Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Dejan Benedik

Analiza in uporaba vsebnikov Docker v arhitekturi mikrostoritev

diplomsko delo

univerzitetni študijski program
prve stopnje
računalništvo in informatika

Mentor: prof. dr. Matjaž Branko Jurič

Ljubljana, 2016

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Proučite koncepte virtualizacije, koncepte vsebnikov, vsebniških tehnologij ter mehanizmov v jedru operacijskega sistema Linux. Podrobno analizirajte vsebniško tehnologijo Docker ter jo primerjajte z ostalimi pristopi. Opišite vpliv vsebnikov na arhitekturo programske opreme. Podrobno analizirajte arhitekturo mikrostoritev ter identificirajte relacijo napram vsebniškim tehnologijam. Izdelajte lastno orodje za izvajanje mikrostoritev v oblaku z uporabo vsebnikov Docker ter ovrednotite njegovo primernost.

Zahvalil bi se prof. dr. Matjažu Branku Juriču za mentorstvo in strokovno pomoč pri izdelavi te diplomske naloge.

Iskrena hvala celotni družini za vso podporo med študijem. Hvala tudi Nejcu za pomoč pri lektoriranju.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
REST	Representational State Transfer	predstavitveni prenos stanj
LXC	Linux Containers	vsebniki Linux
CLI	command-line interface	ukazna vrstica
OS	Operating System	operacijski sistem
***NIX**	Unix-like	podoben sistemu Unix
IaaS	Infrastructure as a Service	infrastruktura kot storitev
PaaS	Platform as a Service	računalniška platforma kot storitev
SaaS	Software as a Service	programska oprema kot storitev
CaaS	Container as a Service	vsebnik kot storitev
CPE	Central Processing Unit	centralna procesna enota
SQL	Structured Query Language	strukturirani jezik za poizvedovanje
OCF	Open Container Format	odprti format vsebnikov
SSO	Single Sign On	enkratna prijava
XML	Extensible Markup Language	razširljivi označevalni jezik
SOAP	Simple Object Access Protocol	enostaven protokol za dostop do objektov

Povzetek

Naslov: Analiza in uporaba vsebnikov Docker v arhitekturi mikrostoritev

Avtor: Dejan Benedik

Za namene izvajanja spletnih aplikacij in storitev so vedno bolj priljubljene vsebniške tehnologije [9]. V tem diplomskem delu analiziramo najbolj razširjeno vsebniško tehnologijo Docker ter jo primerjamo z ostalimi vsebniškimi tehnologijami, ki delujejo na sistemih z jedrom Linux. Zaradi hitrega zagona in manjših obratovalnih stroškov so vsebniške tehnologije omogočile nov način zasnove zmogljivih spletnih aplikacij. Gre za arhitekturo mikrostoritev. Primerjamo jo z alternativami, pregledamo pomembne vidike izvajanja takih aplikacij, posebej pa pregledamo, kakšno vlogo imajo pri izvajanju vsebniške tehnologije. V praktičnem delu razvijemo spletno storitev, ki koristi vsebnike Docker ter omogoča osnovno izvajanje mikrostoritev ter horizontalno skaliranje komponent.

Ključne besede: vsebniki, virtualizacija, mikrostoritve, golang, oblak.

Abstract

Title: Docker container analysis and application in microservice architecture

Author: Dejan Benedik

Using container technology to run web applications and services is becoming increasingly common [9]. In this work we analyse the most popular container technology, Docker, and compare it with other Linux-based container technologies. Due to near-instant startup and minimal overhead, containers are facilitating a new way of designing software applications that is called microservice architecture. We analyse the approach and compare it with its alternatives. We examine some aspects of running such applications with container technology. Finally, we present the design and implementation of a Docker-based web service that allows us to run and scale microservices in a basic way.

Keywords: containers, virtualisation, microservices, golang, cloud.

Poglavje 1

Uvod

Med ključnimi faktorji za dobro uporabniško izkušnjo pri uporabi spletnih aplikacij in storitev je njihova hitrost oziroma odzivnost. To najlažje zagotovimo, če strežniki, ki izvajajo storitve, niso preveč obremenjeni. Ko število uporabnikov aplikacije narašča, lahko problem preobremenjenosti zalednega strežnika rešimo s skaliranjem, bodisi vertikalno (aplikacijo namestimo na bolj zmogljiv strežnik) bodisi horizontalno (obstoječemu strežniku dodamo novega, ki bo poganjal isto aplikacijo). Slednji način skaliranja je težko izvedljiv, če je aplikacija zasnovana kot monolit, zato se kot alternativa uveljavlja arhitektura mikrostoritev. V splošnem to pomeni, da aplikacijo razdelimo na manjše samostojne dele, ki so bolj primerni za sočasno izvajanje in skaliranje. Kot vsaka rešitev ima tudi arhitektura mikrostoritev nekaj pomanjkljivosti, npr. zapletenost orkestracije in težavno zagotavljanje enakega stanja v vseh instancah izvajane aplikacije. Vidiki arhitekture mikrostoritev so opisani v nadaljevanju.

Kljub velikemu številu uporabnikov interneta zmogljivosti spletnih strežnikov pogosto niso popolnoma izkoriščene. Z virtualizacijo računalniških virov lahko na istem strežniku vzpostavimo več izvajalnih okolij, izvajamo več aplikacij ter tako bolje izkoristimo strežniške vire. Virtualizacija v nekaterih izvedbah povzroča razmeroma visoke dodatne stroške. Najmanj dodatnih računalniških virov za virtualizacijo porabijo vsebniške tehnologije

(ang. *containerisation*), imenovane tudi lahka virtualizacija. Gre za izvajanje programa, ki ga operacijski sistem z vgrajenimi mehanizmi izolira od ostalih računalniških procesov in virov.

V tem diplomskem delu so opisani nekateri pristopi k virtualizaciji, podrobneje je analiziran primer orodja za lahko virtualizacijo Docker. Nato sledi primerjava orodja Docker z drugimi vsebniškimi tehnologijami. Definiramo arhitekturo mikrostoritev ter obravnavamo vidike izvajanja mikrostoritev z vsebniki. Na koncu razvijemo orodje, ki koristi vsebniške tehnologije in omogoča osnovno izvajanje mikrostoritev.

Poglavje 2

Virtualizacija in osnove vsebniških tehnologij

V poglavju predstavimo virtualizacijo in mehanizme, ki omogočajo vsebniške tehnologije v operacijskih sistemih z jedrom Linux. Najprej razložimo nekaj pojmov operacijskega sistema UNIX [31], ki veljajo tudi za GNU/Linux in so bistveni za razumevanje tematik, obravnavanih v nadaljevanju.

2.1 Osnovni pojmi

2.1.1 Proces

Proces je računalniški program v izvajanju. V pomnilniku je ponavadi sestavljen iz dveh segmentov, izvajalne kode in podatkov. Poleg tega v sodobnih operacijskih sistemih procesu pripadajo še deskriptorji računalniških virov (npr. datoteke, omrežne povezave), varnostni atributi (dovoljeni sistemski klici), podatki o lastniku procesa in drugo.

Pomembna pojma sta tudi starševski proces (ang. *parent process*), ki ustvari vsaj en otroški oziroma podrejen proces (ang. *child process*). Starševski proces mora med drugim obravnavati nekatere signale svojih podrejenih procesov, npr. ob končanju procesa ta svojemu staršu pošlje signal SIGCHLD. Če starševski proces konča z izvajanjem pred svojimi podrejenimi

procesi, ti postanejo sirote (ang. *orphaned process*). *Sirote* posvoji ter jim nudi vlogo starševskega procesa (predvsem obravnavanje signalov) poseben proces, imenovan *init*. To je prvi proces, ki se ustvari ob zagonu operacijskega sistema in je *prednik* vseh drugih izvajanih procesov. Od zagona do zaustavitve sistema se izvaja kot demon, torej proces, ki se izvaja v ozadju in ga uporabnik neposredno ne upravlja. Ponavadi je demon (posvojen) otrok procesa *init*. Proces *init* samodejno posvoji vse procese, ki postanejo sirote, v nekaterih izvedbah pa z mnogo mehanizmi nadzira delovanje operacijskega sistema. Primer programa init je systemd [16].

2.1.2 Korenska mapa

Korenska mapa ali korenski datotečni imenik (ang. *root filesystem*) je mapa na vrhu datotečnega sistema. Vsi ostali imeniki v sistemu so hierarhično pod korensko mapo in so bodisi neposredno dostopni bodisi priklopljeni nekje v datotečni hierarhiji.

2.1.3 Priklop

Mount je mehanizem, ki datoteko ali datotečni sistem priklopi na želeno mesto v korenskem datotečnem imeniku. Po priklopu nov datotečni imenik prekrije starega, vsebina starega datotečnega imenika pa ni dosegljiva do odklopa (ang. *unmount*). Datoteke znotraj priklopljenega datotečnega sistema so na razpolago uporabniku skladno z njegovimi pravicami branja in pisanja.

2.2 Virtualizacija

Virtualizacija je postopek, ki strojni ali programski vir preslika v navidezni vir, tega potem odjemalec koristi z enakimi mehanizmi, kot bi koristil pravi vir. Virtualiziran vir je lahko strojna oprema, jedro operacijskega sistema, omrežna povezava, datotečni sistem, datoteka in podobno. Gostiteljskemu sistemu virtualizacija v nekaterih izvedbah omogoča spremljanje in omejeva-

nje porabe sistemskih virov. Pomembna prednost uporabe navideznih virov je prenosljivost. Z virtualizacijo namreč lahko zagotovimo enak navidezni vir oziroma enake pogoje za izvajanje programa na različni strojni opremi, zato lahko v nekaterih primerih navidezne sisteme preprosto prenašamo med različnimi izvajalnimi okolji.

Pojem gostitelj (ang. *host*) v kontekstu virtualizacije pomeni sistem, ki nudi virtualizirano izvajalno okolje, pojem gost (ang. *guest*) pa nakazuje izvajan program, aplikacijo ali operacijski sistem, ki koristi to virtualizirano okolje.

Različne izvedbe virtualizacije povzročijo različno veliko dodatno porabo računalniških virov. Ker je namen virtualizacije izpostavljanje navideznih virov za odjemalca, je smiselno, da so dodatni stroški izvajanja čim manjši. Tako odjemalcu za izvajanje ostane večji delež računalniških virov, hkrati je manjša tudi poraba virov celotnega sistema. Naštejemo nekaj glavnih pristopov k virtualizaciji:

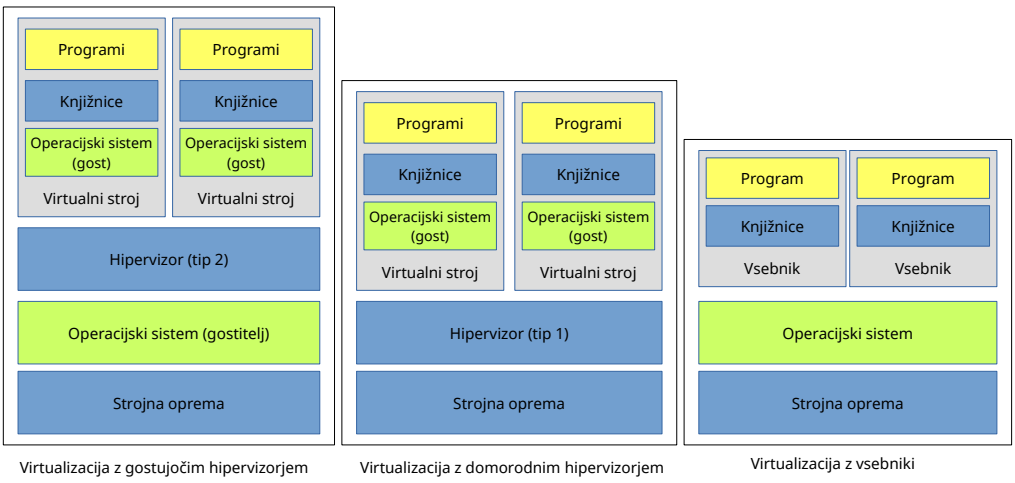
- Emulacija strojne (in programske) opreme (ang. *emulation*). Gre za zelo neučinkovit način virtualizacije, saj mora gostitelj programsko izvajati vse operacije, ki naj bi jih nudil emuliran sistem. Primer take izvedbe so bile prve verzije programa Virtualbox.

- Strojno podprta virtualizacija (ang. *hardware-assisted virtualization*) v primerjavi s prej omenjeno izvedbo izkoristi zmožnosti strojne opreme namenjene virtualizaciji, predvsem CPE. Tu se pojavi koncept hipervizorja (ang. *hypervisor*), ki nadzoruje virtualizirane vire in ustvarja, zaganja, nadzira in ustavlja virtualne stroje. V grobem lahko hipervizorje delimo na dve skupini, domoroden (ang. *native*) oziroma tip 1 in gostujoči (ang. *hosted*) oziroma tip 2. Hipervizor tipa 1 neposredno upravlja strojno opremo, medtem ko se tip 2 izvaja znotraj operacijskega sistema, kar pomeni, da strojno opremo upravlja OS. V praksi tako razlikovanje ni povsem točno, saj obstajajo hipervizorji, za katere lahko trdimo, da spadajo v obe skupini, npr. *Kernel-based Virtual Machine* (KVM) jedra Linux. Hipervizor KVM po eni strani deluje

kot drugi procesi, tako da uporablja vire, ki jih nudi jedro, po drugi strani pa je del jedra, zato ga prav tako lahko uvrstimo med domorodne hipervizorje.

- Posebej lahko omenimo še navidezne stroje namenjene izvajanju programov. Omogočajo enoten programski jezik, ki se lahko potem izvaja na različnih platformah (če seveda obstaja izvedba navideznega stroja za dotično platformo). Kot primere lahko navedemo JVM (*Java Virtual Machine*), ki izvaja *Java bytecode* in posledično podpira mnogo programskih jezikov, CLR (*Common Language Runtime*) na platformi Windows in BEAM (*Bogdan/Björn's Erlang Abstract Machine*), navidezni stroj za programski jezik Erlang.

- Vsebniške tehnologije (ang. *container technology*), virtualizacija na nivoju operacijskega sistema (ang. *operating-system-level virtualization*) oziroma lahka virtualizacija. Pristop je bolj podrobno opisan v nadaljevanju, v osnovi gre za izolacijo procesov in virtualizacijo virov, do katerih ti procesi dostopajo.



Slika 2.1: Primerjava nekaterih izvedb virtualizacije.

Na sliki 2.1 je diagram, ki ponazarja izvajanje programa v virtualnih strojih z različnima hipervizorjema ter v vsebniku. Bistvena razlika je, da pri vsebnikih vlogo hipervizorja igra OS, poleg tega znotraj vsebnika izvajanje dodatnega operacijskega sistema ni potrebno.

Našteli smo nekaj glavnih pristopov k virtualizaciji, ki so neposredno namenjeni izvajanju programov. Lahko bi preučili še virtualizacijo pomnilnika, podatkov, podatkovnih nosilcev, omrežij in podobno, a to za razumevanje vloge vsebnikov ni bistveno. Velja omeniti še, da definicije različnih načinov virtualizacije niso točno določene zaradi nejasnosti mej oziroma razlik med posameznimi načini, kot smo videli v primeru hipervizorja KVM.

2.3 Vsebniki

Točna definicija računalniškega pojma vsebnik (ang. *container*) ne obstaja, v splošnem označuje proces (oziroma skupino procesov), ki je z različnimi mehanizmi operacijskega sistema izoliran od ostalih procesov.

Lxc [25] definira vsebnike kot skupino procesov v operacijskem sistemu z jedrom Linux, katerim upravlja računske vire z nadzornimi skupinami in zagotavlja izolacijo virov z uporabo imenskih prostorov. Kaj točno to pomeni, razložimo v nadaljevanju.

Docker [20] definira vsebnike kot mehanizem, ki program ovije v dokončen datotečni sistem, ki vsebuje vse potrebno za izvajanje, torej programsko kodo, izvajalno okolje, sistemska orodja in sistemske knjižnice. Tako zagotovijo, da se program vedno izvaja pod enakimi pogoji ne glede na izvajalno okolje.

