UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Integracija mehanizma praćenja u Ethereum konsenzus klijent visokih performansi

master rad

Autor:
Vukašin Marković

Mentor:
prof. dr Filip Marić

Beograd, 2026.

Komisija za odbranu

Mentor

prof. dr Filip Marić

Univerzitet u Beogradu, Matematički fakultet

Predsednik

dr Aleksandar Veljković *(van fakulteta)*

Član

prof. dr Saša Malkov

Univerzitet u Beogradu, Matematički fakultet

Sažetak

Ovaj rad se bavi unapređenjem sistema Grandine, produkcijskog klijenta za Ethereum konsenzusni sloj, napisanog u programskom jeziku Rust. Praktični deo rada sproveden je kao autorov doprinos razvoju klijenta Grandine u okviru međunarodnog programa Ethereum Protocol Fellowship, integrisan je u razvojni tok klijenta i, počev od verzije Grandine 2.0.0, deo je njegovih zvaničnih produkcijskih izdanja.

Konsenzusni klijenti predstavljaju kritičnu infrastrukturu Ethereum mreže, jer moraju neprekidno da izvršavaju konsenzusne operacije u visoko asinhronom okruženju i uz stroge zahteve pouzdanosti. U takvom okruženju razumevanje ponašanja sistema i dijagnostikovanje problema predstavljaju značajan izazov za tradicionalne mehanizme logovanja.

U Grandine je integrisan strukturirani sistem za praćenje izvršavanja programa (engl. *tracing*). Postojeći sistem logovanja zamenjen je jedinstvenim mehanizmom koji omogućava strukturirano beleženje događaja i očuvanje konteksta kroz asinhrone tokove izvršavanja. Integracija je obuhvatila i mrežni sloj konsenzusnog klijenta, implementiran kao zasebna biblioteka, čime su i događaji nastali u ovoj komponenti uključeni u isti mehanizam praćenja. Pomoću zajedničkog pretplatnika, događaji iz različitih komponenti sistema mogu se obrađivati kroz jedinstvenu konfiguraciju.

Implementirani su namenski makroi za beleženje događaja i slojevita arhitektura pretplatnika sa odvojenim kanalima za različite vrste dijagnostičkih podataka. Uveden je i mehanizam za izmenu nivoa praćenja tokom rada čvora bez njegovog restartovanja, čime je omogućeno prilagođavanje količine prikupljenih podataka potrebama analize. Pored toga, uveden je zaseban kanal za trajno beleženje kritičnih izuzetaka, uz rotaciju i kompresiju zapisa.

Posebna pažnja posvećena je instrumentaciji asinhronog kôda strukturiranim rasponima (engl. *spans*), koji omogućavaju povezivanje događaja sa kontekstom operacije kroz tok izvršavanja. Pri tome je obezbeđeno njihovo bezbedno korišćenje u asinhronom kôdu, uzimajući u obzir slučajeve u kojima se izvršavanje privremeno prekida i nastavlja nakon `await` tačaka.

Integracija je sprovedena tako da postojeće funkcionalnosti i format ispisa dotadašnjeg sistema za praćenje izvršavanja klijenta ostanu očuvani. Cilj je bio da se ne narušili postojeći kontinuitet rada u produkcionom okruženju.

Ključne reči: distribuirani sistemi, blokčejn, konsenzus, Ethereum, Grandine, Rust, opservabilnost, strukturirano praćenje, tracing, Tokio, asinhrono programiranje.

Sadržaj

Glava 1

Uvod

Distribuirani sistemi uključujući i blokčejn tehnologije predstavljaju jednu od značajnih oblasti savremenog računarstva. U osnovi svakog blokčejn sistema nalazi se problem *konsenzusa*, odnosno postizanja zajedničkog dogovora o stanju sistema među učesnicima koji ne mogu unapred da pretpostave da će ostali učesnici postupati ispravno, pri čemu se mora uzeti u obzir mogućnost otkaza pojedinih učesnika i kašnjenja u komunikaciji.

Jedan od najznačajnijih blokčejn sistema je Ethereum, decentralizovana platforma koja omogućava izvršavanje transakcija i pametnih ugovora i održavanje zajedničkog stanja bez centralnog autoriteta [4, 5]. U Ethereum mreži ostvarivanje konsenzusa povereno je konsenzusnim klijentima. Oni predstavljaju ključnu komponentu Ethereum infrastrukture i njihov rad mora biti pouzdan i stabilan, uz poštovanje vremenskih ograničenja koja nameće protokol.

Sa kontinuiranim razvojem Ethereum ekosistema raste i složenost implementacije protokola, a samim tim i konsenzusnih klijenata. Zbog toga raste i potreba za detaljnim uvidom u njihovo ponašanje. Klijenti se oslanjaju na asinhrono izvršavanje u kom se veliki broj zadataka izvršava istovremeno i uključuje različite komponente sistema. U takvom okruženju logovanje (engl. *logging*), odnosno beleženje događaja tokom izvršavanja sistema kao pojedinačne log zapise, često nije dovoljno za rekonstrukciju toka izvršavanja i povezivanje događaja koji pripadaju istoj operaciji.

Praćenje izvršavanja klijenta treba da omogući da se događajima, pored samog zapisa, pridruže strukturirani podaci i kontekst njihovog izvršavanja. Na taj način prikupljene informacije mogu se povezivati i analizirati pomoću eksternih alata za opservabilnost (engl. *observability*), čime se omogućava detaljniji uvid u tok izvršavanja i ponašanje sistema nego što je to moguće na osnovu pojedinačnih tekstualnih log zapisa.

Jedan od pristupa koji omogućava ovakav način praćenja jeste *tracing*, koji pored poje-

dinačnih događaja uvodi i *raspone* (engl. *span*) za praćenje konteksta i toka izvršavanja operacija. Zbog toga, *tracing* je posebno pogodan za sisteme sa složenim i asinhronim tokovima izvršavanja.

Ovaj rad bavi se integracijom strukturiranog sistema praćenja izvršavanja u produkcijski Ethereum konsenzusni klijent *Grandine*. Implementacija je zasnovana na radnom okviru (engl. *framework*) `tracing` [27] u izvršnom okruženju `tokio` [31], sa ciljem unapređenja mogućnosti praćenja i dijagnostike klijenta, uz očuvanje zahteva u pogledu pouzdanosti i kontinuiteta rada.

Uvedeni mehanizam je deo razvojnog toka klijenta i trajan doprinos koji čini osnovu za dalje unapređenje njegove dijagnostike i opservabilnosti. Kako pouzdanost i mogućnost dijagnostikovanja konsenzusnih klijenata neposredno utiču na stabilnost i dalji razvoj Ethereum protokola, uvedeni mehanizam doprinosi samom protokolu, ali i zajednici koja se na njega oslanja.

Doprinos je uključen u zvanične repozitorijume klijenta [37, 39, 40, 41, 42]. Izdanje 2.0.0 [23] prvo je izdanje koje sadrži rezultate ovog rada, koji od tada čine sastavni deo svih narednih verzija klijenta Grandine. Rad je sproveden u okviru programa Ethereum Protocol Fellowship [33], koji organizuje i finansira Ethereum fondacija, a uloga autora bila je da doprinosi razvoju konsenzusnog klijenta Grandine [35, 34]. Autor je danas alumnista tog programa i deo open-source zajednice koja razvija i održava Ethereum protokol.[1]

1.1 Predmet i cilj rada

Predmet rada obuhvata integraciju centralnog sistema za praćenje izvršavanja, njegovo povezivanje sa različitim komponentama klijenta, kao i instrumentaciju odabranih delova sistema u asinhronom izvršnom okruženju.

Cilj rada je da se uspostavi jedinstven i fleksibilan mehanizam za prikupljanje i obradu dijagnostičkih podataka, koji omogućava detaljniji uvid u izvršavanje klijenta i istovremeno odgovara zahtevima njegovog produkcionog rada. To podrazumeva obezbeđivanje strukturiranog i kontekstno svesnog beleženja događaja, mogućnost upravljanja nivoom praćenja tokom rada klijenta, kao i instrumentaciju asinhronih tokova izvršavanja na način koji ne narušava njihovu bezbednost i očekivano ponašanje. Mehanizam treba da omogući usmeravanje različitih vrsta događaja ka odgovarajućim izlaznim kanalima, u skladu sa njihovom namenom.

Poseban cilj rada je da se navedeni mehanizam integriše uz očuvanje postojećih funkcio-

[1] Rezultate ovog doprinosa autor je prezentovao na svetskom sajmu Devconnect [24] u Buenos Ajresu u novembru 2025. godine, pred liderima industrije i Ethereum zajednice. Snimak izlaganja dostupan je na [25].

nalnosti i formata logovanja na koje se oslanjaju operateri i korisnici klijenta Grandine. Ovaj zahtev je posebno značajan u kontekstu konsenzusnih klijenata koji učestvuju u validaciji, čiji rad zahteva kontinuitet i neometano izvršavanje konsenzusnih dužnosti. Zbog toga uvođenje novog mehanizma praćenja ne treba da zahteva promenu postojećeg načina korišćenja klijenta niti alata koji se nadovezuju na njega. Nova verzija klijenta treba da zadrži postojeće funkcionalnosti, uz istovremeno unapređivanje novim mogućnostima.

Pored same implementacije, rad razmatra odabrane tehničke izazove koji se javljaju pri integraciji sistema za praćenje u složenom asinhronom softverskom sistemu i prikazuje pristup njihovom rešavanju u okviru klijenta Grandine.

1.2 Struktura rada

Rad je organizovan na sledeći način. Poglavlje 2 uvodi problem distribuiranog konsenzusa, a poglavlje 3 ga smešta u kontekst blokčejna i Ethereum konsenzusa, uključujući podelu čvora na konsenzusni i izvršni klijent. Poglavlje 4 predstavlja klijent Grandine, njegovu arhitekturu i asinhroni model izvršavanja, dok poglavlje 5 opisuje polazno stanje logovanja, motivaciju za integraciju i razloge zbog kojih je `tracing` izabran kao rešenje. Poglavlje 6 izlaže inkrementalni pristup radu, a poglavlje 7 izdvaja zahteve i izazove koji su oblikovali integraciju.

Središnji deo rada čini poglavlje 8, koje opisuje samu integraciju sistema za praćenje. U njemu se redom obrađuju jedinstveni pretplatnik i namenski makroi, arhitektura pretplatnika, kontrola nivoa praćenja tokom rada, trajno beleženje kritičnih izuzetaka, prostiranje praćenja na mrežni sloj u biblioteci `eth2_libp2p`, instrumentacija ključnih komponenti i primer zapisa tokom rada čvora. Potom poglavlje 9 opisuje testiranje, a poglavlje 10 zaključuje rad.

Glava 2

Distribuirani konsenzus

Distribuirani sistem čini veći broj računara, odnosno *čvorova*, koji zajedno obavljaju neki zadatak razmenjujući poruke preko mreže. Za razliku od centralizovanog sistema, u distribuiranom sistemu ne postoji zajednička memorija niti jedan čvor koji u svakom trenutku raspolaže potpunim i tačnim pogledom na stanje sistema. Poruke mogu kasniti ili biti izgubljene, pojedini čvorovi mogu prestati da rade, a komunikacija između njih može biti nepouzdana. Uprkos tome, od sistema se često očekuje da kao celina donese doslednu odluku.

Upravo taj zadatak naziva se *konsenzus* – postizanje saglasnosti većeg broja čvorova o jednoj vrednosti ili odluci, tako da svi ispravni učesnici na kraju prihvate isti ishod. Konsenzus je jedan od centralnih problema distribuiranih sistema jer omogućava da se skup nezavisnih čvorova ponaša kao jedinstvena, pouzdana celina i u uslovima otkaza i nepredvidivih kašnjenja.

Težina problema postaje posebno izražena kada nije moguće pretpostaviti da će se svi učesnici ponašati prema pravilima sistema. Ovaj slučaj je ilustrovan *Vizantijskim problemom generala*, koji su 1982. godine formalizovali Lamport, Šostak i Piz [1]. U originalnoj analogiji, grupa generala mora da donese zajedničku odluku o napadu ili povlačenju, pri čemu se komunikacija odvija preko glasnika, a pojedini generali mogu biti izdajice i drugim učesnicima slati protivrečne informacije. Problem se, u opštem obliku, svodi na pitanje kako postići saglasnost ispravnih učesnika kada se deo sistema može ponašati proizvoljno.

Problem distribuiranog konsenzusa dobija novu dimenziju u otvorenim *peer-to-peer* sistemima, odnosno sistemima ravnopravnih čvorova, u kojima učesnici nisu unapred poznati niti postoji centralni autoritet koji određuje ispravno stanje sistema. Jedan od prvih sistema koji je pokazao praktičan način postizanja saglasnosti u takvom okruženju bio je Bitcoin, predstavljen 2008. godine. Uvođenjem mehanizma *Proof of Work* i pravila za izbor lanca, Bitcoin je omogućio učesnicima mreže da bez centralnog autoriteta održavaju zajedničku

istoriju transakcija [3]. Ovaj pristup postavio je temelje za razvoj savremenih blokčejn sistema, koji distribuirani konsenzus koriste za održavanje zajedničkog stanja u otvorenim mrežama.

Glava 3

Blokčejn i Ethereum konsenzus

Blokčejn je oblik distribuiranog sistema u kojem veći broj učesnika zajednički održava stanje sistema bez oslanjanja na centralni autoritet [2]. Za razliku od opšteg modela distribuiranog konsenzusa, blokčejn sistemi organizuju stanje u obliku sekvence blokova koji predstavljaju istoriju promena sistema.

Podaci u blokčejnu organizovani su u blokove koji su međusobno kriptografski povezani u lanac. Konsenzusni mehanizam definiše pravila prema kojima učesnici proveravaju blokove, odlučuju koji od mogućih nastavaka lanca treba prihvatiti i održavaju zajedničko stanje sistema.

Različiti blokčejn sistemi koriste različite mehanizme konsenzusa:

- **Bitcoin** – ***Proof of Work.*** [3] Učesnici koriste računarsku snagu za rešavanje kriptografskog problema čije je rešenje teško pronaći, ali jednostavno proveriti. Učesnik koji prvi pronađe odgovarajuće rešenje može predložiti novi blok, dok ostali učesnici mogu proveriti njegovo rešenje i validnost predloženog bloka. Na taj način računarski resursi određuju ko ima priliku da proširi lanac.
- **Ethereum** – ***Proof of Stake.*** [18] Učesnici konsenzusnog procesa zaključavaju određenu količinu sredstava kao ekonomski ulog i time postaju *validatori*. Protokol na osnovu validatora i njihovog uloga određuje ko učestvuje u predlaganju i potvrđivanju blokova, dok ekonomski mehanizmi podstiču učesnike da postupaju u skladu sa pravilima protokola.

