



Реализация поддержки асинхронного программирования для фреймворка DSLab

Выполнил: Макогон Артём Аркадьевич, БПМИ206

Руководитель: Сухорослов Олег Викторович, к.т.н., доцент НИУ ВШЭ

23 мая 2023 г.

Введение

Описание предметной области

- Большой объем данных/вычислений
- Надежность
- Масштабируемость



Введение

Описание предметной области

- Большой объем данных/вычислений
- Надежность
- Масштабируемость



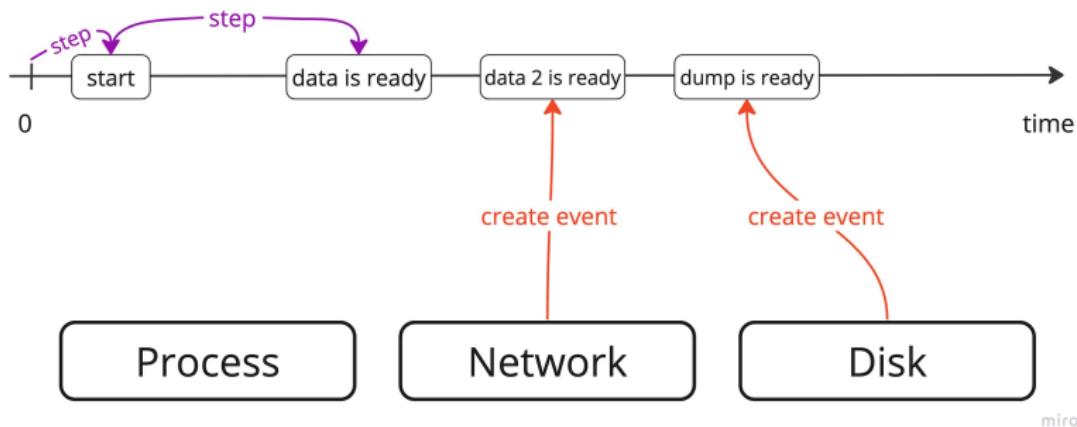
- > Недетерминированные алгоритмы
- > Сложное тестирование





Введение

Дискретно-событийное моделирование

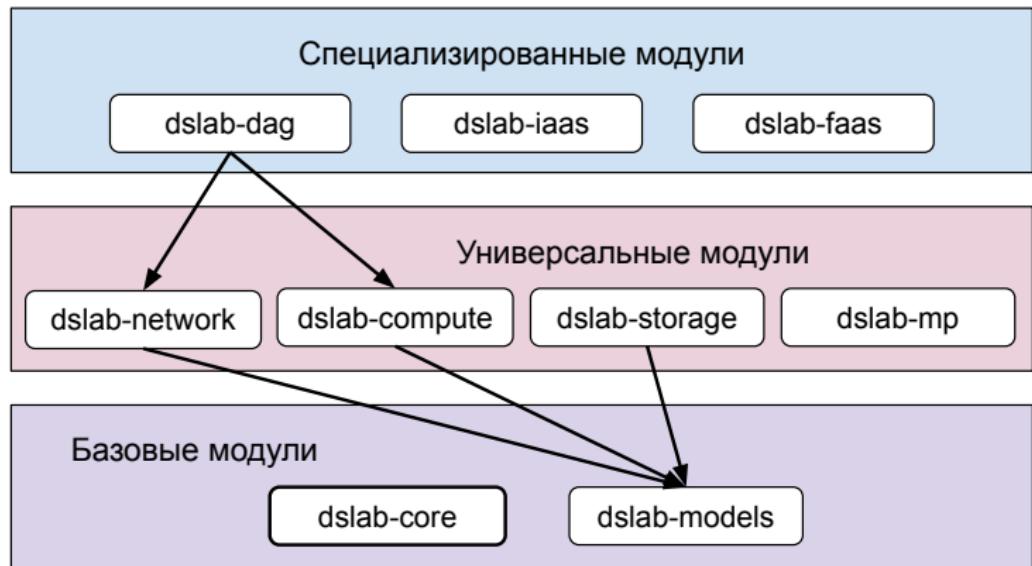


Исполнение симуляции



Введение

Архитектура проекта DSLab





Введение

Callback модель. Фрагментированная реализация Process.

```
fn on(&mut self, event: Event) {  
    cast!(match event.data {  
        Start {} => {  
            self.on_start();  
        }  
        DownloadCompleted {data} => {  
            self.on_download_completed(data);  
        }  
        DiskWriteCompleted => {  
            self.on_disk_write_completed();  
        }  
    })  
}
```

Реагирование на события.

```
impl Process {  
    fn on... {  
          
    }  
    fn on... {  
          
    }  
    fn on... {  
          
    }  
}
```

Реализация
callback-ов



Введение

Преимущества асинхронного подхода

```
async fn collect_file_from_replicas(replicas) {
    result = download_from_all(replicas).await;

    if result.has_quorum {
        dump_file_to_storage(result.file).await;

        send_ok_message_to_user();
    } else {
        send_reject_message_to_user();
    }
}
```

Псевдокод асинхронного взаимодействия компонент в симуляции



Введение

Постановка задачи

Цель проекта – реализация поддержки асинхронного программирования для фреймворка DSLab. Для этого необходимо:

- Реализовать эффективное асинхронное расширение для существующего ядра `dslab-core`. Поддержать возможности:
 - запустить асинхронную задачу;
 - «заснуть» на определенное время внутри симуляции;
 - асинхронно ждать событий от других компонент;
 - комбинировать ожидания с помощью методов `futures`.
- Добавить примеры, написать документацию и тесты.



Введение

Дизайн и структура ядра dslab-core

Event:

```
id: EventId  
src: Id,  
dest: Id,  
time: f64,  
data: dyn EventData
```

Simulation

Component 1

Component 2

miro

Внутреннее устройство dslab-core

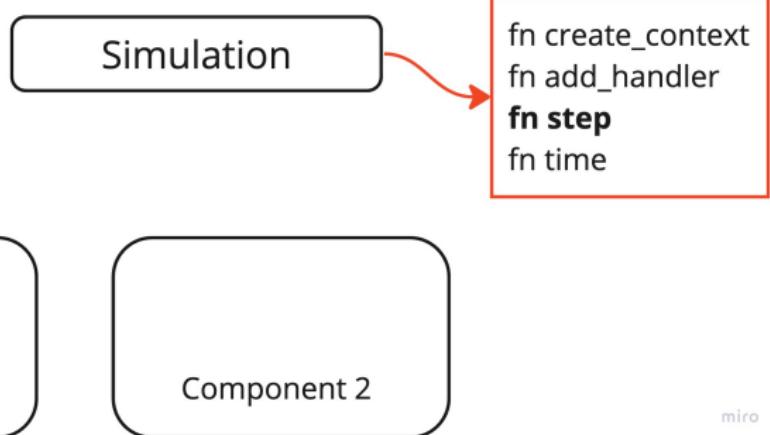


Введение

Дизайн и структура ядра dslab-core

Event:

```
id: EventId  
src: Id,  
dest: Id,  
time: f64,  
data: dyn EventData
```

