Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

David Rubin

Zasnova in razvoj oblačne platforme za zajemanje in shranjevanje podatkovnih tokov iz pametnih naprav

diplomsko delo

univerzitetni študijski program
prve stopnje
računalništvo in informatika

Mentor: doc. dr. Sebastijan Šprager

Ljubljana, 2018

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Preučite koncepte integracije naprav IoT na podatkovnem nivoju. Razmislite o primernih možnostih zajemanja in shranjevanja takih podatkov. Razvijte enostavno platformo za integracijo podatkovnih tokov pametnih naprav, njihovo shranjevanje, upravljanje in vizualizacijo. Delovanje pokažite na konkretnih primerih z uporabo treh različnih senzorskih naprav. Razmislite in opredelite koncepte za prenos in skaliranje vaše rešitve v oblačno infrastrukturo.

Zahvala gre vsem, ki so me med študijem podpirali. Predvsem bi se rad zahvalil staršema, sestri, obema dedkoma in babici. Brez vas to delo ne bi obstajalo. Posebna zahvala gre tudi mentorju. Hvala pa tudi prijateljem, ki so mi popestrili študijske dni.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
IoT	Internet of Things	internet stvari
API	Application programming interface	aplikacijski programski vmesnik
HDFS	Hadoop Distributed File System	Hadoop porazdeljen datotečni sistem
YARN	Yet Another Resource Negotiator	Upravljalec virov
SQL	Structured query language	strukturiran poizvedovalni jezik
NoSQL	non/not only SQL	ne/ne samo SQL
I2C	Inter-Integrated Circuit	/
SDA	Serial Data Line	Serijska linija podatkov
SCL	Serial Clock Line	Serijska linija ure
JSON	Javascript Object Notation	Javascript objektna notacija
IMEI	International Mobile Equipment Identity	Mednarodna identiteta mobilne opreme
RAID	Redundant array of independent disks	čezmerno polje samostojnih diskov
RSA	Rivest–Shamir–Adleman cryptosystem	Algoritem za šifriranje z javnim ključem
AWS	Amazon Web Services	Amazonove oblačne storitve
IaaS	Infrastructure as a service	Infrastruktura kot storitev
PaaS	Platform as a service	Platforma kot storitev
SaaS	Software as a service	Programska oprema kot storitev

Povzetek

Naslov: Zasnova in razvoj oblačne platforme za zajemanje in shranjevanje podatkovnih tokov iz pametnih naprav

Avtor: David Rubin

Pri nalogi smo zasnovali in izdelali platformo, ki omogoča zbiranje in shranjevanje podatkovnih tokov iz pametnih naprav. Med pametne naprave sodijo tako mobilni telefoni, kot tudi naprave interneta stvari. Sposobne so generirati veliko podatkov v realnem času, katere pa je težko obvladati, torej ne le zajeti ampak tudi shraniti za kasnejše analize. Pomemben je tudi hitri dostop do teh shranjenih podatkov. Z nalogo smo razvili šibko sklopljeno platformo, ki omogoča porazdeljeno shranjevanje podatkov in nudi možnost poizvedovanja po njih. Razvili smo tudi IoT senzorske aplikacije, s katerimi smo pokazali, da naša platforma deluje s heterogenimi podatkovnimi tokovi. Platformo smo na koncu še preizkusili in omogočili vizualizacijo zajetih podatkov.

Ključne besede: oblak, internet stvari, velepodatki, pametne naprave, porazdeljena shramba podatkov, Hadoop.

Abstract

Title: Designing and developing a cloud based platform for capturing and storing data streams from smart devices

Author: David Rubin

In this thesis we designed and developed a cloud based platform which offers storage for data gathered by smart devices. Smart devices can include smartphones, the devices of the internet of things and many more. They are capable of generating a large amount of data in real time and it is a challenge to not only capture the data, but to also store it for a possible later analysis. It is also important that we have quick access times to the stored data. In this thesis we developed a loose-coupled platform that can offer distributed storage for the big data from the devices and also an option for querying that data. We also developed some IoT sensor applications that demonstrate efficient operability of our platform on heterogeneous data flows. At the end we tested the platform and plotted some of the data collected.

Keywords: cloud, internet of things, big data, smart devices, distributed storage, Hadoop.

Poglavje 1

Uvod

Zamislimo si, da imamo doma pametno budilko, ki je povezana na internet. Budilka ima zmožnost, da nas vsak dan zbudi od ravno pravem času, da ne zamudimo v službo ali v šolo. Vremenska postaja ji pove, da dežuje in da raje vzemimo avtobus namesto kolesa. Povezani avtomobili ji sporočajo, da so zastoji in da se moramo na pot odpraviti prej. Še več, lahko obvesti naš pametni kavni avtomat in opekač, naj nam ob pravem času pripravita zajtrk. Zamislimo si, da nam hladilnik lahko pove, česa nam primankuje in bi bilo potrebno kupiti. Kaj če smo hladilniku naročili, naj nam sam naroča mleko, ko nam ga zmanjka? Podobne stvari se že dogajajo, v prihodnosti pa bo tega še več [13].

S pametnimi napravami se dandanes srečujemo vsepovsod. Med pametne naprave štejejo elektronske naprave, ki so večinoma povezane z drugimi napravami preko različnih brezžičnih protokolov. Večina ljudi ima vsaj eno takšno v svojem žepu, to je mobilni telefon, poznamo pa tudi naprave interneta stvari (v nadaljevanju IoT). IoT bi lahko opisali kot vse naprave, ki so zmožne povezave do interneta [13]. Napravam je tudi skupno, da so sposobne zbirati podatke o svoji okolici s pomočjo senzorjev. Večina mobilnih telefonov ima senzor za svetlobo, ki samodejno uravnava svetlost ekrana, prav tako v njem najdemo pospeškometer, ki meri pospešek v vse tri smeri v prostoru in še mnoge druge. Če pogledamo naprave IoT, pa je izbira senzorjev še večja.

Lahko si zgradimo vremensko postajo in beležimo temperaturo, zračni tlak, hitrost in smer vetra, količino padavin ... Morda si kdo želi sistem za zalivanje rastlin in bo tako beležil podatke o vlažnosti zemlje, jakosti svetlobe in kdaj ter kako intenzivno je bila posamezna rastlina zalita. Do interneta lahko danes dostopamo od praktično vsepovsod, saj 84% populacije živi v območju pokritim z mobilnim širokopasovnim omrežjem. Prav tako pa se znižujejo stroški dostopa do interneta [5]. Nižajo se tudi cene tehnologij in vedno več naprav ima vgrajene senzorje in sprejemnike za brezžično omrežje. Vse to skupaj je ustvarilo odlično okolje za rast IoT. Revija Forbes je v preteklem letu objavila članek, v katerem navajajo, da je 60% organizacij pričelo z uvajanjem naprav IoT [10]. Prav tako napovedujejo, da bo število povezanih IoT naprav naraslo na 20.8 milijard do leta 2022 [2]. Pri primerjavi običajnih relacijskih bazah ugotovimo, da bi z njihovo uporabo hitro prišli do performančnih težav in tudi do zgornje meje velikosti shrambe [23]. Ali je potem sploh mogoče obvladati podatke, ki bi jih vse te naprave proizvedle?

Kot odgovor na to vprašanje smo zasnovali in razvili platformo, ki omogoča shranjevanje večjih količin podatkov in kasnejše poizvedovanje po njih. Tega izziva smo se lotili iz nule, torej sami smo si zamislili tri primere senzorskih naprav, katere smo seveda implementirali, vzpostavili pa smo tudi strežnik, na katerem teče Hadoop in prehod za podatke v obliki spletnega strežnika. V nadaljevanju je besedilo razdeljeno na več poglavij. V poglavju 2 opisujemo problem, ki ga rešujemo. V poglavju 3 so opisani primeri aplikacij, ki zbirajo in pošiljajo podatke. V poglavju 4 si bomo ogledali Apache Hadoop, iz katerih modulov je sestavljen, kako deluje in našo postavitev te arhitekture. V poglavju 5 je predstavljen naš prehod za podatke, v poglavju 6 pa podamo še primer MapReduce programa in vizualizacijo zbranih podatkov.

Poglavje 2

Arhitektura platforme

Živimo v času, ko imamo s pomočjo svetovnega spleta dostop do ogromnih količin podatkov. Podatki tako imenovanega digitalnega vesolja se vsako leto skoraj podvojijo. Leta 2013 smo imeli 4.4 zetabajtov podatkov. Če predstavimo malce drugače, je vsako gospodinjstvo v povprečju na leto proizvedlo dovolj podatkov, da napolni 65 IPhonov (32GB različic). Napovedi kažejo, da bi se naj količina povečala na 44 zetabajtov do leta 2020 [11]. Vsi ti podatki pa ne pripadajo samo velikim podjetjem in organizacijam. Vzemimo za primer fotografije. Doma imamo album, kjer je pred časom moja mama združila vse starejše, po večini še črno-bele, fotografije naših prednikov in jim dopisala kratke opise. Album ima okoli 100 strani in na vsaki se najde do 5 fotografij. Fotografije segajo od leta 1900 in se razprostirajo po naslednjih 80-90 let. Povprečno so torej na leto naredili pet fotografij. Danes pa imamo za zadnjih deset let pet albumov, poleg njih pa še skoraj tri popolnoma zasedene trde diske s kapaciteto 500GB, kjer so shranjene še preostale fotografije, ki niso prišle v izbor za tisk.

Z vsemi temi podatki, ki jih neprestano zbiramo o okolici, lahko pridemo do precej zanimivih ugotovitev. Kot primer lahko omenimo prepoznavo osebe samo s podatki pospeškometra na mobilnem telefonu [21]. V zdravstvu je IoT odprl popolnoma nove zmožnosti spremljanja pacientov. Spremljamo lahko stanje pacientov, pošiljamo podatke senzorjev v oblak in posredujemo ugoto-

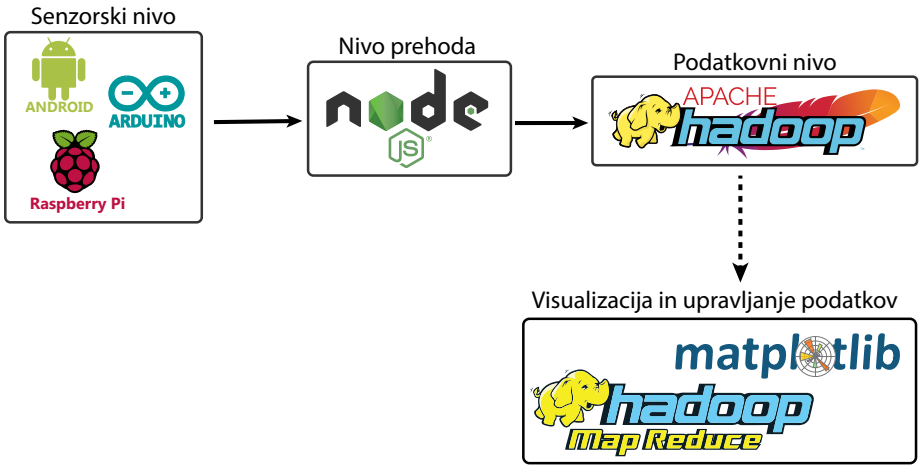
vitve ustreznim zdravstvenim uslužbencem za nadaljnje ukrepe [6]. Kamorkoli pogledamo, najdemo možnost zbiranja podatkov za analize in raziskave, ki bi morda izboljšale oziroma olajšale trenuten način življenja.

Za analizo podatkov ni dovolj, da nam jih senzorji le prikazujejo, potrebno je imeti dovolj veliko količino shranjenih podatkov, kar pa je pri večjih razsežnostih lahko problem. Ena izmed težav je čas dostopa do podatkov. Pomembno je, da prejete podatke hitro shranimo, prav tako pa da lahko za namene učinkovitega procesiranja do njih tudi hitro dostopamo. Rešitev je skaliranje platforme oziroma posameznih nivojev, to pa dosegamo s pomočjo oblaka. Da bi vse te podatke lahko shranili, tako da so kasneje dostopni za poizvedovanje, so se namesto tradicionalnih relacijskih podatkovnih baz uveljavile NoSQL podatkovne baze. Takšne podatkovne baze se od tradicionalnih razlikujejo po načinu shranjevanja podatkov. Ti se ne shranjujejo v tabelah, povezanimi preko relacij, ampak po različnih principih. Med drugim poznamo ključ-vrednost (angl. *key-value*), dokumente (angl. *document*) in stolpične (angl. *column*). Med slednje spada tudi baza HBase, ki je v uporabi pri Hadoopu, o kateri bomo povedali več v prihajajočih poglavjih. V tem delu bomo torej razvili platformo s pomočjo odprtokodnega ogrodja Apache Hadoop, ki se uporablja za procesiranje velikih podatkov. Izbran je bil predvsem iz razloga, ker je možno vzpostaviti delujočo aplikacijo na običajnih strežnikih, torej ni potrebe po precej dragi strežniški infrastrukturi, ki se uporablja v podatkovnih centrih. Z uporabo lastnega strežnika imamo tudi popolni dostop do postavitve.

2.1 Razdelitev na podprobleme

Da prikažemo šibko sklopljenost sistema, smo si celotno platformo razdelili na tri nivoje. Vsak nivo stoji sam kot celota in za prehode oziroma komunikacijo med njimi so definirana ustrezna pravila. Posledično se posamezen nivo v sistemu lahko spreminja, dokler upošteva pravila za komunikacijo z drugimi. Implementacije so lahko v različnih tehnologijah, prav tako se lahko

dodela že obstoječo aplikacijo, da se vključi v platformo. Na sliki 2.1 je prikazan diagram nivojev naše rešitve vključno z tehnologijami, ki so uporabljene znotraj njih.



Slika 2.1: Diagram nivojev jedra naše rešitve s ključnimi tehnologijami za izdelavo le-teh. Vir: lasten, 2018

Prvi nivo so pametne naprave (senzorski nivo). To so prej omenjene naprave IoT in pa mobilni telefoni. Njihova naloga je zbiranje podatkov o okolici in posredovanje le-teh v nadaljnji prehod. Nivo je omejen le z pogojem, da mora posamezna naprava imeti možnost pošiljanja podatkov do nivoja prehoda. Struktura podatkov je načeloma lahko poljubna, mi smo se odločili za JSON in definirali standardno strukturo za vsak primer uporabe. Za ta nivo smo z primeri uporabe pokazali tudi možnost zbiranja heterogenih podatkovnih tokov.

Drugi nivo (nivo prehoda) je prehod med senzorski napravami in shrambo. Tudi ta nivo je neodvisen od tehnologije v kateri je implementiran. Shramba ima na voljo spletni vmesnik za upravljanje, tako da je izbira za prehod, ki je pri nas spisan v ogrodju NodeJS, popolnoma po želji razvijalca. Za tehnologijo NodeJS smo se odločili, ker je relativno enostavna in učinkovita za uporabo in imamo nekaj izkušenj pri njeni uporabi. Ta del usmerja podatke senzorjev, da se pravilno shranijo.

Postavitev ogrodja Hadoop (podatkovni nivo) je tretji nivo. Nudi shrambo za podatke na gruči računalnikov. Zaradi omejitev strojne opreme, smo namesto običajne gruče računalnikov uporabili samo enega in tako imenovani *single-node setup*, kar v grobem pomeni samo eno vozlišče. Na enem računalniku smo tako postavili vse storitve potrebne za delovanje Hadoopa. Postavitev na gručo računalnikov oziroma skaliranje naše implementacije deluje po enakem pristopu. Bistvena razlika pri uporabi gruče bi bili stroški, torej pojavi se še vprašanje, koliko strojne opreme potrebujemo za učinkovito uporabo nivoja v naši platformi.

Za potrebe po demonstraciji delovanja smo dodali še četrti nivo (upravljanje in vizualizacija podatkov). V diagramu je povezan s prekinjeno puščico, saj bi ta nivo načeloma implementiral uporabnik naše platforme, torej nekdo, ki nad podatki shranjenimi preko naše platforme izvaja svoje analize. Da pokažemo, da naša platforma deluje, smo podatke na tem nivoju grafično prikazali in predstavili.

Poglavje 3

Senzorski nivo

Ponavadi si želimo čim več in čim bolj natančnih meritev. V CERN-u (Evropska organizacija za jedrske raziskave) za primer vsako sekundo senzorji proizvedejo en petabajt podatkov. Zaradi fizičnih omejitev strojne opreme, nato z določenimi metodami izluščijo samo pomembne podatke in tako na dan proizvedejo „le“ nekaj deset petabajtov podatkov. Tako so 29. junija leta 2017 presegli 200 petabajtov trajno shranjenih podatkov v svojem centru [17]. To so skrajni primeri, v naši platformi se takšnim številkam nikakor ne moremo niti približati, lahko pa pokažemo principe, po katerih se ti podatki zbirajo in shranjujejo. V naslednjih podpoglavjih bomo opisali naše primere aplikacij in naprav, katerih namen je izključno zbiranje podatkov. Odločili smo se, da izdelamo več različnih prototipov in pokažemo, da se v platformo lahko vključi kakršnakoli naprava. V našem primeru so to naprave, ki temeljijo na Raspberry Pi, Arduinu in mobilni telefoni z operacijskim sistemom Android. Podatke pošiljamo in shranjujemo v formatu JSON. To je lahek in tudi dobro berljiv format, najpomembneje pa je to, da se odlično obnese z našim prehodom, ki je realiziran v jeziku JavaScript. Tako smo lahko z malo dodatnimi koraki dosegli predpomnjenje posameznih meritev kar na nivoju prehoda. Slednje je bilo koristno pri Arduinu, ki zaradi majhnega pomnilnika ni mogel predpomniti večje količine podatkov. Pri vseh treh primerih smo se odločili, da med podatke vključimo enolični identifikator, časovni po-

snetek dogodka, nekakšen identifikator za senzor in vrednosti za senzor. Po tej strukturi se ravnamo pri Arduinu (rastlina) in Raspberry Pi (vremenska postaja). Pri mobilni aplikaciji smo tej osnovni strukturi podatkov senzorja dodali še podatke o stanju baterije naprave in morebitni brezžični omrežni povezavi.

