ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
ФГАОУ ВО НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук
Образовательная программа «Прикладная математика и информатика»

УДК XXXXX

Отчет о программном проекте на тему:

Реализация поддержки асинхронного программирования для фреймворка DSLab

Выполнил:

студент группы БПМИ206 — 11.05.2023
Макогон Артём Аркадьевич _____ (подпись) _____ (дата)

Принял руководитель проекта:

Сухорослов Олег Викторович
Научный сотрудник
Факультета компьютерных наук НИУ ВШЭ

_____ (подпись) _____ (дата)

Москва 2023

Содержание

Аннотация

DSLab - программный фреймворк для имитационного моделирования и тестирования распределенных систем.

В проекте используется дискретно-событийный подход описания моделей и приложений, где события обрабатываются в пользовательских функциях (callback-ах). В рамках проекта добавлена возможность управлять событиями асинхронно.

Ключевые слова

Распредленные системы, симуляция, асинхронность

1 Основные термины и определения

1. Распределенная система – система, выполняющая определенную задачу по обработке данных и компьютерных вычислений и использующая для этого несколько независимых машин (узлов), каждый из которых обладает своими ресурсами, памятью и запущенными процессами.

2. Симуляция – это процесс создания модели реальной системы или процесса, которая может быть использована для изучения и предсказания поведения системы в различных условиях.

3. DSLab – программный фреймворк для имитационного моделирования и тестирования распределенных систем.

4. Асинхронное программирование – это парадигма программирования, основанная на неблокирующем ожидании, которое позволяет программам переключаться на выполнение других задач в ожидании длительных операций.

5. Callback – это функция или код, который передается в качестве аргумента другой функции или метода, для выполнения в более поздний момент в ответ на событие или операцию.

6. Логирование – хронологическая запись операций, выполненных в системе. Логирование позволяет сохранять информацию о состоянии системы, которую можно использовать для ее исследования и отладки.

2 Введение

2.1 Описание проекта DSLab

В силу широты охвата областей применения фреймворка он организован в виде набора слабо связанных программных модулей, использование которых будет осуществляться через их API. Это даст возможность пользователям фреймворка (исследователям, разработчикам, преподавателям) гибким образом собирать из модулей решения под свои цели, например симуляторы для конкретных типов систем или постановок задач.

Входящие в состав фреймворка модули можно условно разделить на три типа:

1. Базовые, функциональность которых используется остальными модулями (например, реализация дискретно-событийного моделирования)
2. Универсальные, функциональность которых может быть использована в различных предметных областях (например, модели сети);
3. Специализированные, которые заточены под определенную предметную область (например, библиотеки для моделирования облачных инфраструктур, исследования алгоритмов планирования заданий на кластерах или тестирования решений учебных заданий).

Архитектуру DSLab можно схематично представить в виде трех слоев (Рис. 2.1), включающих модули соответствующего типа. На рисунке также указаны текущие модули и зависимости между ними. Зависимости от dslab-core (от него зависят все имеющиеся универсальные и специализированные модули) не указаны, чтобы не загромождать рисунок. Таким образом, модули могут зависеть от модулей с нижних слоев, но не наоборот.

