Факултет техничких наука

Универзитет у Новом Саду

Паралелне и дистрибуиране архитектуре и језици

Поређење имплементација вишенитног сервера у програмским језицима *Rust* и *Go*

Аутор:
Александар Стојановић

Индекс:
E2 119/2023

23. децембар 2023.

Садржај

Списак изворних кодова

Списак слика

Списак табела

1 Увод

Циљ овог семинарског рада је имплементација једноставног, вишенитног веб сервера који би опслуживао само два захтева, *GET* и *PUT* над ентитетом који се састоји од три поља *id (int64), description(string)* и *value (float64)*. За имплементацију одабрани су језици *Rust* и *Golang*, чији ће се приступи решавању проблема које ова имплементација носи, као и перформансе које они постижу упоређивати кроз цео рад.

У даљем тексту, уколико се имплементација значајно не разликује у оба језика, пример кода биће приказан у *Rust*-у и језик имплементације неће бити наведен.

2 Коришћени језици

2.1 *Rust*

Rust је модерни програмски језик који је осмишљен са циљем обезбеђивања сигурности, перформанси и практичности. Развијен од стране *Mozilla research*-а [3], истиче се својом снажном подршком за паралелно и конкурентно програмирање, чиме омогућава програмерима да ефикасно искористе савремене вишејезгарне процесорске архитектуре. Овај језик је познат по својој јединственој карактеристици - власничком систему типова, који омогућава прецизно управљање меморијом без угрожавања безбедности. *Rust* такође пружа богат скуп функционалности, укључујући алгебарске типове података, макро систем и једноставну синтаксу. Својом комбинацијом перформанси и безбедности, често се користи у различитим областима, од системског програмирања до веб развоја, нудећи програмерима снажан алат за изградњу поузданих и ефикасних софтверских решења.

2.2 *Golang*

Golang, или *Go*, је модерни програмски језик који је развијен у оквиру *Google*-а. Основни принципи дизајна овог језика усмерени су ка једноставности, перформансама и ефикасности развоја софтвера. Истиче се брзим компајлирањем, чиме омогућава ефикасно испоручивање извршних фајлова, чак и за велике пројекте. Језик подржава конкурентно и паралелно програмирање као део свог језичког система, што га чини посебно прикладним за развој софтвера који захтева ефикасно управљање вишејезгарним процесорским архитектурама. *Go* такође промовише чист и једноставан код, олакшавајући одржавање и разумевање софтверских пројеката. Често се користи у разним областима, укључујући серверски развој, мрежно програмирање и рад са контејнерима. Својим фокусом на продуктивност програмера и ефикасност извршавања кода, постао је популаран избор у индустрији софтверског инжењеринга.

3 Ослушкивање захтева

Да би се омогућило слање захтева на сервер потребно је да се сервер подеси да ослушкује *TCP* захтеве на одређеном порту. Обе имплементације проблем решавају на сличан начин, креирањем *TCP listener*-а, који у бесконачној петљи прима конекције иницијализоване од стране клијента и примљене *HTTP* захтеве даље прослеђује на обраду нитима из *threadpool*-а 1, о којима ће више бити речено у наредном поглављу. Петља бива прекинута у току *graceful* гашења сервера које се иницира слањем *SIGINT* сигнала серверу.

```
for stream in listener.incoming() {
    //Check for termination
    let terminate = termntd_clone.lock().unwrap();
    if *terminate {
        break;
    }

    let stream = stream.unwrap();
    let repo_instance = Arc::clone(&repo);

    thread_pool.execute(move || {
        handle_connection(stream, repo_instance);
    })

}

```

Изворни код 1: Ослушкивање конекција

Http захтев се затим парсира и из њега се извлаче нопходне информације потребне за његову обраду, међу којима се налазе и ентитет који треба додати или освежити у складишту података (у случају *PUT* захтева) и *id* ентитета (у случају *GET* захтева) 2.

```rust
pub struct Entity{
    pub id: i64,
    pub description: String,
    pub value: f32
}

pub struct HttpRequest{
    pub method: HttpMethod,
    pub path: String,
    pub headers: Vec<String>,
    pub body: Option<Entity>
}


```

Изворни код 2: *Entity* и *HttpRequest* структуре података

4 *Threadpool*

Акценат овог сервера је паралелна вишенитна обрада захтева, стога је потребно обезбедити механизме за њену реализацију. Оба језика у својим стандардним пакетима садрже подршку за конкурентно програмирање, које подразумева коришћење нити, механизме за екслузиван приступ ресурсима, као и канале за размену порука између нити. Треба напоменути да *Rust* када ради са нитима ради са системским нитима, док *Go* ради са зеленим нитима, које заузимају знатно мање ресурса у односу на системске нити и о алокацији и деалокацији њима потребних ресурса брине се посебан планер. На свакој зеленој нити може се покренути једна го рутина.

