an understanding of the process of compiling and inserting Linux kernel modules using bash.

1. Current System

This project builds on the current “additional drivers” program in Ubuntu which allows proprietary firmware to be installed using a graphical interface.

<image restricted drivers program>

This application, part of the software and updates package, is automatically launched in situations where a device is found in the system for which a suitable driver implementation can be found. Proprietary drivers can be installed using this system.

Users can install device drivers from this window, but as far as I can see they cannot un-install them.

The phrase proprietary when included in this tool implies that open source device configurations cannot be installed using this application. Instead the Ubuntu team have worked with technology companies such as Nvidia and Ati to collaborate on providing drivers for their respective devices.

There appears to be no platform for submitting device drivers to be used as part of the distribution, leading me to believe that the drivers provided in Ubuntu are made in conjunction with the hardware manufacturers of devices available as part of a system configuration.

This project goes further than the Ubuntu restricted drivers program by offering a platform for anyone to submit code to make a device work. This project also goes further than the current implementation by allowing non-proprietary devices to be submitted then installed to the user’s operating system as well as proprietary drivers and those provided by the package managers for Linux which in the case of Ubuntu is the apt package manager.

The ability to un-install configured devices, upgrade current configurations and roll back previous installations is desirable from the produced program. The ability to restore a previous configuration compensates the situations in which installing a driver causes problems with the configuration of the host machine. In cases where the graphics adapter is not working a terminal based recovery system can be used to restore the devices to their previous configuration.

2. Goals, Objectives, and Rationale for New or Significantly Modified System

2.1 Project Purpose

Create a brand new system based on the current system, Ubuntu additional drivers to install, uninstall, roll back and upgrade device drivers.

2.2 User Community Description

Two types of user will take advantage of this system. The client users will benefit from not having to use the shell to install kernel modules, which many people find obscure and intimidating. The developers, with knowledge of kernel modules and how to engineer them, will suggest configuration processes, with the incentive of having their details recommended to the user installing the device configuration they have authored. For the purposes of this document the contributing developers are referred to as developers, and the client users as users.

2.3 System Goals and Objectives

Make it easier for novice Ubuntu users to configure hardware. Provide a platform for developers to contribute open and closed source code for a client to install. Provide a system for new device configurations to be proposed with a vetting process to ensure these configurations do not contain malicious code.

2.4 Proposed System

This project enables a user of Linux with limited technical knowledge to easily install Linux kernel modules using a graphical interface. It relies on the Linux community to suggest configurations for each device being installed based on the identifier of that device.

As an incentive for contributions the developers will be able to link to their their git profile, plus a web page to the devices they contribute to the platform. When the users have successfully installed a kernel module the developer who contributed the configuration will have a link to their web page, email address, git log and git biography displayed by the program. The user chosen will be based on the most successful contribution to the device

installed by the user, as voted by other users on the success or failure of an install.

In order to access the GitHub repositories I am using the oauth protocol. This allows a developer with a GitHub account to authorise the tuxconfig product to act on the developer's behalf. From this web based protocol the back-end can derive the public repositories the developer owns and then allow him or her to submit a repository to the project. The database stores the URL which is cloned by the front-end as part of the configuration of a client device, along with the commit hash to ascertain a snapshot of the current repository.

In order to prevent malicious code being added to the project a vetting page will be created which will allow those running this system to ensure the Makefile is not malicious and that any device binaries are from genuine sources. The approved commit of a git repository will directly reference the code submitted to the team vetting the configurations, and not subsequent code. When pulling a git repository to the client system this commit is "checked out" to be the same as the state of the repository when the code was submitted. The vetting page will be accessible only on localhost. Access to this host can be used by port forwarding when an authorised user with the correct SSH key logs into the host.

In it's initial concept three types of customisation were planned, configuration of software available from the repositories using apt-get and yum, kernel parameter configuration (for the kernel make config process), and firmware downloads, possibly using all methods for one device configuration.

The initial idea was to allow users to submit config flags from which the kernel could be recompiled for the client user. This would require recompiling the kernel, building a kernel image then rebooting the system with that compiled kernel. The scope of things to go wrong with this method is large, and in certain cases can lead to an un-bootable system with the generic user being lost on where to proceed.

