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

Факультет компьютерных наук
Образовательная программа «Прикладная математика и информатика»

Отчет о программном проекте на тему:

Реализация поддержки асинхронного программирования для фреймворка DSLab

Выполнил:

студент группы БПМИ206
Макогон Артём Аркадьевич

_____ (подпись)

17.05.2023 (дата)

Принял руководитель проекта:

Сухорослов Олег Викторович
Кандидат технических наук, доцент
Факультета компьютерных наук НИУ ВШЭ

_____ (подпись)

17.05.2023 (дата)

Москва 2023

Содержание

Аннотация

Современные информационные и вычислительные системы все чаще становятся распределенными из-за увеличения объема вычислений и данных, а также требований к производительности и масштабируемости. Исследования по алгоритмическому обеспечению распределенных систем проводятся для разработки эффективных алгоритмов управления ресурсами, планирования задач, балансировки нагрузки и других задач. Аналитические и имитационные модели используются для упрощения и ускорения экспериментов, изучения поведения таких систем в разных ситуациях. `DSLab` – программный фреймворк для имитационного моделирования и тестирования распределенных систем, предоставляющий множество возможностей как для написания новых алгоритмов, так и использования готовых модулей. В основе любой симуляции `DSLab` лежит дискретно-событийный подход описания моделей и приложений, где события обрабатываются в пользовательских функциях (`callback`-ах). В рамках проекта добавлена возможность управлять событиями асинхронно, а также комбинировать эти подходы. Благодаря новому функционалу можно писать более выразительный и понятный код, что ускоряет процесс разработки алгоритмов и делает фреймворк более привлекательным для исследователей.

Ключевые слова

Распределенные системы, симуляция, асинхронность, DSLab

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 исполнения имитационной 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 предметной области

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.

`DSLab`[16] предлагает широкий набор инструментов для тестирования различных алгоритмов. Более подробно про проект можно прочитать в его описании [7], его требования [19] и сравнение с аналогами [18].

2.2 Архитектура 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.

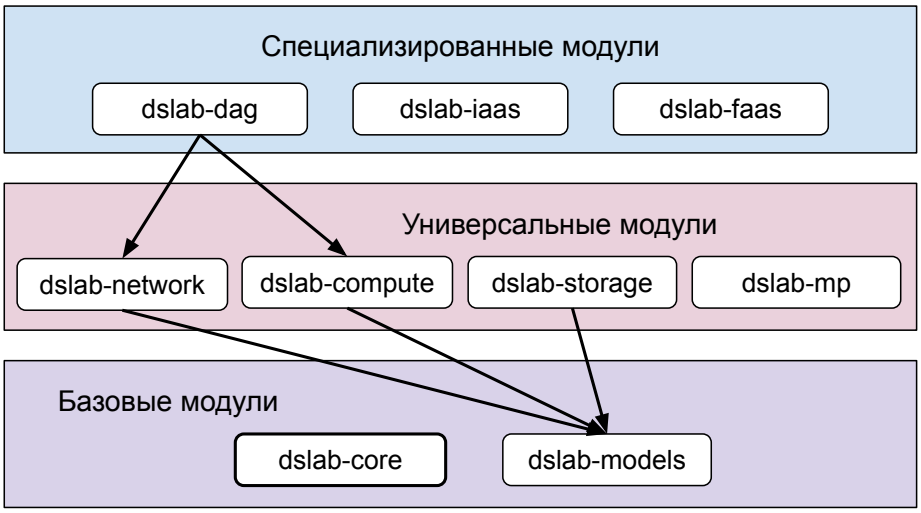

Рис. 2.1: Архитектура DSLab

Описание архитектуры основано на документации проекта [6]. Реализации модулей представлены в основном репозитории [16].

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, 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 точкой входа многие последовательные логичные действия оказываются фрагментированы (разбросаны по разным участкам кода, подробнее можно смотреть разбор примеров из главы 10).

Хотелось бы иметь альтернативную возможность писать на языке программирования более понятный с первого взгляда алгоритм. Как раз такой возможностью является написание асинхронного кода.

2.3 Преимущества асинхронного подхода

Для описания моделей, тестов и алгоритмов в фреймворке используется язык `Rust`, который на уровне языка поддерживает возможность писать асинхронный код. Главные преимущества асинхронной парадигмы программирования – удобство при написании сложных многоступенчатых алгоритмов и повышение читаемости 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. 2.2: Псевдокод асинхронного взаимодействия нод в симуляции

Посмотрев на функцию можно понять, что она делает, потому что логика последовательна

и удобно разбита на подфункции. Используя асинхронность, у нас есть возможность написать такой алгоритм на верхнем уровне.

2.4 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 (подробнее о реализации см. в главе 9), 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.

2.5 Постановка задачи

Целью является реализация поддержки асинхронного программирования для фреймворка `DSLab`. Для этого необходимо:

- Реализовать асинхронное расширение для существующего ядра dslab-core.
- Добавить примеры использования нового функционала высокоуровневыми компонентами.
- Написать подробную документацию нового API и покрыть реализацию тестами.