Како се не би за сваки добијени захтев креирала нова нит и на тај начин могуће покренуо превелик број нити које би искориштавале превише системских ресурса и тиме угрозиле перформансе, у имплементацију се уводи *threadpool* чија је улога да на почетку рада сервера алоцира одређен број нити (овако се смањује *overhead* за креирање нити, јер се ради само на почетку рада сервера) и затим за обраду сваког од захтева посуди једну нит из њега, која се на крају обраде захтева враћа у њега и могуће ју је затим доделити некоме другом.

4.1 *Rust* имплементација

Ради манипулације унутар *threadpool*-а, нит је обмотана структуром *Worker* 3, која садржи идентификатор нити, као и *JoinHandle* структуру која омогућава приступ и манипулацију над нити. Приликом креирања *Worker*-а додељује му се идентификатор и покреће се нит која тренутно не ради ништа, већ чека на поруку од *threadpool*-а са задатком који треба да обави.

```rust
struct Worker {
    id: usize,
    thread: Option<thread::JoinHandle<()>>,
}

impl Worker{
    fn new(id: usize, job_listener:
     ↪  Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker{
       //First we spin the thread and then assign its handle
        ↪  to worker

       let thread = thread::spawn(move || loop {
           //All threads will wait for their turn to lock
            ↪  the channel and by that receive some job
         let message = job_listener.lock().unwrap().recv();

           match message {
              Ok(job ) => {
                  println!("Worker {id} got a job;
                   ↪  executing.");
                  job();
              },
               //Error will occur when job_dispatcher is
                ↪  closed
               Err(_) => {
                   println!("Worker {id} disconnected;
                    ↪  shutting down.");
                   break;
               }
           }
       });

        Worker{
            id,
            thread: Some(thread)
        }
    }
}

```

Изворни код 3: *Worker* имплементација *(Rust)*

Threadpool 4 у себи садржи вектор доступних *worker*-а, а комуницира са њима и задаје им задатке преко канала за размену порука. Будући да *Rust*-ова имплементација канала за размену порука функционише по принцпипу вишеструки произвођачи - јединствен конзумент, а *threadpool*-у је потребан обрнут модел, где он представља јединственог произвођача порука које се шаљу ка више *worker*-а, с тиме да само један *worker* може да прими исту поруку, *threadpool* приликом свог креирања креира један канал за слање порука, којег додељује самом себи и један канал за пријем порука који, да би омогућио његово коришћење свим *worker*-има у ексклузивном режиму, умотава у `Mutex<Arc>` и као таквог га додељује свим *worker*-има. Сада су *worker*-и у могућности да када добију ексклузиван приступ каналу за пријем ишчитају поруку и обраде је, где чим прочитају поруку из канала, екслузиван приступ каналу дају некоме другом.

```rust
pub struct ThreadPool{
    workers: Vec<Worker>,
    job_dispatcher: Option<mpsc::Sender<Job>>
}

impl ThreadPool{
    pub fn new(size: usize) -> Self{
        assert!(size > 0);

        let (job_dispatcher, job_listener) =
         ↪  mpsc::channel();

        let job_listener =
         ↪  Arc::new(Mutex::new(job_listener));

        let mut workers = Vec::with_capacity(size);

        for id in 0..size{
            workers.push(Worker::new(id,
             ↪  Arc::clone(&job_listener)));
        }

        ThreadPool{
            workers,
            job_dispatcher: Some(job_dispatcher)
        }
    }
}


```

Изворни код 4: *Threadpool* имплементација *(Rust)*

Threadpool задаје задатке *worker*-има тако што као поруку шаље *closure* који треба да се изврши 5.

```rust
impl ThreadPool{
    //Wraps passed closure in Box and dispatches it to
     ↪  workers
   pub fn execute<F>(&self, func: F)
   where
        //Fulfills Job type
        F: FnOnce() + Send + 'static
   {
       let job = Box::new(func);


        ↪  self.job_dispatcher.as_ref().unwrap().send(job).unwrap();
   }
}
```

Изворни код 5: *Execute* метода *threadpool*-а *(Rust)*

Веома битан аспект *threadpool*-а је његово деалоцирање заједно са деалоцирањем свих *worker*-а, које се постиже имплементирањем *Drop trait*-а 6. У тренутку када се прекине бесконачна петља која ослушкује *TCP* конекције 1, *threadpool* излази из *scope*-а и позива се његов *Drop trait* који деалоцира канал за слање порука, што за ефекат има прекид бесконачне петље унутар *worker*-а која чека на нове поруке 3. Након прекида рада свих *worker*-а, *threadpool* чека да сви *worker*-и заврше обраду захтева којег су последњег узели и тако постиже *graceful shutdown* сервера.

```rust
impl Drop for ThreadPool{
    fn drop(&mut self) {
        drop(self.job_dispatcher.take());

        for worker in &mut self.workers{
            //Waits for currently running jobs in threads to
             ↪ finish
            if let Some(thread) = worker.thread.take(){
                thread.join().unwrap();
            }
        }

    }
}
```