Čeprav se definicije razlikujejo, lahko vsebnike glede na število izvajanih procesov razdelimo na dva tipa, aplikacijske (ang. *application container*) in sistemske (ang. *system container*) vsebnike. Prvi tip vsebnika je namenjen izvajanju samo enega procesa, medtem ko drugi že vsebuje poljubno implementacijo programa init ter tako omogoča izvajanje več procesov, kar je zelo podobno virtualizaciji z navideznimi stroji.

Program znotraj vsebnika se izvaja na istem jedru operacijskega sistema

kot ostali (normalni) programi, a je s sistemskimi pravili izoliran, tako da ne more dostopati do virov ostalih procesov in virov operacijskega sistema. To je izvedeno s kombinacijo mehanizmov, ki so opisani v naslednji sekciji. V primerjavi s klasično zagnanim procesom ima proces v vsebniku omejen nabor dovoljenih sistemskih klicev, pogosto lahko dostopa samo do virtualiziranega dela datotečnega sistema. To pomeni, da na določeni datotečni poti vidi drugo datoteko, ki ni enaka prvotni datoteki v datotečnem sistemu. Ker so datoteke v operacijskih sistemih *NIX pogosto uporabljena abstrakcija, je s tem zagotovljena virtualizacija večine računalniških virov, ki jih proces uporablja.

Prednosti tega pristopa so hitrost izvajanja[1], avtomatizacija in enostavna uporaba. Ena od večjih slabosti vsebnikov je močna odvisnost od (jedra) operacijskega sistema, kar pomeni, da istega vsebnika ne moremo zagnati na OS GNU/Linux in OS Windows. Včasih obstaja neskladje celo med različnimi verzijami jedra Linux. Vsebniško orodje Docker se neskladnosti v Windows in OS X izogne z virtualizacijo sistema Linux, kar negativno vpliva na zmogljivost. Prav tako so neskladni formati različnih vsebniških tehnologij. Podjetja v industriji se omenjenih problemov zavedajo, zato je bilo leta 2015 ustanovljeno združenje *Open Container Initiative* [14], ki stremi k standardnemu vsebniškemu formatu.

2.4 Vsebniški mehanizmi v jedru Linux

V tem poglavju so opisani mehanizmi, ki so uporabni za izvedbo vsebnikov. Dodatno so predstavljene še nekatere alternative vsebniškim tehnologijam, ki jih ponujajo operacijski sistemi UNIX.

[1] Vsebniške tehnologije v primerjavi z drugimi izvedbami virtualizacije povzročijo minimalne dodatne stroške izvajanja.

2.4.1 chroot

Chroot [24] je funkcionalnost *NIX operacijskih sistemov, ki procesu (in vsem njegovim otrokom) zamenja pravo korensko mapo z navidezno. Takrat pravimo, da je proces v chroot ječi (ang. *chroot jail*). Ime je okrajšava za *change root*, kar prevedeno pomeni menjavo korena. Chroot se danes v Linux operacijskih sistemih za vsebniške tehnologije ne uporablja, ker ima mnogo pomanjkljivosti. Naštejemo le nekaj bistvenih:

- Proces z dovolj pravicami lahko izstopi iz chroot ječe s ponovnim klicem chroot funkcije.
- Za zadovoljivo delovanje moramo procesu v chroot ječi priskrbeti vse programe in knjižnice, ki jih uporablja. To se ponavadi izvede s kopiranjem relevantnih datotek v navidezno mapo procesa, kar je prostorsko ter časovno zelo neučinkovito.
- Zaradi možnosti, da si proces sam nastavi dovolj pravic (ang. *privilege escalation*) in izvede napad iz prve alineje, morajo imeti datoteke zelo natančno nastavljena dovoljenja za branje in pisanje, kar pa je, če upoštevamo drugo alinejo, zelo zapleteno in zamudno opravilo.

Mehanizem je bil za bolj resno uporabo zamenjan z vsebniki Linux, ječami BSD in *Solaris Zones*, vendar je še vedno vreden omembe zaradi zgodovinske vrednosti in osnovne ideje, ki so jo prevzele novejše rešitve.

2.4.2 Varni način računanja (*seccomp*)

SECure COMPuting [23] je funkcionalnost jedra Linux, s katero procesu omejimo dovoljene sistemske klice. Ker proces lahko dostopa do (navideznih) sistemskih virov samo preko sistemskih klicev, ta mehanizem poskrbi za zelo varno izvajanje programov. Prvotna verzija je po prehodu v varni način računanja procesu omogočila samo štiri sistemske klice: *read*() in *write*() nad prej odprtimi datotečnimi deskriptorji, *sigreturn*() in *exit*(). Prva dva klica sta uporabna za branje in pisanje datotek, tretji procesu omogoča uspešno

obravnavo signalov, s četrtim klicem pa proces zaključi izvajanje. Če je proces izvedel kak drug sistemski klic, je bil prisilno zaustavljen.

Problem mehanizma *seccomp* je bil preveč omejen nabor možnih sistemskih klicev, kar je precej omejilo njegovo uporabno vrednost. Zato so pozneje predstavili razširitev *seccomp-bpf* [17], ki omogoča natančno določanje dovoljenih in prepovedanih klicev z **BPF** oziroma *Berkley Packet Filter* pravili. BPF deluje kot interpreter pravil, podanih v obliki programa. To pomeni, da moramo za uporabo *seccomp-bpf* napisati ločen program, preko katerega bo BPF filtriral sistemske klice. Mehanizem primarno ni namenjen vsebniškim tehnologijam, ampak le zmanjšanju na napad izpostavljenih površin jedra Linux. To je uporabno za vsebniške tehnologije, saj ima proces v varnem načinu računanja manj vpliva na sistem in ostale procese.

2.4.3 Nadzorne skupine (*cgroups*)

Nadzorne skupine [3] združujejo procese v hirearhične skupine, te lahko podsistemi nadzornih skupin nadzorujejo, poleg tega opazujejo, omejujejo in prioritizirajo njihovo porabo sistemskih virov.

Med pomembnejše podsisteme nadzornih skupin sodijo:

- *Cpuset* določi, na katerem jedru procesorja se bo izvajala skupina in kje se bo nahajal njen pomnilnik.
- *Cpuacct* spremlja porabo procesorskega časa nadzorne skupine.
- *Cpu* nadzorni skupini omeji ali zagotavlja določen delež procesorskega časa.
- *Blkio* omeji pasovno širino do bločnih naprav, npr. trdih diskov, *flash* pomnilnika, CD/DVD naprav, datotek in drugo.
- *Memory* spremlja in omejuje porabo pomnilnika.
- *Pids* omeji največje število procesov v nadzorni skupini.

- *Freezer* omogoči prekinitev procesov v skupini. V kombinaciji z drugimi mehanizmi je ta podsistem uporaben za prenašanje procesov (med sistemi), ki se že izvajajo.
- *Hugetlb* omeji uporabo velikih pomnilniških strani (ang. *huge pages*), ki procesu omogočijo uporabo nestandardno velikih kosov virtualnega pomnilnika.
- *Net_prio* omogoča prioritizacijo omrežnih vmesnikov za skupine procesov.
- *Net_cls* označi odhodne omrežne pakete skupine procesov, označene pakete lahko potem upravljajo drugi mehanizmi. Kot primer navedemo požarne zidove in preusmerjanje paketov nadzorne skupine skozi oddaljeno virtualno omrežje.

Pod enakim imenom je bila v jedru Linux verzije 4.5 omogočena nova verzija tega mehanizma. V izogib zmedi se imenuje *cgroups v2*, vmesnik oziroma podsistemi pa so organizirani na drugačen način. Trenutno nudi tri podsisteme:

- *io* nadzoruje in omejuje porabo vhodno-izhodnih virov,
- *memory* regulira porabo pomnilnika,
- *cpu* regulira porabo procesorskega časa.

2.4.4 Imenski prostori

Imenski prostori [22] (ang. *namespaces*) ponujajo virtualizacijo sistemskih virov za skupine procesov. Procesi znotraj nekega imenskega prostora tako lahko vidijo in dostopajo samo do tistih (virtualiziranih) virov, ki jih ta imenski prostor ponuja, nimajo pa dostopa do virov v zunanjih imenskih prostorih in do glavnega, sistemskega vira. Funkcionalnost je v osnovi zelo podobna mehanizmu *chroot*, le da ponuja nadzor nad več različnimi sistemskimi viri in nima tako velikih varnostnih pomanjkljivosti.

V jedru Linux obstaja sedem imenskih prostorov:

1. **Cgroup** virtualizira podatke o nadzornih skupinah. Namen tega imenskega prostora je skrivanje informacij procesu, da ta ne more ugotoviti, da se izvaja znotraj vsebnika. Uporaben je tudi za lažje prenašanje izvajanih vsebnikov, saj lahko na drugem sistemu zagotovimo enake podatke nadzornih skupin.

2. **Network** virtualizira sistemske vire, ki so povezani z omrežji, npr. omrežne naprave, požarni zid, vrata, pravila za usmerjanje in tako dalje. Fizična omrežna naprava je dostopna samo v enem imenskem prostoru. Z virtualno omrežno napravo (ang. *virtual ethernet*) lahko vzpostavimo povezavo med več imenskimi prostori in tako procesom v drugih imenskih prostorih omogočimo dostop do omrežja.

3. **Mount** poskrbi za izolacijo seznama priklopov (ang. *mount*). Procesi znotraj takega imenskega prostora dostopajo do navideznega datotečnega sistema, kjer so pravi priklopi prekriti z navideznimi. Upravljamo lahko stopnjo deljenja, ki nam omogoča ali izolacijo ali deljenje posameznega navideznega priklopa. Stopnje deljenja so sledeče:

   - Deljeni priklop (ang. *shared mount*) omogoči navidezni priklop, ki je lahko dostopen procesom v več imenskih prostorih, spremembe znotraj tega priklopa so vidne povsod.
   - Zasebni priklop (ang. *private mount*) je dostopen samo znotraj imenskega prostora, v zunanjih imenskih prostorih spremembe niso vidne.
   - Podrejeni priklop (ang. *slave mount*) deluje podobno kot deljeni priklop, le da se spremembe propagirajo samo v smeri od nadrejenega k podrejenem imenskem prostoru. V podrejenem imenskem prostoru torej priklop deluje kot zasebni, v nadrejenih pa kot deljeni priklop.

- Nepovezljivi priklop (ang. *unbindable mount*) je funkcionalno skoraj enak zasebnemu, dodatno pa znotraj navideznega priklopa onemogoči *bind mount*, torej priklop datotek in datotečnih imenikov.

4. **IPC** virtualizira vire za medprocesno komunikacijo. Bistvena razlika v primerjavi z *mount* imenskim prostorom je drugačen način dostopa do objektov, ki služijo za medprocesno komunikacijo. Ti objekti namreč niso dostopni preko datotečnih poti.

5. **PID** imenski prostor izolira ID (identifikator) procesa. ID je tako zagotovo unikaten samo znotraj imenskega prostora. Proces ne vidi PID števil iz ostalih imenskih prostorov in zato ne more ugotoviti, koliko ostalih procesov se trenutno izvaja. Imenski prostor PID je uporaben tudi za prenašanje izvajanih vsebnikov, iz podobnih razlogov kot imenski prostor *cgroup*.

6. **User** imenski prostor izolira identifikatorje in atribute, ki so povezani z lastnikom izvajanega procesa, npr. ID uporabnika (ang. *user id*) oziroma *uid*, ID skupine (ang. *group id*) oziroma *gid*, zmožnosti uporabnika (ang. *capabilites*), itd. Znotraj uporabniškega imenskega prostora lahko proces izvaja privilegirane operacije, četudi zunaj njega nima dovolj pravic. Proces tako ne more vplivati na izvajalno okolje ostalih privilegiranih procesov, kar bi bila iz vidika varnosti sistema zelo nevarna sposobnost.

7. **UTS** imenski prostor virtualizira identifikatorja gostiteljskega sistema, ime gostitelja (ang. *hostname*) in omrežno ime skupine računalnikov (ang. *NIS domain name*).

2.4.5 Zmožnosti uporabnika (*capabilities*)

Klasične implementacije operacijskih sistemov UNIX so glede na dovoljene operacije procese delile v dve skupini – privilegirane (ang. *privileged*), izva-

jane s pravicami super uporabnika (ang. *root user*) in neprivilegirane (ang. *unprivileged*), z izvajalnimi pravicami normalnega uporabnika. Procesi s privilegiranimi dovoljenji lahko izvajajo poljubne akcije, neprivilegirani procesi pa morajo delovati skladno z dovoljenji operacijskega sistema.

V praksi se je pristop izjalovil, saj je skoraj vsak program za normalno delovanje potreboval nekaj operacij, omogočenih samo privilegiranim procesom. Za zmanjšanje tveganja so v jedru Linux predstavili zmožnosti (ang. *capabilities*), mehanizem, ki razdrobi pravice super uporabnika na bolj obvladljive enote. Tako lahko normalnemu uporabniškemu procesu OS dodeli samo del privilegiranih dovoljenj, v slučaju izgube nadzora pa program ne more vplivati na celoten sistem.

2.4.6 Mandatory Access Control (MAC)

Obvezna kontrola dostopa je mehanizem, kjer operacijski sistem pobudniku (ang. *initiator*) omeji, dovoli ali prepreči dostop oziroma izvajanje določene akcije nad nekim predmetom (ang. *object*). Pobudnik akcije je ponavadi proces, predmet pa je nek računalniški vir, npr. datoteka, drug proces v sistemu, omrežna povezava, del pomnilnika, vhodno-izhodna naprava, datotečni imenik in podobno.

V jedru Linux je MAC omogočen z modulom, ki implementira vmesnik *Linux Security Module* [11]. Tak pristop je bil izbran zaradi nasprotujočih si mnenj o pravilni izvedbi tega varnostnega mehanizma. Posledično je na voljo več implementacij, med glavne sodijo *SELinux*, *AppArmor* in *TOMOYO*. Navedene implementacije so del jedra Linux.

2.4.7 Podobne tehnologije v ostalih sistemih UNIX

V operacijskem sistemu FreeBSD so leta 2000 z različico FreeBSD 4.0 predstavili ječe [4] (ang. *jails*). Mehanizem je nadgradnja sistemskega klica *chroot*. Poleg datotečnega sistema procesu virtualizira tudi uporabnike, ostale procese in dostop do omrežja. Ječo definirajo štiri lastnosti: korenski imenik

ječe[2], ime gostitelja (ang. *hostname*), IP naslov ječe in ukaz, ki bo zagnal proces, izvajan znotraj ječe. V dokumentaciji svarijo, da ima rešitev še vedno nekaj varnostnih pomanjkljivosti, a je z vidika izvajanja procesov v ječi primerljiva z vsebniki Linux. Ti so bolj popularni samo zaradi razširjenosti operacijskih sistemov z jedrom Linux.

Še ena omembe vredna tehnologija so *Solaris Zones*, ki delujejo na podobnih principih kot FreeBSD ječe, dodatno pa izkoriščajo funkcije datotečnega sistema ZFS, ki jim omogoči ustvarjanje posnetkov (ang. *snapshot*) in klonov (ang. *clone*), torej učinkovit način replikacije virtualiziranih okolij.

Omenimo lahko še dve starejši tehnologiji, ki sta osnovani na jedru Linux – OpenVZ (Open Virtuozzo) in Linux-VServer. Čeprav sta obe rešitvi ponujali vsebnikom ekvivalentno funkcionalnost, se nista uveljavili, ker njune rešitve niso bile sprejete v glavno različico jedra Linux. Izvedeni sta z dodatnimi mehanizmi v jedru, tako kot ostale vsebniške tehnologije. Pojavile so se težave z združljivostjo med temi mehanizmi in nekaterimi deli jedra Linux, zato je bila večina mehanizmov dostopna le preko modificiranega (ang. *patched*) jedra, ki je pogosto zaostajalo za glavno različico. Razvijalci obeh tehnologij so nekaj rešitev prispevali v jedro Linux, nekaj pa v ostale vsebniške tehnologije, npr. LXC.