2.6 Полученные результаты

После проделанной работы удалось добавить возможность управлять событиями в симуляции асинхронно. Были написаны схожие примеры с уже существующими, проведены замеры производительности и сравнение семантики получившегося кода(глава 10). Код расширенного асинхронного ядра доступен в репозитории[15].

3 Существующие работы и решения

Подобный подход уже был реализован в других симуляторах, например в SimGrid[2]. Главное отличие подхода `DSLab` от `SimGrid` – наличие дискретно-событийного моделирования.

В SimGrid компоненты (акторы) общаются между собой асинхронно через MailBox-ы (аналог канала из языка Go). Код получается довольно объемный и непростой, но поскольку это самый близкий аналог, поддерживающий асинхронное программирование, его стоит разобрать. Ниже приведена (очень сокращенная) реализация самого простой примера процесса ping-pong в фреймворке SimGrid. Полный код примера можно посмотреть в репозитории[14].

```
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 and send PONG back */
        } else if (msg->type == MessageType::PONG) {
            /* handle PONG and report Complete if applicable */
        } else if (msg->type == MessageType::STOP) {
            stopped = true;
        }
    }
}
```

Рис. 3.1: Код процесса ping-pong в фреймворке SimGrid

Из преимуществ сразу можно отметить, что процесс хранит свое состояние (количество оставшихся ping-ов, которые нужно сделать) на стеке функции и удобно его использует. Исполнение прерывается в методе MailBox::get() до момента получения сообщения. Подобный подход к асинхронности будет внедрен в DSLab.

Дальше можно выделить ряд недостатков, которых хотелось бы избежать. У этого процесса

все еще единая точка входа (`MailBox* in`), и нужно обрабатывать все возможные варианты приходящего события. В этом плане нет большого различия с моделью `callback`-ов. Даже в более сложном примере `SimGrid-master-workers`[13] все равно есть единая точка входа обработки событий, которые распределяются по хэндлерам на основе строки-префикса названия события. Также во всей симуляции некоторое количество кода уходит на ручное управление памятью (выделение с помощью `new` и освобождение с помощью `delete`). Такой подход повышает вероятность ошибки, и, как следствие, утечки памяти из симулятора. Это скорее является особенностью языка `C++`, а поскольку `DSLab` написан на `Rust`, подобной проблемы не предвидится.

Альтернативной реализацией асинхронности в симуляции является использование корутин языка `Kotlin` в проекте `OpenDC` [17][3]. К сожалению, проект не содержит достаточно разнообразных примеров использования симуляции в асинхронном контексте, но простой пример запуска симуляции [12] 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.

4 Описание функциональных и нефункциональных требований к проекту

4.1 Функциональные

- Код расширенного ядра должен иметь полную обратную совместимость со стандартным ядром, т.е. поддерживать возможность добавлять зависимости от уже реализованных модулей и программировать в модели `callback`-ов.
- Поддержка асинхронности должна предоставлять следующие возможности:
  - запустить асинхронную задачу;
  - остановить задачу на определенный промежуток времени внутри симуляции;
  - асинхронно дождаться произвольного события от другого компонента;
  - произвольно комбинировать ожидания используя методы из пакета `futures`[4].

- Реализация должна содержать логирование происходящего в асинхронном контексте, а так же обработку ошибок, которая бы помогала пользователю искать ошибки в коде.

4.2 Нефункциональные

- Код проекта должен быть написан на языке Rust и опубликован как расширение модуля `dslab-core` в репозитории проекта `DSLab`[16].
- Работа нового ядра должна быть продемонстрирована в использовании на разработанных примерах симуляций, в которых используется стандартная `callback-based` модель. Должны быть описаны преимущества и недостатки нового подхода.
- Расширение для ядра должно быть реализовано эффективно: не должны возникать дополнительные накладные расходы (асимптотически) по сравнению со стандартным ядром.
- Код должен быть написан в соответствии со стайлгайдом проекта `DSLab`

5 Дизайн и структура ядра DSLab (dslab-core)

5.1 Событие

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

- `id: EventId` – уникальный идентификатор события во всей симуляции
- `time: f64` – время наступления события
- `src: Id` – идентификатор компонента симуляции, создавшего 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.

5.2 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.

5.3 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 компонента с именем `name` для отправки событий в симуляцию.
  - `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 компоненте `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, отмененные события,

состояние генератора последовательности случайных чисел и текущее внутреннее время симуляции.

- `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.
  - `SimulationContext::cancel_event(id: EventId)` – отменить наступление события с заданным идентификатором.

6 Управление исполнением

Чтобы дать возможность пользователю писать асинхронный код, нужно создать саму возможность обрабатывать события асинхронно. Для этого нужно реализовать `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 `std::future`[5] и пакета `futures`[4] такое поведение может быть реализовано (что на простом примере описано в официальной документации [1]).

В основе подхода асинхронного исполнения в языке `Rust` лежат `stackless`-корутины. Каждая асинхронная функция превращается в `state-machine`. В первом приближении это объект с функциями-`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 «разрезан» на части в местах ожидания событий. В асинхронном подходе мы пишем классическую функцию, используя `.await` для ожидания, а 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. 6.1: Сравнение синхронного и асинхронного кода на языке `Rust`

Как можно ожидать, ни один из этих примеров не исполняется сам. В случае синхронного примера (6.1a) очевидно, что у него нет «единой точки запуска», т.е. каждую из трех функций-`callback`-ов нужно отдельно вызвать (это делает ядро исполнения симуляции `dslab-core`).

