VISOKA ŠKOLA ELEKTROTEHNIKE I RAČUNARSTVA STRUKOVNIH STUDIJA

JANKOVIĆ Marko

ANALIZA ARHITEKTURE PLATFORME ZA SKALABILNE VEB APLIKACIJE

- diplomski rad -

Beograd, 2016

Kandidat: **Janković Marko**

Broj indeksa: **14/05**

Studijski program: **Nove računarske tehnologije**

Tema: **ANALIZA ARHITEKTURE PLATFORME ZA SKALABILNE VEB APLIKACIJE**

Osnovni zadaci:

1. Analiza problema skalabilnosti

2. Analiza Node.js platforme

3. Primena Node.js u kreiranju skalabilnih aplikacija

Hardver: 0% Softver: 0% Teorija: 100%

Mentor:

Beograd, Septembar 2016

_____

dr Boško Nikolić, prof. VIŠER

Izvod

Cilj ovog rada je analiza arhitekture Node.js platforme, programiranja vođenog događajima i neblokirajućeg I/O modela, koji imaju cilj da optimizuju skalabilnost u veb aplikacijama.

Abstract

This work aims to analyze architecture of Node.js platform, event-driven programing and non-blocking I/O model, which aim is to optimize scalability in Web applications.

SADRŽAJ

1. UVOD

Prve veb **(web)** stranice su bile veoma jednostavne i statične, sastojale su se od mnogo informacija koje su korisnici mogli da čitaju, ali pri tome nisu mogli da utiču na sadržaj ili izgled stranice. **Web** stranice su isključivo imale zadatak da prenesu informacije i reference efikasno prema drugim. Potreba za boljom funkcionalnošću sajtova i interaktivnijim sadržajem na **web**-u postala je sve veća i zahtevnija. Jedan od uslova dobrog **web** sajta pored njegovog dizajna i sadržaja je i njegova interakcija sa korisnikom. U tu svrhu je kreiran JavaScript. Ideja je bila napraviti jednostavan jezik koji će se lako implementirati u HTML dokument. Danas JavaScript možemo koristiti za kreiranje kompletne aplikacije korišćenjem istog jezika i u ***web browser***-u (pretraživaču) i na serveru.

Predmet analize ovog rada je arhitektura Node.js platforme i uloga komponenti od kojih je sačinjena. Počevši od uvoda u istoriju JavaScript-a i generalnog opisa problema skalabilnosti, u radu će biti predstavljena Node.js platforma, način na koji ona obezbeđuje veću skalabilnost u odnosu na tradicionalna rešenja, kao i njene prednosti, osnovne karakteristike, izvršni model i asihrono programiranje. Posebna pažnja se posvećuje pristupima i algoritmima implementiranim za efikasniju upotrebu memorije i procesora kao i izazovima koji proističu korišćenjem ovog pristupa.

2. ISTORIJA JAVASCRIPT-A

JavaScript je objektno-orijentisan skriptni jezik nastao u kompaniji Netscape i napravio ga je Brendan Eich za 10 dana u Maju 1995 godine. JavaScript nije uvek bio poznat kao JavaScript. Prvobitno ime je bilo Mocha, ime izabrano od Marka Andresena, osnivača Netscape-a. U Septembru 1995 godine ime je promenjeno u LiveScript, da bi u Decemberu iste godine, posle dobijanja zaštitnog znaka od Sun-a (danasnji Oracle), ime JavaScript bilo usvojeno. U to vreme Netscape je blisko sarađivao sa Sun-om, kompanijom zaslužnom za kreiranje Java programskog jezika. Izbor imena izazvao je dosta špekulacija. Mnogo su mislili da Netscape pokušava da iskoristi popularnost Jave koja je u to vreme postala jako popularna i da je ta promena imena bila marketinški trik. Nažalost, izbor imena izazvao je dosta konfuzije jer su mnogi automaski predpostavili da su dva jezika nekako povezana. U realnosti imaju jako malo zajedničkog osim onih stvari koje su oba nasledila iz C jezika.

Uprkos konfuziji, JavaScript postaje veoma uspešan klijentski (***client-side***) skriptni jezik. Kao odgovor na uspeh JavaScript-a, Microsoft kreira svoj jezik i naziva ga JScript, koji ugrađuje u Internet Explorer 3.0 u Avgustu 1996. U Novemberu 1996 Netscape podnosi zahtev Evropskoj asocijaciji proizvođača računara (ECMA – European Computer Manufacturers Association) za prvu verziju jezičke specifikacije poznate kao ECMAScript. U Junu 1997 godine JavaScript postaje standard ECMA-262. Od tada, svi pretraživači Microsofta i ostali popularni pretraživači implementiraju verzije standarda ECMAScript baziranog na radu Netscape-a. ECMAScript postaje ime za oficijalni standard, sa JavaScript-om kao najpoznatijom implementacijom.

Standard je nastavio da se usavršava u ciklusima, sa izdanjima ECMAScript 2 1998 godine i ECMAScript 3 1999 godine, koji je osnova za moderniji JavaScript. Izdanje "JS2" ili "original-ES4" na čelu sa Valdemarom Horvatom, započetim 2000 godine i na početku sa Microsoft-om, koji se činilo da učestvuje i čak implementira neke od predloga u njihov JScript.net jezik. Vremenom, postalo je jasno da Microsoft nema nameru da sarađuje ili implementira standard u IE (internet explorer), iako nemaju kontra predloge pa se 2003 odustaje od JS2/original-ES4 standarda.

Sledeći veliki ciklus desio se 2005 godine, sa dva glavna događaja u JavaScript istoriji. ECMA postaje neprofitabilna organizacija i počinje se sa radom na ECMA-357 poznatijem kao E4X (ECMAScript for XML) u saradnji sa Macromedia, koja implementira E4X u ActionScript 3 (AS3). Sa Macromedia (kasnije kupljen od Adobe), ECMAScript 4 je opet postao aktuelan sa ciljem standardizacije onog sto je AS3 implementirao u SpiderMonkey okruženje. U tom cilju, Adobe je kreirao javnu biblioteku (***open source***), AVM2 (ActionScript ***Virtual Machine*** 2) i nazvao je Tamarin.

Takođe jos uvek su postojale velike razlike među bitnim igracima. Douglas Crockford (u to vreme iz Yahoo!), 2007 godine u saradnji sa Microsoft-om

suprotstavlja se ECMAScript 4, što dovodi do kreiranja ECMAScript 3.1. Dok se sve to dešavalo, programeri i zajednice programera otkrivali su šta sve može biti urađeno sa JavaScript-om i taj napor je rezultirao da 2005 godine James Garrett napisao članak u kom je skovao termin ***Ajax*** (*Asynchronous JavaScript and XML*). ***Ajax*** je omogućio kreiranje ***web*** aplikacija gde podaci mogu biti učitani u pozadini, bez potrebe za učitavanjem čitavog novog dokumenta. Jesse je opisao set tehnologija, baziranih na ***XMLHttpRequest***-u i JavaScript objekat-u, koji dopušta slobodnu asinhronu komunikaciju klijenta sa serverom - dopremanje informacija u običnom tekstualnom ili struktuiranom (XML ili JSON) obliku klijentu što je omogućilo pravljenje dinamičkih aplikacija. To je dovelo do preporoda JavaScript-a vođenjim ***open source*** bibliotekama i zajednicama formiranim oko njih, sa bibliotekama kao što su Prototype, jQuery, Dojo i Mootools i nastanku “Jednostranih aplikacija” (“Single Page App” ili “SPA”). U Julu 2008 godine u Oslu došlo je do dogovora koji je rezultirao da se 2009 godine ECMAScript 3.1 preimenuju u ECMAScript 5.

Tokom godina, JavaScript je ostao standard za razvoj aplikacija u ***web browser***-u ali je razvoj aplikacija na serveru pripadao je jezicima kao što su PHP i Java. Određen broj projekata je implementiran u JavaScript-u na serverskoj strani ali nijedan od njih nije bio uspešan. Dve velike prepreke blokirale su JavaScript za široku primenu na server. Prva je bila reputacija - na JavaScript se dugo vremena gledalo kao "jezik igračka", pogodan samo za amatere. Druga prepreka bile su slabe performanse u poređenju sa ostalim jezicima. Međutim, JavaScript je imao jednu veliku prednost. ***Web*** se neverovatno brzo razvijao, kao i "rat" među pretraživačima. Kao jedini jezik podržan na svim pretraživačima, JavaScript je dobijao pažnju od kompanija kao što su Google, Apple i druge kompanije što je dovelo do ogromnih poboljšanja u JavaScript performansama.

Sve to nas je dovelo do toga da danas JavaScript ulazi u potpuno novu evoluciju inovacija i standarda, sa novim platformama koje omogućuju upotrebu JavaScript-a na serveru. “***Server-side***” JavaScript je termin koji se odnosi na JavaScript koji se izvršava na serveru. Ekstenzija JavaScripta na serveru omogućena je pomoću JavaScript okruženja - “JavaScript ***engine***”. Neki od njih su SpiderMonkey implementiran u programskom jeziku C, Rhino implementiran u programskom jeziku Java.

Krajem 2009 godine, na JavaScript konferenciji u Berlinu, Ryan Dahl je predstavio projekat na kome je radio. Na iznenađenje svih prisutnih projekat nije bio dizajniran da radi u ***web browser***-u već na serveru. Projekat je nazvan Node.js i danas je među programerima poznatiji samo kao Node. Node.js je pisan u C++ i JavaScript-u, koji koristi Google-ov JavaScript V8 engine, i u prvoj verziji bio je podržan samo na Linux-u. Od samog početka, Node je dobio pažnju velikih igrača u industriji kao sto su Joyent i Microsoft koji su 2011 godine kreirali i Node.js verziju za Windows. Iste godine predstavaljen je registar paketa (***package manager***) za

Node.js, nazvan NPM, koji omogućava programerima da objave i dele module i biblioteke pisane u Node.js.

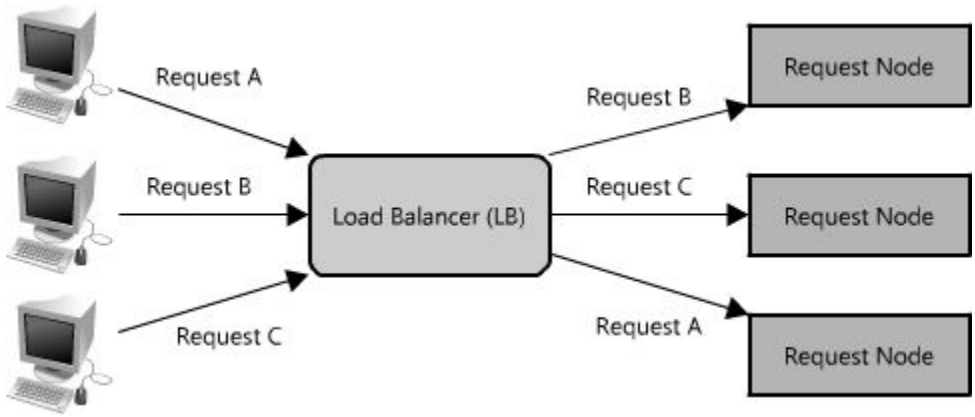
3. SKALABILNOST

U praksi se za isporuku manjih sajtova uglavnom koristi jedan računar koji ima instalirane sve potrebne komponente - ***Web server***, server baze podataka i drugo. Jedan takav računar može služiti za isporuku čak i više stotina manje opterećenih sajtova. U slučaju povećanja opterećenosti sajta prvi korak za podizanje performansi jeste njegovo izmeštanje na namenski računar, fizički ili virtualni, a zatim na dva računara od kojih jedan obavlja samo funkciju ***web*** servera a drugi servera baze podataka.

3.1. Vertikalno I horizontalno skaliranje

U slučaju da ni namenski računari za pojedinačne funkcije ne mogu da odgovore na opterećenje, prvi korak jeste unapređivanje njihovih nosećih komponenata, prvenstveno kroz dodavanje više naprednijih centralnih procesora sa više jezgara i veće količine radne memorije. Ovakav način povećanja performansi serverskih sistema naziva se "**vertikalno skaliranje**". U slučaju da se dođe do granice mogućeg vertikalnog skaliranja, a da još uvek nije postignut željeni nivo performansi, pribegava se **horizontalnom skaliranju**, balansiranju opterećenja i upotrebi farmi servera.

Horizontalno skaliranje predstavlja pristup povećanja performansi serverskih sistema kroz dodavanje većeg broja računara koji obavljaju istu funkciju, na primer isporuku ***Web*** prezentacija i aplikacija, ili serviranje baza podataka. Ovakve grupe servera nazivaju se klasterima servera ili farmama servera.



