Seamless Integration of Docker-based Applications into Linux Servers

Dr. Rüdiger Berlich[*][a], **Stefan Hug**[†][b], **Stefan Hacker**[b], **Dr. Ralf Mikulla**[b], **Henry Kehbel**[b], **Reinhard Ottmann**[b]

[a]*Gemfony scientific UG (haftungsbeschränkt)*
Leopoldstr. 122
76344 Eggenstein-Leopoldshafen / Germany
E-Mail: r.berlich@gemfony.eu
Web: http://www.gemfony.eu

[b]*PTV Planung Transport Verkehr AG*
Haid-und-Neu-Str. 15
76131 Karlsruhe / Germany
E-Mail: stefan.hug@ptvgroup.com
Web: http://www.ptvgroup.com

This paper presents a workflow for integrating Docker-based applications with the RPM and Debian packaging environment as well as the systemd init system to form an integrated solution, shielding users from the complexities of containerization. As an additional benefit, the Docker-specific separation of application and surrounding host allows application developers to focus on a single deployment platform rather than having to deal with sometimes subtle differences between target platforms, particularly on the side of dynamic libraries.

International Symposium on Grids and Clouds 2016
13-18 March 2016
Academia Sinica, Taipei, Taiwan

[*]Speaker.
[†]Main contact

1. Context and Technical Background

The work presented in this paper was performed by the authors at and for PTV Group [1], a Karlsruhe / Germany-based company with a focus on traffic simulation as well as logistic geographic components[1]. One of PTVs products – the xServer[2] [2] – will usually be installed in the form of a medium-sized Web Application Server, which is then accessed over the intranet by Internet-facing, custom designed servers. The application comprises a Java frontend, which deals with all networking-related tasks, as well as modules implemented in C++, in order to achieve good performance for particularly computing-intensive tasks.

The xServer is meant to run on both Windows and Linux hosts. However, while portability between Windows hosts has not been a source of significant problems so far, subtle differences between Linux distributions may make it cumbersome to deliver a binary application to customers running different brands of Linux. This may appear counter-intuitive, as both on the server and the desktop side a wealth of precompiled Linux applications is readily available. Very rarely will typical Linux users need to compile general-purpose applications themselves. However, closer inspection reveals that binary applications, which are delivered by Linux distributors, are not only specific to their particular brand, but are indeed made available separately for each new version.

Apart from other sources of incompatibilities[3], dynamic linkage is commonly used for C/C++-based applications, meaning that executables may have to rely on a mix of libraries already present on the target host. Dynamic libraries accessed by an application must “fit” together in the sense that all expected functions and symbols are present, and perform a task in the way it is expected to by an application[4]. This may be achieved either by only installing (dynamically linked) binaries that already fit the target host, or by delivering all required libraries together with an application.

Users in HPC and research with a requirement to frequently run code on different Linux targets, as well as organizations wishing to supply commercial (server-)applications in binary format to Linux users, may thus face additional complications, particularly where re-compilation is not an option – be it because it is impractical, or because an organization does not want to reveal the source code. Many commercial application providers will also want direct control of the properties of an application’s dependencies, so they can rely on a known behavior of “external” libraries. The same applies to code in natural sciences and engineering that requires a particular behavior of mathematical libraries (e.g. rounding issues, lack of bugs, stability . . .).

While static linkage may help, some key libraries may *not* easily be linked statically into an application. As an example, the dominant Linux C/C++-Compiler `gcc` will silently link a library called `libgcc`, but explicitly warns that “*There are several situations in which an application should use the shared libgcc instead of the static version. The most common of these is when the application wishes to throw and catch exceptions across different shared libraries.*” [8]. Most C++ applications *may* throw exceptions, and *could* do so across different libraries, particularly as it is a design goal of the C++ exception mechanism that the entity throwing an exception does not need to know who catches it.

[1]. . . such as routing- and mapping-applications

[2]. . . which is not to be confused with the X11 window system common on Unix and Linux

[3]. . . such as different filesystem layouts, location of configuration files, etc.

[4] This situation is sometimes referred to as “dependency hell”

Shipping a defined set of dynamic libraries along with applications is thus a more attractive option. Libraries may be “preloaded” (by adding them to the `LD_PRELOAD` environment variable or to the corresponding `-library-path` option of the Linux loader[5]), so that an application only “sees” compatible libraries. In essence, by following this procedure, an application is *isolated* from a host’s Operating System. While this is generally a valid solution, it requires further work on the side of the application-provider, as not only the libraries accessed *directly* by the application need to be collected, but also those that are accessed *indirectly*. All of this can be dealt with, but procedures may be complicated and may introduce other, unforeseen platform dependencies.
Hardware virtualization, using one of the many (sometimes freely available) options provides a far more complete separation from the host system, and the application may see exactly the execution environment it expects, including its own filesystem. However, alongside possible licensing issues, the performance penalty of the virtualization layer may prevent adoption of this solution as the basis for application delivery. And running many virtual machines on the same physical host may consume larger amounts of memory and CPU, far beyond what is needed by the application alone.

2. Containerization as an Alternative

On Linux, a relative newcomer to this game is “containerization”, with its current main representative “Docker”. Container environments combine the benefits of (partial or complete) isolation of an application from its host Operating System with near-native performance. Only very few dependencies on the host system remain, mainly between the application and the host’s Linux kernel. Applications, including their execution environments, may be delivered to target systems in a single file or via repositories, making them as easy to handle as virtual machines[6].
From the application provider’s perspective, a Docker-based execution environment isolates an application much further from the surrounding host than through mere preloading of libraries. Core libraries *still* need to “fit” the host kernel. But inside of the Docker image the execution environment is self-contained and the application cannot distinguish it from a stand-alone machine. Docker thus allows the provider to focus development efforts on a single platform. Most host-dependencies are shifted to the Docker environment, so that maintenance efforts for different target hosts may be reduced and the application provider may ideally follow a “compile once, run anywhere” policy.
On the downside, Docker itself may introduce complexities and may be subject to incompatibilities. Access to data inside of the container, such as logs and configuration files, must be granted to users; data inside of a container may have to be persisted; and the application may require access to outside data or may need to connect to a database management system running on the host system. While this can be handled on the technical level, it may require long and complex calls to the Docker daemon on the command line, making the usage of a Docker-based application cumbersome for the end user. And while, in the experience of the authors, compatibility of Docker with both hosts and hosted applications is good (see section 6 for a discussion), some significant differences *do* exist between different Docker versions. As just one example, the size of `/dev/shm` was hardwired to 64 MB in Docker versions prior to 1.10, leading to some hard-to-track failures for some applications.

