Image Based Upgrades with Debian

Marc Leeman

August 9, 2023

Abstract

Debian is a flexible and powerful Linux based operating system. It has a large repository of readily available and well maintained software on one hand and it has a wide range of supported architectures to run on the other hand. As a result, the operating system is used on many development systems and servers and is known for its long running support and stability.

One of the central components of this success is the software deployment and upgrade tool: `apt-get` and `dpkg`.

These tools use pre-compiled software (in `deb` packages) to upgrade the different systems in a modular way from multiple storage locations and network resources. This results in systems that are up to date with the latest released software and security fixes; but it can also lead to systems that diverge from one another since their final software configuration depends on the time the upgrade has been performed. For device software, this is not desirable and re-initalising the devices with an re-install to restore the baseline is not practical, time and resource intensive.

The rest of the document will describe how Debian can be used to counter this apparent contradiction by using image based upgrades, while keeping the strengths of incremental and modular upgrades when needed.

Contents

1 Introduction **3**

2 Related Work **3**

3 Debian Based Setup **5**
3.1 Controller Environment 5
3.2 Debian as Operating System 6
3.3 Image Based Upgrades 8
3.3.1 Installer 8
3.3.2 Upgrade Artifact 10
3.3.3 A/B deb Delivery 10
3.3.4 What about Read/Write Access? 11
3.4 Package Upgrades on Running Images 13
3.4.1 Non-Persistent Package Install and Upgrade 13
3.4.2 Persistent Package installs and Upgrades 14

4 Summary **15**

1 Introduction

One of the many strengths of Debian is the availability of wide range of well tested libraries and software and support for CPU architectures. The latest Debian release supports over 64,000 of these *packages* and supports no less than 9 different CPU architectures [4].

The use of Debian for other than trivial software stacks cuts down development cost and lets the team focus on the added value instead of integrating software in yet another embedded and highly customised build environment for a proof of concept (PoC) and production software.

For both PoC and development, the ability to select and pick prepared software components is paramount to avoid time consuming cycles of compilation and integration. Also, the availability to run software on the development stations instead of on the (often slower) target devices is also an important win.

The final goal of the development cycle must be a well defined software stack that is deployed on a large number of devices. These devices themselves are not identical and have some variety in their hardware configurations, both peripherals can change as the instruction set architecture of the processor itself. As a whole, the afore mentioned software should abstract these changes and operate, within limits, the same. In turn, this contradicts the flexibility of the development.

In the following, a method to reconciles these two ends of the deployment flexibility will be explained.

2 Related Work

One of the identifying features of Linux systems is that they are very flexible to use. The fact that most of the software is open source has lead to a large number of variants that range from a general purpose design to an OS that is tweaked towards a very dedicated device; from commercial to free systems. Listing all these variants would be near to impossible. As a result, a selection of relevant examples is used.

One of the most open source and device minded solutions that has a very dedicated following is BuildRoot [10].

BuildRoot is highly configurable and is centered around a set of Makefiles that pull in software from a network location and compiles the full system from a predefined configuration, kernel, core libraries and software. The result is a set of images that are minimal and can be written to the device. Since BuildRoot is focussed for embedded devices, these images are typically small and all non-essentials are removed: documentation and licence files are missing from these images. Being make based, it does try to re-use compilation as long as the workspace is kept. The downside is that the images are (in general) pretty fixed and limited after the configuration has been determined. While it does offer the fixed configuration images, it lacks the flexibility for easy run-time adjustment and cycle time between development and deployment should not

be underestimated. It does have a wide range of of integrated software that can be used (depending on the particular configuration used) with more or less features, but the support is no where near what one would expect from a major distribution. Furthermore, the integration of upstream software releases are often delayed. The nature of BuildRoot also puts a strong boundary between the target and the development environment.

Another significant player for Linux device software is The Yocto Project [13]. Where BuildRoot grew from open source initiatives that offered an alternative for MontaVista [14], the Yocto Project was launched by the Linux Foundation [5] in cooperation with important industry players. Yocto is known to build images for different architectures and it supports packages to configure an already installed system as well. However, it does have strong support for System on Chip (SoC) and is often the system of choice when contacting the SoC vendor. Experience shows that, depending on the subsystem, it has a lower delay when integrating new upstream releases. Most of the use-cases seem to be centered on a make → build → deploy cycle of the images, not unlike the one described for BuildRoot.

Both of these allow for some kind of network based image upgrade while incremental changes can be slow. They also have a strong boundary between the development environment and the target. This does not always benefit the quality of the software since many engineers have the tendency to write for the device itself and break functionality when any change is presented to the environment.