[2]V nasprotju s *chroot* proces znotraj ječe ne more dostopati do poti nad prej določenim korenskim imenikom.

Poglavje 3

Docker in ostale vsebniške tehnologije

V poglavju 2 smo predstavili virtualizacijo in mehanizme, ki omogočajo vsebniške tehnologije, v tem poglavju pa se osredotočimo na trenutno najbolj razširjen [9] format vsebnika, Docker. Poleg tega pregledamo še nekatere vidike, pomembne za izvajanje aplikacij v vsebnikih in primerjamo vsebniško tehnologijo Docker z nekaterimi drugimi vsebniškimi tehnologijami.

3.1 Zgradba vsebnika Docker

Vsebnik Docker [20, 28, 29] je proces z nekaj dodatki. Od mehanizmov, ki so predstavljeni v poglavju 2, privzeto uporablja imenske prostore (ang. *namespaces*) in nadzorne skupine (*cgroups*). Omogoča tudi dodatno koriščenje funkcionalnosti, ki jih nudijo *seccomp-bpf*, *AppArmor* in *SELinux*. Poleg teh mu ključne funkcionalnosti nudijo implementacije *UnionFS* in vsebniški format, podrobnosti so predstavljene v nadaljevanju tega poglavja.

3.1.1 UnionFS

UnionFS [33] temelji na navideznem zlivanju (ang. *merge*) ločenih datotečnih sistemov v zaključeno celoto. Ta funkcionalnost je izvedena z drugačno vrsto

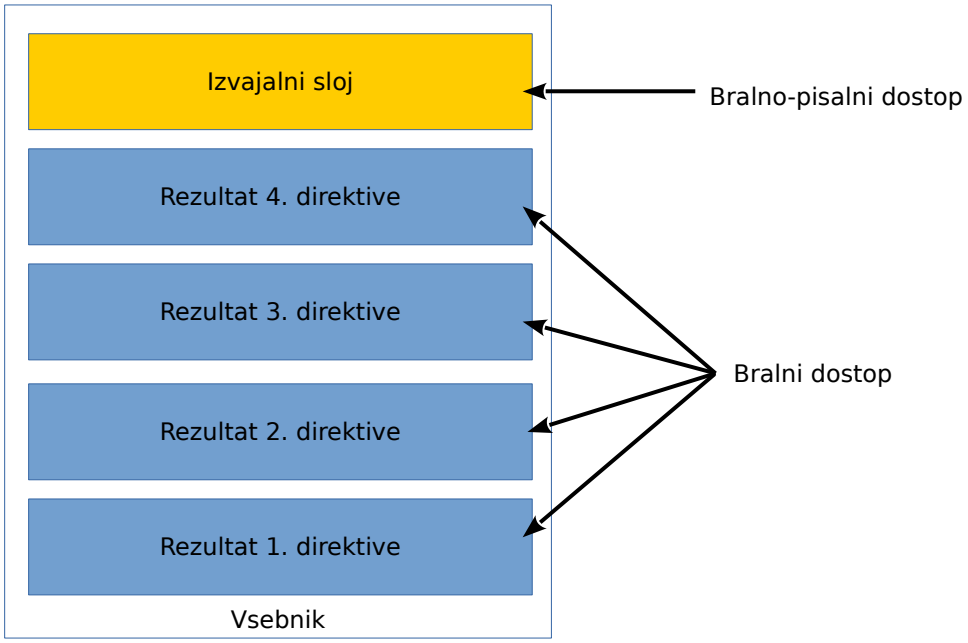
priklopa. Namesto prekrivanja priklopljene datotečne poti združi, tako da so hkrati dostopne datoteke iz priklopljenega sistema in datoteke, ki so se na datotečni poti nahajale pred priklopom. Prekrite so le datoteke z enakim imenom, v takem primeru je možen samo dostop do tiste iz zadnjega priklopa.

Čeprav UnionFS omogoča priklapljanje celotnih datotečnih sistemov, je v praksi bolj pogost priklop zelo majhnih delov datotečnih sistemov, imenovanih sloji (ang. *filesystem layers*). Tak pristop omogoči lažje obvladovanje sprememb datotek v datotečnem sistemu, kar se izkaže za zelo uporabno lastnost. Na nivoju datotečnega sistema omogoča upravljanje z različicami (ang. *version management*), tako lahko uporabnik v primeru težav povrne stabilno stanje datotečnega sistema. Datotečni sistemi, ki implementirajo koncept UnionFS, so sledeči: *AUFS*, *btrfs*, *xfs* in *DeviceMapper*.

Datotečni sistem vsebnika Docker je sestavljen iz sklada slojev datotečnega sistema UnionFS. Vrhnji oziroma nazadnje dodan sloj omogoča bralni in pisalni dostop, vsi ostali (nižji) sloji pa samo bralni dostop. Ob zagonu vsebnika proces Docker za nižje sloje uporabi vsebniško sliko (ang. *container image*), preko katere priklopi nov, prazen bralno-pisalni datotečni sloj. Vsebniška slika se ustvari s postopkom gradnje (ang. *build*). Med gradnjo proces Docker prebere direktive[3] iz datoteke *Dockerfile*. Za posamezno direktivo proces ustvari nov vsebnik, kot vsebniško sliko uporabi sloje prejšnjih direktiv in izvede z direktivo podane ukaze. Rezultate direktive zapiše v vrhnji sloj vsebnika ter prekine njegovo izvajanje. Novo ustvarjena vsebniška slika sestoji iz vseh slojev tega vsebnika. Na sliki 3.1 je prikazan zagnan vsebnik; pripadajoča vsebniška slika je bila zgrajena na podlagi štirih direktiv.

V javno dostopnem repozitoriju *Docker Hub* [7] so na voljo vsebniške slike programov in distribucij operacijskega sistema GNU/Linux. Med gradnjo nove slike vsebnika proces Docker privzeto prenese temeljne slike iz omenjenega repozitorija. Vsebniške slike distribucij vsebujejo osnovno programje in knjižnice, potrebne za izvajanje v okolju, ki je podobno določeni verziji (določene distribucije) operacijskega sistema GNU/Linux. Posledično lahko

[3]Navodila za gradnjo slojev.



Slika 3.1: Diagram datotečnega sistema izvajanega vsebnika Docker.

na primer v trenutno najnovejši verziji distribucije *Ubuntu 16.04* s pomočjo vsebnika Docker vzpostavimo izvajalno okolje druge distribucije, npr. *Fedora 22*. Vredno je poudariti, da izvajalno okolje ni povsem enako, kot bi bilo v pravi distribuciji, saj se proces še vedno izvaja na jedru distribucije Ubuntu. Za izvajanje programov so bistvene prave verzije odvisnih knjižnic (ang. *library dependency*), ki so priložene v vsebniški sliki.

Slike vsebnikov omogočajo tudi preprosto nadgrajevanje, saj kot osnovo pri gradnji nove slike lahko uporabimo katerokoli staro sliko. Ob izvajanju to lahko pripomore k še boljši izrabljenosti prostora v datotečnem sistemu, ker nekatere implementacije UnionFS v času izvajanja procesa znotraj vsebnika omogočajo souporabo istih slojev.

3.1.2 Format vsebnikov

Izvirna verzija vsebnikov Docker je bila osnovana na vsebnikih Linux (LXC), ki so jih nadgradili z orodji za lažjo gradnjo slik in s spremembo namembnosti. Čeprav je LXC že takrat omogočal hkratno izvajanje več procesov znotraj istega vsebnika, so se z vsebniki Docker osredotočili na izvajanje po enega procesa, kar imenujemo vsebnik za aplikacijo (ang. *application container*).

Docker je format vsebnikov LXC zamenjal za lastno rešitev *libcontainer*. Bistveni nameni prehoda so bili preprostejša podpora lastnim razširitvam vsebnikov in neodvisnost od LXC, kar bi lahko omogočilo domorodno (ang. *native*) izvajanje vsebnikov tudi v drugih operacijskih sistemih.

Danes vsebniki Docker temeljijo na vsebniškem formatu *runc*, ki je nadgradnja formata *libcontainer*, skladno s specifikacijo OCF.

3.1.3 Omrežja in podatki v vsebnikih

Čeprav vsebniški formati tega ne definirajo, so za uporabnost vsebnikov ključni načini vzpostavitve povezave z drugimi vsebniki in internetom ter pristopi za (trajno) shranjevanje podatkov zunaj kratkoživih vsebnikov.

Izvajalno okolje vsebnikov Docker ima za mreženje na voljo prekrivna (ang. *overlay*) omrežja. Vsebniki, ki so člani nekega omrežja, lahko dostopajo do ostalih vsebnikov v tem omrežju preko imena vsebnika (ang. *hostname*) in omrežnih vrat. Prav tako je mogoča istočasna prisotnost v več navideznih omrežjih. Bistvena prednost prekrivnih omrežij je trajnost, saj jih ustvarimo (in uničimo) neodvisno od vsebnikov, ki koristijo njihove zmožnosti.

Vsebniki Docker imajo na voljo štiri različne izvedbe omrežij:

- *Bridge* je privzeto izbrano omrežje, ki ponuja funkcionalnost mosta, torej navidezne naprave, ki prepušča omrežne pakete do prave omrežne naprave in zunanjega omrežja. Vsak vsebnik v takem omrežju je virtualno povezan z mostom in ima svoj naslov IP, ki je dostopen znotraj navideznega omrežja. Možno je tudi ustvarjanje novih, ločenih navideznih omrežij, ki so funkcionalno ekvivalentni omrežju *bridge*.

- *Host* je izvedba omrežja brez mosta, torej je vsebnik del omrežja gostiteljskega sistema. Zaradi odsotnosti navideznega mosta je ta način hitrejši in posledično smiseln za posredniške (ang. *proxy*) in predpomnilniške (ang. *cache*) strežnike. Za druge namene ni priporočen predvsem iz varnostnih razlogov, saj lahko vsebnik v tem omrežju dostopa do vseh izpostavljenih omrežnih vrat gostitelja, kar je lahko nevarno.

- *None* je zaprto omrežje, ki ne ponuja dostopa do drugih vsebnikov ali omrežij.

- *Container* je omrežje, ki uporablja omrežni imenski prostor drugega vsebnika in je ostanek funkcionalnosti, ki so bile v uporabi, preden so bila na voljo prekrivna omrežja. Uporabno je v primeru izvajanja več enakih vsebnikov, saj lahko z enim prednastavljenim vsebnikov ustvarimo prilagojeno omrežje, ki ga uporabljajo podvojeni vsebniki. Tak pristop za nekatere primere skaliranja uporablja orodje Kubernetes, ki je bolj podrobno opisano v poglavju 4.2.2.

Za dostop do vsebnika preko omrežja je nujna neka omrežna pot. Izvajalno okolje Docker nudi izpostavljanje omrežnih vrat (ang. *expose port*), torej preslikavo dostopne točke vsebnika v neka omrežna vrata gostiteljskega sistema.

Omenimo lahko še star način mreženja vsebnikov Docker, ki je bil aktualen pred prekrivnimi omrežji. Dostop do ostalih vsebnikov je bil možen z eksplicitno povezavo (ang. *link*), vzpostavljeno ob zagonu vsebnika, kar je bilo v primeru večjega števila vsebnikov zamudno in zapleteno. Ker je imel vsak vsebnik znotraj omrežja unikaten naslov IP, je bilo povezovanje izvedeno z zapisom naslova IP v datoteko /etc/hosts zaganjanega vsebnika. V primeru ciklične odvisnosti (vsebnik povežemo z drugim vsebnikom, slednjega pa bi tudi želeli povezati s prvim) je ta pristop popolnoma odpovedal, ker izvajanim vsebnikom ni mogoče uspešno spremeniti omenjene datoteke, s prekinitvijo izvajanja pa bi vsebniku spremenili naslov IP. Težavo je rešila uvedba dodatnega vsebnika, preko katerega sta komunicirala problematična procesa.

Osredotočimo se še na problem shranjevanja podatkov. Vsebniki Docker so prednostno namenjeni izvajanju kratkoživih, nadomestljivih programov. V primeru napak pri izvajanju je standardna praksa ustavljanje, brisanje in zagon novega vsebnika, kar lahko privede do izgube podatkov. To se rešuje z uporabo podatkovnih nosilcev (ang. *data volume*) in vsebnikov, namenjenih samo shranjevanju podatkov (ang. *data container*). Prva rešitev uporablja nosilce in datotečni sistem gostiteljskega sistema, zato je primerna za vse potrebe po shranjevanju podatkov. Podatkovni vsebniki so alternativa, uporabna v primeru, ko so podatki ključni samo za izvajanje vsebnika in želimo vsebnik med izvajanjem ustaviti ter zamenjati z novo verzijo. Slabost podatkovnih vsebnikov je možnost nenamernega izbrisa ob odstranjevanju starih ustavljenih vsebnikov.

Zanimivo rešitev za problem shranjevanja podatkov ima tudi orkestracijsko ogrodje Kubernetes. Pristop je opisan v sekciji 4.2.2.

3.2 Smiselnost uporabe vsebnikov

Uporaba vsebniških tehnologij ima veliko prednosti v primerjavi z drugimi načini virtualizacije (tudi v primerjavi s pristopom brez virtualizacije), ima pa tudi nekaj pomanjkljivosti. Oba vidika sta bolj podrobno razložena v nadaljevanju. Večina obravnavanih aspektov velja za vsebniške tehnologije v splošnem, posebej so poudarjene določene podrobnosti vsebnikov Docker.

Vsebniške tehnologije v primerjavi z drugimi izvedbami virtualizacije skoraj ne povzročijo dodatnih stroškov, oziroma so ti minimalni. Če kot primer vzamemo virtualne stroje, morajo ti za svoje delovanje najprej zagnati ločeno jedro operacijskega sistema, vse knjižnice in servise, ki jih ta OS potrebuje, šele nato lahko začnejo z izvajanjem virtualizirane aplikacije. V okviru vsebniških tehnologij se v večini primerov takoj zažene virtualizirana aplikacija, ki uporabi kar jedro gostiteljskega OS.

Z uporabo vsebnikov, kot jih ponuja Docker, lahko rešimo problem neskladnosti verzij odvisnih knjižnic. Pri programiranju je pogosta uporaba

knjižnic (ang. *library*), ki priskrbijo neko funkcionalnost. Neuporaba knjižnic pri pisanju programa ni smiselna, saj programer s tem izgublja čas in denar ter z večjo količino lastne programske kode povečuje tudi možnost napak. Ker se knjižnice tako kot vsa druga programska oprema redno posodabljajo, se pogosto zgodi, da program za delovanje potrebuje starejšo različico knjižnice. Možno je tudi obratno, da so knjižnice v sistemu zaradi zagotavljanja stabilnosti starejše, želimo pa pognati program, ki se opira na novejšo verzijo knjižnice. Vse te težave okrepi dejstvo, da je pogosto nemogoče imeti na istem sistemu hkrati več različnih verzij enake knjižnice. Z vsebniki se temu izognemo, saj lahko točno določimo verzije vseh uporabljenih knjižnic in verzijo izvajanega programa.

Vsebniške tehnologije nam lahko omogočijo relativno konsistentno izvajalno okolje v vseh stopnjah razvoja in izvajanja programa. Razvijalci lahko med razvojem testirajo izdelek v vsebnikih in enako okolje potem reproducirajo še v drugih fazah, vključno s produkcijskim okoljem. Tako se izognejo mnogim težavam, ki so posledica razlik med izvajalnimi okolji: različne verzije knjižnic in programja ter različni ne-virtualizirani računalniški viri, npr. podatkovne in omrežne naprave.

Vsakršna oblika virtualizacije omogoča uporabniku določeno mero avtomatizacije. S ponovljivim izvajalnim okoljem lahko vzpostavimo ustrezne pogoje za orodja, ki potem lahko avtomatizirajo večino opravil povezanih z gradnjo, testiranjem in izvajanjem aplikacij v vsebnikih.

Zaradi izvajanja na istem jedru in pripadajočih prihrankov računalniških virov so vsebniške tehnologije primerne za izvajanje v oblačnih arhitekturah. Sem sodijo ponudniki IaaS, PaaS in SaaS, ki lahko z uporabo vsebnikov prihranijo na račun manjše porabe virov. Zaradi fleksibilnosti vsebnikov so se nekateri ponudniki usmerili samo v organizacijo in orkestracijo storitev s pomočjo vsebnikov, te imenujemo vsebniki kot storitev (ang. *Containers as a Service*, CaaS).

