Korištenje Docker kontejnerske platforme za implementaciju IoT aplikacija

Jurak, Robert

VISOKO UČILIŠTE ALGEBRA

DIPLOMSKI RAD

Korištenje Docker kontejnerske platforme za implementaciju IoT aplikacija

Robert Jurak

Zagreb, rujan 2019.

Predgovor

Zahvaljujem mentoru mag. ing. el. Vedranu Dakiću na iskazanom povjerenju, vodstvu i vremenu koji je izdvojio za prihvaćanje još jednog diplomskog rada. Također zahvaljujem Visokoj školi za primijenjeno računarstvo, svim djelatnicima, profesorima i asistentima na suradnji, ugodnom studiranju i stečenim znanjima.

Prilikom uvezivanja rada, Umjesto ove stranice ne zaboravite umetnuti original potvrde o prihvaćanju teme diplomskog rada kojeg ste preuzeli u studentskoj referadi

Sažetak

Tema ovog rada fokusira se na implementaciju IoT-a (Internet of Things) pomoću Docker kontejnera i na posljedice takvog pristupa. Važnost IoT-a raste globalno bez znakova stajanja, sa 7 milijardi uređaja krajem 2018. godine taj broj mogao bi porasti na 10 milijardi do kraja 2020. Time se postavlja pitanje kako jednostavno i efikasno upravljati velikim brojem takvih uređaja. Docker i kontejnerske tehnologije općenito predstavljaju zanimljivo rješenje koje još uvijek nije rašireno. U radu se prezentiraju jedinstvena rješenja kontejnerskih tehnologija s kojima se mogu riješiti neki od glavnih IoT problema, uz objašnjenja vezana za implementaciju i rezultate koje možemo očekivati njihovom primjenom.

Ključne riječi: IoT, Docker, kontejneri.

Abstract

The theme of this paper is the implementation of IoT devices through the use of Docker containers and the consequences of such an approach. IoT importance is growing on a global scale without showing any signs of slowing down, from 7 billion devices in 2018. the number could grow to 10 billion by the end of 2020. With that we have to ask the question how to simply and efficiently manage such a number of devices. Docker and container technologies in general offer an interesting solution that is not yet widespread. The results of this paper include the benefits of container technologies towards solving some of IoT-s biggest problems, while explaining how to implement the technology and showing us the results we can expect from using it.

Key words: IoT, Docker, containers.

Sadržaj

1. Uvod

IoT tržište raste zastrašujućom brzinom i pronalazi sve veću ulogu u optimizaciji poslovanja i omogućavanju novih poslovnih modela. Unatoč tome što se ulaganja u IoT na godišnjoj razini već sada mjere u milijardama dolara, još uvijek postoje mnogi izazovi kod distribucije softvera IoT uređajima. Neki od tih izazova uključuju minimalne dostupne hardverske resurse, geografski odvojene uređaje, ograničen mrežni pristup i različita okruženja u kojima se uređaji nalaze.

Mnogi od tih problema mogu se adresirati tako da te aplikacije stavimo u visoko fleksibilne, apstraktne kontejnere na koje ne utječe okolina u kojoj ih pokrećemo. Unatoč velikom potencijalu, korištenje ovakvog pristupa još uvijek nije široko rasprostranjeno. Interes za kontejnerske tehnologije među IoT developerima je, uz rijetke iznimke, još uvijek iznenađujuće malen.

Cilj ovog rada je poboljšanje razumijevanja kontejnerskog pristupa IoT-u te proučavanje održivosti i isplativosti ovakvog rješenja za implementaciju istog.

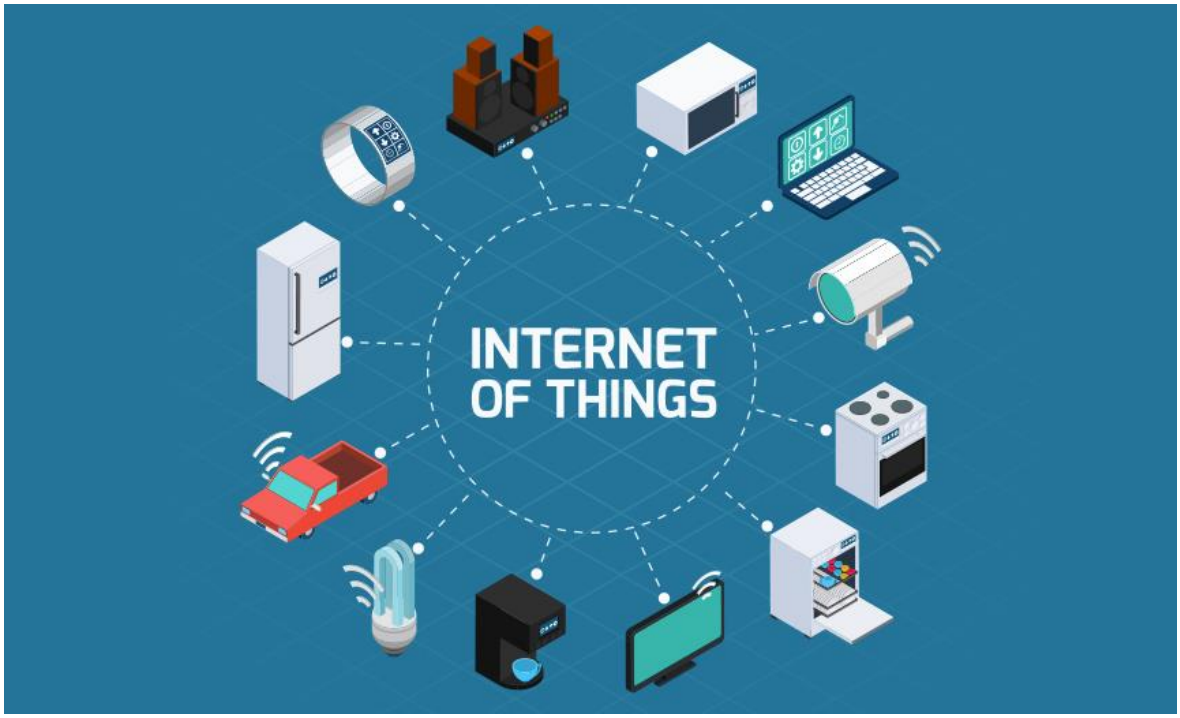
Pomoću Docker alata u radu će se obuhvatiti proces implementacije, zahtjevi, prednosti i nedostaci kontejnerizacije aplikacija u usporedbi s klasičnim rješenjima, uz demonstracije i praktične primjere.

2. IoT – Internet of Things

2.1. Što je IoT

Internet of Things (Internet Stvari) je računalni koncept gdje sve stvari, uključujući svaki fizički objekt, mogu biti povezane na internet, čime ti objekti postaju inteligenti, programabilni i u stanju komunicirati s ljudima prikupljanjem i razmjenom podataka[1].

U većini slučajeva takvi uređaji su jednostavni i jeftini, sami po sebi nisu ništa posebno. Povezivanjem tih uređaja međusobno i preko interneta na računala koja su u stanju obraditi te podatke činimo ih pametnima. Ukratko, cilj je proširiti utjecaj interneta izvan opsega klasičnih računala i mobitela kao što je prikazuje Slika 2.1. To znači da svaki uređaj možemo daljinski nadzirati i njime upravljati što je ogroman skok unaprijed kod automatizacije velikog broja svakodnevnih procesa.



Slika 2.1 Ilustracija kategorija IoT uređaja[2]

[1]https://www.ieee.org/membership-catalog/productdetail/showProductDetailPage.html?product=CMYIOT736
[2] Izvor: https://www.bankinfosecurity.com/gao-assesses-iot-cybersecurity-other-risks-a-9926

2.2. Tehnologije koje su omogućile IoT

2.2.1. Napredak u razvoju procesora

Smanjenjem potrošnje energije, cijene i veličine procesora opće namjene došlo je do njihovog korištenja u ugrađenim sustavima. Tu ulogu su prije imali mikrokontroleri koji su bili specijalno dizajnirani za što efikasnije izvođenje određenih zadataka. Ti mikrokontroleri bili su ograničeni činjenicom da ne mogu pokretati cijele operativne sustave. Prelaskom na procesore opće namjene to ograničenje nestaje te donedavno jednostavan uređaj dobiva podršku za mrežne protokole i sve popularne programske jezike.

2.2.2. Razvoj „laganih“ operativnih sustava i protokola

Vezujući se na razvoj procesora, Windows i Linux kao najrašireniji operativni sustavi već duže vrijeme nisu postavljali nove hardverske zahtjeve za njihovo korištenje. Dogodila se upravo suprotna stvar, počele su se razvijati jednostavnije verzije operativnih sustava u svrhu poboljšanja performansi i postizanja kompatibilnosti s uređajima koji inače ne bi pružali zadovoljavajuće iskustvo.

Slična stvar počela se događati kod razvoja komunikacijskih protokola. Kreirane su jednostavnije verzije TCP/IP stoga i novi protokoli namijenjeni za okruženja gdje su procesorska snaga i komunikacije ograničeni. Bežična komunikacija je također doživjela znatan napredak. WiFi je postao brži svakom iteracijom. Bluetooth niske energije omogućio je značajno smanjenje potrošnje energije i troškova, toliko da jedna baterija može trajati od nekoliko mjeseci do nekoliko godina. Također je došlo do pojave novih niskoenergetskih protokola kao što su: Z-Wave, Zigbee i sl.

2.2.3. Cloud computing

Konvergencijom razvoja procesora i softvera došlo je do toga da su uređaji za minimalan dodatan trošak mogli slati podatke i biti kontrolirani preko interneta.

Posljedica su bile velike količine podataka koje su se morale prikupiti i analizirati da bi se iz njih mogao izvući neki smisao. Cloud computing platforme postale su savršeno rješenje za taj problem. Uz visoku dostupnost, propusnost, skalabilnost i potrebne alate olakšane su implementacije IoT projekata. Velike platforme kao što su Microsoft Azure IoT Suite i Amazon IoT postali su poveznica između uređaja od korisnika i njihovih aplikacija koje analiziraju i prikazuju prikupljene podatke.

2.3. Važnost IoT-a – „Podaci su nova nafta“

Izreka „Podaci su nova nafta“ ne odnosi se na to da će podaci zamijeniti naftu kao resurs. Naftu i podatke uspoređujemo na osnovi toga da je nafta u jednom trenutku počela dominirati svim aspektima naših života. Od automobila, plastike, proizvodnje energije pa čak i umjetnog gnojiva.

Slična pojava očekuje se kod upotrebe podataka. Ljudski faktor predstavlja ograničenje efikasnosti bilo kojeg procesa. Cilj prikupljanja i analize podataka je ukidanje tog ograničenja na način da se kroz IoT postignu automatizirana povećanja efikasnosti, prihoda i postizanje ušteda.

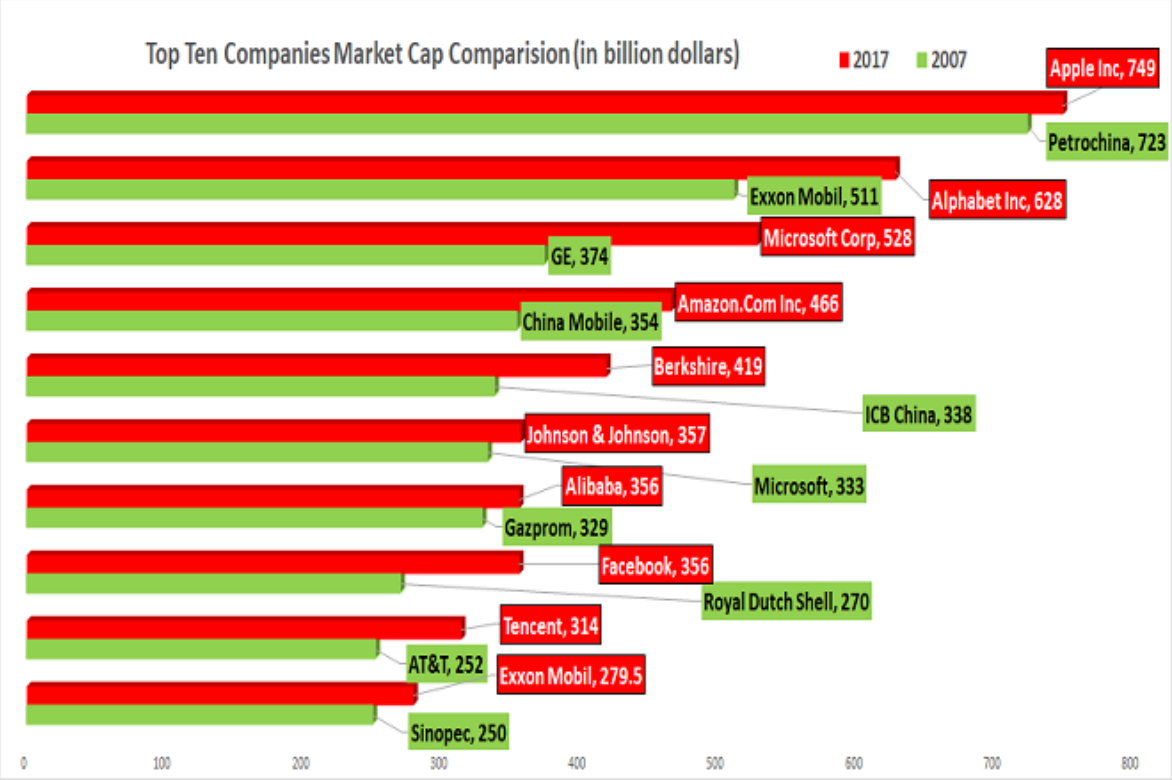
Danas postoje već mnogi primjeri korištenja IoT-a u poslovanju. Poljoprivrednici koriste senzore vlage kako bi usjeve zalijevali točnom količinom vode. UPS dostavna služba koristi IoT senzore na dostavnim vozilima za prikupljanje podataka o brzini, potrošnji, zdravlju motora i sl. Tvrtka Disney preko IoT narukvica prati aktivnosti svih posjetitelja u svrhu optimizacije i regulacije poslovanja.

