

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 2565

**Analiza performansi sustava za
udaljeno izvršavanje programskog
kôda**

Herman Zvonimir Došilović

Zagreb, srpanj 2021.

**SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA**

Zagreb, 12. ožujka 2021.

DIPLOMSKI ZADATAK br. 2565

Pristupnik: **Herman Zvonimir Došilović (0036480275)**

Studij: Računarstvo

Profil: Računarska znanost

Mentor: izv. prof. dr. sc. Igor Mekterović

Zadatak: **Analiza performansi sustava za udaljeno izvršavanje programskog kôda**

Opis zadatka:

Sustavi za udaljeno izvršavanje programskog kôda neizostavan su dio moderne arhitekture web aplikacija koje omogućuju svojim korisnicima programiranje iz internet preglednika. Takvih aplikacija je sve više i najčešće se koriste u svrhu edukacije, ispitivanja znanja, natjecanja i sl. Sustavi za udaljeno izvršavanje programskog kôda moraju biti sigurni, brzi i učinkoviti kako bi programiranje za krajnjeg korisnika bilo pristupačnije i ugodnije. U tu svrhu potrebno je procijeniti koliko opterećenje sustav može podnijeti kako bi se odgovarajuće projektirao i u konačnici skalirao. Ovo pitanje nije trivijalno jer opterećenje ne ovisi samo o broju korisnika već i o programskom jeziku i o samom programu kojeg korisnik pokreće. Potrebno je upoznati se s više sustava za udaljeno izvršavanje programskog kôda i osmisliti testove i metrike s kojima bi se izvana evaluirali takvi sustavi. Napraviti podesivi program koji će to implementirati i omogućiti ispitivanje i analizu performansi sustava za udaljeno izvršavanje programskog kôda. Program treba omogućiti izvještaje i vizualizacije kojima će se ostvariti uvid u performanse testiranih sustava s obzirom na različite obrasce korištenja i korisničke profile. Donijeti ocjenu ostvarenog pristupa te smjernice za budući razvoj.

Rok za predaju rada: 28. lipnja 2021.

Deo grátias.

SADRŽAJ

1. Uvod	1
2. Arhitektura ekosustava sustava za udaljeno ocjenjivanje	4
2.1. Zaštićeno okruženje	5
2.2. Modul za izvršavanje programskog kôda	7
2.3. Sustav za udaljeno izvršavanje programskog kôda	8
2.3.1. Sustav Sphere Engine	11
2.3.2. Sustav Piston	12
2.3.3. Sustav Judge0	13
2.4. Sustav za udaljeno ocjenjivanje	14
2.4.1. Interakcija sa sustavom za udaljeno izvršavanje programskog kôda	15
2.5. <i>Web</i> aplikacija za udaljeno programiranje	17
2.6. Specijalizirane izvedbe sustava za udaljeno ocjenjivanje	17
3. Analiza performansi sustava za udaljeno izvršavanje programskog kôda	21
3.1. Dinamike višekorisničkog opterećenja	21
3.1.1. Impulsno-determinističko opterećenje sustava	22
3.1.2. Kontinuirano-determinističko opterećenje sustava	23
3.1.3. Kontinuirano-stohastičko opterećenje sustava	24
3.2. Scenariji korištenja i programski jezici	25
3.2.1. Jednostavan scenarij	26
3.2.2. Scenarij procesorskog opterećenja	26
3.2.3. Scenarij mrežnog opterećenja	26
3.2.4. Scenarij procesorskog i mrežnog opterećenja	27
3.3. Strategije izvršavanja	27
3.4. Metrike za analizu performansi	28

4. Aplikacija Hélory	29
4.1. Korištene tehnologije	29
4.2. Pregled komandno-linijskog sučelja	30
4.2.1. Pokretanje višekorisničkog opterećenja	32
4.2.2. Odabir dinamike višekorisničkog opterećenja	32
4.2.3. Odabir sustava za udaljeno izvršavanje programskog kôda . .	33
4.2.4. Odabir scenarija	35
4.2.5. Odabir jezika	35
4.3. Pregled sučelja grafičkog izvještaja	36
5. Primjer korištenja aplikacije Hélory	40
5.1. Analiza performansi sustava Piston	40
5.2. Analiza performansi sustava Judge0	42
6. Budući razvoj	47
7. Zaključak	48
Literatura	49

1. Uvod

Računarsko razmišljanje (engl. *computational thinking*) je termin kojeg je prvi put upotrijebio Seymour Aubrey Papert 1980. godine u svojoj knjizi *Mindstorms: Children, Computers, and Powerful Ideas* (Papert, 1980) u kojoj, između ostalog, razmatra korištenje računala kao alata koji može utjecati na način na koji ljudi uče i razmišljaju. Njegova vizija korištenja računala u edukaciji djece bila je naučiti djecu programirati računalo s ciljem da djeca već u ranoj dobi dobiju neke od najdubljih ideja iz raznih znanstvenih disciplina. Papert programiranje računala opisuje kao komunikaciju između čovjeka i računala jezikom koji oboje razumiju, i time je njegova ideja razvoja računala, s kojim je prirodno naučiti komunicirati, temeljna ideja koja se provlači kroz cijelu knjigu. Njegove ideje i inovacije promijenile su način na koji milijuni djece stvaraju i uče (MIT Media Lab, 2016).

Termin *računarsko razmišljanje* ponovo dobiva značajnu pažnju akademske zajednice (Grover i Pea, 2013) 2006. godine objavom istoimenog rada Jeannette M. Wing koja računarsko razmišljanje svrstava u fundamentalnu vještina pored čitanja, pisanja i aritmetike (Wing, 2006). Njezin je rad potaknuo intenzivnije istraživanje o važnosti uključivanja računarskog razmišljanja u edukacijski sustav (Lockwood i Mooney, 2017). U svom izvješću iz 2014. godine internacionalna neprofitna organizacija European Schoolnet prepoznala je važnost integracije računarskog razmišljanja u edukacijski sustav te navode da već tada programiranje počinje biti sve ključnija vještina koju će morati steći svi mлади, ali i radnici iz širokog spektra industrije (Balanskat i Engelhardt, 2014). Navode, također, da je programiranje dio logičkog rasuđivanja i da ono pripada ključnim vještinama 21. stoljeća, što dodatno potkrjepljuje ideje Jeannette M. Wing iz 2006. godine.

Programiranje je od 2015. godine uključeno u nastavni plan i program 16 Europskih zemalja koje kod svojih učenika žele razviti vještine logičkog razmišljanja i rješavanja problema, ali i povećati mogućnost zapošljavanja (Balanskat i Engelhardt, 2015). Budući da se vještina programiranja stječe vježbanjem (Kurnia et al., 2001), institucije, koje u svom nastavnom planu i programu nude programiranje, svojim studentima

trebaju omogućiti pristup biblioteci problemskih zadataka. Programska rješenja studenata konačno netko treba i ocijeniti, a tradicionalno ručno ocjenjivanje programskog kôda unosi dodatne probleme (Kurnia et al., 2001).

Probleme koje unosi tradicionalno ručno ocjenjivanje programskog kôda rješavaju **sustavi za udaljeno ocjenjivanje** (engl. *online judges*, krat. *OJs*) koji poboljšavaju proces edukacije i ocjenjivanja, ali i omogućuju studentima da prevedu (engl. *compile*) i izvrše (engl. *execute*) svoj programski kôda prije predaje (Wang et al., 2021). Sustavi za udaljeno ocjenjivanje, poput *web* aplikacije Edgar (Mekterović et al., 2020) razvijene na Fakultetu elektrotehnike i računarstva Sveučilišta u Zagrebu, omogućuju nastavnicima izradu *online* provjera znanja koje pomoću ispitnih primjera automatski ocjenjuju rješenja studenata predana u obliku programskog kôda i time ubrzavaju proces ispravljanja i ocjenjivanja, i smanjuju mogućnost uvođenja grešaka koje se mogu dogoditi prilikom ručnog ocjenjivanja. Također, sustavi za udaljeno ocjenjivanje omogućuju studentima brzi pristup problemskim zadacima koje mogu rješavati i time vježbati svoju vještina programiranja. Sustavi za udaljeno ocjenjivanje najčešće omogućuju studentima programiranje u pregledniku odnosno kroz *web* aplikaciju koju koristi njihova institucija. Mogućnost programiranja i predaje rješenja kroz *web* aplikaciju olakšava studentima, pogotovo početnicima, proces postavljanja okoline za programiranje u programskom jeziku koji pokriva nastavni plan i program koji pohađaju.

Jasno je dakle da se programski kôd studenta ne može prevesti i izvršiti direktno na njegovom računalu budući da na njemu nema postavljenju okolinu koja to omogućuje. Programski kôd studenta kojeg piše u svom pregledniku izvršava se na poslužitelju sustava za udaljeno ocjenjivanje. Sustav za udaljeno ocjenjivanje, poput već spomenute *web* aplikacije Edgar, oslanja se na **sustav za udaljeno izvršavanje programskog kôda** (engl. *online code execution system*, krat. *OCES*) koji mu osigurava da će se programski kôd studenta prevesti i izvršiti bez negativnih posljedica za poslužitelj i ostalih korisnika aplikacije. Stoga OCES predstavlja ključnu komponentu u arhitekturi sustava za udaljeno ocjenjivanje o kojoj ovisi prije svega sigurnost i integritet poslužitelja, a zatim i korisničko iskustvo, i kvaliteta usluge.

Sustavi za udaljeno izvršavanje programskog kôda nude *web* aplikacijsko sučelje koje sustavi za udaljeno ocjenjivanje mogu transparentno koristiti za stvaranje zahtjeva za izvršavanje programskog kôda, dohvati informacija o izvršavanju i dohvati rezultata izvršavanja. Sučelje koje takvi sustavi nude je jednostavno za korištenje, međutim, usluga koju nude je resursno i sigurnosno zahtjevna. Dobro izveden i skaliran sustav za udaljeno izvršavanje programskog kôda donosi veliku vrijednost sustavima za udaljeno ocjenjivanje budući da se razvojni inženjeri tih sustava tada mogu u potpunosti

fokusirati na razvoj proizvoda i njegovih funkcionalnosti što krajnjem korisniku donosi bolje korisničko iskustvo i kvalitetniju uslugu.

Budući da se sustavi za udaljeno izvršavanje programskog kôda u literaturi razmatraju kao zasebne komponente arhitekture sustava za udaljeno ocjenjivanje tek od (Đošilović i Mekterović, 2020), ne postoji u literaturi radni okvir za analizu performansi i ocjenu kvalitete i pouzdanosti takvih sustava. Performanse, kvaliteta i pouzdanost sustava za udaljeno izvršavanje programskog kôda dolaze do izražaja pri intenzivnom višekorisničkom opterećenju koje, u kontekstu sustava za udaljeno izvršavanje programskog kôda, predstavljaju zahtjevi za izvršavanje programskog kôda koje sustav dobiva kroz *web* aplikacijsko sučelje.

Ovaj rad predstavlja prvi radni okvir za analizu performansi i ocjenu kvalitete i pouzdanosti usluge koju nude sustavi za udaljeno izvršavanje programskog kôda. Ovaj rad također predstavlja i aplikaciju Hélory¹ koja implementira predstavljeni radni okvir za analizu performansi i ocjenu kvalitete i pouzdanosti za tri sustava za udaljeno izvršavanje programskog kôda: Sphere Engine, Piston i Judge0. Razvijen u obliku komandno-linijske aplikacije, Hélory nudi jednostavno pokretanje višekorisničkog opterećenja na željenom sustavu, a nakon eksperimenta Hélory će generirati i pohraniti izvještaj o pokrenutom eksperimentu koji sadrži detaljne informacije o svakom pojedinom zahtjevu za izvršavanje i grafički prikaz metrika od interesa.

U poglavlju 2 opisana je arhitektura OJ ekosustava, dan je pregled slučajeva uporabe (engl. *use-cases*) i opisana je problematika udaljenog izvršavanja programskog kôda. Dodatno, poglavlje 2 donosi formalnu definiciju OCES-a kroz funkcionalne i nefunkcionalne zahtjeve, i daje pregled sustava koje Hélory podržava. Poglavlje 3 predstavlja radni okvir za analizu performansi i ocjenu kvalitete i pouzdanosti usluge OCES-a, dok poglavlje 4 daje pregled funkcionalnosti razvijene aplikacije Hélory. U poglavlju 5 pokazuje se primjer korištenja aplikacije Hélory u analizi performansi sustava Piston i Judge0. Konačno, poglavlje 6 daje smjernice za budući razvoj radnog okvira i aplikacije Hélory.

¹Sv. Ivo Hélory (1253. – 1303.), po kome je aplikacija dobila ime, zaštitnik je, između ostalog i sudaca (engl. *judges*).

2. Arhitektura ekosustava sustava za udaljeno ocjenjivanje

Sustavi za udaljeno ocjenjivanje dio su arhitekture OJ ekosustava prikazane na slici 2.1, a koja započinje zaštićenim okruženjem (engl. *sandbox*) koje osigurava sigurno prevođenje i izvršavanje programskog kôda korisnika na poslužitelju, i završava specijaliziranim izvedbama sustava za udaljeno ocjenjivanje koji se koriste u raznim slučajevima uporabe. Ovo poglavlje daje pregled svake komponente arhitekture OJ ekosustava i motivaciju za njezino korištenje.



Slika 2.1: Arhitektura OJ ekosustava. (Došilović i Mekterović, 2020)

2.1. Zaštićeno okruženje

Sustavi za udaljeno ocjenjivanje koriste se za evaluaciju programskog kôda korisnika na unaprijed zadanom skupu primjera za ispitivanje, što povlači pitanje gdje će se programski kôd korisnika prevesti i izvršiti budući da se bez toga ne može ocijeniti njegovo rješenje. Najjednostavnije bi bilo da korisnik samostalno prevede i izvrši svoj program, i da zatim u sustav postavi rješenja koja je dobio. Ovakav pristup bio bi u redu za napredne korisnike koji znaju instalirati i koristiti odgovarajući prevoditelj za odabrani programski jezik, međutim, za početnike bi ovakav pristup rezultirao jako lošim korisničkim iskustvom. Ovo naime nije jedini argument zašto nije dobro da korisnik samostalno prevodi, izvršava i postavlja rješenja svojih programa. Ispitni primjeri trebaju ostati nepoznati korisniku kako ih ne bi zloupotrijebio, jer je za neke zadatke dovoljno imati standardni ulaz (engl. *standard input*) iz kojeg se lako dobije očekivani izlaz (engl. *expected output*). Već se samo sa ovim snažno opravdanim zahtjevom može zaključiti da je programski kôd korisnika potrebno izvršiti na poslužitelju kojem korisnik nema pristup. Ovakav zahtjev otvara novi spektar problema koje je potrebno riješiti i koji su opisani u (Kurnia et al., 2001).

Pokretanje programskog kôda korisnika na udaljenom poslužitelju nameće izazov da se korisnikov program mora pokrenuti u zaštićenom okruženju kako ne bi negativno utjecao na rad poslužitelja i ostalih procesa. Postoje razne tehnike opisane u (Yi et al., 2014) koje se koriste za izvršavanje programskog kôda u zaštićenom okruženju, i u kontekstu sustava za udaljeno ocjenjivanje od zaštićenog okruženja očekujemo sljedeće funkcionalnosti:

- inicijalizaciju zaštićenog okruženja,
- mogućnost prenošenja programskog kôda korisnika u zaštićeno okruženje,
- prevođenje i izvršavanje programskog kôda u zaštićenom okruženju uz navođenje procesorskih i memorijskih ograničenja,
- prikupljanje standardnog izlaza (engl. *standard output*) programa i ostalih meta podatka o izvršavanju, i
- čišćenje (engl. *cleanup*) zaštićenog okruženja.

