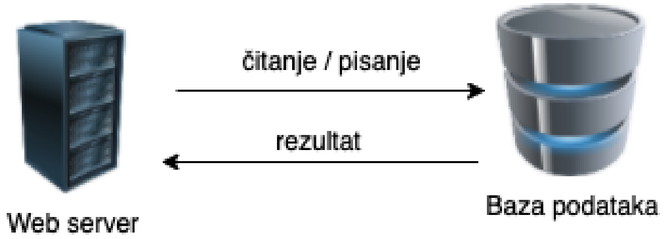
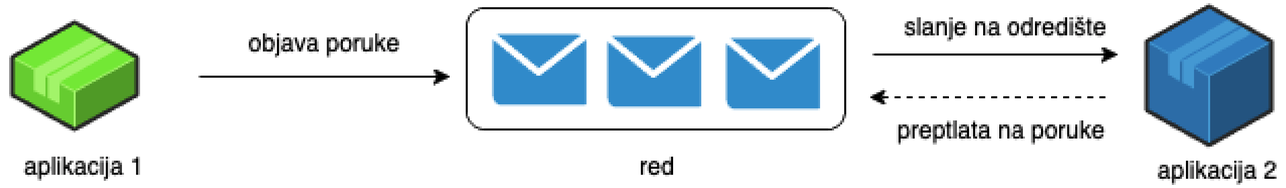
SERVERSKE ARHITEKTURE

SCENARIJI TOKOVA PODATAKA

TRI NAJČEŠĆA SCENARIJA

- Direktna komunikacija klijent <-> server kroz poziv servisa
- Serverska aplikacija <-> Baza podataka
- Asinhrona komunikacija razmenom poruka preko reda poruka (message queue)

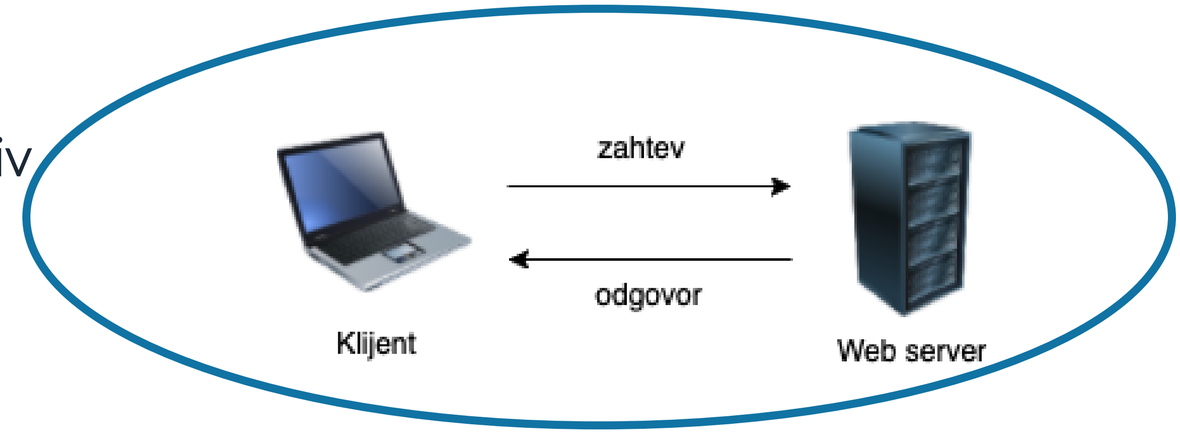


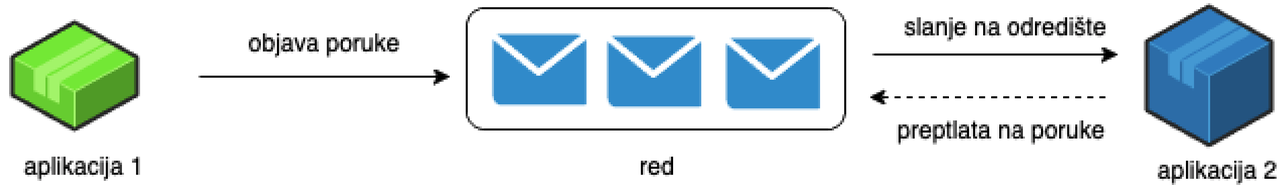


SCENARIJI TOKOVA PODATAKA

TRI NAJČEŠĆA SCENARIJA

- Direktna komunikacija klijent <-> server kroz poziv servisa
- Serverska aplikacija <-> Baza podataka
- Asinhrona komunikacija razmenom poruka preko reda poruka (message queue)





WEB SERVER

ČEMU SLUŽI?

- Komponenta odgovorna za prihvatanje HTTP zahteva i vraćanje odgovora
- Dizajniran je da služi statički sadržaj (ali mnogi imaju podršku za scripting jezike poput Pearl, PHP, itd. za generisanje dinamičkog HTTP sadržaja)

POZNATI SERVERI

- Apache HTTP Server
- Nginx
- lighttpd
- ...

APLIKATIVNI SERVER

ČEMU SLUŽI?

- Generiše dimanički sadržaj koji dolazi kao rezultat izvršavanja poslovne logike
- Većina aplikativnih servera ima web server kao svoj sastavni deo, te može da radi iste stvari
- Dodatno može imati podršku za *connection pooling*, *messaging* servise, itd.
- Nije limitiran na korišćenje samo HTTP protokola

POZNATI SERVERI

- Apache Tomcat
- WildFly (JBoss)
- Oracle WebLogic
- Gunicorn
- ...

KORACI ZA OBRADU ZAHTEVA

PRIHVATANJE ZAHTEVA

- Ako je u pitanju novi zahtev, prvo mora da se uspostavi HTTP konekcija preko TCP konekcije

INICIJALNA OBRADA ZAHTEVA

- Podrazumeva čitanje bajtova (*I/O bound*) i parsiranje HTTP zahteva (*CPU bound*)
- Ako je u pitanju POST ili PUT zahtev, zahteva se dodatno procesiranje podataka koji se šalju u zahtevu

SLANJE ZAHTEVA APLIKACIJI NA DALJU OBRADU

- Podrazumeva slanje zahteva sloju poslovne logike
- Prosleđivanje poziva se može svesti na čitanje podataka sa fajl sistema/baze podataka, slanje zahteva preko mreže na neki message queue, RPC poziv,... (*I/O bound*)

KORACI ZA OBRADU ZAHTEVA

KREIRANJE ODGOVORA

- Kada je zahtev obrađen i resurs je spreman (HTML stranica, slika, video, ...), vraća se klijentu pisanjem na socket

