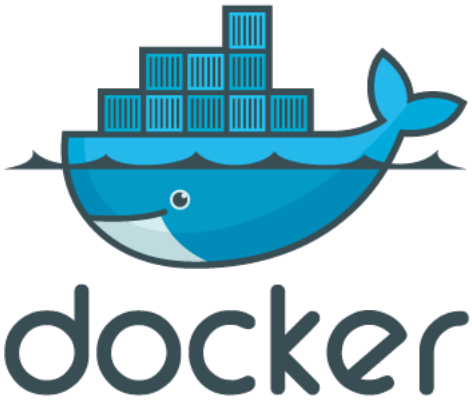
University of Groningen

Software Patterns
Team 2

Pattern-based Recovery & Evaluation: Docker



Authors:
Putra, Guntur
Fakambi, Aurélie
Schaefers, Joris
Menninga, Wouter

Sunday 31st January, 2016

Version 0.4

Authors

Name	E-Mail
Putra, Guntur	G.D.Putra@student.rug.nl
Fakambi, Aurélie	A.Fakambi@student.rug.nl
Schaefers, Joris	J.Schaefers@student.rug.nl
Menninga, Wouter	W.G.Menninga@student.rug.nl

Revision History

Version	Author	Date	Description
0.1	Menninga	12-12-15	Added stakeholders and concerns.
	Schaefers	11-12-15	Added stakeholder concern table
	Schaefers	11-12-15	Added key drivers
	Putra	13-12-15	First draft of introduction chapter.
	Fakambi	11-12-15	System context
0.2	Menninga	29-12-15	Improved stakeholders section
	Menninga	02-01-16	Added Client-Server pattern
	Putra	03-01-16	Revised the introduction.
	Menninga	03-01-16	More additions client-server pattern + added security to stakeholders
	Putra	03-01-16	Added layers pattern documentation.
	Menninga	04-01-16	Added Plugin pattern.
	Menninga	04-01-16	Some improvements to System Context.
	Schaefers	04-01-16	Changed the names of the stakeholders concerns to include both the main- and sub quality
	Schaefers	04-01-16	Added image showing the relation between the stakeholders and the concerns
	Schaefers	04-01-16	Changed the key drivers, added references, changed the explanations
0.3	Putra	08-01-16	Revised introduction
	Menninga	09-01-16	Finished/improved Plugin and Client-Server
	Menninga	09-01-16	Improvement stakeholder concerns
	Putra	10-01-16	Revised layers pattern
	Menninga	10-01-16	Client server in logical view
	Menninga	10-01-16	Added proxy pattern
	Putra	10-01-16	Added few text of logical view and raw intro of evaluation
	Menninga	10-01-16	Security as key driver
	Fakambi	10-01-16	Shared Repository,Event-Driven and Direct Authentication Patterns
	Menninga	11-01-16	Intro for pattern documentation
	Schaefers	11-01-16	Fixed the .lib to make the references resolve again
	Schaefers	11-01-16	Added a start to the logical view in the architecture
	Schaefers	11-01-16	Figure updated about docker registry in chapter 5
0.4	Menninga	15-01-16	Some improvements to stakeholders
	Putra	16-01-16	Small revision introduction. Rewriting layers pattern.
	Putra	16-01-16	Added layers pattern illustration.
	Putra	17-01-16	Added layers pattern evaluation.
	Putra	17-01-16	First draft of conclusion.
	Menninga	17-01-16	Improvements to plugin/client-server/proxy pattern (also added some figures)
	Menninga	17-01-16	Added broker pattern
	Menninga	17-01-16	Evaluation for client-server, proxy, broker and plugin.
	Schaefers	17-01-16	Added general dependency information at the beginning of the software architecture description.

Version	Author	Date	Description
	Schaefers	17-01-16	Added traceability information to the client-server pattern by looking at the dependencies of the packages.
	Schaefers	17-01-16	Added glossaries page and made overall minor improvements.
0.5	Schaefers	11-01-16	
	Putra	31-01-16	Added several FRMs and plugin illustration.
	Putra	31-01-16	Minor modifications.
	Fakambi	11-01-16	
	Menninga	11-01-16	
	Menninga	11-01-16	

Contents

List of Figures

List of Tables

1 Introduction

This document presents the architecture recovery of Docker[1] by identifying software patterns and performing an evaluation of the architecture based on the identified patterns. This document is part of the Software Pattern course assignment at the University of Groningen.

Docker is an open-source project that automates the deployment of applications inside software containers, by providing an additional layer of abstraction and automation of operating-system-level virtualization on Linux [27]. Docker can also run on Windows and Mac OSX, although it is actually run through a VM using Docker Machine because Docker Engine itself can only run on Linux.

The rest of the document is organized as follows. Chapter 2 gives brief explanation with regard to Docker. Chapter 3 elaborates on the stakeholders involved in the Docker project and its corresponding key-drivers. Chapter 4 depicts the architecture of Docker using logic and process views to make it more clear. The patterns that were identified in the Docker project, are documented in chapter 5. Evaluation is presented in chapter 6. Chapter 7 gives several recommendation for the Docker project. Lastly, a conclusion is drawn in chapter 8.

Obtaining and gathering information about Docker is done using the IDAPO [30] approach[2]. This approach defines the process of gathering the information in a clear and structured way. This approach is designed to gather the relevant information that will eventually lead to exposing the architecture of docker and with it, the architectural patterns.
The recovered patterns will be evaluated using the [3] method [19]. This method provides a time-efficient yet reliable way of reviewing the the software architecture and the software patterns.
The IDAPO process consists of 12 distinct steps. Each step will be handled and discussed in this document. The following table, Table 1.1 relates the IDAPO steps to its discussion in this document.

Table 1.1: IDAPO steps

Step	**Description**	**Chapter**
1	identifying the type and domain	2&3
2	identifying used technologies	4
3	studying used technologies	4
4	identifying candidate patterns	4
5	reading patterns literature	4
6	studying the documentation	5
7	studying the source code	5
8	studying components & connectors	5
9	identifying patterns and variants	5
10	validating identified patterns	5
11	getting the feedback from community	5
12	register pattern usage	5

Chapter 2 and Chapter 3 correspond to step (1) of IDAPO. Step (2) to (5) are related to Chapter 4, where used technologies are presented in the architecture. Chapter 5 corresponds to step (6) to (12), where the result of those steps is the pattern documentation.

While IDAPO is used to recover the patterns from the Docker project, PBAR is used to review and evaluate this recovered architecture. This review is documented in Chapter 6.

[1] `https://www.docker.com/`
[2] Identifying Architectural Patterns in Open Source Software
[3] Pattern-Based Architecture Reviews

2 System Context

In this chapter an global information about the Docker project is presented. This includes a general overview of the working of docker and the history of the docker project.

2.1 Overview

Docker is an open-source project, created by Solomon Hykes, written in the "Go" programming language [17]. The goal of docker is making the deployment and distribution of applications easier. Docker started as an internal project within dotCloud, a platform-as-a-service company with four main developers. The first open-source version of Docker was released in March 2013.

Docker automates the deployment of applications inside so-called "containers" by making use of operating-system-level virtualization. They state that "Docker provides an integrated technology suite that enables development and IT operations teams to build, ship, and run distributed applications anywhere"[13].

Operating-system-level virtualization is a virtualization method where the kernel of an operating system allows the isolation of multiple instances of the user space of the operating system. Leveraging this type of virtualization over full virtualization (i.e. hardware emulation) offers a significant performance increase, because there is no need for emulation of the hardware.[31]

Figure 2.1 illustrates the difference between virtualization using hardware emulation and operating-system-level virtualization. As opposed to a virtual machine, a container does not have a guest operating system. Additionally, it does not need a hypervisor for emulating virtual hardware and trapping privileged instructions. This saves a lot of performance overhead.

One tool that makes use of operating-system-level virtualization is Docker. Docker can be used to package applications and dependencies in a 'container'. Running this container using Docker will run it within a separate user space using operating-system-level virtualization.
Docker containers are instances of running Docker images. The process of creating a Docker image is relatively uncomplicated. It involves the creation of a so-called "Dockerfile", which is a type of Makefile, containing instructions that are to be executed to install the application and dependencies (using the `RUN` instruction), and copy relevant data into the container (using `ADD`). Every Dockerfile extends another image (using the `FROM`

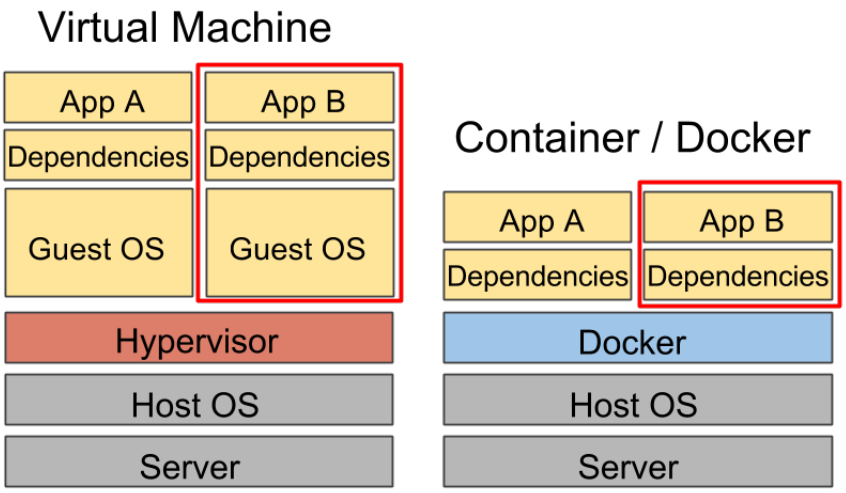


Figure 2.1: Virtual machines compared to containers
Source: [26]

instruction), which is either a previously created image or a base image (often containing the files of a specific operating system).

Docker images use a layered file system. Every command executed during the build-proces, creates a new layer where all modifications to the file system are stored. Docker supports exporting images to tar-archive that are usable by similar versions of Docker.

2.2 Community

Docker has an active community and the managers encourage developers around the world to contribute. [8]

There are several ways to get in contact with the community of Docker:

IRC	Here, the most knowledgeable Docker users reside. At the `#docker` and `#docker-dev` groups on `irc.freenode.net`.
Google Groups	The developers are active in the `docker-dev` Google group.
Stack Overflow	Questions about Docker itself can be asked on the Stack Overflow.

The main organizations that contribute to Docker are:

- Docker team
- Red Hat
- IBM
- Google
- Cisco Systems
- Amadeus IT Group

3 Stakeholders and Concerns

This section defines all stakeholders of the Docker system and describes the concerns of these stakeholders. A stakeholder is a person, group of persons, or organization that is involved in Docker. This section uses the quality attributes as described in ISO-25010[23]. This ISO defines categories of quality attributes (main concerns). These categories contain individual specific quality attributes (specific concerns).

3.1 Stakeholders

This section presents the stakeholders and their concerns. The main concerns are listed in bold and, where applicable, the specific concern in parentheses.

Docker has the following stakeholders:

- Software developers
- Software Maintainers
- Cloud providers
- The open container initiative
- Docker developers
- Plugin developers

Software Developers

The developers are those that use Docker to create their software systems. Docker is used by developers to deploy software and create architectures consisting of multiple containers. Developers using Docker for their system, expect it to work according to the documentation and have a good usability.

Concerns

Usability	Since the software developers are using Docker, they want it to have good usability. This means it is easy to learn how to use Docker and it is not hard to work with. In fact, this is one of the main benefits of Docker, since it makes it easier to containerize applications, which could be done before, but was very difficult to do in practice.
Functional Suitability *(Functional correctness)*	Software developers care about the functional correctness, since bugs in the Docker software make the development of their software more difficult.

Software Maintainers

Software maintainers are responsible for deploying the software and keeping the software product running. Docker is often used for software deployment (especially to the cloud). The software maintainers expect Docker to be reliable, portable, efficient and secure.

Concerns

Reliability	Software maintainers are responsible for keeping the software working while it is deployed. They will only use Docker if it is reliable. It has to be tolerant against failing containers.
Portability	The software maintainers want to be able to run Docker and its containers on a variety of different environments.
Performance efficiency	The software maintainers want Docker to have a good performance. They want the resources of their servers to be used as efficiently as possible and

	therefore want Docker to have almost no overhead, like Virtual Machines often do have.
Security	Software maintainers want Docker to be secure, so applications can be run inside the containers, without Docker negatively affecting the security of this application.

