Univerzitet u Nišu

Elektronski fakultet

Ilija Milosavljević

Simulacija računarskih resursa u Python-u

Završni rad

Studijski program: Elektrotehnika i računarstvo

Modul: Računarstvo i informatika

Kandidat:

Ilija Milosavljević, broj indeksa:16210

Mentor:

Prof. dr. Ivan Milentijević

Niš, Jun 2021. godine

Završni rad:

Simulacija računarskih resursa u Python-u

Simulation of computer resources using Python

Zadatak:

Proučiti principe modelovanja sistema za razvoj simulatora diskretnih događaja, kao i biblioteku SimPy. Projektovati simulacioni model sistema za dodelu računarskih resursa studentima kojie bi oni koristili u procesu učenja. Realizovati aplikaciju za izvođenje simulacionih eksperimenata i beleženje rezultata simulacije. Ispitati različite politike pripremanja računarskih resursa i predložiti najefikasniju politiku u pogledu količine angažovanih resursa i vremena čekanja na resurs.

Komisija za odbranu:

1. _____ Datum prijave: _____

2. _____ Datum prijave: _____

3. _____ Datum prijave: _____

Sadržaj

1. Uvod

Primena novih tehnologija donela je sa sobom brojne benefite i unapređenja u raznovrsnim spektima ljudskog društva. Mnogi sistemi i organizacije su, nošeni ovim razvojem značajno unapredili svoj rad. Međutim, ono što često biva zanemareno jesu mogućnosti koje nam računar, kao centralna komponenta, pruža ukoliko se kombinuje sa postojećim eksperimentalnim i teorijskim znanjima.

Simulacije diskretnih događaja predstavljaju jedan ovakav spoj računarske tehnologije i znanja sakupljenog brojim istraživanjima iz oblasti statistike i psihologije ljudskog ponašanja. Neminovno je da su računari drastično povećali obim podataka koji je moguće obraditi. Ova činjenica predstavlja prekretnicu u načinu testiranja sistema i njegovog odgovora na promene. Komjuteri su omogućili kreiranje kompleksnih modela koji će simulirati komplikovane i nepredvidive sisteme. Kombinovanjem naizgled nespojivih grana nauke pruža se potpuno novi uvid u način njihovog rada i diskretne veze između njegovih komponenti.

Ovaj rad upravo prikazuje razvoj jedne takve aplikacije, koja za cilj utvrđivanje optimalne konfiguracije sistema za dodelu računarskih resursa. Analiza realnog sistema predstavlja komplikovan i, ono što je možda i važnije, skup proces, zbog čega on velikom broju organizacija nije validna opcija. U ovakvim slučajevima simulacija predstavlja prirodno rešenje. Konkretno, rad se bavi izradom aplikacije koja bi fakultetu omogućila da testira i utvrdi parametre sistema za dodelu resursa studentima. Korišćenjem simulacije kao alata, bavićemo se proverom različitih konfiguracija sistema, pri čemu krajnji cilj predstavlja pronalaženje podešavanja koje bi dalo najbolje performanse, a sve ovo uz brzinu i sigurnost koju simulacija pruža.

1.1. Šta je simulacija i zašto je koristiti?

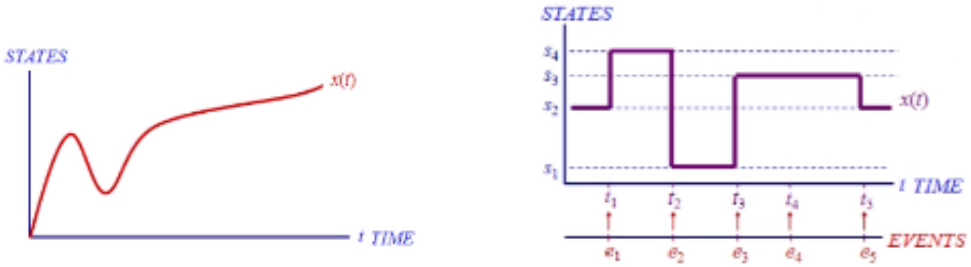
Definicija: Simulacija diskretnih događaja predstavlja seriju tehnika, koje kada se primene na dinamički sistem diskretnih događaja, generišu sekvencu jednostavnih putanja koje karakterišu njegovo ponašanje.

Modelovanje kompleksnih sistema je postalo nezamenljivo u mnogim granama nauke, kao što su matematika, inženjerstvo, vojska, zdravstvo, socijalne, transportne ili telekomunikacione nauke. Eksplozija u popularnosti je došla pre svega zbog toga što simulacija predstavlja relativno jeftin način sakupljanja informacija koje mogu pomoći prilikom izgradnje realnog sistema. Obzirom da veličina i kompleksnost navedenih sistema retko dozvoljava korišćenje analitičkih metoda koje bi pružile takve informacije, simulacija diskretnih događaja često postaje jedini izbor. Primenom simulacije pruža se mogućnost testiranja slučajeva koji se u realnim sistemima dešavaju veoma retko, i na taj način omogućavaju testiranje različitih scenarija koji se mogu javiti kao odgovor na njih. Na taj način pruža se mogućnost testiranja i boljeg razumevanja alternativnih pristupa u rešavanju problema. Primena simulacije omogućava da se lako ispita uticaj različitih parametara na ponašanje sistema kao celine i praćenje promena koje izmena nekog atributa može uzrokovati u ostatku sistema.

Uprkos svemu navedenom, neophodno je skrenuti pažnju na činjenicu da merodavnost rezultata simulacije u velikoj meri zavisi od njene valjanosti. Tačnije, ukoliko generisane vrednosti i veze komponenti unutar simulacije ne odgovaraju onim u realnom sistemu, jasno je da će se izlazni rezultati odstupati od onih primećenih u stvarnom sistemu. Stoga je potrebno poštovati određene korake prilikom njenog kreiranja, i upravo iz tog razloga u nastavku ovog rada oni su detaljno opisani. Ono što je sada potrebno istaći jeste da, iako ona sa sobom donosi brojne pogodnosti, simulacija ne daje nužno ispravne podatke, ukoliko se ne implementira na adekvatan način.

1.2. Diskretni sistemi

Simulacija diskretnih događaja je metod za simuliranje ponašanja i praćenje performansi realnih sistema ili procesa. Simulacija predstavlja aproksimaciju sistema iz svakodnevnog života. Važno je naglasiti da su realni sistemi, po svojoj prirodi, kontinualni. Pod tim podrazumevamo sistem kod koga se promene stanja dešavaju u kontinuitetu, u toku vremena. Nasuprot njima, kod diskretnih sistema promene se dešavaju u diskretnim vremenskim intervalima (Slika 1).



Slika 1: Kontinualne i diskretne vrednosti

Simulacija diskretnih događaja analizira dinamički sistem aproksimiranjem njegovog ponašanja kao sekvencu trenutnih događaja. Aproksimacijom kontinualnih događaja iz simulacije se izbacuju svi oni aspekti realnog sistema koji nisu od interesa za predmet ispitivanja.

2. Modelovanje sistema

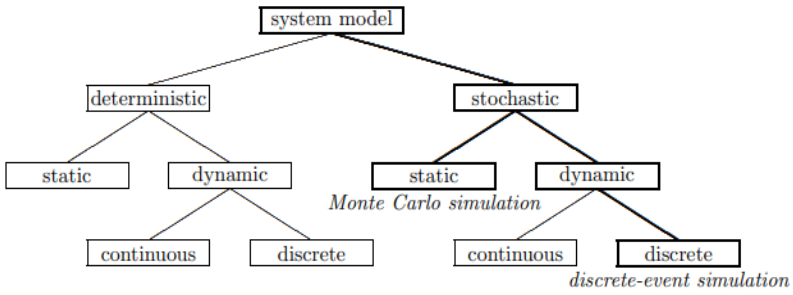
Uspeh simulacije i merodavnost sakupljenih podataka najviše zavisi od modela koji se koristi prilikom simulacije, i njegove sposobnosti da na pravilan način aproksimira ponašanje realnog sistema. Kao što je statističar George E. P. Box rekao „svi modeli su pogrešni, ali su neki korisni“. Razumevanje problema koji se ispituje i cilja same simulacije direktno utiče na to kako će se sistem aproksimirati i utiče na model sistema koji će se koristiti.

Uzmimo primer simulacije leta aviona, u slučaju da želimo da simuliramo let jedne letelice i pratimo parametre i ponašanje letelice tokom leta korisno je znati u kom se tačno stanju nalazi letelica u svakom trenutku i pratiti vrednosti kao što su visina, brzina, broj putnika i td. Međutim u slučaju da se ispituje ponašanje svih aviona u jednom regionu i uticaj vremenskih prilika na letova, gore navedene vrednosti postaju suvišne.

2.1 Karakteristike modela

Model sistema može biti deterministički ili stohastički (Slika 2). Pod determinističkim modelom sistema podrazumevamo onaj koji nema nasumičnu (stohastičku) komponentu. Iako pojedini sistemi na prvi pogled mogu delovati kao dinamički, kao na primer rad mašine, to najčešće nije slučaj. Makar na određenom nivou, svi sistemi poseduju određenu stohastičku komponentu; kvar mašine, zahtevi koje se dešavaju u nasumičnim vremenskim intervalima i td. Upravo zbog svega navedenog modeli koji se koriste u simulacijama diskretnih događaja poseduju mogućnost ubacivanja stohastičke komponente, bez dramatičnog povećavanja kompleksnosti modela i izračunavanja potrebnih za rad simulacije.

Druga podela modela je na statičke i dinamičke. Statički modeli su oni kod kojih vreme ne predstavlja značajnu komponentu. Kod simulacija koje zahtevaju takav model najčešće je potrebno odrediti verovatnoću neke pojave ili događaja, bez obzira na vremenski trenutak, kao u primeru određivanja verovatnoća dobitaka na lutriji u jednom kolu. Nasuprot njih su dinamički sistemi kod kojih se parametri sistema menjaju u toku vremena. Dinamički sistemi mogu biti kontinualni ili diskretni. Većina tradicionalnih dinamičkih sistema koji se koriste prilikom analitičkih izračunavanja su kontinualni. Sa druge strane, dinamički sistemi, kojima se bavimo u ovom radu, pripadaju krupi diskretnih sistema, kod kojih je vremenska osa podeljena na konstantne diskretne delove.



Slika 2: Podela modela sistema

Model koji se koristi u simulacija diskretnih događaja je ujedno i stohastički (nasumičan) i dinamički, sa dodatnom osobinom da se stanje varijabli koje opisuju sistem menja u diskretnim vremenskim intervalima.

Zaključak: Model sistema diskretnih događaja je:

- *Stohastički* - makar neki od parametara sistema su nasumični
- *Dinamički* - promena varijabli u toku vremena je od velikog značaja za sistem
- *Diskretan* - promene parametara sistema su povezane sa događajima koji se dešavaju u diskretnim vremenskim intervalima

2.2 Razvoj modela

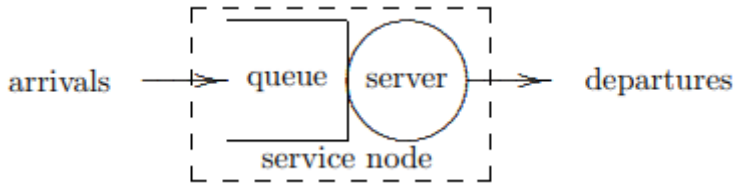
Proces razvoja modela koji će se koristiti u simulaciji diskretnih događaja može se podeliti u određen broj koraka, s tim što se neki koraci (od 2 do 6) mogu ponavljati više puta kako bi se došlo do modela koji će na validan način predstaviti realan sistem [1].

1. Odrediti *ciljeve* i *parametre* koji su predmet analize simulacije. Jasno definisani ciljevi daju smisao preostalim koracima u izradi modela. Ciljevi mogu biti jednostavne *boolean* odluke (da li povećati broj kasa u prodavnici) ili *numeričke* odluke (koliko servera je potrebno da bi se opslužili svi zahtevi korisnika).
2. Na osnovu prvog koraka izrađuje se *konceptualni model,* koji bi trebalo da, ukoliko je pravilno izrađen, da odgovore na pitanja navedena u nastavku. Koji parametri definišu stanje sistema, kako su oni međusobno povezani i u kolikoj meri ih je moguće menjati. Koliko sveobuhvatan treba da bude model? Koji parametri sistema su bitni za predmet i cilj analize, a efekat kojih je zanemarljiv? Ovaj korak najčešće zahteva analiziranje sistema i njegovo dobro poznavanje i ne sme biti zanemaren ako želimo da model koji na adekvatan način opisuje realan sistem.
3. Konvertovanje konceptualnog u *specifikacioni* model. Ukoliko se uradi na adekvatan način ovaj korak značajno olakšava sve naredne. Karakteristično za ovaj korak je to da on obuhvata prikupljanje i statističku analizu podataka koji će predstavljati ulazni skup podataka za model i omogućiti rad simulacije. U nedostatku navedenih podataka moguće je koristiti stohastičke metode koje će pružiti reprezentativan ulazni skup nasumično generisanih podataka.
4. Izrada *računarskog modela* na osnovu specifikacionog, tačnije izrada kompjuterskog programa. U ovom koraku je potrebno doneti odluku koji će se programski jezik koristiti za izradu simulacije.
5. *Verifikacija.* Kao i kod svih kompjuterskih programa, računarski model treba da bude konzistentan sa specifikacionim modelom. Potrebno je utvrditi da li smo na adekvatan način implementirali model.
6. *Validacija.* Da li je kompjuterski model konzistentan sa sistemom koji je predmet analize - da li smo izgradili model koji odgovara realnom sistemu. Imajući u vidu sklonost ljudi da slepo veruje kompjuterskim programima (i rezultatima simulacije) od fundamentalne je važnosti utvrditi merodavnost izlaznih podataka. Jedna od najčešće korišćenih metoda za validaciju je upoređivanje izlaznih podataka koje je dala simulacija diskretnih događaja sa stvarnim podacima sakupljenim analizom realnog sistema sa istim ili sličnim početnim parametrima. Analizu ovih podatak obavlja stručno lice upoznato sa radom realnog sistema. Validnost modela se potvrđuje ukoliko lice koje vrši proveru podataka nisu u mogućnosti da utvrdi koji od izlaznih skupova podatka predstavlja generisan a koji onaj sakupljen analizom realnog sistema.

2.3. Konceptualni model

Osnovu svakog modela sistema čini nekoliko osnovnih komponenti (Slika 3):

- Server koji obrađuje zahteve(događaje)
- Red čekanja u koji se smeštaju zadaci
- Dolasci (zadaci, korisnici)



Slika 3: Simulacija diskretnih događaja – servisni čvor

Zahtevi (događaji, korisnici) dolaze do servisa (servisnog čvora ili mreže servisnih čvorova) u predefinisanim vremenskim trenucima i zahtevaju izvršenje određenog zadatka. Vreme potrebno da se zadatak izvrši se nasumično određuje, nakon toga zadatak je izvršen i prelazi se na izvršenje narednog. Ukoliko je servis zauzet u trenutku kada zadatak stigne, on se smešta u red čekanja. Nakon što završi izvršavanje jednog zadatka servisni čvor proverava red čekanja i ukoliko on nije prazan prelazi na izvršenje sledećeg zadatka. Način rada reda čekanja zavisi od tipa (FIFO, LIFO, SIRO, Prioritet), ali se u najvećem broju slučajeva koristi FIFO red. Najveći broj zadataka koje jedan servis može imati na čekanju zavisi od kapaciteta reda. Kapacitet može biti ograničen ili neograničen, u slučaju ograničenog, nakon popunjavanja reda čekanja svi naredni zadaci se odbacuju sve dok se on ne isprazni.

2.4. Specifikacioni model

Nezavisno od toga da li se koristi skup podataka sakupljenih analiziranjem sistema ili se za njegovo generisanje koriste stohastičke metode, u najvećem broju slučajeva za svaki korisnički zahtev, u nastavku zadatak, moguće je definisati određen skup parametara:

- Vreme dolaska korisnika (zadatka)
- Vreme odlaganja početka izvršenja i čekanje u redu
- Trenutak kada zadatak počinje sa izvršenjem
- Vreme potrebno za izvršenje zadatka
- Vreme koje zadatak provede u servisnom čvoru (zbir vremena čekanja i izvršenja)
- Trenutak kada se završi izvršenje zadatka

Tokom izrade konceptualnog modela, potrebno je analizom sistema doći do ulaznih podataka koji su potrebni za početak simulacije. Međutim ukoliko je njih nemoguće sakupiti, kao što je to bio slučaj prilikom izrade simulacije o kojoj je reč u ovom radu, moguće je korišćenje alternativnih metoda, konkretno, podatke je moguće generisati pomoću stohastičkih (nasumičnih) metoda. Pored toga, čak i u slučaju sistema kod koga skup ulaznih podataka dostupan, pojedini parametri sistema su uvek, po svojoj prirodi nasumični. Imajući u vidu sve navedeno lako je zaključiti da je gotovo nemoguće uspešno realizovati simulaciju diskretnih događaja bez dobrog poznavanja načina za generisanje nasumičnih komponenti modela. U sledećim poglavljima ćemo se detaljnije upoznati sa matematičkim i računarskim alatima potrebnim za generisanje takvih stohastičkih komponenti sistema.

2.4.1. Slučajne promenljive

Ako posmatramo skup elementarnih događaja Ω, i ako se elementarni događaji ω mogu predstaviti kao realni brojevi, onda se eksperiment može zamisliti kao izbor jedne promenljive. Promenljiva veličina koja te brojne vrednosti uzima sa određenim verovatnoćama naziva se *slučajna promenljiva*.

Definicija: Funkcija X koja svakom slučajnom događaju $\omega \in \Omega$ dodeljuje realni broj X(ω) zove se slučajna promenljiva. Slučajne promenljive obeležavamo velikim slovima X,Y,Z.....

Znači, slučajna promenljiva je preslikavanje skupa Ω u skup realnih brojeva, za razliku od verovatnoće, koja je preslikavanje skupa Ω u skup [0,1].

Razlikujemo dva osnovna tipa slučajnih promenljivih, diskretne i neprekidne slučajne promenljive. Podela se vrši u zavisnosti da li slučajna promenljiva uzima vrednosti u konačnom, odnosno prebrojivom ili neprebrojivom skupu vrednosti.

Generisanje mogućih vrednosti za jednu slučajnu promenljivu obavlja se korišćenjem odgovarajućih algoritama. Vrednosti generisane na taj način mogu posedovati malo, ali u nekim slučajevima bitno odstupanje od teorijskih matematičkih vrednosti. Međutim, u našem, kao i u mnogim simulacijama koje se bave sistemima iz svakodnevnog života, a ne osetljivim naučnim eksperimentima, ovo odstupanje je zanemarljivo.

2.4.2. Diskretne slučajne promenljive

Diskretne nasumične vrednosti, su po definiciji, one nasumične vrednosti čiji je skup mogućih vrednosti konačan ili barem prebrojivo beskonačan. U simulacijama diskretnih događaja ove varijable se najčešće koriste za celobrojne vrednosti koje se koriste za brojanje.

Neka slučajna promenljiva X može da uzme vrednosti $x_1,x_2...x_n$, sa verovatnoćama $p_1,p_2...p_n$, pri čemu je $p_1+p_2+...+p_n=1$. Skup parova $(x_1,p_1=P\{X=x_1\})$,i= 1,2,..., n ili

$$\begin{pmatrix} x_1 & x_2 & \dots \\ p(x_1) & p(x_2) & \dots \end{pmatrix}$$

čine zakon raspodele verovatnoća slučajne promenljive X. Zakon raspodela verovatnoća slučajne promenljive je pravilo po kome svakoj vrednosti slučajne promenljive X pridružujemo odgovarajuću verovatnoću p. Zakonom raspodele, ukupna verovatnoća, koja je jednaka 1 , raspodeljena je na pojedine vrednosti slučajne promenljive [6].

2.4.3. Kontinualne slučajne promenljive

Cilj ovog poglavlja je upoznavanje sa kontinualnim nasumičnim vrednostima, algoritmima za njihovo generisanje i primenama raspodela koje one koriste. Kontinualne nasumične vrednosti se mogu koristiti za generisanje slučajnih elemenata modela, i upravo će one biti osnova za generisanje nasumičnih aspekata sistema. U nastavku ćemo se baviti nekim od najkorišćenijih raspodela, sa fokusom na one koje se će se koristiti u simulaciji koja je tema ovog rada.

Po definiciji, kontinualne nasumične promenljive su one koje mogu imati beskonačno mnogo vrednosti. U najvećem broju slučajeva ove vrednosti se koriste za "stvarne" varijable iz realnih sistema, kao što su proteklo vreme, učestalost, količina, rastojanje i slično.