Изворни код 6: *Drop trait threadpool*-а *(Rust)*

4.2 *Go* имплементација

Будући да је имплементација механизма за слање порука путем канала у *Go*-у имплементирана по моделу јединствен произвођач-вишеструки конзументи , која директно належе на потребе комуникационог модела којим се *threadpool* користи, имплементација *threadpool*-а у *Go*-у је нешто једноставнија.

У овој имплементацији нема потребе за креирањем специјалних структура података за *threadpool* и *worker*-е, већ је довољно направити само један канал кроз који ће се преносити пристигле *TCP* конекције као и *wait* група која ће се постарати за безбедно прекидање рада нити приликом *graceful shutdown*-а. 7

```go
connChan := make(chan net.Conn)
wg := sync.WaitGroup{}
```

Изворни код 7: Иницијализација канала и *wait* групе *(Go)*

Затим се покреће одређен број го рутина који одговара величини *threadpool*-а. Њима је приликом креирања прослеђен канал за размену порука којег све оне ослу-

шкују и у тренутку када из њега приме *TCP* конекцију одмах је обрађују.8

```
for i := 0; i < poolSize; i++ {
  wg.Add(1)
  go func(threadNum int) {
    defer wg.Done()

    for {
      select {
      case conn, ok := <-connChan:
        {
          if !ok {
            return
          }
          fmt.Printf("Thread %d handles request\n",
          ↪  threadNum)
          handleConnection(conn, repo, mapMux)
        }
      }
    }
  }(i)
}

```

Изворни код 8: Покретање го рутина *(Go)*

Задаци се прослеђују го рутинама тако што се унутар бесконачне петље која ослушкује нове конекције при пристизању нове конекције она проследи у канал за слање порука.9

```
go func() {
  for {
    conn, err := listener.Accept()
    if err != nil {
      fmt.Println("Error accepting:", err)
      continue
    }

    //Check for termination
    shutdownServerFlag.Lock()
    if shutdownServerFlag.close {
      close(connChan)
      shutdownServerFlag.Unlock()
      return
    } else {
      shutdownServerFlag.Unlock()
    }

    //Dispatch request to thread pool
    connChan <- conn
  }
}()

```

Изворни код 9: Прослеђивање конекција го рутинама *(Go)*

Приликом прекидања бесконачне петље 9, канал за слање порука се затвара што сигнализира свим го рутинама да прекину са својим радом и чека се да се све рутине заврше захваљујући *wait* групи. На овај начин постиже се *graceful shutdown* сервера.

5 Конкурентан приступ складишту података

У претходном поглављу описано је како се постиже паралелна обрада више захтева, помоћу нити и *threadpool*-а. Након тога потребно је омогућити конкурентан приступ складишту података, како оно не би представљало уско грло паралелног рада нити и њихов паралелан рад ипак претворило у секвенцијалан. Складиште података за потребе овог сервера представљаће мапа чији ће кључ бити идентификатор ентитета, а вредност читав ентитет.

Пошто је потребно имплементирати *GET* и *PUT* методе, захтеви се могу издвојити у специфичне случајеве приступања подацима у мапи:

1. Читање података приликом *GET*-а које не захтева никакав вид ексклузивног приступа подацима, ни на нивоу мапе, нити на нивоу појединачног елемента мапе.

2. *PUT* захтев са ентитетом чији се идентификатор не налази у скупу кључева мапе захтева закључавање читаве мапе како би се додао нови кључ са одговарајућом вредношћу.

3. *PUT* захтев са ентитетом чији се идентификатор налази у скупу кључева мапе захтева закључавање само елемента мапе чији кључ одговара кључу ентитета којег желимо да упишемо.