[5]For example `/lib/ld-linux.so.2` under Ubuntu 16.04
[6]Appendix A gives a more detailed overview of Docker.

Ideally, for the end-user, running Docker-based applications should not be different from running "native" applications (in the sense of applications running under the control of the host's Operating System environment). Application providers may then have to invest further work in order to make usage as transparent as possible to the end-user.
The following sections present means of achieving transparent integration of Docker-based applications into target systems, on the example of the work performed for the PTV xServer. The discussion follows a top-down approach, starting with the scripting needed for the integration into the user's host, followed by wrapping images and scripts into RPM- and Debian-packages and ending with the design of the Docker images themselves.

3. The Control Script

"Transparent integration" in the context of the last paragraph means that users[7] may control the application[8] through easy commands, such as "`start`", "`stop`", "`reload`" and "`status`". This follows the conventions used for System V and LSB[9] init scripts [19].
No knowledge of the actual command line arguments used to start the container should be required on the side of the user, but all actions should nevertheless be configurable. The control script(s) thus need to perform the following actions:

- Perform sanity checks (availability of the Docker installation, availability of the payload, ...)
- Locate the Docker image either locally[10] or in a repository.
- Parse a config file, holding information such as the data to be mapped into the container, the size of the shared memory section or environment variables to be set inside of the container
- Parse the command line for the required action
- Assemble suitable command line calls for the Docker daemon and interact with the Docker daemon to control (start, stop, ...) the container
- Interact with the application running inside of the container (reload, establish status, ...)
- Route logs to a location defined in the config file
- Clean up after the application has ended (remove unused containers, ...)

Particularly the need to interact with the container and to control the application inside of it requires knowledge about the image's design, which is custom-built for this purpose (see section 5).. Information about the payload is made available via the config file. Listing 1 shows the actual Docker command assembled by the control script[11].

[7]Henceforth the term "user" means a person wishing to start a given application inside of a Docker container

[8]Henceforth the term "application" denotes an executable, including all required libraries and OS components, installed inside of a Docker image

[9]**L**inux **S**tandard **B**ase

[10]... as a tar archive created with the `docker save` command

[11]Note that site-specific information was replaced by dummy information

Listing 1: The command assembled from the information in the config file

```
/usr/bin/docker run --cap-drop=all --name=dockerpayload \
   -v /tmp/payload.log:/tmp/log:rw  --shm-size=2G  -u 995:995 \
   -v /etc/timezone:/etc/timezone:ro \
   -v /etc/localtime:/etc/localtime:ro \
   --rm=true  -e http_proxy=some.local.proxy:8080 \
   -e https_proxy=some.local.proxy:8080 \
   -e no_proxy="127.0.0.1,localhost" \
   -p 50000:50000 \
   -v /opt/payload:/opt/payload \
   doreka/runtime-image:16.04 \
   /opt/payload/server.sh console
```

Mapping data into the container (option `-v`)[12] plays an important role, and so do environment variables, such as proxy settings to be made available to the application. Network ports are configurable, and the application to be started is passed as an argument to the container, including arguments needed by the application. This design was chosen as all data, including the payload, is stored outside of the running container. This allows to keep Docker images mostly generic, so they become less of a moving target. As the payload will be owned by a normal user rather than `root`, the control script determines the id of the user (995 in listing 1) and makes sure that the payload is started under the same id inside of the container. As a direct consequence, all data written by the payload is owned by the correct user, which is beneficial for practical and security reasons.
The control script is currently written in the `bash` scripting language, which corresponds to the way in which init scripts were traditionally written. An alternative might be Python scripts, as it comprises very powerful modules for controlling Docker and for parsing configuration files. A disadvantage of using Python would be the additional requirement this puts on the user's system.
As specific knowledge about the payload (for example the path to the main application to be started, the name of the Docker-image, ports etc.) has been abstracted away using the configuration file, the startup script is generic, i.e. does not need to have any special knowledge about the payload and can indeed be used verbatim for other applications than "only" the PTV xServer[13], as long as they follow a few simple conventions.

4. Integration into the Init System

As the startup script is modeled after the LSB init scripts, integration into the target host's init system becomes easy. This allows the payload to be started and stopped automatically, when the host boots or shuts down, which is beneficial for server-type applications. In the ideal case, no manual interaction with the payload is necessary anymore from the user side.
On most modern Linux distributions `systemd` is used as the init system. `systemd` may be made aware of a custom service by registering a `payload.service` file[14]. Many tutorials explain

[12]For example, all console output of the payload is logged to a file mapped into the container from the outside

[13]. . . it may indeed start multiple Docker-based applications at once

[14]The term `payload` is meant as a placeholder and should be replaced by a custom name for a given payload

how such a service file must be set up, so this process will not be described here. However, obvious information needed by the service file are the commands needed to start and stop a service. With the control script, creating the service file becomes easy, as it only needs to refer to this script. At this point, users may control the payload with the familiar `systemctl` commands[15]:

```
# systemctl start payload

# systemctl stop payload
```

This assumes that the payload's service script is called `payload.service` and is placed in the appropriate directory (such as `/lib/systemd/system`). Docker will usually also be started as a `systemd` service, so that the Docker daemon may be started automatically before the payload service is started. Instantiating the actual payload container is then done by means of the control script, which is in turn started indirectly through `systemctl`. The command

```
# systemctl enable payload
```

integrates the service file into the `systemd` chain, so that it is started automatically at boot time. Issuing the command

```
# systemctl disable payload
```

makes sure that servers controlled by the script are no longer started automatically at boot-time.