Kontinualna nasumična promenljiva X je jedinstveno definisana skupom mogućih vrednosti *X*, gde se za *X* može reći da predstavlja otvoreni interval (a,b), pri čemu a može biti $-\infty$, a b ∞. Pored toga X određuje odgovarajuća funkcija gustine verovatnoće (eng. Probability density function - PDF), koja je realna funkcija $f(\cdot)$ definisana za svako $x \in X$, takvo da:

$$\int_a^b f(x)dx = Pr(a \leq X \leq b)$$

za interval $(a, b) \subseteq X$. Po definiciji, $x \in X$ je moguća vrednost za X ako i samo ako $f(x) > 0$. Dodatno $f(\cdot)$ se definiše kao:

$$\int_x f(x)dx = 1$$

gde se integral traži za sve $x \in X$.

Standarda devijacija

Standardna devijacija je u statistici apsolutna mera disperzije u osnovnom skupu. Ona nam govori, koliko u proseku elementi skupa odstupaju od aritmetičke sredine skupa. Označava se grčkim slovom sigma, σ. Formula za njeno izračunavanje je:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

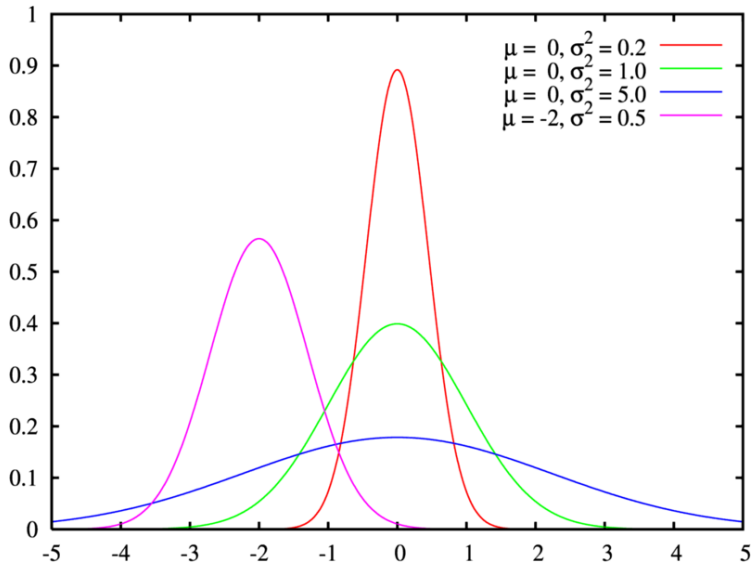
gde je N broj elemenata skupa, μ aritmetička sredina skupa a x_i i-ti član skupa.

Normalna slučajna promenljiva

Kontinualna nasumična promenljiva X je normalna ako i samo ako ima raspodelu $N(\mu, \sigma)$ koja je zadata funkcijom:

$$f(x) = \left(\frac{1}{\sigma\sqrt{2\pi}}\right)^{(x-\mu)^2/2\sigma^2}$$

gde je μ matematičko očekivanje i σ standardna devijacija. Primeri normalne raspodele u zavisnosti od različitih vrednosti za matematičko očekivanje i standardnu devijaciju dati su na slici 4.



Slika 4: Normalna raspodela

Normalna raspodela ili Gausova raspodela, je važna grupa neprekidnih raspodela verovatnoće, sa primenama u mnogim poljima. Članovi grupe normalne raspodele su definisani preko dva parametra, matematičko očekivanje, i varijansa (standardna devijacija) - σ.

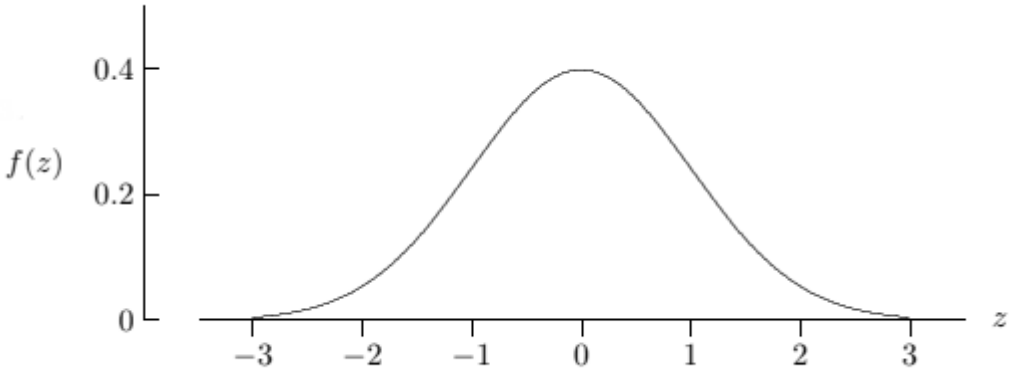
Važnost normalne raspodele kao modela kvantitativnih fenomena u prirodnim i društvenim naukama je posledica centralne granične teoreme. Mnoga psihološka merenja i fizički fenomeni se mogu dobro aproksimirati normalnom raspodelom. Iako su mehanizmi koji leže u osnovi ovih fenomena često nepoznati, upotreba modela normalne raspodele se teoretski opravdava pretpostavkom da mnogo malih, nezavisnih uticaja aditivno doprinose svakoj opservaciji.

Normalna raspodela se javlja i u mnogim oblastima statistike. Na primer, srednja vrednost uzorka ima približno normalnu raspodelu, čak i ako raspodela verovatnoće populacije iz koje se uzorak uzima nije normalna. Normalna raspodela je najčešće korišćena familija raspodela u statistici, i mnogi statistički testovi su bazirani na pretpostavci normalnosti. U teoriji verovatnoće, normalne raspodele se javljaju kao granične raspodele više neprekidnih i slučajnih familija raspodela.

Standardna normalna slučajna promenljiva

Kontinualna nasumična vrednost Z je standardna normalna promenljiva sa normalnom raspodelom (Slika 4) - *N(0, 1)* ako i samo ako je skup svih mogućih vrednosti Z = (−∞, ∞) i ako je PDF:

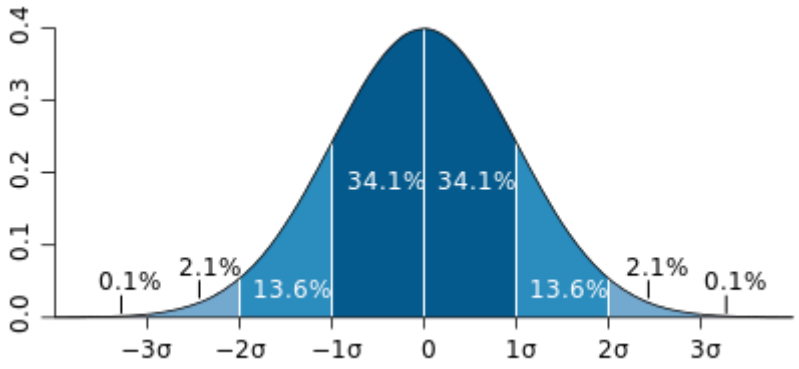
$$f(z) = \left(\frac{1}{\sqrt{2\pi}}\right)^{-z^2/2}$$



Slika 5: Standardna normalna raspodela

Z ima normalnu raspodelu - $N(\mu, \sigma)$ gde je μ matematičko očekivanje i σ standardna devijacija, sa vrednostima $\mu = 0$ *i* $\sigma = 1$.

U praksi, često se pretpostavlja da su podaci iz približno normalno raspodeljene populacije. Ako je ta pretpostavka opravdana, onda se oko 68% vrednosti nalazi u intervalu od plus-minus jedne standardne devijacije od aritmetičke sredine, oko 95% vrednosti se nalazi u intervalu od plus-minus dve standardne devijacije, a oko 99,7% se nalazi unutar plus-minus 3 standardne devijacije. Ovo je poznato kao Pravilo 68-95-99,7, ili empirijsko pravilo (Slika 6).



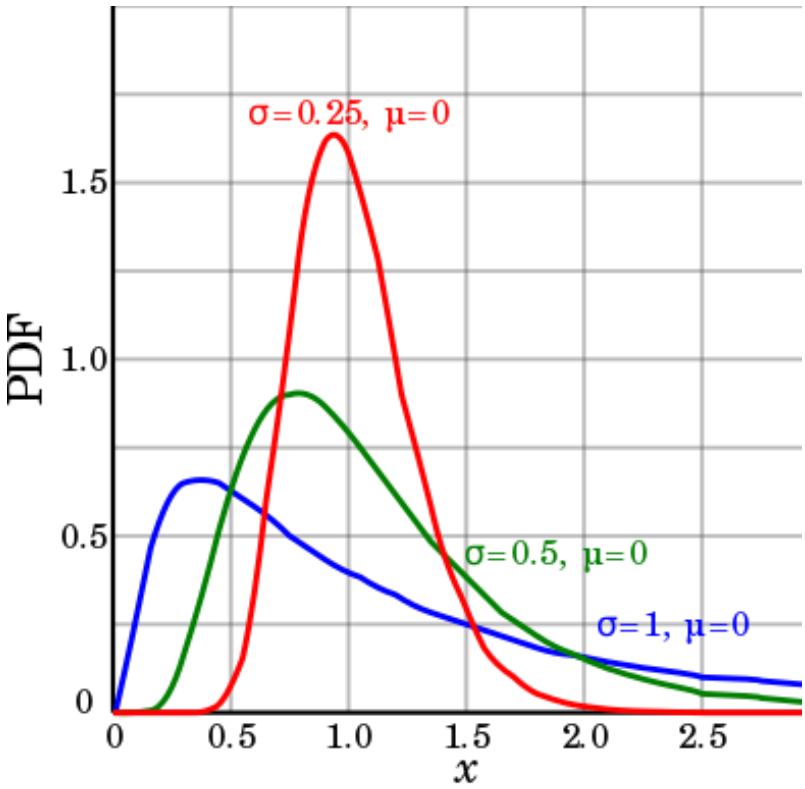
Slika 6: Raspodela verovatnoće normalne slučajne promenljive

Logaritamska normalna slučajna promenljiva

Slučajna promenljiva X je logaritamska normalna ili skraćeno log-normalna - $L(\mu, \sigma)$ ako i samo ako važi

$$X = e^{\mu+\sigma Z}$$

gde je Z normalna slučajna promenljiva *N(0, 1).* Primeri log-normalne raspodele u zavisnosti od različitih vrednosti za matematičko očekivanje i standardnu devijaciju dati su na slici 7.



Slika 7: Logaritamska raspodela

Kao i normalna slučajna promenljiva $N(\mu, \sigma)$, i $L(\mu, \sigma)$ se zasniva na transformaciji *N(0,1)* slučajne promenljive. Međutim, transformacija u ovom slučaju nije linearna, što sa sobom donosi veoma važne prednosti. $L(\mu, \sigma)$ slučajna promenljiva je po svojoj prirodi pozitivna i samim tim pogodna za modelovanje vrednosti koje moraju biti pozitivne, kao što je to slučaj sa vremenima obrade zahteva, čekanja i slično.

Eksponencijalna slučajna promenljiva

Slučajna promenljiva X ima eksponencijalnu raspodelu $\varepsilon(\lambda)$sa parametrom λ, λ>0, ako njena funkcija gustine verovatnoće[1], $f(x)$, i funkciju raspodele, F(x) , imaju sledeći oblik:

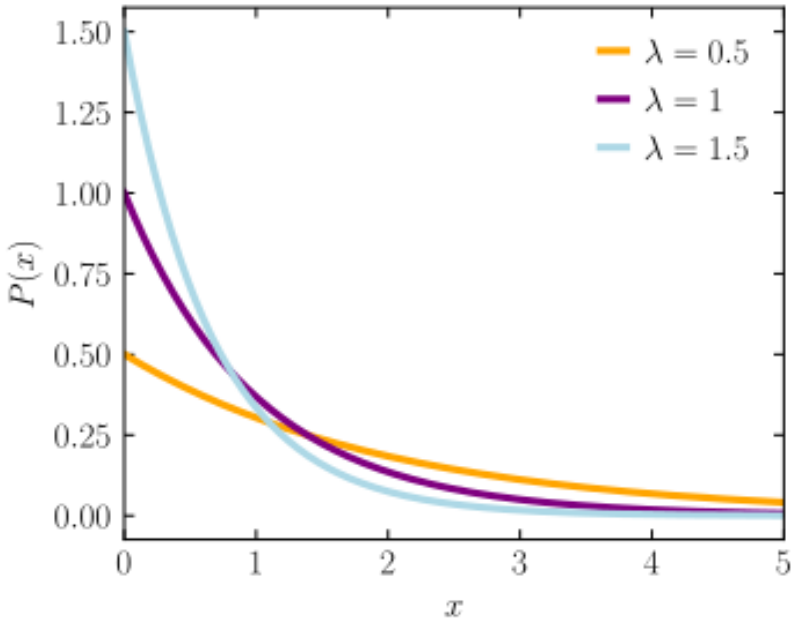
$$f(x) = \begin{cases} \lambda e^{-\lambda x}, & x \geq 0 \\ 0, & x < 0 \end{cases} \quad \text{i} \quad F(x) = \begin{cases} 1 - e^{-\lambda x}, & x \geq 0 \\ 0, & x < 0 \end{cases}.$$

Funkcija gustine verovatnoće eksponencijalne raspodele je monotono opadajuća funkcija na desno i karakteriše je eksponencijalno opadanje desnog repa. Ona ima sledeći oblik:

$$F(x) = e^{-\lambda x}$$

što znači da se događaji sa visokom vrednošću dešavaju sa verovatnoćom koje teži nuli.

Na slici je prikazana funkcija gustine verovatnoće za različite vrednosti parametra λ (Slika 8).



Slika 8: Eksponencijalna raspodela

[1] Funkcija gustine verovatnoće je funkcija čija se vrednost u datom uzorku (ili tački) u prostoru uzoraka (skupu mogućih vrednosti koje slučajna promenljiva može da poprimi) može protumačiti kao *relativna verovatnoća* da će vrednost slučajne promenljive biti jednaka tom uzorku.

Gama slučajna promenljiva

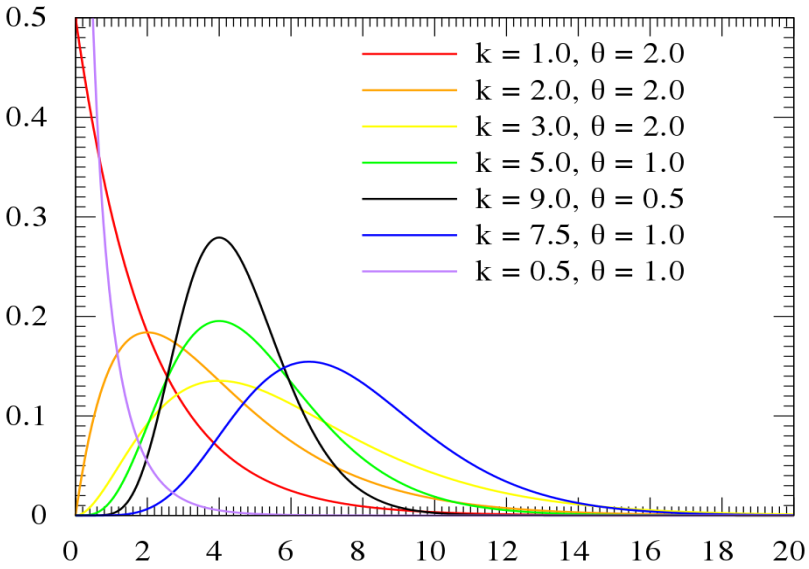
Gama raspodela je uopštenje eksponencijalne raspodele. Ova raspodela pripada porodici dvoparametarskih kontinualnih raspodela, sa parametrima *k-oblik* i *Θ-skala*, ili korišćenjem parametara $\alpha = k$ i inverznog parametra $\beta = 1/\theta$. Pri čemu su u oba slučaja parametri pozitivni realni brojevi.

Za slučajnu promenljivu X kažemo da ima gama raspodelu ako ima sledeću funkciju gustine verovatnoće:

$$f(x;\alpha,\beta) = \frac{\beta^{\alpha} x^{\alpha-1} e^{-\beta x}}{\Gamma(\alpha)}$$

za x > 0 i $\alpha,\beta > 0$. Gde je $\Gamma(\alpha)$ gama funkcija (Slika 9) i za sve pozitivne celobrojne vrednosti α ima vrednost:

$$\Gamma(\alpha) = (\alpha - 1)!$$



Slika 9: Gama raspodela

2.5. Računarski model

Nakon definisanja specifikacionog modela, izrada kompjuterskog programa predstavlja relativno jednostavan zadatak. Iako mnogi moderni programski jezici sadrže sve potrebne alate za implementaciju osnovnih gradivnih komponenti modela, za izradu simulatora diskretnih događaja o kojem je reč u ovom radu koristićemo Python programski jezik i biblioteke koje on poseduje.

Jedna od najkorisnijih biblioteka koje pružaju alate za jednostavniju implementaciju simulatora diskretnih događaja je SimPy framework. U ovom poglavlju ćemo objasniti osnovne koncepte koje SimPy pruža.

2.5.1. SimPy framework

SimPy je biblioteka koja se koristi u simulacijama diskretnih događaja. Ponašanje aktivnih komponenti sistema modelovano je *procesima (processes)*. Pri čemu se svi aktivni procesi obavljaju u okviru definisanog *okruženja (environment)*. Tokom „života“ procesi interaguju sa okruženjem i sa ostalim procesima definisanim u okviru okruženja preko *događaja (event).* Iako se, u teoriji, može koristiti i za kontinualne simulacije, SimPy je razvijen pre svega za simulacije disktetnih događaja, gde pruža alate za izvođenje i praćenje simulacija kod kojih procesi međusobno interaguju. [7]

Okruženje – Environment

Okruženje (Enviorment) simulacije zaduženo je za kontrolu vremena simulacije, planiranje izvršenja procesa, kao i obradu i procesuiranje događaja. Pored toga okruženje pruža mogućnost kontrole rada simulacije. Najvažnija metoda Enviorment klase jeste *run()* pomoću koje se započinje izvršenje simulacije koje traje unapred definisan vremenski period ili sve dok se ne završi izvršenje određenog procesa, tačnije dok se ne desi određen događaj. Drugi, podjednako važan metod ove klase jeste *process()* koji se koristi za definisanje procesa unutar trenutnog okruženja. Na slici 10 dat je primer jednostavne simulacije koja pokreće proces *my_proc* i izvršava se sve dok navedeni proces traje. [7]

```
>>> import simpy
>>>
>>> def my_proc(env):
...     yield env.timeout(1)
...     return 'Monty Python's Flying Circus'
>>>
>>> env = simpy.Environment()
>>> proc = env.process(my_proc(env))
>>> env.run(until=proc)
'Monty Python's Flying Circus'
```

Slika 10: Primer jednostavne simulacije

Simulacije se mogu koristiti za ispitivanje i analizu promena u okviru sistema ne samo u bliskoj budućnosti, već se može baviti dugoročnim posledicama izmene određenih parametara. U ovakvim slučajevima je nepraktično koristiti okruženje u kojem se promene dešavaju sinhrono realnom satu. SimPy sadrži mehanizam za prevazilaženje ovog problema - *RealtimeEnvironment.* Za razliku od normalnog okruženja (*Environment*), *RealtimeEnvironment* daje mogućnost definisanja *factor* parametra pomoću koga se određuje koliko realnog vremena prođe između koraka simulacije.

Na slici 11 prikazno je kako je moguće konvertovati „normalnu“ u „real-time“ simulaciju. Podešavanjem parametra factor na vrednost 0.1, između koraka simulacije prolazi desetina sekunde. Sa druge strane ukoliko se vrednost podesi na factor=60 između koraka simulacije će prolaziti 60 sekundi, što znači da će simulacija teći 60 puta brže.

```
>>> import time
>>> import simpy
>>>
>>> def example(env):
...     start = time.perf_counter()
...     yield env.timeout(1)
...     end = time.perf_counter()
...     print('Duration of one simulation time unit: %.2fs' % (end - start))
>>>
>>> env = simpy.Environment()
>>> proc = env.process(example(env))
>>> env.run(until=proc)
Duration of one simulation time unit: 0.00s
>>>
>>> import simpy.rt
>>> env = simpy.rt.RealtimeEnvironment(factor=0.1)
>>> proc = env.process(example(env))
>>> env.run(until=proc)
Duration of one simulation time unit: 0.10s
```

Slika 11: Primer upotrebe - RealtimeEnviorment

Proces

Prethodni primeri su pokazali na koji način je moguće pokrenuti proces unutar okruženja, a sada ćemo detaljnije opisati šta je zapravo SimPy procesi. Proces je definisan korišćenjem python generatora i može biti metod klase ili funkcija definisana u okviru nekog paketa. Tokom svog izvršenja procesi kreiraju događaje i vraćaju ih korišćenjem *yield* naredbe. Yield je ključna reč u Paython-u koja se koristi za povratak iz funkcije (slično kao return), sa tom razlikom da yield neće uništiti lokalno stanje i vrednosti lokalnih varijabli unutar funkcije. Prilikom sledećeg poziva izvršenje funkcije se nastavlja od poslednje yield naredbe. Svaka Python funkcija koja sadrži yield označava se kao generator. [8].