Sustavi Judge0 i Piston, o kojima će biti riječ nešto kasnije u ovom poglavlju, koriste Isolate (Mareš i Blackham, 2012) i Docker (Merkel, 2014) kao zaštićena okruženja za prevođenje i izvršavanje programskog kôda. Isolate i Docker oboje koriste kontrolne grupe (engl. *control groups*), značajke sustava Linux koje omogućuju definiranje resursnih ograničenja pojedinog procesa. Odabir korištenja jednog, odnosno

drugog zaštićenog okruženja ovisi, između ostalog, o preferencijama razvojnih inženjera.

Važnost korištenja zaštićenog okruženja koje osigurava da programski kôd korisnika ne šteti radu poslužitelja i ostalih procesa prikazuju isječci 2.1 i 2.2 napisani u programskom jeziku C.

Isječak 2.1 prikazuje ispravan C program koji uključuje uređaj pseudo-slučajnih brojeva sustava Linux. Budući da uređaj generira beskonačni slijed pseudo-slučajnih brojeva, uključivanje tog uređaja dovodi do beskonačnog vremena prevođenja. Ukoliko više korisnika napravi zahtjev za izvršavanjem ovakvog ili sličnog programa, na poslužitelju brzo dolazi do iscrpljivanja procesorskih i memorijskih resursa i time postaje neupotrebljiv. Iz ovog je jednostavnog primjera jasno da je proces prevođenja potrebno pokrenuti u zaštićenom okruženju koje će imati ograničene procesorske i memorijске resurse.

```
1 #include </dev/random>
2 int main() {
3     return 0;
4 }
```

Isječak 2.1: Ispravan C program s beskonačnim vremenom prevođenja.

Izoliranje izvršavanja programskog kôda korisnika još je važnije od izolacije prevođenja, budući da je za zloupotrebu neizoliranog procesa prevođenja potrebno nešto više znanja o prevodiocima i programskom jeziku koji se koristi. Isječak 2.2 prikazuje C program koji će beskonačno mnogo puta stvoriti novi proces, i svaki novostvoreni proces napraviti isto. Broj novostvorenih procesa može se opisati eksponencijalnom funkcijom $f(x) = 2^x$, gdje je x broj ponavljanja petlje. Dovoljno je da jedan korisnik pokrene ovakav program da se na poslužitelju iscrpe procesorski i memorijski resursi koji će ga učiniti neupotrebljivim.

```
1 #include <unistd.h>
2 int main() {
3     while(1) {
4         fork();
5     }
6     return 0;
7 }
```

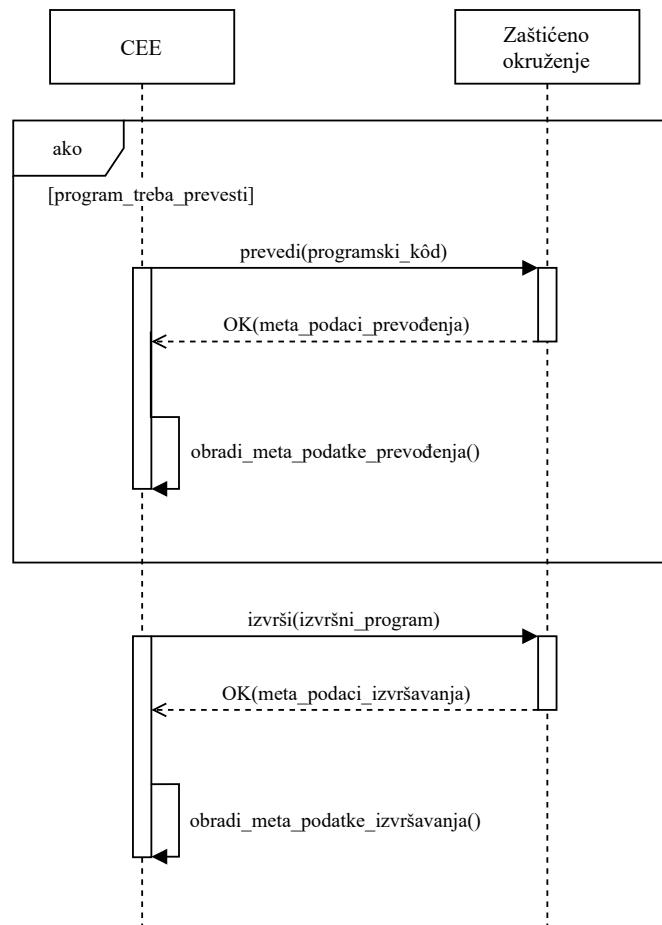
Isječak 2.2: C program s beskonačnim grananjem novih procesa.

Oba isječka potvrđuju da je u sustavima za udaljeno ocjenjivanje nužno koristiti zaštićeno okruženje koja će prevesti i izvršiti programski kôd svakog korisnika.

Gledajući iz perspektive sustava za udaljeno ocjenjivanje svaki programski kôd korisnika smatra se opasnim za poslužitelj, stoga se u kontekstu sustava za udaljeno ocjenjivanje govori o nepouzdanom programskom kôdu (engl. *untrusted source code*) kojeg je potrebno prevesti i izvršiti.

2.2. Modul za izvršavanje programskog kôda

Modul za izvršavanje programskog kôda (engl. *code execution engine*, krat. *CEE*) iduća je komponenta u arhitekturi OJ ekosustava koja koristi zaštićeno okruženje za prevođenje i izvršavanje programskog kôda, i koja zatim obrađuje meta podatke o prevođenju i izvršavanju koje generira zaštićeno okruženje. Slika 2.2 prikazuje dijagram toka interakcije između modula za izvršavanje programskog kôda i zaštićenog okruženja.

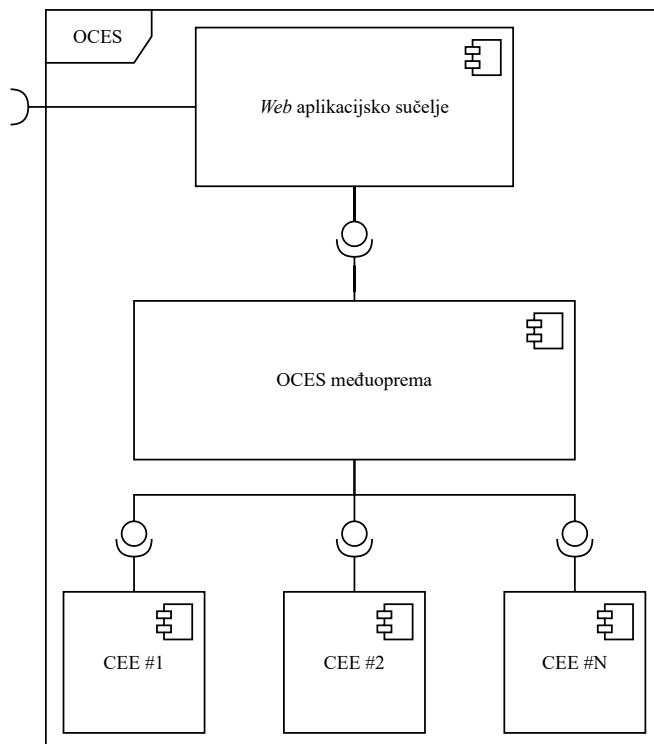


Slika 2.2: Interakcija modula za izvršavanje programskog kôda s zaštićenim okruženjem.

2.3. Sustav za udaljeno izvršavanje programskog kôda

Sustav za udaljeno izvršavanje programskog kôda je sustav koji nudi *web* aplikacijsko sučelje (engl. *web API*) za prevođenje i izvršavanje proizvoljnog programskog kôda. Arhitekturalni prikaz na slici 2.1 izdvaja sustav za udaljeno izvršavanje programskog kôda kao zasebnu komponentu na koju se oslanjaju sustav za udaljeno ocjenjivanje, aplikacija za udaljeno programiranje, a zatim i specijalizirane izvedbe sustava za udaljeno ocjenjivanje razvijene za specifičan slučaj uporabe. Sustav za udaljeno izvršavanje programskog kôda najčešće je strogo povezan i zavisan o ostatku komponenata u arhitekturu sustava za udaljeno ocjenjivanje (Došilović i Mekterović, 2020), međutim, u ovom radu razmatraju se oni sustavi za udaljeno izvršavanje programskog kôda koji su neovisni o ostatku arhitekture koja se na njih oslanja.

Slika 2.3 prikazuje dijagram komponenti sustava za udaljeno izvršavanje programskog kôda koji preko *web* aplikacijskog sučelja nudi uslugu izvršavanja programskog kôda (lijeva priključnica na slici 2.3) koju koriste sustav za udaljeno ocjenjivanje ili aplikacije za udaljeno programiranje.



Slika 2.3: Dijagram komponenti sustava za udaljeno izvršavanje programskog kôda.

Putem OCES međupreme (engl. *middleware*), zahtjev za izvršavanje programskog kôda dolazi do modula za izvršavanje programskog kôda kojih u sustavu može biti više kako bi se prilikom intenzivnog višekorisničkog opterećenja zahtjevi za izvršavanje efikasnije obavili. Način izvedbe OCES međupreme ovisi o pojedinom sustavu za udaljeno izvršavanje programskog kôda kao što se npr. međuprema sustava Judge0 sastoji od PostgreSQL baze podataka koja pohranjuje sve zahtjeve za izvršavanje programskog kôda i rezultate prevođenja i izvršavanja, i Redis baze podataka koja se koristi kao red poslova (engl. *job queue*) (Došilović i Mekterović, 2020).

Formalna definicija sustava za udaljeno izvršavanje programskog kôda dana je u (Došilović i Mekterović, 2020) kroz funkcionalne i nefunkcionalne zahtjeve. Funkcionalni zahtjevi podijeljeni su u nužne i dovoljne, odnosno na zahtjeve koje sustav nužno mora ispuniti i dodatne neobavezne zahtjeve. Nefunkcionalni zahtjevi su također neobavezni, ali preporučeni.

Funkcionalni zahtjevi

Sustav za udaljeno izvršavanje programskog kôda **mora** pružiti:

- dobro dokumentirano *web* aplikacijsko sučelje,
- podršku za prevođenje i izvršavanje proizvoljnog programskog kôda za barem jedan programski jezik,
- izvršavanje u zaštićenom okruženju s prepostavljenim procesorskim i memorijskim ograničenjima,
- podršku za navođenje proizvoljnog standardnog ulaza, i
- podršku za dohvata rezultata izvršavanja programa koji sadrži barem standardni izlaz programa.

Dodatno, ali neobavezno, sustav za udaljeno izvršavanje programskog kôda **može** pružiti podršku za:

- navođenje dodatnih opcija prevodioca,
- navođenje komandno-linijskih argumenata,
- navođenje resursnih ograničenja poput npr. procesorskih i memorijskih ograničenja,
- inicijalizaciju zaštićenog okruženja s dodatnim datotekama,
- dohvata detaljnih meta podataka o izvršavanju programa (npr. vrijeme izvođenja programa ili sadržaj standardnog izlaza za pogreške (engl. *standard error*)),

- skupne zahtjeve za izvršavanje,
- višestruke standardne ulaze, i
- autentifikaciju.

Čim više funkcionalnih zahtjeva sustav ispunjava tim više slučajeva uporabe može zadovoljiti. (Došilović i Mekterović, 2020)

Nefunkcionalni zahtjevi

Preporuča se da sustav za udaljeno izvršavanje programskog kôda bude:

- skalabilan,
- konfigurabilan,
- siguran,
- lako upogonljiv,
- efikasan,
- otporan,
- proširiv, i
- robustan.

Čim više nefunkcionalnih zahtjeva sustav ispunjava tim ga je lakše integrirati u sustav specifično izведен za pojedini slučaj uporabe (Došilović i Mekterović, 2020). Skalabilnost i efikasnost sustava za udaljeno izvršavanje programskog kôda dolazi do izražaja prilikom izrazitog višekorisničkog opterećenja kada krajnji korisnici neposredno preko sustava za udaljeno ocjenjivanje stvaraju zahtjeve za izvršavanje. Tada se od sustava za udaljeno izvršavanje programskog kôda očekuje da neovisno o broju zahtjeva efikasno izvrši programski kôd korisnika i vrati rezultate izvršavanja sustavu za udaljeno ocjenjivanje jer će u protivnom krajne korisničko iskustvo biti narušeno dugim vremenom čekanja. Konfigurabilnost i proširivost sustava za udaljeno izvršavanje programskog kôda omogućuje razvojnim inženjerima da sustav prilagode svojim zahtjevima prilikom razvoja sustava za određeni slučaj uporabe. Laka upogonljivost sustava za udaljeno izvršavanje programskog kôda omogućava razvojnim inženjerima da brzo napreduju u razvoju sustava za određeni slučaj uporabe. Sigurnost sustava za udaljeno izvršavanje programskog kôda uvelike ovisi načinu na koji sustav iskorištava funkcionalnosti zaštićene okoline i koristi li ju uopće. Od sustava se očekuje da niti jedan programski kôd ne naruši sigurnost i integritet poslužitelja. Ako se i dogodi da neki programski kôd uzrukuje neuobičajeno ponašanje, od sustava se očekuje da bude

otporan i robustan na unutarnje pogreške koje se mogu dogoditi i da neovisno o njima vrati korisniku rezultat izvođenja.

U nastavku ove cjeline dan je pregled tri različita sustava za udaljeno izvršavanje programskog kôda koje podržava aplikacija Hélory.

2.3.1. Sustav Sphere Engine

Sustav za udaljeno izvršavanje programskog kôda Sphere Engine razvila je Poljska tvrtka Sphere Research Labs 2008. godine. Sphere Engine je jedan od vodećih komercijalnih sustava za udaljeno izvršavanje programskog kôda kojeg koristi više od 1000 institucija u 180 država. Jedna od najpoznatijih *web* aplikacija koja koristi Sphere Engine je SPOJ (Sphere Research Labs Sp. z o.o., 2004) koju je razvila ista tvrtka, a na kojoj korisnici mogu rješavati razne problemske zadatke iz domene natjecateljskog programiranja. Osim SPOJ-a, CodeChef (Turakhia, 2009) također koristi Sphere Engine za udaljeno izvršavanje programskog kôda.

Budući da je Sphere Engine zatvoreni komercijalni sustav, nisu dostupne informacije o njegovoj izvedbi, niti o tome u kojim tehnologijama je razvijen.

U svrhu ovog rada dovoljno je spomenuti da sustav omogućuje prevođenje i izvršavanje programskog kôda u mnoštvo programskih jezika i da nudi bogatu dokumentaciju *web* aplikacijskog sučelja.



Slika 2.4: Vizualni identitet sustava Sphere Engine. (Sphere Research Labs Sp. z o.o., 2008)

2.3.2. Sustav Piston

Sustav za udaljeno izvršavanje programskog kôda Piston (Seymour, 2018) je sustav otvorenog kôda koji je dostupan na platformi GitHub. Razvijen je i koristi se u edukativne svrhe zajednice koju vodi autor projekta.

Kao zaštićeno okruženje u kojoj se izvršava programski kôda korisnika, sustav Piston koristi Docker i nudi oskudnu dokumentaciju *web* aplikacijskog sučelja, međutim, kao i Sphere Engine, podržava mnoštvo programskih jezika koji se mogu koristiti. Zajednica koja razvija sustav Piston također je razvila i komandno-linijsku aplikaciju koja se povezuje sa sustavom i tako korisnicima omogućava da iz ljudske (engl. *shell*) pokrenu proizvoljni programski kôd, dobivajući tako privid da se njihov program izvršio izravno na njihovom računalu.