Slika 1. Load Balancer (LB)

Složenost kod korišćenja serverskih klastera potiče od potrebe da se opterećenje raspodeli na dostupne servere, kao i da se njihovi podaci sinhronizuju. Za ovu funkciju se koristi ***load balancing,*** balansiranje opterećenja, a nju obavlja

server koji se ponaša kao dispečer zahteva. Odluku o tome kome će od servera u klasteru proslediti zahtev dispečer može donositi jednostavno - na primer korišćenjem ***round-robin*** algoritma ili zasnovano na aktuelnim informacijama o tome koji je server najmanje opterećen.

Jedan od bitnih uticaja na performanse i dostupnost ***Web*** servisa dolazi i od izbora operativnog sistema, kao osnove za rad Web servera. Implementacije sa javno dostupnim izvornim kôdom uglavnom imaju podršku za sve popularne serverske operativne sisteme. Međutim, zatvorene implementacije često su ograničene na podršku za samo operativne sisteme istog ili preferiranog proizvođača. Osim podrške za operativne sisteme, očekivane dostupnosti i broja neuspešnih zahteva, bitan kriterijum čini i podrška za tehnologije na strani servera.

Skalabilnost aplikacija je jedan od najbitnijih problema sa kojima se programeri ***web*** aplikacija suočavaju. Tradicionalni pristup pograzumevao je ***thread based*** skaliranje, skaliranje Nitima. Zbog nekoliko osnovnih nedostataka koje limitiraju sistemsko skaliranje mnogo veliki igrači počeli su da usvajaju alternativni programski ***event-driven*** model (vođen događajima) kako bi ostvarili efikasniju skalabilnost.

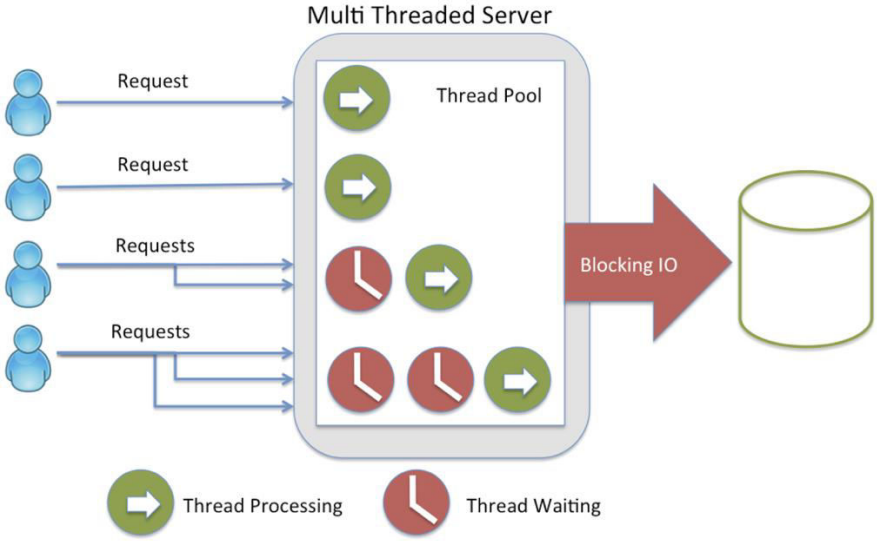
3.2. Višenitna obrada (*Multithreading*)

Već dugi niz godina, performanse računara uglavnom su bile ograničene na brzinu jednog mikroprocesora na kojem CPU može da radi samo jednu stvar u isto vreme. Tradicionalno, programiranje se obavlja sinhrono, linija kôda se izvršava, sistem čeka rezultat, rezultat se procesuira i zatim se izvršavanje programa nastavlja. Ponekad taj sistem izvršavanja zahteva dugo čekanje, npr. čitanje sa baze podataka. Kako je brzina pojedinačnih procesora počela da dostiže svoje granice praktičnosti, proizvođači čipova počeli su da se prebacuju ***multi-core*** (višejezgarni) dizajn, dajući računaru priliku da koristi više Niti (***threads***) na jednom CPU.

Serverski (***Server-side)*** jezici, poput Jave i PHP, koriste izvršni model koji povezuje svaki korisnikov zahtev (***request)*** za jedan ***thread***, poznat kao ***thread-per-request*** (jedna Nit za jedan zahtev). Izvršni modeli sastoje se od ***thread dispatcher****-a* (dispečera niti) koji dođeljuje svaki novi dolazni ***request*** pravi poseban ***thread*** izvršenja iz svojih rezervi niti (***thread pool***) i koristi blokirajuće I/O operacije. To znači da ***thread*** (servisni proces ili niti) koji je izdat za obradu konekcije ostaje blokiran dok se on ne izvrši, server izdaje zahtev i onda čeka na izvršenje upita i na kraju vrati odgovor.

Rezultat toga je da u bilo kojem trenutku, server povećava broj ***thread***-ova kako se povećava broj dolaznih ***request***-ova i aplikacija će obraditi jedan ***request*** po

thread-u u isto vreme bez obrade paralelnih ***request***-ova. Kako se broj klijenta povećava, tako da se povećava i broj aktivnih ***thread***-ova, koji zauzimaju memoriju i potencijalno mogu dostići maksimalnu dostupnu količinu memorije. Zbog toga, operativni sistemi obično imaju maksimalni broj ***thread***-ova koje podržavaju, što bitno ograničava serversku skalabilnost. Ukoliko su svi ***thread***-ovi zauzeti, novi ***request*** čeka da se jedan ***thread*** oslobodi pa se stavlja naglasak na što brže generisanje odgovora i oslobađanje ***thread***-a i izbegavanje zastoja (***deadlocks)***.



Slika 2. Blokirajući I/O

Proces ima stanje i uglavnom ne radi ništa dok se I/O ne završi što bi moglo da traje od 10 ms do nekoliko min, u zavisnosti od kašnjenjenja I/O operacije. Kašnjenje takođe može bi uzrokovano i nekim neočekivanim uzrocima, npr. ***hard disk*** je usao u "režim održavanja" i pauzirao sve I/O operacije ili je upit na bazu podataka spor zbog povećanog opterećenja.

Multithreaded metodologija programiranja je jednostavna, intuitivana ali ima neke fundamentalne nedostatake kao što su ***deadlocks***, pristupačnost i zaštita deljenih resursa između ***thread***-ova gde rešenja dodaju značajnu kompleksnost. Programeri takođe gube određeni nivo kontrole jer operativni sistem obično odlučuje koji će ***thread*** izvršiti i koliko dugo. Za rešavanje ograničenja skalabilnosti, moderni ***data-centri*** rade skaliranje hardverskih resursa. Umesto skaliranja broja ***thread***-ova na jednom serveru, ***thread***-ovi se distribuiraju i balansiraju preko velikog broja serverski mašina. Povećanjem broja fizičkih mašina zahteva više energije, hlađenje i administrativne zahteve koje direktno utiču na ukupne troškove. ***Multithread***-ing je poželjan za tradicionalne ***web*** aplikacije, ali ne i za aplikacije koje zahtevaju čestu razmenu poruka u skoro realnom vremenu sa serverom, što zahteva potencijalno desetine hiljada konekcija koje ostaju otvorene.

4. NODE.JS PLATFORMA

Node.js je ***single-threaded*** (jedno-nitna) serverska JavaScript platforma, bazirana na Google Chrome ***V8 engine-u,*** za kreiranje brzih i skalabilnih mrežnih aplikacija.

Kako bi koordinisala ulaz i izlaz podataka preko raznih distribuiranih sistema, platforma mora biti fleksibilana i pouzdana. Da bi se to postiglo Node arhitektura ima tri glavne karakteristike: ***Single-Thread*** *(Jednu Nit)****, Non-blocking I/O*** *(Nebolokirajući I/O)* ***i Event Loop*** *(Petlju Događaja),* koje su ostvarene kombinacijom ***V8 engine,*** Libuv bibliotekom i drugim C/C++ komponentama koje za povezivanje jednog jezika sa drugim koriste ***Bindings.***

S obzirom da je V8 ***engine*** namenjen prvenstveno za rad u ***web browser***-u, a ne na serveru, bilo je potrebno proširiti njegovu funkcionalnost kako bi odgovaralo zadacima i na serverskoj strani. U tu svrhu napisani su C i C++ moduli koji koristeći ***Bindings*** obezbeđuju interakcije sa serverima na niskom nivou da bi se uspostavile bitne serverske funkcionalnosti kao što su umrežavanje, kompresija, enkripcija, manipulacija binarnim podacima, operacije sa I/O i datotekama, pristup bazama podataka itd.

Bindings su izvršni kôd koji omogućava da V8 i druge biblioteke pisane u različitim jezicima međusobno komuniciraju. U slučaju Noda, ***bindings*** omogućava JavaScript-u da “dohvati” glavne Node komponente kao što su c-ares (asinhroni DNS ***requests***), zlib (kompresija i dekompresija), OpenSSL (sigurnost prenosa podataka) i http-parser (HTTP parser). Takođe, ***Bindings*** nam omogućava da uključimo u aplikaciju i neke spoljne ili namenske biblioteke pisane u C/C++ ali je potreno napisati poseban ***addon***, kôd koji bi “zalepio” našu biblioteku za aplikaciju.

Korišćenje JavaScript-a na serveru, omogućava nam da koristimo ***event-based*** model (baziran i vođen događajima) koji je pogodan i za asinhrono programiranje. Takođe, omogućava nam korišćenje jedinstvenog jezika na serveru i u ***web browser***-u što znači da kompletnu aplikaciju možemo realizovati pomoću jednog programskog jezika. JavaScript je sveprisutan jezik, poznat milionima programera koji već imaju značajno iskustvo radeći na ***client-side*** (klijentskim) aplikacijama. Prednost toga je da većina programera neće morati da uči novi jezik da bi počeli da prave Node aplikacije, što smanjuje krivu učenja Noda.

Struktura podataka takođe mogu biti deljene između klijenta i servera. Korišćenje iste strukture omogućava lakšu i bržu razmenu podataka i eliminiše dodatno trošenje vremena na konvertovanje. Najpoznatiji način razmene podataka je JSON (JavaScript objektna notacija) format koji je ugrađen u JavaScript.

4.1. Sinhrono I/O programiranje

U sekvencijalnom (sinhronom) izvršenju kôda, svaka I/O operacija, obično pozivi na različite sistemske ili mrežne funkcije, izazvaće zastoj. Kôd će blokirati aplikaciju i aplikacija će izgledati kao zaustavljen proces sve dok se jedna operacija ne završi. Procesi blokiranja se mogu bolje razumeti kao procesi mirovanja, a takvi procesi su uska grla celokupnog toka aplikacije.

Datoteka može biti vrlo velike veličine i može potrajati neko vreme dok se učita u memoriji. Nakon što se završilo čitanje sadržaja u memoriju, program će se nastaviti sa izvršenjem sledeće linije na kojoj će biti prikazan sadržaj datoteke. To znači da ako želimo sinhrono da čitamo ili pišemo u datoteku, proces će morati da sačeka dok hardver završi fizički I/O, i vrati informaciju o statusu operacije, uspehu ili neuspehu. Nakon toga, program će se nastaviti sa sledećom datotekom pa će izvršenje sledećih linija kôda "čekati", sve dok izvršenje prethodne operacije ne završe.

4.2. Asinhrono I/O programiranje

Asinhroni I/O, ili neblokirajući I/O, je forma ulazno/izlazne obrade koja omogućava drugim procesima da nastave izvršenje pre nego što se prethodni završi. Nakon što asinhroni proces izda zahtev za čitanje/pisanje, i prosledi se na “red čekanja” ili hardveru, sistemski poziv se odmah vraća. Tako izvršenje procesa nije blokirano, jer ne mora da čeka rezultat iz sistemskog poziva, pa se ga zato zovemo ne-blokirajući I/O.

4.3. Programiranje vođeno događajima (*Event-driven*)

U računarskom programiranju, ***event-driven*** programiranje ili programiranje bazirano na događajima je programska paradigma u kojoj je se protok programa određuje događajima.

Event-driven programe definišu dva faktora - ***events*** (događaji) i ***callbacks*** (povratni pozivi, korisnici tih događaja). Događaji mogu biti sistemski događaji niskog nivoa, kao sto su "datoteka je spremna za upis". ***Callbacks*** su funkcije koje se pozivaju i izvršavaju kada se dogodi određeni događaj, npr. kada upiti na bazu podataka vrati rezultat.

Stil programiranja gde se umesto korišćenja povratne vrednosti, definišu funkcije koje će biti pozvane od strane sistema kada se pojave određeni događaji, naziva se ***event-driven*** programiranje. ***Event-driven*** programiranje nudi efikasniju, skalabilnu alternativu konvencionalnom ***thread-per-request*** pristupu, koja omogućava programerima puno više kontrole. U ovom modelu, aplikacija se oslanja

na sistemske funkcije kao što su Unix sistemski pozivi **select()** i **pool()**, Linux **epoll** servis, ili **kqueue** i **kevent** pozivi dostupni u OS X.

