SUSTAVI S DISTRIBUIRANIM PROCESIRANJEM

Vidaković, Dominik

VISOKO UČILIŠTE ALGEBRA

ZAVRŠNI RAD

SUSTAVI S DISTRIBUIRANIM PROCESIRANJEM

Dominik Vidaković

Zagreb, siječanj 2019.

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

U Zagrebu, 22.1.2019.

Predgovor

Zahvaljujem se profesorima Visoke Škole za Primijenjeno Računarstvo radi svih znanja koje su uz strpljenje i dobru volju prenijeli na mene. Posebno se zahvaljujem svome mentoru prof. Aleksanderu Radovanu radi pomoći pri izradi ovoga završnog rada.

Također se zahvaljujem svojoj obitelji jer bez njih ne bi imao priliku studirati, te bez njihove podrške ne bi dogurao do izrade završnog rada.

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

Sažetak

Ovaj rad pokazuje prednosti korištenja paralelnih distribuiranih sustava pri razvijanju softvera namijenjenog za procesiranje ogromnih količina podataka. Prvo su objašnjeni osnovni pojmovi vezani za distribuirane sustave, nakon toga su predstavljene tri tehnologije koje se koriste pri razvijanju distribuiranih softverskih rješenja. Te tehnologije su Apache Hadoop, Apache Spark i Apache Storm. Rad također pokazuje primjer razvoja, pokretanja i eksperimentiranja s aplikacijom napravljenom za rad u distribuiranom sustavu. Aplikacija će biti programirana u programskom jeziku Java koristeći Apache Storm tehnologije. Njena svrha bit će prepoznavanje vremenskih nepogoda na bazi dolaznih podataka u stvarnom vremenu. Rezultat završnog rada je aplikacija spremna za rad u distribuiranom sustavu te spoznaje programera koji prvi puta radi s distribuiranim rješenjima.

Ključne riječi: podaci, distribuiran, sustav, Storm.

ABSTRACT

This thesis aims to show the advantages of using parallel distributed systems when developing software for processing huge amounts of data. First, basic terms regarding distributed systems will be explained, after that, three different technologies intended for usage in developing distributed software solutions will be presented. Those technologies being Apache Hadoop, Apache Spark and Apcahe Storm. This work also shows an example of development, deployment and experimenting with a application made to work in a distributed system. The application will be programmed in Java while using Apache Storm, and its purpose will be recognizing weather resulting in disaster on the basis of incoming weather data. The result of this thesis will be an application ready for deployment in a distributed system, and a brief comprehension by a programmer who for the first time developed with a distributed system in mind.

Keywords: data, distributed, system, Storm

Sadržaj

1. Uvod

Informacijsko doba i rast korištenja računala u poslovne i osobne svrhe došao je s ogromnom količinom podataka koju ta ista računala nisu sposobna procesirati, ako rade samostalno. Informacija se smatra najvrjednijim resursom u poslovanju. Informacija je temelj na kojemu se donose odluke. Oskudne informacije rezultiraju u nesigurnosti pri donošenju odluka, a zbog te nesigurnosti dolazi do loših odluka. Zato je u poslovanju ključno pribaviti sve potrebne informacije što prije moguće.

Kako se dolazi do informacija? Informacije se dobivaju analizom skupova podataka, a u modernom poslovanju ti skupovi mogu biti veliki do stotine i stotine TB, možda i tisuće. Toliko podataka jedno računalo ne može procesirati u prihvatljivom vremenskom roku. Jedino rješenje je skup hardvera i softvera razvijen za procesiranje podataka u što bržem roku, paralelni distribuirani sustav.

U idućem poglavlju rada bit će objašnjeni osnovni pojmovi vezani za distribuirane sustave. Nakon toga će biti objašnjene 3 tehnologije koje se baziraju na rad u distribuiranom sustavu, a to su Apache Hadoop, Apache Storm i Apache Spark. Zatim slijedi dio rada koji prolazi kroz korake razvoja i pokretanja aplikacije napravljene uz pomoć Storm tehnologija.

2. Osnove distribuiranih sustava

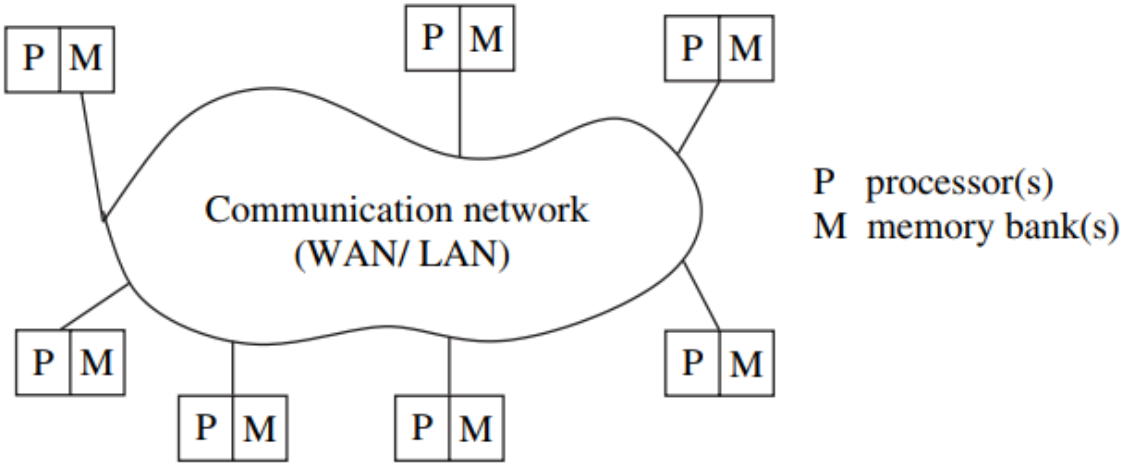
Distribuirani sustav je kolekcija samostalnih entiteta koji zajedno rade da bi riješili problem koji individualno ne bi mogli riješiti.

Po toj definiciji distribuirani sustavi su posvuda u prirodi, npr. ljudsko tijelo je distribuirani sustav zato što se sastoji od više organa koji sami ne mogu puno postići, ali zajedno čine biće koje je sposobno raditi čuda. Isto tako se može reći da je čopor vukova distribuirani sustav sastavljen od predatora koji su evoluirali da što učinkovitije love, a učinkovitiji su kada love zajedno, nego kada love sami.

To su bili primjeri distribuiranih sustava u prirodi, ali u ovom radu se istražuju distribuirani sustavi u računalnoj znanosti i njihovi oblici. Distribuirani sustav u računalnoj znanosti se definira kao skup nezavisnih procesora koji komuniciraju preko određene komunikacijske mreže, te imaju iduća svojstva:

- Nemaju zajednički radni takt – procesori ne moraju raditi sinkrono, tj. u istom vremenu.
- Nemaju zajedničku memoriju – ovo se odnosi na fizičku memoriju kojoj bi procesori pristupali u isto vrijeme, ali pošto nemaju usklađena vremena, tako nešto bi bilo teško izvedivo. Koriste se distribuirane memorije kojima se šalju poruke za komunikaciju.
- Geografska odvojenost - procesori ne moraju biti jako udaljeni jedan od drugoga, ali njihova udaljenost je ono što ih čini distribuiranim sustavom, u tome je i smisao cijele priče. Najpoznatiji primjer bio bi internet, distribuirani sustav koji može povezati sva računala bez obzira na njihovu geografsku lokaciju. Svi ti sustavi gube na brzini i učinkovitosti što su međusobno udaljeniji procesori, pogotovo sustavi s paralelnim procesiranjem koji su razvijeni za svrhu brze obrade podataka. U slučaju paralelnog procesiranja procesori bi trebali biti spojeni u LAN mreži.
- Autonomnost i heterogenost – procesori mogu raditi u različitim brzinama i izvoditi različite operacijske sustave. Rad u specifičnom distribuiranom sustavu im ne mora biti primarna svrha. (Ajay D.Kshenaklyani, 2008)

Svako računalo ima procesor i memoriju, procesor obavlja kalkulacije, dok memorija sadrži naredbe za procesor, te sva računala komuniciraju preko mreže kao što je prikazano na slici 1.



Slika 1. Primjer distribuiranog sustava (Ajay D.Kshenaklyani, 2008)

2.1. Kada i zašto koristiti distribuirane sustave?

Postoje određene motivacije i razlozi za postojanje i razvoj distribuiranih sustava, evo par primjera:

- Inherentno distribuirano računanje – ako je potrebno doći do konsenzusa između dvije stranke koje su geografski udaljene jedna od druge, tada je funkcionalnost sustava inherentno distribuirana. Primjer su transferi sredstava u bankarstvu.
- Dijeljenje resursa – podataka ima previše da bi ih se sve pohranjivalo na jednom računalu, to je nespretno rješenje zato što bi to računalo bilo usko grlo (engl. *bottleneck*) u sustavu. Isto tako bi bilo preskupo sve podatke replicirati na svakom računalu. Zato se podaci distribuiraju po sustavu. Ako jedno računalo treba podatke koje nema, "znati" će gdje te podatke tražiti u sustavu, te kako doći do njih.
- Pristup podacima i resursima koji su geografski udaljeni – podataka može biti previše ili mogu biti preosjetljivi da bi ih se repliciralo na više lokacija, npr. stanja bankovnog računa ili ostale povjerljive informacije. Takve informacije se drže u centralnom serveru koji prima upite od strane računala koja imaju ovlašteni pristup.
- Povećana pouzdanost – distribuirani sustavi imaju sposobnost repliciranja podataka na više lokacija, tako se umanjuje opasnost od gubitka podataka. Pouzdanost u distribuiranim sustavima se opisuje s ova 3 svojstva:
    - dostupnost – resursi bi trebali biti dostupni korisnicima u svakome trenutku
    - integritet – podaci moraju biti točni i ne smiju biti neželjeno alterirani

- tolerancija kvarova – sustav mora imati sposobnost oporavka od eventualnih kvarova

Brewerov teorem kaže da distribuirani sustav ne može u isto vrijeme imati savršenu dostupnost, integritet i toleranciju kvarova. Drugo ime za taj teorem je CAP teorem. (Bashir, 2017)

- Bolji omjer performanse i cijene – zbog dijeljenja resursa na više računala moguće je ta računala maksimalno iskoristiti, tako da jedan zadatak bude podijeljen na više manjih zadataka, te ti manji zadaci budu podijeljeni računalima u sustavu na obradu.
- Skalabilnost i modularnost – heterogeni procesori mogu biti dodani u sustav bez negativnog utjecaja na performanse, jedini uvjet je da procesori izvode iste međusoftverske (engl. *Middleware)* algoritme. Postojeći procesori u sustavu se isto tako mogu bez problema zamijeniti novim procesorima. (Ajay D.Kshenaklyani, 2008)

2.2. Paralelni sustavi

Paralelni sustavi imaju samo neke od gore navedenih svojstva distribuiranog sustava, te se dijele na 3 skupine: multiprocesorski sustavi, višeračunalni sustavi i *array* procesore.

2.2.1. Multiprocesorski paralelni sustav

Paralelni sustav u kojemu više procesora imaju izravan pristup dijeljenoj memoriji zove se multiprocesorski sustav. Njegovi procesori najčešće ne rade u zajedničkom taktu.

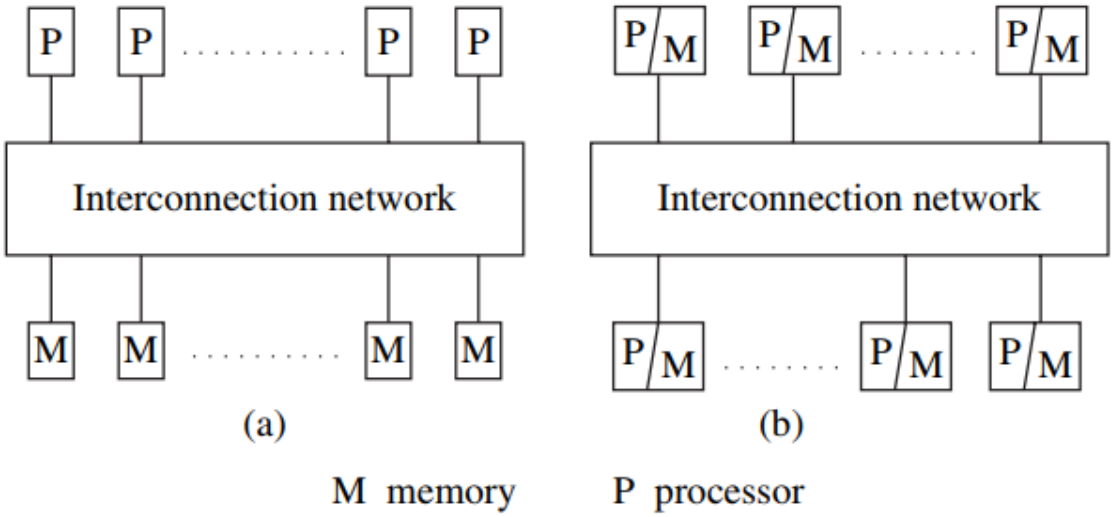
Multiprocesorski sustav komunicira preko *uniform memory access* (UMA) arhitekture u kojoj je vrijeme čekanja za pristup bilo kojem djelu memorije isti za svaki procesor u sustavu. Procesori su fizički jako blizu jedan drugome te su povezani interkonekcijskom mrežom (engl. *interconnection network*). Procesori, u pravilu, komuniciraju uz pomoć operacija za čitanje i pisanje (engl. *read and write*) na dijeljenoj memoriji, iako je uz emulaciju na dijeljenoj memoriji moguća i komunikacija uz pomoć slanja poruka.