Slika 2.5: Vizualni identitet sustava Piston. (Seymour, 2018)

2.3.3. Sustav Judge0

Razvoj sustava Judge0 započet je 22. kolovoza 2016. godine na inicijativu autora ovog rada. Cilj razvoja sustava Judge0 bio je izgradnja novog, robustnog, skalabilnog i lako uporabljivog sustava za udaljeno izvršavanje programskog kôda koji se može jednostavno integrirati u razne *web* aplikacije koje trebaju funkcionalnost izvršavanja programskog kôda. Razvijen je u programskom jeziku Ruby koristeći radni okvir Ruby on Rails, a kao zaštićeno okruženje koristi Isolate (Mareš i Blackham, 2012) koji se također koristi u popularnom sustavu za udaljeno ocjenjivanje CMS (Maggiolo i Massellani, 2012b). Sustav CMS koristio se na najvećim međunarodnim natjecanjima iz natjecateljskog programiranja poput informatičke olimpijade (IOI) i srednjoeuropske informatičke olimpijade (CEOI) (Maggiolo i Massellani, 2012a). Akademskoj zajednici sustav Judge0 napokon je formalno predstavljen 2020. godine radom (Došilović i Mekterović, 2020).



Slika 2.6: Vizualni identitet sustava Judge0.¹

U vrijeme pisanja ovog rada, javna instanca sustava Judge0 izvršila je preko 12,7 milijuna programa korisnika diljem svijeta koji su sustav Judge0 integrirali u svoje proizvode. Dodatno, u vrijeme pisanja ovog rada, zabilježeno je preko 200 aktivnih instanci sustava Judge0 kojeg su razne organizacije pokrenule na svojim poslužiteljima.

Od akademske godine 2018./2019. sustav Judge0 aktivno se koristi na Fakultetu elektrotehnike i računarstva Sveučilišta u Zagrebu kao integracijski dio *web* aplikacije Edgar (Mekterović et al., 2020). Time je sustav Judge0, neposredno preko *web* aplikacije Edgar, koristilo preko 3000 studenata kroz nekoliko predmeta.

Sustav Judge0 odlikuje se prije svega bogatom dokumentacijom *web* aplikacijskog sučelja koje omogućuje jednostavno korištenje sustava s mnoštvom konfiguracijskih opcija. Obilje funkcionalnosti koje nudi Judge0 omogućuje razvojnim inženjerima laku integraciju s drugim sustavima raznih slučajeva uporabe.

¹Vizualni identitet sustava Judge0 osmislio je i izradio Emanuel Loborec.



Slika 2.7: Vizualni identiteti organizacija koje koriste sustav Judge0. (Došilović, 2020)

2.4. Sustav za udaljeno ocjenjivanje

Sustav za udaljeno ocjenjivanje formalno je definiran u (Wasik et al., 2018) kao udaljena usluga (engl. *online service*) koja u oblaku (engl. *cloud*) izvodi barem jednu od sljedećih faza:

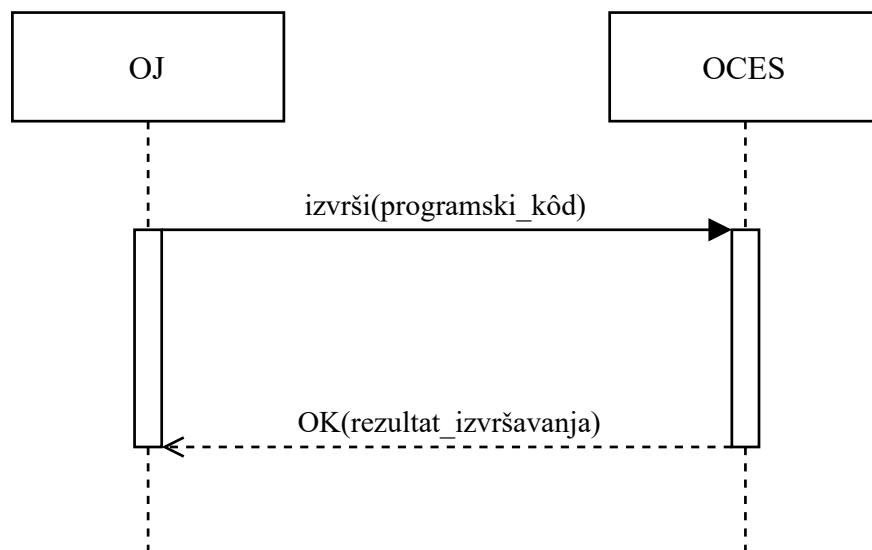
- podnošenje zahtjeva za izvršavanje programskog kôda (engl. *submission*),
- procjena podnesenoga programskog kôda (engl. *assessment*), i
- ocjenjivanje podnesenoga programskog kôda (engl. *scoring*).

Podnošenje zahtjeva za izvršavanje programskog kôda podrazumijeva prihvatanje programskog kôda u sustav, prevođenje po potrebi i verifikaciju da je program spreman za izvršavanje. U procjeni podnesenoga programskog kôda program se pokreće za svaki ispitni primjer i provjerava se uspješnost izvođenja programa sa zadanim procesorskim i memorijskim ograničenjima. Također, u istoj fazi, provjerava se standardni izlaz programa i uspoređuje ga se s očekivanim izlazom za trenutni ispitni primjer. Konačno, u fazi ocjenjivanja, dodjeljuje se ocjena na temelju rezultata iz prethodnih faza. Tako npr. na ocjenu u natjecateljskom programiranju može utjecati vrijeme koje je bilo potrebno natjecatelju da riješi pojedini zadatak. Što kasnije natjecatelj riješi ispravno zadatak to će manje bodova dobiti čak ako su mu svi ispitni primjeri ispravni uz zadana procesorska i memorijska ograničenja.

2.4.1. Interakcija sa sustavom za udaljeno izvršavanje programskog kôda

Interakcija sustava za udaljeno ocjenjivanje i sustava za udaljeno izvršavanje programskog kôda može se dogoditi na dva načina: sinkrono i asinkrono.

Slika 2.8 prikazuje dijagram toka sinkrone interakcije koja započinje zahtjevom za izvršavanjem koji OCES prihvati i u odgovoru vraća rezultate izvršavanja. Budući da izvršavanje programskog kôda može trajati i po nekoliko sekundi ova interakcija s OCES-om može iscrpiti njegove poslužiteljske resurse prilikom intenzivnog višekorisničkog opterećenja.

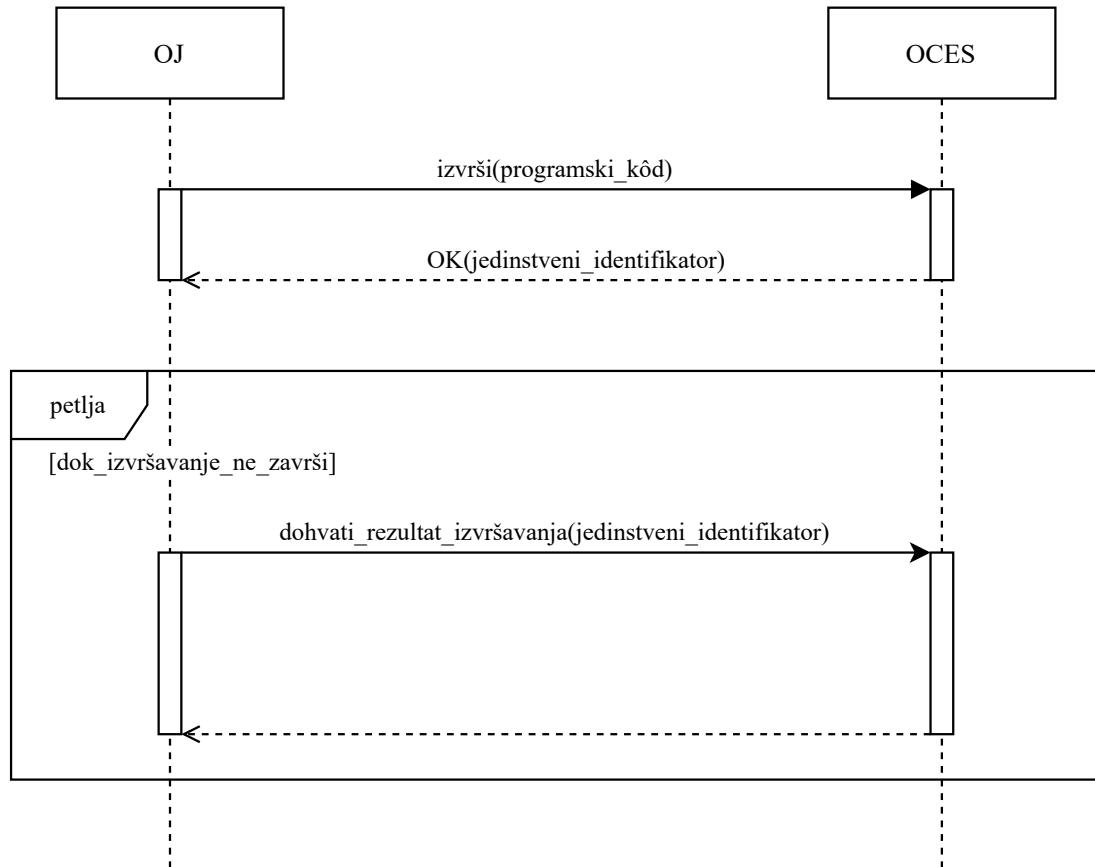


Slika 2.8: Sinkrona interakcija OJ-a i OCES-a.

Asinkrona interakcija (slika 2.9) započinje na isti način kao i sinkrona - zahtjevom za izvršavanje koji OCES prihvati, međutim, sada OCES vraća jedinstveni identifikator zahtjeva. Nakon što OJ primi jedinstveni identifikator svog zahtjeva na njemu je odgovornost da posebnim zahtjevom provjerava njegov status dok ne utvrdi da je izvršavanje završilo.

Sustav Piston podržava sinkronu interakciju, sustav Sphere Engine podržava asinkronu, dok sustav Judge0 podržava oba načina interakcije. Postoji zapravo još jedan način interakcije između OJ-a i OCES-a, a to je preko povratnog URL poziva (engl. *URL callback*), međutim, u ovom radu takva vrsta interakcije se ne razmatra. Ukratko, prilikom stvaranja zahtjeva OJ navodi URL kojeg OCES mora iskoristiti za stvaranje povratnog zahtjeva koji u svom tijelu (engl. *body*) sadrži rezultat izvršavanja

programa. Ovakva vrsta interakcije se ne razmatra u ovom radu jer klijent treba biti realiziran i kao poslužitelj koji može prihvati i obraditi povratni zahtjev koji dolazi od OCES-a. Sphere Engine i Judge0 podržavaju i ovakav način interakcije.



Slika 2.9: Asinkrona interakcija OJ-a i OCES-a.

2.5. Web aplikacija za udaljeno programiranje

Web aplikacija za udaljeno programiranje poput aplikacije Judge0 IDE (Došilović, 2016b) najjednostavniji je primjer uporabe sustava za udaljeno izvršavanje programskog kôda koje korisnicima omogućuju pisanje, prevođenje i izvršavanje programskog kôda u jednom od podržanih programskih jezika (slika 2.10).

The screenshot shows the Judge0 IDE interface. On the left is a code editor with a file named 'main.cpp' containing C++ code for Dijkstra's algorithm. The code includes imports for std::algorithm, std::cin, std::cout, std::limits, std::set, std::utility, and std::vector. It defines Vertex, Cost, Edge, Graph, and CostTable types. The main function implements Dijkstra's algorithm using a priority queue (minHeap) and a cost table. On the right, there are three tabs: 'STDIN' containing a sequence of numbers, 'STDOUT' showing the execution results, and 'STDERR' showing no errors. Below the tabs is a 'COMPILE OUTPUT' section which is empty. At the bottom, there are links for 'Become a Patron' and 'Donate with PayPal', and a note indicating the code was accepted with a runtime of 0.008s and a memory usage of 3324KB.

Slika 2.10: Sučelje web aplikacije Judge0 IDE.

2.6. Specijalizirane izvedbe sustava za udaljeno ocjenjivanje

Najčešća izvedba sustava za udaljeno ocjenjivanje su *web* aplikacije za natjecateljsko programiranje poput: Codeforces (Mirzayanov, 2009), CodeChef (Turakhia, 2009) i SPOJ (Sphere Research Labs Sp. z o.o., 2004) gdje se od korisnika očekuje da iz teksta zadatka prepozna i implementira algoritam i odgovarajuću strukturu podataka koji će zadovoljiti zadana vremenska i memorijska ograničenja.

Izvedba sustava za udaljeno ocjenjivanje kao *web* aplikacije za e-učenje poput *web* aplikacija Edgar (Mekterović et al., 2020) i CodeChum (Maranga et al., 2019) koristi se za ocjenjivanje (engl. *scoring*) programskog kôda kojeg student prilaže kao rješenje zadatka iz npr. domaće zadaće ili provjere znanja.



Slika 2.11: Sučelje *web* aplikacije Edgar iz perspektive učitelja.

Web aplikacije poput HackerRank (Ravisankar i Karunanidhi, 2012), TestDome (Švedić i Živić, 2013) i Filtered (Bilodeau, 2016) koriste se u regrutaciji i procesu zapošljavanja novih softverskih inženjera. Nakon što se prijave na otvorenu poziciju, kandidatima se automatski šalje elektronička pošta s uputama za rješavanje programskih zadataka. Zadatke koje će kandidat rješavati priprema tvrtka za čiju poziciju se kandidat natječe, a težina i vrsta zadataka ovisi o otvorenoj poziciji. Aplikacije za regrutaciju i automatizaciju procesa zapošljavanja najčešće nude svoju biblioteku zadataka koje tvrtke mogu odabrati prilikom izrade ispita za pojedini natječaj. Ovakve *web* aplikacije korisne su tvrtkama koje svakodnevno dobivaju preveliku količinu prijava koje odjel za ljudske resurse može pogledati, stoga im automatizirano ispitivanje kandidata pomaže u inicijalnom filtriranju prilikom zapošljavanja. Osim filtriranja kandidata ove *web* aplikacije pomažu i pri provođenju *online* razgovora gdje kandidati svoje vještine rješavanja problemskih zadataka trebaju pokazati pred ispitivačem koji vodi razgovor (slika 2.13).

Osim što pomažu u regrutaciji i procesu zapošljavanja, sustavi za udaljeno ocjenjivanje dolaze i u izvedbi kao *web* aplikacije za vježbanje i pripremu za razgovore za posao koji uključuju rješavanje problemskih zadataka. *Web* aplikacije poput: Algo-Expert (Mihailescu i Pourchet, 2017), AlgoDaily (Zhang, 2018) i LeetCode (Chang i Tang, 2015) svojim korisnicima nude bogatu biblioteku problemskih zadataka, njihovih rješenja i mnogo sadržaja za učenje algoritama i struktura podataka, a sve u svrhu kvalitetne pripreme za tzv. *coding interview*.

Slika 2.12: Sučelje web aplikacije CodeChum.