Rasprostranjeno korištenje IoT-a također se može očekivati u sljedećim granama poslovanja:

- Proizvodnja
- Prijevoz
- Agrikultura
- Logistika

- Osiguranje
- Bankarstvo
- Pametne zgrade
- Komunalne usluge
- Zdravstvo

Ukratko, važnost podataka raste zajedno sa ulaganjima. Slika 2.2 prikazuje kakav se pomak dogodio u razmaku od 10 godina, energetske tvrtke koje su 2007 bile najvrjednije na tržištu, 2017. su pretečene u tržišnoj vrijednosti od strane tehnoloških kompanija od kojih se mnoge djelomično ili u potpunosti oslanjaju na prikupljanje i obradu podataka za ostvarivanje prihoda.



Slika 2.2 Energetske kompanije za razliku od tehnoloških gube na vrijednosti[3]

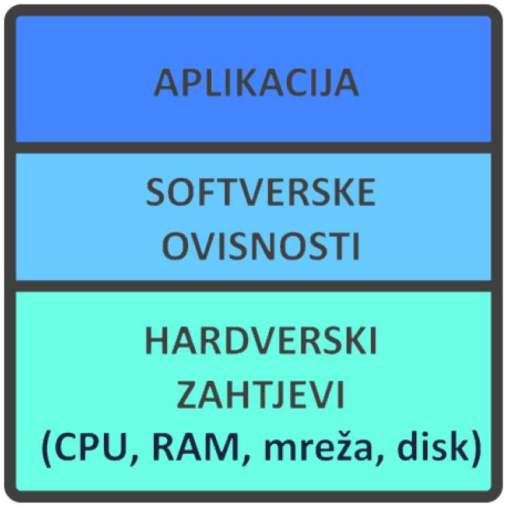
[3]Izvor:https://www.financialexpress.com/market/mukesh-ambani-is-right-after-all-data-is-indeed-the-new-oil-here-is-evidence/885798/

3. Kontejneri

3.1. Što su kontejneri

Kada pričamo o kontejnerima (engl. *containers*) u kontekstu aplikacija i softvera možemo ih usporediti sa stvarnim kontejnerima za otpremu robe. Prije njihovog izuma i standardizacije slanje robe bio je zahtjevan i skupocjen proces. Ovisno o vrsti robe i metodi slanja bilo je potrebno prilagođavati način pakiranja. Standardizacijom kontejnera izjednačen je način slanja robe neovisno o vrsti, obliku, veličini i načinu prijevoza – brodom, vlakom ili kamionom. Količina posla je smanjena uz pojednostavljenje procesa a postignute su i znatne vremenske i cjenovne uštede[4].

Sličan princip leži iza softverskih kontejnera. U kontejnere pakiramo samo ono što nam je potrebno da pokrenemo aplikaciju. Taj kontejner zatim možemo prenositi i pokretati bilo gdje. Umjesto pakiranja cijelog operativnog sustava i svog popratnog softvera, softverski kontejner najčešće sadrži samo sljedeće komponente: samu aplikaciju, potrebne biblioteke, komponente o kojima aplikacija ovisi i konfiguracijske datoteke (Slika 3.1). Aplikacija tako postaje neovisna o vrsti distribucije operativnog sustava i infrastrukturi na kojoj ju pokrećemo.



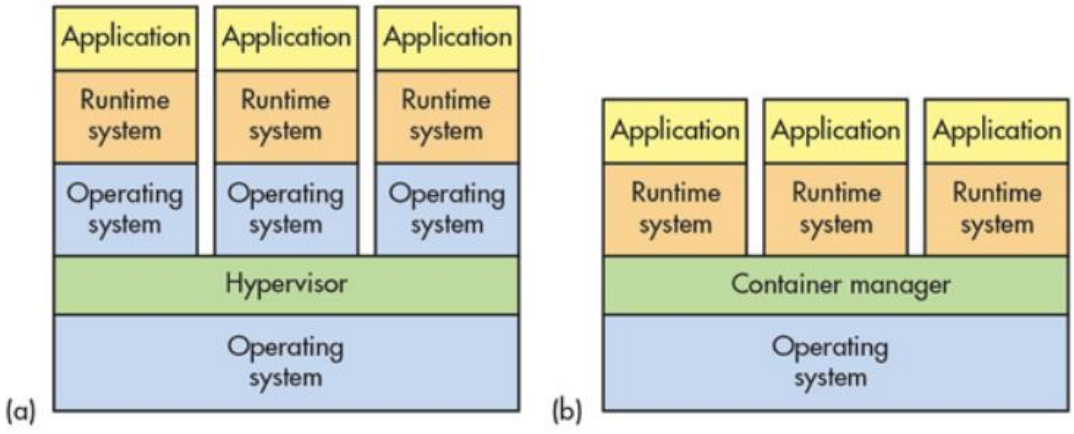
Slika 3.1 Arhitektura kontejnera

[4] JANGLA, S. Accelerating Development Velocity Using Docker

3.2. Usporedba sa virtualnim strojevima

Kontejneri i virtualni strojevi (engl. *virtual machines*, skraćeno VM) zapravo su dva načina implementacije više izoliranih servisa na istoj platformi. Virtualni strojevi pojavili su se kao odgovor na rastuće performanse servera koje standardne aplikacije nisu bile u stanju u potpunosti iskoristiti. Specijaliziran softver zvan hipervizor (engl. *hypervisor*) emulira hardverske resurse servera u potpunosti što omogućava dijeljenje istih između većeg broja VM-ova. Zbog toga je svaki VM u stanju pokretati bilo koju vrstu operativnog sustava (engl. *operating system*, skraćeno OS) ovisno o potrebi, sa skoro svim mogućnostima operativnog sustava kao da je instaliran na klasičnom računalu. Softver u VM-u tako može biti prenijet na drugi server i unatoč različitoj okolini još uvijek raditi pouzdano.

Između dva pristupa postoji nekoliko glavnih razlika. Kontejnerski sustav treba podložni operativni sustav samo za osnovne servise te je dijeljen između svih kontejnera. On također pruža podršku za virtualnu memoriju kako bi se kontejneri mogli izolirati. Hipervizor s druge strane u svakom VM-u vrti zaseban operativni sustav koji također zahtjeva virtualnu kopiju svog potrebnog hardvera od strane operativnog sustava. Usporedba VM i kontejnerske arhitekture prikazana je na Slika 3.2



Slika 3.2 Usporedba arhitekture virtualnog stroja i kontejnera

3.3. Prednosti kontejnera

Kao što smo vidjeli u prijašnjoj usporedbi, glavna prednost kontejnera su efikasnost korištenja resursa i fleksibilnost. Nedostatak operativnog sustava tako smanjuje zauzeće prostora za pohranu s nekoliko gigabajta na nekoliko desetaka ili stotina megabajta. Iz toga slijedi da automatski možemo pokrenuti veći broj kontejnera u odnosu da koristimo virtualizaciju. Uz to su smanjeni procesorski zahtjevi i vrijeme potrebno da pokrenemo kontejner. Procesi koji su pokrenuti na host operativnom sustavu ne moraju se duplicirati unutar kontejnera u kontrastu s virtualnim strojem kojemu treba cijeli operativni sustav za svaku instancu.

Fleksibilnost proizlazi iz toga da su sve datoteke potrebne za pokretanje aplikacije sadržane u kontejneru. Kontejner može imati zasebna mrežna sučelja koja se mogu razlikovati od onih na *host-u* što bi inače moglo dovesti do konflikata pri korištenju pojedinih brojeva portova. Kao posljedica prenosivosti također je olakšano testiranje i praćenje *bug-ova* jer se isti kontejneri koriste kod testiranja i u produkciji.

Resursi se kontejneru ne moraju dodijeliti prije pokretanja kao kod virtualnog stroja. Njihova dodjela se čak može i ograničiti pojedinim kontejnerima pri čemu kontejner neće zauzeti sve dostupne resurse ako mu to nije potrebno. Efikasnost u startu se tako može još više poboljšati.

3.4. Izazovi pri korištenju kontejnera

Kontejnere možemo smatrati minimalnim verzija virtualnih strojeva tj. oni sadrže samo ono što je apsolutno potrebno za pokretanje željene aplikacije. Iz toga bi se moglo zaključiti da je također jednostavnije pobrinuti se za sigurnost istih, ali to nije slučaj. Zbog prirode kontejnera nisu nam dostupni već utvrđeni sigurnosni alati ni sigurnosni protokoli s kojima smo već upoznati. Uz to, kontejneri dijele kernel, program koji je jezgra operativnog sustava na kojem su pokrenuti. To znači da su manje izolirani od virtualnih strojeva. Bilo koja ranjivost kernela operativnog sustava tako može ugroziti sigurnost svih kontejnera.

Izazov također predstavlja činjenica da kontejnere s različitim operativnim sustavima ne možemo pokrenuti na istom serveru što može biti veliko ograničenje kod kompleksnijih

implementacija. Osiguravanje mrežne povezivosti nije jednostavan zadatak ako kontejnere želimo dostatno međusobno izolirati. Dodatan problem predstavlja činjenica da na način koji su dizajnirani, kontejneri se prema zadanim postavkama ne mogu međusobno vidjeti i komunicirati.

Unatoč izazovima, Docker alat je u stanju adresirati neke od tih problema i pružati dodatne značajke u odnosu na slična rješenja što će biti detaljnije obrađeno u poglavlju 4.

3.5. Zašto koristiti kontejnere za implementaciju IoT aplikacija?

IoT aplikacije imaju dvije glavne karakteristike: potencijal za skaliranje ogromnih razmjera i potencijal za kreiranje ekstremnih količina podataka. Kombinacija te dvije karakteristike lagano može dovesti do toga da u budućnosti dobijemo sustav koji je u stanju proizvesti stotine petabajta podataka. Ovakav sustav izvan je granica skaliranja s kakvim smo do sada upoznati, kao što su na primjer web aplikacije. Kako bi se mogli nositi s takvim sustavima potrebna je moderna mikroservisna arhitektura čiju realizaciju omogućuju upravo kontejneri.

IoT aplikacije također se suočavaju s nekoliko problema koji se mogu adresirati ili ublažiti njihovom kontejnerizacijom.

Mnogim IoT uređajima naime nedostaje procesorska snaga i memorija, zbog toga su njihove mogućnosti da procesiraju softverske nadogradnje ograničene. U slučaju kontejnera, kada želimo instalirati neku nadogradnju, dovoljno je postaviti novi *image* i ukloniti stari bez ikakve potrebe da se nadogradnja procesira.

Ograničenja vezana za geografsku distribuciju IoT uređaja i ograničeni ili povremeni mrežni pristup mogu znatno otežati distribuciju softvera na takve uređaje. Kontejnerske platforme pružaju mogućnost korištenje više manjih lokaliziranih repozitorija za bolju pokrivenost i korištenje delta nadogradnji softvera kojima se nadograđuje samo onaj dio koda u kontejneru koji se promijenio.

Fragmentacija softvera koji se koristi u IoT uređajima predstavlja prepreku za distribuciju aplikacija koje je potrebno prilagođavati za svaku vrstu Linux distribucije kako bi se osigurala potpuna funkcionalnost. Korištenjem kontejnera verzije operativnih sustava i druge softverske varijable gube na važnosti čime bilo kakve specifične konfiguracije postaju nepotrebne.

4. Docker

Docker je softver koji je moguće koristiti na Linux i Windows operativnim sustavima. To je alat dizajniran za kreiranje, implementaciju i pokretanje aplikacija koristeći kontejnere. Prvenstvena namjena je za programere koji za svoj posao mogu odabrati željenu platformu te na njoj programirati bez brige o operativnom sustavu na kojem će se njihova aplikacija pokrenuti. Mi ćemo u ovom radu detaljnije proučiti njegove primjene u IoT-u, prvenstveno implementaciju i administraciju IoT uređaja.

4.1. Zašto Docker?

Kontejnerizacija aplikacija nije novost, ideja postoji već nekoliko desetljeća. Docker kao softver pojavio se 2014 godine, a izgrađen je na osnovi Linux kontejnera predstavljenih 2008. godine. Docker također nije jedini pružatelj ovakvih usluga. Uz ove informacije postavlja se pitanje zašto je izabran baš Docker.

Kao prvo, Docker je najpopularnija platforma što se tiče kontejnera, zauzimajući 83% tržišta u 2018. godini. Najviše korisnika će se susresti upravo s ovom platformom kada dođu u doticaj s kontejnerima. Posljedica toga je veća dostupnost materijala za edukaciju korisnika što uvelike olakšava korištenje platforme.

Iako se konkurentski proizvodi neprestano razvijaju uz tržište koje polako sazrijeva Docker još uvijek pruža najviše mogućnosti u odnosu na konkurentske proizvode.

U Tablici 4.1 vidimo usporedbu Dockera s konkurentima iz čega se jasno vidi da je Docker najzrelija platforma koja nudi najveću fleksibilnost i najviše značajki.

Tablica 4.1 Usporedba kontejnerskih platformi

Svojstvo/Kont. tehnologija	Docker	CoreOS rkt	LXC Linux	Open VZ
Podržane Platforme	Linux, Windows	Linux	Linux	Linux
Limitacija resursa kontejnera	Da	Djelomično	Djelomično	Djelomično
Prenosivost i Live migracija	Da	Da	Ograničeno	Da
Nested virtualizacija	Da	Djelomično	Da	Djelomično
Upravljanje udaljenim pristupom	Da	Ne	Ne	Da
Ineroperabilnost i standardizacija	Da	Ne	Ne	Ne

4.2. Arhitektura

Prije nego uđemo u korištenje Dockera moramo steći osnovno poznavanje o njegovoj arhitekturi i kako različite komponente međusobno djeluju.