Pozivom yield-a proces prelazi u suspendovano stanje, i za taj proces kažemo da je yield-ovao događaj (event). SimPy će ponovo nastaviti izvršenje procesa koji je suspendovan kada se desi događaj na koji je on čekao. Važno je naglasiti da više procesa može čekati na isti događaj. Na slici 10 dat je primer jednostavnog procesa za simuliranje rada automobila. Proces prikazan u primeru sa slike tokom svog „života“ prolazi kroz dva stanja – vožnja i parkirano stanje. Korišćenjem yield naredbe definisano je vreme koje će proces provesti u svakom od ovih stanja.

```
>>> def car(env):
...     while True:
...         print('Start parking at %d' % env.now)
...         parking_duration = 5
...         yield env.timeout(parking_duration)
...
...         print('Start driving at %d' % env.now)
...         trip_duration = 2
...         yield env.timeout(trip_duration)
```

Slika 12: Primer jednostavnog procesa

Događaj – Event

Predstavljanje događaja u okviru simulacije obavlja se korišćenjem Event klase. Događaj mogu imati jedno od sledećih stanja:

- Možda će se desiti (*not triggered*)
- Desiće se (*triggered*)
- Desilo se (*processed*)

Tokom svog „života“ događaj prolazi kroz navedena stanja jednom i tačno u tom redosledu. Takođe, oni su usko spregnuti sa vremenom i vreme uzrokuje promenu stanja događaja. Inicijalno, svaki događaj se nalazi u stanju *not triggered.* Prilikom prelaska u stanje *triggered*, kada se ispune za to potrebni uslovi, događaj se smešta u red čekanja i pritom mu se dodeljuje vremenski trenutak kada će se izvršiti. On prelazi u sranje *processed* u trenutku kada ga SimPy pročita iz reda čekanja i tada se nastavljaju procesi koji su čekali na njegovo izvršenje.

Događaji mogu biti različitih tipova i imati brojne funkcije, međutim jedan od značajnijih jeste *Timeout.* Ovaj tip događaja se aktivira nakon nekog definisanog vremenskog perioda. Timeout pružaju mehanizam zaustavljanja procesa, tokom kojeg se proces nalazi u stanju *spavanja* tokom kojeg se stanje njegovih parametara ne menja. Ovaj, kao i ostali događaji, se mogu kreirati pozivom odgovarajuće funkcije okruženja - *Environment.timeout().*

Resurs

SimPy sadrži i alate za rešavanje problema u modelovanju. Osnovna komponenta za definisanje resursa ograničenog kapaciteta predstavljena je objektima *Resources* klase. Jednom resursu istovremeno može pristupati ograničen broj procesa. Broj procesa koji može koristiti jedan resurs definisan je *kapacitetom* resursa. Procesi zahtevaju korišćenje određenog resursa pozivom *request()* metode i nakon završene upotrebe vrši se njegovo oslobađanje. Svakom od objekata Resources klase pridružen je red čekanja u koji se procesi smeštaju ukoliko je resurs zauzet, sve dok ne dođu na red za njegovo korišćenje. Osim osnovne Resource klase, čijim objektima je pridružen fifo red, SimPy podržava i kreiranje zahteva za korišćenje resursa različitih prioriteta. Ovo omogućavaju *PriorityResource* i *PreemptiveResource* klase kod kojih se određenim zahtevima daje viši prioritet.

2.6. Rad simulacije

Nakon uspešnog razvoja modela potrebno je definisati korake simulacije koji će iskoristiti izrađen računarski model (kompjuterski program).

1. Definisanje simulacionih eksperimenata. Iako ovaj korak može delovati jednostavno, u slučaju da postoji veliki broj parametara u okviru modela, od kojih svaki ima veći broj vrednosti od interesa, primenom osnovne kombinatorike, dolazimo do broja eksperimenata koji eksponencijalno raste sa brojem varijabli.
2. Izvesti neophodna pokretanja simulacije koja će generisati potrebne izlazne podatke. Tokom rada programa potrebno je pamtiti kako izlazne podatke tako i vrednosti parametara u svakom od koraka simulacije.
3. Analiza rezultata simulacije. Analiza izlaznih podataka je po svojoj prirodi statistička i uključuje korišćenje osnovnih statističkih alata, kao što su srednje vrednosti, odstupanja, percentili, korelacije, histogrami i td.
4. Donošenje odluka. U najvećem broju slučajeva nakon statističke analize dolazi se do rezultata koji iziskuju pokretanje određenih akcija i promena u realnom sistemu koji je bio predmet analize. Nakon ovoga, korisno je pratiti do koje mere vrednosti dobijene simulacijom odstupaju od vrednosti dobijenih nakon analize izmenjenog sistema. Ovo je od posebnog interesa ukoliko je potrebno ponovo upotrebiti isti kompjuterski program, i može dovesti do promena u korišćenom modelu ukoliko su uočena značajna odstupanja.
5. Dokumentovanje rezultata. Čak i u slučaju neuspeha, dokumentovanje rezultata je od velike važnosti kako bi se unapredili svi naredni pokušaji izrade simulacija. Pored toga, dobra dokumentacija olakšava izradu simulacija sa sličnim ciljevima.

3. Simulacija računarskih resursa korišćenjem Python-a

Korišćenjem navedenih teorijskih metoda u nastavku ćemo obraditi simulaciju sistema za dodelu računarskih resursa studentima. Sistem koji se simulira za cilj ima da studentima (ili uopšteno korisnicima) omogući pristup računarskim resursima kao što su virtuelne mašine ili programske licence. Organizaciji kojoj sistem pripada dostupan je određen, ograničen broj licenci i mašina, i koje se mogu dodeliti korisnicima. Broj resursa koji će biti dostupan u velikoj meri zavisi od finansijskih mogućnosti same organizacije, međutim, čak i kada to nije slučaj i ustanova može priuštiti veći broj resursa, najčešće broj potrebnih licenci znatno manji od ukupnog broja korisnika. Dalje, prirodno je pretpostaviti da će se studenti međusobno razlikovati u pogledu vremena korišćenja određenog resursa, kao i po trenutku pristupanja resursu. Uzimajući ovo u obzir jasno je da će broj zauzetih resursa u svakom trenutku biti znatno manji od ukupnog broja studenata.

Analitičke metode pružaju mogućnost ispitivanja ponašanja sistema, studenata i njihovih navika. Na osnovu ovako sakupljenih podataka moguće je utvrditi navike studenata i proceniti maksimalan broj resursa koje će biti potreban u jednom trenutku i na osnovu toga doneti odluku o broju licenci koje će organizacija pribaviti. Međutim ovo može biti dugotrajan i veoma skup proces, koji je teško opravdati u slučaju organizacije koja pokušava da uštedi izbegavajući prekomerne troškove koji dolaze sa pribavljanjem nepotrebno velikog broja resursa. Upravo zbog toga, primena simulacije diskretnih događaja predstavlja poželjnije rešenje. Primena simulacije omogućava ispitivanje ponašanja sistema i dobijanje povratnih informacija čije prikupljanje bi trajalo mesecima, u samo nekoliko minta. Benefiti korišćenja simulacije takođe uključuju i mogućnost brzog testiranja promene koju izmena određenog parametra unosi u celokupni sistem, nešto što primenom drugih metoda jednostavno nije moguće.

Cilj ovog rada je upravo kreiranje simulacije koja će omogućiti fakultetu (ili bilo kojoj organizaciji) da utvrdi broj resursa koji su potrebni za zadovoljenje potreba korisnika. Potrebno je utvrditi optimalan skup, koji će minimizovati vreme čekanja studenata i ujedno sprečiti pribavljanje prekomernog broja licenci.

3.1 Ciljevi i parametri sistema

Kao što smo već prethodno naveli aplikacija koja je predmet ovog rada razvijena je sa ciljem da ustanovama koje planiraju automatizaciju dodele resursa studentima pruži uvid u performanse sistema. Prioritet, i jedna od osnovnih karakteristika koju jedan takav sistem mora da ima, je mogućnost dodele resursa svim korisnicima u što kraćem roku, pri čemu se kao krajnji cilj može postaviti sistem koji pruža trenutna dodela resursa. Uzimajući u obzir promene frekventnosti zahteva u odnosu na period dana, kao i određene trenutke od značaja (npr. ispitni rok), potrebno je obezbediti dovoljnu količinu resursa za ispunjenje svih zahteva. Stoga, upravo određivanje potrebnog maksimalnog broja resursa predstavlja drugi važan cilj simulacije.

Pored ovoga, podjednako važno za performanse sistema je vreme potrebno da se resurs pripremi za upotrebu. Obzirom da je ovaj parametar u najvećem broju slučajeva fiksan i zavisi od samog resursa, ono na šta se može uticati jeste broj radnika ili računara koji će biti zadužen za pripremu resursa. Međutim, verovatno i važniji aspekt sistema i ono na šta treba obratiti

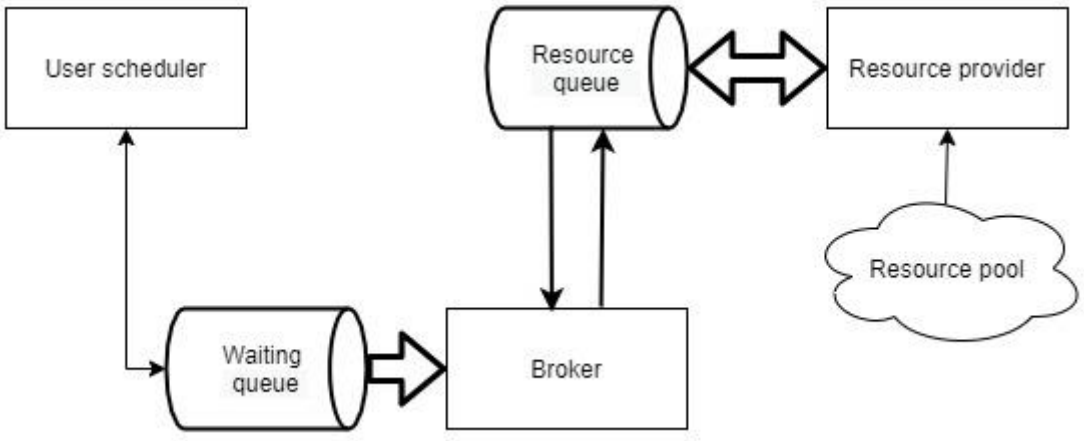
pažnju prilikom projektovanja jeste trenutak u kojem će se započeti sa pripremom resursa kako bi on bio spreman u trenutku kada korisnik zatraži pristup. Tačnije, moguće je održavati skup pripremljenih resursa koji su spremni da budu dodeljeni korisniku na korišćenje. Jasno je da je prilikom ovakvog ključno određivanje optimalnog broja resursa koji će biti na "čekanju". Trenutak pripreme resursa moguće je odrediti na osnovu više parametara sistema, koji su, zbog svoje važnosti, predmet istraživanja ovog rada, i utvrđivanje njihovih optimalnih vrednosti, kao i maksimalan broj spremnih resursa možemo istaći kao još jedan od ciljeva simulacije.

Simulacija diskretnih događaja o kojoj govori ovaj rad, za pojedine vrednosti korisni nasumično generisane promenljive, o kojima je bilo reči u prethodnim poglavljima. Međutim za pravilno funkcionisanje simulacije ipak je potrebno uneti određen broj ulaznih parametara kako bi se omogućilo generisanje adekvatnih izlaznih podataka. Vrednosti ovih promenljivih date su na slici 13. Pored broja studenata i resursa dostupnih organizaciji, moguće je uočiti i parametre potrebne za izračunavanje nasumičnih promenljivih. Vrednosti ovih parametara utvrđene su eksperimentalnim putem i omogućavaju generisanje vrednosti koje odgovaraju onim prisutnim u realnom sistemu. Detalji upotrebe i način korišćenja biće izneti u narednim poglavljima.

```
USER_COUNT = 200

NEXT_LOGIN_MEAN = 5
NEXT_LOGIN_STD = 3

USERS_PER_LOGIN_MEAN = 20
USERS_PER_LOGIN_STD = 3

READY_COUNT = 10
MAX_AVAILABLE_RESOURCES = 150

RESOURCE_USAGE_TIME_MEAN = 60
RESOURCE_USAGE_TIME_STD = 30

RESOURCE_PREPARE_TIME_MEAN = 5
RESOURCE_PREPARE_TIME_STD = 2

CRITICAL_UTILISATION_PERCENT = 0.7
RESOURCE_ADD_NUMBER = 1
RESOURCE_ADD_RATE = 3

GAMMA_25_SHAPE = 0.181
GAMMA_25_SCALE = 0.56
GAMMA_75_SHAPE = 0.36
GAMMA_75_SCALE = 0.12

EXPONENTIAL_LAMBDA = 5/3
```

Slika 13: Parametri simulacije

3.2 Arhitektura sistema



Slika 14: Arhitektura sistema

Aplikacija je razvijena oko centralne komponente - Broker. Broker predstavlja servis simulacije i zadužen je za obradu zahteva korisnika čije generisanje obavlja User scheduler. U okviru User scheduler komponente obavlja se generisanje ulaznog skupa podataka na osnovu parametara aplikacije. Tokom rada Broker obradu zahteva vrši u komunikaciji sa Resource provider-om koji održava skup spremnih resursa i obavlja pripremu novih u trenucima određenim algoritmom koji se u tom trenutku primenjuje. Obzirom da jednovremena obrada svih zahteva nije moguća u realnom sistemu, u okviru aplikacije ova pojava se simulira pridruživanjem redova čekanja Broker-u i Resource provide-u u koje se smeštaju zahtevi koje nije moguće obraditi u tom trenutku. U nastavku ćemo objasniti rad navedenih komponenti, njihove zadatke i ciljeve.

3.3 User Scheduler

Prikupljanje ulaznog skupa podataka analitičkim metodama u cilju istraživanja rada sistema za dodelu resursa studentima je dugotrajan i mukotrpan posao, koji iziskuje opsežno istraživanje, i pre svega značajna finansijska sredstva. Pored toga otvoreno je pitanje da li bi se na taj način dobijen skup statistički značajno razlikovao od onog generisanog matematičkim metodama. Ponašanje grupe ljudi je proučavan i dobro istražene proces i upravo zbog toga sa velikim procentom sigurnosti možemo tvrditi da je generisanje ulaznog skupa podataka za potrebe ovog rada ne samo adekvatno nego i preporučljivo, jer na taj način dolazimo do merodavnih rezultata bez značajnijih vremenskih i finansijskih ulaganja.

User scheduler je komponenta sistema čiji je zadatak da, korišćenjem statističkih metoda, pripremi ulazni skup podataka potreban za rad simulacije. Prilikom pokretanja simulacije generišu se podaci koji će odrediti učestalost korisničkih zahteva, kao i vreme korišćenja resursa.

U svrhu generisanja stohastičkih parametara u okviru ove simulacije biće korišćeno više različitih tipova nasumičnih promenljivih. Obzirom da je svaki tip nasumičnih vrednosti pogodan za simulaciju određenih pojava, upravo smo se time vodili prilikom izbora konkretnih raspodela koje će se primenjivati za generisanje potrebnih vrednosti.

Broj korisnika koji pristupa sistemu

Broj korisnika koji će jednovremeno pristupiti sistemu u cilju pribavljanja resursa određuje se korišćenjem normalne (Gausove) raspodele (Slika 15). Korišćenjem ovako generisane slučajne promenljive dobijamo broj koji se u 68% slučajeva nalazi u prostoru plus-minus jedne standardne devijacije u odnosu na srednju vrednost, ostavljajući pritom dovoljnu verovatnoću za nepredviđene skokove ili padove u broju istovremenih zahteva.

Variranjem parametara, matematičkog očekivanja i standardne devijacije, moguće je simulirati kako dane sa manjim brojem zahteva, tako i dane ekstremne zauzetosti sistema, kakav je slučaj, recimo u toku ispitnog roka. Vrednosti matematičkog očekivanja i standardne devijacije određene su parametrima sistema USERS_PER_LOGIN_MEAN i USERS_PER_LOGIN_STD, redom. Takođe u cilju lakšeg upoređivanja izlaznih rezultata simulacije, ukupan broj korisnika tokom jednog pokretanja simulacije fiksira se na konstantnu vrednost koja je određena parametrom USER_COUNT. Simulacija podržava i vremenski ograničen rad, tokom kojeg rad simulacije ne zavisi od broja korisnika već traje unapred definisani vremenski interval. Izbor između ova dva režima rada vrši se pomoću parametra CONSTANT_USER_COUNT_ENABLED.

```
if Properties.CONSTANT_USER_COUNT_ENABLED:
    self.USERS_NUMBER = []
    total_users_count = Properties.USER_COUNT
    while total_users_count > 0:
        user_count = Properties.get_positive_value_gauss(Properties.USERS_PER_LOGIN_MEAN,
                                                         Properties.USERS_PER_LOGIN_STD)
        if user_count>total_users_count:
            user_count = total_users_count

        self.USERS_NUMBER.append(user_count)
        total_users_count -= user_count

else:
    self.USERS_NUMBER = [Properties.get_positive_value_gauss(Properties.USERS_PER_LOGIN_MEAN,
                                                             Properties.USERS_PER_LOGIN_STD)
                         for _ in range(80)]
```

Slika 15: User scheduler – generisanje broja korisnika koji pristupa sistemu

Vremenski trenutak pristupa sistemu

Trenutak u toku rada simulacije u kojem će grupa korisnika zahtevati pristup resursu takođe se određuje korišćenjem normalne raspodele (Slika 16). Gausova slučajna promenljiva u ovom slučaju predstavlja vreme između dve grupe korisnika koje pristupaju sistemu. Ovaj parametar omogućuje pravilnu raspodelu korisničkih zahteva u vremenu i sprečava prekomeran broj jednovremenih korisničkih zahteva.

Vrednosti koje će ovako generisana promenljiva imati zavise od parametra sistema NEXT_LOGIN_MEAN i NEXT_LOGIN_STD.

```
self.INTER_ARRIVAL_TIMES = [Properties.get_positive_value_gauss(Properties.NEXT_LOGIN_MEAN,
                                                                Properties.NEXT_LOGIN_STD)
                            for _ in range(sum(self.USERS_NUMBER))]
```

Slika 16: User scheduler – generisanje trenutka pristupa sistemu

Vreme korišćenja resursa

Generisanje vrednosti za vremenske intervale korišćenja resursa ostvaruje se pomoću gama raspodele, na način prikazan na slici 17. Gama raspodela daje vrednosti maksimalne entropije. Eksperimentalno je utvrđeno da se korišćenjem vrednosti za $\alpha = 0.181$ i $\beta = 0.56$ u 25% slučajeva i vrednosti $\alpha = 0.36$ i $\beta = 0.12$ u 75% slučajeva i zatim linearnim skaliranjem tako dobijenih vrednosti dobija najreprezentativniji skup podataka.

Parametri sistema koji se koriste za definisanje ovih promenljivih su GAMMA_25_SHAPE, GAMMA_25_SCALE, GAMMA_75_SHAPE, GAMMA_75_SCALE. Skaliranje na adekvatnu vrednost vrši se množenjem ovako dobijenih vrednosti sa 268. U cilju simuliranja dana sa većim opterećenjima, kao što su ispitni dani, takođe je moguće podesiti skaliranje tako da dobijene vrednosti nikada ne budu ispod predefinisanog minimalnog vremena, ovo se postiže upotrebom parametra MINIMUM_USAGE_TIME.

```
self.USAGE_TIME = [np.random.gamma(Properties.GAMMA_25_SHAPE, Properties.GAMMA_25_SCALE)
                   for _ in range(math.ceil(sum(self.USERS_NUMBER) * 0.25))]
self.USAGE_TIME.extend([np.random.gamma(Properties.GAMMA_75_SHAPE, Properties.GAMMA_75_SCALE)
                        for _ in range(math.ceil(sum(self.USERS_NUMBER) * 0.75))])

self.USAGE_TIME = [(x * 268) + Properties.MINIMUM_USAGE_TIME for x in self.USAGE_TIME]
```

Slika 17: User scheduler – generisanje vremena korišćenja sistema

Vreme između dva sukcesivna zahteva

Korisnici, kao što je već navedeno sistemu pristupaju u grupama. Obzirom da je se u realnosti retko susrećemo sa situacijom u kojoj veća grupa ljudi istovremeno pokušava pristup sistemu, ove zahteve je potrebno razdvojiti. U tu svrhu se koristi vrednost koja prati eksponencijalnu raspodelu (Slika 18).

