Universitatea din București

Facultatea de Matematică și Informatică

Aplicație de gestionare a mediilor de virtualizare

LUCRARE DE LICENȚĂ

Coordonator științific

Irofti Paul

Student

Butnaru Ioan-Sorin

București, România

Iulie 2020

Cuprins

Abstract

Abstract-ul lucrării.

Capitolul I

Introducere

I.1 Prezentarea generală a temei

Containerizarea, sau virtualizarea la nivel de sistem de operare este o paradigmă a sistemului de operare în care *kernelul* permite existența a mai multor instanțe izolate ale utilizatorului. Astfel de instanțe, numite containere (Solaris, Docker), zone (Solaris), servere private virtuale (OpenVZ), partiții, medii virtuale (VEs), *kerneluri* virtuale (DragonFly BSD) sau închisori (FreeBSD jail sau chroot jail), pot părea ca niște computere reale din punctul de vedere al programelor care rulează în ele. Un program care rulează pe un sistem de operare obișnuit poate vedea toate resursele (dispozitive conectate, fișiere și foldere, *network share*-uri, putere a procesorului, capacități hardware cuantificabile) ale acelui computer. Cu toate acestea, programele care rulează în interiorul unui container pot vedea doar conținutul și dispozitivele destinate containerului. [1]

Pe sistemele de operare similare Unix, această caracteristică poate fi văzută ca o implementare avansată a mecanismului standard *chroot*, care schimbă folderul rădăcină aparent pentru procesul de rulare curent și copiii săi. În plus față de mecanismele de izolare, *kernelul* oferă și funcții de gestionare a resurselor pentru a limita impactul activităților unui container asupra altor containere. [1]

Virtualizarea la nivel de sistem de operare este folosită în mod obișnuit în mediile de găzduire virtuale, unde este utilă pentru alocarea în siguranță a resurselor hardware finite între un număr mare de utilizatori. Administratorii de sistem o pot utiliza, de asemenea, pentru consolidarea hardware-ului unui server, prin mutarea serviciilor de pe gazde separate pe un singur server în containerele. [1]

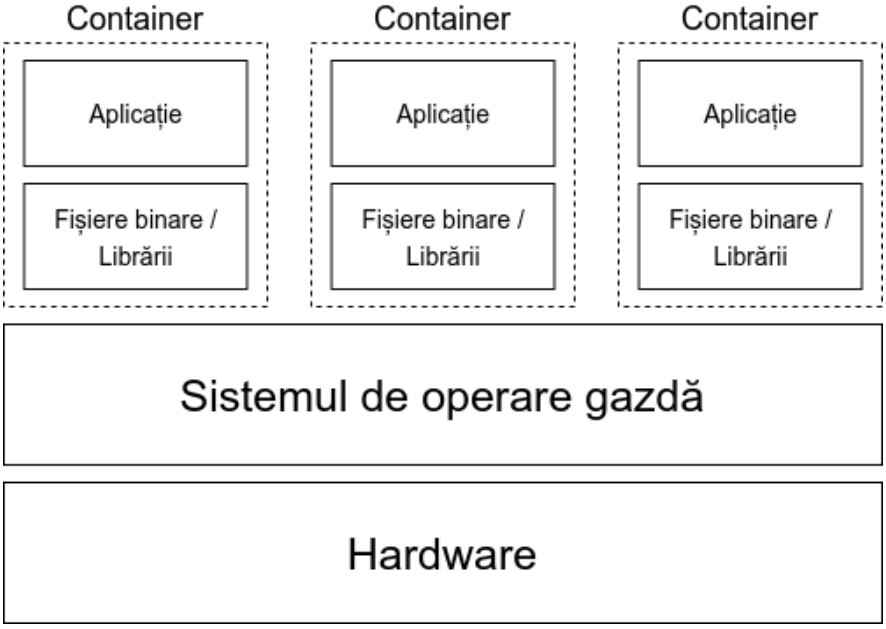


Figura I.1: Virtualizarea la nivel de sistem de operare

Alte scenarii tipice includ separarea mai multor programe în mai multe containere pentru o securitate mai bună, independență hardware și funcții adiționale de gestionare a resurselor. Implementările de virtualizare la nivel de sistem de operare, capabile să migreze direct, pot fi de asemenea utilizate pentru echilibrarea dinamică a resurselor utilizate de containere între nodurile dintr-un cluster. [1]

De asemenea, acest tip de virtualizare este folosită în dezvoltarea de aplicații pentru rezolvarea problemelor de dependențe (lipsa acestora sau conflictele dintre ele) și a diferențelor dintre platforme. [2]

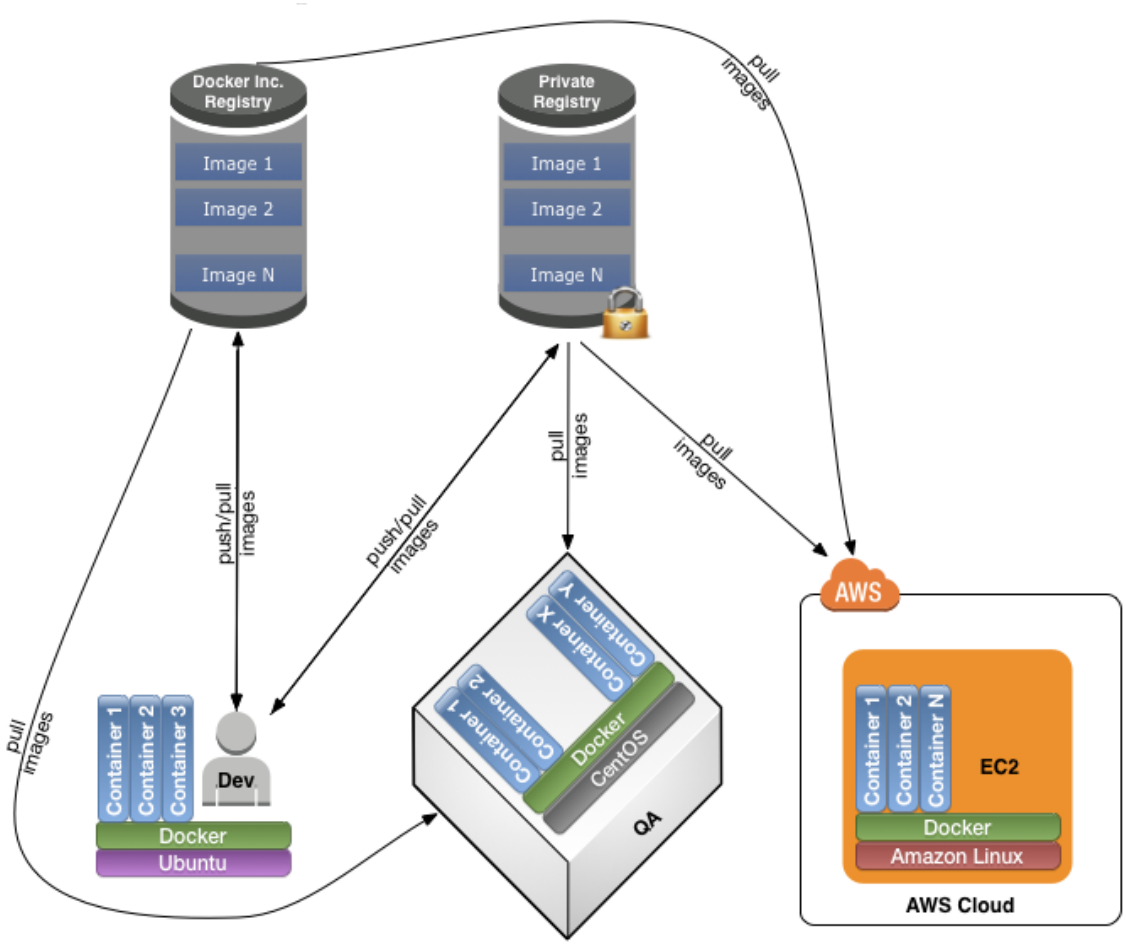


Figura I.2: Fluxul de lucru cu aplicația Docker [2]

I.2 Scopul lucrării

Scopul acestei lucrări este de a dezvolta o aplicație de virtualizare folosind funcționalitățile puse la dispoziție de către *kernelul* Linux. Această aplicație urmând ca exemple principale aplicația Docker și specificațiile OCI.