I purchased a wireless ac adaptor and looked on github.com to download the related firmware and kernel module to make it work as a device. The standard install means when compiling source code in Linux follows the procedure as below:

./configure – generates the Makefile.

Make builds the code

make install – installs the code.

For kernel modules the process is slightly different. Configure is often not needed and once installed to the correct module directory under the “/lib” folder the module can be inserted into the kernel using the “modprobe” command. On restart the module will be inserted at runtime as set up using the “depmod” command, which configures the modules as well as their dependencies.

Rather than recompile the kernel as a whole it became obvious that we can leave the vanilla (default) kernel as it is and build and insert kernel modules to make the devices work. This requires less work and reduces the scope of things which can go wrong compared to rebuilding the entire kernel.

Sometimes packages need to be installed as well as the module to make the device work, or maybe the kernel drivers can be updated using the package manager by itself without the need to build kernel modules. Therefore developers can submit an “apt-get” style package management request by itself without manually building and inserting kernel modules,

When contemplating how to discuss issues with a particular configuration I realized that github.com hosts everything this system could need. This includes but is not limited to Makefiles, source code and firmware files. Therefore rather than host files to be used myself I can reference the github.com repositories as a file host.

Initial ideas were to allow the developer to contribute their own Makefile for the install. The custom Makefile could contain details of how to clone the target git repository, build the firmware using make and insert the module, returning a success or failure exit code once this is done. Client users clone the repository into the host system before checking out a particular commit which is the state of the repository as it was submitted before running the build and install process for that module.

This has subsequently been replaced with a standard Makefile and an optional configuration file called tuxconfig.

The tuxconfig file works in harmony with the Makefile. It contains a list of device IDs, the module name, dependencies which need to be added using

apt-get, the test program to run, a message to describe to the user what the output of the test program should be, as well as a flag to define if a restart is needed after a device install.

After the module has been installed the user will be prompted on the success or failure of the kernel configuration on a per device basis. The initial plan was to use a one time code generated by the back end server to ensure that each device success or fail status can only be reviewed once, in order to attempt to stop saboteurs from manipulating the success or failure of an install, by providing the database with false positives for device installs. Such a system can be easily circumvented by up or down voting particular installs, therefore the idea has been superseeded as described later in this document.

A program to demonstrate the success or failure of a configuration can be launched, as part of the post install process. This could involve playing a sound to ensure the audio device works, or opening a webcam program like cheese to test the video camera. The result of the running of this program is then used by the user to ascertain the success or failure of a device install. This information is used by the system to then recommend the best device install submissions to future client users of the system.

When developers contribute to the system they use the Oauth protocol to give our system permission to act on the behalf of the developer when interacting with Github.com. We can use this permission to write to the developer's issue tracker for that device install listing details of the error logs when a failed configuration is run.

In order to prevent multiple error reporting of the same error the length of the error report being submitted is checked with the error reports already there and subsequent error reports are not uploaded to the github.com issue tracker for that device configuration.

The specifications for submitting a project are as follows:

The package must be installable as a standard "make", "make install", "modprobe <module_name>". As mentioned earlier a full tuxconfig file must be made available, This information is imported to the database having being parsed by the back-end.

2.4.1 System Scope

As this is a proof of concept it's limited to only working with GitHub, running on the Ubuntu platform on the amd64 architecture using kernel version 4.15. As mentioned earlier a system in place to stop false error reporting of failed devices has not been implemented.

2.4.2 Business Processes Supported

The system will be of most use to home users with little experience of configuring Linux, that said it is likely even capable system administrators will use it as it alleviates the need to find the correct firmware projects on git and perform a manual install.

The data held in the database could be made to include the device id's, device descriptions and module names of each device supported by this application, for those who wish to search for device details without using this system,

2.4.3 High-Level Functional Requirements

Developer side:
Allow developer to contribute git project to system using github.
Import details of this repository into the database.

User side:

Installation:

Clone git repository from GitHub.
Install software via apt-get.
Execute shell script.

Back up /etc/modules.* and /lib folder.
Insert kernel module.
Provide feedback for developer.
Provide details of the contributing developer to the user.