Для асинхронного примера (6.1b) действует то же самое правило: 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 – логика, по которой нужно ставить на исполнение ту или иную задачу, – в `dslab-core` уже реализован. Остается только добавить возможность «возобновлять» исполнение асинхронной функции как раз в том месте, где сейчас вызывается заранее зарегистрированный пользовательский `callback`.

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»).

Из этих двух потребностей и вытекает пользовательский интерфейс – нужно уметь блокироваться, ожидая событие от другого компонента. Для этого в `dslab-core` был добавлен `Future` на событие. 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` [15] эта структура называется `AwaitKey` и содержит следующие поля:

- `src: Id` – идентификатор компонента, отправляющего событие;
- `dst: Id` – идентификатор компонента, получающего 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 `AwaitKey` можно посчитать за константное время, таким образом каждый ожидающий компонент обращается к симуляции с подобным ключом, «подписываясь» 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 6.1b).

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, 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. Можно справедливо заметить, что стандартные функции работы с типом `Future` в `Rust` позволяют вывести этот пункт из предыдущих двух и макроса `fututes::select!` из пакета `futures` [4]. 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.

8.2 Добавление деталей к событиям

Поскольку события все находятся в общей очереди, то получать детали внутренней «полезной нагрузки» 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[8].

8.3 Примитивы синхронизации

Был реализован простейший примитив синхронизации – безразмерная блокирующая очередь передачи данных (`UnboundedBlockingQueue<T>`). Семантика очень похожа на канал из языка `Go` бесконечного размера. Блокирующими являются только операции получения данных из канала. Эталонный пример использования можно посмотреть в `async-event-details` [8].

Сейчас этот примитив синхронизации имеет существенное ограничение, каждый полученный `Future` обязательно должен быть завершен и превращен в объект (с помощью стандартного `.await`). В частности это означает, что такую очередь нельзя использовать для альтернативного ожидания (например, с помощью макроса `select!` из библиотеки `futures`[4]). Чтобы преодолеть это ограничение, нужно реализовать более сложный канал передачи данных (см планы на будущее 11.2)

Стоит отметить, что на похожих каналах основано все взаимодействие в фреймворке `SimGrid` [14]. Там акторы общаются между собой через похожие `MailBox-ы` (пример 3.1).

8.4 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. Это значит, что для каждого события типа `T` можно будет использовать только **один** из вариантов асинхронного ожидания:
  - `async_handle_event::<T>` / `async_wait_for_event::<T>`

– `async_detailed_handle_event::<T>` / `async_detailed_wait_for_event::<T>`

Для их использования необходимо «зарегистрировать» функцию, которая будет определять детали события (раздел 8.2).

Это связано с тем, что отсутствие детализированного ожидания приравнивается к ожиданию на «нулевых» деталях, а значит совместить эти подходы за константное время нельзя. Как показывает практика реализации нескольких примеров, такая возможность не является необходимостью.

- 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 соблюдения 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 компонента требуют `immutable` ссылку на `self`. Правильное использование класса `RefCell` в асинхронном контексте требует некоторого опыта, но не является чем-то сложным.

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

8.5 Логирование событий

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

8.6 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 получить максимальную производительность.

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 = "../dslab-core"}` – асинхронность не поддерживается.

  Максимальная производительность.

- `dslab-core = { path = "../dslab-core features = ["async_core"] }` – поддерживается весь интерфейс, кроме функций детализированного ожидания.

- `dslab-core = { path = "../dslab-core features = ["async_details_core"] }` – поддерживается весь функционал API. Производительность примерно вдвое снижена относительно стандартного `dslab-core` (подробнее про замеры производительности в главе 10).

Одну из этих трехх возможностей нужно добавлять в `Cargo.toml` файл соответствующего модуля (или исполняемого примера).

9 Внутреннее устройство и реализация

9.1 Классическая `callback` модель

Процесс доставки событий по модели `callback` остался неизменным:

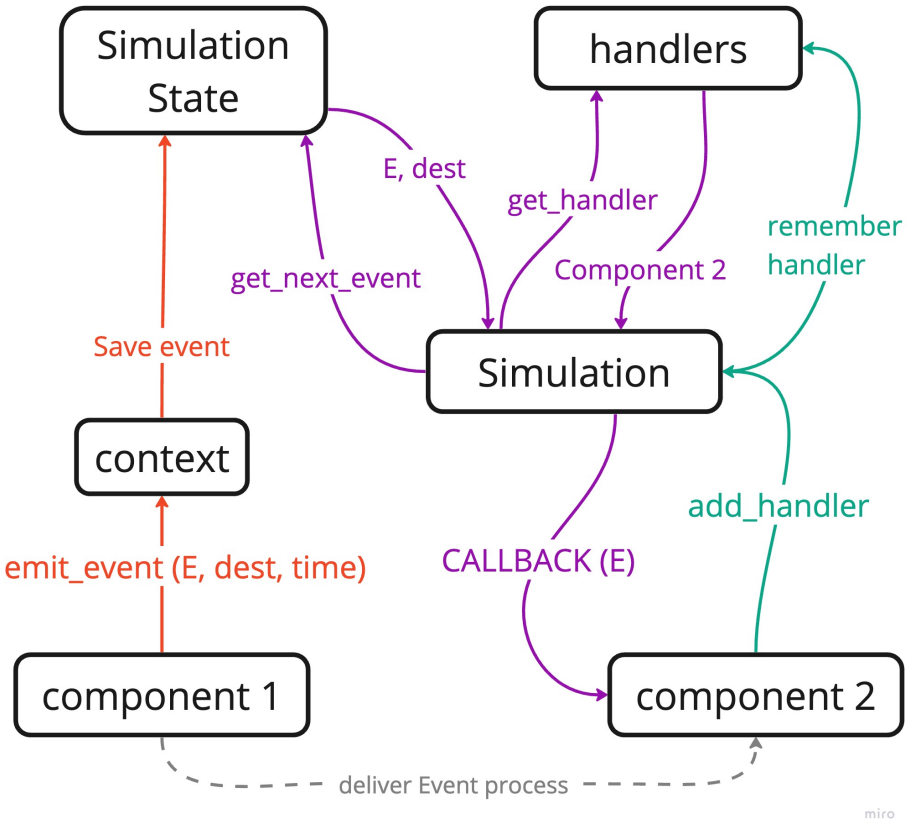

Рис. 9.1: Реализация модели `callback` в `dslab-core`

На схеме 9.1 показан процесс доставки события от `Component 1` к `Component 2`. На картинке разными цветами обозначены процессы, происходящие в разное время. Сначала получатель должен зарегистрировать себя в симуляции (зеленый цвет на схеме 9.1). Затем компонент-отправитель через свой контекст добавляет событие в симуляцию одним из методов `emit`, событие сохраняется в `SimulationState` и ожидает своего времени наступления (оранжевый цвет на схеме 9.1). Когда на очередном шаге `Simulation::step` приходит очередь это событие доставить, по нему определяется `handler` (компонент-получатель), и вызывается соответствующий `callback` с пользовательским кодом (фиолетовый цвет на схеме 9.1). В процессе обработки этого события в симуляцию по точно такому же принципу добавляются новые, цикл замыкается.

9.2 Асинхронная доставка событий

Асинхронная схема доставки события состоит из большего числа шагов. Цепочки действий, происходящие в разное время снова отображены разными цветами.

