УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У НОВОМ САДУ

Марко Николић

Поређење програмских језика Rust и C++ на примеру имплементације асинхроног кода

СЕМИНАРСКИ РАД

Нови Сад, 2025.

САДРЖАЈ

1. УВОД

Асинхроно програмирање постаје кључни концепт у развоју софтверских система који захтевају висок ниво ефикасности и скалабилности. Асинхроно програмирање омогућава програму да истовремено обавља више операција без блокирања главне нити извршавања, што резултује бољим коришћењем ресурса и смањењем времена обраде.

Ова техника је посебно важна у окружењима где се извршава велики број независних задатака који могу бити условљени чекањем на спољне сервисе или на друге дуготрајне процесе. Асинхроне функције омогућавају да се ове операције извршавају без непотребног чекања и оптерећења процесора.

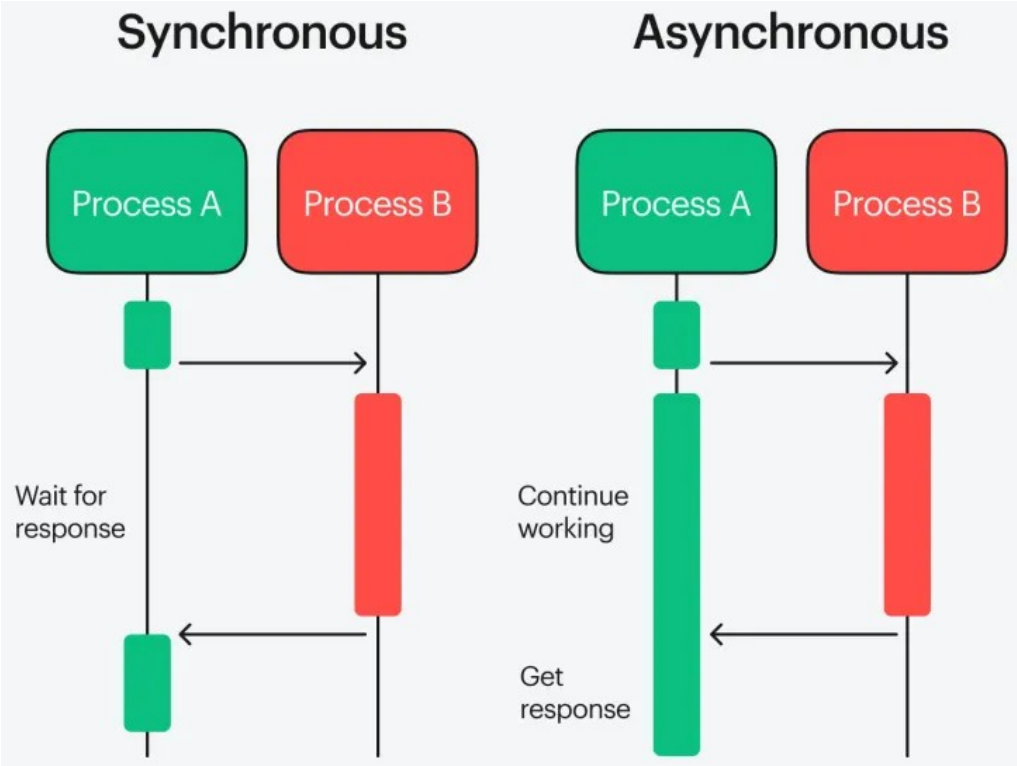
Циљ овог рада је упоређивање приступа асинхроном програмирању у два популарна програмска језика: *Rust* и *C++*. У наставку рада биће приказане имплементације истог сценарија у оба језика. Рад ће упоредити синтаксу, структуру кода и начин како сваки језик решава управљање асинхроним задацима.