3.1 Blokčejn mreža i čvorovi

Blokčejn mrežu čini skup međusobno povezanih čvorova (engl. *nodes*). Čvor predstavlja program koji se izvršava na računaru učesnika mreže i koji implementira pravila konkretnog

blokčejn protokola. Čvorovi međusobno komuniciraju preko *peer-to-peer* (P2P) mreže, bez centralnog posrednika, i razmenjuju informacije potrebne za održavanje zajedničkog stanja sistema.

U zavisnosti od konkretnog protokola, čvorovi mogu imati različite uloge. Mogu održavati lokalnu kopiju stanja, primati i prosleđivati transakcije, proveravati i obrađivati blokove ili učestvovati u procesu postizanja konsenzusa. Zajedničkim radom takvih čvorova mreža održava jedinstveno stanje, iako nijedan pojedinačni čvor nema ulogu centralnog autoriteta.

Da bi se učesnici mogli usaglasiti oko zajedničkog stanja, protokol mora definisati pravila prema kojima se utvrđuje koje promene stanja i koji blokovi treba da budu prihvaćeni. Ovu ulogu ima *konsenzusni mehanizam*. On određuje način na koji učesnici dolaze do dogovora o stanju sistema i redosledu prihvaćenih blokova, kao i način postupanja u slučaju postojanja više mogućih nastavaka lanca.

3.2 Ethereum kao blokčejn protokol

Ethereum je blokčejn platforma koja, pored obrade i evidencije transakcija, omogućava izvršavanje programa poznatih kao pametni ugovori (engl. *smart contracts*) [6]. Za razliku od sistema koji su prvenstveno usmereni na prenos digitalne imovine, Ethereum održava opšte stanje sistema koje se menja izvršavanjem transakcija i programa.

Ethereum treba razlikovati od njegove konkretne programske implementacije. Protokol je definisan skupom pravila i specifikacija, dok različiti razvojni timovi održavaju nezavisne implementacije tih pravila. Takve implementacije nazivaju se *klijenti* i razvijaju se u različitim programskim jezicima, među kojima su Go, Rust, C#, Java, Nim itd.

Postojanje više nezavisnih implementacija, poznato kao *client diversity* [47], predstavlja važan element otpornosti Ethereum mreže. Greška specifična za jednu implementaciju na taj način ne mora istovremeno uticati na učesnike koji koriste druge implementacije.

U današnjoj arhitekturi Ethereum čvora učestvuju dve komplementarne vrste klijenata:

- **Konsenzusni** (engl. *consensus*) klijent
- **Izvršni** (engl. *execution*) klijent

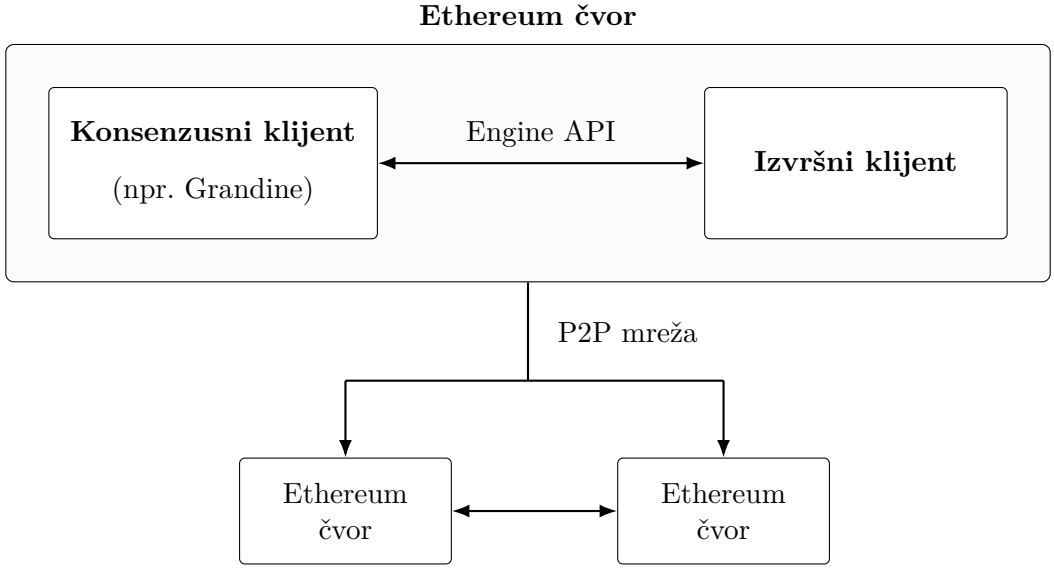
Oni implementiraju različite delove Ethereum protokola i zajednički omogućavaju funkcionisanje čvora.

3.3 Konsenzusni i izvršni sloj

Ethereum je do septembra 2022. godine koristio *Proof of Work* [17], dok je prelaskom na *Proof of Stake*, u okviru događaja poznatog kao *The Merge* [9], promenjen način ostvarivanja konsenzusa. Umesto oslanjanja na računarsku snagu učesnika, novi mehanizam zasniva učešće u konsenzusnom procesu na validatorima i njihovom ekonomskom ulogu u mreži.

U ovoj arhitekturi konsenzusni i izvršni klijent imaju različite, ali međusobno povezane odgovornosti. Izvršni klijent odgovoran je za obradu i izvršavanje transakcija, uključujući izvršavanje pametnih ugovora, kao i za održavanje rezultujućeg stanja sistema. Konsenzusni klijent održava podatke koji opisuju stanje konsenzusa, uključujući informacije o validatorima, njihovim dužnostima i napretku mreže kroz blokove i slotove. Na osnovu tog stanja i pravila konsenzusnog protokola, konsenzusni klijent proverava primljene blokove, učestvuje u izboru kanoničnog lanca i koordinira dužnosti validatora.

Konsenzusni i izvršni klijent međusobno komuniciraju preko *Engine API-ja* [16]. Kada konsenzusni klijent obrađuje blok, podatke potrebne za izvršavanje transakcija prosleđuje izvršnom klijentu. Izvršni klijent izvršava transakcije i utvrđuje rezultat njihovog izvršavanja, koji konsenzusni klijent koristi kao deo provere bloka. Na taj način dva klijenta zajedno ostvaruju funkcionalnost jednog Ethereum čvora, pri čemu su odgovornosti za izvršavanje transakcija i ostvarivanje konsenzusa jasno razdvojene. Slika 3.1 prikazuje odnos konsenzusnog i izvršnog klijenta unutar jednog čvora, kao i međusobno povezivanje čvorova preko *P2P* mreže.



Slika 3.1: Ethereum čvor: konsenzusni klijent (Grandine) i izvršni klijent povezani preko *Engine API*-ja, dok se čvorovi međusobno povezuju preko *P2P* mreže.

Ova podela omogućava nezavisni razvoj dva sloja, ali istovremeno zahteva njihovu pouzdanu koordinaciju. Za potrebe ovog dovoljno je razumevanje značaja konsenzusnog sloja, sloja u

kome se odvijaju vremenski osetljive aktivnosti koje zahtevaju kontinuirano i pravovremeno izvršavanje.

3.4 Ethereum Proof of Stake konsenzus

Konsenzusni sloj Ethereuma zasnovan je na mehanizmu *Proof of Stake* i stanju koje održava Beacon lanac [12, 13]. Za razliku od pristupa zasnovanog na računskoj snazi, u Proof of Stake mehanizmu učesnici konsenzusa su *validatori*, koji zaključavanjem svog uloga stiču pravo da obavljaju konsenzusne dužnosti.

Rad konsenzusnog sloja organizovan je u vremenske intervale koji se nazivaju *slot*-ovi. Svaki slot predstavlja priliku za predlaganje novog bloka, pri čemu protokol određuje validatora koji u datom slotu ima pravo da predloži blok. Veći broj uzastopnih slotova čini *epohu*, koji predstavlja viši nivo organizacije konsenzusnih aktivnosti.

Pored predlaganja blokova, validatori učestvuju u proveri i izboru lanca kroz atestaciju (engl. *attestation*) poruke [20]. Njima validator izražava svoj stav o trenutnom stanju lanca i bloku koji smatra odgovarajućim nastavkom. Konsenzusni klijent prima i obrađuje ove informacije, održavajući lokalni pogled na stanje mreže i koristeći ga u daljem procesu izbora lanca.

Mreža u svakom trenutku može primiti više različitih kandidata za nastavak lanca, na primer usled kašnjenja u komunikaciji ili istovremenog predlaganja blokova. Zbog toga konsenzusni klijent mora da primeni pravila izbora lanca (engl. *fork choice*) kako bi odredio koji se lanac smatra kanoničnim. Ethereum u tu svrhu koristi mehanizam zasnovan na algoritmu LMD-GHOST [8], uz dodatna pravila definisana specifikacijom konsenzusnog sloja.

Pored izbora kanoničnog lanca, Ethereum koristi i mehanizam *finalnosti* zasnovan na principima definisanim algoritmom Casper FFG [7]. Njegova uloga je da, pod uslovima definisanim protokolom, određene delove lanca učini konačnim tako da se njihov status više ne može promeniti u okviru regularnog rada protokola.

Sve navedene aktivnosti odvijaju se kontinuirano i u vremenski ograničenom okruženju. Konsenzusni klijent istovremeno obrađuje podatke pristigle od drugih učesnika, proverava blokove i atestacije, održava konsenzusno stanje, primenjuje pravila izbora lanca i, u slučaju validatora, izvršava dodeljene dužnosti. Ove aktivnosti odvijaju se kroz veliki broj međusobno povezanih i asinhronih tokova izvršavanja.

Takva složenost čini implementaciju konsenzusnog klijenta zahtevnim softverskim sistemom, čije je ponašanje potrebno pouzdano pratiti, analizirati i dijagnostikovati. U nastavku rada pažnja se usmerava na *Grandine*, konkretnu implementaciju Ethereum konsenzusnog

klijenta koja predstavlja osnovu za praktični deo ovog rada.

Glava 4

Grandine: Ethereum konsenzusni klijent

Grandine [21] je nezavisna implementacija Ethereum konsenzusnog klijenta, razvijena u programskom jeziku Rust [32]. Njegova arhitektura organizovana je oko više međusobno povezanih komponenti koje zajedno ostvaruju funkcionalnosti konsenzusnog sloja, uključujući konsenzusnu logiku, mrežnu komunikaciju, upravljanje dužnostima validatora, proizvodnju blokova i Beacon HTTP API. Beacon HTTP API [15] izlaže podatke o blokovima, validatorima, stanju konsenzusa i mrežnim metrikama, i omogućava slanje zahteva poput predlaganja blokova i atestacija.

Grandine može da radi kao Beacon čvor koji prati stanje konsenzusnog sloja i razmenjuje podatke sa drugim učesnicima mreže. Kada se poveže sa odgovarajućim izvršnim klijentom putem Engine API-ja i opremi ključevima validatora, može da učestvuje i u izvršavanju dužnosti validatora [19], uključujući predlaganje blokova i slanje potvrda atestacija. Na taj način Grandine predstavlja konkretnu implementaciju konsenzusnog klijenta čije će arhitektura, model izvršavanja i postojeći sistem logovanja biti analizirani u nastavku rada.

4.1 Arhitektura i organizacija klijenta

Jedna od Grandine karakteristika jeste izražen fokus na performanse, paralelizaciju i efikasno korišćenje računarskih resursa. Takav pristup odražava se i u organizaciji izvornog kôda, koji je podeljen na veliki broj krejtova (engl. *crate*) objedinjenih u okviru jednog Cargo radnog prostora (engl. *workspace*). Krejt predstavlja osnovnu jedinicu kompajliranja i linkovanja u programskom jeziku Rust. Njegov kôd može biti organizovan u više međusobno povezanih modula i može koristiti druge krejtove kao zavisnosti. U zavisnosti od tipa, krejt

se kompajlira u izvršni program ili biblioteku. Na ovaj način se složena funkcionalnost konsenzusnog klijenta deli na manje, funkcionalno zaokružene komponente [22].

Među važnijim komponentama nalaze se `fork_choice_control` i `fork_choice_store`, koji učestvuju u upravljanju izborom grane lanca, kao i `p2p` i `eth2_libp2p`, koji obezbeđuju mrežnu komunikaciju. Komponenta `validator` zadužena je za izvršavanje dužnosti validatora, dok `block_producer` učestvuje u proizvodnji blokova. Komunikaciju sa spoljnim komponentama i korisnicima omogućava `http_api`, dok se različite funkcionalnosti inicijalizacije i pokretanja klijenta oslanjaju na dodatne komponente organizovane u zasebne krejtove.

Ovakva modularna organizacija omogućava da se pojedinačne funkcionalne celine razvijaju i optimizuju nezavisno, ali istovremeno znači da se izvršavanje jedne operacije može prostirati kroz veći broj komponenti. Dodatnu složenost predstavlja činjenica da Grandine u velikoj meri koristi asinhrono i konkurentno izvršavanje.

4.2 Asinhrono izvršavanje

Grandine je implementiran u Rustu i za izvršavanje velikog broja istovremenih zadataka koristi asinhroni model zasnovan na izvršnom okruženju `tokio` [31]. Mrežna komunikacija, obrada konsenzusnih poruka, upravljanje dužnostima validatora i druge aktivnosti mogu se izvršavati istovremeno. Pri tome izvršavanje pojedinačne aktivnosti često obuhvata više komponenti sistema i može biti više puta privremeno prekinuto i nastavljeno usled asinhronih operacija.

Za razliku od sekvencijalnog programa, u kojem se tok izvršavanja može pratiti redosledom operacija, u asinhronom sistemu više zadataka može biti aktivno u isto vreme. Izvršavanje pojedinačnog zadatka može biti privremeno zaustavljeno na tački `await`, dok se u međuvremenu izvršavaju drugi zadaci. Kada se izvršavanje nastavi, događaji koji pripadaju različitim operacijama mogu se pojavljivati međusobno isprepletano.

Ovakav model je važan za performanse konsenzusnog klijenta, jer omogućava efikasno korišćenje računarskih resursa i istovremenu obradu velikog broja mrežnih i konsenzusnih aktivnosti. Međutim, istovremeno povećava složenost analize ponašanja sistema, jer za razumevanje pojedinačne operacije nije dovoljno posmatrati redosled pojedinačnih log zapisa, već je potrebno povezati događaje koji pripadaju istom toku izvršavanja.

Upravo zbog ovakve arhitekture, način na koji se informacije o izvršavanju prate, predstavlja važan deo infrastrukture samog klijenta.

Glava 5

Polazno stanje i motivacija za integraciju

Pre integracije opisane u ovom radu, Grandine nije imao jedinstven mehanizam za praćenje izvršavanja. Postojeći sistem bio je dovoljan za osnovno praćenje rada klijenta, ali je sa povećanjem složenosti sistema postajala potreba za detaljnijim uvidom u njegovo ponašanje.

5.1 Polazno stanje

Postojeći sistem logovanja bio je *hibridan* i zasnovan na dva različita mehanizma. Glavna aplikacija koristila je biblioteku `env_logger` [44], dok je mrežni sloj, implementiran u zasebnoj biblioteci `eth2_libp2p`, koristio biblioteku za strukturirano logovanje `slog` [45]. Kako bi se logovi mrežnog sloja prikazivali zajedno sa logovima glavne aplikacije, ova dva sistema bila su povezana mostom `slog-stdlog` [46].