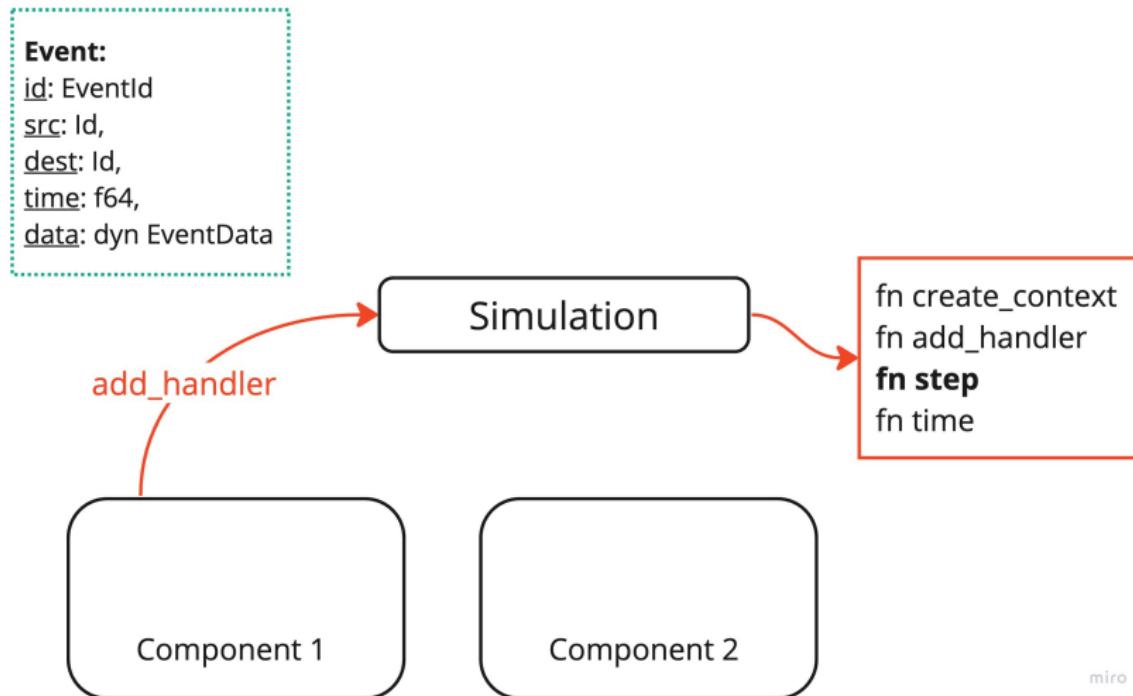


miro

Внутреннее устройство dslab-core

Введение

Дизайн и структура ядра dslab-core

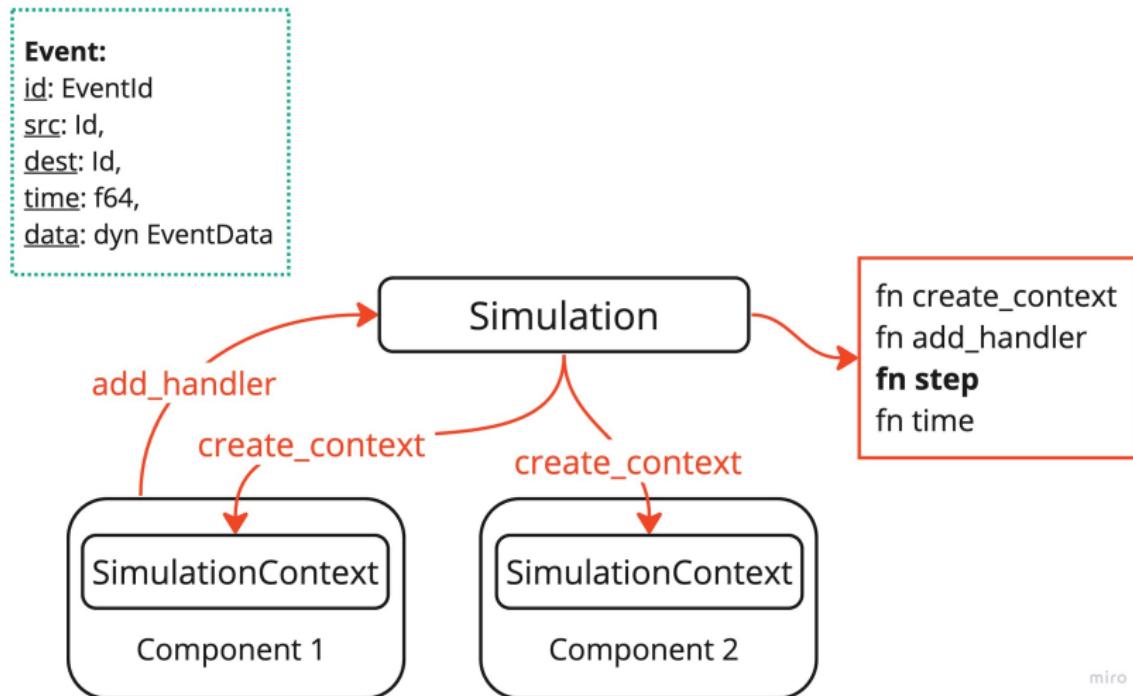


miro

Внутреннее устройство dslab-core

Введение

Дизайн и структура ядра dslab-core

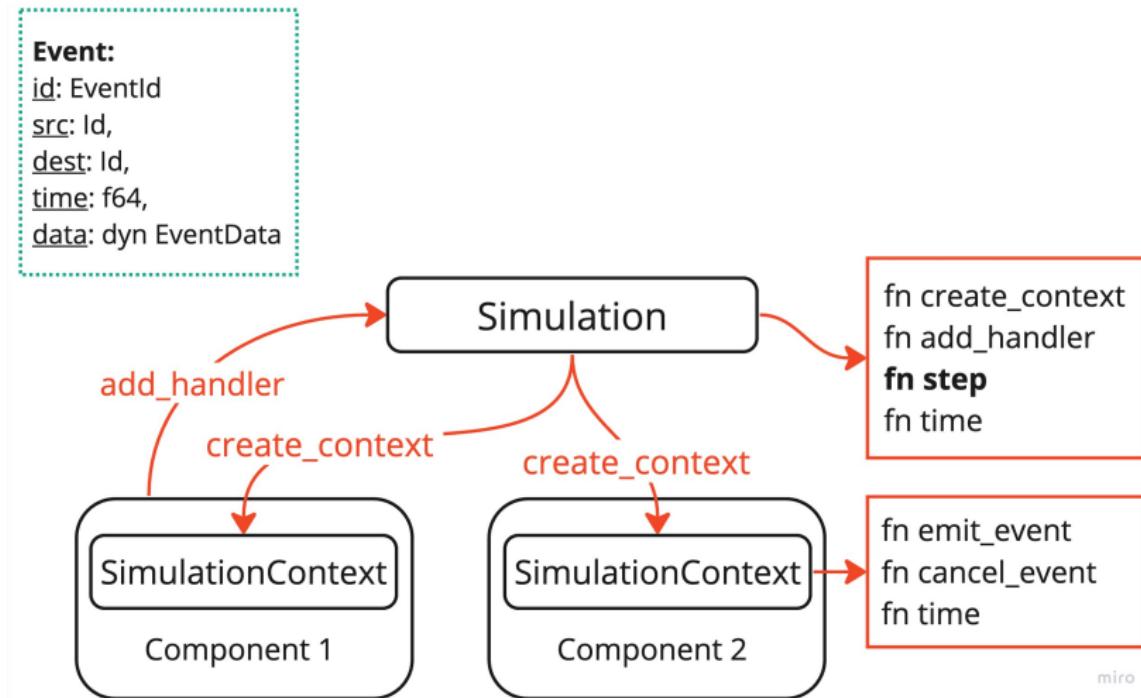


miro

Внутреннее устройство dslab-core

Введение

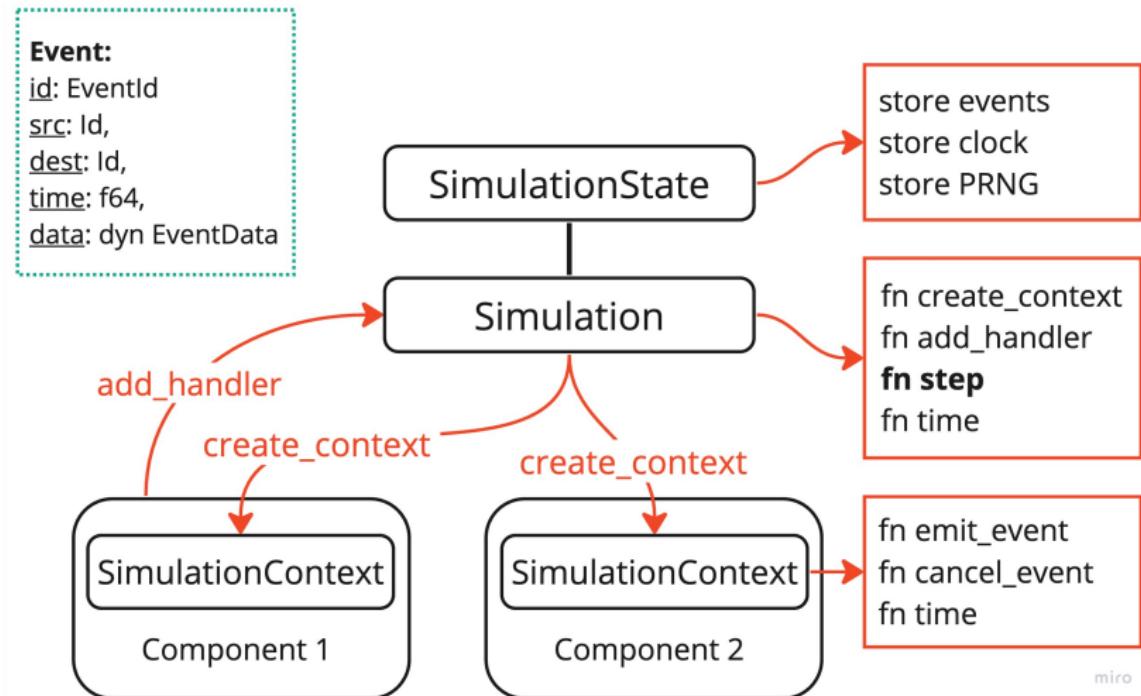
Дизайн и структура ядра dslab-core



Внутреннее устройство dslab-core

Введение

Дизайн и структура ядра dslab-core



miro

Внутреннее устройство dslab-core



Новое API для пользователей

Расширение SimulationContext

- Запустить асинхронную задачу:
 - + `fn spawn(&self, future: impl Future<Output = ()>)`
- «Заснуть» на определенное время:
 - + `async fn async_wait_for(&self, timeout: f64)`