U ***server-side*** asinhronom programiranju, svaki ***request*** korisnika ili dolaznog I/O zahteva smatraju se kao događaja aplikacije. Svaki događaj je povezan sa povratnim pozivom (callback) i smesten u ***event queue***, redu događaja za sekvencijalnu obradu.

Event-driven sistem koristi ***event loop*** (petlju događaja) koja obrađuje jedan po jedan događaj, proverava da li postoji novi I/O događaj u ***event queue*** i izvršava redom sve dostupne ***callbacks*** (povratne pozive). ***Callback*** takođe može pokrenuti dodatne I/O operacije koje se izvršavaju asinhrono uz poštovanje ***event loop***-a kako ne bi blokirao druge zahteve.

```
fileSystem.read("big.txt", function(data) {
  console.log(data.toString());
});

fileSystem.read("anotherbigfile.txt", function(anotherdata) {
  console.log(another.toString());
});
```

Slika 3. Event-driven primer

Funkcija **fileSystem.read** je asinhrone prirode, aplikacija će prikazati sadržaj kada se čitanje završi ali neće čekati, već će nastaviti izvršenje i čitanje sledeće datoteke.

4.4. Node.js programski model

U cilju skaliranja velike količine klijenata i fokusa na performanse i nisku potrošnju memorije, Node arhitektura razlikuje se od tradicionalnih okruženja. Node I/O pristup je strogo asinhroni I/O i otklanjanje blokirajućih procesa pomoću ***event-driven*** modela je primarni organizacioni princip. Svaka funkcija koja obavljanja I/O mora koristiti ***callback***.

Node.js sadrži jedan glavni red događaja (***event queue***), a svaki zahtev izaziva jedan ili više događaja, koji se konkurentno obrađuju što ga čini savršenim za razmenu velike količine podataka u realnom vremenu. Naročito pogodan za aplikacije koje moraju da održavaju perzistentnu konekciju sa serverom, najčešće korišćenjem **WebSocket**-a. Umesto da kreira novi ***thread*** za svaki novi ***request***, Node.js koristi isključivo samo jedan ***thread*** za obradu događaja. Kada se pojavi potencijalno blokirajući proces, kao što je čekanje na podatke iz datoteke, program ne čeka, nego ide na izvršavanje sledeće linije kôda, a događaj se prosleđuje ***thread pool*** (***libuv*** biblioteka) dispečeru. U suprotnom, instrukcija se smešta u ***event queue*** i čeka na ***event loop*** da ga pročita i izvrši. Kada se I/O operacija vrati, pokreće se

callback funkcija i rezultat se procesuira. Takav model omogućuje mu da podrži desetine hiljada konkurentnih konekcija, bez brige o memoriji i menjanju konteksta između ***thread***-ova.

Single-threaded event model nije pogodan za velike obrade i CPU intenzivne operacije će blokirati Node odaziv trenutne konekcije dok će ostatak konekcija drži u ***event queue*** da bi ih izvršio kasnije. Event-based programiranje zahteva od programera duboko razumevanje ***event loop*** arhitekturure i pisanje modula koji će biti neblokirajući da bi se bi se komplikovane i dugotrajne operacija uradile u više koraka obrade. Takvi moduli obično uključuju tehnike ugnezdenih povratnih poziva što povećava složenost razvoja.

4.5. Petlja događaja (*Event loop*)

Event loop je struktura koji obavlja dve funkcije u beskonačnoj petlji. U svakoj iteraciji **event loop** je zadužen za otkrivanje trenutnog događaja (***event***), izvršenje povratnog poziva (***callback***) i vraćanja rezultata aplikaciji. JavaScript se pokreće u ***single-thread***-u ali Node u pozadini koristiti ***thread pool*** na nivou operativnog sistema. Tokom ***request-response*** ciklus, većinu vremena se potroši na I/O operacije pa su sve I/O operacije asinhrone priorode, kontrolisane kroz Node “glavni proces” (***main process***). Svaka operacija pokrenuta je različitom ***thread***-u unutar ***thread-pool***-a (rezervoar niti), garantovajući asinhrono izvršavaje, bez blokiranja ***event loop***-a što omogućava upravljanje velikog broja operacija, puštajući ***event loop*** da efikasno optimizuje izvršenje zadatka i upravljanje ***thread pool***-a.

Ova sposobnost ima značajan uticaj na performanse aplikacije, pojednostavljuje se asinhrono programiranje, što Node čini snažanim alatom za programere. Serveri mogu obraditi hiljade konkurentnih ***request***-ova koristeći vrlo ograničen broj procesa, obično jedan proces po jezgru na ***multi-core*** serverima. Za svaki I/O biće dodeljen poseban ***worker*** u pozadini. ***Main process*** i ***thread***-ovi koji rade u pozadini komuniciraju pomoću reda (***queue)*** koji predstavlja dodeljene I/O zadatke, koji će obavestiti ***main process***-u da je zadatak završen pozivom **callback**-a (povratna funkcija). JavaScript sloj može da komunicira sa ***main process***-om samo preko C sloja (libuv biblioteka). U ***main process***-u, Node serverski poziv nije blokiran jer proces samo delegira I/O zadatake ***thread worker***-ima i odgovora klijentu kad god se zadaci završe. Osim toga, ***main process*** neprekidno iterira kroz petlju da opsluži nove ***request***-ove.

Event loop je skriven od programera i njegova svrha je da postavi odgovarajuće događaje (***events***) koji omogućuju programeru da tretira ***event-driven*** programiranje kao skup ***callback*** funkcija koje će se izvršiti. Budući da se ***event loop*** pokreće u ***single-thread***-u, dok su sve I/O operacije vezane za operativni

sistem, programer piše kôd koji će se izvršavati za svaki ***event*** unutar svakog ***callback***-a, bez brige o problemima konkurentnosti. Postoje situacije kada je moguće prekinuti ili blokirati ***event loop.*** Na primer, ***while*** petlja u primeru ispod, se nikad neće završiti.

```
var stop = false;
setTimeout(function() {
        stop = true;
}, 1000);
while(stop === false) {};
```

Iako bi se moglo očekivati da će se posle jedne sekunde stanje varijable ***stop*** promeniti u **Boolean true**, to se nikada neće dogoditi. While petlja blokira ***event loop,*** konstantno proveravajući vrednost koja nikada nije imala priliku da se promeni, jer ***event loop*** nikada nije imao priliku da registruje **setTimeout callback**.

4.6. Višeprocesorska obrada (*Multiple Processes*)

Većina ***framework***-a baziranih na ***event-base*** modelu, dizajnirani su da iskoriste ***multiple core*** mogućnost, tako što vrše replikaciju procesa. Ovaj pristup prisiljava programera da usvoji strategije bazirane na ***master-worker*** principu.

Svaka Node instanca je predstavljena kao jedan proces. Kako bi bilo moguće iskoristiti dodatne hardverske resurse ***multi-core*** procesora, Node.js podržava Horizontalno Skaliranje preko ugrađenog Cluster modula, proces koji kreira više ***Worker***-a (pod-procesa) na istom serveru. Svaki od procesa pokreće posebnu instancu Node aplikacije, efikasno koristeći operativni sistem kao neku vrstu ***load balancer***-a.

To je uobičajena metoda skaliranja servera, čime se omogućuje korišćenje svih dostupnih procesorskih jezgara. Ove Node instance mogu deliti isti ***socket*** a raspodela opterećenja između Node instanci može biti prepuštena operativnom sistemu, ili u ***round-robin*** algoritmu ugrađenom u ***Master*** proces. Ovakva arhitektura pojednostavljuje raspodelu opterećenja i upravljanje podprocesima, kao i mogućnost da se automatski kreira i pokrene nova instanca u slučaju da aplikacija “padne”. Postoji više načina da se to postigne, u zavisnosti od operativnog sistema i mehanizmi koji o tome brinu su izdvojeni u Libuv biblioteci.

5. LIBUV

libuv je ***cross-platform*** (međuplatformska) biblioteka za podršku i izvorno je napisana za NodeJS u programskom jeziku C. Libuv nudi isti ***API*** (interfejs) za Windows i Unix. Dizajniran oko ***event-driven*** asinhronog I/O modela, odgovoran je za implementacije i događaje vezane za sistemske datoteke, **event loop**, ***thread-pool*** i kreiranje podprocesa. Biblioteka pruža mnogo više od same apstrakcije različitih I/O ***polling*** mehanizama.

Libuv ***thread-pool*** iz kojeg se jedan ***thread*** dođeljuje za svaku I/O operaciju je fiksne veličine. Delegiranjem dugotrajnih operacija na Libuv modul, V8 engine i ostatak Noda je slobodan da nastavi izvršavanje drugih ***request***-ova. Pre 2012. godine, Node se oslanjao na dve odvojene biblioteke, Libio i Libev, kako bi se podržao ***event loop i*** osigurao asinhroni I/O. Libev je radio samo na Unix a kako je rasla popularnost Noda, bilo je važno osposobiti ga da radi na Windows operativnom sistemu. Windows ekvivalent sistemskog mehanizama za događaje, kao što su **kqueue** ili (e)**pool** je IOCP.

Da bi ugradili podršku za Windows, Libio je bio urađen kao apstrakcija oko Libev. Kao su programeri i dalje menjali Libev, postalo je jasno da će održavanje i povećanje performansi jednostavnije biti rešeni ako se kreira nova bibliteka. Na primer, Libev unutrašnja petlja je imala nepotrebne funkcionalnosti za Node projekat pa su njegovim uklanjanjem, programeri bili u mogućnosti da povećaju performanse za gotovo 40%. Sa uvođenjem Libuv u Node verziji 0.9.0, Libev i Libio su u potpunosti uklonjene. Od tada libuv nastavlja da sazreva i postaje samostalna biblioteka za sistemsko programiranje i korišćena je, osim Noda, i u drugim plaformama kao sto su Mozilla Rust jezik, Luvit, Julia, pyuv itd.

Neke od funkcija koje **libuv** podržava:

- Petlja događaja - Event Loop - epoll, kqueue, IOCP
- Asinhroni TCP i UDP soket
- Asinhroni DNS
- Asinhroni file system operacije
- File system događaji
- Kreiranje podprocesa - child processes
- Bazen niti - Thread pool
- Upravljanje signalima

5.1. Libuv petlja događaja (*Event loop*)

Event loop je središnji deo libuv funkcionalnosti, brine o ***pooling***-u I/O operacija i raspoređivanju ***callback***-a koji su pokrenuti od različitih izvora događaja, a to znači da je vezan za ***single-thread***. Event loop se može pokrenuti više puta sve dok je svaki pokrenut u posebnom ***thread***-u ali je bitno napomenuti da ***event loop*** nije ***thread-safe.***

Event loop prati ***single-threaded asynchronous*** I/O pristup: svaki I/O, izvodi se na neblokirajućem ***socket***-u koji koriste najbolje mehanizame dostupane na određenoj platformi: epoll na Linux, kqueue na OSX i IOCP na Windowsima. Kao deo petlje iteracije, petlja će blokirati čekajući I/O aktivnosti na ***socket***-u i ***callback*** će biti trigerovan ukazujući na stanje socket-a. Odgovornost prikupljanja događaja iz operativnog sistema ili praćenje drugih izvora događaja je na libuv, a korisnik može registrovati ***callback*** koji će biti uključen kada se događaj desi. Neki primeri događaja su "Datoteka je spremana za pisanje" ili "Socket ima podatke spremne za čitanje".

Libuv koristi bazen niti (***thread pool***) kako bi obezbedio asinhrone I/O operacije nad fajlovima, ali se mrežni I/O uvek izvodi u jednoj niti (***single-thread***). Iako je ***pooling*** mehanizam drugačiji, libuv izvršeni model je konzistentan kroz Unix sistem i Windows. libuv će obično koristi pozadinske ***worker threads*** kako bi izvršio zadatke u neblokirajućem maniru. Libav biblioteka održava bazen niti koje se koriste Node.js za obavljanje operacija koje dugo traju, bez blokiranja svoje glavne niti. ***thread pool*** se koristi tako što pošalje “izvršni zahtev” (***work request***) na red (***queue***). Pre slanja ***work request*** V8 će konvertovati JavaScript objekte u C/C++. V8 nije ***thread-safe***, tako da je puno bolja ideja da se to uradi u ovom sloju. Nakon što se izvrši funkcija u odvojenom thread-u, libuv će vratiti rezultate natrag u V8 objekat i pozvati JavaScript ***callback***.