3.1 Rastlina, ki se zaliva sama

Spoznajte mojo rastlino, ki sem jo poimenoval Leon. Spada v rod spatifilov (znanstveno ime *Spathiphyllum*) in za preživetje ne potrebuje veliko sončne svetlobe ali vode. Kljub temu so pri njegovi negi nastajale težave, saj je med študijem prebival v študentskem domu, ki je okoli sto kilometrov oddaljen od mojega stalnega prebivališča. Med vikendi kot tudi kakšen teden med počitnicami je ostajal sam in noben (niti sostanovalci) ni imel možnosti, da bi ga zalival. Odločili smo se, da enkrat in za vselej rešimo težave z zalivanjem. Leonu smo omogočili, da se zalije sam. S pomočjo naprav IoT imamo sedaj rastlino, ki se zalije sama in podatke o njenem stanju lahko spremljamo tudi na daljavo. Z analizo teh podatkov bi lahko določili interval zalivanja, oziroma lahko bi napovedovali, kdaj je najboljši čas za zalivanje. Vse skupaj smo realizirali s pomočjo razvojne ploščice Arduino. Arduino je projekt, ki vsebuje tako strojno kot programsko opremo in temelji na enostavnosti za uporabo. Uporablja se v veliko projektih, kjer je potreba po krmiljenju razne elektronike. Na Arduino razvojne ploščice lahko priključimo vse od svetlečih diod do 3D tiskalnikov. V nadaljevanju sta v podpoglavjih opisana fizični del sistema in pa programerski del.

3.1.1 Strojna oprema

Za realizacijo sistema za zalivanje smo vzeli senzorje za vlago zemlje, ploščico Arduino, vodno črpalko in komponento, ki skrbi za povezavo s brezžičnim omrežjem (oznaka slednje je ESP8266). Na nasprotnih straneh lončka smo vstavili po en senzor vlage, vsak pa ima tudi svoj dovod vode iz vodne črpalke.

Za lažje spremljanje dogajanja je na Arduino priključen tudi zaslon, ki izpisuje podatke senzorjev, čas delovanja in čas pretečen od zadnjega zalivanja. Za vzpostavitev povezave med temi komponentami so potrebne še nekatere druge, kot so recimo uporniki ali pa kondenzatorji. Seznam vseh uporabljenih komponent je prikazan v tabeli 3.1.

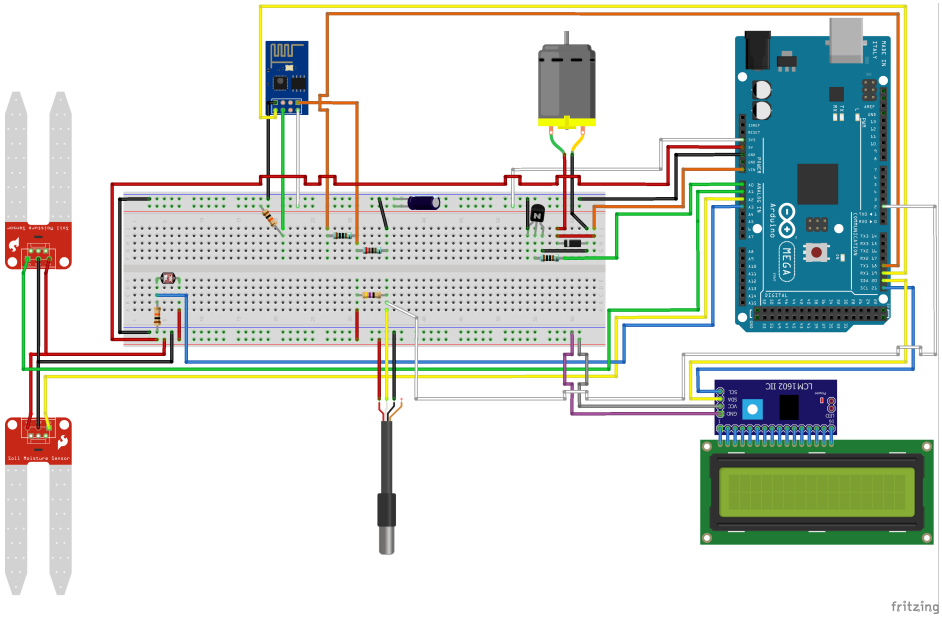
Na senzorju vlage so označeni štirje pini. Napajanje smo povezali na 5 V in senzor ozemljili. Pin A0 pa smo povezali z enim izmed analognih pinov na Arduinu. V našem primeru smo namenili pina A1 in A2 za vsak senzor. Črpalka ima le dve žici, napajanje smo s pomočjo tranzistorja povezali na vhodno napetost (kar je 12 V zaradi adapterja, ki napaja sistem) in na enega izmed analognih pinov na Arduinu. Izbrali smo si A0. Pri črpalki smo vstavili še diodo, da pri vklopu in izklopu ne bi mogel tok teči v obratno smer in povzročiti poškodbo na črpalki. Svetlobno celico smo povezali na 5 V in drugi pin s pinom A3 na Arduinu. Na drugi podatkovni pin smo dodali še *pull-down* upor. Senzor za temperaturo smo povezali na 5 V in ozemljitev, podatkovni pin pa z digitalnim pinom D2 in mu dodali še 4.7 kΩ *pull-up* upor. Zaslonu smo dodali I2C komponento in potem še povezali SDA, SCL, napajanje in ozemljitev z enakimi na Arduinu.

Na ESP8266 verzije 1 smo za boljšo stabilnost in lažje delo pri programiranju predhodno naložili NodeMCU strojno-programsko opremo. Pri vezavi je bilo potrebno paziti, saj ESP8266 obratuje na 3.3 V, torej ga 5 V, kar je izhod na pinih Arduina, lahko poškoduje. Napajalni pin smo priključili na 3.3 V in ESP8266 ozemljili. Pin označen s *chip power down* smo preko 10 kΩ upora povezali na 3.3 V. Pin *reset* smo med navadnim obratovanjem pustili plavajoč. Za komunikacijo smo si izbrali pine *Serial1* na Arduinu. Oddajni pin na ESP8266 smo brez težav povezali s sprejemnim pinom na Arduinu. Oddajni pin na Arduino pa kot izhod pošilja 5 V, zato smo si naredili enostaven pretvornik napetosti s pomočjo dveh upornikov, v našem primeru z 1 kΩ in 2 kΩ uporom. ESP8266 se napaja preko Arduina, in ker pini ne dosegajo dovolj velikega izhodnega toka za običajno obratovanje modula (tudi do 250 mA), smo dodali še kondenzator s kapaciteto 1000 µF. Celotno vezje

Oznaka opreme	opis
Arduino Mega 2560	ploščica, ki uravnava in nadzira delovanje sistema
ESP8266-01	skrbi za povezljivost v omrežje in pošilja podatke
16x2 zaslon	prikazuje podatke o trenutnem stanju sistema, prav tako je nadgrajen z I2C vmesnikom
Senzor za vlago	dva, ki merita na nasprotnih straneh lončka
DS18B20	senzor temperature
črpalka za vodo	potopna, ima 12 V DC motor
2N2222A	tranzistor za dovajanje električnega toka do črpalke
1N4001	dioda, ki skrbi, da tok teče le v eno smer
kondenzator	skrbi, da ESP8266 dobi dovolj toka
svetlobna celica	lahko zaznava spremembe svetlobe preko sprememb upora
senzor temperature	v obliki ene žice meri temperaturo okolice
uporniki	*pull-up/-down* upor, tudi za pretvorbo iz 5 V na 3.3 V
žice	/
plošče za vezavo	na njih bomo povezali komponente med sabo
12 V adapter	pretvarja 220 V iz zidne vtičnice na 12 V s 3 A toka.

Tabela 3.1: Strojna oprema v končni verziji sistema.

je prikazano na sliki 3.1.



Slika 3.1: Skica povezav med komponentami za rastlino, ki se zaliva sama. Vir: lasten, 2018

3.1.2 Programiranje Arduina

Arduino je programiran s pomočjo 5 knjižnic in sicer *Wire*, *LiquidCrystal_I2C*, *ArduinoJson*, *OneWire* in *DallasTemperature*. Prva se uporablja za komunikacijo z zaslonom, druga poenostavi delo z modulom na zaslonu, tretja podatke oblikuje v veljaven JSON, zadnji dve pa poenostavita branje podatkov iz temperaturnega senzorja. Za začetek smo inicializirali vse potrebne spremenljivke in v funkciji *setup* nastavili načine pinov, pri čemer so senzorji vhodni, edini izhodni pa je A0, torej pin za vklop črpalke. Pričnemo serijsko komunikacijo z modulom ESP8266 na pinih Serial1 in počakamo, da se ta modul postavi v stanje pripravljenosti in nam pove trenuten čas. Več o tem modulu bomo povedali malce kasneje. V glavni zanki se vsake tri sekunde zamenja operacija. Spisali smo več zank, takšne kot so prikazane na kodnem bloku 3.1, kjer se vsaka zanka ponavlja tri sekunde preden preide

na naslednjo. Znotraj zanke se opravljajo operacije beleženja, pošiljanja in prikazovanja vrednosti senzorjev, pri čemer je med vsakim prehodom dodan zamik, saj nekateri izmed senzorjev fizično niso zmožni hitreje pridobivati podatke. Za primer vzemimo senzor za temperaturo, ki lahko poda vrednost vsakih 750 ms. Za zalivanje se vzame povprečje obeh senzorjev vlage, vendar se lahko rastlina zaliva le enkrat na 60 sekund in to za vnaprej določenih 5 sekund. To omejitev smo uvedli, da ne pride do prekomernega zalivanja, če se rastlina preveč izsuši v času, ko sistem ni priključen v električno omrežje, saj se vlaga meri v zgornjih 5 cm zemlje, voda pa na začetku odteče do dna. Poleg tri sekundnih ciklov beleženja podatkov, sta vključena tudi tri sekundna cikla, kjer se na zaslon izpisujejo podatki o času delovanja in o času pretečenem od zadnjega zalivanja. V zankah, kjer se zbirajo podatki, se po vsakem branju meritve pošlje prebrane vrednosti v ustrezni JSON obliki v prehod. Posamezna meritev je tako JSON objekt, kateremu se poleg imena in vrednosti senzorja doda še časovni posnetek dogodka in enolični identifikator naprave. V času izdelave nisem imel pri roki komponente, ki je namenjena beleženju časa, zato smo uporabili ESP8266, da nam pove čas s pomočjo omrežne povezave, katerega nato povečujemo z Arduinovo vgrajeno funkcijo *millis*, ki vrača milisekunde od zagona naprave.

```
void loop()
{
  unsigned long displayChange = millis();
  while ((millis() - displayChange)/1000UL <= 3UL )
  {
    // Branje podatkov senzorja
    ...
  }
  ...
  // Enake zanke, ki izvajajo druge operacije
}
```

Kodni blok 3.1: Izgled glavne zanke v Arduinu.

ESP8266 pa ima naloženo že prej omenjeno strojno-programsko opremo NodeMCU, katero lahko programiramo v jeziku Lua. Naprej se zažene datoteka *init.lua*, kjer se povežemo z dostopno točko. Dodan je varnostni ukrep, da preden začne izvajati glavni program, počaka pet sekund. Če bi v tem glavnem programu prišlo do neskončne zanke zaradi napak v kodi, bi se modul ob vsakem zagonu prenehal odzivati na uporabnikove ukaze in potrebno bi bilo na novo naložiti strojno-programsko opremo. Iz tega razloga smo dodali ta varnostni ukrep. Za vsem tem se v novi datoteki (*plant.lua*) prične izvajati glavni program, ki je sestavljen iz dveh funkcij. Prva je *getCurrentTime*, ki s pomočjo vmesnika spletne strani `timezonedb.com` pridobi trenuten čas v obliki sekund pretečenih od 1.1.1970. Funkcija *postData* pa pošlje podane argumente kot telo zahtevka na strežnik, torej v nivo prehoda. Združevanje posameznih meritev v seznam se prepusti nivoju prehoda, saj je pomnilnik na Arduinu dokaj majhen. Vsa komunikacija med ESP8266 in Arduinom poteka preko serijske linije s hitrostjo 115200 bitov na sekundo.

3.2 Enostavna vremenska postaja

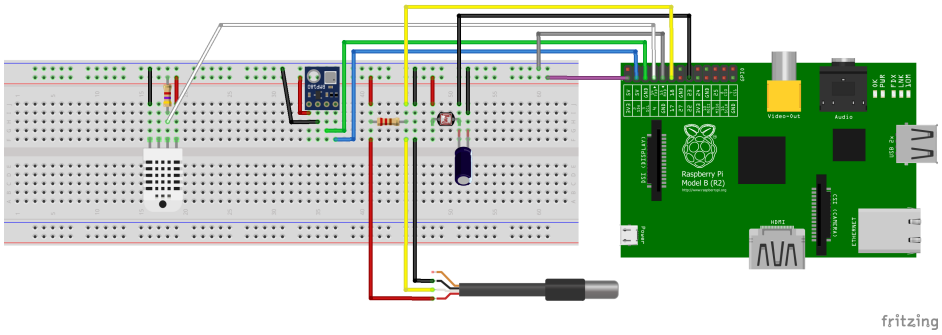
Za naslednji primer smo se odločili, da s pomočjo RaspberryPi ploščice zgradimo enostavno vremensko postajo, ki beleži vlago v ozračju, temperaturo okolice, zračni tlak, nadmorsko višino in količino svetlobe za morebitno ugotavljanje sončnih dni. Za izdelavo smo uporabili RaspberryPi Model B prve generacije. Senzorji so bili sledeči: DHT22 za temperaturo in vlago, BMP180 za zračni tlak, nadmorsko višino, vodoodporni senzor za temperaturo DS18B20, ter svetlobno celico, za ugotavljanje količine svetlobe.

Senzor temperature in vlage smo povezali s pinom GPIO4, dodali pa smo še *pull-up* upor. Pri senzorju za svetlobo oziroma fotocelici, smo en pin povezali s 3.3 V, drugega pa z GPIO22 in preko 1 µF kondenzatorja do ozemljitve. Senzor za tlak in nadmorsko višino pa smo povezali s SDA in SCL. Pri senzorju DS18B20 je bilo potrebno naložiti še gonilnik, preden smo lahko komponento uporabili. Po privzetem uporabljen gonilnik posluša na drugem pinu,

tako da smo v */boot/config.txt* dodali vrstico *dtoverlay=w1-gpio,gpiopin=17*. Slednja pove, naj gonilnik *w1-gpio* uporablja pin 17, kamor smo priključili DS18B20. Zagnali smo še komponento, ki nam omogoča branje vrednosti. Ukazi, ki vse to dosežejo so vidni na kodnem bloku 3.2. Shema komponent je priložena na sliki 3.2.

```
$ printf "#Doda OneWire na pin 17\ndtoverlay=w1-gpio,
    gpiopin=17\n" >> /boot/config.txt
$ modprobe w1-gpio
$ modprobe w1-therm
```

Kodni blok 3.2: Ukazi potrebni za delovanje DS18B20 na RaspberryPi.



Slika 3.2: Skica povezav med komponentami za vremensko postajo. Vir: lasten, 2018

Program, ki se izvaja za zbiranje podatkov, je napisan v jeziku Python. Poleg standardnih knjižnic smo uporabili še dve za učinkovitejše branje senzorjev, to sta *Adafruit_Python_DHT* in *Adafruit_Python_BMP085*. Za fotocelico smo spisali lastno funkcijo, ki šteje koliko časa je vhod postavljen na nizko vrednost. Izkazalo se je, da lahko tako zaznavamo spremembe svetlobe, vendar bi za pretvorbo v standardne enote svetlosti potrebovali možnost kalibracije, katera pa v času izdelave ni bila na voljo. Iz podatkov zopet tvorimo JSON, tokrat ima objekt polje za enolično identifikacijo in polje z imenom *data*. Slednje je seznam meritev, pri čemer je vsaka izmed njih sestavljena iz časovnega posnetka, imena senzorja in do tri vrednosti za vsak senzor.

