План выполнения курсовой работы по теме «Реализация поддержки асинхронного программирования для фреймворка DSLab»

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 предстоит добавить возможность управлять событиями асинхронно.

1 Описание проекта и постановка задачи

1.1 Устройство 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. 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.

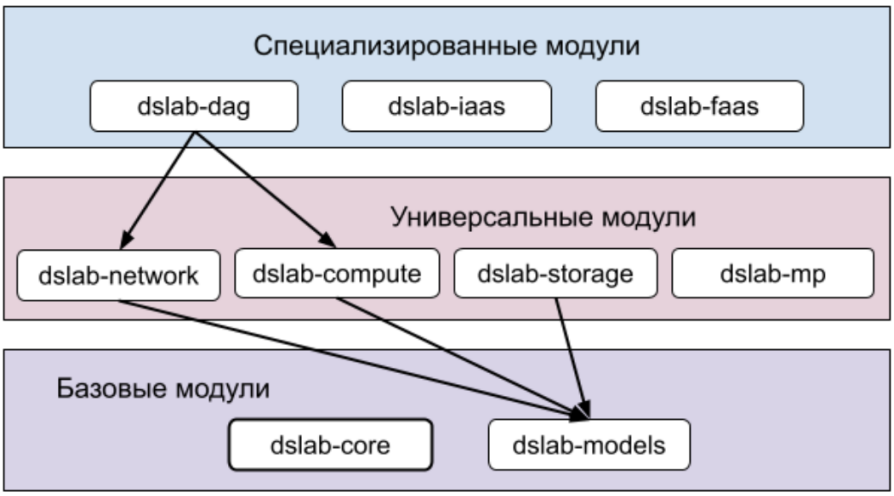

Рис. 1: Архитектура DSLab

Описание архитектуры основано на документации проекта [1]. Более подробно про проект можно прочитать в его описании [2], его требования [3] и сравнение с аналогами [4]. Реализации модулей представлены в репозитории [5].

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

Основным процессом создания симуляции является создание событий и написание процессов реагирования на них. Каждое событие представляет из себя следующую структуру:

1. идентификатор события
2. идентификатор компонента, создавшего событие
3. идентификатор компонента, которому это событие предназначено доставить
4. внутреннее время, когда событие должно наступить
5. произвольные данные события («полезная нагрузка»)

Таким образом в процессе симуляции 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 какого-то события, чтобы продолжить.

1.2 Цель

Добавить возможность пользователю использовать асинхронность при программировании симуляции в DSLab.

1.3 Задачи

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

1.4 Требования

- Код нового ядра должен быть написан на языке Rust и опубликован как модуль проекта DSLab.
- Реализация должна корректно обрабатывать все возможные возникающие комбинации событий в асинхронном контексте. Все возможные случаи должны быть описаны в документации и покрыты тестами.
- Новое ядро должно быть реализовано эффективно: не должны возникать дополнительные накладные расходы по сравнению со стандартным ядром.
- Реализация должна содержать логирование происходящего в асинхронном контексте, а так же обработку ошибок, которая бы помогала пользователю искать ошибки в коде.
- 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.

2 Актуальность и значимость

2.1 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 [2].