Strukturirano logovanje predstavlja način beleženja događaja pri kojem se, pored tekstualne poruke, zapisuju i dodatna imenovana polja u obliku parova ključ–vrednost. Na taj način informacije sadržane u zapisu imaju eksplicitnu strukturu, što omogućava filtriranje, pretraživanje i grupisanje događaja na osnovu pojedinačnih polja. Konkretan format prikaza takvog zapisa može se menjati nezavisno od njegove unutrašnje strukture.

Ovakva organizacija omogućavala je objedinjavanje logova različitih delova klijenta, ali je istovremeno uvodila posredni sloj između dva sistema logovanja. Strukturirani logovi generisani u mrežnom sloju prosleđivali su se kroz `slog-stdlog` u sistem standardnog logovanja glavne aplikacije, gde su tretirani kao standardni log zapisi. Time je bilo moguće zadržati jedinstven izlaz za postojeće korisnike klijenta, ali nije postojao jedinstven mehanizam koji bi omogućio sistematsko povezivanje događaja i praćenje konteksta kroz

različite komponente sistema.

Postojeći sistem je stoga bio dovoljan za osnovno praćenje rada klijenta, ali nije pružao mehanizam namenjen detaljnom praćenju složenih asinhronih tokova izvršavanja.

5.2 Motivacija za integraciju

Sa razvojem Ethereum protokola i porastom složenosti funkcionalnosti koje konsenzusni klijenti moraju da podrže, povećala se potreba za detaljnijim uvidom u njihovo ponašanje. Pojedinačni log zapisi nisu pružali dovoljno konteksta za jednostavno povezivanje događaja koji pripadaju istoj operaciji. U sistemu sa velikim brojem istovremenih asinhronih zadataka to je otežavalo rekonstrukciju toka izvršavanja, identifikaciju uskih grla i analizu performansi.

Dodatno ograničenje predstavljala je mogućnost promene nivoa logovanja samo prilikom pokretanja čvora. Za detaljniju analizu problema tokom rada bilo je potrebno promeniti konfiguraciju i ponovo pokrenuti čvor, što nije poželjno u produkcijskom okruženju. Konsenzusni klijent koji je deo čvora konfigurisanog za rad validatora mora pravovremeno da izvršava dužnosti validatora i da ostane povezan sa mrežom. Zbog toga je bilo poželjno omogućiti promenu nivoa praćenja tokom rada, bez potrebe za restartovanjem čvora.

Razvojni tim klijenta Grandine označio je unapređenje mehanizma praćenja kao jedan od prioriteta. U okviru predloženih projekata za šestu generaciju programa *Ethereum Protocol Fellowship* [33], navedena je upravo *Tokio Tracing* integracija [34], sa ciljem uvođenja strukturiranog logovanja, asinhronog profilisanja performansi i bogatije dijagnostike kroz sistem Grandine.

Pri tome je bilo potrebno očuvati postojeći način korišćenja klijenta. Operateri čvorova mogu koristiti alate za nadzor i obradu logova koji zavise od postojećeg formata ispisa, pa promena mehanizma praćenja nije trebalo da zahteva izmene takvih sistema. Navedena ograničenja ukazala su na potrebu za mehanizmom koji bi omogućio detaljnije praćenje izvršavanja i fleksibilniju konfiguraciju, uz očuvanje postojećeg načina ispisa.

5.3 Zašto `tracing` u okruženju `tokio`

Tokio je asinhrono izvršno okruženje (engl. *runtime*) za programski jezik Rust [31] i predstavlja osnovu izvršavanja klijenta Grandine. Kao što je opisano u prethodnom poglavlju, veći broj zadataka može napredovati istovremeno, pri čemu se izvršavanje pojedinačnog zadatka može privremeno zaustaviti na tački `await` i kasnije nastaviti. Posledica toga je da se događaji nastali tokom izvršavanja različitih zadataka mogu međusobno preplitati, iako pripadaju različitim logičkim operacijama.

U takvom okruženju klasično logovanje predstavlja događaje kao međusobno nezavisne zapise. Sam redosled log zapisa zato često nije dovoljan da se utvrdi kojoj operaciji pojedinačni događaj pripada niti da se povežu događaji nastali u različitim delovima istog asinhronog toka. Za detaljnije praćenje izvršavanja potrebno je, pored informacije o tome da se određeni događaj dogodio, sačuvati i kontekst u kojem je događaj nastao.

U ovom radu za tu namenu koristi se `tracing`, radni okvir za instrumentaciju programa napisanih u programskom jeziku Rust i prikupljanje strukturiranih dijagnostičkih podataka [27]. Njegov osnovni model zasniva se na dve vrste informacija:

- *događaji* (engl. *event*) predstavljaju pojedinačne pojave tokom izvršavanja, analogno zapisima u strukturiranom logovanju;
- *rasponi* (engl. *span*) predstavljaju vremenske intervale tokom kojih se izvršava određena operacija ili je aktivan određeni kontekst. Raspon može sadržati strukturirana polja sa dodatnim informacijama i biti ugnježden u drugi raspon, čime se formira hijerarhijska struktura izvršavanja.

Događaji nastali tokom aktivnog raspona mogu se povezati sa njegovim kontekstom, što omogućava da se informacije iz različitih delova istog toka izvršavanja posmatraju kao deo iste operacije kojoj pripadaju.

Pored samog modela događaja i raspona, `tracing` definiše i arhitekturu za njihovu obradu, zasnovanu na razdvajanju mesta na kome se podaci generišu od mesta na kome se obrađuju. Instrumentovani kôd je zadužen za emitovanje događaja i otvaranje raspona, ne znajući šta se sa njima dalje dešava. Njihovu obradu preuzima *pretplatnik* (engl. *subscriber*), centralna komponenta koja prima sve događaje i raspone nastale u programu. Kada koristimo `tracing`, podrazumeva se jedan globalni pretplatnik, koji se postavlja jednom pri pokretanju i kome se potom obraća čitav program.

Pretplatnik može biti sastavljen od jednog ili više *slojeva* (engl. *layer*), pri čemu svaki sloj obrađuje događaje i raspone na svoj način i usmerava ih ka svom izlazu. Svaki sloj može imati i svoj *filter*, koji odlučuje koje će događaje taj sloj primiti, a koje odbaciti. Time prikupljanje informacija u kôdu ostaje odvojeno od njihove dalje obrade i prikaza.

Upravo kombinacija strukturiranih događaja, vremenskih raspona i fleksibilne arhitekture za njihovu obradu čini `tracing` pogodnim za praćenje asinhronog izvršavanja u okruženju `tokio`. Njegova integracija u Grandine i prilagođavanje potrebama konsenzusnog klijenta predstavljaju predmet narednih poglavlja.

Glava 6

Metodologija integracije

Migracija sistema logovanja na `tracing` sprovedena je inkrementalno, kroz niz zaokruženih faza [36]. Ovakav pristup bio je posebno važan jer je Grandine aktivan konsenzus klijent koji se kontinuirano razvija i podleže strogim zahtevima za ispravnost i stabilnost. Umesto da se čitava migracija razvija kao jedna velika izmena koja bi bila primenljiva tek nakon potpunog završetka, svaka faza je projektovana tako da na kraju ostavi klijent u ispravnom i primenljivom stanju. Na taj način je svaka završena celina mogla nezavisno da bude proverena, integrisana i, po potrebi, uključena u narednu verziju klijenta.

Redosled faza pratio je prirodnu zavisnost komponenti:

1. **Osnovna postavka** – uključivanje biblioteke `tracing` kao zavisnosti projekta, konfiguracija `tracing` pretplatnika i podrška za upravljanje filtriranjem `tracing` događaja preko ENV promenljivih iz okruženja.
2. **Migracija glavne aplikacije** – uvođenje specifičnih `tracing` makroa i prevođenje svih logova glavnog repozitorijuma na `tracing`.
3. **Migracija mrežnog P2P sloja** – uklanjanje mosta i serijalizacije i direktno povezivanje biblioteke `eth2_libp2p` na `tracing` pretplatnika glavnog repozitorijuma (odeljak 8.5).
4. **Instrumentacija** – uvođenje raspona nad mrežnim (HTTP API, P2P) i konsenzusno kritičnim komponentama (odeljak 8.6).
5. **Dinamička kontrola `tracing` pretplatnika** – dinamički pretplatnik i HTTP krajnja tačka za izmenu nivoa (odeljak 8.3).
6. **Jedinični testovi** – napisani za testiranje ponašanja praćenja u različitim scentarijima (poglavlje 9).

7. **Sloj izuzetaka** – uveden sloj za događaje izuzetke, `tracing` pretplatnik prebačen na višeslojnu arhitekturu sa slojem za zasebno kanalisanje događaja izuzetaka (odeljci 8.2 i 8.4).

8. **Rotacija zapisa** – ograničenje veličine arhive za kritične izuzetke i njena kompresija (odeljak 8.4).

9. **Istraživanje integracije metrika** – pravci daljeg rada ka sistemima za opservabilnost.

Ostatak rada prati ovu logiku, ali je radi preglednosti organizovan tematski, a ne strogo hronološki.

Glava 7

Zahtevi i izazovi integracije

Uvođenje novog sistema za praćenje u postojeći konsenzusni klijent zahtevalo je više od same implementacije funkcionalnosti. Za razliku od linearnog dodavanja jedne izolovane funkcionalnosti, trebalo je uvesti promenu koja ima dodira sa svakim delom kôda, kôda koji je ostatak razvojnog tima istovremeno i intenzivno menjao i proširivao. Taj posao bio je dodatno požurivan rokovima uoči Ethereum hardforka[1] Fusaka [10], dok je izmene trebalo pažljivo uklopiti u asinhronu arhitekturu klijenta bez ometanja tekućeg razvoja. Pre prikaza same integracije, u nastavku se izdvajaju tehnički i organizacioni zahtevi i izazovi koji su oblikovali rešenja izložena u narednim poglavljima.

Razvojna grana i konflikti pri spajanju. Inicijalno, u glavnom repozitorijumu klijenta uvedena je sporedna grana za razvoj integracije mehanizma za praćenje. Na toj grani, promene su uvođene postepeno u sve delove postojećeg kôda, dok se sam razvoj klijenta istovremeno intenzivno nastavljao, neprekidno ažurirajući glavnu granu repozitorijuma. Održavanje zasebne grane za implementaciju praćenja zato je zahtevalo kontinuirano usklađivanje sa promenama u glavnoj grani. Svaki skup izmena preuzet sa glavne grane morao je biti pregledan i, po potrebi, prilagođen postojećoj implementaciji praćenja, čak i u slučajevima kada prilikom spajanja nije dolazilo do formalnog konflikta.

Koordinacija različitih komponenti. Integracija nije bila ograničena na glavni repozitorijum klijenta. Bilo je potrebno uskladiti promene i sa mrežnim slojem implementiranim u zasebnoj biblioteci `eth2_libp2p`, kao i sa spoljnim zavisnostima koje već koriste `tracing`. Krajnji cilj predstavljalo je obezbeđivanje da događaji iz ovih komponenti mogu da koriste

[1] Hardfork je unazad nekompatibilna izmena pravila Ethereum protokola koja se aktivira u unapred određenom bloku, od kog svi čvorovi u mreži počinju da rade po novim pravilima. Zbog toga implementacije svih klijenata i svih validatora moraju biti ažurirane do tog trenutka, a operateri čvorova moraju preći na nove verzije, jer čvorovi sa starom implementacijom nakon tog bloka više ne mogu da učestvuju u mreži. Pregled dosadašnjih Ethereum hardforkova dat je u [11].

isti sistem praćenja i isti mehanizam upravljanja kontekstom kao i ostatak klijenta. Način na koji je to ostvareno opisan je u odeljku 8.5.

Uticaj na performanse. Mehanizam za praćenje ne sme nepotrebno da opterećuje kritične putanje izvršavanja. Zbog toga je bilo potrebno pažljivo odrediti mesta na kojima se uvode rasponi i izbegavati njihovo prekomerno ugnježđavanje u često pozivanom kôdu. Istovremeno, detaljniji nivoi praćenja treba da ostanu dostupni kada su potrebni, ali da ne predstavljaju uobičajeno opterećenje pri radu sa podrazumevanom konfiguracijom. Konkretne odluke u vezi sa tim opisane su u odeljku 8.6.

Kontrola detaljnosti praćenja. Jedan od izazova bio je osmisliti najpogodniji način za kontrolu detaljnosti praćenja, što je uključilo i konsultacije sa razvojnim timom o stvarnim potrebama operatera i korisnika klijenta Grandine. U dotadašnjem sistemu logovanja *debug* (detaljniji) režim izveštavanja koristio je gotovo isključivo razvojni tim, prilikom otklanjanja grešaka u razvojnim fazama ili istraživanja prijavljenih propusta. Operateri i korisnici klijenta Grandine gotovo se nikada nisu opredeljivali za korišćenje čvora u pomenutom detaljnijem režimu, budući da je to zahtevalo da čvor od samog pokretanja bude fiksiran za taj režim i da tako učestvuje u mreži. Zbog toga je bilo potrebno da se omogući uključivanje detaljnijeg izveštavanja pojedinih komponenti ili celog čvora dok isti već radi u produkciji i obavlja svoje dužnosti. Cilj je bio da ta funkcionalnost bude dostupna u svakom trenutku u kom se posumnja na neuobičajeno ponašanje ili se ukaže potreba za dubljim uvidom u rad klijenta, bilo zbog trenutnog stanja *peer-to-peer* mreže ili zbog zlonamernih napada na čvor ili pak na celokupan konsenzus Ethereum protokola. Rešenje ovog zahteva opisano je u odeljku 8.3.

Složenost asinhronog kôda. Prostiranje konteksta praćenja kroz asinhrone tokove zahtevalo je pažljiv izbor načina instrumentacije. Poseban izazov predstavljalo je pravilno vezivanje raspona za asinhroni tok, kako bi se izbeglo njihovo zadržavanje preko `.await` granica. Zbog toga je bilo potrebno primenjivati različite načine instrumentacije u zavisnosti od toga da li se radi o asinhronom ili sinhronom kôdu. Primenjeni načini instrumentacije opisani su u odeljcima 8.6 i 8.6.1.

Kompatibilnost sa alatima za opservabilnost. Uvođenje strukturiranog praćenja zahtevalo je i pažljiv izbor imena polja, metapodataka i konteksta događaja. Ovi podaci se ne koriste samo pri samom generisanju događaja, već predstavljaju osnovu za njihovu kasniju pretragu, filtriranje i analizu u spoljnim alatima za opservabilnost. Zbog toga je bilo potrebno voditi računa o tome da struktura prikupljenih podataka ostane konzistentna kroz različite komponente klijenta.