Eksponencijalna raspodela daje vrednosti bliske 0 u najvećem broju slučajeva i zbog toga je potrebno skalirati generisane vrednosti tako da između dva sukcesivna zahteva uvek postoji neki minimalni, unapred definisani razmak. Ova raspodela određena je jednim parametrom EXPONENTIAL_LAMBDA.

```
self.TIME_BETWEEN_LOGINS = [random.expovariate(Properties.EXPONENTIAL_LAMBDA) + 1
                            for _ in range(sum(self.USERS_NUMBER))]
```

Slika 18: User scheduler – generisanje vremena između zahteva

3.4 Resource provider

Ključan aspekt simulacije sistema za dodelu resursa predstavlja valjano predstavljanje istih. Imajući ovo u vidu aplikacija je implementirana korišćenjem gotovih metoda i biblioteka koje nudi SimPy framework, uz dodatak funkcionalnosti potrebnih za adekvatno predstavljanje sistema koji se simulira.

Resource klasa SimPy framework-a predstavlja srž resursa implementiranog u simulaciji. Ona pruža osnovne funkcionalnosti, obezbeđujući elementarne atribute potrebne za praćenje stanja resursa. SimPy framework omogućava praćenje kapaciteta resursa, stanja u kojem se nalazi(spreman ili zauzet) i pruža uvid u broj korisnika koji pokušava da mu pristupi. Korišćenje SimPy resursa se zahteva pozivom metode *request()*. Pristup se odobrava u slučaju da je red čekanja pridružen resursu prazan, ukoliko to nije slučaj zahtev se pamti sve dok ne dođe red na njegovo izvršenje.

Framework daje mogućnost kreiranja resursa različitih kapaciteta, međutim za potrebe ove aplikacije koristi se resurs jediničnog kapaciteta kako bi se obezbedilo pamćenje dodatnih informacija u klasi implementiranoj nad njim. Pored samog resursa, unutar klase pamti se njegov jedinstveni identifikator, kao i tip kojem pripada (licenca, virtuelna mašina...), što je prikazano na slici 19.

```
class Resource:
    def __init__(self, env, resource_id, resource_type: ResourceType):
        self.env = env
        self.simpy_resource = simpy.Resource(self.env)
        self.id = resource_id
        self.resource_type = resource_type

    def is_ready(self):
        return self.simpy_resource.count == 0

    def get_queue_size(self):
        return len(self.simpy_resource.queue)
```

Slika 19: Resource – implementacija klase

ResourceProvider klasa sadrži listu resursa koji su trenutno spremi da budu dodjeljeni korisniku na korišćenje. Pored toga u okviru ove klase definisan je i worker koji je zadužen za pripremu novih resursa, i o njemu će više reči biti u nastavku (Slika 20).

```
class ResourceProvider:
    def __init__(self, env):
        print("Starting simulation")
        self.env = env
        self.ready_resources = [Resource(env, 1, ResourceType.LICENCE) for _ in range(Properties.READY_COUNT)]

        self.worker = simpy.Resource(self.env)
```

Slika 20: Resource provider – implementacija klase

Kao što je već navedeno, Resource provider komponenta ima zadatak održavanja skupa resursa koji su spremni. Obzirom da je u svakom trenutku samo određen broj resursa pripremljen i spreman za dodelu korisnicima, ovaj servis je zadužen i za pripremu novih resursa kada je to potrebno. Maksimalan broj resursa dostupnih organizaciji je određen parametrom MAX_AVAILABLE_RESOURCES i ukoliko se ovaj broj dostigne, kreiranje novih resursa više nije moguće i preostali zahtevi će biti smešteni u red čekanja.

Algoritam dodele (Slika 21) osmišljen je sa ciljem da omogući najbrži mogući odgovor na svaki zahtev. Ukoliko u sistemu postoji resurs koji je spreman, on će biti dodeljen na korišćenje, u suprotnom iz skupa se bira onaj sa najmanjim brojem korisnika koji čeka na njegovo oslobođenje minimizujući na taj način vreme čekanja korisnika.

```
def get_resource(self) -> Resource:

    for resource in self.ready_resources:
        if resource.is_ready():
            return resource

    tuples = list(zip(range(len(self.ready_resources)), self.ready_resources))  # tuples of (i, line)
    shortest = tuples[0][0]
    for i, resource in tuples:
        if len(resource.simpy_resource.queue) < len(self.ready_resources[shortest].simpy_resource.queue):
            shortest = i
            break
    return self.ready_resources[shortest]
```

Slika 21: Algoritam dodele resursa

Druga neophodna funkcionalnost koju Resource provider pruža jeste kreiranje novih resursa. Kao što je to poznato iz iskustva korišćenja raznovrsnih sistema iz realnog života, priprema bilo čega zahteva određeno vreme i nije moguće pripremati neograničen broj stvari istovremeno. Simuliranje ovakvog ponašanja je obezbeđeno je korišćenjem deljenog resursa *worker*. Obzirom da je implementiran kao SimPy resurs, on poseduje sebi pridružen red čekanja u koji se smeštaju zahtevi za kreiranje novih resursa, čime izbegnuta mogućnost istovremenog pripremanja velikog broja mašina, kao što bi to bio slučaj i u realnom sistemu. Pored toga poznato je da je za pripremu resursa potrebno određeno vreme. Upotrebom slučajnih promenljivih za vreme čekanja na pripremu, moguće je uspešno simulirati i ovaj aspekt sistema koji predstavljamo aplikacijom. Konkretno, upotrebom normalne slučajne promenljive generiše se vremenski interval u kojem će resurs biti pripremljen pre nego što postane dostupan za korišćenje (Slika 22).

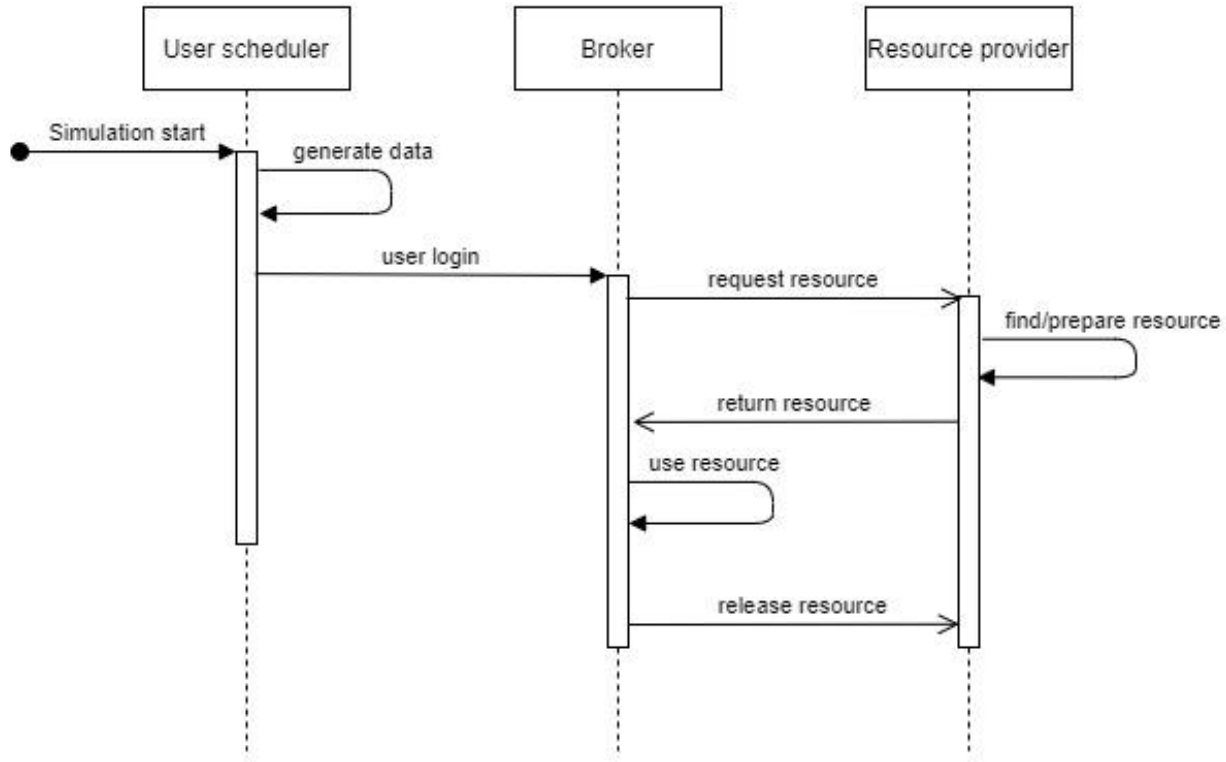
```
def prepare_one_resource(self, env):
    with self.resource_provider.worker.request() as worker:
        yield worker

        yield env.timeout(self.get_positive_value_gauss(Properties.RESOURCE_PREPARE_TIME_MEAN,
                                                        Properties.RESOURCE_PREPARE_TIME_STD))

        self.resource_provider.prepare_new_resource()
        self.analytics.register_new_resource_prepared(self.resource_provider.get_resource_count(), env.now)
```

Slika 22: Algoritam priprema resursa

3.5 Broker

Broker predstavlja centralu komponentu simulacije. Zadatak ovog servisa je prihvatanje korisničkih zahteva i njihova obrada. Jasno je da pravilna implementacija brokera neophodna kako bi se osigurao adekvatan rad simulacije. Imajući to u vidu, u nastavku će biti opisan njegov rad i detalji iza svake odluke u implementaciji.

Simulacija logovanja korisnika u sistem i slanje zahteva za resursom teče sledećim tokom (Slika 23):



Slika 23: Sekvencijalni dijagram – logovanje korisnika

1. User scheduler generise korisnički zahtev, uz potrebne dodatne parametre koji određuju vreme korišćenja resursa
2. Pozivom *login()* metode u Broker-u se registruje korisnički zahtev
3. U zavisnosti od zauzetosti brokera zahtev čeka na obradu određen vremenski period
4. Resource provider daje informaciju o dostupnosti resursa na osnovu kojih se određuje vreme koje korisnik provodi u čekanju na resurs
5. Korisnik koristi resurs određeni vremenski period
6. Resurs se oslobađa i dostupan je za ponovno korišćenje od strane drugih korisnika

Opisane radnje su zajedničke za sve zahteve, bez obzira na algoritam dodele koji Broker koristi, iz tog razloga broker je implementiran kao core klasa, iz koje je izvedeno više specifičnih tipova Broker-a, od kojih svaki koristi različit algoritam dodele i kreiranja resursa.

Nakon registrovanja zahteva pozivom *user_login()* metode korisnik se može naći u jednoj od tri situacije:

1. Ima slobodnih resursa
2. Nema slobodnih resursa ali nije dostignut maksimalan mogući broj i pristupa se pripremi resursa pre korišćenja
3. Nema slobodnih resursa i dostignut je maksimalan mogući broj resursa dostupnih organizaciji i korisnik mora da čeka na oslobođenje nekog resursa

Provera stanja obavlja se prilikom prijama zahteva korisnika u okviru *user_login()* metode (slika 24).

```
if self.resource_provider.get_resource_count() == Properties.MAX_AVAILABLE_RESOURCES:
    # nema slobodnih i ne mogu da se kreirati novi
    resource = self.resource_provider.get_resource()

elif self.resource_provider.get_resource_ready_count() == 0:
    # nema slobodnih i pristupa se kreiranju novih
    yield self.env.timeout(
        self.get_positive_value_gauss(Properties.RESOURCE_PREPARE_TIME_MEAN,
                                       Properties.RESOURCE_PREPARE_TIME_STD))

    self.prepare_new_resources(self.env)
    resource = self.resource_provider.get_resource()

else:
    # ako ima slobodnih resursa
    # self.prepare_new_resources(self.env)
    resource = self.resource_provider.get_resource()
```

Slika 24: Stanja korisničkog zahteva

Obzirom da navedena sekvenca ne zavisi od specifičnosti pojedinih implementacija Broker klase, ovaj niz provera implementiran je u BrokerCore klasi (Slika 25). Ono po čemu se različite implementacije Brokera mogu razlikovati jeste politika održavanja skupa već spremnih resursa. Različiti pristupi navedenom problemu daju drugačije performanse sistema, o čemu će detaljnije biti reči prilikom osvrta na specifičnosti različitih implementacija Broker klase.

```
@abstractmethod
def prepare_new_resources(self, env: RealtimeEnvironment):
    # priprema nove reurse
    # odredjuje kad se krece sa kreiranjem novih
    return
```

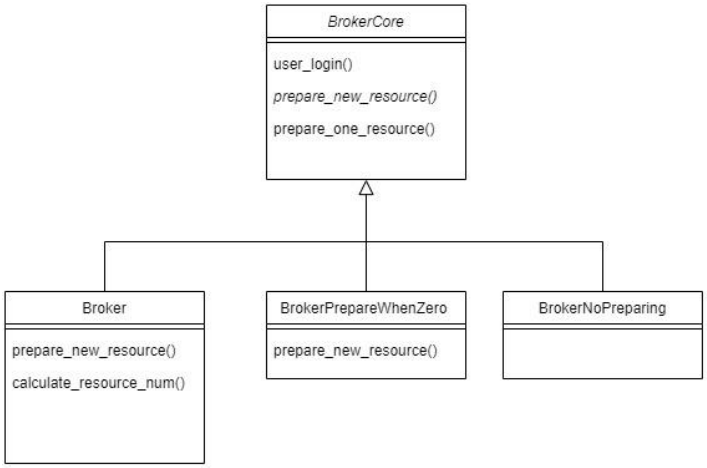
Slika 25: Apstraktna metoda za pripremu resursa

Iako postoji razlika u broju i trenutku kreiranja resursa, ono što je zajedničko za sve implementacije jeste proces kreiranja jednog resursa. Kreiranje resursa obavlja worker iz ResourceProvider-a, na način prikazan na slici 26.

```
def prepare_one_resource(self, env):
    with self.resource_provider.worker.request() as worker:
        yield worker

        yield env.timeout(self.get_positive_value_gauss(Properties.RESOURCE_PREPARE_TIME_MEAN,
                                                        Properties.RESOURCE_PREPARE_TIME_STD))

        self.resource_provider.prepare_new_resource()
        self.analytics.register_new_resource_prepared(self.resource_provider.get_resource_count(), env.now)
```

Slika 26: Metoda za pripremu jednog resursa

Aplikacija će se baviti razlikama u performansama sistema koje donose različite politike pripreme resursa. Konkretno ispitivaće se tri slučaja implementacije brokera prikazanih na slici 27:



Slika 27: Hijerarhija broker klasa

1. **Broker** - pristupa kreiranju novih resursa pre nego što se svi resursi iz skupa spremnih dodele na korišćenje
2. **BrokerPrepareWhenZero** - novi resursi se kreiraju kada se svi iz skupa dostupnih dodele korisnicima
3. **BrokerNoPreparing** - granični slučaj sistema kod koga se ne pripremaju novi resursi, cilj implementacije je ispitivanje štetnih efekata ovakvog pristupa

Prvi tip brokera je najpogodniji za primenu u realnom sistemu. Implementiran je sa idejom da se pripremi resursa pristupi u trenutku kada je sistem pod većim opterećenjem (Slika 28). Kritičan nivo zauzetosti sistema definiše se parametrom CRITICAL_UTILISATION_PERCENT, čime je omogućeno ispitivanje uticaja promene ovog parametra na performanse.

```
def prepare_new_resources(self, env: RealtimeEnvironment):
    if self.resource_provider.get_resource_used_count() > \
            Properties.CRITICAL_UTILISATION_PERCENT * self.resource_provider.get_resource_count():
        for _ in range(Properties.RESOURCE_ADD_NUMBER):
            env.process(self.prepare_one_resource(env))
        Properties.RESOURCE_ADD_NUMBER += self.calculate_resources_to_prepare_count()
```

Slika 28: Metoda za pripremu resursa – Broker sa pripremom unapred

Još jedan parametar koji može uticati na rad sistema jeste broj resursa koje će sistem kreirati u slučaju da dođe do preopterećenja i način njegovog izračunavanja dat je na slici 29. Broj resursa koji će se kreirati obrnuto je proporcionalan broju već kreiranih, što odlaže dostizanje maksimalnog broja resursa ukoliko učestalost zahteva nije dovoljno velika. Ideja ovakvog pristupa je omogućavanje pripremanja većeg obima resursa ukoliko dođe do pika u broju zahteva ali i izbegavanje kreiranja nepotrebno velikog skupa spremnih resursa. Druga veličina koja direktno utiče na izlaz ove metode jeste broj korisnika koji čekaju na resurs, i jasno je da će broj resursa koje je potrebno kreirati veći ukoliko je broj zahteva veći.

```
def calculate_resources_to_prepare_count(self):
    resource_count = self.resource_provider.get_resource_count()
    max_resources_count = Properties.MAX_AVAILABLE_RESOURCES
    number_to_add = math.floor(
        (max_resources_count / resource_count)*0.2 *
        self.resource_provider.get_users_waiting_count())

    if number_to_add < 1 and max_resources_count / resource_count < 0.05 \
            and self.resource_provider.get_users_waiting_count()>10:
        number_to_add = Properties.RESOURCE_ADD_NUMBER

    return number_to_add
```

Slika 29: Metoda za pripremu resursa – Broker sa pripremom unapred

Za razliku od ove, implementacija BrokerPrepareWhenZero ne uzima u obzir zauzetost sistema prilikom pripreme novih resursa (Slika 30). Iako ovaj pristup može delovati primamljivo zbog svoje jednostavnosti, prirodno je pretpostaviti da će on sa sobom doneti značajan pad u performansama sistema.

```
def prepare_new_resources(self, env: RealtimeEnvironment):
    if self.resource_provider.get_resource_count() == 0:
        for _ in range(Properties.RESOURCE_ADD_NUMBER):
            env.process(self.prepare_one_resource(env))
        Properties.RESOURCE_ADD_NUMBER += Properties.RESOURCE_ADD_RATE
```

Slika 30: Metoda za pripremu resursa – Broker bez blagovremene pripreme

Značaj blagovremenog reagovanja sistema na veći broj zahteva je istaknut u ovom primeru, ali pogubne efekte loše implementacije Brokera najbolje prikazuje primer Brokera koji ne pristupa kreiranju novih resursa. Upravo je i prikaz ovog graničnog slučaja glavni cilj poslednje od tri implementacije.

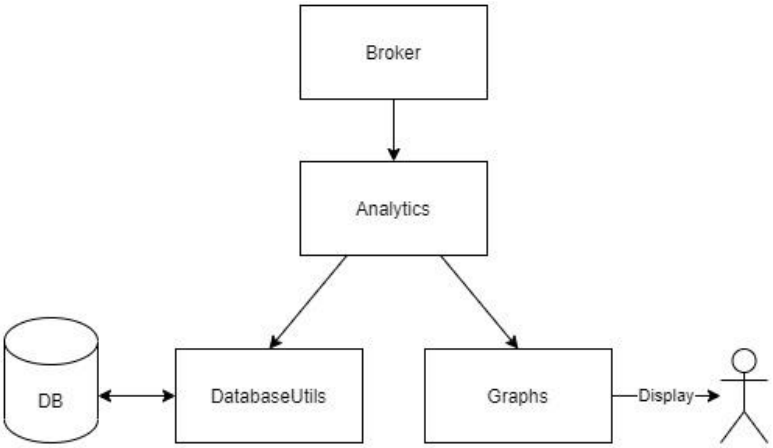
Na kraju, potrebno je skrenuti pažnju na *end_process()* metodu klase (Slika 31), kojim se emituje događaj koji označava kraj simulacije. Događaj se kreira u trenutku kada svi korisnici završe sa upotrebom resursa, i njegova upotreba biće detaljnije opisana u kasnijim poglavljima.

```
def end_process(self):
    if self.resource_provider.is_all_resource_free():
        simulation_end = self.env.event()
        yield simulation_end
```

Slika 31: Metod za emitovanje događaja koji označava kraj simulacije

3.6 Logovanje i baza podataka

Praćenje rada simulacije i kasnije analiziranje rezultata omogućeno je logovanjem značajnih događaja i njihovim pamćenjem u okviru baze podataka. Pored toga, korisniku se tokom rada simulacije prikazuju osnovni podaci koji pružaju uvid u tok simulacije. Na slici 32 prikazana je arhitektura dela sistema koje obavljaju navedene funkcije.