Glavne Docker komponente su:

- Docker platforma
- Docker engine
- Docker arhitektura
  - Docker klijent
  - Docker daemon
  - Docker registri
- Docker objekti
  - Images

    - Kontejneri
    - Servisi
- Docker Hub

4.2.1. Docker platforma

Kako bi se aplikacije mogle pakirati u izolirana okruženja Docker pruža platformu pomoću koje možemo objediniti sve ovisnosti (engl. *dependencies*) i ostale komponente kao što su varijable okruženja, konfiguracijske datoteke, postavke itd. u jednu cjelinu.

Zahvaljujući tome što su izolirane, takve cjeline ne utječu jedna na drugu pa tako možemo imati različite kontejnere koji su istovremeno pokrenuti na *hostu.* Ti kontejneri pokrenuti su direktno na kernelu *host* računala što štedi resurse, a kao posljedica toga čak je moguće pokrenuti Docker kontejnere unutar virtualnih strojeva.

Docker platforma je zapravo skup alata koji omogućavaju upravljanje životnim ciklusom kontejnera. Pokriveni su:

- Razvoj aplikacija i svih popratnih komponenti
- Priprema za testiranje i distribuciju pomoću kontejnera
- Implementacija aplikacije u produkciju, bilo da je produkcijsko okruženje lokalni server, cloud ili hibridno okruženje

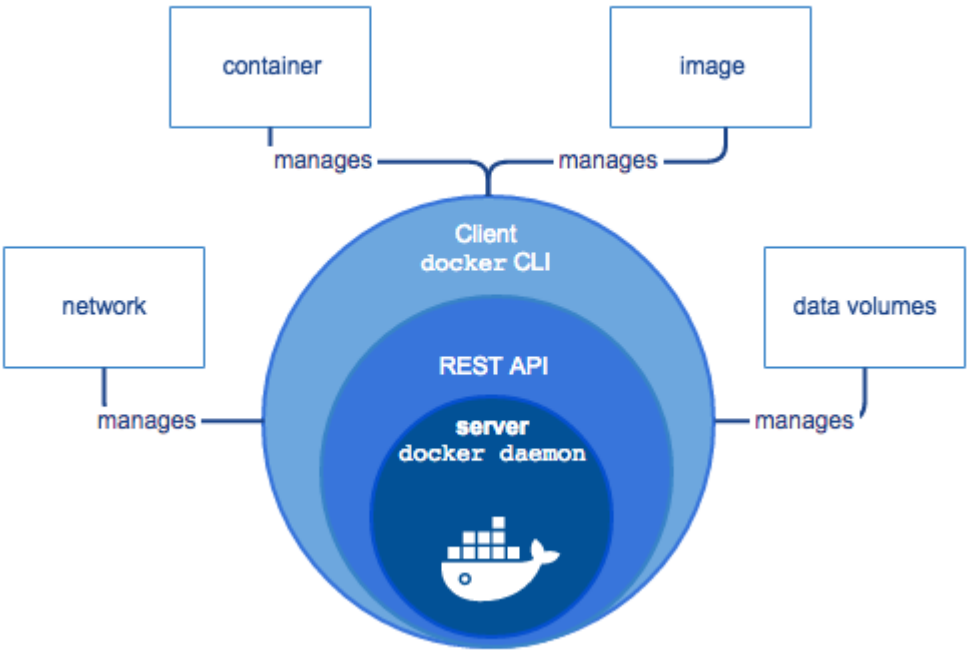
4.2.2. Docker Engine

Docker Engine je klijent-server aplikacija sa sljedećim komponentama[5]:

- Server koji je vrsta dugotrajnog procesa zvanog *daemon.* Taj proces vrti se u pozadini i odgovara na zahtjeve koji su mu upućeni. Pokreće se naredbom dockerd
- REST (engl. *Representational State Transfer)* API (engl. *Application Programming Interface*) specificira sučelja koja programi koriste da bi mogli pričati s *daemonom* i slati mu instrukcije.
- Komandno sučelje (engl. *Command line interface,* skraćeno CLI)

[5] JANGLA, S. Accelerating Development Velocity Using Docker, 2018

Komandno sučelje koristi Docker REST API za kontrolu i interakcije s Docker *daemonom* kroz skripte ili izravne CLI komande. *Daemon* kreira i upravlja s Docker objektima kao što su kontejneri, mreže i volumeni. Slika 4.1 pokazuje kako su Docker komponente međusobno povezane.



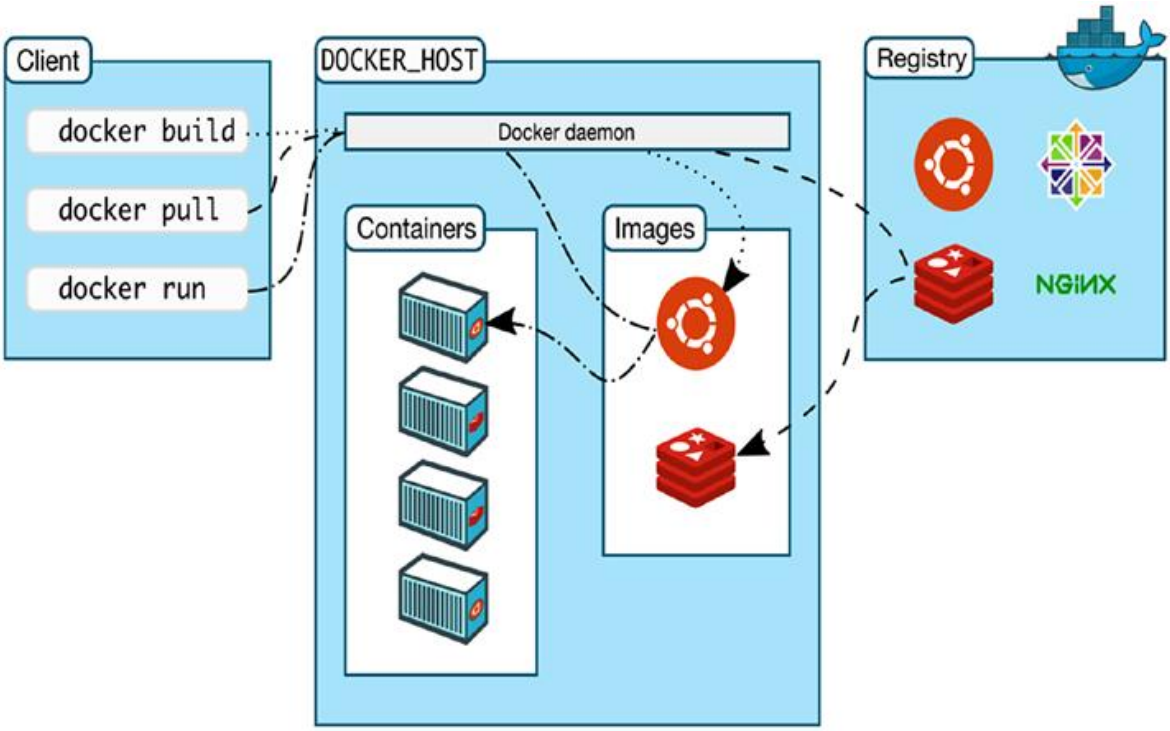
Slika 4.1 Komponente Docker Engine-a[6]

4.2.3. Docker arhitektura

Docker arhitektura bazirana je na klijent-server modelu te definira funkcioniranje i odnose osnovnih Docker komponenti.

Slika 4.2 prikazuje veze između pojedinih komponenti u klijent-serverskoj arhitekturi. Klijent i *daemon* mogu i ne moraju biti na istom serveru, prema potrebi klijent je moguće spojiti na *daemon* udaljenim pristupom.

[6] Izvor: https://docs.docker.com/engine/docker-overview/



Slika 4.2 Detaljan prikaz Docker arhitekture[7]

Docker klijent primarni je način interakcije korisnika s Dockerom. Korištenjem naredbi u komandnom sučelju, one se prosljeđuju Docker daemonu koristeći API sučelje. Docker daemon zatim izvršava te naredbe. Bitno je napomenuti da klijent ima mogućnost komunikacije s više Docker daemona.

Docker daemon je serverski proces koji dugotrajno radi u pozadini. Kontinuirano nadzire REST API sučelje i traži dolazne zahtjeve za procesiranje naredbi.

Softverske slike (engl. *software images*) moraju biti pohranjene na nekoj lokaciji zbog jednostavnosti pristupa – ta lokacija je Docker registar (engl. *registry*). Docker Hub je javan registar koji svatko može koristiti i Docker je prema zadanim postavkama podešen da koristi Docker Hub. Korisnici prema potrebi mogu koristiti svoj privatan registar.

Naredba *docker pull* dohvaća željenu sliku iz konfiguriranog registra dok *docker push* stavlja sliku u registar.

[7] Izvor: https://docs.docker.com/engine/docker-overview/

4.2.4. Docker objekti

Korištenjem Dockera generiraju se mnogi objekti prvenstveno od strane Docker daemona. Objekti uključuju slike (engl. *image*), kontejnere, servise i pohranu.

Docker slika je datotečni sustav namijenjen samo za čitanje koji sadrži upute za kreiranje Docker kontejnera koji može pokretati neku aplikaciju. Korisnici mogu koristiti javne slike s Docker Hub-a ili kreirati vlastite. Dockerfile datoteka koristi se za izgradnju Docker slike, ona sadrži jednostavne naredbe koje daemon izvrši kako bi kreirao i pokrenuo sliku. Svaka naredba u Dockerfile-u kreira poseban sloj unutar slike. Kada kasnije mijenjamo Dockerfile kako bi izmijenili sliku mijenjaju se samo oni slojevi koji sadrže neku promjenu. To je glavni razlog zašto slike zauzimaju puno manje resursa od virtualnih strojeva.

Docker kontejner je instanca slike, slika je pokrenuta unutar kontejnera. Kontejnerom se upravlja naredbama *stop, start i delete.* Na kontejner možemo spojiti jednu ili više mreža, pohranu ili kreirati novu sliku na osnovi njegovog trenutnog stanja. Kontejnere definiramo priloženom slikom i konfiguracijskim opcijama koje smo priložili prilikom pokretanja. Brisanjem kontejnera nestaju svi podaci vezani za njega koji nisu bili trajno pohranjeni.

Korištenjem *docker run* naredbe pokreće se novi kontejner čime se u pozadini odvijaju sljedeći koraci:

1. Docker slika se povlači iz konfiguriranog registra.
2. Kreira se novi Docker kontejner.
3. Docker alocira datotečni sustav za čitanje i pisanje kontejneru što mu omogućava kreiranje i modifikaciju datoteka u njegovom lokalnom datotečnom sustavu.
4. Kontejner se povezuje na zadanu mrežu, osim ako je definirana druga mreža. Također se dodjeljuje IP adresa kontejneru.
5. Docker pokreće kontejner i veže ga za lokalni terminal kako bi mogli njime upravljati.
6. Kontejner je nakon toga moguće zaustaviti ili ukloniti u bilo kojem trenutku pomoću terminala.

Servisi korisnicima dopuštaju skaliranje kontejnera preko više Docker instanci. Na taj način više kontejnera može raditi zajedno i međusobno prosljeđivati podatke kako bi obavili neki zadatak. Krajnjem korisniku će to izgledati kao da sav posao odrađuje jedna aplikacija dok u pozadini zapravo radi više kontejnera na različitim Docker daemon-ima koji komuniciraju preko Docker API-ja.

4.2.5. Docker Hub

Docker Hub je primarna lokacija za pohranu Docker slika. To je javni registar baziran na oblaku s kojega se slike mogu povlačiti ili pohranjivati. To je dućan namijenjen za centraliziranu distribuciju i otkrivanje slika. Korisnik može kupovati ili prodavati slike na Docker Hub-u ili ih distribuirati besplatno. Slike se mogu pretraživati koristeći Docker Hub sučelje ili komandno sučelje.

4.3. Verzije

Docker je dostupan u dvije verzije:

- Community Edition (CE)
- Enterprise Edition (EE)

Docker Community Edition idealan je za eksperimentiranje s kontejnerima te se preporučuje za početnike i manje timove koji počinju raditi s Dockerom.

Docker Enterprise Edition dizajniran je za poduzeća i profesionalne timove koji distribuiraju kritične aplikacije koje se koriste u produkciji.

Tablica 4.2 prikazuje razlike u mogućnostima između verzija. U ovom radu će se za sve svrhe koristiti isključivo CE izdanje.

Tablica 4.2 Razlike između Docker verzija

Mogućnosti	Community Edition	Enterprise Edition
Kontejnerski *engine*, ugrađena orkestracija, mrežna sigurnost	✓	✓
Certificirana infrastruktura, pluginovi		✓
Upravljanje slikama		✓
Upravljanje kontejnerskim aplikacijama		✓
Sigurnosno skeniranje slika		✓

4.4. Docker Instalacija

4.4.1. Linux

S obzirom na to da je Docker izgrađen na osnovi Linux kontejnera instalacija na Linux baziranim operativnim sustavima prilično je jednostavna.

Potrebno je izvršiti nekoliko naredbi u terminalu s administratorskim ovlastima i Docker će biti spreman za korištenje.