I.3 Soluții existente

I.3.1 chroot

chroot este un apel de sistem care schimbă directorul rădăcină aparent pentru procesul de rulare curent și pentru copiii săi. Acesta a fost introdus în cursul dezvoltării versiunii 7 Unix în 1979 și a fost adăugat la BSD de Bill Joy la 18 martie 1982. [3]

Operația *chroot* nu este destinată să se apere împotriva manipulării intenționate de către utilizatorii privilegiați (root). În majoritatea sistemelor, contextele *chroot* nu sunt adăugate corect și programele *chroot*-ate cu privilegii suficiente pot efectua un al doilea *chroot* pentru a ieși din mediul respectiv. Pentru a atenua riscul acestei slăbiciuni în materie de securitate, programele trebuie să renunțe la privilegiile de administrator imediat după operația de *chroot*, sau ar trebui utilizate în schimb alte mecanisme - cum ar fi închisorile FreeBSD. Unele sisteme, cum ar fi FreeBSD, iau măsuri de precauție pentru a preveni un al doilea atac *chroot*. [4]

I.3.2 FreeBSD Jail

jail este o implementare a virtualizării la nivel de sistem FreeBSD, care permite administratorilor de sistem să partiționeze un sistem FreeBSD în mai multe mini-sisteme independente numite închisori, toate care au același kernel. [5] Acesta este implementat printr-un apel de sistem, *jail(2)*, precum și o utilitate de tip userland, *jail(8)*. [6]

I.3.3 LXC

LXC (Linux Containers) este o metodă de virtualizare la nivel de sistem de operare folosită pentru rularea mai multor sisteme Linux (containere) izolate pe o gazdă utilizând un singur *kernel* Linux. [7]

Kernelul Linux oferă funcționalitatea cgroups care permite limitarea și prioritizarea resurselor (procesor, memorie, I/O, rețea etc.) fără a fi necesară pornirea unor mașini virtuale și, de asemenea, funcționalitatea de izolare a spațiului de nume care permite izolarea completă a unei aplicații vedere a mediului de operare. [7]

LXC combină cgrupurile nucleului și suportul pentru spațiile de nume izolate pentru a oferi un mediu izolat pentru aplicații. Versiunile anterioare de Docker au folosit LXC

ca driver de execuție a containerelor, deși LXC a devenit opțional în v0.9, iar suportul a fost oprit în Docker v1.10. [8][7]

I.3.4 docker

Docker este un set produse de tipul *platform as a service* (PaaS) care utilizează virtualizarea la nivel de sistem de operare pentru a livra software în pachete numite containere. Containerele sunt izolate unele de celelalte și conțin propriile programe, biblioteci și fișiere de configurare; ele pot comunica între ele prin canale bine definite. Toate containerele sunt administrate de un *kernelul* sistemului de operare și, prin urmare, folosesc mai puține resurse decât mașinile virtuale. [9]

I.4 Open Container Initiative (OCI)

Inițiativa Open Container (OCI) este proiect realizat de către fundația Linux, cu scopul de a crea standarde în jurul formatelor de containere și în jurul rulării acestora. OCI a fost lansată pe 22 iunie 2015 de Docker, CoreOS și alți lideri din industria containerelor.

OCI conține în prezent două specificații: Specificația pentru rulare (runtime-spec) și Specificația pentru imagini (imagine-spec). Specificația pentru rulare prezintă modul de a rula un „pachet de sisteme de fișiere" (*filesystem bundle*) care este despachetat pe disc. La un nivel înalt, o implementare OCI ar descărca o imagine OCI, apoi ar despacheta imaginea în sistemul de fișiere al unui *runtime* OCI. Acest pachet urmează să fie rulat de către *runtime*.

Formatul de imagine OCI conține suficiente informații pentru a lansa o aplicație pe o platforma țintă (de exemplu, comandă, argumente, variabile de mediu etc.). Această specificație definește modul de creare a unei imagini OCI, care va fi realizată în general de către un sistem de compilare, și de generare a unui *image manifest*, a unei serializări

de sistem de fișiere (*layer*) și a unei configurații a imaginii. La un nivel înalt, *image manifest*-ul conține metadate despre conținutul și dependențele imaginii. Configurația imaginii include informații, cum ar fi argumentele aplicației, medii, etc.

I.5 Motivație

Motivul pentru care am ales această temă este pentru a mă familiariza cu mecanismele care stau in spatele multor aplicații de virtualizare folosite în industrie și pentru a învăța un nou limbaj de programare. Am ales limbajul de programare rust datorită similarității acestuia cu limbajul C++ și datorită designului sau care pune accentul pe performanță și siguranță. [10]

Capitolul II

Tehnologii utilizate

II.1 Linux

II.1.1 namespace

Un *namespace* înfășoară o resursă globală de sistem printr-o abstractizare care face ca procesele din *namespace* să creadă că au propria lor instanță izolată a resursei globale. Modificările aduse resursei globale sunt vizibile altor procese care sunt membre ale *namespace*-ului, dar sunt invizibile pentru alte procese.[11]

În prezent, Linux implementează opt tipuri diferite de *namespace*-uri.

Mount namespace

Mount namespace-ul (CLONE_NEWNS, Linux 2.4.19) izolează setul de *mount point*-uri a sistemelor de fișiere văzute de un grup de procese. Astfel, procesele din diferite *mount namespace*-uri pot avea vederi diferite ale ierarhiei sistemului de fișiere. Odată cu adăugarea *mount namespace*-urilor, apelurile de sistem *mount()* și *umount()* au încetat să funcționeze pe un set global de *mount point*-uri vizibile pentru toate procesele din sistem și în schimb au efectuat operațiuni care au afectat doar *mount namespace*-ul asociat procesului de apelare. [12]

O utilizare a *mount namespace*-urilor este crearea de medii care sunt similare cu închisorile chroot. Cu toate acestea, spre deosebire apelul de sistem *chroot()*, *mount namespace*-urile sunt un instrument mai sigur și mai flexibil pentru această sarcină. Sunt posibile și alte utilizări mai sofisticate ale *mount namespace*-urilor. De exemplu, *mount namespace*-urile separate pot fi configurate într-o relație *master-slave*, astfel încât evenimentele de montare să fie propagate automat de la un *namespace* la altul; acest lucru permite, de exemplu, unui hard disk care este montat într-un *namespace* să apară automat în alte *namespace*-uri. [12]

Mount namespace-ul a fost primul tip de *namespace* care a fost implementat pe Linux, apărut în 2002. Acest fapt explică numele destul de generic "NEWNS" (scurt pentru "new namespace"). [12]

UTS namespace

UTS namespace-urile (CLONE_NEWUTS, Linux 2.6.19) izolează două identificatoare de sistem, *nodename* și *domainname*, returnate de apelul de sistem *uname()*; acestea putând fi setate folosind apelurile de sistem *sethostname()* și *setdomainname()*. În contextul containerelor, funcția *UTS namespace*-urilor permite fiecărui container să aibă propriul *hostname* și propriul *NIS domain name*. Acest lucru poate fi util pentru scripturile de inițializare și configurare care își adaptează acțiunile pe baza acestor nume. Termenul *UTS* derivă din numele structurii transmise apelului de sistem *uname()*: struct utsname. Numele acestei structuri provine, la rândul său, de la *UNIX Time-sharing System*. [12]

IPC namespace

IPC namespace-urile (CLONE_NEWIPC, Linux 2.6.19) izolează anumite resurse de comunicare între procese (*IPC*; *interprocess communication*), și anume, obiecte *IPC System V* și (de la Linux 2.6.30) cozi *POSIX* de mesaje. Caracteristica comună a acestor mecanisme IPC este că obiectele IPC sunt identificate prin alte mecanisme

decât *pathname*-urile sistemelor de fișiere. Fiecare *IPC namespace* are propriul set de identificatori *IPC System V* și propriul sistem de fișiere pentru cozile *POSIX* de mesaje. [12]

PID namespace

PID namespace-urile (CLONE_NEWPID, Linux 2.6.24) izolează spațiul numerelor de identificare ale proceselor. Cu alte cuvinte, procesele din *PID namespace*-uri diferite pot avea același PID. Unul dintre avantajele principale ale *PID namespace*-urilor este faptul că containerele pot fi migrate între gazde păstrând aceleași PID-uri pentru procesele din interiorul containerului. *PID namespace*-urile permit, de asemenea, fiecărui container să aibă propriul său *init* (PID 1), "strămoșul tuturor proceselor" care gestionează diverse activități de inițializare a sistemului și reface procesele copil orfane atunci când se termină. [12]

Din punctul de vedere al unei anumite instanțe de *PID namespace*, un proces are două PID-uri: PID-ul din *namespace* și PID-ul în afara *namespace*-ului, din sistemul gazdă. Un proces poate vedea (de exemplu, să vizualizeze prin */proc/PID* sau să trimită semnale cu *kill()*) doar procesele conținute în *PID namespace*-ul său și *namespace*-urile aflate sub acel *PID namespace*. [12]

