Joker: Implementing Docker Containerization Functionality

Taras Yaroshko, Nazar Demchuk, Oleksandr Shchur, Liubomyr Oleksiuk
Mentor: Hermann Yavorskyi
Computer Science Programme, Faculty of Applied Sciences, Ukrainian Catholic University
Lviv, Ukraine

***Abstract*—This report presents the implementation of Joker, which is the analog of Docker written in Rust and C++. The report reveals the design, features, and challenges encountered during the software development process.**

***Index Terms*—Docker, containerization**

I. Introduction

In the landscape of the modern software development industry, there is always a demand for the scalable, flexible, and efficient deployment of applications. Docker, a tool for containerization (a much more lightweight form of virtualization that enables the packaging of an application and its dependencies/configurations into a standardized unit called a container), has revolutionized how we build, ship, and run applications.

II. How Docker Works

Before delving into Joker's implementation details, it is important to understand the fundamental principles of Docker and its components, which are crucial in understanding the containerization workflow, file loading, and resource management, Fig. 1.

Docker utilizes the containerization concept to encapsulate applications and their dependencies, enabling seamless deployment across different environments. Containers are not only portable and lightweight but also ensure consistency in the execution of applications. Containers are the answer to the question, "Why does it work on your machine but does not seem to work on mine?".

A. Docker Engine

At the core of Docker is the Docker Engine. It is the software responsible for building, running, and managing containers. The engine consists of a server and a lightweight (but powerful) command-line interface (CLI) that communicates with the Docker daemon, managing container operations.

B. Containers

Containers are instances of Docker images. They encapsulate the application and its dependencies, running in an isolated environment. Containers ensure consistency across various development, testing, and production environments, fostering a "run anywhere" philosophy.

C. Dockerfile

To create a Docker image, you have to use a text file called a Dockerfile. This file contains a set of instructions for building a Docker image, specifying the base image, application code, dependencies, and runtime configurations.

III. Solution Overview

Joker emulates the interface and functionality of Docker by using Linux kernel tools such as namespaces and cgroups to isolate container execution and manage the existing resources. The general structure of the project is presented in Fig. 2.

IV. Details Of Implementation

A. Server-Client Communication

In our C++ implementation, the server-client communication system is designed to facilitate asynchronous file transfers and execution. Here is how it works:

1) Listening for Connections: The Server creates a socket and binds it to a specified port (e.g., 8080). It listens for incoming client connections and accepts them when they arrive.

2) Connecting to the Server: The client creates a socket and connects using it to the Server's IP address and port.

3) Client Request Processing: The Server's main thread continuously accepts client connections and processes their requests. Upon receiving a client request, it distinguishes between different types based on the first byte received:

- If the first byte is 0, it indicates the container files transfer request.
- If 1 – a request for a trace of daemon *stdout* outputs.
- If 2 – a request for the *stdout* output of a certain container.

4) Sending the Container Data: The client can send binary files and corresponding configuration files to the Server. It sends the Server the names of the files, their respective sizes, and the files themselves (binary file to be executed or configuration file).

5) Requesting Log and Trace Information: The client can request log information from the Server by sending a specific request identifier. Upon receiving the log request, the server sends back the log content to the client, which is the trace of the daemon or the log of the specific container.

6) Closing the Connection: After completing the file transfer or log request, the client closes the connection with the Server.

