Available online at www.sciencedirect.com

ScienceDirect

Procedia CIRP 72 (2018) 659–664

www.elsevier.com/locate/procedia

51st CIRP Conference on Manufacturing Systems

Using docker for factory system software management: Experience report

Richard Senington[a], Balazs Pataki[b], Xi Vincent Wang[c]

[a]*Högskolan i Skövde, Högskolevägen Box 408, 541 28 Skövde, Sweden*
[b]*Hungarian Academy of Sciences Institute for Computer Science and Control (MTA SZTAKI) H-1111 Budapest, Kende U. 13-17.*
[c]*KTH Royal Institute of Technology, Brinellvgen 68, 114 28 Stockholm, Sweden*

* Corresponding author. Tel.: +4- 650-044-8597. *E-mail address:* richard.james.senington@his.se

Abstract

As factories become increasingly computerised, and with the increasing interest in Cyber-Physical-Systems and the Internet-of-Things, the issues of software management, deployment, configuration and integration are expected to become increasingly important. This paper reports on the ongoing experiences of using the Docker container technology in a major EU research project targeting smart factories. Docker is used to distribute, deploy and manage the configuration of multiple software modules between multiple teams and demonstrator sites in multiple locations, where each module can use its own mixture of protocols, programming languages and platforms.

Keywords: System Integration, Docker, Software Containers

1. Introduction

It is expected that in the future our industrial manufacturing systems will be required to be more flexible than ever before and that this flexibility will be achieved through closer integration of the IT and software infrastructure with the machinery of our factories. In this environment the software systems could be built of many different components serving very different functions, including databases, management interfaces, automatic optimisation and planning and robot control. These software components need not be provided by a single software supplier and hence frequently make use of different software infrastructures, platforms and data formats [1], while needing to be configured, integrated with one another and deployed throughout the factory. The requirement of flexibility in the manufacturing system, now dependent upon these software tools, means that it is unwise to expect the configuration or the deployment of the components to remain unchanged during the lifetime of even a single production run.

These issues of heterogeneous software development and deployment are mirrored in Symbio-tic[2], a recent EU research project into human-robot collaboration. The objective of this project is to explore how to enable humans and robots to work closely together in order to utilise the advantages of each to create better productivity than either alone. The system is further empowered to manage a production process and allocate a set of open tasks to all available resources (human, robot or both at the same time) to achieve the best results possible. The resulting system consists of a number of different components providing the following functions;

- track the location of the human workers,
- provide appropriate instructions to the human workers,
- control the robots and robot tools,
- handle avoiding collisions between the humans and robots in close proximity and
- construct efficient work schedules taking into account the locations and capabilities of the human workers and robots.

The project is composed of teams from both universities and companies spread across the EU. To best utilize each teams existing experience and expertise, and to speed the process of development, each team was allowed to make their own choice of technologies for the modules they were developing. This mix of technologies is broadly analogous to the result of industries purchasing different *best of breed* tools from different suppliers and hence makes our problem one that industry itself must also cope with. The modules of the system, their interconnections and technologies can be seen illustrated in figure 1, and include;

- both Linux and Windows operating systems,
- a mixture of programming technologies such as Java (with and without Spring), Javascript/NodeJs and C++
- and several databases.

Finally this mix of heterogeneous tools are required to be integrated, configured and deployed to both demonstration and test systems in 4 locations around the EU. This is a complex task and one that we expect to appear more in manufacturing industry as advanced IT systems become more widespread. It is also a task that provides ample opportunity for human error and the project sought out a tool to alleviate these difficulties.

With these tasks in mind the consortium considered possible approaches to support the distribution and deployment of software for the project. While install shells are an option it was decided that these require too much development time and tend to be tied to specific systems, such as Windows installer programs. Virtual Machines (VM) were also considered but were thought to be unwealdly due to both the heavy processor load they incure and the size of the virtual machine images themselves. Containers (operating system-level virtualization) were then examined. These have found significant use in shared computing environments and especially cloud computing where they provide an alternative way for customers to deploy software which can subsequently be managed and moved by the service provider to optimise their own infrastructure usage, in a similar way to VMs. Containers are however more lightweight, allowing software to run directly on the CPU and share the Kernel services while preserving key properties of virtualisation such as system resource management and isolation from other running processes. These containers seemed to provide the balance that was sought, with better performance but still providing isolation of individual modules from one another, and relatively easy deployment. The Docker [3] platform was chosen for the project because although it is not quite as fast as some other container technologies [4] it offers greater cross-platform capability (Windows, Mac and Linux where most will only support one of these) and a large existing public ecosystem of modules.