2. АСИНХРОНО ПРОГРАМИРАЊЕ

2.1 Концепт асинхроног програмирања

У асинхроном програмирању, код је организован тако да један задатак може чекати спољни ресурс, док остали задаци настављају са извршавањем, чиме се повећава ефикасност. На овај начин, нит или процес могу наставити са извршавањем других задатака док се чека одговор спољног ресурса, као што је *API* позив или завршетак дуготрајне операције. Овај приступ је нарочито важан у модерним апликацијама које обрађују велики број захтева или интерагују са спољним сервисима, јер омогућава ефикасније коришћење доступних ресурса.

За разлику од асинхроног, синхрони код блокира извршавање програма све док задатак не буде завршен. На пример, у синхроном приступу, сваки *API* позив мора да се заврши пре него што програм пређе на следећи. Ово може резултирати непотребним чекањем и смањењем укупне ефикасности програма, посебно када се обрађује велики број независних захтева.



Слика 1 - Разлика између синхроног и асинхроног кода

2.2 Асинхроно програмирање у програмском језику *C++*

Стандардни *C++* нуди основне механизме за асинхрону обраду задатака, који се базирају на *std::async*, *std::future* и *std::promise*. Ови алати омогућавају покретање функција у позадини и добијање резултата када они постану доступни, без потребе за ручним управљањем нитима.

std::async је функција у стандардном *C++*-у која омогућава покретање функције асинхроно, односно у позадини, без блокирања нити која је позвала функцију. Позив *std::async(std::launch::async, func, args...)* ствара нову нит која одмах извршава задатак, док опција *std::launch::deferred* одлаже извршење функције све док се резултат не затражи преко *future.get()*. *std::async* увек враћа *std::future*.

std::future је класа која представља будући резултат асинхроног задатка. Она омогућава једној нити да сачека и преузме резултат који ће бити доступан тек када се задатак заврши. Метода *get()* блокира нит која је позвала све док резултат не буде спреман, након чега враћа вредност.

std::promise је класа која омогућава једној нити да постави вредност која ће бити доступна другој нити преко *std::future*. Тело асинхроног задатка или друге нит може поставити резултат коришћењем методе *set_value()*, а нит која поседује повезани *std::future* може потом да позове *get()* да преузме ту вредност.

Код на слици 2 приказује пример асинхроног извршавања у стандардном *C++*-у користећи *std::async* и *std::future*. Функција *compute* се покреће у позадинској нити, док главна нит наставља са другим радом. Позивом *result.get()* главна нит чека само ако задатак још није завршен и потом преузима резултат из функције *compute*.

```cpp
#include <iostream>
#include <future>
#include <thread>
#include <chrono>

int compute(int x) {
    std::this_thread::sleep_for(std::chrono::seconds(2));
    return x * 2;
}

int main() {
    std::cout << "Starting async task...\n";

    std::future<int> result = std::async(std::launch::async, compute, 21);

    std::cout << "Doing other work while task runs...\n";

    int value = result.get();
    std::cout << "Received result = " << value << "\n";
}
```

Слика 2 - Пример коришћења *std::async* и *std::future*

За сложеније сценарије, као што су управљање великим бројем асинхронхних позива и *event-driven* архитектуре, могу се користити напредне библиотеке попут *Boost.Asio*. *Boost.Asio* пружа низ алата за асинхроно програмирање, укључујући *io_context*, асинхроне тајмере (*steady_timer*) и ко-рутине, које омогућавају да се код пише на сличан начин као синхрони, али да извршавање задатака буде асинхроно.

У оквиру *Boost.Asio* библиотеке, функција *io_context::run()* представља централни механизам за управљање асинхроним задацима и може се посматрати као *event loop* који омогућава реализацију *event-driven* архитектуре. Када се у програму покрене ко-рутина помоћу *co_spawn*, она се региструје унутар *io_context*, али се њено извршавање суспендује на местима где се користи *co_await* над асинхроним операцијама, као што су *async_connect*, *async_read* или *steady_timer::async_wait*. У тим тренуцима нит није блокирана, већ *io_context* наставља да обрађује друге задатке или да чека сигнале од оперативног система. Када оперативни систем јави да је нека операција завршена, резултат догађаја се смешта у унутрашњи ред *io_context*. Током извршавања i*o_context::run()*, догађај се преузима из реда и ко-рутина која је била суспендована наставља своје извршавање од места *co_await*. На овај начин се добија илузија да је функција блокирана, док је у стварности нит остала слободна и ефикасно искоришћена за паралелно управљање великим бројем асинхроних задатака.