Network namespace

Network namespace-urile (CLONE_NEWNET, începute în Linux 2.4.19 2.6.24 și completate în mare parte în aproximativ Linux 2.6.29) asigură izolarea resurselor de sistem asociate cu rețelelor. Astfel, fiecare *network namespace* are propriile dispozitive de rețea, adrese IP, tabele de rutare IP, director */proc/net* și numere de port. [12]

Network namespace-urile din rețea fac ca containerele să fie utile dintr-o perspectivă de rețea: fiecare container poate avea propriul dispozitiv de rețea (virtual) și propriile aplicații care se leagă la al porturile *namespace*-ului; reguli de rutare adecvate în sis-

temul gazdă pot direcționa pachetele de rețea către dispozitivul de rețea asociat unui anumit container. Astfel, de exemplu, este posibil să existe mai multe servere web containerizate pe același sistem gazdă, fiecare server fiind legat la portul 80 în *namespace*. [12]

User namespace

User namespace-urile (CLONE_NEWUSER, începute în Linux 2.6.23 și completate în Linux 3.8) izoleaza spațiile numerelor de identificare ale utilizatorului și grupului. Cu alte cuvinte, ID-urile de utilizator și de grup ale unui proces pot fi diferite în interiorul și în afara unui *user namespace*. Cel mai interesant caz este faptul că un proces poate avea un ID de utilizator normal neprivilejat în afara unui *user namespace*, având în același timp un ID de utilizator 0 în interiorul *namespace*-ului. Acest lucru înseamnă că procesul are privilegii root complete pentru operațiunile din *user namespace*, dar nu este privilegiat pentru operațiuni în afara *namespace*-ului. [12]

Cgroup namespace

Cgroup namespace-urile (CLONE_NEWCGROUP, Linux 4.6) izoleaza vizualizarea cgrupurilor unui proces, care pot fi accesate prin */proc/[pid]/cgroup* și */proc/[pid]/mountinfo*. [13]

Fiecare *namespace* are propriul set de directoare cgroup rădăcină. Aceste directoare rădăcină sunt punctele de bază pentru locațiile relative afișate în din fişierul */proc/[pid]/cgroup*. Când un proces creează un nou *cgroup namespace* folosind *clone(2)* sau *unshare(2)* cu *flag*-ul CLONE_NEWCGROUP, directoarele sale cgroup curente devin directoarele cgroup rădăcină ale noului *namespace*. [13]

Time namespace

Time namespace-urile izoleaza valorile a două ceasuri de sistem: CLOCK_MONOTONIC și CLOCK_BOOTTIME.[14]

Procesele dintr-un *time namespace* împărtășesc valorile pentru aceste ceasuri, lucru care afectează diferite API-uri care măsoară folosind aceste ceasuri, inclusiv: *clock_gettime (2)*, *clock_nanosleep(2)*, *nanosleep (2)*, *timer_settime (2)*, *timerfd_settime (2)* și */proc/uptime*. [14]

În prezent, singura modalitate de a crea un *time namespace* apelând *unshare (2)* cu *flag*-ul CLONE_NEWTIME. Acest apel creează un nou *time namespace*, dar nu plasează procesul de apelare în noul *namespace*. În schimb, copiii ulterior creați ai procesului de apelare sunt plasați în noul *namespace*. Aceasta permite compensări de ceas pentru ca noul *namespace* să fie setat înainte ca primul proces să fie plasat în *namespace*. Legătura simbolică */proc/[pid]/ns/time_for_children* arată *time namespace*-ul în care copiii procesului for fi creați. [14]

II.1.2 cgroup (Control group)

Grupur de control, denumit de obicei cgroup, este o caracteristică a *kernelului* Linux care permite organizarea ierarhică a proceselor în grupuri, și limitarea și monitorizarea diferitelor tipuri de resurse care pot fi utilizate de acestea. Interfața cgroup este furnizată prin intermediul unui pseudo-sistem de fișiere numit *cgroupfs*. Gruparea este implementată în codul nucleului, în timp ce monitorizarea resurselor și limitele sunt implementate într-un set de subsisteme pentru fiecare tip de resursă (memorie, procesor, etc.). [15]

Un subsistem este o componentă a *kernelului* care modifică comportamentul proceselor dintr-un cgroup. Au fost implementate diferite subsisteme, care fac posibilă realizarea unor lucruri precum limitarea timpului de procesor și a memoriei disponibile pentru un cgroup, contabilizarea timpului de procesor folosit de un cgroup și înghețarea și reluarea execuției unor procese dintr-un cgroup. Subsistemele sunt uneori cunoscute și sub denumirea de *controllere* de resurse. [15]

Cgrupurile pentru un *controller* sunt aranjate într-o ierarhie. Acestă ierarhie este definită prin crearea, eliminarea și redenumirea subdirectoarelor din sistemul de fișiere cgroup. La fiecare nivel al ierarhiei, atributele (de exemplu, limitele) pot fi definite. Controlul sau limitele oferite de cgrupuri au, în general, efect în întreaga sub-ierarhie de sub cgroupul în care atributele sunt definite. Astfel, de exemplu, limitele plasate pe a cgroup la un nivel superior în ierarhie nu pot fi depășite cu cgrupuri descendente. [15]

II.1.3 overlayFS

Sistemele de fișiere Union sunt o soluție care permite combinarea virtuală a mai multor directoare, păstrând conținutul lor real separat. Sistemul de fișiere Overlay (OverlayFS) este un exemplu dintre acestea, deși este mai mult un mecanism de montare decât un sistem de fișiere. [16]

Adăugat în kernelul Linux în versiunea 3.18, OverlayFS permite suprapunerea conținutului (atât fișierele, cât și directoarele) unui director peste altul. Directoarele sursă pot fi pe diferite volume și pot fi chiar sisteme de fișiere diferite, ceea ce creează un mecanism care permite modificarea temporară a fișierelor și directoarelor *read-only.* [16]

Un sistem de fișiere *overlayFS* combină două sisteme de fișiere - un sistem de fișiere "superior"(*upper*) și un sistem de fișiere "inferior"(*lower*). Când un nume există în ambele sisteme de fișiere, obiectul din sistemul de fișiere "superior" este vizibil în timp ce obiectul din sistemul de fișiere "inferior" este ascuns sau, în cazul directoarelor, combinat cu obiectul "superior". [17]

II.2 rust

Rust este un limbaj de programare multi-paradigmă axat pe performanță și siguranță, în special concurgență sigură. Rust este similar sintactic cu C++, dar asigură securitatea memoriei fără a folosi *garbage collection*. [18]

Securitatea memoriei

Rust este proiectat să fie *memory safe* și, prin urmare, nu permite pointeri nuli, *dangling pointeri* sau *data race*-uri în cod sigur. Pentru a reproduce funcția pointerilor din alte limbi, de a fi valizi sau NULL, cum ar fi în listele înlănțuite sau în arborii binari, biblioteca de bază Rust oferă un tip de dată opțiune, care poate fi utilizat pentru a testa dacă un pointer are o valoare (*Some*) sau nu (*None*). De asemenea, Rust introduce sintaxa suplimentară pentru a gestiona valabilitatea (*lifetime*), iar compilatorul verifică acest lucru prin intermediul unui *borrow checker*. Codul nesigur care poate subverti unele dintre aceste restricții poate fi scris folosind cuvântul cheie *unsafe*. [18]

Gestionarea memoriei

Rust nu folosește un sistem automat de *garbage collection* precum cele utilizate în Go, Java sau .NET Framework și nu utilizează *Automatic Reference Counting* folosit în limbaje precum Swift și Objective-C. În schimb, memoria și alte resurse sunt gestionate prin convenția "achiziția resurselor este inițializarea" (RAII; resource acquisition is initialization), cu *reference counting* opțional. [18]

Siguranța utilizării *ponterilor* este verificată la compilare de către *borrow checker*, prevenind *dangling pointerii* și alte forme de comportament nedefinit. [18]

Tipuri și polimorfism

Sistemul de tip acceptă un mecanism similar claselor, numit "trăsătură"(*trait*), inspirat direct de limbajul Haskell. Aceasta este o facilitate pentru polimorfismul ad-hoc, obținută prin adăugarea de constrângeri la declarațiile de tip variabil. [18]

Rust folosește inferența de tip, pentru variabilele declarate cu cuvântul cheie *let*. Astfel de variabile nu necesită o valoare alocată inițial pentru a determina tipul acestora. O eroare de compilare rezultă daca o ramură a codului nu reușește să atribuie o valoare variabilei. Variabilele alocate de mai multe ori, mutabile, trebuie marcate cu cuvântul cheie *mut*. [18]

Sistemul de obiecte din Rust se bazează în jurul implementărilor, trăsăturilor și tipurilor structurate. Implementările îndeplinesc un rol similar cu cel al claselor din alte limbi și sunt definite cu ajutorul cuvântului cheie *impl*. Moștenirea și polimorfismul sunt asigurate de trăsături; ele permit definirea metodelor și amestecarea lor în implementări. Tipurile structurate sunt utilizate pentru a defini câmpurile. Implementările și trăsăturile nu pot defini singure câmpurile și numai trăsăturile pot fi folosite pentru moștenire. Un beneficiu al acestor trasaturi este că se previne problema diamantului, ca în C++. [18]

Capitolul III

Arhitectura aplicației

Funcțiile de bază ale acestei aplicații sunt gestionarea containerelor și a imaginilor folosite pentru crearea acestora, și execuția proceselor în aceste medii izolate.

