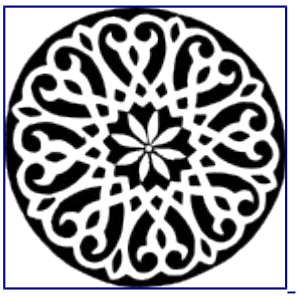

Pid Eins

Posted on Mon 01 September 2014 by Lennart Poettering.

Revisiting How We Put Together Linux Systems

In a previous blog story I discussed Factory Reset, Stateless Systems, Reproducible Systems & Verifiable Systems, I now want to take the opportunity to explain a bit where we want to take this with systemd in the longer run, and what we want to build out of it. This is going to be a longer story, so better grab a cold bottle of Club Mate before you start reading.

Traditional Linux distributions are built around packaging systems like RPM or dpkg, and an organization model where upstream developers and downstream packagers are relatively clearly separated: an upstream developer writes code, and puts it somewhere online, in a tarball. A packager than grabs it and turns it into RPMs/DEBs. The user then grabs these RPMs/DEBs and installs them locally on the system. For a variety of uses this is a fantastic scheme: users have a large selection of readily packaged software available, in mostly uniform packaging, from a single source they can trust. In this scheme the distribution vets all software it packages, and as long as the user trusts the distribution all should be good. The distribution takes the responsibility of ensuring the software is not malicious, of timely fixing security problems and helping the user if something is wrong.

Upstream Projects

However, this scheme also has a number of problems, and doesn't fit many use-cases of our software particularly well. Let's have a look at the problems of this scheme for many upstreams:

- Upstream software vendors are fully dependent on downstream distributions to package their stuff. It's the downstream distribution that decides on schedules, packaging details, and how to handle support. Often upstream vendors want much faster release cycles then the downstream distributions follow.
- Realistic testing is extremely unreliable and next to impossible. Since the end-user can run a

variety of different package versions together, and expects the software he runs to just work on any combination, the test matrix explodes. If upstream tests its version on distribution X release Y, then there's no guarantee that that's the precise combination of packages that the end user will eventually run. In fact, it is very unlikely that the end user will, since most distributions probably updated a number of libraries the package relies on by the time the package ends up being made available to the user. The fact that each package can be individually updated by the user, and each user can combine library versions, plug-ins and executables relatively freely, results in a high risk of something going wrong.

- Since there are so many different distributions in so many different versions around, if upstream tries to build and test software for them it needs to do so for a large number of distributions, which is a massive effort.
- The distributions are actually quite different in many ways. In fact, they are different in a lot of the most basic functionality. For example, the path where to put x86-64 libraries is different on Fedora and Debian derived systems..
- Developing software for a number of distributions and versions is hard: if you want to do it, you need to actually install them, each one of them, manually, and then build your software for each.
- Since most downstream distributions have strict licensing and trademark requirements (and rightly so), any kind of closed source software (or otherwise non-free) does not fit into this scheme at all.

This all together makes it really hard for many upstreams to work nicely with the current way how Linux works. Often they try to improve the situation for them, for example by bundling libraries, to make their test and build matrices smaller.

System Vendors

The *toolbox* approach of classic Linux distributions is fantastic for people who want to put together their individual system, nicely adjusted to exactly what they need. However, this is not really how many of today's Linux systems are built, installed or updated. If you build any kind of embedded device, a server system, or even user systems, you frequently do your work based on complete system images, that are linearly versioned. You build these images somewhere, and then you replicate them atomically to a larger number of systems. On these systems, you don't install or remove packages, you get a defined set of files, and besides installing or updating the system there are no ways how to change the set of tools you get.

The current Linux distributions are not particularly good at providing for this major use-case of Linux. Their strict focus on individual packages as well as package managers as end-user install and update tool is incompatible with what many system vendors want.