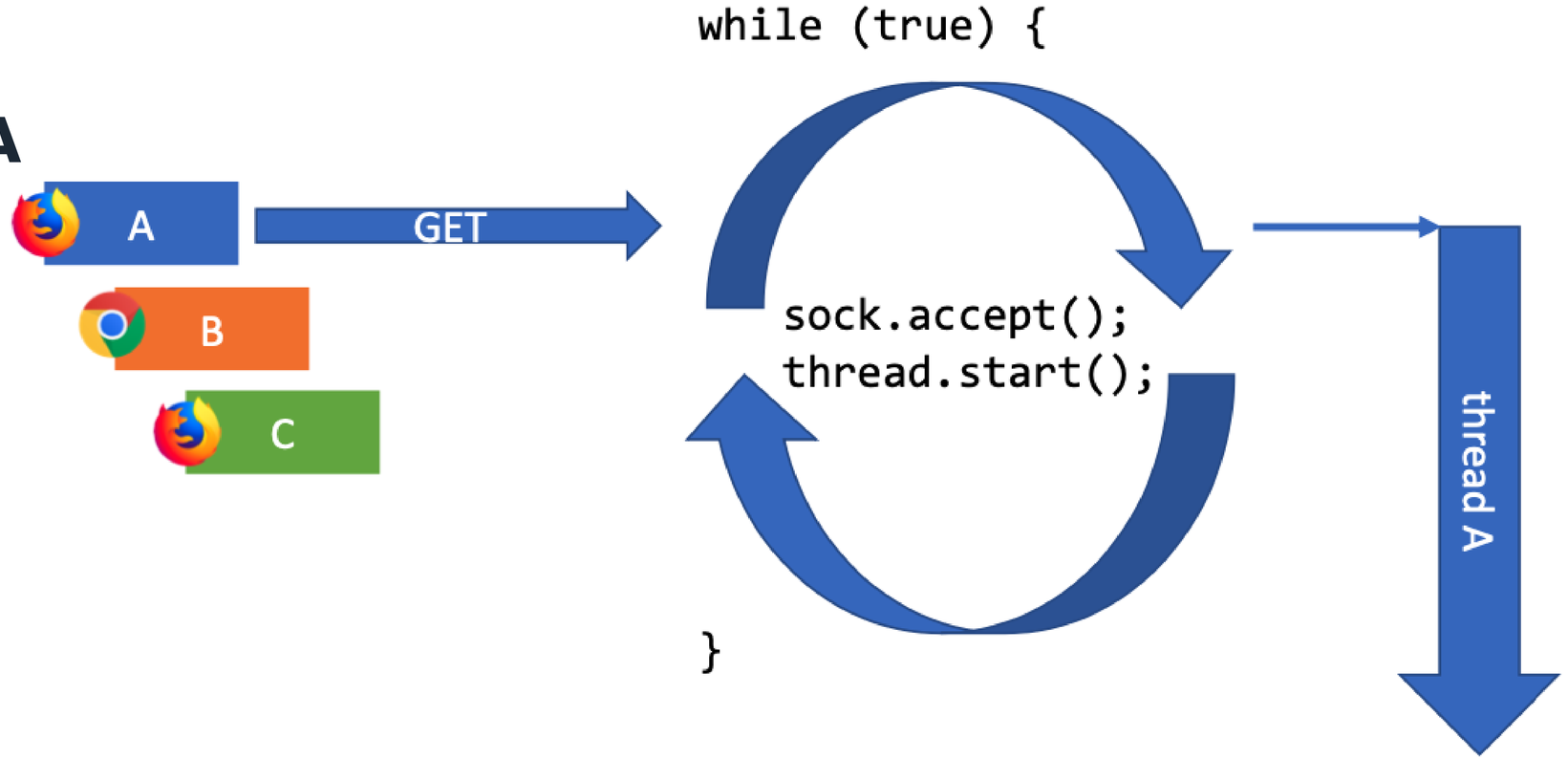
ZAVRŠETAK OBRADE ZAHTEVA

- Web server zatvara konekciju ili se vraća na prvi korak i čeka sledeći zahtev

SERVERSKE ARHITEKTURE BAZIRANE NA NITIMA

TOK OBRADE ZAHTEVA

- Klijent A šalje GET zahtev serveru
- Serverska glavna petlja pokreće novu nit za obradu zahteva
- Nit počinje obradu zahteva paralelno sa glavnom petljom
- Glavna petlja ponovo čeka novi zahtev



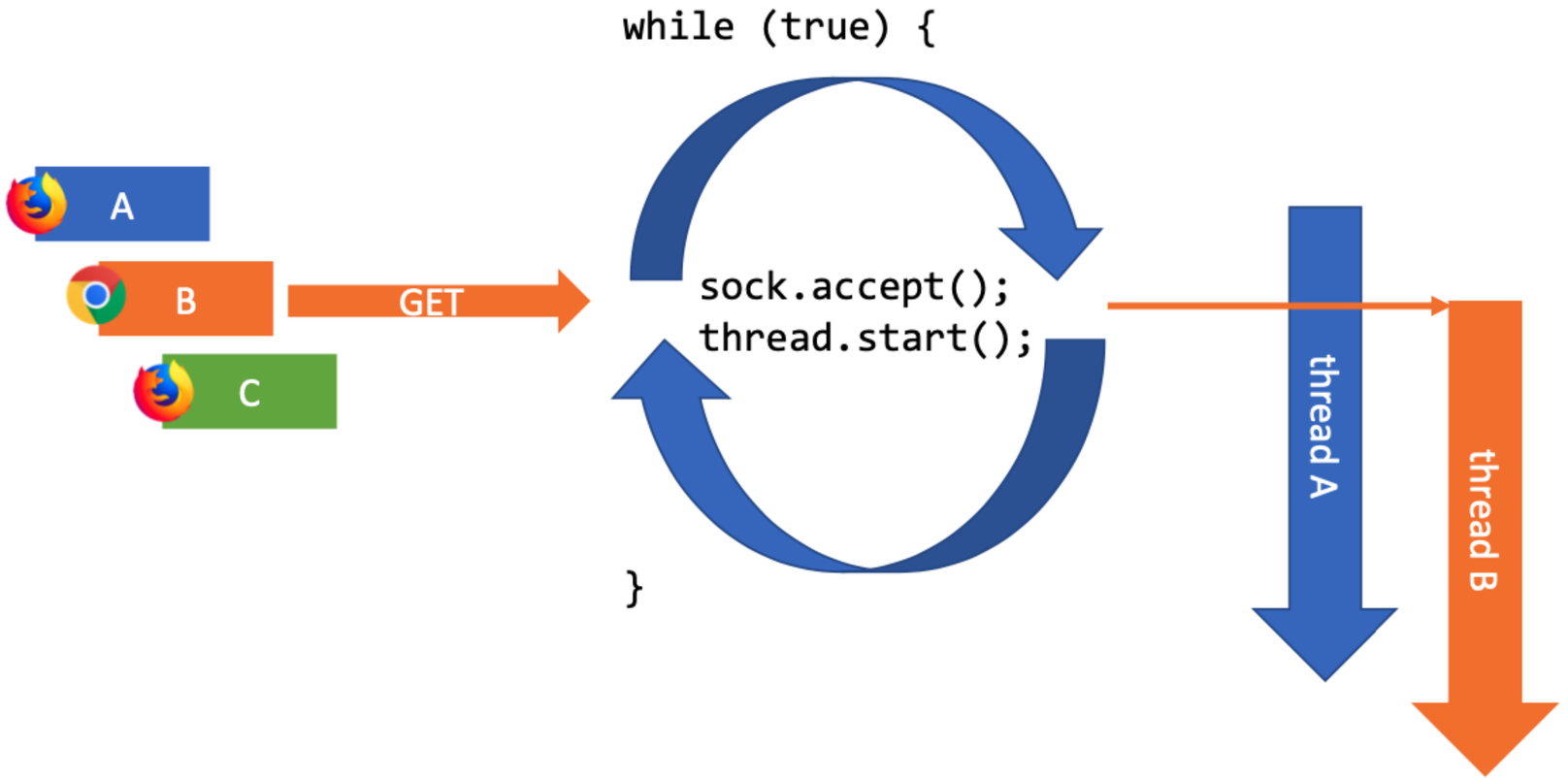
slika preuzeta sa https://github.com/mbranko/isa19/blob/master/01-threads/thread-models.pdf

thread-based-server primer

SERVERSKE ARHITEKTURE BAZIRANE NA NITIMA

TOK OBRADE ZAHTEVA

- Klijent B šalje GET zahtev serveru
- Serverska glavna petlja pokreće novu nit za obradu zahteva
- Nit počinje obradu zahteva paralelno sa glavnom petljom
- Glavna petlja ponovo čeka novi zahtev
- Prethodna nit za obradu zahteva još nije završila rad