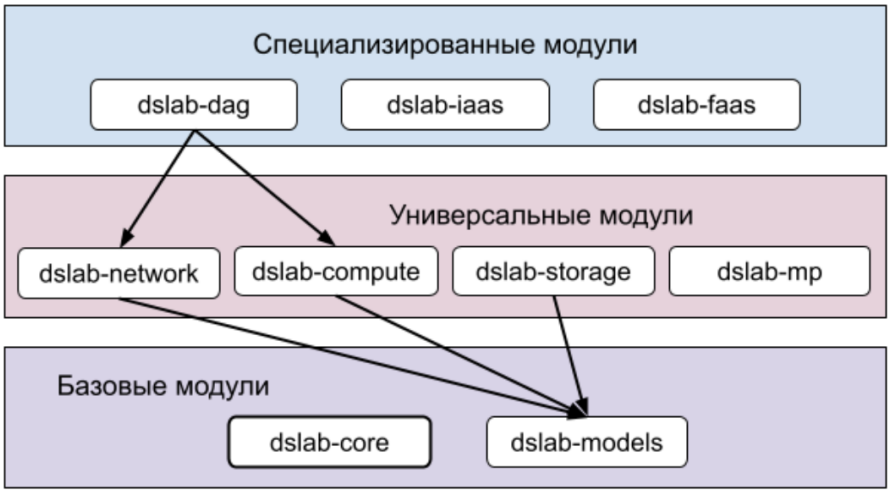

Рис. 2.1: Архитектура DSLab

Описание архитектуры основано на документации проекта [3]. Более подробно про проект можно прочитать в его описании [4], его требования [15] и сравнение с аналогами [14]. Реализации модулей представлены в репозитории [12].

Таким образом пользователь при разработке собственной симуляции может либо использовать уже готовые разработанные компоненты, либо реализовывать свои и произвольно их связывать.

Основным процессом в симуляции является создание событий и исполнения процессов реагирования на них.Разные компоненты генерируют события друг для друга и с помощью низкоуровневого модуля dslab-core обмениваются ими. Чтобы как-то реагировать на событие, каждый компонент реализует единственный обобщенный метод в который dslab-core передает нужное событие. При обработке события компоненты генерируют новые и таким образом цикл симуляции замыкается.

Получается, что сейчас реагировать на пришедшие события можно только в одном месте – в той самой функции-callback которую позовет dslab-core. В случае, когда событий немного – это лаконично выглядит и этим удобно пользоваться. Но в более сложных алгоритмах с большим количеством различных событий, и, что важнее, цепочками событий, на которые нужно последовательно реагировать, подобная модель становится не самой удобной. С такой единой точной входа многие последовательные логичные действия оказываются фрагментированы (разбросаны по разным участкам кода).

Хотелось бы иметь альтернативную возможность писать на языке программирования более понятный с первого взгляда алгоритм.

Как раз такой возможностью является написание асинхронного кода. Тогда различные сложные куски логики можно было бы выносить в отдельные функции, держать их вместе, просто исполнение бы прерывалось в ожидании событий.

2.2 Цель и основные требования

- Реализовать асинхронное расширение для существующего ядра dslab-core.
- Добавить примеры использования нового функционала высокоуровневыми компонентами.
- Написать подробную документацию нового API и покрыть реализацию тестами.
- Новое ядро должно быть реализовано эффективно: не должны возникать дополнительные (асимптотически) накладные расходы по сравнению со стандартным ядром.

3 Актуальность и значимость

3.1 Значимость проекта DSLab

Практически все современные информационные и вычислительные системы являются распределенными. Связано это с растущими объемами вычислений и обрабатываемых данных, требованиями к производительности, надежности и масштабируемости. Алгоритмическое обеспечение распределенных систем (например, алгоритмы управления ресурсами, планирования задач, балансировки нагрузки, членства в группе, консенсуса) является предметом активных исследований. В силу масштабов современных систем, сложности их реализации и недетерминированного характера, исследование алгоритмов и тестирование их реализаций в реальных системах существенно затруднено. Поэтому часто подобные исследования проводятся на аналитических и имитационных моделях, описывающих существенные для решаемой задачи аспекты поведения системы и проводимых вычислений. Использование моделей позволяет значительно удешевить эксперименты, сократить время их проведения и обеспечить воспроизводимость их результатов [4].

3.2 Преимущества асинхронного подхода

Как уже было описано в мотивации постановки цели этой работы, главное преимущество асинхронной модели – удобство при написании сложных многоступенчатых алгоритмов.

Это повысит читаемость кода. Процесс принятия решения о сохранении файла в распределенном хранилище в такой парадигме мог бы выглядеть таким образом (псевдокод):

```
async fn add_file_to_storage(some_file) {
    send_file_to_all_replicas(some_file);
    result = wait_for_confirmation_from_all().await;
    if result.has_quorum {
        send_commit_to_replicas(result.nodes);
        wait_for_commit_confirmation_from(result.nodes).await;
        send_ok_message_to_user();
    } else {
        send_reject_message_to_user();
    }
}
```

Рис. 3.1: Псевдокод асинхронного взаимодействия нод в симуляции

Посмотрев на функцию можно понять, что она делает, потому что логика последовательна и удобно разбита на подфункции. У нас есть возможность написать такой алгоритм на верхнем уровне, а не обрабатывать сообщения всех типов от всех реплик в единой точке входа.

