VIZUALIZACIJA PODATAKA ZAPISNIKA PROCESORA, DISKA I RADNE MEMORIJE

Mamuša, Stjepan

Undergraduate thesis / Završni rad

2020

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Split / Sveučilište u Splitu**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:228:780476

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-03-29**

Repository / Repozitorij:

Repository of University Department of Professional Studies

zir.nsk.hr

UNIVERSITY OF SPLIT

dabar
DIGITALNI AKADEMSKI ARHIVI I REPOZITORIJI

SVEUČILIŠTE U SPLITU
SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE
Preddiplomski stručni studij Elektronike

STJEPAN MAMUŠA

ZAVRŠNI RAD

VIZUALIZACIJA PODATAKA ZAPISNIKA PROCESORA, DISKA I RADNE MEMORIJE

Split, rujan 2020.

SVEUČILIŠTE U SPLITU
SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE
Preddiplomski stručni studij Elektronike

Predmet: Računalne mreže

ZAVRŠNI RAD

Kanditat: Stjepan Mamuša

Naslov rada: Vizualizacija podataka zapisnika procesora, diska i radne memorije

Mentor: Toni Jončić

Komentor: Tonko Kovačević

Split, rujan 2020.

Sadržaj

Sažetak 1

1. UVOD 2

1.1 SDI – Software Defined Infrastructure 3

1.2 Opis komponenti sustava 5

2. RAZVOJ APLIKACIJE 11

2.1 Node.js 11

2.1.1 Parser zapisnika opterećenja procesora 14

2.1.2 Parser zapisnika iskorištenja radne memorije 17

2.1.3 Parser zapisnika iskorištenje memorije diska 20

2.1.4 Pomoćni kod 23

2.2 InfluxDB 24

2.3 Grafana 25

2.4 Dokumentacija 25

3. GLAVNI DIO I UPORABA 26

3.1 Glavni dio programa – main.js i bin.js 26

3.2 Uporaba 31

4. ZAKLJUČAK 33

5. LITERATURA 34

6. POPIS SLIKA 35

Sažetak

Vizualizacija podataka zapisnika procesora, diska i radne memorije

Rad obrađuje tematiku izrade aplikacije za sintaktičku analizu (eng. *parsing*) i vizualizaciju podataka iz zapisnika (eng. *logs*) prikupljenih unutar Ericsson HDS 8000 sustava za podatkovne centre. Vrste podataka koji se prikupljaju uključuju, ali nisu ograničene na, opterećenje procesora, zauzeće pohrambenog prostora na diskovima i to po direktorijima (Linux naredba „*df*“), te zapise o iskorištenoj i ukupnoj radnoj memoriji (RAM). Sustav analize i vizualizacije čine tehnologije Node.js, InfluxDB i Grafana. Zapisnici se skupljaju pomoću već postojoće infrastrukture koda nadzirući niti izvršavanja (eng. *threads*) Java virtualne mašine (JVM).

Summary

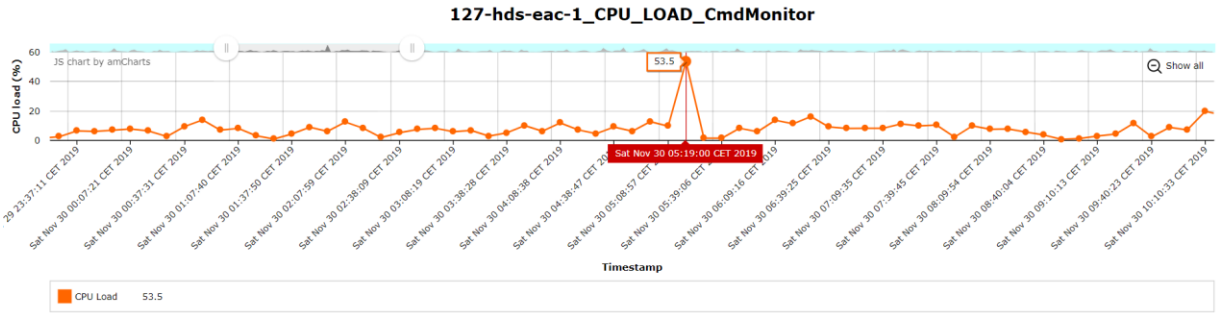
Visualization of CPU, disk and RAM log data

The thesis discusses the topic of parsing and visualizing log data collected within Ericsson HDS 8000 data center system. Collected data types include, but are not limited to, CPU load, disk usage per directory and mount point (Linux command „*df*“) and logs concerning used and total RAM. Analysis and visualization system is comprised of Node.js, InfluxDB and Grafana technologies. The logs are collected via existing code infrastructures by thread monitoring within Java Virtual machine (JVM)

1. UVOD

Tema rada je izrada aplikacije za sintaktičku analizu (u daljnjem tekstu *parsiranje*) podataka iz zapisa o opterećenju procesora, zauzeću diskovne i radne memorije, te vizualizacija istih tih podataka. Aplikacija je razvijana u tvrtci Ericsson Nikola Tesla d.d. u Splitu, gdje je dodijeljena tema na temelju potreba u svakodnevnom izvršavanju zadataka. Kroz uvod će biti dan uvid u proizvod HDS (Hyperscale Datacenter System) 8000 (u daljnjem tekstu SDI – Software Defined Infrastructure), problematiku s kojom se timovi susreću, načine funkcioniranja suradnje između dva tima, a uz to će biti objašnjeno porijeklo odluke o izradi ove aplikacije. Glavni dio rada baviti će se samom izradom, ali potpuni sustav uz aplikaciju pisanu u programskom jeziku JavaScript – tehnologija Node.js, čine još baza podataka InfluxDB, koja je vremenski orijentirana baza podataka, dakle nema relacijskih tablica kao u standardnim relacijskim bazama tipa MySQL, već je se umjesto ideje tablica koriste mjerenja, a vizualizaciju odrađuje web aplikacija Grafana. Ove dvije popratne komponente biti će naknadno pojašnjene. Razvoj proizvoda ima ciklus unutar kojeg su različite faze, stoga je bitno napomenuti da je autor dio tima *Release System Test – Test Automation* (RST – TA), dakle to je zadnja faza u razvojnom ciklusu proizvoda, a bavi se testiranjem sustava pred distribuciju, odnosno pred isporuku kupcu. *Test Automation* tim čini potporu *Release System Test* (RST) timu na način da se provjerene specifikacije testova sustava automatiziraju za to predviđenim alatima. Kako je inženjerima koji izvode testove na temelju razvijenih testnih specifikacija bitno nadzirati hardware-sku opremu, točnije pratiti rad procesora pojedinih komponenti, iskorištenje radne memorije, iskorištenje pohrambenog prostora, te pratiti rad mrežne opreme, tako je razvijeno prvotno rješenje koje opremu nadzire prateći rad niti izvršavanja (eng. *threads*) automatiziranih testova unutar Java virtualne mašine (JVM – Java Virtual Machine). To rješenje se zasniva na otvaranju veze s računalne jedinice koja izvodi testiranje (to može biti pojedino računalo zaposlenika, mrežni čvor u obliku poslužitelj ili pak glavni centralni poslužitelj) prema komponenti koja se nadgleda, a sve komponente na sebi imaju SSH poslužitelja. Veza se ostvaruje SSH protokolom, gdje se po potrebi otvaraju nove SSH veze u ovisnosti komponente koja se nadzire. Ovisno od parametra koji se nadzire preko SSH veze izdaju se komande na domaćinskom operativnom sustavu (strana gdje je SSH poslužitelj) pa se tako na primjer za nadzor procesora mogu koristiti Linux naredbe **sar**, **top** i slične. Izlaz naredbe se onda

proslijedi u nit izvršavanja, a dio koji nadzire niti onda sintaktičkom analizom pretvara taj izlaz u pogodan oblik podataka u Java programskom jeziku te se ti podaci na kraju vizualiziraju pomoću prvotnog rješenja za vizualizaciju. Na Slika 1.1 Prikaz prvotnog rješenja na primjeru procesora vidi se kako izgleda već postojeće rješenje za vizualizaciju. Glavna baza koda iz prikupljenih podataka stvara **.html** datoteke s grafovima iz kojih je moguće iščitati bilo kakvu anomaliju ili očekivano ponašanje, ali problem kod ovog pristupa je kod mrežnih komponenti sustava gdje se po preklopniku nadzire promet na 90 priključaka, a uz to postoji više parametara prometa za nadzor, pa tako u samom početku postoji 90 grafova koji onda vizualizaciju čine ne upotrebljivom zbog velikog opterećenja u pregledniku. Ovaj dio je riješen tako da se na prezentacijskom dijelu koristi gotov servis Grafana. Uz vizualizaciju, nedostatak postojećeg rješenja je u tome što nema spremanja podataka u centralnu bazu te nije moguće vršiti analizu bez da svaki zaposlenik preuzme cijeli sadržaj zapisnika uključujući generirane **.html** datoteke. U ovom dijelu rješenje je postignuto uvođenjem baze podataka InfluxDB. Opis proizvoda i komponenti je u nastavku teksta.