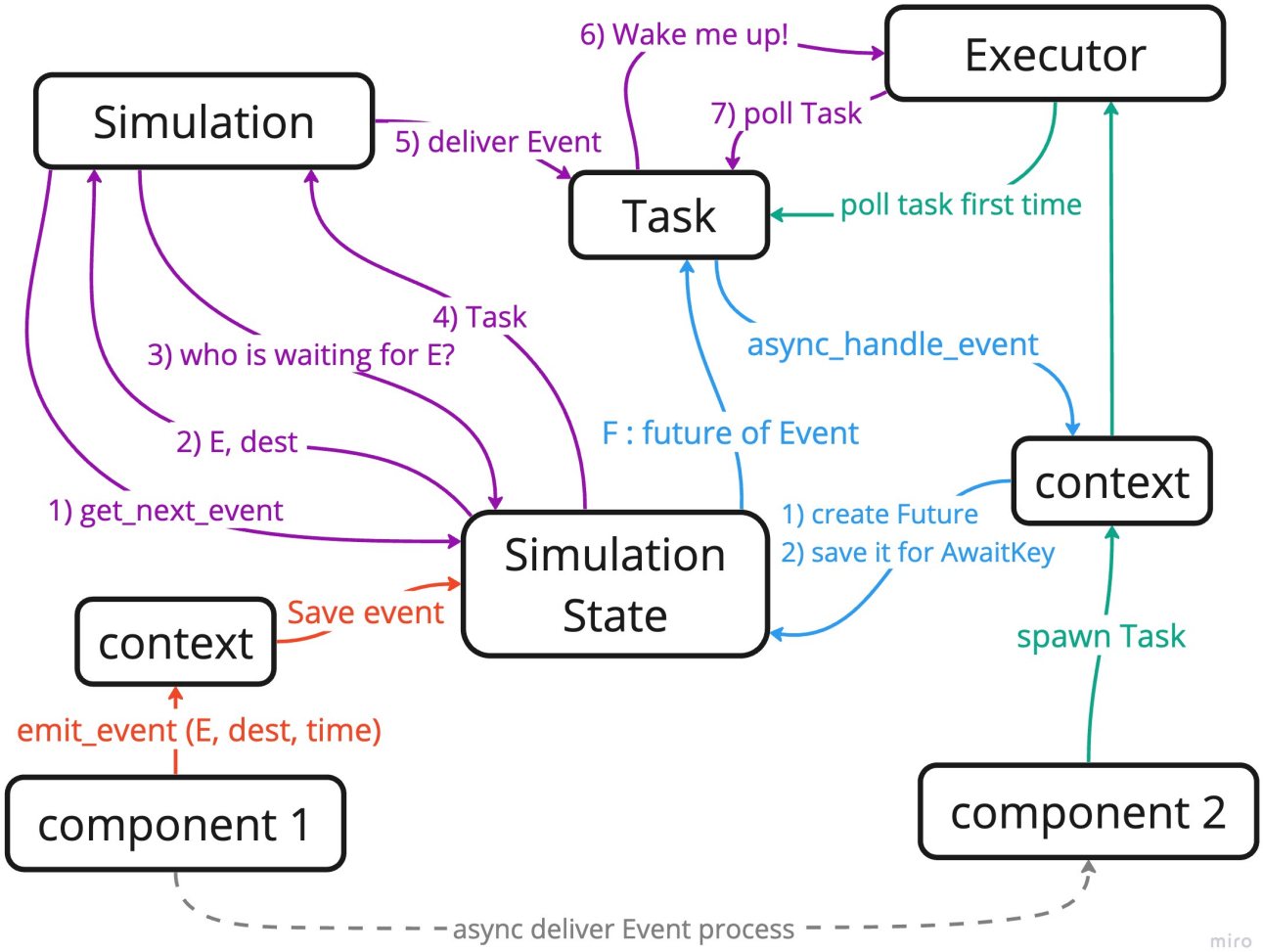

Рис. 9.2: Реализация асинхронной модели доставки событий в `dslab-core`

Задача (Task) Любая асинхронная активность начинается с метода `spawn` (обозначено зеленым цветом на схеме 9.2). Таким образом корневая асинхронная функция становится задачей, у которой будет возобновляться 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). Именно задача является тем, кто ждет какое-либо событие для компонента. Для того чтобы асинхронно дождаться какого-то события нужно обратиться к `SimulationContext` за `Future` на это событие. При создании этого `Future` считается AwaitKey, полученная информация об ожидающей задаче сохраняется по ключу в `SimulationState`, ожидание начинается, задача

прерывается (этот процесс обозначен синим цветом на схеме 9.2). Процесс отправки события никак не отличается от классической модели 9.1 – событие попадает ровно в ту же очередь.

Стоит отметить, что метод spawn не порождает никаких параллельных активностей. Для детерминизма симуляции и воспроизводимости результатов все задачи выполняются строго последовательно.

Разделяемое состояние (SharedState) Чтобы передавать данные между `dslab-core` и пользовательским кодом асинхронно, у каждого `Future` есть разделяемое состояние – ссылка на разделяемую структуру `AwaitEventSharedState` в динамической памяти. Эта ссылка сохраняется в `SimulationState` (на этапе создания `Future`, отмечен синим на схеме 9.2) по вычисленному ключу ожидания, чтобы в эту «корзинку» можно было «положить» пришедшее событие (пункт 5 в процессе доставки сообщения, отмечен фиолетовым на схеме 9.2). В конце процесса доставки задача снова ставится на исполнение и событие «достается» из разделяемого состояния.

Исполнитель задач (Executor) К полям `Simulation` добавляется еще одно поле – `Executor`. Он предоставляет единственный метод: `fn process_task()` – выполнить первую в очереди задачу. Каждая задача может сама положить себя в очередь, когда ей будет доставлено событие (последние шаги `6` и `7` доставки события, отмечено фиолетовым цветом на схеме 9.2) В случае, если асинхронная активность завершилась, ее объект нужно разрушить, чтобы не было утечек памяти.

Процесс доставки события Итого, часть отправки события полностью совпадает с классическим `callback` методом (обе части отмечены оранжевым цветом на схемах 9.1 и 9.2). Главные различия происходят в методе `Simulation::step` (также отмечен одинаковым фиолетовым цветом на обеих схемах). Опишем шаги этого метода а асинхронной модели доставки (со схемы 9.2):

1-2. Берется первое событие, которое должно наступить.

3-4. Происходит вычисление `AwaitKey` этого события.

- Если по такому ключу никто не ждет, переходим в классический режим `callback`-модели – ищем нужный `handler` и вызываем `callback` как показано на схеме 9.1.
- Если нашлась ожидающая задача, то переходим к следующему пункту.

5. Доставляем событие в разделяемое состояние (которое мы нашли по ключу-ожидания).
6. Добавляем задачу в очередь на исполнение `Executor`.
7. Исполняем задачу. В процессе ее исполнения могут порождаться новые события, так цикл симуляции замыкается.

Это лишь краткое и наглядное описание реализации. Полный код реализации приведен форке репозитория `DSLab`[15].

10 Тестирование и замеры производительности

10.1 Ping-pong

`Ping-pong` – простейший пример, который можно реализовать в `DSLab`. Вместе с этим он является идеалом для того, чтобы протестировать производительность симуляции. В симуляции регистрируются определенное количество одинаковых компонентов, которые обмениваются между собой простыми сообщениями: отсылают `Ping`-сообщения своим «соседям» в случайном порядке, и на каждое `Ping`-сообщение отвечают сообщением `Pong`. Процесс повторяется много раз.