Uninstallation:

blacklist the modules in /etc/modprobe.d/blacklist file.

Remove the kernel module using "modprobe -r"#

<implement restart if module cannot be removed>

upgrade:

Run the install procedure again.

Restore:

Restore the tar to the operating system and reboot if necessary

<implement reboot on restore if not already there>

2.4.4 Summary of usage:

It may be worthwhile contacting the Linux kernel team about this project when both the code and write up is submitted. From there they might be able to provide a team to do the vetting process or take on the project from me. Alternatively I might be able to continue the project and moderate the github.com contributions to this system using git pull requests (submitted improvements to the code), maintaining the project myself.

3. Factors Influencing Technical Design

3.1 Relevant Standards

The use of GET and POST HTTP standards, the JSON standard for communicating between and and front end. QT libraries for graphical display, the use of Java beans in the back-end written in Java with a similar concept in the front end in c++. The Oauth protocol allowing developers to list and then add their repositories and for error reporting.

3.2 Constraints

Multiple distribution , architecture and kernel support is beyond the scope of this project.
The end user must have an internet connection, to download the package and during the programs execution. Restoring from a failed command can be possible offline, in cases where the wireless card has ceased to function. Therefore the necessary details on how to restore a device configuration must be stored on the client user's system, as it will not be available to download from online without internet access.
The user must have the device to be installed with a matching configuration submitted to the database in order for an appropriate driver to be downloaded.

3.3 Design Goals

Incentive to contribute is crucial.

User interface must be easy to use.

Vetting of configurations to ensure malicious code is not used is crucial. As we are effectively inserting code into the kernel this cannot be circumvented, even using chroot. In it's place is a vetting process.

System must be extendable for future use, for future LTS kernels and different Linux distributions.

Ease of use to contribute at GitHub (add tuxconfig file).

Rating configurations should be tamper proof. One way (beyond the scope of this project) is to allow each user to only vote once on each device install

success or failure. This would require a database table of users with email addresses and passwords etc. Even this may not be satisfactory.

One way round this is for the administrators of this site to inspect the errors logged at the git issue tracker to see if they are relevant to the fail and ask them to intervene at the database level. Also public private key technology to ensure the application and only the application is talking to the back end could be deployed.

The client application must be easy to install and preferably run automatically on install.

4. Proposed system

4.1 High-Level Operational Requirements and Characteristics

In order for a device to be used in this system it must first be contributed. This is done by the developer logging into the site https://linuxconf.feedthepenguin.org/hehe/GitAuth.html. From here the developer has the option of logging into GitHub using the Oauth protocol. This allows the application to read the developer's public repositories, and post to the repository's issue tracker on behalf of the program. Information of the developer, the bio, location, avatar image, email address and the users website are pulled from GitHub and stored in the back-end database.

From here the developer chooses which repository to add. The server application clones the repository in the /tmp directory and ensures the "tuxconfig" file is there with it's attributes set correctly, before importing the "deviceid" array and "modulename" into the database along with the URL for the repository and the current commit hash.

The user downloads the client program as a Debian package from the host server. The user then installs the client package and is directed to launch the client program on the user's machine. Using xdg-open the package can be opened with the Ubuntu software centre, which will also take care of installing the dependencies. The package install process can use post install hooks to run the application after it has been installed.

The client program identifies the device by it's unique device ID. It contacts the server and if a configuration has been submitted for that device and that that configuration has been authorized as not malicious. The server returns a link to the GitHub project for the client to download and then install using the Makefile with configuration options in the "tuxconfig" file).

The client clones the repository from GitHub, installs any packages (if listed) and builds the kernel modules, if part of the repository. It then attempts to insert the newly built kernel module into the kernel, returning a success or failure result. In the case where an install has failed the client program tries again with the next most successful configuration. Successful or failed results of running the test program are uploaded to the database.

If the configuration fails the errors logs for that configuration are uploaded to the GitHub tracker for that repository.

In situations where the computer needs to be restarted the client embeds it's startup in the X windows startup folder "~.config/autostart/tuxconfig.desktop" as well as in the ~/.bashrc file in cases where a terminal restore is needed. On voting for the success or failure of an install this entry into the start up process of a machine is removed.