Slika 2.13: Sučelje web aplikacije HackerRank. (Ravisankar i Karunanidhi, 2012)

The screenshot shows the LeetCode homepage. At the top, there are navigation links: Explore, Problems, Interview, Contest, Discuss, and Store. Below the navigation is a banner for 'Category - Algorithms' featuring a 'Daily Challenge' section for June 17, and categories for Algorithms, Database, Shell, and Concurrency. A progress bar indicates 6/1723 solved problems, with 4 Easy, 2 Medium, and 0 Hard. A search bar allows users to search by question title, description, or ID. To the right of the search bar are filters for Difficulty, Status, Lists, Tags, and a 'Pick One' button. A sidebar on the right displays three contests: 'LeetCode's Pick', 'Weekly Contest 246' (Sunday, June 19, 2:30 - 4:00AM UTC), and 'Biweekly Contest' (Every other Saturday, 2:30 - 4:00PM UTC). Below the contests is a 'Your Progress' section with a pie chart showing completion status for 1717 total problems, 6 solved, and 0 attempted. It also includes a 'Top Hits' section and a link to 'LeetCode Curated Algo ...'.

Slika 2.14: Sučelje web aplikacije LeetCode.

3. Analiza performansi sustava za udaljeno izvršavanje programskog kôda

Analiza performansi sustava za udaljeno izvršavanje programskog kôda ne spominje se u literaturi budući da se tek od (Došilović i Mekterović, 2020) sustavi za udaljeno izvršavanje programskog kôda razmatraju kao zasebne komponente u arhitekturi sustava za udaljeno ocjenjivanje. U radu (Drung et al., 2011) razmatra se optimizacija performansi postojećeg sustava za udaljeno ocjenjivanje, međutim, ono je usmjereno na ugađanje opcija operacijskog sustava i algoritama raspoređivanja zahtjeva za izvršavanje.

Ovaj rad predstavlja prvi radni okvir za analizu performansi i ocjenu kvalitete i pouzdanosti (u dalnjem tekstu: *radni okvir*) usluge sustava za udaljeno izvršavanje programskog kôda (u dalnjem tekstu: *sustav*). Prema (Nidhra i Dondeti, 2012) ovakva vrsta analize u spektru testiranja softvera naziva se *system testing*, a proučava kako sustav funkcionira u proizvodnjoskom okruženju iz perspektive korisnika. Također, prema uvidu u sustav koji se testira, ovakva vrsta testiranja naziva se *black box testing*.

Za analizu performansi i ocjenu kvalitete i pouzdanosti pojedinog sustava potrebno je promatrati rad sustava s različitim dinamikama višekorisničkog opterećenja, različitim scenarijima korištenja i različitim načinima interakcije sa sustavom. To su dimenzije po kojima možemo promatrati rad sustava, i njihov opis slijedu u nastavku ovog poglavlja.

3.1. Dinamike višekorisničkog opterećenja

Budući da sustav za udaljeno izvršavanje programskog kôda neposredno preko sustava za udaljeno ocjenjivanje koristi više krajnjih korisnika istovremeno, u analizi performansi potrebno je razmatrati višekorisničko opterećenje.

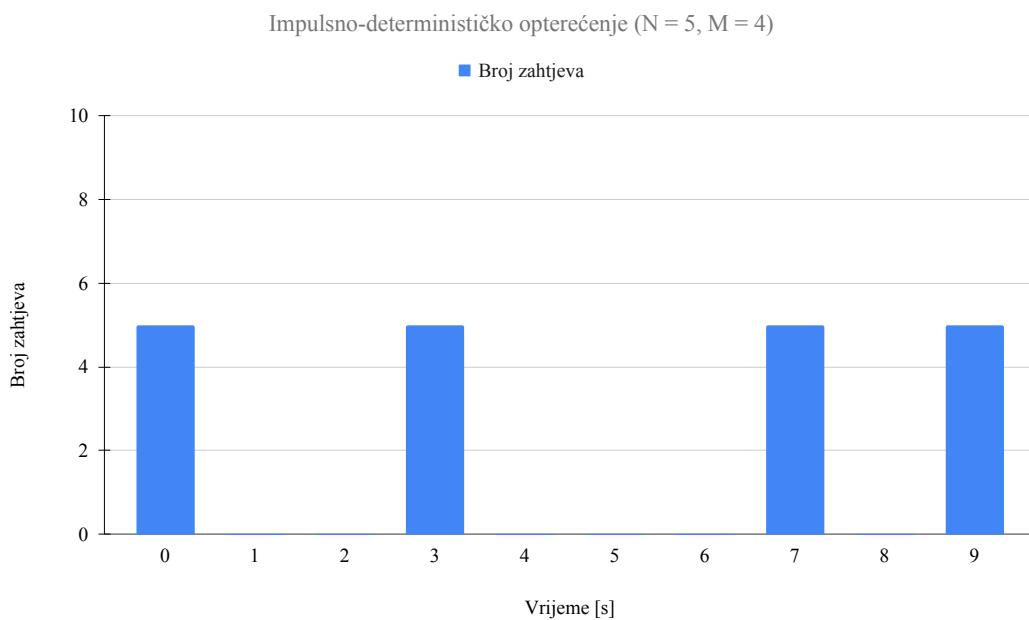
Radni okvir omogućuje razmatranje tri različite dinamike višekorisničkog opterećenja koje se koriste prilikom stvaranja zahtjeva za izvršavanje programskog kôda (u nastavku teksta: *zahtjevi*):

- impulsno-determinističko opterećenje,
- kontinuirano-determinističko opterećenje, i
- kontinuirano-stohastičko opterećenje.

Svaka vrsta višekorisničkog opterećenja detaljno je opisana u nastavku ove cjeline.

3.1.1. Impulsno-determinističko opterećenje sustava

Impulsno-determinističko opterećenje sustava definirano je parametrima N i M , gdje N predstavlja broj zahtjeva koje treba istovremeno poslati, dok M predstavlja broj iteracija koje je potrebno napraviti uz ogragu da zahtjevi poslani u iteraciji i smiju biti poslani tek nakon što se izvrše zahtjevi u iteraciji $i - 1$. U svakoj od M iteracija poslat će se N istovremenih zahtjeva, odnosno, na kraju eksperimenta očekujemo da je u sustav pristiglo sveukupno $N \times M$ zahtjeva.



Slika 3.1: Impulsno-determinističko opterećenje sustava ($N = 5, M = 4$).

Slika 3.1 prikazuje primjer dinamike impulsno-determinističkog opterećenja sustava s 5 istovremenih zahtjeva kroz 4 iteracije. U trenutku $t = 0$ započinje slanje prvih 5 zahtjeva koje je sustav obrađuje do trenutka $t = 3$ kada započinje slanje idućih 5

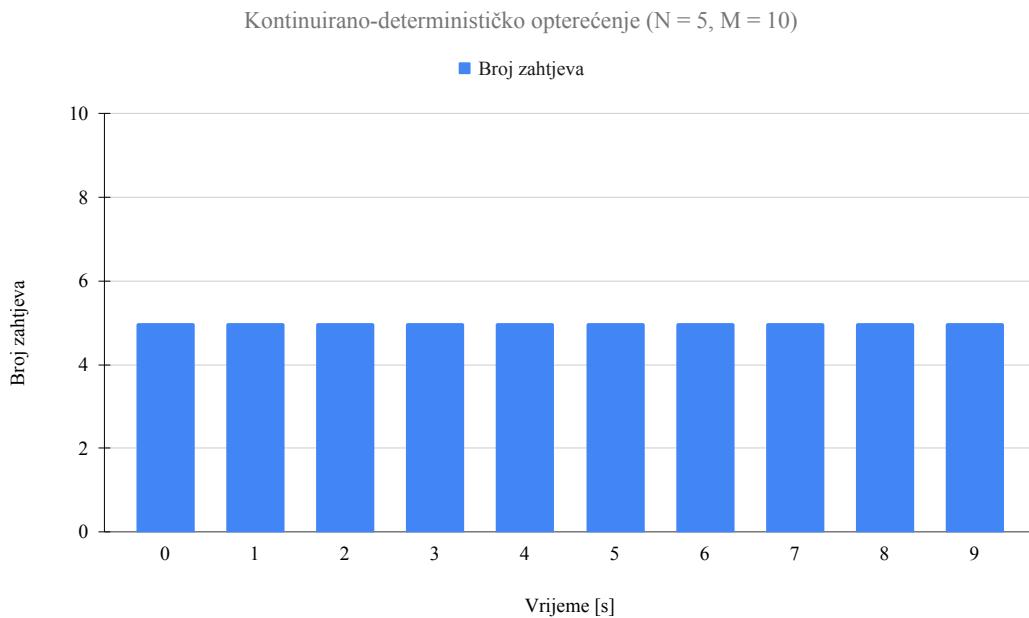
zahtjeva itd. Primjer na slici 3.1 također pokazuje da iako se u svakoj iteraciji šalje ista količina zahtjeva to ne znači nužno da će sustavu trebati isti vremenski period da ih obradi.

Impulsno-determinističko opterećenje sustava predstavlja najjednostavniju dinamiku opterećenja sustava, međutim, takva dinamika ne oponaša stvarno proizvodnjsko okruženje.

3.1.2. Kontinuirano-determinističko opterećenje sustava

Kontinuirano-determinističko opterećenje sustava definirano je također parametrima N i M , međutim, za razliku od prethodne vrste opterećenja, istovremeni zahtjevi u iteraciji i smiju biti poslani i prije nego što se izvrše zahtjevi u iteraciji $i - 1$, ali tek u vremenu $t_i = t_{i-1} + 1$, gdje je vrijeme t izraženo u sekundama. Budući da pri intenzivnom višekorisničkom opterećenju vrijeme obrade zahtjeva može trajati i po nekoliko desetaka sekundi, korak od jedne sekunde predstavlja najmanji interval vremena koji ima smisla promatrati kada se govori o broju zahtjeva koji dolaze u sustav po jedinici vremena.

Slika 3.2 prikazuje primjer dinamike kontinuirano-determinističkog opterećenja od 10 iteracija gdje je u svakoj iteraciji napravljeno 5 zahtjeva.



Slika 3.2: Kontinuirano-determinističko opterećenje sustava ($N = 5, M = 10$).

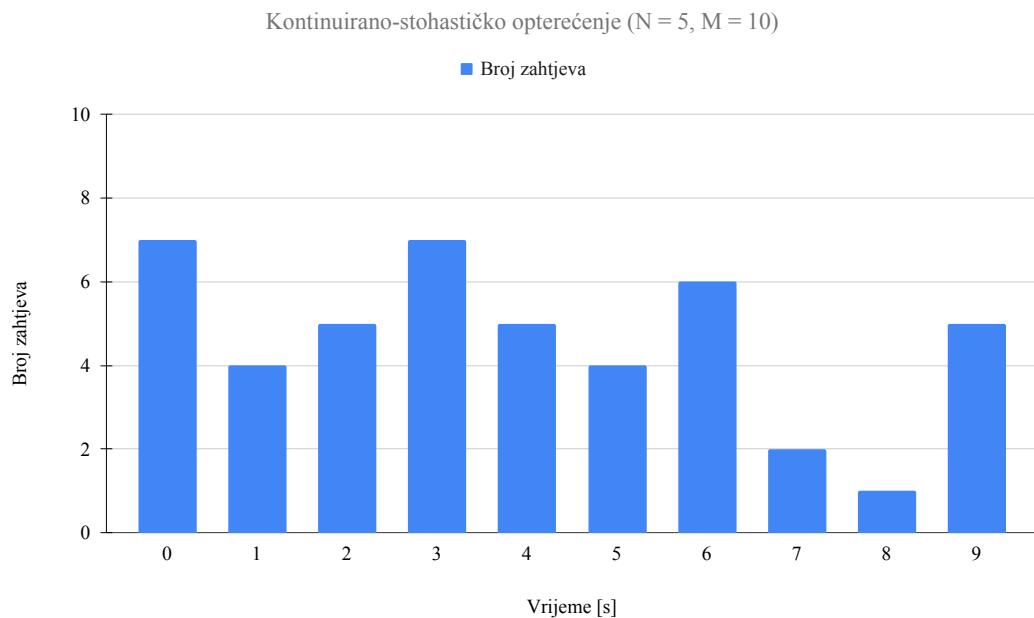
3.1.3. Kontinuirano-stohastičko opterećenje sustava

Za razliku od kontinuirano-determinističkog opterećenja koje u svakom koraku i koristi nepromjenjivi N , kontinuirano-stohastičko opterećenje sustava koristi Poissonovu distribuciju s parametrom $\lambda = N$ za određivanje broja istovremenih zahtjeva u koraku i . Slika 3.3 prikazuje primjer dinamike kontinuirano-stohastičkog opterećenja s parametrima $N = 5$ i $M = 10$. Niz pseudo-slučajnih brojeva iz primjera generiran je pomoću isječka 3.1 napisanog u programskom jeziku Python.

```
1 import numpy  
2 N, M = 5, 10  
3 print(numpy.random.poisson(N, M))
```

Isječak 3.1: Generiranje pseudo-slučajnih brojeva iz Poissonove distribucije.

Kontinuirano-stohastičko opterećenje sustava najbliže oponaša stvarnu dinamiku korištenja sustava prilikom višekorisničkog opterećenja. Parametrom N regulira se prosječan broj korisnika u sekundi, dok se parametrom M regulira duljina trajanja opterećenja.



Slika 3.3: Kontinuirano-stohastičko opterećenje sustava ($N = 5$, $M = 10$).

3.2. Scenariji korištenja i programski jezici

Nakon odabira dinamike višekorisničkog opterećenja potrebno je odabrati scenarij korištenja i programski jezik. U produkcijskom okruženju nemoguće je predvidjeti koji programski kôd će sustav trebati izvršiti, međutim, programski kôdovi koji pristižu u sustav mogu se podijeliti u četiri scenarija korištenja:

- jednostavan scenarij,
- scenarij procesorskog opterećenja,
- scenarij mrežnog opterećenja, i
- scenarij procesorskog i mrežnog opterećenja.

Svaki scenarij pažljivo je odabrani zadatak koji treba riješiti, međutim, za razliku od uobičajenih zadataka, scenarij također daje i uputu na koji način zadatak treba riješiti. Ova ograda važna je da bi se mogla raditi analiza između dva različita scenarija. Tako se npr. uz pažljivo odabранe scenarije mogu uspoređivati scenarij procesorskog opterećenja, i scenarij procesorskog i mrežnog opterećenja ako i samo ako se zna da je drugi zaista mrežno opterećena inačica prvog.

Prilikom analize prepostavlja se da su programski kôdovi scenarija ispravni, odnosno da se mogu uspješno prevesti i izvesti u svakom sustavu koji se promatra. Ova prepostavka uspješnosti prevođenja i izvršavanja ključna je u analizi budući da pri intenzivnom višekorisničkom opterećenju može doći do preopterećenja sustava koje rezultira npr. neuspješnim prevođenjem zbog manjka radne memorije.

Implementacija svakog scenarija korištenja dostupna je u tri programska jezika: C++, Java i Python. Iako se u okviru ovog rada promatraju samo navedeni programski jezici, radni okvir analize nije ograničen samo na njih. Također, u ovom radu ne razmatraju se višedretvene implementacije scenarija.

Svaki scenarij sastoji se od: standardnog ulaza, programskog kôda i očekivanog standardnog izlaza. Standardni ulaz i očekivani standardni izlaz isti su za svaki programski jezik u kojem je scenarij implementiran. Sadržaj standardnog ulaza dovodi se na ulaz programa koji je nastao prevođenjem programskog kôda scenarija u odabranom programskom jeziku, a očekivani standardni izlaz koristi se za usporedbu sa standardnim izlazom programa nakon njegovog izvršavanja. Ova provjera važna je kako bi se utvrdilo da je program zaista odradio zadani scenarij.

3.2.1. Jednostavan scenarij

Jednostavan scenarij zahtjeva da korisnički program ispiše `Hello, world` na standardni izlaz.