3.3 Возможные недостатки асинхронного подхода

Программировать в парадигме асинхронного взаимодействия требует соответствующей подготовки пользователя. Не смотря на то, что в любой момент времени исполняется только одна функция, и проблем многопоточного программирования вида `data-race` не возникает, нужно постоянно держать в голове, что «параллельно» могут быть запущены другие процессы, которые могли поменять общую память в то время, когда функция была неактивна.

4 Существующие работы и решения

Подобный подход уже был реализован в других симуляторах, например в SimGrid [10]. Главное отличие подхода DSLab от SimGrid – наличие дискретно-событийного моделирования. В SimGrid компоненты общаются между собой асинхронно через `MailBox`-ы (аналог канала из языка Go). Также код из примера сложно назвать легко читаемым, потому что SimGrid – низкоуровневый фреймворк.

Более близким к желаемой реализации является использование корутин языка Kotlin в проекте OpenDC [13]. К сожалению, проект не содержит достаточно разнообразных примеров использования симуляции в асинхронном контексте, но простой пример запуска симуляции в асинхронном контексте [9] очень похож на ожидаемый опыт использования асинхронности в DSLab.

Опыт других проектов не очень хорошо подходит как опора для разработки нового решения из-за специфики языка Rust и внутренней архитектуры симулятора.

5 Управление исполнением

Чтобы дать возможность пользователю писать асинхронный код, нужно создать саму возможность обрабатывать события асинхронно. Для этого нужно написать свой executor задач в DSLab, который будет исходя из внутренней логики (наступление времени, когда событие нужно доставить получателю) понимать, когда и какую асинхронную задачу нужно разбудить, и дать ей продолжить исполнение.

К счастью, язык Rust дает возможность пользователям самим управлять процессом рантайма: с помощью библиотеки futures [2] такое поведение может быть реализовано (что на простом примере и описано в официальной документации [1]).

В основе подхода асинхронного исполнения в языке Rust лежат stackless-корутины. Каждая асинхронная функция превращается в state-machine. На высоком уровне это объект с фукнциями-callback-ами, внутри которых нет асинхронного ожидания. Т.е. компилятор «разрезает» за нас эту функцию на части, которые нам нужно будет исполнить. Это ровно то, что сейчас разработчик, использующий DSLab делает самостоятельно. В существующей модели исполнения каждый компонент предоставляет ядру набор callback-ов на каждое событие, которые должен написать пользователь. Из-за этого код получается искусственно «разре-

зан» на части в местах ожидания событий. В асинхронном подходе мы пишем стандартную фукнцию, используя `.await` для ожидания, и компилятор разрезает эту функцию за нас. Наглядно можно продемонстрировать разницу на вот таком участке эквивалентного псевдокода:

```
fn on_start_action(&self) {
    // do 1
}
fn on_first_event(&self) {
    // do 2
}
fn on_second_event(&self) {
    // do 3
}
```

(a) Синхронный код (разрезан на части разработчиком)

```
async fn action(&self) {
    // do 1
    wait_for_first_event().await;
    // do 2
    wait_for_second_event().await;
    // do 3
}
```

(b) Асинхронный код (разрезан на части компилятором)

Рис. 5.1: Сравнение синхронного и асинхронного кода на языке Rust

Как можно ожидать, ни один из этих примеров не исполняется сам. В случае синхронного примера (5.1a) очевидно, что у него нет «единой точки запуска», т.е. каждую из трех функций-callback-ов должно позвать ядро исполнения симуляции (dslab-core).

Для асинхронного примера (5.1b) действует то же самое правило: на при наступлении соответствующего события нужно «разбудить» эту функцию и дойти до следующего места ожидания. Именно такую возможность и нужно поддержать в dslab-core.

Главный компонент исполнителя задач – логика, по которой нужно ставить на исполнение ту или иную задачу в dslab-core уже реализована. Это и есть основа всей дискретно-событийной симуляции. Остается только добавить возможность «возобновлять» исполнение асинхронной функции как раз в том месте, где сейчас вызывается заранее зарегистрированный пользовательский callback.