5. Image Design

The control script was designed to be generic, and the Docker image[16] was similarly designed not to have any particular knowledge about the "payload" application it will harbor. It *may*, at the image designer's choice, contain additional supporting software (such as Java), but will not contain the application itself (but see section 5.3 below for some additional notes on this topic). Keeping the runtime environment generic allows it to be used with only few changes for many purposes, which in turn facilitates the build chain described in section 7. It is also consistent with the accepted policy to keep Docker images as thin as possible.

The runtime image may be created using the standard Docker facilities[17] (i.e. setup of a `Dockerfile` which may itself call scripts to model the image according to user-requirements). As such, the creation of the "runtime image" may be automated. In the environment described in this paper the runtime image is based on the standard Ubuntu 14.04 and 16.04 Docker images, but may build on any other Linux distribution available through the Docker Hub [18].

The images differ from the standard images by a few settings, which are described below.

[15]...these come with `systemd`

[16]...also called "runtime image" below, as it harbours the runtime environment for the payload

[17]Another possibility is *Ansible Container* [16] – which was released short time before this paper was finalized.

5.1 The Init System

One inherent difficulty of Docker containers is what is often called the "zombie reaping problem" (a thorough discussion together with a possible solution may be found in [6]). In short, in a Unix environment, the process with id 1 is responsible for "reaping" (or clearing) orphaned processes, whose parents no longer exist and who are not performing any more work. As Docker containers are designed to run a single application only, this application may receive the process id 1. As it will usually not be designed to eliminate orphaned, dead processes, the process table may fill up with "zombies". Using `systemd` inside of a Docker container is not a viable option, as it requires access to some system calls that are not usually granted to applications running inside of a container[18]. [6] describes a minimal init system that may perform this task and may also help to run standard Unix SysV init-scripts. However, one of its disadvantages is that it is Python-based, resulting in a requirement to install this interpreter, together with many supporting libraries, in the Docker image. As a result, these images will become much larger than they need to be for most application scenarios (except those requiring Python themselves).
The Docker environment described in this paper now uses a simple init system with the telltale name "`dumb-init`" [14]. It has a very reduced functionality – its main duties are handing signals to applications and handling the "zombie" responsibilities. `dumb-init` is statically linked and thus has no external dependencies.

5.2 Handling Application Startup

One disadvantage may be that `dumb-init`[19], at the time of writing, has no support for running any init-scripts at startup or shutdown time of the Docker container. This may be necessary, for example, in order to set the locale inside of the container, or any other custom environment variables. However, this functionality may be added in a simple way by letting dumb-init call a shell script that in turn performs user-defined tasks. By "mapping" modules for this script into the running container, such functionality may even be steered from outside of the container. This script may also be used to start the payload under a specific user id, which is made known to the container through the switch `-u` of the Docker daemon (compare listing 1).

5.3 Aggregating the Payload with the Runtime Image

As the payload is kept outside of the runtime image, it needs to be mapped into the container at startup time. Docker contains all necessary functionality (the "`-v`" switch of "`docker run`"). Keeping the payload outside of the image has several advantages, including

- The runtime-environment is kept generic
- Any data written in the payload's filesystem tree is persisted, i.e. it will "survive" when the container is destroyed (which is the common procedure when the payload shuts down, in order not to "litter" the host with containers remaining from former runs[20].

[18] Applications running inside of a Docker container should be shielded as much as possible from the host environment, both for security reasons and in order to keep services implemented via Docker images as generic as possible.

[19] `dumb-init` was initially developed by *Yelp*, a recommendation portal for restaurants and shops. At the time of writing, it is available under the MIT license from the GitHub Open Source portal [15]

[20] `docker run` will create a new container each time the command is called.

- Reading and writing to the payload's filesystem tree does not need to go through the overlay filesystem maintained by Docker
- Data written by the payload and the payload itself may use the same user id

A disadvantage of this procedure may be that access rights to the payload's filesystem tree need to be tailored both to the host[21] and to the way Docker works.
Two alternatives exist:

- The usage of Docker's data containers instead of plain directories stored on the host filesystem. The data inside of this particular container type is persisted.
- All data, including the payload and all supporting data, may be stored inside of the image. This may result in very large images, but brings Docker containers closer to common usage patterns of virtual machines. It may also be beneficial for roll-backs, i.e. in situations, where several different versions of the same payload and/or data must be accessible at short notice.

6. Notes on Performance and Compatibility

Section 2 has referred to the "near-native" performance of Docker containers. In general, applications running inside of a container do access the same kernel available on the surrounding host and may, under ideal circumstances, achieve the same performance as a "native" application (i.e. an application not running under the control of Docker). Docker *does* use an overlay filesystem, though, which *may* cost performance. Docker uses *bridged networking* by default. And it *may* be suspected that the Docker management layer (including the facilities used for containerization in the Linux kernel) introduce a sizeable runtime overhead. The performance of the payload should thus be monitored. Tests with the PTV xServer have not shown a sizeable degradation of performance. General statements might be difficult, though, as not all usage patterns can be tested, and the performance will depend on the particular patterns being used. The design decision to store data and payload outside of the image may have helped to achieve good performance, though.
In order to further understand the performance available through Docker, a "toy application" tried to model key aspects of xServer, simulating particularly high network- and file-IO workloads as well as high CPU-loads. Like in the case of the xServer, these tests did NOT involve the overlay filesystem, but have instead used an external directory on the host system which was mapped into the container. The toy application consisted of a server which filled up collections of objects holding random numbers. 50 clients running on a remote host could contact the server and request a collection. The server would then serialize the buffer (either in binary- or XML-format) using Boost.Serialization and ship it to the client, using Boost.ASIO.
The client would then send a sorted collection back to the server. Optionally, the sorted collection could be written to disc in order to simulate file-IO. Thanks to the usage of an external native directory and this IO pattern, no significant overhead is expected for read operations, though. The size of the collection could be adapted in order to model different types of workloads. Tests were conducted between two servers that were connected over the intranet, as this conforms to a