Options to restore the configuration files are made available at startup by graphical and non graphical means. If the program is started without a grpahical display available it defaults to the recover process.

I feel it important to mention malicious code. If an install wrecks a system due to a bad makefile I would lose credit as being the cause of it. Also if trojan horses are bundled into the configuration businesses and users could be at risk. The triage system could involve only users with a high stack overflow rating to triage applications. Many users with a high reputation could be created to manipulate the triage system in favour of exploiting this process to leave projects in the system containing malware. Reference to these users could be made using Oauth requests as previously mentioned. Another alternative is to leave a 2 week wait on code in the hope that anything malicious will be removed from the site. In cases where a rare configuration is used it could still cause havoc on the client machine.

We are inserting code into the kernel. Nothing can mitigate against this directly, even running the code in a virtual machine and seeing the consequences of running the code may miss time based malicious behaviour.

4.2 Technical Architecture

The system relies on a MySQL database, maria DB, accessed using Java EE Servlets using get and post requests of JSON type parameters, running on tomcat8. Bash will also be used extensively to clone the git repository and insert the kernel modules, executed from the client program. Rather than writing communications protocols for communication between the C++ front end and the Java back end I propose using HTTP requests. These can be debugged using a standard web browser.

Data collected:
Device information.
Developer login information (using oauth for git).
Information on the success or failure of a particular configuration.

The system uses a two tier architecture with an attached databsae, the GUI uses MVC principles.

Database: MariaDB, an open source MySQL compatible database.

Backend: JavaEE provides a web based framework, using Java beans to communicate with the back end database with ease, as well as the serving of web pages as JSP dynamic pages. An alternative would have been PHP.

Frontend: I could have written this in Java, which does support the execution of commands. However the community, if this project is adopted would continue to develop this in C++, as most software for Linux is written in this language (and C).

Host operating system: Ubuntu is the most popular operating system for novice users of Linux. It supports the installation of custom software using a graphical environment. Whilst it would be nice to support all versions of Linux on multiple architectures with multiple kernels this is beyond the scope of my project, which might mutate into the multiple distribution support after this project has been shared with the Linux community.

Web server: Apache. SSL support is enabled using lets encrypt, a free SSL certificate provider. This is needed to prevent warnings about insecure forms appearing on connecting web browsers when the developer is submitting information.

Copyright:

Seeing as this project uses parts of the Linux kernel, as far as I am aware that means that this code must also be released as an open source project. This is also convenient to bypass the university's copyright issues, also ensuring that, if the project is taken and passed on by someone else, it has my name attributed to it.

Makefile identifiers:

in order to parse information from the tuxconfig configuration file we need to use non obvious identifiers to the strings we are referencing. Otherwise a situation may arise whereby the contributor references a variable of the same name and the system becomes poorly configured. The "ARCH" variable is one example of this.

Versioning:

If the user wishes to make more than one version of a repository available this is possible using different commits, of course a new commit will have to have it's commit hash verified by the maintainers of this project, to ensure that the snapshot of code which was submitted is the version being used at the user end.

Hosting:

Kent University provide a dedicated cluster with open stack deployments of Linux servers. However the servers are currently only available to local users at Kent university or those of a Virtual Private Network, as they use private IP addresses. For now I am using a digital ocean droplet to host the code. This is necessary as the oauth callbacks must point to a public URL in order to respond with the correct Oauth token form github to the back end.

The path to the Java servlet instance is behind an obfuscated URL, in this case "/hehe". This is to stop bots and other people observing this project and completing it for themselves.

Updates:

If a device configuration comes out which is more popular than the current one (as derived by upvotes – downvotes) this result can be compared to the popularity of the current device configuration and a new version can be downloaded and run. This does not cause a positive recursion problem as the

success or failure of installing new version will also be voted on by the client user. Successful device configurations will have a higher success vote, leading them to be downloaded by default.