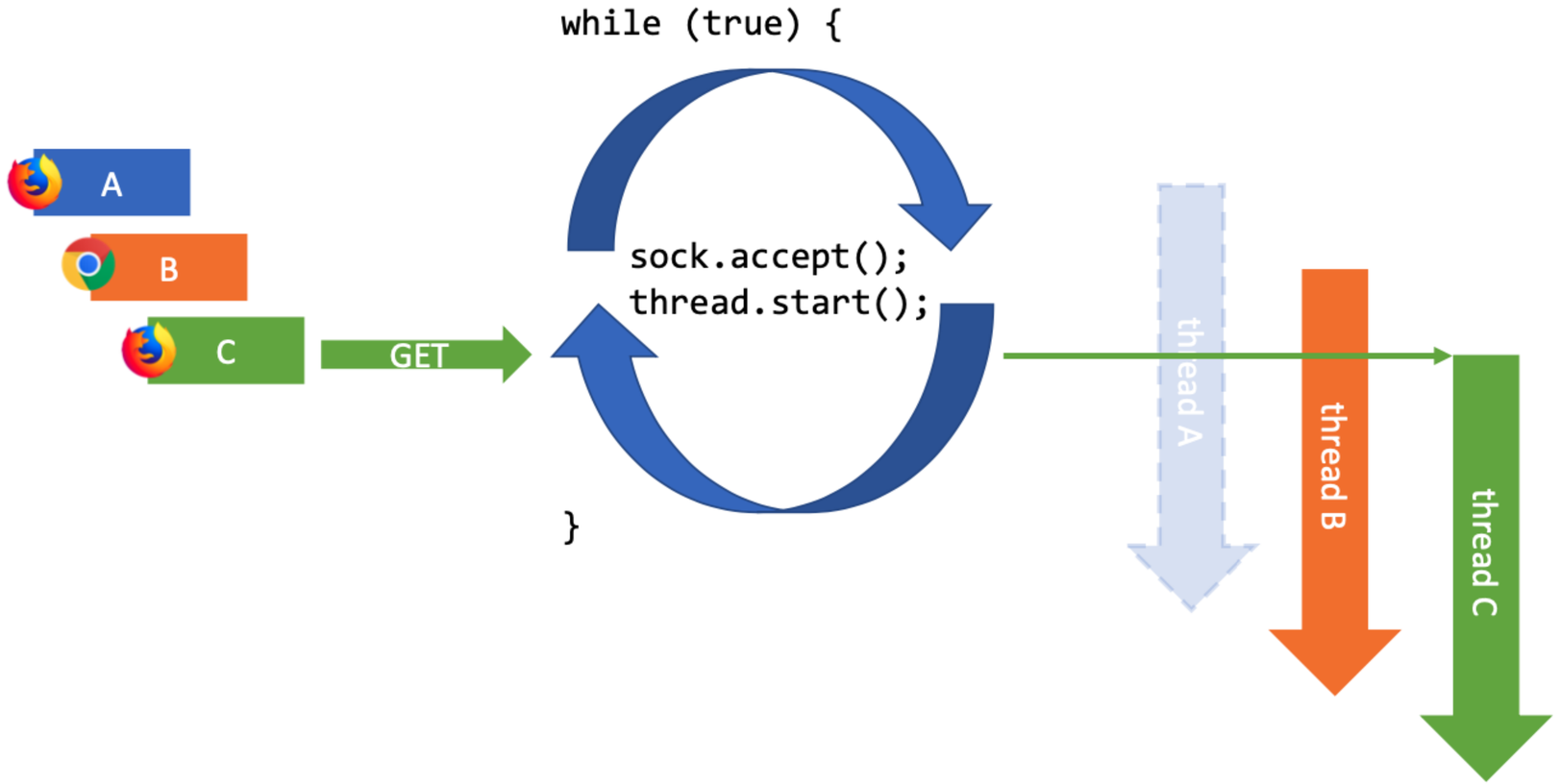
slika preuzeta sa https://github.com/mbranko/isa19/blob/master/01-threads/thread-models.pdf

thread-based-server primer

SERVERSKE ARHITEKTURE BAZIRANE NA NITIMA

TOK OBRADE ZAHTEVA

- Klijent C šalje GET zahtev serveru
- Serverska glavna petlja pokreće novu nit za obradu zahteva
- Nit počinje obradu zahteva paralelno sa glavnom petljom
- Glavna petlja ponovo čeka novi zahtev
- Prva nit za obradu zahteva je završila rad
- Druga nit još uvek radi



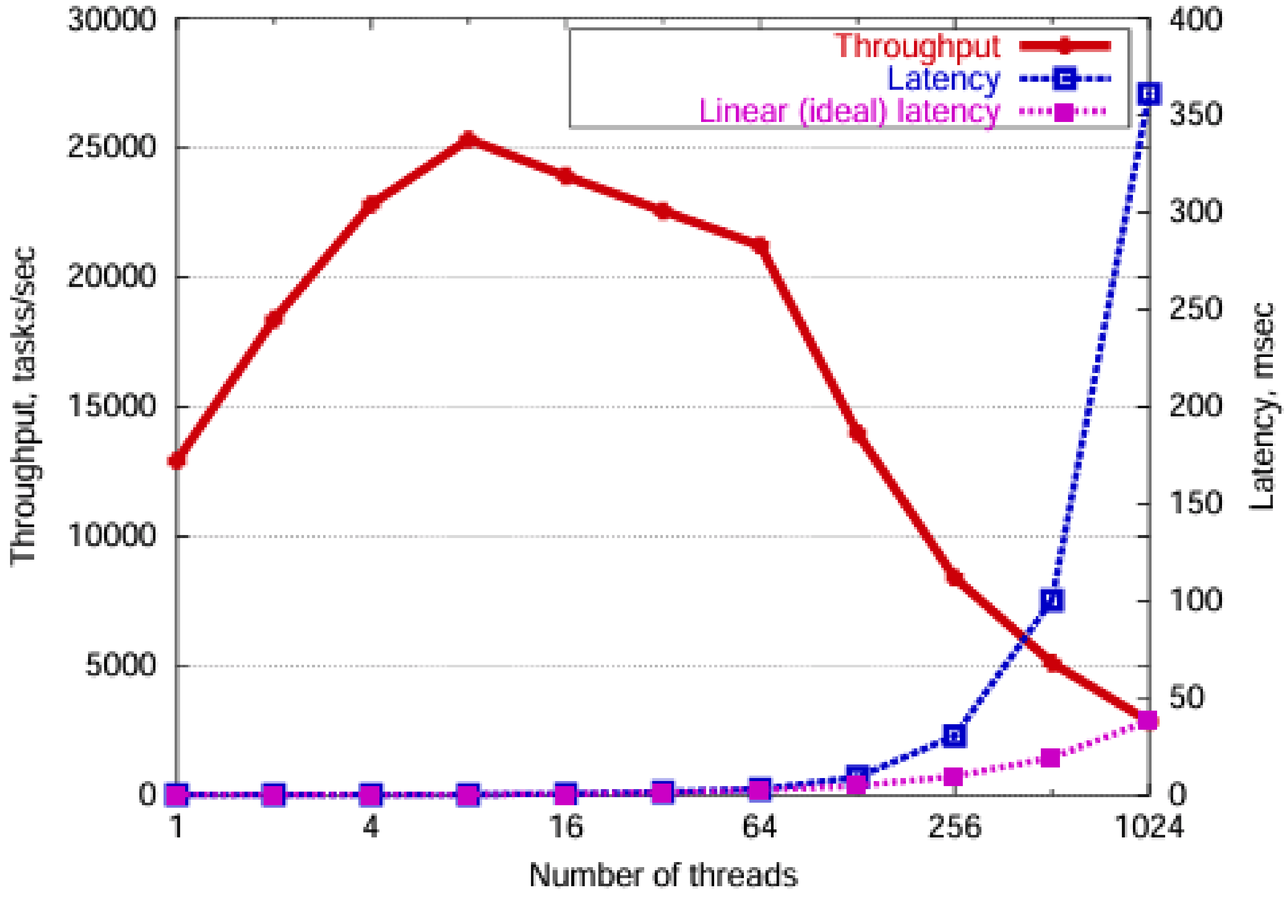
slika preuzeta sa https://github.com/mbranko/isa19/blob/master/01-threads/thread-models.pdf

thread-based-server primer

SERVERSKE ARHITEKTURE BAZIRANE NA NITIMA

DEGRADACIJA PROPUSNOG OPSEGA

- Kreira se nit za svaki novi zadatak
- Nit čita 8KB sa diska
- Niti su prealocirane kako bi se eliminisao overhead za startovanje
- Server implementiran u C
- 500MHz Pentium III sa 2GB RAM i Linux 2.2.14
- Kako se broj konkurentnih zahteva povećava, propusni opseg se povećava dok broj niti ne postane prevelik i onda dolazi do degradiranja
- Vreme odgovora postaje neograničeno kako se dužina redova zadataka povećava



slika preuzeta iz https://people.eecs.berkeley.edu/~brewer/papers/SEDA-sosp.pdf

C10K PROBLEM

KAKO WEB SERVERI MOGU DA IZAĐU NA KRAJ SA 10.000 ISTOVREMENIH ZAHTEVA?

- Dan Kegel 1999. objavio članak [1]
- Danas predstavlja osnovni resurs za diskusiju o skalabilnosti web servera
- Smatra se da (u to vreme) hardver ne mora biti usko grlo sistema za obradu konkurentnih konekcija
- Za server sa 2GB memorije i Gigabit Ethernet karticom, istovremeno rukovanje sa 10.000 zahteva zahtevalo bi da svaki zahtev koristi manje od 200KB (2GB/10.000) memorije i 100Kbit (1000Mbit/10.000) mrežnog propusnog opsega
- Fizički resursi su bili dovoljni, pa se prirodno nametnulo da je u pitanju softverski problem, posebno I/O model
- TL;DR - Problem je rešen izmenama u kernelu OS i prelaskom na event-driven servere (npr. Ngnix i Node bazirane)
- Osnova za nove probleme C100K, C1M, C10M,...