awaitable<T> представља резултат асинхроног задатка који може да се сачека коришћењем *co_await*. Он омогућава писање асинхроног кода на синхрони начин, функција изгледа као да блокира док чека, али нит заправо није блокирана. *awaitable* је кључни део ко-рутина у *Boost.Asio* и користи се као повратни тип за функције које ће се извршавати асинхроно.

co_spawn се користи за регистровање ко-рутине унутар *io_context*. Он не креира нову нит, већ само каже *event loop*-у да треба управљати тим задатком.

co_await се користи за суспензију ко-рутине док асинхрони задатак не заврши. За разлику од класичног *future.get()*, *co_await* не блокира нит, већ само суспендује текућу ко-рутину. Када резултат буде доступан, ко-рутине настављају из места суспензије.

steady_timer омогућава асинхроно чекање одређеног временског периода. У комбинацији са *co_await timer.async_wait()*, тајмер суспендује извршавање ко-рутине без блокирања нити. Када тајмер истекне, ко-рутине настављају са извршавањем. Ово је корисно за симулацију кашњења.

Код на слици 3 приказује основни пример асинхроног извршавања у *Boost.Asio* користећи ко-рутине и *co_spawn*. Функција *sayHello* симулира задатак који чека 500 милисекунди користећи *steady_timer*, а затим исписује поруку. Позивом *co_spawn* ко-рутине се региструју у *io_context*, који управља њиховим извршавањем унутар *event loop*-а. Када ко-рутине наиђу на *co_await timer.async_wait()*, њихово извршавање се суспендује, али нит није блокирана и може наставити са другим ко-рутинама. Након што тајмер истекне, ко-рутине настављају из места суспензије и исписују поруку.

```cpp
#include <boost/asio.hpp>
#include <boost/asio/steady_timer.hpp>
#include <boost/asio/awaitable.hpp>
#include <boost/asio/co_spawn.hpp>
#include <boost/asio/detached.hpp>
#include <boost/asio/use_awaitable.hpp>
#include <iostream>
using namespace std::chrono_literals;
using namespace boost::asio;

awaitable<void> sayHello(int id) {
    steady_timer timer(co_await this_coro::executor);
    timer.expires_after(500ms);
    co_await timer.async_wait(use_awaitable);

    std::cout << "Hello from task " << id << "\n";
}

int main() {
    io_context ctx;

    for (int i = 1; i <= 3; ++i) {
        co_spawn(ctx, sayHello(i), detached);
    }

    ctx.run();

    std::cout << "All tasks scheduled\n";
}
```