Provera ponašanja tokom rada. Automatizovani testovi i *consensus-spec* testovi [14] predstavljali su osnovu za proveru ispravnosti implementacije, ali nisu mogli da obuhvate sva svojstva koja se ispoljavaju tokom rada čvora. Zbog toga je bilo potrebno dodatno proveriti ponašanje klijenta u stvarnim uslovima rada, uključujući pokretanje, sinhronizaciju i redovno izvršavanje. Na ovaj način provereno je da uvođenje mehanizma za praćenje funkcioniše u okviru celokupnog sistema, a ne samo u izolovanim testovima. Postupak testiranja opisan je u poglavlju 9.

Glava 8

Integracija sistema za praćenje izvršavanja

Ovo poglavlje čini jezgro rada i opisuje samu integraciju sistema za praćenje izvršavanja u klijent Grandine. Najpre se uvodi jedinstveni pretplatnik i namenski makroi kojima se emituju događaji, potom arhitektura tog pretplatnika i funkcionalnosti koje ona pruža, dok se na kraju opisuju prostiranje praćenja na mrežni sloj i instrumentacija ključnih komponenti.

8.1 Jedinstveni pretplatnik i makroi

Pretplatnik je, kao što je opisano u odeljku 5.3, opšta komponenta `tracing` sistema na kojoj se obrađuju svi događaji i rasponi. Doprinos ovog dela rada je da se ta komponenta iskoristi za potpunu zamenu starog hibridnog sistema logovanja (`env_logger` u glavnoj aplikaciji i `slog` u mrežnom sloju) jedinstvenim, globalnim `tracing` pretplatnikom [28], konfigurisanim na jednom mestu prilikom pokretanja klijenta. Taj jedinstveni pretplatnik time postaje centralno dijagnostičko čvorište celog klijenta.

Sada bilo koji deo kôda koji koristi `tracing` može, pozivom odgovarajućeg makroa, da emituje odgovarajuće `tracing` događaje za koje je globalni pretplatnik u tom trenutku podešen. Time je uklonjen raniji mehanizam kojim su se informacije iz celokupnog sistema, korišćenjem mosta, serijalizacije i prosleđivanja instance logera dostavljale logeru centralne aplikacije.

Događaji se u `tracing` sistemu ne emituju pozivom funkcije, već pozivom makroa. Makro je u jeziku Rust konstrukcija koja se izvršava tokom prevođenja i tada se razvija u odgovarajući kôd, a od poziva funkcije razlikuje se po znaku uzvika na kraju imena, na primer `info!` ili `error!`. Biblioteka `tracing` nudi po jedan takav makro za svaki nivo,

redom `trace!`, `debug!`, `info!`, `warn!` i `error!`. Svaki događaj emitovan na ovaj način nosi svoj nivo i svoj *cilj* (engl. *target*). Cilj je oznaka porekla događaja, podrazumevano naziv krejta ili modula iz kog događaj potiče, i kasnije služi za njegovo filtriranje i usmeravanje. Upravo na osnovu nivoa i cilja pretplatnik odlučuje da li će i kako obraditi svaki pojedinačni događaj.

8.1.1 Prelazni makroi i kontekst povezanih čvorova

Za konsenzusni klijent, povezani čvor (engl. *peer*) predstavlja drugi čvor mreže sa kojim je klijent uspostavio vezu i preko kojeg razmenjuje informacije potrebne za održavanje konsenzusa. Broj trenutno povezanih čvorova jedan je od važnih pokazatelja povezanosti klijenta sa ostatkom mreže, jer može ukazati na promene u dostupnosti mrežnih veza i pomoći pri tumačenju događaja u mrežnom sloju. Automatsko prilaganje tog podatka svakom događaju bio je i izričit zahtev tima Grandine.

Da bi ta informacija bila dostupna bez ručnog dodavanja u svaki poziv, uveden je globalni brojač `PEER_LOG_METRICS` i skup prelaznih makroa koji ga automatski prilažu svakom `tracing` događaju kao strukturirano polje `peers`.

```
pub static PEER_LOG_METRICS: PeerLogMetrics = PeerLogMetrics::new(0);
#[macro_export]
macro_rules! info_with_peers {
    ( $( $rest:tt )* ) => {
        ::tracing::info!(
            peers = %$crate::PEER_LOG_METRICS,
            $( $rest )*
        )
    };
}
```

Listing 8.1: Globalni brojac povezanih čvorova i primer prelaznog makroa `info` nivoa (krejt `logging`).

Na listingu 8.1 prikazan je jedan takav makro, `info_with_peers!` za `info` nivo. On poziv prosleđuje izvornom `tracing` makrou `info!`, ali mu pre toga automatski dodaje polje `peers` sa trenutnom vrednošću globalnog brojača `PEER_LOG_METRICS`. Oznaka `$rest` u zaglavlju makroa označava da makro prima proizvoljan skup argumenata i prosleđuje ih dalje nepromenjene, dok znak `%` bira tekstualni (`Display`) prikaz vrednosti brojača. Ovi makroi napisani su u `logging` krejtu klijenta Grandine i u kôdu se koriste umesto izvornih `tracing` makroa, čime se broj povezanih čvorova prilaže svakom događaju bez dodatnog rada na mestu poziva.

Analogni makroi implementirani su za sve nivoe (`warn_with_peers!`, `error_with_peers!`,

`debug_with_peers!` i `trace_with_peers!`).

Uvođenje ovog sloja ne donosi nikakvo kašnjenje ni trošak u performansama, jer makroi `*_with_peers` nisu funkcije, nego zamena teksta na nivou prevođenja. Oni se razvijaju u poziv originalnog `tracing` makroa, kome se polje `peers` prosleđuje kao još jedan argument. Ako pretplatnik ne osluškuje na priloženom nivou, do samog emitovanja i obrade `tracing` događaja iz makroa uopšte ne dolazi, kao ni do formatiranja njegovih argumenata, koji se izračunavaju lenjo.

8.1.2 Makro za kritične izuzetne događaje

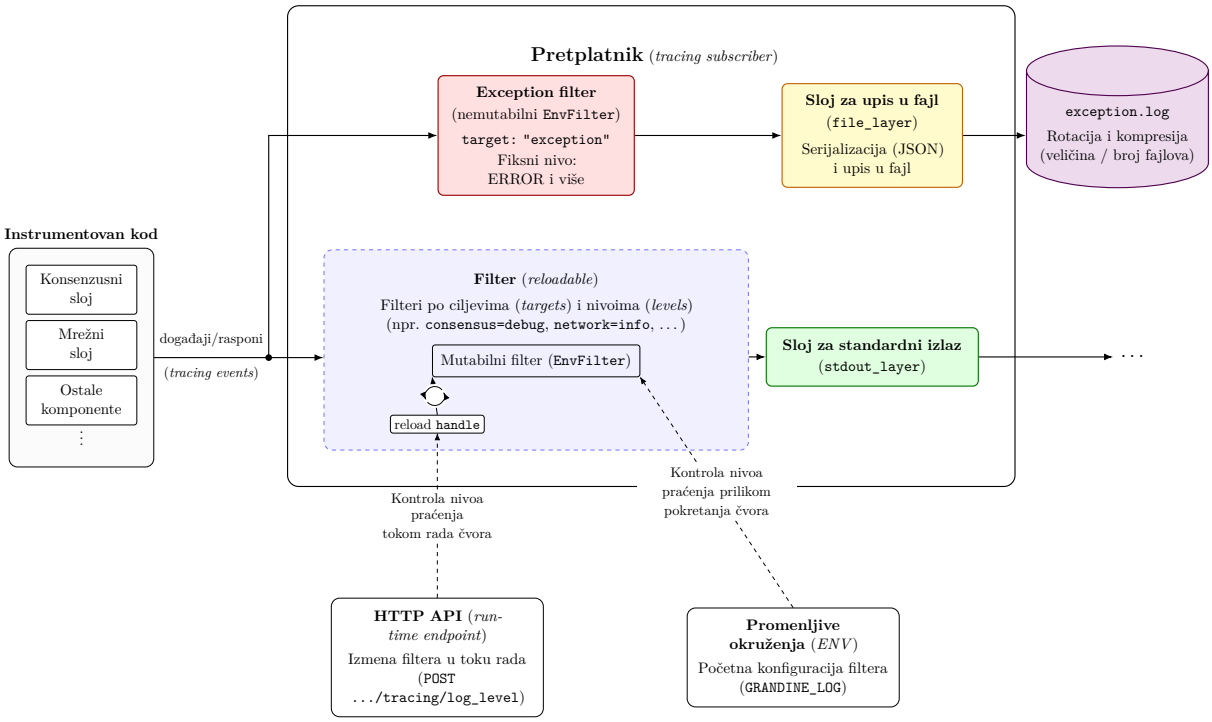
Pored uobičajenih nivoa, uveden je i poseban makro namenjen kritičnim izuzetnim događajima. On je prvobitno nazvan `crit!`, a kasnije preimenovan u `exception!` radi jasnoće (odeljak 8.2). Za razliku od običnog `error!`, ovaj makro koristi ranije opisani pojam cilja tako što svaki događaj emituje na posebnom cilju `exception` (u kôdu `target: "exception"`). Time se takvi događaji kasnije mogu prepoznati po cilju i usmeriti u zaseban izlazni kanal, nezavisno od ostalog toka emitovanih događaja.

8.1.3 Format ispisa

Pri formatiranju se vodilo računa o kompatibilnosti unazad, tako da postojeći sistemi i parseri koje su dotadašnji korisnici klijenta Grandine koristili mogu da nastave da rade bez izmena. Uvođenjem `tracing` biblioteke tekstualno logovanje i njegov format više nisu u centru pažnje kao jedini način za prikaz i obradu dijagnostičkih podataka. Težište se pomera ka mogućnostima koje pruža strukturirano praćenje, pri čemu se generisani podaci mogu prosleđivati spoljnim sistemima za prikupljanje, obradu i analizu. Kao primer takvog sistema može se navesti `OpenTelemetry` [51], koji omogućava standardizovano prikupljanje i prosleđivanje `tracing` podataka, dok `Grafana` [50] omogućava njihovu vizuelizaciju i analizu.

Pored osnovnih funkcionalnosti, prilikom uvođenja `tracing` biblioteke bilo je potrebno uzeti u obzir i njene dodatne mogućnosti koje mogu uticati na kompatibilnost postojećeg načina rada. Jedan od primera je ANSI formatiranje izlaza. Iako je prikaz sa bojama pregledniji u terminalu, u okruženjima koja ne podržavaju ANSI kontrolne znakove njihovo prisustvo može dovesti do neželjenih promena izlaza i poremetiti postojeće sisteme koji se oslanjaju na njegov prethodno definisan format. Zbog toga je implementirano automatsko prepoznavanje terminalskog okruženja, tako da se ANSI formatiranje uključuje samo kada se standardni izlaz koristi kao terminal. Pored automatskog ponašanja, omogućeno je i njegovo eksplicitno uključivanje kada je to potrebno.

8.2 Arhitektura pretplatnika



Slika 8.1: Slojevita arhitektura pretplatnika i izolovani izlazni kanali.

Jedinstveni pretplatnik nije samo zamena za staro logovanje, već i mesto na kome se oblikuje ponašanje celokupnog praćenja. Različite vrste dijagnostičkih podataka imaju različite potrebe. Uobičajeni tok logova namenjen je operateru i treba da bude podesiv tokom rada, dok kritični izuzeci moraju biti trajno sačuvani i dostupni za pružanje na uvid razvojnom timu Grandine u slučaju da nešto pođe po zlu.

Da bi se ti različiti zahtevi pomirili u jednom sistemu, arhitektura pretplatnika počiva na slojevitom modelu sa više izolovanih kanala. Njegova arhitektura, omogućila je izmene detaljnosti praćenja tokom rada čvora, bez restartovanja i bez prekida veze sa mrežom (odeljak 8.3), kao i trajno i predvidivo beleženje kritičnih izuzetaka, sa rotacijom i kompresijom koje sprečavaju nekontrolisan rast zapisa (odeljak 8.4). Celokupan raspored slojeva, njihovih filtera i kontrolnih tačaka za podešavanje praćenja prikazan je na slici 8.1.

Umesto jednog sloja za obradu `tracing` događaja, pretplatnik se sastavlja od više nezavisnih slojeva, od kojih svaki ima sopstveni filter i sopstveni izlaz. Filtriranje se u svakom sloju oslanja na `EnvFilter` [29], gotov filter iz biblioteke `tracing_subscriber` koji događaje propušta ili odbacuje prema njihovom cilju (najčešće naziv krejta ili modula) i nivou, na primer `info` ili `debug`.

Filter jednog sloja može biti nepromenljiv, zadat jednom pri pokretanju, ili promenljiv (*mutabilan*), to jest takav da se može izmeniti. Da bi filter bio promenljiv i dok klijent radi, `tracing_subscriber` nudi *sloj za ponovno učitavanje* (`reload`), koji obavija postojeći

filter i daje upravljač preko kog se filter kasnije menja bez restartovanja.

Konkretan pretplatnik čine dva osnovna sloja.

- **`stdout_layer`** – glavni sloj namenjen operateru, čiji je filter promenljiv i obavijen slojem za ponovno učitavanje, pa se detaljnost praćenja može menjati tokom rada (odeljak 8.3).
- **`file_layer`** – namenski sloj sa trajnim izlazom, čiji je filter nepromenljiv i fiksno podešen samo na kritične izuzetke, koje *uvek* beleži u fajl `exception.log`, nezavisno od trenutnog nivoa glavnog sloja.

Ključna ideja je izolacija kanala. Promena detaljnosti filtriranja jednog sloja ne utiče na to kako drugi sloj filtrira i beleži događaje na koje je on pretplaćen, i obrnuto. Operater tako može da menja detaljnost glavnog sloja a da nikada ne izgubi trag kritičnih izuzetaka, koji ostaju trajno zapisani na disku.

```
// početni filter: sve isključeno, pa po jedna =info direktiva po krejtu
let mut filter = EnvFilter::default()
    .add_directive(LevelFilter::OFF.into())
    .add_directive("fork_choice_control=info".parse()?)
    .add_directive("p2p=info".parse()?)
    .add_directive("eth2_libp2p=info".parse()?)
    // ... po jedna =info direktiva za svaki značajan krejt ...

// GRANDINE_LOG direktive se dodaju na kraj, pa imaju prednost
if let Ok(env_filter) = EnvFilter::try_from_env("GRANDINE_LOG") {
    for directive in env_filter.to_string().split(',') {
        filter = filter.add_directive(directive.parse()?);
    }
}

// filter se omotava u reload sloj -> promenljiv tokom rada
let (filter_layer, handle) = reload::Layer::new(filter);

let registry = tracing_subscriber::registry()
    .with(stdout_layer.with_filter(filter_layer));

// zaseban filter: samo izuzeci, nivo error, uvek u fajl
let exception_filter = EnvFilter::default()
    .add_directive(LevelFilter::OFF.into())
    .add_directive("exception=error".parse()?);

let file_layer = fmt::layer()
    .with_ansi(false)
```

```
29      .with_writer(non_blocking)
30      .with_filter(exception_filter);
31
32  registry.with(file_layer).init();
```

Listing 8.2: Sastavljanje pretplatnika – početni filteri, direktive dodate iz promenljive okruženja, reload sloj i dva izlazna sloja sa nezavisnim filterima (krejt `binary_utils`).