Procesori izvode isti operacijski sustav te su im hardver i softver usko povezani. Procesori su istog tipa i drži ih se u istom kontejneru kao i dijeljena memorija. Interkonekcijska mreža preko koje procesori dolaze do memorije može biti sabirnica, ali najčešće se radi o *switch-u*

sa simetričnim dizajnom. Popularni načini izvedbe interkonekcijske mreže su *Omega* i *Butterfly* mreže. (Ajay D.Kshenaklyani, 2008)

2.2.2. Višeračunalni paralelni sustav

Višeračunalni paralelni sustavi sadrže se od više procesora koji nemaju izravan pristup dijeljenoj memoriji. Procesori su blizu jedan drugoga, usko su povezani i spojeni preko interkonekcijske mreže. Procesori međusobno komuniciraju prosljeđivanjem poruka ili preko zajedničkog memorijskog prostora. Višeračunalni sustav koji ima zajednički memorijski prostor najčešće komunicira uz pomoć *non-uniform memory access* (NUMA) arhitekture, u kojoj vrijeme potrebno procesoru da pristupi memoriji varira i ovisi o procesoru. Slika 2 daje jasniji prikaz razlika između UMA i NUMA sustava. (Andrew S. Tanenbaum, 2016)



Slika 2. UMA multiprocesorski sustav (a) i NUMA multiprocesorski sustav (b) (Ajay D.Kshenaklyani, 2008)

2.2.3. *Array* procesori

Array procesor je skup paralelnih računala koja se nalaze na istoj lokaciji, vrlo usko su povezani, imaju isti radni takt, ne dijele memoriju i komuniciraju prosljeđivanjem poruka. Radi se o procesorima koji sinkronizirano i u taktu procesiraju i razmjenjuju podatke u svrhu npr. obrade digitalnih signala ili obrade slike. Ovakvi procesori se koriste u veoma specifične i rijetke svrhe. (Ajay D.Kshenaklyani, 2008)

2.2.4. Svrha i primjena paralelnih sustava

Primarna korist paralelnih sustava je dobivanje boljih performansi pri radu i kalkuliranju, a to se postiže podjelom posla na više procesora. Paralelni sustavi najbolje rade posao koji se može lako podijeliti na više zadataka, koji ne zahtijeva puno komunikacije u svrhu sinkronizacije te se bazira na kalkulacijama numeričkih vrijednosti. (Ajay D.Kshenaklyani, 2008)

2.3. Važni pojmovi u distribuiranim sustavima

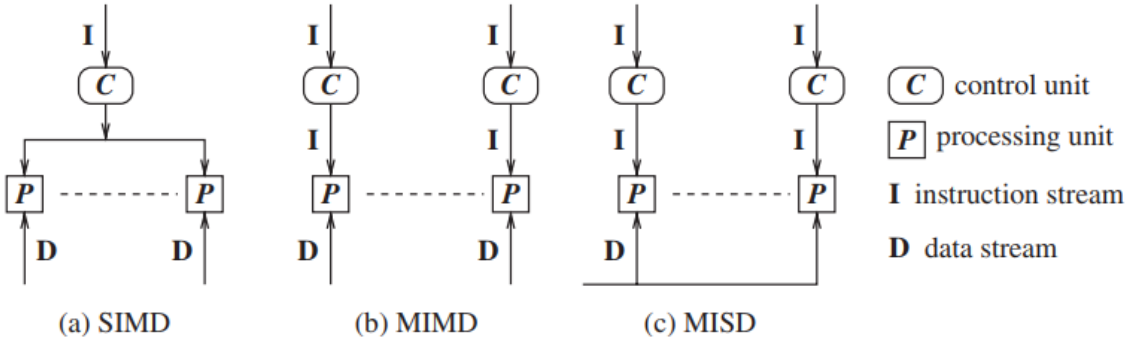
U ovome poglavlju pokriveni su pojmovi koje je potrebno poznavati za bilo kakvu diskusiju u vezi distribuiranih sustava, pojmovi poput Flinnove taksonomije, povezanost, paralelizma, konkurencije i granularnosti. (Ajay D.Kshenaklyani, 2008)

2.3.1. Flinnova taksonomija

Michael J. Flynn podijelio je načine procesiranja ovisno o tome izvršavaju li procesori iste ili različite instrukcijske nizove, i procesiraju li iste ili različite podatke u isto vrijeme (Ajay D.Kshenaklyani, 2008). Podjela je rezultirala 4 načina procesiranja koji su objašnjeni u idućim natuknicama i u slici 3:

- Jedan instrukcijski tok, jedan podatkovni tok (engl . *Single Instruction stream, Single Data stream*) (SISD) – ovaj način procesiranja najbolje predstavlja von Neumannova paradigma. Radi se o jednom procesoru i jednoj memoriji koji su povezani sabirnicom.
- Jedan instrukcijski tok, višestruki podatkovni tok (engl. *Single Instruction stream, Multiple Data stream*) (SIMD) – homogeni procesori koji rade u istom taktu te u isto vrijeme procesiraju različite podatke. Programi koji rade kalkulacije na ogromnim hrpama podataka koje se daju uredno podijeliti najbolje iskorištavaju SIMD način rada.
- Višestruki instrukcijski tok, jedan podatkovni tok (engl. *Multiple Instruction stream, Single Data stream*) (MISD) – ovaj način rada paralelno procesira iste podatke, ali na različite načine te se jako rijetko primjenjuje u praksi, jedan primjer bi bio vizualizacija. Procesori rade u istom radnom taktu.

- Višestruki instrukcijski tok, višestruki podatkovni tok (engl. *Multiple Instruction stream, Multiple Data stream*) (MIMD) – različiti procesori izvršavaju različite operacije nad različitim podacima. Onaj način rada se primjenjuje u distribuiranim i paralelnim sustavima.



Slika 3. Načini procesiranja (Ajay D.Kshenaklyani, 2008)

2.3.2. Povezanost

Povezanost govori o međuovisnosti dviju komponenti u nekom sustavu. Te komponente mogu biti hardver ili softver. Ako su komponente homogene, onda se može reći da su slabo povezane. SIMD i MISD načini procesiranja su veoma usko povezani zato što rade u istom radnom taktu, ali MIMD sustavi mogu biti usko ili slabo povezani ovisno o problemu koji rješavaju.

2.3.3. Paralelizam

U distribuiranim sustavima, paralelizam je mjera koja prikazuje omjer vremena kroz koji procesori izvode instrukcije i vremena kroz koji čekaju da završe komunikacijske operacije. Što je bolje izgrađena logika i algoritam za komunikaciju sustava, to će procesori manje vremena čekati i više raditi nešto produktivno. Paralelizam ima puno veću važnost u paralelnim sustavima, nego u sustavu gdje brzina nije toliko važna. (Ajay D.Kshenaklyani, 2008)

2.3.4. Konkurencija

Konkurencija se često spaja u definiciju paralelizma, pogotovo u kontekstu distribuiranih sustava. Konkurencija opisuje omjer operacija obavljenih na jednom računalu sustava u odnosu na cijeli sustav. Ako npr. jedno računalo u sustavu koji se sastoji od deset

računala obavlja 50% posla, onda to računalo ima visoki stupanj konkurencije u sustavu. Ako sva računala u tom sustavu imaju iste specifikacije što se tiče procesorske moći, jedno računalo ne bi smjelo 50% ukupnog posla sustava rješavati samo, nego bi se taj posao trebao ravnomjerno raspodijeliti (u ovom primjeru, svakome računalu 10% posla) za optimalan rad. (Ajay D.Kshenaklyani, 2008)

2.3.5. Granularnost

Granularnost predstavlja omjer računanja i komunikacije u distribuiranom sustavu. Ako sustav ima nizak stupanj granularnosti, tada procesori u sustavu imaju relativno puno više izvršenih produktivnih instrukcija, nego pokušaja međusobne komunikacije i sinkronizacije. (Andrew S. Tanenbaum, 2016)

2.4. Komunikacija u distribuiranim sustavima

Sustavi dijeljene memorije su sustavi koji sadrže zajednički dijeljeni adresni prostor. U distribuiranim sustavima tu zajedničku memoriju koriste procesori za međusobnu komunikaciju uz pomoć podatkovnih varijabli, a za sinkronizaciju koriste kontrolne varijable. Komunikacija se postiže uz pomoć funkcionalnosti koje dolaze uz dijeljenu memoriju, npr. *lock* ili *monitor* funkcije. Svi višeračunalni sustavi koji nemaju nekakav tip zajedničkog adresnog prostora najvjerojatnije komuniciraju uz pomoć slanja poruka. Puno je lakše programirati sustav koji radi komunikaciju preko dijeljene memorije, nego sustav koji koristi slanje poruka. Iz tog je razloga moguće u distribuiranom sustavu simulirati zajednički adresirani prostor, to se zove distribuirana zajednička memorija. Implementacija takve simulacije dolazi sa svojim nedostacima, ali olakšanje posla programerima je jako veliki plus. Moguće je simulirati slanje poruka uz pomoć dijeljene memorije i obrnuto. (Ajay D.Kshenaklyani, 2008)

2.4.1. Sinkronizacija procesora

Sinkronizirani procesori su procesori koji izvršavaju naredbe u istom radnom taktu i koraku. Takvu sinkronizaciju nije moguće postići u distribuiranom sustavu, ali moguće je pobrinuti se da jedan procesor ne zaostaje iza drugoga u radu. Primjer bi bio komad kôda koji izvršavaju dva procesora, taj komad kôda se može gledati kao korak. Ako jedan procesor izvrši kôd prije drugog, tada on mora čekati drugi procesor da izvrši isti kôd. U sustavu s

puno više sinkroniziranih procesora, niti jedan procesor neće nastaviti izvršavati idući niz naredbi, ako nije dobio informaciju da su svi ostali procesori s njim u koraku. (Andrew S. Tanenbaum, 2016)

2.4.2. Izbor vođe

U distribuiranim sustavima većina algoritama koji se koriste u praksi nisu u potpunosti simetrični, te zato jedan proces mora inicijalizirati algoritam. Ako svaki proces sam inicijalizira algoritam za sebe, došlo bi do nepotrebnog troška resursa i mogućeg gubitka sinkronizacije. Potreban je jedan određeni proces koji svi ostali procesi zajednički prate, taj proces je vođa. Odabir vođe može biti obavljen na veoma jednostavan način, a to je uspoređivanje ID-a. Svaki proces ima svoj ID, te pri odabiru vođe mora obavijestiti ostale procese o svojem ID-u. ID je numerička vrijednost, to znači da nakon što svaki proces pokaže svoj ID, onaj s najvećom numeričkom vrijednošću ID-a je vođa, a svi koji su sudjelovali u odabiru prate njegove instrukcije. Odabir vođe se ponavlja vrlo često, tako je sustav zaštićen od kvara vođe koji bi inače bio konstantan. (Ajay D.Kshenaklyani, 2008)

3. Tehnologije na bazi distribuiranih sustava

Kroz godine hardver se razvijao nevjerojatnom brzinom:

- 1990. godine tipičan tvrdi disk imao je kapacitet od ~1400 MB, 2014. godine je 1 TB norma.
- RAM je u računalima 1993. godine imao 4 MB prostora, osobna računala imaju 16 GB u 2018. godini.
- Procesorska snaga računala 2018. godine je 100 milijardi puta veća nego 1956. godine.
- Od *dial up* modema se prešlo na optičke veze i gigabitne brzine interneta.

Koliko god bio brz i velik taj napredak, nije dovoljno brz da prati naglu potrebu za analiziranjem ogromnih količina podataka. Ljudi u zadnje vrijeme shvaćaju važnost analize podataka u poslovanju te vrijednosti koju te informacije donose. Zahvaljujući zamjeni papirologije za računala u poslovanju, informacije je lakše pohranjivati i obrađivati nego ikada. Te informacije u srednjim i većim firmama zauzimaju čak stotine TB prostora. Jedno računalo koje bi imalo mogućnost pohrane i obrade svih tih podataka bilo bi nepraktično i preskupo. Ako jedan konj nije dovoljno jak da vuče kola, prodati ga te kupiti većeg i skupljeg nema smisla, bolje je upregnuti još jednog konja da vuku kola zajedno. Po toj istoj logici, ima više smisla da dva ili više računala dijele te obavljaju jedan posao bolje i jeftinije, nego jedno računalo.

U računarstvu postoji široki spektar problema koji se rješavaju tehnologijama na bazi distribuiranih sustava. U radu su obrađene tehnologije koje rješavaju iduće probleme:

- Problem pohrane, dohvaćanja i izvođenja upita nad ogromnim količinama podataka rješava se uz pomoć Hadoop-a.
- Problem brzog procesiranja velikih količina podataka, najčešće u svrhu Big Data analitike, rješava Spark.
- Problem analitike u stvarnom vremenu koja mora dati brze rezultate analiziranjem podataka koji dolaze u velikim brzinama i navalama rješava Storm.

Sve navedene tehnologije koriste grupe međusobno umreženih računala kojima što ravnomjernije podijele podatke za obradu, odnosno paralelne distribuirane sustave. Podjela

i uređivanje podataka za procesiranje u paralelnom sustavu nije lagan zadatak, ali postoje tehnike i prakse, jedna od njih je *MapReduce*.

3.1. *MapReduce*

MapReduce je programski model namijenjen za procesiranje podataka uz pomoć klastera računala. Koristi se za velike skupove podataka koje je potrebno podijeliti, organizirati i zapakirati u format koji je primjereniji za raspodjelu zadataka paralelnom sustavu. (White, 2012)

MapReduce procesiranje podataka dijeli na dvije faze, *map* i *reduce*. Obje faze na ulazu i izlazu imaju podatke u obliku ključ-vrijednost (engl. *key-value)* parova. Kao primjer *MapReduce*-a poslužit će algoritam za praćenje trendova koji *Twitter* koristi u svojoj Apache Storm aplikaciji.

Prosječan *tweet* (objava) na *Twitter*-u sadrži riječi označene sa '#' znakom koji se zove *hashtag*. *Hashtag* se koristi pri referenciranju objave na određenu temu iliti trend. *Twitter* koristi *MapReduce* za jednostavno i učinkovito praćenje najpopularnijih trendova u određenom vremenskom periodu. *Map* funkcija pronalazi sve *hashtag*-ove u *tweet*-u te ih spremi u listu. Nakon *tweet*-a koji npr. glasi ovako: „Već je 10 navečer i još radim #pospan #umoran“, *map* funkcija će sve ključne riječi označene s *hashtag*-om spremiti u mapu, a mapa će izgledati ovako:

("pospan", "Već je 10 navečer i još radim #pospan #umoran")

("umoran", "Već je 10 navečer i još radim #pospan #umoran")

Nakon još 2 slična *tweet*-a lista izgleda ovako:

("pospan", "Već je 10 navečer i još radim #pospan #umoran")

("umoran", "Već je 10 navečer i još radim #pospan #umoran")

("umoran', "Jutarnja smjena... #umoran #pospan #ponedjeljak")

("pospan", "Jutarnja smjena... #umoran #pospan #ponedjeljak")

("ponedjeljak", "Jutarnja smjena... #umoran #pospan #ponedjeljak")

("umoran", "Gotov trening #umoran")

Tu *map* funkcija završava svoj posao, proizvedena mapa može se koristiti u puno različitih analitičkih procesa, ali proces u ovom primjeru je praćenje popularnih trendova. *Reduce* je zadužen za prepoznavanje redundancije u mapi te komprimiranje mape u ključ-vrijednost parove, koje u ovome primjeru mora biti jednostavno sortirati po popularnosti. Najpopularniji trend je onaj koji se najviše puta pojavio u na *twitter*-u, zato *reduce* funkcija sada na izlazu daje ključ-vrijednost parove gdje je ključ pojedini trend, a vrijednost broj ponavljanja tog trenda. Ranije proizvedena mapa će nakon *reduce* funkcije izgledati ovako:

("pospan":3)
("umoran":4)
("ponedjeljak":1)

Ovakvu informaciju je lako pohraniti jer na zauzima puno prostora i olakšano je izvršavanje upita za dohvaćanje iste, a to je ono što je najpotrebnije u radu s ogromnim količinama podataka.

3.2. Apache Hadoop

Ako je skup podataka toliko velik da se ne može pohraniti na jednom stroju zbog nedostatka prostora, tada je potrebno te podatke pohraniti na više odvojenih strojeva. Za takve pothvate se koriste posebni datotečni sustavi koji se zovu distribuirani datotečni sustavi. Hadoop ima komponentu pod nazivom *Hadoop Distributed File System* (HDFS) koja je ujedno i jezgra cijelog Hadoop sustava, te će biti jedan od fokusa ovoga rada zbog svojih funkcionalnosti koje se baziraju na paralelne distribuirane sustave. (White, 2012)

3.2.1. Povijest Hadoop-a

Hadoop je razvio Doug Cutting, kreator biblioteke poznate za pretraživanje teksta Apache Lucene. Početak Hadoopa bio je u Apache Nutch-u, *open source* web pretraživaču koji je također bio dio Lucerne-a. Nutch je pokrenut u produkciji 2002. godine, ali ubrzo se shvatilo da postojeća arhitektura neće skalirati dobro kada se uzmu u obzir milijarde stranica na web-u i velike datoteke koje njihov web *crawler* generira pri obradi istih stranica. Za te datoteke jednostavno nisu imali dovoljno prostora za pohranu. 2003. godine Google je izdao članak koji je detaljno opisivao arhitekturu Google-ovog distribuiranog datotečnog sustava, zvao se GFS. GFS je riješio problem premalenog prostora za pohranu podataka. Nakon toga se krenulo razvijanje *open source* implementacije *Nutch Distributed Filesystem*-a (NDFS).