- Дождаться события с «данными» типа T от компонента src:
 - + `async fn async_handle_event<T>(src: Id) -> (Event, T)`
 - + `async fn async_wait_for_event<T>(src: Id, timeout: f64) -> AwaitResult<T>`
- Дождаться события с «данными» T, деталями details от src:
 - + `async fn async_detailed_handle_event<T>(src: Id, details: u64) -> (Event, T)`
 - + `async fn async_detailed_wait_for_event<T>(src: Id, details: u64, timeout: f64) -> AwaitResult<T>`



Новое API для пользователей

Детализированное ожидание. Пример

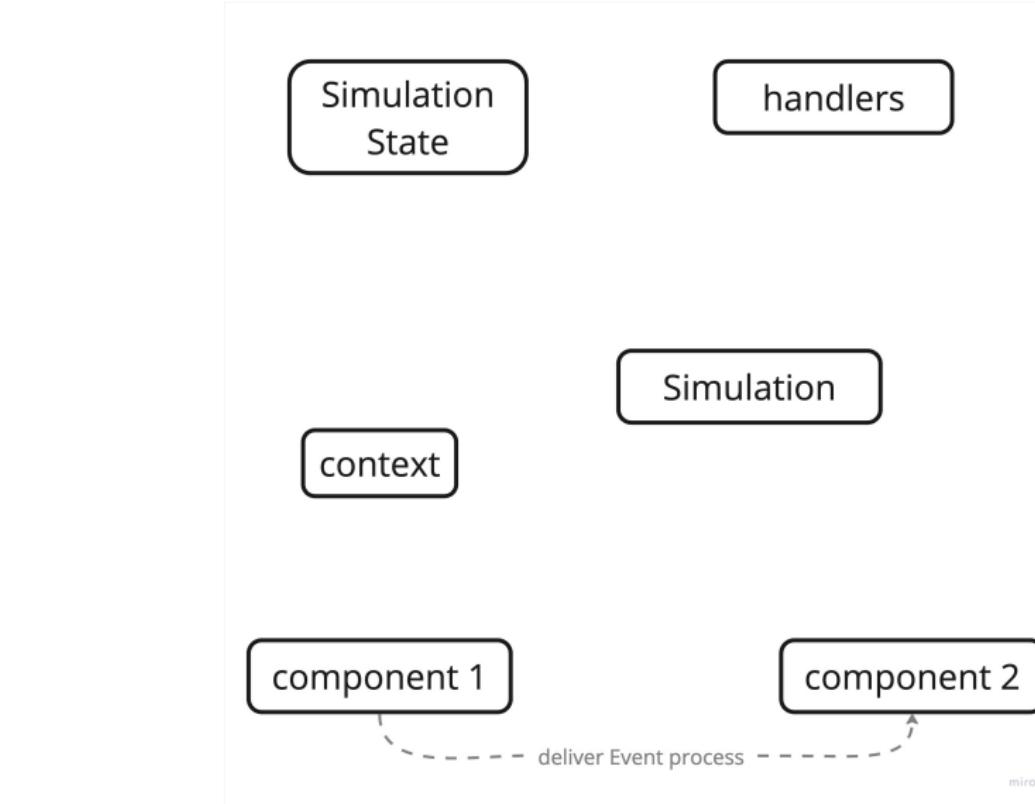
```
async fn process_data_transfer_request(
    &self, request: Request) {
    let request_id = self.network.send(request);

    self.ctx.
        async_detailed_handle_event::<DataTransferCompleted>(
            self.network.id,
            request_id,
        ).await;
}
```