Users

The classic Linux distribution scheme is frequently not what end users want, either. Many users are used to app markets like Android, Windows or iOS/Mac have. Markets are a platform that doesn't package, build or maintain software like distributions do, but simply allows users to quickly find and download the software they need, with the app vendor responsible for keeping the app updated, secured, and all that on the vendor's release cycle. Users tend to be impatient. They want their software quickly, and the fine distinction between trusting a single distribution or a myriad of app developers individually is usually not important for them. The companies behind the marketplaces usually try to improve this trust problem by providing sand-boxing technologies: as a replacement for the distribution that audits, vets, builds and packages the software and thus allows users to trust it to a certain level, these vendors try to find technical solutions to ensure that the software they offer for download can't be malicious.

Existing Approaches To Fix These Problems

Now, all the issues pointed out above are not new, and there are sometimes quite successful attempts to do something about it. Ubuntu Apps, Docker, Software Collections, ChromeOS, CoreOS all fix part of this problem set, usually with a strict focus on one facet of Linux systems. For example, Ubuntu Apps focus strictly on end user (desktop) applications, and don't care about how we built/update/install the OS itself, or containers. Docker OTOH focuses on containers only, and doesn't care about end-user apps. Software Collections tries to focus on the development environments. ChromeOS focuses on the OS itself, but only for end-user devices. CoreOS also focuses on the OS, but only for server systems.

The approaches they find are usually good at specific things, and use a variety of different technologies, on different layers. However, none of these projects tried to fix this problems in a generic way, for all uses, right in the core components of the OS itself.

Linux has come to tremendous successes because its kernel is so generic: you can build supercomputers and tiny embedded devices out of it. It's time we come up with a basic, reusable scheme how to solve the problem set described above, that is equally generic.

What We Want

The systemd cabal (Kay Sievers, Harald Hoyer, Daniel Mack, Tom Gundersen, David Herrmann, and yours truly) recently met in Berlin about all these things, and tried to come up with a scheme that is somewhat simple, but tries to solve the issues generically, for all use-cases, as part of the systemd project. All that in a way that is somewhat compatible with the current scheme of distributions, to allow a slow, gradual adoption. Also, and that's something one cannot stress enough: the *toolbox* scheme of classic Linux distributions is actually a good one, and for many cases the right one. However, we need

to make sure we make distributions relevant again for *all* use-cases, not just those of highly individualized systems.

Anyway, so let's summarize what we are trying to do:

- We want an efficient way that allows vendors to package their software (regardless if just an app, or the whole OS) directly for the end user, and know the precise combination of libraries and packages it will operate with.
- We want to allow end users and administrators to install these packages on their systems, regardless which distribution they have installed on it.
- We want a unified solution that ultimately can cover updates for full systems, OS containers, end user apps, programming ABIs, and more. These updates shall be double-buffered, (at least). This is an absolute necessity if we want to prepare the ground for operating systems that manage themselves, that can update safely without administrator involvement.
- We want our images to be trustable (i.e. signed). In fact we want a fully trustable OS, with images that can be verified by a full trust chain from the firmware (EFI SecureBoot!), through the boot loader, through the kernel, and initrd. Cryptographically secure verification of the code we execute is relevant on the desktop (like ChromeOS does), but also for apps, for embedded devices and even on servers (in a post-Snowden world, in particular).

What We Propose

So much about the set of problems, and what we are trying to do. So, now, let's discuss the technical bits we came up with:

The scheme we propose is built around the variety of concepts of btrfs and Linux file system name-spacing. btrfs at this point already has a large number of features that fit neatly in our concept, and the maintainers are busy working on a couple of others we want to eventually make use of.

As first part of our proposal we make heavy use of btrfs sub-volumes and introduce a clear naming scheme for them. We name snapshots like this:

- `usr:<vendorid>:<architecture>:<version>` -- This refers to a full vendor operating system tree. It's basically a /usr tree (and no other directories), in a specific version, with everything you need to boot it up inside it. The `<vendorid>` field is replaced by some vendor identifier, maybe a scheme like `org.fedoraproject.FedoraWorkstation`. The `<architecture>` field specifies a CPU architecture the OS is designed for, for example `x86-64`. The `<version>` field specifies a specific OS version, for example `23.4`. An example sub-volume name could hence look like this: `usr:org.fedoraproject.FedoraWorkstation:x86_64:23.4`