[1] http://www.kegel.com/c10k.html

C10K PROBLEM

KAKO WEB SERVERI MOGU DA IZAĐU NA KRAJ SA 10.000 ISTOVREMENIH ZAHTEVA?

- Strategije za rešavanje problema
  - Multithreading/Multiprocessing
    - Svakoj konekciji dodeljuje sopstvenu nit ili proces, omogućavajući sistemu da istovremeno rukuje zadacima.
    - Prednosti: Lako se implementira sa postojećim modelima programiranja.
    - Izazovi: Velika upotreba memorije i prekomerno prebacivanje konteksta (engl. *context-switching*) ograničavaju skalabilnost.

SERVERSKE ARHITEKTURE BAZIRANE NA DOGAĐAJIMA

DEMULTIPLEKSER DOGAĐAJA

- Metod kojim se više signala kombinuje u jedan signal preko zajedničkog resursa
- Cilj je da se deli oskudan resurs (npr. CPU)
- To je teorijski model koji se koristi da objasni kako se efikasno rukuje brojnim istovremenim događajima
- Odgovornosti demultipleksera događaja su podeljene u sledeće korake:
  - Identifikacija izvora događaja
  - Registracija izvora događaja
  - Čekanje na događaje
  - Slanje obaveštenja o događaju
- Svaki operativni sistem implementira mehanizam demultipleksera događaja na neki način (npr. epoll (Linux), kqueue (BSD), IOCP (Windows)*)

* IOCP - Input/Output Completion Ports – Windows verzija rada sa višestrukim asinhronim I/O zahtevima

C10K PROBLEM

KAKO WEB SERVERI MOGU DA IZAĐU NA KRAJ SA 10.000 ISTOVREMENIH ZAHTEVA?

- Strategije za rešavanje problema
  - Sinhrono multipleksiranje (select()*/poll()**) – zasnovano na spremnosti
    - Koristi sistemske pozive select() ili poll() da nadgleda više socketa za događaje, omogućavajući jednoj niti da upravlja sa više konekcija.
    - Prednosti: Pojednostavljuje upravljanje konekcijama, smanjujući potrebu za više niti.
    - Izazovi: Performanse opadaju sa mnogo konekcija jer su ovi sistemski pozivi neefikasni u velikim sistemima.

* select() – sistemski poziv koji omogućava programu da nadgleda više fajl deskriptora, čekajući dok jedan ili više njih ne postanu "spremni” za neku klasu I/O operacija (ograničen je na 1024 fajl deskriptora), O(n)

** poll – slično, ali podržava veći skup fajl deskriptora, O(n)

C10K PROBLEM

KAKO WEB SERVERI MOGU DA IZAĐU NA KRAJ SA 10.000 ISTOVREMENIH ZAHTEVA?

- Strategije za rešavanje problema
  - Arhitektura vođena događajima (epoll/kqueue) – zasnovano na spremnosti
    - Poboljšava select()/poll() efikasnim sistemskim pozivima kao što su epoll()* (Linux) ili kqueue() (BSD). Oni omogućavaju registrovanje događaja za mnogo fajl deskriptora.
    - Prednosti: Dobro skalira, koristeći manje sistemskih resursa.
    - Izazovi: Složenost u programiranju u poređenju sa sinhronim modelima.

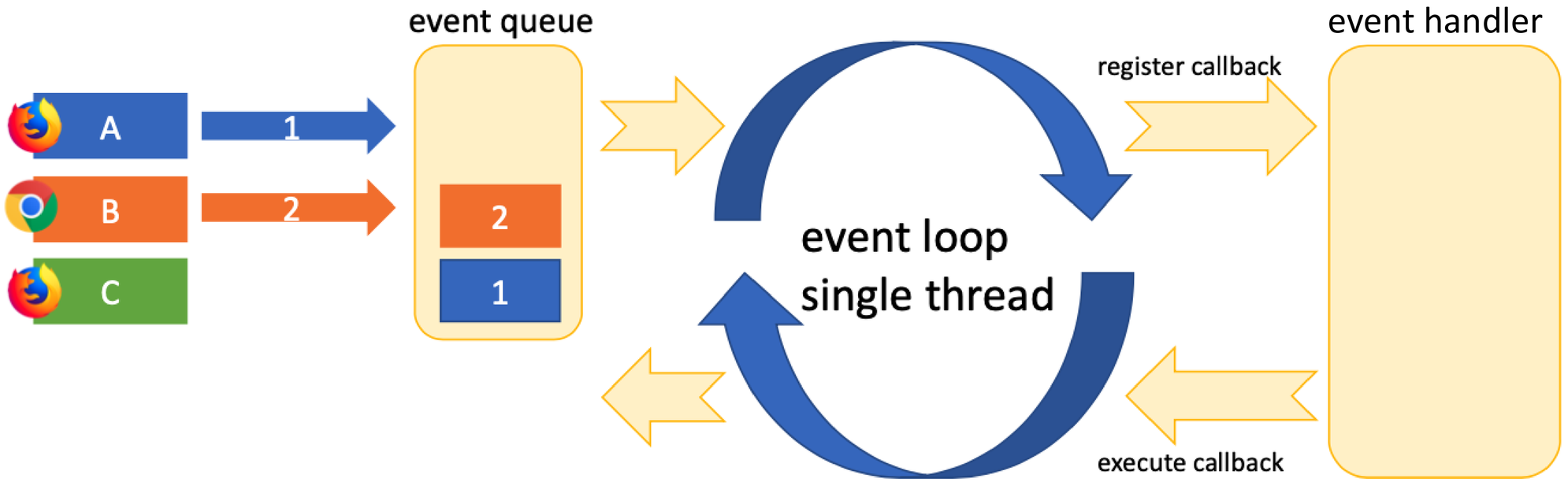
* epoll – sistemski poziv za praćenje više deskriptora datoteka da se vidi da li je I/O moguć na bilo kom od njih, O(1)

C10K PROBLEM

KAKO WEB SERVERI MOGU DA IZAĐU NA KRAJ SA 10.000 ISTOVREMENIH ZAHTEVA?

- Strategije za rešavanje problema
  - Asinhroni O/I (Async I/O / Completion-based I/O)
    - Koristi sistemske pozive asinhronog ulaza/izlaza da kernel sam izvrši I/O operaciju u pozadini, a aplikaciju obavesti kada je operacija završena.
    - Za razliku od modela zasnovanog na spremnosti (epoll/kqueue), gde aplikacija tek nakon obaveštenja sama obavlja neblokirajući read/write, u asinhronom modelu kernel radi ceo posao i vraća rezultat putem povratnog poziva u *completion queue*.
    - Prednosti: Efikasan, posebno za rukovanje velikim brojem konekcija bez blokiranja.
    - Izazovi: Može dovesti do složenog koda sa greškama kojima je teško ući u trag zbog pakla povratnog poziva (engl. *callback hell*).

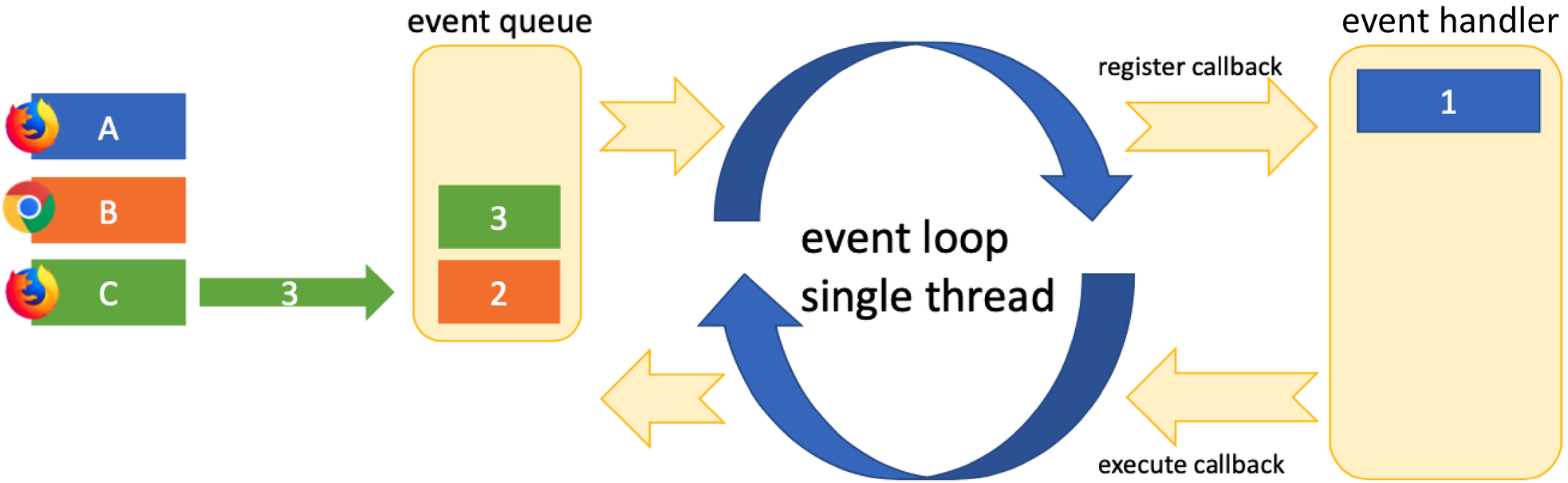
SERVERSKE ARHITEKTURE BAZIRANE NA DOGAĐAJIMA



slika preuzeta sa https://github.com/mbranko/isa19/blob/master/01-threads/thread-models.pdf