Использование детализированного асинхронного ожидания

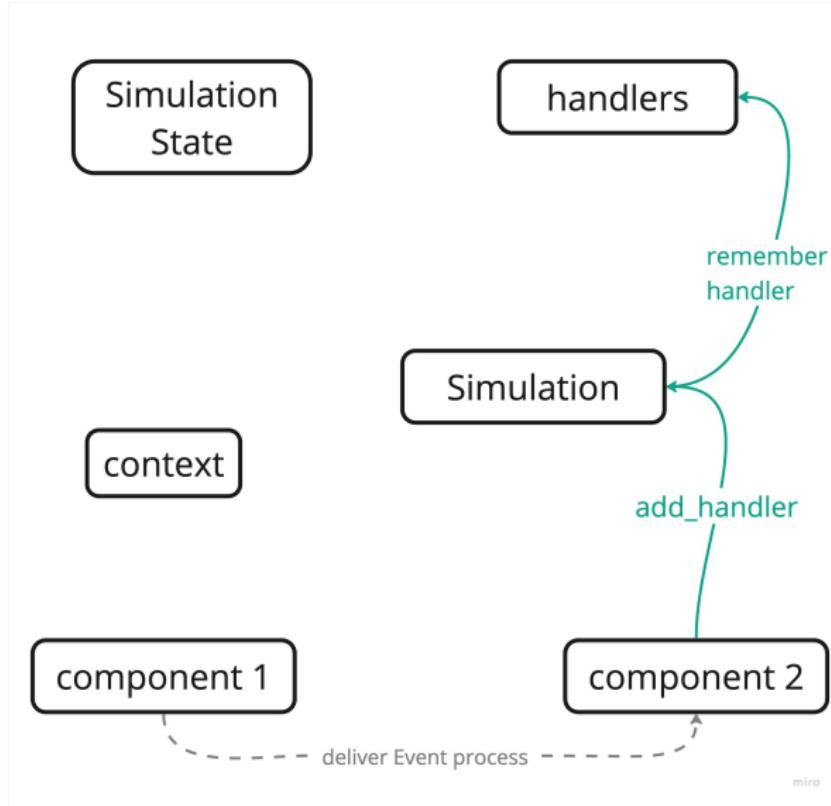
Внутреннее устройство и реализация

Callback-based модель



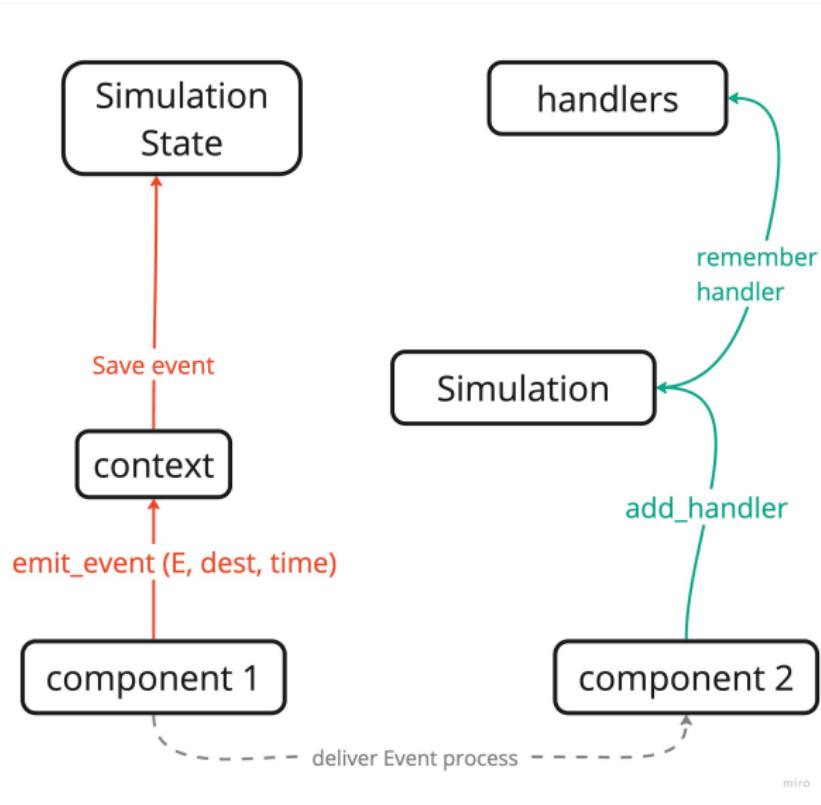
Внутреннее устройство и реализация

Callback-based модель



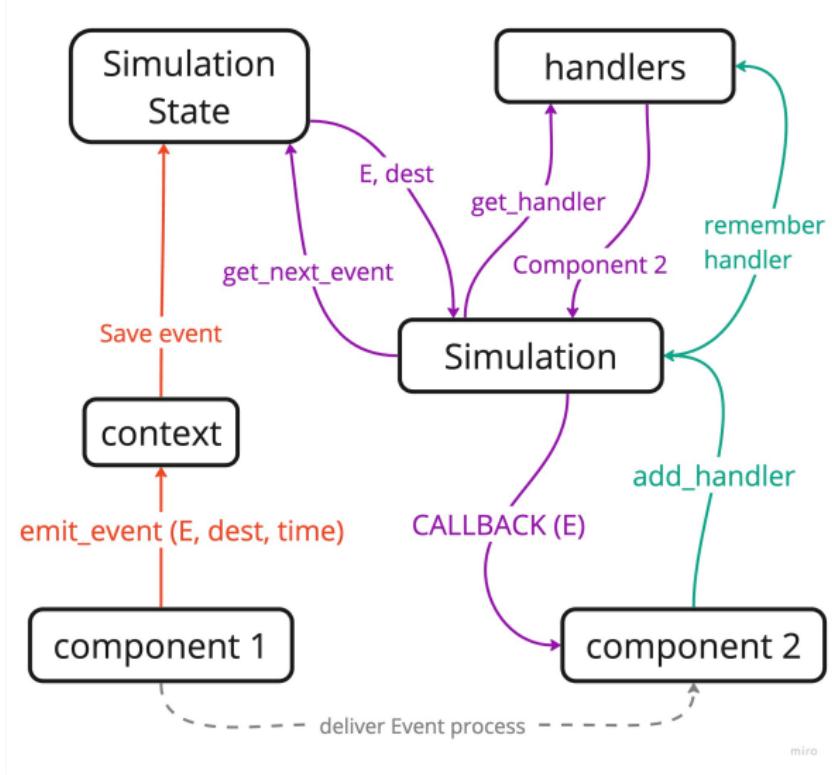
Внутреннее устройство и реализация

Callback-based модель



Внутреннее устройство и реализация

Callback-based модель

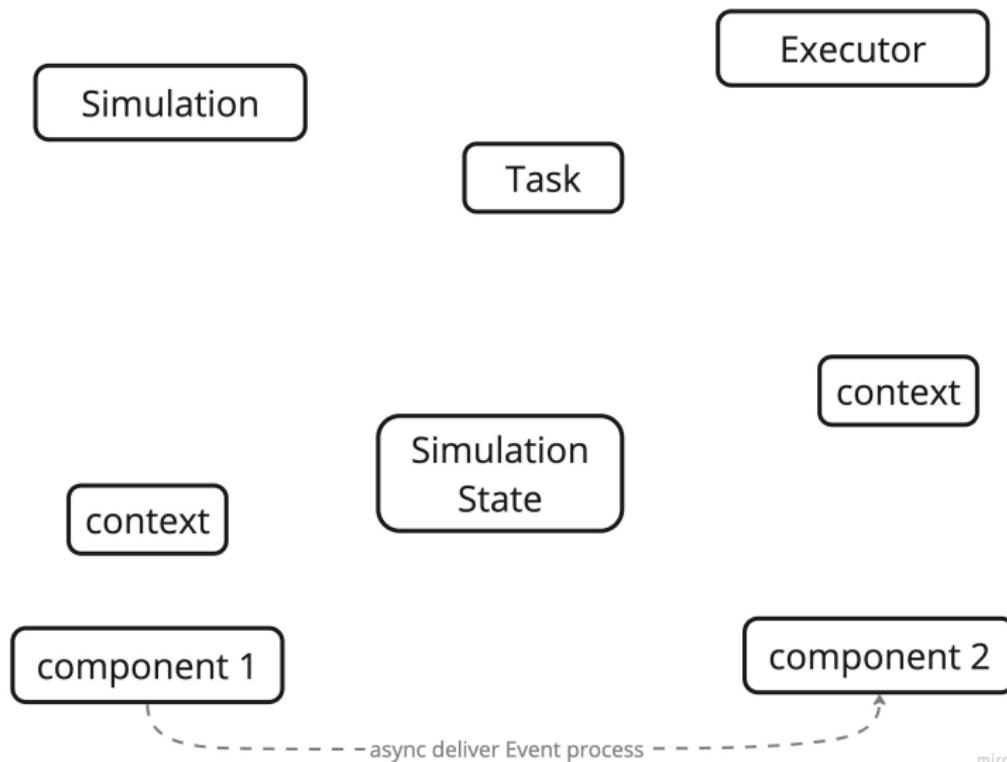


miro



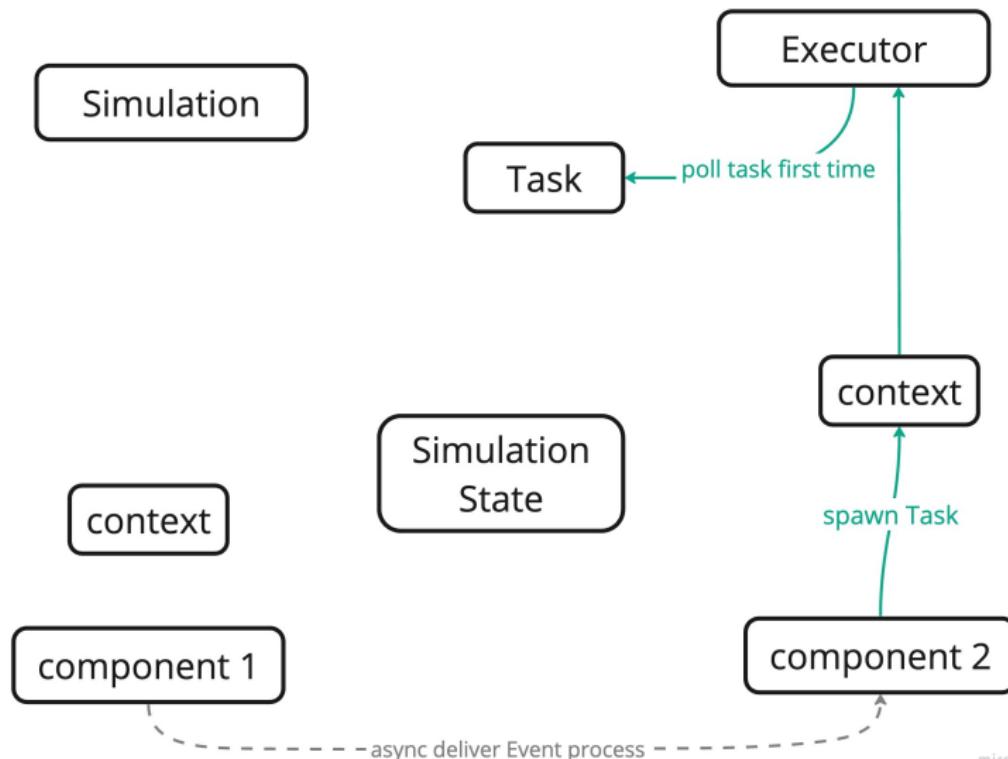
Внутреннее устройство и реализация

Асинхронная модель



Внутреннее устройство и реализация

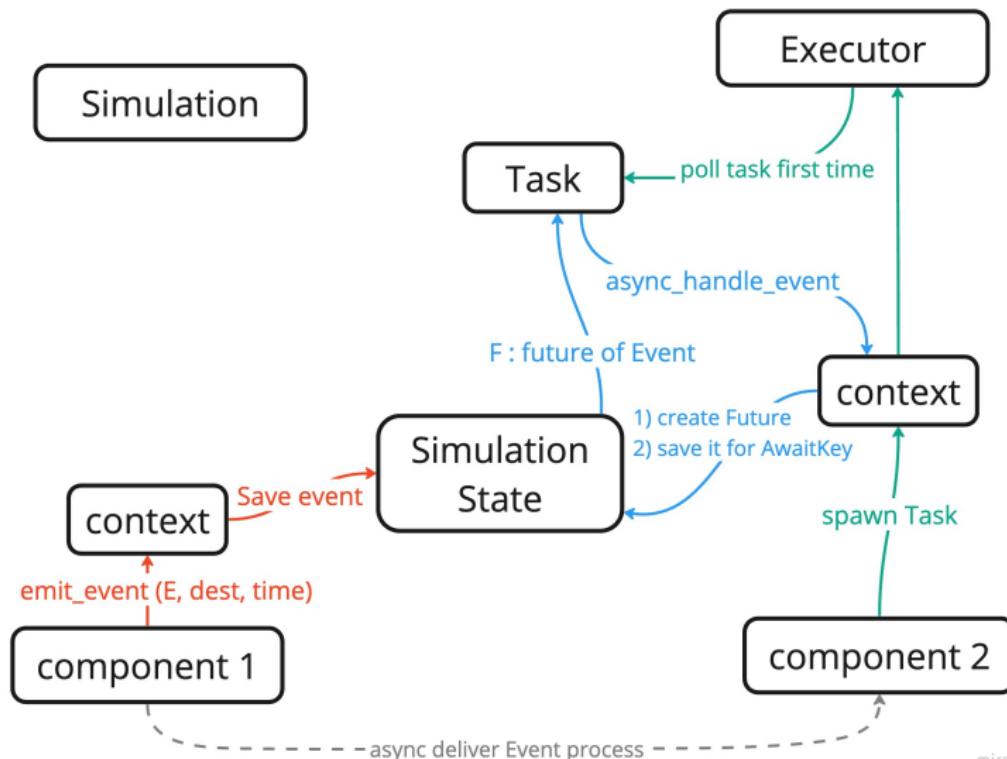
Асинхронная модель



miro

Внутреннее устройство и реализация

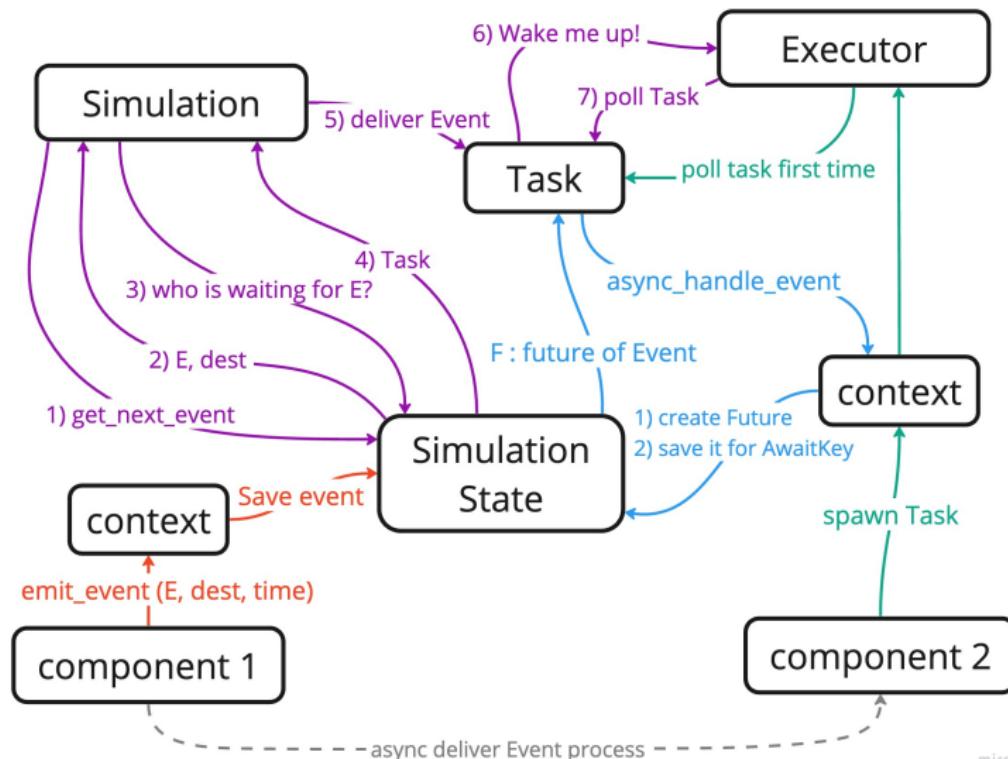
Асинхронная модель



miro

Внутреннее устройство и реализация

Асинхронная модель

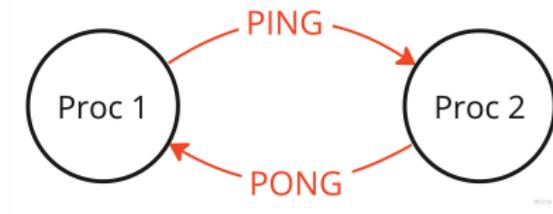


Тестирование

Ping-pong. Производительность

Hosts	100000
Peers per host	100
Iterations	100

Аргументы

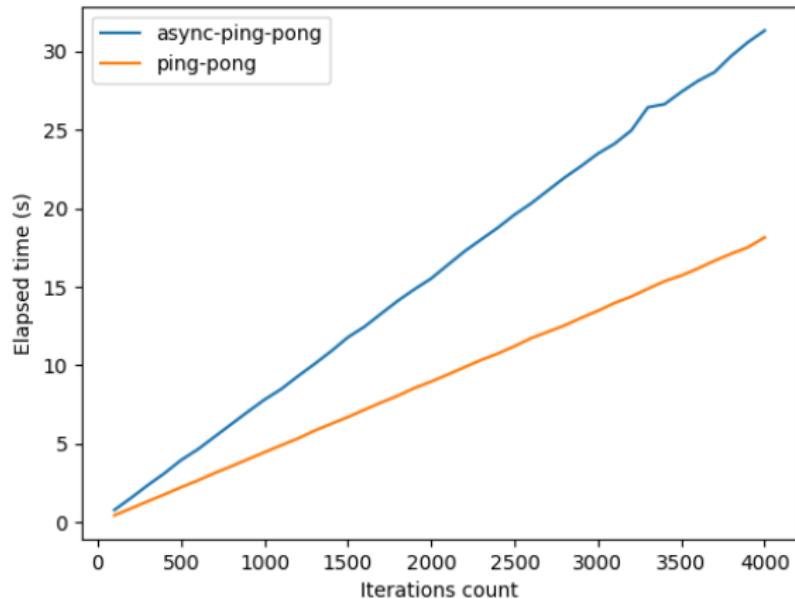


Example	Elapsed time	Events/s	Iterations/s
async-ping-pong	16.45s	1234103	6.08
ping-pong	8.20s	2452670	12.20

Сравнение производительности

Тестирование

Ping-pong. Производительность



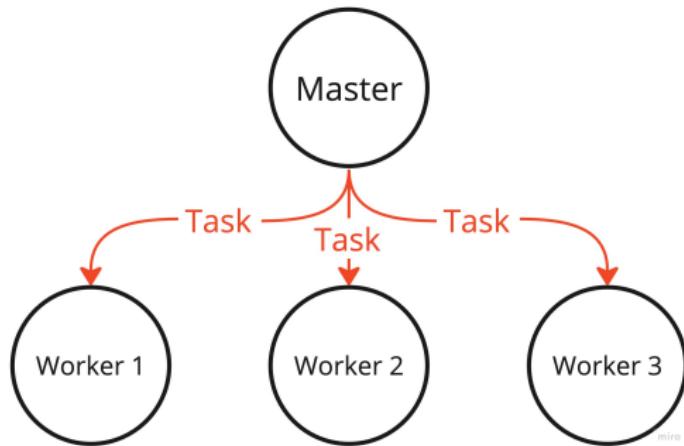
Сравнение производительности

Тестирование

Master-workers. Производительность

Hosts	100
Tasks	100000

Аргументы



Example	Elapsed time	Scheduling time	Events/s
async-master-workers	5.97s	4.31s	285059
master-workers	5.19s	4.31s	328392

Сравнение производительности



Тестирование

Master-workers. Сравнение кода компонента Worker

```
pub struct Worker {  
    tasks: HashMap<u64, TaskInfo>,  
    computations: HashMap<u64, u64>,  
    reads: HashMap<u64, u64>,  
    writes: HashMap<u64, u64>,  
    downloads: HashMap<usize, u64>,  
    uploads: HashMap<usize, u64>,  
}  
  
fn on_task_request(request);  
fn on_data_read_completed(r_id);  
fn on_comp_started(comp_id);  
fn on_comp_finished(comp_id);  
fn on_data_write_completed(r_id);  
  
fn on_data_transfer_completed(*);
