DOI 10.12694/scpe.v17i2.1158
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 17, Number 2, pp. 91–102. http://www.scpe.org

AN EXTENSIBLE SOFTWARE-AS-A-SERVICE SOLUTION FOR DISTRIBUTED INFRASTRUCTURES

JEDRZEJ RYBICKI AND BENEDIKT VON ST. VIETH*

Abstract. Distributed research infrastructures embrace resources, services, and users. From those, services display the highest churn rate. The reasons are twofold. First, new versions or new services emerge, gain popularity, and have to be provisioned by infrastructure operators. Secondly, the demand to make research reproducible inclines the users to share the software they used to obtain their results. Coming to grips with these high churn rates is not easy, but has the potential to speed-up scientific discovery.

In this paper we will describe an extensible Software-as-a-Service (SaaS) solution conceived to facilitate seamless exchange, evolution, and deployment of software services in distributed environments. In opposite to ordinary SaaS solutions, the portfolio of the software services in our approach is extensible. We implemented means for users and developers to make their services readily available in the distributed infrastructure with minimal overhead. In this process, not only the actual software is made available but also the knowledge of how to install and configure it is conveyed, thus running instances of the new service can be provided right away. We will share the experience gained during the implementation of the extensible SaaS solution for DARIAH-DE research infrastructure. It was used to provision on-demand instances of software used in digital humanities. Subsequently, the same solution was reused in a completely different context to provide on-demand instances of UNICORE Grid middleware for training and testing purposes. The operation of our extensible SaaS, yielded additional requirements and extensions. In particular, the need for monitoring and measuring the resource usage were identified and addressed. The presented insights can help to address the problem of exchange and provision of scientific software. Other distributed infrastructures can incorporate them to improve the scalability, maintainability, user-friendliness, and sustainability of their platforms.

Key words: service sharing, distributed systems, cloud computing

AMS subject classifications. 68M14

1. Introduction. Digital methods have become commonplace in modern science. Their popularity is amplified by the recent establishment of the so-called data-driven science, which is expected to provide scientific insights by applying software solutions on digital data. The obvious prerequisite for such approaches is the availability and accessibility of the data. A problem which is acknowledged and addressed (if yet-not-completely-solved). The less obvious problem is how to share the scientific software used for analyzing the data. We have presented the initial solution to this problem in our priory publication [23]. In this paper we will recap our experiences and show that the paradigm is also applicable outside of its original context.

Research software is often a custom-made solution, developed in the course of a research project as an after-effect of answering concrete scientific questions. Other researchers might, nevertheless, benefit from reusing it to solve similar problems or for the validation of the results by applying the software to a different set of data [14]. Sharing software has the potential to accelerate scientific discovery. But even more important is the fact that software which is not sharable is, in the long run, not sustainable. Scientific discoveries done with such a software are, in worst case, not reproducible and thus useless. We believe that sharing data and software is a corner stone for doing science today. In this paper we will brush aside the social challenges of sharing software and data, and concentrate on technical solutions to facilitate such sharing.

It should be stressed that open software repositories like `GitHub` or `sourceforge` do not solve the stated problem completely. The software is only usable when it runs. Installing, configuring, and satisfying the software dependencies often proves to be harder than expected. Especially, when it is conducted on a different platform than it was developed on. Furthermore the knowledge on how to perform these steps is often only present in the heads of the software authors or within the project it was developed for (e. g. closed wiki). Hence the first requirement for effective software sharing is the availability of the abstract, platform-independent, yet executable deployment descriptions. Such descriptions could be, for instance, system images, executable installation scripts, configuration management tools, or container-based solutions. We will discuss the applicability of these approaches in Sect. 2.

The proliferation of distributed research infrastructures [5, 17] adds a new dimension to the problem of sharing scientific software. Such infrastructures offer a quick and cost-effective way of sharing resources. They

*Forschungszentrum Juelich GmbH, Juelich, Germany ({j.rybicki, b.von.st.vieth}@fz-juelich.de)

excel at enabling access to data and often provide some compute resources. The later is offered either in a Infrastructure-as-a-Service or Software-as-a-Service manner (for the definition of the terms see [19]). None of these approaches are optimal from the user's perspective. IaaS solutions put quite a large burden on the users: they have to install, configure, and maintain not only the software they require but also the whole system (e. g. do system upgrades). Furthermore, the created instances cannot be easily shared: this remains in power of the user who created them and is usually technically challenging. The higher-level abstractions in form of Software-as-a-Service are much more attractive to researchers: The effort on their part is reduced and they can directly apply the software in their scientific endeavors. But the effort still exists: The installation work has to be done by the computer centers in the infrastructure. From software sharing perspective the problems are twofold. Firstly, due to the rapid development of digital methods, researchers constantly require new versions or new types of software in the SaaS portfolio. Secondly, the distributed research infrastructures span across multiple resource providers, thus high level of heterogeneity has to be coped with, making the installation harder. To this end, the demanded software deployment descriptions must be platform-independent and portable. To account for the constantly changing requirements, the descriptions should be executable. Finally, the system administrators must be able to audit them, to verify that they are correct, trustworthy, and secure. In this paper, we will show how the deployment descriptions can be used to build a extensible Software-as-a-Service solution, which allows for an easy extension of the offered software portfolio.