event-based-server primer

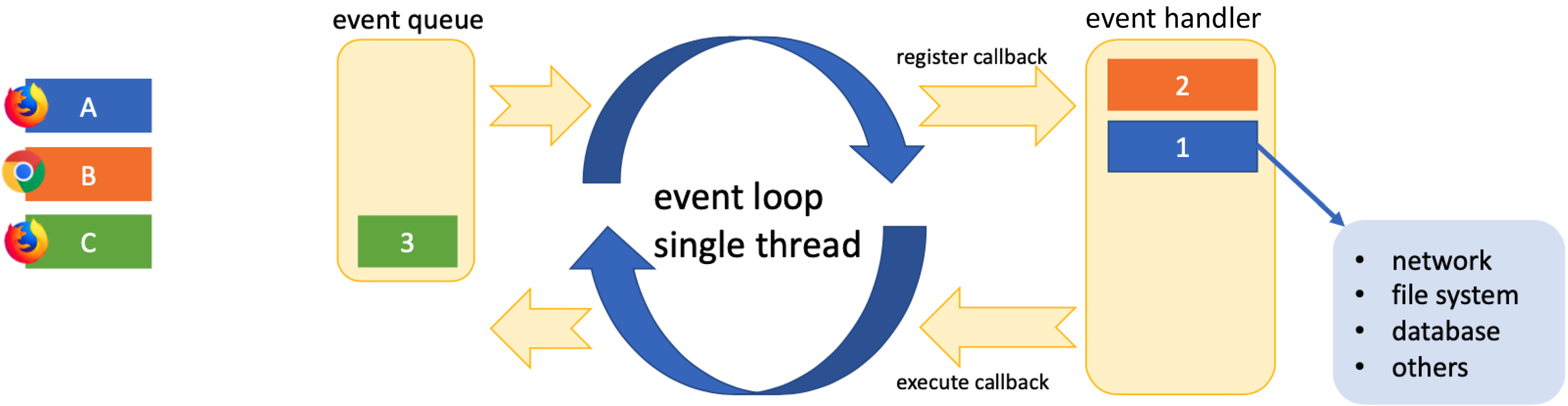
SERVERSKE ARHITEKTURE BAZIRANE NA DOGAĐAJIMA



slika preuzeta sa https://github.com/mbranko/isa19/blob/master/01-threads/thread-models.pdf

event-based-server primer

SERVERSKE ARHITEKTURE BAZIRANE NA DOGAĐAJIMA



slika preuzeta sa https://github.com/mbranko/isa19/blob/master/01-threads/thread-models.pdf

event-based-server primer

SERVERSKE ARHITEKTURE BAZIRANE NA DOGAĐAJIMA

LIBUV[1]

- *Cross-platform* biblioteka originalno pisana za Node.js (pisana u C)
- Treba da obezbedi konzistentan interfejs za neblokirajuće I/O operacije u različitim operativnim sistemima
- Libuv ne samo da se povezuje sa sistemskim demultiplekserom događaja, već uključuje rad sa dve važne komponente
  - Event Queue – struktura podataka gde se svi događaji smeštaju, kako bi se čitali sekvencijalno od strane Event Loop komponente
  - Event Loop – komponenta koja se stalno izvršava i čeka poruke iz Event Queue komponente a zatim ih šalje na obradu

[1] LIBUV - https://docs.libuv.org/en/v1.x/design.html

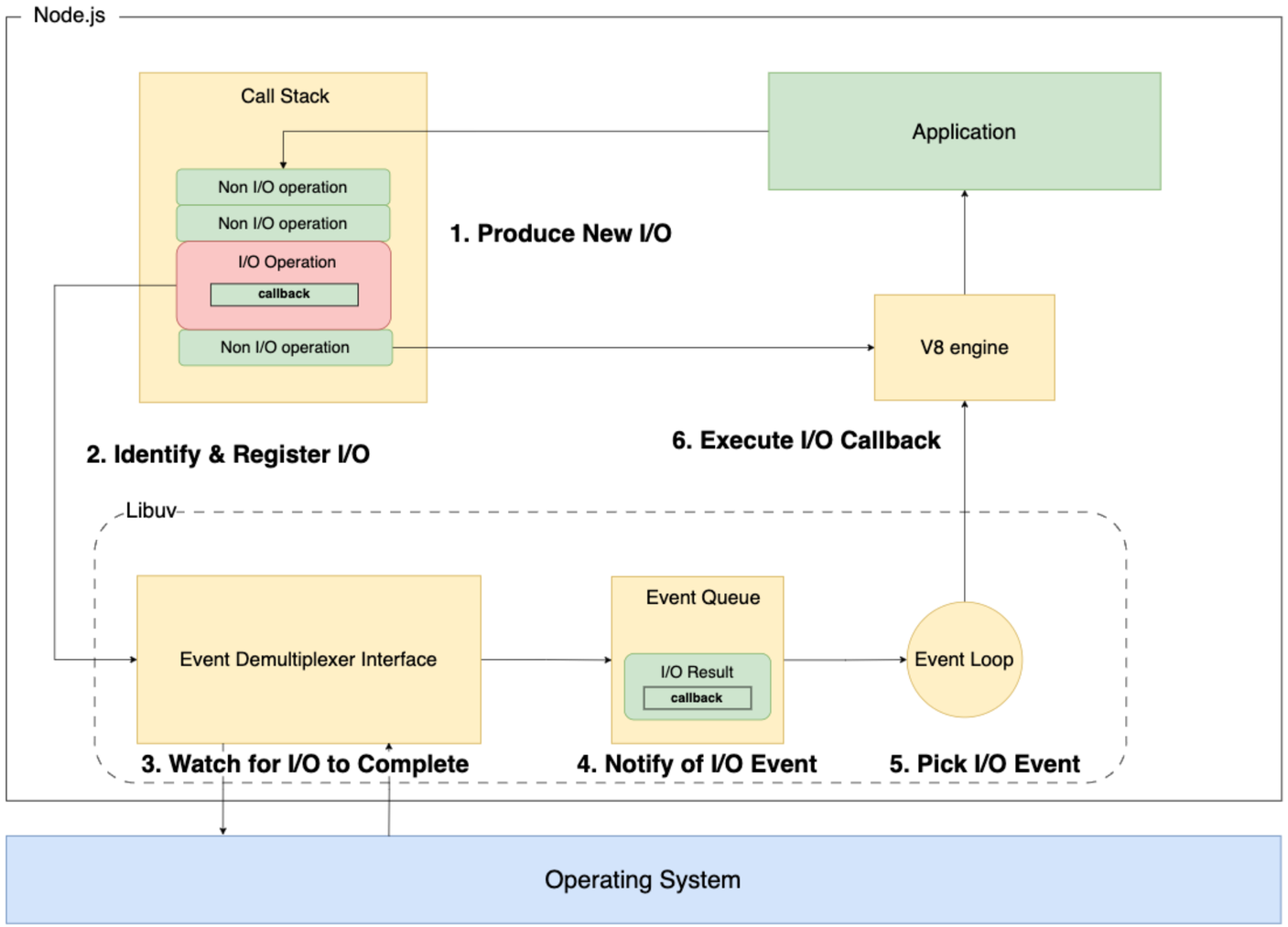
SERVERSKE ARHITEKTURE BAZIRANE NA DOGAĐAJIMA

ŠTA SE DEŠAVA KADA SE POZOVE I/O OPERACIJA U NODE.JS?