On the other side of the spectrum are the general purpose Linux OSes, like RedHat [15], SuSE [16], . . . and Debian [3] with its derivatives (e.g. Ubuntu [2]). The design of these systems is centered around installing it once and maintaining it by upgrading it incrementally: every piece of software can be upgraded at any time and the dependencies between the software packages themselves are minimal. These general purpose Linux OSes are traditionally designed and tweaked for server and desktop use.

By their nature, these distributions have access to a very broad range of software (open source as well as closed), have a focus on long term maintenance and support and can run on different architectures. Most software is pre-compiled and is easy to install on a running system, the learning curve is kept as low as possible. However, in order to re-baseline the system, it is often expected that the device is re-installed. The controllers, where the software is running on, are built in in vehicles and access to these is controllers is challenging at best (hidden behind panels). This becomes even more of a problem if the vehicles are moving around during operation.

One important difference with these general purpose distributions is that they normally do not come with cross compilers but depend on native compilation *for packaging*[1]. Some time ago, this meant that this needed to be compiled on hardware with that instruction set. Luckily, these distributions have multi-architecture support and running e.g. arm code on an x86 machine is all but

[1]Cross-compilers are available by default on Debian for most major architectures.

transparent.

There are many more variants, each with their strengths and weaknesses. Giving an exhaustive summary for all is impossible and far outside of the scope of this work.

3 Debian Based Setup

This section will explain how the ease of use of a general purpose operating system as is Debian, is reconciled with the standardisation of the software running on a series of controllers.

3.1 Controller Environment

The environment where the Debian OSes are used are in the public transportation market and the software is running in a range of x86 hardware (both 32 as 64 bit). The controller systems serve to control a range of embedded devices, is used to stream audio/video on the network (and decode and render it), serves as a network entry point to dedicated networks and collects information and logging. The software can be configured in any of these configurations offering a selection of the functionality. At the time of writing, there are well over 100 different configurations and software combinations that are actively maintained. Finally, these controllers are used in a world-wide market.

A number of important requirements can be derived from the above description:

- There are multiple architectures on the controllers: the older devices are 32-bit, while the newer ones (to replace end-of-life devices) are 64-bit. While the 64-bit systems can be run in 32-bit mode, a lot of the functionality requires running newer vectorized code (eg. AVX2, AVX512) which is not available in older 32-bit CPUs (e.g. encoding and decoding) and running everything in 32-bit mode leaves have of the computing power unused is not cost efficient.
- New controllers are selected for additional performance. This contradicts the possibility to run the systems in 32-bit mode for compatibility purposes. As a result, the support for multiple architectures is mandatory. Furthermore, the software should be available on all configurations; abstracting the software as much from the underlying hardware as possible.
- Whenever possible, decoding and encoding must use the available hardware acceleration blocks: networked video streams are at least 1080p, with 2160p gaining more traction as more advanced codecs are available on the SoC of the network sources (e.g. cameras). Decoding these in software does not only put a high strain on the memory and CPU, it will often lead to higher latency which is not acceptable for monitoring purposes.

- in an x86 environment, End-of-Life (EoL) of components is often around the corner and the easy replacement of these should not require invasive changes in software.
- Matching the above, since the controllers are used in a world wide market, some connectivity components can change between different markets. A typical example is e.g. the 4G/5G connectivity modem between North American and European markets.
- The software often requires customer specific changes and functionality. To this end a standardised software stack that has access to readily available libraries and programs is a huge benefit. Software is written in a range of technologies, including C/C++, Java, Python, Perl, ... and libraries that provide an interesting functionality are often offered or available in some technology, the in-house software needs to interact with all of these. This should not limit the ease of integration.

This list defines all the boundaries to use a general purpose operating system like Debian, while the deployment would traditionally be reserved for more embedded approaches.

3.2 Debian as Operating System

For a number of reasons, Debian was selected as the OS to use, reasons are many and include the availability of software and development tools (which has always been a strong point of Debian) and the long term stability and regular release cycle.

For many use-cases, the availability of pre-compiled media decoding frameworks, Python modules or Java libraries makes for very efficient development. Also once the system is installed, there is little difference for the software to run on the different architectures: when developed for 64-bit (e.g. on the workstation of the developer); code that is written properly will run on 32-bit systems once it has passed the build infrastructure.

The software that ends up on the target systems can be divided in a number of scenarios:

- Software that is being used is packaged upstream and is used on the system in a supportive function or does not need any changes at all. In this case, the package can be used as is and is pulled in from the latest Debian stable via an aptly [9] maintained mirror. The mirror contains not only the packages for the relevant architectures, it also contains the code snapshots that were used to create those packages. By mirroring the source **and** the binary packages, software that is being used in systems can always be patched: the code that was needed to create the binaries is present.
- The software needed is in all intents and purposes the same as upstream (and is packaged in Debian stable). Occasionally, extra functionality is

needed, a bug has been identified and solved or a dependency has changed. In that case, the upstream approach is followed again:

  - The upstream git packaging repository is identified (this is in most cases different than the upstream source code git location) and a patch is added in `debian/patches/` or a another change is done in the `debian/` directory (quilt format in `debian/source/format`).
  - The package is then built in a clean chroot that mimics the final target (e.g. Debian 12) so that all dependencies are resolved cleanly and the system is not polluted by the local installation of the developer or even from the build infrastructure. There are a number of tools that support this (e.g `cowbuilder`); but `mini-buildd` provides a more general and complete solution [7]. Next to building for the target OS, `mini-buildd` builds and provides consistent binaries for the different CPU architectures the software needs to run on (see Figure 1).