5.2. Niti (*Threads*)

Thread*-ovi* se koriste interno za imitiranje asinhrone prirode svih sistemskih poziva. libuv takođe koristi **thread**-ove da omogući izvođenje asinhronih zadatka koji su zapravo blokirajući. Danas postoje dve dominatne ***thread*** biblioteke: Windows i POSIX **pthreadsa**. libuv biblioteka je analogna sa **pthreadsa** i ima sličnu semantiku. Značajan aspekt libuv thread mehanizma je da je samostalan. Dok ostale komponente zavise od **event loop**-a i **callback**-a, thread-ovi su potpuno agnostične.

Libuv thread API je vrlo ograničen jer su semantika i sintaksa ***thread***-ova različite na svim platformama, sa različitim nivoima potpunosti. Postoji samo jedan **event loop,** pokrenuta u jednom ***thread***-u **(main thread)**. Drugi **thread**-ovi ne komuniciraju sa ***event loop.*** Standardna veličina libuv **thread pool**-a je 4, ali se može promeniti kroz Node aplikaciju sa **process.env.UV_THREADPOOL_SIZE**.

5.3. Radni red čekanja (*Work queue*)

Work queue omogućava pokretanje zadatka u posebnom **thread**-u i vraća **callback** koji se aktivira kad je zadatak je završen. **Libuv** ima uv_queue_work funkciju koja omogućuje drugim bibliotekama da koristite **event loop** paradigmu. Kad aplikacija koristi **event loop**, imperativ je da osigura da nema funkciju koja blokira petlju pri obavljanju I/O ili je previše CPU zahtevna, jer to znači da će petlja usporiti.

5.4. Procesi

Libuv nudi kreiranje podprocesa (***child process*** upravljanje), omogućuje komunikaciju s ***child process****-om* pomoću strimova linije ili imenovanih kanala. Čest argument protiv ***event-based*** modela je da on ne može iskoristiti prednost višejezgarnih procesora. U **multi-threaded** programu kernel može obavljati raspoređivanje i dodeliti različite thread-ove različitim jezgarima. Ali obzirom da ***event loop*** ima samo jedan **thread**, rešenje je pokretanje više procesa, svaki proces na zasebnom CPU jezgru i u svakom procesu pokrenut ***event loop***.

5.5. Slanje signala procesima

Libuv podržava standardni **kill** sistemski poziv na Unix-u i implementira jedan sa sličnom semantikom na Windows-u, s jednom razlikom: svi SIGTERM, SIGINT i SIGKILL, dovode do prestanka rada tog procesa. SIGINT - obično se dešava kada korisnik pritisne CTRL + C.

6. V8

V8 je Google ***open source engine*** visokih performansi, napisan u C++ i koristi se u Google Chrome browser-u. V8 je dizajniran za brzo izvršenje velikih JavaScript aplikacija i može se pokrenuti samostalno ili može biti ugrađen u bilo koju C++ aplikaciju. V8 omogućuje bilo kojoj C++ aplikaciji da izloži svoje objekte i funkcije JavaScript kôdu.

V8 je implementira ECMAScript definisan u ECMA-262 standardu i podržan je na Windows, Mac OSX i Linux sistemima koji koriste IA-32, x64 ili ARM procesore. V8 podžava sve vrste podataka, operatore, objekte i funkcije navedene u ECMA standardu, i prvobitno je dizajniran kako bi se povećale performanse izvršenja JavaScript-a unutar ***web browser***-a. Kako bi se dobila brzina, V8 prevodi JavaScript kôd u efikasniji mašinski kôd, umesto korišćenja ***interpreter***-a.

V8 prevodi JavaScript kôd u mašinski kôd prilikom izvršenja pomoću JIT (Just-In-Time) **compiler**-a, upravlja alokacijom memorije, pokreće ***garbage collector*** (sakupljač smeća) za objekte koje više ne koristi. Glavna razlika između V8 i ostalih implementacija je da V8 ne proizvodi binarni kôd ili bilo međujezički kôd.

V8 je sastavljen od nekoliko osnovnih komponenti:

- **Base compiler** (osnovni prevodilac), koji analizira JavaScript i generiše nativni mašinski kôd pre nego što je izvršen i taj kôd na početku nije visoko optimizovan.

- V8 predstavlja objekte u objektni model. Objekti su predstavljeni kao asocijativni nizovi u JavaScript-u, ali u V8 su predstavljeni kao ***hidden classes*** (skrivene klase), koje su interni sistemski tip za optimizovanu pretragu.

- ***Runtime profiler*** (optimizator izvršavanja) koji nadgleda sistem koji se pokreće i prepoznaje "vruće" funkcije (tj. kôd koji se dugo izvršava).

- ***Optimizing compiler*** (optimizujući prevodilac) ponovo prevodi i optimizuje "vrući" kôd pomoću ***runtime profiler***-a, optimizacije kao što su ***inlining*** umetanje.

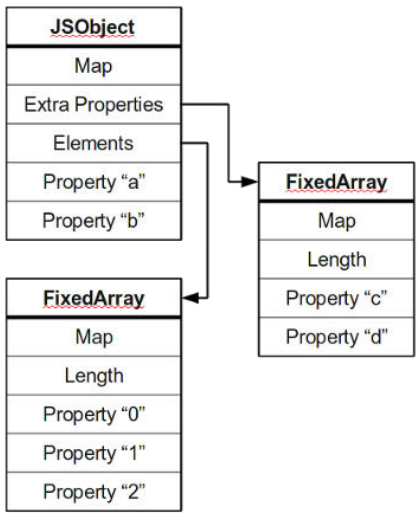
- V8 podržava deoptimizaciju, što znači da prevodilac za optimizaciju može izvući kôd koji se generiše ako otkrije da su neke od pretpostavki koje je napravio o optimizovanom kôdu bile previše optimistične.

- ***Garbage collector***. Razumevanje kako on radi može biti jednako važno kao i optimizacija JavaScript-a.

U nekoliko ***benchmark*** testova (testovi brzine), V8 je mnogo puta brži nego JScript (Internet Explorer), SpiderMonkey (Firefox) i JavaScriptCore (Safari). Koliko je veliko poboljšanje zavisi od toga koliko JavaScript-a je izvršeno i priroda tog JavaScript. Na primer, ako se funkcije u našoj aplikaciji pokreću više puta, poboljšanje performansi će biti veće nego ako se mnogo različitih funkcija pokreću samo jednom.

6.1. Prikaz Objekata (*Object representation*)

U V8, postoje tri primitivne vrste: brojevi (**Numbers)**, logičke promenljive (**Boolean**) i znakovni nizovi (**String)**. Brojevi imaju dva oblika: SMall Integers (SMI), koji su 31-bitni celi brojevi (***integers***), ili normalni objekti u situacijama kao što su parovi (***big numbers***) ili brojevi sa proširenim svojstvima. **String**-ovi imaju dva oblika: jedan je unutar dinamičke memorije (***heap***), a drugi je izvan dinamičke memorije. Takođe postoje i drugi objekti, kao što su nizovi (***arrays***) i nativni ili izvorni objekti, koji nisu u samom ***heap***-u i nisu deo ***Garbage Collector*** obrade.



Slika 4. Object representation

Većina objekata drži sve svoje atribute u jednom bloku memorije ("a" i "b") Svi blokovi memorije imaju pokazivač ka Mapi, koja opisuje njihovu strukturu. Imenovani atributi koji se ne uklapaju u objekat obično se čuvaju u ***overflow array*** ("c" i "d"). Brojivi atributi čuvaju se odvojeno, obično u susednom nizu. Ovaj dijagram pokriva samo najčešći, optimiziran prikaz. Postoji nekoliko reprezentacije koje obrađuju različite slučajeve.

JavaScript standard omogućava programerima da definišu objekte na vrlo fleksibilan način. Objekat je u osnovi skup atributa, uglavnom ključ-vrednost parova. Možemo im pristupiti pomoću dve različite vrste izraza:

```
obj.prop
obj["prop"]
```

Slika 5. Definicija objekta – primer 1

Prema specifikaciji, imena ključeva su uvek stringovi. Ako ćemo koristiti naziv koji nije string, on se implicitno pretvara u niz. Zbog toga, možemo imati i vrednosti sa negativnim indeksom polja.

```
obj[1];    //
obj["1"];  // names for the same property
obj[1.0];  //

var o = { toString: function () { return "-1.5"; } };

obj[-1.5];  // also equivalent
obj[o];     // since o is converted to string
```

Slika 6. Definicija objekta – primer 2

Nizovi u JavaScriptu su objekti sa “magičnim” poljem ***length*** koji vraća broj najvećeg indeksa plus 1. Na primer:

```
var a = new Array();
a[100] = "foo";
a.length;              // returns 101
```

*Slika 7. Primer upotrebe **length** metode*

Interno postoje dve vrste nizova:

- Nizovi u obliku heš tabele (Dictionary, hash table)
- Nizovi za brz pristup (Fast Property Access): linearno skladištenje

6.1.1. Heš tabele

V8 koristi heš tabele da predstavi objekte za koje optimizovani prikaz nije moguć. Međutim, pristup heš tabeli je puno sporiji nego pristup polju na poznatom indeksu. Heš tabele u V8 su veliki nizovi koji sadrže atribute i vrednosti. U početku, svi atributi i vrednosti u heš tabeli su nedefinisani. Kada je ključ-vrednost par umetnut

u tabelu, heš ključ je generisan i koristi se kao početni indeks unošenja. Pošto su stringovi jedinstveni, heš kôdovi se računaju najviše jedanput. Upoređivanjem ključeva po jednakosti su brze operacije u najčešćem slučaju. Međutim, ove rutine nisu uvek trivijalne i njihovo izvršavanje svaki put kada se čita ili upisuje nova vrednost je sporo. V8 će izbeći korišćenje ovog prikaza, kad god je to moguće.

6.1.2. Nizovi za brz pristup - linearno skladištenje

JavaScript je dinamičan programski jezik i novi ključevi se mogu dodati ili obrisati iz objekta u toku rada. U strukturi podataka kao što je heš tabela, pristupanje atributima u JavaScriptu obično puno sporije nego pristupanju instanci varijable u programskom jeziku kao što je Java. U Javi, instance varijabli se nalaze na fiksnim pozicijama koje je odredio prevodilac. Kako bi smanjili vreme potrebno za pristup JavaScript atributima, V8 ne koristi dinamičko pretraživanje. Umesto toga, V8 u pozadini dinamički stvara “skrivene klase” - ***hidden class.***

Kada imamo dva ili više odvojena ali slična objekta u JavaScriptu, oni ne dele zajedničke strukture. ***Hidden classes*** se ponaša kao dodatni objekat koji deklariše broj zajedničkih atributa između ovih različitih objekata. Ova ideja nije nova - ***prototype-based*** programski jezik **Self** koristi Mape da kako bi postigao nešto slično. U V8, objekat menja svoju ***hidden class*** kada se doda novi atribut.

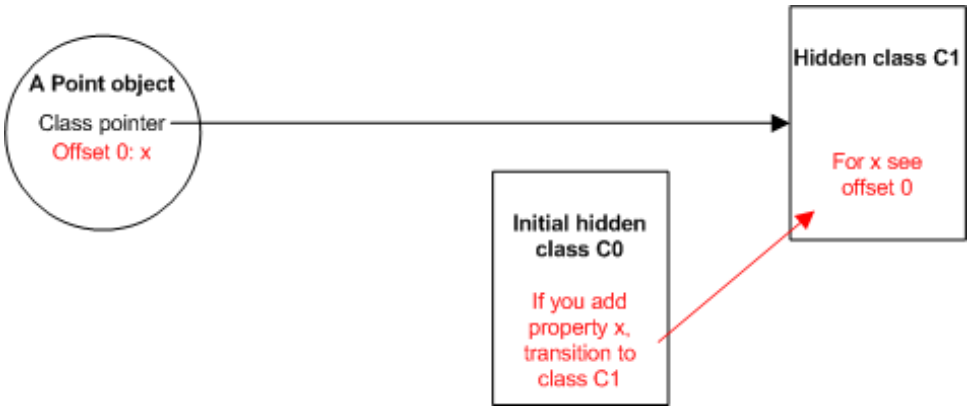
```
function Point(x, y) {
  this.x = x;
  this.y = y;
}
```

Slika 8. Point objekat

Kada izvršimo **new Point(x, y)** novi **Point** objekat je kreiran i kada se desi prvi put, V8 kreira inicijalnu ***hidden class*** od **Point**, nazvanu **C0** u ovom primeru. Kako objekat jos uvek nema nijedan definisan atribut, inicijalna klasa je prazna.