1. Libuv inicijalizuje OS demultiplekser događaja (epoll/kqueue/IOCP)
2. Node.js (V8) izvršava JavaScript kod u *Call Stack*-u (sekvencijalno)
3. Kada se pozove I/O operacija, Node predaje operaciju libuv-u, a *Call Stack* odmah nastavlja rad (ne blokira)
4. Libuv registruje I/O operaciju u OS demultiplekseru koristeći odgovarajuće OS primitive
5. OS demultiplekser asinhrono nadgleda I/O izvore (socket, file, timer, DNS...)
6. Kada se dogodi I/O događaj, demultiplekser obaveštava libuv, koji ubacuje callback u Event Queue
7. Event Loop uzima događaje iz Event Queue i izvršava callback funkcije u glavnoj niti

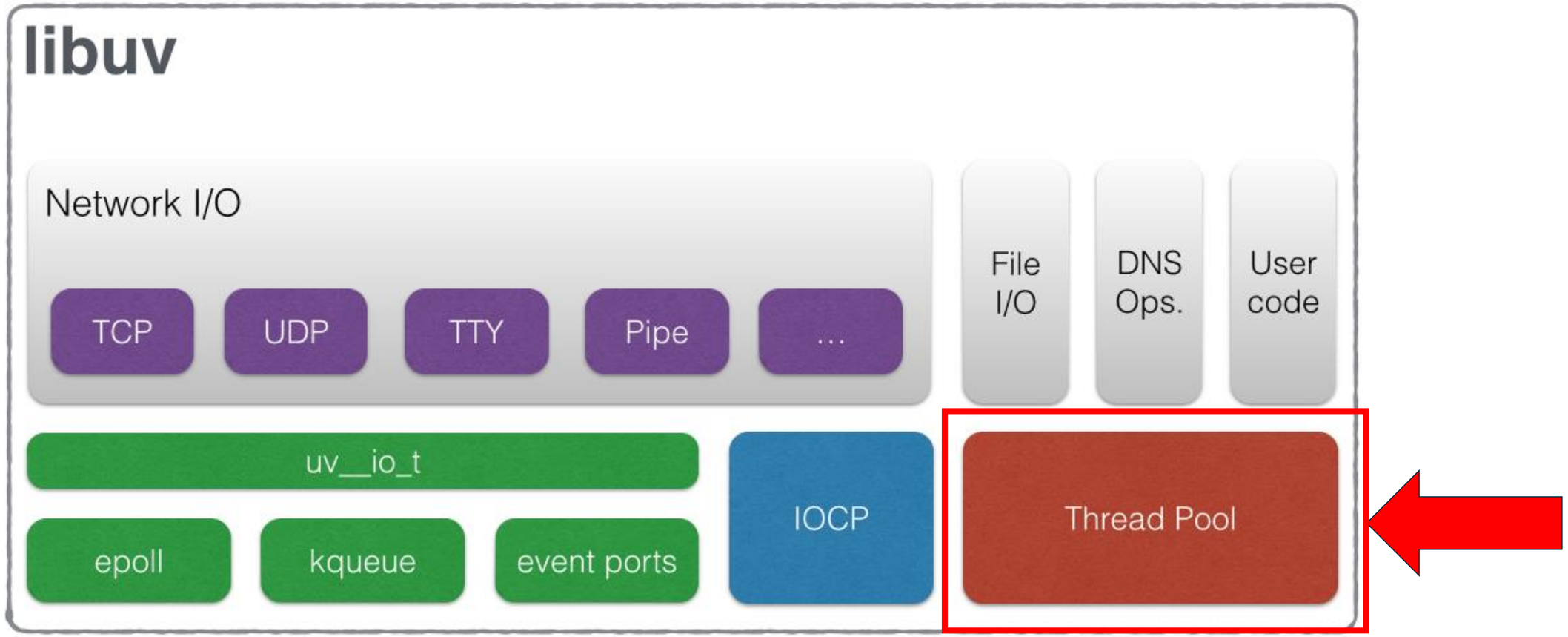
SERVERSKE ARHITEKTURE BAZIRANE NA DOGAĐAJIMA

ŠTA SE DEŠAVA KADA SE POZOVE I/O OPERACIJA U NODE.JS?