Google je 2004. izdao članak o *MapReduce*-u, te su ubrzo nakon toga izašle *MapReduce* implementacije Nutch-a i NDFS-a. Pošto je ta tehnologija imala puno potencijala i izvan Lucerne projekta, predstavljena je kao neovisni projekt pod imenom Hadoop. Odlaskom Douga Cuttinga u Yahoo!, Hadoop je imao sredstva i ljude koji su ga kroz godine razvili u tehnologiju koja je dobila odličnu bazu korisnika. Neki od korisnika su i velike organizacije poput Facebook-a i New York Times-a. NDFS je kasnije preimenovan u HDFS. (White, 2012)

3.2.2. Hadoop Distributed File System

HDFS je datotečni sustav dizajniran s idejom da će pohranjivati veoma velike datoteke uz zadržavanje brzog i lakog protoka podataka, koristeći klaster zamjenjivog hardvera.

Datoteke koje HDFS pohranjuje mogu biti velike od stotine MB do stotine TB. Pohranjivanje TB podataka ne znači ništa, ako brzi pristup tim podacima nije omogućen. Način na koji je osmišljena obrada podataka u HDFS-u je taj da se pri razvijanju sustava imalo na umu da će se iz skupa podataka koji sustav pohranjuje puno češće čitati, nego u njega nešto pisati. Skup podataka nad kojim se rade poslovni procesi nastaje obradom sirovih podataka kroz vršenje raznih analiza. Te analize mogu uključivati cijeli skup podataka, koliko god on bio velik. Zbog takvih se slučajeva brzina dohvaćanja velikih komada podataka u kratkom roku vrednuje mnogo više, nego brzina odaziva sustava.

Hadoop-u nisu potrebni specijalni i skupi hardver da radi, dizajniran je da radi na jeftinijem i pristupačnijem hardveru. Takvim dizajnom je smanjena cijena implementacije sustava koji si je rijetko tko mogao priuštiti. Pošto je poznato da je hardver jeftin i potencijalno nepouzdan, sustav je napravljen tako da u slučaju kvara određenog komada hardvera nastavi raditi bez ikakvih zaostataka. (White, 2012)

3.2.2.1 Blokovi

Svaki disk ima definiranu veličinu bloka. Veličina bloka definira minimalnu količinu podataka koju je moguće čitati ili pisati u jednoj iteraciji. Datotečni sustavi podatke gledaju kao velike blokove koji se sastoje od puno više blokova na disku npr. blok datotečnog sustava je velik oko desetak kB, dok je blok na tvrdom disku velik 512 B. HDFS kao i svi drugi datotečni sustavi ima svoju zadanu veličinu bloka, a ona je 64 MB. U klasičnim datotečnim sustavima podatak koji je premalen da napuni cijeli blok zauzima ostatak njegovog prostora, HDFS tako ne radi. Kada tvrdi disk dobije naredbu da pročita određene

podatke, najviše vremena ne potroši na čitanje podataka, nego na potragu za podatkom, odnosno na čekanje da se čitač postavi na pravu lokaciju na disku. Ako je tražena datoteka na disku napisana u 10000 blokova, tada će čitač trebati 10000 puta tražiti početak svakog od tih blokova da pročita sve podatke. Ako su blokovi puno veći, tada će ta datoteka biti zapisana u samo jednom bloku te će čitač manje vremena potrošiti na sporo traženje podatka i više na brzo čitanje. Zato je veličina bloka u HDFS-u tako velika, jer se tim pristupom omogućuje brzo dohvaćanje ogromnih datoteka. Zahvaljujući HDFS-ovoj apstrakciji blokova nad tvrdim diskovima moguće je jednu datoteku, koja je veća od kapaciteta jednog diska, pohraniti na više diskova. Ta apstrakcija također pojednostavljuje sustav pohranjivanja podataka, a bilo kakvo pojednostavljivanje u distribuiranim sustavima je dobrodošlo zbog njihovih čestih komplikacija i teško razumljivih problema. Zato što je pohrana toliko jednostavnija za koristiti, puno je lakše podatke replicirati u slučaju da dođe do kvara na određenom disku, tako se sprječava gubitak podataka. (White, 2012)

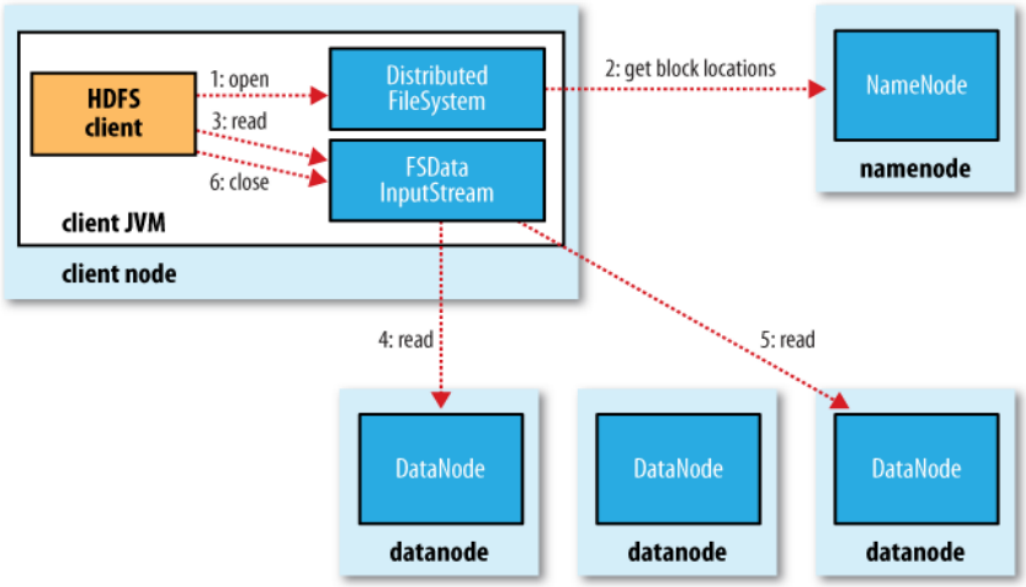
3.2.2.2 Članovi HDFS klastera

Članovi HDFS klastera dijele se na dva tipa:

- Imenski čvor *(engl. Namenode)* – vođa (engl. *master)*
- Podatkovni čvor *(engl. Datanode)* – radnik (engl. *worker*)

Namenode je zadužen za upravljanje imenskog prostora datotečnog sustava, održavanje hijerarhije datotečnog sustava te pohranjivanje metapodataka o svim datotekama u sustavu. *Namenode* također "zna" u kojem *datanode*-u i u kojem bloku se može naći koji podatak, ali te informacije ne pohranjuje lokalno, nego svaki puta pri pokretanju sustava regenerira nove. *Datanode*-ovi su "radnici" kojima *namenode* "diktira" posao. Na naredbu dohvaćaju podatke te ih pohranjuju u blokove i izvještavaju *namenode* o lokacijama blokova.

Slika 4 daje prikaz operacije čitanja iz HDFS-a korak po korak. Klijent želi otvoriti datoteku, te ju zatraži od datotečnog sustava. Datotečni sustav lokaciju blokova na kojoj se nalazi tražena datoteka zatraži od *namenode*-a. Koristeći podatke o lokacijama blokova, datotečni sustav klijentu generira ulazni tok (engl. *input stream).* Klijent tada preko ulaznog toka zatraži od svakog *datanode*-a koji ima blok na kojem je pohranjena datoteka da mu pošalje sadržaje blokova. Kada je zadnji blok pročitan i dostavljen klijentu, ulazni tok se zatvara.



Slika 4. Primjer toka operacija pri čitanju iz HDFS (White, 2012)

3.2.2.3 Sigurnosne mjere

Pošto *namenode* sve informacije o sustavu pohranjuje lokalno, u slučaju njegova kvara sustav gubi sve podatke, zato što *namenode* jedini zna kako rekonstruirati datoteke iz blokova u *datanode*-ovima. Zato *namenode* mora raditi na kvalitetnijem i pouzdanijem hardveru od ostatka sustava, ali opasnost i dalje postoji.

Kako bi se izbjegla ta kritična slabost, u sustavu je implementirana jedna od iduće dvije preventivne mjere. Prva je izrada sigurnosne kopije metapodataka o datotečnom sustavu. To se postiže tako da *namenode* sinkronizirano piše metapodatke na svoj lokalni disk i na neku drugu lokaciju koja neće biti ugrožena u slučaju kvara *namenode*-a. Drugi način je kreiranje sekundarnog *namenode*-a koji će držati iste podatke i stanje kao i primarni *namenode*, ali neće obavljati njegove funkcije. Sekundarni *namenode* zahtijevao bi svoj vlastiti uređaj i kopirao bi periodički stanje primarnog *namenode*-a. Sekundarni *namenode* kopira podatke s primarnoga, to znači da postoji kratki period, između dolaska novog podatka na primarni i kopiranja istog podatka na sekundarni *namenode*, u kojemu sekundarni nema sve podatke primarnog. U slučaju kvara primarnog *namenode*-a tijekom tog nezgodnog trenutka, gubitak podataka bio bi siguran. Iz tog razloga se u praksi najčešće koristi metoda koja uključuje izrade sigurnosne kopije metapodataka.

Namenode drži reference prema blokovima i datotekama u sustavu u trenutnoj memoriji, radi bržeg odaziva. Ako *namenode* održava jako veliki klaster s velikim brojem datoteka,

može se desiti da tome računalu nestane memorije. U tom slučaju Hadoop ima rješenje koje se zove *HDFS Federation*. *HDFS Federation* radi na principu dodavanja više *namenode*-ova u sustav. Svaki *namenode* održava dio imenskog prostora datotečnog sustava.

Ako u se u sustavu koji koristi *HDFS Federation* nalazi jako puno *namenode*-ova, to je vjerojatno zato što sustav u sebi ima pohranjeni veliki broj manjih datoteka, a za svaku tu datoteku *namenode* mora u memoriji imati referencu.

Implementacijom ranije spomenutih rješenja Hadoop se štiti od gubitka podataka u slučaju kvara, ali se ne štiti od gubitka dostupnosti podataka. Ako *namenode* prestane raditi, sustav ne može obavljati svoje zadaće, dok se novi *namenode* na pokrene. Vrijeme koje je potrebno da se novi *namenode* pokrene ovisilo bi o količini podataka za koje je pokvareni *namenode* bio "zadužen". Što više podataka je *namenode* imao "pod sobom", to više metapodataka treba prebaciti iz sigurnosne kopije u novi *namenode*, te više izvješća o blokovima treba primiti novi *namenode* od *datanode*-ova. Zbog svega toga bi sustav predugo ostao nedostupan.

Rješenje za taj problem Hadoop je nazvao *HDFS high-availability* (HA). HA funkcionira tako da *namenode*-ovi dolaze u parovima. Primarni obavlja svoje zadatke normalno, dok sekundarni čeka u stanju pripravnosti. Ako primarni *namenode* padne, sekundarni preuzima njegove obaveze bez vidljivih zastoja u radu sustava.

Da bi takvo rješenje proradilo, potrebne su dodatne promjene u sustavu. Jedna od promjena bila bi dodavanje dijeljene memorije između *namenode*-ova. Također bi trebalo podesiti *datanode*-ove da izvješća o blokovima šalju na oba *namenode*-a. (White, 2012)

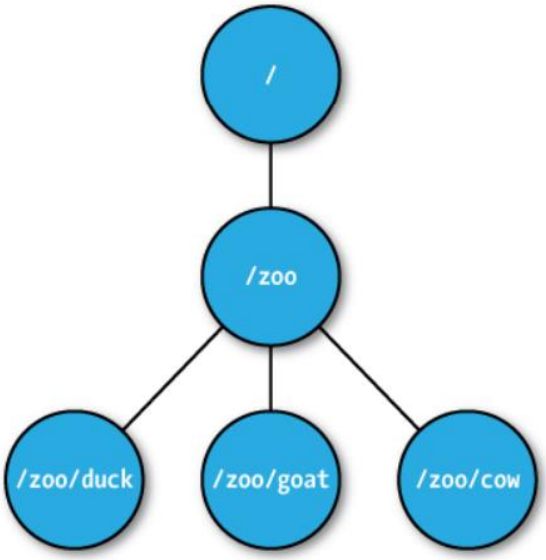
3.2.3. Zaključak za Apache Hadoop

Hadoop je nastao zbog problema uzrokovanih sa strane sporih čitanja i pisanja tvrdih diskova, te taj problem odlično rješava. Sa sve jeftinijim cijenama i većim prostorom za pohranu na SSD (engl. *Solid State Drive*) memorijama, hoće li Hadoop postati nepotreban? Odgovor je ne, Hadoop može poboljšavati brzinu rada u sustavu koji za pohranu koristi SSD memoriju. Zato što je HDFS ključan dio modernih distribuiranih rješenja poput Spark-a, Hadoop može samo rasti u popularnosti.

3.3. Apache Zookeeper

Zookeeper je komponenta koja ima ključnu ulogu u svim spomenutim tehnologijama za distribuirano procesiranje u ovome radu. Zookeeper je dijeljeni datotečni sustav koji članovi distribuiranog sustava koriste za pristupanje dijeljenoj memoriji. Zookeeper je veoma jednostavna biblioteka koja ima velike mogućnosti, ako se njeni jednostavni alati i operacije pravilno koriste. Napravljen je za implementiranje u klaster računala i doprinosi distribuiranom sustavu visoku dostupnost podacima, što rezultira razvojem pouzdanijih aplikacija. Zookeeper puno bolje radi u sustavima koji kroz njega više čitaju, nego upisuju nove podatke.

Zookeeper je datotečni sustav koji nema mape i direktorije, nego čvorove koji se nazivaju *znode-ovi*. Svaki *znode* može biti mjesto za pohranu podataka, ali isto tako može biti i kontejner za druge *znode*-ove. *Znode* je moćan alat kojim je moguće izgraditi svakakve hijerarhije u sustavu. Primjer jedne takve hijerarhije prikazan je u slici 5.



Slika 5. Primjer *znode* hijerarhije (White, 2012)

Zookeeper se primarno koristi za koordinaciju članova nekakvog distribuiranog sustava, a za koordinaciju nema potrebe za zapisivanjem velikih količina podataka u *znode*. Dovoljan je zapis malog podatka koji se ponaša kao zastava (engl. *flag*), uz par dodatnih informacija. Iz tog razloga je maksimalni prostor za pohranu podataka na *znode*-u 1 MB. Nije moguće djelomično čitanje pohranjenih podataka u *znode*-u, korisnik će pročitati ili sve podatke ili ništa. Ako *znode* ima podatak pohranjen u sebi, bilo kakvim pisanjem na *znode* taj stari

podatak se gubi. U *znode* također nije moguće djelomično upisati podatak. *Znode* ima 2 tipa: kratkoročni i dugoročni. Kratkoročni (*ephemeral*) *znode* traje samo onoliko koliko traje sesija klijenta, kada sesija istekne *znode* se briše. Kratkoročni *znode* ne može imati djece. Dugoročni *znode* nije vezan na sesiju klijenta, te se briše samo na izvršavanje izravne naredbe. *Znode* također može biti sekvencijalan. Sekvencijalni *znode* ima na kraju imena dodan redni broj. Ako klijent želi kreirati *znode,* a već postoji jedan sekvencijalni istog imena, tada će novi *znode* poprimiti isto ime, ali će mu biti pridodan novi redni broj, tako da se razlikuju. (White, 2012)

Zookeeper služi za koordinaciju, to se obavlja kroz *znode*-ove. Ako više klijenata koriste određene *znode*-ove za koordinaciju, tada bi na promjene u njima trebali biti obaviješteni. Zookeeper ima način da obavijesti klijenta na promjene *u određenom znode*-u i to radi uz pomoć *watch-a.* Klijent može postaviti *watch* na *znode,* ako želi biti obaviješten o operaciji koja se na njemu zadnja izvršila. *Watch* se okida samo jednom te se mora ponovo postaviti na *znode,* ako klijent želi i dalje pratiti njegove promjene.