Слика 3 - Пример коришћења *Boost.Asio*

2.3 Асинхроно програмирање у програмском језику *Rust*

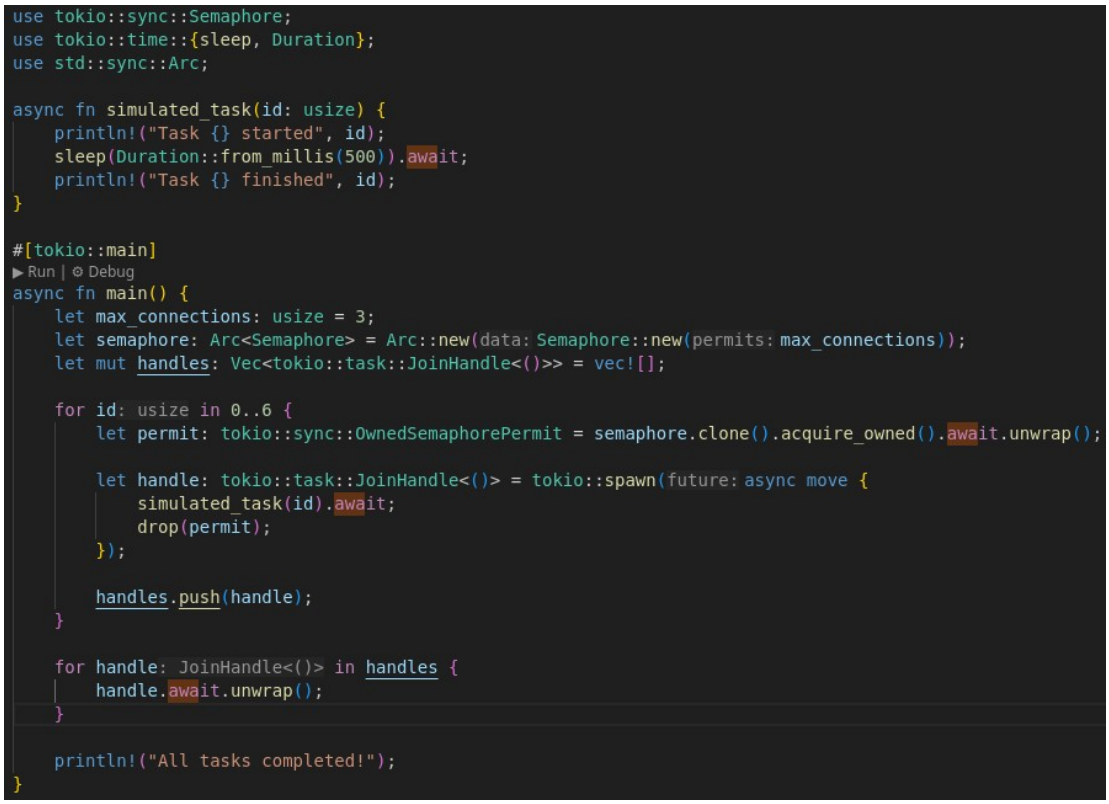
У *Rust*-у се асинхроно програмирање реализује преко *async* функција, *.await* и асинхроног *runtime*-а, као што је *Tokio*.

Асинхроне функције (*async fn*) представљају задатке који се могу суспендовати и наставити касније, без блокирања нити у којој се извршавају. Када се позове *.await* на асинхроној функцији, тренутна асинхрона ко-рутина се суспендује док резултат задатка није доступан, али нит која их покреће остаје слободна да извршава друге задатке.

Tokio runtime у *Rust*-у представља окружење за извршавање асинхроног кода заснованог на *async/await* механизму. Он функционише као *task scheduler* који користи *event loop* да би обрадио велики број истовремених операција без потребе за креирањем великог броја нити. Када се покрене асинхрона функција, *Tokio* је не извршава одмах до краја, већ је региструје као задатак који може да се паузира када наиђе на *await* и настави када су подаци спремни. На тај начин, док једна операција чека, *runtime* пребацује процесор на други задатак, чиме се постиже висока искоришћеност ресурса.

Semaphore у *Tokio*-у (*tokio::sync::Semaphore*) омогућава ограничавање броја задатака који могу истовремено да приступе неком ресурсу. Када задатак заврши, *permit* се ослобађа и други задатак из реда чекања може да почне и да користи ресурс.

Код на слици 4 приказује једноставан пример асинхроног извршавања у *Rust*-у користећи *Tokio runtime* и *async/.await*. Функција *simulated_task* представља задатак који се извршава асинхроно, док главна функција покреће више задатака истовремено користећи *tokio::spawn*. Семафор ограничава број паралелних задатака на 3, тако да само одређени број задатака може истовремено да се извршава. Када задатак заврши, ресурс у семафору се ослобађа, омогућавајући другом задатку да почне. На тај начин се постиже паралелно извршавање задатака без блокирања главне нити и уз контролисан број истовремених извршења.



```rust
use tokio::sync::Semaphore;
use tokio::time::{sleep, Duration};
use std::sync::Arc;

async fn simulated_task(id: usize) {
    println!("Task {} started", id);
    sleep(Duration::from_millis(500)).await;
    println!("Task {} finished", id);
}

#[tokio::main]
▶ Run | ⚙ Debug
async fn main() {
    let max_connections: usize = 3;
    let semaphore: Arc<Semaphore> = Arc::new(data: Semaphore::new(permits: max_connections));
    let mut handles: Vec<tokio::task::JoinHandle<()>> = vec![];

    for id: usize in 0..6 {
        let permit: tokio::sync::OwnedSemaphorePermit = semaphore.clone().acquire_owned().await.unwrap();

        let handle: tokio::task::JoinHandle<()> = tokio::spawn(future: async move {
            simulated_task(id).await;
            drop(permit);
        });

        handles.push(handle);
    }

    for handle: JoinHandle<()> in handles {
        handle.await.unwrap();
    }

    println!("All tasks completed!");
}
```