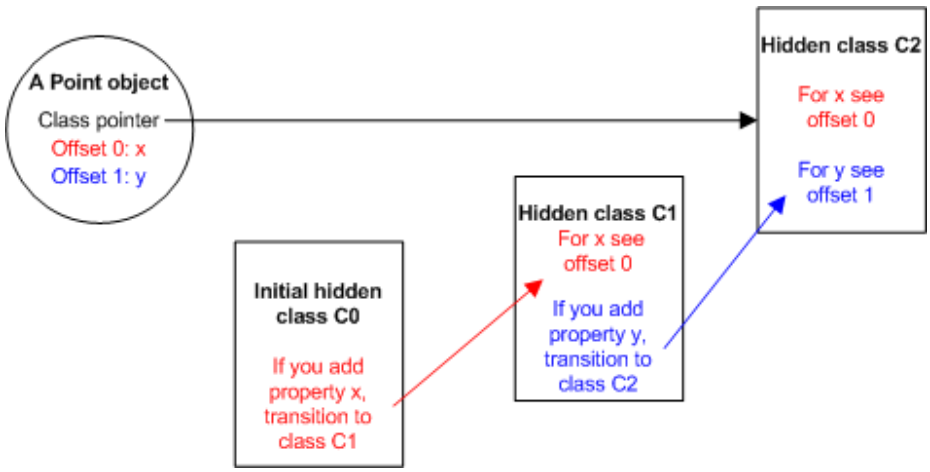
*Slika 9. Prikaz **C0 hidden class** Point objekta*

Izvršavanjem prve naredbe iz klase **Point (this.x = x;)**, V8 kreira još jednu ***hidden class* C1**, baziranu na **C0**, a zatim dodaje informacije C1 koji opisuje objekat koji ima jedan atribut, **X**, čija je vrednost sačuvana na poziciji 0 u Point objektu. Zatim ažurira C0 s klasom tranzicije i pokazuje da ako je atribut X dodat na objekat opisan od C0 tada ***hidden class*** C1 treba koristiti umesto C0. U ovoj fazi ***hidden class*** Point objekta je C1.



*Slika 10. Prikaz **C1 hidden class** Point objekta*

Izvršavanjem druge naredbe iz klase **Point (this.y = y;)** , V8 kreira još jednu ***hidden class* C2**, baziranu na **C1**, a zatim dodaje informacije C2 koji opisuje objekat koji ima jedan atribut, **Y**, čija je vrednost sačuvana na poziciji 1 u Point objektu. Zatim ažurira C1 s klasom tranzicije i pokazuje da ako je atribut Y dodat na objekat opisan od C1 tada ***hidden class*** C2 treba koristiti umesto C1. U ovoj fazi ***hidden class*** Point objekta je C2.



*Slika 11. Prikaz **C2 hidden class** Point objekta*

S obzirom na klase prelaza ***hidden class*** se mogu ponovno iskoristiti. Sledeći put kada se kreira nova klasa **Point**, neće se kreirati nove ***hidden class***, umesto

toga novi **Point** objekat deli klase sa prvim **Point** objektom. ***hidden classes*** omogućava pristup atributima objekta bez korišćenja heš tabela i omogućuje V8 da koristiti ***inline caching***.

6.1.3. Metode I prototipovi

JavaScript nema (prave) klase, što znači da pozivi metoda rade drugačije nego u jezicima kao što su C++ ili Java. Metode u JavaScriptu su obični atributi. U primeru ispod, **distance** je samo atribut **Point** objekata. On pokazuje na funkciju ***PointDistance***.

```
function Point(x, y) {
  this.x = x;
  this.y = y;
  this.distance = PointDistance;
}

function PointDistance(p) {
  var dx = this.x - p.x;
  var dy = this.y - p.y;
  return Math.sqrt(dx*dx + dy*dy);
}
```

Slika 12. Primer Point I PointDistance metoda

Ukoliko bi ***distance*** bila tretirana kao normalno polje u objektu, to bi trošilo puno memorije, jer svaki ***Point*** objekat će imati dodatno polje koje pokazuje na istu stvar.

C++ rešava taj problem s **V-table** (virtuelne tabele). **V-table** su nizovi pointera na virtualne metode. Svaki objekat klase sa virtualnim metodama ima pokazivač na **V-table** za tu klasu. Kada pozovemo virtualnu metodu, program učitava adresu metode iz **V-table**. U V8 postoji slična struktura podataka koja može da ima sličnu ulogu - Mape. Da bi Mape ponašale kao **V-table**, moramo dodati novu vrstu deskriptora - konstantne funkcije. Konstantna funkcija ukazuje na to da objekat ima atribut s određenim imenom, a vrednost tog atributa je sačuvana unutar samog deskriptora, a ne u objektu.

Prelaz se može pratiti samo na Mapi sa konstantnom funkcijom deskriptora sve dok je funkcija koja se dodeljuje je ista ona unutar deskriptora. Dakle, ako je programer ponovo dodelio **PointDistance** varijabli drugu vrednost, prelaz više neće raditi, a mi ćemo morati da kreiramo novu Mapu. V8 ne učitava konstantne funkcije direktno iz Mape kao što to radi **V-table**, umesto toga, optimizovani JavaScript kôd proverava da li objekat ima određenu Mapu, a ako ima, zna da ima određenu konstantnu funkciju. Pokazivač na funkciju je ugrađen direktno u optimizovani kôd.

JavaScript pruža još jedan način za zaobilaženje problema zajedničkih atributa. Svaki konstruktor ima prototip objekat povezan sa njim. Instance konstruktora ponašaju se kao da imaju sve atribute sadržane u njihovim prototipovima. To je ujedno i kanonski način za implementaciju nasleđivanja, jer svaki prototip objekat može imati svoj prototip objekat. V8 će predstaviti metode prototipa pomoću konstantne funkcije deskriptora. Pozivanje metode prototipa može biti malo sporije od poziva "svoje" metode, jer prevedeni kôd mora proveriti ne samo Mapu primalaca, nego i Mape od svog prototip lanca.

6.1.4. Numerisani atributi (*Numbered properties*)

U većini slučajeva za instance istih konstruktora, isti atributi se dođeljuju u istom redosledu. To ne mora biti slučaj sa numerisanim atributima (elemenati kojima se pristupa sa indeksima polja). Po specifikaciji, sva imena atributa su stringovi, i druge vrednosti su konvertovane u stringove pre nego što mogu biti korišćene kao ključevi.

U V8, elementi se čuvaju odvojeno od imenovanih atributa. Svaki objekat ima pokazivač ka svojim elementima, a Mapa objekata pokazuje koji su elementi sačuvani. Mapa ne sadrži opise pojedinih elemenata, ali oni mogu sadržati prelaze inače istih Mapa sa različitim vrstama elementa. Najčešće, objekti imaju brze elemente (***fast elements***), što znači da su elementi sačuvani u susedni niz. Postoje tri vrste ***fast elements***: *fast small integers, fast doubles, fast values*.

JavaScript ne pruža mehanizam koji bi odredio koliko će elemenata objekat da sadrži. Možemo uraditi npr. ***new Array (100)***, ali ovo važi samo za nizove objekata. Ako sačuvamo vrednost indeksa koji ne postoji, V8 će realocirati elemente niza i kopirati stare elemente na veći blok memorije.

V8 pruža generalni interfejs za objekte i koristi odgovarajuće strukture podataka u zavisnosti od upotrebe. Ta prilagodljivost čini da virtualne mašine imaju prednost nad kompajliranim jezicima koji zahtevaju od programera da eksplicitno definišu kako će objekti biti predstavljeni.

6.2. V8 prevodilac (*compiler*)

Kada govorimo o JavaScript ***engine***-u, ono na šta obično mislimo, je njegov ***compiler*** (kompajler), program koji napisani kôd u određenom jeziku pretvara u takozvani mašinski kôd, koji naš računar može da razume.

JavaScript se smatra programskim jezikom visokog nivoa, što znači, da čovek može lako da ga čita i razume. Takođe on poseduje i veliki nivo fleksibilnosti. Posao računara je da pretvori taj programski jezik visokog nivoa u nativne kompjuterske instrukcije.

Najčešće, jednostavni kompajler (**compiler**), ima proces od četiri koraka. ***Lexer*** (Leksička analiza), ***parser*** (parsiranje), ***translator*** (prevodilac) i ***interpreter*** (tumačenje).

- ***Lexer*** skenira izvorni kôd i automatski ga pretvara u atomske jedinice, zvane ***token*** (žeton). To se najčešće postiže obrascem upoređivanja, koristeći regularne izraze.

- Kôd koji je prošao ***Lexer*** analizu se dalje parsira da bi se identifikovala i kodirala njegova struktura i obim u nešto što se zove ***syntax tree*** (sintaksno drvo).

- Onda se ta struktura nalik na grafikon prosleđuje prevodiocu, da bi bila pretvorena u ***bytecode*** (binarni kôd). Najprostija implementacija bi bila nalik velikoj ***switch*** petlji koja mapira ***token*** u svoj ***bytcode*** ekvivalent.

- ***Bytecode*** se dalje prosleđuje ***bytecode interpreter***-u koji ga pretvara u nativni kôd i izvršava.

Ovo je klasični dizajn ***compiler***-a i postoji već niz godina. Međutim, zahtevi ***desktop*** aplikacija su dosta drugačiji od onih u ***web browser***-u. Ova klasična arhitektura pati od određenog broja nedostataka. Inovativni načini kako su ovi nedostaci rešeni rezultat su trke za što većom brzinom ***web browser***-a.

6.2.1. JIT prevodilac (*compiler*)

Glavni problem sa klasičnom arhihekturom je taj što je izvršno vreme ***bytecode*** tumačenja sporo. Performanse mogu da se poboljšaju dodavanjem koraka kompajliranja u kome se ***bytecode*** pretvara u mašinski kôd. Nazalost, čekanje od nekoliko minuta da bi se ***web*** stranica u potpunosti kompajlirala, neće da načini vaš ***web browser*** veoma popularnim.

Rešenje je takozvani ***lazy compilation*** (lenjo prevođenje) od strane JIT-a kompajlera. Kao što ime sugeriše, on prevodi delove kôda u mašinski kôd, u momentu kada je dati deo kôda potreban. JIT kompajleri dolaze u različitim kategorijama, svaka sa svojim strategijama za optimizaciju. Neki, kao na primer kompajleri iz kategorije kompajlera sa regularnim izrazima, su posvećeni optimizaciji

jednog zadatka (izvršavanje jedne komande u jednom momentu), dok neki drugi mogu da optimiziju opšte operacije kao što su ciklusi (petlje) ili funkcije. Moderan JavaScript ***engine*** uključuje u sebe nekoliko ovih kompajlera, koji svi rade zajedno na poboljšavanju performansi našeg kôda.

6.2.2. JavaScript JIT prevodilac (*compiler*)

Prvi JavaScript JIT **compiler** bio je TraceMonkey (Mozilla). Ovo je bio takozvani ***tracing*** JIT ("prateći JIT"), iz razloga što je pratio putanju kroz kôd i tražio kôd koji se najčešće izvršava. Ove takozvane "vruće petlje" su onda prevedene u mašinski kôd. Samo sa ovom optimizacijom, Mozzila je postigla ubrzanje od oko 20% do 40% u odnosu na svoj prethodni ***engine***.

Ubrzo nakon što je TraceMonkey pokrenut, Google je prvi put predstavio svoj Chrome ***web browser*** zajedno sa novim V8 JavaScript ***engine***-om. V8 je bio dizajniran sa ciljem da ubrza izvršavanje. Ključna odluka u arhitekturi dizajna je bila da u potpunosti preskoči ***bytecode*** generisanje i da umesto toga prevodilac emituje nativni kôd.

Za manje od godinu dana od puštanja V8 engine, njegov tim je implementirao registar alokacije, poboljšao ***inline caching*** (linijsko keširanje) i ponovo su napisali njihov engine za regularne izraze, što ga je ubrzalo za 10 puta. Ovo je ubrzalo globalnu brzinu izvršavanja Javascripta u njihovom ***engine***-u za 150%.

U skorije vreme, ***web browser***-i su uveli optimizovane kompajlere sa dodatnim korakom. Posto je ***Directed Flow Graph*** (sintaksno drvo), bilo generisano, ***compiler*** može da koristi tu informaciju da izvrši dodatne optimizacije u zavisnosti od mašinskog kôda. Mozilin IonMonkey i Googlov Crankshat su primeri ovih DFG kompajlera. Krajnji cilj ovih genijalnih projekata je da dovedu izvršavanje JavaScript-a na istu brzinu sa C programskim jezikom. U poslednjih pet godina, učinak JavaScript-a je porastao neverovatno brzo, uglavnom zbog prelaska sa interpretatora na JIT kompajlere u VM (virtuelnim mašinama) Javascript-a. Ovo je značajno povećalo korisnost Javascript-a i web aplikacija uopšteno.

6.2.3. Generacija dinamičkog mašinskog kôda

V8 je bio jedan od prvih JavaScript **engine**-a koji je generisao i izvršavao nativni kôd. On kompajlira sav JavaScript kôd u nativni pre nego ga izvrši. Kod njega ne postoji interpreter i bytecode-a. Najčešće funkcije nisu kompajlirane sve dok nisu

prvi put pozvane. Tako da i ako uključimo neku veliku biblioteku, VM neće gubiti vreme da iskompajlira delove koji se ne koriste.