Prije same instalacije potrebno je nadograditi sve pakete na najnoviju verziju:

```
Yum -y update
```

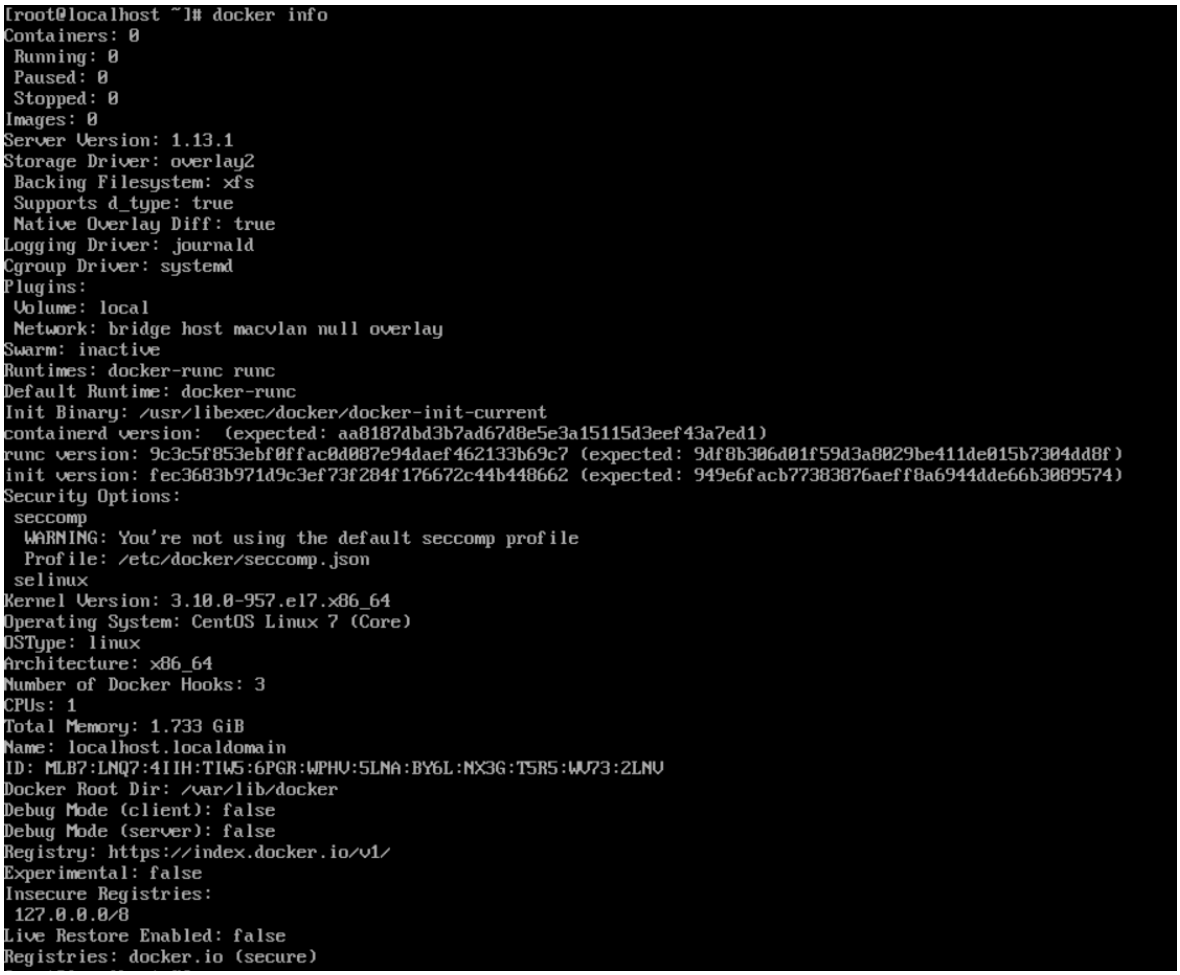
Nakon toga slijedi instalacija dockera:

```
Yum install -y docker
```

Kako bi mogli početi koristiti docker potrebno je pokrenuti docker servis:

```
Service docker start
```

Nakon toga možemo iskoristiti naredbu `docker info` kako bi provjerili ispravnost instalacije i prikazali informacije o Dockeru na našem sustavu. Naredbom dobivamo sljedeći ispis (Slika 4.3) o brojevima kontejnera, verziji, sigurnosnim opcijama itd.



Slika 4.3 Ispis informacija o instalacija i provjera ispravnosti

4.4.2. Windows/MacOS instalacija

Zbog činjenice što Docker koristi neke mogućnosti Linux operativnog sustava proces instalacije za Windows i MacOS se uvelike razlikuje od prijašnjeg pristupa te uključuje dodatne korake. Također postoje dva pristupa instalaciji, stariji proces koji uključuje Docker Toolbox i noviji „Docker za Mac / Windows“ pristup instalaciji.

Docker Toolbox je instalacijski alat koji objedinjuje veći broj komponenti koje su potrebne da bi MacOS ili Windows bili u stanju pokrenuti Docker.

Instalacija uključuje sljedećih 6 alata:

1. Docker CE/EE
2. Docker Compose
3. Docker Machine

4. VirtualBox
5. Docker QuickStart Terminal
6. Kitematic

Zbog toga što se Docker oslanja na komponente Linux sustava kako bi funkcionirao potreban je način da učitamo Linux unutar MacOS/Windows sustava. To se postiže instalacijom VirtualBox-a, hipervizora otvorenog koda koji podržava sve velike platforme. Docker Toolbox izvršava instalaciju i podešavanje VirtualBox-a nakon čega koristi Docker Machine alat kako bi se kreirao virtualni stroj koji sadrži jednostavnu distribuciju Linuxa s instalacijom Docker-a. Kako bi se omogućila interakcija s Dockerom također se vrši instalacija i podešavanje posebnog QuickStart Terminala koji se pri pokretanju automatski spaja na Docker server. Kitematic je opcionalan grafički alat koji omogućava upravljanje Docker slikama i kontejnerima. Docker Compose je alat za definiranje i pokretanje aplikacija koje se protežu kroz više kontejnera.

Docker for Mac / Windows su noviji alati koji ne vrše instalaciju VirtualBox-a. Umjesto toga koriste ugrađene hipervizore koji dolaze ugrađeni u operativne sustave. Za MacOS to je HyperKit, a za Windows Hyper-V.

S ovim alatom instaliraju se sljedeće komponente:

1. Docker CE / EE
2. Docker Compose
3. Docker Machine

Docker daemon će se u ovom slučaju smatrati kao da je pokrenut lokalno tako da više nema potrebe za instalacijom posebnog terminala.

Tablica 4.3 prikazuje kratku usporedbu dvaju alata:

Tablica 4.3 Usporedba dostupnih alata za instalaciju Dockera

Docker Toolbox	Docker za Mac / Windows
Zahtjeva tip 2 hipervizor (pokrenut kao bilo koji drugi program na OS-u)	Zahtjeva tip 1 hipervizor (pokrenut izravno na hardveru računala)
Mountain Lion 10.8+	Yosemite 10.10.3+
Windows 7 Home +	Windows 10 (Pro, Ent, Stu)
Spajanje na daeomon udaljenom konekcijom	Daemon se nalazi na localhostu
Samo QuickStart terminal	Moguće korištenje bilo kojeg terminala (npr. PowerShell)

4.5. Kontejnerizacija jednostavne web aplikacije

Demonstracija se odvija na Windows 10 Pro računalu s instaliranim Docker for Windows alatom. Aplikacija koju ćemo pokrenuti u Docker kontejneru bit će kratka web aplikacija bazirana na Flask-u. Flask je softverski okvir (engl. *software framework*) baziran na Python programskom jeziku koji će nam omogućiti da kreiramo našu aplikaciju. Kao što je prije spomenuto, Docker ne ovisi o programskom jeziku koji koristimo, Flask je u ovom slučaju izabran zbog svoje jednostavnosti. Demonstracija će pokriti sve od pripreme potrebnih datoteka za izradu kontejnera, njegove izrade, pokretanja i demonstracije aplikacije.

4.5.1. Priprema potrebnih datoteka

Prvi korak uključuje pripremu svih datoteka koje će nam biti potrebne da kasnije izradimo kontejner. Dvije glavne datoteke su Dockerfile koji sadrži instrukcije za izradu našeg kontejnera i datoteka koja sadrži kod aplikacije koju namjeravamo staviti u kontejner. Treća datoteka je popis ovisnosti (engl. *dependencies*) koje će nam biti potrebne da uspješno pokrenemo našu aplikaciju u kontejneru.

Kako fokus nije na samoj aplikaciji ona je krajnje jednostavna. Jedina uloga web aplikacije je vratiti vrijednost koju smo zadali, u ovom slučaju poruku „Hello World“. Aplikacija se nalazi u app.py datoteci.

```
from flask import Flask
app = Flask(__name__)
@app.route('/')
def counter():
    return 'Hello World'
```

Kôd 4.1 Web aplikacija koja vraća vrijednost "Hello World"

S obzirom na to da nam je za korištenje aplikacije kao dodatna komponenta potreban samo flask, datoteka *requirements.txt* koja sadrži popis ovisnosti ima samo jednu liniju:

```
Flask==0.12
```

Dockerfile datoteka koja sadrži instrukcije koje su potrebne da bi Docker bio u stanju izgraditi kontejner sadrži sljedeći kod koji će u nastavku biti detaljno objašnjen:

```
FROM python:2.7-alpine

RUN mkdir /app
WORKDIR /app

COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
COPY . .

CMD flask run --host=0.0.0.0 --port=5000
```

Kôd 4.2 Naredbe korištene za izradu Docker image-a

Dockerfile datoteke čitaju se od vrha prema dnu i prva linija koja nije komentar mora sadržavati naredbu `FROM`. Ta naredba nam omogućava da uvezemo osnovni *image* koji će sadržavati programski jezik i operativni sustav potreban da uopće pokrenemo našu aplikaciju u kontejneru. U ovom slučaju odabrana je python distribucija s verzijom 2.7 s projektom Alpine Linux kao operativnim sustavom zbog njegove male veličine (~5MB). Odabrana distribucija nalazi se na Docker Hub-u.

`RUN` naredba omogućava nam pokretanje bilo koje skripte koju bi inače mogli pokrenuti unutar operativnog sustava koji se nalazi u distribuciji koji smo uvezli u prijašnjem koraku. Koristeći *mkdir* kreiramo mapu unutar kontejnera u kojoj će se nalaziti izvorni kod (engl. *source code*) naše aplikacije u kontejneru. Ovo možemo napraviti za bilo koji programski jezik te nije specifično za Flask.

Naredbom `WORKDIR` određujemo direktorij u kojemu će se sve daljnje naredbe odvijati te se tako u budućim naredbama više ne treba specificirati direktorij na koji se uzastopne naredbe odnose.

Naredbom `COPY` kopiramo sadržaj datoteke *requirements* unutar kontejnera u istoimenu datoteku. Na taj način kontejneru postaje dostupan sadržaj datoteke koja sadrži ovisnosti.

Kako Python programski jezik očekuje instalaciju ovisnosti prije nego pokrenemo aplikacija sa sljedećom `RUN` naredbom vršimo instalaciju tih ovisnosti.

Sljedeća `COPY` naredba kopira sve iz naše lokalne mape u *app* direktorij unutar kontejnera uključujući Flask aplikaciju.

Zadnja naredba u Dockerfile-u mora biti `CMD`, ona definira zadanu naredbu koja će se izvršiti kada se naš kontejner pokrene. Razlika između `RUN` i `CMD` naredbi je ta što se `RUN` naredbe izvršavaju samo pri izgradnji kontejnera, `CMD` naredba izvršava se svaki put kada pokrenemo naš *image* u kontejneru. Naredbu koju mi želimo izvršiti s `CMD` je pokretanje Flask servera. Naredba `flask run –host=0.0.0.0 –port=5000` veže flask server na lokalno računalo na port koji smo zadali, u ovom slučaju 5000.

4.5.2. Izgradnja i pokretanje kontejnera

Kako bi Docker daemonu poslali naredbu da izradi kontejner otvaramo terminal i postavljamo se u mapu u kojoj se nalaze datoteke koje smo pripremili. U ovom slučaju to je: `C:\DockerApp\`

