University of Kent

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 to install the drivers to make a device work. Ubuntu, a major Linux distribution, 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 available on the market.

Can there be, 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?

This paper presents tuxconfig, a client application and server back-end to do just that.

This is not only an application to install a significant array 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:

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? My proposed system successfully implements answers to all of these questions.

The nearest tool 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 device configurations can be made available to standard users to install easily and without using the terminal.

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 website. GitHub 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, before defining how the built code will run, then demonstrating a walk through of the built system. It is intended for people with an understanding of Linux.

1. Problem Description

Downloading drivers not automatically supported as part of the kernel in Linux can be a complicated process. If the user is lucky a package will be available in the Linux repositories which can be installed (when running Ubuntu) using the Ubuntu software tool. One example of this is fglrx, the Ati/AMD software driver.

If the drivers for the device the user wishes to configure are not available as part of the distribution software repositories they will, at least, have to:

- Ascertain the device’s identifier.
- Find the software to make the device work (if available).
- Build and install the module.
- Insert the module into the kernel.

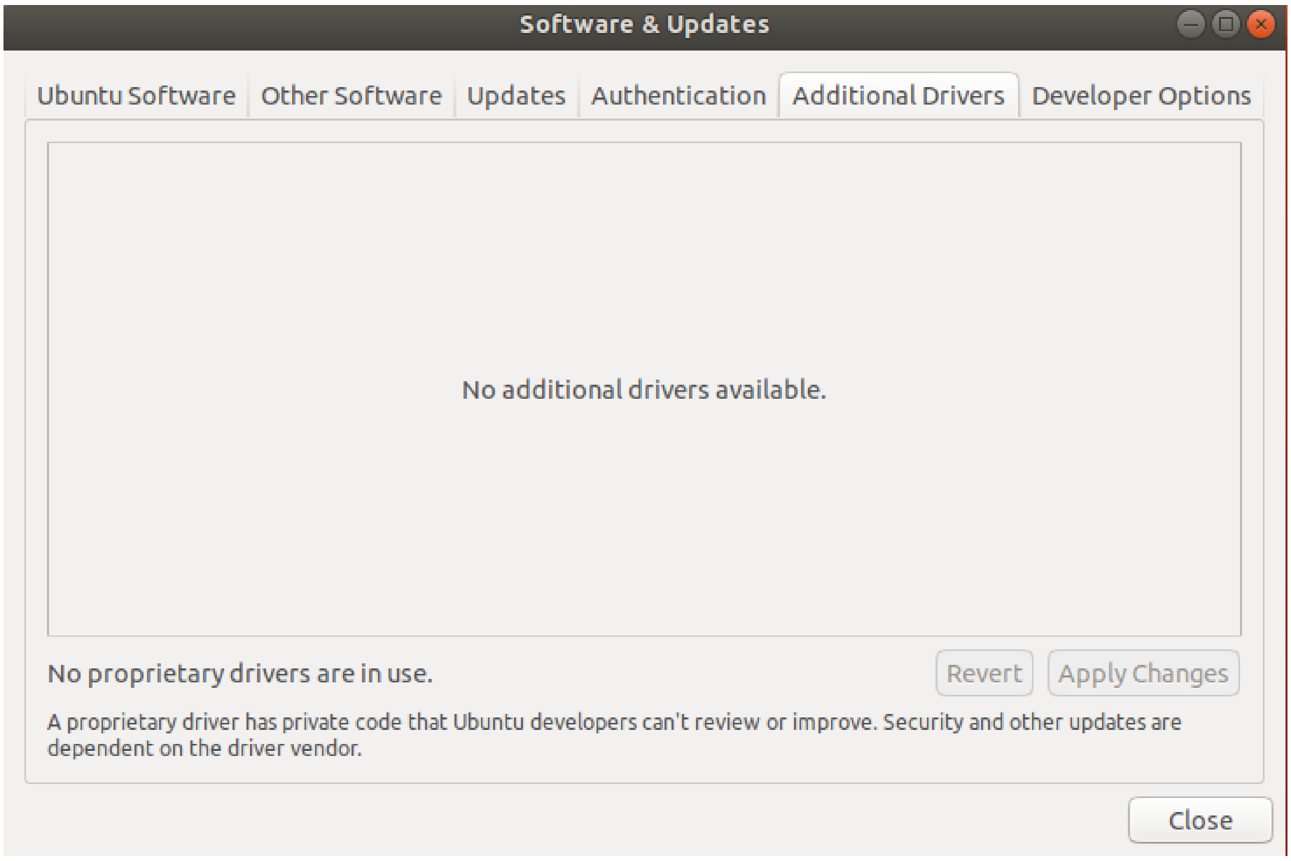
Using the command line terminal.

This process often alienates a lot of users as being arcane. The process of typing commands into the terminal seems outdated since the days of windows ‘95, where the user installs a device from a CD by running the CD and clicking buttons to configure their new device. Whilst windows may not be technically superior to Linux in terms of size and speed, it wins over Linux by providing an easy way to get the devices of the host system up and running.

User’s also have no way of knowing if the provided device driver is malicious, such as containing Trojan horses, or being a risk to the integrity of the system

1.1 Ubuntu restricted drivers program

Ubuntu’s “additional drivers” program reduces the learning curve of new users, but has it’s limitations.



No additional drivers available.

No proprietary drivers are in use.

A proprietary driver has private code that Ubuntu developers can't review or improve. Security and other updates are dependent on the driver vendor.

The 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 and revert these changes.

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.

A similar (command based system) is DKMS, standing for dynamic kernel module support. This automatically rebuilds kernel module when a system is updated, as well as allowing a user to un-install kernel modules when using it as a framework. It does not support the device verification to ensure that a configuration is not malicious, or provide a means of finding device drivers suitable for a host device. Users are expected to find the correct dkms archive for their system, and to use the command line to extract the archive, build the kernel modules and then insert the built module into the system.

As demonstrated the current means to install devices on Linux is hard to use and scares new users away. This project will hopefully go a long way to making novice users feel at ease with the operating system.

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

2.1 Section introduction:

This section describes the purpose of Tuxconfig and it’s user base, before it looks at how tuxconfig builds on the ubuntu restricted drivers program, and defines what it Wil do from a high level of abstraction.

2.2 Project Purpose

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

Provide a centralized database from which users can find the code to make their devices work, without having to use Google to find drivers which may not work and have not been audited, or may not exist.

Allow submitted device configurations to be checked to ensure that they are not malicious.

In cases where a device install has failed provide an alternative configuration to the user where one exists.

In cases where a configuration may break a device allow the user to roll back the configuration to it’s previous state.

Ubuntu is a version of Linux based on ease of use. Its is a good choice of distribution for those transitioning from windows as it is built around the concept of being easy to configure. This project is a proof of concept to demonstrate that easier ways to configure devices can be built than the current methods available.

2.3 Contrast to Ubuntu’s “restricted drivers” program.

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

It 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 this current implementation by allowing non-proprietary devices to be submitted for users to install. In cases where the device can be configured using software from the Ubuntu repositories the proposed system will handle this.

2.4 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, in places where the install is successful. For the purposes of this document the contributing developers are referred to as developers, and the client users as users.

2.5 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.

2.6 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, and git biography displayed by the program. The developer 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 the developer owns this system uses the oauth protocol. This allows a developer with a GitHub account to authorise the tuxconfig program 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 when it was submitted.

In order to prevent malicious code being added to the project, a vetting page will be created which will allow those running this system as moderators 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 revisions to code on GitHub. 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 server page can be used by port forwarding when an authorised user with the correct SSH key logs into the server.