```

master-workers

```
fn on_task_request(req) {  
    self.ctx.spawn(  
        self.process_task(req));  
}  
  
async fn process_task(&self, req:  
    TaskRequest) {  
    let mut task = TaskInfo {req};  
  
    self.download_data(&task).await;  
    self.read_data(&task).await;  
    self.run_task(&task).await;  
    self.write_data(&task).await;  
    self.upload_result(&task).await;  
}
```

async-master-workers



- ✓ Реализовать эффективное асинхронное расширение для существующего ядра `dslab-core`.
- ✓ Добавить примеры, написать документацию и тесты.
- ✓ Реализовать примитив синхронизации: «блокирующая очередь» для передачи произвольных данных.
- ✓ Сделать раздельную сборку ядра \implies классическое ядро не теряет производительность.
- ✓ Сформулировать будущее развитие проекта.



- ✓ Реализовать эффективное асинхронное расширение для существующего ядра `dslab-core`.
- ✓ Добавить примеры, написать документацию и тесты.
- ✓ Реализовать примитив синхронизации: «блокирующая очередь» для передачи произвольных данных.
- ✓ Сделать раздельную сборку ядра \Rightarrow классическое ядро не теряет производительность.
- ✓ Сформулировать будущее развитие проекта.

Главное достижение: можно произвольно совмещать оба подхода:

- Stateful-процесс \Rightarrow используем асинхронный подход.
- Stateless-процесс \Rightarrow используем модель callback-ов.

Вопросы



1. Execution. Stackless-корутины в Rust. Что такое Task?
2. Примитив синхронизации `UnboundedBlockingQueue<T>`
3. Ping-pong. Сравнение кода
4. Раздельная сборка. Feature `async-core`
5. Планы на будущее. Дальнейшее развитие проекта
6. Сравнение с аналогами. `SimGrid`
7. Как устроено дискретно-событийное моделирование



Приложения

Execution. Stackless-корутины в Rust. Что такое Task?

```
fn on_start_action() {  
    // do 1  
}  
  
fn on_first_event() {  
    // do 2  
}  
  
fn on_second_event() {  
    // do 3  
}  
  
/* ... */  
  
on_start_action();
```

\iff

```
async fn action() {  
    // do 1  
  