Zookeeper visoku dostupnost postiže kroz replikaciju, te može raditi dok god većina računala s kojima radi nisu deaktivirana. Skup računala iliti klaster na kojem je podignut Zookeeper zove se ansambl. Ako u ansamblu od 7 računala 3 prestanu raditi, Zookeeper će nesmetano nastaviti rad. Ako u ansamblu od 8 računala 4 prestanu raditi, tada Zookeeper prestaje raditi, zato što 4 nije većina. Zookeeper se brine da svaki podatak bude repliciran na većinu ansambla, na taj način će stanje svakog podatka biti sačuvano dokle god većina ansambla radi. Zookeeper koristi komunikacijski protokol pod imenom Zab. Zab pri zapisu novoga podatka obavlja replikaciju kroz 2 koraka, elekcija "vođe" i emitiranje.

U fazi elekcije "vođe", računala ansambla odabiru jednog člana i imenuju ga "vođom", a ostali postaju njegovi "sljedbenici". U ovoj fazi je cilj dovesti većinu "sljedbenika" u sinkronizirano stanje s "vođom". Većina "sljedbenika" se zove kvorum. Nakon te faze slijedi faza emitiranja podataka kroz kvorum. Podatak koji klijent želi zapisati prosljeđuje se "vođi", koji tada isti podatak proslijedi ostatku kvoruma. Kada svaki član kvoruma uspješno pohrani podatak, "vođa" potvrdi da se zapis uspješno obavio te obavijesti klijenta o tome. Ako zapis ne uspije na svim članovima kvoruma, tada "vođa" poništava zapis. Podatak mora uspješno biti pohranjen na svim članovima kvoruma, ako nije, zapis se poništava. U slučaju da "vođa" zakaže, pokreće se nova elekcija "vođe" i proces se nastavlja kao i inače (White, 2012).

Svi podaci pri zapisu moraju prvo biti zapisani na disk da bi "vođa" potvrdio uspješno ažuriranje. Taj zapis na disku služi samo kao sigurnosna kopija, jer podatak ostaje u trenutnoj

memoriji nakon zapisa. Podatak je moguće dohvatiti iz bilo kojeg računala, nema potrebe za komunikaciju preko "vođe" kvoruma. Pri dohvaćanju podataka iz Zookeeper-a, podatak se čita iz trenutne memorije, a ne s diska. Podatak se ne čita s diska zato što bi to bilo sporo, a u datotečnom sustavu u kojem se puno više čita, nego piše, to bi bilo nezgodno usko grlo. Podatak se iz memorije brže dohvati i zbog prirode Zookeeper-a koji je napravljen za koordinaciju, a ne za pohranu velikih količina podataka, malo prostora u memoriji ne predstavlja problem. *Znode* ima ograničenje od 1 MB prostora za podatke u sebi, na računalu koje ima 16 GB RAM-a (to je, možda, čak i ispod standarda 2019. godine) 1000 *Znode*-a ne predstavlja preveliki problem.

Zookeeper u Hodoop-u nadopunjuje nedostatke HDFS-a, pošto HDFS ima ulogu pohranjivanja te adresiranja ogromnih količina podataka, Zookeeper je tu da koordinira cijeli klaster na kojem Hadoop radi.

3.4. Apache Spark

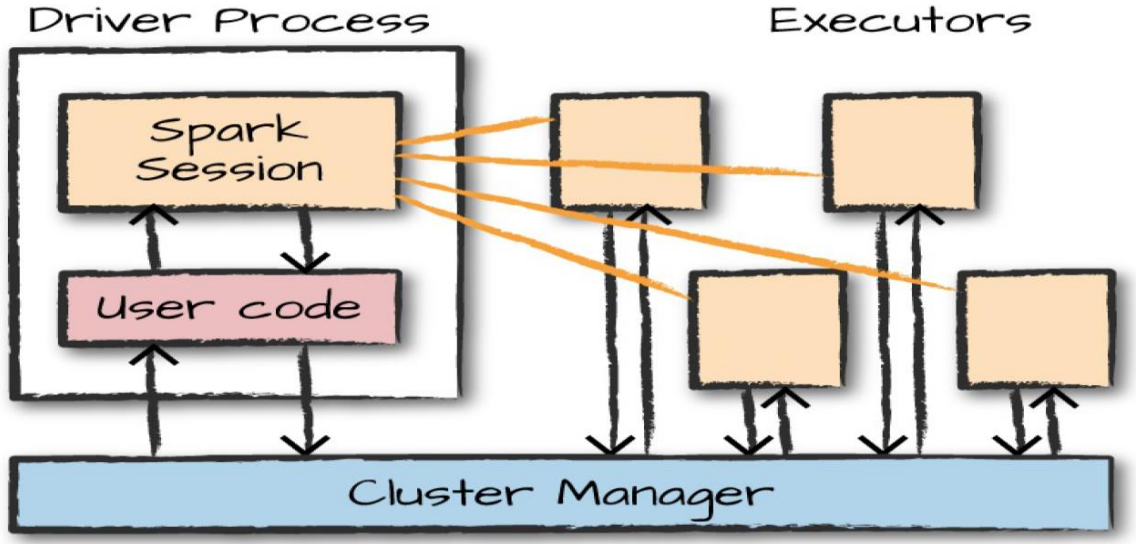
Spark je platforma za distribuirano procesiranje na klasteru računala koja se bazira na korištenju radne memorije. Koristi se za ETL, strojno učenje, analitiku i ostale operacije nad velikim količinama podataka u skupu (engl. *batch*) ili toku (engl. s*tream).*

Za razliku od Hadoop-a koji podatke koje obrađuje, drži i dohvaća sa stalne memorije (tvrdog diska), Spark radi s podacima u trenutnoj memoriji (RAM). To omogućuje puno brže dohvaćanje i obradu podataka, pošto je vrijeme potrebno da se pristupi podatku u stalnoj memoriji puno dulje od vremena potrebnog za pristup radnoj memoriji. Spark omogućuje veliku brzinu, lakoću korištenja, proširivost, interaktivnost i intuitivnost u analitičkim obradama. Koristi se primarno za razvoj aplikacija koje izvršavaju algoritme za višeslojne analitičke operacije, npr. strojno učenje ili napredne SQL upite. To uspijeva zbog apstrakcije koju koristi za obradu podataka na klasteru koristeći trenutnu memoriju, ta apstrakcija zove se *Resilient Distributed Dataset* (RDD). (Holden Karau, 2015)

3.4.1. Arhitektura distribuiranog procesa u Spark-u

Spark aplikacije sastoje se od jednog *Driver* procesa i više *Executor* procesa. *Driver* se pokreće na jednom članu klastera i zadužen je za održavanje informacija o aplikaciji, odgovaranje na korisničke naredbe i upravljanje rada *Executor*-a. *Driver* proces je srce Spark aplikacije. *Executor*-i su zaduženi za obavljanje posla koji im *Driver* proslijedi. Svaki

Executor ima samo dva zadatka: izvršavanje kôda koji mu *Driver* pridoda i izvještavanje *Driver*-a o rezultatima izvršenog kôda. Upravitelj klastera (engl. *Cluster manager)* prati cijeli sustav i njegove resurse. Sve ranije objašnjeno je jasnije prikazano na slici 6.



Slika 6. Arhitektura Spark aplikacije (Holden Karau, 2015)

3.4.2. Spark API

Spark ima dvije vrste API-a: strukturirani API (visoka razina) i nestrukturirani API (niska razina).

Strukturirani API se najčešće pojavljuje u obliku *DataFrame*-a. *DataFrame* predstavlja tablicu podataka koja ima retke i stupce. *Schema* definira stupce i njihove tipove. Ono što razlikuje *DataFrame* od drugih skupova podataka je način na koji se on prikazuje u memoriji. Normalna tablica podataka nalazi se u memoriji jednog računala, dok se *DataFrame* može nalaziti na više računala. Spark se koristi za velike količine podataka, *DataFrame*-u je zato osmišljen takav način rada. Ako je podataka previše za pohranu na jednom računalu ili ako njihova obrada predugo traje na jednom računalu, tada *DataFrame* te podatke idealno priprema za paralelnu i distribuiranu obradu.

DataFrame u pozadini koristi nestrukturirani API, a glavni predstavnik nestrukturiranog API-a je RDD.

3.4.3. Resilient Distributed Dataset

RDD je glavna apstrakcija podataka u Apache Spark-u. RDD je distribuirani skup zapisa raspodijeljen na jednu ili više particija. RDD je zaslužan za mogućnosti razvoja funkcionalnosti paralelnog procesiranja uz API koji je programeru lako koristiti.

Objašnjavajući svako slovo kratice RDD dolazi se do boljeg pogleda u njegova svojstva:

- Otporan (engl. *Resilient*) – ima mogućnost regeneriranja podataka izgubljenih zbog npr. kvara člana klastera.
- Distribuiran (engl. *Distributed*) – podaci su raspodijeljeni na više članova klastera.
- Skup podataka (engl. *Dataset*) – sadrži podatke koji se sastoje od skupova primitivnih vrijednosti.

Osim tih glavnih, RDD ima i iduća svojstva:

- *In-Memory* – podaci RDD-a drže se u trenutnoj memoriji koliko god je to moguće.
- *Immutable* – RDD nije moguće promijeniti jednom kada je stvoren, jedino rješenje za transformaciju je izrada novog RDD-a s preuzimanjem podataka starog.
- *Lazy evaluated* – podaci RDD-a nisu učitani dok se ne zatraži obavljanje neke operacije nad njima, tako se štedi memorija.
- *Cacheable* – moguće je sve podatke držati na dohvat ruke u trenutnoj memoriji, ako to nije moguće zbog veličine skupa podataka, onda se koristi i stalna memorija (sporo i izbjegava se).
- *Parallel* – podaci se obrađuju paralelno.

Spark je zadužen za obradu podataka, ne za pohranu. Spark koristi distribuirane sustave za pohranu podataka poput HDFS-a ili Cassandre. Podaci su u takvim sustavima rastavljeni i pohranjeni na različite particije. Particija je jedan dio određenog podatka. Spark podatke drži podijeljene na particije u trenutnim memorijama klastera, na isti način na koji su podijeljeni u distribuiranom datotečnom sustavu. Tako se dobiva preslika stanja podatka u radnoj memoriji i stanja pohranjenih podataka u stalnoj memoriji. (Bill Chambers, 2018)

Motivacija koja stoji iza kreiranja apstrakcije poput RDD-a su dvije aplikacije koje su 2018. godine popularnije nego ikada, ali nemaju bogati izbor kvalitetnih programskih okvira:

- iterativni algoritmi – npr. strojno učenje
- interaktivni alati za *data mining* –koriste se za dinamički kreirane upite na isti skup podataka

Željeni učinak bio bi kvalitetnije i temeljitije korištenje trenutne memorije i smanjivanje potrebe za kopiranjem velikih skupova podataka preko mreže. RDD za rad mora sadržavati sljedeće:

- Listu RDD-ova o kojima taj RDD ovisi.
- Polje particija po kojima je skup podataka podijeljen.
- Funkcija za obavljanje računalnih operacija nad particijama.
- *Partitioner* koji definira kako su ključevi heširani (engl. *hashed*) i parovi podijeljeni na particije (samo za ključ-vrijednost RDD-ove, Partitioner nije obavezan u RDD-u).
- Konfiguracija preferiranih lokacija za određene particije (nije obavezan).

RDD-ovi se ponašaju kao kontejneri instrukcija koji velike skupove podataka dijele na particije kojima Spark može upravljati. Te podatke podijeljene u particijama tada Spark paralelno obradi, te rezultate obrade zapiše odvojeno za svaku particiju, pojedine particije se obrađuju sekvencijalno.

RDD svoju pravu moć pokazuje u svojoj jednostavnosti i pristupačnosti programeru koji ga koristi. Napraviti apstrakciju iznad entiteta koji je podijeljen na više računala u isto vrijeme te tu apstrakciju predstaviti kao jedan objekt u programskom jeziku (npr. Java ili Python) je nešto što ima ogromnu korist programeru. Moguće je koristiti RDD objekte izravno u programiranju, ali to se koristi samo za posebne i kompleksnije probleme. Za veliku većinu potreba dovoljno je razumijevanje i korištenje *Spark Structured API*-a. *Spark Structured API* programeri koriste za razvoj programskih rješenja za BigData, ali RDD je ono što Spark koristi u pozadini tog API-a. U području kompleksnom poput BigData analitike, Spark daje jednostavno rješenje koje ne zahtijeva veliko znanje o distribuiranim sustavima koji rade u pozadini. (Bill Chambers, 2018)

3.4.4. Distribuirane dijeljene varijable

Distribuirane dijeljene varijable se dodaju pri programiranju funkcionalnosti koje imaju posebna svojstva kada se pokreću na klasteru računala. Postoje dvije vrste distribuiranih dijeljenih varijabli: akumulatori i emitirajuće varijable. Akumulatori služe za dohvaćanje rezultata obrade podataka sa svih klastera i sumiranje tih rezultata u jedan zajednički rezultat. Akumulator je varijabla na *Driver*-u na koju mogu utjecati *Executor*-i koji rade "pod njim". Najčešće se koristi kao nekakav brojač ili kao varijabla za sumiranje, zato Spark ima podršku

za numeričke tipove akumulator varijable, ali po želji moguće je implementirati nove i kompleksnije tipove. Emitirajuće varijable omogućuju održavanje većih vrijednosti na pojedinim članovima klastera u svrhu korištenja tih vrijednosti u radu. Glavna svrha im je izbjegavanje konstantne serijalizacije i deserijalizacije pri prosljeđivanju podataka kroz klaster, to uspijeva spremanjem podataka na sve članove klastera. (Holden Karau, 2015)

3.4.5. Zaključak za Apache Spark

Spark je nadogradnja na Hadoop. Glavna promjena u Hadoop-u sa Spark-om je ta da Spark zamjenjuje Hadoop-ov MapReduce i 20 puta je brži. Spark je najbrže rastuća platforma za rad s ogromnim količinama podataka, bilo to strojno učenje ili *BigData.* Također ima jednostavna rješenja za rad uz ostale Apache distribuirane platforme, tako da ga nije teško implementirati u sustave koji su dulje vrijeme u produkciji.