2.2 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_all_replicas();
        wait_for_commit_confirmation_from_quorum().await;
        send_ok_message_to_user();
    } else {
        send_reject_message_to_user();
    }
}
```

Рис. 2: 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.

2.3 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.

3 Существующие работы и решения

Подобный подход уже был реализован в других симуляторах, например в SimGrid [6], но этот код сложно назвать легко читаемым, потому что SimGrid является низкоуровневым фреймворком.

Более близким к желаемой реализации является использование корутин языка Kotlin в проекте OpenDC [7]. К сожалению, проект не содержит достаточно разнообразных примеров использования симуляции в асинхронном контексте, но простой пример запуска симуляции в асинхронном контексте [8] 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.

4 Предлагаемые подходы и методы

4.1 Изучение технологий

В качестве знакомства с языком программирования Rust и его концепциями я прошел часть курса ШАД по расту [9]. В нем особенно интересовала глава про асинхронное программирование. У Rust есть официальная

документация по асинхронности.[10]. Она еще не завершена, но основные идеи по написанию собственного рантайма там изложены.

4.2 Написание прототипа

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.

Для этого нужно реализовать альтернативу dslab-core (или дополнить уже сущетсвующую реализацию), и затем уже пользоваться новым интерфейсом на более верхних уровнях. К счастью, язык Rust дает возможность пользователям самим управлять процессом рантайма: с помощью библиотеки futures [11] такое поведение может быть реализовано (что на простом примере и описано в официальной документации [10]).

Изучив предлагаемые там примеры и посмотрев материалы был реализован прототип [12]. Он поддерживает простой обмен сообщениями и ожидания прихода сообщений от других компьютеров в симуляции. Синтаксис, которым там можно оперировать, соответствует представленному превдокоду (Рис. 2). В качестве тестирования прототипа был написан пример реплицированного хранения данных на 3 нодах и процесс обмена разными типами сообщений между ними, чтобы поддерживать консистентное состояние [13].

В примере можно видеть, что управление симуляцией (постановка условий и задач для хранилища) так же осуществляется с помощью асинхронных функций. В этом тоже есть удобство: процесс симуляции пишется как сценарий, который легко читать.

4.3 Перенос прототипа в проект DSLab

Основываясь на полученных результатах в процессе разработки прототипа, нужно встроить эту логику в DSLab. Для этого нужно перевести прототип на использование сущностей из DSLab-a, а также перенести пример использования нового асинхронного ядра и добавить другие примеры, иллюстрирующие данных подход, его преимущества и недостатки в сравнении с callback-based методом.

По возможности хочется добиться совместимости уже написанных компонент с новым ядром, тем самым давая пользователю выбор между асинхронной моделью программирования и программирования на callback-ах. Этот вопрос предстоит исследовать.

5 Ожидаемые результаты

1. Основным результатом проекта будет конкретный программный артефакт, в виде нового или расширенного ядра и API симуляции в DSLab, позволяющий пользователю писать собственные модули и симуляции в асинхронной модели.
2. В репозиторий DSLab [5] добавлены примеры кода, документация по использованию асинхронного ядра.
3. Реализация покрыта тестами.

6 План работ

Стратегия разработки совпадает со стандартным процессом создания программного продукта

№	Описание этапа	Примерный срок выполнения
1	Работа с источниками, изучение Rust	01.11.2022 — 31.01.2023
2	Реализация прототипа асинхронного ядра и примеров его использования	01.12.2022 — 31.01.2023
3	Перенос прототипа асинхронного ядра в DSLab, написание примеров	01.02.2023 — 30.03.2023
4	Исследование возможностей совмещения двух парадигм разработки внутри одного модуля	01.03.2023 — 30.03.2023
5	Тестирование всех функций и написание документации	01.04.2022 — 30.04.2023
6	Написание отчетных документов и подготовка к защите	01.05.2023 — 15.05.2023

Таблица 1: Ожидаемые сроки выполнения

Список литературы

1. Архитектура проекта DSLab. — URL: https://docs.google.com/document/d/12CcGpdulqMJppAYoNrOdpIWZYEa3flXn9JMX5fdyAnY/edit (дата обр. 31.01.2023).
2. Описание проекта DSLab. — URL: https://docs.google.com/document/d/1Z8ivtqLRMFG-EfaiRaWBT8dNHFjYjuDhz0bWIYkc-A8/edit (дата обр. 31.01.2023).
3. Требования к проекту DSLab. — URL: https://docs.google.com/document/d/1CprULnQiVSWTXiBkx90CSEi4tgkUhM3R6VeCp1CsRwo/edit (дата обр. 31.01.2023).
4. Существующие решения и аналоги DSLab. — URL: https://docs.google.com/document/d/1H2711nGiS6m7QTo4pMfmrIBmg4llLKUhQzDiFNIw-UU/edit (дата обр. 31.01.2023).
5. Репозиторий DSLab. — URL: https://github.com/osukhoroslov/dslab (дата обр. 31.01.2023).
6. 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).
7. Репозиторий проекта OpenDC. — URL: https://github.com/atlarge-research/opendc (дата обр. 31.01.2023).
8. 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).
9. Shad Rust course repository. — URL: https://gitlab.manytask.org/rust-ysda/public-2022-fall (дата обр. 31.01.2023).
10. Asynchronous Programming in Rust. — URL: https://rust-lang.github.io/async-book (дата обр. 31.01.2023).
11. Rust crate «futures». — URL: https://crates.io/crates/futures (дата обр. 31.01.2023).
12. Прототип асинхронного dslab-core. — URL: https://gitlab.com/makogon2907/rust-async-await-prototype (дата обр. 31.01.2023).
13. Пример использования прототипа асинхронного ядра в случае распределенного хранилища. — URL: https://gitlab.com/makogon2907/rust-async-await-prototype/-/tree/master/samples/replicas (дата обр. 31.01.2023).