- The software that is needed is newer than the target distribution that is being used (e.g. GStreamer 1.22.4 in Debian 11). In this case, a very similar approach as the previous is used and the code is back ported and sent to `mini-buildd`. In Debian, this links to `debian-backports`. By doing this, we can be certain that the code compiles into the target environment (e.g C++20 features that are not supported), we can also assure that the dependencies are correct during build; and as a result, the correct run-time dependencies are generated.

- The software needed is available upstream, but has not been packaged yet. In this case, the aim should be to use the Debian best-practices approach and integrate the code into git with git-buildpackage and pristine-tar (`gbp import-orig --uscan --pristine-tar`). By doing this, the work can be contributed to Debian and/or picked up by someone that contributes it. Even if none of this is the case, the gbp structure is a proven way to maintain it for packaging and keep code in sync with upstream as they release new snapshots.

- The software that needs to be deployed is own software. In that case, the software is packaged as a native package, sent to the `mini-buildd` instance and integrated into OS.

By following these steps, there is enough room to assure that unattended upgrades can work fine and a package based upgrade is an option. The downside of this is practical rather than technical: customers (be it end-users or e.g the Q.A. department) are interested in the combination of the software that offers the functionality and wants to pin a version and timeline on that. The piecemeal upgrade of the system provides a granularity where developers of the relevant projects are interested in, but does not offer the simplicity of the combination of the software that offers an end-functionality.

Figure 1: Screenshot of `mini-buildd` web interface. The screenshot shows that source snapshots are built and released for `amd64` and `i386` architectures, as well as for Debian 11 as Debian 12 systems

3.3 Image Based Upgrades

In order to offer the simplicity of combined software upgraded on a device per device basis, image based upgrades are a proven technique that is used anywhere from embedded controllers to phones, television sets and other end-user electronics.

This section will explain the setup that is used to create an image based upgrade while maintaining the possibility to do package based upgrades.

3.3.1 Installer

As the previous text already hints to, the goal of the overall system and Debian based OS is to stick as close to Debian as possible. Not only does this minimise development and integration efforts, it keeps the threshold for new users low: the system behaves as a standard Debian system and all the files (configuration, programs, libraries) are in the locations the user expects them to be.

This choice entails that the work-flow should mimic the accepted Debian development flow as closely as possible: as with users, developers will find a plethora of available documentation. Because of this, the system starts with the default Debian installer [8] (debian-netinst). The advantage is that it is well known and supports a broad variety of hardware as well as both legacy BIOS boot and EFI boot. Especially the latter is important where hardware is registered to the EFI system before it can be used as a boot device (every device has a unique identifier, so moving images or disks across devices is not as straight forward as it might seem).

To this end, a script was written to author the `debian-netinst` iso file (see Figure 2). `apt-get` is used to download extra packages (with all dependencies) and store them in the iso file. In combination with preseed recipes, the installation can be automated [11]. When this is done properly, the installation can

be all but headless or can just require interaction when an unexpected situation occurs (missing hardware).

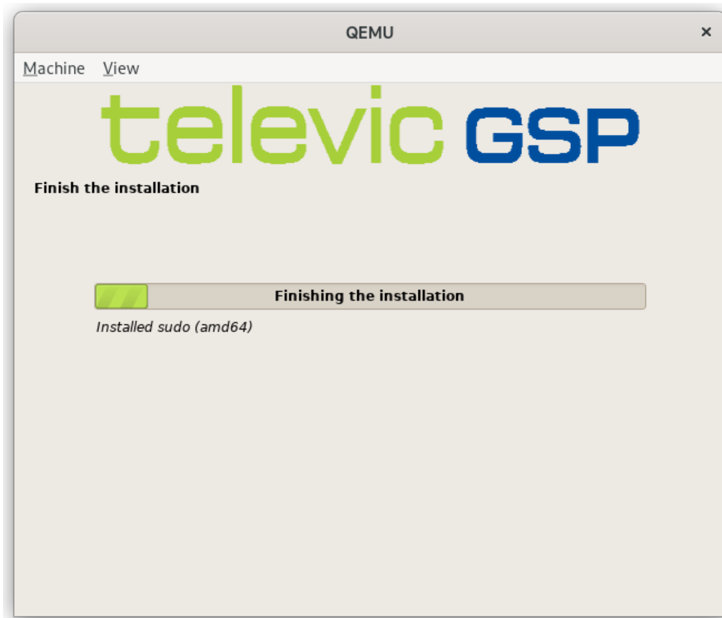

Figure 2: Screenshot of the customised Debian netinst installer. The default Debian graphics have been changed, as has the colour of the progress bar. The installer itself is still the one as provided by Debian.