Слика 4 - Пример коришћења *Tokio runtime*-а

3. ПРИМЕР АСИНХРОНОГ КОДА

3.1 Спецификација проблема

У овом раду разматрамо сценарио у којем апликација мора да шаље више захтева ка спољном *API*-ју. Сви захтеви су независни, али систем који прима захтеве има ограничене ресурсе и у паралели може да има само 5 конекција ка спољном *API*-ју. Поред тога, сваки захтев може да доживи неуспех са вероватноћом од 20%. Уколико дође до неуспеха, захтев мора бити поново послат све док се не добије успешан одговор.

3.2 Имплементација у програмском језику *C++*

За имплементацију примера у програмском језику *C++* искоришћена је библиотека *Boost.Asio*.

Функција *simulateApiCall* (слика 5) представља симулацију асинхроног *API* позива. Она је дефинисана као ко-рутина са повратним типом *awaitable<bool>*, што значи да се може користити са *co_await* и да враћа резултат када операција буде завршена. Унутар функције се креира *steady_timer* који се конфигурише да истекне након 1000 милисекунди, симулирајући време одговора *API* сервиса. Позивом *co_await timer.async_wait(use_awaitable)* ко-рутина се суспендује без блокирања нити, омогућавајући да други задаци буду обрађени у међувремену. Након што тајмер истекне, функција генерише насумичан резултат помоћу *std::uniform_real_distribution* и враћа true у 80% случајева и false у 20%, симулирајући успешан или неуспешан *API* позив.

```
std::mt19937 randomGenerator(std::random_device{}());
std::uniform_real_distribution<> distribution(0, 1);

awaitable<bool> simulateApiCall(int id) {
    int delayMs = 1000;

    steady_timer timer(co_await boost::asio::this_coro::executor);
    timer.expires_after(std::chrono::milliseconds(delayMs));
    co_await timer.async_wait(boost::asio::use_awaitable);

    std::cout << "RUN - Task[" << id << "] finished after " << delayMs << "ms.\n";
    co_return distribution(randomGenerator) < 0.8;
}
```

Слика 5 - Функција *simulateApiCall*

Функција *callApi (слика 6)* представља управљање логиком понављања асинхроног *API* позива све док се не постигне успешан резултат. Она је дефинисана као ко-рутина са повратним типом *awaitable<void>*.

```
awaitable<void> callApi(int id) {
    int tries = 0;
    while (true) {
        ++tries;
        if (co_await simulateApiCall(id)) {
            std::cout << "SUCCESS - Task[" << id << "] after " << tries << " tries.\n";
            co_return;
        } else {
            std::cout << "FAIL - Task[" << id << "] failed.\n";
        }
    }
}
```