[21]For example SELinux or AppArmor might need some special treatment

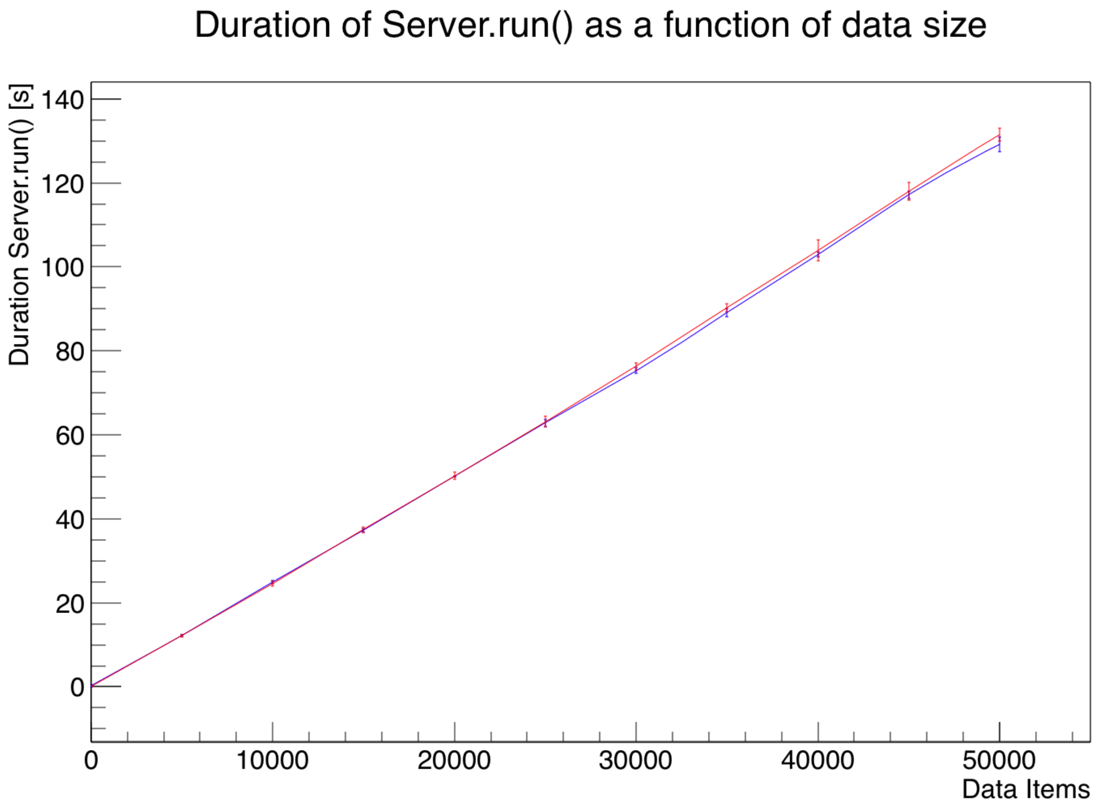

Figure 1: Performance of native (red) and containerized (blue) execution of a networked toy application as a function of the data size being transferred

likely usage pattern for xServer. No major deviations of native operation of the toy-applicationss server compared to "dockerized" deployment were observed for collection sizes up to 50000 objects (equivalent to some 760 kilobytes per message. Slight deviations for larger collection sizes were seen, particularly when writing data to disk. It is not clear whether these were due to Docker or related to some other system and network activities. Figure 1 shows that the performance of the native and "dockerized" execution agree well within their error bounds (multiple measurements were made for each data point).

Further tests of the write performance were carried out using the Unix tool `dd`. In these tests write performance was degraded by a few percent only, even when writing into the overlay filesystem. The data written exceeded the amount of memory available in the system, so the effects of the Linux Virtual Filesystem Layer "VFS" were minimized.

Compatibility of the Ubuntu 16.04-based images was tested on CentOS 7.2, OpenSUSE 42.1 and 13.2 as well as Ubuntu 14.04 . No incompatibilities on the level of the payload were found. However, the tests have shown problems with the size of `/dev/shm` available by default in the Docker images. A short discussion of this topic can be found in section 2. As of version 1.10 this setting may be configured using the `-shm-size` switch (compare listing 1).

7. Assembling RPM- and Debian-Packages

This section describes how scripts, images and payload may be wrapped into native packages for the target platform. The section assumes that the following components (according to the previous description) are available:

- A directory holding all files belonging to the payload
- A "thin" Docker image used as the runtime environment. The image must be compatible with the payload (in particular, it must have all required external libraries)
- A configuration file holding the path to the application to be started inside of the container (plus other supporting information)
- A startup script that understands how to parse the configuration file, map data into a container, set up networking, start, stop and reload the container (including the payload) itself.