Nekorišćenje interpretatora znači da nema generičkog potraživanja polja datog objekta. Mašinski kôd nalazi vrednosti direktno iz skrivenog objekta. Ovo je urađeno prvenstveno kako bi se povećala brzina izvršenja Javascript-a dok on ostaje nezavistan od platforme. Pristup poljima objekata se vrši pomoću ***inline caching***-a (linijskog keširanja) kôda koji može biti dopunjen sa drugim mašinskim instrukcijama kako se V8 izvršava.

Tokom inicijalnog izvršavnja kôda za pristup poljima datog objekta, V8 određuje sve trenutne skrivene klase objekta. V8 optimizuje pristup poljima tako što pretpostavlja da će određena klasa biti korišćena za sve buduće objekte kojima se pristupa u istoj sekciji kôda i koristi informacije u klasi da bi dopunila ***inline cache*** kôd da koristi skrivene klase. U slučaju da V8 pretpostavi tačno, polju se dodeljuje vrednost (ili izvučena) u jednoj operaciji. Ako je pretpostavio pogrešno, V8 dopunjava kôd da skloni optimizaciju.

```
point.x
```

*Slika 13. Primer Javascript kôda koji pristupa polju **x** iz objekta Point*

```
# ebx = the point object
cmp [ebx,<hidden class offset>],<cached hidden class>
jne <inline cache miss>
mov eax,[ebx, <cached x offset>]
```

Slika 14. V8 generiše sledeći kôd za pristup x-u iz objekta Point

Ako se skrivena klasa objekta ne podudara sa keširanom skrivenom klasom, izvršavanje se prebacuje na V8 ***runtime*** (sistem izvršavanja) koji barata promašajima unutar ***inline cache*** i dopunjava ***inline cache*** kôd. Ako postoji podudaranje, koje je najčešći slučaj, vrednost atributa **x** je prosto vraćena.

Kada postoji puno objekata sa istom skrivenom klasom, iste prednosti su dobijene kao i kôd većine statičnih jezika. Kombinacija korišćenja skrivenih klasa (***hidden class***) kako bi se pristupilo poljima sa ***inline cache*** i generisanja mašinskog kôda optimizuje u slučajevima gde se isti tip objekta često kreira i pristupa na sličan način. Ovo u velikoj meri poboljšava brzinu u kojoj većina JavaScript kôda može da bude izvršena.

Zašto ne **bytecode**? Uzmimo u obzir proces kompajliranja oba.

Kompajliranje bytecoda:

- Analiza sintakse (parsiranje)
- Analiza obima
- Prevođenje sintaksnog drveta u bytecode.

Nativno kompajliranje:

- Analiza sintakse (parsiranje)
- Analiza obima
- Prevođenje sintaksnog drveta u nativni kôd.

U oba slučaja, moramo da rasčlanimo izvorni kôd i da proizvedemo apstraktno sintaksno drvo. Moramo da izvršimo polje analize, koje će nam reći da li se svaki simbol odnosi na lokalnu varijablu, varijablu vezanu za određeni kontekst, ili globalno polje. Jedini deo koji je drugačiji je korak prevođenja. Ako želimo da kompajler bude što brži mogući, moramo da uradimo direktno prevođenje - svaki čvor sintaksnog drveta bi bio preveden u fiksni niz **bytecode**-a ili nativne instrukcije.

Implementacija nativnog interpretera za ***bytecode*** bila bi petlja koja dohvata sledeći ***bytecode***, uz pomoć **switch** naredbe izvršava fiksni niz instrukcija. Postoje razni načini za poboljšanje, ali svi oni svode na istu strukturu. Umesto generisanja ***bytecode***-a i korišćenje petlje ***interpreter***-a, emituju se odgovarajući fiksni nizovi instrukcija za svaku operaciju. To uklanja potrebu prevođenja i pojednostavljuje prelaze između neoptimizovanog i optimizovanog kôda. Tačno tako radi ***full compiler.***

6.2.4. Ubrzanje optimizovanog kôda (*Inline Caches IC*)

Najčešća optimizacija u savremenim JavaScript prevodiocima je ***inline-caching***. Kôd generisan od ***full compiler***-a koristi ***Inline-Caches*** (ICS) za obnavljanje znanja o tipovima, dok program radi. Ovo nije nova tehnika, prvi put je implementirana pre 30 godina za Smalltalk prevodilac.

Dinamičko tipiziranje u JavaScript-u je ono što omogućuje jednom istom objektu da u jednom trenutku bude tipa **Number**, a drugom trenutku tipa **String**. Nažalost, ova svestranost zahteva od ***compiler***-a da kreira mnogo više tipova provera što je dosta sporije od provere kôda određenog tipa.

Cilj ***Inline Caching***-a je da efikasno obrađuje tipove, tako što će, dok se kôd izvršava, proveriti da li je pretpostavka tipa tačna, a zatim koristiti ***inline cache***. Međutim, to znači da će opracije koje prihvataju više tipova biti manje efikasne.

Implementacija IC naziva se ***Stub***. ***Stub*** se ponaša kao funkcija u smislu da se funkcije mogu pozvati i da će vratiti rezultat, ali ne moraju nužno postaviti okvir ***stack***-a i pratiti cela pravila pozivanja.

Stubs obično nastaju u hodu, ali za zajedničke slučajeve, oni mogu biti keširani i ponovo upotrebljeni višestrukim ICS. ***Stub*** koji implementira IC obično sadrži optimizovani kôd koji obrađuje tipove operanada na koje je određeni IC naišao u prošlosti što je razlog zašto se zove ***cache*** (skladište).

Ako ***Stub*** naiđe na slučaj koji ne može da identifikuje, Stub ga propušta i poziva C++ izvršni kôd koji obrađuje slučaj, generše novi ***Stub*** koji se može izvršiti za trenutni propušten slučaj, kao i sve prethodne. Poziv na stari Stub je prepisan da zove novi Stub, a izvršenje se nastavlja kao da je Stub normalno pozvan.

Kada **full compiler** prvi put generiše kôd za određenu funkciju, IC počinje u neinicijalizovanom stanju, koristeći prost ***Stub*** koji ne obrađuje nijedan slučaj. Prvi put kada se ***Stub*** pozove, on će "propustiti", a ***runtime*** će generisati kôd za izvršenje bez obzira na slučaj koji je prouzrokovao “propust”.

Svaki objekt ima pokazivač ka mapi, što je uglavnom nepromjenjiva struktura koja opisuje strukturu objekta. Stub će proveriti mapu objekta sa poznatom mapom (mapa koju je video kada je ***Stub*** “propustio”) da brzo proveri da li objekat ima željeno polje na tačnom položaju. Ova provera mape omogućuje nam da izbegnemo skupe pretrage ***hash*** tabele.

Ako je polje odgovarajuće, prethodno generisani ***Stub*** optimizovanog mašinskog kôda se može ponovno iskoristiti. Ako su ***hidden class*** ulaza uvek isti to znači da su oni monomorfni. Ako su se struktura ili tip promenili, npr. neki od argumenata su promenili tip tokom različitih poziva, onda su oni polimorfni. Nakon što je ***compiler*** shvatio strukturu kôda i tipova podataka unutar njega, postaje moguće izvesti čitav niz drugih optimizacija:

- **Linijska ekspanzija (inline expansion - inlining) -** Pozivi funkcije su računski skupi, jer oni zahtevaju neku vrstu pretraživanja pa upiti mogu biti spori. Ideja iza ***inline expansion*** je da se telo funkcije zove i “izbaci” kôd na mesto u kojoj je funkcija pozvana. Time se izbegavaju grananja i ubrzava kôd, ali na račun dodatne memorije.

- **Optimizacija varijabli unutar petlje (hoisting) -** Petlje su glavni kandidat za optimizaciju. Premeštanje nepotrebnog računanja van petlji može uveliko poboljšati performanse. Najčešći primer za to je ***for-loop*** iteracija nad sadržajem niza. Računski je suvišno izračunati dužinu polja prilikom svake iteracije. Duzina može biti unapred sačuvana u varijabli van petlje.

- **Preklapanje konstanti (constant folding) - *Constant folding*** je praksa preračunavanja vrednosti konstanti ili varijabli koje se ne menjaju tokom trajanja programa.

- **Eliminacija zajedničkih podizraza (common-subexpression) -** Slično ***constant folding***, ova optimizacija radi skeniranjem kôda za izraze koji evaluiraju iste vrednosti. Ti izrazi se onda mogu zameniti sa varijablom koja drži unapred izračunate vrednost.

- **Eliminacija "mrtvog" kôda (dead-code elimination) - *Dead code*** je definisan kao neiskorišćeni ili nedostpan kôd. Ako postoji funkcija u programu koja se nikada ne poziva onda optimizacije takvog kôda nema smisla. Takve funkcije se eliminišu, što takođe smanjuje ukupnu veličinu programa.

6.2.5. Potpuni prevodilac (*Full Compiler*)

Full compiler generiše "dobar kôd" za JavaScript ali ne sjajan JIT kôd. Cilj ovog prevodioca je da se kôd generiše brzo. Da bi se postigao svoj cilj, on ne radi bilo koju vrstu analize i ne zna ništa o tipovima. On očekuje da će se tipovi varijabli promeniti u toku izvršenja, koristeći Inline Caches ili "IC" strategije za obavljanje znanja o tipovima, dok je program pokrenut. IC je vrlo efikasan i donosi oko 20 puta poboljšanje brzine. Takođe koristi i ugrađeni profiler za odabir "vruće" funkcija koje će biti optimizovane pomoću Crankshaft-a.

6.2.6. Optimizirajući prevodilac (*Optimizing Compiler*)

Paralelno sa ***full compiler***-om, V8 ponovno prevodi "vruće" funkcije (tj. funkcije koje su pokrenute više puta) pomoću Crankshaft-a. Ovaj prevodilac uzima tipove iz ***Inline Cache*** i donositi odluke o tome kako će optimizovati kôd bolje.

Crankshaft operacije direktno smešta tamo gde su one pozvane. To ubrzava izvršenje ali isto tako i druge optimizacije. Crankshaft nije u mogućnosti da optimizuje sve funkcije. Konkretno, optimiziranje trenutno nije moguće na funkcijama sa try {} catch {} blokovima!

Pošto V8 stvara novu ***hidden class*** skrivenu klasa za svako polje, kreiranje takvih klasa treba svesti na minimum. Da biste se to postiglo, programeri bi trebali da izbegavaju dodavanje polja nakon kreiranja objekta i uvek inicijalizovati članove objekta u istom redosledu (kako bi se izbeglo stvaranje različitih skrivenih klasa).

6.2.7. Deoptimizacija

V8 takođe podržava deoptimizaciju podataka: Ako Crankshaft optimizer napravi pretpostavke koje nisu validne (pretpostavke o različitim tipovima iz Inline Cache). Na primer, ako generisana ***hidden class-a*** nije ono što se očekivalo, V8 odbacuje optimizovan kôd i vraća se na ***full compiler*** da dobije novi tip iz ***Inline Cache***. Ovaj proces je spor i treba izbegavati menjanje funkcija nakon što su optimizovane.

6.3. Sakupljač smeća (*Garbage Collector*)

Svaki programski jezik koji koristi memoriju zahteva mehanizam za rezervisanje i oslobađanje prostora. U C i C++ je postignuto sa malloc() and free() i svaki programer je odgovoran za oslobađanje memorije koja se više ne koristi. Mnogi programski jezici imaju sopstveni sistem koji je zadužen za to, koji pomaže programerima da ne moraju da vode računa o tome. To upravljanje se vrši pomoću ***Garbage Collector***-a. Node.js i JavaScript se kompajliraju u izvorni kôd pomoću V8. To znači da programer ne može da rezerviše ili oslobodi memoriju u JavaScript-u, V8 koristi Garbage Collector mehanizam da o tome brine.

Garbage Collector pojednostavljuje korišćenje jezika time što programer ne mora da se bavi problemima vezanim za memoriju već može da se fokusira na razvoj same aplikacije. ***Garbage Collector*** je najvažniji deo u upravljanju memorijom, koji odgovoran za oslobađanje memorije koja više nije u upotrebi tako što povremeno gleda za rashodovane reference objekata i oslobađa memoriju povezanu sa njima.

Najčešća tehnika koju koristi ***Garbage Collector*** je “Praćenje Referenci i brojanje” (***Monitoring Reference Counting***). To znači da ***Garbage Collector*** za svaki objekat ima brojač koliko se puta referencirao. Kada je taj broj nula on može biti oslobođen iz memorije. Time što programer ne mora da vodi računa o memoriji, V8 otklanja mogućnost pravljenja nekoliko vrsta grešaka: ***Dangling pointer***, ***Double free, Buffer overflows, Memory leaks***.