Slika 32: Arhitektura sistema za prikaz i logovanje

Broker je, kako je to već ranije objašnjeno zadužen za obradu zahteva korisnika. Međutim, analiza rezultata simulacije ne bi bila moguća ukoliko se ne bi pamtila promena parametra sistema i vreme potrebno za obradu svakog od zahteva. Parametri od interesa, koji će se beležiti i na osnovu kojih će se kasnije vršiti analiza su:

1. Trenutak kada se korisnik ulogovao u sistem
2. Vreme čekanja korisnika
3. Trenutak kreiranja novih resursa
4. Broj resursa koji je kreiran
5. Stepen zauzetosti resursa
6. Vreme korišćenja resursa

Vrednosti koje će se pamtiti određuju se tokom rada Broker-a. Stoga je jasno da su tražene vrednosti inicijalno prisutne u okviru Broker komponente. One se očitavaju u predefinisanim koracima obrade zahteva i prosleđuju se do instance Analytics klase. Analytics komponenta sistema zadužena je za preuzimanje informacija i njihovo dalje prosleđivanje do komponenti zaduženih za prikaz i upis u bazu.

Analytics

Klasa Analytics prikazana je na slici 33. U okviru nje vidimo promenljive koje se koriste za pamćenje trenutnog stanja sistema, kao što su upotrebljenost sistema, broj opsluženih i ukupan broj korisnika koji se ulogovan do datog trenutka. Pored toga ova komponenta sadrži parametar kojim se određuje da li će se vrednosti generisane tokom trenutne iteracije simulacije pamtiti u bazi. Vrednosti parametara koji određuju trenutak kada se korisnik logova, njegovo vreme čekanja i zauzetost sistema u određenom trenutku pamte se kao par

(<vremenski trenutak>,<vrednost>)

u okviru nizova arrivals, utilization_percent i watis_for_getting.

```
class Analytics:

    def __init__(self):
        self.database = DatabaseUtils()
        self.log_to_database = True
        self.total_user_count = 0
        self.users_served_count = 0
        self.utilization = 0

    arrivals = defaultdict(lambda: 0)
    utilization_percent = defaultdict(lambda: [])
    waits_for_getting = defaultdict(lambda: [])
```

Slika 33: Klasa Analytics

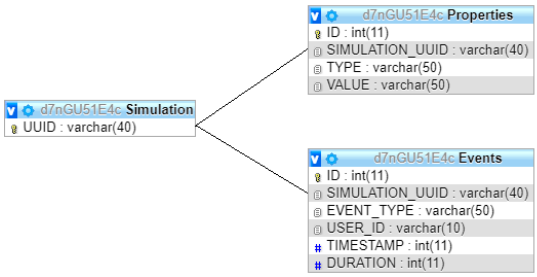
Obzirom da je procedura logovanja i slanja podataka do komponente koja vrši upis u bazu identična za sve parametre, na slici 34 prikazan je primer samo za vreme čekanja. Broker pozivom funkcije *register_user_waiting(...)* potrebnu vrednost dostavlja do komponente Analytics, nakon čega se ona upisuje u potreban niz i u zavisnosti od parametra *log_to_database* šalje do DatabaseUtils komponente.

```
def register_user_waiting(self, queue_begin, queue_end, user: User):
    wait = queue_end - queue_begin
    self.register_wait_for_getting(queue_end, wait)
    print(f"User({user.user_id}) waited {wait} minutes")
    if self.log_to_database:
        self.database.log_event(EventType.USER_WAIT.value, user.user_id, queue_end, wait)

def register_wait_for_getting(self, time, wait):
    self.waits_for_getting[int(time)].append(wait)
```

Slika 34: Primer logovanja vrednosti

DatabaseUtils

Komponenta DatabaseUtils zadužena je za upis vrednosti u bazu podataka. U svrhu ove aplikacije koristi se sql baza jednostavne seme koja se sastoji iz tabela prikazanih na slici 35.



Slika 35: Šema baze podataka

Tabela Simulation pamti uuid simulacija koji se koristi za povezivanje parametara i događaja određene iteracije. Parametri koji se koriste prilikom pokretanja simulacije pamte se u okviru Properties tabele, tok se događaji čuvaju u tabeli Events, pri čemu event_type polje specificira tačan tip događaja o kojem je reč.

Komponenta DatabaseUtils kreira konekciju ka bazi (Slika 36), nakon čega je pozivom odgovarajućih metoda moguć upis parametara i događaja simulacije (Slika 37).

```
class DatabaseUtils:
    def __init__(self):
        self.connection = self.create_server_connection("remotemysql.com", "d7nGU51E4c", "iqiyEKUAlU", "d7nGU51E4c")

    @staticmethod
    def create_server_connection(host_name, user_name, user_password, database):
        connection = None
        try:
            connection = MySQLdb.connect(
                host=host_name,
                user=user_name,
                passwd=user_password,
                database=database
            )
            print("MySQL Database connection successful")
        except Error as err:
            print(f"Error: '{err}'")

        return connection
```

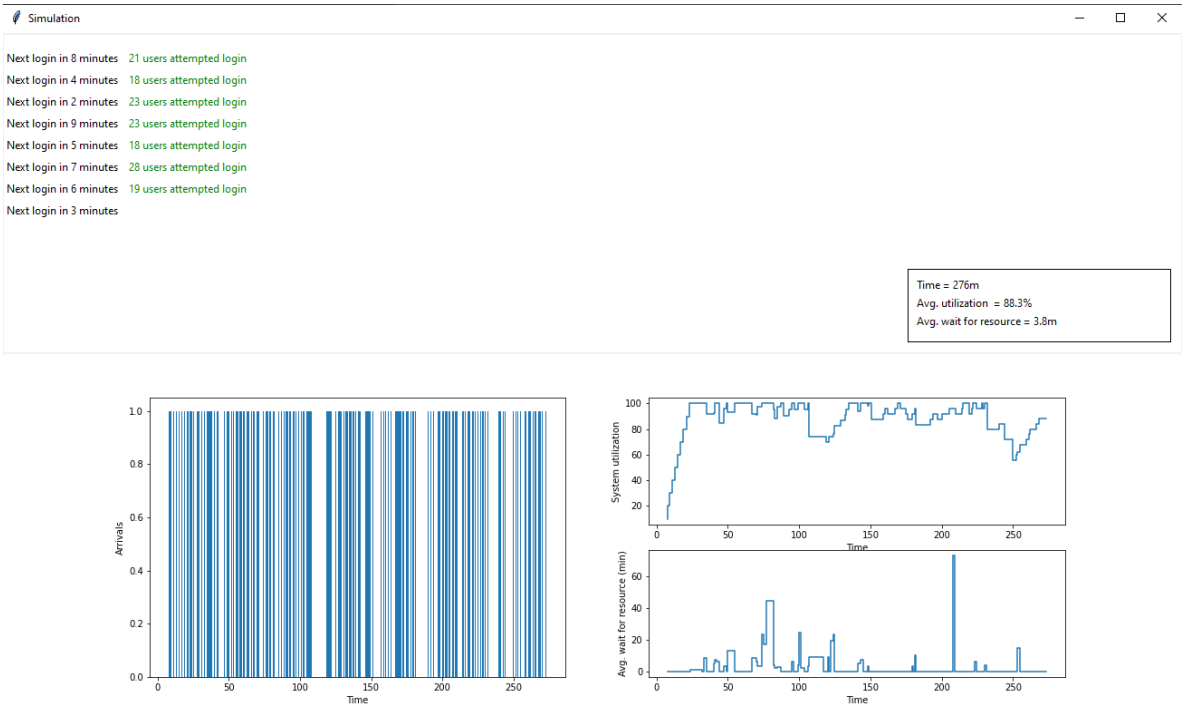
Slika 36: Klasa DatabaseUtils – kreiranje konekcije

```
def execute_query(self, query, values):
    cursor = self.connection.cursor()
    try:
        cursor.execute(query, values)
        print(cursor._executed)
        self.connection.commit()
        # print("Query successful")
    except Error as err:
        print(f"Error: '{err}'")

def log_event(self, event_type, user_id, time, duration):
    self.execute_query(
        "INSERT INTO Events (EVENT_TYPE,SIMULATION_UUID,USER_ID,TIMESTAMP,DURATION) VALUES (%s,%s,%s,%s,%s)",
        (event_type, Properties.SIMULATION_UUID, user_id, time, duration))

def log_simulation_start(self):
    self.execute_query("INSERT INTO Simulation (UUID) VALUES (%s)", [Properties.SIMULATION_UUID])
    self.log_simulation_parameters()
```

Slika 37: Klasa DatabaseUtils – upis podataka u bazu

Graphs

Graphs klasa predstavlja komponentu aplikacije čiji je zadatak prikaz podataka i grafika vrednosti u toku izvršenja simulacije. Na slici 38 je prikazan prozor aplikacije u okviru koga je moguće pratiti informacije o broju korisnika koji pristupa sistemu, trenutnom vremenu simulacije, čekanju korisnika i upotrebljenosti sistema. Kao što se može videti, ova komponenta daje pojednostavljenu analizu performansi trenutne iteracije simulacije, koja može pomoći u stvaranju generalne slike optimalnih vrednosti parametara sistema.



Slika 38: Prozor aplikacije sa graficima koji se iscrtavaju u realnom vremenu

Iscrtavanje grafika obavlja se pomoću biblioteke Pyplot dostupne u paketu Matplotlib. Vrednosti na osnovu kojih se grafici iscrtavaju komponenti Graphs, kako je ranije navedeno, dostavlja instanca Analytics klase. Na slici 39 je prikazan konstruktor klase Graphs koji prihvata nizove podataka instancirane u okviru Analytics klase.

```
class Graphs:
    def __init__(self, canvas, main, utilization, wait_for_resource, arrivals):
        self.x1, self.y1, self.x2, self.y2 = 1000, 260, 1290, 340
        self.time = 0
        self.canvas = canvas

        self.utilization = utilization
        self.wait_for_resource = wait_for_resource
        self.arrivals = arrivals
```

Slika 39: Graphs klasa - konstruktor

Nakon instanciranja objekta Graphs klase osvežavanje prikaza obavlja se na svaki „otkucaj“ sata, definisanog u okviru main paketa, o kojem će biti reč u nastavku (Slika 40).

```
def tick(self, time):
    self.canvas.delete(self.time)
    self.canvas.delete(self.avg_utilization)
    self.canvas.delete(self.avg_resource_wait)

    self.time = self.create_text_canvas_minutes("Time = " + str(round(time, 1)), 10, 10)
    self.avg_utilization = self.create_text_canvas(
        "Avg. utilization  = " + str(self.avg_wait(self.utilization)) + "%", 10, 30)
    self.avg_resource_wait = self.create_text_canvas_minutes(
        "Avg. wait for resource = " + str(self.avg_wait(self.wait_for_resource)), 10, 50)

    self.a1.cla()
    self.a1.set_xlabel("Time")
    self.a1.set_ylabel("System utilization")
    self.a1.step([t for (t, waits) in self.utilization.items()],
                 [np.mean(waits) for (t, waits) in self.utilization.items()])

    self.a2.cla()
    self.a2.set_xlabel("Time")
    self.a2.set_ylabel("Avg. wait for resource (min)")
    self.a2.step([t for (t, waits) in self.wait_for_resource.items()],
                 [np.mean(waits) for (t, waits) in self.wait_for_resource.items()])

    self.a3.cla()
    self.a3.set_xlabel("Time")
    self.a3.set_ylabel("Arrivals")
    self.a3.bar([t for (t, a) in self.arrivals.items()], [a for (t, a) in self.arrivals.items()])

    self.data_plot.draw()
    self.canvas.update()
```

Slika 40: Iscrtavanje grafika na otkucaj sata

Može se primetiti da je za iscrtavanje pojedinih vrednosti potrebno izračunati srednju vrednost elemenata niza što se obavlja upotrebom metoda prikazanih na slici 41.

```
@staticmethod
def avg_wait(raw_waits):
    waits = [w for i in raw_waits.values() for w in i]
    return round(np.mean(waits), 1) if len(waits) > 0 else 0
```

Slika 41: Metoda za izračunavanje srednje vrednosti

3.7 Pokretanje simulacije

Pokretanje simulacije i objedinjavanje svih navedenih komponenti sistema obavlja se u okviru main skripte. U okviru ove komponente sistema vrši se odabir algoritma Brokera, kreiranje okruženja simulacije i iniciranje svih klasa sistema (Slika 42).

```
env = simpy.rt.RealtimeEnvironment(factor=1 / Properties.TIME_SPEEDUP, strict=False)
Properties.SIMULATION_UUID = str(uuid.uuid4())

analytics = Analytics()
database = DatabaseUtils()
user_scheduler = UserScheduler()

user_scheduler.real_mod()
database.log_simulation_start()

log = DisplayLog(canvas, 5, 20)
graph = Graphs(canvas, main, analytics.utilization_percent, analytics.waits_for_getting,
               analytics.arrivals)
resource_provider = ResourceProvider(env)

broker = BrokerPrepareWhenZero(log, resource_provider, user_scheduler, env)
# broker = BrokerNoPreparing(log, resource_provider, user_scheduler, env)
# broker = Broker(log, resource_provider, user_scheduler, env)

process = env.process(start_simulation(env, broker, user_scheduler))
env.process(create_clock(env))

if Properties.CONSTANT_USER_COUNT_ENABLED:
    env.run()
else:
    env.run(until=Properties.SIMULATION_DURATION_MINUTES)

main.mainloop()
```

Slika 42: Main – pokretanje simulacije

Kako je veza između komponenti sistema već objašnjena u ranijim poglavljima, ovde se nećemo ponovo vraćati na to. Zadatak main skripte jeste njihovo povezivanje u smislenu celinu. U okviru main-a definiše se uuid simulacije i inicira upis parametara u bazu podataka. Vrši se odabir željene instance Broker klase, nakon čega se registruju dva osnovna procesa definisanih funkcijama *start_simulation(...)* i *create_clock(...)*.

Kako je ranije već napomenuto, funkcija „sata“ (eng. clock) jeste da omogući iscrtavanje grafika i ispisivanje podataka tokom rada aplikacije. Ona definiše proces koji će svake sekunde obaviti poziv metode *tick()* klase Graphs i na taj način osvežiti prikaz u okviru glavnog prozora aplikacije (Slika 43).

```
def create_clock(environment):
    while True:
        yield environment.timeout(1)
        graph.tick(environment.now)
```

Slika 43: Funkcija create_clock

Funkcija *start_simulation()* (Slika 44) definiše proces koji inicira rad simulacije. U okviru ove funkcije se, sve dok se ne iscrpe svi podaci, pribavljaju vrednosti generisane u UserScheduler-u, koje definišu broj korisnika koji će pristupiti sistemu i vremenski trenutak u kojem će to učiniti. Zahtevi korisnika se registruju u okruženju kao procesi definisani *user_login()* metodom Broker klase, nakon čega se pozivom naredbe yield proces zaustavlja na kraći vremenski period pre nastavka rada. Proces se završava u trenutku kada stigne događaj kojeg emituje *end_process()* metod klase Broker (Slika 31). Ovim događajem signalizira se da su svi korisnici završili sa upotrebom resursa i da je trenutna iteracija simulacije završena.

```
def start_simulation(env: RealtimeEnvironment, broker, user_scheduler):
    user_id = 1
    next_person_id = 0
    while len(user_scheduler.INTER_ARRIVAL_TIMES) > 0 and len(user_scheduler.USERS_NUMBER) > 0:

        # unapred kreirati vektor sa vremenma dolaska, zadrzavanja i broja ljudi koji dodju
        next_arrival = user_scheduler.INTER_ARRIVAL_TIMES.pop()
        users_number = user_scheduler.USERS_NUMBER.pop()

        # Wait for the bus
        log.next_arrival(next_arrival)
        yield env.timeout(next_arrival)
        log.arrived(users_number)

        # self.analytics.register_user_login() below is for reporting purposes only
        database.log_event(EventType.USER_LOGIN.value, None, env.now, users_number)
        for user in range(users_number):
            user = User("user", user_id)
            user_id += 1
            env.process(broker.user_login(user))

            yield env.timeout(user_scheduler.TIME_BETWEEN_LOGINS.pop())

    env.process(broker.end_process())

    create_window()
```

Slika 44: Funkcija start_simulation

4. Rezultati simulacije

Istraživanje će se sprovoditi variranjem sledećih parametara:

1. Algoritam brokera
2. Vreme pripreme novih resursa i broj „radnika“ koji ih pripremaju
3. Vreme dolaska i zadržavanja korisnika

Promena varijabli sistema vrši se sa ciljem utvrđivanja optimalnih vrednosti koje će pružiti najbolje performanse sistema. Analiziraće se vreme čekanja korisnika, prosečna zauzetost sistema, kao i broj resursa koji je upotrebljen za zadovoljavanje svih zahteva.

Prva grupa testova koristiće algoritam koji kreiranje novih resursa započinje tek kada broj spremnih resursa padne na nula. U drugoj grupi će se koristiti algoritam koji unapred priprema resurse, sa ciljem da se izbegne situacija u kojoj nema spremnih resursa za dodelu korisniku. Treću grupu predstavlja test sistema koji ne koristi nijedan algoritam, sa ciljem da se pokažu negativni efekti ovakvog pristupa.

Na osnovu znanja stečenog u prethodnim primerima, u poslednjoj grupi testova, biće primenjeni oni parametri koji daju najbolje performanse. Broj spremnih resursa dostupnih na početku rada biće određen na osnovu ukupnog broja resursa koji su bili potrebni za rad simulacije u primerima od 1 do 3.

4.1. Broker bez blagovremene pripreme resursa

Prvi tip algoritma čije performanse se ispituju pripremi resursa pristupa kada broj spremnih resursa dostigne 0.