In its initial concept three types of customisation were planned, configuration of software available from the repositories using apt-get and yum (package dependency managers), 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 developers 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.

Case study, wireless USB adaptor:

This device can be made to work with a vanilla (standard) kernel by building a kernel module and inserting it into the kernel. The standard procedure for compiling code from source in a Linux machine is as follows:

- ./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 (using “make install”) 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.

Revised process for this project:

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,

Rather than host the necessary files on the database for this project it became apparent that GitHub hosts everything this system could need. This includes but is not limited to Makefiles, source code and firmware files. Therefore the system can point to GitHub repositories, using it 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 superseded as described later in this document.

A program to demonstrate the success or failure of a configuration is then 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.

As stated, 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. 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 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, with reference to the module id and devices available. The restart needed flag, test command and text asking about the success of the test command are read by the front end.

2.6.1 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.

2.6.2 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 is made to include the device id's, device descriptions, device names and module names of each device supported by this application, for those who wish to search for device details without using this system.

2.6.3 High-Level Functional Requirements

Developer side:
Allow developer to contribute git project to system using GitHub.
Store git hub details of developer to display to user in successful install.
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 /et/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 /et/modprobe.d/blacklist file.

Remove the kernel module using “modprobe -r”

upgrade:

Run the install procedure again.

Restore:

Restore the tar to the operating system and reboot if necessary.

The system successfully implements each feature defined in this section.

3. Factors Influencing Technical Design

This section describes how the platform will be built.

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 on the user’s machine. 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 installed via this program.

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).

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 user then has the option of running the program themselves.

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 start-up in the X windows start-up folder by copying the tuxconfig desktop shortcut file from /usr/share/applications/desktop to “~/.config/autostart/tuxconfig.desktop” as well as in the ~/.bashrc file in cases where a terminal restore is needed.

Options to restore the configuration files are made available at start-up by graphical and non graphical means. If the program is started without a graphical display available it defaults to the recover process, believing that the graphical interface is broken.

The concept of malicious code should be mentioned. 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.

This project works by 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 this system uses 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 front end uses a two tier architecture with an attached database, the GUI uses MVC principles.

Database: MariaDB, an open source MySQL compatible database.

Back-end: 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: If this project is adopted by the Linux community it will be developed in C++, as most software for Linux is written in this language (and C). This project could have been written in Java, which would have been easier to implement than 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 multiple distribution support after this project has been shared with the Linux community for contributions to code, after it has been submitted.

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, under the GPL licensing this may imply that this code must also be released as an open source project. This is also convenient to bypass the university's copyright issues, 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 be verified by the maintainers of this project. The git commit has will reference that particular contribution in the database.

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 the project is hosted as a droplet using the digitalocean cloud provider. 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, for use in accessing the contributor's details and his available public repositories.

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. 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. In Ubuntu gksudo is depreciated, with pkexec as an alternative.

Being able to use pkexec requires an entry in the /usr/share/polkit-1/actions directory informing the system as to which file to give privileges to. A message as part of the window to grant permissions to the user is recorded "password is needed to configure Linux devices".

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 start-up process as specified in the .bashrc file.

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

Multiple device support:

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. This information is stored in the tuxconfig file on the git repositories with each device_id entry delimited by a white space.

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.

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. This system can be used to reset the package installs of attempts to configure the device.

Ideally apt-undo would record instances where the install of a package has failed. However, aptitude, the program referenced by this wrapper returns an exit code of 0 in cases where the package is already installed in the system. Therefore apt undo may ask the user to remove packages which have nothing to do with the dependencies of a device install. Having looked at the exit status of apt-get it seems that deriving such a failed install a not possible using these tools. The user should be informed of this potential problem.

Naming:

The name Linuxconf is taken, so the name tuxconfig is being used. Tux is the Linux mascot.

Contributing:

A system to test the devices to be configured for the developer could be in place. This would ensure that the an install works for a particular device. A developer may not possess the device to be contributed but still have knowledge on how to make it work. Therefore this process has not been implemented, relying on developers to submit their code by authorising access to Gtihub repositories.

Cloning repositories:

If a repository is removed from GitHub the back-end 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, 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 by the developer.

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.

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/tuxconfig/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. The general install tab uses this information to offer upgrades, uninstalls and reinstalls to the user.

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

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, using the hexadecimal identifier for each device.

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 back-end these must be taken into account.

Therefore on the back-end 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.

Running commands as current user:

As the front end runs as root using the “pkexec” command the system will by default run all spawned applications as root. Running applications as this users is ill advised where not necessary.   Firefox and google chrome will refuse to run as this user. Running an email client to send an email to the developer or running a web browser to display the developer’s homepage as root will not be possible.

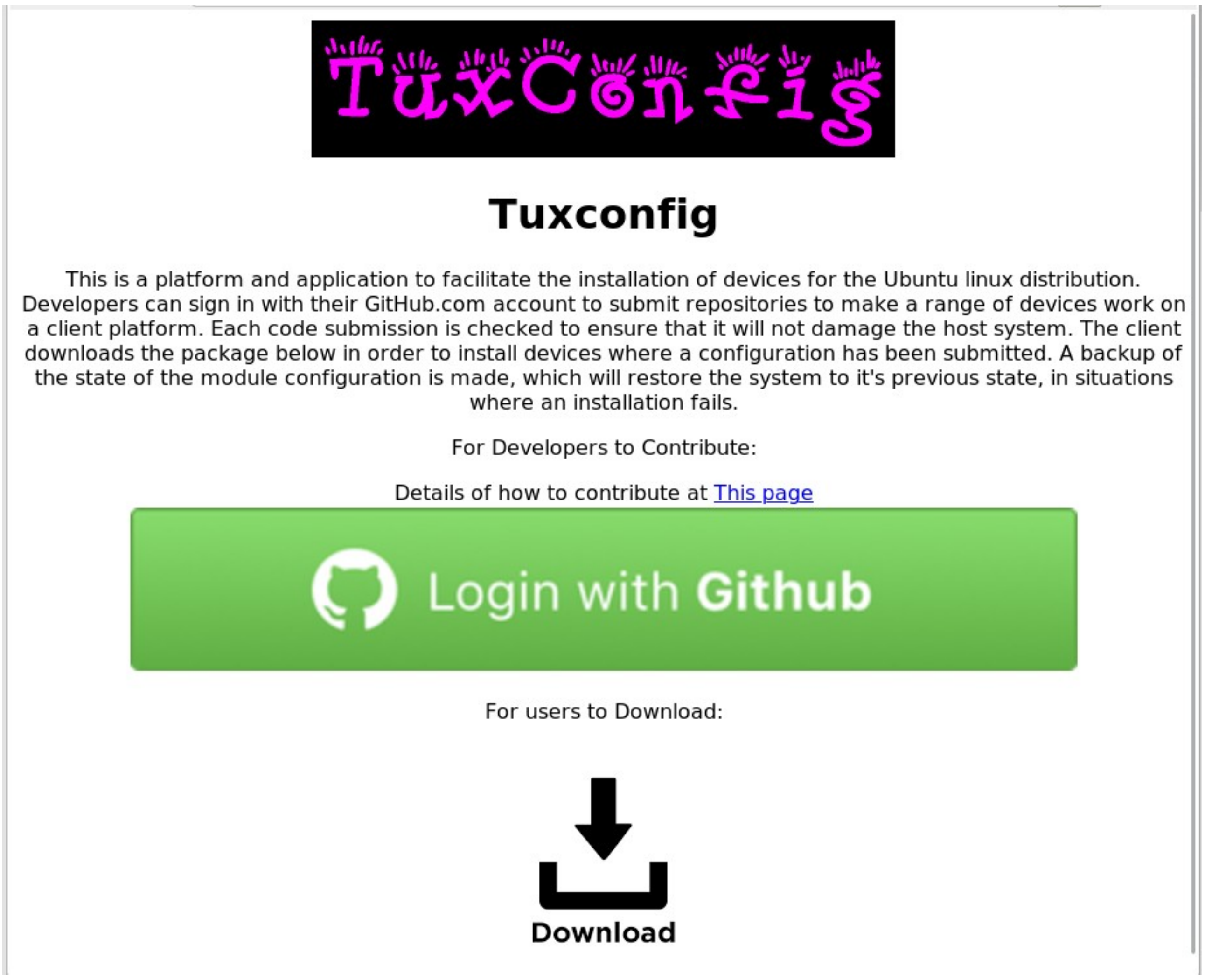
The current username cannot be exported to pkexec as it discards exported virtually all variables. The environment includes the “PKEXEC_UID” variable which declares the UID of the calling user. By referencing the /etc/passwd file the user’s home directory and name can be derived, allowing the program to run necessary programs using the “SUDO” command. When run as a terminal application the sudo command can be used to grant privileges instead of pkexec.