Standard procedures and tools exist for all major Linux distributions to package all of the above into a single file. The Debian package format is commonly used on Ubuntu (and of course Debian ...), while RPM is used by Fedora / RedHat / CentOS as well as SUSE. Such "`.rpm`" or "`.deb`"-Packages may then be stored in package repositories or simply downloaded over the Internet. They may be installed using commands specific to each distribution (such as "`rpm -i`" on RPM-based distributions or "`gdebi`" and "`apt` / `apt-get`" on Debian / Ubuntu)
Installation of a package will unpack the data in directories specified upon package creation and will register these files in a local database. The target host then knows, which files belong to which package and may manage them henceforth. This also facilitates the removal of a package.
RPM- and Debian-packages also allow us to automatically run scripts provided by the package designer before and after installation and removal of a package. This facility may be used to register the init-scripts with `systemd` (or any other init system) and to prepare the environment or clean up the local Docker registry after the removal of the package. The post-installation script also takes care of loading the runtime image into the local Docker registry.
Another nice feature of RPM- and Debian-packages is the ability to handle dependencies. For example, a package holding a Docker-environment will have an obvious dependency on the Docker package available for a given Linux distribution. Hence, upon installation of the package, all dependencies will automatically be installed as well. **The final packages then make the entire installation process transparent to the user and automate the integration into the target system up to the point, where the service is started and shutdown automatically by the host.** The installation of the entire environment can then be done using a single command such as (in the case of Ubuntu):

```
# apt install mypackage.deb
```

Here "mypackage.deb" represents the package holding the image and scripts. The package will register the image with Docker, register the start-up script with `systemd` and will unpack the payload including its data and with the correct permissions in the desired location. The need for user-interaction is minimized in this way (although it might be necessary to copy license files or add additional data not contained in the package).

7.1 Building RPM- and Debian-Packages for different target platforms

In order to facilitate the production of packages (e.g. in the case of code changes in the payload), all steps of the procedure should be automated through a build system. It should be able to create RPM- and Debian-packages for all desired target distributions, without a need to use remote hosts or virtual machines. This may be difficult, as even platforms with the same package format have different conventions for the production of packages. In our case, Ubuntu 16.04 is used as the build host. While it does have some support for RPM, this is certainly not the ideal environment for building them. However, as we are already acting in a Docker environment, we may easily create so called "build images", i.e. Docker images holding the desired target distribution with all tools needed to build the RPM and Debian packages for a specific version of a given Linux distribution. Containers created from these build images may then be executed on the build host, so that the entire procedure can be centralized, while using the specialized tools available for each target distribution.

7.2 Build Orchestration

Each step of the build chain described above consists of a larger amount of Unix, Docker and package-manager commands. Most steps are performed in sequence, and intermediate results of the build chain may depend on preceding results. Other steps, such as the creation of build images or packages for different target platforms, may be performed in parallel.

Various build environments are available that help to manage this situation, but are usually targeted at the compilation of code rather than steering complex Unix/Linux processes. In the context of the work presented in this paper, attempts were made to leverage `CMake` and `Maven` for this process. `Maven` even has a dedicated Docker plugin, and both tools can be "coerced" into issuing custom external commands. Both tools have however proven to be problematic for this purpose, as calling external commands, while generally being possible, is cumbersome and some automatisms in the Docker plugin make fine-grained control of the procedure difficult.

In a *Keep it Simple*-move, plain GNU Make was finally used, as it covers all core requirements of the build procedure, such as build automation, the ability to "cascade" builds and fine-grained control. In addition, it has excellent support for dependency management[22], and even very complex external commands may be called with ease via a shell environment.

Figure 2 gives an overview of the entire workflow, from the creation of runtime images over the assembly of Debian- and RPM-packages up to the installation on the user's system. It has been implemented and tested in prototypical form for the PTV xServer. However, the techniques described above may also be applied to other problem domains, such as the distribution of scientific applications to target clusters with a diverse infrastructure.

The runtime images might also be usable for image-orchestration tools such as Kubernetes or Docker Swarm, although self-contained images (i.e. images holding the entire payload) might be a better choice for these environments.

[22]Dependencies are specified manually in the form of rules, but are sufficiently static that this is not a problem.

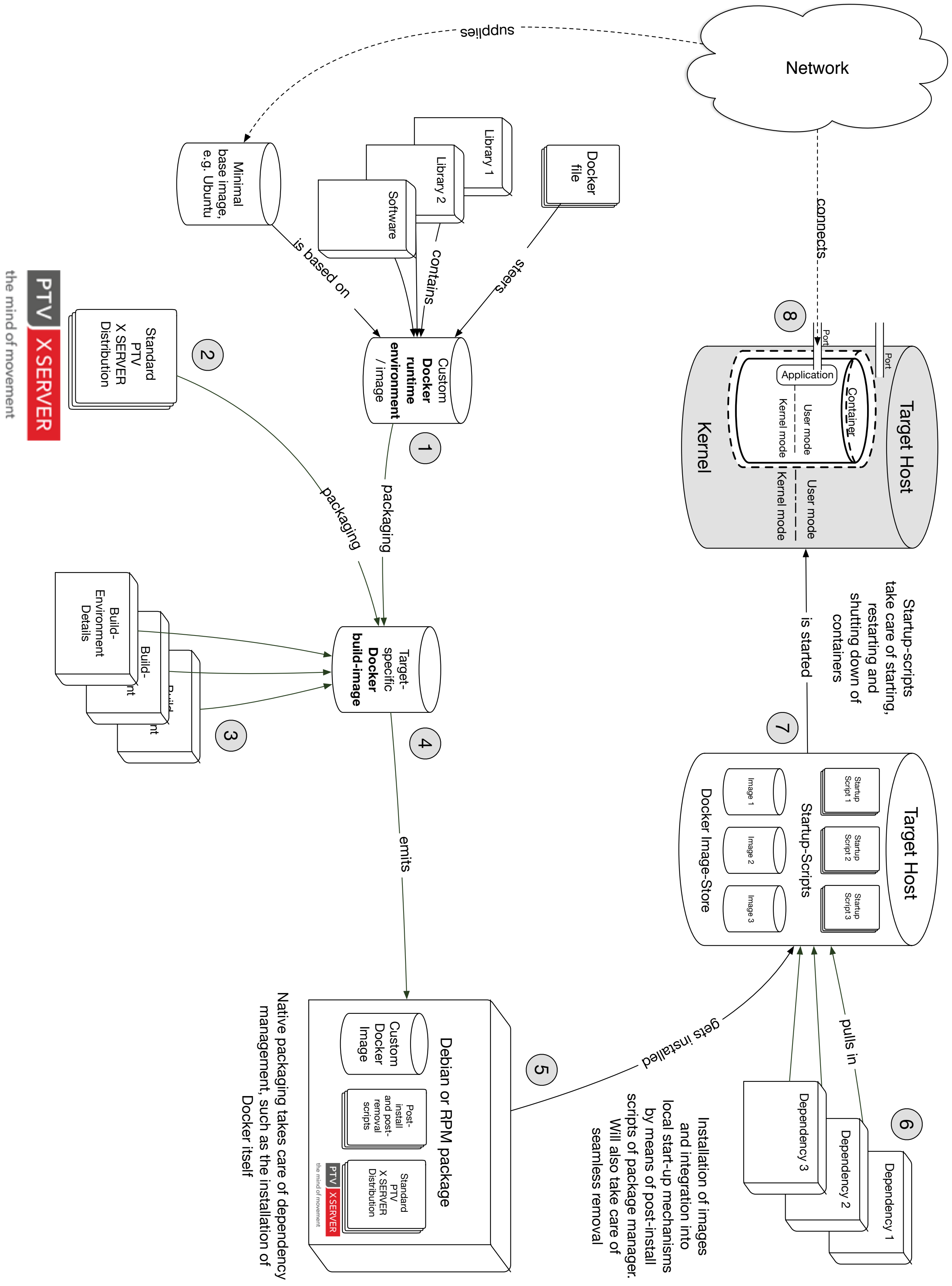
Network
supplies
connects
Minimal base image, e.g. Ubuntu
Library 1
Library 2
Software
Docker file
is based on
contains
steers
Custom Docker runtime environment / image
1
2
Standard PTV X SERVER Distribution
PTV X SERVER the mind of movement
packaging
packaging
Build-Environment Details
3
Target-specific Docker build-image
4
emits
Debian or RPM package
Custom Docker Image
Post-install and post-removal scripts
Standard PTV X SERVER Distribution
Native packaging takes care of dependency management, such as the installation of Docker itself
5
gets installed
Target Host
Startup-Scripts
Startup Script 1
Startup Script 2
Startup Script 3
Image 1
Image 2
Image 3
Docker Image-Store
6
pulls in
Dependency 1
Dependency 2
Dependency 3
Installation of images and integration into local start-up mechanisms by means of post-install scripts of package manager. Will also take care of seamless removal
7
is started
Startup-scripts take care of starting, restarting and shutting down of containers
Target Host
Kernel
User mode
Kernel mode
Container
Application
Port
8