5.1 *Rust* имплементација

Складиште података моделовано је *Repo* 10 структуром која у себи садржи мапу са кључем `i64` који представља идентификатор ентитета и вредношћу `Arc<RwLock<Entity>` која представља енетитет. `RwLock` представља структуру која омогућава истовремени приступ подацима у случају читања, а ексклузиван приступ у случају измене података, што омогућава већу флексибилност приликом приступања подацима. На исти начин како је обмотана вредност мапе у `Arc<RwLock>` обмотава се и читава *Repo* структура, како би се омогућио ексклузиван приступ на нивоу читаве мапе, као и на нивоу појединачног елемента мапе.

```rust
pub struct Repo{
    entities: HashMap<i64,Arc<RwLock<Entity>>>
}

let repo = Arc::new(RwLock::new(Repo::new()));


```

Изворни код 10: *Repo* структура и његова заштита од истовремених уписа *(Rust)*

Први случај 5 решава се постављањем *read lock*-а на нивоу мапе и елемента 11.

```rust
impl Repo{
    pub fn get_by_id(&self, id: i64) ->  Option<Entity>{
        match self.entities.get(&id){
          Some(entity_lock) => {
              let entity = entity_lock.read().ok()?;

              Some((*entity).clone())
           },
          None => None
        }
    }
}

{
...
if let Ok(ro_repo) = repo.read(){
     match ro_repo.get_by_id(id){
        Some(entity) => {
            let json_entity =
             ↪  serde_json::to_string(&entity).unwrap();
            let ok_resp = "HTTP/1.1 200 OK \r\n";
            let content_len = json_entity.len();

            let response = format!(
                "{ok_resp}Content-Length:
                 ↪  {content_len}\r\nContent-Type:
                 ↪  application/json\r\nConnection:
                 ↪  close\r\n\r\n{json_entity}"
            );

            stream.write_all(response.as_bytes()).unwrap();
        }
        None =>{
            let not_found_resp = "HTTP/1.1 404 NotFound
             ↪  \r\n\r\n";
            stre-
             ↪  am.write_all(not_found_resp.as_bytes()).unwrap();
            return;
        }
    };
}
...
}

```


Изворни код 11: Конкурентно решење првог случаја 5 (Rust)

Други случај 5 решава се постављањем *write lock*-а на нивоу мапе 12.

```
{
...
// Doesnt exist: lock whole map and add
let mut w_repo = repo.write().unwrap();
let new_id = body.id;
let new_ent =  Arc::new(RwLock::new(body));
w_repo.entities.insert(new_id, new_ent);
...
}
```

Изворни код 12: Конкурентно решење другог случаја 5 (Rust)

Трећи случај 5 решава се постављањем *read lock*-а на нивоу мапе, како би се добавио елемент мапе, и постављањем *write lock*-а на нивоу елемента, како би се омогућио ексклузиван приступ елементу приликом његове измене 13.

```
{
...
if exists {
    //just mutate entry without locking whole map
    if let Some(ro_repo) = repo.read().ok() {
        let entry = ro_repo.entities.get(&body.id).unwrap();

        //Locking and changing entity inside repo
        if let Some(mut rw_entity) = entry.write().ok() {
            *rw_entity = body;
        }else{
            let response = "HTTP/1.1 500 InternalServerError
             ↪  \r\n\r\n";
            stream.write_all(response.as_bytes()).unwrap();
            return;
        }
    } else {
        let response = "HTTP/1.1 500 InternalServerError
         ↪  \r\n\r\n";
        stream.write_all(response.as_bytes()).unwrap();
        return;
    }
} else {
...
}
```

Изворни код 13: Конкурентно решење трећег случаја 5 (Rust)

5.2 *Go* имплементација

Будући да *Go* не поседује концепт власничког система типова, није могуће умотати *Repo* структуру као и вредност мапе у нешто попут `Arc<RwLock>`, већ се проблему мора приступити мало другачије 14. *Repo* структура моделује се на сличан начин, као омотач око мапе, чији је кључ идентификатор ентитета `int64`, са разликом у вредности мапе која је моделована као структура која садржи сам ентитет, али и структуру `RWMutex` која омогућава ексклузиван или истовремен приступ неком делу кода. Такође, креира се додатан `RWMutex` који ће се користити за ексклузиван приступ на нивоу читаве мапе. На овај начин у могућности смо да сваки пут када нам треба ексклузиван приступ неком елементу или читавој мапи закључамо спе-

цифичан `RWMutex` који је везан за тај елемент или мапу и тако уколико неко други покуша да измени елемент или мапу, прво ће покушати да закључа исти `RWMutex` који је претходно већ закључан што ће га натерати да чека док тренутна го рутина не откључа `RWMutex`.

```
type Repo struct {
        Entries map[int64]MapEntry
}


type MapEntry struct {
        Mux    *sync.RWMutex
        Entity Entity
}

func main(){
...
        repo := NewRepo()
        mapMux := &sync.RWMutex{}
...
}

```

Изворни код 14: Неопходне структуре за конкурентан приступ мапи (Go)

Први случај 5 решава се *read lock*-овањем `RWMutex`-а мапе и *read lock*-овањем `RWMutex`-а елемента 15.

```
{
...
mapMux.RLock()
entry, ok := repo.Entries[id]


entry.Mux.RLock()
jsonBytes, err := json.Marshal(entry.Entity)
entry.Mux.RUnlock()

mapMux.RUnlock()
...
}
```