6.3.1. Viseći pokazivač (*Dangling pointer*)

Dangling pointer su pokazivači koji ne pokazuju na validan objekat odgovarajućeg tipa. To su reference koje ne vode do validne destinacije tj. postaju trajno nedostupni. ***Dangling pointer*** se pojavljuje tokom uništenja objekta, kada objekat koji ima dolazeću referencu je obrisan ili alociran bez promene vrednosti pokazivača, pa pokazivač još uvek pokazuje na mesto u memoriji koje je već obrisano.

Sistem može da relocira ranije oslobođenu memoriju drugom procesu, a ako izvorni program zatim dereferencira viseći pokazivač, to može dovesti do nedefinisanog ponašanja, jer memorija sada može da sadrži potpuno različite podatke. U ovom slučaju više delova programa menjaju isti memoriski blok. Ova greška može biti veoma teška za nalaženje.

6.3.2. Dvostruko oslobađanje (*Double free*)

Ovo se dešava kada se oslobodi mesto u memoriji a onda se oslobodi ponovo. U međuvremenu ono može biti ponovo dodeljeno i korišćeno od nekog drugog dela aplikacije, uništavajući pristup tom bloku. Ovaj problem je sličan prethodnom gde dva dela upravljaju istim blokom. U ovom slučaju jedan će pokušati da koristi dok će drugi obrisati podatake.

Kada se program pozove free() metod dva puta sa istim argumentom, struktura podataka postaju oštećene i mogu dopustiti zlonamernom korisniku pisanje vrednosti u proizvoljnim memorijskim prostorima. Ova korupcija može uzrokovati pad programa, u nekim okolnostima i menjanje toka izvršavanja. Prepisivanjem određenog registra ili memorijskog mesta, napadač može prevariti program u izvođenju kôda.

6.3.3. Prekoračenje bafera (*Buffer overflows*)

Ovo je anomalija kada program u toku upisivanja u ***buffer*** premašuje granice ***buffer***-a i pregazi susedne memorijske lokacije. U ovom slučaju program dodeli memorijski blok negde gde treba više prostora nego što ima. Ovde se dogodi da se ne detektuje i ponovo dodeli potreban prostor.

Buffer overflow može biti pokrenuti ulazima koji su dizajnirani za izvršavanje kôda ili da promene način na koji program radi. To može dovesti do čudnog ponašanja programa, uključujući greške prilikom pristupanju memorije, pogrešne rezultate, pa programa ili kršenje sigurnosti sistema. Dakle, oni su osnova mnogih softverskih ranjivosti i mogu se zlonamerno iskoristiti.

6.3.4. Curenje memorije (*Memory leaks*)

Memory leak je konstantan gubitak raspoložive memorije (curenje memorije). To se događa ako se “izgubi” pokazivač na dodeljeni prostor pre nego što se oslobodio. U ovom slučaju taj prostor ostaje rezervisan do kraja izvršenja programa bez mogućnosti pristupanja ili oslobađanja. Ova greška može dovesti do potpunog zaustavljanja programa. Node.js aplikacije mogu patiti od ovog problema posredno zbog ***Garbage Collector***-a. To obično nije ***Garbage Collector*** krivica, ali je uzrokovana nekim uništavanjem objekta koje se ne odvija kada je trebalo, a to nije teško kada koristimo ***event-driven*** arhitekturu.

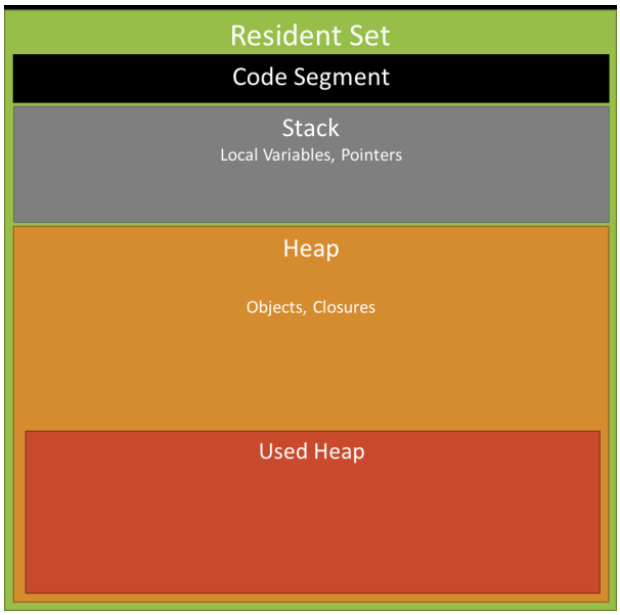
6.3.5. Organizacija memorije

Predstavlja mrežu elementa obično brojeva, stringova i objekata. To se može prikazati u obliku grafikona međusobno povezanih tačaka. Memorija može biti korišćena za čuvanje informacija ili referenci ka drugim objektima. Memorija korišćena od samih objekata zove se **Shallow size**. Koristi se za smeštanje neposrednih vrednost, i obično, samo stringovi i nizovi su značajnih veličina.

Veličina prostora koji će se osloboditi brisanjem objekta zove se **Retained size**. Ta veličina je veličina objekta plus veličina referenciranih objekata koja će se osloboditi dereferenciranjem. Program koji se izvršava uvek je predstavljen kroz neki prostor dodijeljen u memoriji. Ovaj prostor se naziva ***Resident Set***.

V8 koristi sličnu šemu kao Java ***Virtual Machine*** i deli momoriju na segmente:

- Kôd: konkretan kôd koji se izvršava
- ***Stack***: Sadrži sve tipove (integer ili boolean) sa pokazivačima ka referenciranim objektima. Pokazivači definišu kontrolu toka programa.
- ***Heap***: Segment memorije zadužen za smeštanje objekata i stringova.



Slika 15. V8 meorijska šema

U Node.js trenutno korišćenje memorije može se lako dobiti pozivom process.memoryUsage(). Ova funkcija će vratiti objekat koji sadrži:

- RSS - Resident Set Size

- heapTotal - Ukupna veličina ***Heap***-a
- heapUsed - Koliko je zauzet ***Heap***-a

6.3.6. V8 Sakupljač smeća (*V8 Garbage Collector*)

Ako ***Garbage Collector*** nađe objekat koji je još uvek referenciran od strane drugih objekta iako se više ne koristi u aplikaciji, ostaće u ***heap***-u i premestiće ga u ***old space***. Obično objekat živi veoma dug period, od pokretanja aplikacije ili veoma kratak vremenski period opslužujući određenog klijenta. V8 ***Garbage Collector*** je dizajniran da iskoristi prednosti od ova dva najčešće korišćena tipa objekata.

Garbage Collector ciklus obično čisti kratkovečne objekte, a ako misli da su ti objekti i dalje korisni (kada prežive više od jednog ciklusa), premešta ih u veću zonu gde akumulira đubre. Kada ova zona počne da postaje sve veća, trajanje ***Garbage Collector*** ciklusa postaje veće i aplikacija će početi da primećuje kompletne prekide na sekund ili čak nekoliko sekundi. ***Garbage Collector*** ciklus postaje veoma procesorski zahtevan za velike kolekcije objekata. Zbog toga trebamo da posmatramo ***Garbage Collector*** memorijskom upravljanje i da ako je moguće izbegnemo prekomerno korišćenje memorije.

Garbage Collector troši resurse kada pretražuje memoriju koja se koristi ili odlučuje kada će da dereferncira objekte i time pravi nepredvidive pauze tokom izvršavanja aplikacije. Takođe vreme kada se ***Garbage Collector*** startuje je van naše kontrole i to može dovesti do nepredvidivih pogoršanja u performansama. Nema načina za blokiranje ***Garbage Collector***-a, ali ga zato možemo ručno startovati. Programer može manuelno da forsira ***Garbage Collector*** pomoću V8 gc() metode ali ne može kontrolisati kada će se on pokrenuti. ***Garbage Collectors*** troši resurse proporcionalno broju referenciranih objekata, tako da ukoliko ga pokrećemo nakom smanjenog broja referenciranih objekata, možemo smanjiti resurse potrebne za njegov rad. Garbage Collector zaustavlja izvršavanje aplikacije, što utiče na performanse. Da bi se to popravilo V8 koristi 2 tipa ***Garbage Collection***:

- ***Scavenge*** - koji je brz ali nekompletan (Cheney algoritam)

- ***Mark-Sweep*** and ***Mark-compact*** koji je spor ali potpuno oslobađa nereferenciranu memoriju.

6.3.7. Dinamička memorija (Object heaps)

Garbage Collector smešta objekte u ***Object Heap***. U velikom broju programa, životni vek objekata je kratak dok veoma mali broj objekata ima duži životni vek. Da bi iskoristio prednosti ovakvog ponašanja, V8 razdvaja ***Heap*** u dve generacije. Objekti se alociraju u novi prostor (***new-space*)** koji je veoma mali (između 1 do 8MB). Alociranje u ***new space*** je veoma jeftino, dizajnirano da bude brzo analizirano od strane ***Garbage Collectora,*** i ima pointer alokacije (***allocation pointer***) koji se uvećava svaki put kada se rezerviše prostor za novi objekat.

Budući da većina objekata rano umire, V8 strategija omogućava ***Garbage Collector-u*** da obavlja redovna i kratka čišćenja pomoću ***scavenges***, poznatije ***smaller young generation***, bez potrebe za praćenje objekata u staroj generaciji (***old space***). Kada ***allocation pointer*** dostigne maksimalnu veličinu **new-space-a**, poziva se **scavenge** (**Minor Garbage Collection Cycle**) koji brzo briše nekorišćene objekte. Objekti koji su preživeli dva **scavenge** ciklusa**,** jer su još uvek referencirani, promovišu se u **old-space.** Ovde se mogu desiti ***memory leaks***.

Old-space predstavlja đubre skupljeno tokom glavnog ciklusa, **mark-sweep** ili **mark-compact** (**Major Garbage Collection Cycle**) koji ima nekoliko optimizacija kako bi se smanjilo kašnjenje i potrošnja memorije. ***Old space*** može narasti od nekoliko megabajta do gigabajt.

Major Garbage Collection ciklus se aktivira kada određena količina memorije pređe u ***old space***. Trenutak aktiviranja zavisi od veličine **old space** prostora i ponašanja programa. Svaki prostor se sastoji od strana (**pages)**, susednih memorijskih blokova koji sadrže objekte. Svaka strana ima zaglavlja na vrhu i **bitmap** koja govori ***Garbage Collector***-u koje deo strane objekat koristi.

Ovo razdvajanje i pomeranje od jednog do drugog memorijskog prostora stvara određene probleme. Jedan je očigledan - ponovna dodela (***reallocation)***. Drugi je potreba da se zna da li su reference objekta u **new-space** samo u **old-space**. Ova situacija treba da spreči da objekat bude obrisan, ali će i naterati ***Garbage Collector*** da analizira i **old space** što će uticati na performase.

Da bi se ovo izbeglo ***Garbage Collector*** održava listu referenci iz **old-space** u **new-space**. Ona je obično mala pošto je se relativno retko dešava da imamo ovakvu vrstu referenci. Pošto se **scavenges** čišćenje dešava često ono mora biti i brzo. Međutim ako većina objekata preživi **scavenge,** dužina ovog ćićenja može biti značajno duža. ***New-space*** koristi strategiju dodele ***semi-space*** i podeljen je u dve jednake veličine, ***to-space*** i ***from-space***. Većina alokacija se desi u ***to-space***, ali

postoje određene vrste objekata, kao što su izvršni kôdovi koji se uvek alociraju u ***old-space***. Kada se ***to-space*** ispuni, **scavenge** će premestiti objekte u ***from-space***. Objekti koji su premešteni nakon što su već promovisani u **old-space**, smatraju se **long-living**. Jednom kada se objekti pomere, novi **semi-space** postaje aktivan i svi preostali neaktivni objekti postaju odbačeni.

Pošto Garbage Collector ne radi konstantno, između ciklusa objekat može biti kreiran, uništen i dereferenciran nekoliko momenata kasnije. Glavni cilj ***Garbage Collector***-a je da identifikuje đubre u memoriji. Ovo se odnosi na memorijske blokove koje aplikacija više ne koristi. Kada je identifikovana ova memorija može biti ponovo iskorišćena ili oslobođena.

Da bi obezbedio brze alokacije objekata, kratke **Garbage Collection** pauze i izbegao rasipanje memorije, u svakom trenutku ciklusa, ***Garbage Collector*** pauzira V8 izvršavanje (**stop-the-world**)**,** znajući tačno gde su svi objekti u memoriji i koje reference postoje.