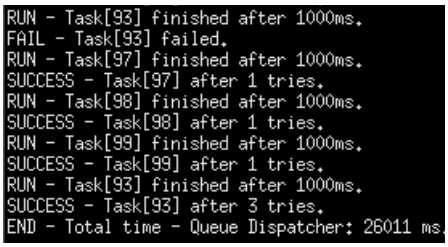
Слика 6 - Функција *callApi*

Функција *runWithQueueDispatcher* (слика 7) реализује ред задатака са ограниченим бројем истовремених асинхронхних операција. Унутар ње се креира *io_context* за управљање свим ко-рутинама и ред (*std::queue<int>*) у који се смештају задаци. Ламбда функција *launch* покреће задатак, позива *co_await callApi(id)* и након завршетка иницира следећи задатак из реда, ако постоји. Број тренутно активних задатака прати се промењливом *running*, а када нема више задатака и сви активни заврше, сигнализира се крај рада. На почетку се покреће *maxConnections* задатака помоћу *co_spawn*, а *ctx.run()* управља *event loop*-ом, настављајући ко-рутине кад *API* позиви или тајмери заврше. Функција на крају враћа укупно протекло време у милисекундама.

```
int runWithQueueDispatcher(int totalTasks, int maxConnections) {
    io_context ctx;

    auto start = std::chrono::high_resolution_clock::now();

    std::queue<int> tasks;
    for(int i = 0; i < totalTasks; i++) {
        tasks.push(i);
    }

    int running = 0;
    boost::asio::steady_timer allDone(ctx);

    std::function<awaitable<void>(int)> launch = [&](int id) -> awaitable<void> {
        ++running;
        co_await callApi(id);
        --running;

        if(!tasks.empty()) {
            int next = tasks.front();
            tasks.pop();
            co_spawn(ctx, launch(next), detached);
        }

        if(running == 0 && tasks.empty()) {
            allDone.cancel();
        }

        co_return;
    };

    for(int i = 0; i < maxConnections; i++) {
        int id = tasks.front();
        tasks.pop();
        co_spawn(ctx, launch(id), detached);
    }

    allDone.expires_at(std::chrono::steady_clock::time_point::max());
    allDone.async_wait(boost::asio::use_awaitable);
    ctx.run();

    return std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::high_resolution_clock::now() - start).count();
}
```

Слика 7 - Функција *runWithQueueDispatcher*

На примеру где је број пожива ка *API*-ју 100 и максималан број конекција 5, програм се ижврши за 25 секунди у просеку (слика 8), али време ижвршења варира од количине неуспешних захтева и поновних покушаја.



Слика 8 - Испис и време ижвршавања

3.3 Имплементација у програмском језику *Rust*

За имплементацију примера у програмском језику *Rust* искоришћен је *Tokio runtime*.

Функција *simulate_api_call* (слика 9) представља симулацију асинхроног *API* позива и дефинисана је као *async fn* која враћа *bool*. Унутар функције се користи *sleep(Duration::from_millis(delay_ms)).await* да се симулира кашњење од 1000 милисекунди, што представља време потребно да *API* сервис одговори. Позивом *.await* извршавање функције се суспендује без блокирања текуће нити, омогућавајући да се други асинхрони задаци истовремено обрађују. Након што тајмер истекне, генерише се насумичан резултат помоћу *StdRng::from_entropy().gen_bool(0.8)*, тако да *API* позив у 80% случајева буде успешан, а у 20% случајева неуспешан.

```rust
async fn simulate_api_call(id: usize) -> bool {
    let delay_ms: u64 = 1000;
    sleep(Duration::from_millis(delay_ms)).await;

    println!("RUN - Task[{}] run ended after {}ms.", id, delay_ms);

    let is_successful: bool = StdRng::from_entropy().gen_bool(0.8);
    is_successful
}
```

Слика 9 - Функција *simulate_api_call*

Функција *call_api* (слика 10) представља управљање логиком понављања асинхроног *API* позива све док се не постигне успешан резултат. Она је дефинисана као асинхрона функција (*async fn*) и користи *.await* за суспензију извршавања без блокирања нити.

```rust
async fn call_api(id: usize) {
    let mut tries: i32 = 0;

    loop {
        tries += 1;

        if simulate_api_call(id).await {
            println!("SUCCESS - Task[{}] finished successfully after {} tries.", id, tries);
            break;
        } else {
            println!("FAIL - Task[{}] failed.", id);
        }
    }
}
```