Permissions:

As the target user will be limited in their knowledge of Linux they may not understand the concept of root as the administrative user. The system must run as root in order to be able to manipulate the operating system. The system was built on my Debian instance for which gksudo is available. When installed under Ubuntu I realised that gksudo was depreciated, with pkexec as an alternative. In it's place I am using sudo -H in the desktop shortcut file and sudo when the program is in terminal mode. Sudo -H asks for the user's sudo password in order to grant priviedges to the user allowing them to act as root. Using the $SUDO_USER variable the program can discern who the calling user was are add to the startup process of this user in cases where a restart is needed. Also running the email client and web browser for this user is needed when selecting the option to email the contributor or visit his or her webpage respectively.

Graphics:

A bash script to ascertain if the GUI is running has been written as the file tuxconfig_cmd. It runs the program in recovery mode if the X server is not running. This implies that a poor configuration of the graphics drivers has been run and therefore the configuration must be recovered. It also explains why the user password is being asked for when the program is run as part of the startup process as specified in the .bsahrc file. Adding the line "Ctrl – C" to exit will show a user how to escape the program if they wish to continue as normal.

As in tradition with Linux the /var/lib/tuxconfig folder will be used as the working directory of the program.

Multiple device support:

I realised having installed a wireless .ac USB dongle that one kernel module can support multiple devices. Therefore the tuxconfig configuration file will allow multiple devices to be entered into the database for each commit and URL. The devices table in the database will need to be split into a table for devices and a table for the configuration variables which apply to those

devices. One is device_id, a table of devices, descriptions, related GithHub.com URLS and commits. One is a table referencing the configuration options for a particular repository, called git_url. The devices table includes the device id, git url and the commit hash for that device configuration. The commit hash and git url columns are referenced by the git_url table as foreign keys <FIX must ask Ian about foreign key problems>.

Multiple distribution support:

For now checks are made to ensure the client program is running from Ubuntu, this could be changed to provide client side identification of other operating systems. <FIX, must implement in final version>

As module source code is made for each operating system on GitHub in source code which is then compiled for each architecture and operating system we could allow one repository to support multiple operating systems. The install platform is referenced in many Makefiles by the $ARCH variable. As this is a proof of concept this functionality hasn't been implemented in this project.

Apt install packages:

Apt-undo is a script which allows the rolling back of repository changes. I propose using this system to reset the package installs of attempts to configure the device. In cases where the system has not made any changes to the package structure (where the dependencies variable is empty) the rollback command will not be run as this would revert genuine changes to the package structure not made by this system. This could confuse the user in cases where the apt-get phase of the install process has failed. Therefore the install script must record the success of an apt-get install. Apt-undo will therefore only roll back packages which have been installed as part of the install process for this system, and not previous package installs.

Naming:

The name Linuxconf is taken, so I propose tuxconfig,

Contributing:

We could use a client program to test the configuration works for a different device, as well as ensuring the parameters are set correctly. This is already done by the back-end when the repository is added to the project, and therefore duplicate code. There are cases where a developer might take

someone else's code and make it available for other devices with the same chipset, therefore not requiring a device to test an install on.

Cloning repositories:

If a repository is removed from github the backend realises this and deletes the repository url in the database. In cases where a malicious repository can be set up with the same name as a valid repository this attack will fail as the git commit will be different for each of the listed repositories.

The client clones a repository from github.com, before checking out a particular commit. It runs a bash file which then runs "make" to build the code, before running "make install" to install the module to the /lib directory. It then inserts the module into the kernel. If any errors occur during this process the error is trapped by a function in the executable bash file and a SIGUSR1 kill signal is sent to the running C++ application. This signal, when sent to the program causes console window to show that the device install has failed.

A log of the running install script is then sent to the back end to be posted on the issue tracker. In order to show the minimum amount of information to the user bash stdout Is sent to /dev/null and stderr is displayed to the user. This reduces the amount of information displayed to the user and only displays errors which might be worth noting.

In cases where an install succeeds without error a SIGUSR2 kill signal is sent to the running application. This signal is caught and the console tab displays the test message as defined in the tuxconfig file as well as the success and failed buttons. The testing program is launched and the user can comment on it's success. The testing program could be "cheese" in installs of a webcam, or x-www-browser youtube.com/video to check the workings of a sound card.

Where successful installs have occurred the user is then directed to a tab displaying the details of the contributor.