Cloud providers

There are numerous cloud providers offering services which are based on Docker[1]. These cloud providers offer Container-based cloud computing, sometimes referred to as CaaS (Containers-as-a-Service).

Concerns

Portability *(Installability)*	The cloud providers want to integrate Docker into their cloud computing architecture.
Compatibility *(Co-Existence)*	Docker has to share the environment with the existing architecture of the cloud providers.
Reliability	The customers using the cloud providers' services expect a high reliability. If cloud providers are to implement Docker in their architectures, they need Docker to be as reliable as running the application naively or inside a VM.
Security	Cloud providers want to offer a secure cloud computing environment to their customers.

The Open Container Initiative

The Open Container Initiative[2] was formed with the purpose of creating an open industry standard for container formats and runtime in June 2015.

They are interested in creating a formal, open, industry specification around container formats and runtime. This specification should be independent of particular clients/orchestration stacks, commercial vendors or projects and should be portable across a wide variety of operating systems, hardware etc.

Docker has donated its container format and runtime, known as 'runC' to this initiative and is one of the members of the initiative, together with a lot of other members (including competing technologies, such as 'rkt' from CoreOS)[3].

Concerns

Portability	The main goal of the initiative is to standardize the format for the container and the runtime. These standards are used by Docker to allow using Docker containers with other container runtimes.
Compatibility *(Interoperability)*	Docker has to be able to work with the containers from the initiative.

Docker developers

The Docker developers are the developers contributing to the Docker code base. Some of these developers work on Docker in their free time, others work on it as part of their job at a company with an interest in Docker and some developers work for Docker Inc itself[1].

Concerns

Maintainability *(Modifiability)*	These developers contribute new features and bugfixes to the existing code base. Therefore, they want the project to have good modifiability, such that additions and improvements can be realized without to much effort.
Security	The Docker developers care about the security of the product they are working on.[4]

[1] `https://www.docker.com/partners#/service`
[2] `https://www.opencontainers.org/`
[3] A list is available at `https://www.opencontainers.org/about/members`
[4] `https://github.com/docker/docker#security-disclosure`

Plugin developers

Docker allows extending its capabilities with plugins[5]. These plugins are created by the plugin developers.

Concerns

Portability *(Adaptability)*	The developers of the plugins want to extend the functionality of Docker in an effective and efficient way.

Overview

The stakeholders and their related concerns are visualized in Figure 3.1 below.

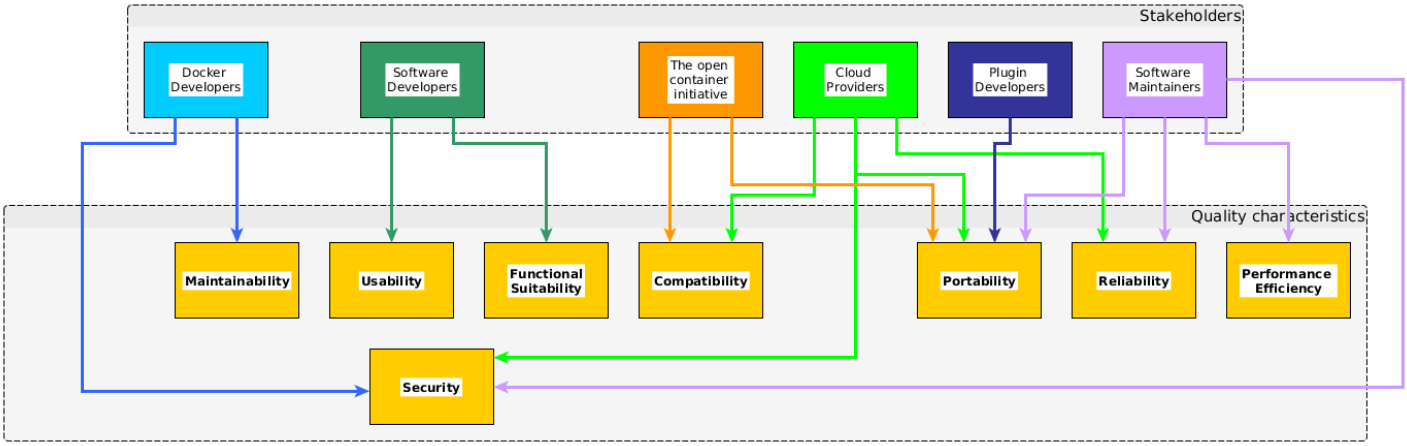


Figure 3.1: Stakeholders and the quality attributes they are concerned about

3.2 Key Drivers

Docker has the following key drivers

- KD- 1 – Portability
- KD- 2 – Security
- KD- 3 – Reliability

3.2.1 Portability

Looking at Figure 3.1, which visualizes the quality concerns of the stakholders, it is apparent that portability is the quality that is most desired among the stakeholders. One of the main features, if not the main feature of Docker, is to provide "agility and control for Development and IT Operations teams to build, ship, and run any app, anywhere"[2].
Docker wants the containers running the applications to be "hardware-agnostic" and "platform-agnostic" [16], meaning that Docker containers do not know anything about the hardware, nor the platform it is run on, thus making it very portable. This allows developers to focus on the actual application, without having to worry about the underlying hardware or platform. The applications the developers create could then be run everywhere where docker is supported.
To provide this quality attribute, Docker is also focused on increasing the performance. To truly be able to replace applications, the docker containers must perform comparable to running the application natively, using a comparible amount of system resources. Running an application inside a VM makes it somewhat portable. However, if the VM can only be run on systems that can cope with the increased use of system resources, the portability is reduces significantly. By making the Docker containers small, yet having a high performance, the containers become usable by a wide variety of systems and are able to replace the need to use any virtual

[5] https://blog.docker.com/2015/06/extending-docker-with-plugins/

machines that would otherwise be used to achieve this level of 0portability[16]. IBM has researched and compared the performance of VM's and containers [32]. This research shows that the performance overhead of docker is indeed minimal, as is shown in table 3.2.

Workload		Native	Docker	KVM-untuned	KVM-tuned
PXZ (MB/s)		76.2 [±0.93]	73.5 (-4%) [±0.64]	59.2 (-22%) [±1.88]	62.2 (-18%) [±1.33]
Linpack (GFLOPS)		290.8 [±1.13]	290.9 (-0%) [±0.98]	241.3 (-17%) [±1.18]	284.2 (-2%) [±1.45]
RandomAccess (GUPS)		0.0126 [±0.00029]	0.0124 (-2%) [±0.00044]	0.0125 (-1%) [±0.00032]	Tuned run not warranted
Stream (GB/s)	Add	45.8 [±0.21]	45.6 (-0%) [±0.55]	45.0 (-2%) [±0.19]	
	Copy	41.3 [±0.06]	41.2 (-0%) [±0.08]	40.1 (-3%) [±0.21]	
	Scale	41.2 [±0.08]	41.2 (-0%) [±0.06]	40.0 (-3%) [±0.15]	
	Triad	45.6 [±0.12]	45.6 (-0%) [±0.49]	45.0 (-1%) [±0.20]	

Figure 3.2: Performance comparison results for PXZ, Linpack , Stream, and Random access. Each data point is the arithmetic mean of ten runs. Departure from native execution is show within parentheses "()". The standard deviation is shown within square brackets.
Source: [32]

To increase the portability even more, docker provides a service to upload and share docker containers, making docker even more portable. This service is called the "Docker hub", which is a "docker registry" and will be discussed in 4.1.5.

3.2.2 Security

The Docker daemon executes software inside containers and exposes an API. If somebody would gain unauthorized access to this API, this attacker would be capable of executing any program on the vulnerable host machine. Therefore, security is an important key driver for Docker.

Additionally, since Docker does not use separated operating systems, but instead operating-system-level virtualization, the security and isolation of the containers is a serious concern if software inside the containers is not trusted. For example, for a cloud provider running containers for clients, isolation of these containers is an important aspect of the security, since clients should not be able to access data of other clients.

3.2.3 Reliability

The main function of a Docker container is to run a certain application. However, a system usually needs to run a set of different applications and it should not be the case that running these applications in Docker compromises the systems reliability.
Dependencies often lead to problems[16], decreasing the reliability of software. The Docker files are text files containing a small set of instructions that will run an application. This means that the instructions needed to run an application are conveniently located in a single place, the Dockerfiles. This allows viewing and the dependencies of all the different applications and making them easily maintainable.
Docker increases the reliability of the applications it runs, by increasing the testability of the containers and applications.
To make sure Docker itself stays reliable, it provides a test framework and test functions that thoroughly test the Docker software. [12]

4 Software Architecture

This section discusses the architecture of Docker. Two views from the 4+1 Architectural View Model[25] will be presented : the logical- and the process view.

4.1 Logical View

Docker is made using the the go language created by Google. Docker is an application categorized as "Operating-system-level virtualization". It consists of:

- 2979205 lines of go code (almost 3 million).
- 1782 '.go' files
- 267 different packages

Several tools have been used to get information about the dependencies of the different packages [21] [24]. Using these tools, statistics can be generated about the usage of the packages. This can give an insight about the most important packages of the system.
The packages that are most used by other packages is listed below in table 4.1. For each package the table lists the amount of references from the package in the "Out" column and the amount incoming references in the "In" column. The "In" column will be increased by one for each reference from any other package to this specific package.

Table 4.1: List of packages sorted by the amount of incoming dependencies.

In	Out	Package
33	11	github.com/docker/docker/image
33	23	github.com/docker/docker/pkg/archive
36	0	github.com/docker/distribution/digest
37	5	github.com/docker/docker/reference
37	12	github.com/docker/docker/daemon/execdriver
40	47	github.com/docker/docker/container
42	0	github.com/opencontainers/runc/libcontainer/configs
42	4	github.com/docker/docker/errors
43	0	github.com/docker/libnetwork/types
47	20	github.com/docker/docker/runconfig
48	4	github.com/docker/docker/cli
55	1	github.com/docker/docker/pkg/mflag
104	9	github.com/docker/docker/api/types
187	0	github.com/Sirupsen/logrus

The most referenced package is the logging package called "logrus". After that it is the api package that is most referenced. This is to be expected, since it is the only way to communicate with the docker daemon [9].
The "pkg" package, the third most referenced package, is, "a collection of utility packages used by the Docker project without being specific to its internals.". This means that it can contain any kind of logic for which the developers decided that it should not go in to the core codebase in order to keep the core small [10]. The specific mkflag package that is referenced here, is responsible for parsing the multiple flags (hence mflag) that are passed to the command-line interface [11].
The command line interface (cli) package is most used after this. The docker container package seems to be used by a lot of packages (In) and reference a lot of packages itself as well (Out). Same goes for the docker daemon and the docker image with increasingly less references.
Next, the the packages that have the most references to other packages are listed in table 4.2.

Table 4.2: List of packages sorted by the amount of outgoing dependencies.

In	Out	Package
1	36	github.com/docker/docker/builder/dockerfile
19	38	github.com/docker/docker/registry
10	43	github.com/opencontainers/runc/libcontainer
0	43	github.com/docker/docker/docker
9	44	github.com/docker/docker/api/client/lib
40	47	github.com/docker/docker/container
10	68	github.com/docker/libnetwork
1	101	github.com/docker/docker/distribution
1	235	github.com/docker/docker/api/client
11	349	github.com/docker/docker/daemon

Not surprisingly, the docker daemon has the most references. This is because it is responsible for making the containers run in a secure and reliable way. While at the same time the daemon handles requests coming from the API, which is the package with the second most amount of outgoing references.
The container package has a lot of references to the runc package, which is a "CLI tool for spawning and running containers according to the Open Container Format (OCF) specification." [22].

The following image 4.1 shows an overview of the Docker architecture.