Изворни код 15: Конкурентно решење првог случаја 5 (Go)

Други случај 5 решава се *write lock*-овањем `RWMutex`-а мапе 16.

```
{
  ...
  //Entry doesn't exist -> lock whole map
  mapMux.Lock()
  repo.Entries[request.Body.Id] =
   ↪  *NewMapEntry(*request.Body)
  mapMux.Unlock()
  ...
}

```

Изворни код 16: Конкурентно решење другог случаја 5 (Go)

Трећи случај 5 решава се *read lock*-овањем `RWMutex`-а мапе и *write lock*-овањем `RWMutex`-а елемента 17.

```
{
...
        if exists {
                mapMux.RLock()
                entry, _ := repo.Entries[request.Body.Id]
                entry.Mux.Lock()
                repo.Entries[request.Body.Id] =
                 ↪  *NewMapEntryWMux(*request.Body,
                 ↪  entry.Mux)
                entry.Mux.Unlock()
                mapMux.RUnlock()
        }
...
}
```

Изворни код 17: Конкурентно решење трећег случаја 5 (Go)

Битно је напоменути да *Go* унутар *sync* пакета већ има имплементирану мапу која се брине о конкурентном приступу, али како су унутар *Rust* имплементације коришћене само примитиве језика, тако је одрађено и овде.

6 *Stress test* рада сервера

До сада, фокус је био на имплементацији сервера, али ни у једном тренутку није извршена провера да ли обе имплементације функционишу како је специфицирано и да ли у неким случајевима улазе у недефинисана стања и доводе до пада сервера. У оквиру овог поглавља биће описан начин на који је извршена верификација правилног рада обе имплементације.

За потребе тестирања коришћен је алат *Apache JMeter* [2]који омогућава дефинисање тест сценариа за тестирање сервера, који може да обухвати велики број паралелних захтева ка серверу, као и да омогући накнадну анализу извршеног сценариа.

За потребе тестирања тренутних имплементација дефинисан је сценарио од:

- 10000 паралелних *GET* захтева, где сваки захтев чита различит елемент мапе.
- 10000 паралелних *PUT* захтева, где сваки захтев уписује елемент мапе који до тада није постојао у мапи.
- 20000 паралелних *PUT* захтева, где свака два захтева покушавају да измене исти елемент који већ постоји у мапи.

Овим сценарином успевамо да тестирамо све случајеве конкурентног приступа наведених у поглављу 5. Конкретан фајл са конфигурацијом сценариа може се наћи на путањи `https://github.com/stojanovic00/rust-go-server-comp/blob/main/profiling/stress_testing/rust_testing.jmx`. Подаци који су се користили за тестирање генерисани су наменском *Go* скриптом `test_data_generator` која генерише насумичне вредности за *descritpion* и *value* поља ентитета за задати број ентитета, док поља *id* инкрементира за један почевши од нуле. Код `test_data_generator` скрипте може се наћи на путањи `https://github.com/stojanovic00/rust-go-server-c` `tree/main/profiling/stress_testing/test_data_generator`

Табеле са резултатима приказане на сликама 1 и 2 јасно приказују да је свих 40000 захтева упућених ка оба сервера обрађено са 0% грешака (колона *Error %*). Остале метрике приказане у табели нису толико меродавне због тога што је *Apache JMeter* намењен за тестирање у окружењу где постоји тест сервер и сервер на коме се покреће *Apache JMeter*, док је за потребе овог тестирања *Apache JMeter* покретан на истом серверу на којем се налазе и имплементације сервера, али за потребе верификације исправног рада имплементација сервера покретање *Apache JMeter* у оваквом окружењу је оправдано.

Label	# Samples	Error %	Throughput	Received ...	Sent KB/sec
PUT new data	10000	0.00%	947.0/sec	35.14	228.96
GET data	10000	0.00%	929.2/sec	137.07	105.16
PUT update data	20000	0.00%	832.5/sec	30.89	201.29
TOTAL	40000	0.00%	882.3/sec	57.09	184.95

Слика 1: Резултати тестирања *Rust* сервера

Label	# Samples	Error %	Throughput	Received ...	Sent KB/sec
PUT new data	10000	0.00%	1503.5/sec	55.80	363.52
GET data	10000	0.00%	1533.7/sec	224.76	173.58
PUT update data	20000	0.00%	1838.2/sec	68.22	444.45
TOTAL	40000	0.00%	1663.3/sec	107.23	348.67