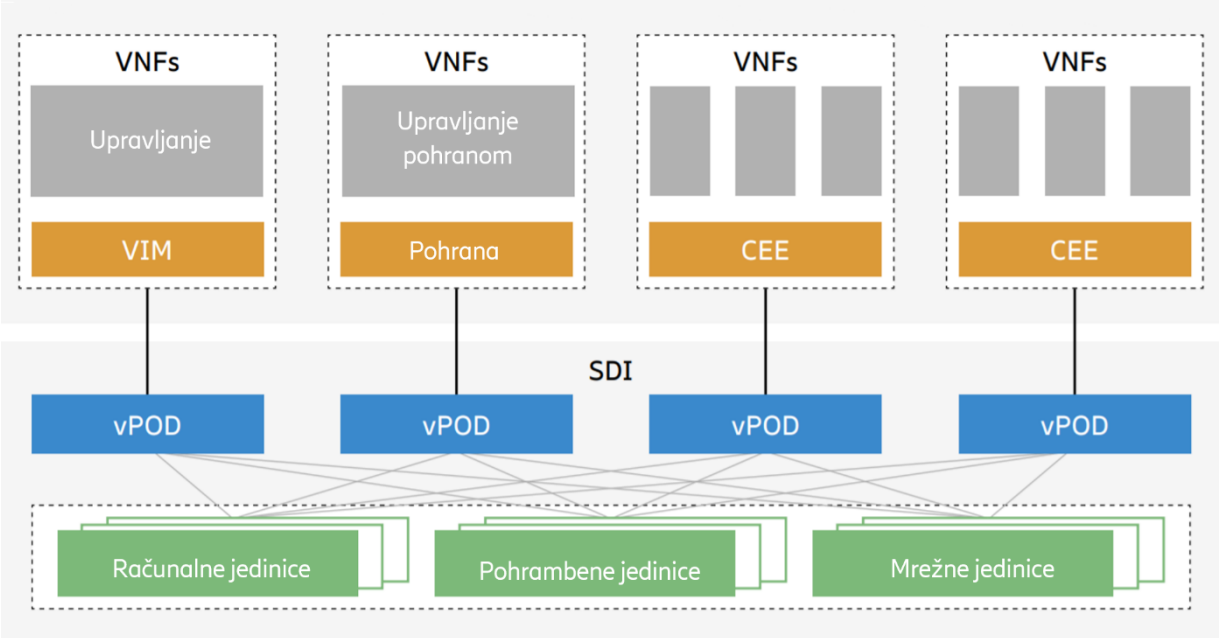


Slika 1.1 Prikaz prvotnog rješenja na primjeru procesora

1.1 SDI – Software Defined Infrastructure

Kako bi se dobio što zorniji prikaz i pružilo razumijevanje sustava i potrebe za aplikacijom rad se otvara s opisom proizvoda unutar čijeg opsega je razvijena. Software-ski definirana infrastruktura, skraćeno SDI, bivšeg naziva HDS 8000, je proizvod razvijen za podatkovne centre unutar Ericsson-ove jedinice za računarstvo u oblaku (eng. *cloud computing*) naziva

Cloud Development (razvoj software-a i hardware-a za računarstvo u oblaku). SDI je sustav za podatkovne centre kojim se omogućuje definiranje infrastrukture korištenjem upravljačkog software-a. Infrastrukturu čine dostupni resursi unutar podatkovnom centra kao što su računalne, pohrambene i mrežne jedinice. Pod definiranjem strukture podrazumijeva se stvaranje virtualnih performansno optimiziranih podatkovnih centara (vPOD – virtual Performance Optimized Datacenter) te kreiranje resursa unutar vPOD-a. Jedan vPOD predstavlja logičku cjelinu te u sebi sadrži svu potrebnu prethodno definiranu strukturu, a nakon dodjele resursa unutar vPOD-a moguće je kreirati i potrebne mreže za povezati dodijeljene resurse. SDI čini dio portofolia NFVI (Network Function Virtualization Infrastructure), odnosno virtualizacija mrežnih funkcija. To je koncept mrežne arhitekture koji koristi tehnologije IT virtualizacije da bi virtualizirao čitave klase mrežnih čvorova u građevinske blokove koje je moguće zajedno povezati s ciljem stvaranja komunikacijskih usluga. Ericsson NFVI rješenje omogućuje operaterima puštanje u pogon virtualnih mrežnih funkcija ili nativnih funkcija u oblaku, različitih proizvođača, kao i sustava aplikacija za operativnu podršku (OSS) ili poslovnu podršku (BSS) i sve to s brzinom, dok se ukupna cijena vlasništva drži niskom. Na slici ispod vidi se logička organizacija sustava.



Slika 1.2 Prikaz položaja SDI proizvoda u NFVI portfoliju

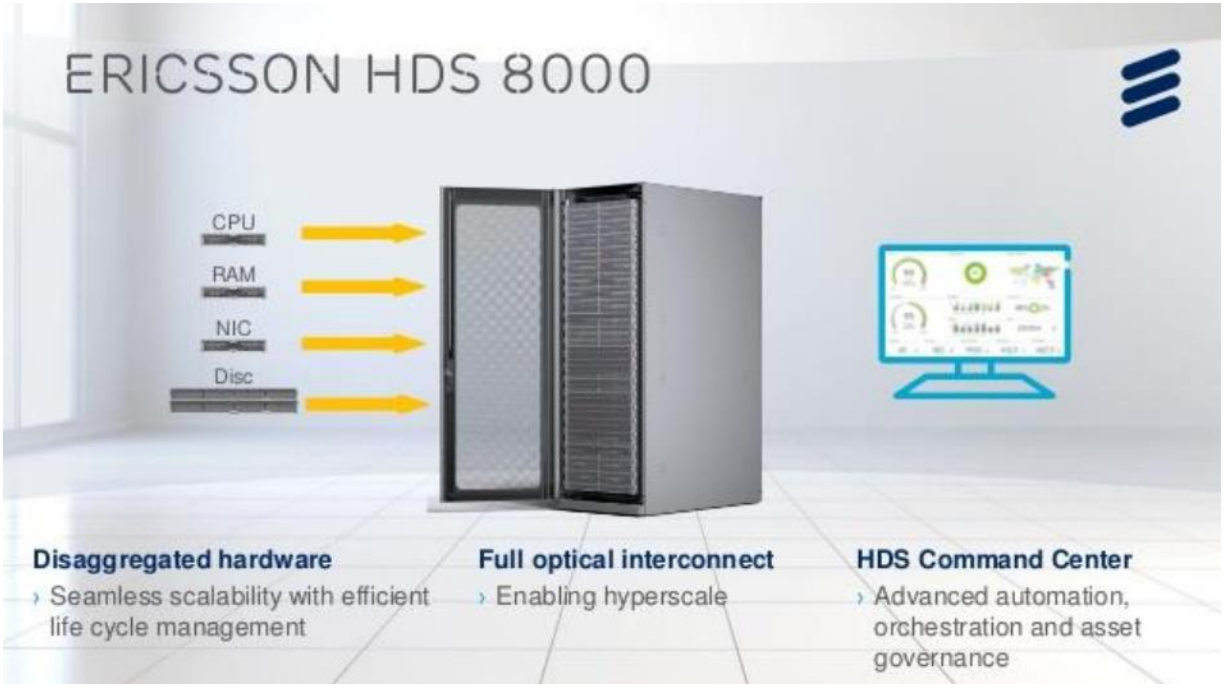
1.2 Opis komponenti sustava

Proizvod prati Intel Rack Scale Design specifikacije. Komponente sustava podijeljene su u logičke cjeline:

- Hardware
    - Šasija i upravljačka jedinica šasije (Chassis Management Unit – CMU),
    - Računalna ladična jedinica (Compute Sled Unit - CSU),
    - Pohrambena ladična jedinica (Storage Sled Unit – SSU) te
    - Mrežna ladična jedinica (Network Sled Unit – NSU).
- Software
    - Command Center Manager (CCM + UI) – Upravljački software + korisničko sučelje,
    - Command Center Agent te
    - Equipment Access Manager (EAM = EAS + EAC) – skup upravljačkih programa za opremu – direktno upravljanje opremom.

Prikazana je podjela po pitanju hardware-a prikazuje samo različite građevne elemente dok se pojedinačni proizvodi u svakoj komponenti mogu pronaći na mrežnim stranicama Ericsson SDI proizvoda.

Prikaz sustava je na Slika 1.3 Pregled HDS 8000 sustava.



Slika 1.3 Pregled HDS 8000 sustava

U nastavku su slikovni prikazi ostalih hardware-skih komponenti kao i upravljačkog centra HDS Command Center Manager koji čini glavno korisničko sučelje, odnosno dio sustava koji kupac vidi i kojim upravlja sustavom.

Slika 1.4 CSU - Compute Sled Unit

Značajke računalne jedinice (CSU):

- Intel procesorski podsustav (dvo-priključni (eng. Dual socket) Intel XEON E5-2600),
- 24 DDR4 priključka do ukupno 1546 GB,
- Intel Fortville mrežni podsustav (XL710),
- Luxtera optički pogon te
- Potrošnja energije 200 – 500 W (ovisno o konfiguraciji).

Slika 1.5 SSU - Storage Sled Unit

Značajke pohrambene jedinice (SSU):

- Optički SAS (Serijski spojeni SCSI): 12 Gb proširnik s dvostrukim priključkom (port),
- Integrirani optički Luxtera primopredajnik,
- Potrošnja energije 150 – 350 W (ovisno o konfiguraciji),
- Podržava diskove : SSD 2.5", HDD 2.5", HDD 3.5", (12x3.5" HDD/SSD, 20x2.5" HDD/SSD) te
- Sav SW se učitava / ažurira preko SAS-a, Ethernet-a ili IPMB.