Toate fișierele aplicației sunt stocate în directorul *minato* aflat în directorul */var/lib/* al utilizatorului care rulează aplcația. Acest director conține containerele (*minato/containers*), imaginile (*minato/images*) si alte fișiere auxiliare.

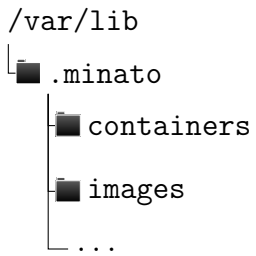


Figura III.1: Directorul principal al aplicației

Aplicația este compusă din trei componente principale: Managerul de containere, Managerul de imagini, Daemon/client.

III.1 Managerul de containere

Scopul managerului de containere este de a crea o interfață între utilizator și aplicație. Acesta poate efectua următoarele operații:

Operație	**Descrierea operației**
create	Crearea unui container
run	Rularea unui container
open	Deschiderea unui container
stop	Oprirea unui container
list	Afișarea tuturor containerelor create
delete	Ștergerea unui container

Tabela III.1: Operațiile efectuate de către managerul de containere

III.1.1 Crearea unui container (*create*)

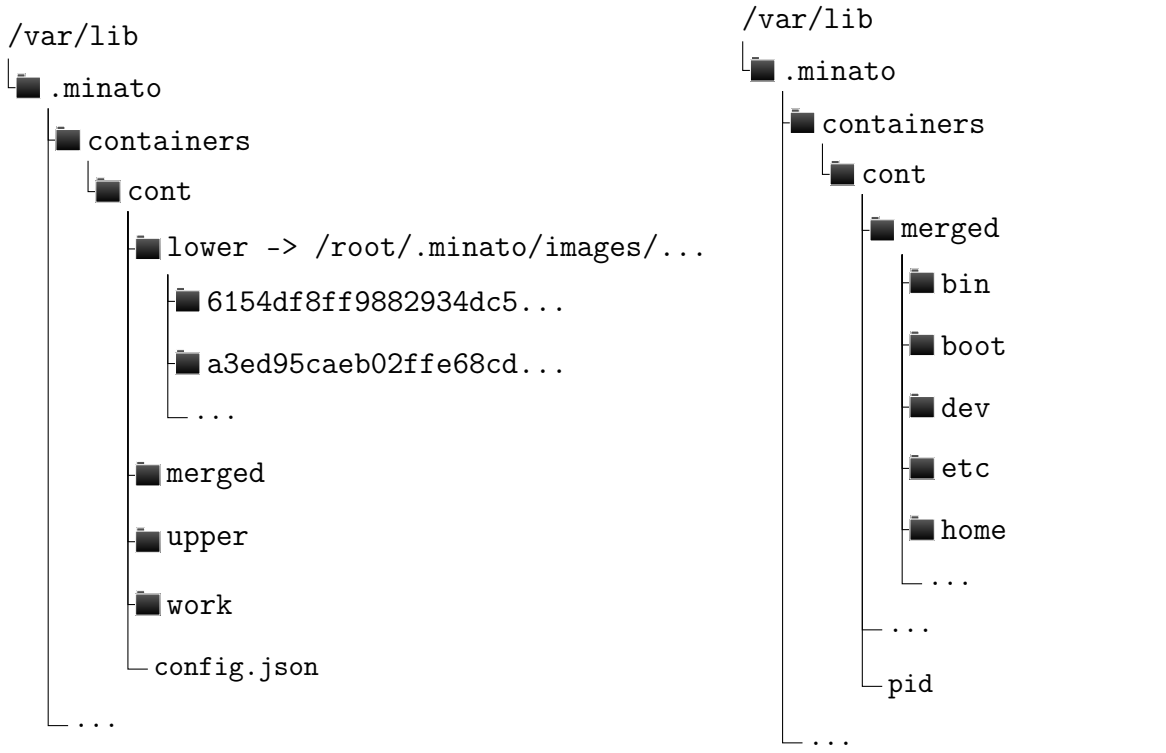
Această operație constă în crearea directorului în care se vor stoca fișierele containerului și se va monta sistemul de fișiere. Inițial acesta va conține directoarele necesare pentru operația de montare, *upper*, *lower*, *work* și *merged*, care vor fi goale, cu exceptia directorului *lower*, care va fi o legătură simbolică către directorul în care sunt stocate straturile imaginii care urmează să fiee utilizate de către container.

De asemenea, în directorul containerului va fi creat fișierul *config.json* în care se află toate configurările care trebuie aplicate asupra containerului. Printre acestea se regăsesc: maparea unu la unu dintre userul din sistemul de operare și userul din container, *namespaceurile* în care trebuie izolat containerul, hostname-ul containerului, etc.

```
1 pub fn create(&self) -> Result<(), Box<dyn std::error::Error>> {
2     info!("creating container");
3 
4     if Path::new(&self.path).exists() {
5         info!("container exists. skipping creation...");
```

```
6           return Ok(())
7       }
8       self.create_directory_structure()?;
9       self.generate_config_json()?;
10
11      info!("created container.");
12      Ok(())
13  }
```

Listarea III.1: Crearea unui container



(a) Directorul unui container

(b) Directorul unui container în timpul rulării

Figura III.2: Directorul unui container

III.1.2 Rularea unui container (*run*)

Operația de rulare reprezintă crearea propriu-zisă a spațiului izolat în sistemul de operare. Pentru realizarea acestei operații, se efectuează mai mulți pași.

```rust
fn run(&self, daemon: bool, volume: Option<String>) -> Result<(), Box
    ↪ <dyn std::error::Error>> {
    self.mount_container_filesystem()?;
    self.prepare_container_mountpoint()?;
    self.mount_volume(volume)?;
    self.prepare_container_directories()?;
    self.prepare_container_networking()?;
    self.mount_container_directories()?;
    self.prepare_container_id_maps()?;
    self.pivot_container_root()?;
    self.execute_inner_fork(daemon)?;

    Ok(())
}
```

Listarea III.2: Rularea unui container

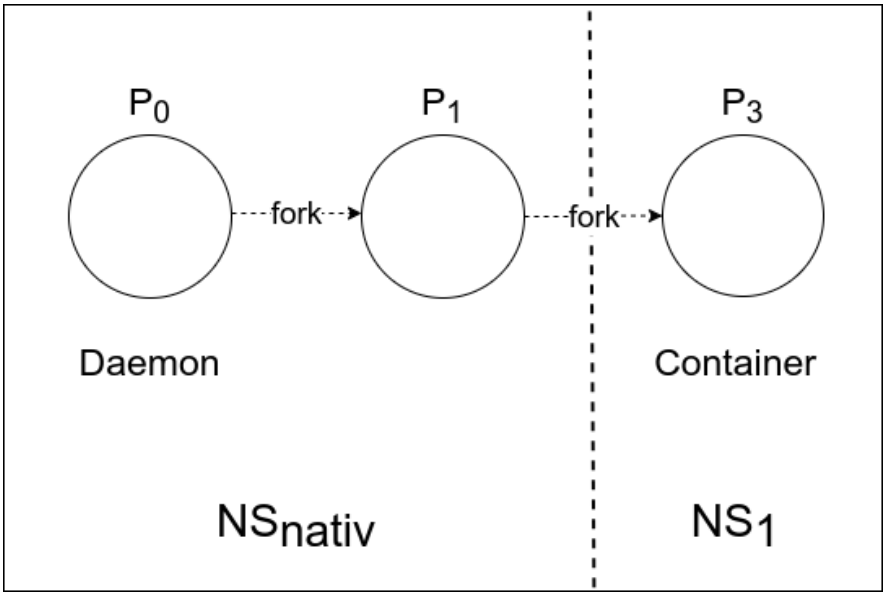


Figura III.3: Crearea procesului în noul namespace

Efectuarea primului *fork*, care are rolul de a separa procesul containerului de procesul *daemonului* atunci când programul trebuie sa ruleze pe fundal și de a îl păstra pe acesta în *namespaceul* inițial al sistemului de operare gazdă. De asemenea, este creat fișierul *pid* care este folosit pentru a verifica daca container-ul rulează deja și, în cazul afirmativ, care este PID-ul acestuia. Crearea fișierului este efectuată de către părintele din *fork*, resul pașilor de către copil.