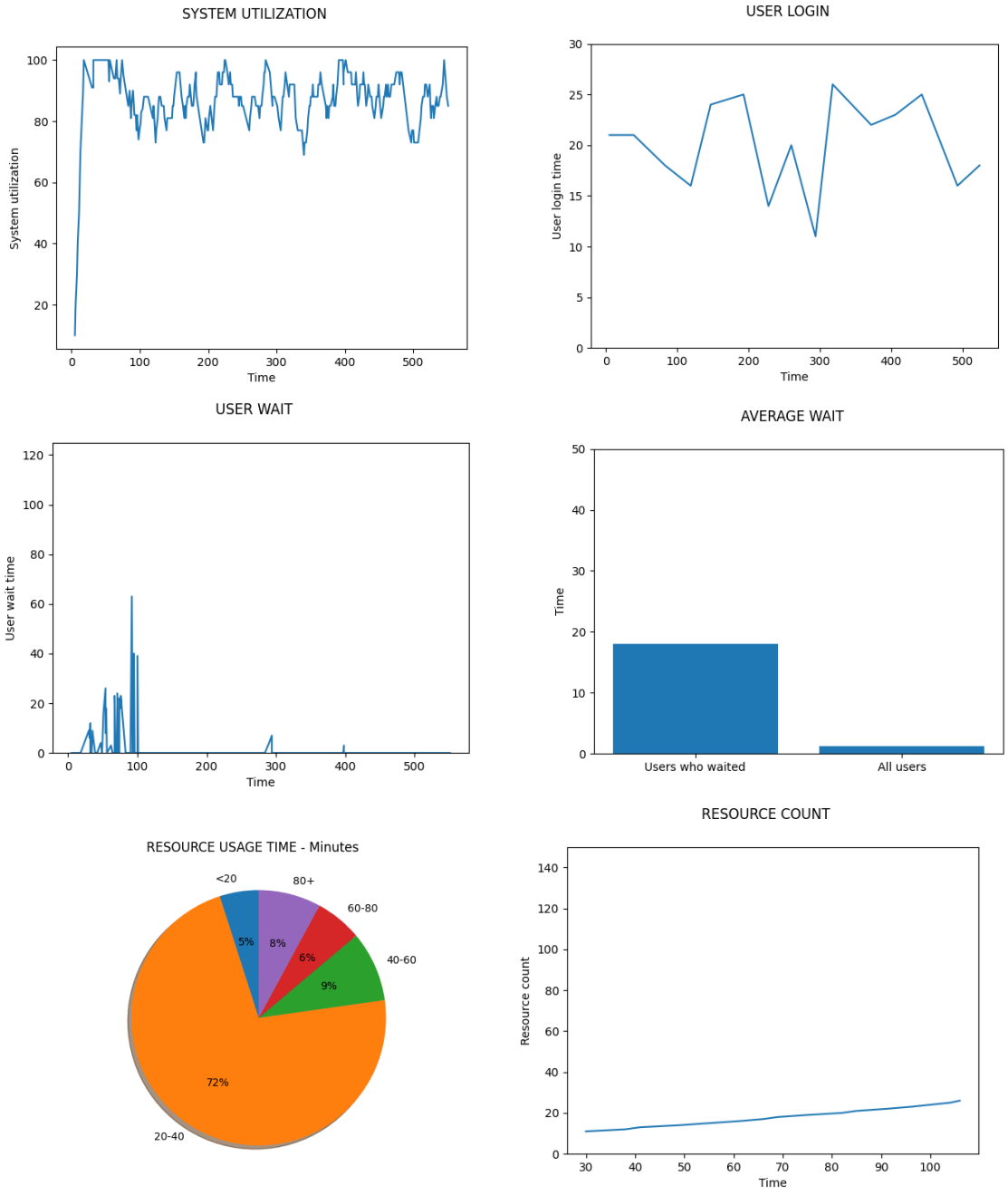
Primer 1.1

- *Prosečan broj korisnika:* 10 na 5 minuta
- *Standardno vreme korišćenja*
- *Prosečno vreme potrebno za pripremu resursa:* 5 minuta
- *Ukupan broj korisnika:* 300

Slika 45: Rezultati testiranja – primer 1.1

Primer 1.2

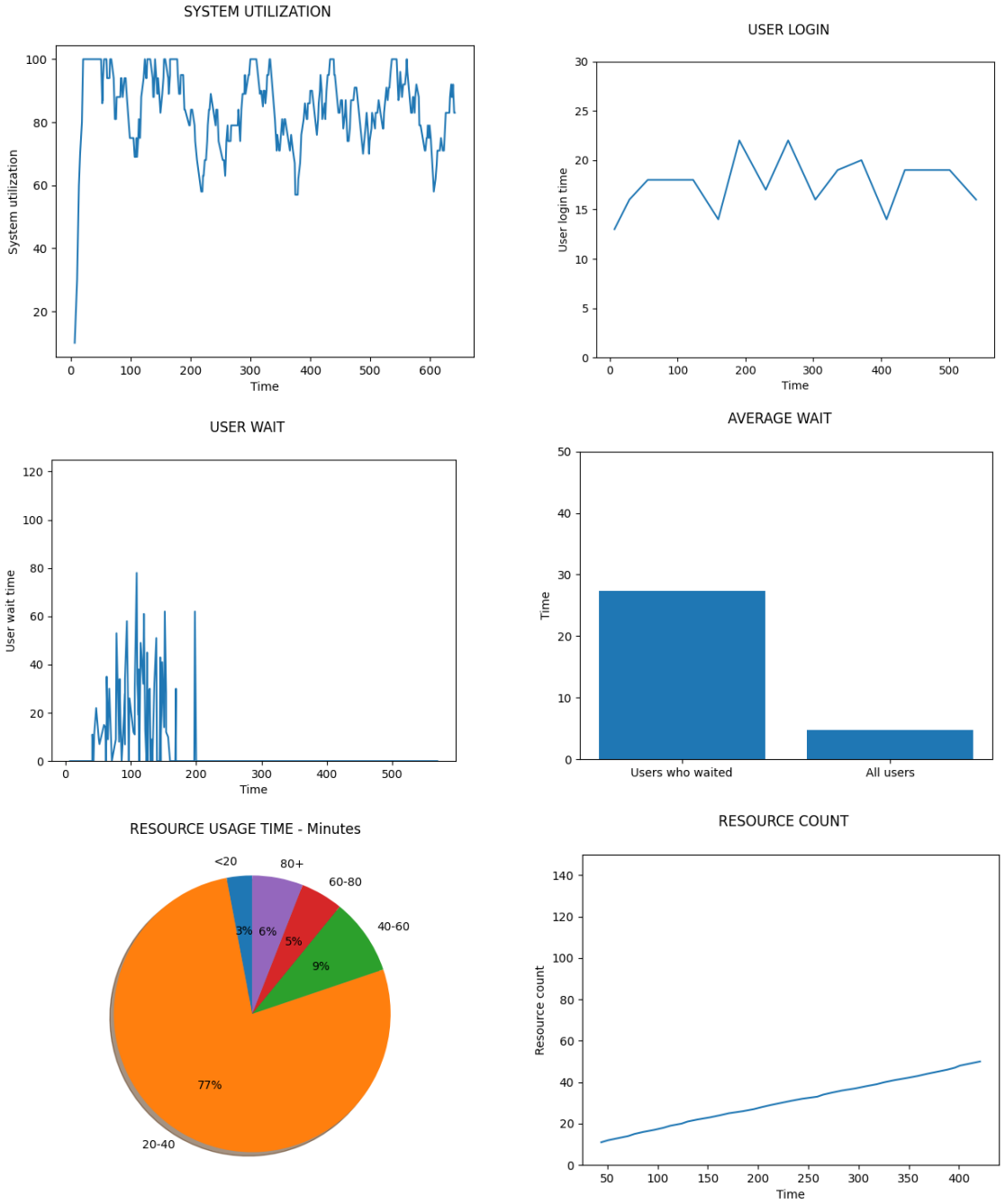
- *Prosečan broj korisnika*: 20 na 5 minuta
- *Standardno vreme korišćenja*
- *Prosečno vreme potrebno za pripremu resursa:* 5 minuta
- *Ukupan broj korisnika:* 300



Slika 46: Rezultati testiranja – primer 1.2

Primer 1.3

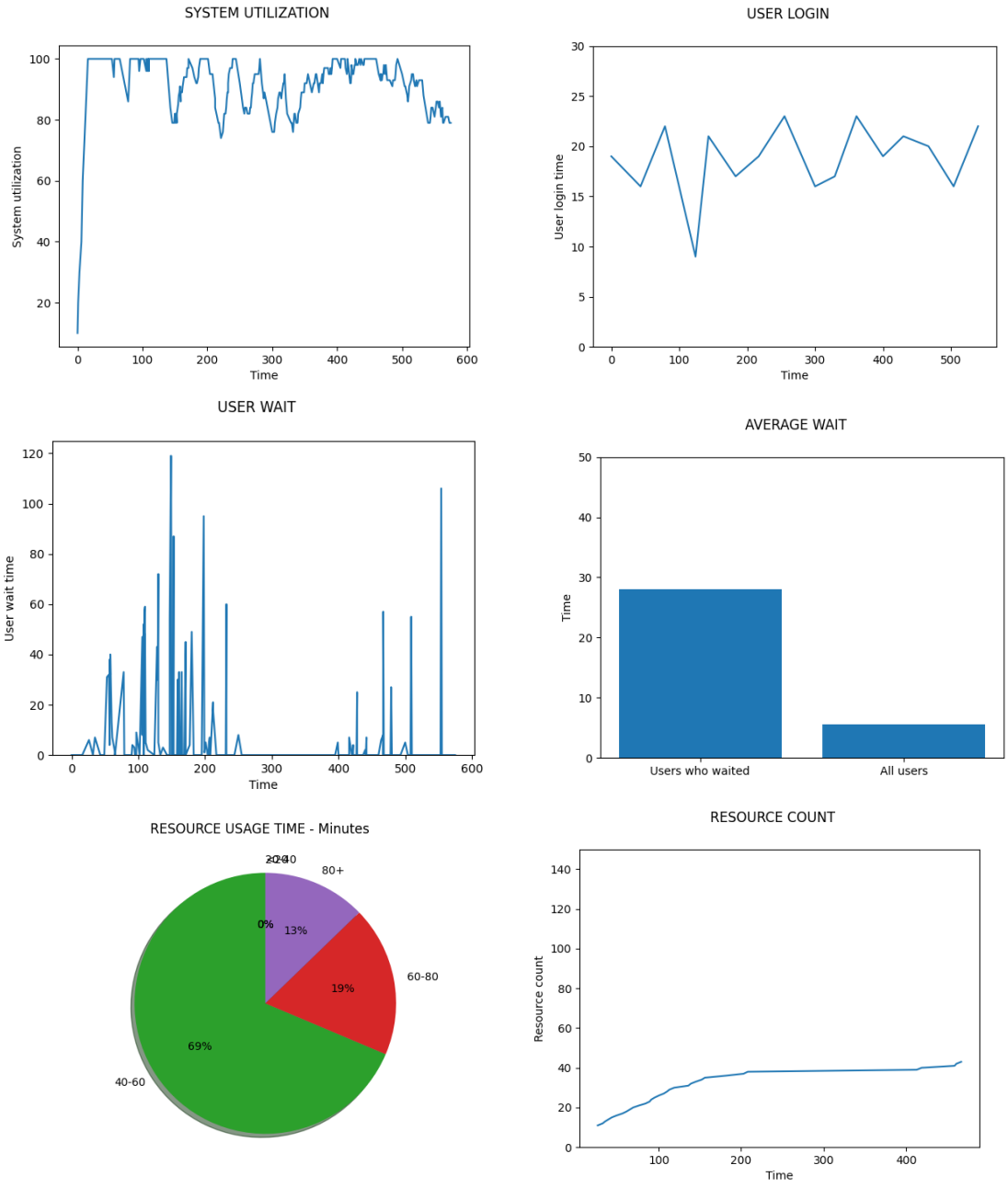
- *Prosečan broj korisnika:* 20 na 5 minuta
- *Standardno vreme korišćenja*
- *Prosečno vreme potrebno za pripremu resursa:* 10 minuta
- *Ukupan broj korisnika:* 300



Slika 47: Rezultati testiranja – primer 1.3

Primer 1.4

- *Prosečan broj korisnika:* 20 na 5 minuta
- *Duže vreme korišćenja* - veće opterećenje sistema
- *Prosečno vreme potrebno za pripremu resursa:* 5 minuta
- *Ukupan broj korisnika:* 300

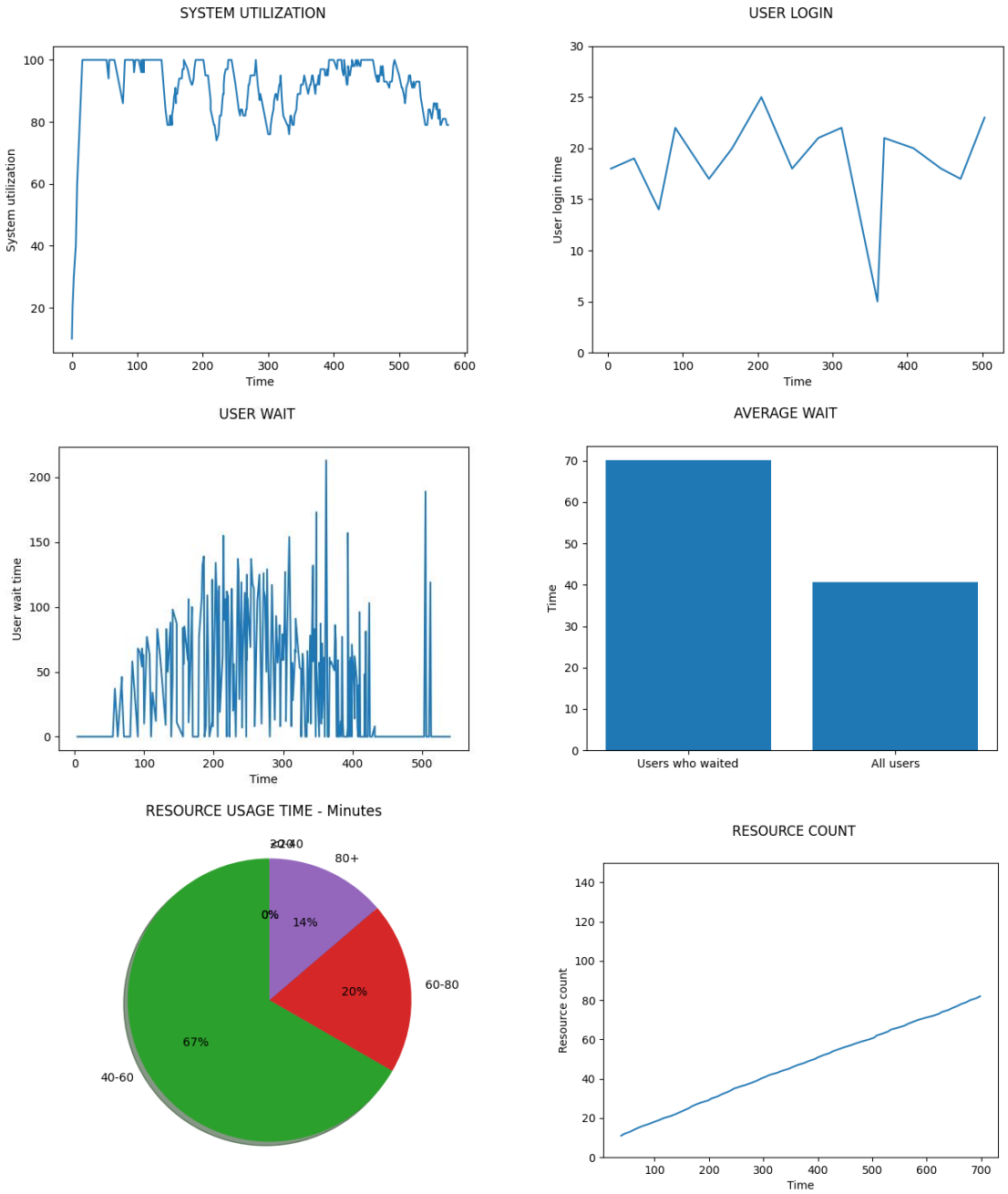


Slika 48: Rezultati testiranja – primer 1.4

Primer 1.5

- *Prosečan broj korisnika:* 20 na 5 minuta
- *Duže vreme korišćenja* - veće opterećenje sistema
- *Prosečno vreme potrebno za pripremu resursa:* 10 minuta
- *Ukupan broj korisnika:* 300

Napomena: Obratiti pažnju na skalu prosečnog vremena čekanja koja ja znato veća u odnosu na ostale primere



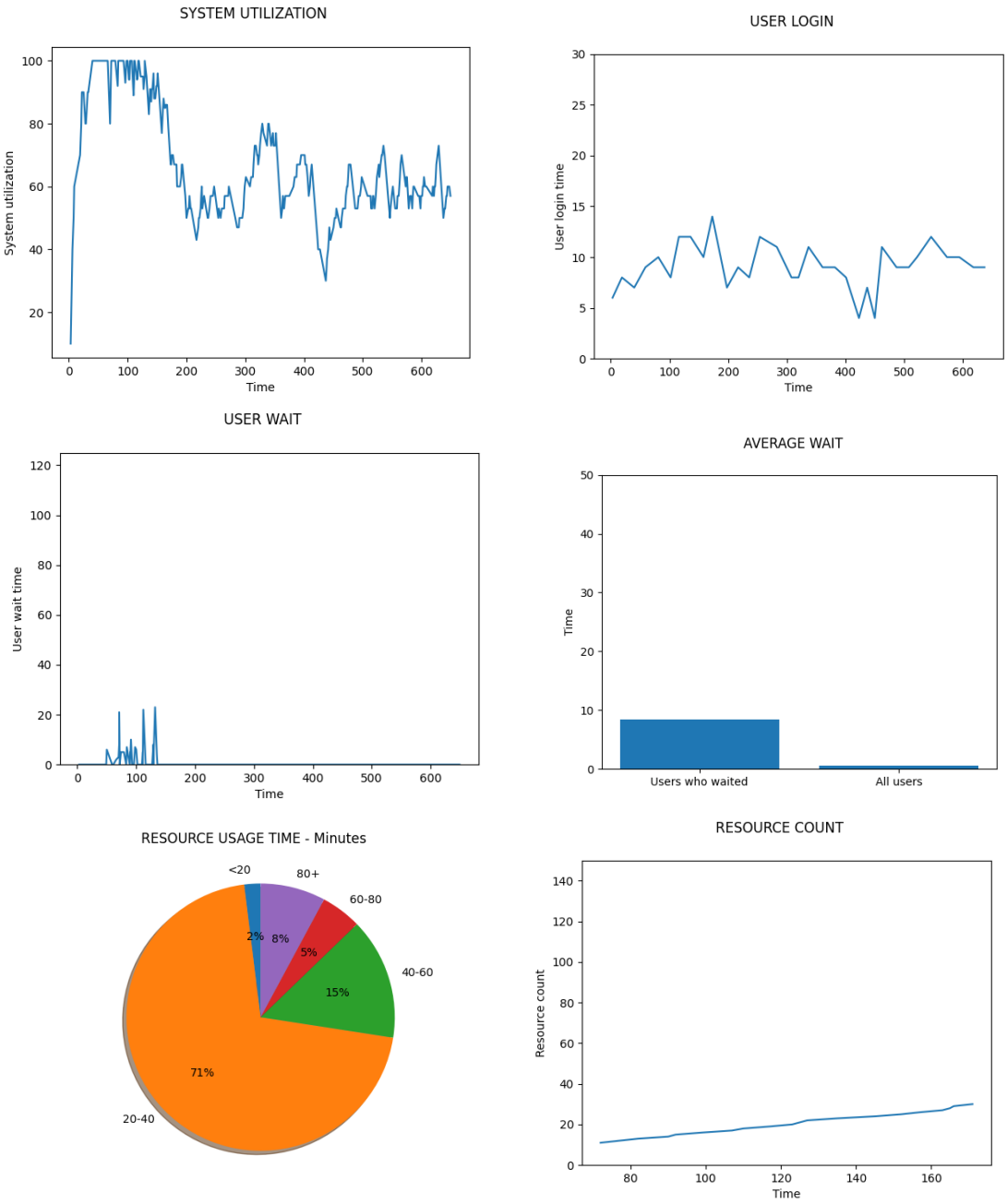
Slika 49: Rezultati testiranja – primer 1.5

4.2. Blagovremena priprema resursa

U drugoj grupi testova koristiće se algoritma koji kreiranje resursa započinje pre nego što se utroše svi raspoloživi resursi.