Slika 1.6 NSU – Network Sled Unit

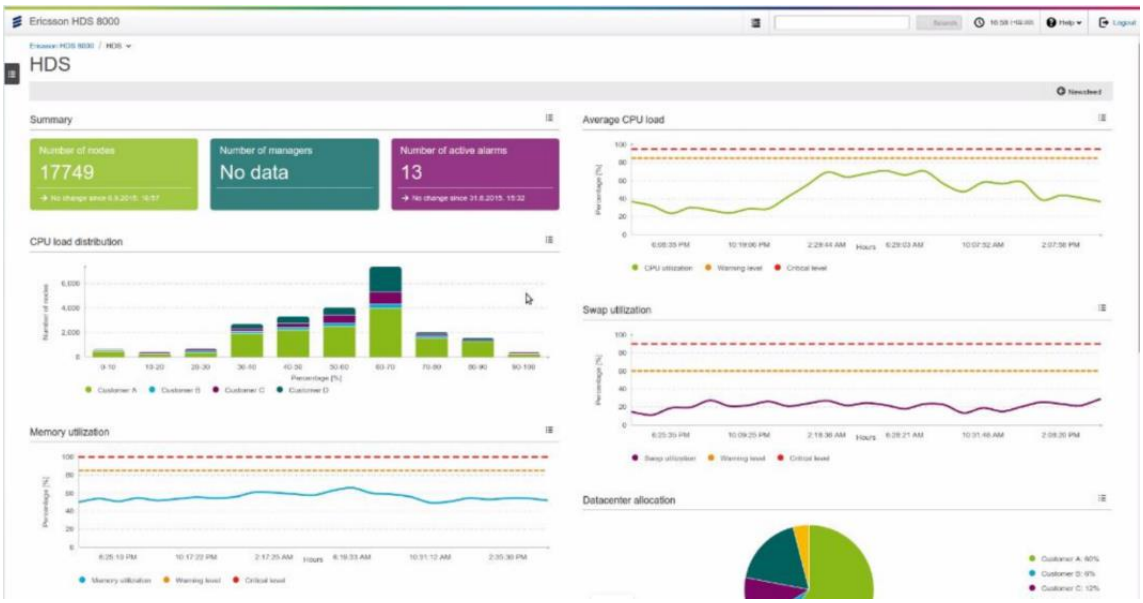
Značajke mrežne jedinice (NSU):

- Podatkovna mreža,
- 16 x 40Gb portova po ladici, ukupno 1680 Mbps,
- Netvisor operativni sustav (SDN) te
- Potrošnja energije: 400 – 1100 W.

Slika 1.7 EAS - Equipment Access Switch

Značajke pristupnog preklopnika za opremu (EAS):

- Upravljačka mreža,
- 48 x RJ45 Ethernet, 10 x modular SFP+ te
- Potrošnja energije: 350W max.



Slika 1.8 CCM - Command Center Manager

Značajke upravljačkog centra (CCM):

- Sadrži nekoliko software-skih komponenti čija je namjena upravljanje Ericsson HDS 8000 sustavom i serverima treće stranke,
- CCM je hardware-ski neovisan tako da može raditi na različitim mašinama (HP, Dell),
- Komponente:
    - CCM korisničko sučelje,
    - CCM jezgreni software,
    - Sparklog te
    - Directive.

- Neke od funkcionalnosti:
    - Skeniranje mreže,
    - Menadžment i upravljanje (postavljanje redoslijeda bootanja, firmware ažuriranje,
    - Prikupljanje metrika te
    - Postavljanje operativnih sustava.

Kombinacija iznad prikazanih komponenti čini jedan tzv. „blok“ opreme. Što se mrežnog dijela tiče koristi se „Spine – Leaf“ (u prijevodu Kičma – List) mrežna arhitektura preklopnika koja se zasniva na reduciranoj Clos mrežnoj arhitekturi. Tako jedan minimalni blok opreme mora imati po jednu CSU, SSU, NSU, EAS jedinicu, a u mrežnom dijelu to je jedan „Spine“ preklopnik i dva „Leaf“ preklopnika.

2. RAZVOJ APLIKACIJE

Unutar SDI komponente skupljaju se informacije s računalnih jedinica, jedinica podatkovne pohrane te mrežnih jedinica – preklopnika (eng. Switch) na način na koji je objašnjeno u uvodu. Podatci koji se skupljaju uključuju postotak iskorištenja procesora – CPU, temperature procesora na pojedinim komponentama, zauzeće diska na različitim virtualnim mašinama, iskorištenje radne memorije – RAM, podatkovni promet po priključku (eng. Port) na preklopnicima, broj okretaja pojedinih ventilatora na preklopnicima, podatkovni promet po virtualnom toku, količina neprenesenih i prenesenih podataka po preklopniku i mnogi drugi.

Nakon što su definirani zahtjevi bilo je potrebno odlučiti u kojoj je tehnologiji izraditi, a izbor je bio sužen na programske jezike Java i JavaScript. Kako je je aplikacija trebala raditi samo dvije stvari, analizirati tekstualnu datoteku i poslati te podatke u bazu podataka, mnoge prednosti koje Java ima nad JavaScript-om su bile zanemarene pa je tako odabran JavaScript. Pored jednostavnosti jezika prednost je bilo već postojeće znanje JavaScript-a te prethodno iskustvo s radom u Node.js tehnologiji što je bilo veliki plus iz perspektive tehničkog duga

2.1 Node.js

Izrada koncepta prvotno je uključivala obradu podataka jednog zapisnika, postotak iskorištenosti procesora. Nakon kratke analize datoteke zapisnika, provedeno je istraživanje kako najbolje obraditi tekstualnu datoteku, počela je izrada aplikacije. Glavna tehnologija je Node.js, okruženje koje dopušta uporabu programskog jezika JavaScript, kojemu je prvotna namjena bila uporaba u Internet preglednicima za prikaz dinamičkih sadržaja, da se izvodi na višestrukim platformama prvotno na poslužiteljskoj strani odnosno van Internet preglednika. To okruženje razvio je inženjer Ryan Dahl, a samo okruženje isprogramirano je u jezicima C i C++. Okruženje razvojnom programeru pruža mogućnosti učitavanja datoteka i rad s istima na znatno jednostavniji način što je u konačnici rezultiralo minimalnim tehničkim dugom (kraće vrijeme razvoja aplikacije i veća dostupnost razvojnih programera

koji poznaju JavaScript). Prvi cilj bio je napraviti prototip parserske funkcije za obradu ovakve datoteke i u sklopu tog cilja trebalo se upoznati s regularnim izrazima, koji su ukratko najmoćniji alat za obradu teksta. Srž funkcionalnosti aplikacije čini zapravo učitavanje tekstualne datoteke u RAM memoriju računala gdje je cijela datoteka predstavljena kao niz znakova (eng. *string*) i na tom nizu znakova raditi operacije čiji su rezultat obrađeni podatci, odnosno strukturirani programski prikaz „sirovog“ teksta. Primjer tekstualne datoteke je na Slika 2.1 Primjer neobrađenog zapisnika iskorištenosti procesora.

```
1  Timestamp: 1594483397056 || Data: Sat Jul 11 18:03:13 CEST 2020LF
2  9LF
3  LF
4  Timestamp: 1594484000246 || Data: Sat Jul 11 18:13:17 CEST 2020LF
5  7.3LF
6  LF
7  Timestamp: 1594484603438 || Data: Sat Jul 11 18:23:20 CEST 2020LF
8  7.7LF
9  LF
10 Timestamp: 1594485206631 || Data: Sat Jul 11 18:33:23 CEST 2020LF
11 2.2LF
12 LF
```

Slika 2.1 Primjer neobrađenog zapisnika iskorištenosti procesora

Aplikacija je strukturirana modularno tako da se svaka funkcionalnost aplikacije može odvojeno izmjenjivati i ažurirati. Glavni dijelovi aplikacije – parseri (dio koji obavlja sintaktičku analizu) odvojeni su u zasebnu pod strukturu – module, a prateći taj uzor tako je odvojen i dio koda koji vrši pripremu podataka, odnosno priprema objekte koji predstavljaju vremenske trenutke, a odvojen je i dio koda koji pruža potporu glavnom dijelu kao što je provjera mape koja sadrži zapisnike, ispravljanje kraja datoteke (provjera da li datoteka završava s dvama **LF** znakovima) i slično. Važno je napomenuti da je ovo CLI (Command Line Interface – Upravljačko sučelje) aplikacija i da se koristi striktno kao zasebna naredba unutar terminala, tako da aplikacija ne prati ni jedan od uzoraka dizajna vezanih za Node.js budući da su ti uzorci orijentirani na web aplikacije i razvoj REST API-ja. Aplikacija strukturalno prati Node.js načelo modularizacije koda odvajajući tako svaku funkcionalnost u zasebnu logičku cjelinu. Iako je modularna, određene cjeline nisu samostalne, posebice parseri koji koriste pomoćne kodove ali je modularnost postignuta razdvajanjem pojedinih parsera i pojedinih funkcionalnosti pomoćnog koda tako da ukoliko je potrebno napraviti

izmjenu u parseru moguće ju je napraviti bez utjecaja na pomoćni kod, ali izmjene u pomoćnom kodu se propagiraju na sav parserski kod pa je te izmjene potrebno preduhitriti izmjenama u parserima.

Struktura koda prikazana je sljedećim spiskom:

- app,
    - src,
        - components.
            - customInflux,
            - parsers,
                - index.js,
                - parseCpu.js,
                - parseDisk.js,
                - parseRam.js,
            - utils,
                - config.json,
                - Global-Type-Definitions.js,
                - globalDebugSwitch.js,
                - index.js,
                - influxHelper.js,
                - utilities.js,
    - main.js i
    - bin.js.

Aplikaciju čine zapravo dvije glavne komponente, parseri (sintaktički analizatori) i pomagala (eng. *utils*). Svaki od tri parsera biti će posebno obrađen u svom potpoglavlju. Nakon obrade parsera i pomoćnog koda poglavlje o razvoju aplikacije biti će zaključeno analizom glavnog dijela programa sadržanog u *main.js* datoteci, datoteka *bin.js* je pomoćna datoteka preko koje se omogućuje pakiranje aplikacije u samostalnu izvršnu datoteku.