Jednostavan scenarij predstavlja najjednostavniji smisleni programski kôd koji korisnik može izvesti putem sustava. Ovaj scenarij također predstavlja minimalno mrežno, procesorsko i memorijsko opterećenje na poslužitelju sustava.

Vremenska i memorijksa složenost algoritma koji implementira rješenje ovog scenarija je $\mathcal{O}(1)$.

3.2.2. Scenarij procesorskog opterećenja

Scenarij procesorskog opterećenja zahtjeva ispis medijana od 20.000 cijelih brojeva tipa *integer* koji se na standardni ulaz dovode silazno sortirani. Scenarij također zahtjeva da se pročitanih 20.000 brojeva najprije uzlazno sortira mjehurićastim sortiranjem (engl. *bubble sort*).

Vremenska složenost algoritma koji implementira rješenje ovog scenarija je $\mathcal{O}(n^2)$, a memorijksa složenost je $\mathcal{O}(n)$, gdje je n duljina ulaznog niza brojeva.

Odabir 20.000 za duljinu ulaznog niza brojeva i zahtjev da brojevi budu sortirani suboptimalnim algoritmom mjehurićastog sortiranja nije slučajan, nego pažljivo odabran budući da vrijeme izvođenja programa koji implementira rješenje tog scenarija traje zamjetno dulje od jednostavnog scenarija. Konkretno, na procesoru Intel Core i7 radnog takta 2,6 GHz računala MacBook Pro (16-inch, 2019) C++ implementacija rješenja ovog scenarija traje oko 2,5 sekundi.

Ulagni niz brojeva koji se na standardni ulaz dovode silazno sortirani odabran je zato što predstavlja najgori mogući slučaj (engl. *worst case*) za algoritam mjehurićastog sortiranja koji brojeve treba sortirati uzlazno.

3.2.3. Scenarij mrežnog opterećenja

Scenarij mrežnog opterećenja očekuje da program ispiše sumu 750.000 cijelih brojeva tipa *integer* koji se na standardni ulaz dovode silazno sortirani.

Mrežno opterećenje ovog scenarija očituje se u veličini sadržaja standardnog ulaza u iznosu od 5,1 MB.

Vremenska i memorijksa složenost algoritma koji implementira rješenje ovog scenarija je $\mathcal{O}(n)$, gdje je n duljina ulaznog niza brojeva.

3.2.4. Scenarij procesorskog i mrežnog opterećenja

Scenarij procesorskog i mrežnog opterećenja spoj je prethodna dva scenarija. Ovaj scenarij zahtjeva da program ispiše medijan prvih 20.000 brojeva iz niza od 750.000 brojeva koji se na standardni ulaz dovode silazno sortirani. Također, kao i u scenariju procesorskog opterećenja, od programa se očekuje da brojeve sortira suboptimalnim algoritmom mješuričastog sortiranja.

Standardni ulaz ovog scenarija isti je kao i standardni ulaz scenarija mrežnog opterećenja, i algoritam rješenja ovog scenarija isti je kao algoritam rješenja scenarija procesorskog opterećenja.

Vremenska složenost algoritma koji implementira rješenje ovog scenarija je $\mathcal{O}(n^2)$, a memorijska složenost je $\mathcal{O}(n)$, gdje je n duljina ulaznog niza brojeva, uz ogragu da n ne može biti veći od 20.000.

3.3. Strategije izvršavanja

Odabir strategije izvršavanja programskog kôda svodi se na odabir načina interakcije sa sustavom koji su objašnjeni u pododjeljku 2.4.1.

Bilo da se radi o sinkronoj ili asinkronoj interakciji prvi zahtjev prema sustavu naziva se *zahtjev narudžbe* (engl. *order request*) koji sadrži sve potrebne informacije koje sustav treba za izvršavanje programskog kôda, a to su: programski kôd, identifikator programskog jezika, standardni ulaz i očekivani standardni izlaz.

Ako se koristi sinkrona interakcija sa sustavom onda će odgovor na zahtjev narudžbe biti rezultat izvršavanja programskog kôda.

Međutim, ako se koristi asinkrona interakcija tada je potrebno raditi tzv. *zahtjev ispitivanja* (engl. *probe request*) tako dugo dok se ne utvrdi da je izvršavanje završilo. Implementacija radnog okvira u aplikaciji Hélory radi jedan zahtjev ispitivanja po sekundi s time da između zahtjeva narudžbe i prvog zahtjeva ispitivanja mora proći dvije sekunde. Ova dinamika slanja zahtjeva ispitivanja odabrana je jer ju sustav Sphere Engine preporuča u svojoj dokumentaciji.

Sustav Piston podržava sinkronu strategiju izvršavanja, sustav Sphere Engine asinkronu, a sustav Judge0 podržava obje.

3.4. Metrike za analizu performansi

Radni okvir razmatra tri metrike koje se koriste u analizi performansi i ocjeni kvalitete i pouzdanosti sustava: uspješnost izvršavanja, vrijeme obrade, i maksimalno opterećenje.

Uspješnost izvršavanja

Budući da prilikom intenzivnog višekorisničkog opterećenja može doći do preopterećenja sustava, zahtjevi mogu biti odbačeni ili može doći do greške uslijed prevođenja ili izvršavanja zbog manjka slobodnih resursa na poslužitelju. Ovo su samo neki od brojnih razloga zbog čega sustav može neuspješno izvršiti programski kôd korisnika i razmatranje svakog pojedinog razloga izlazi iz opsega ovog rada, i ovisi o izvedbi svakog pojedinog sustava. Uspješnost pojedinog zahtjeva izvršavanja poprima Booleovu vrijednost, a promatrana metrika uspješnosti izvršavanja izražava se kao udio uspješno obrađenih zahtjeva.

Vrijeme obrade

Vrijeme obrade (engl. *turn around time*) najvažnija je metrika iz perspektive krajnjeg korisnika koje se definira kao vrijeme proteklo između trenutka kada je korisnik zahtjevom narudžbe poslao sve potrebne podatke sustavu, do trenutka kada je korisnik ustanovio da je izvršavanje završilo bilo zahtjevom ispitivanja u asinkronoj interakciji, bilo primanjem odgovora na zahtjev narudžbe u sinkronoj interakciji.

Maksimalno opterećenje

Maksimalno opterećenje koje sustav podržava definira se kao maksimalan broj istovremenih zahtjeva N koje sustav može u potpunosti uspješno obraditi za vrijeme kontinuirano-stohastičkog opterećenja, a da medijan vrijeme obrade zahtjeva ne raste s trajanjem eksperimenta.

4. Aplikacija Hélory

Aplikacija Hélory implementira radni okvir (opisan u poglavlju 3) za analizu performansi i ocjenu kvalitete i pouzdanosti usluge sustava za udaljeno izvršavanje programskog kôda. Komandno-linijsko sučelje aplikacije Hélory omogućuje pokretanje višekorisničkog opterećenja na tri različita sustava za udaljeno izvršavanje programskog kôda: Sphere Engine, Piston i Judge0.

Kao rezultat eksperimentalnog višekorisničkog opterećenja Hélory u tekućem direktoriju pohranjuje izvještaj u HTML formatu koji sadrži sve relevantne informacije o eksperimentu, grafički prikaz metrika od interesa i neobrađene podatke u formatu JSON prikupljene za vrijeme eksperimenta koji naknadno mogu biti dodatno iskorišteni.

Ovo poglavlje daje pregled korištenih tehnologija u razvoju aplikacije Hélory, a zatim i pregled komandno-linijskog sučelja i sučelja grafičkog izvještaja.

4.1. Korištene tehnologije

Hélory je razvijen u statičkom i tipiziranom programskom jeziku Go (Donovan i Kernighan, 2015) koji se brzo prevodi u strojni kôd i kojeg odlikuje ekspresivnost, preciznost, jednostavnost i čistoća u pisanju programskog kôda, ali i jednostavnost pisanja konkurenčnih programa koji efikasno iskorištavaju resurse domaćina. Izvještaj kojeg generira Hélory nakon eksperimenta napisan je u opisnom jeziku HTML uz korišteњe CSS-a i JavaScripta, a za prikaz metrika u grafičkom obliku koristi se biblioteka Chart.js.

4.2. Pregled komandno-linijskog sučelja

Isječak 4.1 prikazuje ispis aplikacije Hélory prilikom pokretanja bez komandno-linijskih argumenata. U ispisu su navedene sljedeće naredbe dostupne unutar aplikacije koje omogućuju:

- `endpoints` - ispis dostupnih instanci sustava za udaljeno izvršavanje programskog kôda koje je moguće koristiti prilikom pokretanja eksperimenata,
- `languages` - ispis dostupnih programskeh jezika,
- `providers` - ispis podržanih sustava za udaljeno izvršavanje programskog kôda,
- `run` - pokretanje višekorisničkog opterećenja, i
- `scenarios` - ispis dostupnih scenarija korištenja.

Naredbom `providers` aplikacija će ispisati podržane sustave za udaljeno izvršavanje programskog kôda, a to su: Sphere Engine, Piston i Judge0. Međutim, budući da je za svaki podržani sustav moguće imati više dostupnih instanci, naredba `endpoints` ispisuje koje su sve instance sustava za udaljeno izvršavanje programskog kôda na raspolaganju za odabir prilikom pokretanja eksperimenata.

```
$ ./helory
Helory is a performance analysis tool for online code execution systems.

Usage:
  helory [command]

Available Commands:
  endpoints      List all available endpoints
  help          Help about any command
  languages      List all available languages
  providers      List all supported providers
  run            Run benchmark
  scenarios      List all available scenarios

Flags:
  -h, --help           help for helory
  --log-level string   (default "info")

Use "helory [command] --help" for more information about a command.
```

Isječak 4.1: Pokretanje aplikacije Hélory iz ljske Bash.

Isječak 4.2 prikazuje primjer ispisa naredbe `endpoints` koji prikazuje dostupnost tri OCES instance jedinstvenih identifikatora: `judge0_ce`, `judge0_extra_ce` i `piston_public`. Jedinstveni identifikatori OCES instanci koristit će se u pokretanju eksperimenata kako bi aplikacija znala koji OCES da koristi za višekorisničko opterećenje.

```
$ ./helory endpoints
id: judge0_ce
name: Judge0 CE
provider: judge0
url: https://ce.judge0.com
supported_execution_strategies: both
default_execution_strategy: async

id: judge0_extra_ce
name: Judge0 Extra CE
provider: judge0
url: https://extra-ce.judge0.com
supported_execution_strategies: both
default_execution_strategy: async

id: piston_public
name: Piston (Free Public Instance)
provider: piston
url: https://emkc.org/api/v2/piston
supported_execution_strategies: sync
default_execution_strategy: sync
```

Isječak 4.2: Primjer ispisa dostupnih OCES instanci.

U nastavku ovog odjeljka slijedi pregled sučelja za pokretanje višekorisničkog opterećenja i odabir: dinamike višekorisničkog opterećenja, OCES-a, scenarija korištenja, i programske jezike.

4.2.1. Pokretanje višekorisničkog opterećenja

Aplikacijska naredba `run` ispisuje opcije (engl. *flags*) koje su dostupne prilikom pokretanja višekorisničkog opterećenja. Ispis naredbe `run` prikazan je isječkom 4.3 koji prikazuje da se pomoću naredbi `deterministic` i `stochastic` pokreće višekorisničko opterećenje odabrane dinamike.

```
$ ./helory run
Run benchmark

Usage:
  helory run [command]

Available Commands:
  deterministic  Run deterministic benchmark with parallel users
  stochastic     Run stochastic benchmark with parallel users

Flags:
  --config string          config file (default "./config.yaml")
  --endpoint string         online code execution endpoint (default "judge0_ce")
  --execution-strategy string code execution strategy (default "default")
  -h, --help                help for run
  --language string          programming language (default "cpp")
  --scenario string          benchmark scenario (default "hello_world")

Global Flags:
  --log-level string        (default "info")

Use "helory run [command] --help" for more information about a command.
```

Isječak 4.3: Dostupne opcije pri pokretanju višekorisničkog opterećenja.

4.2.2. Odabir dinamike višekorisničkog opterećenja

Eksperiment višekorisničkog opterećenja impulsno-determinističke dinamike, opisane u pododjeljku 3.1.1, pokreće se naredbom: `./helory run deterministic`. Odabir parametara N i M vrši se opcijom `--users` koja prihvata više formata za fleksibilan odabir parametara. Formati i njihova značenja koje opcija `--users` prihvata su sljedeći:

- A - odabire parametre $N = A$ i $M = 1$ (npr. `--users 5`),
- AxB - odabire parametre $N = A$ i $M = B$ (npr. `--users 5x10`),
- A₁, A₂, ..., A_i, ..., A_K - pokreće K opterećenja s parametrima $N_i = A_i$ i $M = 1$ (npr. `--users 10,5,7`),

- $A1, A2, \dots, Ai, \dots, AKxB$ - pokreće K opterećenja s parametrima $N_i = A_i$ i $M = B$ (npr. `--users 10,5,7x3`),
- $A1:A2:A3$ - pokreće opterećenja s parametrima $M = 1$ i parametrima $N_i = A_1 + i \cdot A_3$, $i \in \mathbb{N}_0$ uz ogragu $N_i \leq A_2$ (npr. `--users 5:30:2`), i
- $A1:A2:A3xB$ - pokreće opterećenja s parametrima $M = B$ i parametrima $N_i = A_1 + i \cdot A_3$, $i \in \mathbb{N}_0$ uz ogragu $N_i \leq A_2$ (npr. `--users 5:30:2x7`).

Eksperiment višekorisničkog opterećenja kontinuirano-determinističke dinamike, opisane u pododjeljku 3.1.2, pokreće se na isti način kao i eksperiment impulsno-determinističke dinamike, ali uz navođenje dodatne opcije `--no-wait`.

Eksperiment višekorisničkog opterećenja s kontinuirano-stohastičkom dinamikom pokreće se naredbom `./helory run stochastic`, gdje opcije `--intensity` i `--duration` služe za odabir parametara N i M . Dodatno, budući da ova vrsta dinamike koristi generator pseudo-slučajnih brojeva, opcijom `--seed` moguće je odabrati konstantu za inicijalizaciju generatora pseudo-slučajnih brojeva. Ako konstanta nije postavljena ovom opcijom, aplikacija će ju samostalno odabrati nasumičnim odabirom.

4.2.3. Odabir sustava za udaljeno izvršavanje programskog kôda

Navođenjem jedinstvenog identifikatora OCES-a, opcijom `--endpoint`, odabire se sustav za udaljeno izvršavanje programskog kôda za koji će biti pokrenuto višekorisničko opterećenje. Isječak 4.4 prikazuje primjer pokretanja višekorisničkog opterećenja impulsno-determinističke dinamike parametara $N = 5$ i $M = 10$ na OCES-u `judge0_ce`.

```
$ ./helory run deterministic --users 5x10 --endpoint judge0_ce
```

Isječak 4.4: Primjer pokretanja višekorisničkog opterećenja.