6 Дизайн и структура ядра DSLab (dslab-core)

6.1 Событие

Основа дискретно-событийного моделирования – события. В фреймворке DSLab они представленны структурой `Event` со следующими полями:

- `id: EventId` – уникальный идентификатор события во всей симуляции
- `time: f64` – время наступления события
- `src: Id` – идентификатор компонента симуляции, создавший это событие
- `dest: Id` – идентификатор компонента симуляции, которому это событие предназначается
- `data: Box<dyn EventData>` – произвольная «полезная нагрузка» события, которая может иметь любой тип и определяется пользователем.

6.2 Обработчик событий

Обрабатывать события симуляции в DSLab может любой класс, реализующий интерфейс `EventHandler`. Этот интерфейс содержит единственную функцию:

```
fn on(&mut self, event: Event)
```

Именно эта функция вызывается как callback для обработки полученного события.

6.3 Основные классы симуляции

Модуль dslab-core состоит из нескольких основных классов, логически разделенных друг от друга:

- `Simulation` – основной класс, отвечающий за симуляцию. С него начинается создание любой симуляции. Он же предоставляет API для контроля за событиями. Основные методы:

- `Simulation::step() -> bool` – сделать один «шаг» симуляции. Это значит взять из очереди событие с наименьшим временем наступления и вызвать обработчик у компонента, которому это событие предназначается. Возвращает `true` если все перечисленные действия успешно завершились.
- `Simulation::time() -> f64` – вернуть текущее время симуляции (вещественное число). Оно совпадает с временем последнего обработанного события.
- `Simulation::create_context(name: &str) -> SimulationContext` – Создать контекст для компонента с именем `name` симуляции, с помощью которого тот будет отправлять события.
- `Simulation::add_handler(name: &str, handler: dyn EventHandler)` – Добавить обработчик событий для компонента с именем `name`. Этот метод принимает любой класс, реализующий публичный интерфейс EventHandler. Когда компоненту `name` нужно будет доставить какое-либо событие (в рамках исполнения метода Simulation::step()), именно эта функция будет вызвана для обработки события.
- Набор методов для генерации псевдослучайных последовательностей. Опустим эти детали, поскольку эта часть не очень важна в данной работе.

- `SimulationState` – класс состояния симуляции. Является полем класса `Simulation` и хранит в себе всю информацию о симуляции: очереди событий и отмененные события.
- `SimulationContext` – контекст одного компонента симуляции. Является продуктом вызова функции Simulation::create_context. В предполагаемом дизайне системы является полем класса, отвечающем за функционал отдельного компонента симуляции. Основные методы API для взаимодействия с симуляцией:
  - `SimulationContext::emit<T>(data: T, dest: Id, delay: f64) -> EventId` – добавить в симуляцию событие. Принимает данные произвольного типа T и создает событие с этими данными, которое должно быть доставленно компоненту симуляции с идентификатором `dest` через время `delay` (разумеется, речь идет о внутреннем времени симуляции). В момент, когда событие будет доставлено, у соответствующего класса будет вызывана функция-callback. Тот класс в свою очередь (уже с помощью своего SimulationContext::emit) сможет добавлять новые события в симуляцию, и таким образом цикл замыкается, симуляция продолжается.
  - `SimulationContext::cancel_event(id: EventId)` – отменить наступление события с заданным идентификатором.

7 Добавление асинхронности

Поскольку существующий `dslab-core` – уже готовый планировщик задач, то было решено расширить его функционал для работы с асинхронными функциями. При таком подходе сохраняется обратная совместимость со всеми уже написанными компонентами: можно использовать модель callback-ов для других компонент и даже совмещать две парадигмы внутри одного компонента.

7.1 Ожидающие компоненты

Ключевой вопрос, который предстояло решить: «Каким образом будет организовано асинхронное ожидание внутри симуляции?» Для этого нужно обратиться к смыслу дискретно-событийного моделирования: компоненты обмениваются между собой событиями. Соответственно, разрыв в процессе работы какого-то компонента может произойти по двум причинам:

1. Необходимо дождаться какого-то события от другого компонента
2. Необходимо подождать какое-то количество времени (внутри симуляции) и возобновить исполнение (иными словами «завести таймер в симуляции»). Поддержать такое становится возможно с помощью предыдущего пункта и отправки событий самому себе.