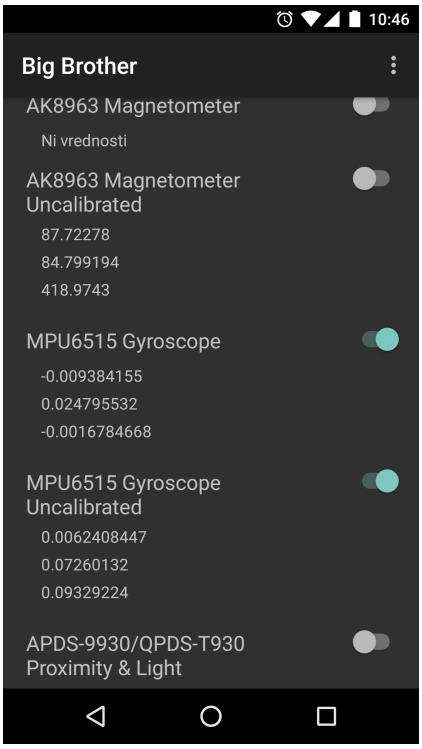
Program tudi predpomni 1125 vrednosti in jih nato pošlje v nivo prehoda. Z merjenjem trajanja zanke, kjer se izvajajo branja senzorjev, smo prišli do povprečnega časa 1.6 sekunde za en obhod. Če to pomnožimo z velikostjo predpomnilnika dobimo časovni interval pošiljanja na vsakih 30 minut, kar se nam zdi primerno za zbiranje testnih podatkov.

3.3 Mobilna aplikacija

Za zadnji primer pa smo še spisali aplikacijo za mobilni operacijski sistem Android, ki beleži celo kopico podatkov o mobilni napravi in posledično uporabniku. Sodeč po raziskavi podjetja Gartner je v prvem četrtletju leta 2017 kar 86% prodanih novih telefonov imelo nameščen ta operacijski sistem [9]. Tako imamo na voljo ogromno uporabnikov, katere lahko profiliramo, torej ugotavljamo kje se zadržujejo, merimo fizično aktivnost ... S takšnimi podatki lahko na primer uporabniku posredujemo bolj specifične oglase, lahko pa tudi odkrijemo in nadzorujemo navade posameznikov. Aplikacija vzame vse senzorje, ki so napravi na voljo, in jih izpiše na ekran s stikali za vklop. Poleg tega pa še beleži vse dotike po zaslonu, kot tudi stanje baterije in podatke o morebitni aktivni brezžični omrežni povezavi.

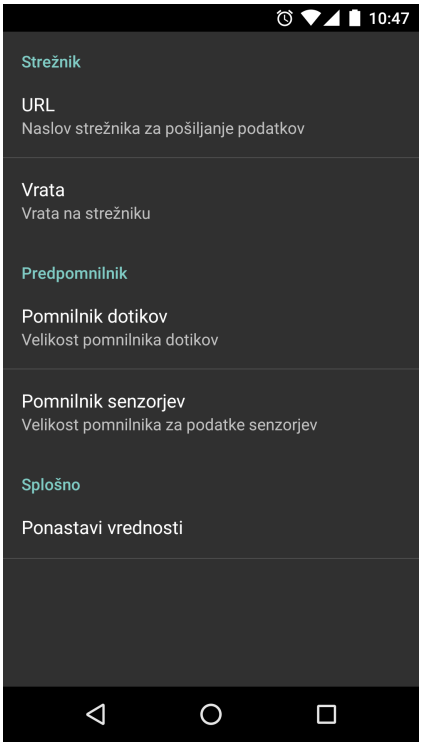
Za realizacijo mobilne aplikacije smo uporabili knjižnico *Volley*, ki vpelje vrsto za pošiljanje datotek preko omrežne povezave. Spisali smo lasten *GsonRequest*, ki naš Java objekt s podatki naprave pretvarja v JSON in ga pošlje na ustrezen naslov do našega prehoda. Posamezen JSON objekt za meritev je sestavljen iz enoličnega identifikatorja, ki je pri nas IMEI (mednarodna identiteta mobilne opreme), časovnega posnetka, imena senzorja, tipa senzorja, do tri vrednosti za senzor in podatki o bateriji in morebitni brezžični povezavi. Podatki o bateriji so JSON objekt s stopnjo napolnjenosti in podatkom o morebitnem napajanju. Če se naprava napaja je še dodano ali tok prihaja preko povezave USB ali zidne vtičnice. Podatki o brezžični povezavi so prav tako JSON objekt, kjer se zapiše ali je povezava vzpostavljena, če je, so dodana še polja za moč signala in ime dostopne točke (angl. *service set*

identifier oziroma *SSID*). Za prikaz vseh senzorjev na ekranu se uporablja *RecyclerView*, kateri izpiše ime senzorja, poleg imena pa se nahaja stikalo za vklop tega senzorja. Če je stikalo vključeno, se pod njim pričnejo prikazovati in osveževati vrednosti, ki jih senzor vrača. Če senzor še ni bil vklopljen oziroma ob zagonu aplikacije, pa namesto samih vrednosti piše „Ni vrednosti“. Na sliki 3.3 je prikazan izgled aplikacije. V grafični vmesnik smo dodali še nekaj nastavitev, katerih namen je lažje testiranje in uporaba aplikacije. Te nastavitve z opisi so prikazane na sliki 3.4.



Slika 3.3: Izgled vmesnika mobilne aplikacije za beleženje podatkov. Vir: lasten, 2018

Dotiki po zaslonu se beležijo vedno, ko je aplikacija aktivna. Dogodke dotikov obravnavamo kot senzorske in uporabljamo enako JSON strukturo za shranjevanje. Dodan je le tip dotika, ki je lahko na primer samo pritisk, ali pa uporabnik vleče po zaslonu. Preden se pošljejo, se podatki nabirajo v medpomnilniku. Ta počaka, da se združi nekaj dogodkov (privzeto 12.000),



Slika 3.4: Nastavitve dostopne znotraj aplikacije. Vir: lasten, 2018

nato pa vse skupaj pošlje v naslednjo točko, to je prehod. Knjižnica Volley, ki se uporabi za pošiljanje meritev, se obnese odlično pri pošiljanju majhnih količin podatkov, ima pa težave pri pošiljanju večjih, saj objekte prepiše v pomnilnik (angl. *in-memory*). Posledično se upočasni delovanje aplikacije med pošiljanjem in z nekaj poskusi smo določili mejo 12.000 meritev, kjer se aplikacija še dobro odziva. Pri senzorjih smo uporabili najkrajši možen zamik med branjem vsake vrednosti, da pridobimo čim več podatkov o vsakem uporabniku.

Poglavje 4

Podatkovni nivo

Kot smo že večkrat omenili, smo za shranjevanje podatkov uporabili Apache Hadoop. Poglejmo si najprej kratko zgodovino projekta. Projekt *Nutch* so leta 2002 želeli uporabiti kot iskalnik po spletu. Njegovi ustanovitelji so dokaj hitro ugotovili, da se projekt slabo skalira za iskanje po milijardah strani. Idejo za Hadoop so nato dobili v članku, kjer je bil opisan *Google distributed filesystem* [20]. Leto kasneje je Google predstavil še MapReduce programski model [14]. Tako je leta 2006 ustanovitelj projekta Doug Cutting združil arhitekturi in ustanovil Hadoop. Kot zanimivost naj omenimo, da ime Hadoop izhaja iz plišaste igrače, ki jo je poimenoval sin ustanovitelja. K nadaljnjemu razvoju je veliko prispevalo podjetje Yahoo!, kjer so med drugim leta 2009 uspeli razvrstiti 1 terabajt podatkov v 62 sekundah s pomočjo Hadoopa [19]. Leta 2011 je Yahoo! imel Hadoop gručo sestavljeno iz 42.000 vozlišč. Leta 2012 pa je Facebook oznanil, da imajo na voljo 100 petabajtov shrambe v Hadoop gruči [3]. V nadaljevanju bomo na kratko opisali posamezne komponente, ki so v uporabi, nato pa bomo še predstavili našo postavitev.

Apache Hadoop projekt razvija odprtokodno programsko opremo za zanesljivo, skalabilno in distribuirano procesiranje [1]. Slednje omogoča tudi na večjih podatkovnih množicah, pri tem pa uporablja enostavne programske modele. Kar je pa za nas še predvsem pomembno, postavimo ga lahko na samo enem ali pa ga skaliramo na gručo strežnikov. Če si bralec želi izvedeti

več o samem delovanju in arhitekturi Apache Hadoopa priporočamo knjigi *Hadoop: The Definitive Guide* in *Hadoop Operations* (vira [22] in [8]), na kateri se skozi naslednja poglavja navezujemo tudi sami.

V osnovi je Hadoop sestavljen iz štirih večjih modulov: *Hadoop Common*, *Hadoop Distributed File System* (v nadaljevanju HDFS), *Hadoop YARN* in *Hadoop MapReduce*. Prvi izmed naštetih je imenovan tudi jedro Hadoopa, saj vsebuje knjižnice in orodja, ki podpirajo delovanje ostalih modulov. Ta modul vsebuje tudi datoteke in skripte, ki so potrebne za zagon samega Hadoopa. Kot je že iz imena razvidno, je HDFS modul, ki nudi porazdeljeno shrambo z visoko propustnostjo dostopa do podatkov. Porazdeljena shramba pomeni, da so podatki shranjeni v omrežju na več napravah, kar pa seveda lahko pripelje do težav. Ena izmed največjih je preprečevanje izgube podatkov, v primeru da eno izmed vozlišč preneha delovati. Več o samem sistemu si lahko preberemo v podpoglavju 4.1. YARN je Hadoopov sistem za upravljanje virov gruče strežnikov. Običajno ga uporabnik ne kliče direktno, ampak uporablja višje nivojske programske vmesnike od ostalih ogrodij. Več o njem v podpoglavju 4.2. MapReduce je programski model, ki pri zadnjih verzijah Hadoopa temelji na YARNu in se uporablja za paralelno procesiranje velikih podatkov. Sestavljen je iz dveh postopkov: *Map*, ki opravi filtriranje in sortiranje in *Reduce*, ki nato opravi procesiranje podatkov. Več o samem modelu si lahko preberete v podpoglavju 4.3. Na sliki 4.1 je prikazan ekosistem Hadoopa, kjer zasledimo tudi te štiri osnovne module.

4.1 Hadoop porazdeljen datotečni sistem

Danes imamo na voljo vedno višje kapacitete trdih diskov. Medtem ko so trdi diski leta 2005 dosegali kapacitete okoli 200GB [1], imamo danes na voljo že diske velikosti 12TB [2]. Žal pa je faktor zviševanja hitrosti pisanja in branja podatkov po disku manjši. Tako so diski leta 2005 imeli povprečno hitrost

[1] Za podatke smo uporabili trdi disk Seagate Barracuda 7200.7 (ST3200822AS)

[2] Pridobljeno iz specifikacij trdega diska Seagate Iron Wolf Pro 12TB (ST12000VN0007)

Slika 4.1: Ekosistem Hadoopa. Vir: lasten, 2018

branja okoli 50MB/s, medtem ko so novejši diski blizu 250MB/s. Leta 2005 smo rabili približno 1 uro, da smo prebrali vse podatke na disku, danes pa bi za večji disk rabili že več kot 13 ur. Ena izmed rešitev za večjo propustnost podatkov je v uporabi več diskov, ki delujejo paralelno. Žal pa se izkaže, da se pri uporabi več delov strojne opreme poveča verjetnost za odpoved enega izmed teh delov. RAID deluje po principu, da podatke podvaja ali računa paritete, tako da v primeru odpovedi strojne opreme ostanejo bodisi kopije podatkov bodisi možnost izračuna izgubljenih bitov. HDFS (*Hadoop Distributed Filesystem* oziroma Hadoop porazdeljen datotečni sistem) pa deluje malce drugače.

HDFS so zasnovali tako, da lahko shrani datoteke velike več deset gigabajtov in vzdržuje datotečne sisteme velike po več deset petabajtov. Za skaliranje se uporabi cenejša strežniška oprema in kup diskov, dostopnost in propustnost pa zagotavlja aplikacijska replikacija podatkov. Optimizirali so ga za velika pretočna branja, kar posledično poveča latenco. Prav tako je bil zasnovan za uporabo z modelom MapReduce. Tako kot trdi diski uporablja koncept blokov (v nadaljevanju se z besedo blok sklicujemo na HDFS bloke),

kar prinaša več prednosti. Prva prednost so shranjevanje datotek, ki so večje kot celoten disk. Razdelimo jo na bloke in porazdelimo po več diskih. Poenostavi se tudi datotečni sistem, saj je lažje shranjevati bloke, ki so stalne velikosti, kot pa datoteke, ki so lahko različnih velikosti. Bloki pa se obnesejo dobro tudi pri replikaciji (s katero zmanjšamo možnost za izgubo podatkov), saj se bloki običajno prepišejo na več lokacij. Ker so bloki v HDFS shranjeni po principu *write once, read many times*, se problem glede spreminjanja originalov in popravljanja repliciranih blokov ne pojavi. Tako lahko pri dostopu do bloka vzamemo kar tistega, ki nam je v omrežju najbližji oziroma imamo na izbiro več opcij. V primeru odpovedi strojne opreme se lahko zgodi, da se število kopij blokov zmanjša. V takšnih situacijah sistem to zazna in ustvari nove kopije na drugih lokacijah. HDFS ne potrebuje RAID sistema za shrambo, saj je pred izgubo podatkov zaščiten s tem repliciranjem.

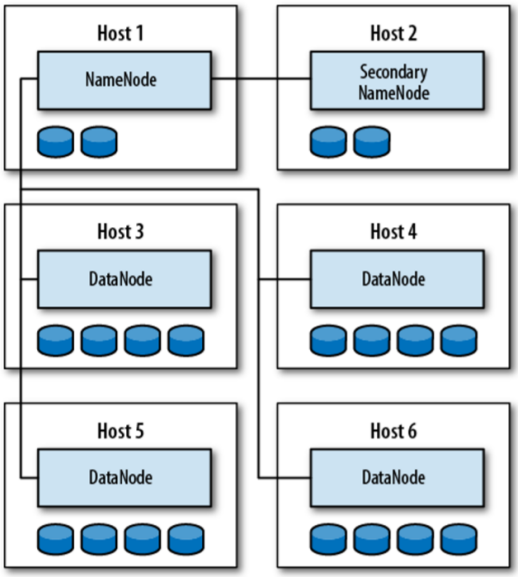
Pri HDFS poznamo dva tipa vozlišč, ki delujejo po principu gospodar-suženj. Med gospodarje se štejejo tako imenovana *namenode* vozlišča. Le-ta skrbijo za datotečno strukturo, torej vodijo evidenco o vseh datotekah in mapah v sistemu, katero shranijo na lokalni disk. Brez *namenode* uporaba datotečnega sistema ni mogoča. Če bi nekdo uničil vsa ta vozlišča, bi izgubili vse datoteke v sistemu, saj ne bi znali rekonstruirati teh datotek iz shranjenih blokov. Da se to ne zgodi, ima Hadoop več varnostnih mehanizmov. Prvi je izdelava varnostnih kopij prej omenjenih metapodatkov o sistemu, ki so sinhrona in atomična pisanja. Hadoop 2 je vpeljal koncept visoke razpoložljivosti, kjer imamo dva *namenode* vozlišča v konfiguraciji *active-standby*. Za realizacijo je bilo potrebno uvesti še nekaj arhitekturnih sprememb, toda sedaj pri uporabi tega koncepta v primeru odpovedi primarnega vozlišča sekundarno prevzame v manj kot sekundi, kar je veliko hitreje kot pa tako imenovan hladni zagon, kjer mora nov *namenode* prepisati vse metapodatke v delovni spomin in dobiti dovolj podatkov od *datanode* vozlišč, da lahko prične delovati. Čas potreben za hladen zagon *namenode* vozlišča je običajno več kot 30 minut. *Datanode* vozlišča so sužnji, ki shranjujejo ali pridobivajo bloke iz lokacij, ki so vnaprej podane bodisi od vozlišča *namenode*

bodisi uporabnika. Njihova naloga je tudi periodično pošiljanje informacij o vseh blokih, ki jih trenutno hranijo. Medtem ko se podatki, o tem katere datoteke so v katerem bloku, shranijo na disk, se lokacije teh blokov zapišejo le v delovni spomin. Ta pristop prinaša več prednosti. Lahko se spremeni omrežni naslov (ang. *hostname*) naprave in sistem bo še zmeraj deloval. Še več, odpove lahko matična plošča sistema in diske lahko le prestavimo v drug *datanode* in malo počakamo, da se pošljejo vsa poročila o blokah in *namenode* bo zopet stregel te podatke. Slabost je ta, da mora ob zagonu gruče *namenode* pridobiti vsa poročila o shranjenih blokah. Ker je veliko podatkov v spominu *namenode* vozlišč, ugotovimo, da pridemo do omejitev, saj je količina pomnilnika končna. Ta problem so rešili tako, da več *namenode* vozlišč skrbi za en datotečni sistem. Za primer lahko *namenode1* skrbi za vse v mapi */user*, *namenode2* naj vzdržuje mapo */hbase* in tako naprej. Koncept se imenuje *namenode federation*. Na sliki 4.2 je prikazana poenostavljena shema arhitekture HDFS.