The motivation for looking into a particular software service can be manyfold. In many cases it can be just curiosity, a need for testing its applicability, or training purposes. Our solution is primary made for these applications. We will show in this paper that it can even provide on-demand instances of quite a complicated Grid middleware stack. They are configured to start, and provide basic functionality, making it possible to get an impression of how particular software works. The configuration might, however, not necessarily be sufficient for long-running, multi user, installations. There are examples in the literature of more sophisticated, higher-level approaches, e. g., [2] which can be used complementary to our extensible SaaS.

The high development pace of cooperative science together with the increasing importance of interdisciplinarity, bears one more use case for sharing research software: exchanging software between e-Infrastructures. This use case reinforces the previously mentioned requirement of making software deployment descriptions executable, auditable, portable, and sharable. The potential users of research software from different disciplines will have to invest some time to understand how to use it. The time should not be increased by forcing the user to understand how to install, and deploy the software.

This paper is structured as follows. Related work is discussed in Sect. 2. We provide a short introduction in the technology which we will use as a basis for the software sharing in Sect. 3. Section 4 presents how we used executable software deployment descriptions to implement an extensible Software-as-a-Service solution for DARIAH-DE. Section 5 provides a short summary of further applications and extensions of our solution. We conclude our paper with a summary in Section 6.

2. Related work. Let us first reflect on how software reuse can be achieved. In the first step, the scientific software has to be made discoverable. This step involves some social challenges, for instance willingness to share and use software, or efficient communication between the parties, involving software citation. In this paper, however, we focus on the technical challenges and assume that open-source repositories are sufficient for discovery. As stated above, the software is only useful if it can be applied to a new problem, and this is only possible when the software runs. Hence, the discovered software has to be installed. The process is conducted either locally on the researchers machine or (in a more modern fashion) on a remote machine running somewhere in the e-Infrastructure Cloud. The installation usually involves the configuration of the operating system (OS) and the provision of run-time environment, additional libraries, and actual software. The knowledge on how to achieve this might be, to some extent, conveyed in the service documentation. Experience shows, that the documentation is seldom up-to-date and does not cover all the corner cases encountered in the real-life deployments e. g., library dependencies or support for differing operating systems. When it comes to a migration of the service, the process must be started again, often by a different person. Even such a down-to-earth, recurring situation like the OS upgrade for the underlying machine might render a service unusable and result in the need for a re-installation. Since the process described above is quite laborious and error-prone, let us now discuss ways in which it can be either simplified or conducted in such a way that other researchers can

use the software without repeating the installation process.

2.1. Virtual machines. One way of sharing "already-installed" software is to prepare images from which virtual machines can be spun off. The most obvious case for such applications are Clouds offering an IaaS interface [4, 20], although local deployments of images are conceivable as well. VM image preparation involves repeating the above steps in a virtual machine while accounting for special properties of the given Cloud. The later is often subsumed under the term: *contextualization*. When the image is created and uploaded to a Cloud image repository, it can be instantiated by other users to obtain running instance of the application. VM images suffer, however, under some limitations. First of all, images are black boxes: one has to trust the creator and has very limited capabilities of auditing the actual content of the image. Thus it is hard to safe-guard that the images do not include some malicious software or are not misconfigured so that running instances can be easily hacked or destroyed by malicious third parties. Virtual images include a complete software stack comprised of the required software, dependencies, operating system, and kernel. This results in a high overhead in terms of both image size (what makes the sharing of images harder) and performance. The inclusion of the operating system results in the need for periodic updates of the image: for instance to apply new security patches. The further problem with images is, that they are infrastructure-specific: they strongly depend on the hypervisor used and contextualization solution provided in the particular Cloud. Clouds include their own image repositories but moving images between the infrastructures, i. e., between different Clouds, remains an unsolved problem. A problem with obvious ramification for software sharing.

2.2. Configuration management tools. One way to create trustworthy and reproducible service deployments that are, to some extent, infrastructure, hypervisor, and OS agnostic, is to use configuration management tools like Puppet [21] or Chef [6]. With these tools it is possible to describe the service deployment in a declarative way. One defines which packages should be installed, which configuration changes should take place or which commands should be executed instead of applying all these steps manually. The tools can be used to provide a generic description of the service deployment that interested partners can apply to virtual machines or servers. Configuration management tools do not offer any isolation nor virtualization by themselves. Here the same limitations with respect to overhead of running VMs apply. On the upside, the descriptions are human-readable and can be reviewed and adjusted before the actual deployment, this contributes to their trustworthiness. Configuration management tools enable reproducible service deployments, but they fail to some extent at abstracting the underlaying platform, for this the developer has to spent additional effort.

One example for this is the deployment of an Apache HTTP server [3], a software often used to provide webservices. Listing 1 shows an excerpt from a deployment description that is used to deploy the software by installing (section `package` on Listing 1), configuring (`file`), and starting (`service`) it. Puppet brings an abstractions for package managers and it also provides wrappers for different `init` systems, but it has some limitations with respect to the different package, path, and service naming. To make them portable, one has to prepare deployment descriptions for all the supported platforms. The initial lines of the Listing 1 show how to cope with the fact that the same software is available under different names in RedHat and Debian repositories.