    wait_for_first_event().await;  
  
    // do 2  
  
    wait_for_second_event().await;  
  
    // do 3  
}  
  
/* ... */  
  
let future = action();  
future.poll(**/);
```

Синхронный код (разрезан на части разработчиком)

Асинхронный код (разрезан на части компилятором)



Приложения

Ping-pong. Сравнение кода.

```
pub struct Process {  
    iterations: u32,  
}  
  
fn on_start(&mut self) {  
    let peer = /*random peer*/;  
    self.send(Ping {}, peer);  
}  
  
fn on_ping(&mut self, from: Id) {  
    self.send(Pong {}, from);  
}  
  
fn on_pong(&mut self, from: Id) {  
    self.iterations -= 1;  
    if self.iterations > 0 {  
        let peer = /*choose peer*/  
        self.send(Ping {}, peer);  
    }  
}
```

ping-pong

```
fn on_start(&self) {  
    self.ctx.spawn(self.process());  
}  
  
fn on_ping(&mut self, from: Id) {  
    self.send(Pong {}, from);  
}  
  
async fn process(&self) {  
    for _i in 0..self.iterations {  
        let peer = /*choose peer*/;  
        self.send(Ping {}, peer);  
  
        // stop until receive Pong  
        self.ctx.async_handle_event  
            ::<Pong>(peer).await;  
    }  
}
```

async-ping-pong



Приложения

Примитив синхронизации `UnboundedBlockingQueue<T>`.

```
pub struct Worker {
    task_chan: UnboundedBlockingQueue<TaskInfo>,
}

fn on_task_request(&self, task_info: TaskInfo) {
    self.task_chan.send(task_info);
}

async fn work_loop(&self) {
    loop {
        let task_info = self.task_chan.receive().await;
        /* process task */
    }
}
```

Использование `UnboundedBlockingQueue<T>` как буфер для задач



Приложения

Раздельная сборка. Feature `async-core`.

Пакет `dslab-core` можно подключить с разными опциями:

- `dslab-core = { path = "../dslaсb-core"}` – асинхронность не поддерживается. Максимальная производительность.
- `dslab-core = { path = "../dslaсb-core features = ["async_core"] }` – поддерживается весь асинхронный интерфейс. Скорость симуляции снижена.



Приложения

Планы на будущее. Дальнейшее развитие проекта.