Poleg teh dveh tipov vozlišč pa poznamo še tako imenovan sekundaren *namenode*. Ta ni kopija primarnega vozlišča, ampak le skrbi za njegove metapodatke o sistemu.

4.2 Upravljalec virov YARN

Apache YARN (ang. *Yet Another Resource Negotiator*) upravlja vire, ki so na voljo v gruči računalnikov. Višje ležeča ogrodja kot so MapReduce ali Spark od te komponente pričakujejo porazdelitev posameznih opravil njihovih aplikacij na vozlišča v gruči, dodeljevanje virov posameznemu opravilu preko vsebnikov, nadzorovanje posameznih opravil in podobno. Ekipa pri podjetju Yahoo! je naletela na težave v arhitekturi prvotnega sisema, ko so sistem skalirali na 4000 vozlišč [15]. Velike težave je povzročal *JobTracker*, ki je takrat skrbel za vsa opravila (angl. *jobs*) pri MapReduce. Pri tej veliki postavitvi *JobTracker* ni mogel vzdrževati vseh opravil, ki bi naj tekla in so se posledično izgubljala. Kot rešitev so leta 2012 namesto *JobTracker*-ja

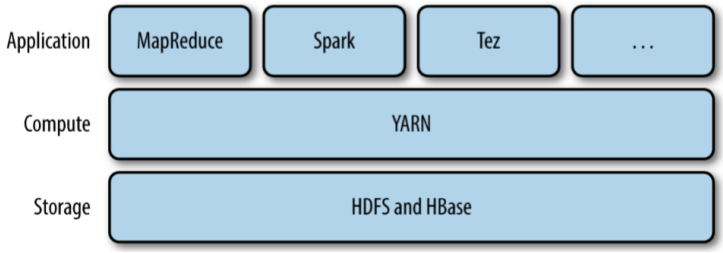


Slika 4.2: Pregled arhitekture datotečnega sistema HDFS. Vir: Sammer E., Hadoop Operations, stran 10

vpeljali četrti modul imenovan YARN [4].

Deluje kot vmesnik med višjenivojskimi aplikacijami in viri. Prav tako teži k temu, da je za uporabnika proces upravljanja virov skrit. Na sliki 4.3 lahko vidimo nivoje pri aplikacijah, ki uporabljajo YARN. Seveda obstajajo še višje nivojske aplikacije, ki delujejo nad temi prikazanimi na sliki, kot sta *Pig* in *Hive*. Prvotno implementacijo enega procesa, ki upravlja z viri, je YARN izboljšal tako, da si je proces razdelil v dva dela. Del, ki je v *JobTrackerju* upravljal z viri, je postal upravljalec virov (angl. *resource manager*), ki se pojavi enkrat na gručo. Ta dodeljuje vire k več aplikacijam, vsaka takšna aplikacija pa ima nato svojega gospodarja (angl. *application master*). Pri takšni postavitvi nimamo več centraliziranega upravljalca virov. Posamezni gospodarji so med sabo izolirani in tudi če eden izmed njih odpove, lahko ostali delujejo brez problemov. Prav tako lahko gospodarji uporabljajo različno programsko opremo in posledično enostavno vpeljejo tekoče posodobitve aplikacij. Preostali del, ki je pa upravljal s procesi MapReduce opravil, pa je postal upravitelj vozlišč (angl. *node manager*), ki pa

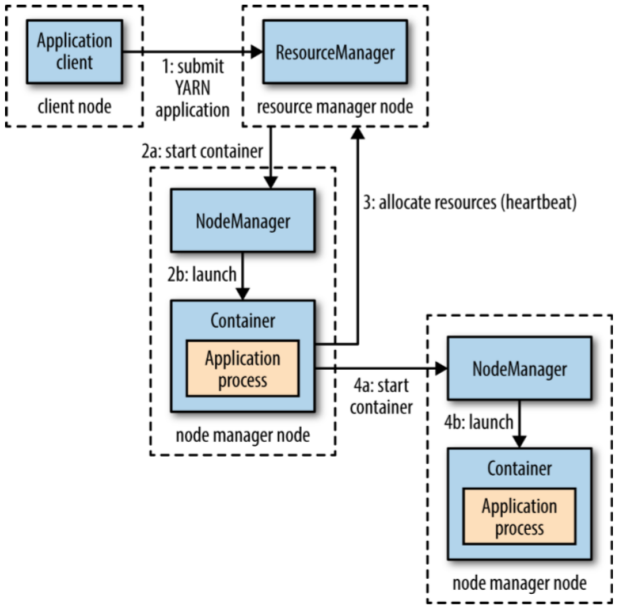
teče na vsakem vozlišču. Zaganja in nadzoruje procese v vsebnikih. Ti omejujejo procese z viri (pogosto pomnilnik, procesna moč, ...). Postal je tudi bolj splošen: ni nujno, da zaganja le procese, ki so v povezavi z MapReduce, ampak kakršnekoli. Tako je mogoče, da YARN upravlja tudi aplikacije, ki niso povezane z MapReduce.



Slika 4.3: Distribuirane aplikacije, ki tečejo kot YARN aplikacije na procesnem nivoju in uporabljajo HDFS in HBase na nivoju shrambe. Vir: White T., Hadoop: The Definitive Guide, Fourth Edition, stran 79

Na sliki 4.4 je prikazan proces zagona aplikacija s pomočjo YARN. Najprej mora uporabnik poslati zahtevo za nov proces (korak 1) k upravljalcu virov. Ta najde prosto vozlišče in na njem zažene vsebnik s procesom (koraka 2a in 2b). Večinoma se nato iz procesa zaradi distribuiranega procesiranja kličejo zahtevki za več virov (korak 3) in se zatem ustvarijo še novi vsebniki (koraka 4a in 4b). Pri zahtevanju virov imamo možnost določevanja, kje naj se dodelijo. Tako lahko za večjo pasovno širino med vsebniki določimo vozlišča, kjer naj tečejo. To je bodisi na istem vozlišču bodisi na istem stojalu (angl. *rack*), če pa to ne uspe, pa lahko rečemo naj ga kreira kjerkoli. Kot primer: MapReduce pri funkciji *map* poskusi vsebnik ustvariti na enem izmed vozlišč, ki hrani podatke potrebovane v procesu. Prav tako lahko vnaprej zaprosimo za vse vire, ali pa med samim procesom dinamično zvišujemo ali znižujemo razpoložljive vire.

V primerjavi z MapReduce 1 (implementacija MapReduce preden se uporabi YARN), se pojavi več prednosti. Kot že omenjeno je gruča bolj skalabilna in lahko dosežemo 10000 vozlišč [18]. Omogoči se tudi visoka razpoložljivost. Podvajamo podatke o stanju, tako da lahko drug prikrit proces



Slika 4.4: Proces zagona aplikacije pri YARN. Vir: White T., Hadoop: The Definitive Guide, Fourth Edition, stran 80

(angl. *daemon*) prevzame opravila, v primeru da prvotni odpove. Pri MapReduce 1 so viri bili razporejeni fiksno po predalčkah. Določena so bila tudi števila predalčkov za opravila *map* in za opravila *reduce*. Pri tem se opravila *map* niso mogla zaganjati v predalčkah namenjenim *reduce* in obratno. Pri YARN pa imamo *pool of resources*, kjer si opravilo samo izbere količino virov. Ne pride več do situacij, da bi *map* ali *reduce* opravilo čakalo na prazne predalčke svoje vrste. Izboljšala se je tudi izkoriščenost, saj lahko vire za posamezno opravilo dinamično spreminjamo (ni potrate v primeru da proces ne potrebuje vnaprej določene količine virov, ali napake v primeru da predalček nima dovolj virov za proces). Verjetno največja prednost pa je ta, da se Hadoop lahko uporablja z drugimi distribuiranimi aplikacijami, ki ne rabijo nujno uporabljati MapReduce.

4.3 Hadoop MapReduce

Idejo o programskem modelu MapReduce so razvili pri podjetju Google leta 2004 [14]. Zamišljen je bil za procesiranje velikih podatkovnih množic. Uporabniki pišejo opravila, ki so v glavnem sestavljena iz dveh delov: *map* (maperji) in *reduce* (reducerji). Vso upravljanje z viri, distribucija na več vozlišč, nadzor in reševanje težav naj bo prepuščeno ogrodju. Slednje omogoča, da tudi razvijalci z malo izkušnjami z distribuiranimi sistemi lahko uporabljajo vire le-teh. Glavne prednosti so torej enostavnost, saj se skrije distribuiranje opravil, skalabilnost na vedno več vozlišč in pa odpornost na napake. Zadnje je izvedeno tako, da če se opravila na enem izmed vozlišč ne izvedejo do konca (na primer vmes pride do izpada strojne opreme), se vsa nedokončana opravila ponovno zaženejo na nekem drugem zdravem vozlišču samodejno.

Pri MapReduce se torej pišejo opravila. Uporabnik definira funkciji *map* in *reduce* in še poda nekaj parametrov za zagon novega opravila. Ogrodje (YARN) nato poskrbi, da se to opravilo razdeli na procese, jih razporedi po vozliščih, nadzira njihovo stanje in skrbi za morebitne napake. Večinoma se za vhod vzame množica datotek in se kot izhod vrne podatkovna množica. Funkcija *map* vhodne podatke spremeni v vmesne pare tipa ključ in vrednost. Ti se nato kot vhod podajo funkciji *reduce*, ki vzame nek ključ in nato obdela vse vrednosti zanj, ki so lahko porazdeljene po vozliščih. Večinoma se kot rezultat funkcije *reduce* vrne le ena vrednost. Za enostaven primer vzemimo štetje pojavov posamezne besede v količini dokumentov. Psevdokoda 4.1 prikazuje funkciji *map* in *reduce*. Pri prvi se sprehodimo skozi vsebino posameznega dokumenta in v njem preštejemo pojave posameznih besed vsebovanih v njem. V naslednjem koraku pa za vsako posamezno besedo preštejemo, kolikokrat se je pojavila po vseh dokumentih.

Vhodni podatki v MapReduce program se porazdelijo na kose (angl. *input splits* ali kar *splits*), katerih velikost je določena vnaprej. Nad vsakim kosom se nato zažene opravilo *map*, ki kliče uporabnikovo funkcijo nad vsakim zapisom v kosu. Obdelava enega kosa je veliko hitrejša, kot pa če bi

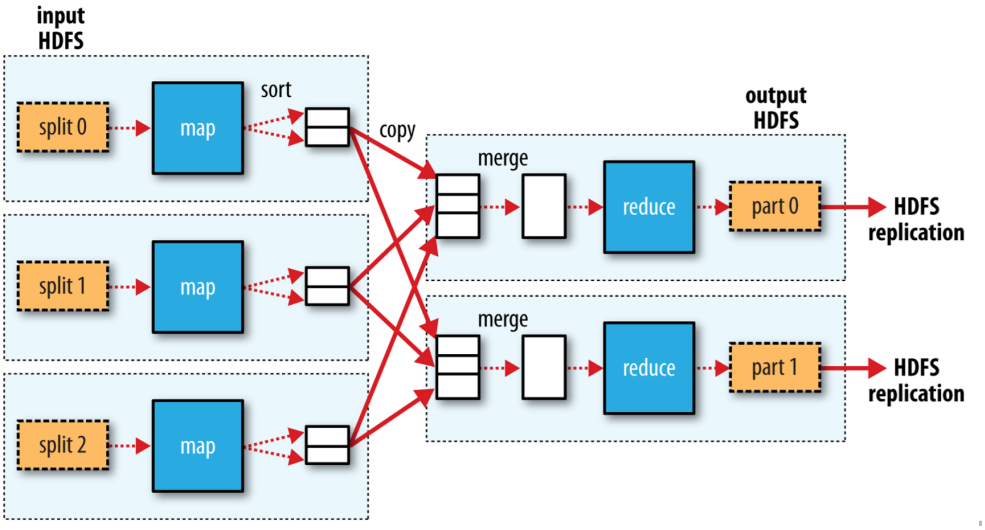
```
map(String kljuc, String vrednost):
        // kljuc: ime dokumenta
        // vrednost: vsebina dokumenta
        for each beseda b in vrednost:
                EmitIntermediate(b, "1")

reduce(String kljuc, Iterator vrednosti):
        // kljuc: beseda
        // vrednosti: seznam prestetih besed
        int rezultat = 0
        for each v in vrednosti:
                rezultat += int(v)
        Emit(kljuc, rezultat)
```

Kodni blok 4.1: Psevdo koda štetja besed v dokumentih.

gledali celotno množico podatkov. Kose pa lahko obdelujemo paralelno in se zato celotna obdelava pohitri. Porazdeljevanje bremena (angl. *load balancing*) je tu bistvenega pomena. Idealen scenarij je, ko vsa vozlišča v gruči opravljajo enako veliko dela, kar je enostavneje doseči z manjšimi kosami podatkov. Toda premajhni kosi lahko povzročijo preveč režije in upočasnijo obdelavo. Večinoma je za velikost enega kosa dobra mera kar velikost HDFS bloka (privzeto 128 MB). Hkrati je to tudi največja garantirana velikost podatkov shranjenih na enem vozlišču. Če bi recimo vzeli dva bloka, se lahko zgodi, da sta (vključno z kopijami) shranjena na dveh precej oddaljenih vozliščih. V takem primeru bi tratili dragoceno pasovno širino gruče, saj bi blok prenašali preko omrežja. Opisano načelo se imenuje optimizacija lokalnosti podatkov (angl. *data locality optimization*), o katerem smo govorili že pri YARN (zagon procesov na istih vozliščih kjer so podatki). Vmesni podatki, to so izhodi funkcije *map*, se ne zapisujejo v HDFS. Te vrednosti so le začasne in se ob končanem *reduce* opravilu lahko zavržejo. Repliciranje teh podatkov v sistemu HDFS bi bila potrata, zato jih shranimo na lokalen disk (ogrodje Spark za dodatno pohitritev vrednosti shrani v delovni pomnilnik).

Pri opravilih *reduce* pa lokalnost ni mogoča. Vhodi zanje so izhodi od vseh maperjev, ki pa so tipično razporejeni po omrežju. Prav tako se izhod opravila *reduce* zapiše v sistem HDFS za večjo zanesljivost ohranjanja rezultata. Število reducerjev ni pogojeno z velikostjo vhoda ampak ga lahko določimo. Na sliki 4.5, lahko vidimo potek aplikacije pri dveh reducerjih. Maperji svoje izhode porazdelijo na particije: vsak jih naredi toliko, koliko je reducerjev. Particija lahko vsebuje veliko različnih ključev, vendar se vse vrednosti za določen ključ vedno nahajajo v samo eni izmed particij. Particije se nato porazdelijo po reducerjih v fazi imenovani *shuffle*. Ta faza je bolj kompleksna kot nakazuje slika in določanje parametrov zanjo lahko precej vpliva na celoten čas izvajanja. Poseben primer je, ko nimamo reducerjev. Takrat za rezultat vzamemo kar izhode maperjev in jih zapišemo v HDFS.



Slika 4.5: Diagram poteka MapReduce pri dveh *reduce* opravilih. Vir: White T., Hadoop: The Definitive Guide, Fourth Edition, stran 33

Prizadevamo si za čim manjšo obremenitev omrežja, saj je pasovna širina dragocena. Za dodatno varčevanje z njo lahko definiramo tudi združitvene funkcije (angl. *combiner functions*). Te se kličejo med maperji in reducerji in zmanjšujejo količino podatkov, ki se prenašajo med njimi. Razložimo koncept na primeru. Imamo podatke o meritvah temperaturev nekem kraju

po posameznih letih. Želimo najti najvišjo izmerjeno temperaturo za leto 2016. Recimo, da je prvi maper proizvedel naslednji izhod:

(2016, 0)
(2016, 20)
(2016, 10)

drugi pa takšnega:

(2016, 25)
(2016, 15)

Običajno bi klicali reducer s seznamom vseh vrednosti:

(2016, [0, 20, 10, 25, 15])

Kar bi nam vrnilo izhod 25, saj je najvišja vrednost. Z uporabo združitvene funkcije bi, enako kot pri reducerju, iskali maksimum, vendar bi to storili že pri izhodih maperjev. Tako bi reducer dobil naslednji vhod:

(2016, [20, 25])

pri čemer sta 20 in 25 maksimuma pri obeh posameznih maperjih. Rezultat je v tem primeru ostal enak, kar pa ne velja za vsak primer. Za primer lahko bralec poskusi enak postopek za računanje povprečne vrednosti. Prav tako ni zagotovljeno kolikokrat se bo funkcija klicala. Lahko da samo enkrat, lahko večkrat, lahko pa da sploh ne. Zato je pomembno, da ne glede kolikokrat jo kličemo, če sploh, da preverimo, da reducer proizvede enak rezultat. Kljub vsemu se zmanjšuje količina podatkov, ki se prenaša med maperji in reducerji in samo iz tega vidika je vredno pretuhtati ali lahko uporabimo združitveno funkcijo.

4.4 Drugi projekti

Z leti se je razvilo vedno več projektov, ki razširjajo zmožnosti Hadoopa. V tem podpoglavju bomo na kratko opisali nekaj izmed njih, da dobimo občutek kaj vse Hadoop še zmore. Seznam odprtokodnih projektov v povezavi s Hadoopom je na voljo na spletnem naslovu `hadoopecosystemtable.github.io` (nazadnje dostopano 25.10.2017).