Из этих двух потребностей и вытекает пользовательский интерфейс – нужно уметь блокироваться, ожидая событие от другого компонента. Для этого в `dslab-core` был добавлен `Future` для события. Таким образом каждый, кто хочет дождаться какого-то события асинхронно, отправляет соответствующий запрос к симуляции (или через SimulationContext) и получает на выход `Future` на это событие.

7.2 Ключи ожидания

Для того, чтобы сохранить константную асимптотику обработки события внутри ядра, необходимо для каждого события понимать, куда его нужно доставить за константное время. Было решено каждому событию сопоставить некоторый «ключ». В реализации `dslab-core` [11] эта структура называется `AwaitKey` и содержит следующие поля:

- `src: Id` – идентификатор компонента, отправляющий событие

- `dst: Id` – идентификатор компонента, получающий событие
- `msg_type: TypeId` – идентификатор типа «полезной нагрузки», которую несет в себе событие – уникальный для каждого типа в рамках программы
- `details: u64` – внутренние детали «полезной нагрузки» для более точной настойки ожидания, получаемые за константное время (заполняются нулем, если таковые отсутствуют).

Как можно заметить, при наличии события, такой вот `AwaitKey` можно посчитать за константное время, таким образом каждый ожидающий компонент обращается к симуляции с подобным ключом, «регистрируясь» на ожидание события конкретного типа от конкретного другого компонента.

При наступлении следующего события в методе Simulation::step() можно за константное время посчитать его `AwaitKey` и проверить, подписывался ли кто-то на получение конкретно этого события или нет.

В случае, если такой ожидающий компонент находится, событие доставляется уникально в это место ожидания. После этого возобновляется работа прерванной на ожидание задачи до следующего ожидания (см пример 5.1b).

В случае, если конкретно на полученном `AwaitKey` никто не ждет, задействуется стандартная процедура вызова функции-callback-а.

7.3 Детализированное ожидание

В рамках асинхронного ожидания мы можем ждать сообщение какого-то конкретного типа от какого-то конкретного компонента. Оказывается, этого не всегда достаточно.

Рассмотрим простейший пример: есть несколько нод и они общаются друг с другом через компонент сети. Тогда сигналом о завершении передачи данных будет соответствующее событие (например `DataTransferCompleted`), полученное от компонента-сети. Проблема в том, что таким образом будет заканчиваться любая передача данных в такой симуляции. Такая модель ограничивает нас в каждый момент времени держать ровно одну активную передачу данных, иначе невозможно будет однозначно определить по паре `{id сети, тип события}` какая именно из передач завершилась, и какую именно активность стоит продолжить в рамках симуляции.

С такими жесткими ограничениями использование такой модели не представляет никакого интереса, поэтому нужно еще сильнее сузить и конкретизировать запрос на ожидание события. Ровно для этого в ключе ожидания есть дополнительное поле `details`. Обычно подобные взаимодействия сопровождаются неким `request_id` или подобным полем, однозначно идентифицирующим конкретный запрос или процесс.

Возвращаясь к примеру: при отправке запроса на передачу данных мы можем возвращать пользователю идентификатор этой передачи (за которым следит компонент сети), и поскольку в сообщении `DataTransferCompleted` тоже будет присутствовать этот идентификатор, мы можем добавить его как подробности для ожидания. Получится примерно такое использование:

```
async fn process_data_transfer_request(&self, request: Request) {
    let request_id = self.network.send(request);
    async_detailed_handle_event::<DataTransferCompleted>(
        self.network.id,
        request_id,
    ).await;
}
```

Рис. 7.1: Псевдокод асинхронного ожидания детализированного события

8 Асинхронный интерфейс для пользователя

8.1 Описание API

Поскольку компоненты взаимодействуют с симуляцией через SimulationContext, то именно это API было расширено. Новый функционал `SimulationContext` можно разделить на 4 категории:

1. Старт асинхронных активностей. Чтобы пользоваться асинхронным ожиданием, нужно создать корневую асинхронную активность, из которой уже будут вызываться асинхронные функции контекста. Для этого предоставляется единственная функция:
    - `fn spawn(&self, future: impl Future<Output = ()>)` – запуск асинхронной активности без возможности как-либо на нее повлиять. Стандартный паттерн для

асинхронного рантайма. Такой активностью может быть любая асинхронная функция или метод класса с пустым возвращаемым значением.