U sljedećem koraku koristimo naredbu: `docker image build -t web1 .`

Ova naredba gradi Docker *image* zvan web1 u trenutnom direktoriju koji ćemo kasnije moći pokrenuti unutar kontejnera. Nakon što pokrenemo naredbu docker kreće u izgradnju našeg *image-a* izvršavajući naredbe koje su opisane u prijašnjem poglavlju pri čemu dobivamo sljedeći ispis (Slika 4.4):

```
PS C:\DockerApp> docker image build -t web1 .
Sending build context to Docker daemon  4.096kB
Step 1/7 : FROM python:2.7-alpine
2.7-alpine: Pulling from library/python
e7c96db7181b: Pull complete
 1819f4b92bc2: Pull complete
  da13060a0845: Pull complete
   2b929b7be260: Pull complete
    Digest: sha256:6f2af74e4ecf234b902d6a5adff91d51d4cd10f124a389450cbf08f7ffbe3805
Status: Downloaded newer image for python:2.7-alpine
 ---> ee70cb11da0d
Step 2/7 : RUN mkdir /app
 ---> Running in 1e7d11c79cd3
Removing intermediate container 1e7d11c79cd3
 ---> 85032282a8e6
Step 3/7 : WORKDIR /app
 ---> Running in e7bea8ea2293
Removing intermediate container e7bea8ea2293
 ---> 5099b8901f2a
Step 4/7 : COPY requirements.txt requirements.txt
 ---> 15ddcad86a69
Step 5/7 : RUN pip install -r requirements.txt
 ---> Running in e9935500e9c2
DEPRECATION: Python 2.7 will reach the end of its life on January 1st, 2020. Please upgrade your Python as Python 2.7 w
on't be maintained after that date. A future version of pip will drop support for Python 2.7.
Collecting Flask==0.12 (from -r requirements.txt (line 1))
  Downloading https://files.pythonhosted.org/packages/0e/e9/37ee66dde483dceefe45bb5e92b387f990d4f097df40c400cf816dcebaa
4/Flask-0.12-py2.py3-none-any.whl (82kB)
Collecting itsdangerous>=0.21 (from Flask==0.12->-r requirements.txt (line 1))
  Downloading https://files.pythonhosted.org/packages/76/ae/44b03b253d6fade317f32c24d100b3b35c2239807046a4c953c7b89fa49
e/itsdangerous-1.1.0-py2.py3-none-any.whl
Collecting click>=2.0 (from Flask==0.12->-r requirements.txt (line 1))
  Downloading https://files.pythonhosted.org/packages/fa/37/45185cb5abbc30d7257104c434fe0b07e5a195a6847506c074527aa599e
c/Click-7.0-py2.py3-none-any.whl (81kB)
Collecting Jinja2>=2.4 (from Flask==0.12->-r requirements.txt (line 1))
  Downloading https://files.pythonhosted.org/packages/1d/e7/fd8b501e7a6dfe492a433deb7b9d833d39ca74916fa8bc63dd1a4947a67
1/Jinja2-2.10.1-py2.py3-none-any.whl (124kB)
Collecting Werkzeug>=0.7 (from Flask==0.12->-r requirements.txt (line 1))
  Downloading https://files.pythonhosted.org/packages/9f/57/92a497e38161ce40606c27a86759c6b92dd34fcdb33f64171ec559257c0
2/Werkzeug-0.15.4-py2.py3-none-any.whl (327kB)
Collecting MarkupSafe>=0.23 (from Jinja2>=2.4->Flask==0.12->-r requirements.txt (line 1))
  Downloading https://files.pythonhosted.org/packages/b9/2e/64db92e53b86efccfaea71321f597fa2e1b2bd3853d8ce658568f7a1309
4/MarkupSafe-1.1.1.tar.gz
Building wheels for collected packages: MarkupSafe
  Building wheel for MarkupSafe (setup.py): started
  Building wheel for MarkupSafe (setup.py): finished with status 'done'
  Stored in directory: /root/.cache/pip/wheels/f2/aa/04/0edf07a1b8a5f5f1aed7580fffb69ce8972edc16a505916a77
Successfully built MarkupSafe
Installing collected packages: itsdangerous, click, MarkupSafe, Jinja2, Werkzeug, Flask
Successfully installed Flask-0.12 Jinja2-2.10.1 MarkupSafe-1.1.1 Werkzeug-0.15.4 click-7.0 itsdangerous-1.1.0
Removing intermediate container e9935500e9c2
 ---> 97ad6c8681b7
Step 6/7 : COPY . .
 ---> 906584ed6f4f
Step 7/7 : CMD flask run --host=0.0.0.0 --port=5000
 ---> Running in 53df8c422f0a
Removing intermediate container 53df8c422f0a
 ---> 522597309568
Successfully built 522597309568
Successfully tagged web1:latest
```

Slika 4.4 Izrada Docker *image*-a

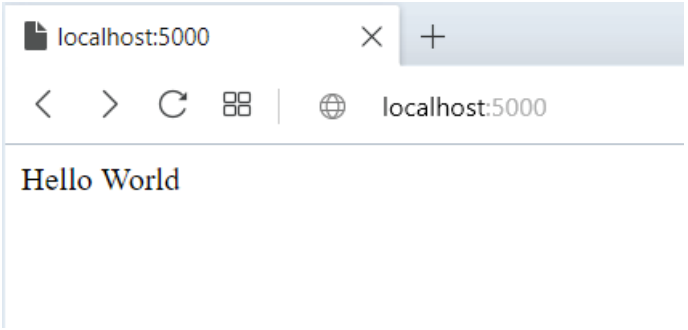
Nakon što smo uspješno izradili *image* preostaje samo pokrenuti ga unutar kontejnera što postižemo sljedećom naredbom:

```
docker container run -it -p 5000:5000 -e FLASK_APP=app.py web1
```

Argumenti -it Dockeru dozvoljavaju obradu UNIX signala kao kombinaciju Ctrl+C koja se koristi za prekidanje nekog procesa. Zastavica -p određuje portove koji se koriste na lokalnom računalu i u Docker kontejneru te ih čini dostupnima za korištenje. Zastavica -e

dopušta prosljeđivanje varijabli okruženja, a Flask softverski okvir očekuje varijablu zvanu FLASK_APP koja pokazuje na našu aplikaciju. Zadnja varijabla koju trebamo priložiti je ime našeg Docker *image-a*.

Izvršavanjem naredbe naša web aplikacija pokreće se unutar Docker kontejnera te je spremna za korištenje. Ako posjetimo adresu localhost:5000 u web pregledniku na *host* računalu web aplikacija će uistinu vratiti poruku „Hello World“ kao što vidimo na slici 4.5.



Slika 4.5 Web aplikacija koja iz kontejnera vraća ispis

5. Demonstracija pristupa IoT hardveru kroz Docker kontejner

5.1. Uvod

Svrha ove demonstracije je pokazati koliko brzo i jednostavno se IoT aplikacija može distribuirati pomoću Docker kontejnera. Korišteni hardver jeftin je i široko dostupan što dodatno olakšava implementaciju. Kako fokus nije na programskom dijelu za demonstraciju je korišten već izgrađen, javno dostupan *image*, hypriot/wiringpi[8] s Docker Hub-a od autora Govinda Fichtner. Izvorni kod Dockerfile-a korišten za kreiranje Docker *image*-a dostupan je na GitHub-u[9]. Spomenuti *image* sadrži sve potrebne komponente za komunikaciju s našim IoT uređajem koji je povezan na Raspberry Pi. Primjer je jednostavan i uključuje osnovnu funkcionalnost očitavanja temperature i vlage s DHT 22 senzora. Naravno, što se tiče funkcionalnosti, primjene ovakvog i sličnih senzora daju se uvelike proširiti. Vrijednosti se mogu očitavati u vremenskim intervalima i pohranjivati u bazu podataka za kasniju analizu, podaci se u stvarnom vremenu mogu slati na mobilnu aplikaciju ili na osnovi podataka upravljati drugim uređajima.

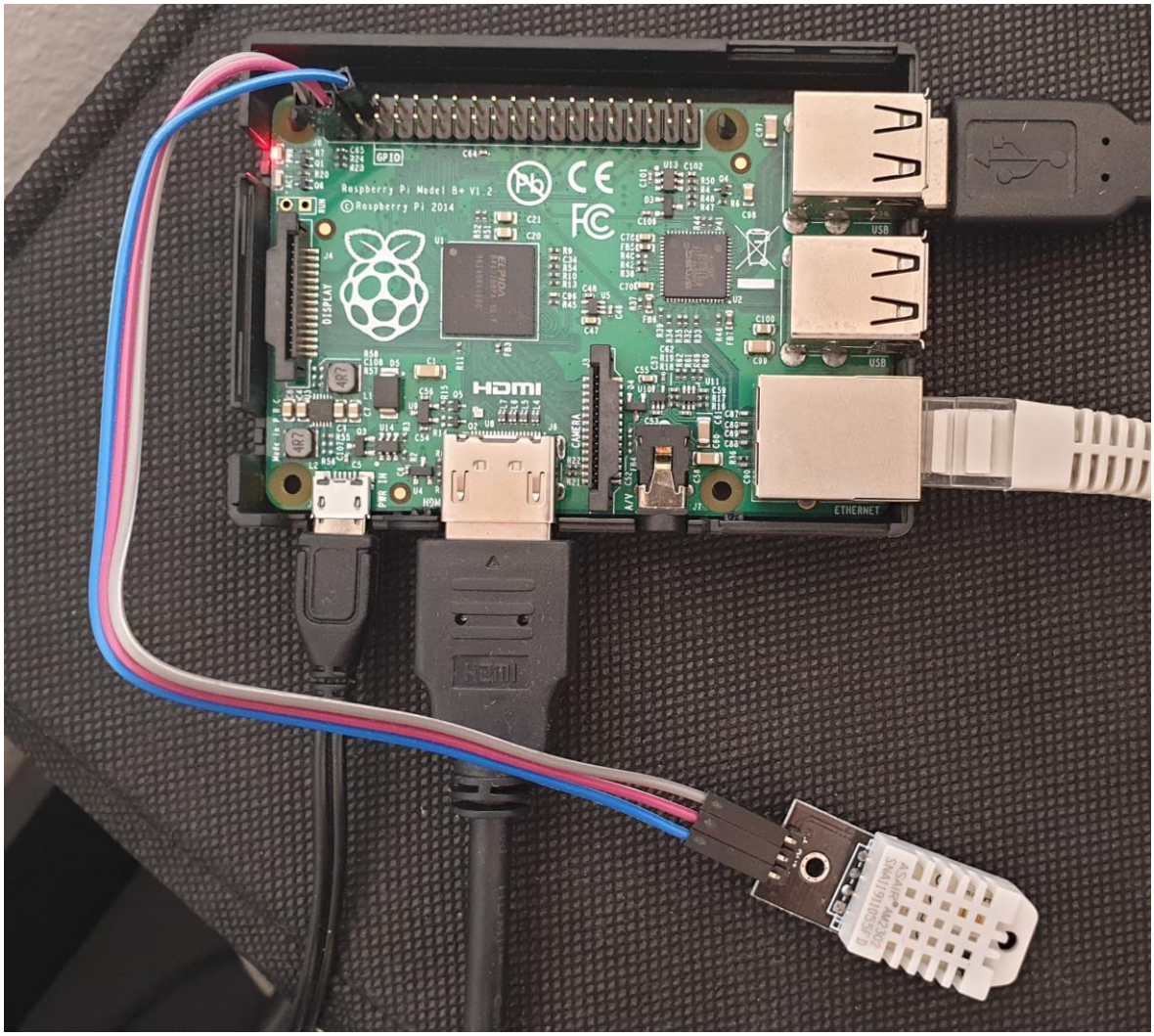
Za demonstraciju korišteni su:

- Raspberry Pi B+ verzija 1.2
- SanDisk microSD kartica 16GB
- HypriotOS[10] operativni sustav
- Balena Etcher[11] flashing alat
- DHT22 senzor za mjerenje temperature i vlage
- hypriot/wiringpi Docker image

HypriotOS je operativni sustav specijaliziran za pokretanje Docker-a na Raspberry Pi uređajima s ARM arhitekturom. Docker alat dolazi predinstaliran s postavkama koje su optimizirane za njegovo pokretanje.

[8] https://hub.docker.com/r/hypriot/wiringpi/
[9] https://gist.github.com/Govinda-Fichtner/1d48543a6b9c2e21b7c1#file-dockerfile-final
[10] https://blog.hypriot.com
[11] https://www.balena.io/etcher/

Korišteni hardver vidljiv je na Slika 5.1.



Slika 5.1 DHT22 senzor povezan na Raspberry Pi

5.2. Implementacija i očitanje senzora

Prvi korak uključuje spajanje DHT22 senzora na Raspberry Pi GPIO (engl. *general purpose input/output*) pinove koji su definirani u Docker image-u. Naš Docker image sadrži WiringPi[12] biblioteku koja nam omogućava komunikaciju s GPIO pinovima.

Referenciranjem dokumentacije WiringPi projekta možemo vidjeti koji pinovi se mogu koristiti za napajanje, uzemljenje i čitanje podataka. U našem slučaju odabran je fizički pin

[12] http://wiringpi.com

broj 3 za čitanje podataka, koji je u wiringpi biblioteci označen s brojem 8. Kasnije ćemo u naredbi definirati taj pin za očitavanje vrijednosti.

Iz samog kontejnera također možemo dobiti ispis svih pinova i njihovih funkcija kao što vidimo na Slika 5.2.

```
root@black-pearl:~# docker run --device /dev/ttyAMA0:/dev/ttyAMA0 --device /dev/
mem:/dev/mem --privileged -ti hypriot/wiringpi bash
root@0dd1b26011ad:/data# gpio readall
 +-----+-----+---------+------+---+--B Plus--+---+------+---------+-----+-----+
 | BCM | wPi |   Name  | Mode | V | Physical | V | Mode | Name    | wPi | BCM |
 +-----+-----+---------+------+---+----++----+---+------+---------+-----+-----+
 |     |     |    3.3v |      |   |  1 || 2  |   |      | 5v      |     |     |
 |   2 |   8 |   SDA.1 |   IN | 1 |  3 || 4  |   |      | 5V      |     |     |
 |   3 |   9 |   SCL.1 |   IN | 1 |  5 || 6  |   |      | 0v      |     |     |
 |   4 |   7 | GPIO. 7 |   IN | 1 |  7 || 8  | 1 | ALT0 | TxD     | 15  | 14  |
 |     |     |      0v |      |   |  9 || 10 | 1 | ALT0 | RxD     | 16  | 15  |
 |  17 |   0 | GPIO. 0 |   IN | 0 | 11 || 12 | 0 | IN   | GPIO. 1 | 1   | 18  |
 |  27 |   2 | GPIO. 2 |   IN | 0 | 13 || 14 |   |      | 0v      |     |     |
 |  22 |   3 | GPIO. 3 |   IN | 0 | 15 || 16 | 0 | IN   | GPIO. 4 | 4   | 23  |
 |     |     |    3.3v |      |   | 17 || 18 | 0 | IN   | GPIO. 5 | 5   | 24  |
 |  10 |  12 |    MOSI |   IN | 0 | 19 || 20 |   |      | 0v      |     |     |
 |   9 |  13 |    MISO |   IN | 0 | 21 || 22 | 0 | IN   | GPIO. 6 | 6   | 25  |
 |  11 |  14 |    SCLK |   IN | 0 | 23 || 24 | 1 | IN   | CE0     | 10  | 8   |
 |     |     |      0v |      |   | 25 || 26 | 1 | IN   | CE1     | 11  | 7   |
 |   0 |  30 |   SDA.0 |   IN | 1 | 27 || 28 | 1 | IN   | SCL.0   | 31  | 1   |
 |   5 |  21 | GPIO.21 |   IN | 1 | 29 || 30 |   |      | 0v      |     |     |
 |   6 |  22 | GPIO.22 |   IN | 1 | 31 || 32 | 0 | IN   | GPIO.26 | 26  | 12  |
 |  13 |  23 | GPIO.23 |   IN | 0 | 33 || 34 |   |      | 0v      |     |     |
 |  19 |  24 | GPIO.24 |   IN | 0 | 35 || 36 | 1 | OUT  | GPIO.27 | 27  | 16  |
 |  26 |  25 | GPIO.25 |   IN | 0 | 37 || 38 | 0 | IN   | GPIO.28 | 28  | 20  |
 |     |     |      0v |      |   | 39 || 40 | 0 | IN   | GPIO.29 | 29  | 21  |
 +-----+-----+---------+------+---+----++----+---+------+---------+-----+-----+
 | BCM | wPi |   Name  | Mode | V | Physical | V | Mode | Name    | wPi | BCM |
 +-----+-----+---------+------+---+--B Plus--+---+------+---------+-----+-----+
```

Slika 5.2 Ispis svih dostupnih GPIO pinova i njihovih funkcija

Nakon spajanja senzora potrebno je izvršiti *flash* operativnog sustava na SD karticu pomoću Balena Etcher alata nakon čega je on spreman za korištenje na Raspberry Pi-u. Sve što nas sada dijeli od dobivanja očitanja je nekoliko naredbi.