Apache Flume nudi možnost, da v Hadoop vpeljemo podatkovne tokove. Večinoma se uporablja za dnevniške datoteke, tipičen primer je zbiranje teh datotek iz spletnih strežnikov in ustvarjanje novih datotek primernih za shrambo v HDFS. Proces je sestavljen iz izvirov, ponorov in kanalov. Dogodki, ki se proizvajajo v izvirih, se shranjujejo v kanalu do določene mere in se nato posredujejo ponoru, ki je ponavadi kar HDFS.

Apache Sqoop je odprtokodno orodje, ki omogoča pretvorbo strukturiranih podatkov za obdelavo teh v okolju Hadoop. Pogosto so podatki shranjeni v relacijskih bazah in Sqoop omogoča, da tudi takšne podatke procesiramo z MapReduce. Pridobljene rezultate lahko shrani tudi nazaj v prvotno bazo, da so na voljo ostalim uporabnikom.

MapReduce omogoča, da definirate funkciji *map* in *reduce*, vendar morate sami nato prilagoditi podatke, da ustrezajo temu postopku. Velikokrat je to težek izziv, zato **Apache Pig** vpelje bogatejše podatkovne strukture. Uporablja svoj jezik imenovan Pig Latin, ki deluje podobno kot SQL pri relacijskih bazah. Uporablja pa še svojo okolje za zagon Pig Latin programov, ki pa je bodisi lokalno na virtualnem Java stroju bodisi v gruči Hadoop. Program v tem jeziku je sestavljen iz več operacij in transformacij nad vhodnimi podatki. Če bi pogledali natančneje, bi videli, da Pig uporablja MapReduce opravila, da izvede svoje operacije. Uporabniku je to skrito in se tako lahko osredotoči na podatke ne pa na izvedbo. Celoten projekt je bil zasnovan tako, da je razširljiv. Vse operacije (nalaganje, shranjevanje, filtriranje, združevanje ...) lahko prilagodimo z uporabniškimi funkcijami. Slabost je, da v nekaterih primerih deluje počasneje kot aplikacije v MapReduce. Vsekakor pa pisanje poizvedb v Pigu prihrani veliko časa in je tudi enostavnejše.

Pri Facebooku so razvili ogrodje za podatkovno skladiščenje imenovano **Hive**. Na začetku so ga uporabljaji le interno, po nekaj mesecih pa so ga kot uraden podprojekt Hadoopa odprli javnosti pod licenco Apache 2.0. Uporablja SQLu podoben jezik imenovan HiveQL. Tako lahko podatke shranjene v HDFS analizirajo tudi uporabniki, ki so bolj vešči jezika SQL kot pa Jave. Seveda ta pristop ni uporaben vsepovsod (na primer pri kompleksnih algo-

ritmih strojnega učenja), vendar je široko poznan in se lahko uporabi pri določenih analizah. Kot alternativo je potrebno omeniti **Apache Impala**, ki nudi podobne funkcionalnosti.

Apache Spark je ogrodje za procesiranje podatkov na večji ravni. Z razliko prej opisanih ogrodij ta ne uporablja MapReduce za pogon aplikacije. Je tesno povezan z Hadoopom: lahko teče na YARN in uporablja HDFS. Ima zmožnost, da obdrži večje količine podatkovnih množic v pomnilniku med opravili. To lahko izkoristijo aplikacije, ki iterirajo na podatki (jih kličejo znova in znova dokler ni izpolnjen ustavitveni pogoj) ali pa aplikacije z interaktivno analizo, kjer uporabnik niza poizvedbe po podatkih. Nekateri so mnenja, da je tudi uporabniška izkušnja boljša kot pa pri MapReduce, saj ima bogato izbiro programskih vmesnikov.

HBase je distribuirana stolpično orientirana (angl. *column-oriented*) baza zgrajena na osnovi HDFS. Uporablja se pri potrebi po pisanju in poizvedbah v realnem času po velikih podatkovnih množicah. Kljub veliko rešitvam za shranjevanje podatkov večina ne podpira zelo velikega obsega podatkov in porazdeljenosti. HBase rešuje problem z lineranim skaliranjem, torej dodajnjem vozlišč po potrebi. Ni relacijska baza in ne podpira SQL, vendar lahko gostuje zelo velik obseg podatkov v redkih tabelah v gruči z običajno strežniško opremo. Tipičen primer uporabe bi bila spletna tabela (angl. *webtable*). Vsebuje zapise o spletnih straneh, katere brskajo tako imenovani *web crawlers*, ponavadi so ključi kar spletni naslovi, med atribute pa sodijo na primer jezik strani. Te tabele so precej velike in lahko dosežejo milijardno število vrstic. Na njih se poganjajo MapReduce opravila, katera računajo statistike, dodajajo podatke za indeksiranje s strani spletnih brskalnikov ... Naključne vrstice ob naključnem času posodabljajo tudi prej omenjeni *web crawlers*, zato je hitrost dostopa pomembna.

Zookeper rešuje eno izmed težav pri pisanju distribuiranih aplikacij, ki je delna napaka sistema. Vzemimo primer, kjer si dve vozlišči pošiljata sporočilo in med procesom odpove omrežje. Pošiljatelj ne ve ali je sporočilo prispelo na cilj ali ne. Lahko da je pred napako prišlo skozi, morda pa je še

poleg omrežja propadel tudi sprejemnikov proces. Če pošiljatelj želi izvedeti, mora ponovno vzpostaviti povezavo in sprejemnika vprašati. Zookeeper teh napak ne odpravi , nudi pa orodja, da z njimi rokujemo. Med drugim se lahko uporablja za porazdeljene vrste in izbiro voditelja med soležniki. Nudi visoko razpoložljivost, služi pa lahko tudi kot posrednik med dvema procesoma, ki drug o drugem ne vesta ničesar. Prav tako vsebuje odprtokodno knjižnico z implementacijami pogosto uporabljenih koordinacijskih postopkov.

Nazadnje omenimo še **Hue** (angl. *Hadoop User Experience*) od Cloudere, ki je odprtokoden in nudi spletni vmesnik za brskanje, poizvedovanje in vizualizacijo podatkov in **Apache Oozie**, ki v grobem razvršča Hadoop opravila.

4.5 Naša postavitev

Za dokaz koncepta naše platforme smo Hadoop tudi sami namestili na računalnik, v katerem ima na voljo dvojedrni procesor, 8 gigabajtov delovnega pomnilnika in 1 terabajt velik trdi disk. Odločevali smo se med distribucijama, ki jih ponujata Cloudera in Hortonworks. Cloudera je podjetje, katerega so ustanovili inženirji iz podjetij Google, Yahoo!, Oracle in Facebook, danes pa je pri njih zaposlen tudi že omenjen Doug Cutting, ki je ustanovitelj Hadoopa. Za svoje dosežke so prejeli že veliko nagrad [12]. Pri CDH (angl. *The Cloudera distribution of Apache Hadoop*) nam ponujajo (poleg osnovnih komponent Hadoopa seveda) tudi svoje produkte. Zaenkrat omenimo le *Cloudera Manager*, ki nudi spletni uporabniški vmesnik za upravljanje postavitev CDH gruč in pa Hue, ki je odličen pri poizvedovanju po podatkih. Na njihovi spletni strani si lahko prenesemo posnetek operacijskega sistema ali Docker vsebnik, v katerem je že skonfigurirana osnovna postavitev Hadoopa [3]. Ta ponudnik je odličen za vse, ki bi radi preizkusili Hadoop z malo napora pri postavitvi in uporabi. Žal pa za delovanje potrebuje malce boljšo

[3]Spletni naslov za prenos: `https://www.cloudera.com/downloads/quickstart_vms/5-12.html`

opremo, kot pa je bila na voljo v našem primeru. Tako da smo se odločili za ponudnika Hortonworks in njihov HDP (Hortonworks Data Platform), ki je v bistvu distribucija Apache Hadoopa. Podjetje so ustanovili leta 2011 z (poleg ostalih) 24 inženirji, ki so bili prisotni v prvotni ekipi za razvoj Hadoopa. Tamkajšnji zaposleni so tudi znani po tem, da prispevajo največje deleže k projektu. Medtem ko napredne opcije pri Clouderi zahtevajo plačilo za uporabo, je pri Hortonworks vsa funkcionalnost platforme na voljo zastonj. Hortonworks se je torej usmeril v model odprte kode in razvoja, medtem ko so pri Clouderi usmerjeni v komercialnost.

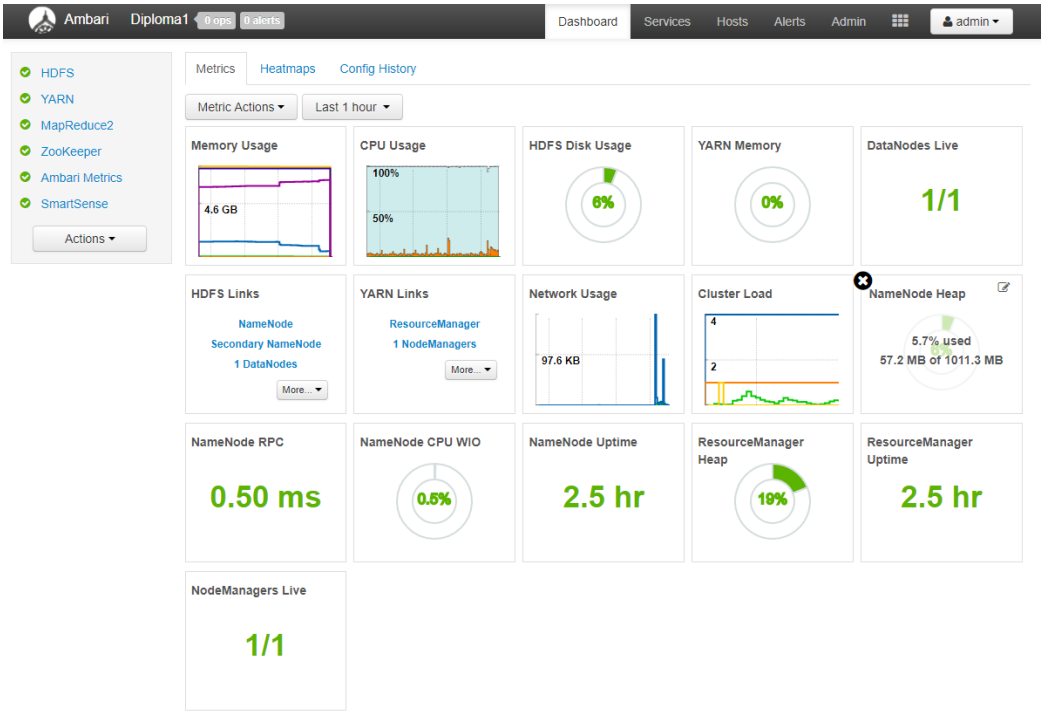
Na našem strežniku teče operacijski sistem Ubuntu 16.04. Za potrebe HDP imamo nameščen Python (verzije 2.7.12) in Javo (verzije 8). Na kratko bomo še povzeli postopek za namestitev Apache Ambari, ki deluje kot upravljalec gruče preko grafičnega spletnega vmesnika, in nato namestitev komponent s pomočjo Ambarija. Podrobnejša navodila so na voljo na spletni strani za dokumentacijo Hortonworks platforme [4]. Za namestitev komponent potrebuje Ambari dostop brez gesla do *root* računov na posameznem računalniku v gruči. Da smo to dosegli, smo se prijavili kot uporabnik *root* in dodali javni RSA ključ v datoteko *~.ssh/authorized_keys*. Preveriti je potrebno tudi, da so pravilno nastavljena domenska imena računalnikov. Za pravilno delovanje morajo biti odprta nekatera vrata v požarnem zidu, ker se gre le za demonstracijo, smo požarni zid kar izklopili, saj ne dajemo poudarek na varnost. Na koncu smo še izklopili *SELinux*, za kar smo uporabili ukaz *setenforce 0*. V naslednjem koraku smo dodali Ambari apt repozitorij in namestili paket *ambari-server*. Pred zagonom tega strežnika je potrebna še dodatna konfiguracija (preko ukaza *ambari-server setup*), več o tej je napisano v prej omenjeni dokumentaciji Hortonworksa.

Ko so vsi zgoraj opisani koraki zaključeni, lahko zaženemo Ambari strežnik preko ukazne vrstice z ukazom *ambari-server start*. Sedaj lahko odpremo

[4]Spletni naslov za podrobnejša navodila: `https://docs.hortonworks.com/HDPDocuments/Ambari-2.6.0.0/bk_ambari-installation/content/ch_Getting_Ready.html`

spletni brskalnik in ga usmerimo na vrata 8080 na našem strežniku. Odpre se stran za prijavo, kjer vnesemo privzeto uporabniško ime in geslo admin/admin. Po prijavi lahko pričnemo z namestitvijo Hadoop gruče s klikom na gumb *Launch Install Wizard*. Vnesemo ime gruče (mi smo izbrali Diploma1), izberemo verzijo HDP (v času pisanja je bila na voljo 2.6.3.0), ko nas povpraša po domenskih imenih računalnikov v gruči vnesemo ime našega strežika (pri nas je bil na voljo le en računalnik, pri pravi postavitvi se tukaj vnese več imen). Potrebno je še vnesti privatni ključ računa *root*, katerega javni par smo prej shranili v datoteko, oziroma RSA ključ od računa, ki ima pravico do izvedbe *sudo* ukazov brez potrebe po vnosu gesla. Naša odločitev, da uporabimo kar račun *root* izhaja iz tega, da nismo namenili veliko časa varnosti sistema in je praktično uporabiti že račun, ki je predhodno nameščen v sistem. Po nekaj korakih, ko je potrebno še potrditi strežnike, pridemo do izbire storitev, ki naj tečejo v naši gruči. Tu smo se odločili za minimalno število teh, to so HDFS, Yarn (z MapReduce2), ZooKeeper, Ambari Metrics in Smart Sense. Od teh še nismo omenili Ambari Metrics in Smart Sense. Prva je storitev, ki zbira raznorazne metrike v strežniškem okolju in nam jih izpisuje preko grafičnega vmesnika (recimo zasedenost pomnilnika). Smart Sense pa je obvezna pri postavitvi in nam nudi hitro pridobivanje podatkov (konfiguracijske parametre, metrike, dnevniške datoteke) o HDP storitvah. Odvisno od storitve obstaja možnost, da je v naslednjem koraku potrebno še dodatno konfigurirati kakšne parametre (recimo vnos gesel, ali pa vzpostaviti povezavo do podatkovne baze). Uporabniku prijazno so vsi ti dodatni koraki jasno označeni in kjer je bilo zahtevano smo te podatke še izpolnili. Na sliki 4.6 je prikazan Ambari spletni vmesnik po končani namestitvi, ki teče na našem strežniku.

Tako smo prišli do naše postavitve Hadoopa, ki nam teče v načinu enojnega vozlišča, kar pomeni, da vse storitve tečejo le na enem strežniku. Da se izognemo opozorilom, smo pri storitvi HDFS spremenili parameter minimalne replikacije blokov iz privzete vrednosti 3 na 1. Imamo namreč le en *datanode* in je potemtakem replikacija bloka na 3 vozlišča nemogoča. Prav



Slika 4.6: Spletni vmesnik za delo s Hadoop gručo, ki teče na našem strežniku. Vir: lasten, 2018

tako pri delu z HDFS za lažjo preglednost uporabljamo svojega uporabnika, katerega smo si ustvarili. Uporabili smo račun na operacijskem sistemu, ki ima uporabniško ime *david*. V okolju HDFS smo ustvarili mapo */user/david* in uporabniku *david* dodali pravice za upravljanje s to mapo. Ukaza, ki to dosežeta, sta napisana v kodnem bloku 4.2. Če povzamemo, imamo delujočo Hadoop gručo enojnega vozlišča, dostop do spletnega vmesnika Ambari na vratih 8080, kjer lahko upravljamo s to gručo in svojega uporabnika v HDFS, ki bo imel v lasti podatke poslane iz prehoda.

```
$ sudo -u hdfs fs -mkdir /user/david
$ sudo -u hdfs fs -chown -R david /user/david
```

Kodni blok 4.2: Ukaza za dodajanje uporabnika david v HDFS

4.5.1 Postavitev v oblak

Konceptualno smo našo postavitev platforme razširili še tako, da smo jo izpostavili v oblak. Nakup nove strojne opreme, če oziroma ko bi morali našo postavitev skalirati, bi prinesel veliko dodatnih stroškov, katere si kot posamezniki navadno ne moremo privoščiti. Oblak nam nudi naročninski model zakupa virov, kjer je strošek odvisen od uporabe virov skozi čas. Slednje omogoča skalabilnost in zanesljivost in občutno zniža začetne stroške. Po želji lahko dodajamo vire, če recimo izvajamo veliko MapReduce opravil, in tudi odvzemamo vire, če bi recimo le sprejemali in shranjevali zahtevke iz naprav. Oblak prinaša tudi večjo varnost in zanesljivost. Prednosti oblaka so opisane tudi v članku [24], kjer je med drugim omenjeno tudi to, da so se za oblak odločili pri ameriški vesoljski agenciji NASA, ameriški vladi in obveščevalni službi CIA.