Dostupne OCES instance korisnik navodi u glavnoj konfiguracijskoj datoteci u YAML formatu. Ako se drugačije ne navede opcijom `--config`, aplikacija prepostavlja da je glavna konfiguracijska datoteka dostupna u tekućem direktoriju pod nazivom `config.yaml`. Glavna konfiguracijska datoteka također sadrži i podatke o putanji do konfiguracijskih direktorija scenarija i programske jezike koji su dostupni aplikaciji.

```

1 endpoints:
2   judge0_ce:
3     &judge0_ce
4     name: Judge0 CE
5     provider: judge0
6     url: https://ce.judge0.com
7     supported_execution_strategies: both
8     default_execution_strategy: async
9
10    judge0_extra_ce:
11      <<: *judge0_ce
12      name: Judge0 Extra CE
13      url: https://extra-ce.judge0.com
14
15    piston_public:
16      name: Piston (Free Public Instance)
17      provider: piston
18      url: https://emkc.org/api/v2/piston
19      supported_execution_strategies: sync
20
21    sphere_engine:
22      name: Sphere Engine
23      provider: sphere
24      url: https://5def13a.compilers.sphere-engine.com/api/v4
25      supported_execution_strategies: async
26
27 scenarios:
28   data_dir: ./scenarios_data
29
30 languages:
31   data_dir: ./languages_data

```

Isječak 4.5: Primjer sadržaja glavne konfiguracijske datoteke.

4.2.4. Odabir scenarija

Ako se drugačije ne navede zastavicom `--scenario`, prilikom pokretanja višekorisničkog opterećenja koristit će se jednostavan scenarij čiji je jedinstveni identifikator `hello_world`, dok su jedinstveni identifikatori ostalih scenarija: `cpu_intensive`, `network_intensive`, i `cpu_and_network_intensive`.

```
$ ./helory run deterministic --users 5x10 --endpoint judge0_ce --scenario cpu_intensive
```

Isječak 4.6: Primjer odabira scenarija procesorskog opterećenja.

4.2.5. Odabir jezika

Prilikom pokretanja višekorisničkog opterećenja opcijom `--language` moguće je odabrati programski jezik koji će aplikacija koristiti za odabrani scenarij. Programske jezike odabire se navođenjem njegovog jedinstvenog identifikatora koji je definiran u konfiguracijskom direktoriju za programske jezike dostupne aplikaciji. Primjer sadržaja konfiguracijskog direktorija programskih jezika prikazan je isječkom 4.7, dok je sadržaj konfiguracijske datoteke `cpp.yaml` za programske jezike C++ prikazan u isječku 4.8. Ime konfiguracijske datoteke programske jezike, bez njezinog nastavka (engl. *extension*), služi za definiranje jedinstvenog identifikatora jezika. Konfiguracijska datoteka programske jezike objedinjuje jedinstvene identifikatore programskih jezika koje koriste pojedini sustavi za udaljeno izvršavanje programske kôde.

```
./languages_data/
|---- cpp.yaml
|---- java.yaml
|---- python.yaml
```

Isječak 4.7: Primjer sadržaja konfiguracijskog direktorija programskih jezika.

```

1 name: C++
2 filename: main.cpp
3 providers:
4   judge0_ce:
5     language_id: 54
6   judge0_extra_ce:
7     language_id: 2
8   piston:
9     language_id: cpp
10    version: 10.2.0
11   sphere:
12     language_id: 1

```

Isječak 4.8: Primjer sadržaja konfiguracijske datoteke programskog jezika C++.

4.3. Pregled sučelja grafičkog izvještaja

Aplikacija Hélory nakon eksperimenta generira izvještaj u HTML formatu kojeg je moguće otvoriti u internet pregledniku. Ova cjelina daje pregled sučelja grafičkog izvještaja na primjeru izvještaja koji je nastao eksperimentom pokrenutim naredbom prikazanom u isječku 4.9.

```
$ ./helory run stochastic --intensity 5 --duration 10 --endpoint judge0_ce
```

Isječak 4.9: Pokretanje eksperimenta koji generira izvještaj opisan u ovom odjeljku.

Izvještaj se sastoji od četiri dijela. Prvi dio izvještaja prikazan je na slici 4.1 i sastoji se od tablice na lijevoj strani koja prikazuje sve važne detalje o pokrenutom eksperimentu. Tablica prikazuje da je eksperiment višekorisničkog opterećenja pokrenut koristeći kontinuirano-stohastičku dinamiku s parametrima $N = 5$ i $M = 10$ na OCES instanci Judge0 CE, i koristeći C++ implementaciju jednostavnog `hello_world` scenarija. Tablica također prikazuje da je korištena asinkrona interakcija između aplikacije i sustava. Eksperiment je trajao 14 sekundi, što znači da je navedeno opterećenje OCES instance Judge0 CE uspjela obraditi u tom vremenu. S desne strane iste slike prikazan je graf uspješnosti koji u ovom primjeru prikazuje da je OCES instance sve zahtjeve uspješno obradila. U donjem desnom kutu slike 4.1 prikazan je padajući izbornik (engl. *dropdown*) koji omogućuje odabir jedinice vremena koja će se koristiti kao korak vremenske osi na grafovima koji slijede.

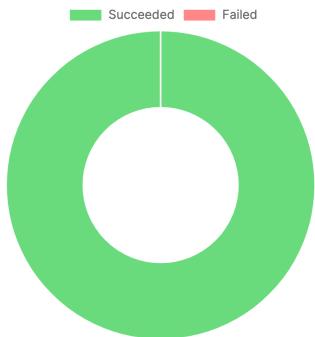


Hélory Benchmark Report

[Download as PDF](#)

Scenario	Hello World	
Description	Plain "hello, world" program.	
Language	C++	
Endpoint	Judge0 CE	
Strategy	async	
Type	stochastic	
Configuration	duration	10
	intensity	5
	seed	1624807317443342000
Started At	27/06/2021, 17:21:57	
Finished At	27/06/2021, 17:22:11	
Duration	00:00:14	

Success Rate



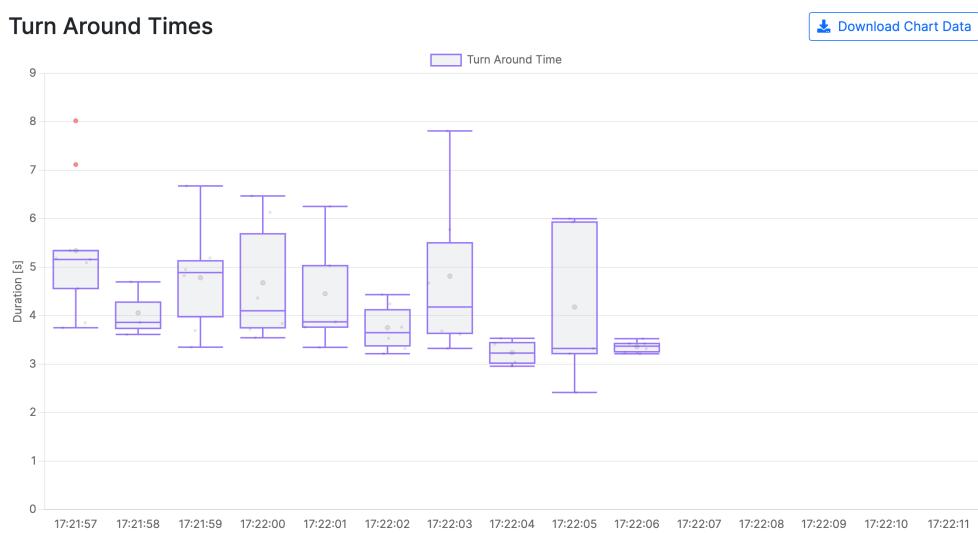
Interval 1s ▾

Slika 4.1: Prikaz detalja o pokrenutom eksperimentu.

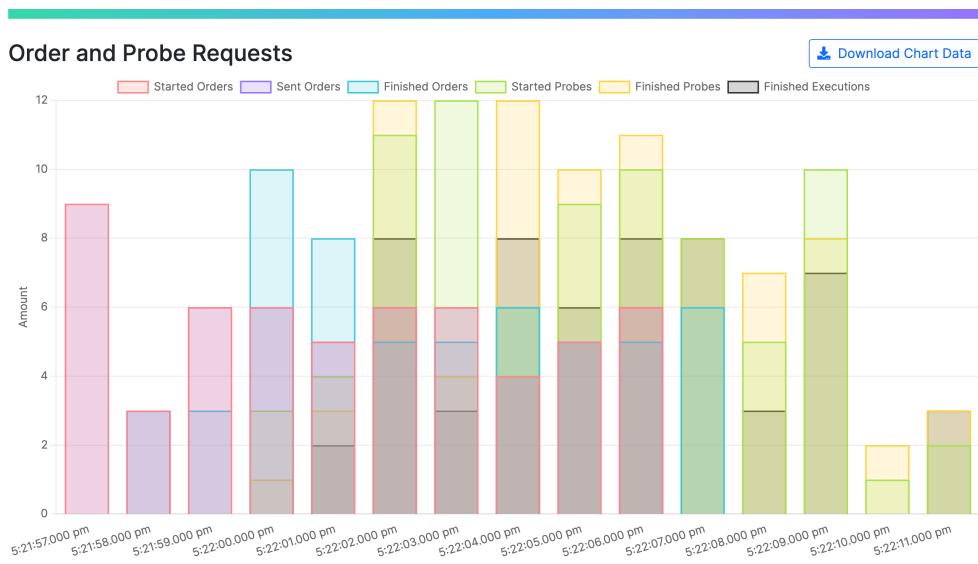
Slika 4.2 prikazuje drugi dio grafičkog sučelja izvješća koji prikazuje kutijaste dijagrame (engl. *box plots*) vremena obrade zahtjeva grupiranih prema vremenu završetka slanja podataka u zahtjevu narudžbe koristeći odabrani korak jedinice vremena. U ovom slučaju odabrani korak jedinice vremena je 1 sekunda, što znači da će u isti kutijasti dijagram biti grupirani svi zahtjevi koji su u istoj sekundi završili slanje podataka u zahtjevu narudžbe.

Sljedeći graf koji prikazuje izvještaj je graf prikazan na slici 4.3 koji prikazuje koliko je zahtjeva u istom vremenskom trenutku odabranog koraka:

- *Started Orders* - započeo zahtjev narudžbe,
- *Sent Orders* - poslao zahtjev narudžbe,
- *Finished Orders* - završio zahtjev narudžbe,
- *Started Probes* - započeo zahtjev ispitivanja,
- *Finished Probes* - završio zahtjev ispitivanja, i
- *Finished Executions* - završio zahtjev izvršavanja.



Slika 4.2: Prikaz dijagrama vremena obrade zahtjeva.



Slika 4.3: Prikaz grafa zahtjeva narudžbe i zahtjeva ispitivanja.

Konačno, slika 4.4 prikazuje sirove podatke u formatu JSON koji su prikupljeni za vrijeme trajanja eksperimenta, a na temelju kojih su napravljeni prethodno navedeni grafovi. Ove podatke moguće je preuzeti u formatu JSON i dodatno obraditi.

Raw Benchmark Data

[Download](#)

```
{  
    "id": "2021-06-27T17:21:57+02:00-stochastic-hello_world-cpp-judge0_ce-async-int_5-dur_10",  
    "name": "Hello World in C++ via Judge0 CE with stochastic behavior.",  
    "started_at": "2021-06-27T17:21:57.443642+02:00",  
    "finished_at": "2021-06-27T17:22:11.449519+02:00",  
    "scenario": "Hello World",  
    "scenario_description": "Plain \"hello, world\" program.",  
    "language": "C++",  
    "endpoint": "Judge0 CE",  
    "endpoint_url": "https://ce.judge0.com",  
    "strategy": "async",  
    "experiment_type": "stochastic",  
    "experiment_configuration": {  
        "duration": 10,  
        "intensity": 5,  
        "seed": 1624807317443342000  
    },  
    "executions": [  
        {  
            "success": true,  
            "started_at": "2021-06-27T17:21:57.444496+02:00",  
            "finished_at": "2021-06-27T17:22:01.194987+02:00",  
            "order_started_at": "2021-06-27T17:21:57.444718+02:00",  
            "order_finished_at": "2021-06-27T17:22:01.194987+02:00",  
            "order_index": 1  
        }  
    ]  
}
```

Generated by [Hélořy](#) on Sun Jun 27 2021

Slika 4.4: Prikaz sirovih podataka prikupljenih za vrijeme eksperimenta.

5. Primjer korištenja aplikacije Hélory

U ovom poglavlju prikazan je primjer korištenja aplikacije Hélory u analizi performansi sustava Piston i Judge0 koji su postavljeni na dva različita poslužitelja istih resursnih kapaciteta (2 CPU-a i 8 GB radne memorije) kupljenih preko platforme Digital Ocean. Cilj eksperimenta je ocijeniti kvalitetu i pouzdanost usluge koju nude ova dva sustava, međutim, treba uzeti u obzir da kvaliteta i pouzdanost usluge također ovisi i o resursima poslužitelja, ali i načinu na koji ih sustavi iskorištavaju. Također, važno je naglasiti da su sustavi na poslužitelje instalirani prema javno dostupnim uputama bez dodatnih modifikacija.

U analizi sustava promatrati će se jednostavan scenarij u C++ implementaciji koristeći kontinuirano-stohastičku dinamiku s parametrima $N \in \{1, 5, 10\}$ i $M = 60$. Budući da sustav Piston podržava isključivo sinkronu interakciju, u eksperimentima u kojima se koristi sustav Judge0 također će se koristiti sinkrona interakcija kako bi se ta dva sustava mogla pravedno usporediti.

Dodatno, na kraju je napravljena analiza performansi instance sustava Judge0 koja se koristi na Fakultetu elektrotehnike i računarstva Sveučilišta u Zagrebu.

Od nešto više od 500 eksperimentalnih rezultata, u ovom poglavlju napravljena je analiza nekoliko najznačajnijih rezultata za gore navedene instance.

5.1. Analiza performansi sustava Piston

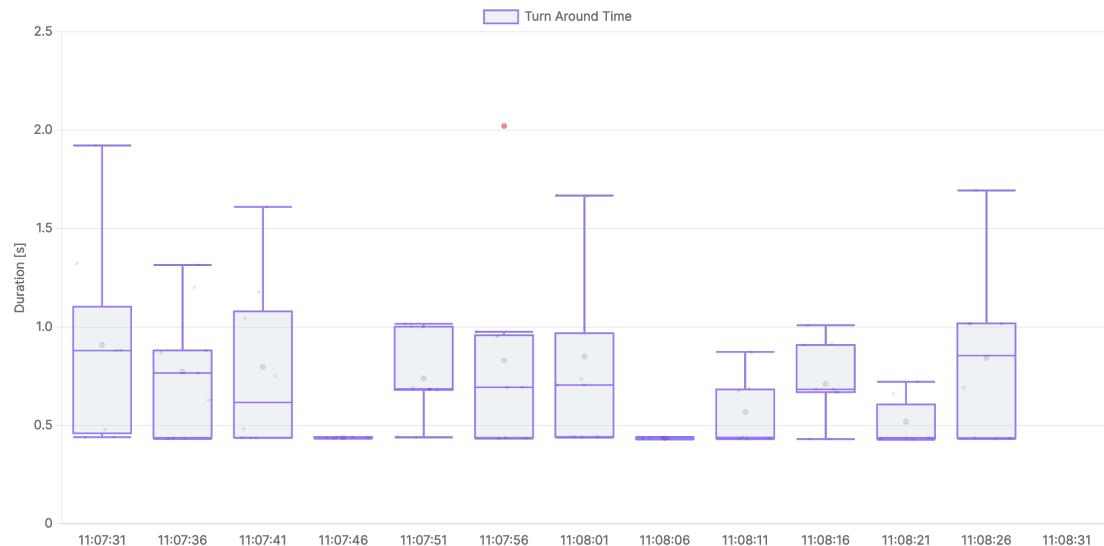
Prvi eksperiment višekorisničkog opterećenja kontinuirano-stohastičke dinamike s parametrima $N = 1$ i $M = 60$ trajao je 60 sekundi i svi su zahtjevi uspješno obrađeni. Slika 5.1 prikazuje vrijeme obrade zahtjeva grupiranih u intervalu po 5 sekundi. Medijan vrijeme obrade svake grupe zahtjeve je između 0,5 i 1 sekunde što znači da sustav dobro podnosi opterećenje.