3.5. Apache Storm

Storm je distribuiran i pouzdan sustav za *stream processing*. *Stream processing* je čin stalnog inkorporiranja novih podataka u svrhu računanja i davanja rezultata. Dolazeći podaci nemaju definirani početak i kraj, nego dolaze u seriji događaja koji nemaju konstantnu frekvenciju. Ti podaci na dolasku u aplikaciju prolaze kroz više koraka obrade koja može dati i više od jednog rezultata iz računanja jedne vrste podataka. *Batch processing* je obrada nad podacima koji imaju definiranu količinu, početak i kraj, ta obrada daje rezultat samo jednom. *Stream i batch processing* u praksi često rade skupa. *Stream processing* aplikacije često nadopunjuju ili rade pridruživanja (engl. *join)* na postojeći skup podataka koji je nastao periodičkim izvršavanjem *batch processing* zadatka. Na drugoj strani, podaci na kraju *stream processinga* skoro uvijek budu pohranjeni u neke tablice nad kojima *batch processing,* eventualno, vrši upite. (Holden Karau, 2015)

Aplikacije koje u stvarnom vremenu paralelno procesiraju podatke na klasteru računala programeru nisu ni malo jednostavne razvijati, zato je Storm razvoj takvih aplikacija učinio što jednostavnijim. Storm je najlakše koristiti u Java programskom jeziku, ali on ima i podršku za veliki broj drugih jezika, dok god oni implementiraju određene Storm biblioteke. Također ima ugrađenu funkcionalnost koja u slučaju kvara jednog računala u klasteru, sve njegove zadatke automatski prenese na drugo. U slučaju da korisnik želi dodati novo računalo u klaster, nije potrebno prolaziti kroz svakakve mjere opreza, nego se računalo

samo doda u konfiguracijama i Storm će mu dodjeljivati zadatke čim prije moguće, toliko je skalabilan. Storm je napravljen za što veću brzinu u procesiranju podataka, zato za komunikaciju u mreži koristi *zeromq,* biblioteku koja slanje podataka preko mreže prema klasteru obavlja brže nego TCP. U novijim verzijama Storm-a, umjesto *zeromq*-a koristi se *Netty.* Sva navedena svojstva su tu da olakšaju razumijevanje i održavanje aplikacija za procesiranje u stvarnom vremenu programerima, koji uz sve zadatke koje svakodnevno rješavaju žele nešto što je jednostavno i brzo kao rješenje njihovog problema (Jonathan Leibiusky, 2012).

3.5.1. Storm klaster

U Storm klasteru, članovi klastera organizirani su prema glavnom članu koji se zove *Master*. *Master* u sebi ima pokrenut *Nimbus,* proces koji je zadužen za raspodjelu kôda, dodjelu zadataka i rješavanje kvarova u klasteru. *Master* "pod sobom" ima i članove klastera tipa *Worker*. *Worker* je zadužen za izvršavanje kôda i zadataka koje mu *Master* dodijeli. Proces koji izvršava dio Storm aplikacije na *Worker-u* zove se *Supervisor*. *Nimbus* i *Supervisor*-i međusobno se koordiniraju uz pomoć Zookeeper-a.

3.5.2. Komponente Storm topologije

Storm topologija je apstrakcija koja predstavlja svaki dio Storm aplikacije spojen u jednu funkcionalnu cjelinu. Glavni dijelovi Storm aplikacije zovu su *Spout* i *Bolt*. *Spout* je ulazna točka Storm aplikacije. *Spout* dohvaća sirove podatke i prosljeđuje ih dalje kroz aplikaciju. Moguće je *Spout* podesiti tako da prosljeđuje podatke prema različitim komponentama aplikacije ovisno o njihovom sadržaju, ali zbog bolje prakse to je bolje prepustiti *Bolt*-u. *Bolt* je komponenta Storm topologije zadužena za obradu podataka. *Bolt* se može koristiti za određene agregacije nad podacima, za usmjeravanje podataka na točno određeni *Bolt*, za pohranu podataka ili slanje podataka nekom vanjskom sustavu. Podaci se pri slanju iz jednog Bolt-a u drugi, spremaju u entitet koji se zove *Tuple*. Svaki *Tuple* se pri slanju doda u ranije deklarirano polje (engl. *Field*).

Svaki *Spout i Bolt* povezuju se prema jednom od više pravila grupiranja toka podataka (engl. *Stream Grouping*). Nasumično grupiranje (engl. *Shuffle Grouping*) povezuje *Spout* ili *Bolt* s idućom instancom *Bolt*-a nasumično, tj. izlazni podatak se šalje na nasumično odabranu instancu idućeg *Bolt*-a. Grupiranje po polju (engl. *Fields Grouping)* koristi se kada je potrebno izlazni podatak poslati na određenu instancu idućeg *Bolt*-a u toku, ovisno o

sadržaju određenog izlaznog polja. Pri slanju izlaznih podataka koji su u obliku *Tuple*-a pohranjenih u poljima, definira se jedno polje po kojemu će Storm odrediti gdje proslijediti sadržaje svih tih polja. Opće grupiranje (engl. *All Grouping)* služi za slanje izlaznog podatka na sve instance idućeg *Bolt*-a u toku. Slanje podataka za obradu na taj način ne bi imalo smisla, zato se to grupiranje koristi za slanje određenih signala koji mogu izazvati posebno ponašanje svakog *Bolt*-a koji primi signal. (Jonathan Leibiusky, 2012)

Ovo je bio pogled na Apache Storm s visoke razine, detaljnije informacije nalaze se u idućem poglavlju, gdje je dan bolji pogled na Storm i njegove funkcionalnosti kroz praktičan primjer.

4. Aplikacija za prepoznavanje meteoroloških nepogoda u stvarnom vremenu

Aplikacija za prepoznavanje meteoroloških nepogoda u stvarnom vremenu koristi Apache Storm tehnologije i napisana je u programskom jeziku Java. Aplikacija kao ulazne vrijednosti prima: podatke o meteorološkim vrijednostima koji mogu izravno i negativno utjecati na sigurnost stanovništva, podatke o državi i gradu pojedinog mjerenja. Aplikacija daje mape država ili gradova sortirane po količini vremenskih nepogoda u određenom vremenskom periodu. Mape dolaze u obliku sumiranih rezultata svih vrsta vremenskih nepogoda, i u obliku rezultata za jednu određenu vrstu nepogode (npr. obilna kiša ili preniska temperatura).

4.1. Tehnologije za razvoj softvera

U ovome poglavlju bit će nabrojane sve tehnologije koje su se koristile pri izradi ovoga završnog rada tj. sve programske jezike, alate i drugo, te će također u nastavku biti detaljnije objašnjene.

Programski jezik na kojemu se temelji cijeli završni rad je Java. Programiranje u Javi radit će se u Eclipse okruženju. *Build* i menadžment ovisnosti (engl. *Dependency management)* riješen je uz pomoć *Maven*-a. Aplikacija je napravljena za pokretanje na Storm *nimbus*-u, te ju je zato potrebno razvijati uz Storm Core biblioteku. Storm *nimbus* radi na klasteru računala, ta računala su virtualne mašine koje pokreću Linux Ubuntu operacijski sustav uz pomoć Oracle VM VirtualBox platforme. Na tim računalima pokretat će se Storm aplikacija te će se testirati njene performanse u različitim postavkama i uvjetima.

4.1.1. Java

Java je najkorišteniji programski jezik na svijetu. Sintaksu je naslijedila od programskog jezika C, a svoja objektno orijentirana svojstva od C++-a. Javu su razvili programeri u Sun Microsystems u 1991. godini.

Java je na tržištu objavljena u doba početka interneta. Njena najsjajnija točka bila je razvijanje aplikacija za internet, a tu je uspijevala najviše zbog neovisnosti o platformi na

kojoj radi. Tu neovisnost o platformi Javi omogućuje *Java Virtual Machine* (JVM). Java kompajler ne proizvodi izvršni kôd (*.exe*), nego *bytecode*. *Bytecode* je set instrukcija napravljen da se izvršava isključivo u JVM-u, a JVM je zapravo interpreter *bytecode*-a. (Schildt, 2014)

Čemu dodavati još jedan sloj pri izvršavanju kôda, ne usporava li performanse programa cijeli proces prevođenja *bytecode*-a u JVM-u? Istina, taj dodatni sloj usporava performanse, ali *bytecode* je veoma dobro optimiziran te ga JVM procesira veoma brzo, tako da su te prepreke performansama gotovo i zanemarive. Ali i taj mali problem dobio je svoje rješenje u obliku tehnologije HotSpot. HotSpot Javi pridonosi *Just-In-Time* (JIT) kompajler za *bytecode*. JIT je dio JVM-a koji pri izvođenju programa određene dijelove *bytecode*-a pretvara u izvršni kôd, te tako ubrzava samo izvođenje. Da bi Java radila na računalu, potrebno je implementirati JVM. Kada bi se Java izvršavala bez JVM-a, tada bi program trebao biti kompatibilan s procesorom ili ga jednostavno procesor ne bi znao izvršiti. JVM je tu da se pobrine da bilo koji procesor može pokrenuti Java program. (Schildt, 2014)

Programska rješenja za internet Java je imala u obliku appleta i servleta. Applet je Java program napravljen tako da se može prenositi putem interneta te izvršiti u web pregledniku. Klikom na link koji bi na drugoj strani imao applet, applet bi se preuzeo s interneta te automatski pokrenuo i prikazao u web pregledniku. Takav način izvršavanja programa bio je lak je za korištenje, pogotovo u to doba kada je rijetko tko znao upravljati osobnim računalom. U isto vrijeme program koji se izvršava samo na klik miša ne zvuči najsigurnije, jer taj program može biti štetan za računalo. JVM ograničava procese koji se odvijaju tijekom izvršavanja Java programa, tako da dopušta programu da se izvršava samo u svojoj izvršnoj okolini te mu ne dopušta pristup u druge dijelove računala. Zato su appleti i drugi Java programi bili sigurni za korištenje. Applete je zamijenio JavaScript.

Applet se izvršava na klijentu, servlet se izvršava na serveru. Servlet stvara dinamički generirane podatke te ih servira klijentu. Klijent od servleta samo mora zatražiti informaciju, a servlet će klijentu poslati generiranu web stranicu koja sadrži sve što je klijent tražio. Servleti su zbog JVM-a bili kompatibilni s mnogim okolinama, te su zato bili veoma zastupljeni na tržištu. (Schildt, 2014)

4.1.2. Eclipse

Eclipse je integrirano razvojno okruženje (engl. *Integrated Development Environment)* (IDE) za programiranje, jedan od najpopularnijih za programiranje u Javi. Može se koristiti za programiranje u bilo kojem jeziku. Eclipse je nastao kao zamjena za Visual Age for Java od IBM-a, ali na kraju je završio kao *open source*. Njime trenutačno upravlja neovisna i neprofitna organizacija pod nazivom Eclipse Foundation. Od trenutka kada je postao *open source* softver 2001. godine, Eclipse je dobio milijune korisnika.

Da bi se Eclipse mogao koristiti za rad u Javi, potrebno je s interneta preuzeti Java development kit (JDK).

Pri inicijalnom pokretanju Eclipse-a on pita korisnika gdje želi izraditi svoj *workspace*, što je *workspace*? *Workspace* je lokacija u kojoj se nalazi sav kôd, datoteke i postavke napravljene kroz izradu softvera. Zbog tog načina stvaranja radnog okruženja moguće bez puno napora "skakati" s npr. jednog *workspace*-a u kojemu se radi Java server na drugi *workspace* u kojemu se rade statističke analize i izvještaji u Python-u.

Glavni prozor u Eclipse-u zove se *workbench* te sadrži sve najosnovnije komponente koje su potrebne funkcionalnom IDE-u, a to su:

- Pogledi
- *Editor*
- Perspektive
- Izborna traka
- Alatna traka

Pogledi su prozori koji pokazuju informacije o trenutnom projektu, npr. prikaz paketa te njihovu hijerarhiju. Neizbježni pogledi su preglednik paketa, problemi, konzola i *Javadoc*. U pregledniku paketa moguće je vidjeti hijerarhiju paketa projekta. Ikone na paketima mijenjaju izgled u raznim situacijama npr. kada se u jednoj njegovoj datoteci nalazi greška, kada taj dio projekta nije *upload-an* na repozitorij kôda, itd. Na taj način Eclipse lako daje do znanja u kakvom je stanju projekt i gdje bi bilo dobro usmjeriti programerovu pažnju. Pogled problema na jednom mjestu prikazuje sve probleme u projektu i sadrži poveznice na iste probleme. Duplim klikom se dolazi do izvora svakoga problema koji Eclipse može uočiti. Konzola daje ključne informacije o izvođenju aplikacija u stvarnom vremenu. *Javadoc* sadrži dokumentaciju o klasama i metodama koje se koriste. Dokumentacija sadrži

opis funkcionalnosti klasa i metoda te pošteđuje programera od raznih istraživanja i potraga za tim informacijama.

Editor iliti urednik je prostor u kojemu se pišu i izmjenjuju datoteke. U slučaju ovoga rada te datoteke su najčešće Java klase. Editor u Eclipse-u ima ugrađeno znanje o specifičnim programskim jezicima. Java *editor* savršeno poznaje Java sintaksu te tijekom pisanja pomaže upozoravajući na pogreške, preporukama uz auto-ispunu varijabli, metoda ili klasa, čak i upozoravanjem na neiskorištene varijable, *import*-ove i nezatvorene *stream*-ove.

Perspektive se mijenjaju u odnosu na ono što se trenutačno radi u Eclipse-u. Pri korištenju editora za pisanje Java kôda koristi se Java perspektiva. Pri pokretanju aplikacije u *debug* načinu tada koristi se *debug* perspektiva. Kad se radi o pretraživanju repozitorija kôda tada se gleda SVN (Apache Subversion) ili Git perspektiva itd. Alatne trake sadrže prečace za često korištene akcije i operacije u Eclipse-u. Kao što su spremanje promjena, pokretanje aplikacije, *debug* aplikacije, itd. (Burnette, 2005)

4.1.3. Maven

Maven je alat za upravljanje projektima koji obuhvaća *project object model* (POM), skup standarda, životni tijek projekta, sustav za upravljanje ovisnostima te logiku za izvršavanje *plugin*-a u različitim dijelovima životnog tijeka projekta. Ovo je bila jedna od skromnijih definicija Maven-a, teško ga je opisati kada se uzme u obzir sve što on može napraviti. Maven kao alat ne postoji da riješi jedan konkretan problem, jer je kroz godine dobio ogroman broj funkcionalnosti i mogućnosti, od alata za generiranje izvršnih aplikacija iz izvornog kôda pa do olakšavanja komunikacije između članova tima.

Kada se radi o Maven projektu, tada se radi o projektu s veoma strogo definiranom strukturom i hijerarhijom. Projekt će se zvati „zavrsniRad“. Izvorni kôd se nalazi u "zavrsniRad/src/main/java", a njegovi resursi se nalaze u "zavrsniRad/src/main/resources". Testovi se nalaze u "zavrsniRad/src/test/java", a njihovi resursi u "zavrsniRad/src/test/resources". Konvencije i dobre prakse govore da "main" i "test" paketi sadrže istu hijerarhiju, npr. ako se REST API nalazi u "zavrsniRad/src/main/java/rest", tada će se testovi za REST API nalaziti u "zavrsniRad/src/test/java/rest". Maven će kompajlirati *bytecode* u "zavrsniRad/target/classes", a JAR će generirati u "zavrsniRad/target". Ako se programer nauči pratiti tu konvenciju, tada će Maven pri *build*-u najteži dio posla obaviti za njega. Po ovome se može zaključiti da Maven više poštuje konvenciju od konfiguracije i da

se korisnik mora prilagoditi njegovim pravilima. Iako to je najbolja i najsigurnija praksa, nije obavezno držati se konvencije, te je moguće konfigurirati stvari poput lokacije generiranja JAR-a, također je moguće korištenje raznih *plugin*-ova te uređivanje Maven-a po ukusu.