Kot že rečeno bomo le omenili koncepte za to skaliranje, saj so po večini te storitve plačljive. Raziskovanje vseh teh opcij pa je izven dosega te naloge. Kot prvo opcijo omenimo *Cloudera Altus*, ki je oblačna storitev, ki omogoča uporabo njihove postavitve Hadoopa v oblaku. Uporablja se lahko z Amazonovim oblakom (AWS). V grobem nudi vmesnik, ki preko vašega AWS računa ustvari, nadzira in tudi ugasne Hadoop gručo. Če si želimo malce večji nadzor in bogatejše funkcionalnosti, lahko pri istem ponudniku uporabimo *Cloudera Director*. Ta nudi vmesnik, s katerim preidemo iz upravljanja strojne opreme na upravljanje oblačnih instanc. Uporablja se ga lahko z javnimi oblaki kot so Amazon Web Services (AWS), Microsoft Azure in Google Cloud Platform (GCP). Podpira tudi delovanje tako imenovanega hibridnega oblaka, kjer lahko del aplikacij ostane znotraj podjetja, del pa jih lahko prenesemo v oblak. Pri podjetju Hortonworks pa imajo orodje imenovano *Cloudbreak*, ki lahko njihovo postavitev Hadoopa izpostavi v oblak. Je del njihove Hadoop Data Platforme in deluje preko vmesnika Apache Ambari. Poleg javnih oblakov AWS, Microsoft Azure in GCP podpira tudi odprtokoden Openstack. Seveda je izbire še ogromno, tudi ponudniki javnih oblačnih storitev imajo svoje implementacije: Amazon nudi Amazon EMR, Google

ima Cloud Dataproc, Microsoft Azure pa HDInsight.

Oblačna postavitev in izbira količine virov je že precej obsežen problem pri načrtovanju postavitve Hadoopa. Odvisna je predvsem od primera uporabe, kjer imamo ponavadi omejene finančne vire, pa tudi večje količine podatkov in posledično večje delovne obremenitve, kjer nam skalirana postavitev omogoča hitrejše procesiranje podatkov. Za nas je zaenkrat dovolj en strežnik in možnost da v prihodnosti uporabimo oblak, če bo po njem potreba.

Poglavje 5

Nivo prehoda

V tem poglavju bomo opisali našo implementacijo nivoja za prehod podatkov iz senzorskih naprav v shrambo. Spisali smo spletni strežnik s pomočjo NodeJS, ki usmerja podatke. Strežnik uporablja ogrodje *Express*. Knjižnica *morgan* doda beleženje dogodkov in nam omogoča spremljanje dogajanja. Knjižnica *request* pa nam poenostavi delo s HTTP zahtevki potrebnimi za komunikacijo s HDFS. Na strežniku sta izpostavljena dva vira. Prvi je na domači lokaciji (pod *'/'*) in vrača le tekstovni odziv, drugi pa se nahaja pod *'/data'* in sprejema POST zahtevke s podatki. Potrebno je omeniti, da ima drugi vir 2 verziji, prva sprejema podatke v obliki seznama, torej predpostavlja, da se podatki predpomnijo na napravi, ki jih zbira, druga verzija pa posluša za posameznimi meritvami in jih združi do približno 1.1 megabajta velikosti, preden se shranijo v HDFS. Med testiranjem so se podatki sprejemali iz ene same naprave s povprečno hitrostjo treh zahtevkov na sekundo. S podatkom o velikosti posameznega zahtevka (približno 100 bajtov) pridemo do časa med klici za shranjevanje datotek v HDFS, ki znaša eno uro. Za procesiranje datotek je ponavadi bolje, da so te čim večje, saj se nato ustvari manj *map* opravil in je obremenitev manjša. Velikost enega megabajta in čas ene ure se nam je zdela dobro razmerje med čakanjem na podatke in velikostjo shranjene datoteke. Vsi prej opisani viri so prikazani v tabeli 5.1. Mejo količine podatkov, ki jih lahko preberemo v enem zahtevku, smo dvi-

Lokacija	Metoda	Opis
/	GET	Vrača tekstovni odziv, za testiranje ali je prehod aktiven
/v1/data	POST	Shrani prejete podatke kot celoto v HDFS
/v2/data	POST	Prejete podatke predpomni in jih shrani v HDFS šele, ko prispe določeno število klicev

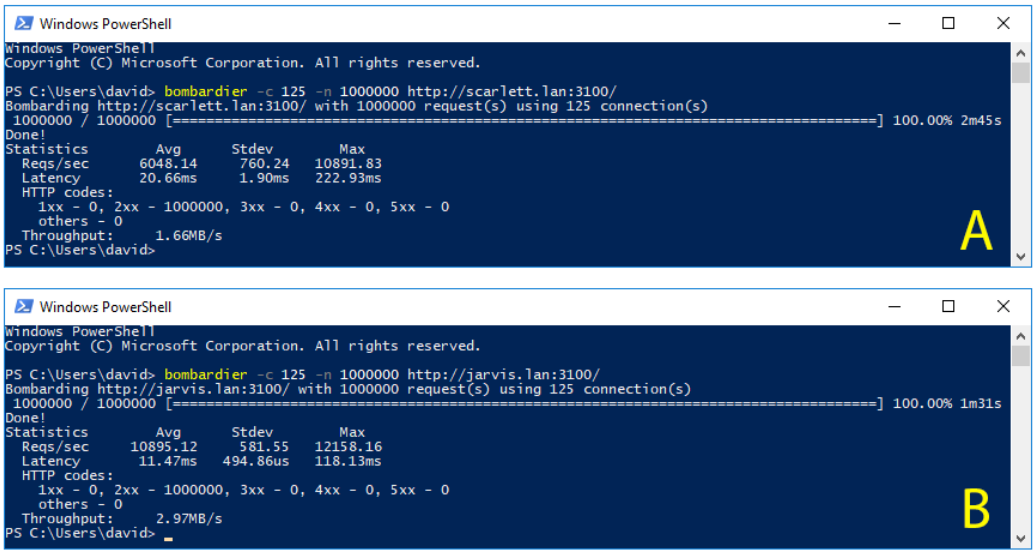
Tabela 5.1: Tabela z viri, ki so na voljo na našem prehodu

gnili na 64MB. Za shranjevanje v HDFS smo implementirali svoj standard za poimenovanje datotek. Tako smo se odločili, da uporabimo podmape, ki nakazujejo leto, mesec, dan v mescu in uro, ime datoteke pa je časovni posnetek trenutka, ko je bil prejet (primer: /2017/oct/13/15/ kot podmape in 2017_10_13_15_44_32_678Z kot ime datoteke). Časovni posnetek pri imenu je natančen do milisekunde, tako da teoretično omogoča sprejemanje do 1000 zahtevkov na sekundo. S to potjo in imenom datoteke se nato kliče PUT zahtevek preko *WebHDFS* programskega vmesnika na *namenode*. Ta nam posreduje lokacijo ustreznega *datanode* vozlišča, ki je v naši postavitvi eno in edino, kamor nato podamo še en PUT zahtevek in dokončno shranimo prejete podatke.

5.1 Skaliranje prehoda

Naša trenutna rešitev deluje, vendar jo uporablja le nekaj naprav. Da bi ugotovili približno mejo, koliko naprav lahko prehod hkrati streže, smo izvedli obremenitveni test s pošiljanjem zahtevkov na naš prehod. Zahtevke smo pošiljali na vir, ki vrača tekstovni odziv (glej tabelo 5.1), uporabili pa

smo program *bombardier* [1]. Program je generiral milijon zahtevkov preko sto petindvajset povezav. Na sliki 5.1 (A) lahko razberemo povprečno število zahtevkov na sekundo, ki je okoli 6000. Za doseganje te meje bi prej seveda morali popraviti delovanje prehoda pri poimenovanju datotek, ker bi lahko prihajalo do konfliktov pri shranjevanju datotek z enakim imenom. S povečanjem naprav se približamo tej meji in jo seveda tudi lahko presežemo. To pa pomeni, da se bodo naše zahteve pričele izgubljati.



Slika 5.1: Zmogljivost prehoda postavljenega na starejšem strežniku (A) in na zmogljivejšem prenosniku (B) Vir: lasten, 2018

Po privzetem vsi procesi znotraj NodeJS aplikacije tečejo v eni sami niti. To pomeni, da uporabljamo le eno procesorsko jedro in se preostala ne uporabljajo iz vidika aplikacije. Z NodeJS modulom imenovanim gruča (angl. *cluster* [2]) lahko v pogon spravimo tudi preostala jedra, za kar lahko rečemo, da je eden izmed načinov skaliranja NodeJS aplikacije. Deluje po principu gospodarja in sužnjev oziroma delovnih vozlišč (angl. *worker no-*

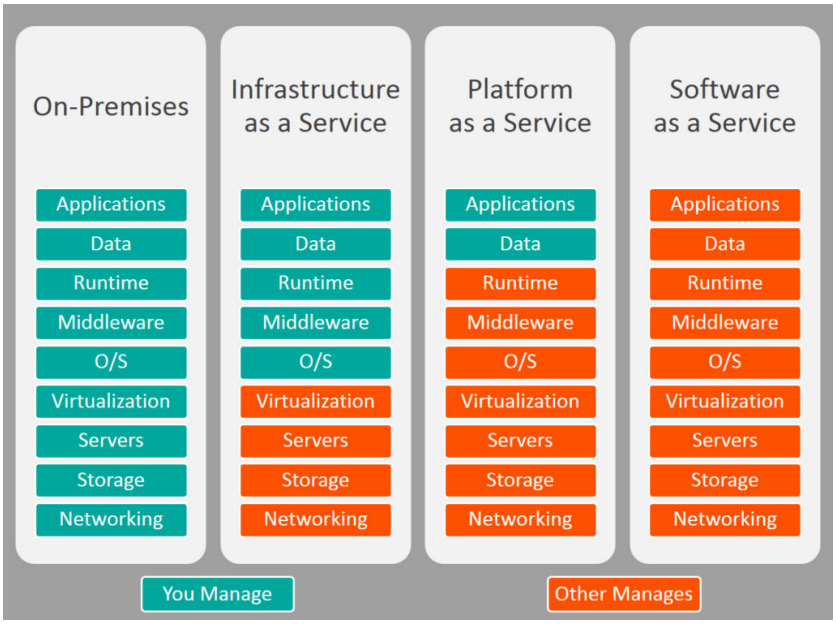
[1] Program *bombardier* je na voljo na naslovu `https://github.com/codesenberg/bombardier`

[2] Dokumentacija za ta modul je na voljo na naslovu `https://nodejs.org/api/cluster.html`

des). Bolj podrobno koncepta ne bomo opisovali, razvidno pa je, da tudi tu hitro dosežemo meje zmogljivosti pri enem samem strežniku. Problem skaliranja lahko rešimo tako, da prehod postavimo na močnejši strežnik. Tako smo pri nas prehod iz strežnika prestavili na bolj zmogljiv prenosnik. Koncept se imenuje vertikalno skaliranje in iz primerjalnega testa zmogljivosti na sliki 5.1 (B) ni razvidnih obetavnih rezultatov, saj smo mejo le dvignili za nekaj tisoč naprav. Test je tekel preko *localhost* omrežja in je, enako kot pri prejšnjem, poslal milijon zahtevkov preko sto petindvajset povezav. Ostane nam še koncept horizontalnega skaliranja, katerega bomo uporabili tudi za naš prehod. V grobem bomo uporabili več računalnikov oziroma v našem primeru več vsebnikov (angl. *containers*). Zaradi pomanjkanja lastne strojne opreme in stroškov zakupa le-te smo ta korak izvedli konceptualno.

Kot je sedaj že v navadi, smo za rešitev uporabili oblak. Precej enostaven način za to je, da uporabimo enega izmed ponudnikov PaaS (angl. *Platform as a Service*), ki podpira NodeJS. Na takšnih platformah skrbimo le za pisanje naše aplikacije, procesi postavitve, izenačevanja obremenitve, skaliranja in še mnogi drugi so upravljani s strani ponudnika in za nas tako rekoč nevidni. Takšni ponudniki so na primer AWS Elastic Beanstalk, Google App Engine in Heroku. Za lažje razumevanje bomo na kratko razložili osnove, kako bi lahko uporabili IaaS (angl. *Infrastructure as a Service*) in si zgradili svoj PaaS. Na sliki 5.2 je prikazana primerjava upravljanja virov pri različnih stopnjah zakupa storitev. Nismo še omenili SaaS (angl. *Software as a Service*), kjer smo le uporabniki določene aplikacije. Kot primer takšne storitve lahko vzamemo Dropbox, ki nudi oblačno shrambo za datoteke. Če se vrnemo nazaj na razliko med IaaS in PaaS, vidimo, da pri IaaS dobimo le dostop do strežnikov in moramo sami poskrbeti za postavitev naše aplikacije nanje, za možnosti skaliranja glede na uporabo, za nadzor samih procesov in podobno.

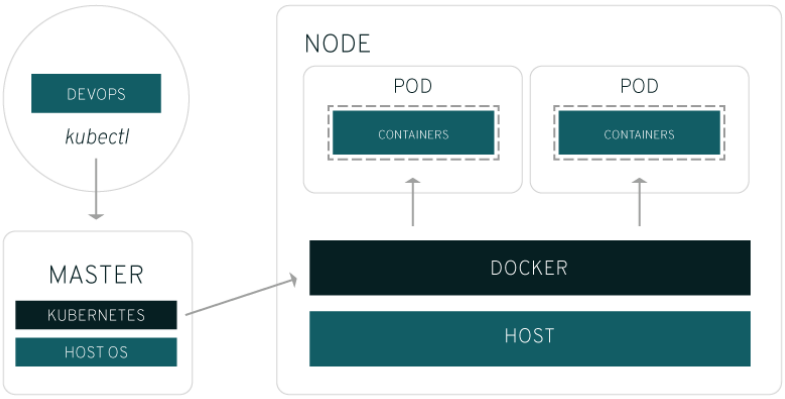
Da smo vse to storili bolj učinkovito, smo naš enostaven prehod vključili v vsebnik (angl. *container*). Vsebnik je lahek in samostojen paket programske opreme, ki vsebuje vse, kar je potrebno za njegov zagon [7]. Povedano



Slika 5.2: Primerjava upravljanja pri lastni strežniški arhitekturi, IaaS, PaaS in SaaS Vir: bmc.com, 2018

drugače, vsebnik z našo programsko opremo lahko prenašamo med okolji (na primer Windows in Linux) in bo deloval enako. V primerjavi z virtualnimi napravami vsebniki porabijo manj prostora, znotraj enega operacijskega sistema jih lahko teče več in zagon enega vsebnika je skoraj takojšen [7]. Ko imamo naš prehod znotraj vsebnika, lahko uporabimo sistem za avtomatizirano postavljanje, skaliranje in nadzor nad vsebniki. To nam nudi Kubernetes [16]. Poleg zaganjanja aplikacij nam nudi tudi nadzor zdravja aplikacij, tekočih posodobitev, porazdeljevanje obremenitve, skratka nudi podobne funkcionalnosti kot PaaS, pri čemer ohranimo zmožnosti IaaS. Seveda nam ne nudi vsega kar bi dobili pri pravem PaaS, je pa zato precej prilagodljiv uporabnikovim željam [16]. Za delo s Kuberenetes uporabljamo objekte *Kubernetes API*, s katerimi opišemo želeno stanje naše gruče. Povemo kateri vsebniki naj tečejo, kolikokrat naj bodo namnoženi, kateri viri jim naj bodo na voljo ... *Kubernetes Cluster Services* nato upravljajo delovna vozlišča in poskrbijo, da se stanje gruče prilagaja našim željam. Posplošena arhitektura je prikazana

na sliki 5.3. Kubernetes platforma je obširna in bi si zaslužila svoje poglavje za osnoven opis, ker pa to presega cilje te naloge, se bomo tu ustavili. Več o samih konceptih Kubernetesa si lahko preberemo na njihovi spletni strani [3].



Slika 5.3: Posplošena arhitektura Kubernetesa. Vir: redhat.com, 2018

Za konceptno skaliranje našega prehoda smo torej uporabili PaaS, saj prikrije večino procesov potrebnih za skaliranje aplikacije in je po našem mnenju bolj enostavno kot gradnja lastne infrastrukture. Prav tako za naše potrebe PaaS ponuja dovolj prilagodljivosti in ne najdemo pomanjkljivosti, ki bi nam preprečevala njegovo uporabo. V določenih primerih pa se lahko zgodi, da PaaS omejuje razvijalca. Takrat se lahko obrnemo na platforme, kot je Kubernetes.

[3]Dokumentacija Kubernetesa je na voljo na `https://kubernetes.io/docs/concepts/`

Poglavje 6

Upravljanje in vizualizacija zbranih podatkov

V tem poglavju bomo predstavili še četrti nivo, ki se nanaša na upravljanje s podatki. Sestavljen je iz primera MapReduce programa in vizualizacije podatkov zbranih preko naše platforme. Z MapReduce programom želimo poračunati povprečne temperature na vsake pol ure za merjeno obdobje. Rezultate programa lahko nato nazorno prikažemo v obliki stolpičnega diagrama [1]. V drugem delu pa bomo vizualizirali podatke z grafi in primeri shranjenih podatkov za vse tri naprave priključene v našo platformo.