Navodila oziroma skripte za gradnjo in zagon programov v vsebnikih imajo tudi vlogo dokumentacije izvajalnega okolja. V primerjavi z ročno

namestitvijo na strežniku lahko namreč v navodilih za gradnjo vsebnikov najdemo vse relevantne knjižnice ter njihove uporabljene različice. To sicer ni unikatna značilnost te tehnologije, ampak je zgolj posledica omogočene avtomatizacije.

Kljub prej naštetim prednostim imajo vsebniške tehnologije tudi slabe strani. Med najbolj očitne sodijo dodatna kompleksnost izvajalnega okolja, varnostni pomisleki in neprimernost vsebnikov za shranjevanje podatkov, kar se na primer pokaže pri izvajanju podatkovnih baz.

Najprej se posvetimo dodatni kompleksnosti. Čeprav uporaba vsebnikov poenostavi vzpostavljanje in izvajanje programov, nam hkrati omogoči izvajanje še več programov z različnimi okolji. Poleg tega uporaba dodatnih orodij (za nadzor in vzpostavitev vsebnikov) dvigne nivo znanja, potrebnega za razumevanje izvajalnega okolja neke aplikacije. To lahko predstavlja oviro za nove člane razvojne ekipe.

V primerjavi z drugimi virtualizacijskimi pristopi je z varnostnega vidika slabost vsebnikov izvajanje na jedru gostiteljskega OS. Kljub celi vrsti mehanizmov, ki skrbijo za nadzor in izolacijo, je za izgubo oblasti nad procesom včasih dovolj že ena sama ranljivost. Proces znotraj virtualnega stroja mora za podoben napad izkoristiti vsaj dve ranljivosti, eno v virtualnem stroju in drugo v jedru gostitelja.

Soroden problem se je pojavil, ker Docker za nadzor in izvajanje vsebnikov uporablja demona s pravicami super uporabnika (ang. *root user*) [5]. Ranljivost je bila izkoriščena v okviru izgradnje slike novega vsebnika; takrat je škodljiva programska koda prevzela nadzor nad demonom Docker in je zaradi pravic super uporabnika proces pridobil nadzor nad celotnim sistemom.

Naslednja varnostno problematična stvar je specifična za slike vsebnikov Docker. Ponujajo namreč register, ki je vsem uporabnikom dostopen za nalaganje in prenos vsebniških slik. Čeprav lahko z uporabo prenešene slike razvijalec prihrani precej časa, obstaja tveganje, da iz registra prenese vsebnik z zlonamerno vsebino.

Dodaten problem pri gradnji slik vsebnikov je, da v praksi ni možno

zagotoviti povsem ponovljive gradnje, torej da bi z istimi navodili vedno zgradili sliko istega vsebnika. Razlog so ukazi, ki pri gradnji slike prenašajo programe, knjižnice in ostale podatke iz oddaljenih virov. Izvajalno okolje tako ne more biti vedno zajamčeno konsistentno. Temu problemu se v veliki meri da izogniti z uporabo zasebnih repozitorijev slik vsebnikov, knjižnic in ostalih podatkov.

3.2.1 Open Container Initiative

Zaradi izrednega povečanja uporabe in zanimanja za vsebniške tehnologije so podjetja iz industrije leta 2015 ustanovile *Open Container Initiative* [14].

Glede na trenutne trende je vsebnik Docker sinonim za vsebniške tehnologije. To za celotno industrijo ni vzdržno, saj je za skladno delovanje orodij za nadzor in orkestracijo vsebnikov ter za prenosljivost aplikacij z uporabo vsebnikov ključno, da obstaja enoten, standarden format. OCI je zato predstavila specifikacijo OCF (ang. *Open Container Format*). Ta definira vsebnik, ki ni posebej vezan na specifike izvajalnega okolja ali orodja za orkestracijo, je prenosljiv med različnimi operacijskimi sistemi, strojnimi (ang. *hardware*) in oblačnimi platformami ter ni tesno povezan z nobenim podjetjem oziroma projektom.

3.3 Primerjava vsebniških tehnologij

V nadaljevanju predstavimo ključne lastnosti drugih vsebniških tehnologij ter jih primerjamo z Dockerjem. Ker so vsebniki večinoma izvedeni z mehanizmi naštetimi v sekciji 2.4, se v splošnem ne razlikujejo veliko, zato poleg samih vsebnikov primerjamo tudi druge elemente tehnologije in posebej poudarimo razlike v primerjavi z vsebniškim okoljem Docker. V tabeli 3.1 so primerjane vsebniške tehnologije glede na mehanizme, katerih funkcionalnosti so na voljo uporabnikom. Izkaže se, da je po kriteriju izpostavljenih mehanizmov drugačen systemd-nspawn, ki privzeto ponuja samo izolacijo imenskih prostorov, nekatere druge funkcionalnosti (npr. urejanje nadzornih skupin) pa

ponujajo ostala orodja pod okriljem projekta systemd.

tehnologija	nadzorne skupine	imenski prostori	seccomp	LSM	zmožnosti
Docker	da	da	da	da	da
LXC	da	da	da	da	da
LXD	da	da	da	da	da
systemd-nspawn	**ne**	da	**ne**	**ne**	**ne**
rkt	da	da	da	da	da

Tabela 3.1: Primerjava uporabljenih mehanizmov v vsebniških tehnologijah.

Če vsebniške tehnologije primerjamo po ostalih elementih, kot so npr. primarni tip vsebnika, gradnja in deljenje vsebniških slik ter sposobnost migracije izvajanih vsebnikov, lahko opazimo nekaj razlik. Tehnologije so podrobno primerjane v nadaljevanju, povzetek je v tabeli 3.2.

tehnologija	orodja za gradnjo slik	migracija izvajanih vsebnikov	uporabniški vmesnik	primarni tip vsebnika
Docker	da, lasten format	ne	CLI	aplikacijski
LXC	ne	ne	CLI	sistemski
LXD	ne	da	REST,CLI	sistemski
systemd-nspawn	ne	ne	CLI	aplikacijski
rkt	da, standardna UNIX orodja	je mogoče, odvisno od okoliščin	CLI	aplikacijski

Tabela 3.2: Ostale lastnosti primerjanih vsebniških tehnologij.

3.3.1 LXC

Ime je okrajšava za vsebnike Linux (ang. *linux containers*). LXC [10] je izveden kot kombinacija nadzornih skupin in imenskih prostorov in v nasprotju z vsebniki Docker privzeto uporablja tudi mehanizme LSM (opisano v sekciji 2.4.6), bolj konkretno AppArmor in SELinux. V okviru LXC je vsebnik lahko izoliran proces oziroma skupina izoliranih procesov, imenovana tudi sistem. Tako omogoča izvajanje vsebnikov, ki so po številu izvajanih procesov bolj podobni virtualnim strojem, kot vsebnikom Docker.

Prve implementacije vsebnikov Docker so bile osnovane na LXC, zato ni bistvenih razlik v načinu izvajanja. Močno se razlikujeta na področju kreiranja in deljenja vsebniških slik. LXC vsebuje nekaj predlog (ang. *template*) za ustvarjanje slik vsebnikov, ki vsebujejo izvajalna okolja razširjenih distribucij GNU/Linux. Ponuja tudi pisanje lastnih predlog, vendar nima mehanizmov, ki bi omogočali njihovo nadgrajevanje oziroma uporabo že zgrajenih slik vsebnikov, kar je velika pomanjkljivost. Vse to pomeni, da je pisanje lastnih predlog zelo zamudno. Poleg tega LXC nima centralnega repozitorija, kjer bi lahko uporabniki delili svoje predloge oziroma slike. Menimo, da so omenjene razlike glavni razlogi za večjo priljubljenost vsebnikov Docker.

3.3.2 LXD

LXD [10] upravlja in nadzoruje vsebnike LXC. Po funkcionalnosti bi ga lahko enačili s hipervizorji, saj ustvarja in zaganja navidezne stroje v obliki sistemskih vsebnikov. Med drugim ponuja upravljanje preko ukazne vrstice in preko REST vmesnika. Ključna razlika v primerjavi z vsebniki Docker je ta, da se vsebnik LXD smatra kot navidezni stroj, ki se bo izvajal dalj časa in bo med izvajanjem omogočal posodabljanje in menjavo komponent. Bolj splošno je Docker namenjen izvajanju aplikacijskih, LXD pa sistemskih vsebnikov. Ker LXD ni tesno povezan z LXC (ta skrbi za izvajanje in nadzor posameznega vsebnika), je ena od zanimivih funkcionalnosti možnost prenašanja že zagnanih vsebnikov med izvajalnimi okolji. To pomeni, da LXD vsebnik začasno

ustavi in ga prenese npr. na drug strežnik. Tam s pomočjo vsebniških mehanizmov vzpostavi navidezno izvajalno okolje, ki je dovolj enako okolju na prejšnjem strežniku, da lahko sistem znotraj vsebnika normalno nadaljuje z izvajanjem. LXD nadgradi LXC z naprednejšim upravljanjem vsebniških slik. Deljenje slik omogoča kar preko procesa LXD, statičnih strežnikov ali drugih protokolov. Kljub temu LXD ne ponuja centralnega repozitorija, kamor bi uporabniki lahko nalagali (in delili) svoje slike.

3.3.3 systemd-nspawn

Projekt systemd [19] skrbi za razvoj in podporo sistemskih programov, ki so nujni za delovanje vsakega računalniškega sistema. Najbolj znan je njihov istoimenski nadomestek za proces init. Ker so razvijalci projekta systemd mnenja, da bi vsebniki morali biti funkcionalnost, ki jo v celoti priskrbi operacijski sistem (v okviru privzetih knjižnic in sistemskih programov), razvijajo alternativno vsebniško izvajalno okolje, imenovano systemd-nspawn. Ta je po funkcionalnostih najbolj podoben vsebnikom LXC, torej lahko poganja tako vsebnike z enim procesom kot tudi celotne sisteme znotraj vsebnikov. Razlikuje se v uporabljenih mehanizmih, saj ponuja le funkcionalnosti imenskih prostorov in ne podpira nadzornih skupin ter ostalih. V primerjavi z Docker in LXC ima eno prednost, kot del projekta systemd je namreč privzeto nameščen na večini sistemov z operacijskim sistemom GNU/Linux, kjer vsebnike nadzorujejo kar proces init in ostali sistemski programi, npr. machinectl. Po drugi strani je systemd-nspawn samo vsebniško izvajalno okolje brez izgradnje in deljenja (ang. *sharing*) vsebniških slik ter podobnih naprednih funkcionalnosti. Pričakovan način uporabe je ročno (ang. *manual*) ustvarjanje datotečnega imenika, primernega za izvajanje vsebnika. Pravzaprav bi lahko systemd-nspawn označili kot boljšo implementacijo mehanizma chroot, brez očitnih varnostnih pomanjkljivosti.

3.3.4 rkt („rocket“)

Rkt je vsebniško izvajalno okolje, ki se od Dockerja razlikuje v mnogo pogledih. Projekt je bil ustanovljen leta 2014 ob odkritju več varnostnih ranljivosti v arhitekturni zasnovi vsebnikov Docker in pripadajočega ekosistema. Rkt je modularno izvajalno okolje za appc format vsebnika, ostali aspekti pa so prilagodljivi. V nadaljevanju so prikazane ključne razlike med obema tehnologijama.

Vsebniške slike

Za gradnjo vsebniških slik rkt uporablja standardna (in uporabnikom znana) skriptna orodja v UNIX okolju, npr. bash. Gradnja vsebniških slik Docker se izvede z datoteko, ki vsebuje direktive, te pa izvajajo želene ukaze. Direktive so unikatne, zato se mora uporabnik za gradnjo slik najprej naučiti specifičnih pravil za pisanje navodil za gradnjo.

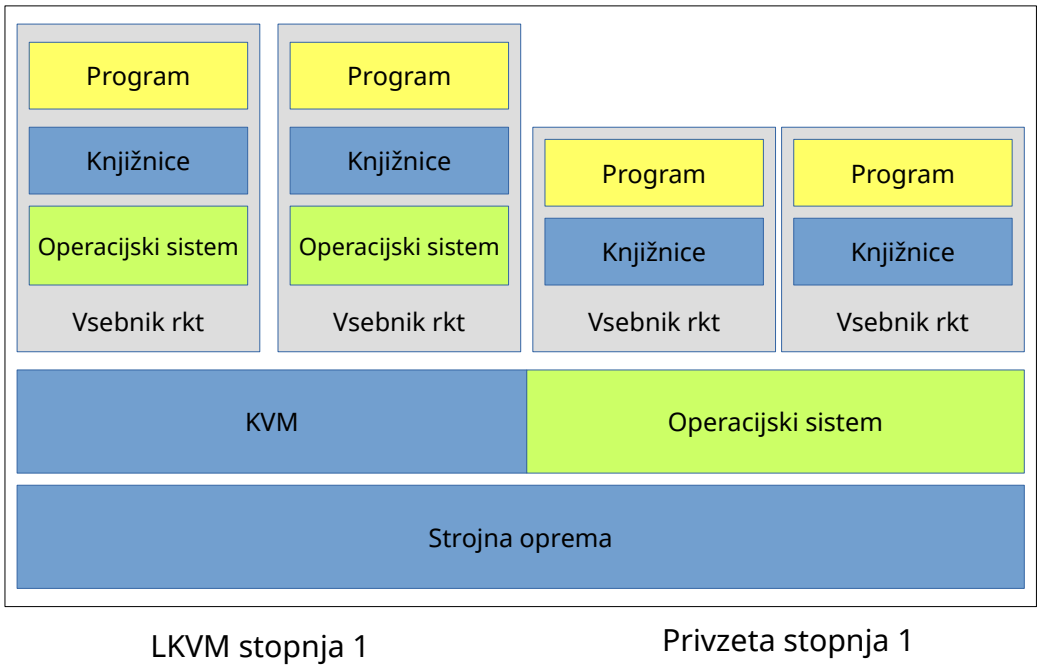
Rkt privzeto preveri podpis vsebniške slike pri prenosu iz oddaljenega repozitorija in tako garantira pristnost (avtorja) slike. Docker omogoča preverjanje podpisa slike, a ker privzeto tega ne naredi, obstaja nevarnost napada *Man-In-The-Middle*.

Za distribucijo vsebniških slik je pri rkt dovolj statični spletni strežnik s HTTPS, prav tako je možno prenašanje slik preko *p2p* protokola BitTorrent. Docker po drugi strani omogoča prenos in nalaganje slik iz zunanjih strežnikov samo v primeru, da ti strežniki izvajajo *Docker Registry*.

Zaganjanje vsebnikov

Zaradi omogočanja modularnosti je izvajanje vsebnika rkt razdeljeno na tri stopnje (ang. *stage*). V stopnji 0 rkt pripravi podatke, datotečni sistem in vsebniške slike za naslednjo stopnjo, v okviru stopnje 2 se znotraj vsebnika izvaja ciljni proces. Najbolj zanimiva je stopnja 1, kjer izbrano orodje vzpostavi vsebniško okolje. Privzeto je uporabljen systemd-nspawn, ki ustvari podobne vsebnike kot Docker ali LXC. Obstaja tudi implementacija, ki s

hipervizorjem KVM ustvari navidezni stroj, znotraj tega zažene proces init in izbrani program. Ta pristop, imenovan LKVM stopnja 1, izboljša varnost „vsebnika“ na račun zmogljivosti ter hkrati zamegli mejo med vsebniki in drugimi načini virtualizacije. Vseeno menimo, da je koncept dober, saj z enakim programskim vmesnikom ponuja možnost bolj varnega načina izvajanja. Na sliki 3.2 je diagram, ki nazorno pokaže razliko med vsebniki glede na implementacijo stopnje 1. Po zagonu je nadzor nad vsebnikom prepuščen procesu init (npr. systemd), ali pa orodju za orkestracijo, npr. Kubernetes.