Listing 1
Puppet example for deploying Apache httpd

```
if $::osfamily == 'RedHat'{
  $user = 'apache'
  $name = 'httpd'
} elsif $::osfamily == 'Debian' {
  $user = 'www-data'
  $name = 'apache2'
} else {
  fail("Unsupported os: ${::osfamily}")
}

file { "/etc/$name/$name.conf":
  owner => $user
  ensure => 'present'
}
package { $name:
  ensure => 'latest'
```

```
}
service { $name:
  ensure   => 'running'
}
```

We establish that tools like Puppet solve a slightly different problem. They are perfectly suited for managing the configuration (changes) of software running in a distributed infrastructure. For this purpose, they allow for configuration changes of the running machines (such changes are then automatically applied on the managed machines). The scientific software sharing, on the other hand, will be rather used to create quickly disposable instances of given services. The instances will be used, e. g., to verify the researcher's hypothesis or validate previous results and they will be discarded afterwards. We argue that, at least in some of the cases, there will be no need to manage the configuration changes in one instance, for the modified setup new instance will be spun off. Researcher will not be willing to spent much effort into the configuration management when it would be possible to simply, quickly, and cheaply create new instances. This assumption is especially true in Cloud environments. Since running instances costs money, they incentivize an approach of "run-and-discard."

2.3. Dynamic provisioning of research software. Another dimension of our work is the actual problem of software sharing and exchange regardless of the technology used. The problem is not new. The advent of affordable pay-per-use Cloud offerings in form of either public [4] or private Clouds [20], paved the way for many solutions to this problem. The works in this field cover a broad spectrum of subjects, starting from the general problem of reproducibility and repeatability, and how Clouds can help to alleviate it [11]. A lot of work was put into providing re-usable tools to support data-driven research, ranging from generic solutions like dedicated Linux distributions similar to bioKepler ones [29], up to Cloud-based provision workflow systems Galaxy Cloud [1]. The recent works on this field [2] concentrate on providing and executing data-intensive workflows. This work bears some similarities with our approach but operates on higher level of abstraction. From a technical point of view, on which we concentrate in this paper, there are no arguments preventing the usage of our solution to provide software packages constituting the workflows.

3. Container-based virtualization. Container-based virtualization solutions like Linux VServer [28] or Docker [9] became very popular for building, shipping, and running applications across many machines. They constitute the perfect basis for an efficient sharing of scientific software. In this section, we will first provide the reader with some basic information on how Docker works, and also argue why it better suits our use cases than the previously described system images and configuration management tools. The section will be concluded with a proposal on how the software sharing life cycle could be implemented in a distributed infrastructure.

3.1. Introduction to Docker. Docker is used as the basis for our software-sharing solution. This section will equip the reader with enough information about this technology to understand the rest of the paper. More details can be found in the extensive Docker documentation [9].

Docker is a lightweight, open-source, virtualization solution. In contrast to full-stack virtualization solutions, Docker images do not include a guest operating system kernel. It lowers the overhead in terms of both performance and size of the images, allowing near-native performance. Docker is based on mature Linux kernel technologies (`cgroups` and `namespaces`, among the others) to isolate independent application containers. Docker images use layered AuFS file system, enabling sharing libraries between images on one hand, and inspecting the changes done in the images, on the other. The later feature permits a rudimentary provenance tracking of images.

Let us now briefly discuss the Docker terminology and usage. Applications running with Docker are called containers, they are created from images. Images should be understood as hierarchical templates. It is possible to start with one image, add some software, and save the result as a new image. It is also easily possible to share images by using image repositories either private or public ones, like the `Docker Hub`. There are two main ways of shipping applications with Docker. In the first one (let us call it interactive), the developer starts a basis Docker image (e. g.: official Debian Linux image) and installs application, its dependencies, and everything else by issuing ordinary Linux commands. As soon as the application is installed, a snapshot of the running container can be created. Such a snapshot is, in fact, an image from which new containers can be created. An alternative way of creating images follows a declarative approach. The developer can describe the installation steps in a