Na voljo imamo podatke naše vremenske postaje za obdobje od 12. do 15. decembra leta 2017, kar je okoli 160.000 meritev temperature. Želimo si nazoren graf, s katerega lahko razberemo povprečno temperaturo za vsako polovico ure. Problem bi radi rešili s pomočjo programskega modela MapReduce. Za MapReduce program smo uporabili orodje Maven in urejevalnik IntelliJ. V meniju urejevalnika smo izbrali nov Maven projekt, obkljukali *Create from archetype* in na seznamu izbrali *maven-archetype-quickstart*. Pod *GroupId* smo vpisali *com.david.hadoop* in pod *ArtifactId* smo vnesli *average-temp*.

[1] Podoben diagram za urne vrednosti temperatur imajo tudi na spletni strani meteroloških in ekoloških podatkov za Koper na spletnem naslovu `http://193.95.233.105/econova1/Html/Urne_02.aspx?mesto=Koper`

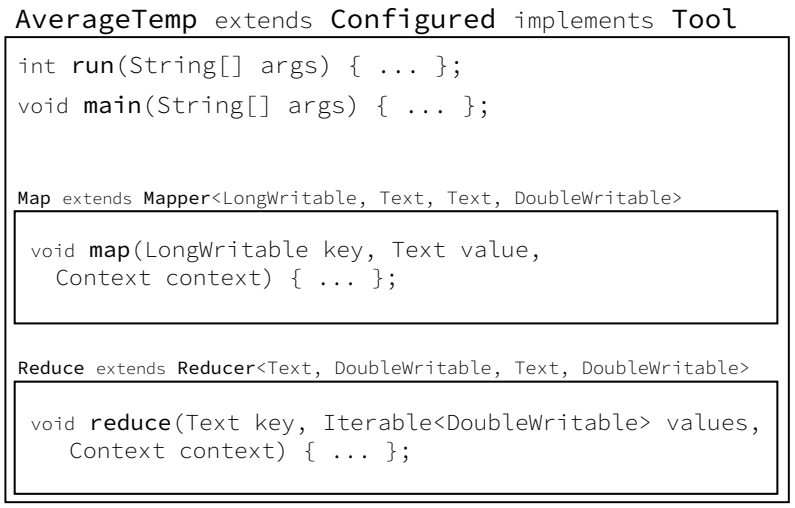
Ime projekta je *AverageTemp* (angleška okrajšava za povprečna temperatura). Vhodni podatki so JSON datoteke v sistemu HDFS. Na sliki 6.1 so podatki senzorja DS18B20 in BMP180, ki sta merila temperaturo, zračni tlak in nadmorsko višino, po katerih smo oblikovali svoj Java objekt. Imenovali smo ga *RaspberryPiDataType*. Vsebuje identifikator naprave in seznam meritev senzorjev. Seznam smo poimenovali *RaspberryPiData* in vsebuje časovni posnetek dogodka, ime senzorja in do tri vrednosti za ta senzor.


```
{
    "id": "RaspberryPi_2_00000000b28bef06",
    "data": [
        ... {
            "values": {
                "y": 97917,
                "x": 22.9,
                "z": 287.76122635387765
            },
            "timestamp": 1513192809.937962,
            "sensor": "bmp180"
        }, {
            "values": {
                "x": 23.062
            },
            "timestamp": 1513192810.837529,
            "sensor": "ds18b20"
        }, ...
    ]
}
```


Slika 6.1: Izsek iz datoteke s podatki vremenske postaje po kateri smo modelirali Java objekt. Vir: lasten, 2018

V nadaljnjem koraku pa smo se lotili pisanja *map* in *reduce* funkcij, zato smo dodali novo odvisnost v datoteko *pom.xml* na *hadoop-client*, ki vsebuje osnovne funkcije za pisanje MapReduce programov. V času pisanja smo uporabljali verzijo 2.9.0. Za boljšo preglednost bomo v nadaljevanju namesto *org.apache.hadoop* pisali kar *hadoop*. Ustvarili smo nov razred imenovan *AverageTemp*, ki razširja *hadoop.conf.Configured* in implementira vmesnik *hadoop.util.Tool*. Na sliki 6.2 je prikazan objektni diagram nad strukturo programa (razredi in funkcije, ki ga sestavljajo).

Znotraj razreda smo ustvarili še razred *Map*, ki razširja razred *hadoop.mapreduce.Mapper* in sprejme štiri argumente. Vsi so podatkovni tipi in to



Slika 6.2: Objektni diagram našega MapReduce programa. Vir: lasten, 2018

za vhodni ključ, vhodno vrednost, izhodni ključ in izhodno vrednost. Razred zahteva, da so ti tipi iz razreda *hadoop.io*, zato so, če jih zapišemo po vrsti, *LongWritable*, *Text*, zopet *Text* in *DoubleWritable*. Zatem smo prepisali metodo *map*, ki sprejme *LongWritable* ključ, *Text* vrednost in *hadoop.mapreduce.Mapper.Context* kontekst. Za deserializacijo smo uporabili GSON, ki omogoča, da poskusimo vhodno datoteko pretvoriti v naš razred *RaspberryPiDataType*. V primeru, da se proži izjema *JsonSyntaxException*, vse nadaljnje korake preskočimo, saj nam izjema pove, da JSON v datoteki ne ustreza našemu Java objektu, torej ne izvira iz vremenske postaje. V primeru, da deserializacija uspe, se z *for* zanko sprehodimo po posameznih meritvah, preverimo ime senzorja pri meritvi in če se ujema z našim, prožimo nov *context.write()* s ključem, za katerega smo se odločili, da bo enolično določal polovico ure v času meritev in seveda z vrednostjo v tistem času. Za primer vzemimo vhodno dvojico meritve s časom 19:33:45 in vrednostjo temperature 19.270 stopinj Celzija. Izhod v tem primeru je časovni posnetek za čas 19:30:00 in vrednost 19.270.

Podobno kot razred *Map* smo ustvarili še razred *Reduce* z argumenti tipa *Text*, *DoubleWritable* in še enkrat *Text* in *DoubleWritable*. Ti argumenti

nam kažejo, da so vhodi metode enaki kot naši izhodi. Sprejemamo tekstovni ključ (časovni posnetek kreiran malce višje) in vse številčne vrednosti temperature za ta ključ. Prepisali smo metodo *reduce*, ki pri nas sprejme ključ tipa *Text*, vrednosti tipa *Iterable<DoubleWritable>* in kontekst tipa *hadoop.mapreduce.Reducer.Context*. V metodi se sprehodimo po vseh vrednostih, jih seštejemo, na koncu pa prožimo še *context.write(čas, new DoubleWritable(seštevek / dolžinaSeznama)*. Časovni posnetek smo pretvorili iz sekund v uporabniku bolj prijazen format za izpis.

Pri konfiguraciji za zagon našega programa smo uporabili prej omenjeni razred *Configured* in vmesnik *Tool*. V metodi *main* kličemo metodo *hadoop.util.ToolRunner.run*, ki sprejme 3 argumente. Prvi je konfiguracija tipa *hadoop.conf.Configuration*, ki je pri nas kar nova (*new Configuration()*). Drugi je orodje, kar je naš razred *AverageTemp*, tretji pa se uporablja za prenos argumentov iz metode *main* v metodo *run*. Znotraj razreda *AverageTemp* smo prepisali metodo *run* in v njej ustvarili nov *hadoop.mapreduce.Job* z imenom „povprečna temperatura“. Podali smo še razrede, ki predstavaljajo vhodne in izhodne podatke, pot do vhodnih datotek, pot do izhodne datoteke in povezali razrede za *map* in *reduce*. Na koncu smo opravilo pognali z *job.waitForCompletion(true)*, kar v osnovi pomeni, da program počaka, da se opravilo izvede in sproti izpisuje dogajanje na zaslon.

Za zagon tega programa smo s pomočjo orodja Maven generirali JAR datoteko. Za ta korak smo v *pom.xml* definirali *maven-compiler-plugin*. Z ukazom *mvn clean install* v mapi projekta smo ustvarili novo datoteko z imenom *average-temp-1.0.SNAPSHOT.jar*. Datoteko smo prenesli na strežnik, kjer teče Hadoop in pognali še ukaz, ki program doda v vrsto za izvedbo: programu *hadoop* povemo, da želimo pognati JAR, mu podamo pot do te JAR datoteke, glavni razred znotraj datoteke, nato pa še argumente za metodo *main* znotraj razreda. Ukaz, ki vse to izvede za naš primer, je sledeči:

hadoop jar average-temp-1.0-SNAPSHOT.jar com.david.hadoop.AverageTemp /user/david/2017/dec//* /user/david/average_temp/output*

Prva pot predstavlja vhod in pomeni, da naj pregleda vse podmape dni v decembru leta 2017 in pregleda vse podmape ur posameznega dneva. Druga pot pa je izhodna mapa, kamor se bo ustvarila datoteka z izhodnimi vrednostmi. Če poiščemo to datoteko, bomo (med drugimi) zasledili naslednje vrstice:

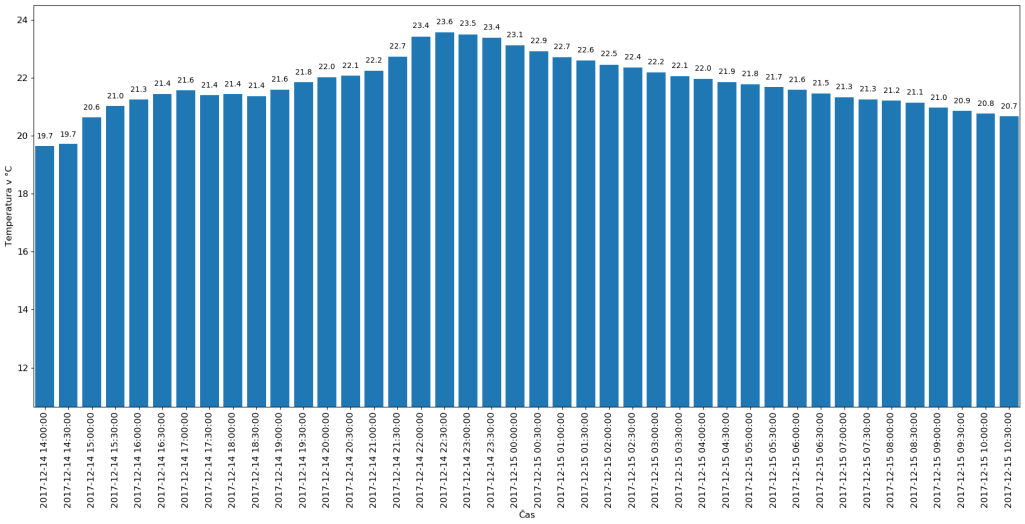
14.12.2017T14:00:00 19.659241057542715

14.12.2017T14:30:00 19.722253918495237

14.12.2017T15:00:00 20.64440909090904

14.12.2017T15:30:00 21.031826709061953

ki prikazujejo povprečne temperature 14.12.2017 v časovnem obdobju od 14:00 do 15:30. S pomočjo te datoteke lahko sedaj ustvarimo še stolpični diagram, izsek katerega je prikazan na sliki 6.3.



Slika 6.3: Izsek povprečnih temperatur za obdobje od 14. do 15. decembra 2017. Posamezen stolpec predstavlja polovico ure. Vir: lasten, 2018

6.1 Vizualizacija

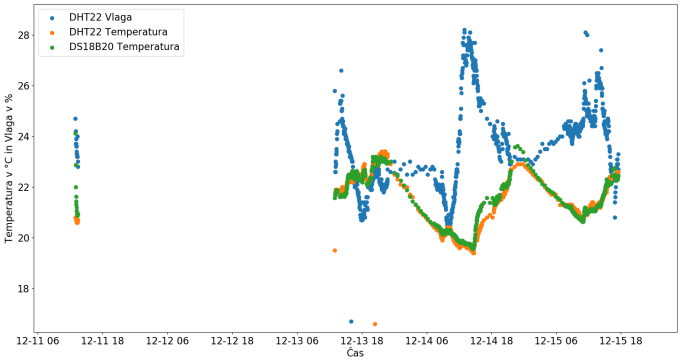
Za vse naprave, ki so nam bile na voljo, smo preverili, ali so podatki, ki so se pridobili preko njih, na voljo za obdelavo. Za dokaz tega smo te podatke vizualizirali s pomočjo Python knjižnice *matplotlib*. Za vsako izmed naprav smo spisali program, ki se sprehaja po datotekah shranjenih v HDFS, jih prebere in ugotovi kateri napravi pripada. Datoteke naprave Raspberry Pi smo ločili kar po strukturi JSON, saj je v njih objekt, medtem ko so v datotekah za Arduino in Android JSON seznami. Slednja lahko ločimo po enoličnem identifikatorju, katerega smo zasnovali tako, da se lahko razbere tudi vrsta naprave. V grafih je na abscisni osi vedno čas, na ordinatni osi pa so vrednosti senzorjev.

6.1.1 Vremenska postaja

Za izris grafov pri vremenski postaji smo zaradi velike količine podatkov in posledično visoke obremenitve nad knjižnico *matplotlib* vzeli le vsako 50. vrednost. Na sliki 6.4 lahko vidimo graf narejen iz podatkov temperature, katere smo uporabili za prej opisani MapReduce program. Na istem grafu so zabeležene še vrednosti vlage in temperature senzorja DHT22 za isto časovno obdobje. Na sliki 6.5 pa je na voljo še graf nad podatki zračnega tlaka, ki se je meril v istem obdobju.

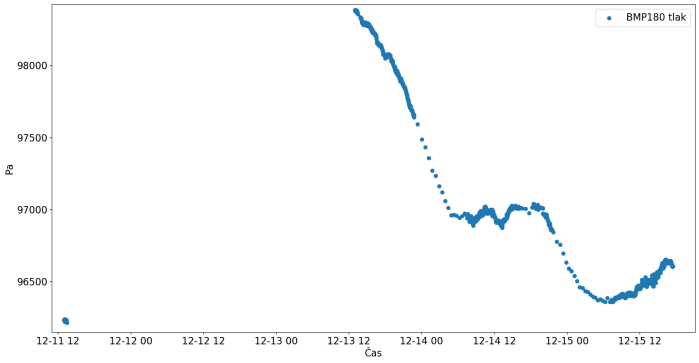
6.1.2 Rastlina, ki se zaliva sama

V začetku februarja leta 2018 smo tudi zbirali podatke o rastlini, katera je opisana v poglavju 3.1. Primer strukture prebranega podatka je na sliki 6.6. Na sliki 6.7 so prikazane vrednosti iz obeh senzorjev vlage. Vlaga je bila na začetku nizka, kar senzor prikaže z visokimi vrednostmi. Bralec z ostrim očesom bo morda opazil nenaden padec vrednosti vlage, ki se je zgodil med 2. in 3. februarjem. Razlog tega padca vrednosti je poškodba, ki se je pripetila že nekoliko prej na enem izmed senzorjev. Našli smo prekinjeno povezavo do senzorja, rešitev pa je bila menjava modula senzorja v zemlji. Preven-



Slika 6.4: Vrednosti temperature in vlage prebrane na RaspberryPi. Vir: lasten, 2018

tivno smo preverili tudi drugi senzor, ki je bil nepoškodovan. Pri ponovni namestitvi senzorjev v zemljo so se nekoliko zamaknile prvotne pozicije teh in posledica tega je prikazan padec v vrednosti vlage. Pri grafu vrednosti fotocelice na sliki 6.8 se lepo vidijo razlike med dnevom in nočjo, pri čemer visoke vrednosti predstavljajo več svetlobe oziroma dan. Seveda pa so se tudi tukaj pojavile težave. Na začetku so listi občasno zakrivali senzor, umik slednjih pa je povzročil nenaden poskok vrednosti svetlobe, ki ga na grafu opazimo dvakrat med 1. in 3. februarjem. Med nočnim časom pa prav tako opazimo poskoke v vrednosti svetlobe. Razlog teh pa je luč, ki je bila prižgana v sobi. Iz obeh grafov so razvidna tudi obdobja, kjer vrednosti bodisi manjkajo bodisi odstopajo od ostalih vrednosti. Eden izmed razlogov je izpad omrežja, ki se je pripetil med 6. in 7. februarjem. Med podatki smo pa opazili tudi popačenost ključev in vrednosti, za katere domnevamo da so se zgodili pri serijski komunikaciji med Arduinom in ESP8266 (na primer namesto pričakovanega ključa *timestamp* se je shranila vrednost *timesTamp*).