Listing 8.2 prikazuje sastavljanje ovakvog pretplatnika u kôdu. Najpre se gradi početni filter u kome je podrazumevano sve isključeno, pa se za svaki značajan krejt dodaje po jedna direktiva na nivou `info`. Zatim se, ako je zadata, čita promenljiva okruženja `GRANDINE_LOG` i njene direktive se dodaju na kraj, tako da mogu da nadjačaju podrazumevane. Tako sastavljen filter se pozivom `reload::Layer::new` obavija u sloj za ponovno učitavanje, čime postaje promenljiv, i vezuje za `stdout_layer`. Poseban, nepromenljiv filter propušta samo događaje sa cilja `exception` i vezuje se za `file_layer`, koji ih upisuje u fajl. Na kraju se oba sloja objedinjuju u jedinstveni pretplatnik pozivom `init`.

Da upis izuzetaka u fajl ne bi opterećivao izvršavanje, `file_layer` koristi neblokirajući pisac `non_blocking` iz biblioteke `tracing_appender` [30]. Taj pisac formatirane zapise ne upisuje odmah na disk, već ih predaje zasebnoj radnoj niti, čime se ulazno-izlazni trošak skida sa kritičnih putanja kôda. Čuvar (engl. *guard*) tog pisca čuva se u `OnceLock` promenljivoj čime se obezbeđuje da njegov životni vek traje tokom rada klijenta, odnosno da radna nit ne bude prekinuta pre nego što se svi zapisi isprazne.

8.3 Kontrola nivoa praćenja tokom rada

Važno je da se kontrola nivoa praćenja izvršava unutar samog konsenzus klijenta, a ne da se filtriranje odvija na standardnom izlazu ili u spoljnim alatima za prikupljanje i prikaz dijagnostičkih podataka, u kojima bi se prikaz naknadno ograničavao po nivou. Ukoliko bi klijent emitovao događaje svih nivoa, dok bi se njihovo prikazivanje naknadno ograničavalo u eksternom alatu, samo bi se smanjila količina prikazanih podataka operateru, ali ne i trošak njihovog generisanja i obrade unutar samog klijenta. Takav pristup mogao bi značajno da poveća opterećenje konsenzusnog čvora, i time ugrozi pravovremeno izvršavanje njegovih konsenzusnih operacija. Ethereum konsenzus ne bi tolerisao takvo ponašanje klijenta, čiji rad zahteva poštovanje strogih vremenskih ograničenja i stabilno izvršavanje tokom rada mreže.

Zbog toga je filtriranje implementirano direktno u okviru `tracing` pretplatnika, tako da se `tracing` događaji koji ne odgovaraju trenutno podešenom nivou filtriranja i ne emituju.

Isti mehanizam obuhvata i spoljne zavisnosti klijenta koje i same koriste `tracing`. Takva

je `discv5` [43], biblioteka koju Grandine koristi za otkrivanje drugih čvorova u mreži (Ethereum Node Discovery protokol). Pošto `discv5` svoje događaje emituje kroz `tracing`, a pretplatnik je globalan, nivo i vidljivost tih događaja podešavaju se na isti način kao i za interne module Grandine, bez dodatnog mosta ili posebnog mehanizma. Time se jedinstvenim pretplatnikom obuhvataju i komponente izvan samog klijenta.

8.3.1 Podešavanje nivoa pri pokretanju čvora

Pri pokretanju, filter se sastavlja iz inicijalno implementiranog skupa podrazumevanih direktiva – po jedna za svaki značajan *krejt* sistema, sve na nivou `info` – čime se dobija razuman i predvidiv obim `tracing` događaja na koje smo pretplaćeni (listing 8.2).

Nivo praćenja željenih krejtova moguće je izabrati i pri pokretanju čvora, putem promenljive okruženja `GRANDINE_LOG`. Njena vrednost se čita i razlaže na pojedinačne direktive, koje se potom parsiraju i dodaju na kraj već sastavljenog filtera (listing 8.2). Upravo zato što se dodaju poslednje, one imaju prvenstvo u slučaju konflikta i za isti cilj nadjačavaju podrazumevanu direktivu, pa operater može da izabere početnu detaljnost za bilo koji modul još pre pokretanja. Na primer, vrednost `GRANDINE_LOG=info,pubkey_cache=debug` postavlja globalni nivo `info` na sve krejtove, a samo za krejt `pubkey_cache` uključuje detaljniji nivo `debug`.

8.3.2 Podešavanje nivoa tokom rada čvora

Da bi se omogućila izmena nivoa praćenja bez restartovanja čvora, filter glavnog toka nije statičan. On je obmotan slojem za ponovno učitavanje (`tracing_subscriber::reload`), koji obezbeđuje upravljač (engl. *handle*) pomoću kojeg se filter može izmeniti tokom rada. Taj upravljač je u kôdu predstavljen vrednošću tipa `TracingHandle`.

Upravljač nastaje pri inicijalizaciji pretplatnika i propagira se kroz izvršno okruženje (`runtime`) do HTTP API-ja klijenta Grandine. Taj HTTP API deo je samog klijenta (krejt `http_api`) i inače izlaže Beacon HTTP API (odeljak 4), a ovde je proširen dodatnom krajnjom tačkom za izmenu nivoa praćenja. Pošto se HTTP API može konstruisati i u okruženjima u kojima globalni pretplatnik nije inicijalizovan, poput pojedinih testnih okruženja, upravljač se čuva kao *opciono* stanje tipa `Option<TracingHandle>`. U redovnom radu čvora upravljač postoji i krajnja tačka je dostupna, dok se u suprotnom vraća status `503`.

Izmena filtera glavnog toka izložena je preko pomenutog HTTP API-ja, na krajnjoj tački (engl. *endpoint*)

`POST /eth/v2/debug/tracing/log_level`.

Telo zahteva sadrži cilj (`target`) i željeni nivo (`level`), na primer `fork_choice_control`

i `debug`, od kojih se sastavlja direktiva `fork_choice_control=debug`. Direktiva se zatim parsira i primenjuje na filter preko `TracingHandle`. U slučaju neispravne direktive vraća se `400 Bad Request`, dok se u slučaju nedostupnog mehanizma za praćenje vraća `503 Service Unavailable`.

Serverski deo klijenta koji ovu logiku sprovodi prikazan je u listingu 8.3. Raščlanjena direktiva se, kada upravljač postoji, pozivom `modify_log` dodaje u promenljivi filter, a kada ga nema, vraća se odgovor `503`.

```
let directive_str = format!("{}={}", req.target, req.level);
match directive_str.parse() {
    Ok(directive) => {
        if let Some(handle) = handle {
            handle.modify_log(|filter| {
                let old = core::mem::take(filter);
                *filter = old.add_directive(directive);
            })?;
            (StatusCode::OK, "log level updated".to_owned())
        } else {
            (StatusCode::SERVICE_UNAVAILABLE, "tracing not
                available".to_owned())
        }
    }
    Err(e) => (StatusCode::BAD_REQUEST, format!("invalid directive:
        {e}")),
}
```

Listing 8.3: Primena direktive na promenljivi filter tokom rada (krejt `http_api`).

Na ovaj način operater može, na primer, privremeno da poveća detaljnost i nivo praćenja samo za izabrane module dok analizira trenutni problem, a zatim da ih vrati na prvobitno stanje bez prekida rada čvora i njegovog izlaska iz mreže.

Konkretan primer upotrebe prikazan je u listingu 8.4. Slanjem jednostavnog `POST` zahteva na opisanu krajnju tačku, nivo praćenja modula `fork_choice_control` menja se na `debug` tokom rada čvora.

```
curl -X POST http://localhost:5052/eth/v2/debug/tracing/log_level \
  -H "Content-Type: application/json" \
  -d '{
    "target": "fork_choice_control",
    "level": "debug"
  }'
```

Listing 8.4: Primer zahteva alatom `curl` kojim se tokom rada menja nivo praćenja modula `fork_choice_control`.

Praktičan prikaz ove izmene tokom rada čvora, u kome se nivo modula `fork_choice_control` menja iz `info` u `debug` i potom vraća na `info`, dostupan je u pratećem video snimku.[1]

Ovakva mogućnost je posebno značajna za probleme koji se javljaju nepredvidivo ili samo pri određenim kombinacijama uslova u mreži. Umesto da se čvor prethodno konfiguriše za detaljnije praćenje, potrebni nivo detaljnosti može se uključiti tek kada se problem pojavi. Time se omogućava detaljnija analiza ponašanja klijenta u realnom vremenu, uz izbegavanje nepotrebnog opterećenja koje bi stalno uključeno detaljno praćenje moglo da izazove.

8.4 Trajno beleženje kritičnih izuzetaka

Za razliku od uobičajenih logova koji prate redovan rad klijenta, kategorija *kritičnih izuzetaka* obuhvata događaje koji ukazuju na odstupanje od očekivanog ponašanja konsenzusnog klijenta. Oni mogu biti posledica spoljašnjih uticaja, poput neispravnog ili zlonamernog povezanog čvora, ali i neočekivanog ponašanja unutar samog klijenta. Namena ovog kanala je dvostruka. Takvi događaji ne treba da opterećuju običnog korisnika ni operatera klijenta, ali istovremeno nikada ne smeju biti izgubljeni, kako bi, ako nešto pođe po zlu, mogli biti priloženi razvojnom timu Grandine na analizu.

Zbog toga je odgovarajući deo pretplatnika izdvojen u zaseban sloj koji nezavisno od trenutnog nivoa praćenja beleži kritične izuzetke i trajno ih upisuje u fajl `exception.log`. Na standardnom izlazu oni su za korisnika podrazumevano skriveni, dok ih je moguće dodatno prikazati putem konfiguracije pri pokretanju ili preko HTTP krajnje tačke tokom rada. Ukoliko ciljni direktorijum ne postoji, on se automatski kreira, dok se nad fajlom `exception.log` primenjuje rotacija zasnovana na njegovoj veličini.[2]

Taj ciljni direktorijum jeste data direktorijum čvora, a ime fajla i njegova lokacija nisu izloženi konfiguraciji, već su fiksni i vezani za verziju klijenta. Fajl se tako za datu verziju uvek nalazi na istom, unapred poznatom mestu, pod imenom `exception.log`. Time se u slučaju problema uvek pouzdano može pronaći i proslediti razvojnom timu Grandine, nezavisno od okruženja u kom čvor radi.

Kritični izuzeci se po svojoj prirodi javljaju retko, jer označavaju stvarna odstupanja u radu klijenta, a ne redovne događaje. Rotacija fajla `exception.log` nije uvedena zbog očekivanog obima izuzetaka tokom normalnog rada čvora i mreže, već kao mera predostrožnosti za slučaj da nešto pođe po zlu. Budući da konsenzusni klijent radi kontinuirano i neograničeno dugo, neograničen rast ovog fajla predstavljao bi latentni rizik u atipičnim

[1] https://www.youtube.com/watch?v=c8RrdKiOQno

[2] Rotacija je uobičajen postupak upravljanja log fajlovima, kod kog se aktivni fajl, kada dostigne zadatu granicu, zatvara i arhivira, dok se beleženje nastavlja u novom, praznom fajlu, uz čuvanje samo ograničenog broja arhiviranih fajlova.

situacijama, poput dužih problema u mrežnom okruženju, učestalih zlonamernih napada na čvor ili ponovljene interne greške. Rotacija zasnovana na veličini fajla garantuje gornje ograničenje zauzeća prostora na disku i u takvim slučajevima.

Kada `exception.log` dostigne veličinu od 1 GB, aktivni fajl se rotira, a upisivanje se nastavlja u novom fajlu. Prethodni fajl se zatim kompresuje korišćenjem `gzip` formata. Čuva se najviše pet fajlova, pri čemu je najnoviji aktivni fajl nekompresovan, dok su prethodna četiri kompresovana. Kada bi rotacijom nastao novi fajl iznad ovog ograničenja, najstariji se automatski uklanja. Na ovaj način zadržava se do približno 5 GB istorije izuzetaka pre kompresije, dok je stvarna zauzetost prostora zahvaljujući kompresiji do 2 GB. Time je količina sačuvane istorije ograničena, čime je sprečen nekontrolisan rast zauzeća prostora na disku.

Navedene veličine namerno su fiksne i nisu izložene korisničkoj konfiguraciji, u skladu sa zahtevom razvojnog tima Grandine. Potpuni čvor (engl. *full node*) već zauzima prostor reda veličine terabajta, pa je izabrana granica takva da je svaki pokrenuti čvor može podržati bez dodatnog uticaja na korisnika. Pošto se kritični izuzeci javljaju retko, u najvećem broju slučajeva se ni prvi fajl ne popuni. Fiksne vrednosti istovremeno obezbeđuju da ponašanje ovog kanala bude predvidivo i jednako na svim čvorovima iste verzije klijenta.

Za implementaciju rotacije i kompresije korišćena je biblioteka `logroller`, čime je izbegnuto ručno pisanje te funkcionalnosti. U listingu 8.5 se tako podešen rotirajući pisac dodatno obavija neblokirajućim piscem iz biblioteke `tracing_appender`, a njegov čuvar se čuva u `LOG_GUARD` kako radna nit ne bi bila prekinuta pre nego što se svi zapisi isprazne.

```
let rotator = LogRollerBuilder::new(data_dir,
    Path::new("exception.log"))
     .rotation(Rotation::SizeBased(RotationSize::GB(1)))
     .max_keep_files(5)
     .compression(Compression::Gzip)
     .build()?;

let (non_blocking, guard) = tracing_appender::non_blocking(rotator);
LOG_GUARD.set(guard).ok();
```

Listing 8.5: Rotirajući pisac za `exception.log` (krejt `binary_utils`).

8.5 Prostiranje praćenja na mrežni sloj

Mrežni (*peer-to-peer*) sloj klijenta Grandine nije deo glavnog repozitorijuma, već zasebna biblioteka `eth2_libp2p` [38], nastala kao *fork* (izdvojena, samostalno održavana kopija) biblioteke koju je razvio tim klijenta Lighthouse [26], a koju tim Grandine održava u

sopstvenom repozitorijumu. U polaznom stanju ova biblioteka koristila je sistem logovanja `slog` [45], gde je od glavne aplikacije dobijala instancu logera preko koje je prosleđivala svoje bogate i strukturirane dijagnostičke podatke, uključujući identifikatore povezanih čvorova, adrese i portove. Glavna aplikacija je te zapise primala preko mosta (`slog-stdlog`) i prosleđivala ih svom `log/env_logger` ispisu. Kako taj ispis radi sa porukom, a ne sa poljima, strukturirana polja su se pri prelasku serijalizovala u tekst poruke i time gubila strukturu.