Client machine install:

The program and it’s related files have been packaged in a Debian package format titled tuxconfig.deb. This is available form the landing page of the website at https://linuxconf.feedthepenguin/hehe. In ubuntu downloading it will provide the option to install the package plus it’s dependencies from the command line. The user then has the option of launching it from the desktop shortcut menu in their toolbar when running Gnome.

Walk through:

Webpage:



Tuxconfig

This is a platform and application to facilitate the installation of devices for the Ubuntu linux distribution. Developers can sign in with their GitHub.com account to submit repositories to make a range of devices work on a client platform. Each code submission is checked to ensure that it will not damage the host system. The client downloads the package below in order to install devices where a configuration has been submitted. A backup of the state of the module configuration is made, which will restore the system to it's previous state, in situations where an installation fails.

For Developers to Contribute:

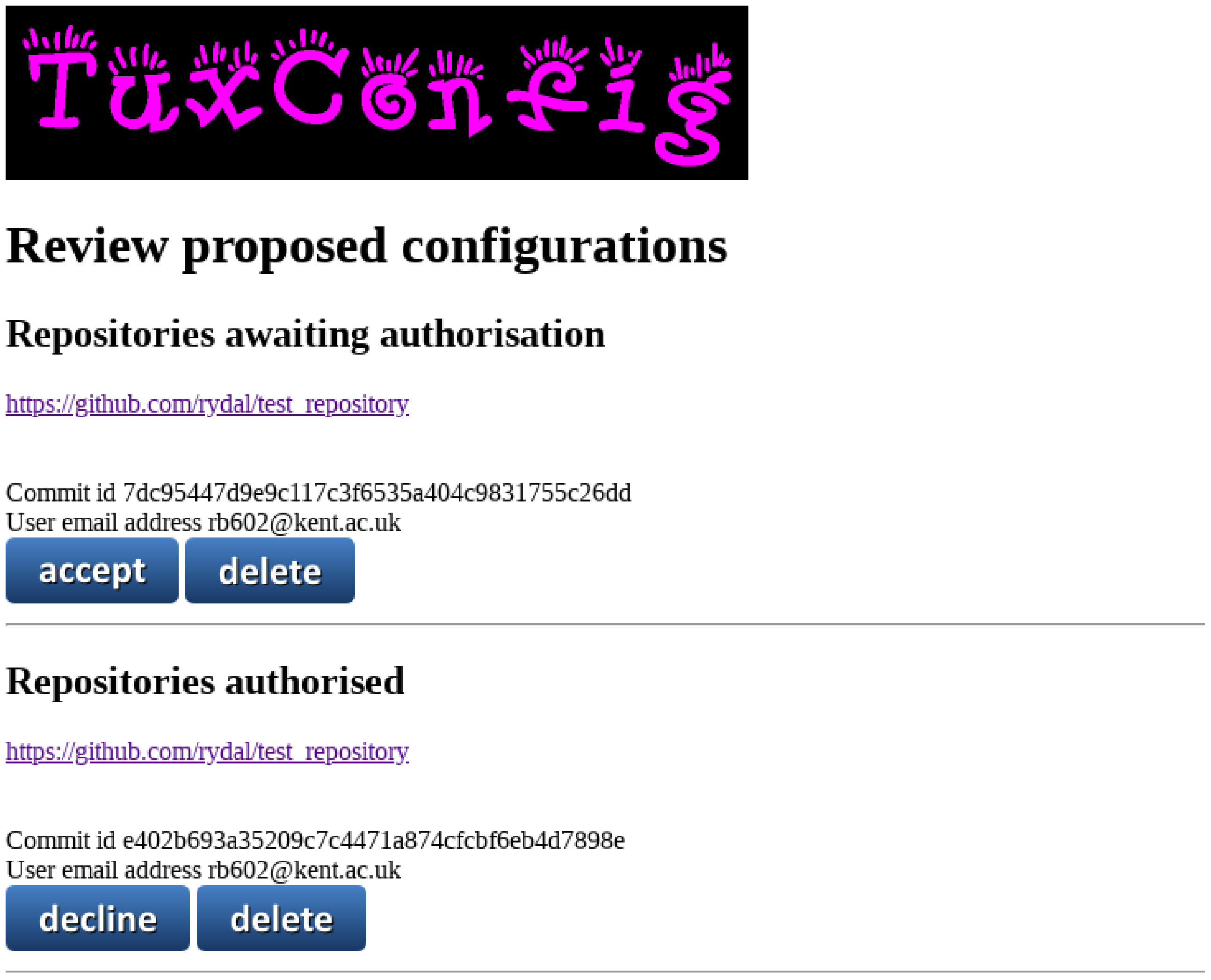
Details of how to contribute at This page

For users to Download:

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 developer’s public repositories.

The login with GitHub link derives the information from GitHub to ascertain the developer’s details which is parsed from a received JSON array using the “public_repo” scope. The “public_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 tuxconfig 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.

The vet configurations page Is below:



Review proposed configurations

Repositories awaiting authorisation

https://github.com/rydal/test_repository

Commit id 7dc95447d9e9c117c3f6535a404c9831755c26dd
User email address rb602@kent.ac.uk