- Повышение удобства синтаксиса.
- Увеличение производительности симуляции.
- Расширение асинхронной функциональности:
 - > `yield` – уйти в конец очереди планировщика без продвижения по времени.
 - > `Cancellation`. Возможность отменять задачи.
- Расширение «стандартных» асинхронных инструментов:
 - > Полноценный канал из Go
 - > `ConditionVariable`?



Приложения

Сравнение с аналогами. SimGrid.

```
void Process(int id, Mailbox* in, vector<Mailbox*> peers,
 int iters) {
    // wait for Start message
    auto* msg = in->get<Message>();
    bool stopped = false;
    bool wait_reply = false;
    while (!stopped) {
        if (pings_to_send > 0 && !wait_reply) {
            MailBox* peer_mailbox = /* choose peer */;
            peer_mailbox->put_init(new Message(/*create message*/));
            wait_reply = true;
            pings_to_send -= 1;
        }
        msg = in->get<Message>();
        if (msg->type == MessageType::PING) {/* handle PING */}
        else if (msg->type == MessageType::PONG) {/* handle PONG */}
        else if (msg->type == MessageType::STOP) { stopped = true; }
    }
}
```

Код процесса ping-pong в фреймворке SimGrid



Приложения

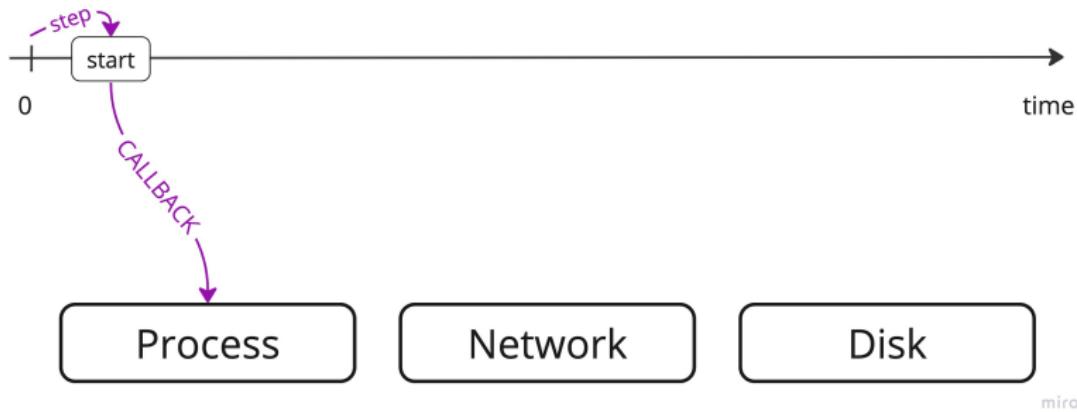
Как устроено дискретно-событийное моделирование



Исполнение симуляции

Приложения

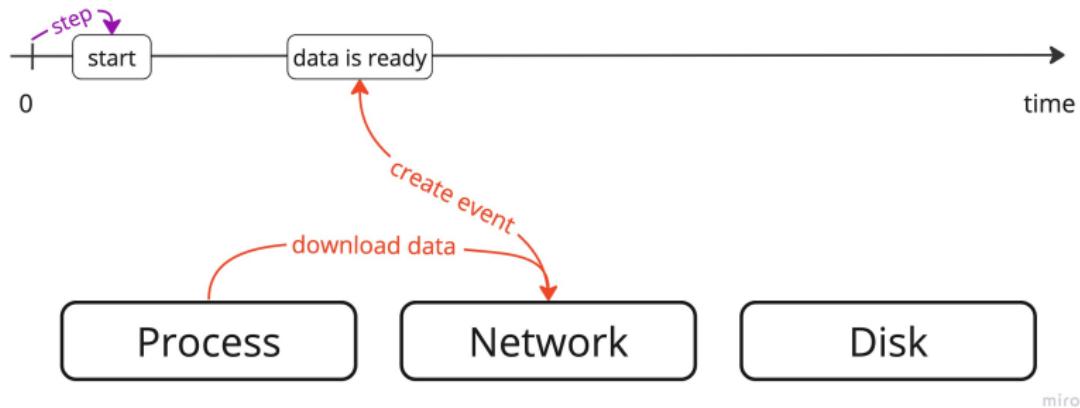
Как устроено дискретно-событийное моделирование



Исполнение симуляции

Приложения

Как устроено дискретно-событийное моделирование

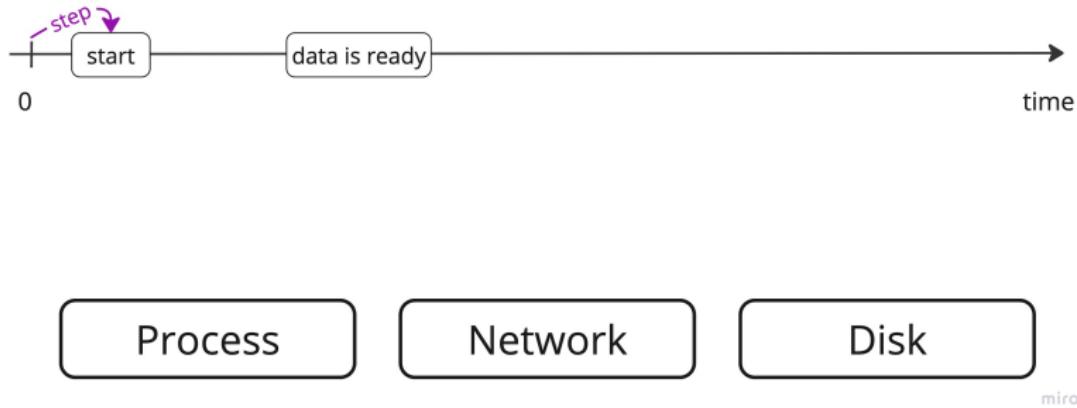


Исполнение симуляции



Приложения

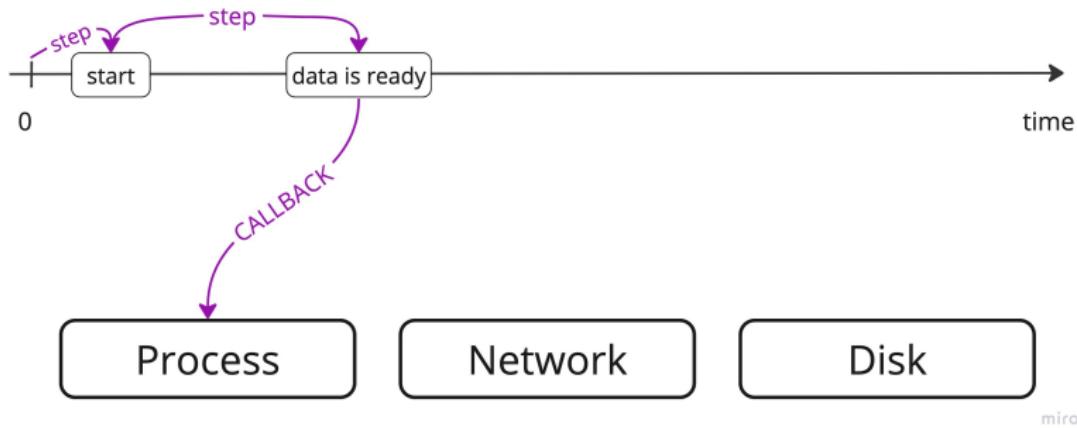
Как устроено дискретно-событийное моделирование