For the image based upgrades, a symmetrical design is chosen: the system provides a configuration for two systems, one of them active and the other is not. Both of these systems are fully functional and contain the software that is needed during operation. The choice for this was driven by simplicity and to reduce downtime[2].

An important part of the preseed file is the partitioning. Since image based upgrades are used, a ping/pong approach is used: when system A is booted (partition), the system B partition is written during the image upgrade and the other way around. A number of approaches can be used: primary partitions are often limited to 4; more flexibility is available with extended partitions but the most flexibility is probably in using Logical Volume Management (LVM).

Per system (A/B) a `boot` and `root` partition is needed where the boot files are stored (kernel, initrd) and the full system is stored respectively. In the case

[2]The alternate approach is that the system needs be brought down in a mode from which it can be upgraded. This is often the boot loader (that has extra functionality) or a dedicated Linux system. Systems that use this range from large servers (e.g. IBM), to routers or even android devices. However, there is always more downtime involved, either by preparing the system to go in upgrade mode and having it in the upgrade mode, or by applying the upgrade during the next boot.

of EFI, an extra EFI VFAT partition is needed. The preseed partitioning will, at least, foresee (`EFI`), `boot_a`, `root_a`, `boot_b` and `root_b`, after installation, (`EFI`), `boot_a` and `root_a` will be in use while `boot_b` and `root_b` are empty[3].

3.3.2 Upgrade Artifact

The previous installer contains a bootable system after which the debootstrap [12] is used by the installer to prepare the initial system. Deboostrap can be used to create the upgrade artifact itself, but in our case, we opted to combine the test the installer with the installation of iso and use the resulting archive of the partitions to create the upgrade artifact. For virtualisation, QEMU is used and the resulting disk can be loop mounted or can be mounted with kpartx (when the disk is in compressed qcow2 format).

The iso file is fed to a virtual machine that boots from a virtual USB drive. Just as what happens during production step on the real hardware, the installer takes over, detects the hardware and partitions the target media. Internally, the installer uses debootstrap to install the packages that are defined in the preseed file on this virtual boot disk. When the Virtual machine shuts down, the resulting image file is processed and the `boot_a`/`root_a` partitions are archived.

The result of this procedure is a set of archives (`boot` and `root`). These are enough to implement the A/B upgrade system by simple scripting:

- When A is booted, format `boot_b` and `root_b`
- Extract the boot archive on `boot_b` and the root archive on `root_b`
- Swap the boot by changing the boot configuration. Depending on what is used as boot loader or boot manager, options are rewriting the grub configuration file, using a simple EFI boot manager like rEFInd or using a system that boots based on the labels of the filesystems and swapping the ext4 labels after extracting the archives (e.g. `extlinux`).

The same reasoning is in place when the B system is booted.

3.3.3 A/B deb Delivery

As a final piece, the functionality to deliver the upgrade archives needs to be determined.

When upgrading, the upgrade functionality is often embedded in the system that needs to be upgrade. In reality, there can be months or years between upgrades so the upgrade is in essence being activated by code that was written years ago. When it is certain that the upgrade system will never change or will never need bug fixes, this can be a valid approach. However, to our experience, this is rarely the case and combining the upgrade methodology with the upgrade artifacts provides room for much more retroactive changes.

[3] A production install will only fill the *A* system. After a first upgrade, both systems will be *occupied* and swapping between the two is possible.

What is known, is that the system that is being upgraded is a debian system and has `dpkg` as part of the standard tooling.

For this reason, the image archives are packed in a *fat debian* file, this allows to bundle the upgrade functionality (scripts, applications) in the upgrade artifact itself and these do not need to be part of the system that needs upgrading. Bundling it this way provides much more functionality for fixing and re-configuring the system long after it has been deployed in the field. The downside is that some caution is needed when standard tools that run on the target are used.