Repositories authorised

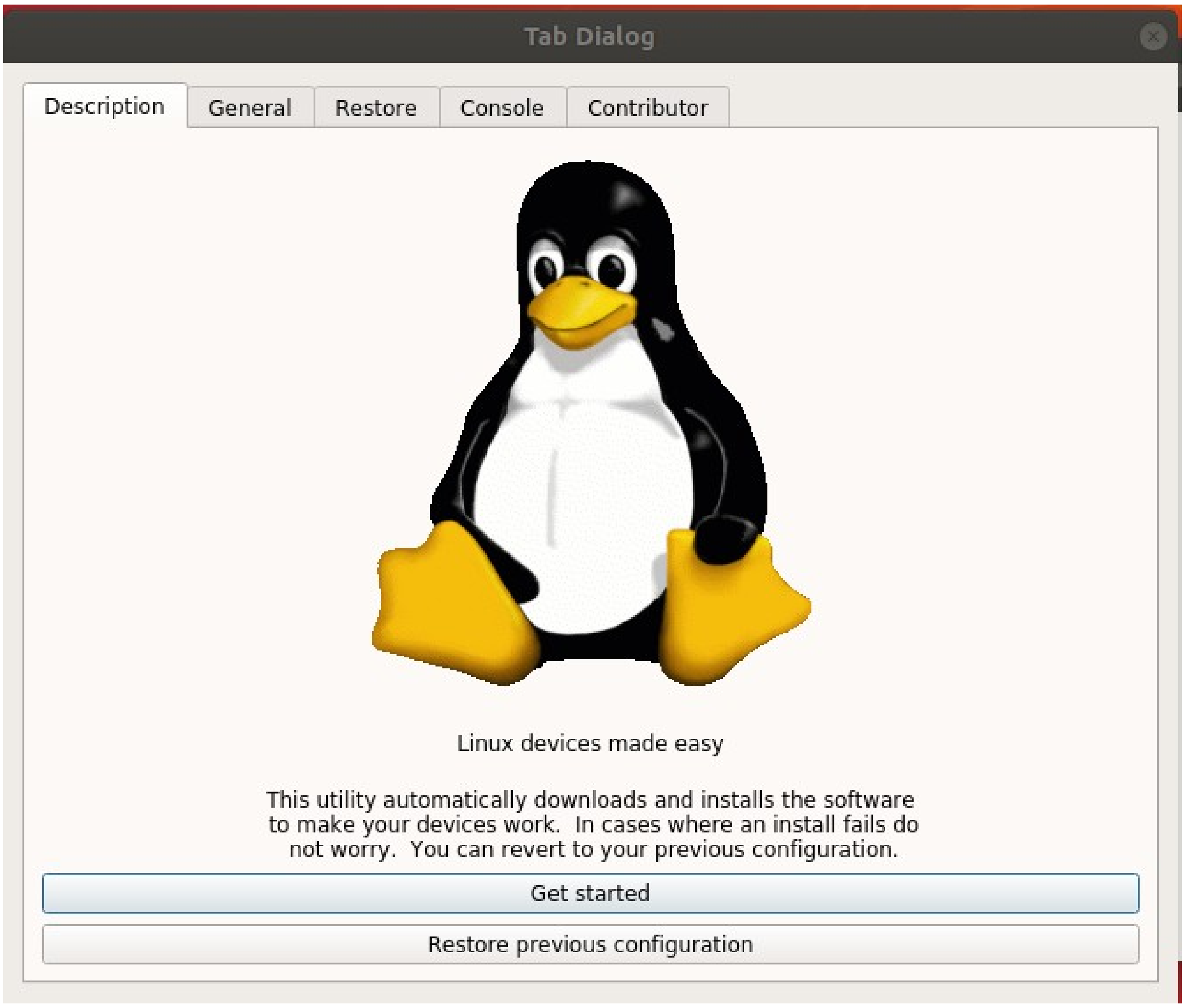
https://github.com/rydal/test_repository

Commit id e402b693a35209c7c4471a874cfcbf6eb4d7898e
User email address rb602@kent.ac.uk

This page allows an administrator logged into localhost to approve, revoke or delete a submitted repository submitted by a developer, accessed via localhost on port 8080.

GUI:

Initially the system was coded using multiple windows for notifications. This was hard to navigate, involving an application tab, running console and test program with a success or failure dialogue 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.

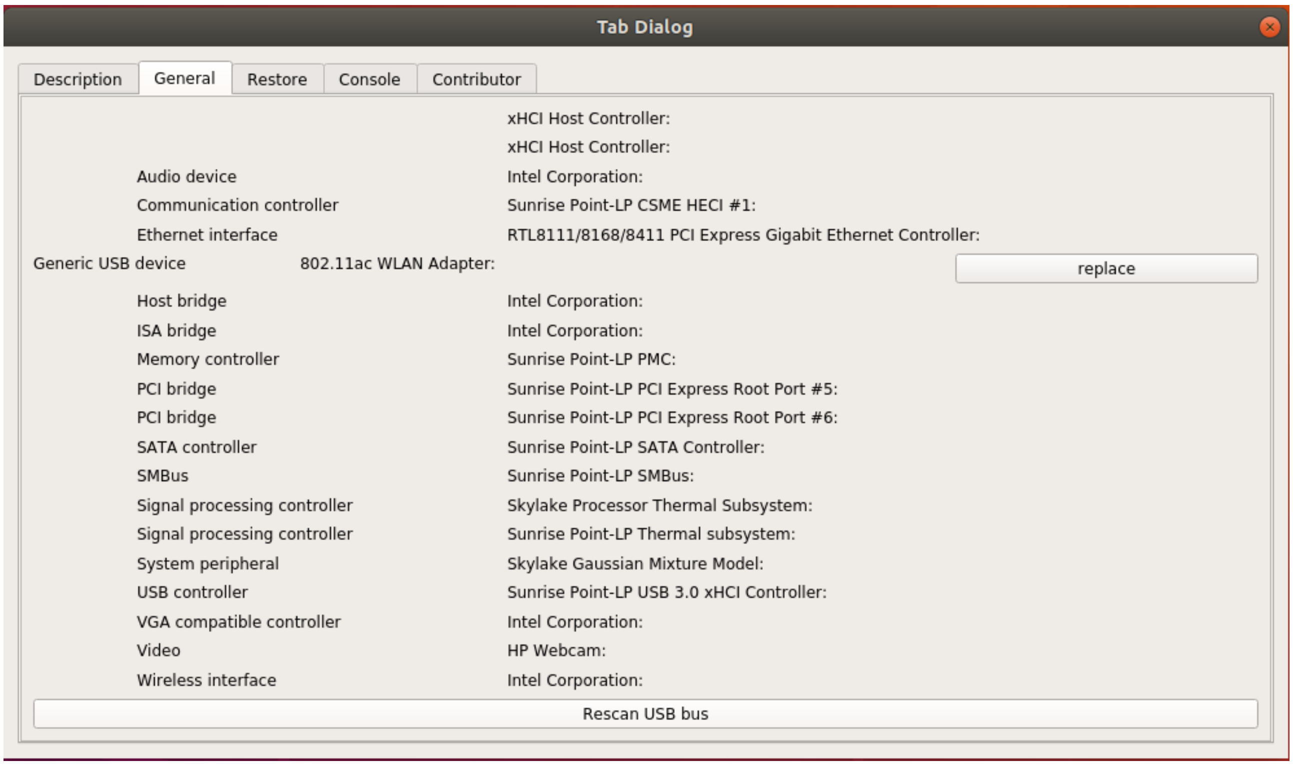


Linux devices made easy

This utility automatically downloads and installs the software to make your devices work. In cases where an install fails do not worry. You can revert to your previous configuration.