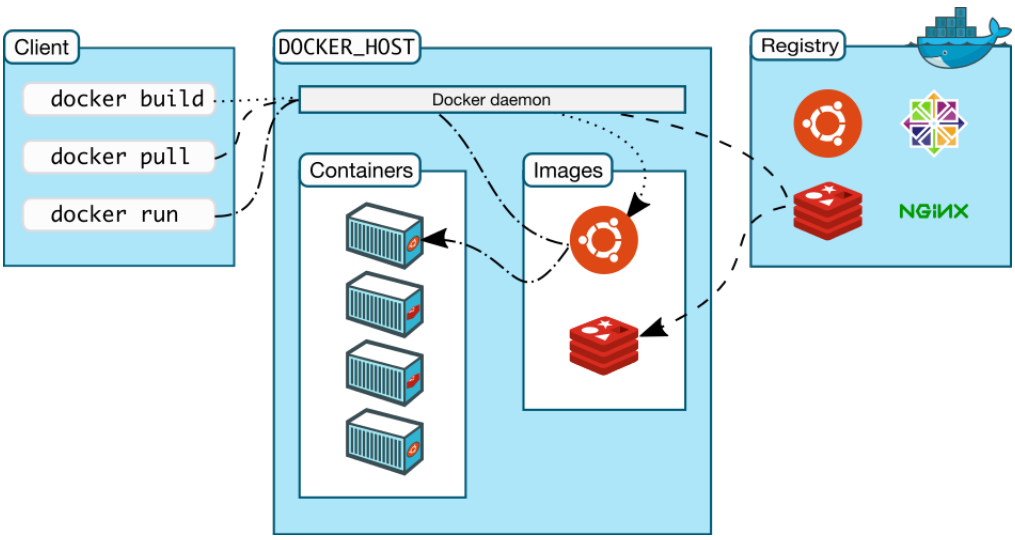


Figure 4.1: A high-level overview of the Docker architecture
Source: [4]

Docker uses a client-server architecture. The client (a command-line tool) acts as the primary user interface and talks to the Docker daemon. The daemon is a background process which does all the heavy lifting, e.g. the building and running of the containers.

The following sections will discuss the most important parts of docker in more detail.

4.1.1 Client

The client is a small binary and acts as the primary user interface to Docker. The user inputs commands into the client and the client forwards these commands to the Docker daemon, which executes commands. The Docker client is capable of connecting to daemons running on the local machine, as well as connecting to daemons running on remote machines over the internet.

4.1.2 Docker host: Daemon (server)

The Docker daemon is a process running on the host machine (as can be seen in Figure 4.1). This process is usually started when the host machine starts and runs in the background. It exposes a REST interface and listens for requests coming from clients on the same or remote hosts.

4.1.3 Docker host: Images

Docker images can be interpreted as a read-only template from which Docker containers are started. A Docker image consists of a stack of layers which are bonded by a union file system. These layers support reusability and sharing.

Docker images can be built from text files ('Dockerfiles'), which contain instructions like installing software and copying files.

4.1.4 Docker host: Containers

Docker containers are based on Docker images. These images consist of read-only layers. Docker containers contain an additional thin writable layer on top of the images to perform operations on them. Docker containers basically consist of the files of the operating system, user-added files, application files, and data files. Multiple containers may run based on the same image. In this case, Docker will not create separate copies for each containers. Instead, they will share the same image with their own writable layer.

To run Docker containers, Docker uses the `libcontainer` library, which is part of the Open Container Initiative.

4.1.5 Registry

A Docker registry is a place where Docker images can be stored and retrieved. The Docker daemon fetches desired Docker images from this repository. This registry can be either public or private and may thus reside behind a firewall. Docker Hub[1] is one example of a docker registry hosted by Docker. A registry provides an easy way to distribute images among different hosts.

4.1.6 Client-Server rationale

The daemon is a background process, which supplies the requested services to the client. The daemon exposes a REST interface. For Docker, the client can be configured to connect to other daemon processes than the one running on the local machine. It can be configured to connect to remote Docker daemons as well, allowing the user to issue commands to daemons running remotely.

4.1.7 Client-server solution

Docker uses a Client-Server architecture. The client, a binary supplying a command-line interface, acts as the primary interface for the user. The user enters commands into this client, which are then sent to a server: the Docker daemon.

4.1.8 Broker authentication rationale

Docker discovers plugins by looking for .sock, .spec or .json files in the plugin directories on the host system. These files describe how Docker can communicate with the plugins using the REST API (usually via a Unix socket). The plugins themselves run as separate processes on the same host as the Docker daemon and implement an HTTP server listening for requests from the Docker daemon. After a user requires a plugin (this is indicated e.g. as a command-line parameter when starting a container using the Docker client) Docker uses the discovery algorithm (see also Section 4.3.1). After that, Docker sends a handshake to the plugin and the plugin returns a list of which subsystems this plugin implements. For these subsytems Docker will replace the default implementation by a Proxy, that forwards all calls over the REST interface to the plugin process.

[1] `https://hub.docker.com/`

4.2 Proxy rationale

Using the proxy pattern allows the daemon to communicate with the plugins, without requiring direct access to these plugins. Also, because the subsystem component has the same interface as its proxy, the implementation of software using this component does not depend on whether the proxy is used or the original subsystem.

4.3 Process View

This section discusses the system processes, how they communicate and the runtime behaviour of the system.

4.3.1 Plugins

Docker supports extending its capabilities using third-party plugins. Docker allows extending the functionality of the volumes, which are responsible for storing files outside containers. And Docker allows extending the functionality of the network subsystems that provides inter-container communication over the network. This functionality will be extended in the future[3].

A plugin runs in its own separate process and communicates with the Docker daemon using the REST API. In order for the Docker daemon to know about the plugins existence, a file has to be placed in a predefined directory.

In Figure 4.2 the process of using a plugin can be seen.

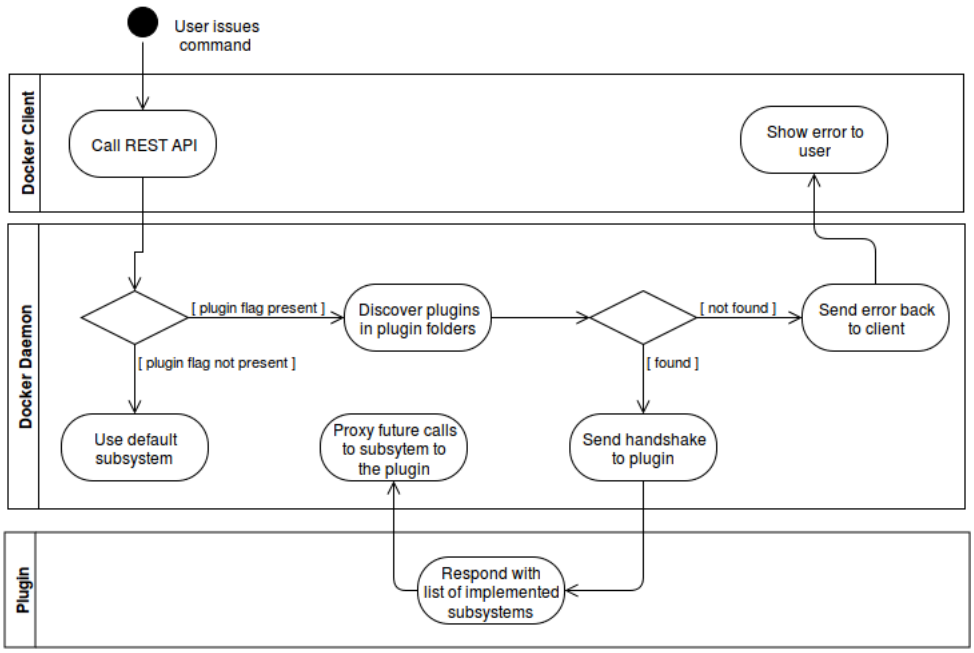


Figure 4.2: An activity diagram showing the process of selecting a plugin

A user indicates that he/she wants to use a plugin by passing a command-line parameter to the Docker client when starting a container. If this parameter is present, then the daemon will start looking for the plugin in the plugin directory. If the plugin is not found, it returns an error to the user. When the plugin is found, the daemon will send a handshake to the plugin using a UNIX socket and the plugin will reply with a list of subsystems it implements. The Docker daemon will then use a Proxy to forward calls to this subsystem to the plugin process.

4.3.2 Brokered Authentication

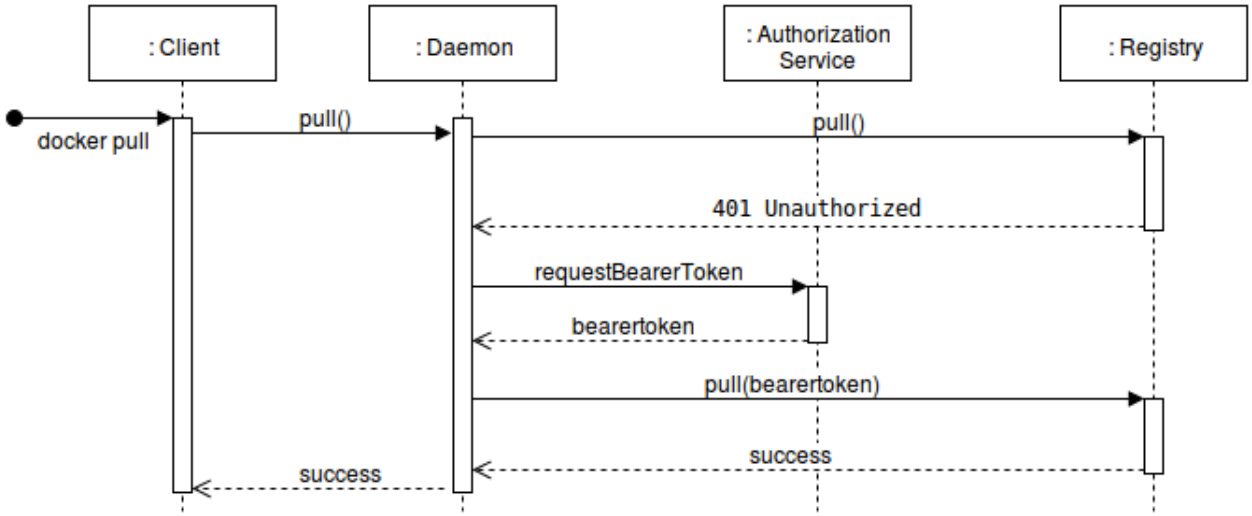


Figure 4.3: A sequence diagram showing the brokered authentication process for the registry

The Docker Registry uses a brokered authentication scheme. An independent authentication broker issues authentication tokens to clients that want to use the registry.

The sequence diagram in Figure 4.3 shows globally the process of authenticating with the registry using an authentication broker (authorization service).

After the client has issued a command (e.g. a pull command) that interacts with the registry (and the client forwarded it to the daemon), the daemon attempts to start the pull-operation with the registry. If the registry requires authentication it returns a `401 Unauthorized` http response. Then, the daemon will connect to the authorization service and request a bearer token and after receiving this bearer token the daemon will retry the pull-operation on the registry with the token. If the token is valid and accepted by the registry it will start the pull-operation.

5 Pattern Documentation

This chapter describes the patterns used in Docker. The patterns are documented as described by Harrison et al.[18], including additional information about concerning where the pattern was found (traceability) in the code or documentation.

5.1 Client-Server

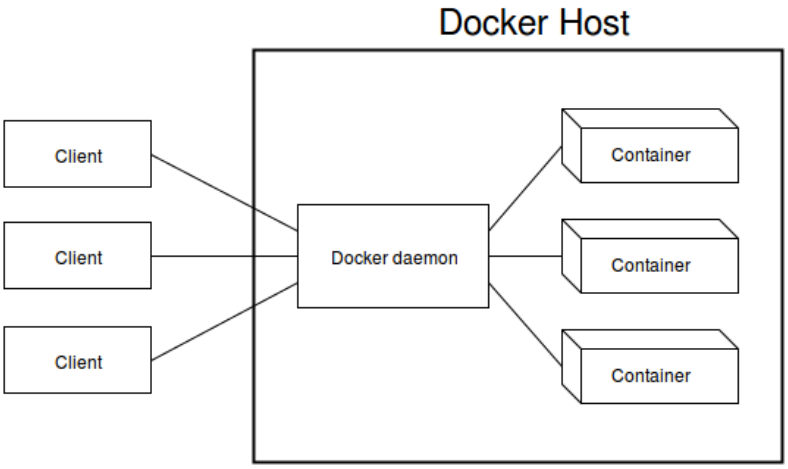


Figure 5.1: Visualization of the client-server pattern used by the Docker clients (client) and the Docker daemon (server).