The Docker system was first made available in 2013 and in addition to being used in cloud computing [5,6] has also been used for the distribution of research projects [7] and as a framework for parallel processing. In line with containers in general the key features of Docker that made it attractive for this project were;

- it is not a full VM, meaning that while it isolates your program from other programs on the same system it is not as slow as a full VM [8]
- the isolated environments mean that the requirements of each component, including library versions, cannot conflict
- it packages the dependencies of each component so that they do not need to be installed manually at each site
- it enables fast distribution and activation of the components

While useful, the Docker tool is not without drawbacks. Firstly it should be noted that while the overhead of use is low it is still present and this could be an issue for industrial control software that requires very fast responses. Secondly that the use of a further platform in a software stack presents a new security risk for the system [9]. Finally, while the project has found that the active community and large ecosystem of public projects provides a great way to develop quickly these themselves present a significant security risk, though one we hope will be overcome in the future though a system of container certification. These dangers are believed to contribute to the lack of examples of Docker being used in the manufacturing and heavy industry sectors although there are examples of experimentation in robotics [10] and for IoT [11] which could find use in future smart factory systems.

This paper will begin by giving a more in depth introduction to Docker and the usage of Docker before describing the experiences of using Docker to manage the deployment of software in this research project.

2. An introduction to Docker

Docker provides the ability to package and run an application in a loosely isolated environment called a container. Containers are lightweight because they don't need the extra load of a hypervisor, but instead share the host machines kernel but with strict limits on how much of the machines resources they can see and/or use. This means one can run more containers on a given hardware combination than if one was using virtual machines [12].

Containers provide a number of advantages over virtual machines.

- Lightweight. Containers provide operating system virtualization as opposed to hardware level virtualization available in VMs. This means that all containers share the low level capabilities of the host operating system - like memory management, process management, I/O, etc - while providing isolation of processes running in each container, with their own separate dependencies.
- Native performance. Because the low level operations of containers are managed by the same host operating system, the processes running inside the containers run at the same speed as any other process on the host computer. Every process run in containers is actually a process in the host operating system and can be handled as such. For example, they can be listed with the 'ps' command in Linux. This should make them attractive to industry which requires quick response times from their control software.
- Require less machine power. Since containers don't require full hardware virtualization the same host machine can host many more containers than virtual machines.
- Faster "booting". Starting up a process inside a container is nearly the same speed as starting the process on the host computer, minus some overhead of docker itself. In case of a virtual machine, starting just a single process inside the VM requires an actual boot of a separate operating system first and then running the process.

Despite these advantages of containers there are cases where VMs have preferable properties, for example if the software environment to be deployed is heterogeneous from an operating system point of view. For example, if deployment requires both Linux and Windows software to be run side by side docker - or any other container technology - won't be able to support it. In these cases a full blown VM is required to run Linux inside Windows or vice-versa. In situations where the system architecture is based around the deployment of Virtual Machines (such as is found in many cloud computing providers), con-