Montarea rădăcinii containerului, cu sistemul de fișiere OverlayFS folosind comanda **mount -t overlay overlay -olowerdir=/lower,upperdir=/upper, workdir=/work /merged**. În urma acesteia, în directorul *merged* se va găsi structura de directoare specifică *kernelului* Linux.

```
fn mount_container_filesystem(&self) -> Result<(), Box<dyn std::error
    ↪ ::Error>> {
    info!("mounting container filesystem...");

    let container_path = Path::new(&self.path);
    let subdirectories = container_path.join("lower")
        .read_dir().unwrap()
        .map(|dir|
            format!("{}", dir.unwrap().path().display()))
        .collect::<Vec<String>>()
        .join(":");
    let lowerdir_arg  = format!("lowerdir={}",       subdirectories);
    let upperdir_arg  = format!("upperdir={}/upper", &self.path);
    let workdir_arg   = format!("workdir={}/work",   &self.path);
    let mergeddir_arg = format!("{}/merged",         &self.path);
    let full_arg = format!("{},{},{}",
        lowerdir_arg, upperdir_arg, workdir_arg
    );
```

```
18
19      mount(
20          Some("overlay"),
21          mergeddir_arg.as_str(),
22          Some("overlay"),
23          MsFlags::empty(),
24          Some(full_arg.as_str())
25      )?;
26
27      info!("mounted␣container␣filesystem.");
28      Ok(())
29  }
```

Listarea III.3: Montarea directorului în care este sistemul de fișiere al containerului

Apelul funcției *unshare* cu *flag*-urile specifice fiecarui *namespace*. În urma acestuia, toate procesele copil ale procesului curent vor fi create în *namespace*-uri separate.

Setarea sistemului de fișiere al mașinii gazdă ca privat. Acest pas are rolul de a izola toate evenimentele de montare sau de demontare care au loc în sistemul de operare gazdă.

Legarea executabilului folosit ca proces inițial (init). Acest pas constă în efectuarea unui *bind mount* între fișierul *tini*, aflat în directorul *.minato*, și un fișier nou creat în sistemul de fișiere al containerului. *tini* este un program open-source care are aceleași funcționalitați ca procesul *init* găsit în sistemele de operare Unix, de exemplu, protecția împotriva creării proceselor *zombie*.

Crearea directoarelor *dev*, *sys*, *proc*, *old_proc* și *put_old*. Directoarele *dev*, *sys* și *proc* conțin sisteme de fișiere specifice sistemelor de operare Unix, necesare pentru

funcționarea containerului. Directorul *old_proc* este folosit pentru montarea pseudo-sistemului de fișiere *proc*, iar directorul *put_old* este folosit în operația de schimbare a rădăcinii procesului curent, *pivot_root*.

Crearea legăturii de rețea dintre container și sistemul de operare gazdă folosind un dispozitive virtuale de *ethernet*(*veth*) și un *bridge*. În executarea acestei operații se folosește aplicația **iproute2** din kernelul Linux.

```
ip link add example-br0 type bridge
ip link set dev example-br0 up
ip link add example-veth0 type veth peer name example-veth1
ip addr add 192.168.1.10/24 dev example-veth0
ip link set example-veth0 up
ip link set dev example-veth1 master example-br0
ln /proc/67539/ns/net /var/run/netns/example-ns
ip link set example-veth1 netns example-ns
ip netns exec example-ns ip link set lo up
ip netns exec example-ns ip link set example-veth1 up
ip netns exec example-ns ip addr add 192.168.1.11/24 dev example-
    ↪ veth1
ip netns exec example-ns ip route add default via 192.168.1.11/24
```

Listarea III.4: Comenzile utilizate pentru crearea legăturii de rețea

Montarea directoarelor *dev* și *proc*, și a ierarhiei cgroup. Directorul *dev* va fi legat cu cel din sistemul gazdă pentru a avea acces la dispozitivele acesteia, iar directorul *proc* va fi de asemenea legat cu cel din sistemul gazdă, dar doar temporar. Pentru crearea ierarhiei cgroup este nevoie doar de crearea și montarea directoarelor principale (*freezer*, *memory*, *blkio*, etc.) ale acesteia. Odata montate, acestea sunt populate automat de către *kernel*.

```rust
fn mount_container_directories(&self) -> Result<(), Box<dyn std::
    ↪ error::Error>> {
    info!("mounting␣container␣directories...");

    info!("mounting␣proc␣to␣old_proc...");
    mount(
        Some("/proc"),
        "old_proc",
        None::<&str>,
        MsFlags::MS_BIND | MsFlags::MS_REC,
        None::<&str>,
    )?;

    self.mount_container_cgroup_hierarchy()?;

    info!("mounting␣dev␣to␣dev...");
    mount(
        Some("/dev"),
        "dev",
        None::<&str>,
        MsFlags::MS_BIND | MsFlags::MS_REC,
        None::<&str>,
    )?;

    info!("mounted␣container␣directories.");
    Ok(())
}

```

```rust
fn mount_container_cgroup_hierarchy(&self) -> Result<(), Box<dyn std
    ↪ ::error::Error>> {
    info!("mounting␣container␣cgroup␣hierarchy...");

    let directories = vec![
        "freezer", "hugetlb", "memory", "blkio", "cpuset",
        "cpu,cpuacct", "devices", "pids",
        "net_cls,net_prio", "perf_event", "rdma"
    ];

    for dir in directories {
        let dir_path = format!("sys/fs/cgroup/{}", dir);
        if Path::new(&dir_path).exists() {
            fs::remove_dir_all(&dir_path)?;
        }
        fs::create_dir_all(&dir_path)?;

        info!("mounting␣cgroup␣{}...", dir);
        let cgroup_version = "cgroup";
        mount(
            Some(cgroup_version),
            dir_path.as_str(),
            Some(cgroup_version),
            MsFlags::MS_NOSUID | MsFlags::MS_NODEV | MsFlags::
    ↪ MS_NOEXEC,
            Some(dir),
        )?;
    }
```

```
54
55      info!("mounted container cgroup hierarchy.");
56      Ok(())
57  }
```

Listarea III.5: Montarea directoarelor *dev* și *proc*, și a ierarhiei cgroup

Schimbarea id-urilor utilizatorului pentru noul *user namespace*, folosind fișierele */proc/self/uid_ map* și */proc/self/gid_ map*. Acestea permit crearea unei legături unu la unu între id-urile din sistemul de operare gazdă si id-urile din container. După crearea unui *user namespace* nou, fișierul *uid_ map* al containerului poate fi scris o singură dată pentru a defini maparea ID-urilor utilizatorului în noul spațiu de utilizator. Încercarea de a scrie de mai multe ori într-un fișier *uid_ map* dintr-un *user namespace* eșuează cu eroarea EPERM. Reguli similare se aplică pentru fișierul *gid_ map*. [19]

Schimbarea rădăcinii sistemului de fișiere din container folosind operația *pivot_ root*. Directorul *put_ old* este un director aflat în rădacina containerului, necesar pentru această operația. Odata folosit, el este demontat si șters.

```
1  fn pivot_container_root(&self) -> Result<(), Box<dyn std::error::
       ↪ Error>> {
2       info!("pivoting container root..");
3
4       info!("pivoting root...");
5       pivot_root(".", "put_old")?;
6
7       info!("unmounting pivot auxiliary folder...");
8       umount2("/put_old", MntFlags::MNT_DETACH)?;
9       if Path::new("/put_old").exists() {
10          info!("removing auxiliary folder...");
11          std::fs::remove_dir_all("/put_old")?;
```

```
12      }
13
14      info!("pivoted␣container␣root.");
15      Ok(())
16      }
```

Listarea III.6: Schimbarea rădăcinii sistemului de fișiere

Efectuarea *fork*-ului final pentru aplicarea efectelor operației *unshare*. Acesta va crea un nou proces copil, în care se va executa primul proces al containerului folosind comanda *execve*.