- `root:<name>:<vendorid>:<architecture>` -- This refers to an *instance* of an operating system. Its basically a root directory, containing primarily /etc and /var (but possibly more). Sub-volumes of this type do not contain a populated /usr tree though. The `<name>` field refers to some instance name (maybe the host name of the instance). The other fields are defined as above. An example sub-volume name is `root:revolution:org.fedoraproject.FedoraWorkstation:x86_64`.
- `runtime:<vendorid>:<architecture>:<version>` -- This refers to a vendor *runtime*. A runtime here is supposed to be a set of libraries and other resources that are needed to run apps (for the concept of *apps* see below), all in a /usr tree. In this regard this is very similar to the `usr` sub-volumes explained above, however, while a `usr` sub-volume is a full OS and contains everything necessary to boot, a runtime is really only a set of libraries. You cannot boot it, but you can run apps with it. An example sub-volume name is: `runtime:org.gnome.GNOME3_20:x86_64:3.20.1`
- `framework:<vendorid>:<architecture>:<version>` -- This is very similar to a vendor runtime, as described above, it contains just a /usr tree, but goes one step further: it additionally contains all development headers, compilers and build tools, that allow developing against a specific runtime. For each runtime there should be a framework. When you develop against a specific framework in a specific architecture, then the resulting app will be compatible with the runtime of the same vendor ID and architecture. Example: `framework:org.gnome.GNOME3_20:x86_64:3.20.1`
- `app:<vendorid>:<runtime>:<architecture>:<version>` -- This encapsulates an application bundle. It contains a tree that at runtime is mounted to `/opt/<vendorid>`, and contains all the application's resources. The `<vendorid>` could be a string like `org.libreoffice.LibreOffice`, the `<runtime>` refers to one the vendor id of one specific runtime the application is built for, for example `org.gnome.GNOME3_20:3.20.1`. The `<architecture>` and `<version>` refer to the architecture the application is built for, and of course its version. Example: `app:org.libreoffice.LibreOffice:GNOME3_20:x86_64:133`
- `home:<user>:<uid>:<gid>` -- This sub-volume shall refer to the home directory of the specific user. The `<user>` field contains the user name, the `<uid>` and `<gid>` fields the numeric Unix UIDs and GIDs of the user. The idea here is that in the long run the list of sub-volumes is sufficient as a user database (but see below). Example: `home:lennart:1000:1000`.

btrfs partitions that adhere to this naming scheme should be clearly identifiable. It is our intention to introduce a new GPT partition type ID for this.

How To Use It

After we introduced this naming scheme let's see what we can build of this:

- When booting up a system we mount the root directory from one of the `root` sub-volumes, and then mount /usr from a matching `usr` sub-volume. *Matching* here means it carries the same `<vendor-id>` and `<architecture>`. Of course, by default we should pick the matching `usr` sub-volume with the newest version by default.
- When we boot up an OS container, we do exactly the same as the when we boot up a regular system: we simply combine a `usr` sub-volume with a `root` sub-volume.
- When we enumerate the system's users we simply go through the list of `home` snapshots.
- When a user authenticates and logs in we mount his home directory from his snapshot.
- When an app is run, we set up a new file system name-space, mount the `app` sub-volume to `/opt/<vendorid>/`, and the appropriate `runtime` sub-volume the app picked to `/usr`, as well as the user's `/home/$USER` to its place.
- When a developer wants to develop against a specific runtime he installs the right framework, and then temporarily transitions into a name space where `/usr`is mounted from the framework sub-volume, and `/home/$USER` from his own home directory. In this name space he then runs his build commands. He can build in multiple name spaces at the same time, if he intends to builds software for multiple runtimes or architectures at the same time.

Instantiating a new system or OS container (which is exactly the same in this scheme) just consists of creating a new appropriately named `root` sub-volume. Completely naturally you can share one vendor OS copy in one specific version with a multitude of container instances.

Everything is *double-buffered* (or actually, n-fold-buffered), because `usr`, `runtime`, `framework`, `app` sub-volumes can exist in multiple versions. Of course, by default the execution logic should always pick the newest release of each sub-volume, but it is up to the user keep multiple versions around, and possibly execute older versions, if he desires to do so. In fact, like on ChromeOS this could even be handled automatically: if a system fails to boot with a newer snapshot, the boot loader can automatically revert back to an older version of the OS.