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

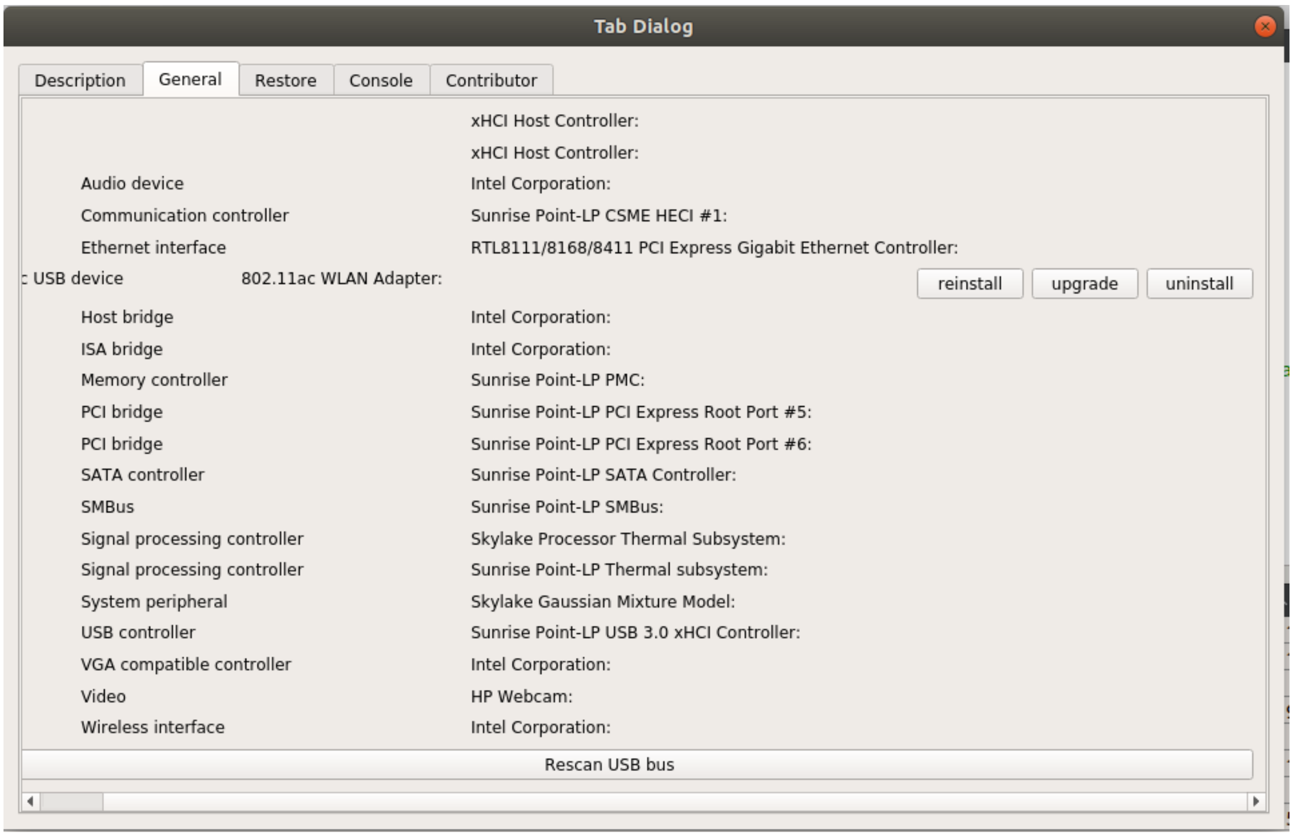
General tab is below:



	xHCI Host Controller:
	xHCI Host Controller:
Audio device	Intel Corporation:
Communication controller	Sunrise Point-LP CSME HECI #1:
Ethernet interface	RTL8111/8168/8411 PCI Express Gigabit Ethernet Controller:
Generic USB device	802.11ac WLAN Adapter:
Host bridge	Intel Corporation:
ISA bridge	Intel Corporation:
Memory controller	Sunrise Point-LP PMC:
PCI bridge	Sunrise Point-LP PCI Express Root Port #5:
PCI bridge	Sunrise Point-LP PCI Express Root Port #6:
SATA controller	Sunrise Point-LP SATA Controller:
SMBus	Sunrise Point-LP SMBus:
Signal processing controller	Skylake Processor Thermal Subsystem:
Signal processing controller	Sunrise Point-LP Thermal subsystem:
System peripheral	Skylake Gaussian Mixture Model:
USB controller	Sunrise Point-LP USB 3.0 xHCI Controller:
VGA compatible controller	Intel Corporation:
Video	HP Webcam:
Wireless interface	Intel Corporation:

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

Once a device has been installed:



	xHCI Host Controller:
	xHCI Host Controller:
Audio device	Intel Corporation:
Communication controller	Sunrise Point-LP CSME HECI #1:
Ethernet interface	RTL8111/8168/8411 PCI Express Gigabit Ethernet Controller:
USB device	802.11ac WLAN Adapter:
Host bridge	Intel Corporation:
ISA bridge	Intel Corporation:
Memory controller	Sunrise Point-LP PMC:
PCI bridge	Sunrise Point-LP PCI Express Root Port #5:
PCI bridge	Sunrise Point-LP PCI Express Root Port #6:
SATA controller	Sunrise Point-LP SATA Controller:
SMBus	Sunrise Point-LP SMBus:
Signal processing controller	Skylake Processor Thermal Subsystem:
Signal processing controller	Sunrise Point-LP Thermal subsystem:
System peripheral	Skylake Gaussian Mixture Model:
USB controller	Sunrise Point-LP USB 3.0 xHCI Controller:
VGA compatible controller	Intel Corporation:
Video	HP Webcam:
Wireless interface	Intel Corporation:

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.

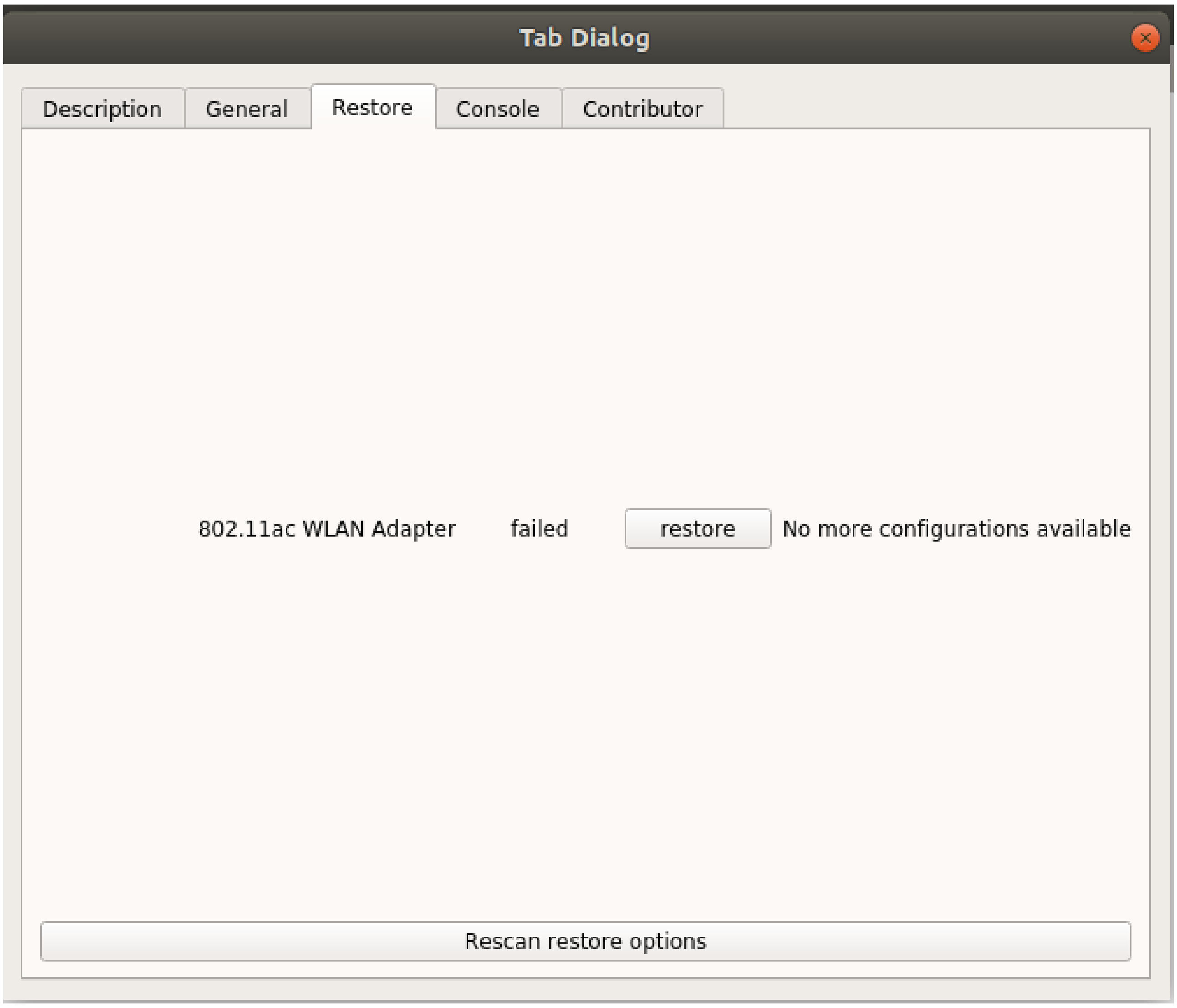
Upgrade option: when the device is installed a log of the device’s vote count is kept in the history file /var/lib/tuxconfig/history. If the database has a higher rated install for a particular device id the option to upgrade the device is shown. As the success of a device is upvotes – downvotes this feature is not time critical.