Prva naredba uključuje povlačenje *image-a* s javnog repozitorija, Docker Hub-a koji će nakon što se povuče odmah biti spreman za korištenje:

```
docker pull hypriot/wiringpi
```

Sve što nam preostaje je pokrenuti kontejner i dobiti očitanje. To postižemo sa sljedećom naredbom u kojoj kontejneru dajemo pristup potrebnim fizičkim uređajima i definiramo pin s kojega želimo očitati podatke:

```
docker run --device /dev/ttyAMA0:/dev/ttyAMA0 --device /dev/mem:/dev/mem
--privileged -ti hypriot/wiringpi /loldht 8
```

Rezultat je vidljiv na Slika 5.3:

```
root@black-pearl:~# docker run --device /dev/ttyAMA0:/dev/ttyAMA0 --device /dev/
mem:/dev/mem --privileged -ti hypriot/wiringpi /loldht 8
Raspberry Pi wiringPi DHT22 reader
www.lolware.net
Data not good, skip
Humidity = 46.80 % Temperature = 28.20 *C
```

Slika 5.3 Ispis temperature i vlage

Što, uz minimalna odstupanja, odgovara stvarnim uvjetima kada usporedimo očitane vrijednosti s kućnim termostatom (Slika 5.4):

Slika 5.4 Provjera ispravnosti dobivenih podataka

5.3. Prednosti vidljive iz praktičnog primjera

Praktičan primjer pokazuje koliko malo konfiguracije je zapravo potrebno kada sve potrebne programske komponente imamo spremne u Docker kontejneru. U našem slučaju samo dvije naredbe su nas dijelile od dobivanja očitanja s IoT uređaja. Čak uz pripremu operativnog

sustava IoT uređaj se na nekoj lokaciji može implementirati u svega nekoliko minuta. Kako se isti kontejner koristi za testiranje i implementaciju, kada dođe do izmjena dovoljno je izmijeniti program jednom, ažurirati kontejner i ponovno pokrenuti pull naredbu. Nije potrebna nikakva deinstalacija ili uklanjanje starih verzija, Docker jednostavno skine noviju verziju i zamijeni staru.

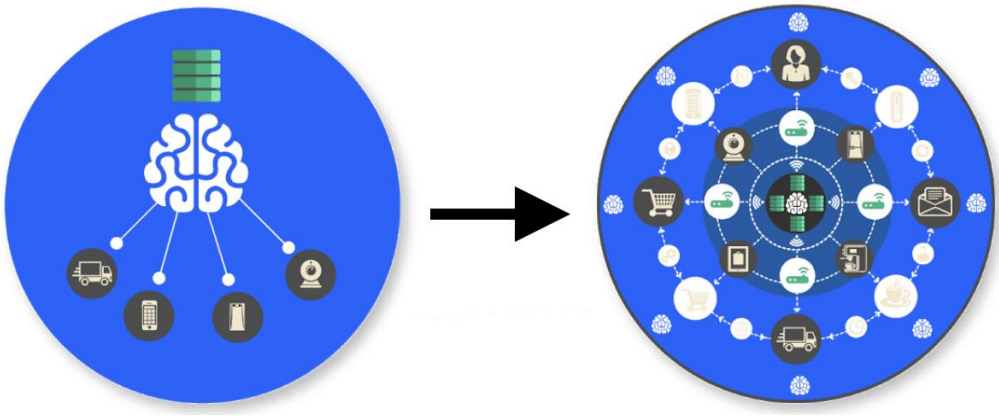
Ovakva ušteda vremena i smanjivanje kompleksnosti implementacije postaje ogromna prednost kada počnemo koristiti na stotine ili tisuće IoT uređaja koje je potrebno orkestrirati – automatski ih konfigurirati, koordinirati i njima upravljati.

6. Mjerenje performansi kontejnera

6.1. Uvod

U radu je već spomenuto kako Docker, zahvaljujući svojoj arhitekturi, ima manji *overhead* u smislu zauzeća prostora za pohranu i memorije. Kako bi dobili bolji uvid u tehnologiju potrebni su nam i podaci koji govore o performansama kontejnera. U slučaju kada kontejnere koristimo za jednostavne zadatke kao što je očitanje senzora i slanje podataka performanse ne igraju bitnu ulogu. Unatoč tome, kao i većina tehnologija, IoT se nastavlja razvijati.

Trenutno, većina podataka od strane IoT uređaja prikuplja i analizara se u oblaku ili centraliziranim serverima. Svi smo upoznati s ovakvim provjerenim pristupom gdje ne moramo previše razmišljati o ograničenjima resursa pri radu s podacima. Unatoč pouzdanom pristupu, u IoT industriji se događaju pomaci prema novom modelu zvanom IoT Edge Computing. Ideja je približiti analitičke sposobnosti i sposobnosti donošenja odluka rubu mreže kako bi se adresirali neki od trenutnih i predstojećih izazova. Neke od prednosti uključuju samostalno donošenje odluka, postepena degradacija usluge umjesto potpunog ispada i uštede kod količine prenesenih podataka. Slika 6.1 ilustrira očekivane promjene u IoT mrežama. Do kraja 2019. godine očekuje se generiranje preko 500 zettabajta[13] (1 zettabajt = trilijun gigabajta) podataka od strane svih ljudi i uređaja povezanih na internet, tako da novi pristup itekako ima smisla.



Slika 6.1 Prelazak tradicionalnih IoT mreža na Edge Computing [14]

[13] http://intelzone.com/iot-sensors-edge-computing/
[14] Izvor: https://www.postscapes.com/iot-edge-computing-software/

Kako bi provjerili prikladnost kontejnera za buduće implementacije IoT edge computing-a usporedit ćemo performanse Docker-a s performansama virtualnih strojeva i performansama fizičkog stroja. Cilj je dobiti okvirnu usporedbu kako bi se na osnovi tih podataka moglo zaključiti kakvo ponašanje kontejnera možemo očekivati u odnosu na postojeću implementaciju gdje većinu stvari virtualiziramo.

Performanse računala bez ikakvih virtualizacijskih slojeva poslužit će nam kao referentna vrijednost za olakšavanje usporedbe. Za mjerenje performansi korišten je sintetički alat "sysbench[15]" zbog modularnosti, mogućnosti pokretanja unutar terminala i velikog broja opcija za podešavanje mjerenja.

Provedena su mjerenja performansi procesora, memorije, čitanja i pisanja podataka. Sva mjerenja ponovljena su 10 puta.

6.2. Okruženje

Korišteni softver

- Ubuntu 19.04 (64-bit) – korišten kao *host* operativni sustav na kojem su dobivene referentne vrijednosti i kasnije pokrenuti Docker i VirtualBox
- Docker verzija 19.03
- VirtualBox verzija 6.0.6
- Sysbench verzija 1.0.15

Korišteni hardver

- Host računalo sa: Processor Intel Core i7-6700HQ @ 2.60GHz, 8 GB RAM, Intel SSD p600 256GB

[15] http://manpages.ubuntu.com/manpages/disco/en/man1/sysbench.1.html

6.3. Metodologija

Kao host operativni sustav izabrana je najnovija stabilna Ubuntu verzija, Ubuntu 19.04. Na *host* računalu izvršena je instalacija s najnovijim ažuriranjima uz instalaciju sysbench alata. Mjerenja na *host*-u izvršena su prije instalacije bilo kakvih drugih aplikacija. Zatim je instalirana Docker verzija 19.03 te skinut Docker *image* koji sadrži Ubuntu distribuciju iste verzije kao i naš *host*. Veličina tog *image-a* je svega 64.2 MB. Mjerenja su provedena nakon što je *image* u kontejneru ažuriran i instaliran sysbench. Nakon dobivenih rezultata s *host*-a je uklonjen Docker te instaliran VirtualBox u kojem je postavljen virtualni stroj s Ubuntu 19.04 distribucijom. Virtualnom stroju dodijeljena je polovica resursa od *host*-a na kojem se nalazi. To uključuje 2 fizičke jezgre računala i 4GB radne memorije. Stroju je također dodijeljeno 50GB prostora za pohranu što je petina kapaciteta host-a. Sva provedena mjerenja su *single threaded*, što znači da se oslanjaju na samo jednu jezgru procesora. Fizički stroj tako ne može postići bolje rezultate na osnovi većeg broja dostupnih jezgri.

Sva provedena mjerenja su sintetička čime smo u stanju detaljno i kontrolirano odrediti parametre koje želimo testirati uz pomoću bogatog izbora opcija.

6.4. CPU performanse

Prvi test uključuje testiranje i usporedbu performansi procesora između fizičkog računala, docker kontejnera i virtualnog stroja. Korištena naredba je sljedeća:

```
sysbench cpu run --threads=1 --events=1 --cpu-max-prime=8000000
```

Naredba definira testiranje procesora pomoću jednog događaja koji se odvija na jednoj dretvi. Događaj vrši provjeru prostih brojeva do vrijednosti koju smo zadali u opciji `cpu-max-prime`.

Kao što je i očekivano, u Tablica 6.1 vidimo da su performanse na fizičkom računalu bolje od virtualizacijskih tehnologija. Docker kontejneri imaju skoro neznatan utjecaj na performanse od 1.9%, u odnosu na 5% kod virtualnog stroja.

Tablica 6.1 Mjerenja CPU performansi

cpu	Fizički stroj	Kontejner	Virtualni stroj
1	9.0716	9.2068	9.5419
2	9.1002	9.4389	9.7195
3	9.1249	9.2020	9.4950
4	9.0754	9.2536	9.4560
5	9.1084	9.3480	9.4832
6	9.0690	9.2504	9.5.34
7	9.1050	9.2203	9.4686
8	9.0633	9.2413	9.5139
9	9.1555	9.2090	9.4792
10	9.0961	9.3301	9.5234
prosjek	9.0969	9.27	9.5521

6.5. Performanse memorije

Testiranje brzine pristupa memoriji provedeno je pomoću naredbe:

```
sysbench --test=memory --memory-block-size=1K --memory-total-size=100G --num-threads=1 run
```

Naredba provodi zapisivanje podataka u obliku nasumičnih brojeva u radnu memoriju računala pri čemu se koristi samo jedna dretva. Način pristupa memoriji je nasumičan dok se podaci zapisuju u blokovima veličine 1 kilobajta. 100 gigabajta podataka ne označava količinu alocirane memorije nego količinu podataka koji će se zapisati u memoriju prije nego test završi. Testom dobivamo brzinu zapisivanja podataka u RAM memoriju u obliku MiB/sec (megabajta po sekundi). U Tablica 6.2 vidimo da fizički stroj ponovno ima najbolje rezultate dok virtualni stroj u ovom slučaju ima 3.5% bolji rezultat od kontejnera.

Tablica 6.2 Brzina zapisivanja podataka u memoriju

Memorija	Fizički stroj	Docker	Virtualni stroj
1	5716.82	4958.38	5062.22
2	5678.03	4830.56	4874.40
3	5655.52	4865.27	5054.30
4	5665.99	4710.70	5050.60
5	5657.89	4804.30	5021.70
6	5670.55	4884.10	4943.43
7	5597.20	4805.14	5012.84
8	5587.68	4826.54	4880.52
9	5641.26	4801.67	4965.35
10	5689.66	4798.12	5116.66
Prosjek	5656.06	4828.48	4998.2

6.6. I/O performanse

Testiranje performansi čitanja i pisanja podataka zadnje je provedeno mjerenje. Za razliku od prijašnjih mjerenja ovdje je potrebno prvo pripremiti datoteke ili skup datoteka koje ćemo koristiti pomoću naredbe:

```
sysbench fileio prepare --file-num=10 --file-total-size=10G --file-extra-flags=direct
```

`--file-total-size` određuje ukupnu veličinu datoteka dok sa `–file-num` određujemo njihov broj. Kako ne bi došlo do čitanja podataka iz cache memorije što bi rezultiralo nerealnim brzinama čitanja koristimo opciju `–file-extra-flags=direct`. Na taj način izbjegavamo cache memoriju, ali može doći do narušavanja perfomansi. U našem slučaju nije bitno ako ne postignemo maksimalne brzine čitanja i pisanja, cilj nam je dobiti okvirnu usporedbu brzina čitanja.

Naredba korištena za mjerenje I/O performansi je:

```
sysbench fileio run --file-num=10 --file-total-size=10G --file-test-mode=rndrw --file-extra-flags=direct --time=60
```

U prvom dijelu naredbe definiramo datoteke koje smo pripremili, način testiranja je nasumično čitanje i pisanje pri čemu svaki test traje 60 sekundi.

Iz podatak koji su priloženi u tablicama 6.3 i 6.4 vidimo da Docker kontejneri zaostaju za performansama virtualnog stroja. Docker ima 11.19% sporije performanse kod čitanja podataka i 11.5% kod pisanja podataka.