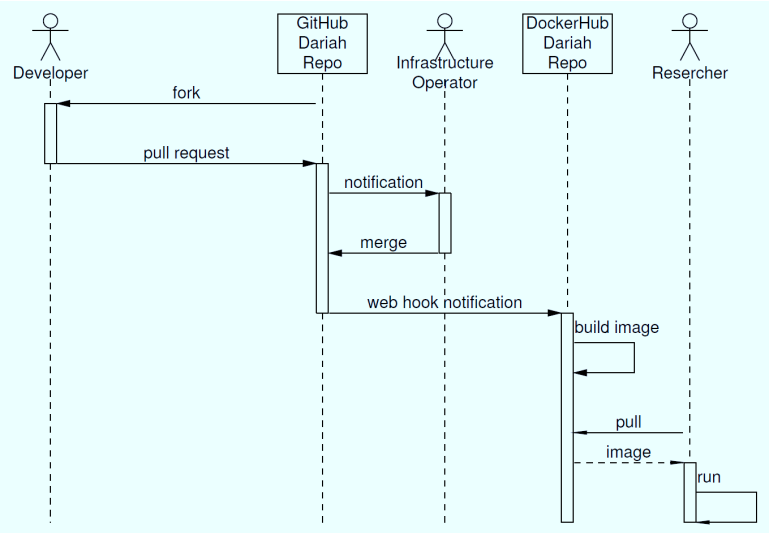

FIG. 3.1. *Proposed software sharing life cycle*

special `Dockerfile`. An image is created from such a description by issuing the `docker build` command. A generalization of the declarative approach are automatic builds. To setup an automatic build one has to publish the `Dockerfile` in a public code repository (like `GitHub`). Afterwards a web hook has to be created, so that each time the `Dockerfile` is modified, a new image build is automatically triggered. Regardless of the way in which an image was created, it is possible to view all the changes done in the image during the installation of the software. The automatic builds go an extra mile in terms of the trustworthiness of images: each user can review not only the content of the image but also the way they were created, since the description is publicly available.

Docker has a very important abstraction of data volumes which allows to separate software and data. The data are not included in the image, as it often happens in VM images. Data volumes, on the other hand, enable an easy injection of the data into the running containers. Hence the software in a container can be used for analyzing various sets of data. The separation of the software and the data diminishes one of the common barrier for sharing software: Researcher might be afraid of leaking out the data in the software they publish.

Docker is available on almost all Linux-based platforms, there is a support for MacOS and recently announced support on Windows platforms. Since the images encapsulate the application and all its dependencies, starting new containers on a new system is like starting a statically linked, self-contained program: no modification in the host system are required.

3.2. DARIAH-DE software sharing life cycle. The first application of our extensible SaaS solution took place in the DARIAH-DE distributed, research infrastructure. This use case is pretty typical for software sharing, thus explaining it in detail will help the reader to understand the trade-offs of software sharing, real-world applicability of Docker, and the design of the actual extensible Software-as-a-Service which we present later in this paper. Most of these points are relevant also outside of the DARIAH context.

DARIAH, the Digital Research Infrastructure for the Arts and Humanities, aims to enhance and support digitally-enabled research and teaching across the arts and humanities. It comprises a number of national initiatives, there is also a Germany-based effort, DARIAH-DE [5], where the described work was conducted. Efficient software sharing is essential for digitally-enabled research of its users. Also sharing software between infrastructures is becoming relevant in this context [16]. Software developed in one part of DARIAH might be passed over to and reused by other national efforts.

The technology for creating executable and deployable descriptions of software must be integrated into the scientific workflows and play well with existing e-Infrastructures. On Fig. 3.1 we present how such an integration could be conducted. We differentiate between three roles: developers, infrastructure operators, and researchers.

We have also two central components: a code repository, where the deployment descriptions are stored, and `Docker Hub` repository where the images are stored and pulled from.

The process of making scientific software available kicks off with the developer wanting to advertise and make his software available. He can propose the deployment description by follow typical `GitHub` cooperation workflow. It includes forking of the official `GitHub` repository of DARIAH-DE Docker files [13], adding a new description (or updating an existing one), and creating a pull request.[1] The operators of the repository are notified about the pull request. They can review the proposed solution, adjust it when required, or even reject it if it does not work. The DARIAH-DE `GitHub` repository is connected to the `Docker Hub` via web hooks. In short, this mechanism triggers an HTTP request to the `Docker Hub` each time the `GitHub` repository is modified. Upon such a request, a new build of the provided Dockerfile is started. The infrastructure operators can view the details of the build and act accordingly in case of a failure. Successfully built images are published into the official DARIAH-DE image repository [10] in the `Docker Hub`. The researchers can explore the repository, select the software they are interested in, pull the images and start it on their machines. Pulling and starting requires just one command (`docker run`). The `Docker Hub` allows for storing additional information about the software. The developer can describe the typical applications or provide a link to a more extensive documentation.

The workflow described above can be repeated multiple times, e. g., each time a new version of the software is released. Moreover, the roles can be distributed or handed-over: Both repositories allow for registration of organizations. Such organizations can have multiple administrators. Hence it is not required to designate just one infrastructure operator but rather a group of people can be delegated for this task. Similarly, all the `GitHub` users can fork, and modify the Dockerfiles. The trustworthiness of the images is guaranteed by the reviews conducted by the operator who has to accept the pull requests. Of course, each researcher can view the publicly available descriptions. The images (and Dockerfiles) are publicly available and they can be used by the researchers from outside of DARIAH-DE. All interested parties can found Dockerfiles in DARIAH-DE `GitHub` repository [13] and Docker images can be retrieved from DARIAH-DE `Docker Hub` [10].

Fig. 3.1 depicts the simplest workflow, which ends with a local deployment of the software. It is sufficient to explain how the workflow works but clearly the local deployment is not always the optimal solution. In the next section we will show how an extensible SaaS solution can be build, based on the proposed workflow.