Slika 3.2: Primerjava vsebnikov rkt z različnimi implementacijami stopnje 1.

Docker za nadzor in podobne operacije nad vsebniki uporablja demona, ki za zagon vsebnikov potrebuje pravice super uporabnika. Ta demon med drugim izvaja nadzorne operacije, za katere že obstajajo zmogljive rešitve, npr. programi init. Pristop ima tudi varnostne pomanjkljivosti, te so opisane v drugem delu podpoglavja 3.2.

Format vsebnika

Docker ima lasten format vsebnika runC, ki je skladen z OCF. Rkt po drugi strani podpira dva različna formata, vsebniški format runC, ki ga uporablja Docker, in vsebniški format appc [1]. Ta specifikacija definira format slike vsebnika, načine odkrivanja slik, način združevanja več vsebnikov v eno izvajalno (logično zaključeno) skupino[4] in izvajalno okolje takih skupin. Enako kot vsebnik Docker so tudi vsebniki formata appc namenjeni izvajanju aplikacij, ne pa sistemov.

3.4 Vpliv vsebnikov

Pred uveljavitvijo vsebniških tehnologij so bili za namene izvajanja aplikacij in storitev pogosto uporabljeni virtualni stroji ter pripadajoča orodja za avtomatizacijo (npr. Ansible, Chef in Puppet). Glavna problema uporabe virtualnih strojev sta bila počasen zagon ter večja poraba računalniških virov zaradi virtualizacije celotnega operacijskega sistema z namenom izvajanja le nekaj aplikacij. Vsebniki nudijo v primerjavi z virtualnimi stroji hitrejši zagon [32] ter tudi hitrejše izvajanje ciljne aplikacije [27]. Zaradi hitre vzpostavitve vsebnika postane smiselna izvedba aplikacije razdeljene na manjše neodvisne enote, ki jih v primeru napak preprosto zamenjamo z novimi instancami. Zaradi instantnega zagona vsebnika postane uporabno tudi horizontalno skaliranje posamezne enote – tako se lahko aplikacija hitro in brez težav odzove na povečano obremenitev. Vsebniki so torej spodbudili nov arhitekturni pristop h gradnji aplikacij imenovan arhitektura mikrostoritev. Arhitekturo mikrostoritev bolj podrobno opišemo v naslednjem poglavju.

[4]Taka skupina se imenuje *pod*, konceptualno je enaka istoimenskim skupinam v orkestracijskem ogrodju Kubernetes.

Poglavje 4

Mikrostoritve

Arhitekturna zasnova je eden od vidikov, ki močno vpliva na delovanje in uporabnost aplikacije. Pravilno zastavljen sistem ima večino časa dovolj kapacitet za obravnavo vseh uporabnikov, to pa zagotavlja za čim manjšo ceno. Dober primer zasnove, ki omogoča tak sistem tudi pri velikem številu uporabnikov z mnogo različnimi načini dostopa, je arhitektura mikrostoritev.

V nadaljevanju tega poglavja pojem aplikacija pomeni programje, ki skupaj tvori mikrostoritev.

4.1 Definicija

Arhitektura mikrostoritev [12, 30] opisuje aplikacijo, sestavljeno iz majhnih, zaključenih enot. Vsaka enota ima naslednje lastnosti:

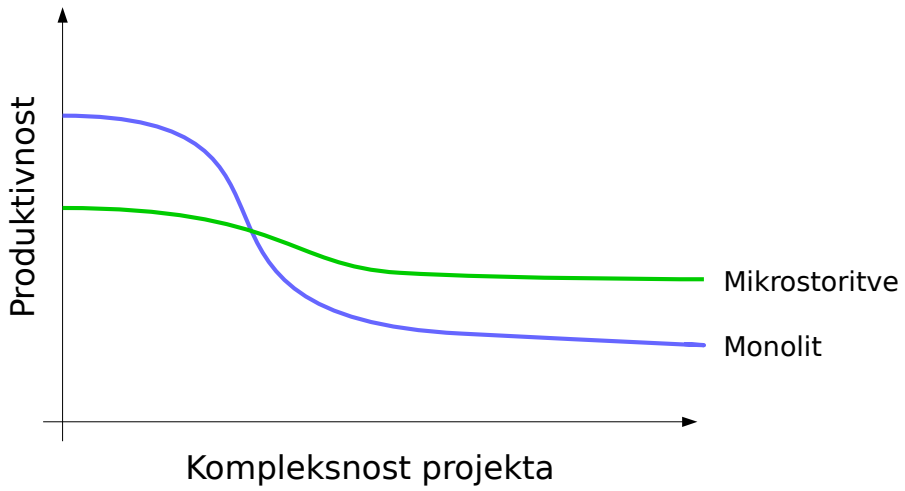
- ima točno določen namen oziroma funkcijo,
- je napisana v primernem programskem jeziku,
- komunicira z drugimi enotami preko poljubnih povezav (pogosto je to HTTP),
- ponavadi skrbi za del podatkov, ključen samo za delovanje te enote,

- v primeru napak na ostalih enotah nadaljuje z izvajanjem, če je to mogoče,
- je izvajana neodvisno od ostalih enot, kar omogoči preprosto nadgradnjo ali menjavo posamezne enote,
- omogoča preprosto dodajanje novih enot,
- je brez stanja, kolikor je to mogoče.

Da bi razumeli razloge za arhitekturo mikrostoritev, moramo najprej poznati njihovo nasprotje, torej monolitno arhitekturo. Take aplikacije vse funkcionalnosti ponujajo v okviru enega samega programa, kar precej zmanjša kompleksnost izvajanja, a se v primeru večjega števila uporabnikov ne obnese dobro. Prav tako je težko zagotavljanje visoke razpoložljivosti (ang. *high availability*) – če to zagotavljamo z več izvajalnimi strežniki, lahko z monolitno aplikacijo naletimo na težave pri sinhronizaciji med posameznimi instancami. Prav tako moramo na vsakem strežniku zagnati celotno aplikacijo, čeprav je pod visoko obremenitvijo samo določena komponenta aplikacije. Dodaten problem lahko nastane s prepletenostjo komponent znotraj aplikacije, kar oteži popravljanje in nadgrajevanje aplikacije. Monolitni pristop je bolj primeren za preprostejše projekte, namenjene manjšemu številu (sočasnih) uporabnikov. Pri takih projektih uporaba mikrostoritev ni smiselna.

Na grafikonu 4.1 je nakazana relativna odvisnost produktivnosti programerske ekipe glede na kompleksnost projekta in izbrano arhitekturno zasnovo. Pri manj kompleksnem projektu je monolitna zasnova vsekakor boljša izbira, ko pa se kompleksnost poveča, je padec produktivnosti z arhitekturo mikrostoritev manjši. Vredno je omeniti, da so razmerja prikazana relativno in so zelo odvisna od sposobnosti programerske ekipe ter ostalih faktorjev.

Velikost posamezne enote v mikrostoritvi ni določena s številom vrstic programske kode, s časom vloženim v razvoj ali z velikostjo datoteke izvršljivega programa, temveč s številom različnih opravil, ki jih ta enota opravlja. Zaželeno je, da so vsa opravila ene enote namenjena nudenju neke zmožnosti (ang. *capability*) oziroma funkcionalnosti, npr. spletni uporabniški vmesnik,



Slika 4.1: Primerjava produktivnosti glede na kompleksnost projekta.

vodenje inventarja, integracija s storitvami drugih podjetij, beleženje aktivnosti, itd. Če se med razvojem ugotovi, da je neka enota postala preveč kompleksna oziroma služi več različnim namenom, je priporočena delitev na manjše, bolj preproste enote s točno določeno funkcijo.

Posamezne enote med seboj komunicirajo preko različnih (omrežnih) protokolov. Med pogosto uporabljene načine komunikacije sodijo *Representational State Transfer* (REST), *Simple Object Access Protocol* (SOAP), klici za oddaljeni postopek (ang. *remote procedure call*, RPC), različni sporočilni protokoli (npr. *Asynchronous Messaging Queue Protocol*, AMQP), komunikacija z raznimi binarnimi formati in podobno. Načine komunikacije lahko razdelimo na sinhrone in asinhrone. Pri prvem načinu komponenta odda zahtevo drugi komponenti ter počaka na odgovor, medtem ne izvaja drugih operacij. Z asinhrono komunikacijo komponenta samo odda zahtevo za neko akcijo ter nadaljuje z izvajanjem. Pogosto niti ne preveri, če se je akcija sploh izvedla. Čeprav sta prednosti prvega načina komunikacije precej preprostejša implementacija in lažje razumevanje celotnega sistema, so v okviru mikrosto-

ritev bolj vidne slabosti tega pristopa. Čakanje oziroma blokiranje zmanjša zmogljivost komponente, poleg tega lahko vodi v izpade celotne mikrostoritve, če komponente niso pravilno povezane. Več o ohranjanju stabilnosti mikrostoritve je napisano v sekciji 4.2.4. V splošnem velja, da naj se uporabi tisti protokol, ki je najbolj primeren za izvedbo neke funkcionalnosti. Na primer zahtevek za izvedbo nekega dalj časa trajajočega opravila bo izveden preko RPC, rezultati pa se ob zaključku opravila posredujejo preko protokola REST. Če se pojavi potreba po izvedbi mnogo opravil iste enote, bodo zahteve za ta opravila oddana preko sporočilne vrste in tako dalje.

Uporaba iste podatkovne baze v okviru mikrostoritev pogosto ni smiselna. Razlogov je več: vse enote ponavadi ne potrebujejo dostopa do vseh podatkov, če pa si že delijo določene podatke, so ti pogosto interpretirani na različne načine, oziroma je zanje relevanten samo nek manjši del. V izogib tem nevšečnostim lahko vsaka mikrostoritev uporablja svojo, ločeno podatkovno bazo. Dodatna prednost tega pristopa je, da kriterij za izbiro podatkovne baze ni cela aplikacija, ampak samo potrebe obravnavane komponente. Tako poleg relacijskih podatkovnih baz za določene namene postanejo smiselne tudi ne-relacijske podatkovne baze (ang. *NoSQL database*).

Podobno kot pri izbiri podatkovne baze imajo razvijalci relativno proste roke tudi pri izbiri tehnologije posamezne komponente. Za določene namene se v praksi pogosto izbere kar enega izmed odprtokodnih programov, ki rešujejo določen problem. Navedemo lahko razne podatkovne baze (npr. MySQL, PostgreSQL, MongoDB, Redis, Cassandra, itd.), spletne strežnike (npr. Nginx, Apache HTTP Server, httpd, Caddy, itd.), programe za beleženje (ang. *logging*), avtentikacijo, avtorizacijo in tako dalje. Glavni kriteriji pri izbiri programskega jezika oziroma tehnologije so:

- znanje določenega programskega jezika in pripadajočih knjižnic,
- dostopnost ogrodij, ki močno olajšajo razvoj v nekem jeziku,
- primernost tehnologije za pisanje programov z želeno funkcijo,
- želje in zahteve naročnika.

Čeprav arhitektura mikrostoritev omogoča implementacijo vsake komponente v drugačni tehnologiji, je izbiro smiselno zmanjšati na tiste programske jezike in ogrodja, ki jih razvojna ekipa obvlada in so primerna za namen komponente. Če je npr. zahtevana komunikacija preko protokola SOAP, bo za to najbolj primerna implementacija v tehnologijah JavaEE ali .NET, ki imata za ta protokol boljšo podporo kot ostale tehnologije. Še en primer bi bila komponenta, ki mora nuditi visoko zmogljive povezave WebSocket. Za tak namen bi bile primerne implementacije v programskih jezikih C++ ali Go.

Enote mikrostoritev naj bi bile izvajane neodvisno od ostalih enot, kar pomeni, da napake v ostalih enotah ne bi smele vplivati na obravnavano enoto. Tako obratovanje je mogoče, če aplikacija ni močno prepletena. Če se enote res izvajajo neodvisno, to programerjem omogoči nadgrajevanje, popravljanje in tudi dodajanje povsem novih enot brez prekinitev v izvajanju ostalih delov aplikacije.

Še ena pogosto pomembna lastnost mikrostoritev je, da obratujejo brez stanja (ang. *stateless*), torej da zaporedna opravila, ki jih enota opravlja, niso medsebojno odvisna. Nasprotje so transakcije oziroma opravila s stanjem (ang. *stateful*), kjer se več opravil izvede v okviru ene seje. Podatke o posamezni seji mora storitev ohranjati vsaj do zaključka le-te. Izvajanje storitev s stanjem je težko ustaviti brez prekinjanja trenutno odprtih sej, zato tak pristop za mikrostoritve ni primeren. Dodatna prednost, ki jo prinesejo enote mikrostoritev brez stanja, je enostavnost horizontalnega skaliranja. Ker storitvam ni treba ohranjati podatkov o sejah, v primeru več izvajanih enot ni pomembno, katera enota bo obdelala uporabnikovo zahtevo. Če pa skaliramo storitev s stanjem, mora vsak zahtevek v okviru iste seje prispeti do iste instance izvajane storitve, kar poveča kompleksnost usmerjanja oziroma izenačevanja obremenitve (ang. *load balancing*).

Kljub prej omenjenim lastnostim se med posameznimi komponentami še vedno lahko ustvarijo medsebojne odvisnosti, ko npr. neka enota za delovanje nujno potrebuje dostopno podatkovno bazo. Obstajajo orodja, ki pri zagonu te odvisnosti upoštevajo, npr. Docker Compose, ki poskrbi, da so odvisne

komponente že zagnane ter Kubernetes, ki omogoči interakcijo šele takrat, ko je odvisna komponenta pripravljena na uporabo.

Mikrostoritve omogočajo tudi lažje spreminjanje števila programerjev in razvojnih ekip v organizacijah. Če je vsaka razvojna ekipa zadolžena za nekaj komponent, se s tem zmanjša nivo potrebnega znanja o celotni aplikaciji, zato se novi člani preprosteje pridružijo ustaljenim razvojnim ekipam. Prav tako lahko razvoj nove komponente prevzame povsem druga ekipa.

V povezavi z ekipami in arhitekturo aplikacij lahko omenimo Conwayev zakon. Ta pravi, da so sistemi, ki jih izdela neka organizacija, oblikovani podobno kot komunikacijska struktura znotraj organizacije. Razlog za to je komunikacija, potrebna za usklajevanje med razvojnimi skupinami. Če vsaka od teh skupin skrbi samo za svoj arhitekturni nivo aplikacije (npr. podatkovna baza, zaledje (ang. *backend*) s poslovno logiko, spletni uporabniški vmesnik), lahko spremembe v zahtevah povzročijo veliko dodatnega dela, saj se morajo skupine za uspešno rešitev uskladiti. Nasprotje opisani organizaciji ekip, ki se imenuje silosna delitev, je delitev po poslovnih zmožnostih (ang. *business capability*), oziroma po domeni ali funkcionalnosti. Tako je posamezna ekipa odgovorna za vse nivoje svojega dela aplikacije in v primeru sprememb v zahtevah (ki so pogosto povezane samo z eno domeno) ni potrebe po komunikaciji z ostalimi ekipami. Slednji način organizacije je priporočen za razvoj mikrostoritev.

4.1.1 Zgodovina

Princip uporabe majhnih programov s točno določeno funkcionalnostjo je bil eno od vodil pri razvoju prvih verzij operacijskega sistema UNIX v 70. letih prejšnjega stoletja. Filozofijo [2] razvoja tega OS lahko strnimo v naslednje točke:

- Programi naj opravljajo samo eno nalogo in to opravijo dobro.
- Izhod programa bo lahko uporabljen kot vhod drugega, trenutno še neznanega programa, zato naj bodo podatki oddani v obvladljivem

formatu.

- Programi naj bodo napisani tako, da lahko sodelujejo.