SERVERSKE ARHITEKTURE BAZIRANE NA DOGAĐAJIMA

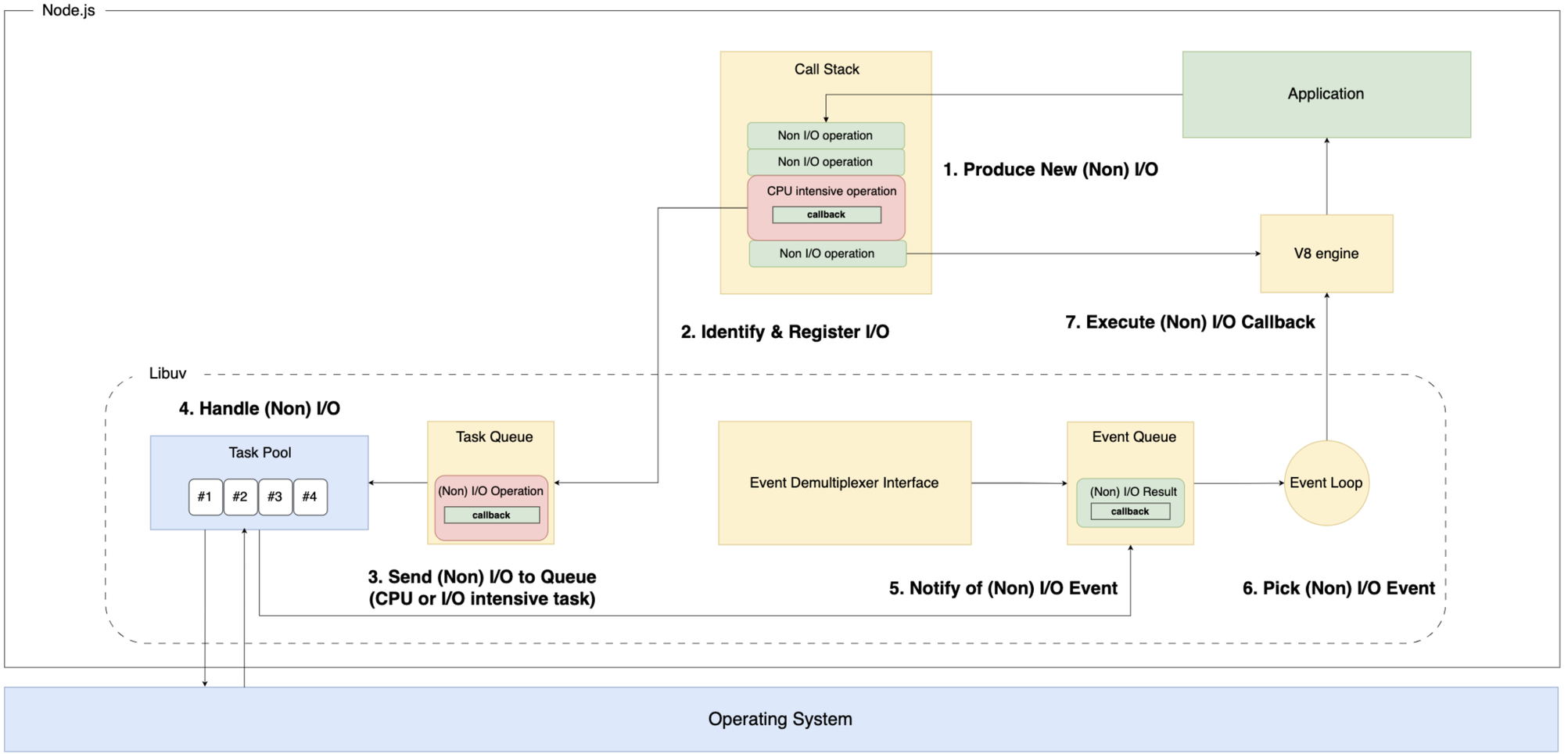
LIVUB KOMPONENTE



slika preuzeta sa https://docs.libuv.org/en/v1.x/design.html

SERVERSKE ARHITEKTURE BAZIRANE NA DOGAĐAJIMA

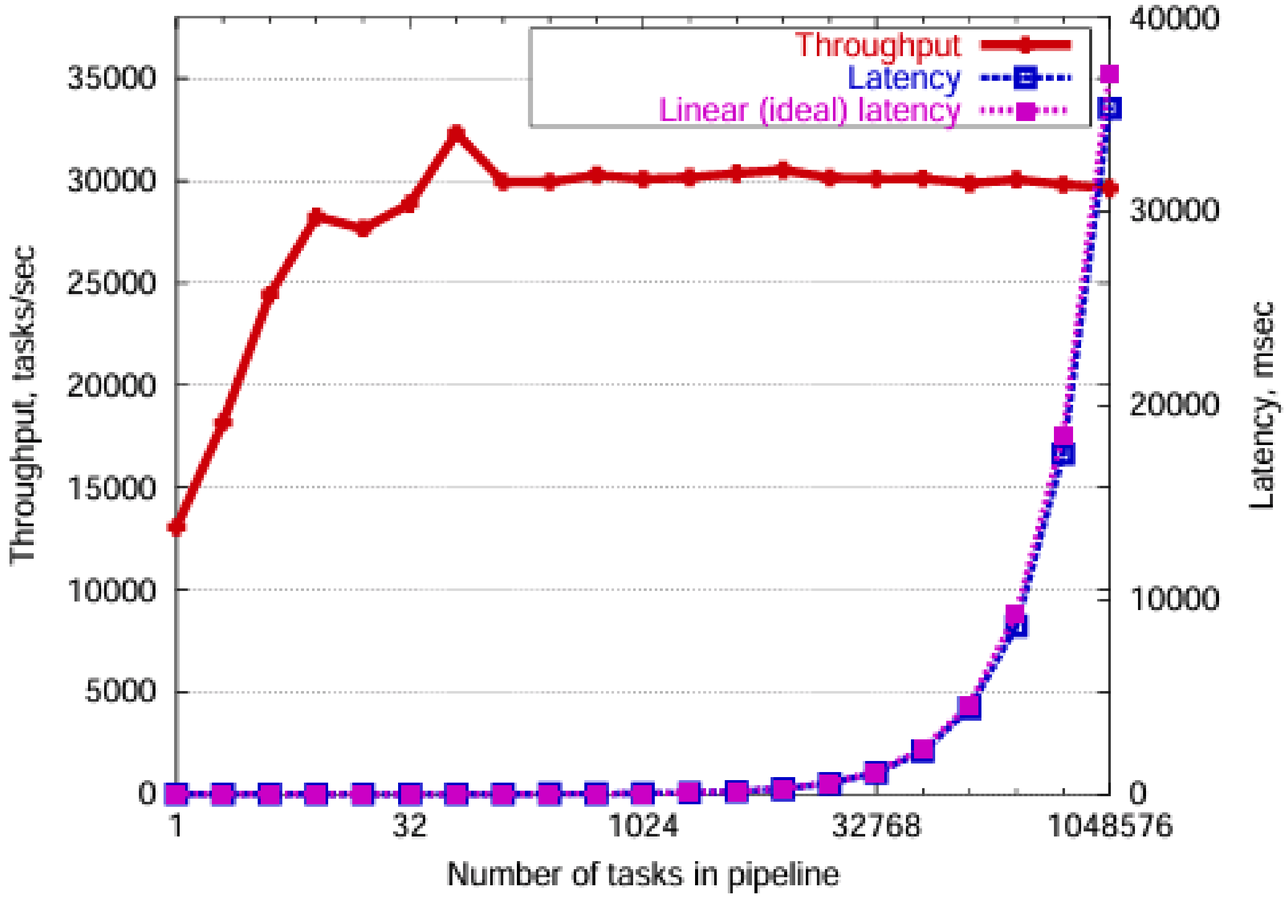
ŠTA SE DEŠAVA KADA SE POZOVE BLOKIRAJUĆA OPERACIJA U NODE.JS?



SERVERSKE ARHITEKTURE BAZIRANE NA DOGAĐAJIMA

DEGRADACIJA PROPUSNOG OPSEGA

- Koristi se jedna nit za zadatke
- Nit čita 8KB sa diska
- Server implementiran u C
- 500MHz Pentium III sa 2GB RAM i Linux 2.2.14
- Propusni opseg ostaje konstantan kak se opterećenje povećava na veoma veliki broj zadataka
- Vreme odgovora je linearno

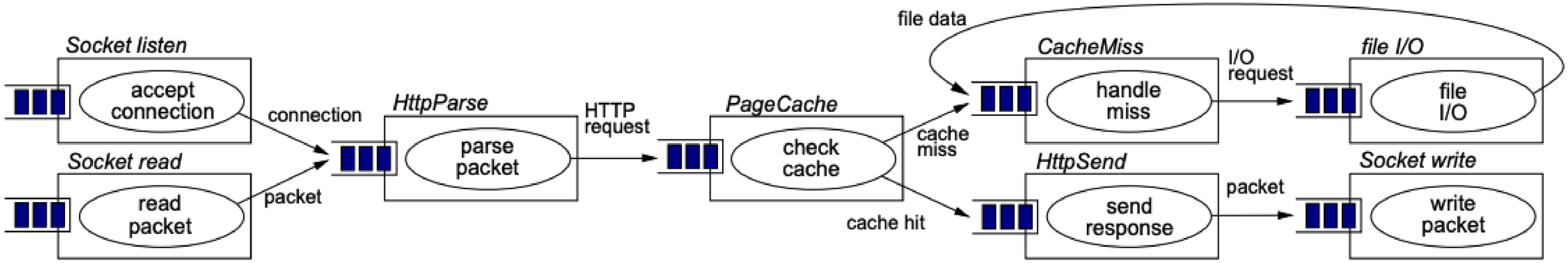


slika preuzeta iz https://people.eecs.berkeley.edu/~brewer/papers/SEDA-sosp.pdf

STAGED EVENT-DRIVEN ARCHITECTURE (SEDA)

KOMBINOVANA ARHITEKTURA KOJA KORISTI I NITI I DOGAĐAJE

- Predložili Matt Welsh, David Culler i Eric Brewer [1] s namerom da unaprede performanse
- U osnovi je podeljena serverska logika u lanac striktno definisanih faza
- Faze su povezane pomoću redova
- Zahtevi se prosleđuju iz jedne u drugu fazu tokom procesiranja
- Svaka faza ima nit ili skup niti (*thread pool*) koji mogu da se konfigurišu dinamički



[1] https://people.eecs.berkeley.edu/~brewer/papers/SEDA-sosp.pdf

slika preuzeta iz istog rada

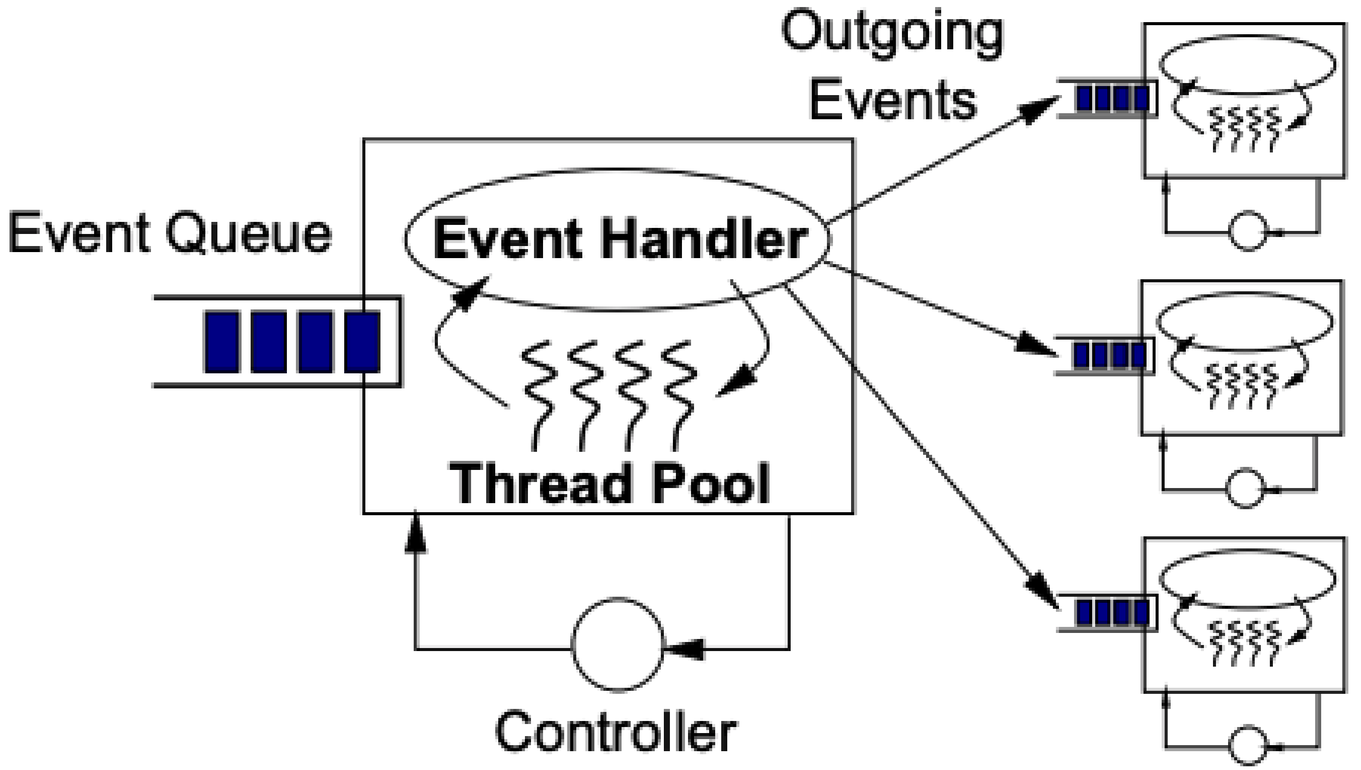
STAGED EVENT-DRIVEN ARCHITECTURE (SEDA)