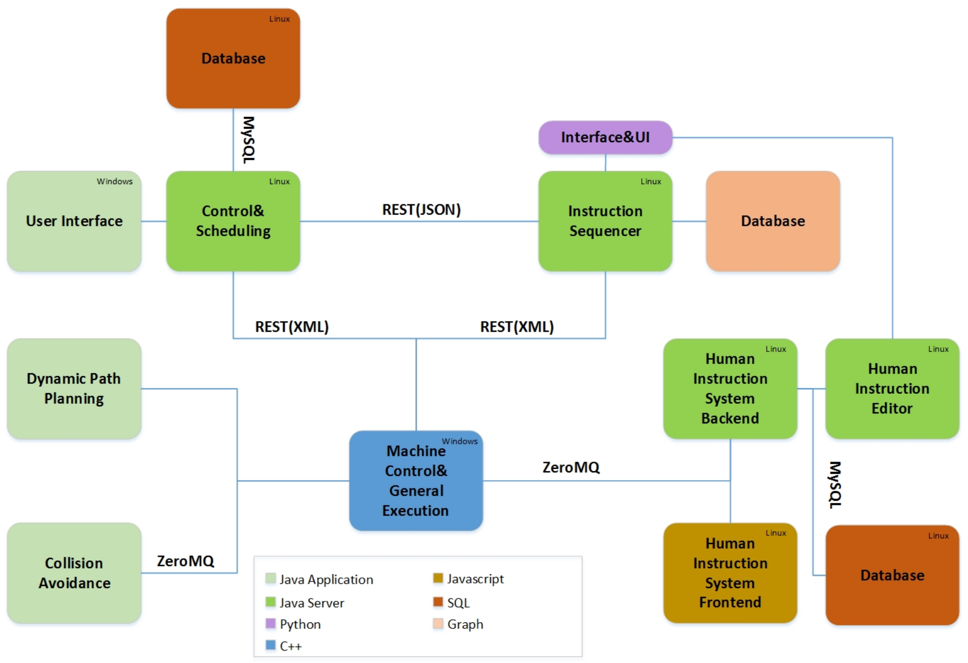

Fig. 1. A summary of the architecture of the project, illustrating the mixture of components, technologies and protocols used.

tainer technology can still be used to group together services which require a particular class of environment (for example x86 Linux), rather than having a full virtual machine for each one. It should also be noted that VMs are a more mature technology and this track record is likely to be more attractive to manufacturing when assurances of reliability are sought.

2.1. Operations of Docker

Docker was originally based on LXC (Linux Containers), which is the built-in container technology available in Linux since 2008. Over the years Docker has been ported to Windows and macOS. On Windows Docker can work with Windows Containers natively, while it requires virtualization to run Linux Containers. In case of macOS the operating system itself doesn't provide containers and thus virtualization is required to run Linux Containers. In case of both Windows and macOS, however, Docker may use the built-in hypervisor technology (Hyper-V for Windows and HyperKit for macOD) to provide virtualization, which is more efficient than a generic VM environment like Oracle's Virtual Box, which is also supported by Docker [12].

2.2. Images vs containers

So far we only talked about containers, which are actually runtime instances of an image. An image is a stand-alone executable package that includes everything needed to run a piece of software, including the code, a runtime, libraries, environment variables, and configuration files. What the image becomes in memory when actually executed is what we refer to as container throughout this paper.

Docker manages both images and containers. It supports downloading ready made images from repositories like the Docker Hub [13]), but also creation of custom images. Docker Hub provides images for 100,000+ free apps, which practically include all major open source systems including MySQL, Postgres, Java/Spring, NodeJS, etc.

Creating a container based on a Docker Hub image is a 2 step process

1. Download the necessary image, eg. an Ubuntu environment:
   `docker pull ubuntu`
2. Run a process inside the container:
   `docker run ubuntu echo 'Hello world!'`

In this example the image chosen was *Ubuntu*, which is really a collection of executable files providing some interface to Kernel functionality, just as in Linux generally. In this case the single element we are interested in is the `echo` command. This is run with the parameter `Hello world!` is in a container created based on the *Ubuntu* image downloaded in step 1.

2.3. Connecting containers

Modern large scale applications are often composed of several modules (each of which can be of significant size and complexity of course) which must be able to talk to one another. Where modules are deployed as Docker containers on the same physical machine Docker provides a default bridge network called 'bridge' created automatically when you install Docker. The Docker daemon also runs an embedded DNS server which allows for DNS resolution among containers by container name. Every container is automatically configured to use this bridge but Docker must be told to allow such connections when the container starts.

For example a backend Java server in one container must be able to access the database running in another container. These could be connected like this:

```
docker run   --name=database
             postgres
docker run   --link database
             --name=server
             javaserver
```

These two command start up a postgres database and Java server. From inside the 'server' container the Java server may access the postgres process on the standard port using the name 'database' and the database cannot be connected to from anywhere else. It is of course possible to expose a Docker service to outside processes and this is done with the '-p' option, which maps a port from the host environment into the container environment.

2.4. Storage and file access

Containers may store data in a number of ways. The simplest of these is to make use of the Containers own writable storage space and this is kept isolated from other running processes providing a degree of security, however it will be deleted when the container is deleted. The second method is to create a *volume*, a Docker managed and named writable space. This has the advantage of being able to be shared between running containers. The final alternative is to mount a file or folder of the host system within the container. This tends to be the least secure, since it is now possible for external processes to change the files and the docker process to interact directly with the host system. It can however be used with a read-only flag, a method that proves useful for managing configuration files. More generally it is likely that a database, using the standard network protocols, will be used, though when that is run within a container it is expected that volumes or host space will be used for persistence.