Слика 2: Резултати тестирања *Go* сервера

7 Поређење перформанси

За мерење перформанси, услед недостатка адекватних алата, кориштен је скуп оркестрираних наменских скрипти и алата. За мерење заузећа меморије и процесора кориштен је *pidstat CLI* алат [4], док је за остале параметре кориштен *Apache Benchmark* [1]. Тестирање започиње покретањем сервера и добављањем идентификатора његовог процеса који се затим прослеђује двема инстанцама *pidstat*-а (једна за процесор друга за меморију). Затим се покреће *Apache Benchmark* и након завршетка његовог рада, сви програми се гасе и своје податке чувају у одређеним фајловима. Наменска *Go* скрипта затим учитава све генерисане фајлове, парсира битне податке и уписује их у једну линију *CSV* фајла. Наведени кораци описани су унутар *SHELL* скрипте која као улазне параметре прима величину *threadpool*-а, број захтева и број паралелних конекција. Да би се максимално аутоматизовао процес, ова скрипта се позива више пута са различитим параметрима унутар још једне *SHELL* скрипте. Добијени резултати агрегирани су и приказани у табелама 1 и 2, где су називи колона, због недостатка простора, редуковани, али у тексту испод налази се легенда за њихово тумачење. Све горе наведене скрипте као и оригинални агрегирани подаци могу се пронаћи на путањи `https://github.com/stojanovic00/rust-go-server-comp/tree/main/profiling/shell_profiling`.

- lang - language
- psz - pool size
- reqs - requests
- conns - connections
- avgcpu - avg cpu[%]
- maxcpu - max cpu[%]
- avgmem - avg mem[%]
- maxmem - max mem[%]
- ttltstt - total test time[s]
- preqmt - per request mean time[ms]
- trrc - transfer rate rcvd[kB/s]
- trs - transfer rate sent[kB/s]
- connlat - connection latency[ms]

- connproc - connection processing time[ms]

lang	psz	reqs	conns	avgcpu	maxcpu	avgmem	maxmem	ttltstt	preqmt	trrc	trs	connlat	connproc
golang	2	100000	10	116.556	119.000	0.117	0.120	9.215	0.922	1345.860	0.000	0.000	1.000
golang	4	100000	10	119.100	129.000	0.113	0.120	10.408	1.041	1191.650	0.000	0.000	1.000
golang	8	100000	10	120.608	132.000	0.113	0.120	11.418	1.142	1086.200	0.000	0.000	1.000
golang	16	100000	10	123.100	131.000	0.111	0.120	9.920	0.992	1250.180	0.000	0.000	1.000
golang	32	100000	10	117.572	132.000	0.111	0.120	12.000	1.200	1033.550	0.000	0.000	1.000
golang	64	100000	10	120.800	125.000	0.110	0.110	10.554	1.055	1175.080	0.000	0.000	1.000
golang	2	100000	100	103.889	107.000	0.101	0.110	9.795	9.795	1266.150	0.000	4.000	6.000
golang	4	100000	100	108.667	111.000	0.100	0.100	9.245	9.245	1341.550	0.000	4.000	5.000
golang	8	100000	100	110.444	113.000	0.100	0.100	9.313	9.313	1331.750	0.000	4.000	5.000
golang	16	100000	100	103.200	114.000	0.115	0.120	10.687	10.687	1160.460	0.000	5.000	6.000
golang	32	100000	100	104.900	115.000	0.109	0.110	10.142	10.142	1222.850	0.000	5.000	5.000
golang	64	100000	100	108.900	116.000	0.113	0.120	10.176	10.176	1218.770	0.000	5.000	5.000
golang	2	100000	1000	94.700	100.000	0.104	0.110	9.987	99.868	1241.870	0.000	42.000	57.000
golang	4	100000	1000	91.333	95.000	0.112	0.120	9.623	96.232	1288.800	0.000	43.000	53.000
golang	8	100000	1000	89.111	92.000	0.118	0.120	9.872	98.717	1256.350	0.000	45.000	53.000
golang	16	100000	1000	87.600	91.000	0.128	0.130	10.106	101.063	1227.190	0.000	47.000	53.000
golang	32	100000	1000	90.300	96.000	0.115	0.120	10.008	100.076	1239.290	0.000	47.000	53.000
golang	64	100000	1000	92.100	98.000	0.114	0.120	9.916	99.155	1250.800	0.000	47.000	52.000
rust	2	100000	10	99.000	102.000	0.040	0.040	7.000	0.700	1813.550	0.000	0.000	0.000
rust	4	100000	10	106.429	108.000	0.030	0.030	7.468	0.747	1699.870	0.000	0.000	0.000
rust	8	100000	10	117.000	118.000	0.030	0.030	7.702	0.770	1648.270	0.000	0.000	0.000
rust	16	100000	10	119.143	121.000	0.040	0.040	7.723	0.772	1643.810	0.000	0.000	0.000
rust	32	100000	10	122.000	123.000	0.040	0.040	7.685	0.769	1651.930	0.000	0.000	0.000
rust	64	100000	10	124.571	126.000	0.060	0.060	7.661	0.766	1657.210	0.000	0.000	0.000
rust	2	100000	100	96.429	103.000	0.030	0.030	7.087	7.087	1791.450	0.000	3.000	4.000
rust	4	100000	100	107.286	108.000	0.030	0.030	7.084	7.084	1792.080	0.000	3.000	4.000
rust	8	100000	100	118.429	120.000	0.040	0.040	7.378	7.378	1720.800	0.000	3.000	4.000
rust	16	100000	100	123.000	124.000	0.020	0.020	7.427	7.427	1709.430	0.000	4.000	4.000
rust	32	100000	100	124.857	126.000	0.040	0.040	7.633	7.633	1663.280	0.000	4.000	4.000
rust	64	100000	100	119.750	122.000	0.060	0.060	7.972	7.972	1592.430	0.000	4.000	4.000
rust	2	100000	1000	94.286	102.000	0.030	0.030	7.246	72.465	1751.930	0.000	37.000	35.000
rust	4	100000	1000	103.286	112.000	0.040	0.040	7.235	72.351	1754.690	0.000	37.000	35.000
rust	8	100000	1000	107.714	122.000	0.030	0.030	7.524	75.241	1687.280	0.000	41.000	34.000
rust	16	100000	1000	109.000	113.000	0.020	0.020	7.800	78.005	1627.510	0.000	41.000	36.000
rust	32	100000	1000	114.571	124.000	0.030	0.030	7.754	77.542	1637.230	0.000	42.000	36.000
rust	64	100000	1000	116.857	119.000	0.069	0.070	7.749	77.493	1638.240	0.000	42.000	35.000