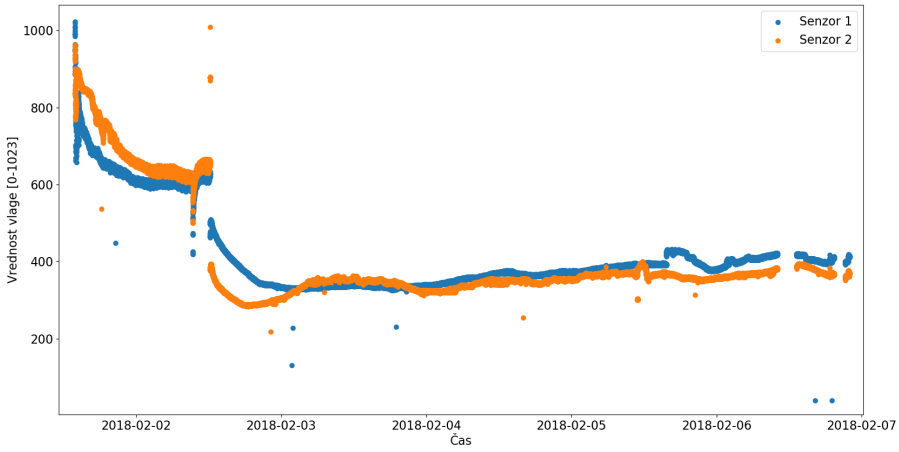
Slika 6.5: Vrednosti zračnega tlaka prebranega na RaspberryPi. Vir: lasten, 2018


```
[
    ... {
        "id": "Arduino_Mega_2560_Leon",
        "timestamp": 1517623065,
        "sensor": "moisture",
        "values": {
            "x": 330,
            "y": 346,
        }
    }, ...
]
```

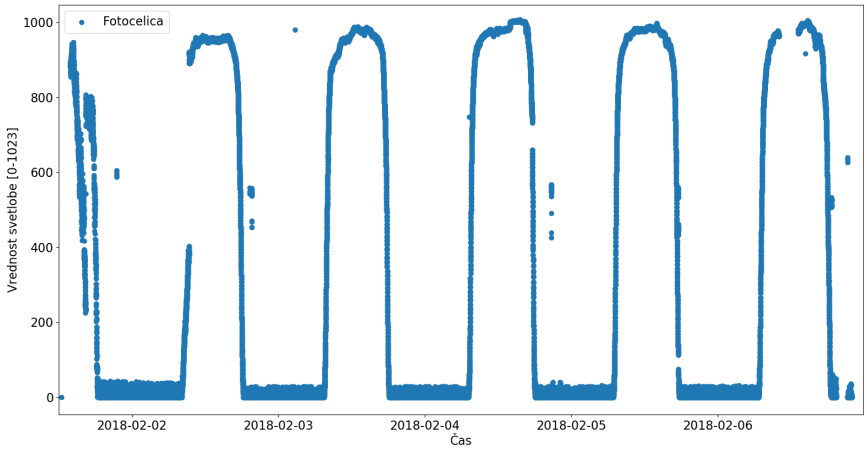

Slika 6.6: Primer shranjenega podatka o vlagi zemlje pri rastlini. Vir: lasten, 2018

6.1.3 Mobilna aplikacija

Podatke pa smo zbirali tudi na naši mobilni aplikaciji. Za raznolikost med zbranimi podatki je bilo napravo z aplikacijo potrebno uporabljati, tako da je časovni interval nad zbranimi podatki tukaj manjši kot pa pri prejšnjih napravah. Primer shranjenega podatka meritve pospeškometra je prikazan na sliki 6.9. Graf nad podatki pospeškometra je prikazan na sliki 6.10. Malo pred 11. uro dopoldan smo aplikacijo vključili in jo postavili na mizo za določevanje začetnih vrednosti pri mirovanju. Nato smo v intervalih po deset minut menjavali med fizičnimi aktivnostmi in mirovanjem. Prvi interval



Slika 6.7: Vrednosti vlage v zgornjih 5 cm zemlje prebrane na naši rastlini. Vir: lasten, 2018



Slika 6.8: Vrednosti fotocelice prebrane v bližini naše rastline. Vir: lasten, 2018

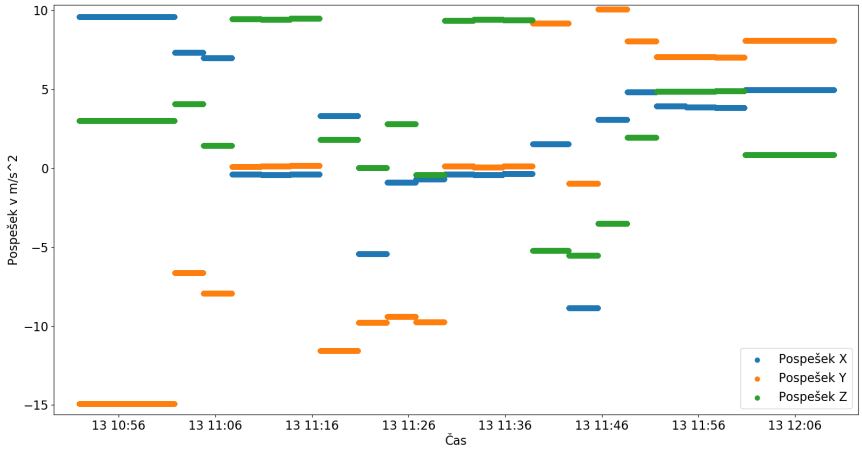
smo hodili po hiši z napravo v roki, kar se na grafu okoli 11. ure vidi kot sprememba vrednosti v pospeških. Nato je sledilo deset minut mirovanja, pri čemer je naprava ležala na mizi. Naslednjih deset minut smo napravo

nosili s sabo v hlačnem žepu, kar se na grafu zopet vidi kot sprememba v vrednostih pospeška. Po še enem deset minutnem premoru smo telefon nosili s sabo v oprsnem žepu na srajci. Proti koncu, okoli deset minut pred opoldnevom, smo z napravo v žepu še sedeli na stolu in opravljali delo za računalnikom. V istem obdobju smo merili še vrednosti barometra, ki je bil na voljo na napravi. Graf nad temi podatki je prikazan na sliki 6.11. Tukaj lahko večkrat opazimo poskok v vrednosti tlaka. Razlog za to je preprost. Kot smo že omenili, se je v času zbiranja podatkov naprava premikala po hiši, kar je vključevalo tudi premikanje med posameznimi nadstropji. S takšno spremembo nadmorske višine se je spremenila tudi vrednost zračnega tlaka. S temi podatki in s podatki o temperaturi in zračnem tlaku v Kopru istega dne ob istem času [2] lahko poračunamo višinsko razliko, ki je pri nas znesla okoli pet do šest metrov. Vrednost nakazuje na razliko v višini treh nadstropij, kar pa se sklada z dejanskimi dogodki.

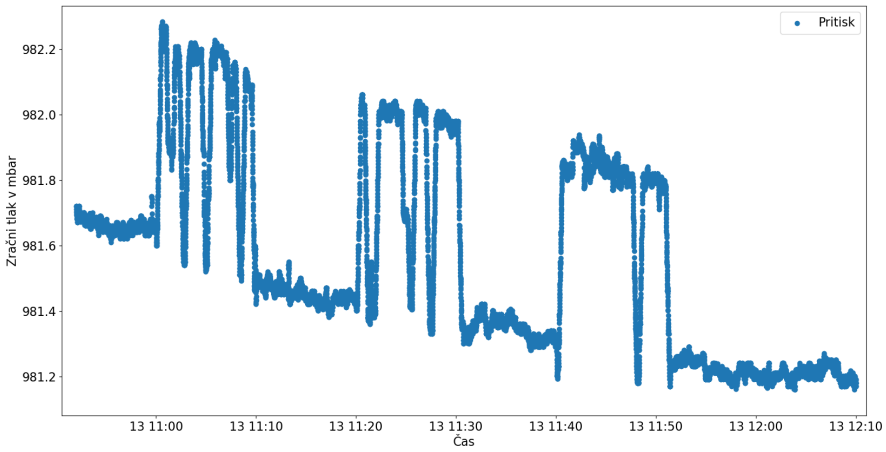

```
[
    ... {
        "ID": "357478060482889",
        "batteryData": {
            "batteryLevel": 73,
            "isCharging": false
        },
        "sensor": "MPU6515 Accelerometer",
        "sensorType": "android.sensor.accelerometer",
        "timestamp": "1518519650775",
        "values": [4.972519, 8.104797, 0.8689728],
        "wifiData": {
            "connected": true,
            "signalStrength": -49,
            "ssid": "Wifi david"
        }
    }, ...
]
```


Slika 6.9: Primer shranjenega podatka o meritvi pospeškometra. Vir: lasten, 2018

[2] Podatki so vzeti iz spletne strani `http://193.95.233.105/econova1/Html/Urne_02.aspx?mesto=Koper`. Vzeli smo temperaturo zraka 6.7 °C in zračni tlak 1007.2 kPa



Slika 6.10: Vrednosti pospeškometra pridobljene preko naše mobilne aplikacije. Vir: lasten, 2018



Slika 6.11: Vrednosti barometra pridobljene preko naše mobilne aplikacije. Vir: lasten, 2018

Poglavje 7

Sklepne ugotovitve

Živimo v času, ko je relativno lahko zbirati podatke o nekem okolju. Problem, ki pri tem lahko nastane, je obvladovanje nekega sistema, kjer količina podatkov preseže določene vrednosti in si s tradicionalnimi rešitvami relacijskih podatkovnih baz ne moremo več pomagati. Kot rešitev tega problema, smo v tem delu zasnovali platformo, ki deluje po principih uporabljenih pri obvladovanju velike količine podatkov. Za demonstracijo delovanja smo razvili tudi nekaj primerov aplikacij za zbiranje podatkov v različnih okoljih. Strežnik, na katerem je tekel Hadoop smo preko prehoda povezali s temi „zbiratelji podatkov“ in tudi grafično prikazali nekaj izmed zbranih vrednosti. Za strežniški del (postavitev Hadoopa in prehod za podatke) smo tudi konceptualno opisali, kako se lahko skalira in uporabi za tisoče naprav, ki generirajo podatke. Prikazali smo tudi primer programa spisanega po programskem modelu MapReduce, ki se uporablja za porazdeljeno procesiranje podatkov. Del podatkov iz naših primerov senzorskih naprav smo tudi vizualizirali in prišli do nekaj ugotovitev. Pri nameščanju senzorja svetlobe na rastlino je potrebno paziti, da listi ne morejo prekriti senzorja in povzročiti napačne vrednosti. Na isti senzor lahko tudi vpliva umetni vir svetlobe, kot je luč v večernem času. Pri vremenski postaji smo ugotovili, da temperatura v sobi niha med 19.5 in 23 stopinj Celzija, na mobilni aplikaciji pa smo s pomočjo vgrajenega barometra lahko poračunali približno višinsko razliko,

ki jo je uporabnik premagal med zbiranjem podatkov.

Za Hadoop, ki se uporablja pri velikih podatkih, bi na prvi pogled rekli, da je za demonstracijo njegove uporabe potrebna draga gruča računalnikov. V tem delu smo prikazali, da temu ni tako, saj smo delujočo postavitev imeli na precej starem računalniku. Seveda je to daleč od zmogljivosti prave postavitve Hadoop platforme z več tisoč vozlišči, smo pa sistem zasnovali tako, da bi moral delovati tudi na takšnih razsežnostih. Ugotovili smo tudi, da so tehnologije za obvladovanje velikih podatkov lahko dostopne, nekatere tudi odprtokodne. Težava je v tem, da za izrabo prednosti, ki jih takšne tehnologije prinesejo, potrebujemo veliko strežniške opreme. Tako menimo, da je v večini primerov, ko si želimo porazdeljeno procesiranje nad podatki, ceneje koristiti ponudnike Hadoop storitve, kot pa kupiti in postaviti lastno gručo strežnikov. Če je nekomu, ki procesira veliko podatkov, na voljo uporaba gruče računalnikov, bi mu vsekakor predlagali tudi MapReduce programski model. Kot argument za uporabo naj še enkrat omenimo, da jim je že pred kar nekaj leti uspelo razvrstiti 1 terabajt podatkov v skoraj minuti [19].

Za vse nadaljnje korake pri razvoju naše platforme bi namesto konceptualnih skaliranj morali vpeljati dejanske implementacije teh. Kot začetek bi potrebovali veliko več naprav, ki generirajo podatke. S številom teh naprav bi lahko zasnovali obseg gruče v Hadoop platformi, katero bi najverjetneje imeli postavljeno kar pri enem izmed ponudnikov oblačnih storitev. S skaliranjem naprav in Hadoop gruče bi nato sledilo še nadgrajevanje prehoda, pri katerem bi se gotovo pojavile težave zaradi večjega števila zahtevkov na časovno enoto. V takšnem scenariju bi se nam postavila vprašanja o obsegu strežnikov v Hadoop gruči, o zmogljivosti posameznega strežnika, o učinkovitosti prehoda, o stroških in o varnosti platforme.

Literatura

[1] Welcome to apache™ hadoop®! http://hadoop.apache.org/, 2017. [Spletni vir; dostopano 30.1.2018].

[2] Gartner Inc. and/or its Affiliates. Iot. http://view.ceros.com/gartner/iot/p/1, 2016. [Spletni vir; dostopano 25.7.2017].

[3] Ryan Andrew. Under the hood: Hadoop distributed filesystem reliability with namenode and avatarnode. https://www.facebook.com/notes/facebook-engineering/under-the-hood-hadoop-distributed-filesystem-reliability-with-namenode-and-avata/10150888759153920, Junij 2012. [Spletni vir; dostopano 23.10.2017].

[4] Murthy C. Arun. Introducing apache hadoop yarn. Avgust 2012. [Spletni vir; dostopano 23.10.2017].

[5] ICT Data and Statistics Division. Ict facts and figures 2016. http://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2016.pdf, Junij 2016. [Spletni vir; dostopano 25.7.2017].

[6] Niewolny David. How the internet of things is revolutionizing healthcare. https://cache.freescale.com/files/corporate/doc/white_paper/IOTREVHEALCARWP.pdf, Oktober 2013. [Spletni vir; dostopano 23.7.2017].

[7] Docker. What is a container. https://www.docker.com/what-container, 2018. [Spletni vir; Dostopano 2.2.2018].

[8] Sammer Eric. *Hadoop Operations*. O'Reilly Media, Inc., 2012.

[9] Inc. Gartner. Gartner says worldwide sales of smartphones grew 9 percent in first quarter of 2017. https://www.gartner.com/newsroom/id/3725117, Maj 2017. [Spletni vir; Dostopano 18.2.2018.

[10] Press Gil. Internet of things by the numbers: What new surveys found. https://www.forbes.com/sites/gilpress/2016/09/02/internet-of-things-by-the-numbers-what-new-surveys-found/#48d6585b16a0, September 2016. [Spletni vir; dostopano 23.7.2017].

[11] IDC. The digital universe of opportunities: Rich data and the increasing value of the internet of things. https://www.emc.com/leadership/digital-universe/2014iview/index.htm, April 2014. [Spletni vir; dostopano 23.7.2017].

[12] Cloudera Inc. Cloudera company background information and management and board. https://www.cloudera.com/more/about.html, 2017. [Spletni vir; dostopano 29.10.2017].

[13] Morgan Jacobl. A simple explanation of 'the internet of things'. https://www.forbes.com/sites/jacobmorgan/2014/05/13/simple-explanation-internet-things-that-anyone-can-understand/#2eab7f6a1d09, Maj 2014. [Spletni vir; dostopano 25.7.2017].

[14] Dean Jeffrey and Ghemawat Sanjay. Mapreduce: Simplified data processing on large clusters. https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf, 2004.

[15] Shvachko V. Konstantin and Murthy C. Arun. Scaling hadoop to 4000 nodes at yahoo! September 2008. [Spletni vir; dostopano 23.10.2017].

[16] Kubernetes. What is kubernetes? https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/, 2018. [Spletni vir; Dostopano 2.2.2018.

[17] Gaillard Mélissai. Cern data centre passes the 200-petabyte milestone. https://home.cern/about/updates/2017/07/cern-data-centre-passes-200-petabyte-milestone, Julij 2017. [Spletni vir; dostopano 24.7.2017].

[18] Arun C. Murthy, Chris Douglas, Mahadev Konar, Owen O'Malley, Sanjay Radia, Sharad Agarwal, and Vinod K V. Architecture of next generation apache hadoop mapreduce framework. 2011. [Spletni vir; dostopano 23.10.2017].

[19] O'Malley Owen and Murthy C. Arun. Winning a 60 second dash with a yellow elephant. April 2009. [Spletni vir; dostopano 23.10.2017].

[20] Ghemawat Sanjay, Gobioff Howard, and Leung Shun-Tak. The google file system. Oktober 2003. [Spletni vir; dostopano 23.10.2017].

[21] Sprager Sebastijan and Zazula Damjan. A cumulant-based method for gait identification using accelerometer data with principal component analysis and support vector machine. *WSEAS Transactions on Signal Processing*, 5(11):369–378, 2009.

[22] White Tom. *Hadoop: The Definitive Guide, Fourth Edition*. O'Reilly Media, Inc., 2015.

[23] Wikipedia. Comparison of relational database management systems — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Comparison%20of%20relational%20database%20management%20systems&oldid=791156337, 2017. [Spletni vir; dostopano 23.7.2017].

[24] Lanich Zach. The benefits of moving to the cloud. https://www.forbes.com/sites/forbestechcouncil/2017/05/19/the-

benefits-of-moving-to-the-cloud/, May 2017. [Spletni vir; Dostopano 30.1.2018].