The split of upgrade functionality in the debian package is:

- `debian/preinst`: prepare the system for the upgrade: determine the system that is in stand by and format the relevant partitions.
- `debian/postinst`: extract the archives to the relevant locations, do some validity tests and provide feedback. If everything checks out, swap the boot configuration to use the new system. When EFI is used, a one-time boot could be used: if there is an unexpected problem; a hard reboot will fall back to the previous system.

```
admin@boxpc001-75cf1fe1f4a34d4baa83eb5bcc1ee88e: ~
⚠ Could not find /run/televic-auto-deploy/article
/dev/sda2
✋ This is a EFI system
mount: /boot: /dev/sda3 already mounted on /boot.
       dmesg(1) may have more information after failed mount system call.
⚠ Could not mount /dev/sda3 (already mounted?), continue, ...
mount: /dev/mapper/tlv00-root_b: can't find in /etc/fstab.
⚠ Could not mount /dev/mapper/tlv00-root_b (already mounted?), continue, ...
B system active
Upgrading: active: /dev/sda3 (boot), /dev/mapper/tlv00-root_b (root)
Upgrading: backup: /dev/sda2 (boot), /dev/mapper/tlv00-root_a (root)
mount: /media/boot_a: /dev/sda2 already mounted on /media/boot_a.
       dmesg(1) may have more information after failed mount system call.
⚠ Could not mount /dev/sda2, continue...
mount: /media/root_a: /dev/mapper/tlv00-root_a already mounted on /media/root_a.
       dmesg(1) may have more information after failed mount system call.
⚠ Could not mount /dev/mapper/tlv00-root_a, continue...
Found a valid televic-unified-upgrade list file
Bxpc_33.96.0666_2.00.03_bookworm-unstable-amd64-20230727-195143_boot.tar.gz: OK
Bxpc_33.96.0666_2.00.03_bookworm-unstable-amd64-20230727-195143_root.tar.gz: OK
Extracting /data/autodeploy/33.96.0666/Bxpc_33.96.0666_2.00.03_bookworm-unstable-amd64-20230727-195143_root.tar.gz to /media/root_a
Extracting /data/autodeploy/33.96.0666/Bxpc_33.96.0666_2.00.03_bookworm-unstable-amd64-20230727-195143_boot.tar.gz to /media/boot_a
⚠ Detected virtual environment, inserting ttyS0 console
⚠ Detected virtual environment, inserting ttyS0 console
Cleaning up /data/autodeploy/33.96.0666/
admin@boxpc001-75cf1fe1f4a34d4baa83eb5bcc1ee88e:~$ sudo dpkg -i /data/autodeploy/33.96.0666/Bxpc_33.96.0666_2.00.03_bookworm-unstable-amd64-20230727-195143.deb
```

Figure 3: Upgrading the system with a *fat debian package*, where dpkg is used with a patched debian package to perform the upgrade on the inactive partitions before rebooting into the new system. This functionality is hidden behind a user friendly web interface and is executed in parallel to enable bulk upgrades.

In Figure 3, the low level output is shown when an upgrade deb is installed on the system. This package is not installed in the classical way since it is running in memory (cf. infra), but the pre and post scripts are used to format and extract the system.

3.3.4 What about Read/Write Access?

One thing that puts low level embedded systems apart from general purpose software, is the use of storage: a lot of software that was designed to run and

operate on an embedded system does either not use any storage at all, or uses very specific locations to read and write temporary files. On the other hand, many general purpose software depend on the use of e.g. pid files, scratch locations to store information. When the software is well written, these locations are clearly defined and are in well defined places (e.g. `/run/`, `/tmp/`, ...).

In order to keep systems as resilient as possible (and to avoid wear and tear on e.g. NAND based storage devices) the system is best kept read-only. On the other hand, running systems need at least some kind of read/write (RW) access (e.g for storing pid or temporary files) at run-time or some space to store persistent data.

A straightforward approach would be to isolate these locations and do a *bind mount* on a memory location; a way of working that is already in use for many Linux operating systems for the `/tmp/` location. When the system reboots, the memory is cleared and all the volatile data is gone again.

Unfortunately, this does not rule out one or another subsystem accessing a different location by accident (or program error).

A solution that is used here is to use an overlay filesystem in memory. These systems have been around for some time (see. e.g. [6]). Most importantly, these systems are used for live CDs where the system boots from the CD/USB image, allows the user to look around, execute some functionality or decide to install the system.

When the functionality was made popular by `aufs`, the downside of this is that it is not part of the main Linux kernel. A successor `overlayfs` is part of the kernel and needs no extra steps to use. `overlayfs` is used on the `root` partition [1]. Overlayfs emulates a system that has full read/write access but these changes are stored in memory and are not persistent; these are not written to persistent storage. A simple reboot will restore the system to the default system before changes were made.

Overlayfs is part of the kernel and can easily be configured after installing the user land tools (see figure 4).

```
admin@boxpc001-75cf1fe1f4a34d4baa83eb5bcc1ee88e:~$ dpkg -l | grep overlay
ii  overlayroot                    0.47ubuntu1~televic12+1          all        use an overlayfs on top of a read-only
 root filesystem
admin@boxpc001-75cf1fe1f4a34d4baa83eb5bcc1ee88e:~$ cat /etc/overlayroot.conf| grep -v "^#"
overlayroot_cfgdisk="disabled"
overlayroot="tmpfs:swap=1,recurse=0"
admin@boxpc001-75cf1fe1f4a34d4baa83eb5bcc1ee88e:~$
```

Figure 4: Simple configuration of the system to use `overlayfs`. After installing `overlayroot`, a simple configuration file ensures that a memory layer is put over the root, shielding it from changes.