Слика 10 - Функција *call_api*

Функција *run_with_semaphore* (слика 11) управља покретањем више асинхронхних задатака уз ограничење броја истовремених извршења. Она користи семафор да ограничи број паралелних позива на *max_connections*. За сваки задатак се добија дозвола (*permit*) из семафора, након чега се позива *call_api(id).await*. Када задатак

заврши, дозвола се ослобађа (*drop(permit)*), што омогућава да следећи задатак из реда буде покренут. Сви задаци се чувају у вектору *handles* и чекају се да се заврше помоћу *join_all*. Функција на крају враћа укупан протекли временски интервал.

```rust
async fn run_with_semaphore(task_count: usize, max_connections: usize) -> std::time::Duration {
    let start: Instant = Instant::now();

    let semaphore: Arc<Semaphore> = Arc::new(data: Semaphore::new(permits: max_connections));
    let mut handles: Vec<tokio::task::JoinHandle<()>> = vec![];

    for id: usize in 0..task_count {
        let permit: tokio::sync::OwnedSemaphorePermit = semaphore.clone().acquire_owned().await.unwrap();

        let handle: tokio::task::JoinHandle<()> = tokio::spawn(future: async move {
            call_api(id).await;

            drop(permit);
        });

        handles.push(handle);
    }
    futures::future::join_all(iter: handles).await;

    start.elapsed()
}
```

Слика 11 - Функција *run_with_semaphore*

На примеру где је број пожива ка *API*-ју 100 и максималан број конекција 5, програм се ижврши за 25 секунди у просеку (слика 12), али време ижвршења варира од количине неуспешних захтева и поновних покушаја.

```
SUCCESS - Task[93] finished successfully after 2 tries.
SUCCESS - Task[96] finished successfully after 1 tries.
SUCCESS - Task[97] finished successfully after 1 tries.
RUN - Task[99] run ended after 1000ms.
RUN - Task[98] run ended after 1000ms.
SUCCESS - Task[99] finished successfully after 1 tries.
SUCCESS - Task[98] finished successfully after 2 tries.
END - Total time - Semaphore: 25.039569514s
```

Слика 12 - Испис и време ижвршавања

3.4 Поређење имплементација у програмским језицима *C++* и *Rust*

У погледу читљивости и сложености кода, *Rust* имплементација пружа краћи и прегледнији код. Супротно томе, *Boost.Asio* захтева више *boilerplate* кода. Међутим, *Boost.Asio* омогућава већу контролу над извршавањем и флексибилност на нижем нивоу.

Модел асинхроне обраде у оба примера је сличан. И *Boost.Asio* и *Tokio* користе *event loop* архитектуру, при чему се задаци могу суспендовати без блокирања извршне нити.

Ефикасност оба приступа је слична у посматраном сценарију. Код 100 захтева са ограничењем на пет паралелних конекција, и *C++* и *Rust* имплементације осигуравају да се истовремено покреће највише пет задатака, поштујући ограничење ресурса. У исто време, 20% захтева намерно не успева, а систем поново покреће неуспеле задатке. Ова комбинација ограничења конекција и механизма поновног покушаја резултује укупним временом обраде од око 25 секунди које одговара очекиваном трајању сценарија, показујући да оба система ефикасно распоређују задатке и одржавају скалабилност.

4. ЗАКЉУЧАК

У овом раду приказано је поређење приступа асинхроном програмирању у програмским језицима *C++* и *Rust* на примеру симулације *API* позива са ограниченим ресурсима и могућношћу неуспеха. Показао се значај асинхроног приступа за ефикасно управљање више независних задатака, без блокирања главне нити.

Имплементација у *Rust*-у је једноставнија и прегледнија, захваљујући *async/.await* синтакси и *Tokio runtime*-у, док *C++* уз *Boost.Asio* омогућава већу контролу и флексибилност на нижем нивоу, али уз нешто сложенији код.

У погледу перформанси, оба језика показују сличну ефикасност у сценарију са ограниченим бројем паралелних конекција и поновним покушајима неуспелих захтева. Ово потврђује да су и *Rust* и *C++* погодни за развој високоефикасних асинхронхних система.