Maven sam po sebi može samo raditi s XML dokumentima i pratiti životni tijek aplikacije uz par *plugin*-ova. Maven je dizajniran da posao prebaci na skup Maven *plugin*-ova. On sam po sebi ne može obaviti određenu radnju, dok mu se ne pridruži *plugin* za obavljanje te radnje. *Plugin-i* se povlače iz Maven *repository*-a. Kada se u Maven projektu izvrši naredba `mvn install`, tada se svi najosnovniji i krucijalni *plugin*-ovi skinu s *Central Maven Repository*-a. Maven povlači *plugin*-ove i ovisnosti s repozitorija ovisno o sadržaju POM datoteke.

Upravljanje ovisnostima (engl. *dependency management*) se koristi lakše nego ikada. Sve što je potrebno da se povuku ovisnost je set atributa koji se unosi u POM. Ti atributi su identifikator grupe, identifikator artefakta te verzija ovisnosti. Svi *plugin*-ovi rade po logici definiranoj u POM-u.

Što je zapravo POM? POM je XML datoteka koja definira svaki Maven projekt. U POM-u je definirana struktura projekta, konfiguriran *build* proces te su opisane poveznice s drugim projektima.

POM "diktira" Maven-u o kakvom se projektu radi i kako se tim projektom upravlja. POM sadrži četiri kategorije opisivanja konfiguracije:

- Osnovne informacije o projektu (engl. *General project information*)
- Postavke *build*-a (engl. *Build settings*)
- Okruženje *build*-a (engl. *Build environment*)
- Odnosi POM-ova (engl. *POM relationships*)

General project information sadrži glavne informacije o projektu, a to su ime projekta, URL projekta, organizaciju koja razvija projekt i listu programera ili dionika te licence njihovih projekata.

Build settings je dio POM-a gdje se navodi kako će se Maven ponašati kod *build*-a. Ovdje je moguće mijenjati lokacije izvornog kôda i testova, dodavati nove *plugin*-e te definirati gdje će u životnom tijeku aplikacije koji *plugin* obavljati svoj zadatak.

Build environment se sastoji od profila koji se mogu aktivirati za korištenje u drugim okruženjima. Na primjer, možda je potrebno u različitim situacijama aplikaciju pokretati na različitim mjestima. Takve stvari se navode u ovom dijelu POM-a koji se često nadopunjava sa *settings.xml* datotekom, koja se u pravilu nalazi u "~/{USER}/.m2" mapi.

POM *relationships* najčešće sadrži informacije o POM-u *parent* projekta, s kojega je potrebno naslijediti određene postavke. Svaki POM svakog Maven projekta nasljeđuje Super POM. Super POM definira vrijednosti i atribute koji postoje u svakom Maven projektu. (Tim O'Brien, 2011)

4.2. Komponente i funkcionalnosti aplikacije

Ovo poglavlje pokriva razvoj Storm aplikacije u Java programskom jeziku. Koristeći IDE kreiran je Maven projekt pod imenom "storm".

4.2.1. Preuzimanje potrebnih ovisnosti uz Maven

U "pom.xml" potrebno je dodati iduću ovisnost (engl. *dependency*):

```
<dependencies>
   <dependency>
      <groupId>org.apache.storm</groupId>
      <artifactId>storm-core</artifactId>
      <version>1.2.2</version>
      <scope>compile</scope>
   </dependency>
</dependencies>
```

Kôd 1. Storm Core ovisnost u "pom.xml" datoteci

Nakon dodavanja te ovisnosti, potrebno je izvršiti `mvn install` naredbu da bi Maven preuzeo s repozitorija Storm Core biblioteku. Kada je Storm Core biblioteka preuzeta, moguće je krenuti programirati *Spout* i *Bolt*-ove.

4.2.2. Ulazni podaci

Storm aplikacija će dohvaćati podatke o državi, gradu, temperaturi zraka (°C), količini kiše (mm, 1 milimetar kiše je jednak 1 litri kiše po četvornom metru, koja je pala u periodu

od jednog sata), zagađenju zraka (PM2.5, količina čestica koja se zadržava u zraku na dulji period vremena) i brzini vjetra (km/h).

Te vrijednosti bit će odvojene zarezom i bit će dohvaćene iz datoteke, ovako izgleda dio datoteke:

Tanzania,Tukuyu,43,1.93,103,299

United States,Warren,22,9.96,11,196

Indonesia,Krajan Bejagung,35,6.15,10,322

France,Dijon,14,6.84,194,296

Russia,Privolzhskiy,-29,4.7,152,159

Vrijednosti su nasumično generirane, ali i dalje unutar granica koje su u stvarnosti moguće.

4.2.3. `InputReader` Spout

Spout je ulazna točka svake Storm topologije. U praksi, Storm se najčešće koristi uz Apache Kafku. Kafka je rješenje za primanje i organizaciju velikog broja ulaznih podataka. Kafka te podatke sprema te, na zahtjev klijenta, dostavlja. Taj klijent bi u ovom primjeru bila Storm aplikacija, točnije *Spout*. Zato što u ovome radu nema rješenja poput Kafke, *Spout* će simulirane meteorološke podatke dohvaćati iz datoteke. *Spout* se zove `InputReader` i radi se o Java klasi koja nasljeđuje `BaseRichSpout`. Koristit će se 3 metode iz `BaseRichSpout` klase, a to su `open()`, `nextTuple()`, i `declareOutputFields()`.

Metoda `open()` pokreće se pri inicijalizaciji *Spout* klase, te služi za pripremu resursa koje su potrebne *Spout*-u da normalno radi. U slučaju kôda 2, `open()` će pripremiti datoteku s podacima za čitanje i inicijalizirati kolektor (engl. *Collector*).

```
public class InputReader extends BaseRichSpout
{
    private String fileToRead;
    private SpoutOutputCollector collector;
    private FileReader reader;
    private BufferedReader buffReader;

    @Override
    public void open(Map map, TopologyContext
topologyContext, SpoutOutputCollector spoutOutputCollector)
```

```
    {
        this.fileToRead=map.get("fileToRead").toString();
        try{
            this.reader = new FileReader(fileToRead);
    }
        catch (FileNotFoundException e)
        {
            throw new RuntimeException("Error reading file -
"+map.get("fileToRead"));
        }
        this.collector=spoutOutputCollector;
        this.buffReader=new BufferedReader(reader);
    } ...
```

Kôd 2. Deklaracija `InputReader` *Spout*-a i `open()` metoda

`SpoutOutputCollector` prikazan u kôdu 2 je objekt zadužen za skupljanje izlaznog rezultata *Spout*-a i prosljeđivanje istog dalje prema toku topologije. Svaka klasa koja u imenu ima "Collector" u Storm-u ima tu istu zadaću. Parametar `map` u `open()` metodi sadrži konfiguraciju Storm topologije kojoj mogu pristupiti svaki *Spout* i *Bolt*. U ovom primjeru se pristupa konfiguraciji radi dohvaćanja putanje datoteke koja sadrži meteorološke podatke.

`nextTuple()` metoda zadužena je za dohvaćanje ulazne informacije, pretvaranje te informacije u format koji se sprema u *tuple* te šalje isti idućem *Bolt*-u. Metoda je prikazana u kôdu 3.

```
@Override
public void nextTuple()
{
        try{
            String str=buffReader.readLine();
            if(str!=null){
                this.collector.emit(new Values(str));
            } ...
```

Kôd 3. `nextTuple()` metoda

Pročitani redak datoteke se uz pomoć `SpoutOutputCollectora` emitira dalje kroz topologiju. `Values` objekt proslijeđen metodi `emit()` predstavlja *tuple*.

Metoda `declareOutputFields()` je mjesto gdje se deklariraju izlazne vrijednosti *Spout*-a. Primjer je prikazan u kôdu 4. Nije moguće na izlazu emitirati više vrijednosti preko *tuple*-a nego što je deklarirano.

```
@Override
public void declareOutputFields(OutputFieldsDeclarer
outputFieldsDeclarer)
{
```

```
        outputFieldsDeclarer.declare(new Fields("line"));
    }
```

Kôd 4. `declareOutputFields()` metoda

U kodu 4, metodi `declare()` dodaje se onoliko parametara u `Fields` konstruktor koliko je očekivano da će *Spout* emitirati vrijednosti preko *tuple*-a. Konstruktoru `Fields` klase se prosljeđuje ključ vrijednosti preko kojega se kasnije vrijednost može referencirati i dohvatiti u idućem *Bolt*-u.

Spout se koristi samo za dohvaćanje podataka i prosljeđivanje istih prema ostatku topologije, *Spout* ne bi trebao raditi obradu ili podjelu podataka.

4.2.4. `InputNormalizer` Bolt

Bolt je komponenta Storm topologije zadužena za obradu podataka. `InputNormalizer` *Bolt* je zadužen za podjelu retka datoteke poslanog iz `InputReader`-a u format nad kojim se mogu raditi usporedbe i slične analize. `InputNormalizer` je klasa koja nasljeđuje `BaseBasicBolt`. Koristi metode `execute()` i `declareOutputFields()`.

Metoda `execute()` dohvaća emitirani *tuple* koji dolazi od prethodne komponente u topologiji, u ovom slučaju to je `InputReader`, i podijeli sadržaj tog *tuple*-a u samostalne vrijednosti. Ranije objašnjena funkcionalnost je prikazana u kôdu 5.

```
public class InputNormalizer extends BaseBasicBolt
{
    @Override
    public void execute(Tuple tuple, BasicOutputCollector
basicOutputCollector)
    {
        String data= tuple.getString(0);
        String[] values=data.split(",");
        basicOutputCollector.emit(new Values(
                 values[0].trim()
                ,values[1].trim()
                ,Integer.parseInt(values[2].trim())
                ,Double.parseDouble(values[3].trim())
                ,Integer.parseInt(values[5].trim())
                ,Integer.parseInt(values[4].trim())
        ));
    }
```

Kôd 5. Deklaracija `InputNormalizer` *Bolt*-a i metoda `execute()`

Tuple u sebi može imati više vrijednosti, tim vrijednostima se može pristupiti po njihovim rednim brojevima ili po imenu koje im je dano pri deklaraciji izlaznih vrijednosti. U kôdu 5, vrijednost je iz *tuple*-a dohvaćena preko rednog broja, a pošto je poznato da taj

tuple ima samo jednu vrijednost po njegovoj deklaraciji u `InputReader`-u, može se vrijednost dohvatiti preko indeksa 0, ali moguće je i tražiti vrijednost s ključem "line". Dalje se radi o jednostavnoj primjeni `split()` metode nad spojenim meteorološkim informacijama da bi se dobilo polje čije se elemente može pregledno pohraniti u novi *tuple* te ga emitirati na idući *Bolt* preko kolektora.

`declareOutputFields()` ima istu zadaću kao i u ranije spomenutom *Spout*-u, ali u slučaju kôda 6 deklarirani *tuple* puno je bogatiji.

```
@Override
public void declareOutputFields(OutputFieldsDeclarer
outputFieldsDeclarer)
{
    outputFieldsDeclarer.declare(new
Fields("country","city","temperature","rainfall","wind","pollution"));
}
```

Kôd 6. `declareOutputFields()` metoda u `InputNormalizer`-u

4.2.5. `ExtremesEvaluator` Bolt

`ExtremesEvaluator`, isto kao i njegov prethodnik, nasljeđuje `BaseBasicBolt` te koristi iste metode. Uloga ovog *Bolt*-a je usporedba meteoroloških podataka s njihovim pojedinim ekstremnim vrijednostima i slanje obavijesti na idući *Bolt* koje sadrži ime države i tip vremenske nepogode. Ekstremne vrijednosti meteoroloških podataka su vrijednosti u kojima su vremenski uvjeti opasni za čovjeka i njegovu imovinu. Na -20 C°, izloženi čovjek dobiva ozebline. 35 C° je temperatura na kojoj vrućina čovjeku može uzrokovati razne zdravstvene tegobe. Ako kiša pada u količinama većim od 8 (l/m^2)/h, tada su očekivane poplave. Brzine vjetra veće od 150 km/h bacaju sposobne su nositi teže predmete, koji tada mogu nekoga ozlijediti. Zrak koji pokazuje PM2.5 vrijednost veću od 50 nije zdrav za ljudski organizam. Sve te vrijednosti su definirane u kôdu 7. Također postoji slična verzija ovog *Bolt*-a koja umjesto država, povezuje vremensku nepogodu s gradovima.

```
public class ExtremesEvaluator extends BaseBasicBolt
{
    private static final int TEMP_LOW =-20;
    private static final int TEMP_HIGH =35;
    private static final double RAIN =8;
    private static final int WIND =150;
    private static final int POLLUTION =50;

    @Override
    public void execute(Tuple tuple, BasicOutputCollector
basicOutputCollector)
    {
        String country=tuple.getStringByField("country");
```

```
        int
temperature=tuple.getIntegerByField("temperature");
        double rainfall=tuple.getDoubleByField("rainfall");
        int windSpeed=tuple.getIntegerByField("wind");
        int pollution=tuple.getIntegerByField("pollution");

        if(temperature< TEMP_LOW){
            basicOutputCollector.emit(new
Values(country,"temperatureLow"));
        }
        if(temperature> TEMP_HIGH){
            basicOutputCollector.emit(new
Values(country,"temperatureHigh"));
        }
        if(rainfall> RAIN){
            basicOutputCollector.emit(new
Values(country,"rainfall"));
        }
        if(windSpeed> WIND){
            basicOutputCollector.emit(new
Values(country,"wind"));
        }
        if(pollution>POLLUTION){
            basicOutputCollector.emit(new
Values(country,"pollution"));
        }
    }
```

Kôd 7. Deklaracija `ExtremesEvaluator` Bolt-a i `execute()` metoda

Ekstremne vrijednosti definirane na vrhu `ExtremesEvaluator`-a koriste se za prepoznavanje vremenske nepogode u dolazećim podacima. Prepoznavanje vremenske nepogode se obavlja uz pomoć jednostavne provjere je li dolazni podatak premašio njegovu pripadajuću ekstremnu vrijednost. Pri prepoznavanju nepogode emitira se novi *tuple* s informacijom o vrsti nepogode i njenoj lokaciji, kao što je vidljivo na kôdu 7.

Izgled *tuple*-a jasnije se vidi u `declareOutputFielsd()` metodi prikazanoj u kôdu 8.

```
@Override
public void declareOutputFields(OutputFieldsDeclarer
outputFieldsDeclarer)
{
    outputFieldsDeclarer.declare(new
Fields("country","type"));
}
```

Kôd 8. `declareOutputFields()` metoda `ExtremesEvaluator` *Bolt*-a

4.2.6. `ToplistManager` Bolt

`ToplistManager` *Bolt* prima *tuple* koji sadrži podatke o lokaciji i tipu vremenske nepogode te ih bilježi u mape. Mape sadrže imena lokacija i brojač koji odbrojava koliko se

puta određena nepogoda pojavila na toj lokaciji. *Tuple*-ov podatak pohranit će se u dvije mape, jedna sadrži podatke za određeni tip nepogode, a druga sve podatke bez obzira na tip. `ToplistManager` nasljeđuje `BaseBasicBolt` i koristi njegove metode `prepare()`, `execute()` i `cleanup()`. Za svakih *n* primljenih podataka, podaci su sortirani po količini vremenskih nepogoda u tome periodu, te se tada zapisuju u novu izlaznu datoteku. Broj *n* je definiran u konfiguraciji. Ti podaci bi se inače zapisivali u bazu podataka, ali pošto je jedan od ciljeva ovoga rada izmjeriti brzinu rada Storm aplikacije, baza podataka koja nije distribuirana na više računala bila bi usko grlo.