2. Завести таймер и подождать произвольное время внутри симуляции:

   - `async fn async_wait_for(&self, timeout: f64)` – возвращает объект Future, у которого будет пустой результат через `timeout` внутреннего времени симуляции. Чтобы получить результат, нужно позвать встроенный `.await`.

3. Асинхронное ожидание событий без таймаута.

   - `async fn async_handle_event<T>(&self, src: Id) -> (Event, T)` – асинхронно дождаться события с «полезной нагрузкой» типа T от компонента с идентификатором `src`. Возвращает объект типа Future, на который нужно сделать `.await`.
   - `async fn async_detailed_handle_event<T>(&self, src: Id, details: u64) -> (Event, T)` – асинхронно дождаться события с «полезной нагрузкой» типа T и внутренней информацией details (см псевдокод в примере 7.1) от компонента с идентификатором `src`. Возвращает объект типа Future, на который нужно сделать `.await`.

4. Асинхронное ожидание с таймаутом. Тут можно справедливо заметить, что стандартные функции работы с типом `Futures` в `Rust` позволяют вывести этот пункт из предыдущих двух и макроса `fututes::select!` из библиотки `futures` [2]. Но т.к. это довольно востребованный сценарий альтернативного ожидания было принято решение поддержать его отдельно. Это сокращает накладные расходы внутри dslab-core по причинам внутренней реализации.Результатом такого ожидания будет либо событие, либо сообщение об истекшем таймере. Реализовано это с помощью `enum AwaitResult`, который либо содержит событие либо сообщение об истекшем таймере. Методы (все возвращают объект типа `Future`, на которые нужно сделать `.await`):

   - `async fn async_wait_for_event<T>(`
     `&self, src: Id, timeout: f64,`
     `) -> AwaitResult<T>` – для обычного ожидания
   - `async fn async_detailed_wait_for_event<T>(`
     `&self, src: Id, details: u64, timeout: f64,`
     `) -> AwaitResult<T>` – для детализированного ожидания.

8.1.1 Добавление деталей к событиям

Поскольку события все находятся в общей очереди, то получать детали внутренней «полезной нагрузке» необходимо будет самому компоненту `dslab-core`, во время обработки событий. Для этого предоставлен следующий интерфейс:

```
fn register_details_getter_for<T: EventData>(
                                        &self, getter: fn(&dyn EventData) -> u64)
```

Такой метод есть у Simulation и у SimulationContext. Для каждого типа событий, на которых планируется выполнять детализированное ожидание (за все время симуляции), необходимо через один из этих методов зарегистрировать функцию, которая будет определять детали. Подробнее можно посмотреть в примере `async-event-details` в DSLab [5]

8.2 Примитивы синхронизации

Был реализован простейший примитив синхронизации – канал передачи данных. Семантика очень похожа на канал из языка `Go`, только канал бесконечного размера, и блокирующими являются только операции получения данных из канала. Эталонный пример его использования можно посмотреть в `async-event-details` [5].

Стоит отметить, что на похожих каналах основано все взаимодействие в фреймворке `SimGrid` [10] Там акторы общаются между собой через похожие `MailBox-ы`.

8.3 Ограничения при использовании

Для удобства использования асинхронного фреймворка и быстроты его работы необходимо было пойти на некоторые компромиссы:

- Одновременно обычное и детализированное ожидание событий невозможно, потому что отсутствие детализированного ожидание приравнивается к ожиданию на «нулевых» деталях, а значит события с ненулевыми деталями получены за константное время быть не могут.
- При асинхронном взаимодействии можно использовать только `immutable` ссылки. Это связано с ограничениями языка `Rust`: на один объект может быть не более одной

`mutable`-ссылки, одновременно `mutable` и `immutable` ссылки существовать не могут. Из-за этого приходится пользоваться проверкой этих правил в runtime с помощью класса `RefCell`. Заметим, что на объект создается новая ссылка каждый раз при вызове метода spawn (ссылка на `self`), поэтому все методы контекста требуют `immutable` ссылка на `self`

- Объекты, отвечающие за симуляцию компонентов стоит разрушать только методами, предоставленными в симуляции. Нужно, чтобы избежать неопределенного поведения программы. При задуманном дизайне разработки компоненты вообще не придется разрушать и перемещать до окончания работы программы (симуляции).