Migracija `tracing` zato je obuhvatila i mrežni sloj i sprovedena je koordinisano u oba repozitorijuma. Sistem `slog` zamenjen je `tracing` sistemom, tako da događaji mrežnog sloja sada koriste isti model praćenja kao i ostatak klijenta i prosleđuju se zajedničkom pretplatniku. Strukturirana polja se na taj način zadržavaju kao deo `tracing` događaja, bez potrebe za njihovim prevođenjem u tekstualne poruke, dok mrežni sloj može da koristi isti mehanizam za upravljanje kontekstom i nivoom praćenja kao i ostatak sistema.

8.5.1 Prilagođavanje mrežnog sloja modelu `tracing`

Prelazak mrežnog sloja na `tracing` zahtevao je promene u načinu na koji se informacije o izvršavanju prosleđuju kroz njegove komponente. Pošto više nije bilo potrebno koristiti zaseban `slog` sistem, uklonjene su instance tipa `slog::Logger` iz polja i konstruktora komponenti koje su ih koristile. Pozivi za beleženje događaja zatim su zamenjeni `tracing` makroima iz zajedničkog `logging` krejta (listing 8.1), čime mrežni sloj koristi isti mehanizam praćenja kao i ostatak klijenta.

Promena je obuhvatila i način predstavljanja strukturiranih podataka. U prethodnom sistemu je, da bi se neka vrednost zabeležila kao strukturirano polje, njen tip morao da zna da se predstavi `slog` sistemu. To se postizalo implementacijom osobine (engl. *trait*) `slog::Value`, koju `slog` koristi da bi vrednost pretvorio u polje zapisa. Zbog toga su i sopstveni tipovi mrežnog sloja, poput identifikatora podtoka `SubstreamId`, morali da obezbede tu implementaciju.

U modelu `tracing` to nije potrebno, jer se vrednosti mogu direktno proslediti kao strukturirana polja događaja. Način na koji će vrednost biti pretvorena u tekst bira se posebnim znakom ispred nje. Znak `%` bira prikaz `Display`, uredan oblik namenjen čoveku, dok znak `?` bira prikaz `Debug`, razvojni oblik koji vernije odslikava unutrašnju strukturu vrednosti. `Display` i `Debug` su uobičajene osobine jezika Rust za pretvaranje vrednosti u tekst, pa se izborom znaka bira koja će se od njih upotrebiti. Na taj način struktura događaja ostaje očuvana tokom njegovog daljeg obrađivanja, umesto da se podaci unapred ugrađuju u tekstualnu poruku. Prelazak je na konkretnom primeru prikazan u listingu 8.6.

```
1 // PRE (slog)
2
```

```
// sopstveni tip je morao da implementira slog::Value da bi bio polje
impl slog::Value for SubstreamId {
    fn serialize(
        &self,
        record: &slog::Record,
        key: slog::Key,
        serializer: &mut dyn slog::Serializer,
    ) -> slog::Result {
        slog::Value::serialize(&self.0, record, key, serializer)
    }
}

// prosleđuje se instanca logera; polja preko "key" => value
crit!(
    self.log,
    "Duplicate outbound substream id";
    "peer_id" => %self.peer_id,
    "connection_id" => %self.connection_id,
    "id" => self.current_outbound_substream_id,
);
-----------------------------------------------
// POSLE (tracing): bez logera; strukturirano key = value;
   SubstreamId se
// prosleđuje direktno uz ? (Debug), bez slog::Value implementacije
crit!(
    peer_id = %self.peer_id,
    connection_id = %self.connection_id,
    id = ?self.current_outbound_substream_id, "Duplicate outbound
       substream id");
```

Listing 8.6: Prelazak sa `slog` na `tracing` u mrežnom sloju (`eth2_libp2p`).

Listing 8.6 isti događaj prikazuje pre i posle prelaska. U gornjem delu, sa `slog`-om, tip `SubstreamId` najpre implementira `slog::Value`, a makrou `crit!` prosleđuje se instanca logera `self.log`, dok se polja navode u obliku `"ključ-> vrednost`. U donjem delu, sa `tracing`-om, logera više nema, polja se pišu u obliku `ključ = vrednost`, a `SubstreamId` se prosleđuje neposredno uz znak `?`, dakle preko `Debug` prikaza, bez ikakve dodatne implementacije.

8.6 Instrumentacija ključnih komponenti

Pojedinačni `tracing` događaji govore *šta* se desilo, ali ne i *u okviru čega*. U sistemu u kome se istovremeno izvršava mnoštvo asinhronih zadataka, događaji iz različitih operacija se prepliću, pa je iz samog njihovog niza teško rekonstruisati koji je događaj kojoj operaciji

pripadao i koliko je ona trajala. Zato `tracing`, pored događaja, uvodi i *raspone* (*spans*). Raspon je imenovani, strukturirani kontekst izvršavanja koji ima početak i kraj, a može biti aktivan tokom jednog ili više vremenskih intervala, budući da se u njega može ući i iz njega izaći više puta tokom izvršavanja. Svaki događaj koji nastane dok je raspon aktivan u okviru istog zadatka pripada tom rasponu, a rasponi se mogu ugnježdavati i tako oslikati lanac poziva. Time se razdvojeni događaji povezuju u smislenu celinu, uz podatak o trajanju operacije. To je posebno korisno u klijentu Grandine, gde se veliki broj zadataka izvršava konkurentno.

Konkretan primer koristi raspona u dijagnostici potekao je iz prakse drugog konsenzusnog klijenata - klijenta Lighthouse. Tim Lighthouse je pomoću raspona uočio da je mrežni sloj znatan deo vremena trošio na slanje konsenzusnih zahteva novopokrenutom čvoru, iako taj čvor još nije bio sinhronizovan niti spreman da obavlja svoje dužnosti. Takvo ponašanje teško je uočljivo iz pojedinačnih događaja, jer svaki zahtev sam po sebi izgleda ispravno. Tek kada su zahtevi povezani rasponima koji nose kontekst o stanju čvora i toku operacije, postalo je vidljivo da se veliki deo rada troši u periodu u kom čvor još ne može da odgovori na njih. Na osnovu tog uvida uvedene su odgovarajuće provere, čime je smanjeno nepotrebno opterećenje mreže. Ovaj primer pokazuje kako rasponi, povezujući razdvojene događaje u širi kontekst, otkrivaju probleme u ponašanju sistema koje bi inače bilo teško primetiti.

Rasponi su u kôd uvedeni na dva načina. Prvi i najčešći je atribut `#[instrument]` iznad funkcije, koji za svaki njen poziv automatski otvara raspon i u njega upisuje izabrane argumente kao strukturirana polja. Drugi je eksplicitno vezivanje raspona za *future* pomoću `.instrument(span)`, što je potrebno kada raspon treba da prati asinhroni tok kroz više `await` tačaka ili kada se više operacija grupiše pod jednim rasponom. Atribut `#[instrument]` upotrebljen je na velikom broju mesta, reda veličine oko sto osamdeset.

Instrumentovane su kritične komponente sistema, redom mrežni sloj i krajnje tačke unutar HTTP API klijenta Grandine, potom njegovi krejtovi `validator` (dužnosti validatora) i `fork_choice_control` (kontrola izbora grane lanca). Kao i kod makroa iz odeljka 8.1, i ovde se vodilo računa o trošku. Rasponi su podrazumevano postavljeni na niske nivoe (`level = 'debug'`), pa su neaktivni i uz zanemarljiv trošak sve dok se ciljano ne uključe. Tek kada operater podigne nivo, instrumentovani delovi počinju da proizvode raspone koji događajima dodaju kontekst – ko je pozvao koga i koliko je operacija trajala.

Uticaj na metrike složenosti. Anotacije rasponima i prelazni makroi povećavaju formalnu (kognitivnu) složenost funkcija u očima statičkog analizatora `clippy`, iako stvarnu logiku ne komplikuju. Zato je prag `cognitive-complexity-threshold` morao da bude podignut, kako ispravno instrumentovan kôd ne bi generisao lažna upozorenja.

8.6.1 Rasponi i asinhrona bezbednost (`Send`)

Izbor načina instrumentacije nije stvar ukusa, nego ispravnosti. Način na koji se raspon aktivira mora biti usklađen sa asinhronim modelom izvršavanja, jer neusklađen izbor može da naruši dva svojstva. Prvo je bezbednost premeštanja zadatka između niti, a drugo tačnost konteksta koji se pridružuje događajima.

Da bi se ta razlika razumela, potrebno je najpre pojasniti nekoliko pojmova. Asinhrona funkcija, označena ključnom rečju `async`, ne izvršava svoje telo odmah po pozivu. Umesto toga vraća *future*, objekat koji predstavlja operaciju čije je izvršavanje odloženo i koju izvršno okruženje pokreće kasnije. Kao što je opisano u odeljku 4.2, takav *future* se ne izvršava u jednom potezu, već napreduje do naredne `.await` tačke i tu može privremeno da stane, ustupajući nit drugim zadacima, da bi kasnije nastavio odatle. Da bi nastavak bio moguć, izvršno okruženje pri svakom zastoju čuva unutrašnje stanje *future*-a, odnosno one vrednosti koje mu i posle zastoja još trebaju.

Pošto Grandine koristi višenitno `tokio` okruženje, nastavak izvršavanja posle `.await` tačke ne mora da se odigra na istoj niti na kojoj je *future* zaustavljen, jer okruženje premešta zadatke između svojih radnih niti. Zato *future* koji se izvršava kao zaseban zadatak mora biti bezbedan za takvo premeštanje. Rust to svojstvo izražava osobinom (engl. *trait*) `Send`, koja označava da se vrednost datog tipa sme preneti u drugu nit, dok se tipovi bez tog svojstva označavaju kao `!Send`.

Upravo takva `!Send` vrednost može nastati pri radu sa rasponima. Pored atributa `#[instrument]` i metode `.instrument(span)`, opisanih u odeljku 8.6, `tracing` omogućava i ručno ulaženje u raspon pozivom `span.entered()`. Taj poziv vraća čuvara (engl. *guard*) tipa `EnteredSpan` [48], čiji životni vek određuje period tokom kog je raspon aktivan. Kada se čuvar uništi, raspon se napušta. Sam čuvar `EnteredSpan` ne implementira `Send`, jer ulazak u raspon i izlazak iz njega moraju da se dogode na istoj niti.

Prvi problem neposredno sledi iz te činjenice. Ako se čuvar zadrži preko `.await` granice, on postaje deo unutrašnjeg stanja *future*-a, pa i sam *future* postaje `!Send` i više ne može da se izvršava kao zaseban zadatak u višenitnom `tokio` okruženju. Problem stoga nije u samom čuvaru, već u tome što njegov životni vek prelazi granicu suspenzije asinhronog kôda.

Drugi problem ne zavisi od bezbednosti prenosa između niti. Dok je jedan zadatak suspendovan, ista nit može izvršavati drugi zadatak. Kada je raspon aktiviran neposrednim pozivom `entered()`, njegova aktivnost vezana je za trenutnu nit, a ne za suspendovani zadatak, pa se događaji drugog zadatka koji se u međuvremenu izvršava na istoj niti mogu pogrešno pripisati rasponu prvog. Da bi kontekst bio tačan, raspon mora da prati sam *future*, tako da bude aktivan dok se *future* izvršava, a neaktivan dok je suspendovan.

Imajući u vidu oba ograničenja, u radu je usvojena jedinstvena konvencija instrumentacije asinhronog kôda. Rasponi se vezuju za *future* pomoću atributa `#[instrument]` ili metode `future.instrument(span)`, dok se ručno ulaženje u raspon pomoću `entered()` u asinhronom kôdu izbegava. Time se čuvar nikada ne zadržava preko `.await` granice, čime se istovremeno očuva bezbednost prenosa *future*-a između niti i obezbeđuje da kontekst raspona verno prati asinhroni tok izvršavanja. Konvencija je usvojena upravo zbog ovog problema, uočenog tokom migracije, koji je zabeležen i u pratećoj prijavi problema (engl. *issue*) [49].

Način primene ove konvencije prikazan je u listingu 8.7. U sva tri primera funkcija `handle` prima argument `peer` tipa `PeerId`, čija se vrednost beleži kao polje odgovarajućeg raspona, a primeri se razlikuju po tome kako se raspon vezuje za asinhrono izvršavanje.

```
// NEISPRAVNO: cuvar EnteredSpan (iz entered()) je !Send i ostaje
    aktivan
// preko .await granice, zbog toga future postaje !Send
async fn handle(peer: PeerId) {
    let span = info_span!("handle", %peer);
    let _guard = span.entered();
    do_async_work().await;
}


// ISPRAVNO: raspon je vezan za asinhronu funkciju
#[instrument(skip_all, fields(%peer))]
async fn handle(peer: PeerId) {
    do_async_work().await;
}

// ISPRAVNO: raspon je eksplicitno vezan za future
async fn handle(peer: PeerId) {
    let span = info_span!("handle", %peer);
    do_async_work().instrument(span).await;
}
```

Listing 8.7: Načini instrumentacije asinhronog kôda u klijentu Grandine.

Prvi primer je neispravan, i na njemu se najbolje vidi u čemu je opasnost. Poziv `span.entered()` aktivira raspon i vraća čuvara `EnteredSpan`, koji prati princip RAII što ga Rust primenjuje automatski. Po tom principu čuvar se uništava tek kada se izađe iz bloka u kome je nastao, u ovom slučaju na kraju tela funkcije `handle`, i raspon se napušta upravo u tom trenutku. Problem je što se između nastanka i uništenja čuvara nalazi `do_async_work().await`. Ta `.await` tačka je granica suspenzije, mesto na kome *future* može privremeno da stane i kasnije nastavi izvršavanje. Da bi nastavak bio moguć, sve što je u tom trenutku živo mora

da se sačuva, pa i čuvar `EnteredSpan` postaje deo sačuvanog stanja *future*-a. Pošto je taj čuvar `!Send`, ceo *future* postaje `!Send` i izvršno okruženje ga ne može premeštati između svojih radnih niti, zbog čega nije upotrebljiv u višenitnom `tokio` okruženju.

Druga dva primera rešavaju upravo taj problem, tako što raspon vezuju za sam *future* umesto da ga drže otvorenim preko čuvara. Da bi se razumelo kako, važno je kako izvršno okruženje pokreće asinhroni kôd. Ono *future* ne izvršava odjednom, već ga više puta poziva da napreduje do naredne `.await` tačke. Metoda `.instrument(span)` i atribut `#[instrument]`, koji se na nju interno oslanja, umeću raspon oko svakog takvog koraka. Na početku koraka raspon se aktivira, ostaje aktivan dok se *future* izvršava, a napušta se čim *future* vrati kontrolu na `.await` tački. Pri narednom nastavku izvršavanja raspon se ponovo aktivira, po potrebi i na drugoj niti. Aktivni raspon tako nikada ne prelazi granicu suspenzije, jer traje samo koliko i pojedinačni korak izvršavanja.