Makefiles built from a configure script will not be included as part of this project, as there are too many variables to consider when building the Makefile.

History file:

A file stored at /var/lib/tuxonfig/history contains the history of the run of this project. It is used by the restore command to revert the system to it's previous state, as well as informing the general tab in the application as to what has been installed and what can be upgraded. The restore tab uses this file to ascertain the success or failure of a device install.

A template expected from the developers contributing to this project will be available here https://linxuconf.feedthepenguin.org/hehe/Contribute.html.

<implement web page>

Deriving installed devices:

The program lshw, with the argument "-numeric" lists all devices in a system including the device id. The front end parses this document before connecting to the back end to retrieve information on any configuration details which have been added to the system.

Device ids:

These take the format nnnn:nnnn, where n is a hexadecimal character.

When running lshw – numeric devices are listed with zeros at the beginning and end of the device id omitted. In lsusb these are implemented. For the client using device ids in this fashion is OK, however for the backend these must be taken into account.

Therefore on the backend if the device string is less than four characters at each side of their separating colon the zeros are added before the device is logged in the database, and added when a device is retrieved using it's id.

Webpage:

<image gitauth.html>

<implement, rename to index.jsp, add package>

This is the landing page for the application. It allows users to download the project as a Debian package, as well as allowing developers to login using the Oauth protocol to github which then authorises the back end to act on their behalf with regard to the users' public repositories.

<image index.html>

This page derives the information from github to ascertain the developer's details which is parsed from a received JSON array using the "repo" scope. The repo scope allows tuxconfig to get details of the developer's public repositories as well as write on the issue tracker for each repository. The developer can then contribute a repository in it's current state to the system. On selecting the repository the developer would like to contribute the repository is copied into the /tmp directory on the server instance. This is then parsed to get the commit hash of the repository before the tuxconfig file is checked to ensure it exists and the parameters for the tuxcofig file are set correctly. The data copied into the system is not used beyond this by the server, which instead points to the github URL.

<implement delete repository when parsed>

<image VetConfigurations>

This page allows an administrator logged into localhost to approve, revoke or delete a submitted repository submitted by a developer.

GUI:

Initially I was using multiple windows for notifications. This was hard to navigate, involving an application tab, running console and test program with a success or failure dialog box. This has been consolidated into one window of five tabs, each with a different purpose. One tab is a terminal widget from which the scripts will be run for install, uninstall, restore and upgrade.

<image about tab>

Explains what the program is for, links to the General tab and restore tab.

<image general tab>

Replace option: Device install options only available where they exist in the database.

Uninstall option: The option to uninstall devices only exists where one has been installed by this application, otherwise people would uninstall devices where there is no replacement, leaving a naive user having to restore a non-functioning machine.

Upgrade option: when the device is installed a log of the device's vote cound is kept in the history file /var/lib/tuxconfig/history. If the database

has a higher rated install for a particular device id the option to upgrade the device is shown. As the success of a device is upvotes – downvotes this feature is not time critical.

The tab can be refreshed at any time by clicking on the “rescan USB bus” button.

<image restore tab>

Restore tab: Entered automatically when the program is run with a “recover” argument. This allows a user to restore from a previous install, at whichever point it got to. This is done by extracting the most recently created tar of the /lib/ directory and the /etc/modules* files for that device_id. In cases where a user has not yet voted on the success of an install this option is made available. In cases where a device configuration has been altered the tab is updated by parsing the history file again.

<image console tab>

Running qtermwidget this tab displays the actions triggered by the install and recover commands. The user votes on the success of an install using these and the result is sent to the backend and stored in the database. In cases where the install has failed no buttons appear and the error logs are sent to the back end. In cases where an install has failed the program selects the restore tab for the user to restore the configuration.

<image contributor tab>

This displays the details of the contributing user in cases where the device has installed correctly. It comprises of the users’ github profile and bio, along with a website for the user and their email address. On clicking the email link the originating user (remember this process runs as root) the xdg-email program is opened for the logged in user to write to the contributing developer. Similarly the webpage for the user is opened by the logged in user by clicking on the http link for the contributing developer.