2.1.1 Parser zapisnika opterećenja procesora

Ovaj parser je prvi koji je razvijen još u fazi razvitka dokaza koncepta aplikacije. U nastavku je prikaz koda. U prikazima koda biti će izostavljen popratni sadržaj za dokumentaciju.

```
const { basename, parse } = require('path');

const { newLineFix, getLogInfo } = require('../utils');

const parseCpu = function parseCpuFromLogFile(cpuLogFile) {
  // Data object with time-load key-value pairs
  const data = {};
  try {
    // File is read with newLineFix method
    const fileContent = newLineFix(cpuLogFile);

    const regexTimestamp = /(?<=p: )(.*)(?=\x20\x7C)/gmu; // gets timestamps from log
    const regexCpuLoad = /(?<cpuLoad>^\d(\S+|\n))/gm; // get CPU load

    // Get data from logs using regex and assign them
    const timestamps = fileContent.match(regexTimestamp);
    const cpuLoad = fileContent.match(regexCpuLoad);

   // Create new array with numbers instead of strings - enables arithmetic operations on
  // array
    const cpuLoadNumeral = cpuLoad.map((loadPercent) => Number(loadPercent));
    timestamps.forEach((timestamp, cpuload) => {
      data[timestamp] = cpuLoadNumeral[cpuload];
    });
  } catch (e) {
    console.log(e);
  }

  return {
    logData: data,
    logInfo: getLogInfo(basename(cpuLogFile)),
  };
};

module.exports.parseCpu = parseCpu;
```

Slika 2.2 parseCpu.js glavni kod

Na početku datoteke uvoze se potrebne biblioteke za rad, a to su ugrađena biblioteka ***path*** i dodatno razvijena biblioteka s pomoćnim kodom – ***utils***. Iz biblioteke ***path*** koristi se funkcija *basename( )* za dobavljanje baznog imena iz potpunog imena datoteke koje uključuje i putanju do datoteke. Funkcija *parse( )* se koristi za sintaktičku analizu putanje i namijenjena je kao napomena za daljnji razvoj – ne koristi se u trenutnoj verziji parsera. U nastavku se definira glavna funkcionalnost modula *parseCpu.js* i to preko funkcije *parseCpuFromLogFile( )*. Ovdje se vidi pridjeljivanje imena funkciji, jednoznačno spremanju funkcije u varijablu **parseCpu**. Ovo je posebna značajka programskog jezika JavaScript, a omogućuje skrivanje punog imena funkcije. Puno ime funkcije se konvencijski koristi za opis funkcije, dok se dodijeljeno ime koristi za pozivanje funkcije te izvoz iz modula. Na početku funkcije definiramo JavaScript objekt **data** koji će sadržati parove UNIX vremenskog žiga i decimalnog broja koji predstavlja jedno mjerenje, npr: **`{ 1594483396328 : 11.3 }`**.

Primjer neobrađenih podataka izgleda kao u prikazu ispod :

```
Timestamp: 1594483396328 || Data: Sat Jul 11 18:03:13 CEST 2020
11.3
Timestamp: 1594483999517 || Data: Sat Jul 11 18:13:16 CEST 2020
4.3
Timestamp: 1594484602708 || Data: Sat Jul 11 18:23:19 CEST 2020
3.1
```

Nakon definiranog objekta otvaramo tzv. *try-catch* u kojem u dijelu **try** izvodimo glavnu logiku, a pomoću dijela **catch** (hrv. uhvatiti) „hvatamo“ svaku moguću grešku koja može nastati u procesu parsiranja i ispisujemo je u konzolu. Proces parsiranja počinje u **try** bloku učitavanjem sadržaja datoteke zapisnika o opterećenju procesora u radnu memoriju. U ovom parseru je učitavanje obfuscirano pozivom na metodu *newLineFix( )* koja ispravlja ranije spomenute greške sa znakom novog reda u zapisnicima. Za potpuno razumijevanje ovo dijela potrebno se referirati na cjelinu 0. Rezultate metode se sprema u konstantnu varijablue **fileContent**, a u nastavku se definiraju još dvije konstante varijable koje sadrže regularne izraze kojima se opisuje UNIX vremenski žig i postotak iskorištenosti procesora. Regularni izraz spremljen u varijabli **regexTimestamp**, sadržaja `/(?<=p: )(.*)(?=\x20\x7C)/gmu` traži tekst po navedenom uzorku na način da su odgovarajući rezultati tekst između slova „p“

popraćenog znakom „:“ i znakova razmaka i vertikalne crte „ |“. Ako pogledamo u „sirovi“ zapisnik vidimo da se na točno tim pozicijama nalazi vremenski žig. Kako je datoteka učitana sinkronim načinom (cijela datoteka stoji spremljena u RAM računala), tako je moguće bezbrižno primijeniti ovaj način pretrage podataka. Regularni izraz spremljen u varijabli **regexCpuLoad** sadržaja `/(?<cpuLoad>^\d(\S+|\n))/gm` pretražuje datoteku za uzorak kojem odgovara sve što počinje s brojem (^\d) kojeg slijedi bilo koji i bilo koliko znakova (\S+) ili novi red (\n). Uzorke je moguće testirati bilo kojim alatom za regularne izraze. Sada je potrebno definirati varijable u koje će rezultati pretrage biti spremljeni, a to su u primjeru varijable **timestamps** i **cpuLoad**. Te varijable su zapravo nizovi tipa podataka *string*, ali u JavaScript jeziku nije potrebno brinuti o tipovima podataka budući da je jezik dinamičkog tipa odnosno interno na temelju tipa podatka dodijeli svakoj varijabli odgovarajući tip. Na sadržaju datoteke (varijabla **fileContent**) pomoću metode *match( )* se pronalaze iznad definirani uzorci i dodjeljuju se varijablama **timestamps** za vremenske žigove i **cpuLoad** za opterećenje procesora. Iz varijable **cpuLoad** vrši se mapiranje u novu varijablu – niz **cpuLoadNumeral** koja je zapravo istog sadržaja, ali za razliku od **cpuLoad** varijable sadrži vrijednosti u brojevnom obliku što je potrebno pri spremanju u bazu podataka. Mapiranje se vrši pomoću metode *map( )* koja spada pod funkcije višeg reda u JavaScriptu, a toj funkciji se prosljeđuje druga funkcija tzv. **callback** (u prijevodu : poziv nazad), ali je i ta funkcija zapisana u obliku streličastih funkcija (eng. *arrow functions*), gdje je izostavljena ključna riječ ***function*** i umjesto nje se koriste zagrade popraćene znakovima jednakosti i veće od, primjer : () => { „tijelo funkcije“ }. U obične zagrade se unosi definirana ili privremena varijabla (u JavaScriptu privremene varijable nije potrebno defnirati), a u tijelu funkcije se vrše radnje nad tom varijablom, no ukoliko se poziva samo jedna dodatna funkcija moguće je izostaviti tijelo funkcije i direktno izvršiti manipulaciju kao što je u iznad navedenom primjeru koda s pozivom na konstruktorsku metodu *Number( )*. Metoda *map( )* vraća novi niz koji se onda dodijeli varijabli **cpuLoadNumeral**. Sada u igru dolazi na početku definirani objekt **data**. Na nizu podataka spremljenom u varijablu **timestamps** poziva se metoda *forEach( )* koja je također funkcija višeg radi, a obavlja funkcionalnost sličnu „for“ petlji s glavnim izuzetkom da se ne prati red elemenata u nizu, ali radnja bude obavljena za svaki. Redoslijed u objektu nije bitan bitno je samo kreirati objekt s parovima vremenskog žiga i odgovarajućim opterećenjem procesora, a redoslijed se definira unutar baze podataka na temelju žiga budući da baza automatski preslaže podatke po redoslijedu događaja. U metodi *forEach( )* se poziva streličasta anonimna funkcija u kojoj se koriste dvije privremene

(nedefinirane varijable) koje na temelju svog položaja u pozivu imaju dodijeljena značenja, pa tako varijabla **timestamp** ima značenje elementa iz niza tj. varijable **timestamps**, a varijabla **cpuload** ima značenje indeksa (broj) tj. pozicije varijable **timestamp** u nizu **timestamps**. Za svaki element niza **timestamps** se objektu **data** dodaje ključ – vrijednost par (eng. *key-value pair*). Ključ čini vremenski žig, a vrijednost brojčana vrijednost opterećenja procesora. Ključ se uzima iz niza **timestamps**, a vrijednost iz niza **cpuLoad** preko indeksa elementa **timestamp**. Važno je da su nizovi jednake duljine u protivnom će doći greške koja će biti ispisana u bloku **catch**. Metoda vraća anonimni objekt koji sadrži ključeve **logData** s vrijednošću koja je objekt **data** i **logInfo** kojem je vrijednost anonimni objekt vraćen iz funkcije *getLogInfo( )*. Ova funkcija spada pod domenu pomoćnog koda gdje se nalazi potpuna definicija metode. Na kraju izvezemo funkcionalnost modula na način da eksportiramo željene funkcije, to je u ovom slučaju samo **parseCpu.** Izvoz se vrši standardnom Node.js sintaksom **module.exports**.