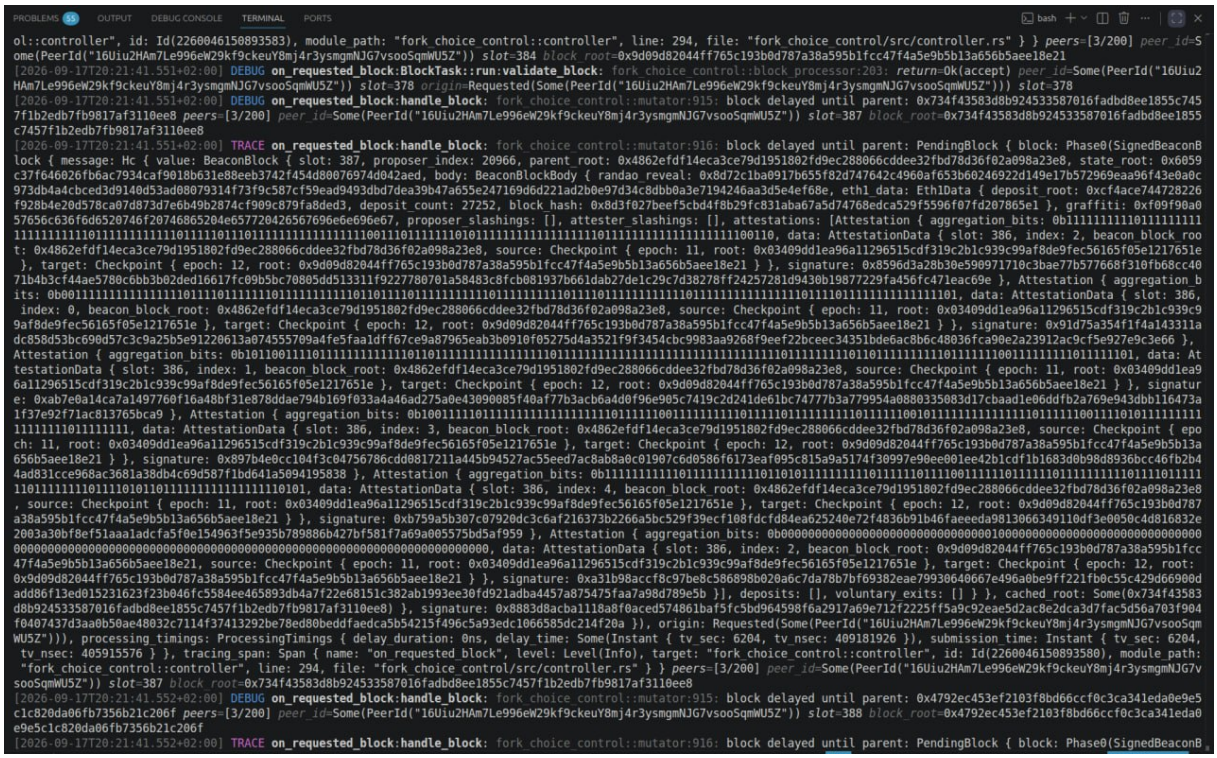
Odatle slede oba bezbednosna svojstva. Između koraka *future* čuva samo objekat raspona, koji je bezbedan za prenos između niti, a ne i njegovog čuvara. Zato nijedna `!Send` vrednost ne živi preko `.await` granice, pa *future* ostaje `Send` i izvršno okruženje ga slobodno premešta između radnih niti. Istovremeno je raspon aktivan tačno dok se taj *future* izvršava, pa se događaji uvek pripisuju operaciji kojoj stvarno pripadaju, bez obzira na to na kojoj se niti izvršavanje odvija.

Razlika između dva ispravna oblika svodi se na to ko kreira raspon. U drugom primeru atribut `#[instrument]` stvara raspon automatski iz potpisa funkcije, pri čemu `fields(%peer)` beleži identifikator čvora kao njegovo polje, a `skip_all` isključuje automatsko beleženje ostalih argumenata. U trećem primeru raspon se najpre ručno napravi makroom `info_span!` i dopuni željenim poljima, a zatim pridruži *future*-u metodom `.instrument(span)`. Taj oblik je pogodan kada rasponu treba dodati posebna polja ili mu prilagoditi kontekst, uz istu bezbednost asinhronog izvršavanja kao u drugom primeru.

8.7 Primer zapisa tokom rada čvora

Da bi se ilustrovali format i obim zapisa koje sistem praćenja proizvodi, na slici 8.2 prikazan je isečak izlaza nastao tokom sinhronizacije čvora. Čvor tada istovremeno dovlači i obrađuje veliki broj blokova zatraženih od povezanih čvorova, pa se zapisi iz više komponenti konsenzusnog klijenta međusobno prepliću.

Svaki zapis počinje vremenom nastanka i nivoom (`DEBUG` ili `TRACE`). Nakon toga sledi putanja koja pokazuje u okviru koje operacije je događaj nastao, propraćeno sa ciljem i brojem linije koji identifikuju komponentu i mesto koje je događaj emitovalo. Nastavak zapisa sačinjava prosleđena poruka događaja sa svojim argumentima, koji mogu ali ne moraju imati sopstvenu unutrašnju strukturu. Kada je imaju, ona se u ispisu prikazuje



Slika 8.2: Isečak stvarnog izlaza tokom sinhronizacije čvora, u boji kako se prikazuje u terminalu (krejt `fork_choice_control`). Prikaz ilustruje format i obim zapisa na različitim nivoima.

izabranim prikazom vrednosti. Na kraju dolaze strukturirana polja iz dva izvora. Prva su polja koja namenski makroi dodaju svakom događaju, poput ranije opisanog polja `peers` (odeljak 8.1). Druga su polja koja potiču iz raspona aktivnih u trenutku beleženja.

Radi čitljivosti, jedan `DEBUG` red sa slike 8.2 izdvojen je u listingu 8.8.

```
[2026-09-17T20:21:41.551+02:00] DEBUG on_requested_block:handle_block:
    fork_choice_control::mutator:915: block delayed until parent:
    0x734f43583d8b924533587016fadbd8ee1855c7457f1b2edb7fb9817af3110ee8 peers=[3/200]
    peer_id=Some(PeerId("16Uiu2HAm7Le996eW29kf9ckeuY8mj4r3ysmgmNJG7vsooSqmWU5Z"))
    slot=387
    block_root=0x734f43583d8b924533587016fadbd8ee1855c7457f1b2edb7fb9817af3110ee8
```

Listing 8.8: Jedan `DEBUG` red iz izlaza sa slike 8.2, izdvojen radi čitljivosti.

U tom redu, posle nivoa `DEBUG`, putanja `on_requested_block:handle_block` pokazuje operaciju u okviru koje operacije je događaj nastao, a `fork_choice_control::mutator:915` identifikuju komponentu i mesto koje je događaj emitovalo. Poruka glasi `block delayed until parent` i uz nju, kao njen argument, stoji koren bloka. Na kraju su strukturirana polja `peers`, `peer_id`, `slot` i `block_root`. Prvo, `peers`, dodaju namenski makroi, a preostala tri dolaze iz raspona aktivnih u tom trenutku.

Putanja `on_requested_block:handle_block` je niz raspona koji su u trenutku nastanka događaja aktivni, a rasponi se otvaraju instrumentacijom opisanom u odeljku 8.6. Raspon

`on_requested_block` nastaje iz atributa `#[instrument]` nad istoimenom funkcijom i u svojim poljima beleži `peer_id` i `slot`, dok se `block_root` beleži u polju raspona `handle_block`. Raspon `handle_block` takođe nastaje iz atributa `#[instrument]` nad istoimenom funkcijom, ali se otvara kasnije i na drugoj niti, dok se preko sačuvanog konteksta vezuje za nadređeni raspon `on_requested_block`. Pretplatnik zato uz svaki događaj ispisuje imena aktivnih raspona, od najšireg ka najužem, a njihova polja pridružuje samom događaju. Time svaki događaj automatski dobija kontekst operacije u kojoj je nastao, bez obzira na to iz koje komponente potiče, što je i osnovna praktična korist raspona.

U listingu 8.9 prikazana su dva mesta u kôdu iz kojih potiču dva zapisa sa slike 8.2, jedan na nivou `DEBUG`, a drugi na nivou `TRACE`. To su dve uzastopne linije koda, pa se izvršavaju praktično u istom trenutku i isti događaj beleže na dva nivoa.

```
// mutator.rs, linija 915
debug_with_peers!("block delayed until parent: {block_root:?}");
// mutator.rs, linija 916
trace_with_peers!("block delayed until parent: {pending_block:?}");
```

Listing 8.9: Dva susedna poziva makroa iz kojih potiču `DEBUG` i `TRACE` zapis (krejt `fork_choice_control`).

Oba poziva vrednost formatiraju znakom `?`, koji bira `Debug` prikaz opisan u odeljku 8.5. Razlika je u tome šta se formatira. Na nivou `DEBUG` to je mala vrednost, koren bloka `block_root`, pa je ispis kratak. Na nivou `TRACE` to je cela struktura `pending_block`, koju znak `?` razvija u ceo blok sa svim atestacijama, potpisima i ostalim poljima, što se na slici 8.2 vidi kao ogroman ispis. Tek posle te poruke slede polja iz aktivnih raspona, `peer_id`, `slot` i `block_root`. Ovaj par poziva zato oslikava namenu nivoa praćenja. Detaljnost ispisa prilagođava se nivou za koji je događaj namenjen, pa isti trenutak na nivou `DEBUG` daje kraći i pregledniji opis, a na nivou `TRACE` bogat i iscrpan opis za dublju analizu. Najveća količina podataka tako ostaje ograničena na najniži nivo praćenja, koji se uključuje samo kada je zaista potreban.

Ovakvi zapisi nastaju u velikom obimu i velikom brzinom, a prikazani primer predstavlja tek delić onoga što se generiše tokom rada čvora, pri čemu obim zapisa može biti i znatno veći. Zbog toga oni po svojoj prirodi nisu namenjeni neposrednom ljudskom čitanju, već obradi u alatima za opservabilnost. Takvi alati automatizovano pretražuju zapise, prepoznaju obrasce i povezuju pojedinačne događaje u širi kontekst operacije. Korisnik sa njima najčešće radi posredno, kroz upitni i vizuelni interfejs, a u novije vreme i kroz mogućnosti zasnovane na veštačkoj inteligenciji.

Glava 9

Testiranje

Pošto izmene u konsenzusnom klijentu moraju da očuvaju postojeću ispravnost, testiranje je sprovedeno na dva nivoa. Prvi nivo odnosio se na očuvanje ispravnosti protokola. Nakon svake faze integracije (poglavlje 6) celokupan skup *consensus-spec* testova [14] morao je i dalje da prolazi, čime je provereno da izmene povezane sa uvođenjem mehanizma praćenja nisu uticale na ponašanje klijenta u odnosu na specifikaciju protokola. Drugi nivo činili su namenski automatizovani testovi kojima je provereno ponašanje implementiranog sistema za praćenje.

Testovima različitih nivoa praćenja provereno je ponašanje filtera i generisanih događaja. Na podrazumevanom nivou provereno je da su vidljivi događaji nivoa `info`, `warn` i `error`, dok kritični izuzeci nisu deo glavnog toka. Nakon izmene filtera proverena je njihova vidljivost, kao i pojavljivanje događaja nivoa `debug` i `trace` kada je uključeno detaljnije praćenje. Dodatno su provereni formatiranje strukturiranih polja i različiti oblici prosleđivanja argumenata kroz prelazne makroe.

Da bi se izlaz mogao automatski proveravati, standardni izlaz se tokom testa preusmerava u međumemorijski bafer (`BufferRedirect`). Testovi se oslanjaju na jednom postavljen pretplatnik, čija je inicijalizacija prikazana u listingu 9.1.

```
struct LoggerWithTempDir {
    handle: TracingHandle,
    _temp_dir: TempDir,
}

static LOGGER: LazyLock<Mutex<Option<LoggerWithTempDir>>> =
    LazyLock::new(|| Mutex::new(None));
```

```rust
fn init_logger_once() -> TracingHandle {
    let data_dir = TempDir::new().expect("should create a temp data dir");
    let mut lock = LOGGER.lock().expect("Failed to acquire LOGGER mutex lock");
    if lock.is_none() {
        let handle =
            initialize_tracing_logger(module_path!(), Some(data_dir.path()), None, false)
                .expect("Failed to initialize tracing logger");

        *lock = Some(LoggerWithTempDir {
            handle: handle.clone(),
            _temp_dir: data_dir,
        });

        handle
    } else {
        lock.as_ref()
            .expect("LOGGER should always be initialized")
            .handle
            .clone()
    }
}
```

Listing 9.1: Jednokratna inicijalizacija globalnog pretplatnika, deljena između testova (krejt `binary_utils`).

Pomoćna funkcija `init_logger_once` obezbeđuje da se globalni pretplatnik postavi tačno jednom. Postavljanje se u `tracing`-u obavlja pozivom `init`, koji registruje pretplatnik za ceo proces i sme da se izvrši samo jednom. Zato se rezultat čuva u statičkoj promenljivoj `LOGGER`. Ona je tipa `LazyLock` iz standardne biblioteke jezika Rust, koji sadržaj statičke promenljive stvara tek pri prvom pristupu i na način bezbedan za istovremeni pristup iz više niti. Sam sadržaj je vrednost tipa `Option`, u početku prazna i smeštena u muteks (`Mutex`). Pošto je statička promenljiva dostupna iz svih niti, muteks obezbeđuje bezbedan i sinhronizovan pristup tom promenljivom globalnom stanju, kako to jezik Rust zahteva. Pri prvom pozivu funkcija pravi pretplatnik pozivom `initialize_tracing_logger`, a dobijeni upravljač `TracingHandle` smešta u `LOGGER` i vraća. Pri svakom narednom pozivu vraća se klon već sačuvanog upravljača koji upućuje na isti, deljeni pretplatnik.

Privremeni direktorijum (`TempDir`) pravi se pri istoj inicijalizaciji i čuva se zajedno sa upravljačem u promenljivoj `LOGGER`. U njega se upisuje fajl kritičnih izuzetaka, pa on postoji tokom celog izvršavanja testova.

Ova inicijalizacija ponavlja se za svaki krejt zato što Rust testove svakog krejta kompajlira u zaseban test-program koji se izvršava kao poseban proces. U svakom takvom procesu globalni pretplatnik postavlja se iznova, pa se ista inicijalizacija posebno nalazi i u krejtu `logging` i u krejtu `binary_utils`. Unutar jednog procesa svi testovi dele isti pretplatnik, zbog čega se izvršavaju serijalizovano (`serial_test::serial`), kako izmena filtera u jednom testu ne bi uticala na ostale.