Čeprav arhitektura mikrostoritev in operacijski sistem UNIX nista zasnovana z enakim namenom, lahko med pristopoma najdemo mnoge podobnosti. Pri obojih gre za združevanje manjših programskih enot oziroma komponent, ki skupaj ponujajo funkcionalnosti sistema. Z uporabo majhnih enot se pristopa izogneta kompleksnosti znotraj programov. Izpostavimo lahko tudi nekaj podrobnosti, kjer se pristopa razlikujeta, npr. format podatkov, način komunikacije in uporabniški vmesnik. To lahko razložimo z razlikami v namembnosti – UNIX je večopravilni, večuporabniški operacijski sistem, medtem ko so mikrostoritve arhitekturni pristop za visoko razpoložljive na okvare neobčutljive (ang. *fault tolerant*) porazdeljene storitve. V operacijskih sistemih UNIX je za prenos podatkov med komponentami priporočeno golo besedilo (ang. *plain text*), razni binarni formati so nezaželeni. Nasprotno arhitektura mikrostoritev ne določa formata komunikacije med posameznimi komponentami, vendar so za ustrezno delovanje priporočeni jasno definirani vmesniki. Če nadaljujemo z vmesniki, je najbolj pogost način komunikacije med procesi, ki v OS UNIX skupaj ponujajo neko funkcionalnost, cevovod (ang. *pipeline*). Operacijski sistem glede na ukaze poveže standardni izhod enega procesa s standardnim vhodom drugega procesa in tako vzpostavi enosmerno komunikacijo. Pristopa se razhajata tudi v načinu povezovanja več komponent. Komponenta mikrostoritve ima ponavadi že med razvojem določene vse ostale komponente, s katerimi bo komunicirala in koristila njihove funkcionalnosti. Na drugi strani program operacijskega sistema UNIX še med izvajanjem zgolj bere vhodne podatke in rezultate oddaja naprej, brez vednosti o ostalih programih, s katerimi komunicira. To je tudi razlog za strogo definiran format golega besedila, saj je le na tak način možno doseči zadovoljivo združljivost različnih programov.

V mnogih pogledih je arhitekturi mikrostoritev podobna tudi storitveno usmerjena arhitektura (ang. *Service-oriented architecture*, SOA) [30]. Ta določa način arhitekture, kjer komponente aplikacije nudijo storitve drugim

komponentam preko nekega komunikacijskega protokola, najpogosteje je to omrežna povezava. Kljub temu lahko opazimo razlike v namembnosti obeh arhitektur. SOA je namenjena organizaciji večjega števila programov z namenom zmanjšanja stroškov in izboljšanja fleksibilnosti podjetij ter organizacij, ki tak pristop uporabljajo, arhitektura mikrostoritev podobne principe (s podobnimi nameni) uveljavlja zgolj na nivoju ene aplikacije oziroma storitve.

4.1.2 Slabosti arhitekture mikrostoritev

V predhodnih sekcijah tega poglavja smo že nakazali večino lastnosti obravnavane arhitekture. V tem delu zberemo in bolj podrobno opišemo nekaj negativnih vidikov uporabe mikrostoritev.

Kot prvo slabost lahko omenimo prepričanje, da uporaba mikrostoritev zmanjša kompleksnost programov. Čeprav je res, da so posamezne komponente v primerjavi z monolitno aplikacijo manjše in preprostejše, se kompleksnosti ne znebimo, le prestavimo jo izven komponente. Kar bi bil v monolitni aplikaciji programski klic neke komponente, je v mikrostoritvi oddaljen klic do neke storitve, za katero nismo prepričani, da deluje. Poleg tega so za organizacijo komponent znotraj monolitne aplikacije (ponavadi) dovolj že mehanizmi, ki jih ponuja ali programski jezik ali izbrano ogrodje, medtem ko za nadzor in organizacijo komponent mikrostoritve nujno potrebujemo dodatno podporno programje in znanje uporabe tega programja.

Naslednja slabost arhitekture mikrostoritev so nekatere lastnosti porazdeljenih sistemov. Ker komponente komunicirajo preko omrežja, se pri obratovanju pojavljajo zakasnitve in s tem povezani problemi. Omenimo lahko tudi izrek CAP [30], ki pravi, da je za porazdeljen sistem nemogoče hkrati zagotavljati naslednje tri lastnosti:

- usklajenost (ang. *consistency*),
- dosegljivost (ang. *availability*),
- odpornost na delitev omrežja (ang. *partition tolerancy*).

Usklajenost pomeni, da na neko zahtevo vse komponente odgovorijo z enakim odgovorom, dosegljivost pomeni, da pobudnik za vsako zahtevo prejme odgovor, odpornost na delitev omrežja pa je sposobnost obratovanja sistema, ko nekatere povezave niso na voljo, torej komunikacija med določenimi komponentami ni mogoča. Ker je odpornost na delitev omrežja nujna, se v praksi pojavljata dva sistema, kombinacija usklajenosti in odpornosti na delitev omrežja (CP) ter alternativa, ki za dosegljivost žrtvuje usklajenost (AP). Eden od namenov uporabe arhitekture mikrostoritev je visoka razpoložljivost, zato jih uvrščamo v AP tip porazdeljenih sistemov. Omenimo lahko pojem zakasnjene usklajenosti (ang. *eventual consistency*), ki je lastnost AP tipa sistemov. Zakasnjena usklajenost nam garantira, da se bo določen podatek sčasoma uskladil na vseh vozliščih oziroma komponentah porazdeljenega sistema, če medtem ne bo spremenjen. Točen čas oziroma trajanje do sinhronizacije ni določeno, to pa od razvijalcev mikrostoritev zahteva dodatne razmisleke in posebno pazljivost pri obravnavi podatkov, saj ne morejo zagotovo vedeti, če gre pri obravnavanih podatkih res za aktualne različice.

4.2 Izvajanje mikrostoritev

Uporaba mikrostoritev je bila zares omogočena šele z dostopnostjo vsebniških in drugih virtualizacijskih tehnologij [26, 29, 30]. V tej sekciji so obravnavani pomembni vidiki izvajanja mikrostoritev z vsebniškimi tehnologijami.

4.2.1 Odkrivanje storitev

Ena od pomembnih lastnosti komponent mikrostoritev je zaganjanje (in ustavljanje) neodvisno od ostalih komponent, zato se pojavi potreba po odkrivanju storitev (ang. *service discovery*). Gre za zmožnost komponente, da se ob zagonu registrira in drugim komponentam javi, da je dosegljiva. Drugi namen odkrivanja storitev je omogočanje iskanja že zagnanih (registriranih) komponent.

V osnovi za problem odkrivanja storitev obstajata dve rešitvi – sistem domenskih imen (DNS) in dinamični registri servisov (ang. *dynamic service registry*). DNS je sicer preprostejši, a ima nekaj pomanjkljivosti v primeru zelo dinamične mikrostoritve, kjer se komponente stalno zaganjajo, ustavljajo in spreminjajo. Naslovi pridobljeni preko DNS imajo življenjsko dobo (ang. *time to live*) – časovno obdobje, ko naj bi bil naslov pravilen. Če mikrostoritvi odvzemamo (ali dodajamo) instance komponent, se lahko zgodi, da bo DNS vnos pri uporabniku kazal na staro storitev, ki mogoče sploh ni več na voljo. Tako situacijo se lahko reši z izenačevalnikom obremenitve, kjer se registrirajo nove komponente. To že meji na drugo skupino rešitev, dinamične registre servisov. Ti večinoma temeljijo na nekem centralnem registru, ki vzdržuje evidenco dosegljivih komponent. Poleg odkrivanja storitev nekatere rešitve omogočajo tudi upravljanje konfiguracije, sinhronizacijo med storitvami in podobno. Naštejemo lahko Apache Zookeper, Consul, etcd in Netflix Eureko.

4.2.2 Orkestracija

Arhitektura mikrostoritev omogoča enostavno horizontalno skaliranje obremenjenih in odstranjevanje odvečnih komponent. Poleg tega obstaja v visoko razpoložljivih aplikacijah potreba po avtomatskem preklopu na rezervne strežnike v primeru napak v glavnem izvajalnem okolju, postopek se imenuje samodejni nadomestni način delovanja (ang. *automatic failover*). Taki sistemi so smiselni le, če se izvajajo na več strežnikih, ki se nahajajo na različnih lokacijah. Pri uporabi večjega števila strežnikov se pojavi tudi potreba po uravnavanju porabe računskih sredstev, kar se izvede s prestavljanjem (navideznih) izvajalnih okolij in zaustavljanjem strežnikov in virtualnih strojev, ki niso nujno potrebni. Vse to omogočijo orodja za gručenje (ang. *clustering*), orkestracijo (ang. *orchestration*) in nadzor (ang. *management*).

Najprej razložimo prej naštete pojme: gručenje je združevanje gostiteljev[5] preko omrežja v enoto, ki odjemalcem izpostavlja skupne (združene)

[5]Lahko gre za virtualne stroje ali normalne strežnike.

računalniške vire. Orkestracija obsega zagon vsebnikov na primernih gostiteljih, vzpostavljanje omrežij med vsebniki, pogosto orkestracijska orodja poskrbijo tudi za skaliranje, nadomestni način delovanja, prestavljanje vsebnikov med gostitelji za uravnoteženje obremenitve in drugo. Nadzor označuje vse operacije, ki nam omogočajo vpogled v stanje sistema in ostale administrativne operacije.

Med orodji, ki so namenjena gručenju in orkestraciji, lahko naštejemo: Docker Swarm, Kubernetes, fleet in Apache Mesos. Platforme bolj namenjene samo nadzoru pa so Docker Cloud, Rancher in Clocker.

Kot zanimivost predstavimo nekaj posebnosti orkestracijskega ogrodja Kubernetes:

- *Pod* je skupina vsebnikov, ki so zagnani istočasno in skupaj ponujajo neko storitev. Znotraj skupine se izvajajo še podporni vsebniki, npr. za beleženje in nadzorne operacije. Skupine so, podobno kot vsebniki, kratkožive in nadomestljive ter se med izvajanjem aplikacije pogosto ustavljajo, odstranjujejo in na novo zaganjajo. Menimo, da se *pod* zelo dobro prilega komponentam mikrostoritev, saj zajame posamezno komponento skupaj z njenimi unikatnimi odvisnostmi. Če na primer neka komponenta edina dostopa do določene podatkovne baze, je smiselno, da sta obe del iste skupine.

- Enotno omrežje (ang. *flat networking space*) se zelo razlikuje od obstoječih načinov mreženja vsebnikov, kot so npr. prekrivna omrežja Docker. Klasično se vsebniki nahajajo v ločenih omrežjih, za komunikacijo z vsebniki na drugih gostiteljih pa potrebujejo posebej vzpostavljene povezave in NAT (ang. *Network Address Translation*). V okviru ogrodja Kubernetes si vsebniki v skupini delijo naslov IP, do ostalih vsebnikov dostopajo preko naslova IP njihove skupine. Znotraj skupine komunicirajo preko odprtih omrežnih vrat na naslovu localhost, zato je pri izvajanju potrebna dodatna pozornost, da ti vsebniki svojih dostopnih točk ne izpostavljajo preko istih omrežnih vrat.

- Označbe (ang. *labels*) so atributi objektov, ponavadi skupin vsebnikov, preko katerih orodje izvaja izbrane orkestracijske in nadzorne operacije.

- Storitve (ang. *services*) so dostopne točke, naslovljive z imenom. S pomočjo označb (ali s fiksnim naslovom IP) so asociriane z eno ali več skupinami vsebnikov, tako ostalim skupinam omogočajo poenostavljeno interakcijo brez zavedanja o podrobnostih njene izvedbe. Na primer aplikacija znotraj ene skupine dostopa do podatkovne baze, ki je dejansko cela gruča vsebnikov, a zaradi abstrakcije, ki jo ponuja storitev, aplikaciji ni potrebo vedeti, točno s katero instanco komunicira, oziroma koliko vsebnikov sestavlja to storitev. Abstrakcija storitev omogoči tudi nemoteno dodajanje in menjavo skupin vsebnikov. Čeprav se ob takih operacijah skupini vsebnikov spremeni naslov IP, bo ta še vedno dostopna preko označbe oziroma fiksnega naslova IP. Dejansko gre za odkrivanje storitev, to smo podrobneje obravnavali v sekciji 4.2.1.

- Krmilniki replikacij (ang. *replication controller*) skrbijo za zagon in izvajanje skupin vsebnikov za določeno storitev. V primeru napak odstranijo problematične skupine in zaženejo nove, odstranijo odvečne skupine, če njihovo izvajanje ni smiselno, ter horizontalno skalirajo skupine v primeru povečane obremenitve.

Ker ogrodje Kubernetes operira s skupinami namesto s posameznimi vsebniki, je smiseln tudi drugačen pristop za hrambo podatkov. Skupinam vsebnikov so tako na voljo naslednje (navidezne) naprave za shranjevanje podatkov:

- *Emptydir* je prazna datotečna mapa, ki se ustvari in izbriše skupaj s skupino vsebnikov. Vsakemu vsebniku v skupini omogoča pisanje in branje in je najbolj primeren za izmenjavo začasnih podatkov.

- *Nfs* omogoča dostop do omrežnega datotečnega sistema (ang. *Network File System*, NFS), podatki v tem nosilcu se ohranijo tudi po uničenju skupine.

- *GcePersistentDisk* in *awsElasticBlockStore* sta namenjena shranjevanju podatkov v Googlovih in Amazonovih oblačnih storitvah, podatki se prav tako ohranijo tudi po uničenju skupine.
- *Secret* je namenjen hranjenju varnostno občutljivih podatkov, npr. gesel, certifikatov in varnostnih žetonov. Ta nosilec je izveden kot začasni datotečni sistem v bralno-pisalnem pomnilniku (ang. *Random Access Memory*) in ni nikoli zapisan na disk, podatki pa obstajajo samo za čas izvajanja skupine vsebnikov.

4.2.3 Mreženje

Ko se komponente mikrostoritve izvajajo na gruči strežnikov, za normalno komunikacijo nujno potrebujejo povezave do pravih komponent. Za klic prave komponente poskrbi orodje za odkrivanje storitev, povezavo pa mora (ponavadi) omogočiti druga rešitev. Če se vsebniki izvajajo na istem sistemu, je situacija preprosta, saj je dovolj navidezno omrežje. Mreženje postane zapleteno, ko povezujemo več strežnikov, ker moramo takrat med njimi zagotoviti omrežne poti, ki lahko potekajo bodisi preko javnega omrežja bodisi preko lokalnega omrežnega stikala. Izkaže se, da so za postavitev takega omrežja na gruči primerna orodja za odkrivanje storitev, orodja za ustvarjanje navideznih omrežij ali kombinacije obeh. Ponovno lahko omenimo orodja iz sekcije 4.2.1, poleg njih pa še rešitve, namenjene posebej mreženju: Weave, Flannel in Calico.

4.2.4 Ohranjanje stabilnosti

Če komponenta mikrostoritve ne deluje zanesljivo, lahko to negativno vpliva na stabilnost celotne aplikacije. Problem se pojavi predvsem pri uporabi sinhronih načinov komunikacije, ko mora tista komponenta, ki je pobudnica komunikacije, počakati na odgovor. Naštejemo lahko nekaj rešitev:

- Časovni iztek (ang. *timeout*) zahteve. Pogosto je smiselno, da po določenem času prenehamo čakati na odziv komponente. Edina težava

je v določanju primernih časovnih okvirov – če se zahteva izteče prekmalu, smo delujočo komponento smatrali kot nestabilno, če pa zahteva traja predolgo, to upočasni celoten sistem. Vsekakor je uporaba časovnih iztekov smiselna, saj se tako izognemo popolni blokadi celotne storitve.

- Varovalka (ang. *circuit breaker*) je arhitekturni vzorec, kjer pred problematično komponento postavimo posrednika, ki posreduje zahteve in spremlja odzive pripadajoče komponente. V primeru težav *circuit breaker* preneha s posredovanjem zahtev in počaka, da komponenta postane ponovno stabilna. Glede na okoliščine lahko med nedostopnostjo še vedno zbira zahteve in jih kasneje posreduje komponenti, lahko pa le vrne odziv o napaki. Prvi način je smiseln za asinhrone načine komunikacije, medtem ko je drugi bolj primeren za sinhrone. *Circuit breaker* je uporaben tudi pri nadomeščanju in nadgrajevanju komponent, saj za čas nedostopnosti komponente preprosto prekine posredovanje zahtev.