9 Внутреннее устройство и реализация

Полная реализация приведена в моем форке репозитория `DSLab` [11]. Этот раздел будет дополнен деталями.

9.1 Задача

Любая асинхронная активность начианется с метода spawn. Таким образом корневая асинхронная функция становится задачей, с которой мы возобновляем исполнение. Задача обладает способностью разбудить себя (хранит в себе ссылку на очередь задач и при вызове соответствующего метода добавляет себя в эту очередь, чтобы потом исполнитель задач ее исполнил).

9.2 Исполнитель задач

К полям Simulation добавляется еще поле типа `Executor` – исполнитель задач. Он предоставляет единственный метод: `fn process_task()` – выполнить первую в очереди задачу. В случае, если асинхронная активность завершилась, ее объект нужно разрушить, чтобы не было утечек памяти.

9.3 Разделяемое состояние

Чтобы передавать данные между `dslab-core` и пользовательским кодом асинхронно, у каждого `Future` есть разделяемое состояние – ссылка на разделяемую структуру в динамической памяти. Эта ссылка передается в `dslab-core`, чтобы в эту «корзинку» можно было «положить» пришедшее событие для конкретного AwaitKey. Эта же ссылка передается в реализованные `Future`, из которой они «достают» результат и возвращают его пользователю.

10 Эксперименты и замеры производительности

10.1 Раздельная сборка

Поскольку необходимость поддерживать асинхронную функциональность замедляет ядро, было принято решение включать ее по опции (в языке Rust к пакетам можно добавлять разные `features`, которые влияют на сборку). Таким образом, если асинхронность не используется в каком-то проекте, ее можно не подключать и не получить никакой просадки производительности.

Аналогичная логика работает с детализированным ожиданием – оно также создает дополнительную нагрузку на ядро на обработку каждого события, поэтому этот функционал тоже включается по опции. Все возможные опции подключения пакета из примеров перечислены ниже (относительный путь до пакета `dslab-core` может быть другим):

- `dslab-core = { path = "../../crates/dslab-core"}` – асинхронность не поддерживается. Максимальная производительность.
- `dslab-core = { path = "../../crates/dslab-core features = ["async_core"] }` – поддерживается весь интерфейс, кроме функций детализированного ожидания.
- `dslab-core = { path = "../../crates/dslab-core features = ["async_details_core"] }` – поддерживается весь функционал API. Производительность снижена относительно стандартного `dslab-core`.

10.2 Замеры на схожих примерах

10.2.1 Ping-pong

Example	Hosts	Peers per host	Iterations	Elapsed time	Events/s	Iterations/s
async-ping-pong	100000	100	100	16.45s	1234103	6.08
ping-pong	100000	100	100	8.20s	2452670	12.20

Таблица 10.1: Сравнение производительности async-ping-pong и ping-pong.

Видим двукратную деградацию в производительности. Естественно это связано с дополнительными расходами на асинхронность. Для каждого события необходимо посчитать его AwaitKey, достаить его в соответствующий `SharedState`, поставить задачу на исполнение в `Executor` и выполнить задачу.

Однако, `ping-pong` – это довольно вырожденный пример. Обычно значительное время работы симуляции занимает как раз работа пользовательского кода (в этом примере она полностью отсутствует). С точки зрения приближенности к реальным условиям показательнее будет следующий пример.

10.2.2 Master-workers

Сравнивался существующий синхронный пример `master-workers` [8] с асинхронным аналогом `async-master-workers` [6]. Примеры различаются лишь подходом к обработке событий, логика и симуляция в примерах полностью совпадают. Оба примера были запущены с параметрами 100 хостов и 100000 задач. Получились такие результаты:

Example	Tasks	Hosts	Elapsed time	Scheduling time	Events per second
async-master-workers	100000	100	5.97s	4.31s	285059
master-workers	100000	100	5.19s	4.31s	328392

Таблица 10.2: Сравнение производительности async-master-workers и master-workers.