By tweaking the installer, we ensure that the root filesystem is mounted as read-only during boot (do the same with `boot` of course). Writes are then captured with `overlayfs` that keeps track of the changes. After a reboot these changes are lost and the system is re-initalised to the contents of the stored `RO` system.

This approach has many advantages: not only does it allow software to run obliviously of their non-persistent storage, it also allows to restore the system

to the boot/installed state, but most importantly, it allows to experiment with the system and try out changes before committing these to persistent storage. This will be tackled in the next section.

3.4 Package Upgrades on Running Images

The previous sections explained what the added value is of upgrading systems image based (as opposed to package based) and how Debian can be used to achieve this. It was also explained that all the software was packaged according to the Debian standards and sent through a builder that keeps track of the consistency between different architectures (among other things).

Certainly, there are alternative approaches where `overlayfs` can be used to merge partial file systems on top of each other, or a system is composed in a *hack and run* fashion by copying compiled files into a location before slapping it in an image. Many of these approaches would advertise the simplicity as an advantage, while ignoring the myriad of inherent problems (e.g. symbol lookup, consistency, . . .).

It is at this point where the system as designed above benefits from the consistent Debian centric packaging and comes full circle.

3.4.1 Non-Persistent Package Install and Upgrade

These systems remain fully functional Debian systems and since all the software was packaged and stored in a Debian repository (provided by e.g. `mini-buildd`), software can be installed with the normal Debian tools. At this point, a distinction needs to be made between installing software persistently (remains in between reboots), or software installs that are not persistent.

In the case of a simple non-persistent package install, the system acts as any normal Debian system does. The overlay file system ensures that the write only filesystem (mounted on `/media/root-ro` behaves as if it was read/write and captures the changes in memory. Installing a package is as simple as adding the normal deb resource to `/etc/apt/sources.list.d` and updating the package descriptions.

As an example `tcpdump` is installed Figure 5. After `apt-get`, the package is available in the root, but it is not present in `/media/root-ro`. All changes are contained in memory (`/media/root-rw`).

This allows to modify a running system to install debugging tools or now version of the software before committing these. As described before, a simple reboot will reset the system to the original version.

This way of working has one important limitation: since the changes are stored in memory; the combined use of the changes must be limited. If these are large, it will eat memory and will impact the functionality of the system. If the combined disk space of the changes exceeds the available memory; the upgrade/install will fail.

```
admin@boxpc001-75cf1fe1f4a34d4baa83eb5bcc1ee88e: ~
admin@boxpc001-75cf1fe1f4a34d4baa83eb5bcc1ee88e:~$ echo "deb http://deb.debian.org/debian/ bookworm main contrib non-free non-free-firmware" |
 sudo tee /etc/apt/sources.list.d/bookworm.list &> /dev/null
admin@boxpc001-75cf1fe1f4a34d4baa83eb5bcc1ee88e:~$ cat /etc/apt/sources.list.d/bookworm.list
deb http://deb.debian.org/debian/ bookworm main contrib non-free non-free-firmware
admin@boxpc001-75cf1fe1f4a34d4baa83eb5bcc1ee88e:~$ sudo apt update
Hit:1 http://deb.debian.org/debian bookworm InRelease
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
All packages are up to date.
admin@boxpc001-75cf1fe1f4a34d4baa83eb5bcc1ee88e:~$ which tcpdump
admin@boxpc001-75cf1fe1f4a34d4baa83eb5bcc1ee88e:~$ sudo apt install tcpdump
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following NEW packages will be installed:
  tcpdump
0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.
Need to get 467 kB of archives.
After this operation, 1364 kB of additional disk space will be used.
Get:1 http://deb.debian.org/debian bookworm/main amd64 tcpdump amd64 4.99.3-1 [467 kB]
Fetched 467 kB in 0s (3624 kB/s)
debconf: unable to initialize frontend: Dialog
debconf: (No usable dialog-like program is installed, so the dialog based frontend cannot be used. at /usr/share/perl5/Debconf/FrontEnd/Dialog
.pm line 78, <> line 1.)
debconf: falling back to frontend: Readline
Selecting previously unselected package tcpdump.
(Reading database ... 77919 files and directories currently installed.)
Preparing to unpack .../tcpdump_4.99.3-1_amd64.deb ...
Unpacking tcpdump (4.99.3-1) ...
Setting up tcpdump (4.99.3-1) ...
Processing triggers for man-db (2.11.2-2) ...
admin@boxpc001-75cf1fe1f4a34d4baa83eb5bcc1ee88e:~$ which tcpdump
/usr/bin/tcpdump
admin@boxpc001-75cf1fe1f4a34d4baa83eb5bcc1ee88e:~$ ls -al /usr/bin/tcpdump
-rwxr-xr-x 1 root root 1269784 Jan 14  2023 /usr/bin/tcpdump
admin@boxpc001-75cf1fe1f4a34d4baa83eb5bcc1ee88e:~$ ls -al /media/root-ro/usr/bin/tcpdump
ls: cannot access '/media/root-ro/usr/bin/tcpdump': No such file or directory
admin@boxpc001-75cf1fe1f4a34d4baa83eb5bcc1ee88e:~$
```