Metoda `prepare()` dohvaća potrebne informacije iz konfiguracije te nakon toga inicijalizira sve mape u kojima će se držati podaci.

```
public class ToplistManager extends BaseBasicBolt
{
    private Map<String, Integer> toplist;
    private Map<String, Integer> toplistTempHigh;
    private Map<String, Integer> toplistTempLow;
    private Map<String, Integer> toplistRain;
    private Map<String, Integer> toplistWind;
    private Map<String, Integer> toplistPollution;

    private static int printInterval;
    private static int counter=0;
    private static int fileNumber=1;

    private String fileLocation;

    @Override
    public void prepare(Map stormConf, TopologyContext context)
    {
        fileLocation = stormConf.get("dirToWrite").toString();
        printInterval=(Integer)stormConf.get("printInterval");
        initialize();
    }

    private void initialize(){
        toplist = new HashMap<String, Integer>();
        toplistTempHigh = new HashMap<String, Integer>();
        toplistTempLow = new HashMap<String, Integer>();
        toplistRain = new HashMap<String, Integer>();
        toplistWind = new HashMap<String, Integer>();
        toplistPollution = new HashMap<String, Integer>();
    }
```

Kôd 9. Deklaracija `ToplistManager` *Bolt*-a i metoda `prepare()`

U kôdu 9 se vidi kako se iz konfiguracije dohvaćaju informacije o putanji izlaznih datoteka i intervalu u kojem će se podaci ispisivati i pohranjuju u lokalne varijable.

Metoda `execute()` razvrstava podatke u pripadajuće mape te provjerava da li je vrijeme za ispis mapa u datoteke.

```
@Override
public void execute(Tuple tuple, BasicOutputCollector
basicOutputCollector)
{
    String location = tuple.getString(0);
    String type = tuple.getString(1);

    if (type.equals("temperatureLow"))
    {
        editToplist(toplistTempLow, location);
    }
    else if (type.equals("temperatureHigh"))
    {
        editToplist(toplistTempHigh, location);
    }
    else if (type.equals("rain"))
    {
        editToplist(toplistRain, location);
    }
    else if (type.equals("wind"))
    {
        editToplist(toplistWind, location);
    }
    else if (type.equals("pollution"))
    {
        editToplist(toplistPollution, location);
    }
    editToplist(toplist, location);

    counter++;
    if(counter>=printInterval){
        outputToplists();
        counter=0;
        fileNumber++;
        if(fileNumber%10==0){
            initialize();
        }
    }
}

private void editToplist(Map<String, Integer> map, String
country)
{
    if (!map.containsKey(country))
    {
        map.put(country, 1);
    }
    else
    {
        Integer i = map.get(country) + 1;
        map.put(country, i);
    }
}

private void outputToplists()
{
    sortToplists();
    printToplist(toplist, fileLocation +
"countryResults"+fileNumber+".txt");
    printToplist(toplistTempLow, fileLocation +
```

```
"countryLowTempResults"+fileNumber+".txt");
    printToplist(toplistTempHigh, fileLocation +
"countryHighTempResults"+fileNumber+".txt");
    printToplist(toplistRain, fileLocation +
"countryRainResults"+fileNumber+".txt");
    printToplist(toplistWind, fileLocation +
"countryWindResults"+fileNumber+".txt");
    printToplist(toplistPollution, fileLocation +
"countryPollutionResults"+fileNumber+".txt");
}
```

Kôd 10. Metoda `execute()u ToplistManager` *Bolt-u*

Nakon svakog desetog pisanja stanja mapa u datoteke, resetiraju se sve mape. Tako se zadržavaju povijesni zapisi i ti stari zapisi ne utječu na poredak u aktualnim mapama. Opisana funkcionalnost se prikazuje u kôdu 10.

Metoda `declareOutputFields()` se ne koristi u ovome *Bolt*-u, zato što on ne šalje nikakav *tuple* dalje kroz topologiju. Metoda `cleanup()` se poziva kada se topologija gasi. U kôdu 11 `cleanup()` metoda pohranjuje bilo kakve podatke koji bi bili izgubljeni zbog gašenja topologije.

```
@Override
public void cleanup()
{
    fileNumber++;
    outputToplists();
}

private void outputToplists()
{
    sortToplists();
    printToplist(toplist, fileLocation +
"countryResults"+fileNumber+".txt");
    printToplist(toplistTempLow, fileLocation +
"countryLowTempResults"+fileNumber+".txt");
    printToplist(toplistTempHigh, fileLocation +
"countryHighTempResults"+fileNumber+".txt");
    printToplist(toplistRain, fileLocation +
"countryRainResults"+fileNumber+".txt");
    printToplist(toplistWind, fileLocation +
"countryWindResults"+fileNumber+".txt");
    printToplist(toplistPollution, fileLocation +
"countryPollutionResults"+fileNumber+".txt");
}
```

Kôd 11. Metoda `cleanup()`

4.2.7. `StormTopology` klasa i `TopologyBuilder`

`StormTopology` je *main* klasa ove aplikacije i prikazana je u kôdu 12. U njoj se koristi `TopologyBuilder` klasa za izradu Storm topologije koristeći ranije prikazane *Bolt*-ove i *Spout*.

```
public class StormTopology
{
   public static void main(String[] args) throws
InterruptedException {
      TopologyBuilder builder=new TopologyBuilder();
      builder.setSpout("input-reader",new InputReader());
      builder.setBolt("input-normalizer",new
InputNormalizer(),3).shuffleGrouping("input-reader");
      builder.setBolt("extremes-evaluator",new
ExtremesEvaluator(),3).shuffleGrouping("input-normalizer");
      builder.setBolt("country-toplist-manager",new
ToplistManager()).shuffleGrouping("extremes-evaluator");

      Config conf=new Config();
      conf.put("fileToRead","/home/dominik1/test.csv");
      conf.put("dirToWrite","/home/dominik1/");
      conf.put("printInterval",1000000);
      try{
         StormSubmitter.submitTopology("HarmfulWeather-
Topology-1331",conf,builder.createTopology());
      }
      catch (Exception e){e.printStackTrace();}}
}
```

Kôd 12. `main()` i `TopologyBuilder`

`TopologyBuilder` se koristi za ulančavanje komponenti prema željenom grupiranju. Na primjeru dodavanja `InputNormalizera`, vidljivo je da je on povezan nasumičnim grupiranjem (`shuffleGrouping()`) s `InputReader` *Spout*-om. Također je vidljivo u trećem parametru `setBolt()` metode da će se instancirati 3 instance `InputNormalizera` i `InputEvaluatora`. Koliko instanci po *Bolt-u* daje najbolje rezultate? To se mjeri u idućem poglavlju.

Nimbus proces topologije prima u obliku ".jar" datoteke, a Maven može iz kôda generirati JAR izvršavanjem naredbe `mvn clean install`. Generirani JAR nalazi se u "target" direktoriju.

4.3. Pokretanje aplikacije na Storm klasteru

Klaster računala dovoljno jak za pokretanje Storm topologije u produkcijskom okruženju vrlo je nepristupačan, primarno zbog svoje cijene. Zato će u svrhu ovoga rada poslužiti od 2 do 5 virtualnih mašina identičnih specifikacija. Virtualne mašine pokrenute su uz pomoć Oracle VM VirtualBox platforme. Oracle VM jednostavan je za korištenje. Nakon preuzimanja Linux Ubuntu operacijskog sustava i njegove instalacije na virtualnu mašinu, podešava se Storm klaster. Podešavanje je objašnjeno u idućim poglavljima.

4.3.1. Instalacija i pokretanje Zookeeper servera

Zookeeper je *open source* i preuzima se s Apache repozitorija. Nakon raspakiravanja preuzete datoteke, instalacija je gotova i sve što je ostalo je konfiguriranje. Računalo mora imati instaliran *Java Development Kit* (JDK) da bi Zookeeper proradio. Zookeeper je potrebno konfigurirati tako da se uspostavi dijeljena memorija između više virtualnih mašina. U mapi "$ZOOKEEPER_HOME/conf'' nalazi se datoteka "zoo.cfg". Ta datoteka sadrži većinu konfiguracije Zookeeper-a, a nakon dodatnog konfiguriranja dobi se prikazano u kôdu 13.

```
syncLimit=5
dataDir=../dataDir
clientPort=2181
server.1=10.0.2.7:2888:3888
server.2=10.0.2.5:2888:3888
server.3=10.0.2.6:2888:3888
server.4=10.0.2.8:2888:3888
#server.5=10.0.2.9:2888:3888
```

Kôd 13. Konfiguracija Zookeeper servera

Parametar `syncLimit` definira koliko maksimalno puta Zookeeper može primiti zahtjev bez da vrati potvrdu da je primio isti, odnosno koliko server smije biti neusklađen s vođom kvoruma. Parametar `dataDir` je mjesto gdje je trenutno stanje memorije pohranjeno i mjesto gdje korisnik definira ID Zookeeper servera. Port na koji se klijenti spajaju definiran je kao 2181. Serveri koji dijele memoriju definirani su u formatu `''server.id=ip:portZaPovezivanjeSaVođomKvoruma:PortZaUlazNaServer''`. Serverov ID je potrebno definirati u "myid" datoteci, a se mora nalaziti u direktoriju definiranom u `dataDir` parametru. Datoteka "myid" nema ekstenziju i sadrži samo broj koji definira ID lokalnog Zookeeper servera. Za pokretanje Zookeeper servera, potrebno je u "$ZOOKEEPER_HOME/bin" direktoriju pokrenuti "zkServer.sh" skriptu koristeći naredbu:

```
./zkServer.sh start-foreground
```

Tu istu naredbu potrebno je izvršiti na svim računalima definiranim u konfiguraciji. Za provjeru funkcionalnosti servera, potrebno je pokrenuti u istom direktoriju "zkCli.sh" skriptu, ovako:

```
./zkCli.sh -server localhost
```

Ta skripta pokreće Zookeeper klijenta u konzoli, gdje korisnik ima pristup naredbama za upravljanje datotečnim sustavom. Naredbama poput `ls` ili `get` moguće je navigirati kroz hijerarhiju i čitati podatke u sustavu. Korištenjem klijenta i izvršavanjem naredbe `create` kreira se *Znode* na jednom serveru. Taj *Znode* bit će repliciran na ostale servere definirane u konfiguraciji. Povezivanje uz pomoć klijenta na bilo koji od tih servera i provjera da li je i tamo vidljiv novo kreirani *Znode*, najbolji je način provjere da li Zookeeper pravilno radi.

4.3.2. Instalacija i konfiguriranje Storm sustava

Storm je, kao i Zookeeper, *open source* i preuzima se s Apache repozitorija. Raspakiravanjem preuzetog finalizira se instalacija. Računalo mora imati instaliran *Java Development Kit* (JDK) i Python 2.6. Konfiguracijska datoteka nalazi se u "$STORM_HOME/conf" direktoriju i zove se "storm.yaml". Nakon konfiguriranja te datoteke, ona bi trebala izgledati kako je prikazano u kôdu 14.

```
nimbus.seeds: ["localhost"]
storm.zookeeper.servers:
     - "10.0.2.7"
     - "10.0.2.5"
     - "10.0.2.6"
     - "10.0.2.8"
supervisor.slots.ports:
    - 6700
storm.local.dir: "dataDir"
```

Kôd 14. Konfiguracija Storm klastera

Parametar "nimbus.seeds" sadrži IP adrese svih računala u klasteru na kojima je pokrenut *nimbus* proces. Parametar `storm.zookeeper.servers` sadrži sve IP adrese Zookeepeer servera ranije definirane u "zoo.cfg" datoteci. Parametar

`supervisor.slots.ports` sadrži listu *port*-ova preko kojih je moguće komunicirati sa *supervisor* procesom. Direktorij za generirane datoteke Storm-a definira se u `storm.local.dir` parametru.

4.3.3. Pokretanje Storm sustava

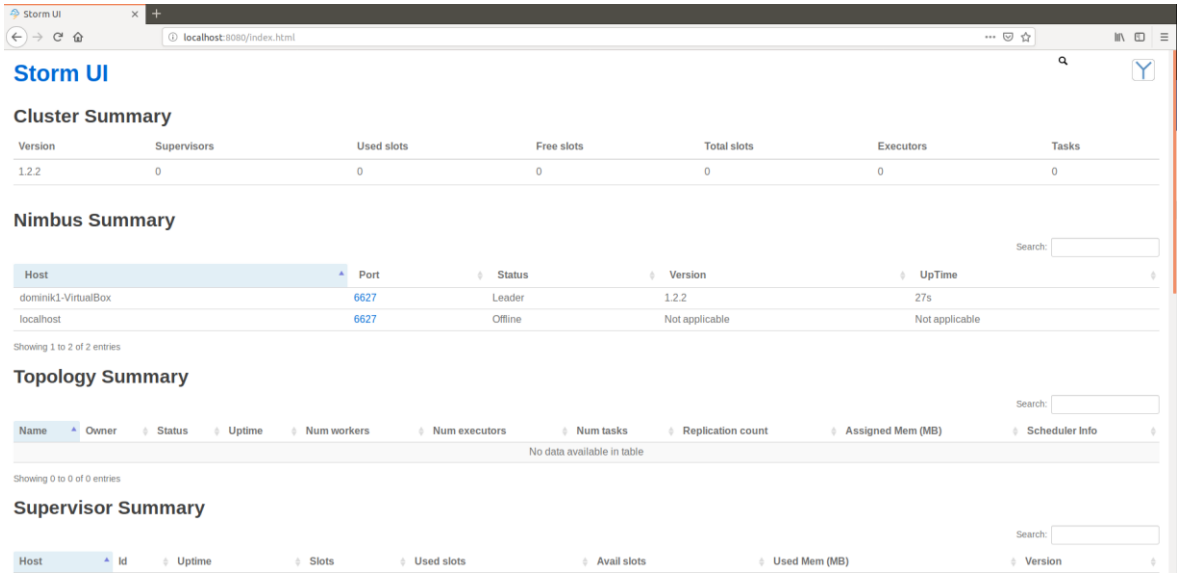
Pozicioniranjem u "$STORM_HOME" direktorij i izvršavanjem iduće naredbe pokreće se *nimbus* proces:

```
bin/storm nimbus
```

Idućom naredbom pokreće se Storm UI:

```
bin/storm ui
```

Storm UI je web aplikacija koja u grafičkom sučelju prikazuje stanje Storm platforme. Storm UI aplikaciji pristupa se preko porta 8080.