2.1.2 Parser zapisnika iskorištenja radne memorije

Idući zapisnik sadrži podatke o iskorištenju radne memorije nadzirane komponente, pa kako je primjer iznad bio s komponente EAC (Equipment Access Manager) koji je virtualna mašina, tako će i u ovom primjeru biti prikazan zapisnik s iste komponente. Prikaz na slici ispod.

```
Timestamp: 1594483393373 || Data: Sat Jul 11 18:03:13 CEST 2020LF
3944LF
830LF
LF
Timestamp: 1594483993384 || Data: Sat Jul 11 18:13:13 CEST 2020LF
3944LF
773LF
LF
```

Slika 2.3 Neobrađeni zapisnik iskorištenja radne memorije

Iz prikaza se lako da zaključiti da nam gornji broj u mjerenju (3944) prikazuje ukupnu radnu memoriju što je na ovoj virtualnoj mašini 4 GB, a donji broj predstavlja iskorištenu memoriju. Princip koda za parser je sličan onome za parser opterećenja procesora s iznimkom što ovdje imamo dva podatka po mjerenju. Kod je priložen na sljedećoj stranici.

```
const { readFileSync } = require('fs');
const { getLogInfo } = require('../utils');

const parseRam = function parseRamFromLogFile(ramLogFile) {
  const fileContent = readFileSync(ramLogFile, { encoding: 'utf8' });

  const logArray = fileContent.split('\n\n');
  const regexTimestamp = /(?<=p: )(.*)(?=\x20\x7C)/gmu; // gets timestamps from log

  const ramData = [];

  for (let i = 0; i < logArray.length - 1; i += 1) {
    const logEntryObj = {};
    const logEntryLines = logArray[i].split('\n');
    const timestamp = logEntryLines[0].match(regexTimestamp)[0];
    const totalRam = logEntryLines[1];
    const usedRam = logEntryLines[2];
    logEntryObj.timestamp = timestamp;
    logEntryObj.totalRam = totalRam;
    logEntryObj.usedRam = usedRam;
    ramData[i] = logEntryObj;
  }

  if (ramData[0].totalRam === undefined || ramData[0].usedRam === undefined) {
    console.error(
      `ERROR : Cannot parse file ${ramLogFile}, check file manually for errors and
       inconsistencies`
       .red.bold
    );
    return;
  }
  return {
    logData: ramData,
    logInfo: getLogInfo(ramLogFile),
  };
};

module.exports.parseRam = parseRam;
```

Slika 2.4 parseRam.js glavni kod

U kodu vidimo sličnu strukturu prijašnjem parseru s iznimkom što je u ovom kodu eksplicitno učitana datoteka u odnosu na obfuscirani raniji primjer. Na početku datoteke uvoze se dvije glavne datoteke i iz njih se uzimaju funkcije *readFileSync( )* za sinkrono učitavanje datoteke i *getLogInfo( )* što je već spomenuta osobna implentacija za dobavljanje

informacija o datoteci a opis metode je u poglavlju o pomoćnom kodu. Definirana je funkcija *parseRamFromLogFile( ),* kojoj je dodijeljen jednostavan naziv **parseRam**. Funkcija kao i svaki drugi parser prima datoteku s odgovarajućim sadržajem. Sadržaj datoteke učitava se u varijablu **fileContent**, pomoću metode (funkcije) *readFileSync( )*, a iz tog sadržaja se kreira niz elementa koji se sastoje od vremenskog žiga i mjerenja (dvije vrijednosti RAM-e). Niz se kreira tako da se sadržaj razdvaja po duplom znaku novog reda „\n\n“, te se nakon toga „for“ petljom prolazi kroz svako mjerenje. U ovom slučaju koristi se standardna „for“ petlja, a umjesto pretrage regularnim izrazom kroz cijelu datoteku koristi se samo regularni izraz po kojem se pronalazi i dodjeljuje vremenski žig. Podaci su spremljeni u niz – polje (eng. array) pod nazivom varijable **ramData**. U nizu unosa u varijabli **logArray** se pri svakoj iteraciji kreira novi privremeni objekt imena **logInfoObj** (objekt koji predstavlja informacije zapisnika jednog unosa). Dalje se po znaku novog reda „\n“ unos u trenutnoj iteraciji rastavlja na pojedine linije koje su u svakoj iteraciji spremljene u privremeni niz **logEntryLines**. Na svakoj liniji zapisa, odnosno svakom elementu spomenutog niza, vrši se izvlačenje informacija, pa se tako iz prve linije preko regularnog izraza uzme vremenski žig i spremi u varijablu **timestamp**, druga linija se spremi u varijablu **totalRam**, a treća linija u varijablu **usedRam**. Vrijednost ovih varijabli se spremi u istoimene ključeve objekta **logInfoObject** koji se, doda u niz **ramData** na indeks jednak trenutnoj iteraciji petlje niza **logArray**. Umjesto **try-catch** bloka ovdje provjerimo da li su vrijednosti na prvom elementu **ramData** nedefinirane i ako jesu vratimo upozorenje bez prekida rada aplikacije. Na kraju se kao rezultat obrade vraća anonimni objekt s ključem **logData** koji sadrži podatke u obliku vrijednosti **ramData** i identičan ključ **logInfo** kao u parseru opterećenja procesora.

2.1.3 Parser zapisnika iskorištenje memorije diska

Zadnji u nizu parsera vrši obradu najsloženijeg zapisnika, jedini složeniji zapisnik vezan je za obradu podataka s preklopnika koji radi ograničenog opsega rada neće biti izložen u radu. Primjer zapisnika je na Slika 2.5 Neobrađeni zapisnik iskorištenja diskovnog prostora.

```
Timestamp: 1594483993055 || Data: Sat Jul 11 18:13:13 CEST 2020
Filesystem                                     1K-blocks    Used Available Use% Mounted on
udev                                             1990048       0   1990048   0% /dev
tmpfs                                             403924   41296    362628  11% /run
/dev/mapper/HDS--vg--eac-root                    6426992 1423336   4637476  24% /
tmpfs                                            2019604       4   2019600   1% /dev/shm
tmpfs                                               5120       0      5120   0% /run/lock
tmpfs                                            2019604       0   2019604   0% /sys/fs/cgroup
/dev/vda1                                         472036   74372    373293  17% /boot
/dev/mapper/HDS--vg--eac-home                     920472    3136    850984   1% /home
/dev/mapper/HDS--vg--eac-tmp                      920472    1376    852744   1% /tmp
/dev/mapper/HDS--vg--eac-var                     2745100  413112   2165692  17% /var
/dev/mapper/HDS--vg--eac-var_tmp                  920472    1236    852884   1% /var/tmp
/dev/mapper/HDS--vg--eac-var_backups_hds         2745100    4516   2574288   1% /var/backups/hds
/dev/mapper/HDS--vg--eac-var_lib_hds             1840976    2952   1721700   1% /var/lib/hds
/dev/mapper/HDS--vg--eac-var_log                 2745100   46212   2532592   2% /var/log
/dev/mapper/HDS--vg--eac-var_lib_hds_inventory   2745100    4400   2574404   1% /var/lib/hds/inventory
/dev/mapper/HDS--vg--eac-var_lib_hds_config       920472   63360    790760   8% /var/lib/hds/config
/dev/mapper/HDS--vg--eac-var_log_audit            920472    5548    848572   1% /var/log/audit
tmpfs                                             403924       0    403924   0% /run/user/1000
total                                           31164988 2084856  27576313   8% -

```

Slika 2.5 Neobrađeni zapisnik iskorištenja diskovnog prostora

Ovdje se vidi znatno složenija struktura zapisnika za razliku od zapisnika opterećenja procesora i iskorištenja radne memorije. Korišteni pristup svodi se na dijeljenje cijelog zapisnika na unose kao što je prikazani na slici iznad. Dalje se zapisnik dijeli na pojedine linije, a linije se dijele po stupcima. Iako je duljina svakog elementa različita svi su elementi razdvojeni minimalnom jednim znakom razmaka što omogućuje jednostavno razdvajanje elemenata. Struktura zapisnika uvijek ostaje ista (raspored stupaca) pa nije bilo potrebe za regularnim izrazima van već korištenog izraza za vremenski žig. Kako se kroz ovaj zapisnik radi razdvajanje pojedinih unosa, pa unutar tog unosa opet na linije algoritam traje duže, ali na malim zapisnicima do par megabajta razlika se ne osjeti. Kod za parser iskorištenja diska koristi iste dvije biblioteke kao i parser za iskorištenje radne memorije, biblioteku za čitanje datoteka i biblioteku za skupljanje informacija. Pregled koda priložen je na idućoj stranici, ali bez uključenih biblioteka, kod se nastavlja na stranicu 22.

```
const parseDisk = function parseDiskFromLogFile(diskLogFile) {
  // Blocks execution until file is read and stored
  const fileContent = readFileSync(diskLogFile, { encoding: 'utf8' });
  // Split at double newline
  const logArray = fileContent.split('\n\n');
  //  datalineArray
  const datalineArray = [];
  // Entire log data
  const dfArray = [];
  // Regex for timestamp
  const regexTimestamp = /(?<=p: )(.*)(?=\x20\x7C)/gm;
  for (let i = 0; i < logArray.length - 1; i += 1) {
    // Last element is new line, hence -1
    // Create new instance of log entry object on each iteration
    const logEntryObj = {};
    // Split each entry on newline
    const logEntryLines = logArray[i].split('\n');

    // Timestamp is the first matched element
    const timestamp = logEntryLines[0].match(regexTimestamp)[0];

    // Set timestamp key of log entry object to matched timestamp
    logEntryObj.timestamp = timestamp;
    // Remove first two lines of log entry
    logEntryLines.splice(0, 2);
    for (let j = 0; j < logEntryLines.length; j += 1) {
      // Split each line by space and remove space characters
      const splitLine = logEntryLines[j]
        .split(' ')
        .filter((element) => element !== '');
      // Create new instance of info object on each iteration
      // Create key - value pairs for dataline
      const dataline = {
        filesystem: splitLine[0],
        _1k_blocks: splitLine[1],
        used: splitLine[2],
        available: splitLine[3],
        use: splitLine[4].replace('%', ''),
        mountedOn: splitLine[5],
      };
      datalineArray[j] = dataline;
    }
    logEntryObj.data = datalineArray;
    dfArray[i] = logEntryObj;
```

Slika 2.6 parseDisk.js glavni kod - DIO 1

```
  return {
    logData: dfArray,
    logInfo: getLogInfo(diskLogFile),
  };
};

module.exports.parseDisk = parseDisk;
```