CILJEVI

- Podrška za masivnu konkurentnost
  - Da bi se izbegla degradacija performansi usled korišćenja niti, SEDA koristi izvršenje vođeno događajima gde god je to moguće.
- Pojednostavljivanje konstrukcije Internet servisa
  - Da bi se smanjila složenost izgradnje Internet servisa, SEDA štiti programere aplikacija od mnogih detalja planiranja i upravljanja resursima. Dizajn takođe podržava modularnu konstrukciju ovih aplikacija i pruža podršku za otklanjanje grešaka i profilisanje performansi.
- Podrška za introspekciju
  - Aplikacije bi trebalo da budu u stanju da analiziraju tok zahteva kako bi prilagodile ponašanje promenljivim uslovima opterećenja. Na primer, sistem bi trebalo da bude u stanju da odredi prioritete i filtrira zahteve da podrži degradiranu uslugu pod velikim opterećenjem.
- Podrška za samopodešavanje (self-tuning) upravljanjem resursima
  - Umesto da zahteva prethodno znanje o zahtevima za resurse aplikacije i karakteristikama opterećenja klijenta, sistem bi trebalo da dinamički prilagodi svoje parametre upravljanja resursima kako bi ispunio ciljeve performansi. Na primer, broj niti dodeljenih fazi može se automatski odrediti na osnovu uočenih zahteva za konkurentnost, umesto da ih programer unapred iskodira.

STAGED EVENT-DRIVEN ARCHITECTURE (SEDA)

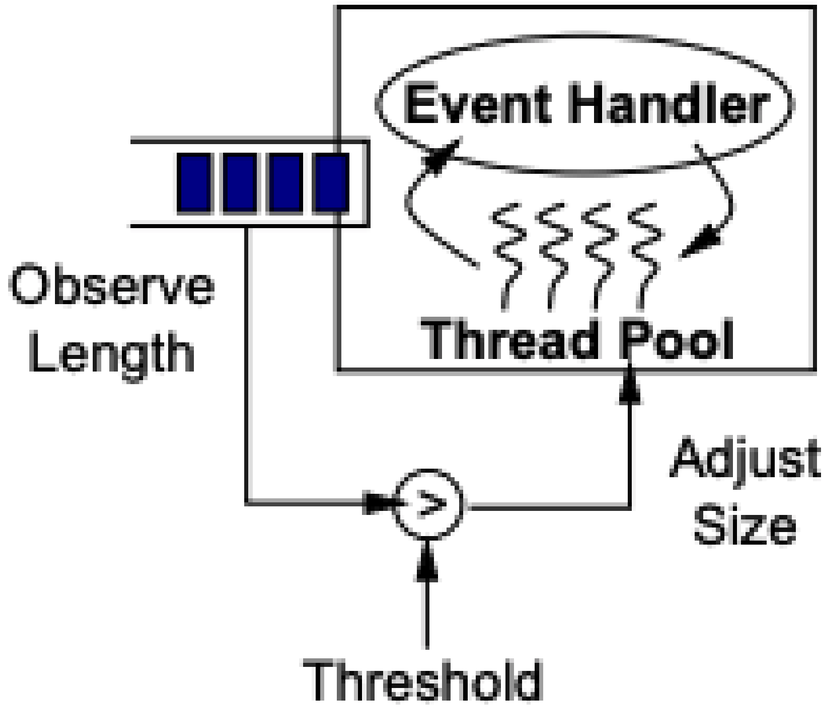
SEDA FAZA



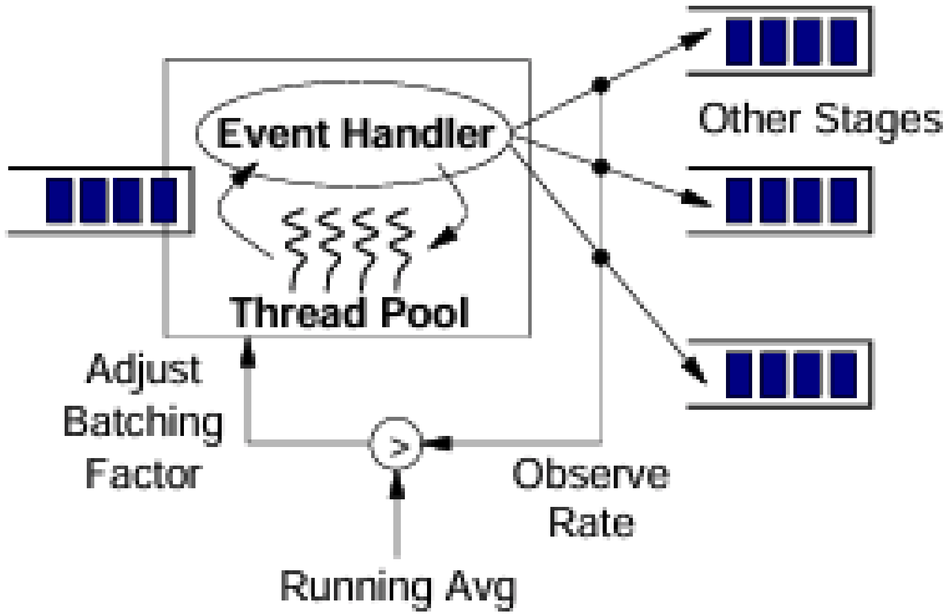
slika preuzeta iz https://people.eecs.berkeley.edu/~brewer/papers/SEDA-sosp.pdf

STAGED EVENT-DRIVEN ARCHITECTURE (SEDA)

THREAD POOL KONTROLER



BATCHING KONTROLER



slike preuzete iz https://people.eecs.berkeley.edu/~brewer/papers/SEDA-sosp.pdf

METRIKE ZA MERENJE PERFORMANSI SERVERA

METRIKE ZA MERENJE PERFORMANSI SERVERA

- Propusni opseg zahteva (*request throughput* meren kao broj zahteva po jedinici vremena – req/sec)
- Propusni opseg podataka (meren u Mbps)
- Vreme odgovora (*response time* meren u ms)
- Broj konkurentnih konekcija (izražen kao broj)

DODATNE METRIKE UKLJUČUJU MONITORING SERVERSKE MAŠINE

- Zauzeće CPU
- Zauzeće memorije
- Broj niti/procesa
- Broj otvorenih soketa
- Broj otvorenih fajlova
- ...

REFERENCE

- PRIMERI PO UZORU NA https://github.com/mbranko/isa19/tree/master/01-threads
- PRIMER ZA MERENJE PERFORMANSI https://github.com/mbranko/isa19/tree/master/01-threads/analyze
- WELSH ET AL. AN ARCHITECTURE FOR WELL-CONDITIONED, SCALABLE INTERNET SERVICES. https://people.eecs.berkeley.edu/~brewer/papers/SEDA-sosp.pdf
- DAN KAGEL. C10K PROBLEM. http://www.kegel.com/c10k.html
- PARIAG ET AL. COMPARING THE PERFORMANCE OF WEB SERVER ARCHITECTURES. https://people.eecs.berkeley.edu/~brewer/cs262/Pariag07.pdf
- NODE.JS EVENT LOOP. https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/
- BENJAMIN ERB. CONCURRENT PROGRAMMING FOR SCALABLE WEB ARCHITECTURES. https://berb.github.io/diploma-thesis/
- NIKHIL MARATHE. AN INTRODUCTION TO LIBUV. https://nikhilm.github.io/uvbook/basics.html

KOJA SU VAŠA PITANJA?