Traceability

The Client-Server pattern can be deduced from the 'What is Dockers architecture?' section of the online documentation[4].
When listing the dependencies using the dependency tools, very few packages use the docker daemon. Looking at the high-level parts of docker, the main package that references the daemon is the docker container. However, the daemon seems to be referenced quite a lot when looking at table 4.1(section 4.1), that lists the packages with the highest amount of incoming references.
This shows that the container part of docker is heavily using the docker daemon.
It is remarkable that when listing the dependencies of the daemon. It seems to only reference the container. So clearly, the docker daemon is a main access point for modifying the containers.
The documentation says that only the docker api communicates with the daemon. Clearly there is no package using the docker daemon, but also the api does not seem to be referencing the docker daemon. This is because the docker api uses the daemon through means of HTTP requests. As is made clear when looking at the code in

$$api/client/cli.go$$

This contains the line:

$$client, err := client.NewClient(host, verStr, clientTransport, customHeaders)$$

Indicating that the client is referencing the daemon through means of sending HTTP requests to the daemon api.

Source

Architectural patterns revisited – a pattern language, P. 29 [29]

Issue

Docker containers can not be be controlled remotely. It should be possible for Docker containers to be

controlled remotely and a single interface should be able to control containers on multiple hosts (e.g. in the cloud).

Additionally, certain operating systems lack the underlying technologies necessary for running containers. The inability to remotely control containers from these operating systems prevents these systems from using docker in any way.

Solution

Docker uses a Client-Server architecture. The client, a binary supplying a command-line interface, acts as the primary interface for the user. The user enters commands into this client, which are then sent to a server: the Docker daemon.

Assumptions/Constraints

- The docker daemon should support the API calls made by the client. This means that if the client is outdated, the daemon is assumed to have the legacy controllers needed to still provide the wanted functionality.
- The REST interface that the daemon offers is assumed to be available and accessible by the client

Rationale

By separating the client and server it is possible to use the same client to issue commands to different daemons, running on different hosts. It is also possible to use the client on operating systems that do not support running containers.

Implications

The use of the Client-Server pattern results in two different executable binaries: a daemon and a client.

The use of the Client-Server pattern increases the interoperability, since the client can send commands to daemons running on remote machines and the local machine.

Additionally, the portability is increased, since the client can run on Operating Systems that are unable run containers themselves.

There is also an impact on security, because the communication between the client and the server needs to be secured. Because docker is composed of different layers, securing this communication can be done from a single place. This is discussed in section 5.9, which discusses the layers pattern.

Related Patterns

- Broker

5.2 Layers

Layers pattern is a architectural pattern that consists of several layers, which each layer provides a set of services to the layer above and uses the services of the layer below. Docker implements this pattern to separate the concerns of each component with independent functionalities. Layers pattern implementation of Docker, based on our investigation, is shown in Figure 5.9.

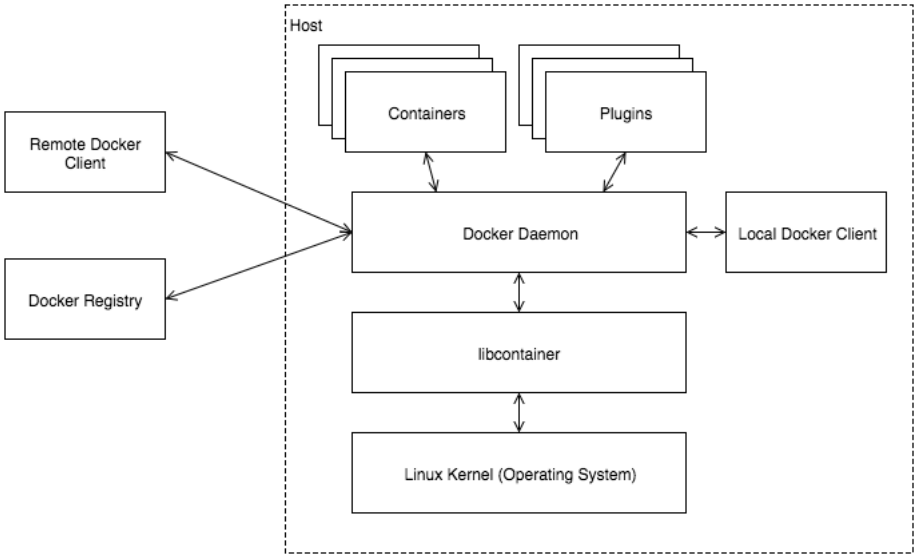


Figure 5.2: A layered overview of the Docker.

Traceability

Layers is implicitly mentioned in the Docker architecture in the online documentation, specifically in the *What is Docker's architecture?* section [4].

Source

Architectural patterns revisited – a pattern language, P. 29 [29]

Issue

Docker provides several features (e.g. hosting the image, creating image, running container, and daemon supervision). Each of the features vary in terms of the functionality they give. Single implementation of features will certainly result in abundant usage of resources, because hosting the image means there must be huge amount of disk space available.

Overlapping may also occur since multiple Docker instances may use the same image or configuration file, which could be stored and organized elsewhere. Docker also has several dependencies, especially the Linux kernel components (`namespaces` and `cgroups`) for OS level virtualization. Furthermore, Docker mostly needs remote supervision or control, since Docker usually runs in the cloud. Therefore, the features must be separated into several independent components and their dependencies and communication have to be clearly separated and maintained.

Assumptions/Constraints

Docker is able to run only in Linux kernel since it uses **`NAMESPACES`** and **`CGROUPS`** technology, which is only available in Linux kernel. However, Docker is also possible to be run on other OS although using virtual machine. The connection between local and remote components are carried out through secured TCP/IP connection.

Solution

Docker is separated into several components that have their own functionality, as can be seen in Figure 5.9. The main components are:

- Containers
- Docker daemon
- Local Docker client
- Libcontainer
- Linux kernel

Everything inside the dotted box is located on the same host. Inter-process communication is utilized to communicate between Docker, containers, libcontainer, and Linux kernel components, but Docker daemon, plugins, and Docker client communicate using Docker API [1]. Remote connections are managed through secured TCP/IP based connection.

[1] `https://github.com/docker/docker/blob/master/api/common.go`

Rationale

Layers are very good in terms of sharing and reusability. In this way, several Docker daemon that uses the same image can just fetch it through the Docker registry. Using Docker registry also reduce disk space for storing images and increase easiness of sharing images. Layers pattern also supports connection to other layer, such as connection to plugins. It is also manageable to supervise remote Docker daemon through remote Docker client.

Implications

This pattern gives positive implication to portability as clear separation of containers, Docker daemon, Docker registry, and Docker client makes Docker portable. Utilizing Docker registry also promotes sharing images, which make it easier to deploy an application without knowing what platform running below it.
A host is open to public in the communication of Docker daemon and Docker registry and remote client. This pattern provides the ability to secure all the communication from single place. This can increase the security, because focus on security is reduces to single place, meaning that strengthening the security there makes everything more secure.
However, a security breach will effect the entire system and not only a single component. This security layer can be easily replicated, having a positive impact on reliability. The security is defined in a single place, and so the level of security that is defined there can be expected from each component.

Related Patterns

- Client-server
- Shared repository

5.3 Shared/Active repository

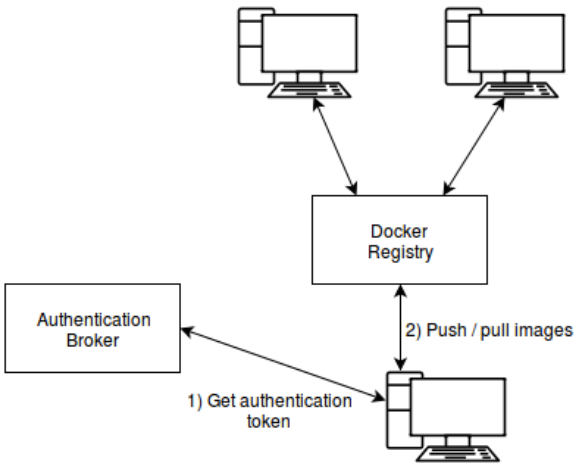


Figure 5.3: The shared repository with brokered authentication

Using the Docker files and the docker commands, personalized images can be created that support a specified project. Docker allows these images to be stored in a docker registry, earlier discussed in subsection 4.1.5. Anyone with access to this registry can pull this image and use it. Users of the application a specific project provides, are able to pull the image. This allows running the application, thereby obtain the desired functionality. The registry also allows the image to be easily shared with other developers working on any related project. This makes the docker registry a shared repository.

The Docker registry also contains functionality for sending notifications when certain events occur [5]. This makes the docker registry an active repository as well. Figure 5.10 shows the illustration of Shared/Active repository pattern in Docker.

Traceability

The Shared Repository pattern can be deducted from the online documentation: [6] “The Registry is a stateless, highly scalable server side application that **stores** and lets you **distribute** Docker images. A registry is a storage and content delivery system.” Everything dealing with the registry is in the “Registry folder” in the Docker repository.

Source
Patterns of Enterprise Application Architecture, P.322 [15]
Architectural Patterns Revisited - A Pattern Language, P.13 [29]
Pattern-oriented Software Architecture - Volume 4, P.202 [14]

Issue
Docker provided a way for the user to control the storage and distribution of images. The user wants to be alerted of new events happening in the registry through notifications.

Assumptions/Constraint
The docker registry is dedicated to handling only docker images. It does not provide shared storage that allows storing any other kind of data. This makes sure that the images stored in the registry are runnable and do not crash in critical ways.

Solution
Docker utilizes a repository to store images. Users who need a particular image may fetch required images through TCP/IP connection. Users interact with a registry by using docker push and pull commands.

Rationale
After the integration of the Shared Repository Pattern each has one central repository containing their images and they can it access using a logging. Users can also access images from others users: The registry is a sharing unit.

Implications
Users can get their own images from the public registry and also the ones created by other users so the registry is a sharing unit. The use of this pattern enhances Re-usability, Changeability, Maintainability, Integrability because the Registry is central.

Related Patterns
Publish Subscribe
- Brokered Authentication

5.4 Publish-Subscribe

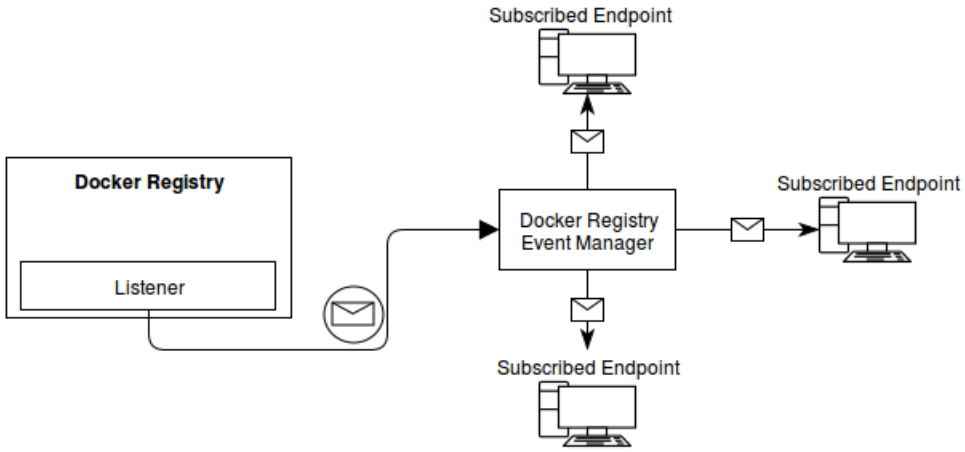


Figure 5.4: Docker Registry's event manager and subscribed endpoints

Traceability
The notification mechanism of the Docker Registry uses the Publish Subscribe Pattern [5].

Source
Pattern Oriented Software Architecture- Volume 4, P.234 [14]
Architectural Patterns Revisited – a pattern language, P.32 [29]

Issue
The user wants to be alerted about certain events/changes that occur in a registry. The notification system needs a mechanism in order to send those events to the user.

Assumptions/Constraints
This pattern is used at a lower level in the Docker Registry/Active Repository Pattern.

Solution
The Publish-Subscribe pattern allows publisher to send messages to endpoints which are subscribed to these messages.

As can be seen in Figure 5.11, a listener listens for events and forwards these to the event manager. The event manager forwards messages of this event to all the subscribed endpoints over HTTP.

Rationale
Using the Publish-Subscribe pattern allows users to configure endpoints, which will receive notifications about the events/changes that occur in the registry.

Implications
The Publish-Subscribe Pattern enhances modifiability because publisher and subscribers are decoupled.

The portability is increased using this pattern, because the decoupling of the publishers and subscribers (combined with the messages send over HTTP) means that the subscribers can be in a completely different environment than the registry itself.