Listing 9.2 prikazuje jedan takav test. Standardni izlaz se najpre preusmerava u bafer, a zatim se pozivom `init_logger_once` dobija deljeni upravljač. Nivo praćenja menja se pozivom `modify_log`, kome se prosleđuje funkcija koja postojeći filter klonira, dodaje mu direktivu `exception` i time ga zamenjuje. To je isti mehanizam kojim se filter menja i u produkciji, iza HTTP krajnje tačke (odeljak 8.3), pa test zapravo proverava istu putanju izmene filtera koja se u radu čvora pokreće `POST` zahtevom.

Nakon toga test emituje događaje nivoa `info`, `warn` i `error`, kao i kritični izuzetak, pa iz bafera čita ceo izlaz i proverava da su sva četiri zapisa prisutna. Kritični izuzetak je pri tome vidljiv upravo zato što je prethodno dodata direktiva `exception`, dok na podrazumevanom nivou ne bi bio deo glavnog toka, što proverava zaseban test. Na samom kraju test vraća stanje filtera dodavanjem direktive `exception=off`. Taj korak je neophodan jer je pretplatnik deljen između svih testova u procesu, pa bi zaostala direktiva promenila ponašanje ostalih testova.

```
#[test]
#[serial]
fn initialize_tracing_logger_developer_info_mode() -> Result<()> {
    let mut buf = BufferRedirect::stdout().expect("failed to redirect
       stdout");

    let handle = init_logger_once();
    handle.modify_log(|env_filter| {
        let new_filter = env_filter
            .clone()
            .add_directive("exception".parse().expect("Failed to
               parse"));
        *env_filter = new_filter;
    })?;

    info_with_peers!("info_with_peers test message");
    warn_with_peers!("warn_with_peers test message");
    error_with_peers!("error_with_peers test message");
    exception!("exception test message");

    let mut output = String::new();
    buf.read_to_string(&mut output)?;
```

```
21
22      assert!(output.contains("info_with_peers test message"));
23      assert!(output.contains("warn_with_peers test message"));
24      assert!(output.contains("error_with_peers test message"));
25      assert!(output.contains("exception test message"));
26
27      // Isključivanje "exception" direktive je neophodno jer globalni
            tracing pretplatnik traje između testova. Bez ovog čišćenja
            zaostala direktiva bi uticala na druge testove i narušila
            njihovo očekivano ponašanje.
28      handle.modify_log(|env_filter| {
29          let new_filter = env_filter
30              .clone()
31              .add_directive("exception=off".parse().expect("Failed to
                    parse"));
32          *env_filter = new_filter;
33      })?;
34
35      Ok(())
36  }
```

Listing 9.2: Test koji cilj `exception` uključuje, a potom isključuje preko upravljača `TracingHandle`. Poruke o grešci u `assert!` su radi preglednosti izostavljene (krejt `binary_utils`).

Automatizovani testovi predstavljaju osnovnu proveru ispravnosti implementiranog sistema, ali ne mogu obuhvatiti sva svojstva koja se ispoljavaju tokom rada čvora. Zbog toga su dopunjeni ručnim posmatranjem ponašanja klijenta pri pokretanju, sinhronizaciji i redovnom radu.

Glava 10

Zaključak

U ovom radu prikazana je i analizirana integracija strukturiranog sistema praćenja izvršavanja u produkcijski Ethereum konsenzusni klijent Grandine. Postojeće, hibridno logovanje – zasnovano na biblioteci `env_logger` u glavnoj aplikaciji i na strukturiranom logeru `slog` u mrežnom sloju, povezanim mostom uz serijalizaciju zapisa – zamenjeno je jedinstvenim, kontekstno svesnim mehanizmom zasnovanim na radnom okviru `tracing` u izvršnom okruženju `tokio`. Integracija je sprovedena inkrementalno i koordinisano, obuhvatajući i zasebnu mrežnu biblioteku `eth2_libp2p` i druge spoljne zavisnosti, uz očuvanje produkcijske spremnosti klijenta u svakoj fazi.

Doprinos rada obuhvata više celina. Uveden je jedinstveni pretplatnik i skup namenskih makroa koji svakom događaju automatski pridružuju kontekst. Slojevita organizacija pretplatnika omogućila je razdvajanje različitih tokova dijagnostičkih podataka, tako da se uobičajeni izlaz namenjen operateru čvora može nezavisno kontrolisati, dok se kritični izuzeci trajno čuvaju, uz rotaciju i kompresiju zapisa. Nivo praćenja može se menjati tokom rada čvora, prema trenutnim potrebama analize, bez restartovanja i bez prekida veze sa mrežom, čime se ne narušava kontinuitet rada klijenta.

Poseban deo integracije odnosio se na instrumentaciju asinhronog kôda. Ključne komponente klijenta Grandine instrumentovane su strukturiranim rasponima, uz posebnu pažnju da ta instrumentacija bude ispravna u višenitnom asinhronom izvršavanju nad okruženjem `tokio` i da događajima pruža dosledan kontekst.

Opisanim pristupom ostvareni su osnovni ciljevi rada. Praćenje izvršavanja postalo je detaljnije i pogodnije za analizu složenih asinhronih tokova, uz mogućnost dinamičkog prilagođavanja nivoa praćenja bez prekida rada čvora. Postojeći format ispisa i funkcionalnosti na koje se korisnici oslanjaju očuvani su radi kompatibilnosti sa postojećim alatima i procesima. Dodatni troškovi ograničeni su na periode aktivnog prikupljanja podataka, budući da su instrumentacija i dodatna polja neaktivni kada odgovarajući nivo praćenja

nije uključen. Praktični rezultat rada uključen je u razvojni tok klijenta Grandine i koristi se u produkcionom okruženju.

Zahvaljujući arhitekturi pretplatnika uvedene u ovom radu, uz razdvajanje generisanja događaja od njihove obrade, stvorena je osnova za povezivanje sa eksternim sistemima za opservabilnost i analizu izvršavanja. U Grandine klijentu, primenljivost takvog pristupa potvrđena je naknadnim dodavanjem sloja `tracing- opentelemetry` [52], koji `tracing` raspone i događaje prevodi u format OpenTelemetry [51] i omogućava njihovo dalje prosleđivanje i korišćenje u savremenim alatima.

Strukturirani podaci koje ovakav sistem za praćenje obezbeđuje predstavljaju osnovu za primenu savremenih alata zasnovanih na veštačkoj inteligenciji. Jasan uvid u ponašanje klijenta pri tome olakšava njegov dalji razvoj, skaliranje i integraciju novih tehnologija u brzo promenljivom Ethereum ekosistemu.

Bibliografija

[1] L. Lamport, R. Shostak, M. Pease, *The Byzantine Generals Problem*, ACM TOPLAS, 1982. https://dl.acm.org/doi/10.1145/357172.357176

[2] Cloudflare learning, *What is blockchain?* https://www.cloudflare.com/learning/security/what-is-blockchain/

[3] S. Nakamoto, *Bitcoin: A Peer-to-Peer Electronic Cash System*, 2008. https://bitcoin.org/bitcoin.pdf

[4] V. Buterin, *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform* (whitepaper), 2014. https://ethereum.org/en/whitepaper/

[5] G. Wood, *Ethereum: A Secure Decentralised Generalised Transaction Ledger* (yellow paper). https://ethereum.github.io/yellowpaper/paper.pdf

[6] Ethereum Foundation, *Introduction to Smart Contracts*. https://ethereum.org/developers/docs/smart-contracts/

[7] V. Buterin, V. Griffith, *Casper the Friendly Finality Gadget*, arXiv:1710.09437, 2017. https://arxiv.org/abs/1710.09437

[8] V. Buterin, D. Hernandez, T. Kamphefner et al., *Combining GHOST and Casper*, arXiv:2003.03052, 2020. https://arxiv.org/abs/2003.03052

[9] Ethereum Foundation, *The Merge*. https://ethereum.org/en/roadmap/merge/

[10] Ethereum Foundation, *Fulu-Osaka (Fusaka)*. https://ethereum.org/en/roadmap/fusaka/

[11] Ethereum Foundation, *History and forks of Ethereum* – vremenska linija svih hard-forkova. https://ethereum.org/en/history/

[12] Ethereum Foundation, *The Beacon Chain*. https://ethereum.org/en/roadmap/beacon-chain/

[13] Ethereum, *Consensus specifications* (`ethereum/consensus-specs`). https://github.com/ethereum/consensus-specs

[14] Ethereum, *Consensus spec tests* (`ethereum/consensus-spec-tests`). https://github.com/ethereum/consensus-spec-tests

[15] Ethereum, *Beacon API specification*. https://ethereum.github.io/beacon-APIs/#/Beacon

[16] Ethereum, *Engine API specification* (`ethereum/execution-apis`). https://github.com/ethereum/execution-apis/tree/main/src/engine

[17] Ethereum Foundation, *Proof-of-work (PoW)*. https://ethereum.org/en/developers/docs/consensus-mechanisms/pow/

[18] Ethereum Foundation, *Proof-of-stake (PoS)*. https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/

[19] Chainlink Education Hub, *What Is a Blockchain Validator?*. https://chain.link/article/validator

[20] Ethereum Foundation, *Attestations* (Proof-of-stake). https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/attestations/

[21] Dokumentacija Grandine. https://docs.grandine.io/

[22] Grandine – izvorni kôd (`grandinetech/grandine`). https://github.com/grandinetech/grandine

[23] Grandine – izdanje 2.0.0, prvo produkcijsko izdanje za *mainnet* sa integrisanim sistemom `tracing`. https://github.com/grandinetech/grandine/releases/tag/2.0.0

[24] Devconnect – međunarodni skup Ethereum zajednice u organizaciji Ethereum fondacije. https://devconnect.org/

[25] V. Marković, izlaganje o integraciji sistema `tracing` u klijent Grandine, Devconnect (Buenos Ajres, 2025), video snimak. https://www.youtube.com/watch?v=1ZktKW1NBzO&t=4791s

[26] Sigma Prime, *Lighthouse* – Ethereum konsenzusni klijent u jeziku Rust. https://github.com/sigp/lighthouse

[27] `tracing` – Application-level tracing for Rust. https://docs.rs/tracing

[28] `tracing-subscriber` dokumentacija. https://docs.rs/tracing-subscriber

[29] `tracing_subscriber::filter::EnvFilter` – dokumentacija. https://docs.rs/tracing-subscriber/latest/tracing_subscriber/filter/struct.EnvFilter.html

[30] `tracing_appender` – neblokirajući upisivač `tracing` zapisa u fajl, sa podrškom za rotaciju. https://docs.rs/tracing-appender/

[31] Tokio: asinhrona izvršna sredina za Rust. https://tokio.rs/

[32] *The Rust Programming Language*. https://www.rust-lang.org/

[33] *Ethereum Protocol Fellowship* – program Ethereum fondacije za doprinose protokolu. https://blog.ethereum.org/2025/04/10/epf-6

[34] Ethereum Protocol Fellowship (Cohort Six) – *Grandine: Implementing Tokio Tracing for Debugging and Performance Analysis*, tema projekta iz liste projektnih ideja programa. https://github.com/eth-protocol-fellows/cohort-six/blob/master/projects/project-ideas.md#grandine-implementing-tokio-tracing-for-debugging-and-performance-analysis

[35] V. Marković, *Grandine: Implementing Tokio Tracing for Debugging and Performance Analysis* – predlog projekta, Ethereum Protocol Fellowship (Cohort Six). https://github.com/eth-protocol-fellows/cohort-six/blob/master/projects/Grandine-Implementing-Tokio-Tracing-For-Debugging-And-Performance-Analysis.md

[36] V. Marković, *Grandine: Implementing Tokio Tracing for Debugging and Performance Analysis* – plan (*roadmap*) projekta, Ethereum Protocol Fellowship (Cohort Six). https://github.com/eth-protocol-fellows/cohort-six/blob/master/projects/Grandine-Implementing-Tokio-Tracing-For-Debugging-And-Performance-Analysis.md#roadmap

[37] V. Marković, Grandine PR #422 – migracija na `tokio-tracing`. https://github.com/grandinetech/grandine/pull/422

[38] Grandine `eth2_libp2p` – mrežna (p2p) biblioteka, fork biblioteke `lighthouse-network`. https://github.com/grandinetech/eth2_libp2p

[39] V. Marković, Grandine PR #432 – dvoslojni pretplatnik, `exception.log` i lokalna vremenska zona. https://github.com/grandinetech/grandine/pull/432

[40] V. Marković, Grandine PR #456 – rotacija `exception.log` preko 1 GB. https://github.com/grandinetech/grandine/pull/456

[41] V. Marković, `eth2_libp2p` PR #21 – migracija na `tokio-tracing`. https://github.com/grandinetech/eth2_libp2p/pull/21

[42] V. Marković, `eth2_libp2p` PR #22 – preimenovanje `crit` → `exception`. https://github.com/grandinetech/eth2_libp2p/pull/22

[43] `discv5` – implementacija Ethereum Node Discovery protokola v5. https://github.com/sigp/discv5

[44] `env_logger` – biblioteka za logovanje u Rustu. https://docs.rs/env_logger/latest/env_logger/

[45] `slog` – strukturirano logovanje za Rust. https://docs.rs/slog/latest/slog/

[46] `slog-stdlog` – adapter (most) između `slog` i `log` u Rustu. https://docs.rs/slog-stdlog/latest/slog_stdlog/

[47] Ethereum Foundation, *Client diversity* (raznovrsnost klijenata). https://ethereum.org/en/developers/docs/nodes-and-clients/client-diversity/

[48] `tracing` – *EnteredSpan in asynchronous code*. https://docs.rs/tracing/latest/tracing/span/struct.EnteredSpan.html#in-asynchronous-code

[49] V. Marković, *Network layer async functions not Send* – issue #14, `sntntn/grandine`. https://github.com/sntntn/grandine/issues/14

[50] Grafana. https://grafana.com/docs/grafana/latest/

[51] OpenTelemetry – posmatrački okvir i OTLP protokol. https://opentelemetry.io/

[52] `tracing-opentelemetry` – sloj koji `tracing` raspone prevodi u OpenTelemetry. https://docs.rs/tracing-opentelemetry/

Biografija

Vukašin Marković rođen je 2. maja 2000. godine u Beogradu. U Beogradu je završio srednju Elektrotehničku školu Nikola Tesla, smer elektrotehničar računara, a obrazovanje je nastavio na Matematičkom fakultetu Univerziteta u Beogradu. Tamo je završio osnovne studije na smeru Informatika, a potom upisao master studije istog smera.

Tokom osnovnih studija razvio je interesovanje za oblasti algoritama i razvoja softvera, što ga je kasnije podstaklo da se posveti istraživanju otvorenih problema u industriji iz tih oblasti. Ethereum protokol, koji je područje primene najnovijih istraživanja iz industrije i akademije i koji se neprekidno suočava sa aktuelnim i još nerešenim izazovima, svojom prirodom privukao je Vukašina da karijeru i svoje doprinose usmeri u tom pravcu. Postao je deo Ethereum zajednice i danas se, kao softverski inženjer i istraživač, bavi razvojem te tehnologije.