Stop-the-world znači da V8:

- Zaustavlja izvršavanje programa tokom rada ***garbage collection*** ciklusa.
- Procesira samo delove **object heap** u **garbage collection** ciklus. Ovo minimizuje uticaj stopiranja aplikacije.
- Uvek zna gde su svi objekti i pokazivači u memoriji. Time se izbegava pogrešno identifikovanje objekata koje može dovesti do ***memory leaks***.

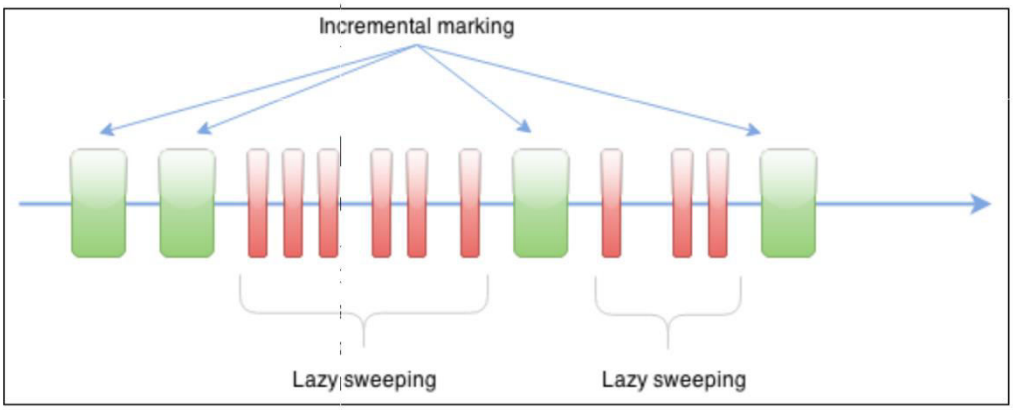
Ako postoji previše referenci, ***Garbage Collector*** će procesirati samo deo **object heap**, minimizujući uticaj pauze.

Scavenge algoritam je odličan za sakupljanje male količine memorije ali ima veliki trošak prostora pošto nam treba fizička memorija za podršku **to-space** and **from-space**. Ovo je prihvatljivo sve dok je **new-space** mali ali je nepraktično koristiti ovaj pristup za više od nekoliko megabajta podataka. Za sakupljanje **old space**, koji može da sadrži nekoliko stotina megabajta podataka se koriste **Mark-sweep** and **Mark-compact**.

U oba algoritma, ***Garbage Collector*** prolazi kroz ***stack*** i označava referencirane objekte. Slika 16. pokazuje kako V8 skenira objekte, obeležava nereferencirane (prvi red u crvenom), odbacuje ih sa liste (drugi red) i kompletira listu brisanjem praznog prostora između objekata (treći red).

*Slika 16. Prikaz **Mark-sweep** skeniranja*

Nakon toga, može se koristiti **mark-sweep** da samo počisti objekte koji nisu bili dostupni oslobađanjem memorije, ili koristiti **mark-compac**t da preraspodeli i kompresuje. Problem kod ova dva algoritma je što mogu trošiti dosta vremena ako koriste za velike dinamičke memorije sa dosta podataka.



Slika 17. Inkrementalno markiranje

Google je 2012. godine predstavio unapređenje koje je značajno smanjilo pauze ***Garbage Collection*** ciklusa. Predstavili su inkrementalno markiranje (**incremental marking)** koje omogućava da dinamička memorija bude obeležena u više manjih koraka. ***Incremental marking*** počinje kada dinamička memorija dostigne određenu kritičnu veličinu. Svaki put kada se aktivira izvršenje se pauzira da bi se izvršilo postepeno obeležavanje. Kada se ***incremental marking*** završi počinje ***lazy sweeping***.

Svi objekti postaju obeleženi kao “mrtvi” ili “živi”, i dinamička memorija tačno zna koliko memorije treba da bude očišćeno. Sva ova memorija ne mora da bude očišćena istog trenutka, odlaganje neće ničemu nauditi. Ne mora da očisti sve u istom trenutku, umesto toga Garbage Collector ih čisti po potrebi dok sve ne očisti. Kada ih sve očisti Garbage Collection ciklus je kompletan i ***incremental marking***

može početi opet. Google je takođe nedavno uveo podršku za **parallel sweeping.** Od tada glavni izvršni ***thread*** neće dirati “mrtve” objekte, strana može biti očišćena sa zasebnim **thread**-ovima sa minimalnom sinhronizacijom.

7. NODE.JS MENADŽER PAKETA

Prava snaga svakog programskog jezika ili platforme je velika zajednica programera. Pored ugrađenih modula moguće je preuzeti veliki broj nezavisnih (**third party**) modula, dostupnih za preuzimanje uz pomoć konzolnog programa pod nazivom NPM (**Node package manager**).

Koncept NPM ima jako puno sličnosti sa **Ruby gem**-ovima i omogućava Javascript programerima deljenje kôda koji su kreirali za rešavanje određenih problema, kao i drugim programerima da taj kôd iskoriste u svojim aplikacijama. Ti komadići za višekratnu upotrebu kôda nazivaju se paketi, ili ponekad moduli. Paket je samo direktorijum sa jednom ili više datoteka u njemu, koji ima fajl "**package.json**" sa meta podacima o paketu. Tipična aplikacija, kao što su **web** stranice, zavise od nekoliko desetina ili stotina paketa. Ovi paketi su često mali a ideja je da svako može da kreira mali deo kôda koji rešava određeni problem. To omogućava programerima da napišu veća, prilagođena rešenja iz više manjih paketa.

NPM predstavlja suštinu modularnog ekosistema Node.js. Ovi moduli uključuju se u program korišćenjem funkcije **require**, koja kao argument prima ime modula ili lokaciju fajla gde je on smešten. Ukoliko putanja postoji, vraća nazad objekat koji može da se koristi kao interfejs za korišćenje tog modula.

Ukoliko dođe do kolizije naziva između modula koji treba da bude uključen i nekog od modula iz jezgra, prioritet ima podrazumevani modul iz osnovne biblioteke. Zadatak NPM modula je da omogući pretraživanja i instalacije dostupnih modula koji se nalaze na udaljenoj lokaciji. NPM moduli mogu biti instalirani globalno ili lokalno, kao što se može zahtevati i određena, konkretna verzija modula. Prilikom instalacije određenog NPM modula dodatno se instaliraju i moduli koji su neophodni za njegov rad. U **root** folderu projekta, u fajlu **package.json**, definisani su svi moduli koji se koriste u projektu, pored ostalih osnovnih informacija vezanih za projekat.

```
{
        "name": "ime aplikacije",
        "version": "1.0.0",
        "scripts": {
                "start": "node server.js"
        },
        "dependencies": {
                "express": "4.2.0",
                "jade": "*"
        }
}
```

Name je jedinstveno ime modula ili aplikacije, **version** predstavlja semantičku verziju paketa i preporučljivo je da se počne sa verzijom 1.0.0. **Script** sadrži dodatne komande koje će se izvršavati u toku životnog ciklusa aplikacije, gde **key** vrednost prestavlja **event**, **value** predstavlja komandu koja se tada izvršava, a u navedenom NPM pokreće izvršenje **server.js** skripte. **Dependencies** se koristi da bi se naveli moduli koji će se preuzeti sa NPM registra i koristiti u aplikaciji. **Dependencies** kao najrelevantniji segment, predstavlja objekat koji sadrži **key** - **value** parove (**key** je ime modula, **value** je verzija modula).

Kao najpopularniji NPM moduli izdvajaju se **express web framework**, **mongodb** koji omogućava API za MongoDB bazu, **underscore utility** biblioteka, **forever** alat koji kroz komandnu liniju kontroliše Node procese i osigurava da aplikacija radi neprekidno.

8. ZAKLJUČAK

U radu je predstavljena i analizirana Node.js platforma koja se odlikuje velikom brzinom i efikasnošću kao i glavne komponente i arhitektura na kojima je Node.js platforma izgrađena. U opštem prikazu Noda opisani su motivi za njegov nastanak, osnovne karakteristike, način na koji Node tehnologija rešava problem skalabilnosti pomoću Libuv sistemske biblioteke, kao i pregled njegovih prednosti i mana. Dat je detaljan opis Google V8 **engine**-a, strategije koje on koristi da bi efikasno kompajlirao i optimizovao JavaScript kôd, kao i njegova uloga u optimizaciji i odrzavanju objekata u memoriji.

Treba napomenuti da i pored svih prednosti Node tehnologije, ona neće u potpunosti zameniti najpopularnije web servere. Ova tehnologija ima svoju sferu primene i većina njegovih ograničenja potiče iz svesne odluke i kompromisa napravljenih sa specifičnim ciljem. Node.js pruža alternativu postojećim aplikativnim sistemima u smislu lakšeg razvoja i postizanja boljih performansi i fleksibilnosti. Posmatrano sa tog aspekta, potpuno je jasno zašto je Node. js tehnologija postala najpopularnija kada je u pitanju razvoj fleksibilnih i brzih client-server aplikacija.

Node.js uživa veliku popularnost i pozicioniran je na zavidan položaj u odnosu na konkurenciju, zbog odgovora na izazove performansi i skaliranja. Node.js ne predstavlja instant rešenja svih mogućih izazova, i nije pogodan za korišćenje u svim mogućim scenarijima. Međutim, u polju njegove namene, može da postigne neverovatne rezultate. Iako Node.js predstavlja zrelu tehnologiju, ne možemo reći da je još dostigao svoj vrhunac. To se može videti po promenama koje se uvode prilikom izlaska novih verzija i, posobno po načinu na koji zajednica otvorenog kôda prati i uzima aktivno učešće u njihovom razvoju, što samo ukazuje na činjenicu da se može očekivati još jako puno od ove tehnologije.

9. INDEKS POJMOVA

10. LITERATURA

1. https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript
2. https://en.wikipedia.org/wiki/Node.js
3. https://nodejs.org/en/about/
4. https://en.wikipedia.org/wiki/V8_(JavaScript_engine)
5. http://steve.vinoski.net/pdf/IC-Node.js.pdf
6. https://www.michaelinthe.cloud/thesis.pdf
7. http://java.ociweb.com/mark/JavaUserGroup/NodeCore.pdf
8. http://matthewhalpern.com/publications/eve-micro-2015.pdf
9. http://www.racunarstvo.matf.bg.ac.rs/MasterRadovi/2014_Aljosa_Sljuka/rad.pdf
10. https://repozitorij.mathos.hr/islandora/object/mathos%3A43/datastream/PDF/view
11. http://www.academia.edu/23674866/Izrada_web_aplikacije_za_prodaju_putem_interneta_na_osnovi_Node.js_tehnologije
12. https://arenli.com/architecture-of-node-js-internal-codebase-57cd8376b71f#.ft3e3qu2d
13. http://blog.kashishgupta.in/2016/05/01/node-js-internal-architecture/
14. https://www.ykode.com/2015/10/10/under-the-hood-1-nodejs.html
15. http://buytaert.net/a-history-of-javascript-across-the-stack
16. https://github.com/nodejs/node/blob/master/doc/topics/blocking-vs-non-blocking.md
17. https://nodesource.com/blog/why-asynchronous/
18. https://docs.npmjs.com/how-npm-works/packages

19. https://rclayton.silvrback.com/speaking-intelligently-about-java-vs-node-performance
20. http://science.webhostinggeeks.com/obilazak-v8
21. http://scg.unibe.ch/archive/masters/Flue14a.pdf
22. http://cs.au.dk/~jmi/VM/GC.pdf
23. http://thibaultlaurens.github.io/javascript/2013/04/29/how-the-v8-engine-works/
24. https://github.com/v8/v8/wiki/Design%20Elements
25. http://jayconrod.com/posts/55/a-tour-of-v8-garbage-collection
26. http://v8project.blogspot.rs/2015/08/getting-garbage-collection-for-free.html
27. https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)
28. http://mcgill-csus.github.io/student_projects/Submission2.pdf
29. https://blog.chromium.org/2010/12/new-crankshaft-for-v8.html
30. http://jayconrod.com/posts/54/a-tour-of-v8-crankshaft-the-optimizing-compiler
31. http://wingolog.org/archives/2011/07/05/v8-a-tale-of-two-compilers
32. http://mrale.ph/v8/resources.html
33. http://nikhilm.github.io/uvbook/An%20Introduction%20to%20libuv.pdf
34. https://www.future-processing.pl/blog/on-problems-with-threads-in-node-js/
35. http://thlorenz.com/learnuv/book/index.html
36. http://www.journaldev.com/7462/node-js-architecture-single-threaded-event-loop
37. http://khan.io/2015/02/25/the-event-loop-and-non-blocking-io-in-node-js/
38. https://www.racunarskemreze.com/Knjiga?Module=BookOnline__Reader&Chapter=145
39. http://blog.mixu.net/2011/02/01/understanding-the-node-js-event-loop/
40. https://developer.mozilla.org/en/docs/Web/JavaScript/EventLoop