Related Patterns
Shared/Active Repository Pattern

5.5 Brokered Authentication

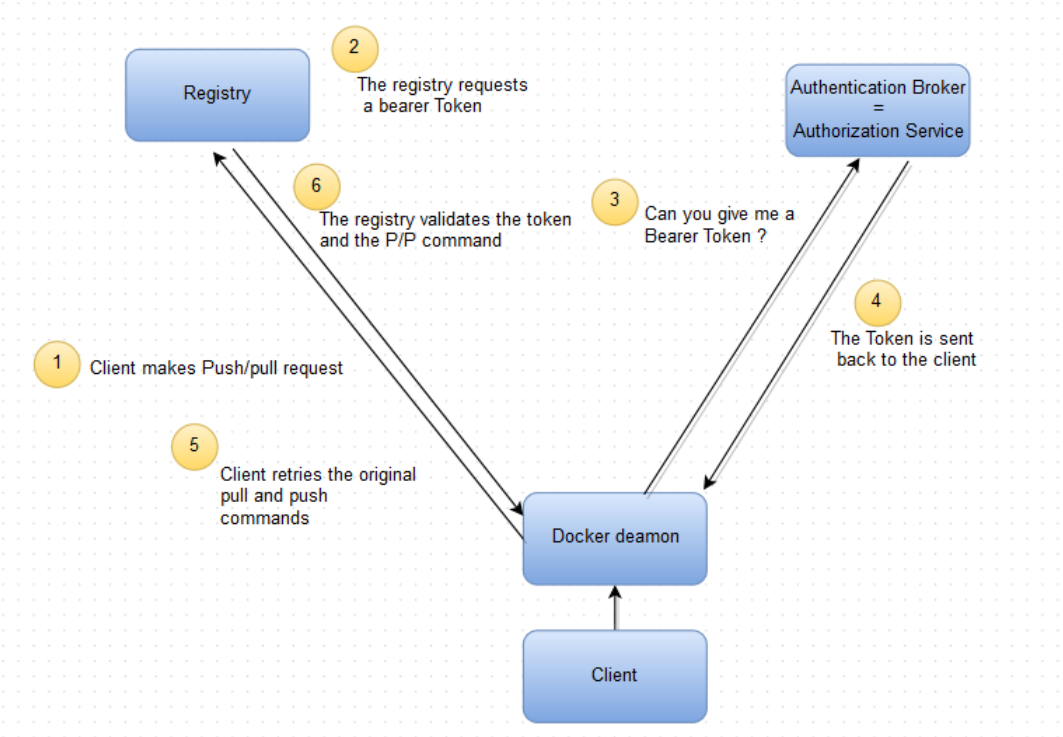


Figure 5.5: Authentication process of the Docker registry

Traceability
From the v2 of Docker the authentication is done through a central service.
The Brokered Authentication pattern can be deducted from the source code through the "auth.go" and the "token.go" files from the Docker Registry Github Repository.
Docker Authentication [7]

The main functions for authentication are :

- func tryV2TokenAuthLogin(authConfig *types.AuthConfig, params map[string]string, registryEndpoint *Endpoint)
- func loginV2(authConfig *types.AuthConfig, registryEndpoint *Endpoint, scope string)

This service is used by the official Docker Registry to authenticate clients and verify their authorization to Docker image repositories.

Source
Brokered Authentication Pattern[28]

Issue
The Docker Registry needs an authentication system in order to ensure security of pulling and pushing images.

Assumptions/Constraints

Solution
An authentication broker that both parties trust independently, issues a security token to the client. The client can use this token to authenticate himself. This means that there is no direct relationship between the user and the registry.

By using the Brokered Authentication the access to the Docker Registry is controlled for each user.

Rationale

Implication
After the integration of the Brokered authentication the textbfSecurity for the registry is ensured. The authentication broker manages trust centrally. This eliminates the need for each client and service to independently manage their own trust relationships.

Related Patterns
Shared/Active Repository Pattern

5.6 Plugin

Plugin pattern basically expands the capability of a particular software. Plugin pattern also provides centralized, runtime configuration[15]. An illustration of plugin pattern is shown in Figure 5.13.

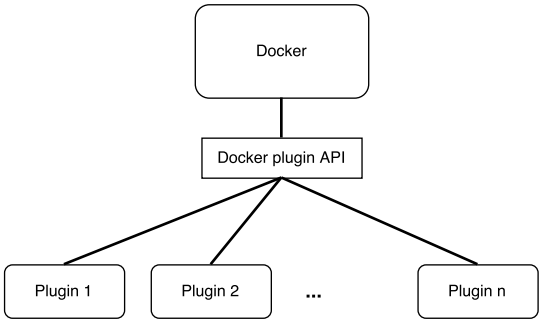


Figure 5.6: Plugin pattern illustration for Docker.

Traceability
The existence of Docker plugin becomes apparent from its documentation, which describes the plugin feature of Docker [3].

Additionally, the directories `docker/pkg/plugin/`[2] and `docker/daemon/graphdriver/plugin.go`[3] (among others) in the project's repository contain the code for discovering plugin and the interfaces the plugin should implement.

[2] `https://github.com/docker/docker/tree/master/pkg/plugin`
[3] `https://github.com/docker/docker/blob/master/daemon/graphdriver/plugin.go`

Source

Patterns of Enterprise Application Architecture, P. 499 [15]

Issue

Several of Docker users desired functionalities are not natively provided by docker. Docker provides the mechanism so that third party custom-built tools that provide this missing functionality to extend Docker. These customizations mean that third party developers are able to write tools, extending Docker's core functionality[20]. The additional functionality that is added, is only usable during run time.

Assumptions/Constraints

- plugin can only extend the functionality of the Docker components if necessary APIs are provided for certain need or use.

Solution

Docker implements the plugin pattern to link several Docker's extendable component's interfaces with third-party code at runtime.

Rationale

By using plugin pattern, any feature that is currently not supported natively by Docker can be added. The users can also develop a custom code that will exploit Docker APIs to fulfill their need.

Implications

The use of the plugin pattern means that the adaptability increases, because plugin allow the application to be adapted with new features. For the security concern, it means that there is extra communication over the API which has to be secured. The security of the plugin themselves cannot be guaranteed by Docker.

Related Patterns

- Broker

5.7 Broker

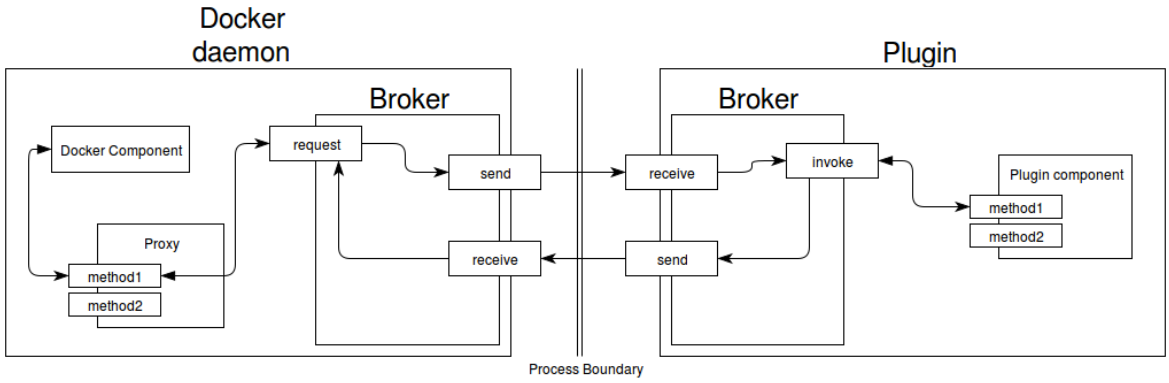


Figure 5.7: Broker for communication between Docker daemon and a plugin.

Traceability

The use of the Broker pattern can be found in the proxy package in the source code: `docker/pkg/proxy/tcp_proxy.go`[4].

Source

Pattern-oriented Software Architecture - Volume 4, P.237 [14]
Architectural Pattern Revisited - A Pattern Language, P.34 [29]

Issue

Docker consists of multiple processes. For example, the client and daemon are separate processes and also the plugins are separate processes.

[4] `https://github.com/docker/docker/blob/master/pkg/proxy/tcp_proxy.go`

These processes have to communicate with each other. However, dealing with the challenges of inter-process communication in the application code would greatly increase the complexity. The application components should be shielded form the details of the inter-process communication.

Assumptions/Constraints

- All the communication between the components in different processes have to go via the Broker.

Solution

Use a Broker to separate the logic for communication between the processes from the application functionality. Use a component-based model (using Proxies) to allow local components to invoke methods on components in remote processes as if they were local.

Rationale

The Broker pattern allows for and handles all the communication between the different processes.

This means that the logic for inter-process communication does not have to be implemented on a per-component basis.

Implications

Using the Broker pattern is good for the portability, because it does not matter if a component is part of a different process (running locally or remotely).

This pattern is also good for the security, since the Broker is a single entry point for all the inter-process communication, which makes it easier to secure. This has been done by using Transport Layer Security for the communication between the processes.

Additionally, the interoperability (compatibility) is increased, because it becomes possible for two components (in two different processes) to exchange information.

The Broker does however introduce a performance overhead, which decreases the performance efficiency.

Related Patterns

- Plugin
- Proxy
- Client-Server

ign

5.8 Client-Server

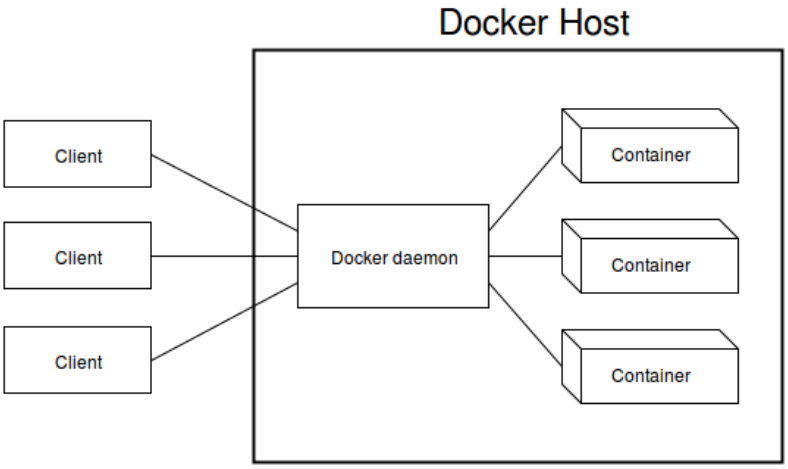


Figure 5.8: Visualization of the client-server pattern used by the Docker clients (client) and the Docker daemon (server).

Traceability

The Client-Server pattern can be deduced from the 'What is Dockers architecture?' section of the online documentation[4].

When listing the dependencies using the dependency tools, very few packages use the docker daemon. Looking at the high-level parts of docker, the main package that references the daemon is the docker container. However, the daemon seems to be referenced quite a lot when looking at table 4.1(section 4.1), that lists the packages with the highest amount of incoming references.
This shows that the container part of docker is heavily using the docker daemon.
It is remarkable that when listing the dependencies of the daemon. It seems to only reference the container. So clearly, the docker daemon is a main access point for modifying the containers.
The documentation says that only the docker api communicates with the daemon. Clearly there is no package using the docker daemon, but also the api does not seem to be referencing the docker daemon. This is because the docker api uses the daemon through means of HTTP requests. As is made clear when looking at the code in

$$api/client/cli.go$$

This contains the line:

$$client, err := client.NewClient(host, verStr, clientTransport, customHeaders)$$

Indicating that the client is referencing the daemon through means of sending HTTP requests to the daemon api.

Source

Architectural patterns revisited – a pattern language, P. 29 [29]

Issue

Docker containers can not be be controlled remotely. It should be possible for Docker containers to be controlled remotely and a single interface should be able to control containers on multiple hosts (e.g. in the cloud).

Additionally, certain operating systems lack the underlying technologies necessary for running containers. The inability to remotely control containers from these operating systems prevents these systems from using docker in any way.

Solution

Docker uses a Client-Server architecture. The client, a binary supplying a command-line interface, acts as the primary interface for the user. The user enters commands into this client, which are then sent to a server: the Docker daemon.

Assumptions/Constraints

- The docker daemon should support the API calls made by the client. This means that if the client is outdated, the daemon is assumed to have the legacy controllers needed to still provide the wanted functionality.
- The REST interface that the daemon offers is assumed to be available and accessible by the client

Rationale

By separating the client and server it is possible to use the same client to issue commands to different daemons, running on different hosts. It is also possible to use the client on operating systems that do not support running containers.