- Izolacija mehanizmov za komunikacijo s problematično enoto. Kot primer bi lahko navedli prej omenjene varovalke, smiselni so tudi drugi pristopi, npr. uporaba bazena povezav (ang. *connection pool*) za vsako komponento posebej.

Za stabilno delovanje storitve je priporočena uporaba vseh treh prej omenjenih pristopov.

4.2.5 Beleženje

Drugačen pristop je pri mikrostoritvah nujen tudi za uspešno beleženje (ang. *logging*) različnih dogodkov [6]. Če se vrnemo na monolitne aplikacije, je v teh beleženje preprostejše zaradi enotnosti programskega jezika in izvajalne platforme. Nasprotno so komponente mikrostoritve izvedene z različnimi ogrodji in programskimi jeziki, ki podatke o dogodkih zapisujejo na različne načine, zato je učinkovito beleženje težko, preprosta agregacija vseh zapisov v isti

format pa praktično nemogoča. Dodatne težave lahko povzroči kratkoživost vsebnikov, ki so na primer lahko ustavljeni in izbrisani preden nam uspe pridobiti zapise procesa, izvajanega v vsebniku.

Za namene beleženja je priporočena uporaba agregatorjev – namenskih programov, ki prevzamejo zapise posameznih vsebnikov, jim dodajo podatke o času (v formatu UTC), izvajalnem okolju ter vsebniku in nato vse skupaj zapišejo v primerno podatkovno bazo. Primeri takih orodij so Fluentd ter sklad ELK (Elacsticsearch, Logstash in Kibana).

4.2.6 Dokumentacija (vmesnikov) komponent

Pri razvoju storitve je smiselno, da so enote in njihovi javni vmesniki dobro dokumentirani. To razvijalcem mikrostoritve olajša razvoj, saj za uporabo ostalih komponent le pregledajo dokumentacijo. Če ta ne obstaja, so razvijalci primorani brati programsko kodo drugih komponent, ki so lahko izvedene v drugačni, bralcu potencialno neznani tehnologiji. Monolitne aplikacije imajo s tega vidika prednost, saj so napisane v istem programskem jeziku. Njihove komponente večinoma komunicirajo preko privzetih mehanizmov, kot so npr. klici javnih funkcij ostalih komponent. V okviru mikrostoritev se pojavi problem, ker za večino uporabljenih načinov komunikacije ni standardnih načinov dokumentacije. Lahko navedemo vmesnik XML *Web Services Definition Language* (WSDL), pogosto uporabljen za dokumentacijo dostopnih točk SOAP. Za mikrostoritve ni primeren, saj se namesto SOAP večinoma uporablja preprostejši način komunikacije REST, ki nima niti standardnega načina modeliranja dostopnih točk niti standardnega načina dokumentacije. Za namene dokumentacije dostopnih točk REST obstajajo mnoga (nestandardizirana) orodja, najbolj uporabljeni med njimi sta *Swagger* [18] in *RAML* [15], omenimo pa lahko tudi težnjo po enotnem formatu dokumentacije vmesnikov REST, OAI [13].

4.2.7 Tajni ključi

Pomembna vidika pri izvajanju aplikacije v arhitekturi mikrostoritev sta tudi avtentikacija in pooblastitev (ang. *authorization*). Prvi koncept je preverjanje istovetnosti pobudnika[6] neke akcije, drugi pomeni dovoljenje za izvedbo te akcije.

Če se omejimo na avtentikacijo uporabnika, se pri mikrostoritvah pojavi problem, ko dostopamo do več različnih komponent. Avtentikacija ob dostopu vsake komponente bi bila nesmiselna ter časovno potratna, zato so se uveljavile drugačne rešitve. Vsem je skupna značilnost, da se uporabniku ob vpisu dodeli nek atribut, preko katerega potem vse komponente ugotavljajo njegovo identiteto in pooblastitve. Pristop se imenuje enkratni vpis (ang. *single sign-on*, SSO) in je lahko izveden na dva različna načina. Pri prvem uporabnik od avtentikacijske komponente pridobi identiteto in nato sam dostopa do ostalih komponent, medtem ko pri drugem vse akcije izvaja preko vmesne enote, ki deluje kot prehod (ang. *gateway*). To pomeni, da enota dostopa do notranjih komponent in rezultate posreduje uporabniku.

Znotraj aplikacije je na nivoju komponent mikrostoritve situacija drugačna. Včasih je dovolj, če so zavarovane samo navzven dostopne točke, medtem ko vse notranje (podporne) storitve obratujejo brez avtentikacije. Med alternativami so še razne implementacije SSO, avtentikacija *HTTP Basic*, uporaba certifikatov, s katerimi komponenta preveri istovetnost druge komponente in podobno. Posebej lahko omenimo še ključe aplikacijskih programskih vmesnikov (ang. *API keys*), s katerimi je lahko znotraj sistema omogočeno merjenje in omejevanje uporabe točno določenih storitev.

Pomemben varnostni vidik je tudi način dostave ključev, gesel in certifikatov v vsebnike. Zapisovanje gesla med gradnjo vsebnika ni priporočeno, saj je to potem dostopno v vsebniški sliki. V nekaterih primerih je gesla smiselno podati preko okoljskih spremenljivk (ang. *environment variables*) ob zagonu storitve, čeprav potem obstaja možnost, da proces v vsebniku izpiše svoje okoljske spremenljivke in tako razkrije gesla. Bolj primeren pristop je zagon

[6] To je lahko uporabnik, program ali storitev.

s priloženim enkratnim geslom, s katerim potem storitev pridobi prijavne podatke od neke druge komponente, npr. od orodja za odkrivanje storitev ali od orodja za orkestracijo. Naslednji način je souporaba priklopljene logične enote, ki vsebuje relevantne certifikate ali datoteke s šifriranimi gesli. Tak pristop je otežen v primeru izvajanja mikrostoritve na gruči strežnikov, saj mora biti nosilec podatkov s certifikati dostopen na istem strežniku, kjer se izvaja dotični vsebnik, kar lahko povzroči dodatne nevšečnosti pri nameščanju certifikata, recimo varnostne pomisleke pri prenašanju certifikata preko omrežja ter probleme s sinhronizacijo, če ustvarimo nov certifikat.

4.2.8 Testiranje komponent

Mikrostoritve zaradi lastnosti porazdeljenega sistema privedejo do potrebe po dodatnih načinih testiranja komponent. Najprej naštejemo klasične načine testiranja, ki so smiselni tudi za monolitne aplikacije:

- Testiranje enot (ang. *unit*). Kratki testi, ki obravnavajo majhne izolirane dele funkcionalnosti enote. Najpogosteje so testirane funkcije oziroma metode.

- Testiranje komponent nadgradi testiranje enot tako, da upošteva kontekst aplikacije in stranske učinke, ki jih neka enota povzroči.

- *End-to-end* testi izvedejo akcije, ki vključujejo večino komponent, da preverijo delovanje celotnega sistema. Ker so to največji in časovno najbolj potratni testi, se ne izvajajo tako pogosto kot načina omenjena v prejšnjih alinejah.

Dodatno lahko omenimo testiranje integracije (ang. *integration testing*), torej preverjanje delovanja povezav in vmesnikov ostalih komponent, ki jih določena storitev uporablja. Za mikrostoritve je to toliko bolj pomembno, ker vmesniki razmejujejo komponente, ki so pogosto izvedene v različnih tehnologijah. Ravno obraten pristop, imenovan *consumer-driven contract*

testing, omogoča razvijalcem že dostopanih storitev razvoj in hkrati ohranjanje nespremenjenega programskega vmesnika, da bo ta še vedno primeren za uporabo.

4.2.9 Avtomatizacija infrastrukture

Da bi bila razvoj in izvajanje aplikacij v arhitekturi mikrostoritev sploh smiselna, je ključna avtomatizacija razvojnega, testnega in izvajalnega okolja. Omenimo lahko sprotno integracijo (ang. *continuous integration*), prakso prispevanja programske kode v repozitorije večkrat dnevno. Ob vsaki taki akciji se potem avtomatično izvedejo prevajanje programa ter bistveni testi. S sprotno integracijo se razvijalci izognejo zapletenemu zlivanju (ang. *merge*) večjih delov programske kode, poleg tega se z avtomatskim testiranjem napake odkrijejo hitreje.

4.3 Nadaljnji razvoj, izvajanje brez aplikacijskega strežnika

Vsebniki so ponudnikom oblačnih platform (IaaS, PaaS in SaaS) omogočili velike prihranke, saj za nudenje enako zmogljivih storitev vsebniške tehnologije povzročijo manj dodatnih stroškov. Prihranke se da v mikrostoritvi še povečati, če povsem ustavimo izvajanje komponent, takrat ko te nimajo dela. Včasih je tudi smiselno uporabljati storitve drugih ponudnikov, tako da čim bolj zmanjšamo obseg lastnih rešitev. Če izvajanje lastnih komponent prenesemo na ponudnika funkcije kot storitve (ang. *Function as a Service*, FaaS), kot je recimo storitev Amazon Lambda, se tak pristop imenuje arhitektura brez aplikacijskega strežnika (ang. *serverless architecture*). Z uporabo take arhitekture se izognemo zapleteni orkestraciji in nadzoru ter plačujemo samo za izvajanje komponente oziroma funkcije.

Poglavje 5

Orodje za izvajanje mikrostoritev v oblaku

V okviru diplomskega dela je bilo razvito orodje za izvajanje mikrostoritev, ki temelji na vsebnikih Docker. Orodje je ponujeno kot spletna storitev z vmesnikom REST, ki uporabniku omogoča:

- nalaganje datotek za gradnjo vsebnikov Docker,
- nalaganje nastavitvenih datotek *Docker Compose* za hkratno organizirano izvajanje več vsebnikov Docker,
- gradnjo, izvajanje, ustavljanje in odstranjevanje vsebnikov,
- skaliranje komponent v določenih primerih,
- dostop do izpostavljenih mikrostoritev.

5.1 Implementacija

Odločili smo se, da orodje razvijemo v programskem jeziku Go. Temu je botrovalo več razlogov:

- Programski jezik je nastal kot odgovor na kompleksnost programov, napisanih v objektno orientiranih jezikih, kot sta Java in C++. Sintaksa

jezika omeji število različnih načinov izvedbe algoritma in tako zmanjša število napak, razvoj pa je preprostejši. Implementacija orodja v drugih programskih jezikih (npr. Java, C) bi bila precej bolj zapletena.

- Večina programskih rešitev s področja vsebniških tehnologij je napisana v jeziku Go. Naštejemo lahko Docker, etcd, flannel, fleet, Kubernetes, Juju, LXD, itd. To pomeni tudi, da je na voljo mnogo knjižnic, ki nam olajšajo delo z vsebniki.

- Go ponuja obsežno standardno knjižnico, ki pogosto nudi dovolj funkcionalnosti, zunanje knjižnice pa so večinoma majhne in preproste za uporabo.

- Go je primeren za razvoj spletnih strežnikov, storitev in tudi bolj nizkonivojskih programov. Naše orodje spada v vsa omenjena področja.

Rezultat prevajanja v jeziku Go je ponavadi statično povezan (ang. *statically linked*) program. To pomeni, da za izvajanje ne potrebuje zunanjih knjižnic. Izvajalno okolje našega orodja je zato razmeroma preprosto – strežnik s poljubno distribucijo sistema GNU/Linux, ki ima zagnan proces Docker.

5.1.1 REST dostopne točke

Ker mora biti orodje dostopno preko omrežja, je upravljanje izvedeno preko vmesnika REST. Zaradi preprostosti izvedbe je uporabljena osnovna avtentikacija HTTP (ang. *basic access authentication*), uporabniško ime in geslo se torej ob vsaki transakciji preneseta v glavi zahtevka HTTP. V tabeli 5.1 so naštete glavne dostopne točke, ki jih izpostavlja orodje. Naslov je podan relativno, torej moramo za uporabo dostopne točke poznati tudi spletni naslov, kjer se orodje nahaja. Besedo znotraj zavitih oklepajev nadomestimo z imenom tistega elementa, ki ga želimo upravljati. Poizvedbi dodamo argumente v obliki ?ime_argumenta=vrednost&ime_drugega=vrednost2. Primer: če se storitev nahaja na naslovu http://www.ime_domene.si in želimo naložiti

arhivsko datoteko projekta z imenom *projekt1* ter argumentom ime=vrednost, bomo uporabili metodo POST na naslovu

`http://www.ime_domene.si/upload/projekt1?ime=vrednost.`

Če želimo *projekt1* zagnati, bomo izvedli zahtevek GET na naslovu

`http://www.ime_domene.si/up/storitev.`

Naslov	Metoda	Akcija
/register	GET	Omogoči kreiranje uporabniškega računa.
/services/{ID storitve}	Vse metode	Dostop do storitve, ki jo ponuja projekt. Podrobnosti so zapisane v podpoglavju 5.1.3.
/upload/{projekt}	POST	Omogoča nalaganje arhivske datoteke z navodili za izgradnjo projekta. Ta dostopna točka hkrati služi tudi za kreiranje novega projekta.
/info/{projekt}	GET	Izpiše določene informacije o zagnanih vsebnikih v projektu.
/build/{projekt}	GET	Izvede gradnjo vsebnikov v projektu.
/up/{projekt}	GET	Zagon vsebnikov v projektu.
/scale/{projekt}	GET	Skalira komponento projekta. Z argumentom poizvedbe „name“ podamo ime komponente, z argumentom „num“ pa število želenih instanc.
/stop/{projekt}	GET	Ustavi izvajanje vsebnikov v projektu.
/rm/{projekt}	GET	Odstrani vsebnike v projektu.

Tabela 5.1: REST dostopne točke in njihove akcije.

5.1.2 Register zagnanih storitev

Ena od funkcionalnosti orodja je pregled nad trenutno izvajanimi storitvami. Za to uporablja ločeno komponento, ki izpostavlja nekaj funkcij nad svojo podatkovno bazo. Omogoča dodajanje asociacij med uporabnikom, njegovim projektom in izpostavljeno storitvijo. Vsaka storitev prejme unikaten ID, s katerim je potem dostopna preko spletnega podnaslova /services/{ID storitve}. Ker gre za preprost projekt, so podatki registra shranjeni kar v pomnilniku, v zgoščeni tabeli (ang. *hash table*). Bolj obsežne rešitve bi verjetno uporabile podatkovno bazo oziroma eno od orodij za odkrivanje storitev, ki so omenjena v prejšnjem poglavju.

5.1.3 Posrednik

Orodje mora omogočati hkratno izvajanje več projektov. Lahko se pojavi problem, da več projektov poskusi izpostaviti storitve preko istih omrežnih vrat, kar pa ni izvedljivo. Ena izmed možnih rešitev je uporaba t.i. posrednika (ang. *reverse proxy*). Ponavadi gre za program, ki prejema omrežne zahtevke na določenem naslovu in glede na prejete podatke zahtevo posreduje določenemu strežniku, odgovor obdela in ga posreduje nazaj do pobudnika zahteve.

V izseku kode 1 je funkcija, ki vrne rutino (ang. *handler*) za posredovanje zahtevkov. V 22. in 23. vrstici kreiramo nov objekt ReverseProxy paketa http, podamo mu še funkcijo Director, ki modificira zahtevek HTTP (ponavadi spremeni pot do ciljnega strežnika, lahko pa tudi spremeni podatke v zaglavju zahtevka). Funkcija tipa Director, ki je definirana med 2. in 20. vrstico, najprej iz naslova zahtevka izlušči ID želene storitve in ciljno pot (ang. *target path*). Nato preko prej omenjenega registra pridobi omrežna vrata storitve, spremeni ciljni naslov zahtevka ter doda izluščeno pot.

```
1  func createProxy() http.Handler {
2    d := func(r *http.Request) {
3      // regex matches /services/{service-id}/{path}
4      regx, _ := regexp.Compile("^/services/(?P<v0>[^/]+)
           (?P<v1>/.*)?$")
5      pathVars := regx.FindStringSubmatch(r.URL.Path)
6
7      // get internal service port from the registry
8      port, err := reg.GetPort(pathVars[1])
9      if err != nil {
10       r.URL.Scheme = "http"
11       r.URL.Host = "localhost" + defaultPort
12       r.URL.Path = "/invalid"
13       return
14     }
15     portStr := fmt.Sprintf("%d", port)
16
17     r.URL.Scheme = "http"
18     r.URL.Host = "localhost:" + portStr
19     r.URL.Path = pathVars[2]
20   }
21
22   return &httputil.ReverseProxy{
23     Director: d,
24   }
25 }
```

Izsek programske kode 1: Programska logika posrednika.