Curățarea spațiului de lucru urmată de sfârșirea procesului început de către container. Aceasta constă în ștergerea dispozitivelor create pentru stabilirea legăturii de rețea dintre container și sistem de operare, și în demontarea sistemului de fișiere *overlay*.

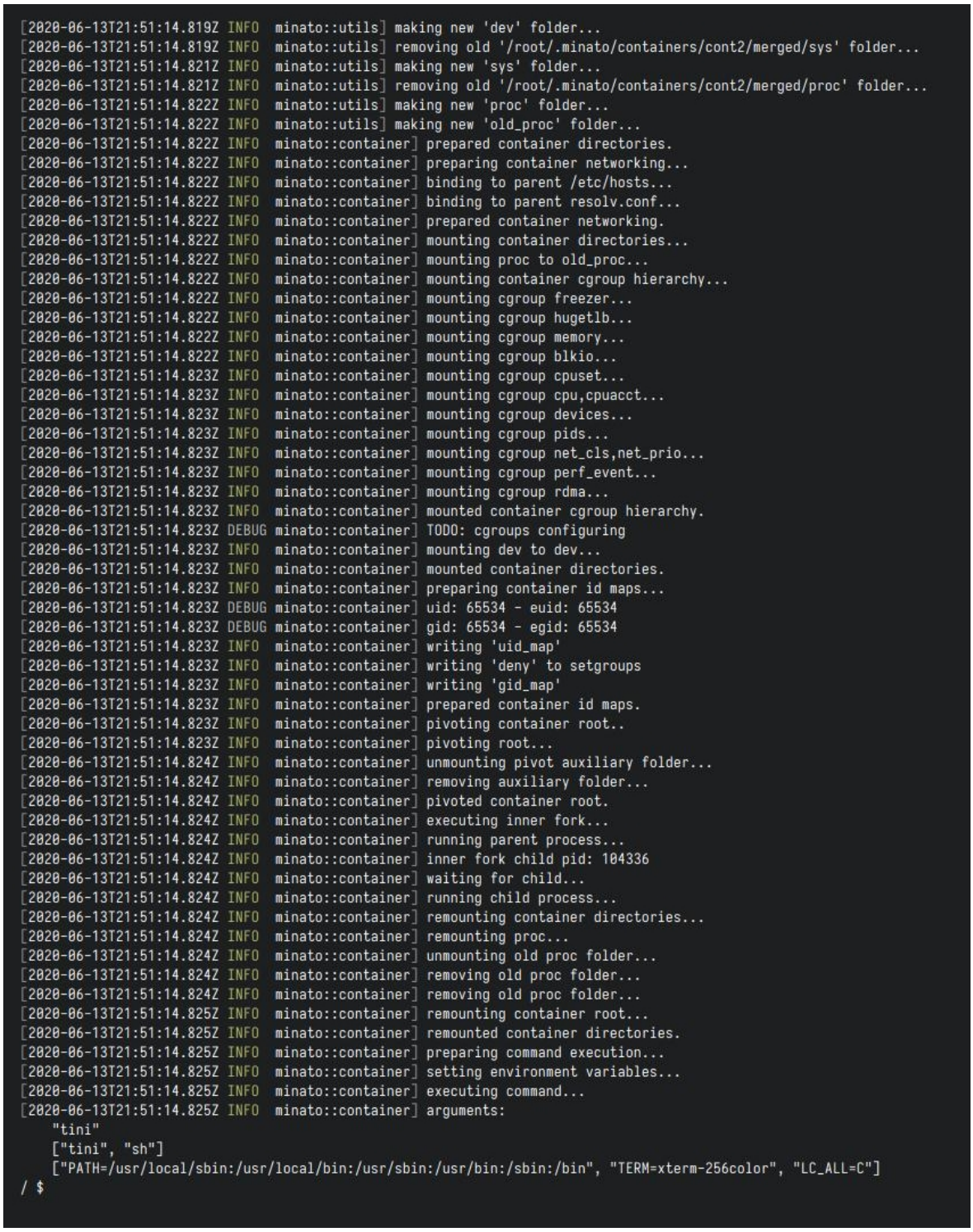


Figura III.4: Rularea unui container cu comanda /bin/sh

III.1.3 Deschiderea unui container (*open*)

Operația de deschidere a unui container se realizează folosind apelul de system *setns* împreuna cu *flag*-urile specifice fiecarui *namespace*, urmat de comanda *execve* cu

interpretorul *sh*.

```
pub fn open(&self, container_name: &str) -> Result<(), Box<dyn std::
    ↪ error::Error>> {
     info!("opening container...");

     let container_pid = match utils::get_container_pid_with_str(
    ↪ container_name).unwrap() {
         None => {
             info!("container isn't running or doesn't exist. exiting
    ↪ ...");
             return Ok(());
         },
         Some(pid) => pid
     };
     info!("container pid: {}", container_pid);
     let mut namespaces = HashMap::new();
     namespaces.insert(CloneFlags::CLONE_NEWIPC, "ipc");
     namespaces.insert(CloneFlags::CLONE_NEWUTS, "uts");
     namespaces.insert(CloneFlags::CLONE_NEWNET, "net");
     namespaces.insert(CloneFlags::CLONE_NEWPID, "pid");
     namespaces.insert(CloneFlags::CLONE_NEWNS, "mnt");
     namespaces.insert(CloneFlags::CLONE_NEWCGROUP, "cgroup");
     namespaces.insert(CloneFlags::CLONE_NEWUSER, "user");
     let pid_path = format!("/proc/{}/ns", container_pid);
     info!("setting namespaces...");
     for namespace in namespaces {
         let ns_path = format!("{}/{}", pid_path, namespace.1);
         self.set_namespace(ns_path.as_str(), namespace.0)?;
```

```
25      }
26      let result = match fork() {
27          Ok(ForkResult::Parent { child, .. }) => {
28              waitpid(child, None)?;
29              Ok(())
30          }
31          Ok(ForkResult::Child) => {
32              self.do_exec("/bin/sh")
33          }
34          Err(e) => {
35              info!("fork␣failed:␣{}", e);
36              Ok(())
37          }
38      };
39      info!("opened␣container.");
40      result
41  }
```

Listarea III.7: Deschiderea unui container

III.1.4 Oprirea unui container (*stop*)

Operația de oprire a unui container se realizează folosind apelul de system *kill* împreuna cu *flag*-ul *SIGTERM* și id-ul PID al container-ului.

III.1.5 Afișarea tuturor containerelor create (*list*)

Operația de afișare a containerelor se realizează prin citirea tuturor directoarelor aflate în directoarele *containers* și *images*. Pentru fiecare container găsit se afișeaza numele acestuia, imaginea folosită pentru crearea lui, locația unde este stocat și PID-ul lui dacă acesta rulează.

III.1.6 Ștergerea unui container (*delete*)

Operația de ștergere a unui container se realizează prin ștergerea directorului în care acesta este stocat.

III.2 Managerul de imagini

Asemănător cu managerul de containere, scopul managerului de imagini este de a crea o interfață între utilizator și aplicație. Acesta poate efectua următoarele operații:

Operație	**Descrierea operației**
pull	Downloadarea și stocarea unei imagini
list	Afișarea tuturor imaginilor downloadate
delete	Ștergerea unei imagini

Tabela III.2: Operațiile efectuate de către managerul de containere

III.2.1 Downloadarea și stocarea unei imagini (*pull*)

Downloadarea imaginilor se face din *repository*-ul Docker, Docker Hub, prin API-ul pus la dispoziție de către aceștia. Acest proces constă în autentificarea cu API-ul și downloadarea fișierului *json* asociat imaginii. Din acest fișier sunt extrase date pentru descărcarea straturilor imaginii, care apoi sunt descărcate și dezarhivate în directorul imaginii.

III.2.2 Afișarea tuturor imaginilor downloadate (*list*)

Operația de afișare a imaginilor se realizează prin citirea tuturor directoarelor aflate în directorul *images*. Pentru fiecare imagine găsită se afișeaza numele acesteia locația unde este stocată.

III.2.3 Ștergerea unei imagini (*delete*)

Operația de ștergere a unei imagini se realizează prin ștergerea directorului în care aceasta este stocată.

III.3 Daemon/client

Componenta daemon/client permite rularea unui proces pe fundal care se ocupă cu gestionarea containerelor și a imaginilor, și comunicarea cu acesta.

Beneficiul principal al acestei componente este faptul că aceasta pune la dispoziție accesul la operațiile containerului și utilizatorilor neprivilegiați.

Comunicarea dintre client și daemon are loc prin *socketuri* Unix, daemonul așteptând constant să primească instrucțiuni de la clienți pe care apoi să le execute. Clienții pot trimite operații specifice programului, printre care și cea de oprire a daemon-ului.