Primer 2.1

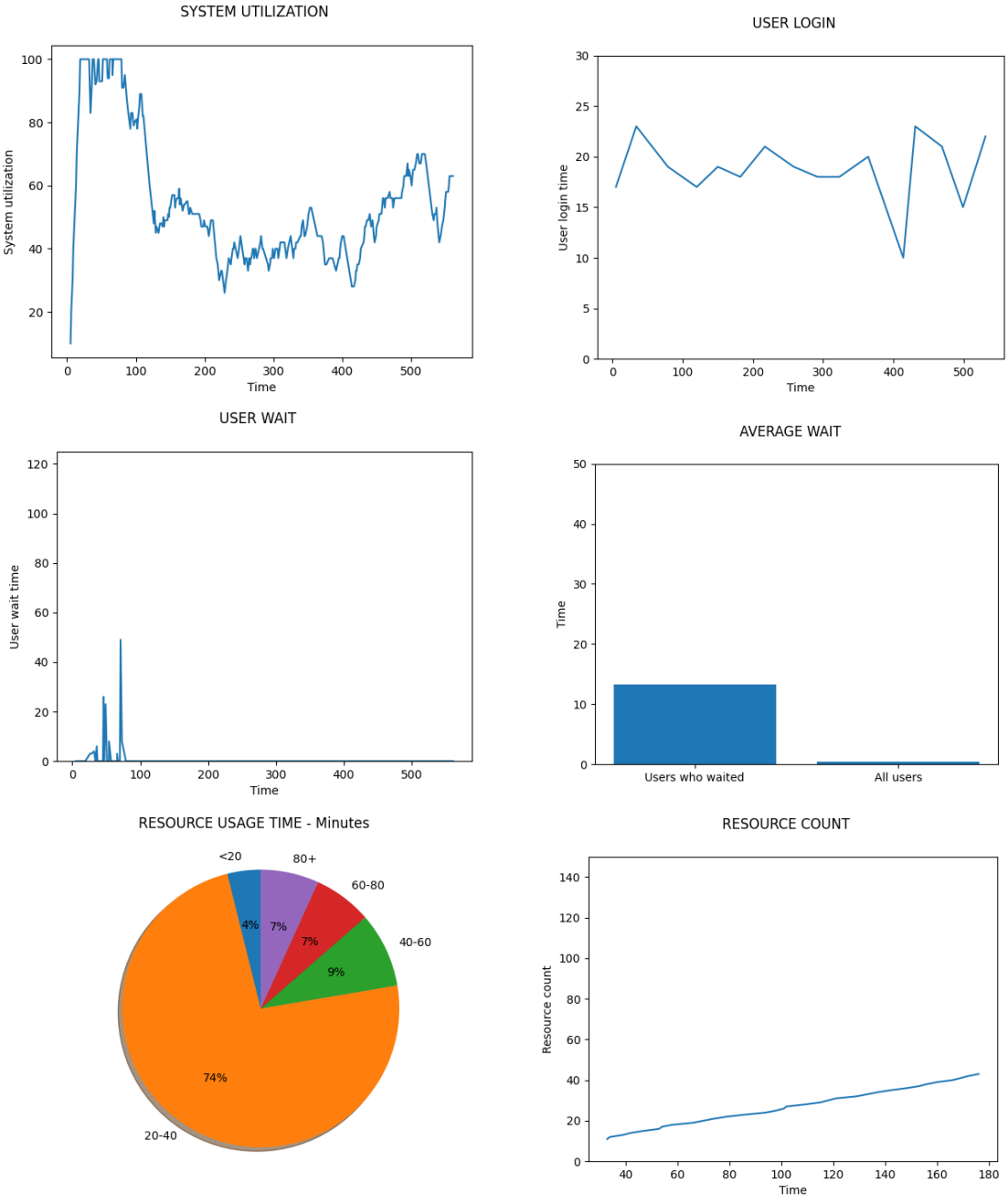
- *Prosečan broj korisnika:* 10 na 5 minuta
- *Standardno vreme korišćenja*
- *Prosečno vreme potrebno za pripremu resursa:* 5 minuta
- *Ukupan broj korisnika:* 300



Slika 50: Rezultati testiranja – primer 2.1

Primer 2.2

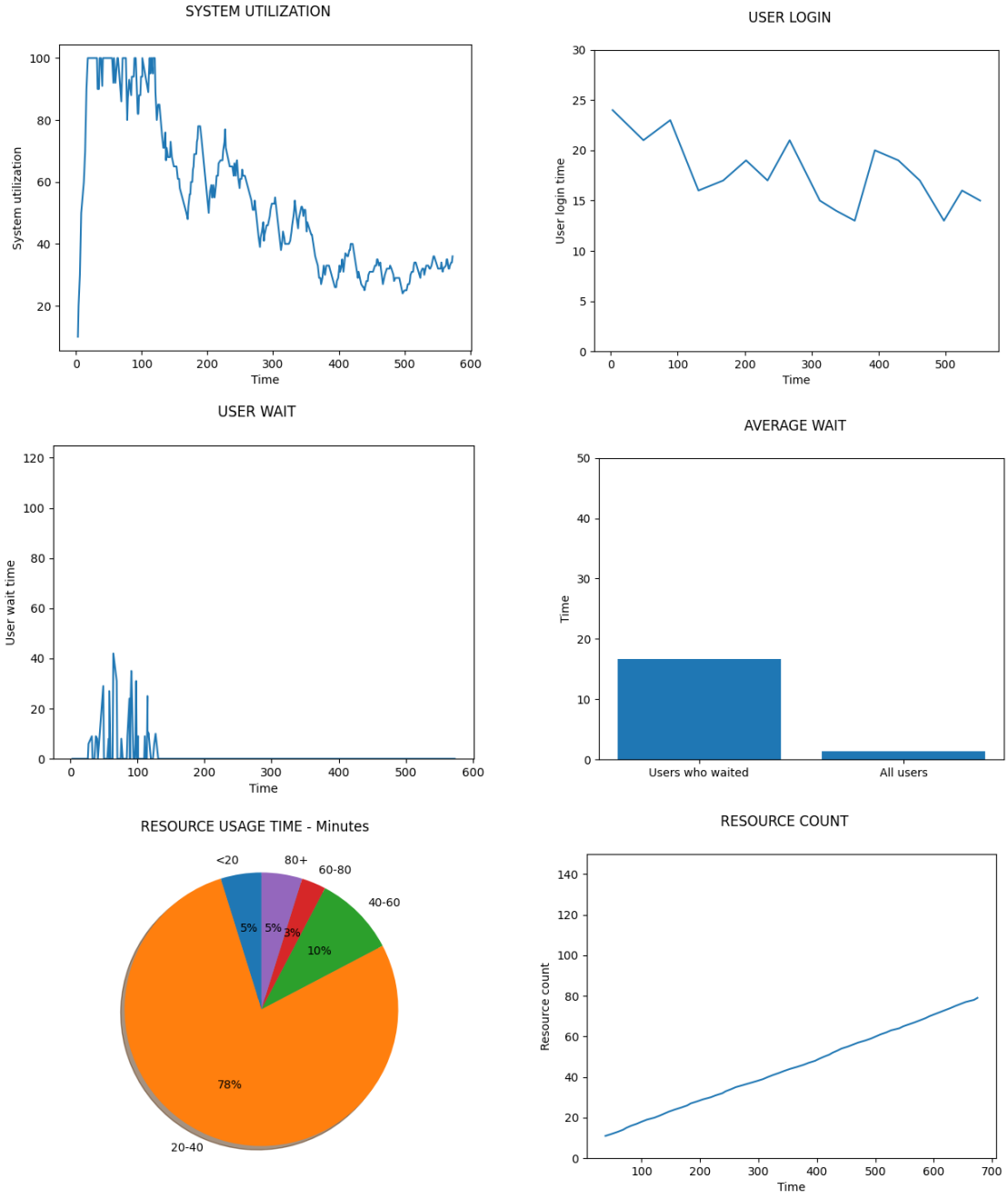
- *Prosečan broj korisnika:* 20 na 5 minuta
- *Standardno vreme korišćenja*
- *Prosečno vreme potrebno za pripremu resursa:* 5 minuta
- *Ukupan broj korisnika:* 300



Slika 51: Rezultati testiranja – primer 2.2

Primer 2.3

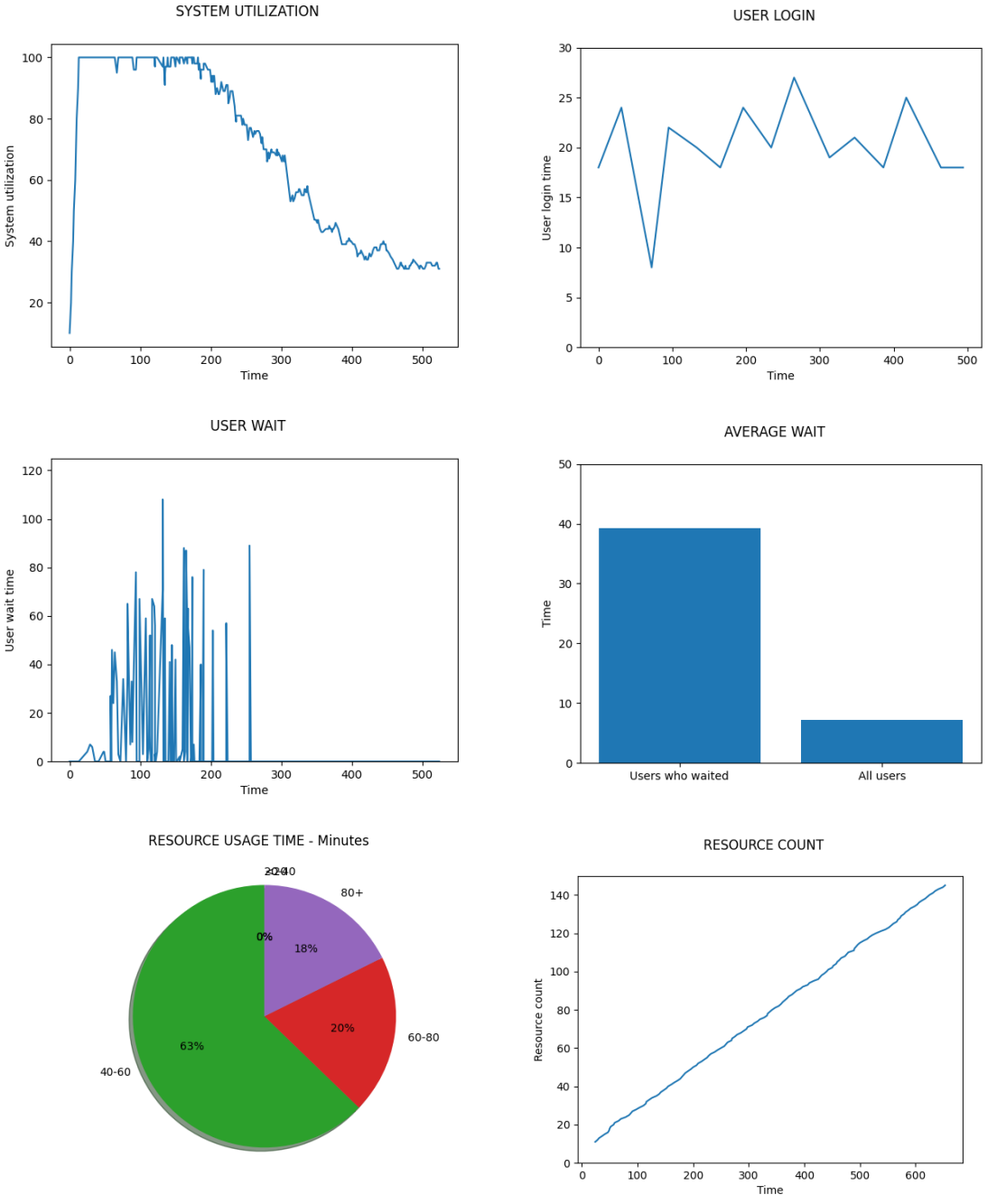
- *Prosečan broj korisnika:* 20 na 5 minuta
- *Standardno vreme korišćenja*
- *Prosečno vreme potrebno za pripremu resursa:* 10 minuta
- *Ukupan broj korisnika:* 300



Slika 52: Rezultati testiranja – primer 2.3

Primer 2.4

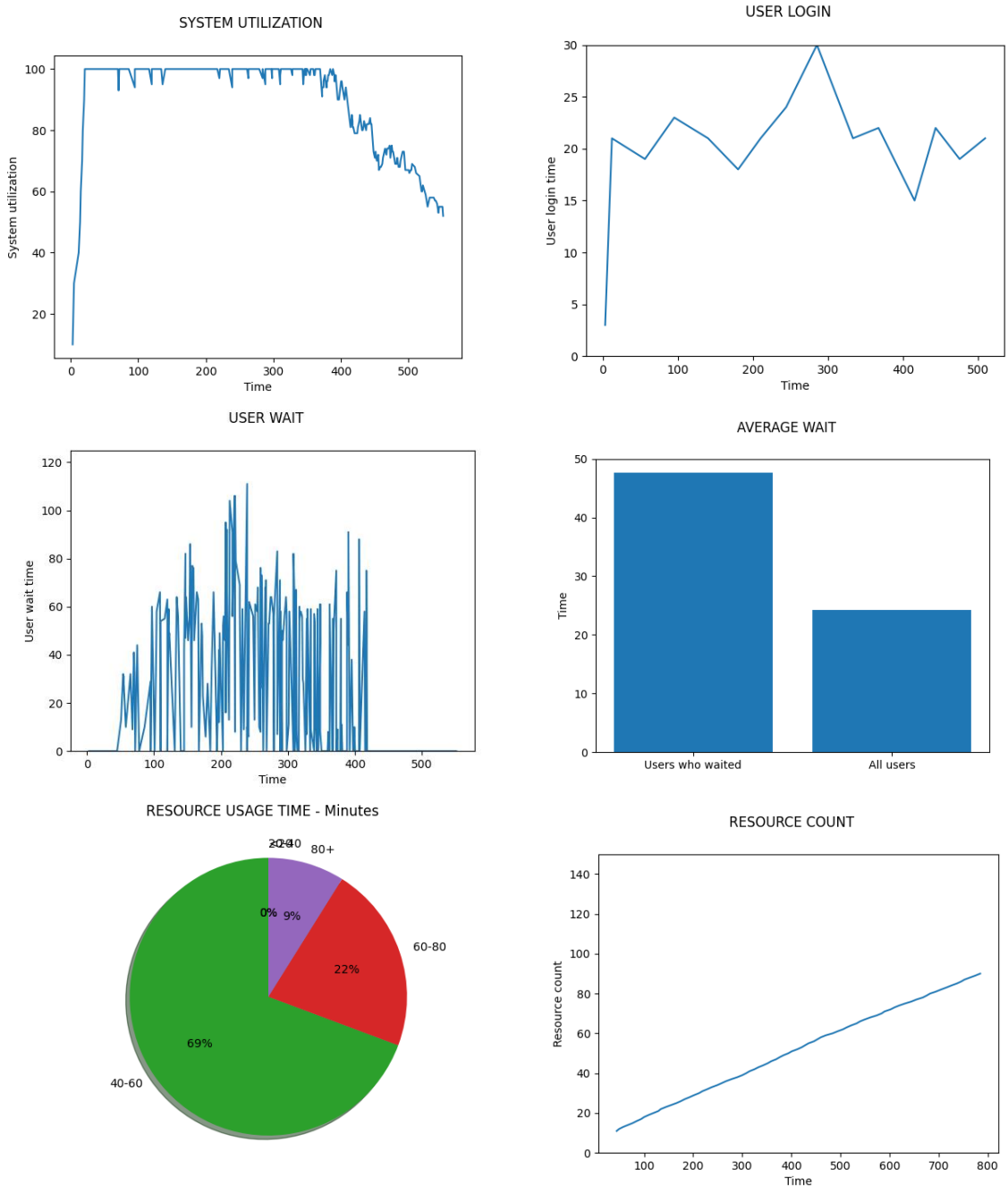
- *Prosečan broj korisnika:* 20 na 5 minuta
- *Duže vreme korišćenja* - veće opterećenje sistema
- *Prosečno vreme potrebno za pripremu resursa*: 5 minuta
- *Ukupan broj korisnika:* 300



Slika Slika 53: Rezultati testiranja – primer 2.4

Primer 2.5

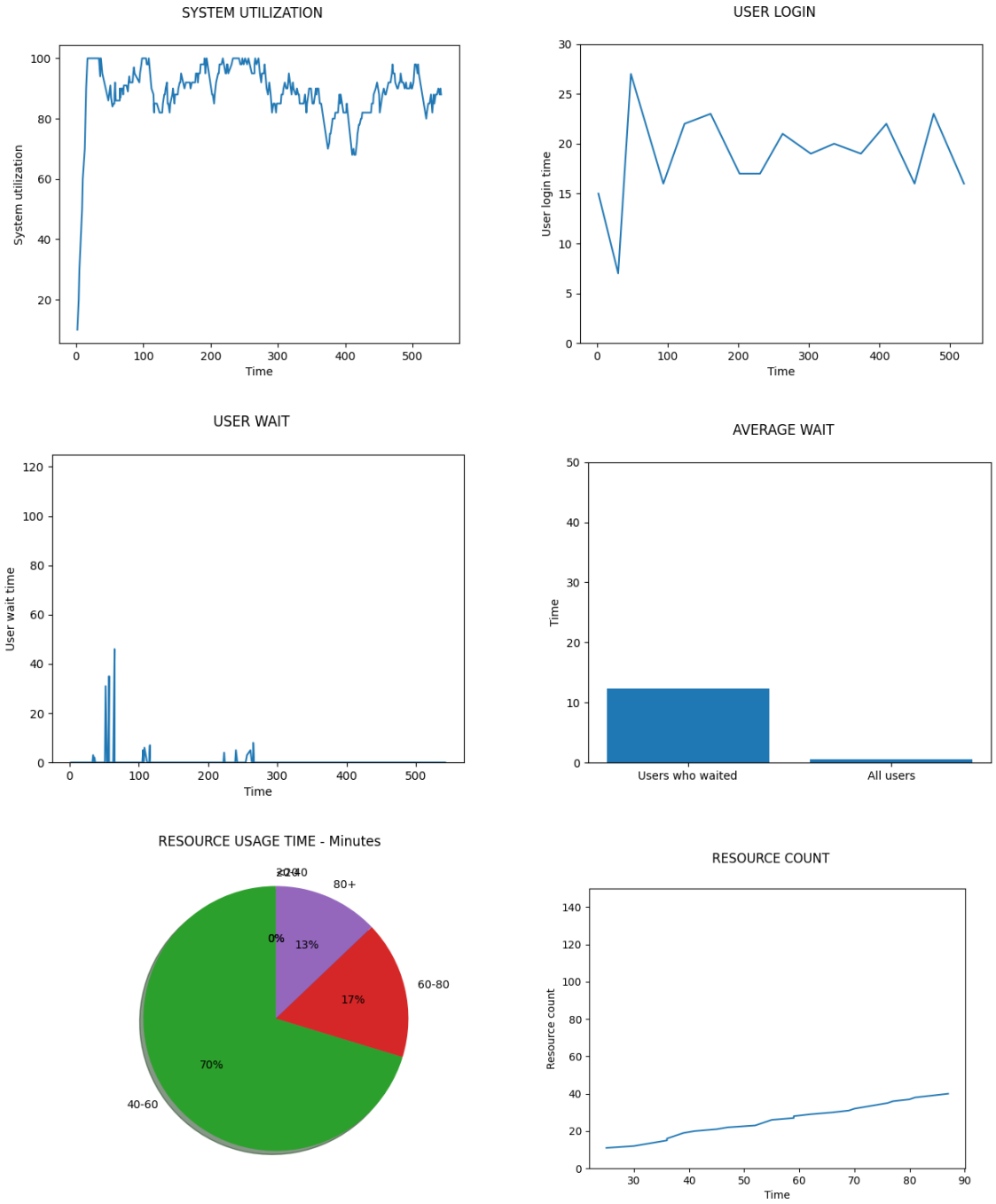
- *Prosečan broj korisnika:* 20 na 5 minuta
- *Duže vreme korišćenja* - veće opterećenje sistema
- *Prosečno vreme potrebno za pripremu resursa:* 10 minuta
- *Ukupan broj korisnika:* 300



Slika 54: Rezultati testiranja – primer 2.5

Primer 3

- *Prosečan broj korisnika:* 20 na 5 minuta
- *Duže vreme korišćenja* - veće opterećenje sistema
- *Prosečno vreme potrebno za pripremu resursa:* 5 minuta
- *Više od jednog "radnika" koji kreira resurse* - broj povećan na 2
- *Ukupan broj korisnika:* 300



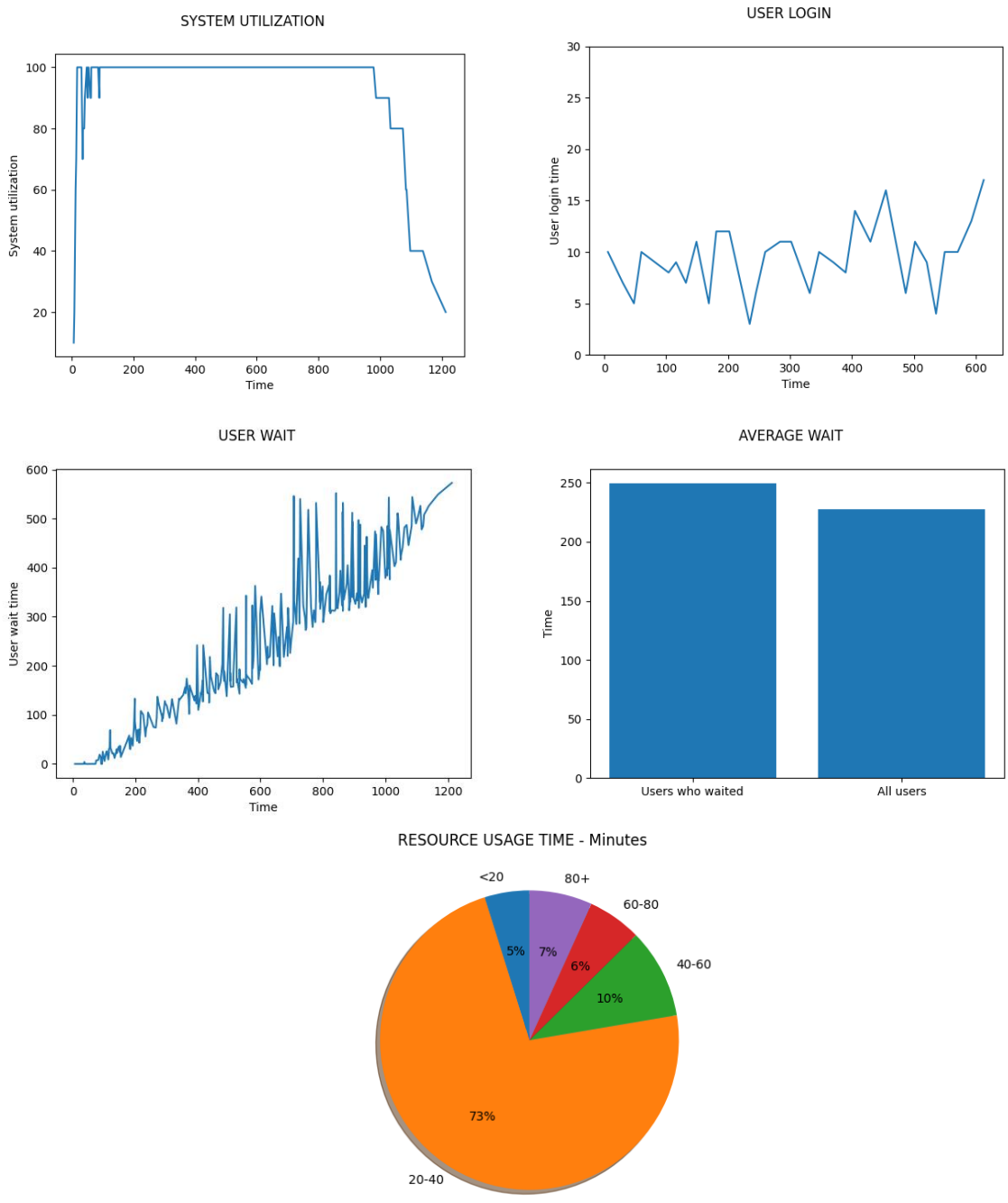
Slika 55: Rezultati testiranja – primer 3