5.2 Primer uporabe

V nadaljevanju je prikazan primer uporabe našega orodja. Pokazati želimo sledeča opravila:

- registracijo novega uporabniškega računa,
- kreiranje projekta in nalaganje datotek za izgradnjo projekta,
- izgradnjo projekta,
- zagon komponent projekta,
- skaliranje glavne aplikacije,
- dostop do skaliranih komponent,
- zaustavitev izvajanja projekta.

Za možnost skaliranja je priročno, če je znotraj projekta zagnan izenačevalnik obremenitve, ki samodejno opazi operacijo skaliranja in temu primerno prilagodi usmerjanje. V našem primeru bomo uporabljali že pripravljen vsebnik Docker s programom HAProxy [8]. Da bi pokazali efekt skaliranja, smo napisali majhen program v programskem jeziku Go, ki na omrežnih vratih 80 izpiše ime gostitelja[7] in zahtevano pot URL (ang. *URL path*). Programska koda je vidna v odseku 2.

[7]Ker bo program izvajan v vsebniku, bo to unikatno, navidezno ime gostitelja.

```
1  package main
2
3  import "fmt"
4  import "os"
5  import "net/http"
6
7  func main() {
8   http.HandleFunc("/", func(w http.ResponseWriter, req *
       http.Request) {
9    hostname, _ := os.Hostname()
10   fmt.Fprintf(w, "Hello␣from␣container␣%q!\n", hostname)
11   fmt.Fprintf(w, "Path:␣%s\n", req.URL.String())
12  })
13  http.ListenAndServe(":80", nil)
14 }
```

Izsek programske kode 2: Testni program.

Pokažemo še nastavitveno datoteko project.yml, ki je skladna z *Docker Compose*. Definira dve storitvi: prox, ki skrbi za usmerjanje in app, ki je prej omenjen program v jeziku Go. Z nastavitvijo links: -app vzpostavimo povezavo med komponentama.

```
> cat project.yml
version: '2'
services:
    prox:
        image: 'dockercloud/haproxy:latest'
        links:
            - app
        ports:
            - "80:80"
        volumes:
            - /var/run/docker.sock:/var/run/docker.sock
```

```
        environment:
            - BALANCE=roundrobin
    app:
        build:
            context: ./app
```

Zahtevke REST pošiljamo s programom curl [21], uporabili bi lahko tudi poljubni spletni brskalnik. Za razumevanje programa curl je ključna razlaga uporabljenih argumentov. Z zastavico -u nakažemo, da bo naslednji argument par uporabniškega imena in gesla, ki bo dodan v zaglavje zahtevka HTTP. Uporaben argument je tudi -F, s katerim sprožimo zahtevo POST, z naslednjim argumentom pa priložimo poljubne podatke. Primer uporabe tega argumenta za nalaganje datoteke datoteka.zip je -F "podatki=@datoteka.zip".

Najprej ustvarimo nov uporabniški račun.

```
> curl -u uporabnik:geslo \
        http://ime_domene.si/register
{'message':'Created user account for "uporabnik".'}
```

Recimo, da uporabnik želi izvajati preprost spletni projekt. Najprej ustvari arhiv *zip*, ki vsebuje navodila in datoteke za izgradnjo vsebnikov Docker ter nastavitveno datoteko project.yml. Ta vsebuje podatke o tem, kako zgraditi vsebnike, kako naj se izvajajo, kako povezati posamezne storitve in preko katerih omrežnih vrat bodo storitve dostopne. Datoteko projekt.zip pošlje storitvi in tako ustvari nov projekt.

```
> curl -u uporabnik:geslo \
        -F ''data=@projekt.zip'' \
        http://ime_domene.si/upload/mojprojekt
{'message':'Upload successfull,
  created project "mojprojekt"'}
```

Po uspešnem nalaganju projekta lahko uporabnik izvede gradnjo slik vsebnikov.

```
> curl -u uporabnik:geslo \
       http://ime_domene.si/build/mojprojekt
{'message':'Project "mojprojekt" build successful.'}
```

Nato lahko projekt zažene in v odgovoru pridobi identifikatorje, s katerimi lahko dostopa do javno izpostavljenih storitev projekta.

```
> curl -u uporabnik:geslo \
       http://ime_domene.si/up/mojprojekt
{'message':'Project "mojprojekt" services started.',
'services':[{'port':80,
'id':'5ec767b1-b8b5-4434-9dbe-cee993181635'}]}
```

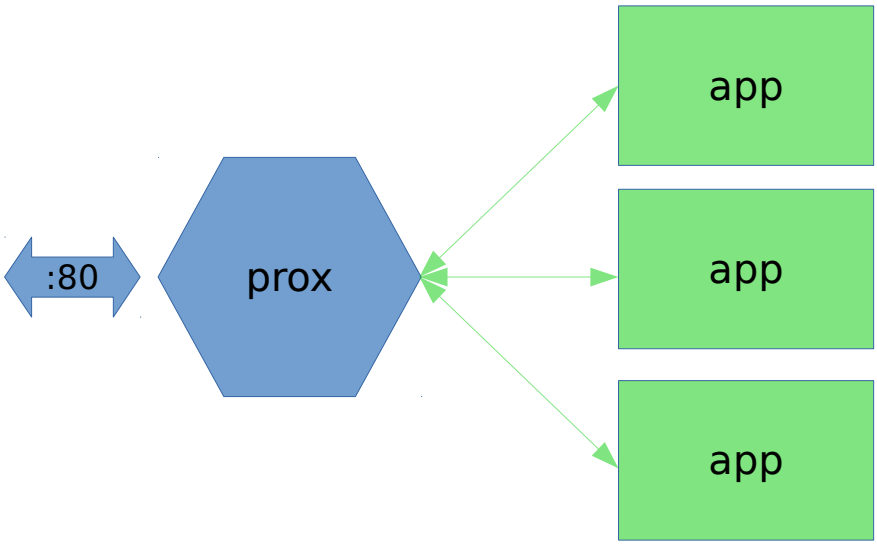
Iz odgovora je razvidno, da ima storitev, ki izpostavlja omrežna vrata 80, identifikator *5ec767b1-b8b5-4434-9dbe-cee993181635*, torej je glede na tabelo 5.1 dostopna na naslovu http://ime_domene.si/services/5ec767b1-b8b5-4434-9dbe-cee993181635.

```
> curl http://ime_domene.si/services/\
       5ec767b1-b8b5-4434-9dbe-cee993181635/\
       dodatna/pot?vrednost=1
Hello from container "e67b4e9efddc"!
Path: /dodatna/pot?vrednost=1
```

Nato izvedemo zahtevo za horizontalno skaliranje komponente app na tri instance.

```
> curl -u uporabnik:geslo \
       http://ime_domene.si/scale/\
       mojprojekt?name=app\&num=3
{'message':'Service "app" of project "mojprojekt"
  scaled to 3 instances.'}
```

Na sliki 5.1 je prikazan diagram izvajanih vsebnikov projekta po uspešni operaciji skaliranja. Izenačevalnik obremenitve prox je znotraj vsebnika do-



Slika 5.1: Pregled komponent projekta po operaciji skaliranja.

stopen na omrežnih vratih 80, zahtevke pa preusmerja trem instancam komponente app.

Po skaliranju še enkrat dostopamo do storitve. Izenačevalnik obremenitve zahtevek posreduje drugemu procesu z drugačnim imenom gostitelja.

```
> curl http://ime_domene.si/services/\
        5ec767b1-b8b5-4434-9dbe-cee993181635/\
        1/2/3
Hello from container "3d6cdef777a1"!
Path: /1/2/3
```

Nazadnje ustavimo izvajanje projekta.

```
> curl -u uporabnik:geslo \
        http://ime_domene.si/stop/mojprojekt
{'message':'Project "mojprojekt" stopped.'}
```

5.3 Primernost orodja za izvajanje mikrostoritev

Kot smo omenili že na začetku poglavja, je razvito orodje precej preprosto. Omogoča osnoven nadzor nad izvajanjem, povezavo komponent znotraj projekta ter horizontalno skaliranje izbranih komponent. Omogoča bistveno lastnost mikrostoritev – komunikacijo med komponentami.

5.3.1 Možne izboljšave

Da bi bilo orodje uporabno v praksi, bi morali dodati ali integrirati še kar nekaj funkcionalnosti, npr. gručenje, odkrivanje storitev, samostojno dodajanje in odvzemanje posameznih komponent, orkestracijske operacije ter ostale zmožnosti, omenjene v poglavju 4.2. Prav tako bi bilo zanimivo samodejno horizontalno skaliranje v primeru večje obremenitve posameznih komponent. Med nadgradnjami, ki bi izboljšale uporabniško izkušnjo, lahko naštejemo:

- Registracija in avtentikacija uporabnika preko ponudnika avtentikacijskega protokola (npr. OAuth) oziroma dostop do storitve z uporabo varnostnih žetonov.
- Spletni uporabniški vmesnik – trenutno je storitev dostopna le preko vmesnika REST.
- Potisna obvestila (ang. *push notification*) o stanju izvajanja uporabnikovega projekta.

Poglavje 6

Sklep

V diplomski nalogi smo obravnavali vsebniške tehnologije ter arhitekturo mikrostoritev. Razvili smo tudi storitev, ki uporablja vsebnike ter preko vmesnika REST omogoča izvajanje ter dostopanje do aplikacij, izvedenih v arhitekturi mikrostoritev.

V poglavju 2 smo predstavili virtualizacijo in vsebnike ter analizirali mehanizme v jedru Linux, ki omogočajo vsebniške tehnologije. V naslednjem poglavju smo natančneje pregledali zgradbo vsebnikov Docker in jih primerjali z ostalimi vsebniškimi tehnologijami. Ugotovili smo, da med izvedbami vsebnikov ni bistvenih razlik, se pa vsebniške tehnologije razlikujejo v načinih kreiranja novih vsebnikov, enostavnosti deljenja in uporabe vsebniških slik ter na področju varnosti. Razen na področju varnosti ima tehnologija Docker prednosti pred ostalimi. Menimo, da so prednosti na omenjenih področjih eni od ključnih razlogov za večjo priljubljenost Dockerja.

Zaradi hitrega zagona, avtomatizacije ter nižjih izvajalnih stroškov se z uporabo vsebnikov uveljavlja nov način zasnove visoko razpoložljivih spletnih aplikacij – arhitektura mikrostoritev. Obravnavali smo jo v poglavju 4. Najprej smo arhitekturo mikrostoritev primerjali z monolitno arhitekturo, nato smo pokazali vzporednice s filozofijo razvoja operacijskega sistema UNIX ter s storitveno orientiranimi arhitekturami. Drugi del poglavja je bil namenjen obravnavi pomembnih vidikov izvajanja mikrostoritev ter kakšno vlogo imajo

pri tem vsebniške tehnologije.

V poglavju 5 smo predstavili delovanje in nekaj podrobnosti implementacije lastnega orodja, s katerim smo želeli pokazati primernost vsebniških tehnologij za izvajanje mikrostoritev. Gre za storitev REST, ki uporabnikom omogoča ustvarjanje in zaganjanje projektov, horizontalno skaliranje komponent ter dostop do izpostavljenih komponent projekta.

Čeprav vsebniške tehnologije to poenostavijo, je izvajanje aplikacij izvedenih v arhitekturi mikrostoritev še vedno zelo kompleksno. To se odraža v velikem številu dodatnih funkcionalnosti, ki bi jih naše orodje potrebovalo za resno uporabo. Na podlagi tega sklepamo, da je za izvajanje mikrostoritev namesto enega kompleksnega programa bolj primerna integracija več preprostih namenskih orodij, kjer posamezno orodje poskrbi za zanesljivo delovanje točno določene funkcionalnosti.

Literatura

[1] App container specification. Dosegljivo: https://github.com/appc/spec/blob/master/SPEC.md. [Dostopano: 23. 07. 2016].

[2] Basics of unix philosophy. Dosegljivo: http://www.catb.org/esr/writings/taoup/html/ch01s06.html. [Dostopano: 24. 07. 2016].

[3] Cgroups. Dosegljivo: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt. [Dostopano: 24. 07. 2016].

[4] Chapter 14. jails. Dosegljivo: http://www.freebsd.org/doc/handbook/jails.html. [Dostopano: 21. 07. 2016].

[5] Cve-2014-9357. Dosegljivo: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-9357. [Dostopano: 11. 08. 2016].

[6] Distributed logging architecture in the container era. Dosegljivo: https://blog.treasuredata.com/blog/2016/08/03/distributed-logging-architecture-in-the-container-era/. [Dostopano: 04. 08. 2016].

[7] Docker hub - explore official repositories. Dosegljivo: https://hub.docker.com/explore/. [Dostopano: 05. 09. 2016].

[8] dockercloud/haproxy. Dosegljivo: https://github.com/docker/dockercloud-haproxy. [Dostopano: 27. 07. 2016].

[9] Facts about docker adoption. Dosegljivo: https://www.datadoghq.com/docker-adoption/. [Dostopano: 12. 08. 2016].

[10] Linux containers. Dosegljivo: https://linuxcontainers.org/. [Dostopano: 23. 07. 2016].

[11] Linux security module framework. Dosegljivo: https://www.kernel.org/doc/Documentation/security/LSM.txt. [Dostopano: 12. 08. 2016].

[12] Microservices. Dosegljivo: http://martinfowler.com/articles/microservices.html. [Dostopano: 07. 04. 2016].

[13] Open api initiative. Dosegljivo: https://openapis.org/. [Dostopano: 12. 08. 2016].

[14] Open container initiative. Dosegljivo: https://www.opencontainers.org/faq. [Dostopano: 10. 08. 2016].

[15] raml.org. Dosegljivo: http://raml.org/. [Dostopano: 02. 09. 2016].

[16] Rethinking pid 1. Dosegljivo: http://0pointer.net/blog/projects/systemd.html. [Dostopano: 11. 08. 2016].

[17] Secure computing with filters. Dosegljivo: https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt. [Dostopano: 23. 07. 2016].

[18] swagger.io. Dosegljivo: http://swagger.io/. [Dostopano: 12. 08. 2016].

[19] systemd system and service manager. Dosegljivo: https://www.freedesktop.org/wiki/Software/systemd/. [Dostopano: 12. 08. 2016].

[20] What is docker? Dosegljivo: https://www.docker.com/what-docker. [Dostopano: 13. 07. 2016].

[21] *curl(1) Curl Manual*, November 2014.

[22] *namespaces(7) Linux Programmer's Manual*, September 2014.

[23] *seccomp(2) Linux Programmer's Manual*, December 2015.

[24] *chroot(2) Linux Programmer's Manual*, March 2016.

[25] *lxc(7)*, June 2016.

[26] Viktor Farcic. *The DevOps 2.0 Toolkit: Automating the Continuous Deployment Pipeline with Containerized Microservices*. 2016.

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

[28] Sean P. Kane Karl Matthias. *Docker: Up and Running*. O'Reilly Media, 2015.

[29] Adrian Mouat. *Using Docker: Developing and Deploying Software with Containers*. O'Reilly Media, 2016.

[30] Sam Newman. *Building Microservices*. O'Reilly Media, 1 edition, 2015.

[31] Dennis M. Ritchie and Ken Thompson. The unix time-sharing system. *Commun. ACM*, 17(7):365–375, July 1974.

[32] Kyoung-Taek Seo, Hyun-Seo Hwang, Il-Young Moon, Oh-Young Kwon, and Byeong-Jun Kim. Performance comparison analysis of linux container and virtual machine for building cloud. *Advanced Science and Technology Letters*, 66:105–111, 2014.

[33] C. P. Wright and E. Zadok. Unionfs: Bringing File Systems Together. *Linux Journal*, 2004(128):24–29, December 2004.