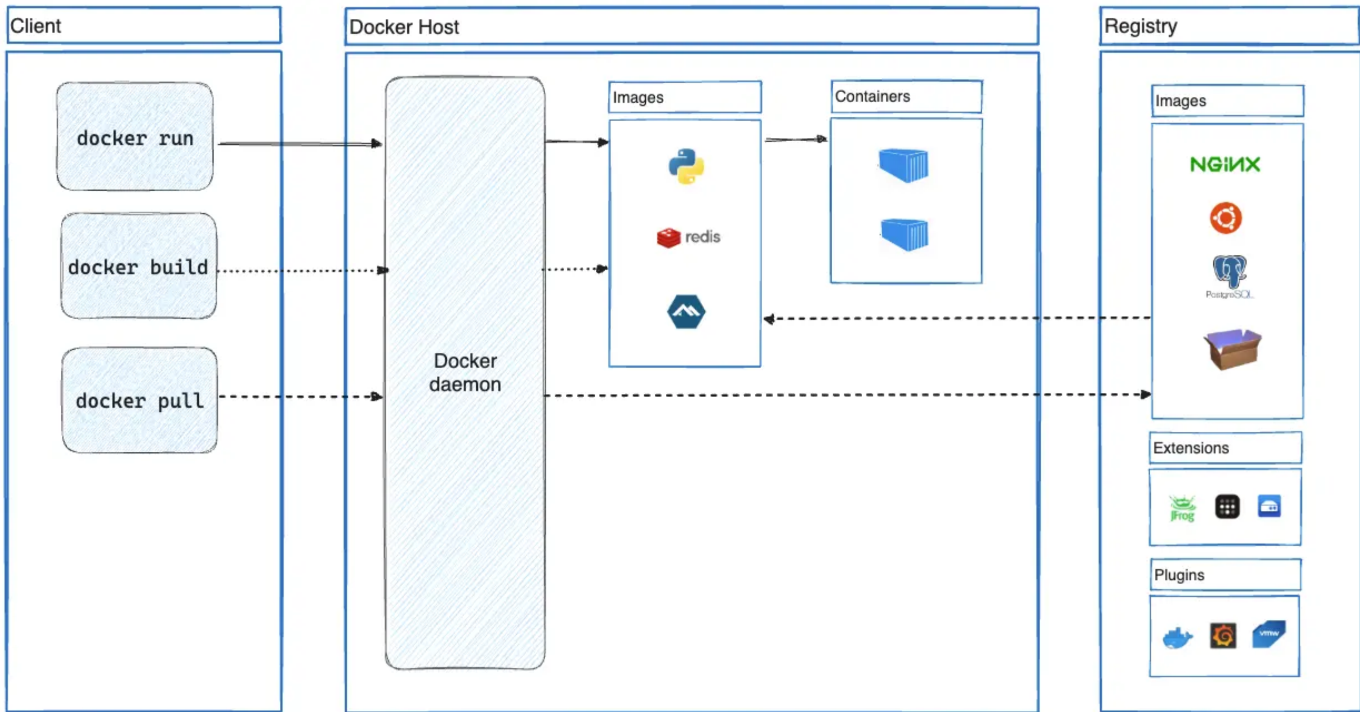

Fig. 1. The architecture of Docker [1].

B. Namespaces

Namespaces [2] is the kernel feature providing a way to virtualize the different aspects of program-system interaction. There are numerous namespaces, each responsible for some part of that interaction. The more namespaces are used, the better isolation from the host system can be ensured.

Let's review the functionality within namespaces, which we used as layers of isolation for the containers:

- **Mount Namespace**
  Each container has an isolated file system environment, which means that changes to the file system within a container do not affect the host or other containers. This namespace was the most important one, as it was crucial to provide separate mount points for each container to ensure adequate encapsulation.
- **IPC Namespace**
  Allows each container to have independent shared memory segments, semaphores, and message queues within the container's processes.
- **Network Namespace**
  Isolates network-related resources so that every container can have individual network interfaces, IP addresses, routing tables, etc.
- **PID Namespace**
  It isolates the process ID space (containers have their own PID namespace, and processes within a container are unaware of processes outside the container). This isolation is fundamental for maintaining container independence.
- **UTS Namespace**
  It isolates the hostname and NIS (Network Information Service), which allows every container to have a unique hostname and domain name independent of other containers and the host system.
- **Time Namespace**
  Allows processes to see different system times similar to the UTS namespace.
- **User Namespace**
  Allows mapping of user and group IDs inside the container to different IDs outside the container. That means that users inside the container can have restricted privileges, while the container can interact with the host as a non-root user, so it avoids the potential risk associated with running processes as root within the container.