Tablica 6.3 Usporedba brzine čitanja

I/O read	Fizički stroj	Docker	Virtualni stroj
1	82.43	72.91	76.73
2	84.21	72.84	79.82
3	84.91	74.15	80.47
4	78.17	73.54	80.19
5	85.80	70.06	80.09
6	86.64	71.42	79.45
7	83.35	72.72	80.68
8	81.54	71.55	80.02
9	85.48	67.94	79.62
10	83.26	69.87	80.24
Prosjek	83.58	71.7	79.73

Tablica 6.4 Usporedba brzine pisanja

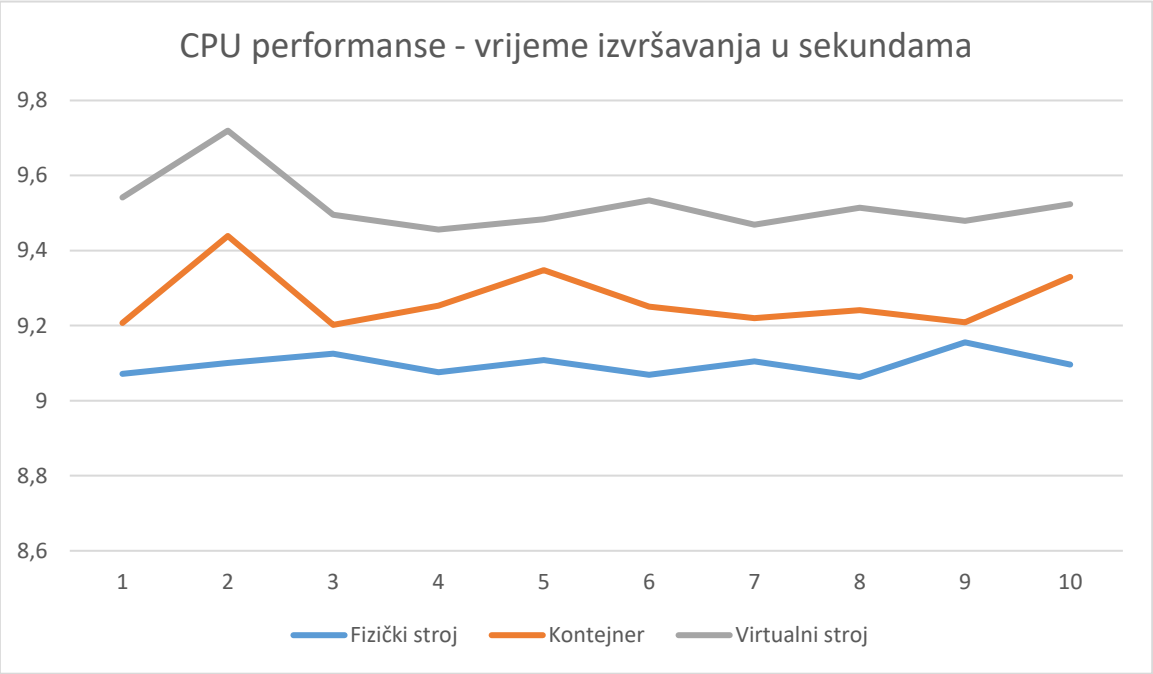
I/O write	Fizički stroj	Docker	Virtualni stroj
1	54.95	48.61	51.15
2	56.14	48.42	53.95
3	56.61	49.44	53.65
4	54.78	48.16	53.50
5	57.20	46.70	53.39
6	57.76	47.25	52.86
7	55.57	48.48	53.79
8	54.12	47.10	52.28
9	56.95	45.96	53.08
10	55.47	46.21	53.47
Prosjek	55.96	47.63	53.11

6.7. Pregled rezultata

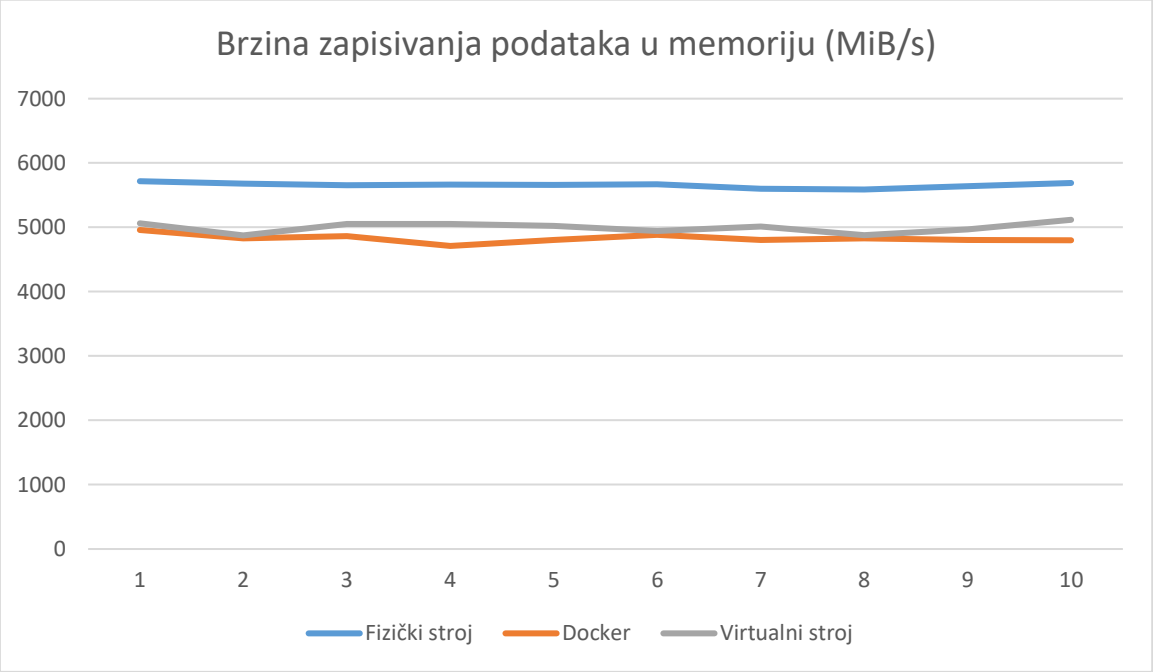
Iz grafičke usporedbe na slikama 6.1-6.4 možemo brzo zaključiti da Docker kontejneri nemaju nužno bolje performanse od virtualnih strojeva. Iz provedenih mjerenja možemo vidjeti da u slučaju iskorištenosti procesora Docker ima prednost, kod RAM performansi postoji mala razlika dok kod I/O operacija virtualni stroj preuzima vodstvo.

Unatoč tome što Docker zaostaje u nekim kategorijama razlike su takve da s obzirom na ostale prednosti nipošto ne predstavljaju razlog zašto bi se kontejneri trebali izbjegavati. Zbog efikasnosti pri korištenju resursa od strane kontejnera može se čak razmišljati o premještanju zahtjevnijih aplikacija u Docker kontejnere. Kada o rezultatima pričamo u kontekstu IoT-a performanse su nebitan čimbenik za trenutne implementacije. Kod mogućih

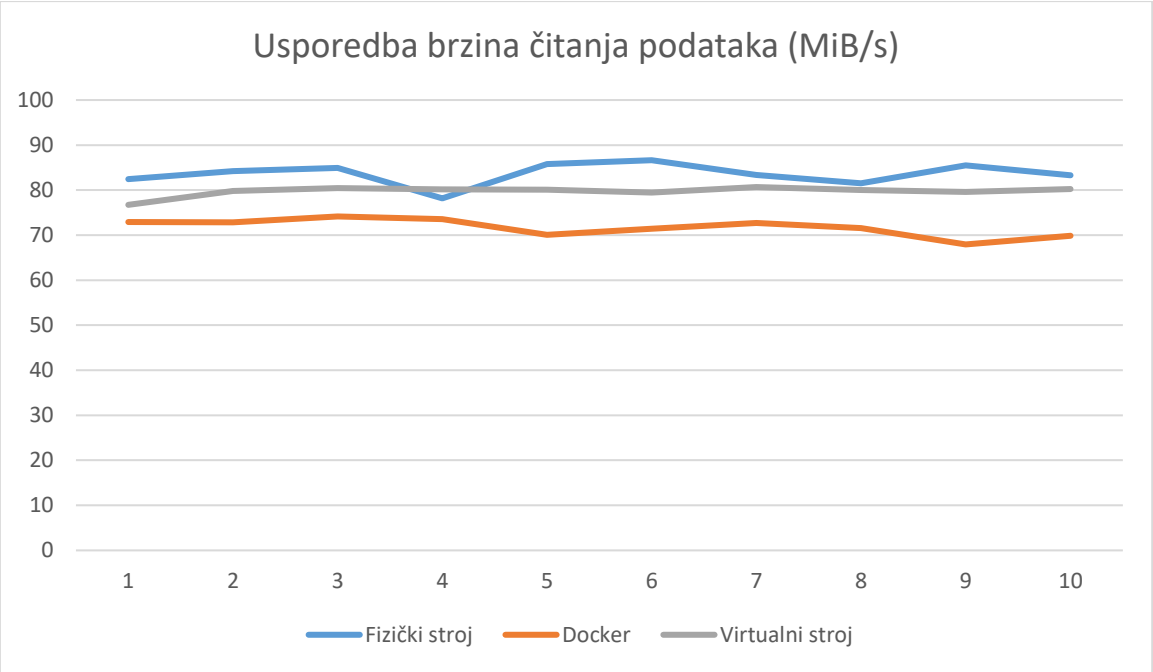
budućih primjena u *edge computing*-u varijacije u performansama su takve da teško možemo opravdati korištenje i miješanje drugih tehnologija uz kontejnerski pristup.



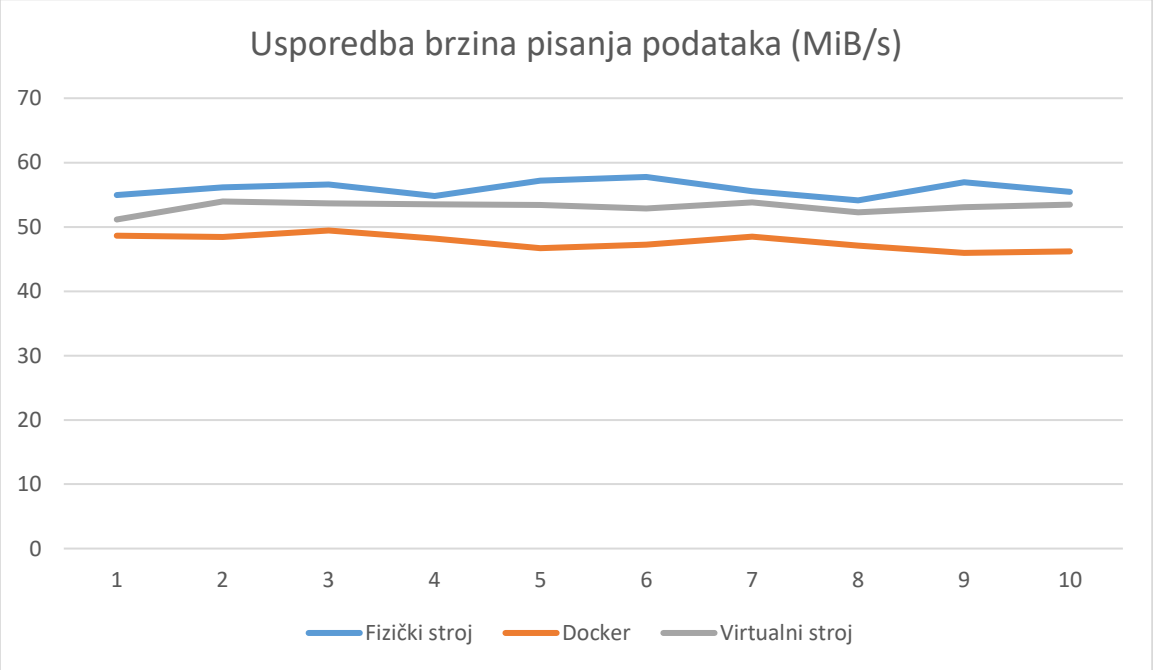
Slika 6.2 Grafički prikaz usporedbe CPU performansi



Slika 6.3 Grafički prikaz usporedbe RAM performansi



Slika 6.4 Grafički prikaz usporedbe brzina čitanja



Slika 6.5 Grafički prikaz usporedbe brzina pisanja

Zaključak

Broj IoT uređaja u zastrašujućem je porastu, milijarde novih uređaja povezuju se na internet svake godine. Razlog tome su nove tehnologije koje su omogućile njihov razvoj i sve veća vrijednost podataka koje nam ti uređaji mogu proslijediti. IoT uređaji više nisu ograničeni mikrokontrolerima zbog optimizacije, minijaturizacije i niske potrošnje današnjih računala. Ta činjenica IoT uređaju omogućuje pokretanje cijelih operativnih sustava, a kao posljedicu toga i izvršavanje koda bilo kojeg programskog jezika.

Unatoč toj pojavi još uvijek ne postoji standard za implementaciju IoT-a. Kod se još uvijek piše za specifične uređaje, sigurnosne prakse ne postoje, a implementacija sigurnosnih zakrpi je dugotrajan proces. Uz velik broj uređaja postavljaju se pitanja kako ubrzati implementaciju, poboljšati sigurnost, optimizirati korištenje resursa, olakšati testiranje itd. Provjereno rješenje koje adresira navedene probleme je virtualizacija, ali postoji i novija tehnologija koja u IoT kontekstu ima zanimljive prednosti – kontejneri.