4. Implementing extensible Software-as-a-Service. So far we have shown that Docker provides a means for a fast deployment of software and how `Docker Hub` and `GitHub` can be used to define a workflow for sharing researcher software in a decentralized yet trustworthy manner. In the optimal case, the user has to issue just one command (`docker run`) to obtain a running instance on her system. The optimal case requires some changes on the users machine, at least the installation of Docker is required. Even if easily possible, the local deployments are not always the best option. One reason for the remote deployments would be efficiency: remote instances can run close to the resources they require. Many data-driven projects require long-lasting software runs. This is hard to perform on the local researcher's machine. Fortunately, e-Infrastructures like DARIAH-DE provide resources which can be used by the researchers for this purpose.

In this section we will show how the availability of deployable software description (as those provided by Docker) can facilitate an extensible Software-as-a-Service solution. This service will offer all the benefits of the SaaS approach (i. e. very low effort is required to use scientific software shared by other researchers) whilst allowing for quick and easy extensions of the software portfolio. We present our prototype solution which runs on low-level compute resources already available in the DARIAH-DE infrastructure. Our goal was a system which provides the researchers with an instance of the software she requires in just one request through a web page.

4.1. Actors. Fig. 4.1 depicts the architecture of the implemented extensible Software-as-a-Service. It includes lots of loosely-coupled components. Messaging is used for exchanging information between components in an asynchronous way. In production we use a `amqp`-based product RabbitMQ [22]. Let us now briefly discuss the roles and responsibilities of the single actors of our architecture.

The *Facade* is the entity facing end-users. It is an entry point from which the commands (like create a new instance of a given software product) are issued. The Facade is stateless and it is possible to run multiple

[1] Readers who are not familiar with the `GitHub` commands are referred to the documentation [12]

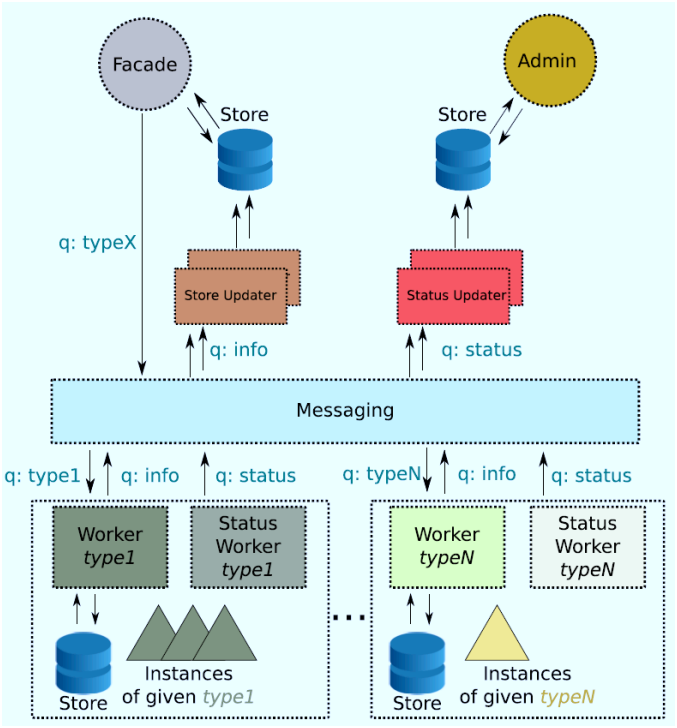

FIG. 4.1. *Architecture of the extensible Software-as-a-Service*

Facades simultaneously. They can offer interfaces of different types, e. g., a html-based for browser access or a json-based programmatic API.

The Facade has access to the messaging bus and a read-only access to the *Store*. Store is a system-wide cache with information about:

1. types of service registered in the system,
2. instances running in the system.

The content of the cache is updated by the *Store Updaters*. These entities subscribe for information about creation of new instances and registration of new types. Upon reception of such messages, Store Updaters write the information in the Store. It should be stressed that the content of the Store itself can be lost as it will be periodically republished. Store is solely used to speed-up the response times.

The *Workers* embody the main functionality of the system. Those are the entities which can manage instances of given service types (e. g. instances of `eXist` or `neo4j` databases). Each Worker upon its creation (and later periodically) informs other components about its capabilities (i. e. supported service type) and its current state (i. e. managed instances). Workers receive commands from users (via Facade) through the messaging system. Next section will be devoted to the communication problem and includes example message exchange.

4.2. Communication with workers. The critical part of our system is the communication between the components. It is dynamic, asynchronous, and subject-based. We required the possibility to change the system capabilities during the run-time. In particular, to add new software to the portfolio of the offered Software-as-a-Service. This is done by adding (or removing) Workers. To become active in the system, Worker needs access to the messaging bus. The incoming Worker publishes its "specialty" (i. e. instance type that it can provide) and subscribes for the messages in this instance-type-queue (like create new instance, delete instance, etc.). Listing 2 shows the initialization message published by a worker able to create and manage instances of `neo4j` graph database. Both the name of the worker (`neo4j`) and the description field are visible to the user. The message also includes a status field (and a human readable status message), a timestamp, and (empty

in this case) environment field. The later one is a feature connector, which can be used to request additional information required to create new instances. An example would be a request for the password value for the database.