C. cgroups

Cgroups [3] allow us to define resource constraints for a group of processes, preventing one container from monopolizing resources at the expense of others. This helps in achieving better resource utilization and isolation in containerized environments. Cgroup manager provides access to create, configure and store cgroups.

Cgroups features are somewhat similar to namespaces. However, they are used in the context of hardware resource usage restrictions.

In our implementation, it is possible to limit the container in the use of the processor, the speed of reading and writing, swap, soft and hard (the container process will be killed in case of using more memory) memory limits, a limit on the number of processes that can be created in the container.

Fig. 2. Joker UML diagram.

V. INTERFACE

A. Client

*1) **joker: Command Line Interface Tool** :* The Joker CLI provides a comprehensive set of commands, enabling users to perform various tasks related to containerization. Here is the list of Docker commands that we implemented:

- **joker add**
  Function: Add a new daemon with a specific IPv4 address and port.
  Example:
  `joker add daemon_name --ip 127.0.0.1 --port 8080`
- **joker checkout**
  Function: Switch to a specific daemon
  Example: `joker checkout daemon_name`
- **joker send**
  Function: Send the configuration file to the daemon
  Example: `joker send config_file_name`
- **joker run**
  Function: Run a specified container on the current daemon
  Example: `joker run container_name`
- **joker trace**
  Function: Trace the events on the daemon (use stdout by default).
  Example: `joker trace`
- **joker logs**
  Function: Get the output of the specific container (use stdout by default).
  Example: `joker logs container_name`

B. Configuration Files

In order to have a generalized format of configuration files for both containers and namespaces/cgroups, we created our own analog of Dockerfile – a file with extension *.joker*. Here's how the configuration file for the container (which is sent to the daemon after the container) looks:

Listing 1. Container configuration file example

```
Container name: communicate_client
IPC namespace name: ipc_ns
User namespace name: user_ns
Mount namespace name: mnt_ns
PID namespace name: pid_ns
Network namespace name: net_ns
Time namespace name: time_ns
UTS namespace name: uts_ns
Cgroup name: cgroup_example
```

Also, here are the examples for cgroups and namespaces configuration files, respectively:

Listing 2. Cgroups configuration file example

```
Cgroup
Cgroup name: pid-limit
Pids max: 1
```

Listing 3. Container configuration file example

```
Namespace-template
ID: 1
Namespace type: 1
New root filesystem path: ./data/alpine_rootfs
Filesystem: ext4
Put old: .put_old1

Namespace-template
ID: 2
Namespace type: 6
Hostname: hostname from template!

Namespace-template
ID: 3
Namespace type: 2

Namespace
Namespace name: test_1
Template-Id: 1
Namespace type: 1

Namespace
Namespace name: test_2
Template-Id: 1
Namespace type: 1

Namespace
Namespace name: uts_1
Template-Id: 2
Namespace type: 6

Namespace
Namespace name: pid_1
Template-Id: 3
Namespace type: 2
```

VI. Future Plans

As for the future plans, we would like to incorporate a few more features into our implementation:

1) Add a TLS handshake (a process when a client and server exchange specific messages to verify each other and establish the encryption algorithms and session keys) to ensure a secure connection between the communicating sides.
2) Develop a standardized format for all the configuration files – container configs (with the extension *.joker*) and daemon configs (with the extension *.ini*).

The project's GitHub repository: *https://github.com/Joker-containers*.

References

[1] Docker Documentation, *Docker Overview*. docs.docker.com/get-started/overview/.

[2] Linux Documentation Project, *Namespaces Manual Page*. https://man7.org/linux/man-pages/man7/namespaces.7.html.

[3] Linux Documentation Project, *Cgroups Manual Page*. https://man7.org/linux/man-pages/man7/cgroups.7.html.