Capitolul IV

Utilizarea aplicației

În acest capitol voi prezenta un exemplu de utilizare a aplicației pentru rularea unui server HTTP Apache (*httpd*) și comenzile necesare pentru realizarea acestui lucru.

Pornirea daemonului

Pentru pornirea daemonului se execută comanda **sudo minato -d**, care va porni programul în modul daemon și va începe să aștepte comenzi de la clienți.

```
[2020-06-15T22:09:46.604Z INFO  minato] running in daemon mode
[2020-06-15T22:09:46.605Z INFO  minato] running as daemon
[2020-06-15T22:09:46.605Z INFO  minato::daemon] creating daemon...
[2020-06-15T22:09:46.605Z INFO  minato::daemon] creating socket...
[2020-06-15T22:09:46.605Z INFO  minato::daemon] socket file already exists. removing...
[2020-06-15T22:09:46.607Z INFO  minato::daemon] listener local address: "/var/lib/minato/socket" (pathname)
[2020-06-15T22:09:46.607Z INFO  minato::daemon] created socket.
[2020-06-15T22:09:46.607Z INFO  minato::daemon] created daemon.
[2020-06-15T22:09:46.607Z INFO  minato::daemon] starting daemon...
[2020-06-15T22:09:46.607Z INFO  minato::daemon] removing pid file...
[2020-06-15T22:09:46.608Z INFO  minato::daemon] waiting for client...
```

Figura IV.1: Pornirea programului in modul daemon

Downloadarea unei imagini

Pentru downloadarea unei imagini se execută comanda **sudo minato -d image pull -i "alpine:latest"**. Aceasta va comunica cu daemonul și îi va trimite comanda pentru downloadarea unei imaginii a distributiei Alpine Linux. *alpine* este id-ul folosit de imagine în Docker Hub, iar *latest* este tag-ul acesteia, versiunea.

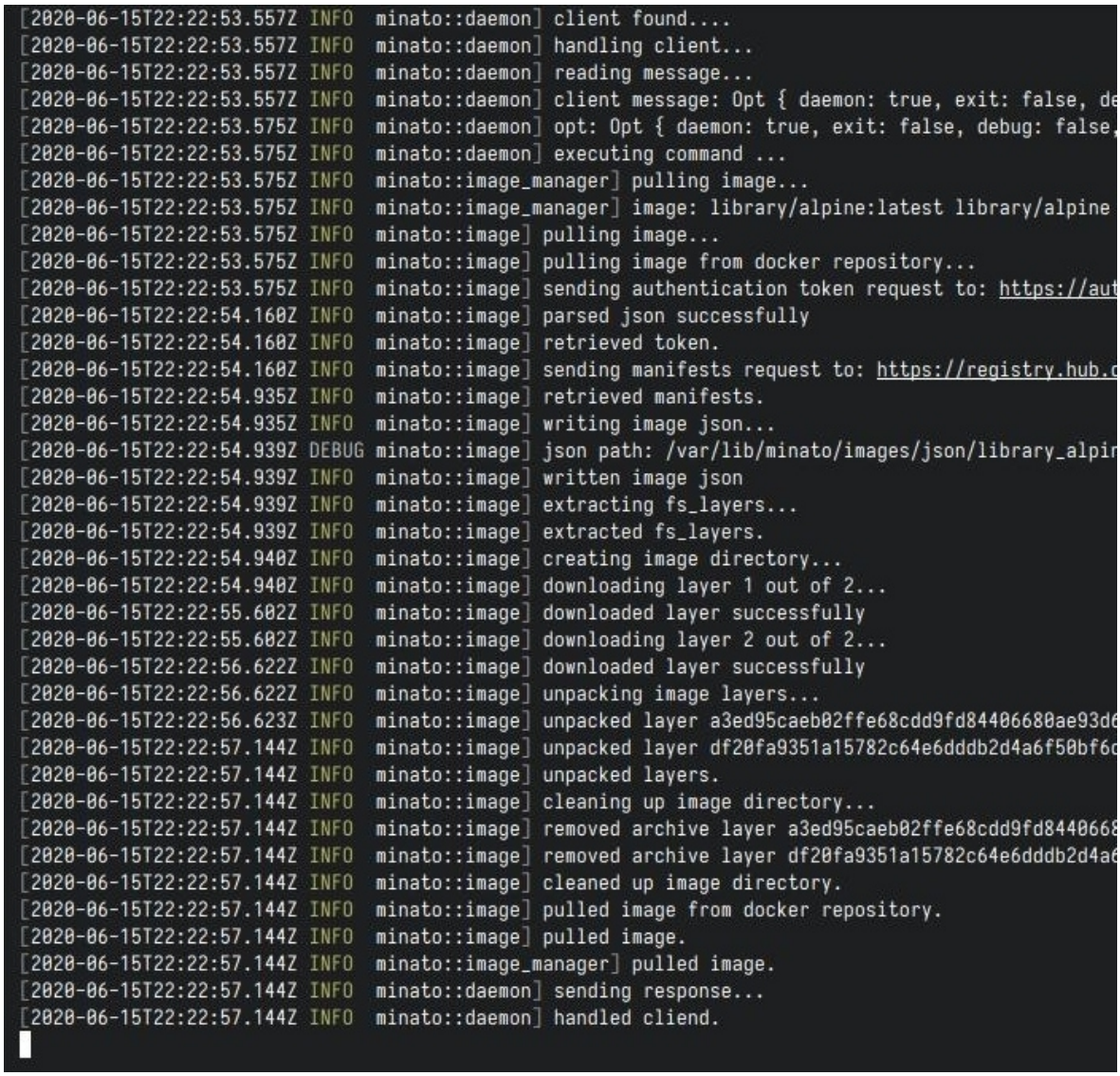


Figura IV.2: Downloadarea unei imagini

Crearea unui container

Pentru crearea unui container se execută comanda **sudo minato -d container create -c "example" -i "alpine:latest"**. Aceasta va comunica cu daemonul și îi va trimite comanda pentru crearea unui container cu numele "example", folosind ca imagine distribuția Alpine Linux.

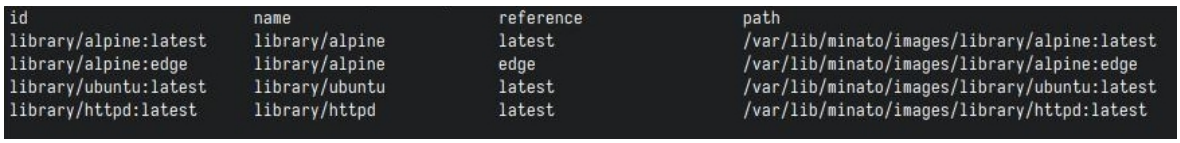
Listarea containerelor și a imaginilor

Pentru listarea containerelor și a imaginilor se execută comanda **sudo minato container list**, respectiv **sudo minato image list**. În lista de containere sunt afișate

numele, imaginea, locația și PID-ul (dacă acesta rulează) fiecărui container, iar in lista de imagini sunt afișate numele, versiunea și locația fiecărui container.

pid	id	image	path
-	test	library/alpine:latest	/var/lib/minato/containers/test
-	httpd_c	library/httpd:latest	/var/lib/minato/containers/httpd_c
-	cont3	library/alpine:latest	/var/lib/minato/containers/cont3
-	example	library/alpine:latest	/var/lib/minato/containers/example

Figura IV.3: Listarea containerelor



id	name	reference	path
library/alpine:latest	library/alpine	latest	/var/lib/minato/images/library/alpine:latest
library/alpine:edge	library/alpine	edge	/var/lib/minato/images/library/alpine:edge
library/ubuntu:latest	library/ubuntu	latest	/var/lib/minato/images/library/ubuntu:latest
library/httpd:latest	library/httpd	latest	/var/lib/minato/images/library/httpd:latest

Figura IV.4: Listarea imaginilor

Rularea containerului

Pentru rularea unui container se execută comanda **sudo minato -d container run -c example -v "/test:var/www/html"**. Aceasta va comunica cu daemonul și îi va trimite comanda pentru rularea containerului cu numele "example". De asemenea, se va crea o legătură (*bind mount*) între directorul */test* din sistemul de operare gazdă și directorul */var/www/html* din container.

Odata ce primește comanda, daemonul va crea un proces nou, izolat față de restul sistemului de operare, apoi se va reîntoarce la starea de așteptare.

Pașii executați pentru rularea unui container sunt descriși în secțiunea III.1.2 și în figura III.4.