Implications

The use of the Client-Server pattern results in two different executable binaries: a daemon and a client.

The use of the Client-Server pattern increases the interoperability, since the client can send commands to daemons running on remote machines and the local machine.

Additionally, the portability is increased, since the client can run on Operating Systems that are unable run containers themselves.

There is also an impact on security, because the communication between the client and the server needs to be secured. Because docker is composed of different layers, securing this communication can be done from a single place. This is discussed in section 5.9, which discusses the layers pattern.

Related Patterns

- Broker

5.9 Layers

Layers pattern is a architectural pattern that consists of several layers, which each layer provides a set of services to the layer above and uses the services of the layer below. Docker implements this pattern to separate the concerns of each component with independent functionalities. Layers pattern implementation of Docker, based on our investigation, is shown in Figure 5.9.

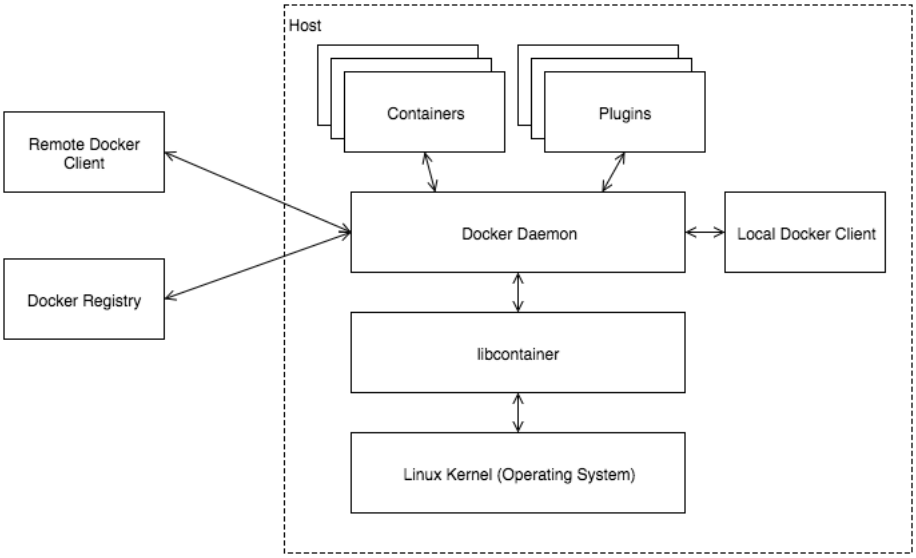


Figure 5.9: A layered overview of the Docker.

Traceability
Layers is implicitly mentioned in the Docker architecture in the online documentation, specifically in the *What is Docker's architecture?* section [4].

Source
Architectural patterns revisited – a pattern language, P. 29 [29]

Issue
Docker provides several features (e.g. hosting the image, creating image, running container, and daemon supervision). Each of the features vary in terms of the functionality they give. Single implementation of features will certainly result in abundant usage of resources, because hosting the image means there must be huge amount of disk space available.

Overlapping may also occur since multiple Docker instances may use the same image or configuration file, which could be stored and organized elsewhere. Docker also has several dependencies, especially the Linux kernel components (`namespaces` and `cgroups`) for OS level virtualization. Furthermore, Docker mostly needs remote supervision or control, since Docker usually runs in the cloud. Therefore, the features must be separated into several independent components and their dependencies and communication have to be clearly separated and maintained.

Assumptions/Constraints
Docker is able to run only in Linux kernel since it uses `NAMESPACES` and `CGROUPS` technology, which is only available in Linux kernel. However, Docker is also possible to be run on other OS although using virtual machine. The connection between local and remote components are carried out through secured TCP/IP connection.

Solution
Docker is separated into several components that have their own functionality, as can be seen in Figure 5.9. The main components are:

- Containers
- Docker daemon
- Local Docker client
- Libcontainer
- Linux kernel

Everything inside the dotted box is located on the same host. Inter-process communication is utilized to communicate between Docker, containers, libcontainer, and Linux kernel components, but Docker daemon, plugins, and Docker client communicate using Docker API [5]. Remote connections are managed through secured TCP/IP based connection.

Rationale

Layers are very good in terms of sharing and reusability. In this way, several Docker daemon that uses the same image can just fetch it through the Docker registry. Using Docker registry also reduce disk space for storing images and increase easiness of sharing images. Layers pattern also supports connection to other layer, such as connection to plugins. It is also manageable to supervise remote Docker daemon through remote Docker client.

Implications

This pattern gives positive implication to portability as clear separation of containers, Docker daemon, Docker registry, and Docker client makes Docker portable. Utilizing Docker registry also promotes sharing images, which make it easier to deploy an application without knowing what platform running below it.
A host is open to public in the communication of Docker daemon and Docker registry and remote client. This pattern provides the ability to secure all the communication from single place. This can increase the security, because focus on security is reduces to single place, meaning that strengthening the security there makes everything more secure.
However, a security breach will effect the entire system and not only a single component. This security layer can be easily replicated, having a positive impact on reliability. The security is defined in a single place, and so the level of security that is defined there can be expected from each component.

Related Patterns

- Client-server
- Shared repository

5.10 Shared/Active repository

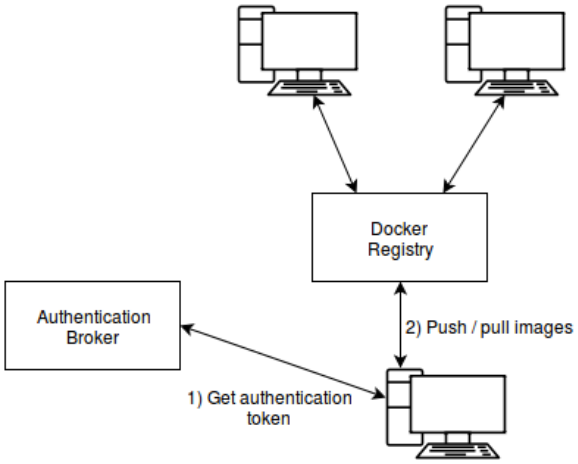


Figure 5.10: The shared repository with brokered authentication

Using the Docker files and the docker commands, personalized images can be created that support a specified project. Docker allows these images to be stored in a docker registry, earlier discussed in subsection 4.1.5. Anyone with access to this registry can pull this image and use it. Users of the application a specific project provides, are able to pull the image. This allows running the application, thereby obtain the desired functionality. The registry also allows the image to be easily shared with other developers working on any related project. This makes the docker registry a shared repository.

The Docker registry also contains functionality for sending notifications when certain events occur [5]. This makes the docker registry an active repository as well. Figure 5.10 shows the illustration of Shared/Active repository pattern in Docker.

[5] `https://github.com/docker/docker/blob/master/api/common.go`

Traceability

The Shared Repository pattern can be deducted from the online documentation: [6] "The Registry is a stateless, highly scalable server side application that **stores** and lets you **distribute** Docker images. A registry is a storage and content delivery system." Everything dealing with the registry is in the "Registry folder" in the Docker repository.

Source

Patterns of Enterprise Application Architecture, P.322 [15]
Architectural Patterns Revisited - A Pattern Language, P.13 [29]
Pattern-oriented Software Architecture - Volume 4, P.202 [14]

Issue

Docker provided a way for the user to control the storage and distribution of images. The user wants to be alerted of new events happening in the registry through notifications.

Assumptions/Constraint

The docker registry is dedicated to handling only docker images. It does not provide shared storage that allows storing any other kind of data. This makes sure that the images stored in the registry are runnable and do not crash in critical ways.

Solution

Docker utilizes a repository to store images. Users who need a particular image may fetch required images through TCP/IP connection. Users interact with a registry by using docker push and pull commands.

Rationale

After the integration of the Shared Repository Pattern each has one central repository containing their images and they can it access using a logging. Users can also access images from others users: The registry is a sharing unit.

Implications

Users can get their own images from the public registry and also the ones created by other users so the registry is a sharing unit. The use of this pattern enhances Re-usability, Changeability, Maintainability, Integrability because the Registry is central.

Related Patterns

Publish Subscribe

- Brokered Authentication

5.11 Publish-Subscribe

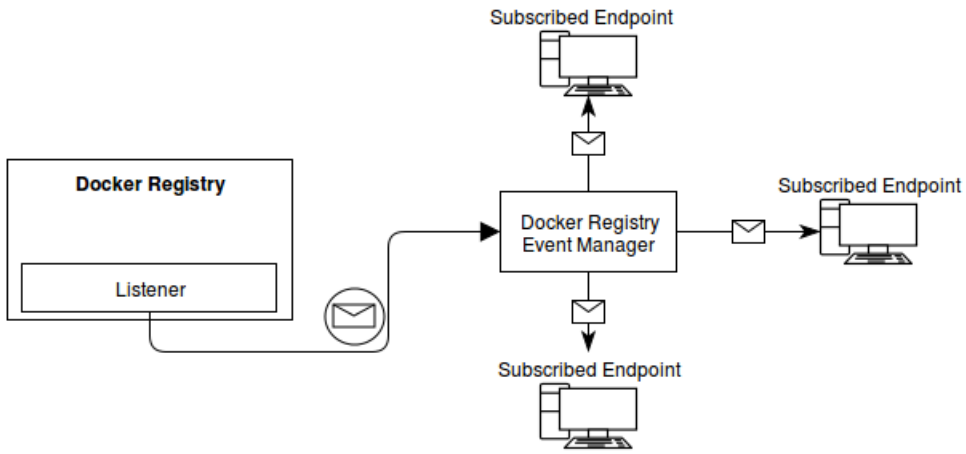


Figure 5.11: Docker Registry's event manager and subscribed endpoints

Traceability

The notification mechanism of the Docker Registry uses the Publish Subscribe Pattern [5].

Source

Pattern Oriented Software Architecture- Volume 4, P.234 [14]
Architectural Patterns Revisited – a pattern language, P.32 [29]

Issue

The user wants to be alerted about certain events/changes that occur in a registry. The notification system needs a mechanism in order to send those events to the user.

Assumptions/Constraints

This pattern is used at a lower level in the Docker Registry/Active Repository Pattern.

Solution

The Publish-Subscribe pattern allows publisher to send messages to endpoints which are subscribed to these messages.

As can be seen in Figure 5.11, a listener listens for events and forwards these to the event manager. The event manager forwards messages of this event to all the subscribed endpoints over HTTP.

Rationale

Using the Publish-Subscribe pattern allows users to configure endpoints, which will receive notifications about the events/changes that occur in the registry.

Implications

The Publish-Subscribe Pattern enhances modifiability because publisher and subscribers are decoupled.

The portability is increased using this pattern, because the decoupling of the publishers and subscribers (combined with the messages send over HTTP) means that the subscribers can be in a completely different environment than the registry itself.