Listing 2
Worker initialization message

```
{
    "available": true,
    "name": "neo4j",
    "description": "neo4j is a graph database",
    "status": "Worker available",
    "ts": 1447977918.137527,
    "environment": []
}
```

As can be seen on Fig. 4.1, for each instance type supported by the system, there is a separate queue. When a user requires a new instance of a given type it sends a message to the respective queue (via Facade). An example of such a message is presented on Listing 3 and is pretty self-explanatory. Client is able to give instances its own ids by which the instances can be referred to in the future, in this example we used a UUID algorithm to generate such an id.

Listing 3
Create instance message

```
{
    "subject": "create_instance",
    "instance": {
        "type": "neo4j",
        "id": "e6da7ab7-9076-41d7-b0c2-c724a92712ab",
        "environment": []
    }
}
```

The message is routed to the proper Worker, it creates an instance and sends a response to the messaging system. This response contains details required for connecting to the just-created instance of a concrete service. Such details typically include IP address, protocol and port, username and password. An example of a message sent by a `neo4j` Worker in response to a request like presented above is shown on Listing 4. We see that `neo4j` is accessible via both, HTTP and HTTPS. A user can directly connect to her new service instance via provided URL.

Listing 4
Instance created response

```
{
    "subject": "instance_info",
    "instance": {
        "type": "neo4j",
        "id": "e6da7ab7-9076-41d7-b0c2-c724a92712ab",
        "status": "created",
        "created": 1447978020.125227,
        "ts": 1447978920.137527,
        "urls": [
            "http://134.194.198.39:9011",
            "https://134.194.198.39:9010"
        ],
        "environment": []
    }
}
```

Workers republish their state periodically (i. e. instance types that they support and state of managed instances). The timestamps included in the messages are used to implement soft-state status management, workers which do not published their status for a longer time will be consider not active or overloaded. A worker has also a possibility to gracefully leave the system by publishing an explicit non-active status message.

Each worker has a dedicated queue for the incoming commands and all workers share a common response queue (the `info` queue on Fig. 4.1).

4.3. Implementation and operation. We have created a prototype of the extensible Software-as-a-Service called DARIAH Meta Hosting. The implementation was done in Python and uses a set of frameworks and well-known products: Flask for the web front end, MongoDB as storage, and RabbitMQ for messaging. We offer two interfaces: one is a human-friendly web site, the other is a programmatic json-based API. We envision the later one to be used to script automatic deployments of software, e. g., in data processing workflows.

One feature of our design was the division of the system into small, autonomous parts connected by messaging. The advantage of such a design was the ability to change the parts independently, for instance to scale up. We use Docker for deploying single parts and the Docker-based orchestration tool `Docker Compose` [7] to create test deployments including all the parts required. An example of such a deployment of our system is show on Listing 5. It can be used to deploy a testing system (or a production system, after some modifications) on every research infrastructure supporting Docker. Docker Compose is also able to scale out the system by spinning off additional instances. In this self-test, the solution we use for the software sharing was successfully applied to the software we have implemented. Our prototype is deployed on a OpenStack Infrastructure-as-a-Service cloud offered in DARIAH-DE. So far the complete system is placed in one data center, but it should be possible to extend it beyond this one entity. This would only require to make the messaging system accessible from the outside and deploy additional workers in different data centers.

Listing 5
Docker Compose description of a test deployment

```
messaging:
  image: rabbitmq:3.4
db:
  image: mongo:latest
  command: --smallfiles
autho:
  image: httpprincess/authorizator:latest
updater:
  image: httpprincess/store-updater:latest
  links:
    - db
    - messaging
neoworker:
  image: benedicere/metahosting-worker
  links:
    - messaging
  environment:
    - DB_LOCAL_COLLECTION=local-instances
    - WORKER_NAME=neo4j
    - WORKER_DESCRIPTION=neo4j is a graph database
    - WORKER_IMAGE=dariah/neo4j:latest
  volumes:
    - ./neo4j-worker.ini:/app/dockerworker.ini
facade:
  image: httpprincess/http-facade:latest
  links:
    - db
    - messaging
  command: python /app/client.py
```

For our prototype we have implemented a generic Docker worker. The deployment of Docker-based instances is usually conducted in the same fashion, regardless of what software is confined in the image. Hence the Worker is initialized with a name of the Docker image containing the software it will be responsible for (see Listing 5). We create an instance of Worker for each software package offered in the extensible SaaS. To scale out the system it is also possible to have multiple workers offering the same type of the software. The requests will be then divided among the workers in a Round-robin fashion, allowing for fair load distribution.

At this stage it should be stressed, that the proposed architecture is extensible. The Workers abstract the technology used for creating the instances. It would be perfectly possible to write a worker which would use

Puppet-based deployment descriptions or any other technology that might come around in the future.

An important feature for the operation of the service was the integration with the DARIAH-DE Authentication and Authorization Infrastructure (AAI). It is based on Shibboleth with project specific Attribute Providers. Flask does not provide a native means for Shibboleth-based authentication, therefore a new plug-in had to be developed. The effort paid out since it potentially opened the extensible SaaS also beyond the group of DARIAH-DE users. There is a Europe-wide federation of Shibboleth-based AAI Infrastructures called eduGAIN [18]. With the current solution it is possible to allow all members of this federation to access the DARIAH-DE Meta Hosting. This means that potentially all European researchers can use it to share their software. This point was crucial for DARIAH-DE since (as already explained) it is part of bigger project federation of national DARIAH efforts.