U sljedećoj tablici možemo vidjeti usporedbu glavnih karakteristika kontejnera i virtualnih strojeva:

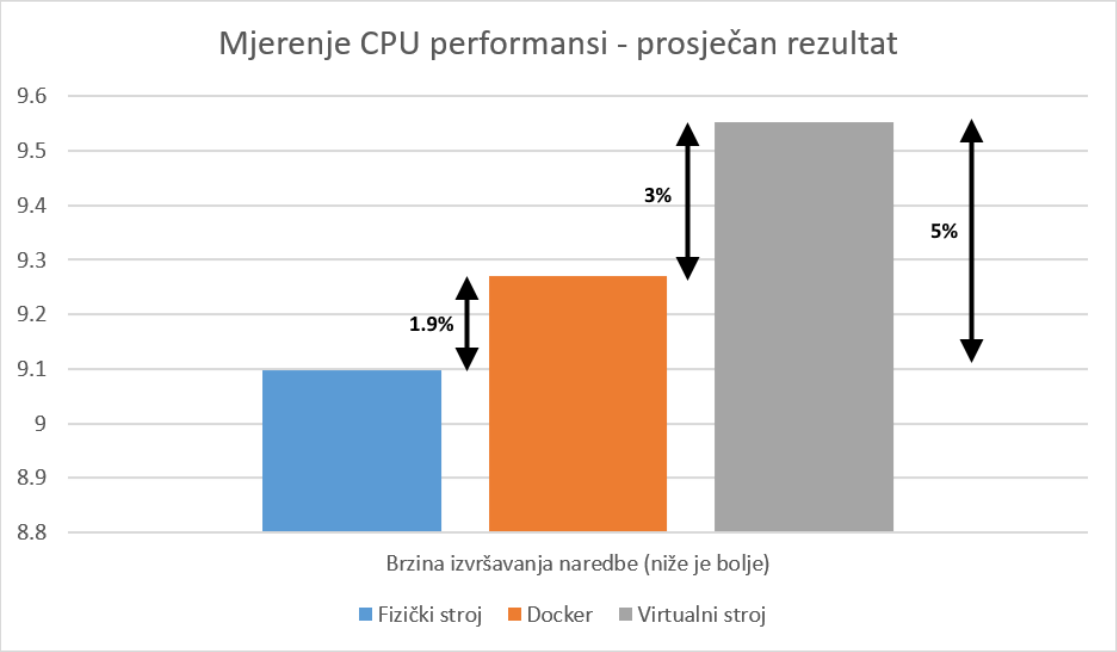
Tablica 7.1 Prednosti i nedostaci kontejnera

Virtualni strojevi	**Kontejneri**
Zauzimaju više resursa	Zauzimaju manje resursa
Teža i sporija nadogradnja	Izmjene se provode jednostavnom zamjenom starog kontejnera sa novim
Pokretanje traje i do nekoliko minuta	Pokreću se za nekoliko sekundi
Resursi se zauzimaju unaprijed	Resursi se zauzimaju prema potrebi
Potpuno izolirani – bolja sigurnost	Izolacija na razini procesa
Bolja opcija za kompleksne aplikacije	Idealni za jednostavnije zadatke

Kada uzmemo u obzir hardverska ograničenja s kojima se IoT uređaji suočavaju i uvjete u kojima se implementiraju prednost kontejnera je očigledna i čini ih savršenima za ovakvu primjenu. Primjer implementacije senzora temperature i vlage sa Raspberry Pi dobar je pokazatelj ograničenja s kojima se susrećemo u stvarnom svijetu. ARM procesor i 512MB RAM memorije potpuno su neprikladni za bilo koju vrstu virtualizacije. Praktičan primjer također je izvrstan pokazatelj kako zahvaljujući kontejnerima za nekoliko minuta možemo implementirati IoT rješenje na nekoj lokaciji pomoću široko dostupnog i jeftinog hardvera. Cijene SBC (engl. *single board computer*) računala kao što su RaspberryPi počinju od $5 dok su cijene senzora zanemarive. U ovakvim uvjetima implementacija IoT uređaja pomoću kontejnera idealno je rješenje zbog kombinacije efikasnog korištenja resursa, fleksibilnosti i jednostavosti.

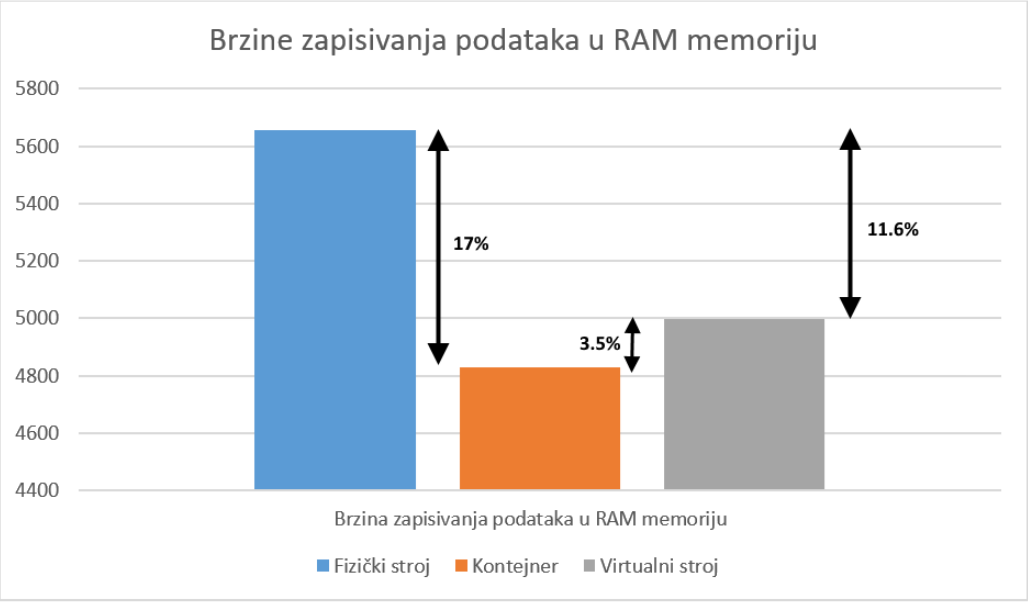
Vezajući se na prijašnje zaključke, relevantnost IoT-a u budućnosti je osigurana, ali zbog pomaka u pristupu ovoj tehnologiji i pojavi Edge computinga postavlja se pitanje da li će isto vrijediti za kontejnere. Trendovi pokazuju da će zahtjevi od rubnih uređaja na mreži rasti i da će se sve više podataka obrađivati lokalno. Kako bi se testirala održivost kontejnerskog pristupa u slučaju kada ćemo imati uređaje koji će podatke obrađivati lokalno, uspoređene su performanse kontejnera sa fizičkim i virtualnim strojevima.

Performanse Docker kontejnera kod CPU intenzivnih zadataka (Slika 7.1) su izvrsne i vrlo blizu onima fizičkog stroja. Obrada podataka, ako ne uzimamo u obzir brzinu pristupa disku, svakako se preporuča u kontejnerima. Iskorištenost procesora ne predstavlja nikakvu prepreku lokalnoj obradi IoT podataka te je bliska nativnim performansama.



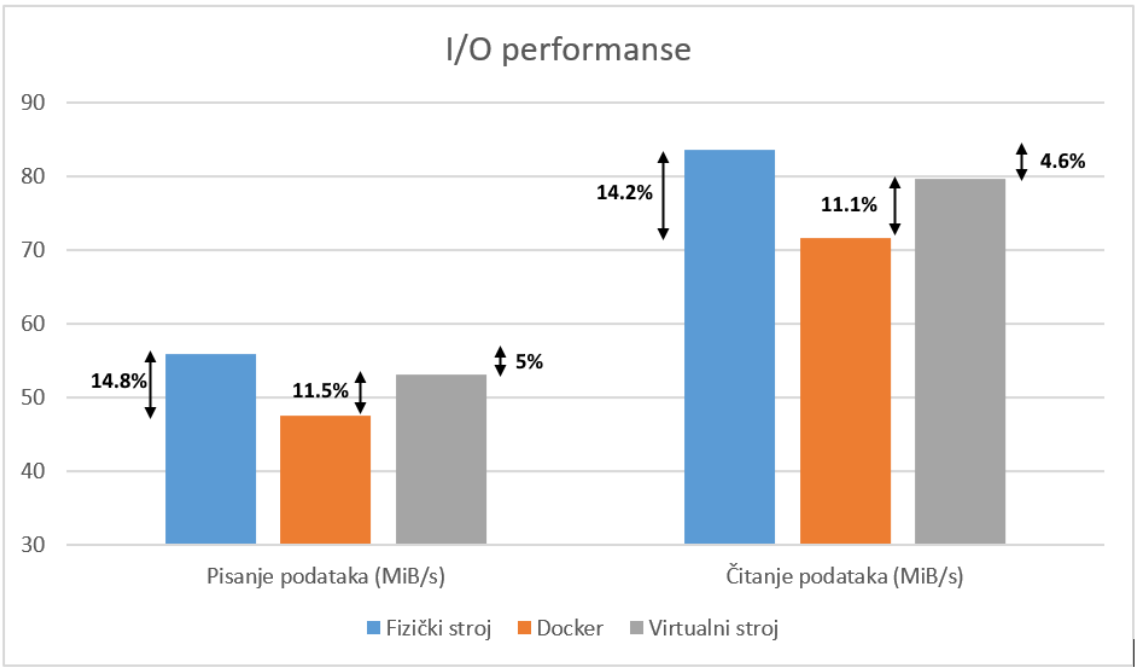
Slika 7.1 Usporedba CPU performansi

Kod mjerenja RAM performansi, osim činjenice da fizički stroj ovdje ima veliku prednost, na osnovi dobivenih podataka (Slika 7.2) ne postoji razlog zašto odabrati jednu tehnologiju umjesto druge. Prednost virtualnog stroja od 3.5% eventualno bi mogla imati važnost kod kritičnih aplikacija koje ionako ne bi stavljali u kontejnere ili na rub mreže. RAM tako ne predstavlja nikakvu prepreku korištenju kontejnera.



Slika 7.2 Usporedba RAM performansi

Zadnja usporedba sastoji se od usporedbe brzine čitanja i pisanja sa medija za pohranu podataka (Slika 7.3). Docker kontejneri u ovom slučaju zaostaju otprilike 11% za virtualizacijom. Razlika se na prvi pogled može činiti znatnom, ali prema mojem mišljenju neće imati negativan utjecaj na zadatke s kakvima se IoT infrastrukture susreću. I/O performanse mogle bi biti kritične ako radimo sa intenzivnim bazama podataka, ali se ponovno nalazimo u situaciji gdje bi baze takve veličine mogle biti od kritične važnosti i nalaziti se u jezgri mreže, ne na njezinom rubu.



Slika 7.3 Usporedba I/O performansi

Iz priloženih podataka lako je zaključiti da su kontejneri više nego adekvatni za obradu IoT podataka, ne samo za implementaciju samih senzora i aplikacija. Konačan zaključan je da kontejnerske tehnologije imaju ogroman potencijal u IoT sektoru i da ne postoje prepreke njihovom širenju. Situacija je baš suprotna, kontejneri bi mogli napraviti znatan doprinos razvoju i standardizaciji IoT-a . Ako u obzir uzmemo i činjenicu da za Docker kontejnersku platformu već sada postoje alati za orkestraciju kao što su Docker Swarm i Kubernetes koji omogućuju upravljanje tisućama kontejnera, budućnost ove tehnologije u IoT-u zasigurno je neizbježna.

Popis kratica

IoT	*Internet of Things*	internet stvari
TCP/IP	*Transfer control protocol/internet protocol*	protokol za kontrolu prijenosa
VM	*Virtual machine*	virtualni stroj
OS	*Operating system*	operativni sustav
REST	*Representational State Transfer*	prijenos reprezentacijskog stanja
API	*Application programming interface*	aplikacijsko programsko sučelje
CLI	*Command line interface*	komandno linijsko sučelje
CE	*Community edition*	javna verzija
EE	*Enterprise edition*	verzija za tvrtke
GPIO	*General purpose input output*	ulaz/izlaz opće primjene
I/O	*Input/output*	ulaz/izlaz
SBC	*Single board computer*	računalo na jednoj matičnoj ploči

Popis slika

Popis tablica

Popis kôdova

Literatura

[1] IEEE IOT COMMUNITY, https://www.ieee.org/membership-catalog/productdetail/showProductDetailPage.html?product=CMYIOT736

[2] JANGLA, S. Accelerating Development Velocity Using Docker, 2018.

[3] HUTTEN, D. Docker: Docker Tutorial for Beginners Build Ship and Run, 2018.

[4] DOCKER PRODUCTS, DOCKER-DESKTOP, https://www.docker.com/products/docker-desktop.

[5] DOCS.DOCKER.COM, WINDOWS INSTALL, https://docs.docker.com/v17.12/docker-for-windows/install/

[6] DOCS.DOCKER.COM, LINUX INSTALL https://docs.docker.com/v17.12/install/linux/docker-ee/ubuntu/#install-using-the-repository

[7] DOCS.DOCKER.COM, SETUP, https://docs.docker.com/v17.12/get-started/

[8] IOTTECHTRENDS, https://www.iottechtrends.com/why-docker-is-useful-iot/.

[9] DOCKS.DOCKER.COM, CONTAINERS: https://docs.docker.com/v17.12/get-started/part2/#your-new-development-environment

[10] SYSBENCH MANPAGE, http://manpages.ubuntu.com/manpages/disco/en/man1/sysbench.1.html

[11] IOT EDGE COMPUTING, http://intelzone.com/iot-sensors-edge-computing/

„Pod punom odgovornošću pismeno potvrđujem da je ovo moj autorski rad čiji niti jedan dio nije nastao kopiranjem ili plagiranjem tuđeg sadržaja. Prilikom izrade rada koristio sam tuđe materijale navedene u popisu literature ali nisam kopirao niti jedan njihov dio, osim citata za koje sam naveo autora i izvor te ih jasno označio znakovima navodnika. U slučaju da se u bilo kojem trenutku dokaže suprotno, spreman sam snositi sve posljedice uključivo i poništenje javne isprave stečene dijelom i na temelju ovoga rada“.

U Zagrebu, 04.09.2019.

Prilog