Slika 2.7 parseDisk.js glavni kod – DIO 2

Kao i prva dva parsera kod počinje s učitavanjem datoteke, gdje se kreira niz unosa kao što je prikazan na Slika 2.5 Neobrađeni zapisnik iskorištenja diskovnog prostora. Niz se kreira razdvajanjem po dvostrukoj novoj liniji. Varijable **dataLineArray** i **dfArray** koriste se za spremanje pojedinih podatkovnih linija unosa u prvu, a u drugu se spremaju objekte koji predstavljaju pojedini unos. Obrada podataka kreće ulazom u niz **logArray** kroz koji se prolazi „for“ petljom i u svakoj iteraciji se kreira novi objekt **logEntryObj**, a nakon njega svakom iteracijom kreira se novi niz koji predstavlja linije jednog unosa, a to je varijabla **logEntryLines** koju se dobije tako što se jedan unos rastavi po znaku novog reda „\n“. Na prvoj liniji preko regularnog izraza, te se taj žig odmah pridruži ključu **timestamp** objekta **logEntryObj**. Onda slijedi obrada pojedine linije koja se rastavlja po znaku razmaka, a nakon toga se elementi spremaju u niz kao varijabla **splitLine** pozivom na funkciju višeg reda – *filter( )*. Nakon što su podaci rastavljeni, formira se objekt koji predstavlja jednu podatkovnu liniju na način da je ključ u objektu ime stupca, a vrijednost vezana za ključ je element niza **splitLine**, na odgovarajućem indeksu stupca, pa je tako vrijednost na prvom mjestu – indeks broj 0 vezana za stupac „filesystem“ što je prvi stupac u neobrađenom zapisniku. Na isti način se kreira ostatak objekta, a svaki se takav objekt dodaje u niz **dataLineArray** kreiran na početku, svaki element ovog niza je jedan objekt koji predstavlja određenu liniju jednog unosa zapisnika pa je tako **dataLineArray[0]** uvijek prva linija pojedinog unosa. Ovaj proces se ponavlja za svaki unos što se vidi u dijelu koda koji veže niz **dataLineArray** kao vrijednost ključa **data** objekta **logEntryObj**. Cjelokupni zapisnik predstavljen je na kraju nizom takvih objekta – **dfArray**. Po uzoru na prva dva parsera funkcija vraća anonimni objekt kojemu je **dfArray** vrijednost ključa **logData**, a ključ **logInfo** vrijednost dobija istom logikom kao i ostali parseri što je objašnjeno u idućem segmentu rada.

2.1.4 Pomoćni kod

Pomoćni kod predstavlja kod koji se koristi kroz ostatak aplikacije, do sada je najviše spominjan dio koda koji uzima informacije o datoteci preko funkcije *getLogInfo( )*. Pomoćni kod sastoji se od sljedećih modula :

- globalDebugSwitch.js
- Global-Type-Definitions.js
- index.js
- influxHelper.js
- utilities.js

Od navedenih modula **globalDebugSwitch.js** sadrži samo globalnu varijablu koja se koristi kao globalni prekidač koji omogućuje ispis grešaka u konzolu, **Global-Type-Definitions.js** sadrži tipove podataka koji služe kao potpora generaciji dokumentacije, a modul **indeks.js** služi samo kao ulazna točka u mapu tako da će se ovaj dio rada baviti glavnim pomoćnim modulima **influxHelper.js** i **utilities.js**. Od funkcija prikazan je samo kod funkcije *getLogInfo( )* kako bi se smanjio opseg rada budući da se u pomoćnom dijelu dosta koda ponavlja s malim izmjenama, a ostale funkcije su nabrojane sa svojim pripadajućim opisima.

Pomoćni modul **utilities.js** od pomoćnih metoda sadrži:

- convertPostFix metodu za pretvorbu jedinica s postfiksom u cijeli broj,
- fixNewLine metodu za ispravke određenih datoteka s neispravnim završetkom,
- getHostId metodu za dobavljanje ID-a domaćina iz imena datoteke,
- getHostType metodu za dobavljanje tipa domaćina iz imena datoteke,
- getLogType metodu za dobavljanje tipa zapisnika iz imena datoteke,
- getBlockId metodu za dobavljanje ID-a bloka opreme iz popisa zapisnika,
- setBlockId metodu za postavljanje ID-a bloka opreme u informacije o zapisniku,
- checkFolder metodu za provjeru mape koja sadrži zapisnike,
- sortLogsIntoArrays metodu za razvrstavanje zapisnika u nizove te naravno
- getLogInfo metodu koja poziva tri druge metode za skupljanje informacija.

Kada se metoda *getLogInfo( )* pozove ona pozove metode *getHostType( ), getHostId( ), getLogType( ),* te postavi bazno ime datoteke. Kod metode prikazan je u nastavak s izostatkom implentacija pojedinih metoda koje poziva.

```
const getLogInfo = function getCompleteLogInfoFromFileName(fileName) {
  const fileBasename = basename(fileName);
  const hostType = getHostType(fileBasename);
  const hostId = getHostId(fileBasename);
  const logType = getLogType(fileBasename);

  return {
    hostType,
    hostId,
    logType,
    blockId: '',
  };
};
```

Slika 2.8 Implementacija metode *getLogInfo( )*

Dakle u parserima se pozivom ove metode ključu **logInfo** dodjeljuje objekt s podacima o zapisniku koji uključuje tip domaćinske komponente, ID domaćina komponente, tip zapisnika, te ID bloka opreme koji je zadan kao prazan, ali se postavlja pozivom na metode *setBlockId( ).*

Pomoćni modul **influxHelper.js** sadrži glavnu funkcionalnost modula unutar metode *createPoints( )* koja kreira objekte koji predstavljaju mjerenje u određenom vremenskom trenutku, dakle metoda kreira objekte koji su ovaj put prilagođeni objekti pojedinih parsera, a prilagođeni su za slanje u bazu kako to definira klijentska biblioteka baze podataka InfluxDB.

2.2 InfluxDB

Odabrana baza podataka je InfluxDB koja vremenski-zasnovana baza podataka što znači da je primarni ključ po kojem se podatci identificiraju vrijeme, odnosno UNIX vremenski žig koji se dosad izdvajao iz svakog pojedinog unosa.To je baza podataka vremenskih serija otvorenog koda (TSDB) koju je razvio InfluxData. Napisana je u programu Go i optimizirana za brzo pohranjivanje i dostupnost podataka o vremenskim serijama s visokom raspoloživošću u poljima kao što su nadzor operacija, mjerni podaci aplikacija, podaci senzora za Internet stvari i analitika u stvarnom vremenu. Baza je sadržana sama u sebi što

znači da ne ovisi ni o kakvom vanjskom software-u. Ima svoj jezik za pristupanje podacima sličan SQL-u, ali se u izradi aplikacije nije koristio jezik direktno, već se koristila klijentska biblioteka za Node.js koja značajno pojednostavnjuje spajanje na InfluxDB server, stvaranje novih baza podataka, dodavanje mjerenja, unos podataka i ostale operacije. Spajanje na bazu podataka kodom opisano je u poglavlju 3.

2.3 Grafana

Aplikacija je uz bazu imala ostvaren samo jedan cilj – spremanje obrađenih podataka za kasniju analizu, ali i dalje ostaje problem kvalitetne vizualizacije podataka. Za vizualizaciju je iskorištena Grafana koja je web aplikacija neovisna o platformi što je točno ono što je traženo u okruženju u kojem se aplikacija razvijala – različiti zaposlenici koriste različite operativne sustave što je potpuno nebitno iz razlog što se Grafana kao web aplikacija izvršava u pregledniku. Kao i InfluxDB Grafana je napisana u programskom jeziku i otvorenog je koda, a podržava spajanje na razne vrste baza podataka kako vremenske tako i relacijske. Grafana koristi koncept upravljačkih ploča gdje je po potrebi moguće kreirati različite vrste ploča, a ukoliko je potrebno napraviti nove slične ploče Grafana podržava izvoz i uvoz ploča u poznatim formatima za prijenos konfiguracijskih datoteka potput .xml i .json formata. Prikaz upravljačkih ploča biti će prikazan u poglavlju 3.

2.4 Dokumentacija

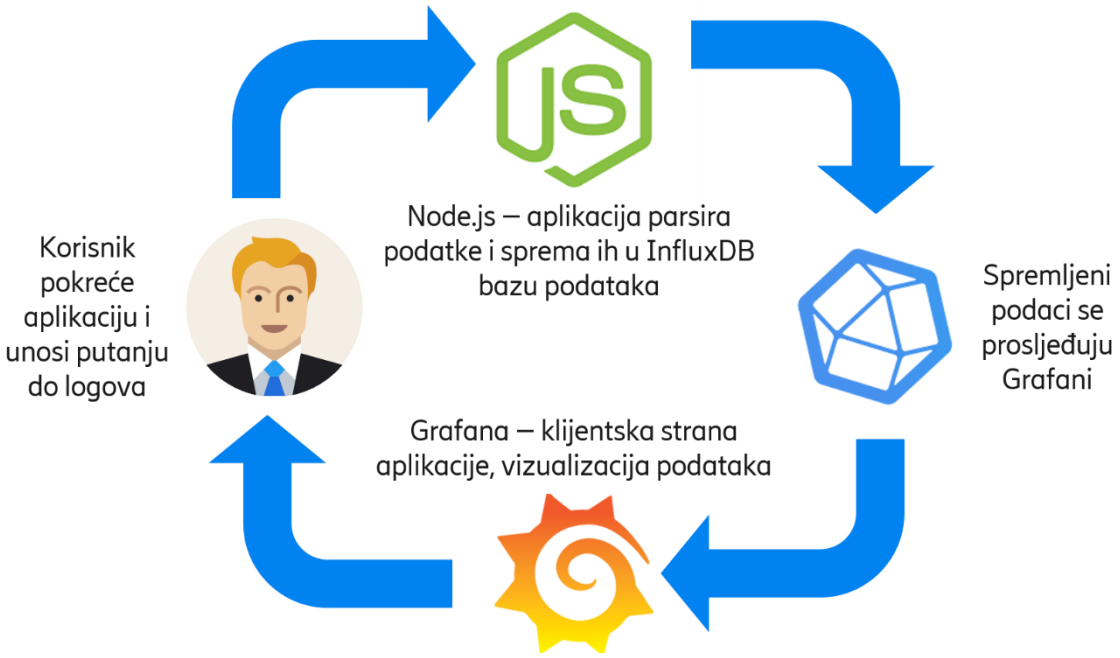
Kao neizostavan dio svakog projekta i ovaj projekt popraćen je dokumentacijom koja se generira iz opisa pisanih u jeziku za označavanje JSDoc. JSDoc funkcionira na način da se pišu anotacije iznad pojedinih metoda. Dokumentacija se generira naredbom „jsdoc“.

```
/**
 * Converts postfix in data size, e.g. from 1.43K to 1430
 *
 * @param {string|number} size
 * Data size written with letter postfix { K | M | G | T | P } or plain number
 * @returns {string|number}
 * Converted data size, or if result is NaN the function returns inputed string
 */
```