The syntax for mounting volumes and host folders is to make use of a *-v* flag followed either by a folder on the host or a volume name, and then where it will be mapped to. For example:

```
docker run  --name=server
            -v ~/project/config:/config:ro
            javaserver
```

This command will make the file `~/project/config` in the host system to be available inside the container under the path `/config` in read-only mode. By changing the values in the `~/project/config` it will be instantly reflected for the processes inside.

2.5. Creation of Custom Images

In an actual use case of Docker we need customised images containing the software and its environment we need to deploy. A file called Dockerfile is used for this purpose, which describes what files should be inside the container, what ports it would expose, what volumes it would use/provide and what the default processes should be.

A really simple Dockerfile for creating an image may look like this:

```
FROM microsoft/nanoserver
COPY testfile.txt c:\
RUN dir c:\
```

When built using the command `docker build` this would create an image, which is based on the 'microsoft/nanoserver' image. By basing our image on an already existing one makes image creation even easier, since we don't have to specify all dependencies that are already available in the base image. Once the base image is downloaded docker copies a file from the host to the container and when the container is started it would run 'dir c:\'.

3. Use Case

The key problem that our project[1] encountered that spurred the use of Docker was the question of software module distribution and deployment. This is supported using a mixture of Docker related technologies such as Docker registries for deployment and docker-compose scripts for configuration, customisation and orchestration of groups of containers on individual machines. Scalability has not yet been an issue so clusters of Docker platforms and Docker-Swarm have not been investigated.

3.1. Streamlining Deployment

Before Docker was introduced the consortium had the difficulty of managing many modules, each with their own configuration method, libraries and platform requirements. Each site that wished to make use of a module would therefore need to develop a non-trival level of understanding of the module and manage any potential conflicts between tools and libraries that might arise. The consortium uses Docker's light weight containers to encapsulate each module while expecting not to significantly sacrifice performance [8].

Each module contains appropriate information related to the basic runtime environment (e.g. Linux), the platform that will be used (e.g. Java and/or Maven) and the various libraries that will be needed. Libraries can be provided either by a Linux repository (e.g. Debian's Apt package manager) or through a platform specific tool (such as Java libraries). Docker provides tooling to capture all of this information in a unique *image*, a template for a running container. Notice that this also provides a clean encapsulation of library and software versions and dependencies. This prevents a situation where a user trying to deploy a module installs the wrong version of a library and hence breaks dependencies, thus saving time within the project. This also lends itself to software certification in the future, where a specific mix of tools can be built into one image and subsequently tested and verified.

Installation of a software module therefore becomes the downloading of a single image file and the use of standard docker tools to install it into the local Docker environment. Subsequently a *container* is created from the Image file and activated as a Docker process. These images, containers and processes maintain a strict separation of the dependencies and platforms of each module so that the users need not worry about it.

To illustrate the simplification that Docker brings to our project we have examined the number of instructions that are required to install and activate a selection of modules used in the project and this can be seen in table 2. The table shows that if a manual or native install is carried out then a user is expected to perform a number of separate steps, where as under Docker it is only the installation and activation steps in every case. While it is possible to capture all of these steps in a script file this assumes a uniform underlying platform (e.g. Debian Linux) and when changing platform or properties of a platform the user is likely to be required to correct issues in the script, a problem that is encountered less under Docker.

[1]The architecture of the modules can be seen in figure 1.

	Instructions		Time(s)			
	Docker	Native	Docker		Native	
			Install	Startup	Install	Startup
Module 1	2	6	18.3	77.8	167.1	46.0
Module 2	2	5	21.9	16.7	75.8	14.0
Module 3	2	4	13.7	33.2	6.7	34.3
2 Databases	4	3	19.6	21.0	36.7	0.0

Fig. 2. A comparison of the installation and startup complexity and time costs between Docker and Native versions of the project

Table 2 also includes some *illustrative* time data, and shows that it is quicker to install the images of docker than to perform a native install. For a complete running system the shorter install and restart times are not expected to be a significant benefit however this is of use to our project with rapid development, repeated testing and multiple demonstration sites.