Međutim, prilikom povećanja opterećenja na $N = 5$ udio uspješno obrađenih zahtjeva pada na 82,98%, a medijan vrijeme obrade zahtjeva raste i do 55 sekundi (slika 5.2). Sustav Piston još teže podnosi opterećenje za $N = 10$ kada udio uspješno obr

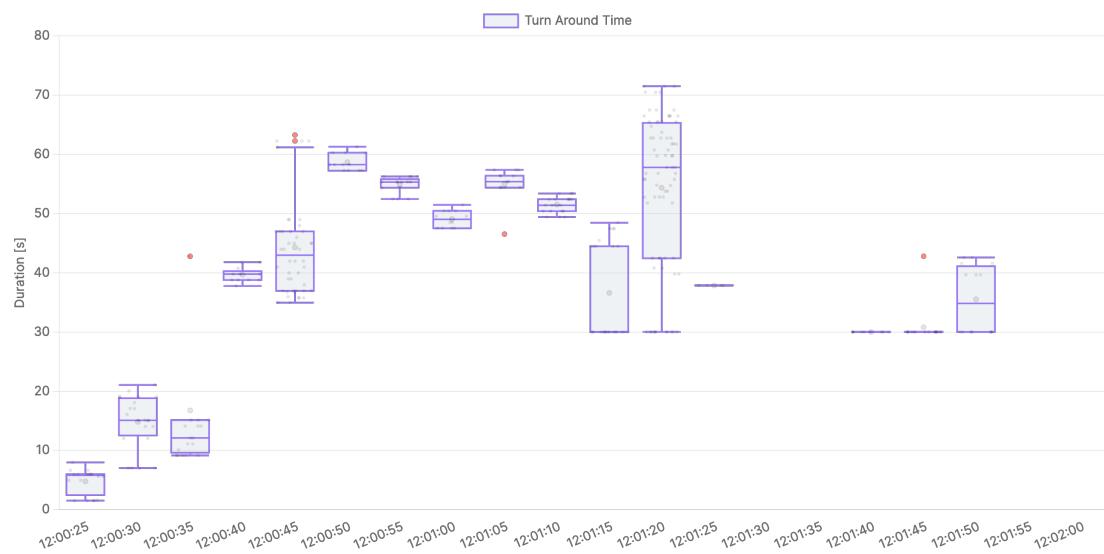
đenih zahtjeva dodatno padne na 42,44%.

Dodatno provedeni eksperimenti pokazuju da sustav Piston ne obradi uspješno niti jedan zahtjev za opterećenje $N = 1$ scenarija procesorskog opterećenja u C++ implementaciji.

Daljnja analiza performansi sustava Piston nije potrebna budući da za jednostavan scenarij korištenja ne prikazuje dobre rezultate niti po pitanju performansi, niti pouzdanosti, a rezultati pokazuju da sustav Piston podnosi maksimalno opterećenje od jednog zahtjeva u sekundi jednostavnog scenarija korištenja u C++ implementaciji.



Slika 5.1: Vrijeme obrade zahtjeva sustava Piston za $N = 1$ i $M = 60$.



Slika 5.2: Vrijeme obrade zahtjeva sustava Piston za $N = 5$ i $M = 60$.

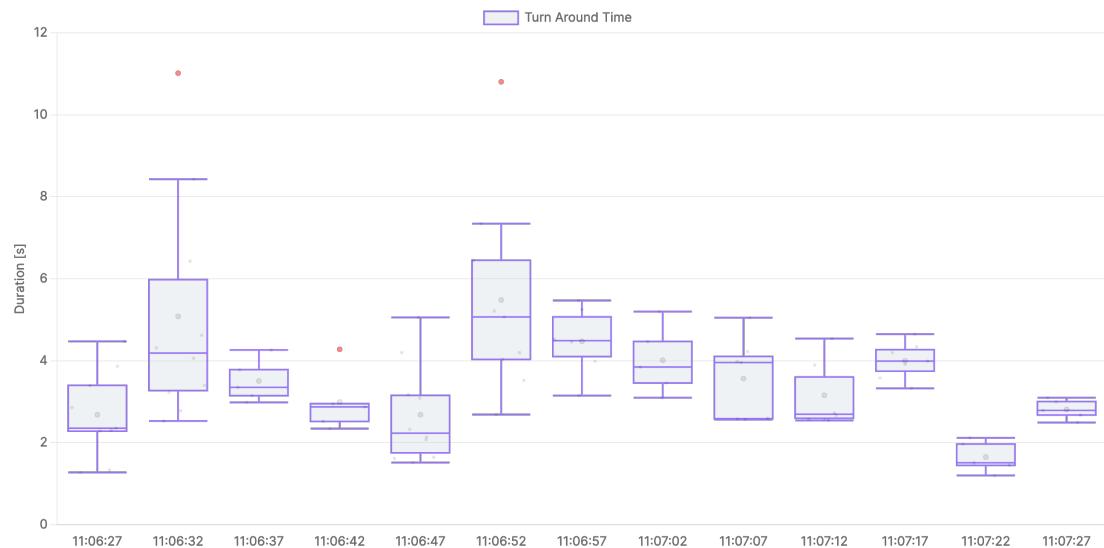
5.2. Analiza performansi sustava Judge0

Sustavu Judge0 pri opterećenju $N = 1$ treba 63 sekunde da uspješno obradi sve zahtjeve, međutim, medijan vrijeme obrade zahtjeva kreće se između 2 i 5 sekundi (slika 5.3) što je nepovoljnije u odnosu na sustav Piston.

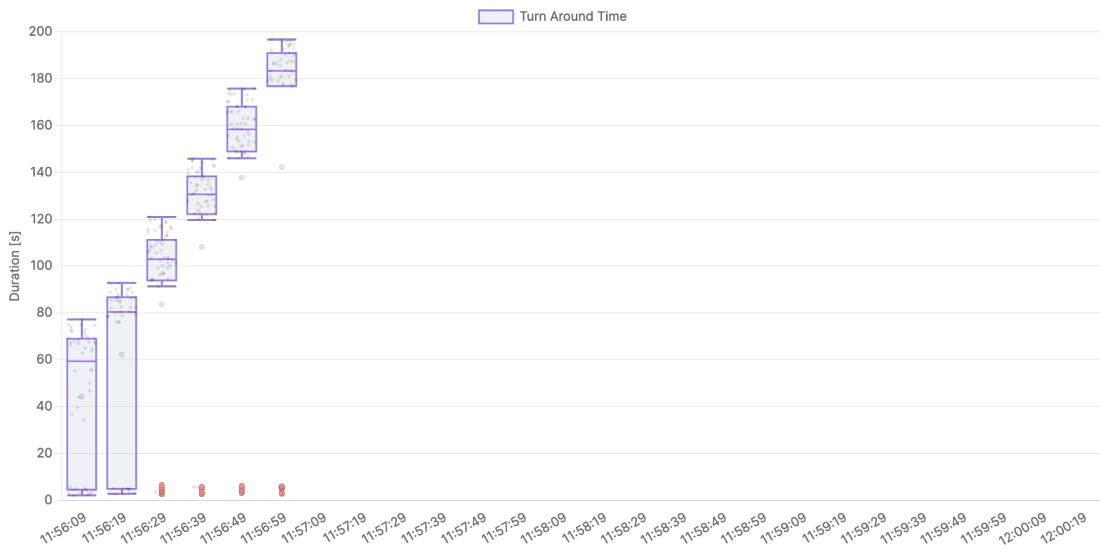
Pri opterećenju $N = 5$ Judge0 zadržava udio uspješno obrađenih zahtjeva, međutim, medijan vrijeme obrade zahtjeva linearno raste s vremenom trajanja eksperimenta kako prikazuje slika 5.4.

Dalnjim povećanjem opterećenja na $N = 10$ udio uspješno obrađenih zahtjeva pada na 81,79%, a medijan vrijeme obrade zahtjeva i dalje linearno raste. Međutim, dodatnim eksperimentima utvrđeno je da pri asinkronoj interakciji istog opterećenja udio uspješno obrađenih zahtjeva 100%, ali medijan vrijeme obrade zahtjeva opet linearno raste.

Na poslužitelju s istim resursnim kapacitetima, sustav Judge0 pokazuje bolju pouzdanost od sustava Piston uz uvjet da se koristi preporučena asinkrona interakcija, međutim, kao i sustav Piston, sustav Judge0 podnosi maksimalno opterećenje od jednog zahtjeva u sekundi jednostavnog scenarija korištenja u C++ implementaciji.



Slika 5.3: Vrijeme obrade zahtjeva sustava Judge0 za $N = 1$ i $M = 60$.



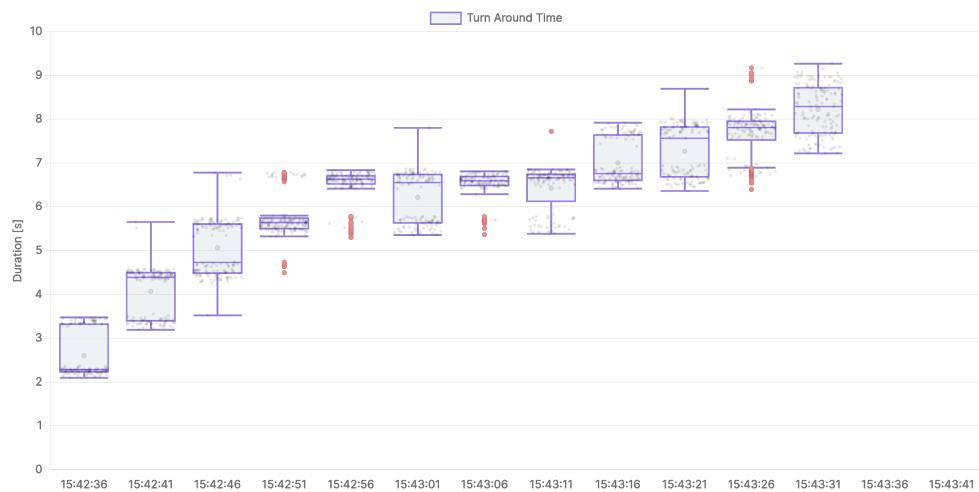
Slika 5.4: Vrijeme obrade zahtjeva sustava Judge0 za $N = 5$ i $M = 60$.

Analiza performansi FER-ove instance sustava Judge0

Instanca sustava Judge0 koja se koristi na Fakultet elektrotehnike i računarstva Sveučilišta u Zagrebu pokazuje zavidno stabilno medjan vrijeme obrade zahtjeva za kontinuirano-stohastičko opterećenje parametara $N = 5$ i $M = 60$ s asinkronom interakcijom za jednostavan scenarij korištenja u C++ implementaciji. Slika 5.5 prikazuje vrijeme obrade zahtjeva grupiranih u intervalu duljine 5 sekundi. Svi zahtjevi uspješno su obrađeni. Povećanjem intenziteta na $N = 20$ sustav zadržava stabilno medjan vrijeme obrade kao i u prethodnom slučaju, međutim, pri intenzitetu od $N = 25$ medjan vrijeme obrade počinje rasti s vremenom trajanja eksperimenta (slika 5.6).

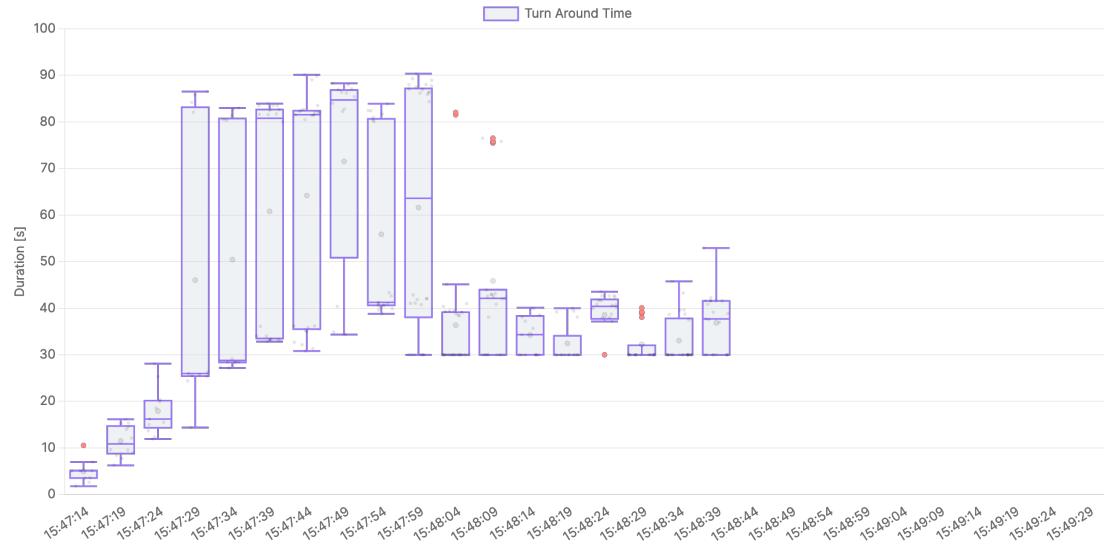


Slika 5.5: Vrijeme obrade zahtjeva FER-ove instance sustava Judge0 za $N = 5$ i $M = 60$ s asinkronom interakcijom.



Slika 5.6: Vrijeme obrade zahtjeva FER-ove instance sustava Judge0 za $N = 25$ i $M = 60$ s asinkronom interakcijom.

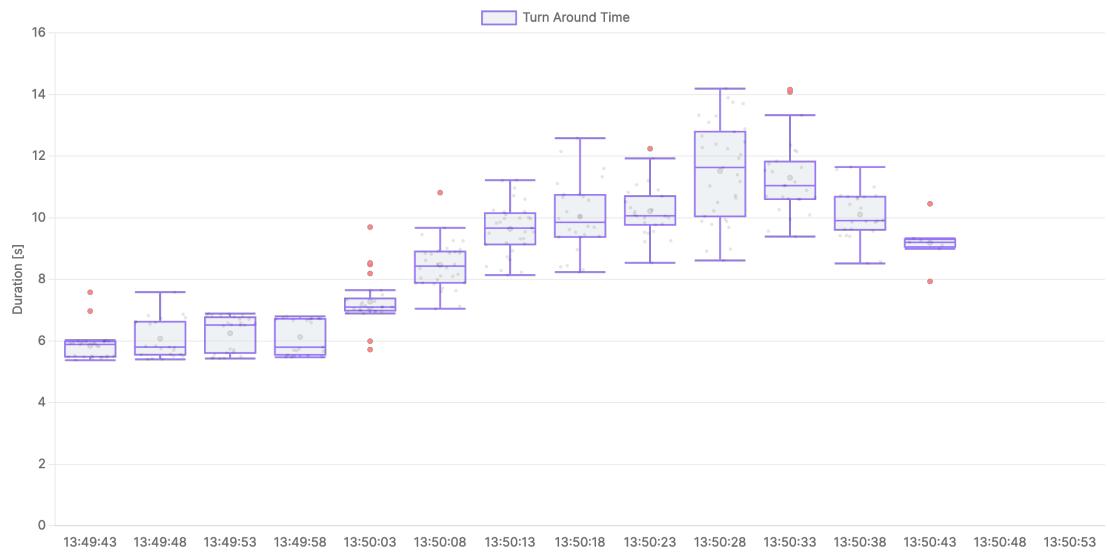
Dodatni eksperimenti napravljeni nad ovom instancom pokazuju kako je za sustav Judge0 uistinu nepovoljno koristiti sinkronu interakciju jer pri opterećenju od $N = 5$ sustav uspješno obradi svega 70.21% zahtjeva, dok medijan vrijeme obrade zahtjeva linearno raste s vremenom trajanja eksperimenta (slika 5.7). Službena, javno dostupna, instanca sustava Judge0 (Došilović, 2016a) nema problema sa rukovanjem ovog intenziteta opterećenja u sinkronoj interakciji. Njezino medijan vrijeme obrade zahtjeva ponaša se kao na slici 5.5, odnosno, kao i FER-ova instanca za asinkronu interakciju.