4.3. Bez pripreme resursa

Primer 4

- *Prosečan broj korisnika:* 10 na 5 minuta
- *Standardno vreme korišćenja*
- *Prosečno vreme potrebno za pripremu resursa:* 5 minuta
- *Ukupan broj korisnika:* 300

Napomena: Obratiti pažnju na skalu prosečnog vremena čekanja koja ja znato veća u odnosu na ostale primere



Slika 56: Rezultati testiranja – primer 4

4.4. Optimalni algoritam i parametri

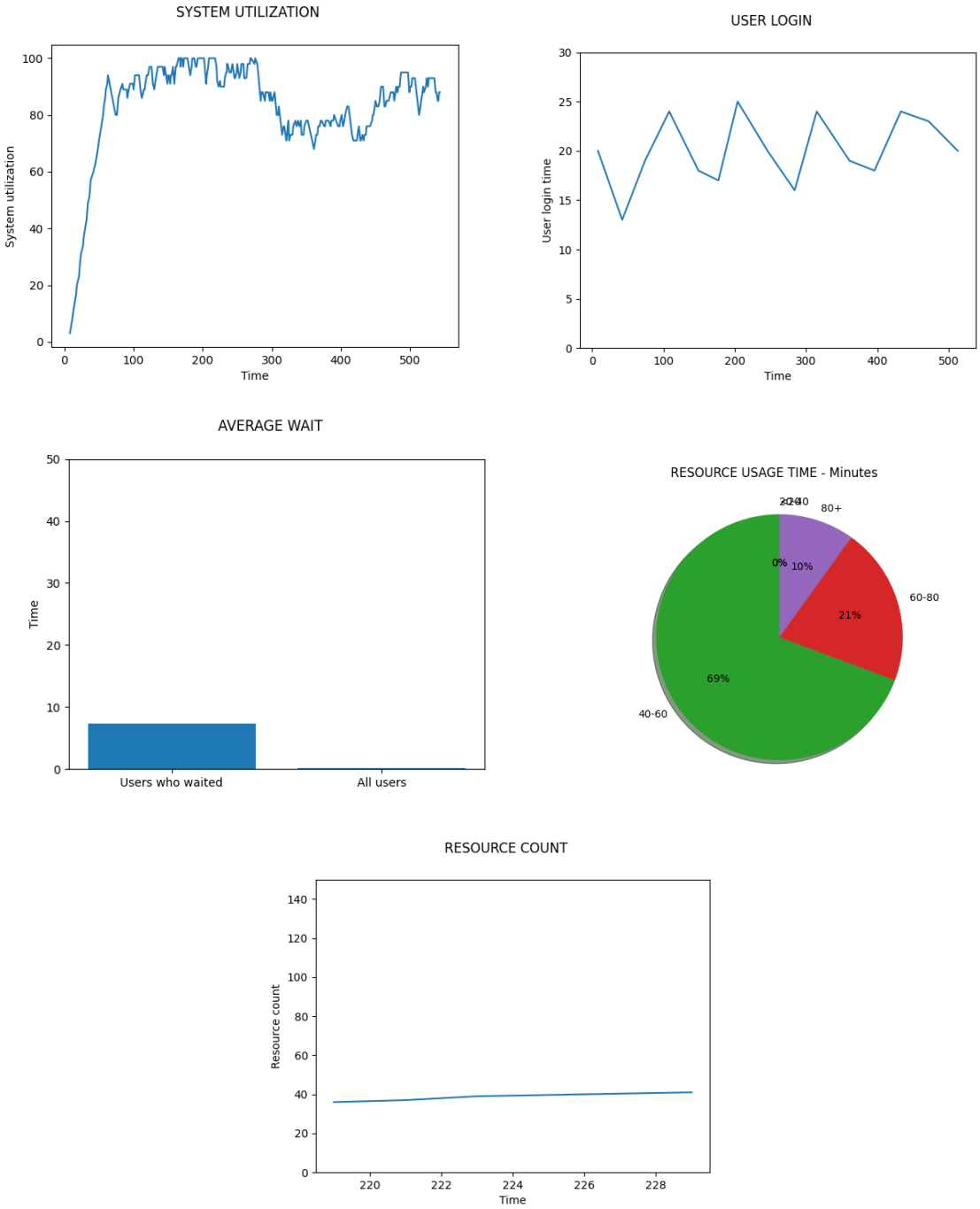
Primer 5.1

- *Prosečan broj korisnika:* 10 na 5 minuta
- *Standardno vreme korišćenja*
- *Prosečno vreme potrebno za pripremu resursa:* 5 minuta
- *Više od jednog "radnika" koji kreira resurse* - broj povećan na 2
- *Početni broj resursa na osnovu prethodnih simulacija* - 35
- *Ukupan broj korisnika:* 300

Slika 57: Rezultati testiranja – primer 5.1

Primer 5.2

- *Prosečan broj korisnika:* 20 na 5 minuta
- *Duže vreme korišćenja* - veće opterećenje sistema
- *Prosečno vreme potrebno za pripremu resursa:* 5 minuta
- *Više od jednog "radnika" koji kreira resurse* - broj povećan na 2
- *Početni broj resursa na osnovu prethodnih simulacija* - 35
- *Ukupan broj korisnika:* 300

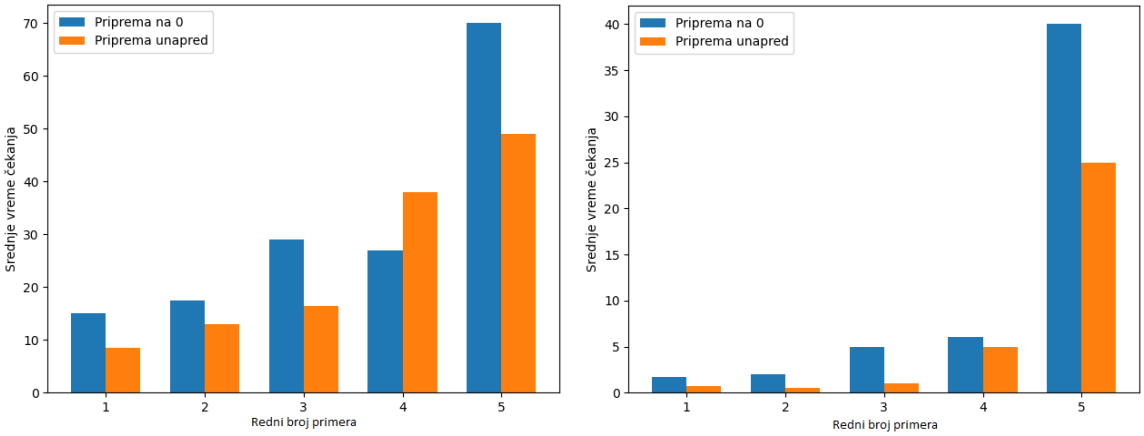


Slika 58: Rezultati testiranja – primer 5.2

5. Analiza rezultata simulacije

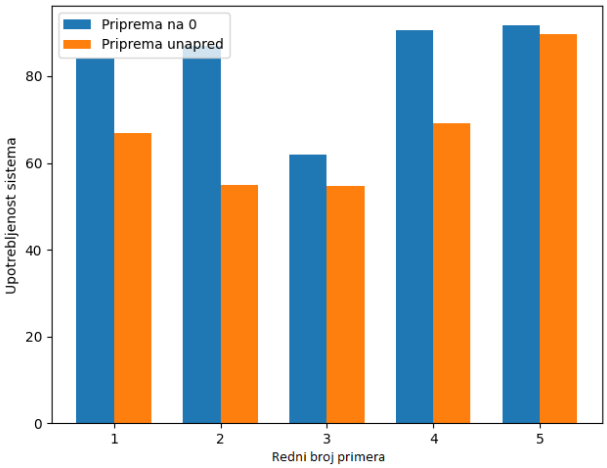
Nakon sprovedenog istraživanja moguće je stvoriti poprilično jasnu sliku osobina koje sistem za dodelu resursa treba da poseduje. U nastavku ćemo se fokusirati na analizu rezultata dobijenih upotrebom prva dva algoritma planiranja resursa, obzirom da sistem koji ne bi koristio ovakav algoritam imao neprihvatljive performanse. Ovo je pokazano primerom 4, u kojem se vidi da takav sistem ne bi mogao da ogovori ni na osnovne zahteve korisnika, osim u slučaju da se svi raspoloživi resursi u svakom trenutku održavaju u spremnom stanju, što je iz praktičnih razloga gotovo neizvodljivo.

Upoređivanjem vremena čekanja u zavisnosti od upotrebljenog algoritma, lako se može zaključiti da priprema resursa unapred daje značajno poboljšanje u performansama sistema. Do ovog zaključka dolazi se analizom testova sprovedenih u primerima 1 i 2. Na prvoj slici je prikazana razlika u prosečnom vremenu čekanja za sve korisnike koji su čekali na dodelu resursa, dok je na drugoj upoređeno prosečno vreme čekanja svih korisnika.



Slika 59: Analiza vremena čekanja

Ukoliko uporedimo upotrebljenost resursa sistema, takođe dolazimo do sličnih zaključaka. Naime, u slučaju algoritma koji ne priprema resurse unapred ovaj broj je znatno bliži stoprocentnoj upotrebi resursa sistema. Čak i u slučaju da se i kod drugog sistema javi ovaj skok u procentu upotrebljenosti on znatno brže opadne i dolazi do balansa u broju resursa i broja zahteva.



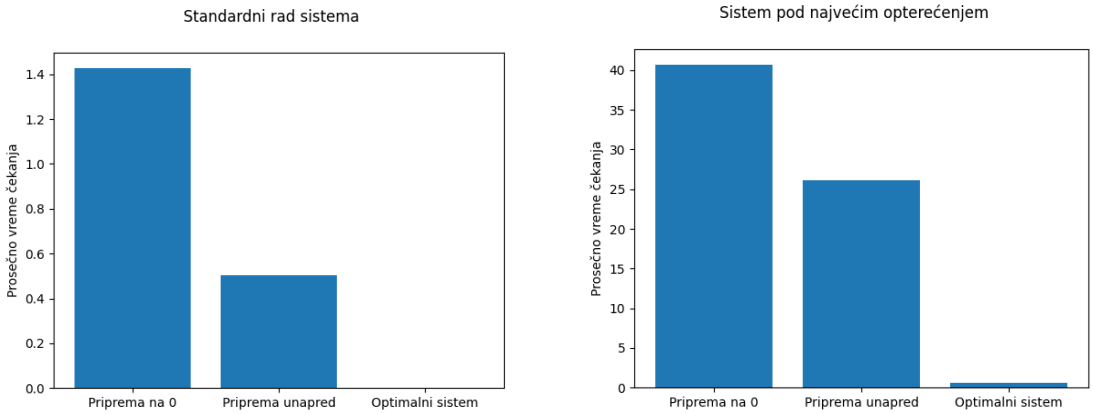
Slika 60: Analiza vremena čekanja

Međutim, ono što se u oba slučajeva može navesti kao mana jeste nemogućnost sistema da obradi veći broj zahteva, što se može videti na osnovu znatnog skoka u vremenima čekanja i upotrebljenosti resursa kod oba sistema. Jasno je da su prosečno čekanje od 40 minuta i konstanta gotovo stoprocentna opterećenost neprihvatljive za bilo koji sistem koji će se upotrebljivati u svakodnevnom životu. Dramatičan skok u vremenima čekanja dolazi kao posledica činjenice da je u svakom od primera korišćen po jedan "radnik" koji će kreirati resurse.

Ono što se može zaključiti na osnovu upoređivanja 5. i 3. podprimera sa ostalim primerima iz grupe jeste da je vreme pripreme resursa od presudnog značaja. S toga se dolazi do zaključka da je neophodno uložiti u taj aspekt sistema. Pored samog vremena čekanja moguća je i "paralelizacija" procesa uvođenjem više od jednog "radnika" za kreiranje novih resursa. Gore navedeno se u stvarnom sistemu postiže korišćenjem vise računarskih jezgra i upotrebom procesora zadovoljavajućih performansi.

Navedena zapažanja dovela su do konfiguracije koja se može videti u primeru 5 u kojem se korišćenjem algoritma za kreiranje resursa unapred, upotrebom više "radnika" (barem dva) i povećanjem broja resursa koji su spremni u trenutku pokretanja sistema dolazi do sistema čije su performanse zadovoljavajuće kako u svakodnevnom radu, tako i u trenucima velikog opterećenja.

Performanse sistema čiji su parametri određeni analizom rezultata simulacija najbolje se mogu videti poređenjem prosečnog vremena čekanja u odnosu na sisteme iz primera 1 i 2.



Slika 61: Analiza vremena čekanja

5.1. Zaključci istraživanja

Kako je to analiza rezultata jasno pokazala, konfiguracija sistema prikazana u primeru 5 daje optimalne performanse sistema. Ovaj primer koristi algoritam blagovremene pripreme resursa, koji podrazumeva održavanje određenog skupa spremnih resursa koji se mogu dodeliti korisniku na korišćenje. Takođe, veoma važan aspekt ovog algoritma predstavlja trenutak u kojem sistem daje signal za kreiranje novih resursa. Naime, sa pripremom se kreće tek kada se detektuje veće opterećenje sistema i u tom trenutku se započinje priprema novog broja resursa, pri čemu se proj resursa koji će biti pripremljen određuje primenom algoritma opisanog u poglavju 3.5, slika 29.

Vrednosti za parametre sisteme koje se koriste u ovom primeru dobijene su nakon sprovedenog istraživanja i analize izlaznih podataka. Na osnovu toga moguće je izvesti zaključak da je za grupu od 300 studenata na početku rada sistema dovoljno pripremiti 35 resursa kako bi se, čak i u slučaju velikog opterećenja, izbegao zastoj u radu. Pored toga, potrebno je obezbediti određen paralelizam u radu za servis koji će obavljati pripremu resursa, obzirom da je sprovedeno istraživanje jasno pokazalo da on u slučaju većeg opterećenja lako postaje usko grlo sistema.

Još jedan parametar koji direktno utiče na vreme čekanja korisnika jeste vreme pripreme resursa. Stoga je jasno da je ovaj proces potrebno maksimalno optimizovati. U primeru koji je prilikom simulacije dao najbolje performanse vreme pripreme iznosilo je u proseku 5 minuta. Na osnovu rezultata u izloženom primeru svaku vrednost, blisku ovoj, možemo smatrati adekvatnom i primenljivom u realnom sistemu.

6. Zaključak

Na kraju, možemo se osvrnuti na rezultate aplikacije koja je razvijena u ovom radu i mogućnosti njihove primene. Primenom simulacije dobili smo uvid u način funkcionisanja sistema za dodelu računarskih resursa i diskretne veze koje postoje između njegovih komponenti. Primena programskog jezika Python, i konkretno SimPy framework-a dodatno je olakšalo i ubrzalo proces implementacije. Na osnovu rezultata istraživanja moguće je izvući jasne zaključke koji bi omogućili pouzdanu i adekvatnu konfiguraciju realnog sistema. Obzirom da je analiza urađena na uzorku od 300 studenata, svi zaključci do kojih se došlo upravo se odnose na naveden slučaj korišćenja, pri čemu je opravdano i očekivanje da će sistem adekvatno odgovoriti i u drugačijim uslovima. Kako je to analiza jasno pokazala, algoritam koji podrazumeva pripremu resursa unapred, uz održavanje određenog skupa spremnih resursa daje optimalne performanse. Pored toga, važan aspekt sistema jesu „radnici“ koji pripremaju resurse, pri čemu je važno implementirati određeni paralelizam u njihovom radu, što podrazumeva postojanje najmanje dva radnika koja će obavljati ovaj posao. Navedena podešavanja, uz početan skup od 35 spremnih resursa i resurse čije je vreme pripreme blisko vrednosti od 5 minuta, daju optimalnu konfiguraciju sistema. Ono što ostaje van opsega ovog rada jeste testiranje navedene konfiguracije u realnom sistemu, kako bi se i empirijski utvrdili dobijeni rezultati.

Obzirom da je pristup rešavanju problema opisan u ovom primenljiv čak i u slučaju najkompleksnihih sistema, jasan je rast popularnosti simulacija diskretnih događaja. One daju novi pogled na sistem, čak i u slučaju da nema potrebe za testiranjem novih ideja, i organizacijama ukazuju na slabosti i pružaju alate za pronalaženje rešenja problema. Rezultati dobijeni ovim istraživanjem jasno prikazuju prednosti korišćenja simulacije diskretnih događaja, obzirom da je konfiguracija sistema određenje bez uvođenja istraživanja u realnom životu. Izbegnute su neprijatnosti i neugodne situacije koje testiranje u stvarnom sistemu donosi pri čemu je pretpostavka da će primena izloženog podešavanja u realnom sistemu doneti slične performanse, ne samo prirodna, već i očekivana. Konačno, na osnovu svega izloženog, možemo zaključiti da navedene osobine čine simulaciju nezamenljivom u društvu koje teži sve većoj optimizaciji rada, pružajući instrumente za povećanje fluidnosti u radu i bolju komunikaciju svih komponenti sistema.

7. Literatura

1. Lawrence Leemis, Steve Park, *Discrete-event simulation a first course,* New York, Williamsburg: The College of William & Mary, December 2004
2. Jonathan Karnon, James Stahl, Alan Brennan, Jaime Caro, Javier Mar, Jörgen Möller, „Modeling using Discrete Event Simulation: A Report of the ISPOR-SMDM Modeling Good Research Practices Task Force-4“, Value in Health, vol. 15, issue 6, pp 821-827, September–October 2012. Dostupno na: https://www.sciencedirect.com/science/article/pii/S1098301512016580
3. George S. Fishman, *Discrete-Event Simulation: Modeling, Programming, and Analysis,* Berlin: Springer, 2001
4. Will Campbell, „Understanding Discrete-Event Simulation“, MathWorks. Dostupno na: https://www.mathworks.com/videos/series/understanding-discrete-event-simulation.html [Pristupljeno: April 2021]
5. Tijana Mandić, „Napredni pristup merenja operativnog rizika“, Master rad, Dep. za matematiku i informatiku, Prirodno matematički fakultet, Univerzitet u Novom Sadu, 2013
6. Nevena Dugandzija, „Slučajne promenljive“, Literatura sa predavanja, Prirodno matematički fakultet, Univerzitet u Novom Sadu. Dostupno na: https://personal.pmf.uns.ac.rs/nevena.dugandzija/wp-content/uploads/sites/42/2016/11/Slucajne_promenljive.pdf [Pristupljeno: Maj 2021]
7. *SimPy Overview*, Zvanična dokumentacija. Dostupno na: https://simpy.readthedocs.io/en/latest/ [Pristupljeno: Mart 2021]
8. *Python documentation,* Zvanična dokumentacija. Dostupno na: https://docs.python.org/3/ [Pristupljeno: Mart 2021]
9. Kevin P. Brown, „Simulating Real-Life Events in Python with SimPy“, Dattivo, London. Dostupno na: https://www.dattivo.com/simulating-real-life-events-in-python-with-simpy/ [Pristupljeno: Mart 2021]