To generate the data in table 2 we wrote code to time how long each step took. Each step is then run 25 times and an average taken. The averages are then summed to give the final times seen here. It should be noted that it has been hard to provide a *fair* comparison, as installation of local images does not require download time where as the native install will tend to make use of standard repositories such as those for Debian Linux. This however is a reasonable illustration of the situation because when performing a manual deployment the user will perform these steps against the global repositories. This could be overcome by downloading all required files for the test, however in many ways this is precisely what the use of Docker images is doing.

3.2. Encapsulating Configuration

Each module handles configuration in it's own way. Some have used parameters on the command line while most have used one or more configuration files in various formats. These configuration files contain settings ranging from the IP address and ports of the other systems to database schema files to database login details. In the initial versions of the modules a programmer had to find the appropriate configuration files in various places in the system and update them correctly.

The approach taken by the project to standardise the configuration process and hide the details from the users was to make use of Docker's support for developer specified *environment variables*. These are passed to a Docker container as it starts up. New scripts were then coded to read these environment variables first and update the default config files where necessary before activating the primary tool of the module.

Additionally Docker hides the networking of the contained software, except where it is specifically instructed to expose it. This has proven to be useful because communication between the modules uses various network protocols with REST being the most popular. Several modules may wish to expose a service on the same port and while these ports are usually a part of the module configuration Docker's networking layer provides a powerful alternative. Using this we can use the standard Docker interface to redirect the conflicting ports at the point where Docker activates the container, rather than having to manage the software's internal configuration.

Finally the use of Docker's interface as a single point of entry for each module simplifies training the users in how to configure the deployments, or for the delivery of standard scripts that configure and start the system.

3.3. Docker for Databases

The Docker Hub provides base images for a wide variety of databases of all kinds. This is used to speed the deployment of databases for the respective modules within this project. Images can be downloaded, moved around and deployed in the same way as other modules within the project. Several databases can be run on the same machine if desired and this has allowed the different teams to develop their own schemas independently and then deploy them at runtime with no changes.

It must be noted that there is a risk with the use of a Database Management System (DBMS) within a container. The use of a container introduces a layer of functionality (however thin) that is outside of the control of the DBMS, and if the container breaks down for some reason then capabilities of the system such as transaction rollback could be interrupted. While this is true, we propose that firstly the use of a cluster for the database with automatic creation of new nodes when old nodes fail can offset these issues, and that for the purpose of a research project this is not a significant threat.

3.4. Encapsulating Module Customisation

One particular module in this project has been built with standard C++ tooling. Distribution of an executable application is possible, however local recompilation is often necessary because this part of the system often requires customisation for specific deployments or use cases. This customisation is achieved by adding a number of code modules and rebuilding the final executable. Local compilation requires significant expertise in the build tools and understanding of the project structure.

Docker has been used to overcome this issue, by providing an encapsulated environment with capabilities for both compilation and execution of the current build of the tool. The environment is delivered with the correct versions of the C compiler, appropriate required libraries, tooling and build scripts. When the image is converted into a running process by Docker a script to rebuild the tool internally is run and the executable created it then also run as the key process. Control of the deployment specific modifications is handled by environment variables as was done with configuration, hence providing a single interface for the user. The user must still understand that they are modifying a C++ program and provide appropriate additional code in the environment variables, however the details of the full project structure is hidden as are the details of build scripts.

4. Conclusion

Docker was introduced to this research project to aid distribution of the software modules. It does this by allowing us to encapsulate the environment of the software such as libraries, interpreters and virtual machines in a single package.

This has been a success, streamlining the deployment process by reducing the work needed to activate the system and reducing the chance of human error in the installation processes. The deployment and activation process has fewer steps than an approach using the native applications and now only uses standard docker commands for activation and configuration through environment variables. This allows for further streamlining through installation scripts that would have previously been time consuming to write.

We have also experimented with compilation within Docker containers in order to prepare specialised executables based on configuration information passed at start up time or updates requested by additional services. While this is not heavily used by the project it does provide the possibility that in future modules could compile customisations into their execution environment allowing for improved performance through static code optimisations rather than relying on configuration or JIT compilation. This would provide better predictability of the runtime performance of code, relative to a specific configuration, than other more dynamic compilation methods. This could be further improved with respect to relatively slow compilation processes that can give rise to very high performance executables, such as using super compilation [14].