Исполнение симуляции

Приложения

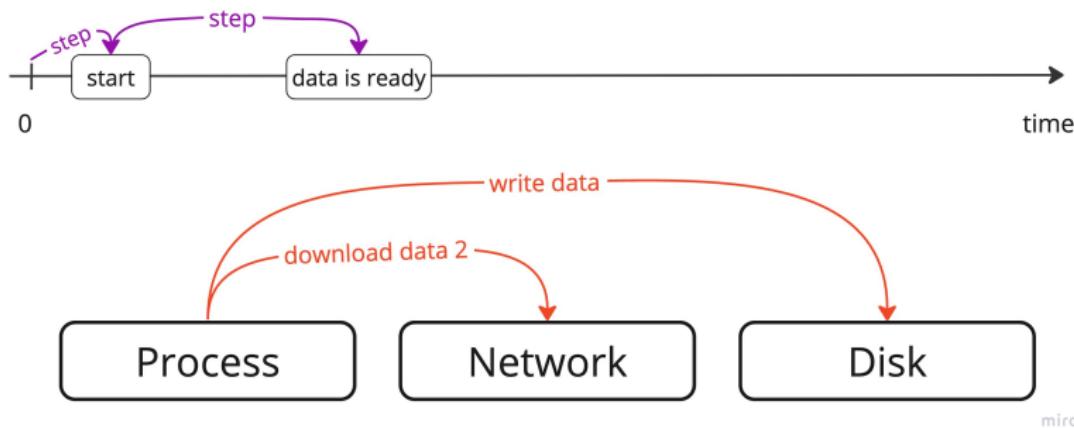
Как устроено дискретно-событийное моделирование



Исполнение симуляции

Приложения

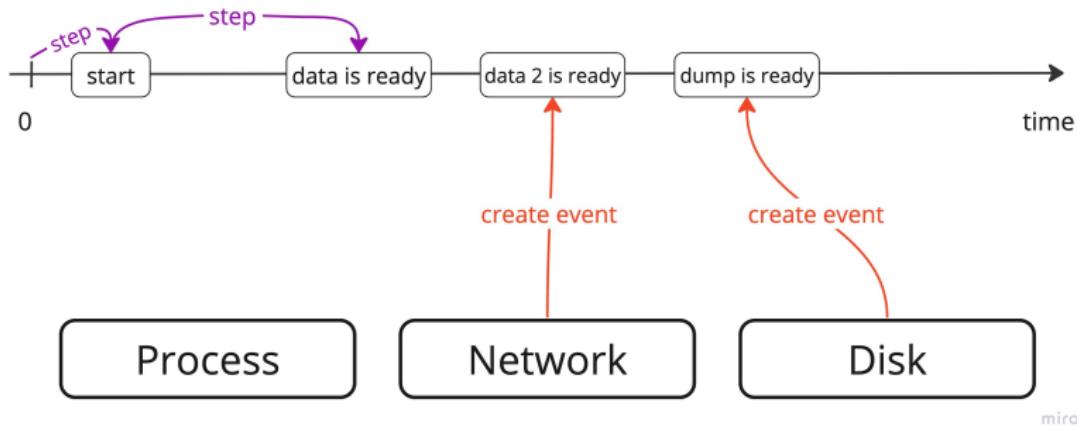
Как устроено дискретно-событийное моделирование



Исполнение симуляции

Приложения

Как устроено дискретно-событийное моделирование

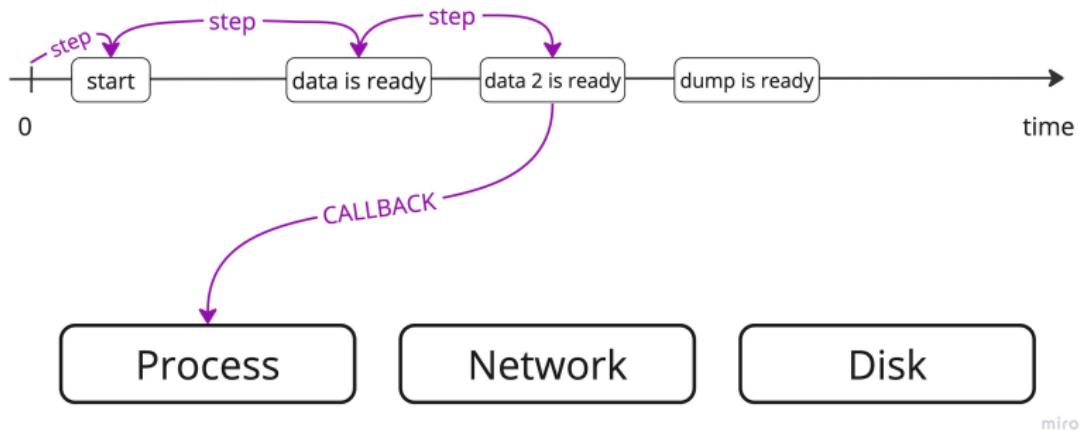


Исполнение симуляции



Приложения

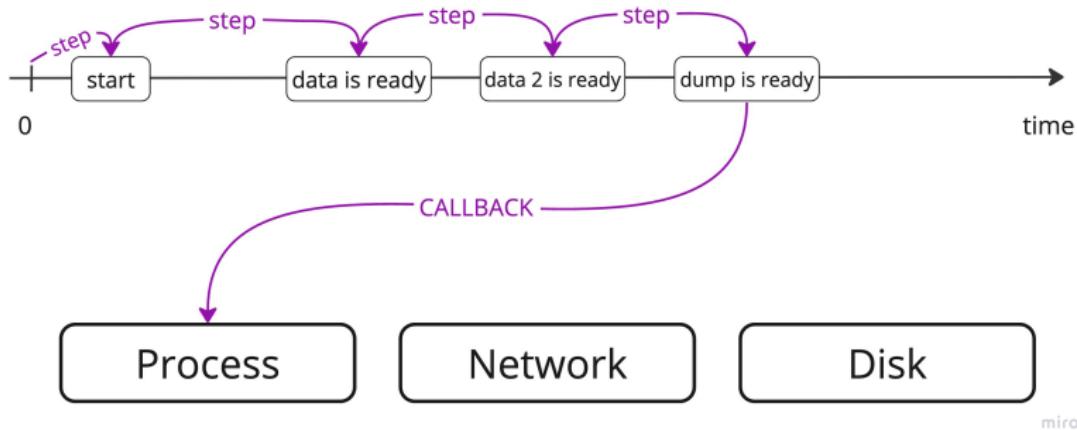
Как устроено дискретно-событийное моделирование



Исполнение симуляции

Приложения

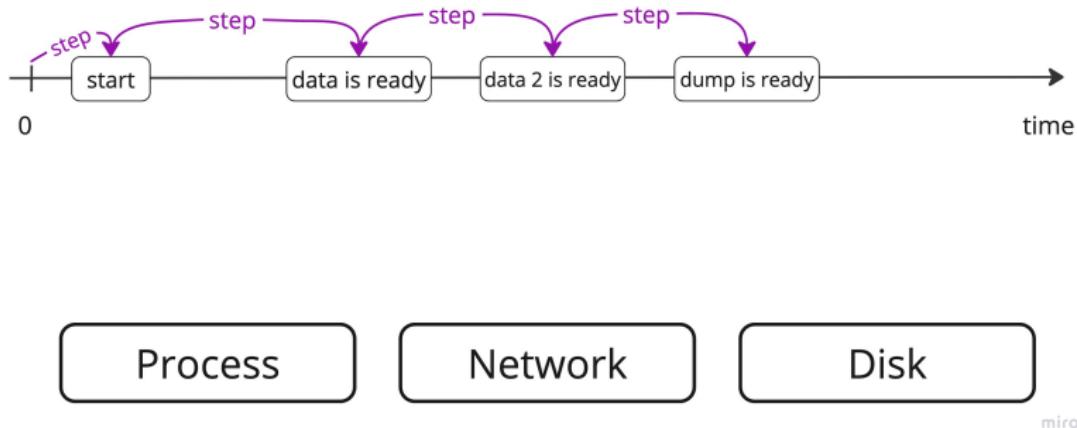
Как устроено дискретно-событийное моделирование



Исполнение симуляции

Приложения

Как устроено дискретно-событийное моделирование



Исполнение симуляции