Deschiderea containerului

Pentru deschiderea unui container se execută comanda **sudo minato container open -c example** într-un terminal separat. Programul va căuta apoi PID-ul containerului cu numele "example" și va crea un proces nou, un shell, în *namespaceurile* acestuia.

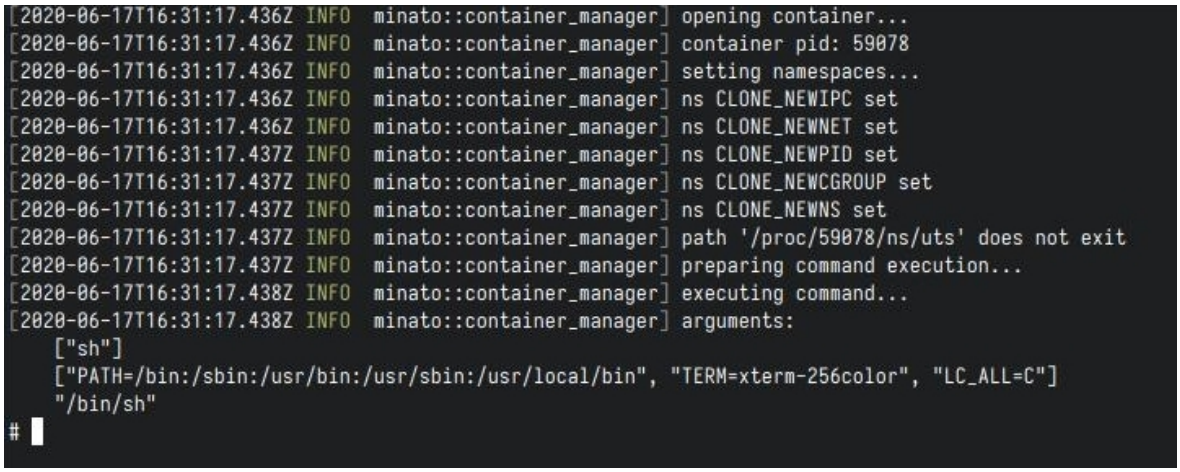


Figura IV.5: Deschiderea unui container

Instalarea și pornirea aplicației de server

Odată deschis containerul, se instalează pachetele *apache2*, aplicația pentru server, si *openrc*, aplicația pentru servicii, folosind comanda **apt install apache2 openrc**. Pornirea serverului se face prin comanda **rc-service apache2 start**.

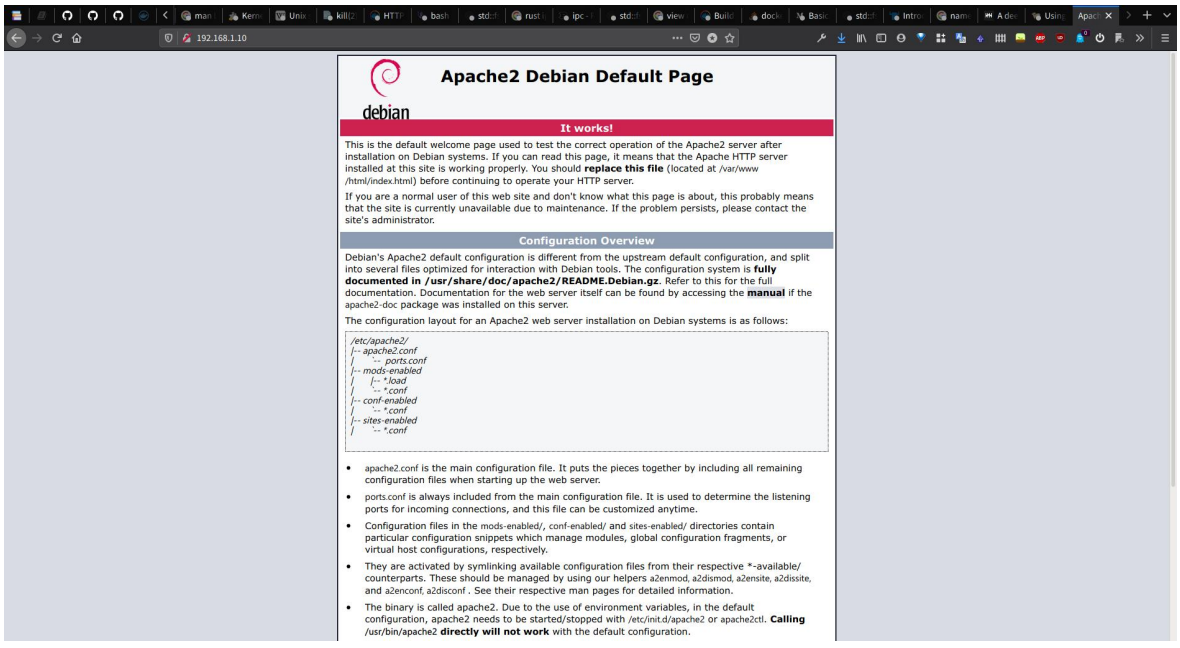


Figura IV.6: Site-ul de start al serviciului httpd

Fișierele site-ului hostat de server sunt stocate în */var/www/html* în container, dar sunt accesibile și din directorul */test* din sistemul de operare. De asemenea, site-ul poate fi vizualizat din sistemul de operare prin IP-ul **192.168.1.10**.

Această configurație permite rularea unui server sau a mai multor servere locale, fiecare asociat unui site și unui IP, și izolat față de celelalte.

Ștergerea containerului și a imaginii

Odată ce containerul și imaginea nu mai sunt necesare, acestea pot fi șterse prin execuția comenzilor **sudo minato -d container delete -c example** și **sudo minato -d image delete -i "alpine:latest"**.

Oprirea daemonului

Daemonul poate fi oprit prin execuția comenzii **sudo minato -e**.

Capitolul V

Concluzii

Bibliografie

[1] Wikipedia contributors. Os-level virtualization — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=OS-level_virtualization&oldid=958577453, 2020. [Online; accessed 7-June-2020].

[2] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[3] Poul-Henning Kamp and Robert NM Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, volume 43, page 116, 2000.

[4] chroot - freebsd manual pages. http://man.freebsd.org/chroot/2. [Online; accessed 7-June-2020].

[5] David Chisnall. Dragonfly bsd: Unix for clusters? https://www.informit.com/articles/printerfriendly/766375, 2020. [Online; accessed 2019-03-06].

[6] Wikipedia contributors. Freebsd jail — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=FreeBSD_jail&oldid=945967483, 2020. [Online; accessed 7-June-2020].

[7] Wikipedia contributors. Lxc — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=LXC&oldid=958749686, 2020. [Online; accessed 11-June-2020].

[8] Docker 0.9: introducing execution drivers and libcontainer - Docker Blog. https://www.docker.com/blog/

docker-0-9-introducing-execution-drivers-and-libcontainer/. [Online; accessed 2018-05-09].

[9] Wikipedia contributors. Docker (software) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Docker_(software)&oldid=960398220, 2020. [Online; accessed 7-June-2020].

[10] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–34, 2017.

[11] namespaces(7) - linux programmer's manual. https://man7.org/linux/man-pages/man7/namespaces.7.html. [Online; accessed 11-Jun-2020].

[12] Namespaces in operation, part 1: namespaces overview. https://lwn.net/Articles/531114/. [Online; accessed 11-Jun-2020].

[13] cgroup_namespaces(7) - linux programmer's manual. https://man7.org/linux/man-pages/man7/cgroup_namespaces.7.html. [Online; accessed 11-Jun-2020].

[14] time_namespaces(7) - linux programmer's manual. https://man7.org/linux/man-pages/man7/time_namespaces.7.html. [Online; accessed 11-Jun-2020].

[15] cgroups(7) - linux programmer's manual. https://man7.org/linux/man-pages/man7/cgroups.7.html. [Online; accessed 12-Jun-2020].

[16] Thom Denholm. Explaining overlayfs – what it does and how it works. https://www.datalight.com/blog/2016/01/27/explaining-overlayfs-%E2%80%93-what-it-does-and-how-it-works/. [Online; accessed 12-Jun-2020].

[17] Neil Brown. Overlayfs - the linux kernel archives. https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt. [Online; accessed 12-Jun-2020].

[18] Wikipedia contributors. Rust (programming language) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Rust_

(programming_language)&oldid=961355473, 2020. [Online; accessed 12-June-2020].

[19] user_namespaces(7) - linux programmer's manual. https://man7.org/linux/man-pages/man7/user_namespaces.7.html. [Online; accessed 11-Jun-2020].

Anexe

Anexa A

Capitol anexă

A.1 Secțiune anexă

Conținut anexă