10.1.1 Производительность

Симуляция была запущена со следующими параметрами на рабочем ноутбуке:

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, доставить его в соответствующий `SharedState`, поставить задачу на исполнение в `Executor` и выполнить задачу.

Можем также пронаблюдать линейный рост потраченного времени на симуляцию с ростом количества итераций на графике 10.1. Видно, что при общей тенденции на линейный рост

времени есть небольшие выбросы, которые можно связать с работой на ноутбуке большого количества фоновых программ, влияющих на производительность системы:

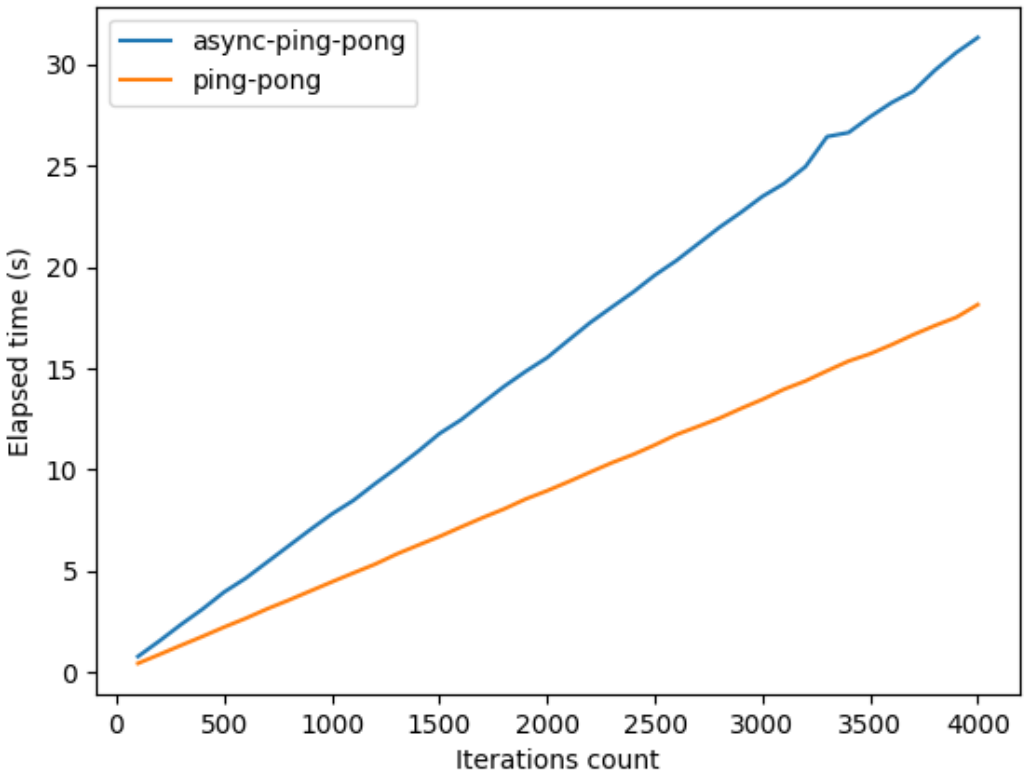

Рис. 10.1: Сравнение производительности примеров `ping-pong` и `async-ping-pong`

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.

10.1.2 Сравнение кода

Пример довольно простой, но уже на нем можно увидеть радикальные различия в подходах. В синхронном примере первый `Ping` посылается на старте, есть функция реагирования на `Ping` сообщения, а при получении `Pong` посылается новый `Ping`. Код разделен на 3 отдельных участка (код намеренно упрощен для наглядности, полный пример можно смотреть в репозитории[10]):

```rust
pub struct Process {
  iterations: u32,
}

fn on_start(&mut self) {
  let peer = /*random peer*/;
  self.send(Ping {}, peer);
}
```

(a) Старт ping-pong

```rust
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);
  }
}
```

(b) Реакция на `Ping` и `Pong` сообщения

Рис. 10.2: `Ping-pong` example

Видно, что для того, чтобы сделать заданное количество итераций процесса `ping-pong`, нужно в полях структуры самого процесса хранить информацию о том, сколько их осталось. Эту роль выполняет переменная `iterations` (определяемая на участке кода 10.2a). По трём отдельным участкам кода единого процесса работы не прослеживается. Его можно увидеть только внимательно вчитавшись в суть предоставленных `callback`-ов.

Теперь посмотрим, как абсолютно аналогичную задачу выполняет асинхронная реализация того же процесса (код также упрощен для наглядности, полный код можно смотреть в примере[10]):

```rust
fn on_start(&self) {
  self.ctx.spawn(
    self.process()
  );
}

fn on_ping(&mut self, from: Id) {
  self.send(Pong {}, from);
}
```

(a) Старт процесса и callback на `Ping`

```rust
async fn process(&self) {
  for _i in 0..self.iterations {
    let peer = /*choose peer*/;
    self.send(Ping {}, peer);

    // stop execution until receive Pong
    self.ctx.async_handle_event::<Pong>(
        peer).await;
  }
}
```

(b) Асинхронная функция работы процесса

Рис. 10.3: `Async-ping-pong` example

Можно видеть, что код функции `process` (участок кода 10.3b) очень четко выражает весь

алгоритм работы. Особенно хорошо видна разница, сравнивая с функциями классической реализации `ping-pong` (участок кода 10.2b) – там 2 отдельные функции, которые описывают только реакции, хотя на самом деле в симуляции происходит процесс, который идеально описывается циклом `for` (10.3b).