An Example

Note that in result this allows installing not only multiple end-user applications into the same btrfs volume, but also multiple operating systems, multiple system instances, multiple runtimes, multiple frameworks. Or to spell this out in an example:

Let's say Fedora, Mageia and ArchLinux all implement this scheme, and provide ready-made end-user images. Also, the GNOME, KDE, SDL projects all define a runtime+framework to develop against. Finally, both LibreOffice and Firefox provide their stuff according to this scheme. You can now trivially install of these into the same btrfs volume:

- usr:org.fedoraproject.WorkStation:x86_64:24.7
- usr:org.fedoraproject.WorkStation:x86_64:24.8
- usr:org.fedoraproject.WorkStation:x86_64:24.9
- usr:org.fedoraproject.WorkStation:x86_64:25beta
- usr:org.mageia.Client:i386:39.3
- usr:org.mageia.Client:i386:39.4
- usr:org.mageia.Client:i386:39.6
- usr:org.archlinux.Desktop:x86_64:302.7.8
- usr:org.archlinux.Desktop:x86_64:302.7.9
- usr:org.archlinux.Desktop:x86_64:302.7.10
- root:revolution:org.fedoraproject.WorkStation:x86_64
- root:testmachine:org.fedoraproject.WorkStation:x86_64
- root:foo:org.mageia.Client:i386
- root:bar:org.archlinux.Desktop:x86_64
- runtime:org.gnome.GNOME3_20:x86_64:3.20.1
- runtime:org.gnome.GNOME3_20:x86_64:3.20.4
- runtime:org.gnome.GNOME3_20:x86_64:3.20.5
- runtime:org.gnome.GNOME3_22:x86_64:3.22.0
- runtime:org.kde.KDE5_6:x86_64:5.6.0
- framework:org.gnome.GNOME3_22:x86_64:3.22.0
- framework:org.kde.KDE5_6:x86_64:5.6.0
- app:org.libreoffice.LibreOffice:GNOME3_20:x86_64:133
- app:org.libreoffice.LibreOffice:GNOME3_22:x86_64:166
- app:org.mozilla.Firefox:GNOME3_20:x86_64:39
- app:org.mozilla.Firefox:GNOME3_20:x86_64:40
- home:lennart:1000:1000
- home:hrundivbakshi:1001:1001

In the example above, we have three vendor operating systems installed. All of them in three versions, and one even in a beta version. We have four system instances around. Two of them of Fedora, maybe one of them we usually boot from, the other we run for very specific purposes in an OS container. We also have the runtimes for two GNOME releases in multiple versions, plus one for KDE. Then, we have the development trees for one version of KDE and GNOME around, as well as two apps, that

make use of two releases of the GNOME runtime. Finally, we have the home directories of two users.

Now, with the name-spacing concepts we introduced above, we can actually relatively freely mix and match apps and OSes, or develop against specific frameworks in specific versions on any operating system. It doesn't matter if you booted your ArchLinux instance, or your Fedora one, you can execute both LibreOffice and Firefox just fine, because at execution time they get matched up with the right runtime, and all of them are available from all the operating systems you installed. You get the precise runtime that the upstream vendor of Firefox/LibreOffice did their testing with. It doesn't matter anymore which distribution you run, and which distribution the vendor prefers.

Also, given that the user database is actually encoded in the sub-volume list, it doesn't matter which system you boot, the distribution should be able to find your local users automatically, without any configuration in /etc/passwd.

Building Blocks

With this naming scheme plus the way how we can combine them on execution we already came quite far, but how do we actually get these sub-volumes onto the final machines, and how do we update them? Well, btrfs has a feature they call "send-and-receive". It basically allows you to "diff" two file system versions, and generate a binary delta. You can generate these deltas on a developer's machine and then push them into the user's system, and he'll get the exact same sub-volume too. This is how we envision installation and updating of operating systems, applications, runtimes, frameworks. At installation time, we simply deserialize an initial send-and-receive delta into our btrfs volume, and later, when a new version is released we just add in the few bits that are new, by dropping in another send-and-receive delta under a new sub-volume name. And we do it exactly the same for the OS itself, for a runtime, a framework or an app. There's no technical distinction anymore. The underlying operation for installing apps, runtime, frameworks, vendor OSes, as well as the operation for updating them is done the exact same way for all.