Табела 1: Поређење перформанси *GET* захтева

lang	psz	reqs	conns	avgcpu	maxcpu	avgmem	maxmem	ttltstt	preqmt	trrc	trs	connlat	connproc
go	2	100000	10	127.375	129.000	0.120	0.120	8.405	0.841	441.510	2393.430	0.000	0.000
go	4	100000	10	129.000	131.000	0.105	0.110	8.476	0.848	437.840	2373.550	0.000	0.000
go	8	100000	10	136.625	138.000	0.129	0.130	8.691	0.869	426.980	2314.670	0.000	1.000
go	16	100000	10	137.330	139.000	0.100	0.100	8.723	0.872	425.400	2306.090	0.000	1.000
go	32	100000	10	138.500	140.000	0.115	0.120	8.720	0.872	425.570	2307.030	0.000	1.000
go	64	100000	10	138.875	141.000	0.100	0.110	8.791	0.879	422.130	2288.370	0.000	1.000
go	2	100000	100	118.750	119.000	0.116	0.120	8.877	8.877	418.050	2266.290	4.000	5.000
go	4	100000	100	128.000	129.000	0.112	0.120	8.668	8.668	428.110	2320.800	4.000	5.000
go	8	100000	100	131.125	135.000	0.115	0.120	8.675	8.675	427.780	2318.990	4.000	5.000
go	16	100000	100	133.125	137.000	0.114	0.120	8.576	8.576	432.720	2345.780	4.000	5.000
go	32	100000	100	132.125	134.000	0.101	0.110	8.589	8.589	432.040	2342.120	4.000	4.000
go	64	100000	100	126.000	128.000	0.129	0.130	9.229	9.229	402.100	2179.780	4.000	5.000
go	2	100000	1000	120.625	122.000	0.117	0.120	8.732	87.317	425.000	2303.920	36.000	51.000
go	4	100000	1000	116.500	119.000	0.111	0.120	8.674	86.745	427.800	2319.120	38.000	49.000
go	8	100000	1000	115.375	117.000	0.100	0.100	8.527	85.268	435.210	2359.280	39.000	46.000
go	16	100000	1000	115.375	118.000	0.114	0.120	8.535	85.347	434.810	2357.110	40.000	45.000
go	32	100000	1000	117.605	120.000	0.086	0.090	8.565	85.653	433.250	2348.680	40.000	45.000
go	64	100000	1000	118.875	123.000	0.125	0.130	8.592	85.918	431.920	2341.440	41.000	44.000
rust	2	100000	10	112.429	115.000	0.020	0.020	7.146	0.715	519.280	2815.030	0.000	0.000
rust	4	100000	10	124.143	126.000	0.020	0.020	7.514	0.751	493.900	2677.450	0.000	0.000
rust	8	100000	10	127.375	140.000	0.040	0.040	8.653	0.865	428.860	2324.870	0.000	1.000
rust	16	100000	10	138.000	144.000	0.040	0.040	8.231	0.823	450.850	2444.090	0.000	0.000
rust	32	100000	10	145.286	148.000	0.060	0.060	7.781	0.778	476.930	2585.460	0.000	0.000
rust	64	100000	10	146.286	148.000	0.060	0.060	7.844	0.784	473.080	2564.570	0.000	0.000
rust	2	100000	100	103.571	107.000	0.020	0.020	7.614	7.614	487.380	2642.120	3.000	4.000
rust	4	100000	100	115.143	118.000	0.040	0.040	7.666	7.666	484.060	2624.120	3.000	4.000
rust	8	100000	100	123.500	129.000	0.020	0.020	8.014	8.014	463.070	2510.350	4.000	4.000
rust	16	100000	100	133.286	135.000	0.040	0.040	7.854	7.854	472.510	2561.510	4.000	4.000
rust	32	100000	100	133.125	137.000	0.040	0.040	8.160	8.160	454.740	2465.190	4.000	4.000
rust	64	100000	100	122.889	136.000	0.060	0.060	9.751	9.751	380.550	2063.000	5.000	5.000
rust	2	100000	1000	97.000	100.000	0.020	0.020	7.790	77.904	476.350	2582.320	40.000	37.000
rust	4	100000	1000	107.851	112.000	0.030	0.030	7.879	78.788	471.000	2553.320	41.000	37.000
rust	8	100000	1000	107.750	116.000	0.040	0.040	8.326	83.264	445.680	2416.080	43.000	40.000
rust	16	100000	1000	110.300	133.000	0.035	0.080	9.867	98.668	376.100	2038.870	52.000	46.000
rust	32	100000	1000	116.125	131.000	0.040	0.040	8.594	85.944	431.790	2340.740	47.000	39.000
rust	64	100000	1000	122.000	131.000	0.060	0.060	8.335	83.354	445.200	2413.470	45.000	38.000