Figure 2: Simplified view of typical workflow for the creation of Docker-based RPM and Debian-Packages

8. Conclusion

Docker provides a simple and convenient way of running custom applications in an isolated environment. Such "containerization" may thus help to run an application in exactly the environment it expects, thus helping to alleviate or even remove the problems involved with what is often called "dependency hell". Crafting portable applications is thus much easier to achieve with the help of container environments such as Docker.

Handling of Docker and integration of Docker images into a target system is more complex, and some work needs to be invested to achieve a seamless integration. This paper has outlined a workflow that results in native packages dealing with the integration into the target system. In the common case, users of these packages will only need to have knowledge of some standard Unix commands for the installation and removal of RPM or Debian packages as well as enabling and disabling services, where needed. Our measurements[23] also indicate, that the performance of "dockerized" applications may be near-native, despite heavy network traffic and including disk-IO.

On the side of the application provider, most steps of package creation may be automated, and most application-specific tasks may be abstracted away to the point where package creation is a generic process. It may even be possible to *compile* and run an application in exactly the same environment, while installing it on a wide array of different Linux distributions. Docker, as a now established and well supported environment, appears to be a static target which may be relied upon. Other container environments exist for Linux, which use the same underlying functionality of the Linux kernel. Hence Docker is also not a "single point of failure".

The remaining dependency on the host's kernel[24] needs to be taken seriously (in the sense that compatibility checks should be performed), however these dependencies have in our experience not presented a problem so far. This is also due to the fact that kernel APIs do not change often. Besides, where such dependencies are a problem, the only remedy would be full virtualization or compilation for each target system. Preloading (using `LD_PRELOAD`) in particular of the core system libraries would result in the same dependencies and thus the same problems as in the case of Docker. Hence, where an `LD_PRELOAD` solution is considered, a Docker solution should be considered as well.

Docker is, by design, also a "Cloud" tool and may thus appear quite heavy-weight, particularly where the "orchestration" of images and other cloud features are not needed. With Canonical's "*Snap*" packages as well as RedHats "*Flatpack*", two other techniques have just appeared that are based on containers. These are directly tied in with package-managers, so that some of the procedures outlined in this paper might be bypassed when these are used. From the application provider's point of view, a single runtime environment would be desirable, and Snap and Flatpack come from competing companies. So while these are promising developments, they are certainly not yet as established as Docker and should be given some time to mature. Docker itself will likely remain a market proposition for a long time and thus currently appears to be the path to follow.

[23]...see also [17]

[24]The container runs directly on the host kernel, so system calls required by the application need to be present

A. An Overview of the Docker Platform

Docker [12] is commercially supported Open Source software, with a VC funding level of $160 million (US) for Docker Inc. (the company behind Docker) at the time of writing [4]. The initial public release dates back to March 2013, with a 1.0 version made available in June 2014. Docker was initially created as an internal project for the PAAS[25] company *dotCloud*, and is partially written in Go, a programming language developed by Google as of 2007. Docker manages (but does not provide) container technologies present in the Linux kernel since 2008. So while Docker has older roots, it must by itself be considered to be relatively new technology.

As quoted from [11]: "*A container is a self-contained execution environment that shares the kernel of the host system and which is (optionally) isolated from other containers in the system.*". The term "execution environment" refers to libraries, programs and configuration files needed by an application, organized in a common filesystem hierarchy. Often this will simply be a standard (minimal) installation of one of the common Linux distributions, with the one difference that the libraries and applications stored inside of a Docker image use the host's operating system kernel at run-time (i.e., use Linux kernel functions and let themselves be managed by the kernel).