The integration with the DARIAH-DE AAI solution was important also for the sake of integrating Meta Hosting with other DARIAH-DE services. By sharing common authentication and authorization mechanism it would be possible to offer a seamless user experience of using multiple services at the same time. An example of such a usage would be an analysis of digital texts collected in the TextGrid repository [26] with the Voyant [24] tools running in the Meta Hosting as implemented in DARIAH-DE DigiVoy tool [8]. It should be stressed that this paper is written from the perspective of the resource provider, the actual use of the resources (i. e. the research tasks tackled on the provided resources) are part of a different domain and could be subject of a future work.

Another extension that was implemented for operational reasons was the status monitoring part. The astute reader may notice some elements on Fig. 4.1 which were not yet explained. In particular elements called: Admin, and Status Worker. The later entity is responsible for monitoring local resource usage and status of Workers. The collected information is passed through a dedicated messaging queue to the Status Store, from which it is presented to the infrastructure operators. All in all, it is very similar to the information flow in case of instance creation and management. There is, however, an important difference. The information in the Status Store are persistent and should not be lost. The reason is simple, the information might be used for accounting and infrastructure debugging purposes. The administrator is able to track down the resource usage and usage induced by single users. Some pieces of this information are also send to the central monitoring system of DARIAH-DE to inform users about the health status of the Meta Hosting service.

5. Extending SaaS. A test of the extensibility of our solution was to provide more complex software through the extensible SaaS solution. We have tested this scenario with a Grid middleware. Uniform Interface to Computing Resources (UNICORE) [25] is a ready-to-run Grid system to make distributed compute and data resources accessible in a seamless way. It has a broad spectrum of applications, for instance, the US-based infrastructure project XSEDE [30], or Human Brain Project [15]. The far-reaching flexibility of the middleware comes at the cost of not-straight-forward deployments, resulting also from the number of components they contain. The installations in the aforementioned big infrastructures are long-running and have to be tailored to specific requirements and resources. Although it would be possible to at least support such installations with Docker images, our use case was a little bit different. There are situations where users want to "train" with the middleware, without the risk of depleting resources provided by the infrastructures, also the infrastructure operators prefer to get the users familiar with the system before allowing the actual access to the scarce resources. The extensible SaaS can be applied exactly for such training purposes. It allows a quick provision of a preconfigured UNICORE middleware, giving the users a possibility to learn how to use it and discard the instances afterwards.

The prerequisite for the inclusion of the UNICORE into extensible SaaS was the availability of the Docker images. As already explained, Docker is a lightweight virtualization solution to provide images with executable services inside. The typical philosophy of the tool is to provide one image for one service. UNICORE is composed of number of service components. Also the way UNICORE is configured (lots of XML files) and how the services mutually authenticate to each other (with x.509 certificates) makes it hard to use with Docker. Luckily, it allows an all-in-one installation which should not be used for production deployments but is sufficient for testing and training purposes. It was used to create the Docker images [27]. Solely, the availability of the proper image was sufficient to include UNICORE in the portfolio of the SaaS. We have created new deployment of the extensible SaaS which can be used for training purposes. We also reuse the Shibboleth-based authentication plug-in, this

time the users were not limited to the DARIAH-DE user base.

6. Conclusion. The two main problems addressed in this paper were scientific software sharing and efficient, seamless, user-friendly access to resources in distributed infrastructures. Sharing of research software is the central component of reproducible data-driven science. We have argued that the problem is manyfold. It involves both technical and social challenges. We focused on the former ones. To this end, we firstly reviewed the technologies which could be used for software sharing. We discussed the software sharing with the selected technology in the context of e-Infrastructure: DARIAH-DE. The presented software sharing workflow is based on established and well-known technologies. It achieves a high level of trust in a decentralized environment.

In the second part of this paper we have shown how the presented software sharing solution can be used to leverage an extensible Software-as-a-Service system. It enables a quick and easy deployment of the software on IaaS resources available in the e-Infrastructure. To underpin the extensibility of the proposed architecture we show that it can be applied beyond its original purpose of sharing software in digital humanities. Based on DARIAH-DE Meta Hosting we have created a test bed of the UNICORE Grid middleware for training purposes. We shared the experiences gained during the implementation and operation of the working prototypes.

Our observation is that both addressed problems: efficient access to resources in distributed infrastructures and user-friendly software sharing intervene to high extent in case of DARIAH-DE. The workflow for software sharing produces artifacts which can be instantly deployed on the available resources with help of the extensible SaaS. For the second discussed use case of dynamic provisioning of UNICORE instances, this is not the case. There is no sharing workflow. It shows that the extensible SaaS is compatible with different sharing approaches. By contrast the sharing workflow can be used also beyond the extensible SaaS, for instance, to create service instances on local resources.

Acknowledgment. The work on Meta Hosting was partially funded by the German Federal Ministry of Education and Research (BMBF) under the project DARIAH-DE (fund number 01UG1110A-M).