Также этот пример демонстрирует обратную совместимость с моделью `callback`-ов. В случае, когда это уместно и выразительно, ее даже нужно использовать – это видно на примере реакции на сообщения `Ping`, – мы должны просто вернуть `Pong` отправителю (участок кода 10.3a).

10.2 Master-workers

Этот пример намного более сложный с точки зрения логики пользователя. Один компонент `Master` распределяет приходящие к нему задачи между большим количеством `Worker`-ов, каждый из которых выполняет длинный pipeline для каждой задачи.

10.2.1 Производительность

Сравнивался существующий синхронный пример `master-workers` [11] с асинхронным аналогом `async-master-workers` [9]. 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 (по сравнению с примером `ping-pong`), 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.

10.2.2 Сравнение кода

Приведем (в очень сжатом виде) сравнение семантики двух примеров. Полный код можно найти в репозитории (классическая реализация `master-workers`[11], и асинхронная реализация `async-master-workers`[9]). На примере ниже показаны функции компонента `Worker` (исполняющего задачи).

```
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(request_id);
fn on_comp_started(comp_id);
fn on_comp_finished(comp_id);
fn on_data_write_completed(request_id);
fn on_data_transfer_completed(data);
```

(a) Выполнение задачи в `master-workers`

```
fn on_task_request(req) {
  self.ctx.spawn(
    self.process_task_request(req));
}
async fn process_task_request(&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;
}
```

(b) Пайплайн задачи в `async-master-workers`

Рис. 10.4: Сравнение `master-workers` и `async-master-workers`

На этом примере видно схожую разницу, только в большем масштабе. В классическом случае с `callback`-ами мы вынуждены хранить промежуточную информацию о процессе выполнения всех задач в полях структуры `Worker` (код 10.4a). Это приводит к нагромождению кода операциями по чтению и записи в эти общие структуры. Более того, по набору декларируемых `callback`-ов непонятно, как они вообще связаны, в каком порядке должны идти. Об это можно догадываться по названиям, но чтобы узнать точно, нужно подробно вчитаться, что делает `Worker` в каждом из `callback`-ов. Отдельное неудобство конкретно в этом примере доставляет то, что `callback` `on_data_transfer_completed` (код 10.4a) вызывается компонентом сети дважды при выполнении одной задачи: после загрузки изначальных данных и после выгрузки результатов, что порождает необходимость ставить внутри условия.

Как с этим справляется асинхронный код можно видеть на примере 10.4b. На запрос задачи все еще реагирует `callback` (что является здесь вполне уместным), но процесс выполнения

устроен совершенно по-другому. Сразу видна последовательность действий, прогресс задачи сохранен в «локальной» переменной и не перегружает структуру `Worker`. Для того чтобы узнать подробности каждого этапа нужно пройти в соответствующие функции (внутри которых тоже есть асинхронные ожидания).

При более внимательном рассмотрении можно увидеть, что код 10.4a в первом приближении является развернутой `state-machine` кода 10.4b (по аналогии с примером 6.1). Как раз это «разворачивание» делает за нас компилятор `Rust`, позволяя нам писать «синхронный» выразительный код.

Из-за полной обратной совместимости с моделью `callback`-ов, код компонента `Master` остался без изменений, как и все используемые в этом примере зависимости (`Network`, `Disk`, `ComputeMulticore`).

10.3 Event-details

Пример является немного упрощенной версией `master-workers` (раздел 10.2): один хост разбирает поступающие задачи и ставит их параллельно на свой процессор (подключенный модуль `ComputeMulticore`). Смысл примера в демонстрации нескольких новых функций: реализованного стандартного примитива синхронизации `UnboundedBlockingQueue<T>` и альтернативного ожидания с помощью макроса `select!` из пакета `futures`. Сделать краткую содержательную выжимку кода этого примера довольно трудно, поэтому с ним можно подробно ознакомиться в репозитории[8].

Основная идея состоит в том, что при старте вычислительной активности модуль `ComputeMulticore` присылает либо событие `CompStarted` (если удалось выделить ресурсы и начать исполнять задачу) либо событие `CompFailed` (если процессор сейчас слишком нагружен для такой задачи). Именно такое альтернативное ожидание используется в примере. Если получили `CompFailed`, то нужно дождаться завершения любого другого вычислительного процесса и попробовать снова.

`UnboundedBlockingQueue<T>` используется для удобного сохранения задач в очереди. `Callback fn on_task_request` складывает сообщение в очередь, а процесс `async fn work_loop` достает, когда это нужно, или блокируется в ожидании следующей задачи.