Related Patterns

Shared/Active Repository Pattern

5.12 Brokered Authentication

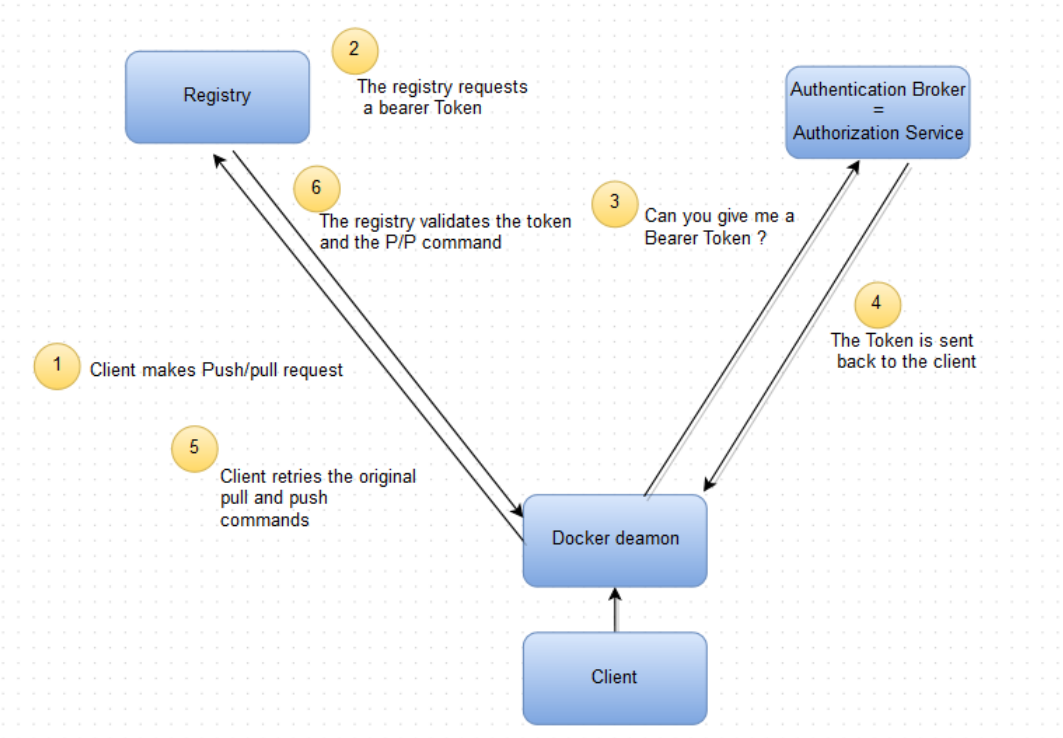


Figure 5.12: Authentication process of the Docker registry

Traceability

From the v2 of Docker the authentication is done through a central service.
The Brokered Authentication pattern can be deducted from the source code through the "auth.go" and the "token.go" files from the Docker Registry Github Repository.
Docker Authentication [7]

The main functions for authentication are :

- func tryV2TokenAuthLogin(authConfig *types.AuthConfig, params map[string]string, registryEndpoint *Endpoint)
- func loginV2(authConfig *types.AuthConfig, registryEndpoint *Endpoint, scope string)

This service is used by the official Docker Registry to authenticate clients and verify their authorization to Docker image repositories.

Source

Brokered Authentication Pattern[28]

Issue

The Docker Registry needs an authentication system in order to ensure security of pulling and pushing images.

Assumptions/Constraints

Solution

An authentication broker that both parties trust independently, issues a security token to the client. The client can use this token to authenticate himself. This means that there is no direct relationship between the user and the registry.

By using the Brokered Authentication the access to the Docker Registry is controlled for each user.

Rationale

Implication

After the integration of the Brokered authentication the textbfSecurity for the registry is ensured. The authentication broker manages trust centrally. This eliminates the need for each client and service to independently manage their own trust relationships.

Related Patterns

Shared/Active Repository Pattern

5.13 Plugin

Plugin pattern basically expands the capability of a particular software. Plugin pattern also provides centralized, runtime configuration[15]. An illustration of plugin pattern is shown in Figure 5.13.

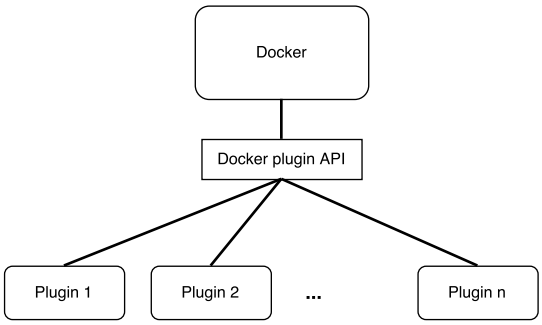


Figure 5.13: Plugin pattern illustration for Docker.

Traceability

The existence of Docker plugin becomes apparent from its documentation, which describes the plugin feature of Docker [3].

Additionally, the directories `docker/pkg/plugin/`[6] and `docker/daemon/graphdriver/plugin.go`[7] (among others) in the project's repository contain the code for discovering plugin and the interfaces the plugin should implement.

Source

Patterns of Enterprise Application Architecture, P. 499 [15]

Issue

Several of Docker users desired functionalities are not natively provided by docker. Docker provides the mechanism so that third party custom-built tools that provide this missing functionality to extend Docker. These customizations mean that third party developers are able to write tools, extending Docker's core functionality[20]. The additional functionality that is added, is only usable during run time.

Assumptions/Constraints

- plugin can only extend the functionality of the Docker components if necessary APIs are provided for certain need or use.

Solution

Docker implements the plugin pattern to link several Docker's extendable component's interfaces with third-party code at runtime.

Rationale

By using plugin pattern, any feature that is currently not supported natively by Docker can be added. The users can also develop a custom code that will exploit Docker APIs to fulfill their need.

Implications

The use of the plugin pattern means that the adaptability increases, because plugin allow the application to be adapted with new features. For the security concern, it means that there is extra communication over the API which has to be secured. The security of the plugin themselves cannot be guaranteed by Docker.

Related Patterns

- Broker

5.14 Broker

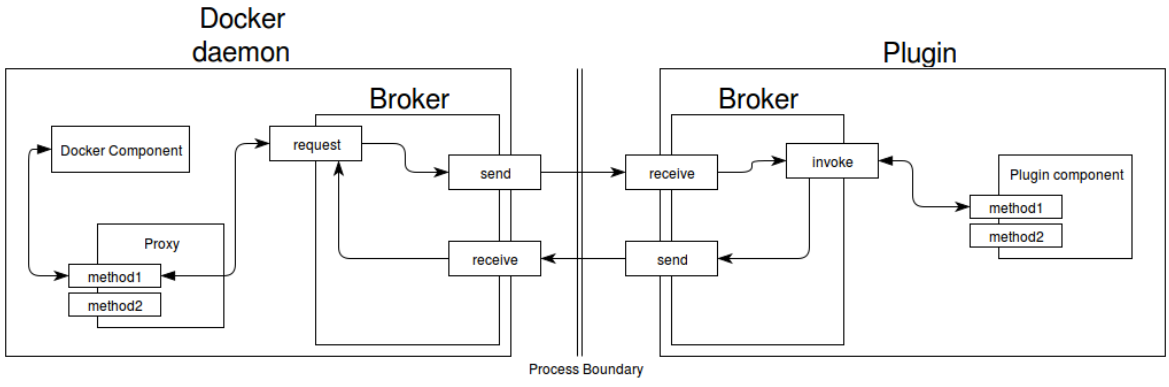


Figure 5.14: Broker for communication between Docker daemon and a plugin.

Traceability

The use of the Broker pattern can be found in the proxy package in the source code: `docker/pkg/proxy/tcp_proxy.go`[8].

[6] `https://github.com/docker/docker/tree/master/pkg/plugin`
[7] `https://github.com/docker/docker/blob/master/daemon/graphdriver/plugin.go`
[8] `https://github.com/docker/docker/blob/master/pkg/proxy/tcp_proxy.go`

Source

Pattern-oriented Software Architecture - Volume 4, P.237 [14]
Architectural Pattern Revisited - A Pattern Language, P.34 [29]

Issue

Docker consists of multiple processes. For example, the client and daemon are separate processes and also the plugins are separate processes.

These processes have to communicate with each other. However, dealing with the challenges of inter-process communication in the application code would greatly increase the complexity. The application components should be shielded form the details of the inter-process communication.

Assumptions/Constraints

- All the communication between the components in different processes have to go via the Broker.

Solution

Use a Broker to separate the logic for communication between the processes from the application functionality. Use a component-based model (using Proxies) to allow local components to invoke methods on components in remote processes as if they were local.

Rationale

The Broker pattern allows for and handles all the communication between the different processes.

This means that the logic for inter-process communication does not have to be implemented on a per-component basis.

Implications

Using the Broker pattern is good for the portability, because it does not matter if a component is part of a different process (running locally or remotely).

This pattern is also good for the security, since the Broker is a single entry point for all the inter-process communication, which makes it easier to secure. This has been done by using Transport Layer Security for the communication between the processes.

Additionally, the interoperability (compatibility) is increased, because it becomes possible for two components (in two different processes) to exchange information.

The Broker does however introduce a performance overhead, which decreases the performance efficiency.

Related Patterns

- Plugin
- Proxy
- Client-Server

6 Architecture review

This chapter presents the evaluation of documented patterns and overall system architecture against the key drivers of the architecture presented in Section 3.2. The Pattern-Based Architecture Reviews (PBAR) [19] method is used to carry out the evaluation. Force Resolution Maps (FRM) are also utilized to present the result.

Table 6.1: Force Resolution Maps definition.

Value	Definition
−2	Big negative impact
−1	Small negative impact
0	Neutral
+1	Small positive impact
+2	High positive impact

6.1 Patterns

This section describes the contributions of the patterns with regard to the key drivers. An FRM is included in each subsection to give the overview of the contributions.

6.1.1 Client-Server

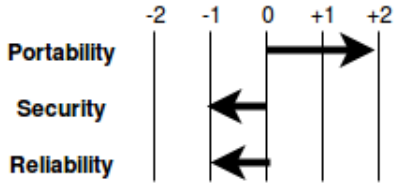


Figure 6.1: Force Resolution Map for Client-Server pattern

For the Client-Server pattern, the portability of the system increases, because the client and server (daemon) can be located on completely different systems.

The security has a slight negative impact because for communication between the client and server to be possible, the client and server have to expose an interface, which has to be secured.

Reliability also has a small negative impact, because the communication between client and server relies on underlying communication infrastructure that could fail.

6.1.2 Layers

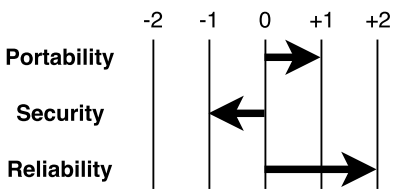


Figure 6.2: Force Resolution Map for Layers pattern.

This pattern gives positive implication to portability as clear separation of containers, Docker daemon, Docker registry, and Docker client makes Docker portable and loosely coupled. Utilizing Docker registry also promotes sharing images, which make it easier to deploy an application without knowing what platform running below it. Exposing TCP/IP connection, specifically REST, contributes to negative effect on the security. It means that any unwanted connections or attacks could possible exploit this hole. However, this weak point is compensated by other patterns.

Furthermore, decoupling some components means that the components below or above can be replicated in order to gain more reliability as it has more availability. Thus, this patterns give positive effect to the reliability.

The summary of the Layers pattern evaluation is shown in FRM in Figure 6.2.

6.1.3 Shared/Active repository

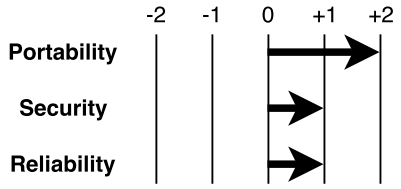


Figure 6.3: Force Resolution Map for Shared/Active Repository pattern.

According to Wikipedia, a reliable service is one that notifies the user if delivery fails, while an "unreliable" one does not notify the user if delivery fails. Based on this definition we can consider that the registry is reliable because in case a push command doesn't work the user is notified (same system as Github for example) by an error message/prompt and can try again.

Security is enhanced because the access to the registry is controlled by a login and passwords for the users. Moreover, the push and pull commands in the registry are also secure (see Brokered Authentication Pattern).

The docker registry is good for portability because whatever the environment of the user everybody can subscribe and access this functionality.

The Shared Repository pattern has effects on other quality attributes such as integrability, modifiability and re-usability. Figure 6.3 shows the FRM of Shared/Active Repository pattern.

6.1.4 Publish-Subscribe

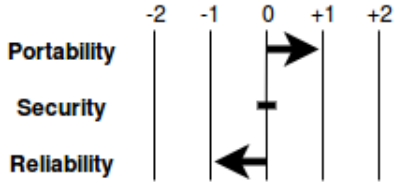


Figure 6.4: Force Resolution Map for Publish-Subscribe pattern

The publish subscribe pattern allows the event occurring in the Docker Registry to be triggered to the user who has previously subscribed to them. When those events happen the user receives a webhook notification.

As mentioned in the documentation[5], Docker's implementation of this pattern places messages in an in-memory queue. Attempts to deliver messages are repeated until they are accepted. This means that when an endpoint is not available for a longer time, the message queue gets larger and drops messages. This has a negative impact on the reliability.

The portability is enhanced by this pattern because whatever the platform used everyone can subscribe to the notification system.

Security is not affected by this pattern.

6.1.5 Brokered Authentication

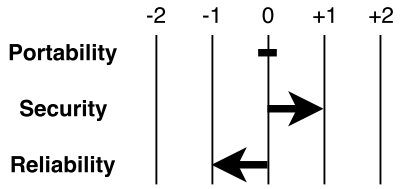


Figure 6.5: Force Resolution Map for Brokered Authentication pattern.