Of course, keeping multiple full /usr trees around sounds like an awful lot of waste, after all they will contain a lot of very similar data, since a lot of resources are shared between distributions, frameworks and runtimes. However, thankfully btrfs actually is able to de-duplicate this for us. If we add in a new app snapshot, this simply adds in the new files that changed. Moreover different runtimes and operating systems might actually end up sharing the same tree.

Even though the example above focuses primarily on the end-user, desktop side of things, the concept is also extremely powerful in server scenarios. For example, it is easy to build your own `usr` trees and deliver them to your hosts using this scheme. The `usr` sub-volumes are supposed to be something that administrators can put together. After deserializing them into a couple of hosts, you can trivially instantiate them as OS containers there, simply by adding a new `root` sub-volume for each instance, referencing the `usr` tree you just put together. Instantiating OS containers hence becomes as easy as

creating a new btrfs sub-volume. And you can still update the images nicely, get fully double-buffered updates and everything.

And of course, this scheme also applies great to embedded use-cases. Regardless if you build a TV, an IVI system or a phone: you can put together you OS versions as `usr` trees, and then use btrfs-send-and-receive facilities to deliver them to the systems, and update them there.

Many people when they hear the word "btrfs" instantly reply with "is it ready yet?". Thankfully, most of the functionality we really need here is strictly read-only. With the exception of the `home` sub-volumes (see below) all snapshots are strictly read-only, and are delivered as immutable vendor trees onto the devices. They never are changed. Even if btrfs might still be immature, for this kind of read-only logic it should be more than good enough.

Note that this scheme also enables doing *fat* systems: for example, an installer image could include a Fedora version compiled for x86-64, one for i386, one for ARM, all in the same btrfs volume. Due to btrfs' de-duplication they will share as much as possible, and when the image is booted up the right sub-volume is automatically picked. Something similar of course applies to the apps too!

This also allows us to implement something that we like to call *Operating-System-As-A-Virus*. Installing a new system is little more than:

- Creating a new GPT partition table
- Adding an EFI System Partition (FAT) to it
- Adding a new btrfs volume to it
- Deserializing a single `usr` sub-volume into the btrfs volume
- Installing a boot loader into the EFI System Partition
- Rebooting

Now, since the only real vendor data you need is the `usr` sub-volume, you can trivially duplicate this onto any block device you want. Let's say you are a happy Fedora user, and you want to provide a friend with his own installation of this awesome system, all on a USB stick. All you have to do for this is do the steps above, using your installed `usr` tree as source to copy. And there you go! And you don't have to be afraid that any of your personal data is copied too, as the `usr` sub-volume is the exact version your vendor provided you with. Or with other words: there's no distinction anymore between installer images and installed systems. It's all the same. Installation becomes replication, not more. Live-CDs and installed systems can be fully identical.

Note that in this design apps are actually developed against a single, very specific runtime, that contains all libraries it can link against (including a specific glibc version!). Any library that is not included in the runtime the developer picked must be included in the app itself. This is similar how apps on Android declare one very specific Android version they are developed against. This greatly simplifies application installation, as there's no dependency hell: each app pulls in one runtime, and the

app is actually free to pick which one, as you can have multiple installed, though only one is used by each app.

Also note that operating systems built this way will never see "half-updated" systems, as it is common when a system is updated using RPM/dpkg. When updating the system the code will either run the old or the new version, but it will never see part of the old files and part of the new files. This is the same for apps, runtimes, and frameworks, too.

Where We Are Now

We are currently working on a lot of the groundwork necessary for this. This scheme relies on the ability to monopolize the vendor OS resources in /usr, which is the key of what I described in Factory Reset, Stateless Systems, Reproducible Systems & Verifiable Systems a few weeks back. Then, of course, for the full desktop app concept we need a strong sandbox, that does more than just hiding files from the file system view. After all with an app concept like the above the primary interfacing between the executed desktop apps and the rest of the system is via IPC (which is why we work on kdbus and teach it all kinds of sand-boxing features), and the kernel itself. Harald Hoyer has started working on generating the btrfs send-and-receive images based on Fedora.