Figure 5: Simple non-persistent package install: not many changes with respect to normal Debian way of working

3.4.2 Persistent Package installs and Upgrades

The split between the active system and the protected root filesystem is something that needs to be taken into account when making persistent upgrades. In the simplest form, a user can use `chroot` to change into the `/media/root-ro` directory and perform the actions there.

```
$ sudo mount -o remount,rw /media/root-ro
$ sudo cp app_1.6.1+1+1+g0232ef5_all.deb /media/root-ro/tmp/
$ sudo chroot /media/root-ro
# dpkg -i /tmp/app_1.6.1+1+1+g0232ef5_all.deb
# rm /tmp/app_1.6.1+1+1+g0232ef5_all.deb
# exit
$ sudo mount -o remount,ro /media/root-ro
```

However, for convenience, putting this in a script that wraps `dpkg` makes it a lot easier and hides the copying, mounting and remounting for the user:

```
$ sudo tlv-dpkg-chroot -i app_1.6.1+1+1+g0232ef5_all.deb
```

Similarly, the above approach can be adjusted for apt, taking into account the chroot location. Figure 6 shows the approach where apt has been wrapped in `tlv-apt-chroot`. After adding a correct `list` file in `/media/root-ro/etc/apt/sources.list.d`[4], `tlv-apt-chroot` can be used to install a package persistently on the root location:

[4] At the moment, the system does not use DNS when operating in the chroot, so the apt repository host needs to be resolved to an IP.

```
admin@boxpc001-75cf1fe1f4a34d4baa83eb5bcc1ee88e: ~
admin@boxpc001-75cf1fe1f4a34d4baa83eb5bcc1ee88e:~$ sudo tlv-apt-chroot update
Hit:1 http://195.234.45.114/debian bookworm InRelease
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
All packages are up to date.
admin@boxpc001-75cf1fe1f4a34d4baa83eb5bcc1ee88e:~$ which tcpdump
admin@boxpc001-75cf1fe1f4a34d4baa83eb5bcc1ee88e:~$ sudo tlv-apt-chroot install tcpdump
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following NEW packages will be installed:
  tcpdump
0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.
Need to get 467 kB of archives.
After this operation, 1364 kB of additional disk space will be used.
Get:1 http://195.234.45.114/debian bookworm/main amd64 tcpdump amd64 4.99.3-1 [467 kB]
Fetched 467 kB in 0s (2974 kB/s)
debconf: unable to initialize frontend: Dialog
debconf: (No usable dialog-like program is installed, so the dialog based frontend cannot be used. at /usr/share/perl5/Debconf/FrontEnd/Dialog
.pm line 78, <> line 1.)
debconf: falling back to frontend: Readline
E: Can not write log (Is /dev/pts mounted?) - posix_openpt (2: No such file or directory)
Selecting previously unselected package tcpdump.
(Reading database ... 77919 files and directories currently installed.)
Preparing to unpack .../tcpdump_4.99.3-1_amd64.deb ...
Unpacking tcpdump (4.99.3-1) ...
Setting up tcpdump (4.99.3-1) ...
Processing triggers for man-db (2.11.2-2) ...
admin@boxpc001-75cf1fe1f4a34d4baa83eb5bcc1ee88e:~$ which tcpdump
admin@boxpc001-75cf1fe1f4a34d4baa83eb5bcc1ee88e:~$ sudo chroot /media/root-ro which tcpdump
/usr/bin/tcpdump
admin@boxpc001-75cf1fe1f4a34d4baa83eb5bcc1ee88e:~$ ls -al /media//root-ro/usr/bin/tcpdump
-rwxr-xr-x 1 root root 1269784 Jan 14  2023 /media//root-ro/usr/bin/tcpdump
admin@boxpc001-75cf1fe1f4a34d4baa83eb5bcc1ee88e:~$
admin@boxpc001-75cf1fe1f4a34d4baa83eb5bcc1ee88e:~$
```

Figure 6: By wrapping apt in a script that executes it with chroot in the remounted /media/root-ro location; the system can be upgraded incrementally and new software can be installed without needing to re-install the full updated image. When using apt without chroot, changes will get lost after a reboot.