Figure 3 provides a schematic comparison between containerized execution, virtualization and native execution on a host. From the view of an application running inside of a container, the main difference to hardware virtualization is that it uses a "foreign" kernel provided by the host OS.

"*namespaces*" and "*control groups*" (or "*cgroups*" for short) inside of the Linux kernel shield containers from the host system, so that applications running in a container may only access very limited resources on the host. Both were initially independent, but related developments. They were merged into the Linux kernel with version 2.6.24 and allow to isolate individual processes or groups of processes, so that resources of other groups are not accessible to them, making accidental or malicious modification of "foreign" resources harder to achieve. Individual tasks and groups of tasks including their children may be partitioned into hierarchies with specific restrictions and privileges, and their resource usage (CPU, memory, swap, ...) controlled. Namespaces include process id, network and user ids. The first process started in a Docker container will always have process id 1, even though the init system of the surrounding host already possesses the same id. Also, ip and ports, routing and iptable firewall rules are separated from the host system and other containers (unless the user of an image allows access).

Access to host-devices from the container-side is limited, but configurable. For example, Docker may explicitly allow a container to access and use the host's network namespace. The host, in comparison, has detailed knowledge of what goes on inside of a container and may restrict the resources being used or suspend or even stop applications running inside of the container. Means of logging the activities inside of the container exist. So while an application running inside of a container is isolated, full control is offered to maintainers outside of the container.

Containers are more efficient than "traditional" virtual machines, as no part of the hardware needs to be emulated. Contrary to virtual machines, however, at the time of writing, only Linux applications may run inside of Docker. Microsoft is working on direct Docker-support for Windows [7] inside of containers, though. Note that Linux containers will always require a Linux host, and Win-

[25]Platform As A Service

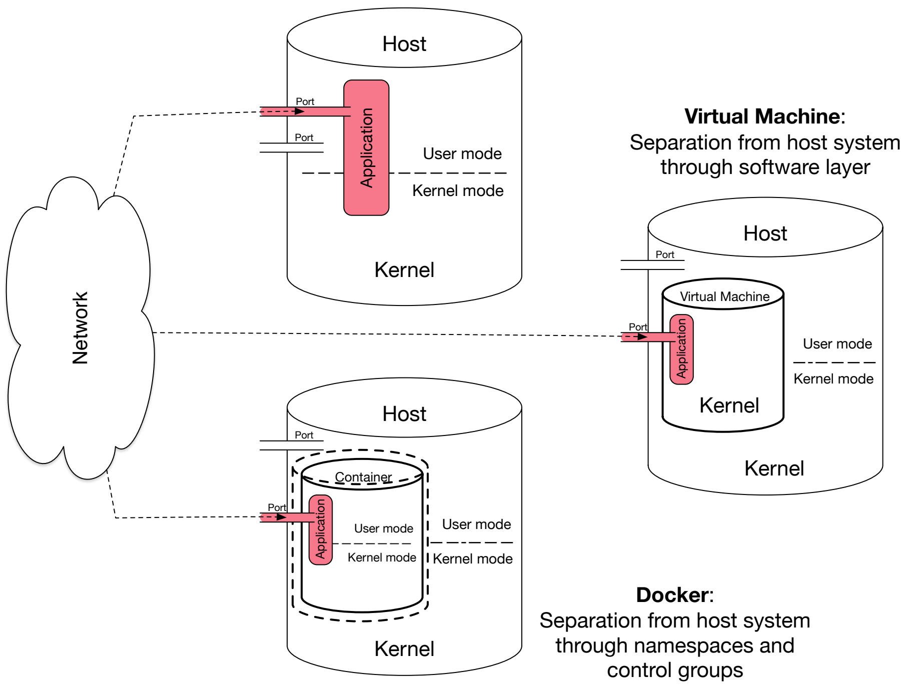

Figure 3: A comparison of native vs. container- and virtual-machine based execution environments

dows containers a Windows host, as the host's operating system kernel is used. "Mixed" operation is only possible with the help of an intermediary virtualized OS[26].
It must be noted that containerization is not a new paradigm – similar features were available for example inside Solaris and are available in BSD. The underlying techniques – namespaces and control groups – have been in production use for close to 8 years. Furthermore, Docker is not the only container-implementation leveraging these capabilities of the Linux kernel. At this point in time, however, Docker seems to have the biggest momentum and is being used in production by a wide array of companies.
Docker-applications are delivered in the form of distinct disk-images[27], similar to what is common for VMs, and may thus be easily transferred between hosts. They may alternatively be stored in a repository. Docker containers are "instances" of images and use an overlay filesystem to store data written inside of the container. Corresponding data is stored in separate "overlay" files which, together with the disk image, form the execution environment "seen" by the application. Docker takes care that data in all files belonging to the container is combined to a single "view". An advantage of this "split storage" is that for updates of a container not all data needs to be transferred – in a way only a "diff" needs to be applied.
(Usually bridged-)networking and the use of an overlay filesystem make it necessary to investigate the performance of an application running inside of a Docker container. There is little reason for concern about the performance of CPU-bound applications, though, as these are using the native host kernel, and the hardware does not need to be emulated. Some tests performed by us [17]

[26]A theoretical possibility might be a "translation layer" like Wine that translates system calls between OS platforms.
[27]. . . albeit layered, but they may be stored in a single file for transportation

indicate, though, that Docker appears to have very little overhead even for disk- and network-IO. It does help and is even a crucial requirement for the deployment of an application via Docker that it is easily possible to make selected host-directories and even individual files available to the Docker container in such a way that the application running inside of it cannot distinguish such data from local files stored inside of the image.
The fact that almost arbitrary Linux execution environments may be stored inside of and run through Docker images also means that, through Docker, portability to most other recent Linux distributions may be achieved. Dependencies mostly exist between the C-libraries and the operating system kernel (the libraries may not attempt to call kernel-functions that do not exist there). It is expected that the effects of this dependency are small. Competing methods, such as LD_PRELOAD, have the same problem.
Efficient and seamless integration into the host's network is crucial for a successful deployment of networked applications like xServer. Docker supports different network modes out of the box:

- Host-Networking makes the hosts network settings directly available to the container. Any network communication coming from the container cannot be distinguished from the host's networking. It has the same IP and name as the surrounding host. This is also reportedly the fastest networking mode. One disadvantage is the possible yet unwanted exposure of ports. However, a firewall on the host system should help to prevent this. Also, since the container has access to the host's network settings in this mode, a privilege escalation inside of the container may have more severe implications than in the case of bridged networking.
- Bridged Networking uses Network Address Translation ("NAT"). The container is assigned its own private IP and hostname, and ports may be reassigned. For example it is possible to let a container communicate externally with port 40040 while the "native" port is 40030. This adds additional flexibility. Also, only those ports that were specified at the start of the container will be visible externally. The container (and its application) are thus better protected than in the case of Host-Networking. Bridged Networking is the Docker default, but may imply some overhead for the address translation. However, tests performed by our team have shown little cause for concern for our application scenario. It must be noted, however, that such statements cannot be generalized, as they are highly application-dependent.
- Additional network modes exist that let Docker containers interact among each other. These are likely not relevant for the deployment scenarios discussed further below and are therefore not discussed here.

Docker Images are commonly organized in repositories, and "standard" images exist for practically all Linux environments and application scenarios. It is easily possible to save Docker images and to restore them on a different host. The creation of Docker images is steered through a "`Dockerfile`", which holds information about what should be stored inside of a container and how it is set up.
Docker wants to "shield" applications from each other. It is thus considered "good style" in the Docker community to only run a single application inside of a container. For some application scenarios (for example where an application needs other, closely coupled supporting applications) this

may be hard to achieve. However it does make sense to keep a Docker-image as "thin" as possible. This may also lead to an additional security benefit not easily available to a native deployment of an application at the user site without Docker.
Containers should be stateless so that they may be suspended, stopped and restarted at will, without disturbing later operation. This mandates the separation of the application from any data written by it, which should be stored natively outside of the container in a dedicated directory. As a side benefit, storage of data outside of the container allows separate distribution of supporting data without access limitations for the application running inside of a container.
In the case of databases being accessed by an application, connections should be closed upon shutdown and the database itself should be stored outside of the container. Note that database access will happen through the external IP of a host, as an application running inside of a container (with bridged networking) will have no access to the IP `127.0.0.1` (for example "`localhost`") of the surrounding host. Host networking may help here, but comes with its own set of problems.
Docker is well supported out of the box by all important recent Linux distributions. Docker Inc.'s description "Install Docker Engine" [13] lists different versions of Red Hat Enterprise Linux, CentOS, Fedora, OpenSUSE and SUSE Linux Enterprise, Ubuntu, Debian (and others) as supported platforms. Docker-packages may thus usually be installed from the native package repositories (if they haven't been installed already by default).
The integration of Docker images into target systems may be automated, but might require further work. One way of hiding this task from the user is the integration of images and startup scripts into the two dominant Linux package formats – Debian- and RPM-packages. Leveraging these package formats also solves further problems and turns out to make the installation and deployment process virtually transparent for the user.

B. Thanks

The authors would like to thank their colleagues that have helped to shape the work presented here, as well as FERCHAU Engineering GmbH for their support of this work.

References

[1] `http://www.ptvgroup.com` – The web presence of PTV Group; visited July 4, 2016

[2] `http://xserver.ptvgroup.com/en-uk/products/ptv-xserver/` – Description of the PTV xServer; visited June 30, 2016

[3] `http://www.gemfony.eu` – The web presence of Gemfony UG (haftungsbeschränkt); visited July 4, 2016

[4] `https://breakoutlist.com/docker` – Docker Inc. has received $160 million (US) funding at the time of writing; visited July 4, 2016

[5] "Docker in Action", Manning Publications, ISBN-13: 978-1633430235; A comprehensive review of Docker technologies

[6] `https://blog.phusion.nl/2015/01/20/docker-and-the-pid-1-zombie-reaping-problem` – A thorough discussion of the process id 1 / zombie problem in docker containers; visited July 4, 2016

[7] `http://www.golem.de/news/docker-vorschau-auf-windows-server-2016-enthaelt-container-1508-115858.html` – Windows Server 2016 may contain Docker; visited July 4, 2016

[8] `https://gcc.gnu.org/onlinedocs/gcc/Link-Options.html` – Warning not to link statically with libgcc under certain circumstances; visited June 30, 2016

[9] `http://www.etalabs.net/compare_libcs.html` – A comparison of different Linux C library implementations; visited June 30, 2016

[10] `http://www.linuxfoundation.org/collaborate/workgroups/lsb` – The Linux Standard Base project; visited July 4, 2016

[11] `https://hub.docker.com/r/dockercore/libcontainer/` – Home of the Docker-component "libcontainer"; visited July 4, 2016

[12] `https://www.docker.com` – Home of Docker Inc.; visited June 30, 2016

[13] `https://docs.docker.com/engine/installation` – Host systems supported by Docker; visited June 30, 2016

[14] `https://engineeringblog.yelp.com/2016/01/dumb-init-an-init-for-docker.html` – the "dumb-init" simple init system for Docker; visited July 4, 2016

[15] `https://github.com/Yelp/dumb-init` – the GitHub download page for `dumb-init`; visited July 4, 2016

[16] `https://www.ansible.com/containers` – The "Ansible Container" Docker-container build-environment

[17] "Dock Dock Go" – a German-language article on using Docker for the creation of portable applications for Linux; iX Magazin für professionelle Informationstechnik; issue 5/2016, pp. 102ff

[18] `https://hub.docker.com` – The Docker Hub; visited July 3, 2016

[19] `https://refspecs.linuxbase.org/LSB\_3.0.0/LSB-Embedded/LSB-Embedded/iniscrptact.html` – actions required for LSB init scripts; visited October 21, 2016