Reliability is ensured because if the user cannot push/pull, an error message is prompt. In that way, the user is notified in case of a failure in his authentication. (see definition of Reliability in the Shared Repository Pattern evaluation).

However, there is also a drawback in the reliability because the centralized authorization service can be a single point of failure. It may need to be online and available, otherwise the client can't get his token. If the brokered authentication becomes unavailable, the registry and the client can't communicate with each other. Redundant and back-up authentication broker are the example solutions of this problem.

Security is ensured because the registry and the client do not communicate directly. The Bearer token is a barrier of protection between the two components. However security tokens must be signed by the authentication broker. If they are not, their integrity cannot be verified. This could result in attackers trying to issue false tokens.

Portability isn't affected by the use of this pattern.

Figure 6.5 concludes the contribution of this pattern.

6.1.6 Plugin

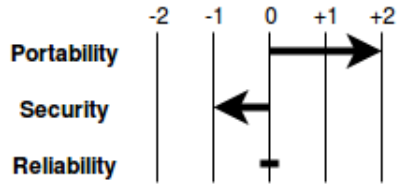


Figure 6.6: Force Resolution Map for Plugin pattern

The Plugin pattern greatly increases the portability, because it increases the adaptability by allowing third parties to develop extensions for Docker.

It has a slight negative impact on the security, because there is communication between the daemon and plugin process which has to be secured. Additionally, the security of the plugins is not controlled by Docker, so defects in a plugins security can affect the security of the Docker daemon as well.

Individual plugins can fail, which affects the reliability. However, because plugins function as a replacement for components in Docker and the implementation of the plugins themselves are not part of Docker, we do not consider this a negative impact on the reliability of Docker.

6.1.7 Broker

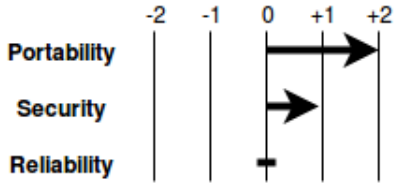


Figure 6.7: Force Resolution Map for Broker pattern

The use of the Broker pattern is very beneficial for the portability, because it allows the use of components in different processes, potentially running on different machines.

This pattern is also positive for the security of Docker, because it is a single entrypoint for all inter-process communication, which is easier to secure than multiple entrypoints.

6.2 Overall system

7 Recommendations

8 Conclusion

This document have presented the architecture recovery of Docker by identifying software patterns and performing an evaluation of the architecture based on the identified patterns according to their impact on the Docker's quality attributes. This is done by implementing PBAR and IDAPO method. IDAPO method is used to identify the patterns within the Docker architecture, while PBAR is utilized to perform the architecture review.

We firstly start by grasping some basic information about Docker. The findings were presented in the System Context chapter, which consists of brief explanation about Docker and its ecosystem, along with the communities supporting and developing it. We then figured out the stakeholders behind Docker project and their concerns. Based on our findings, the stakeholders can be grouped in to six groups, those are Docker developers, Software developers that utilize Docker in their project, the Open Container Initiative, cloud providers, Docker plugin developers, and software maintainers. The summary of stakeholders and their concerns can be seen in Figure 3.1. Based on their concerns, the key-drivers can be extracted and those are *portability, security, and reliability.*

Even if software designers or developers may not be aware of certain software patterns, the patterns may still be present [30]. The IDAPO process is used to identify existing patterns in a open source software. Patterns recovery is very useful for quantifying the software project on a high-level approach. By knowing the patterns inside a particular software project it is also easier to measure or evaluate it by their impact to the software's key-drivers. The IDAPO consists of 12 iterative steps that aim to achieve a robust pattern recovery inside the OSS project. The mapping of IDAPO process and the chapters of this document is described in chapter 1.

A typical evaluation of an architecture requires a lot of effort and cost, for example, ATAM method. PBAR provides a lightweight architecture review process that can be used where traditional architecture review methods would not because of their high cost. PBAR focuses the evaluation on issues of the system's' key- drivers. It makes use of discovered patterns within the architecture to properly document issues. PBAR consists of four main steps, which the third step, the review meeting, is actually a repetitive steps.

One of the reason why Docker is selected to be the target of this Pattern-based recovery and evaluation is of its popularity. The purpose this software is to make the deployment of distributed application easier. As stated in the documentation, Docker provides an integrated technology stacks, which enables the development and IT operation teams to build, ship, and run distributed applications anywhere without having to know the platform running below it.

Our investigation showed that there are seven layers exists in Docker, although more patterns are likely to be there as well. The detailed documentations of those pattern are written in chapter 5. In this project, we also implemented some insight that we achieved when working with the first project of Software Pattern. Hopefully, the enhanced insights and experiences of this second assignment would be beneficial for us in the future.

A Time Tracking

Week 1

Person	Task	Hours
Putra	Coaching session, researching about Docker, working on first draft of introduction	3.5
Fakambi	Coaching session, Research about Docker, Work on the system context	3.5
Schaefers	Coaching session, setting up initial layout and new git repository, defining key drivers, research.	6.5
Menninga	Coaching session, stakeholders and concerns	4.0

Week 2

Person	Task	Hours
Putra	Catching up after winter break, reading Docker docs, revising intro, and writing layers pattern documentation	6
Fakambi	Coaching session, Meeting for discussing the patterns	3
Schaefers	Creating scripts to find patterns, building and running docker, creating various dependency graphs, changing the key driver section, adding stakeholders and concern image	8.5
Menninga	Searching patterns, improving stakeholders, client-server and plugin patterns. Improvements to System context.	7.5

Week 3

Person	Task	Hours
Putra	Coaching session, reading docker documentation and source code, revising intro and layers pattern, small changes in logical view and evaluation	14.5
Fakambi	Coaching Session, small modifications System Context, Work on the Shared Repository, Event-Driven and Direct Authentication	12
Joris	Researching and generating dependency graphs. Coaching session (via skype). Adding a software architecture begin.	11
Menninga	Coaching session, improvements after feedback, process/logic view, client-server, plugin, proxy, changed key drivers	15

Week 4

Person	Task	Hours
Putra	Coaching session, meetings, group 3 review, working on feedback, rewrite and add illustration and evaluation for layers pattern, first draft of conclusion.	12
Fakambi	Coaching session, Meetings,Presentation, Processing Feedback,Improvements patterns	12

Joris	Coaching session, meetings, parsing .dot files, searching for useful dependency/code references, reviewing and processing reviews	11
Menninga	Coaching session, meetings, reviewing document other group, evaluation for patterns, improvements to patterns	12

Week 5-6

Person	**Task**	**Hours**
Putra	Meetings, working on feedback, quick reviewing, improving things, and added several figures.	9
Fakambi		
Joris		
Menninga		

Bibliography

[1] Chris Dawson. Who are the docker developers? http://thenewstack.io/who-are-the-docker-developers/, 2014. [Online; accessed 9th January 2015].

[2] Docker. Docker - build,ship, and run any app, anywhere. https://www.docker.com/, 2015. [Online; accessed 13-December-2015].

[3] Docker. Docker - understand docker plugins. https://docs.docker.com/engine/extend/plugins/, 2015. [Online; accessed 3rd January 2015].

[4] Docker. Docker - understanding the architecture. https://docs.docker.com/engine/introduction/understanding-docker/, 2015. [Online; accessed 30th December 2015].

[5] Docker. Docker notifications. https://docs.docker.com/registry/notifications/, 2016. [Online; accessed 17-Jan-2016].

[6] Docker. Docker registry. https://docs.docker.com/registry/, 2016. [Online; accessed 15-Jan-2016].

[7] Docker. Docker registry v2 authentication via central service. https://docs.docker.com/registry/spec/auth/token/, 2016. [Online; accessed 15-Jan-2016].

[8] Docker. Docker: Where and what you can contribute. https://docs.docker.com/opensource/project/who-written-for/, 2016. "[Online; accessed 13-December-2015]".

[9] Docker. Github: Docker api - a directory containing code pertaining to the docker api. https://github.com/docker/docker/tree/82401a4b13c0581908d46dbc3e7b475c7de48334/api, 2016. "[Online; accessed 17-December-2015]".

[10] Docker. Github: Docker pkg - a collection of utility packages used by the docker project without being specific to its internals. https://github.com/docker/docker/tree/82401a4b13c0581908d46dbc3e7b475c7de48334/pkg, 2016. "[Online; accessed 17-December-2015]".

[11] Docker. Github: Package mflag - command-line flag parsing. https://github.com/docker/docker/tree/82401a4b13c0581908d46dbc3e7b475c7de48334/pkg/mflag, 2016. "[Online; accessed 17-December-2015]".

[12] Docker. Run tests and test documentation. https://docs.docker.com/opensource/project/test-and-docs/, 2016. "[Online; accessed 13-December-2015]".

[13] Docker. What is docker? https://www.docker.com/what-docker, 2016. [Online; accessed 3-January-2016].

[14] K Henney F Buschmann and D.C. Schmidt. *Pattern-oriented Software Architecture - Volume 4*. John Wiley & Sons, Ltd, 2007.

[15] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[16] GitHub. Docker - the open-source application container engine. https://github.com/docker/docker, 2015. [Online; accessed 13-April-2015].

[17] Google. The go programming language. https://golang.org/project, 2015. [Online; accessed 9th January 2015].

[18] N. Harrison, P. Avgeriou, and U. Zdun. Using patterns to capture architectural decisions. *Software, IEEE*, 24(4):38–45, July 2007.

[19] NeilB. Harrison and Paris Avgeriou. Using pattern-based architecture reviews to detect quality attribute issues - an exploratory study. In James Noble, Ralph Johnson, Uwe Zdun, and Eugene Wallingford, editors, *Transactions on Pattern Languages of Programming III*, volume 7840 of *Lecture Notes in Computer Science*, pages 168–194. Springer Berlin Heidelberg, 2013.

[20] Adam Herzog. Extending docker with plugins. https://blog.docker.com/2015/06/extending-docker-with-plugins/, 2015. [Online; accessed 3rd January 2015].

[21] hirokidaichi. Goviz - a visualization tool for golang project dependency. https://github.com/hirokidaichi/goviz, 2014. [Online; accessed 21-December-2015].

[22] Open Container Initiative. Github: Open container initiative - project to create open industry standards around container formats and runtime. https://github.com/opencontainers/runc/tree/54b07da69e0dd5924e279b852e3693e166cc54d4, 2016. "[Online; accessed 17-December-2015]".

[23] ISO. Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. http://iso25000.com/index.php/en/iso-25000-standards/iso-25010, 2011.

[24] kisielk. Godepgraph - a go dependency graph visualization tool. https://github.com/kisielk/godepgraph, 2015. [Online; accessed 21-December-2015].

[25] Philippe Kruchten. Architectural blueprints — the "4+1" view model of software architecture. *IEEE Software 12, pp.42-50*, 1995.

[26] W.G. Menninga. Benchmarking Tool for the Cloud. Bachelor Thesis, University of Groningen, 2015.

[27] Maureen OǴara. Ben golub, who sold gluster to red hat, now running dotcloud. http://maureenogara.sys-con.com/node/2747331, 2013. [Online; accessed 13-December-2015].

[28] SOA Patterns. Brokered authentication. http://soapatterns.org/design_patterns/brokered_authentication, 2016. [Online; accessed 15-Jan-2016].

[29] Architectural patterns revisited a pattern language. Architectural patterns revisited – a pattern language. *Software, IEEE*, Decembber.

[30] Klaas-Jan Stol, Paris Avgeriou, and Muhammad Ali Babar. Design and evaluation of a process for identifying architecture patterns in open source software. In Ivica Crnkovic, Volker Gruhn, and Matthias Book, editors, *Software Architecture*, volume 6903 of *Lecture Notes in Computer Science*, pages 147–163. Springer Berlin Heidelberg, 2011.

[31] John Paul Walters, Vipin Chaudhary, Minsuk Cha, Salvatore Guercio Jr, and Steve Gallo. A comparison of virtualization technologies for hpc. In *Advanced Information Networking and Applications, 2008. AINA 2008. 22nd International Conference on*, pages 861–868. IEEE, 2008.

[32] Felter Wes, Ferreira Alexandre, Rajamony Ram, and Rubio Juan. An updated performance comparison of virtual machines and linux containers. Technical Report RC25482, IBM, Research Division, Austin Research Laboratory 11501 Burnet Road Austin, TX 78758 USA, July 2014. "[Accessed 22-Jan-2016]".