11 Заключение

Разработанное асинхронное расширение для ядра идеально дополняет старый подход, а не является альтернативой. Строго говоря, классическую модель `callback`-ов можно целиком выразить новыми средствами асинхронности, однако это не принесет никаких преимуществ, только симуляция будет работать медленнее (как показал пример `ping-pong`). По примерам разобранным в главе 10 можно проследить четкую закономерность: для `Stateless`-процессов[1] лучше и интуитивнее работает подход `callback`-ов, а для `Stateful`-процессов[2] асинхронность оказывается намного выразительнее и удобнее. В рамках одной симуляции часто приходится работать с большим разнообразием процессов, комбинируя использование асинхронного и классического подхода. Именно возможность бесшовной интеграции со старыми компонентами, используя при этом новые методы по необходимости, является главным достижением этой работы.

11.1 Характеристика результатов работы

В результате было разработано рабочее расширение ядра `dslab-core`[15] и написаны примеры его использования[10][8][9]. Основной функционал покрыт тестами. В тексте этой работы приведено наглядное сравнение подходов (глава 10).

Таким образом, главная цель проекта достигнута, выполнены основные требования.

11.2 Планы на будущее

Будущее развитие проекта можно разделить на 4 направления:

- Повышение удобства синтаксиса. Например, сейчас для использования детализированного ожидания необходимо для каждого типа событий регистрировать функцию по получению этих самых деталей (часть 8.2), что является, на первый взгляд, избыточным. Почти наверняка эту функцию можно красиво и эффективно реализовать с помощью макросов языка `Rust`. Это один из примеров возможного улучшения в этой области.

[1] `Stateless`-процесс не нуждается в хранении никакого состояния и может быть выполнен сразу целиком, не имеет побочных эффектов, которые нужно было бы сохранять. Классический пример – ответить сообщением `Pong` на сообщение `Ping` (код 10.3a).

[2] `Stateful`-процессы выполняются в несколько этапов и требуют сохранения промежуточного состояния между этими этапами. Из разобранных примеров самым наглядным является пайплайн выполнения задачи в `async-master-workers` (код 10.4b)

- Увеличение производительности симуляции. Для этого нужно еще более глубоко погрузиться во внутренне устройство рантайма языка `Rust`, чтобы найти потенциально неэффективные места. Возможно также применить профилирование программы и другие методы диагностики производительности программы. Это самая сложная область дальнейшей деятельности.

- Расширение асинхронной функциональности. Чтобы сделать расширенное ядро еще более выразительным, в него можно добавить, например, функцию `yield` – «уйти в конец очереди планировщика». Это техническая функция позволит сначала обработать все события текущего времени в симуляции, а затем уже продолжить исполнение текущей задачи. Это может быть полезно для реализации более сложных асинхронных алгоритмов и примитивов синхронизации, которые завязаны на передаче управления друг другу без продвижения симуляции по времени.

  Другим возможным расширением асинхронной функциональности будет являться `cancellation` – отмена работающих процессов. Это может сильно упростить написание распределенных алгоритмов, где отмены задач являются частым явлением. `Cancellation` – достаточно сложный вопрос с точки зрения правильного дизайна, который нужно будет тщательно продумать перед реализацией.

  Также расширение асинхронной функциональности является предметом исследования. Скорее всего при накоплении опыта использования появятся новые запросы или откроются проблемы, которые нужно будет решать.

- Расширение «стандартных» асинхронных инструментов. После появления метода `yield` станет возможным реализовать полноценный канал передачи данных из языка `Go`. Такой канал мог бы стать универсальным примитивом синхронизации и идеально дополнял бы асинхронный фреймворк. Однако, его реализация – непростая задача из-за большого количества возможных случаев, которые необходимо будет рассмотреть.

  Еще возможна реализация примитива `condition_variable`, но его применение видится существенно более ограниченным и специфичным чем `Go`-like канала.

  В этом пункте также не исключена исследовательская составляющая и появление новых задач с накоплением опыта использования фреймворка.

Список литературы

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

[2] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson и Frédéric Suter. "Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms". в: *Journal of Parallel and Distributed Computing* 74.10 (июнь 2014), с. 2899—2917. DOI: 10.1016/j.jpdc.2014.06.008. URL: https://inria.hal.science/hal-01017319.

[3] Alexandru Iosup, Georgios Andreadis, Vincent Beek, Matthijs Bijman, Erwin Eyk, Mihai Neacsu, Leon Overweel, Sacheendra Talluri, Laurens Versluis и Maaike Visser. "The OpenDC Vision: Towards Collaborative Datacenter Simulation and Exploration for Everybody". в: июль 2017, с. 85—94. DOI: 10.1109/ISPDC.2017.25. URL: https://atlarge-research.com/pdfs/ccgrid21-opendc-paper.pdf.

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

[5] *Rust std::future library*. URL: https://doc.rust-lang.org/std/future/index.html (дата обр. 31.01.2023).

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

[7] *Описание проекта DSLab*. URL: https://docs.google.com/document/d/1Z8ivtqLRMFG-EfaiRaWBT8dNHFjYjuDhz0bWIYkc-A8/edit (дата обр. 31.01.2023).

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

[9] *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).

[10] *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).

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

[12] *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).

[13] *Пример использования асинхронного кода в фреймворке SimGrid: master-workers*. URL: https://github.com/osukhoroslov/dslab/blob/main/examples-other/simgrid/master-workers/worker.cpp (дата обр. 17.05.2023).

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

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

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

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

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

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