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

Reinstall button:

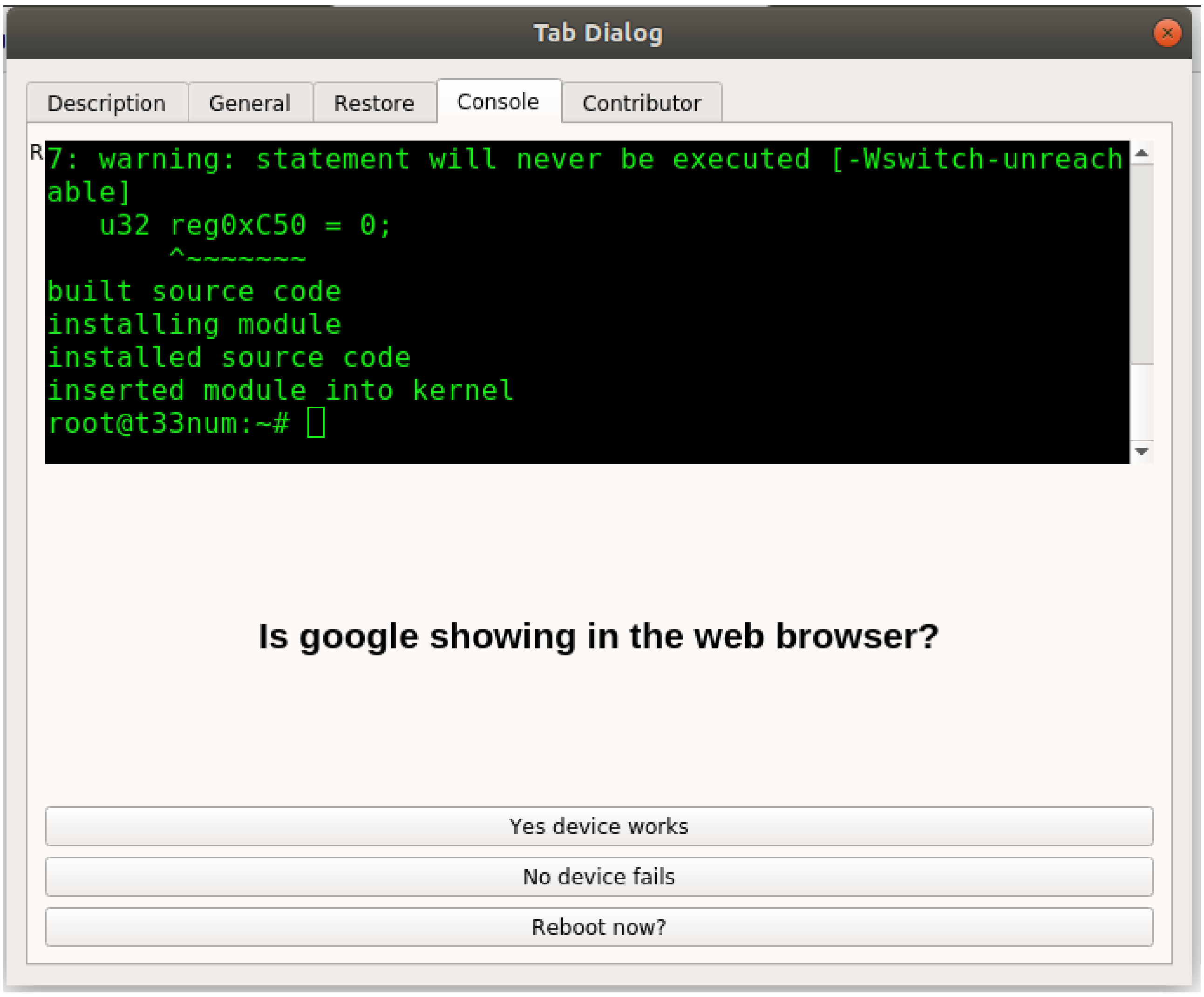
This downloads and installs the device drivers again.

The restore tab is below:


802.11ac WLAN Adapter failed restore No more configurations available

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.

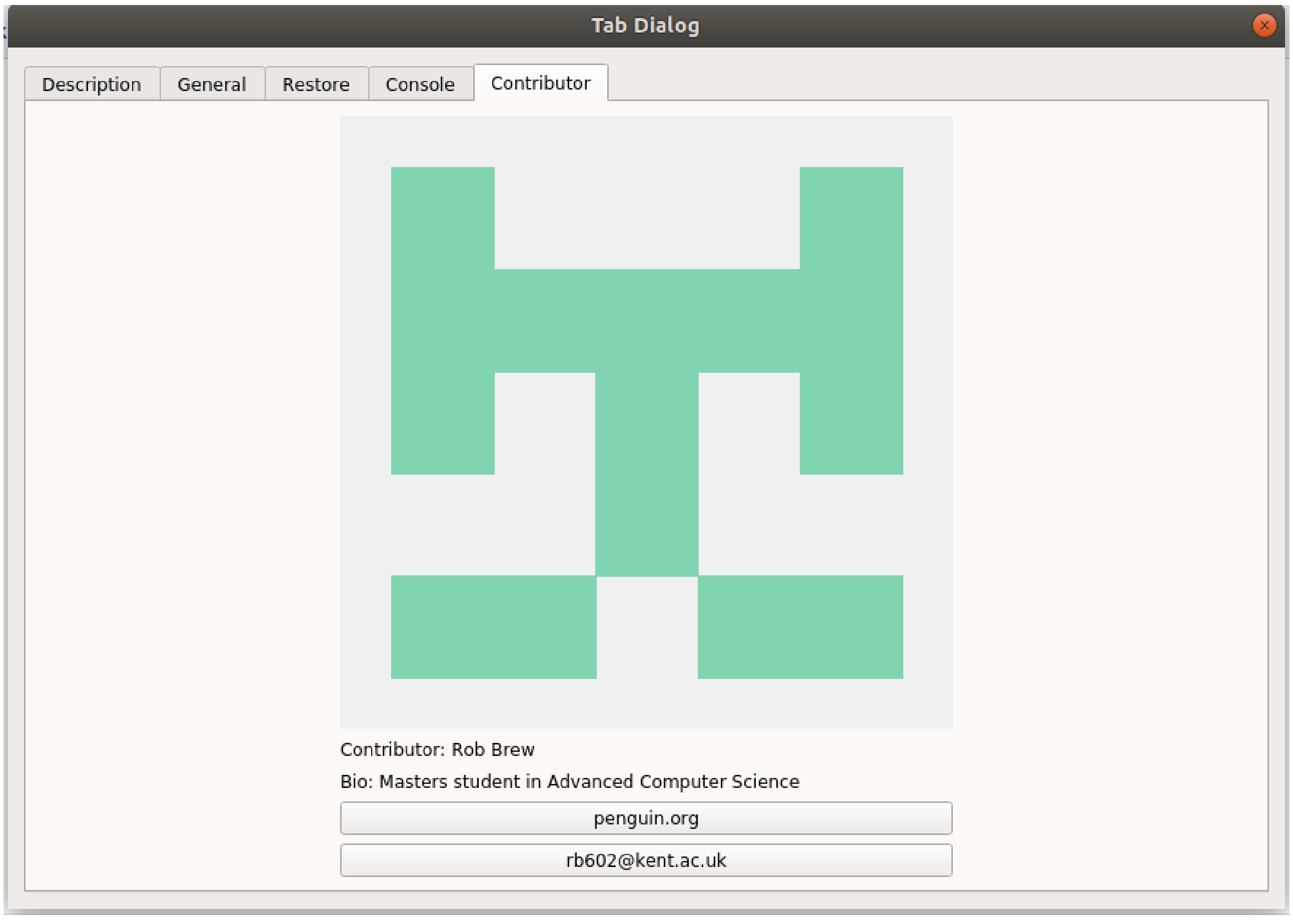
Console tab:



```
7: warning: statement will never be executed [-Wswitch-unreach
able]
   u32 reg0xC50 = 0;
       ^~~~~~~~
built source code
installing module
installed source code
inserted module into kernel
root@t33num:~#
```