Slika 2.9 Primjer uporabe JSDoc anotacija

3. GLAVNI DIO I UPORABA

Logika rada aplikacije sadržana je u modulu **main.js** koji je ulazna točka u aplikaciju, a ranije opisane funkcije parsera su radi spomenute modularnosti odvojene u svoje module kako bi se mogle odvojeno ažurirati bez da se utiče na glavni kod aplikacije.



Slika 3.1 Prikaz toka načina rada aplikacije

3.1 Glavni dio programa – main.js i bin.js

U datoteci **main.js** definira se glavna funkcionalnost aplikacije. Aplikacija pri pokretanju od korisnika zahtjeva unos putanje do mape koja sadrži datoteke sa zapisnicima, nakon toga provjeri da li je mapa uredu na način da pregleda da li mapa sadrži određene datoteke (ukoliko je jedan naziv promijenjen ili je format datoteke neočekivan znači da je došlo do greške i aplikacija će odlučiti koje od datoteka će obraditi a koje ne). Kako je ovo ulazna točka u aplikaciju u njoj su uvezene sve potrebne biblioteku za ostvarivanje glavne funkcionalnosti uključujući i sve prethodno razvijane funkcije i komponente poput parsera i pomoćnih modula. Uz navedene u aplikaciji su u glavnom dijelu uključena još neka pomagala poput praćenja napretka parsiranja, biblioteka za manipulaciju međuspremnikom u konzoli za prikaz pitanja korisniku i slične. Kod glavnog modula **main.js** će zbog veličine biti razložen u nekoliko dijelova te objašnjen nakon čega će u narednom potpoglavlju biti objašnjena uporaba same aplikacije kao i prikaz krajnjeg rezultata.

```
const prompts = require('prompts');
const fs = require('fs');
const path = require('path');
const chalk = require('chalk');
const progress = require('cli-progress');
const logger = require('single-line-log');

const {
  parseCpu,
  parseDisk,
  parseRam,
  parseNru,
} = require('./components/parsers');
const {
  checkFolder,
  sortLogsIntoArrays,
  createPoints,
  setBlockId,
} = require('./components/utils');
const {
  influxCPU,
  influxRAM,
  influxDF,
  influxNRUPortStats,
  influxNRUVflowStats,
  influxNRUSwitchStatus,
  influxNRUSystemStats,
  create,
} = require('./components/utils/influxHelper');
```

Slika 3.2 Biblioteke / moduli u glavnom dijelu aplikacije

Na slici je prikazan cijeli ekosustav biblioteka koje se koriste u glavnom dijelu aplikacije, počevši od biblioteke **prompts** koja omogućuje postavljanje upita korisniku, te biblioteka **fs** za podatkovni sustav i biblioteke **path** koja služi za manipulaciju putanjama. Biblioteke **chalk**, **progress** i **logger** koriste se za bojanje teksta u konzoli, trake napretka i stvaranja zapisnika same aplikacije, ali se one u trenutnoj verziji ne koriste već su predviđene za daljnji razvoj aplikacije. Nakon toga vidimo uvoz različitih metoda iz prethodno spomenutih modula parsera, i pomagala koje se uvoze sintaksom destrukturiranja objekata. Svaki modul ponaša se kao jedan objekt koji u sebi ima izvezene metode koje se ovom sintaksom izvezu iz tog objekta što nam omogućuje uvoz samo određenim metoda iz pojedinog modula / biblioteke. Pored uvezenih metoda uvezeni su i objekti s prefiksom **influx** koji predstavljaju veze s bazom podataka i u sebi sadrže definirane podatkovne sheme za podatke koji će se

tom vezom spremati u bazu. Ovo je naročito korisno zbog toga što će se veza za određenu vrstu zapisnika prekinuti ukoliko je došlo do greške u obradi podataka te u bazu neće biti upisani krivi podatci. Nakon uvoza svih potrebnih resursa slijedi definicija glavne funkcije (aplikacije).

```
const main = async function mainApplication() {
  const cpuPoints = [];
  const ramPoints = [];
  const dfPoints = [];
  let portStatsPoints = [];
  let vflowStatsPoints = [];
  let switchStatusPoints = [];
  let systemStatsPoints = [];
  let logsPath;
  let contents;
  await prompts({
    type: 'text',
    name: 'path',
    message: 'Enter path to log files',
    validate: (pathToLogs) => {
      logsPath = pathToLogs;
      try {
        contents = fs
          .readdirSync(pathToLogs)
          .filter(
            (file) =>
              !file.includes('.html') &&
              !file.includes('.zip') &&
              file !== 'amcharts'
          );
        const isOk = checkFolder(contents);

        if (isOk) {
          return true;
        }
        return 'Path contains no logs with CmdMonitor substring';
      } catch (e) {
        return false;
// 'Error with file path, please check your delimiters\n Backslash
//  \\ for Windows and forwardslash / for Linux';
      }
    },
```

Slika 3.3 Glavna aplikacija - DIO 1

U prvom dijelu aplikacije definiraju se varijable u koje se spremaju točke mjerenja, putanja do zapisnika i sadržaj mape zapisnika. Nakon toga se od korisnika traži unos putanje do zapisnika koja se provjerava, te ako je sve uredu s putanjom, aplikacija nastavlja dalje.

```
try {
  const shortContents = contents;
  contents = contents.map((log) => path.join(logsPath, log));
  const logs = sortLogsIntoArrays(contents);
  logs.RAM = logs.RAM.filter((ramLog) => !ramLog.includes('CMU'));
  const parsedCpuLogs = [];
  const parsedRamLogs = [];
  const parsedDfLogs = [];

  if (logs.CPU.length !== 0)
    logs.CPU.forEach((cpuLog) => {
      const parsedLog = parseCpu(cpuLog);
      setBlockId(parsedLog, shortContents);
      parsedCpuLogs.push(parsedLog);
    });
  if (logs.RAM.length !== 0)
    logs.RAM.forEach((ramLog) => {
      const parsedLog = parseRam(ramLog);
      setBlockId(parsedLog, shortContents);
      parsedRamLogs.push(parsedLog);
    });
  if (logs.DF.length !== 0)
    logs.DF.forEach((dfLog) => {
      const parsedLog = parseDisk(dfLog);
      setBlockId(parsedLog, shortContents);
      parsedDfLogs.push(parsedLog);
    });
```

Slika 3.4 Glavna aplikacija - DIO 2

U drugom dijelu koda zapisnici se sortiraju iz mape u za to definirane nizove, za svaki niz zapisnika se provjerava postoji li barem jedan zapisnik, te ako postoji pozivaju se odgovarajuće parserske metode, postavlja se ID bloka oprema na parsirani zapisnik te se zapisnik dodaje na popis parsiranih zapisnika. Na Slika 3.6 Glavna aplikacija - DIO 3 prikazan je zadnji dio koda glavne aplikacije u kojem završava **try-catch** blok koda, a glavna funkcionalnost se izvozi za uporabu u datoteci **bin.js**. Iz prikaza koda glavnog dijela aplikacije izbačeni su dijelova koda vezani za preklopnike budući da su preklopnici van opsega rada, a uz to je smanjen opseg prikazanog koda. Parseri za preklopnike prate iste principe iznešene u prikazima kodova za parsere procesora, rama i diska.

```
    influxCPU
      .writePoints(cpuPoints, { precision: 'ms' })
      .then(console.log('CPU points written to DB'));
    influxRAM
      .writePoints(ramPoints, { precision: 'ms' })
      .then(console.log('RAM points written to DB'));
    influxDF
      .writePoints(dfPoints, { precision: 'ms' })
      .then(console.log('DF points written to DB'));
  } catch (err) {
    console.log(
      chalk.yellow(
        "Ooops... something went wrong - I'm an error from main application"
      )
    );
    console.log(err);
  } finally {
    //
  }
};

exports.main = main;
```

Slika 3.6 Glavna aplikacija - DIO 3

Nakon izvoza glavne funkcije aplikacije ona se uvozi u modul **bin.js**. Prikaz sadržaja modula je na slici ispod.

```
#!/usr/bin/env node
const { main } = require('./src/main');

// console.log('This is a standalone exec test');
main();
```

Slika 3.5 bin.js modul

Ovaj modul je pomoćni modul za pakiranje Node.js aplikacije u samostalnu izvršnu datoteku neovisnu o vanjskim bibliotekama, te se uz pomoć ovog modula i modula koji se poziva naredbom **pkg** stvaraju izvršne datoteke za Windows, Linux i macOS operativne sustave.