REFERENCES

[1] E. Afgan, D. Baker, N. Coraor, H. Goto, I. M. Paul, K. D. Makova, and J. Taylor, *Harnessing cloud-computing for biomedical research with Galaxy Cloud*, Nature Biotechnology, 29 (2011), pp. 972–974.
[2] E. Afgan, K. Krampis, N. Goonasekera, K. Skala, and J. Taylor, *Building and provisioning bioinformatics environments on public and private clouds*, in MIPRO 15: 38th International Convention on Information and Communication Technology, Electronics and Microelectronics, May 2015, pp. 223–228.
[3] *Apache HTTP server project*. `http://httpd.apache.org/`. [Online; accessed: 2015-12-20].
[4] *Amazon Web Services*. `http://aws.amazon.com/`. [Online; accessed: 2015-12-20].
[5] T. Blanke, M. Bryant, M. Hedges, A. Aschenbrenner, and M. Priddy, *Preparing DARIAH*, in IEEE 7th International Conference on E-Science, Dec 2011, pp. 158–165.
[6] *Chef*. `https://www.chef.io/`. [Online; accessed: 2015-12-20].
[7] *Docker Compose*. `https://www.docker.com/docker-compose`. [Online; accessed: 2015-12-20].
[8] *DIGIVOY*. `https://de.dariah.eu/digivoy`. [Online; accessed: 2016-01-29 (in German)].
[9] *Docker*. `https://www.docker.com/`. [Online; accessed: 2015-12-20].
[10] *DARIAH-DE Docker Hub Account*. `https://hub.docker.com/r/dariah/`. [Online; accessed: 2016-01-27].
[11] J. T. Dudley and A. J. Butte, *In silico research in the era of cloud computing*, Nature biotechnology, 28 (2010), pp. 1181–1185.
[12] *GitHub*. `https://github.com/`. [Online; accessed: 2015-12-20].
[13] *Official DARIAH-DE GitHub Account*. `https://github.com/DARIAH-DE`. [Online; accessed: 2016-01-27].
[14] C. Goble, *Better software, better research*, IEEE Internet Computing, 18 (2014), pp. 4–8.
[15] *Human Brain Project*. `http://www.humanbrainproject.eu/`. [Online; accessed: 2015-12-20].
[16] M. Hedges, H. Neuroth, K. M. Smith, T. Blanke, L. Romary, M. Kster, and M. Illingworth, *TextGrid, TEXTvre, and DARIAH: Sustainability of Infrastructures for Textual Scholarship*, Journal of the Text Encoding Initiative, (2013), pp. 3–13.
[17] D. Lecarpentier, P. Wittenburg, W. Elbers, A. Michelini, R. Kanso, P. Coveney, and R. Baxter, *EUDAT: A new cross-disciplinary data infrastructure for science*, International Journal of Digital Curation, 8 (2013), pp. 279–287.
[18] D. López, *eduGAIN: Federation interoperation by design*, in TERENA Networking Conference, May 2006.
[19] P. Mell and T. Grance, *The NIST definition of cloud computing*, Tech. Report 800-145, National Institute of Standards and Technology, 2011.
[20] *OpenStack*. `http://openstack.org/`. [Online; accessed: 2015-12-20].
[21] *Puppet*. `http://puppetlabs.com/`. [Online; accessed: 2015-12-20].
[22] *RabbitMQ*. `http://www.rabbitmq.com/`. [Online; accessed: 2015-12-20].

[23] J. Rybicki and B. von St Vieth, *DARIAH Meta Hosting: Sharing software in a distributed infrastructure*, in MIPRO 15: 38th International Convention on Information and Communication Technology, Electronics and Microelectronics, May 2015, pp. 217–222.
[24] S. Sinclair and G. Rockwell, *Voyant Tools*. `http://voyant-tools.org`. [Online; accessed: 2016-01-29].
[25] A. Streit, D. Erwin, T. Lippert, D. Mallmann, R. Menday, M. Rambadt, M. Riedel, M. Romberg, B. Schuller, and P. Wieder, *UNICORE: From project results to production grids*, in Grid Computing The New Frontier of High Performance Computing, vol. 14 of Advances in Parallel Computing, Elsevier, 2005, pp. 357–376.
[26] *TextGrid Repository*. `http://textgridrep.de/`. [Online; accessed: 2016-01-29].
[27] B. von St. Vieth, *UNICORE Docker Image*. `https://hub.docker.com/r/benedicere/unicore/`. [Online; accessed: 2016-01-29].
[28] *Linux-VServer*. `http://linux-vserver.org/`. [Online; accessed: 2015-12-20].
[29] J. Wang, D. Crawl, and I. Altintas, *A framework for distributed data-parallel execution in the Kepler scientific workflow system*, in ICCS 12: 1st International Workshop on Advances in the Kepler Scientific Workflow System and Its Applications, 2012, pp. 41–42.
[30] *Extreme Science and Engineering Discovery Environment*. `http://www.xsede.org`. [Online; accessed 2015-12-20].

Edited by: Enis Afgan
Received: December 21, 2015
Accepted: March 31, 2016