Is google showing in the web browser?

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 back-end 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. Where the install has succeeded the contributor tab is shown:



Contributor: Rob Brew

Bio: Masters student in Advanced Computer Science

penguin.org

rb602@kent.ac.uk

This displays the details of the contributing user. 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.

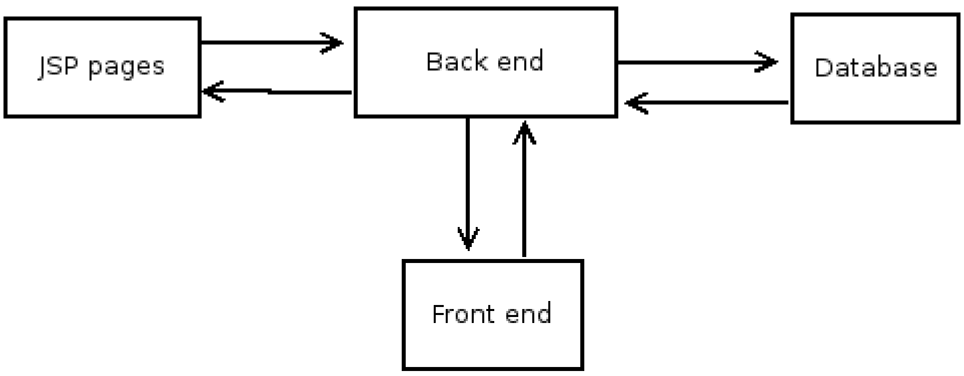
RestoreCmd:

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. The "tuxconfig_cmd" script explains why the user is entering a password for the program and how to bypass this using Ctrl-C. On voting for the success or failure of an install this entry into the start up process of a machine is removed.

The restore feature of the program 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 start-up process.

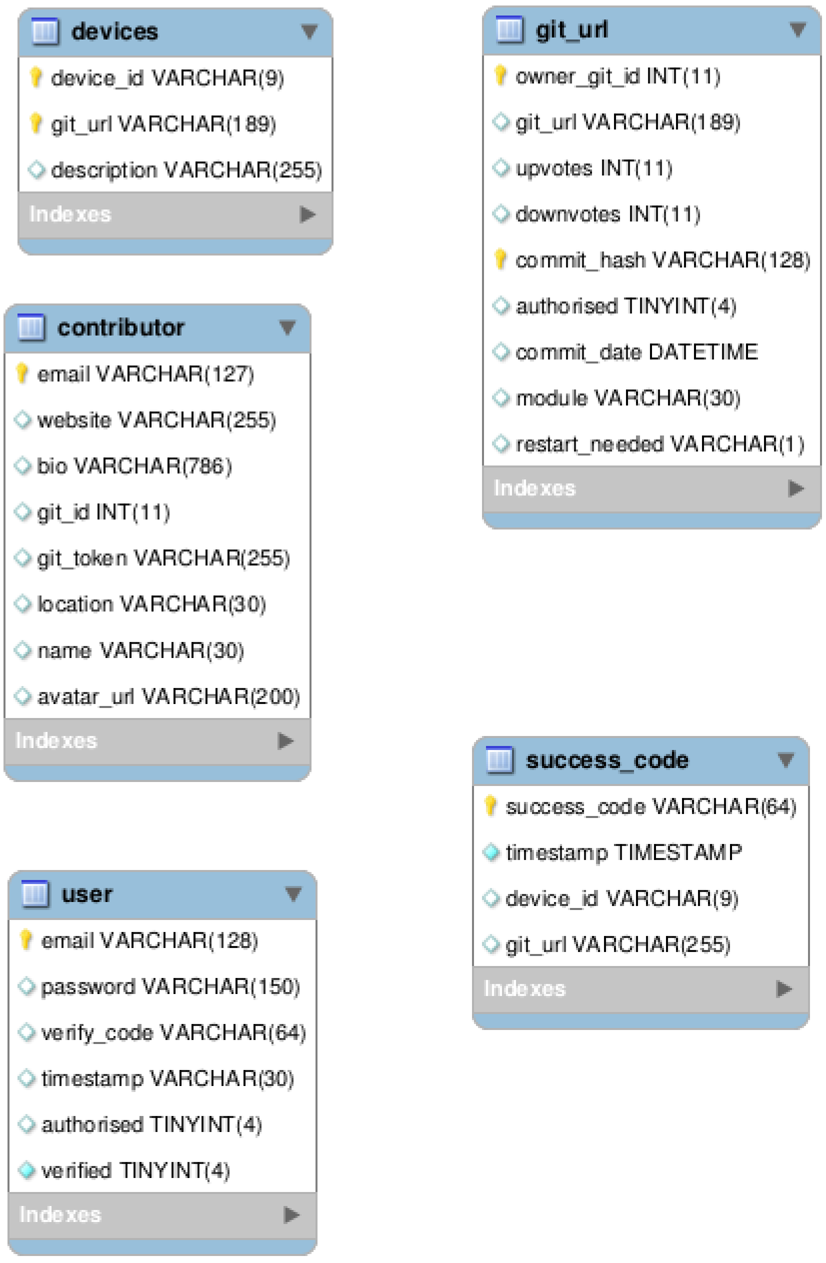
System architecture:

The system comprises four major components, as illustrated below:



Database:

My initial database schema is attached below.



devices
device_id VARCHAR(9)
git_url VARCHAR(189)
description VARCHAR(255)
Indexes

git_url
owner_git_id INT(11)
git_url VARCHAR(189)
upvotes INT(11)
downvotes INT(11)
commit_hash VARCHAR(128)
authorised TINYINT(4)
commit_date DATETIME
module VARCHAR(30)
restart_needed VARCHAR(1)
Indexes