Тут как можно заметить результаты не такие разные, т.к. основное время тратится в пользовательской симуляции (на Scheduling time), а не на обмен событиями. Использование асинхронности никак не замедляет и не ускоряет пользовательский код не связанный с обменом событий.

11 Производительность при увеличении симуляции

Тут будут построены еще графички (...)(.....), показывающие производительность на тех двух примерах с разными входными данными чтобы показать $O(NlogN)$ зависимость от числа компонент и линейную зависимость от числа событий в симуляции.

12 Логгирование событий

Поскольку асинхронное управление событиями было добавленно как расширение к уже существующему ядру, дополнительное логгирование событий не понадобилось, т.к. оно уже присутствовало в `dslab-core`.

13 Результаты и планы на будущее

Главным результатом работы стало работающее расширение модуля `dslab-core` [11] и написанные к нему примеры [7][5][6].

В качестве дальнейшей работы можно улучшить документацию, которая является несовершенной.

Список литературы

[1] *Asynchronous Programming in Rust*. URL: https://rust-lang.github.io/async-book (дата обр. 31.01.2023).

[2] *Rust crate «futures»*. URL: https://crates.io/crates/futures (дата обр. 31.01.2023).

[3] *Архитектура проекта DSLab*. URL: https://docs.google.com/document/d/12CcGpdulqMJppAYoNrOdpIWZYEa3flXn9JMX5fdyAnY/edit (дата обр. 31.01.2023).

[4] *Описание проекта DSLab*. URL: https://docs.google.com/document/d/1Z8ivtqLRMFG-EfaiRaWBT8dNHFjYjuDhzObWIYkc-A8/edit (дата обр. 31.01.2023).

[5] *Пример async-event-details из DSLab*. URL: https://github.com/nogokama/dslab/tree/modify_dslab_core/examples/async-event-details/src (дата обр. 11.05.2023).

[6] *Пример async-master-workers из DSLab*. URL: https://github.com/nogokama/dslab/tree/modify_dslab_core/examples/async-master-workers/src (дата обр. 11.05.2023).

[7] *Пример async-ping-pong из DSLab*. URL: https://github.com/nogokama/dslab/tree/modify_dslab_core/examples/async-ping-pong/src (дата обр. 11.05.2023).

[8] *Пример master-workers из DSLab*. URL: https://github.com/nogokama/dslab/tree/modify_dslab_core/examples/master-workers/src (дата обр. 11.05.2023).

[9] *Пример использования асинхронного кода в фреймворке OpenDC*. URL: https://github.com/atlarge-research/opendc/blob/master/opendc-simulator/opendc-simulator-core/src/main/kotlin/org/opendc/simulator/kotlin/SimulationBuilders.kt (дата обр. 31.01.2023).

[10] *Пример использования асинхронного кода в фреймворке SimGrid*. URL: https://github.com/osukhoroslov/dslab/blob/main/examples-other/simgrid/ping-pong/process.cpp (дата обр. 31.01.2023).

[11] *Реализация асинхронного ядра*. URL: https://github.com/nogokama/dslab/tree/modify_dslab_core/crates/dslab-core/src (дата обр. 11.05.2023).

[12] *Репозиторий DSLab*. URL: https://github.com/osukhoroslov/dslab (дата обр. 31.01.2023).

[13] *Репозиторий проекта OpenDC*. URL: https://github.com/atlarge-research/opendc (дата обр. 31.01.2023).

[14] *Существующие решения и аналоги DSLab*. URL: https://docs.google.com/document/d/1H2711nGiS6m7QTo4pMfmrIBmg4llLKUhQzDiFNIw-UU/edit (дата обр. 31.01.2023).

[15] *Требования к проекту DSLab*. URL: https://docs.google.com/document/d/1CprULnQiVSWTXiBkx90CSEi4tgkUhM3R6VeCp1CsRwo/edit (дата обр. 31.01.2023).