As in the previous example with the volatile installation of a package, `tcpdump` is installed; the command is straightforward, the package (with dependencies if needed) is installed and downloaded. In this example (with `overlayfs`, the package (and functionality) is not immediately present in the running system and will require a reboot to activate the changes on the running system[5].

4 Summary

This document described the use of an image based upgrade for Debian systems at a high level. We started out by describing the advantages of package based upgrade and why image upgrades can be a valid addition to these. After deciding that the software should be packaged just as software is packaged in Debian (and other distributions), the infrastructure that has been put in place was described.

The creation of the images was tackled next. The system was installed with a modified Debian installer, this installer included the custom software packages on top of the default ones already present on the original installer. For aesthetics, some of the artwork was modified. The entire system was wrapped up with preseed files that determine the partitioning to accommodate for a symmetric A/B system and answers to common installation questions. The result is an installer that can be written on a CD or USB stick and that prepares the controllers for an initial install.

[5]Other overlay systems can make this change seen immediately (e.g. `aufs`).

In-kernel virtualisation (QEMU) was chosen to generate the upgrade images by installing the generated iso file in a virtual machine and generating the archives of the `boot` and `root` locations. The advantage of chain testing the installer to generate the `root` and `boot` upgrade artifacts was used over `debootstrap`.

Next, these 2 archives were wrapped in a *fat* debian package which allows to keep all the scripting and upgrade functionality in the upgrade artifact and not part of the original system that needs upgrading. This guarantees much more flexible maintenance. At that point, the image based upgrade of the Debian system was in place.

Finally, the coupling was made to the standard Debian packaging again: the system could still add new software from existing repositories (with both `apt` and `dpkg`). When using `dpkg` and `apt`; there was little change with a normal Debian system. The only difference was that the changes were installed in memory and will not be persistent between reboots.

When using `chroot` to wrap `apt` and `dpkg`, an approach was presented to allow upgrades and installs of extra software that *are* persistent in between upgrades.

This whole process is architecture agnostic. If the company decides to move to e.g. move to Arm or RISC-V, provided the platform uses something similar to EFI, the same build/installation process could be used, even in parallel with the existing i386/amd64 architectures for smoother migrations.

Bibliography

[1] Neil Brown. *Overlay Filesystem*. URL: `https://www.kernel.org/doc/html/latest/filesystems/overlayfs.html`. (accessed: 28.07.2023).

[2] Canonical. *Enterprise Open Source and Linux — Ubuntu*. URL: `https://www.ubuntu.com/`. (accessed: 27.07.2023).

[3] Debian. *Debian – The Universal Operating System*. URL: `https://www.debian.org/`. (accessed: 27.07.2023).

[4] Debian. *Debian 12 "bookworm" released*. URL: `https://www.debian.org/News/2023/20230610`. (accessed: 27.07.2023).

[5] The Linux Foundation. *Decentralized innovation. Built on trust.* URL: `https://www.linuxfoundation.org`. (accessed: 27.07.2023).

[6] Knoppix. *Knoppix.net, from zero to Linux in 5 minutes*. URL: `http://knoppix.net/`. (accessed: 28.07.2023).

[7] Miscellaneous. URL: `http://installiert.net/mini-buildd/`. (accessed: 28.07.2023).

[8] Miscellaneous. URL: `https://www.debian.org/CD/netinst/`. (accessed: 28.07.2023).

[9] Miscellaneous. *Aptly, Swiss army knife for Debian repository management*. URL: `https://www.aptly.info/`. (accessed: 28.07.2023).

[10] Miscellaneous. *Buildroot, Making Embedded Linux Easy*. URL: `https://buildroot.org`. (accessed: 27.07.2023).

[11] Miscellaneous. *DebianInstaller Preseed*. URL: `https://wiki.debian.org/DebianInstaller/Preseed`. (accessed: 28.07.2023).

[12] Miscellaneous. *Debootstrap*. URL: `https://wiki.debian.org/Debootstrap`. (accessed: 28.07.2023).

[13] Miscellaneous. *The Yocto Project. It's not an embedded Linux distribution, it creates a custom one for you.* URL: `https://www.yoctoproject.org/`. (accessed: 27.07.2023).

[14] Montavista. *New to Rocky Linux? Get in touch today!* URL: `https://www.mvista.com/`. (accessed: 27.07.2023).

[15] Redhat. *Red Hat - We make open source technologies for the enterprise*. URL: `https://www.redhat.com/en`. (accessed: 27.07.2023).

[16] SuSE. *Open Source Solutions for Enterprise Servers & Cloud — SUSE*. URL: `https://www.suse.com/`. (accessed: 27.07.2023).