RestoreCmd:

<image restoreCmd>

This is a simple text based aspect of the application which asks the user if everything is installed correctly. It is run automatically by the .bashrc file of

the user's home directory. It first asks if everything was configured OK, before asking the user to enter the device_id of the configuration which failed. The latest written backup tar archive for that device id and commit is then restored to the system. Once the success or failure of an install have been voted on automatically starting the program is removed from the startup process.

The system comprises four major components, as illustrated below:

<image>

System architecture:

Database:

My initial database schema is attached below.

<image schema>

Database normalisation:

I left this activity to the end of the project to be sure that each field in the database holds relevant information for this project. On realising that each device must be linked not only to a git url but also a git commit it has changed slightly:

<image new schema>

The git_url table must also reference the following properties of the device table:

commit_hash

git_url.

These should be constrained by a foreign key constraint existing from the device table into the git_url table as a foreign key.

The git_url table must be constrained so that each owner_git_id exists in the contributors table.

As mentioned earlier the success_code method is open to manipulation, so whilst implemented this table is not used.

Voting on the success or failure of a device is not reported in the client app. My reason for this being it is beyond the scope for a standard user to know or care on the outcome of this.

Realising the device id’s do not need to be displayed in the VetConfigurations console only the commits, hashes and developer id’s are listed.

Foreign keys:

<FIX must implement>

Files:

/usr/share/pixmaps/tux.png (splash screen).

/usr/share/pixmapstux2.png(icon)

/usr/local/bin/apt-undo

/usr/bin/tuxconfig-cmd (bash script to check if graphics are running).

/usr/bin/tuxconfig (main application).

/usr/share/applications/desktop/tuxconfig.desktop.

These are added to a Debian package for the user to download and install.

<fix implement skip network check on recover>

Retrying failed attempts:

This is wired into the restore tab widget. In the event of an install failing the user will be offered the chance to try another install, if one is available. The database on the back-end server is queried by the number of entries in descending vote order as limited by the “attempt number”. The databse returns the most successful configuration after the higher voted ones have been chosen. The number of failed installs is derived from the history file with the “failed” entry being referenced with that device id in the same line.

<implement don’t offer next install where none exists>

Dependencies:

It should be mentioned here that build-essential should be installed on the client device for the user to build everything using g++ and make etc. This is not implicitly obvious from the code documentation.

Packaging details:

Each of the above files will be packaged into a .deb file for Ubuntu users to download and install on their machine. This can be done graphically using a web browser. The program should run automaticity on install.

How it could be improved:

As mentioned in order to restrict the voting mechanism to one user / one install one vote a database of users could be created, ensuring that only one vote can be cast per configuration per user, to stop people providing false positives or negatives to the database, thereby invalidating the vote success information.

Multiple platform, architecture and kernel version support would be nice, as this is a proof of concept this version does not contain such information.

Rather than allow the back-end to post to the contributors github.com page using their oauth token an alternative should be used, maybe pastebin or github.com's gist for error reporting.

A database of device id's with the attached device description and related module name would be useful to advanced user's searching for device information. <implement such a web page>

Conclusion:

I like to think that, using Github and standard protocols, I have made a complicated proposal quite simple.

I've used a variety of technologies, including c++, Java, web based technologies including JavaScript, databases using MySQL as well as Linux commands written in bash, requiring a good knowledge of the Linux filesystem and Linux commands. The process of writing this code has dramatically increased my understanding of C++, which, whilst being harder to master than Java I am starting to enjoy.

I've found that people are surprised at how easy Linux is to use, but not configure. I hope this project solves perhaps the biggest problem of Linux adoption. That of requiring knowledge of the console to configure each device which cannot be installed automatically.

The Linux kernel maintainers have / will be informed of this project, and I hope the Linux community will adopt it as a standardised platform for submitting and implementing drivers for the Linux operating system, not only for Ubuntu but for each of the main distributions out there.

Thanks for reading.

Rob Brew.

Thanks to:

The library writers for CurlPP, JSONPP, Qt and HTTPDownloader.

TOO:

Fix warnings.

Appendices:

Logbook

:Git logs for the back and front end I am using:

I realised these can serve well for the log book, and as of 01 / 8 / 18 I am using the commit messages for this.