Despite our interest in Docker we can see several issues with the technology that should be addressed. The most obvious are related to the performance costs container technology which, while not as significant as virtual machines, is not zero. It has been shown recently that previous benchmarks on Docker struggle to be clear and reproducible [15] and this should be further addressed, in addition to considering whether the performance impact of Docker varies depending on the technology it contains (e.g. Java, C). From the perspective of this paper a particular question should be asked, that of whether the performance impact could make Docker unsuitable for use in a smart factory environment due to safety concerns through lack of responsiveness of physical tools.

A second issue with Docker is that of stability, both runtime stability but also the stability of the technology and protocols, which continue to evolve. Changes between versions of Docker which break backwards compatibility could be seen as excluding it from production engineering for the foreseeable future and this question should be examined in more detail. The runtime stability was previously mentioned with respect to databases. While databases are perhaps a special case that should not be placed within Docker, while processing modules could be, there is an open question here, of how often Docker containers will break and if it is significantly worse than not using the containers at all.

Despite these concerns we feel that there is definite benefit to either Docker or a related container technology in the smart factory environment, supporting the deployment and management of control software on distributed computing hardware, in line with existing research on the use of Docker for edge computing[5]. This would in turn support greater flexibility in factories in general by enabling rapid changes to control software as needed by the process, and this in turn then supports the trend towards mass customisation in manufacturing industry in general. We feel that our project is an opportunity to try out Docker in an environment similar to that of a production environment. With this in mind we intend to further investigate stability and security of the Docker runtime and the impact it has on the performance of complex control software. A final capability of Docker that we wish to explore is its network APIs, and related GUI implementations, for remote server administration. We see this as a way to further streamline our deployment and management operations.

5. Acknowledgements

This paper is based on work performed within the research projects SYMBIO-TIC [637107] sponsored by the European Union and YOU2 [20130303] sponsored by the Swedish Knowledge Foundation.

References

[1] Wang, X.V., Xu, X.W.. DIMP: an interoperable solution for software integration and product data exchange. Enterprise Information Systems 2012;6(3).

[2] The Consortium of SYMBIO-TIC project, . Symbio-tic. 2015. URL: http://www.symbio-tic.eu/.

[3] Anderson, C.. Docker [Software engineering]. IEEE Software 2015;32(3):102–c3.

[4] Kozhirbayev, Z., Sinnott, R.O.. A performance comparison of container-based technologies for the Cloud. Future Generation Computer Systems 2017;68:175–182.

[5] Ismail, B.I., Goortani, E.M., Karim, M.B.A., Tat, W.M., Setapa, S., Luke, J.Y., et al. Evaluation of Docker as Edge computing platform. In: 2015 IEEE Conference on Open Systems (ICOS). 2015, p. 130–135.

[6] Chung, M.T., Quang-Hung, N., Nguyen, M.T., Thoai, N.. Using Docker in high performance computing applications. In: 2016 IEEE Sixth International Conference on Communications and Electronics (ICCE). 2016, p. 52–57.

[7] Boettiger, C.. An Introduction to Docker for Reproducible Research. SIGOPS Oper Syst Rev 2015;49(1):71–79.

[8] Felter, W., Ferreira, A., Rajamony, R., Rubio, J.. An updated performance comparison of virtual machines and linux containers. 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) 2015;00:171–172.

[9] Mohallel, A.A., Bass, J.M., Dehghantaha, A.. Experimenting with docker: Linux container and base OS attack surfaces. In: 2016 International Conference on Information Society (i-Society). 2016, p. 17–21.

[10] Gonzlez-Nalda, P., Etxeberria-Agiriano, I., Calvo, I., Otero, M.C.. A modular CPS architecture design based on ROS and Docker. International Journal on Interactive Design and Manufacturing (IJIDeM) 2017;11(4):949–955.

[11] Rufino, J., Alam, M., Ferreira, J., Rehman, A., Tsang, K.F.. Orchestration of containerized microservices for IIoT using Docker. In: 2017 IEEE International Conference on Industrial Technology (ICIT). 2017, p. 1532–1536.

[12] Docker Inc, . Docker documentation. 2017. URL: https://docs.docker.com.

[13] Docker Inc, . Docker hub. 2016. URL: https://hub.docker.com/.

[14] Bolingbroke, M., Peyton Jones, S.. Supercompilation by evaluation. SIGPLAN Not 2010;45(11):135–146.

[15] Casalicchio, E., Perciballi, V.. Measuring docker performance: What a mess!!! In: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion. ICPE '17 Companion; New York, NY, USA: ACM; 2017, p. 11–16.