Табела 2: Поређење перформанси *PUT* захтева

На основу добијених података можемо донети одређене закључке:

- Повећањем величине *threadpool*-а у обе имплементације долази до повећања искоришћених ресурса процесора.
- *Rust* користи убедљиво мање ресурса процесора, све док величина *threadpool*-а не премаши број системских нити, где тада вођство преузима *Go*. Ова појава може се приписати томе да *Rust* користи системске нити, док *Go* користи зелене нити за покретање својих го рутина.
- За фиксну величину *threadpool*-а, повећањем броја паралелних конекција у обе имплементације долази до смањења искоришћених ресурса процесора.
- Повећањем величине *threadpool*-а у обе имплементације долази до повећања искоришћених меморијских ресурса, с тиме да је релативно повећање меморије драстичније у *Rust* имплементацији.
- *Rust* у свакој ситуацији користи знатно мање меморијских ресурса.
- *Rust* имплементација у сваком случају има брже просечно време одговора на захтев.
- *Rust* имплементација готово увек има већи *transfer rate*, како слања тако и примања података
- Преласком на 1000 паралелних конекција знатно се повећава латенција и смањује брзина обраде конекције, где *Go* има нижу латенцију, али и мању брзину обраде конекције.
- Као што је и очекивано, *PUT* захтев захтева више ресурса за његову обраду.

8 Закључак

У закључку овог истраживања различитих аспеката конкурентног програмирања у програмским језицима Go и Rust на примеру вишенитног сервера, могу се извести одрђени закључци.

Оба програмска језика нуде подршку за конкурентно програмирање, али се различито сналазе у различитим аспектима овог домена. *Go* се истиче по лакшем имплементирању и употреби *threadpool-a* и боље се сналази у случају повећања његове величине. С друге стране, *Rust* пружа интуитивније и безбедније механизме за имплементацију конкурентног приступа складишту података, чиме се смањује ризик од грешака приликом рада са дељеним подацима.

Када је у питању учинак, резултати су комплексни и зависе од конкретних захтева апликације. *Go* се показује као бољи избор у ситуацијама када је потребно повећати величину threadpool-a и задржати заузеће процесорске моћи у дозвољеним границама, док га *Rust* у свим осталим аспектима надмашује. Овим се може закључити да мало изазовнији начин програмирања у *Rust*-у, иако некада главоболан, на самом крају награђује завидним перформансама.

Библиографија

[1] Apache benchmark. https://httpd.apache.org/docs/2.4/programs/ab.html, 2023. Последњи приступ: 23.12.2023.

[2] Apache jmeter. https://jmeter.apache.org/, 2023. Последњи приступ: 22.12.2023.

[3] Mozilla research. https://research.mozilla.org/, 2023. Последњи приступ: 22.12.2023.

[4] pidstat. https://man7.org/linux/man-pages/man1/pidstat.1.html, 2023. Последњи приступ: 23.12.2023.