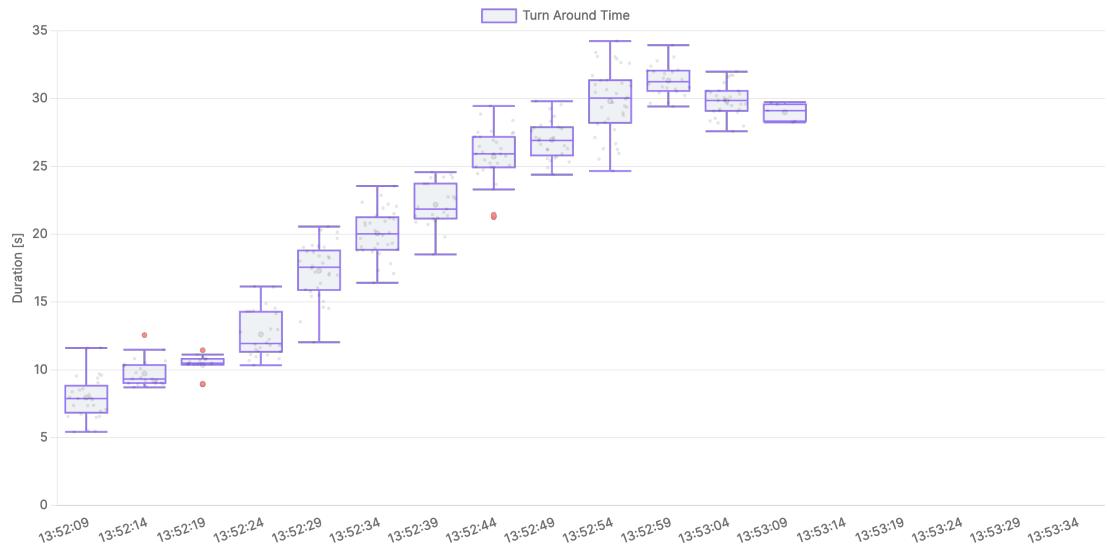
Slika 5.7: Vrijeme obrade zahtjeva FER-ove instance sustava Judge0 za $N = 5$ i $M = 60$ sa sinkronom interakcijom.

Eksperimenti pokazuju da FER-ova instanca sustava Judge0 podnosi maksimalno opterećenje od 20 zahtjeva u sekundi jednostavnog scenarija korištenja u C++ implementaciji, dok za programski jezik Java ista instanca podnosi maksimalno opterećenje od 10 zahtjeva u sekundi, a za programski jezik Python čak 25 zahtjeva u sekundi.

Za razliku od jednostavnog scenarija korištenja, kod scenarija procesorskog opterećenja u C++ implementaciji dolazi od rasta medijan vremena obrade već pri opterećenju od $N = 5$ (slika 5.8), i isto ponašanje uočava se i u Java i Python implementaciji (slika 5.9).



Slika 5.8: Vrijeme obrade zahtjeva FER-ove instance sustava Judge0 za $N = 5$ i $M = 60$ scenarija procesorskog opterećenja u C++ implementaciji.



Slika 5.9: Vrijeme obrade zahtjeva FER-ove instance sustava Judge0 za $N = 5$ i $M = 60$ scenarija procesorskog opterećenja u Java implementaciji.

6. Budući razvoj

Za budući razvoj radnog okvira analize performansi i ocjene kvalitete i pouzdanosti preporučuje se razmatranje dodatnih metrika poput npr. trajanja slanja zahtjeva narudžbe. Za budući razvoj aplikacije Hélory preporuča se razvoj nove funkcionalnosti u grafičkom izvješću koja će omogućiti objedinjavanje i usporedbu rezultata više eksperimenta. Također, preporuča se implementacija automatskog traženja maksimalnog opterećenja koje instanca podržava za odabrani scenarij korištenja i programski jezik. Za to je prije svega potrebno dobro definirati uvjet stabilne podrške nekog opterećenja. Dodatno, preporuča se detektiranje i implementacija dodatnih grafova koji se mogu dobiti koristeći sirove podatke prikupljene za vrijeme trajanja eksperimenta.

7. Zaključak

Sustav za udaljeno izvršavanje programskog kôda služe za sigurno i pouzdano izvršavanje korisničkog programskog kôda u sustavima za udaljeno ocjenjivanje koji se koriste u raznim slučajevima primjene među kojima su i *web* aplikacije za e-učenje. U arhitekturi ekosustava sustava za udaljeno ocjenjivanje, sustavi za udaljeno izvršavanje programskog kôda promatraju se kao zasebne komponente u literaturi tek od (Došilović i Mekterović, 2020). Budući da sustavi za udaljeno izvršavanje programskog kôda imaju ključni utjecaj na korisničko iskustvo, važno je da prilikom intenzivnog višekorisničkog opterećenja korisničko iskustvo ne pati zbog dugog vremena obrade zahtjeva za izvršavanje programskog kôda. Dosada u kontekstu sustava za udaljeno izvršavanje programskog kôda nisu postavljeni radni okviri za analizu njihovih performansi i ocjenu njihove kvalitete i pouzdanosti.

Ovaj rad predstavlja prvi radni okvir za analizu performansi i ocjenu kvalitete i pouzdanosti usluge koju nude sustavi za udaljeno izvršavanje programskog kôda. Ovaj rad također predstavlja i aplikaciju Hélory koja implementira predstavljeni radni okvir za analizu performansi i ocjenu kvalitete i pouzdanosti za tri sustava za udaljeno izvršavanje programskog kôda: Sphere Engine, Piston i Judge0. Razvijen u obliku komandno-linijske aplikacije, Hélory nudi jednostavno pokretanje višekorisničkog opterećenja na željenom sustavu, a nakon eksperimenta Hélory će generirati i pohraniti izvještaj o pokrenutom eksperimentu koji sadrži detaljne informacije o svakom pojedinom zahtjevu za izvršavanje i grafički prikaz metrika od interesa.

Koristeći aplikaciju Hélory eksperimentalno je dokazano da se pri korištenju sustava Judge0 preporuča koristiti asinkronu interakciju sa sustavom. Također provedeni eksperimenti nad instancom sustava Judge0 koja se koristi na Fakultetu elektrotehnike i računarstva Sveučilišta u Zagrebu pokazuju da instance podnosi maksimalno opterećenje od 10 zahtjeva u sekundi jednostavnog scenarija korištenja u C++ implementaciji, dok za programske jezike Java i Python ista instance podnosi maksimalno opterećenje od 5 zahtjeva u sekundi, a za programske jezike C i C++ čak 20 zahtjeva u sekundi.

LITERATURA

Anja Balanskat i Katja Engelhardt. *Computing our Future*. European Schoolnet, 10 2014. doi: 10.13140/RG.2.1.5029.9048.

Anja Balanskat i Katja Engelhardt. *Computing our future: Computer programming and coding - Priorities, school curricula and initiatives across Europe*. European Schoolnet, 10 2015.

Paul Bilodeau. Filtered - The future of hiring is skill-based, 2016. URL <https://www.filtered.ai>. Pristupano: 17.06.2021.

Hercy Chang i Winston Tang. LeetCode - The World's Leading Online Programming Learning Platform, 2015. URL <https://leetcode.com>. Pristupano: 17.06.2021.

Alan AA Donovan i Brian W Kernighan. *The Go programming language*. Addison-Wesley Professional, 2015.

Herman Zvonimir Došilović. Judge0 CE - API Docs, 2016a. URL <https://ce.judge0.com>. Pristupano: 28.06.2021.

Herman Zvonimir Došilović. Judge0 IDE - Free and open-source online code editor, 2016b. URL <https://ide.judge0.com>. Pristupano: 17.06.2021.

Herman Zvonimir Došilović. Judge0 - Where code happens., 2020. URL <https://judge0.com>. Pristupano: 25.06.2021.

Herman Zvonimir Došilović i Igor Mekterović. Robust and Scalable Online Code Execution System. U *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)*, stranice 1627–1632, 2020. doi: 10.23919/MIPRO48935.2020.9245310.

Cheedoong Drung, Jianwen Wang, i Ning Guo. Enhance performance of program automatic online judging systems using affinity algorithm and queuing theory in

SMP environment. U *Proceedings of 2011 International Conference on Electronic & Mechanical Engineering and Information Technology*, svezak 9, stranice 4425–4428. IEEE, 2011.

Shuchi Grover i Roy Pea. Computational thinking in K–12: A review of the state of the field. *Educational researcher*, 42(1):38–43, 2013.

Andy Kurnia, Andrew Lim, i Brenda Cheang. Online judge. *Computers & Education*, 36(4):299–315, 2001.

James Lockwood i Aidan Mooney. Computational thinking in education: Where does it fit? A systematic literary review. *arXiv preprint arXiv:1703.07659*, 2017.

Stefano Maggiolo i Giovanni Mascellani. CMS :: Testimonials, 2012a. URL <http://cms-dev.github.io/testimonials.html>. Pristupano: 26.06.2021.

Stefano Maggiolo i Giovanni Mascellani. Introducing CMS: A Contest Management System. *Olympiads in Informatics*, 6, 2012b.

Jemar Jude A Maranga, Leilla Keith J Matugas, Jorge Frederick W Lim, i Cherry Lyn C Romana. CodeChum: An Online Learning and Monitoring Platform for C Programming. *International Association for Development of the Information Society*, 2019.

Martin Mareš i Bernard Blackham. A New Contest Sandbox. *Olympiads in Informatics*, 6, 2012.

Igor Mekterović, Ljiljana Brkić, Boris Milašinović, i Mirta Baranović. Building a comprehensive automated programming assessment system. *IEEE Access*, 8:81154–81172, 2020.

Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

Clément Mihailescu i Antoine Pourchet. AlgoExpert | Ace the Coding Interviews, 2017. URL <https://www.algoexpert.io>. Pristupano: 17.06.2021.

Mikhail Mirzayanov. Codeforces, 2009. URL <https://codeforces.com>. Pristupano: 16.06.2021.

MIT Media Lab. Professor Emeritus Seymour Papert, pioneer of constructionist learning, dies at 88 | MIT News | Massachusetts Institute of Technology, 2016. URL <https://news.mit.edu/2016/seymour-papert-pioneer-of-constructionist-learning-dies-0801>. Pristupano: 23.06.2021.

Srinivas Nidhra i Jagruthi Dondeti. Black box and white box testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA)*, 2(2):29–50, 2012.

Seymour Papert. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, Inc., USA, 1980. ISBN 0465046274.

Vivek Ravisankar i Hari Karunanidhi. HackerRank, 2012. URL <https://www.hackerrank.com>. Pristupano: 17.06.2021.

Brian Seymour. A high performance general purpose code execution engine., 2018. URL <https://github.com/engineer-man/piston>. Pristupano: 20.06.2021.

Sphere Research Labs Sp. z o.o. Sphere Online Judge (SPOJ), 2004. URL <https://www.spoj.com>. Pristupano: 16.06.2021.

Sphere Research Labs Sp. z o.o. Coding skills assessment and code execution APIs - Sphere Engine, 2008. URL <https://sphere-engine.com>. Pristupano: 25.06.2021.

Bhavin Turakhia. Competitive Programming | Participate & Learn | CodeChef, 2009. URL <https://www.codechef.com>. Pristupano: 16.06.2021.

Miao Wang, Wentao Han, i Wenguang Chen. MetaOJ: A massive distributed online judge system. *Tsinghua Science and Technology*, 26(4):548–557, 2021.

Szymon Wasik, Maciej Antczak, Jan Badura, Artur Laskowski, i Tomasz Sternal. A survey on online judge systems and their applications. *ACM Computing Surveys (CSUR)*, 51(1):1–34, 2018.

Jeannette M Wing. Computational thinking. *Communications of the ACM*, 49(3): 33–35, 2006.

Chao Yi, Su Feng, i Zhi Gong. A comparison of sandbox technologies used in online judge systems. U *Applied Mechanics and Materials*, svezak 490, stranice 1201–1204. Trans Tech Publ, 2014.

Jacob Zhang. AlgoDaily - Software interview prep made easy. Coding Interview Questions., 2018. URL <https://algodaily.com>. Pristupano: 17.06.2021.

Željko Švedić i Mario Živić. Programming and Interview Online Assessment Tests | TestDome, 2013. URL <https://www.testdome.com>. Pristupano: 17.06.2021.

Analiza performansi sustava za udaljeno izvršavanje programskog kôda

Sažetak

Sustavi za udaljeno izvršavanje programskog kôda imaju ključni utjecaj na korisničko iskustvo, i važno je da prilikom intenzivnog višekorisničkog opterećenja korisničko iskustvo ne pati zbog dugog vremena obrade zahtjeva za izvršavanje programskog kôda. Dosada u kontekstu sustava za udaljeno izvršavanje programskog kôda nisu postavljeni radni okviri za analizu njihovih performansi i ocjenu njihove kvalitete i pouzdanosti. Ovaj rad predstavlja prvi radni okvir za analizu performansi i ocjenu kvalitete i pouzdanosti usluge koju nude sustavi za udaljeno izvršavanje programskog kôda. Ovaj rad također predstavlja i aplikaciju Hélory koja implementira predstavljeni radni okvir za analizu performansi i ocjenu kvalitete i pouzdanosti za tri sustava za udaljeno izvršavanje programskog kôda: Sphere Engine, Piston i Judge0. Nakon pregleda osnovnih funkcionalnosti i sučelja aplikacije Hélory dan je primjer korištenja u analizi performansi sustava Piston i Judge0, ali i analiza performansi instance sustava Judge0 koja se aktivno koristi na Fakultetu elektrotehnike i računarstva Sveučilišta u Zagrebu. Konačno, na kraju se donose smjernice za budući razvoj radnog okvira i aplikacije.

Ključne riječi: analiza performansi, sustav za udaljeno izvršavanje programskog kôda, sustav za udaljeno ocjenjivanje, izvršavanje nepouzdanog programskog kôda, sphere engine, piston, judge0

Performance analysis of online code execution systems

Abstract

Online code execution systems have a crucial impact on the user experience, and it is essential that during intense multi-user workloads, the user experience does not suffer due to the long processing time of code execution requests. So far, in the context of online code execution systems, no framework has been made to analyze their performance and assess their quality and reliability. This paper presents the first framework for analyzing the performance and evaluating the quality and reliability of the service offered by online code execution systems. This paper also presents the Hélory application that implements the presented framework for performance analysis and quality and reliability assessment for three systems for online code execution: Sphere Engine, Piston, and Judge0. After reviewing the basic functionalities and interface of the Hélory application, an example of use in the performance analysis of the Piston and Judge0 systems is given, as well as the performance analysis of the Judge0 system instance actively used at the Faculty of Electrical Engineering and Computing, University of Zagreb. Finally, guidelines for the future development of the framework and application are given.

Keywords: performance analysis, online code execution system, online judge system, untrusted code execution, sphere engine, piston, judge0