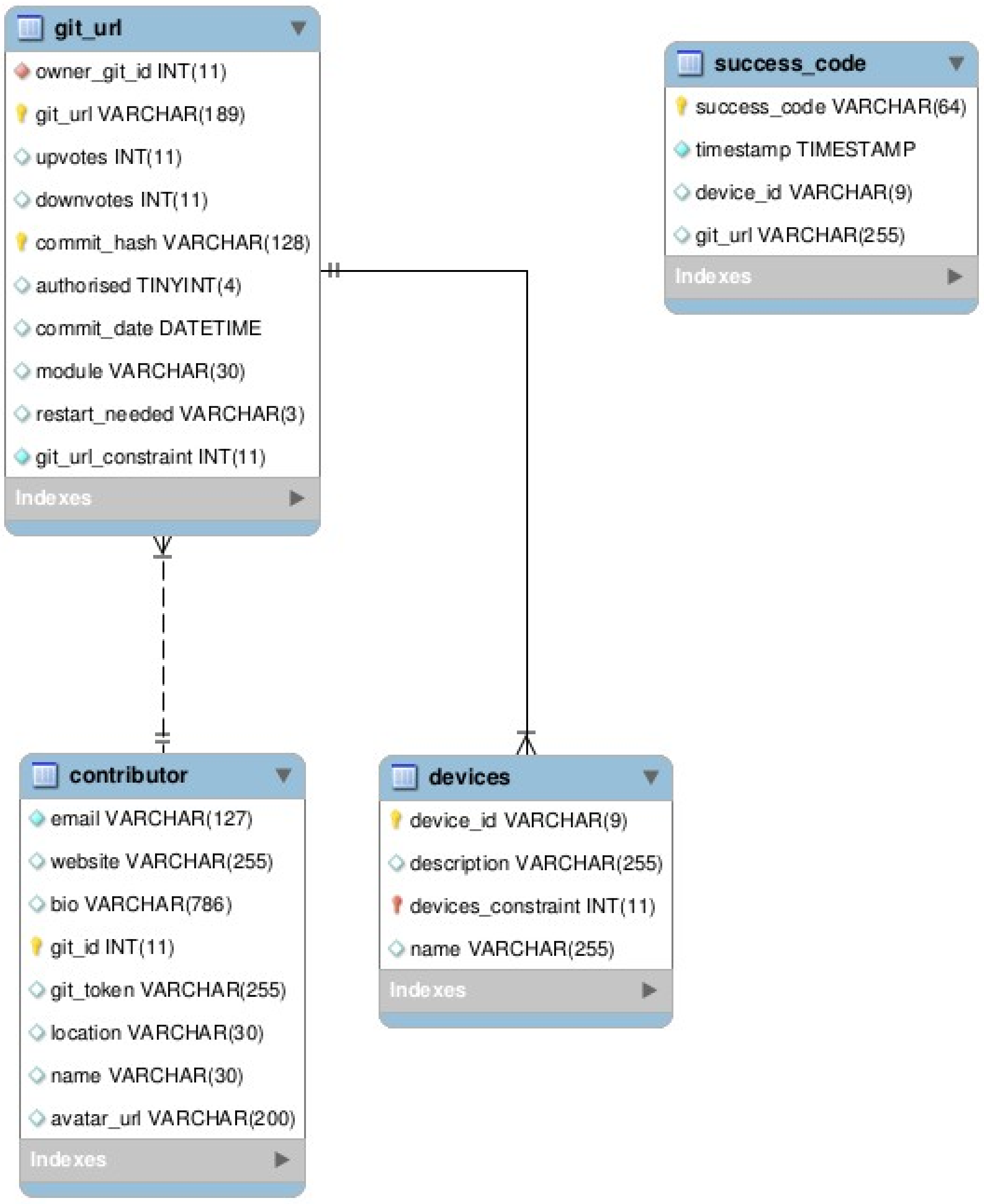
contributor
email VARCHAR(127)
website VARCHAR(255)
bio VARCHAR(786)
git_id INT(11)
git_token VARCHAR(255)
location VARCHAR(30)
name VARCHAR(30)
avatar_url VARCHAR(200)
Indexes

success_code
success_code VARCHAR(64)
timestamp TIMESTAMP
device_id VARCHAR(9)
git_url VARCHAR(255)
Indexes

user
email VARCHAR(128)
password VARCHAR(150)
verify_code VARCHAR(64)
timestamp VARCHAR(30)
authorised TINYINT(4)
verified TINYINT(4)
Indexes

Database normalisation:

This activity has been left 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:



Two foreign key constraints exist, the devices to contributor key would Ideally be the composite key of commit_hash and git_url referenced by the git_url table. However as this is part of the primary key for the devices table a unique index has been used instead.

The git_url table is 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 by 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, or a submission of the error log to the server for positing on the issue tracker at GitHub.

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.

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 database 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 for each device by device id. The restore tab checks the git_url of the next available device configuration. If this is “null” the option to try the next best configuration is not enabled. Otherwise the next best configuration is offered to the user. This option exists by the “try next install” button on the restore tab. If no devices are available the message “no more configurations available” appears instead.

This request to the database for the next best available configuration is only made by the restore tab. The general tab will always look for the best available configuration allowing it to upgrade a device’s configuration where a better one exists.

Upgrade Mode:

Using the “update” flag the program launches if updates are available for any device, otherwise it exits. This could be used as part of a cron script to run each week to update installed devices.

Device listing:

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. This has been made available at https://linuxconf.feedthepenguin.org/hehe/GetDevicesInfo.jsp

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. Git needs to be installed as well for the program to download information from the GitHub website. Build-essential is needed to install make and other building tools. This is not implicitly obvious from the code documentation.

How it could be improved:

In order to ensure that the vote counts cannot be manipulated the front-end program can sign the device id’s with a cryptographic hash using a public key. This signed device id can then be interpreted by the server to ensure that voting the device configurations has been implemented by the program itself, and not a rouge user. The client program can also use the serial number of the device being queried to ensure that each vote can only be made once per device. This will limit the submission of false positives but not end it entirely.. This would also require a check of the hash of the “lshw” program to ensure it hasn’t been altered to produce false devices with false serial numbers. The success code as mentioned previously will be used to ensure that when a device install is marked as working on reboot the works / failed vote will be relevant to the install being attempted before the reboot.

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 page using their oauth token an alternative should be used, maybe pastebin or github.com’s gist for error reporting.

As the software is downloaded using a standardised format, it shouldn’t be too hard to convert the downloaded drivers into a DKMS format kernel module .The advantage of doing this would be to ensure that the kernel modules get rebuilt when the kernel is rebuilt.

Where to from here:

It will 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. I might be able to continue the project and moderate the contributions to this system using git pull requests (submitted improvements to the code), maintaining the project myself, or pass it to the Linux community. Alternatively I could implement the above changes myself. For now. “real artists ship” to quote Steve Jobs.

Conclusion:

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

I’ve found that people are surprised at how easy Linux is to use, but not configure. This proof of concept demonstrates that there may be easier ways of configuring Linux than using the command line. I hope that, with improvement to the code from the Linux community, this project will solve one of the biggest problems of Linux adoption, that of how to install device drivers easily in Linux.

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.

The Linux kernel maintainers have / will be informed of this project, and I hope the Linux community will adopt this system 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.

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.

Bibliography:

Template for this document:

https://www.cms.gov/Research-Statistics-Data-and-Systems/CMS-Information-Technology/XLC/Downloads/HighLvlTechDesign.docx

Output of command in GetOS class:

https://stackoverflow.com/questions/478898/how-to-execute-a-command-and-get-output-of-command-within-c-using-posix/478960

HTTPDownloader in code:

https://techoverflow.net/2013/03/15/simple-c-http-download-using-libcurl-easy-api/

CheckConnection class write to /dev/null adapted from:

https://curl.haxx.se/mail/lib-2016-03/att-0223/CurlPreTransferExample.cpp

Heavy use of Stack Overflow for problems coding:

www.stackoverflow.com

QT library reference at:

http://doc.qt.io/qt-5/reference-overview.html