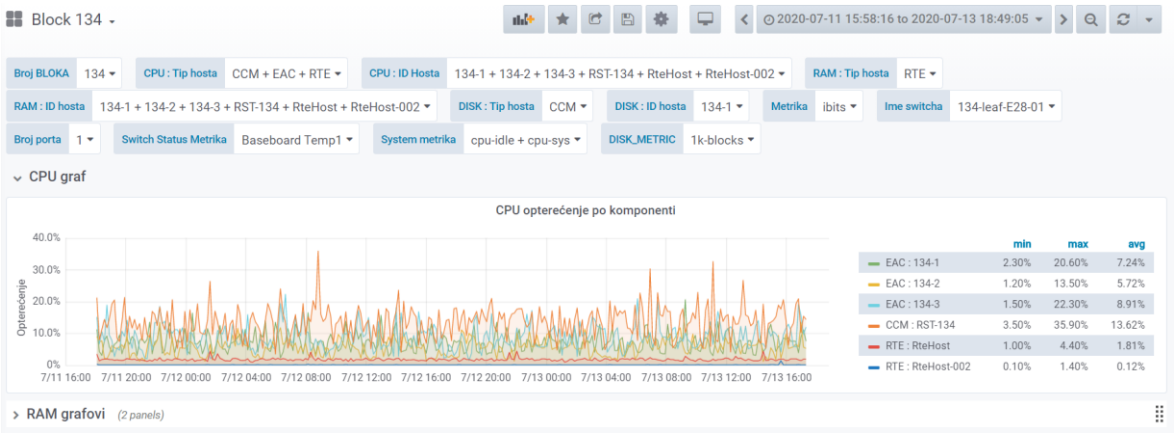
3.2 Uporaba

Uporaba aplikacija je grafički prikazana na Slika 3.1 Prikaz toka načina rada aplikacije, ali je ovaj dio teksta namijenjen za opis upotrebe aplikacije u stvarnom radnom okruženju u kojem je InfluxDB poslužitelj baze pokrenut na centralnom poslužitelju, a korisnik je spojen na lokalnu ili centralnu instalaciju Grafane. Kada korisnik u terminalu pokrene aplikaciju (prikazani način pokretanja je preko prosljeđivanja skripte Node.js – u), dovoljno je da unese putanju do zapisnika nakon čega će aplikacija obaviti parsiranje, spajanje na bazu i spremanje podataka. Rezultat naredbe izgleda kao na slici ispod.

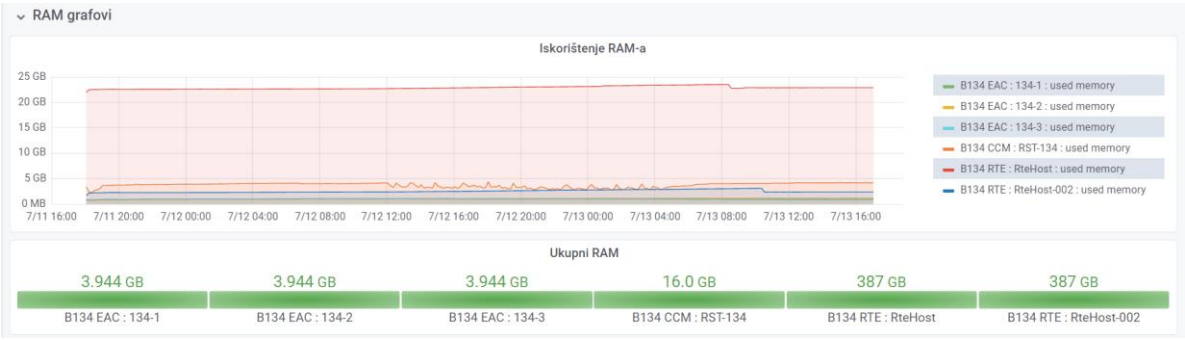
```
PS C:\Users\emamstj\Documents\Code\LogParser\app\src> node .\main.js
✓ Enter path to log files ... C:\Users\emamstj\Documents\Code\LogParser\logs\NOVI_ST_2_14_1
CPU points written to DB
RAM points written to DB
DF points written to DB
PORT Stats written to DB
VFLOW Stats written to DB
SWITCH Status written to DB
SYSTEM Stats written to DB
PS C:\Users\emamstj\Documents\Code\LogParser\app\src>
```

Slika 3.7 Rezultat pokretanja aplikacije u terminalu

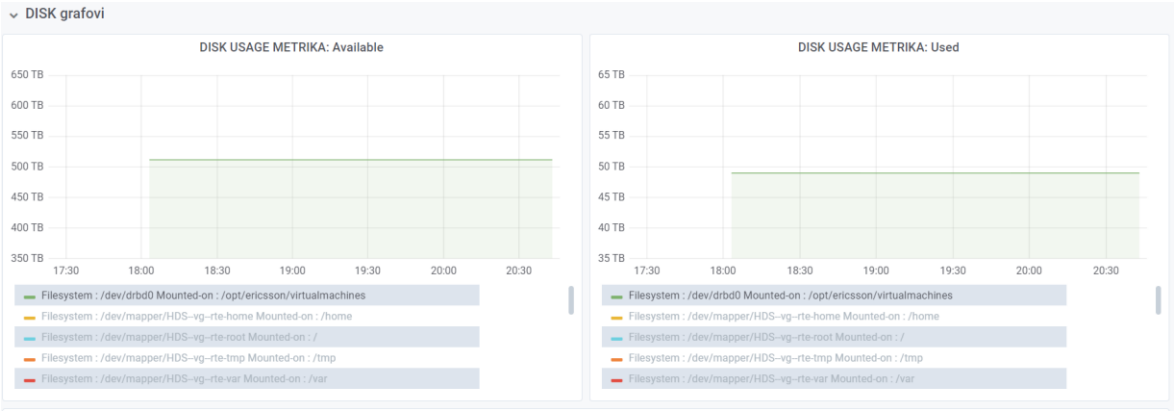
Vizualizirani podatci na Grafani su u nastavku.



Slika 3.8 Opterećenje procesora po komponenti



Slika 3.9 Graf iskorištenja RAM-a



Slika 3.10 Graf iskorištenja diska

4. ZAKLJUČAK

U radu je iznesena problematika spremanja podataka s hardware-skih i software-skih komponenti u podatkovnom centru kao i vizualizacija tih podataka. Dan je uvid u komponente software-ski definirane infrastrukture podaktovnog centra, testiranje sustava te nadzor parametara pri testiranju. Kako nadzor čini bitan dio dnevnih aktivnosti u testiranju pokazano je kako se može razviti rješenje za analizu, obradu i vizualizaciju podataka o sustavu, zapisanih u tekstualne datoteke. Iznesene su moderne tehnologije baze podataka, parsiranja i vizualizacije, razložena struktura koda s naglaskom na najbitnije dijelove, te prikazan način uporabe jednog takvog rješenja u svakodnevnom poslu. Dano rješenje nije jedino, te je alat ovakav kakav je prikazan daleko od gotovog, ali služi kao temelj izgradnje software-a za nadzor u budućnosti, te se ostavlja mjesto za napredak. Poboljšanja se mogu postići u radu parsera, razvoju univerzalnih parsera, kao i u načinu ostvarivanja SSH veza do razine gdje bi se nadziranje moglo vršiti u stvarnom vremenu, umjesto analiziranja „starih“ podataka ponašanja sustava.

5. LITERATURA

Svim poveznicama je zadnji pokušaj pristupa bio dana 21.9.2020.

[1] Node.js - https://nodejs.org/en/

[2] InfluxDB - https://www.influxdata.com/

[3] Grafana - https://grafana.com/

[4] Ericsson SDI - https://www.ericsson.com/en/portfolio/digital-services/cloud-infrastructure/software-defined-infrastructure

[5] Ericsson NFVI - https://www.ericsson.com/en/digital-services/cloud-infrastructure/nfvi

[6] Ericsson HDS 8000 - https://www.ericsson.com/en/news/2017/7/capacity-of-ericsson-hds-8000-boosted-with-new-technology-from-intel

[7] Intel Rack Scale Design - https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html

6. POPIS SLIKA

Slika 1.1 Prikaz prvotnog rješenja na primjeru procesora 3
Slika 1.2 Prikaz položaja SDI proizvoda u NFVI portfoliju 4
Slika 1.3 Pregled HDS 8000 sustava 6
Slika 1.4 CSU - Compute Sled Unit 6
Slika 1.5 SSU - Storage Sled Unit 7
Slika 1.6 NSU – Network Sled Unit 8
Slika 1.7 EAS - Equipment Access Switch 8
Slika 1.8 CCM - Command Center Manager 9
Slika 2.1 Primjer neobrađenog zapisnika iskorištenosti procesora 12
Slika 2.2 parseCpu.js glavni kod 14
Slika 2.3 Neobrađeni zapisnik iskorištenja radne memorije 17
Slika 2.4 parseRam.js glavni kod 18
Slika 2.5 Neobrađeni zapisnik iskorištenja diskovnog prostora 20
Slika 2.6 parseDisk.js glavni kod - DIO 1 21
Slika 2.7 parseDisk.js glavni kod – DIO 2 22
Slika 2.8 Implementacija metode *getLogInfo( )* 24
Slika 2.9 Primjer uporabe JSDoc anotacija 25
Slika 3.1 Prikaz toka načina rada aplikacije 26
Slika 3.2 Biblioteke / moduli u glavnom dijelu aplikacije 27
Slika 3.3 Glavna aplikacija - DIO 1 28
Slika 3.4 Glavna aplikacija - DIO 2 29
Slika 3.5 bin.js modul 30
Slika 3.6 Glavna aplikacija - DIO 3 30
Slika 3.7 Rezultat pokretanja aplikacije u terminalu 31
Slika 3.8 Opterećenje procesora po komponenti 31
Slika 3.9 Graf iskorištenja RAM-a 32
Slika 3.10 Graf iskorištenja diska 32