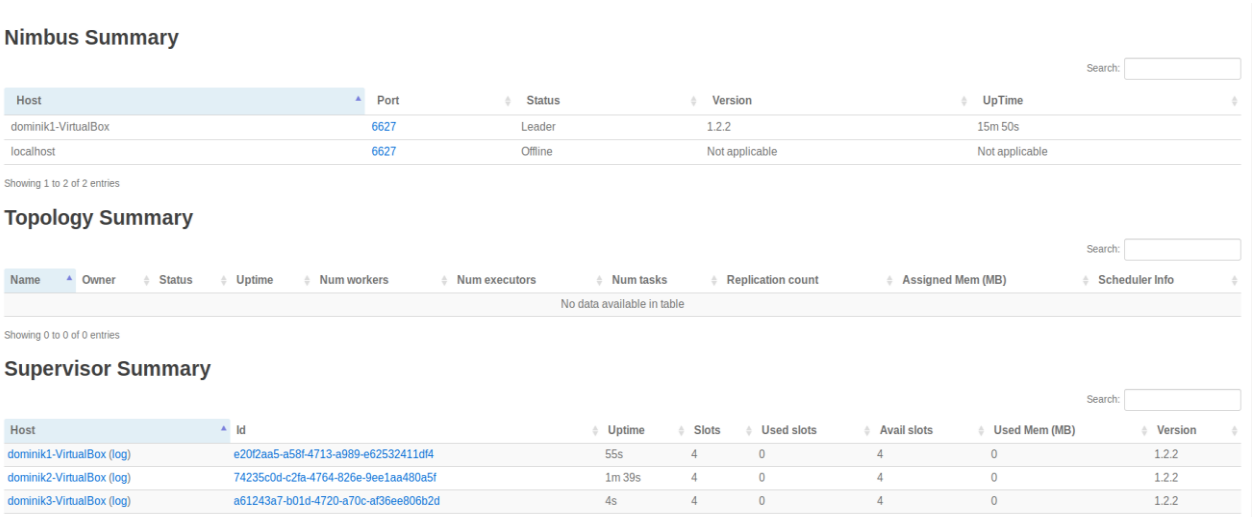


Slika 7. Storm UI

UI prikazan na slici 7 izgleda prazno zato što nema pokrenutog niti jednog *supervisor* procesa niti pokrenute topologije. *Nimbus* je centralna točka koja vodi cijeli sustav, dok *supervisor* procesi obavljaju posao pokrenutih topologija. Sada kada je na jednom računalu pokrenut *nimbus*, treba na ostalim računalima gdje Zookeeper radi pokrenuti *supervisor* procese. *Supervisor* se pokreće naredbom:

```
bin/storm supervisor
```

Nakon pokretanja 3 *supervisor procesa*, Storm UI prikazuje informacije kao na slici 8.



Slika 8. Storm UI s 3 povezana *supervisor*-a

Sada kada je Storm klaster spreman, nastavlja se s pokretanjem Storm topologije.

4.3.4. Pokretanje Storm topologije

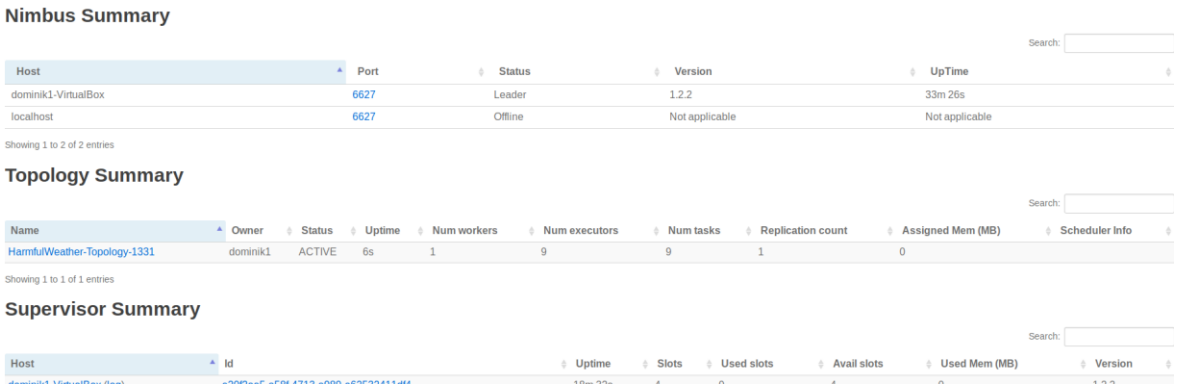
JAR koji sadrži Storm topologiju pohranjen je na poznato mjesto u virtualnoj mašini. Za pokretanje Storm topologije izvršava se naredba:

```
bin/storm jar "lokacijaJARa'' imeMainKlase
```

Naredba za pokretanje aplikacije za prepoznavanje vremenskih nepogoda izgledala je ovako:

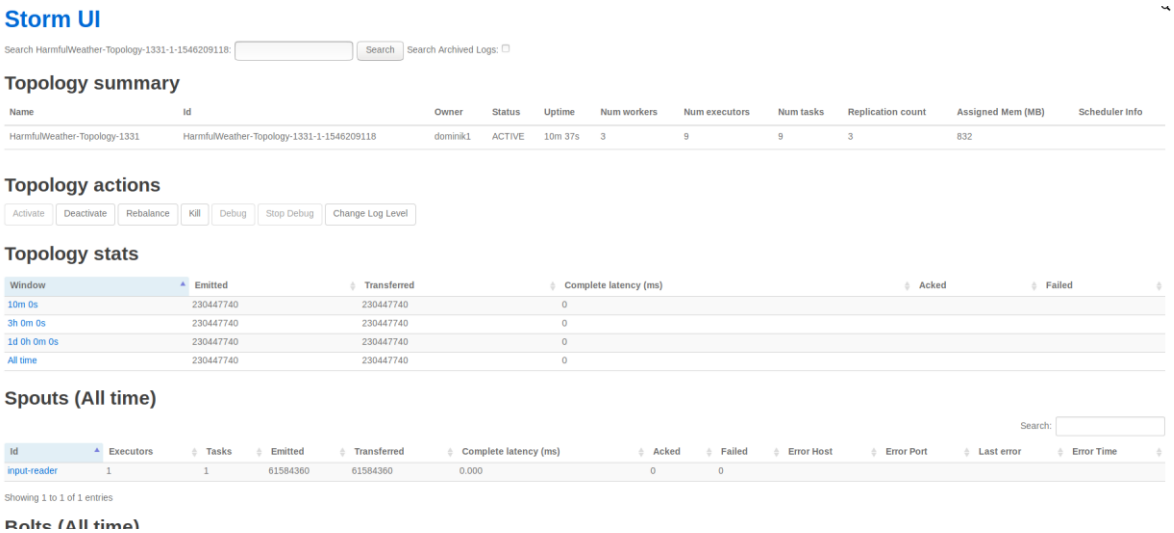
```
bin/storm jar "/home/dominik1/stormJar/storm-0.0.1-SNAPSHOT13312.jar" com.vidakovic.storm.StormTopology
```

Storm UI pokazuje da je topologija uspješno pokrenuta.



Slika 9. Pokrenuta Storm topologija

Klikom na ime topologije vidljivo na slici 9, Storm UI daje detaljniji pregled topologije. Prikazuje i broj emitiranih *tuple*-a u različitim vremenskim periodima.



Slika 10. Detaljan prikaz topologije

Topologija je ispisala svoje rezultate u više datoteka, datoteka koja sadrži sumirani broj vremenskih nepogoda ovako prikazuje najnesretnijih 5 država:

China: 1825487

Indonesia: 1176821

Russia: 585521

Philippines: 464975

Brazil: 424800

Sustav dokazano radi. Sada slijedi testiranje topologije s različitim brojevima *executor-a* (instanci *Bolt*-ova, dretva po boltu) i *supervisor*-a (računala).

4.3.5. Testiranje topologije u različitim uvjetima

Storm UI prati broj emitiranih *tuple*-a kroz razne vremenske periode, u svrhu testiranja koristit će se period od zadnjih 10 minuta. Prvi test uključivat će jedan *supervisor* proces i samo jedan *executor* po *Bolt*-u, drugi će imati isti broj *supervisor*-a, ali 3 *executor*-a za svaki *Bolt*, treći će imati 6 *executor*-a, a četvrti 9. Tako će se nastaviti u 5 krugova testiranja i svaki krug će imati jedan *supervisor* više. Rezultati su prikazani u idućoj tablici:

Tablica 1. Rezultati testiranja

Broj *supervisor*-a (*worker* računala)	Broj *executor*-a (dretvi) po *Bolt*-u	Broj emitiranih *tuple*-a u 10 minuta
1	1	115678556
1	3	135786786
1	6	168971253
1	9	156478624
2	1	156842378
2	3	183512547
2	6	215675752
2	9	211578617
3	1	205678423
3	3	232548642
3	6	276523642
3	9	277562145
4	1	215687754
4	3	245761237
4	6	279842354
4	9	285314852
5	1	175631222

Na rezultatima testiranja se jasno vidi da broj *executor*-a osjetno utječe na brzinu rada topologije, ali za aplikaciju s ovako malo *Bolt*-ova povećanje brzine se smanjuje s prelaskom sa 6 *executor*-a po *Bolt*-u na 9. Brzina rada je naglo pala pri priključenju 5. računala u klaster zato što računalo na kojemu se radi testiranje ne može podnijeti više od 4 ili 5 pokrenutih virtualnih mašina. Virtualne mašine su podešene tako da imaju malo RAM-a i slabu procesorsku moć, zato ja količina procesiranih *tuple*-a ovako mala. Na fizičkim računalima koji se u praksi koriste kao serveri, ovakva jednostavnija topologija emitirala bi najmanje 1000000 *tuple*-a po sekundi.

4.4. Zaključak za Apache Storm

Apache Storm pokazao se veoma pristupačan za programera koji se susreće s njime prvi puta. Kao distribuirani sustav Storm ima nisku povezanost između članova sustava. Svaki *supervisor* može raditi bez drugih *supervisor-a,* i u slučaju kvara *nimbus* procesa njihov nadzor preuzima drugi nimbus, ako postoji, ako ne postoji tada će *supervisor*-i zapamtiti svoje trenutno stanje te nastaviti normalno s radom čim se novi *nimbus* proces pojavi. Što se tiče paralelizma, *nimbus* kôd topologije prenosi na svaki *supervisor* tako da svaki *supervisor* puno više vremena provede izvršavajući instrukcije, nego što čeka instrukcije. Konkurencija između članova Storm sustava je bila dobro raspodijeljena, tj. računala su između sebe ravnomjerno raspodijelila posao. (Jonathan Leibiusky, 2012)

Storm se može primijeniti u bilo kojem sustavu koji prima ogromne količine podataka u kratkom vremenskom roku, i zahtjeva korisne informacije iz tih podataka što prije moguće. Na primjer, grad u kojemu bi svi automobili svakih 5 do 10 sekundi slali svoju lokaciju i odredište mogao bi uz *stream processing* bolje usmjeriti svoj promet u stvarnom vremenu.

Zaključak

Sve više i više firma razumije korist čuvanja i analiziranja podataka. Razvoj računala zaostaje u odnosu na rast količine podataka i distribuirani sustavi su trenutno jedino rješenje. Prepreke u toj grani računarstva su kompleksnost razvoja rješenja koja uključuju distribuirane sustave i cijena hardvera, ali iz godine u godinu te prepreke su sve manje. Još jedna prepreka je ovisnost paralelnih distribuiranih sustava o spajanju u LAN mrežu zbog brzine mrežne komunikacije.

Najveći potencijal predstavlja Apache Spark. Zbog temelja na ranije poznatom Hadoop-u, razvoja rješenja za sve više aktualnih problema (od *stream processing*-a do strojnog učenja) i velike brzine rada, Spark je postao veoma popularan, a velika i brojna zajednica jako puno znači. Primjer koristi velike zajednice oko tehnologije je izrada aplikacije za prepoznavanje vremenskih nepogoda. Zahvaljujući ogromnom broju korisnika Storm tehnologija, količina informacija dostupnih početniku je ogromna. Čak je i postavljanje Storm klastera uspjelo nekome tko nije specijaliziran u sistemskom i mrežnom računarstvu. Uz razumijevanje programiranja i stečeno razumijevanje sustava zahvaljujući mnogobrojnim izvorima, nije preveliki problem izraditi Storm topologiju, nema potrebe za specijalnim znanjem. Aplikacija za praćenje vremenskih nepogoda se uz minimalne promjene može pripremiti za rad u stvarnom svijetu, umjesto virtualne mašine koja prima simulirane podatke. Ovaj rad bi pokazao više puta bolje rezultate da je hardver potreban za rad u distribuiranim sustavima nešto dostupniji. Rad s virtualnim mašinama znatno otežava povezivanje više distribuiranih aplikacija, npr. povezivanje Apache Kafka aplikacije i Apache Cassandra baze podataka sa Storm aplikacijom za prepoznavanje vremenskih nepogoda.

Uz nedostatak alternativa, rast u popularnosti i dolaskom novih i jednostavnih rješenja, sustavi s distribuiranim procesiranjem sve će više rasti i donositi nove vrijednosti sve širem spektru poslovanja.

Popis kratica

API	Application Programming Interface	aplikacijsko programersko sučelje
CAP	Consistency, Availability, Partition tolerance	dosljednost, dostupnost, tolerantnost particija
ETL	Extract, Transform, Load	izvlačenje, transformiranje, punjenje
ID	Identity document	identifikacijski dokument
IDE	Integrated Development Environment	integrirano razvojno okruženje
JAR	Java Archive	Java arhiva
JDK	Java Development Kit	Java razvojni paket
JIT	Just in time	točno na vrijeme
JNDI	Java Naming and Directory Interface	Java nazivanje i sučelje direktorija
JVM	Java Virtual Machine	java virtualna mašina
GFS	Google File System	Google datotečni sustav
HDFS	Hadoop Distributed File System	Hadoop distribuirani datotečni sustav
LAN	Local Area Network	lokalna mreža
MIMD	Multiple Instruction Multiple Data	Višestruki instrukcijski tok, višestruki podatkovni tok
MISD	Multiple Instruction Single Data	Višestruki instrukcijski tok, jedan podatkovni tok
NDFS	Nutch Distributed File System	Nutch distribuirani datotečni sustav
NUMA	Non-Uniform Memory Access	nehomogen pristup memoriji
POM	Project Object Model	projekt objektni model
RAM	Random Access Memory	Memorija nasumičnog pristupa
RDD	Resilient Distributed DataSet	Otporan distribuirani skup podataka
SIMD	Single Instruction Multiple Data	Jedan instrukcijski tok višestruki podatkovni tok
SISD	Single Instruction Single Data	Jedan instrukcijski tok jedan podatkovni tok
SQL	Structured Query Language	strukturirani upitni jezik
TCP	Transmission Control Protocol	protokol kontrole prijenosa
UI	User Interface	Korisničko sučelje
UMA	Uniform Memory Access	Homogen pristup memoriji
XML	Extensible Markup Language	Proširivi označni jezik

Popis slika

Popis kodova

Popis tablica

Literatura

[1] Herbert Schildt;(2014); Java The Complete Reference Ninth Edition; Oracle Press

[2] Danijel Kučak;(2010); Programiranje u Javi II; Algebra d.o.o.;

[3] Andrew S. Tanenbaum, M. v. (2016). *Distributed Systems: Principles and Paradigms.* London: Pearson Education

[4] Bashir, I. (2017). *Mastering Blockchain.* Birmingham: Packt Publishing.

[5] Bill Chambers, M. Z. (2018). Spark - The Definitive Guide - Big data processing made simple. Sebastopol: O'Reilly.

[6] Ajay D.Kshenaklyani, M. S. (2008). Distributed Computing: Principles, Algorithms, and Systems. Cambridge: Cambridge.

[7] Burnette, E. (2005). *Eclipse IDE Pocket Guide.* O'Reilly.

[8] Frampton, M. (2015). *Mastering Apache Spark.* Birmingham: Packt Publishing.

[9] Holden Karau, A. K. (2015). *Learning Spark.* Sebastopol: O'Reilly.

[10] Iuliana Cosmina, R. H. (2017). *Pro Spring 5.* Apress.

[11] Jonathan Leibiusky, G. E. (2012). *Getting Started with Storm.* Sebastopol: O'Reilly.

[12] Mike Keith, M. S. (2015). *Pro JPA 2.* Apress.

[13] Tim O'Brien, M. M. (2011). *Maven; The Complete Reference.* Sonatype.

[14] White, T. (2012). *Hadoop: The Definitive Guide.* Cambridge: O'Reilly.

ALGEBRA VISOKO UČILIŠTE

SUSTAVI S DISTRIBUIRANIM PROCESIRANJEM

Pristupnik: Dominik Vidaković, 0321004725

Mentor: v. pred. Aleksander Radovan, dipl. ing.