Getting to the full scheme will take a while. Currently we have many of the building blocks ready, but some major items are missing. For example, we push quite a few problems into btrfs, that other solutions try to solve in user space. One of them is actually signing/verification of images. The btrfs maintainers are working on adding this to the code base, but currently nothing exists. This functionality is essential though to come to a fully verified system where a trust chain exists all the way from the firmware to the apps. Also, to make the `home` sub-volume scheme fully workable we actually need encrypted sub-volumes, so that the sub-volume's pass-phrase can be used for authenticating users in PAM. This doesn't exist either.

Working towards this scheme is a gradual process. Many of the steps we require for this are useful outside of the grand scheme though, which means we can slowly work towards the goal, and our users can already take benefit of what we are working on as we go.

Also, and most importantly, this is not really a departure from traditional operating systems:

Each app, each OS and each app sees a traditional Unix hierarchy with /usr, /home, /opt, /var, /etc. It executes in an environment that is pretty much identical to how it would be run on traditional systems.

There's no need to fully move to a system that uses only btrfs and follows strictly this sub-volume scheme. For example, we intend to provide implicit support for systems that are installed on ext4 or xfs, or that are put together with traditional packaging tools such as RPM or dpkg: if the the user tries to install a runtime/app/framework/os image on a system that doesn't use btrfs so far, it can just create a loop-back btrfs image in /var, and push the data into that. Even us developers will run our stuff like this

for a while, after all this new scheme is not particularly useful for highly individualized systems, and we developers usually tend to run systems like that.

Also note that this in no way a departure from packaging systems like RPM or DEB. Even if the new scheme we propose is used for installing and updating a specific system, it is RPM/DEB that is used to put together the vendor OS tree initially. Hence, even in this scheme RPM/DEB are highly relevant, though not strictly as an end-user tool anymore, but as a build tool.

So Let's Summarize Again What We Propose

- We want a unified scheme, how we can install and update OS images, user apps, runtimes and frameworks.
- We want a unified scheme how you can relatively freely mix OS images, apps, runtimes and frameworks on the same system.
- We want a fully trusted system, where cryptographic verification of all executed code can be done, all the way to the firmware, as standard feature of the system.
- We want to allow app vendors to write their programs against very specific frameworks, under the knowledge that they will end up being executed with the exact same set of libraries chosen.
- We want to allow parallel installation of multiple OSes and versions of them, multiple runtimes in multiple versions, as well as multiple frameworks in multiple versions. And of course, multiple apps in multiple versions.
- We want everything *double buffered* (or actually n-fold buffered), to ensure we can reliably update/rollback versions, in particular to safely do automatic updates.
- We want a system where updating a runtime, OS, framework, or OS container is as simple as adding in a new snapshot and restarting the runtime/OS/framework/OS container.
- We want a system where we can easily instantiate a number of OS instances from a single vendor tree, with zero difference for doing this on order to be able to boot it on bare metal/VM or as a container.
- We want to enable Linux to have an open scheme that people can use to build app markets and similar schemes, not restricted to a specific vendor.

Final Words

I'll be talking about this at LinuxCon Europe in October. I originally intended to discuss this at the Linux Plumbers Conference (which I assumed was the right forum for this kind of major plumbing level improvement), and at linux.conf.au, but there was no interest in my session submissions there...

Of course this is all work in progress. These are our current ideas we are working towards. As we progress we will likely change a number of things. For example, the precise naming of the sub-volumes might look very different in the end.

Of course, we are developers of the systemd project. Implementing this scheme is not just a job for the systemd developers. This is a reinvention how distributions work, and hence needs great support from the distributions. We really hope we can trigger some interest by publishing this proposal now, to get the distributions on board. This after all is explicitly not supposed to be a solution for one specific project and one specific vendor product, we care about making this open, and solving it for the generic case, without cutting corners.

If you have any questions about this, you know how you can reach us (IRC, mail, G+, ...).

The future is going to be awesome!