Федеральное государственное автономное
образовательное учреждение высшего образования
«Национальный исследовательский университет
«Высшая школа экономики»

Факультет компьютерных наук
Основная образовательная программа
«Прикладная математика и информатика»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Программный проект на тему «Моделирование облаков и распределённых систем»

Выполнил студент группы БПМИ176, 4 курса,

Кускаров Тагир Фаридович

Руководитель ВКР:

Ведущий инженер ключевых проектов

ООО «Техкомпания Хуавэй»

Тихонов Андрей Александрович

Москва 2021

Содержание

Аннотация

Развитие технологий облачных вычислений создаёт потребность в разработке и тестировании новых алгоритмов оптимального распределения ресурсов внутри облака. Проведение подобной работы на настоящем оборудовании — трудоёмкий и ресурсозатратный процесс, который влияет на пользователей и, к тому же, не даёт полной детерминированности. По этой причине исследователи используют симуляторы — окружения, предоставляющие схожий с реальным интерфейс облака, однако проводящие вычисления на одной машине и адаптированные под разработку и анализ выходных данных. Существующие симуляторы часто сложны и дают разработчикам недостаточно свободы в реализации своих оптимизационных алгоритмов, в связи с чем было разработано новое решение, не содержащее большинства найденных проблем. При работе с новым симулятором исследователи смогут использовать удобное для них окружение, автоматически обрабатывать полученные результаты, а также, при необходимости, легко расширять существующий функционал.

Abstract

Cloud computing is a fast growing sphere. This causes a need for creating and5 testing new algorithms for optimal resource sharing in the cloud. This workflow cannot be done well on a real infrastructure as it requires too much resources been withdrawn from users' needs. Moreover, this process is not fully deterministic. For those reasons researches use cloud computing simulators — a software which behaves as a real cloud but emulates all computing on a single machine. This software also provides convenient ways for producing experiments and their analysis. Currently available simulators are often difficult to use and not frexible enough for testing advanced algorithms and making large experiments. In this work a new simulator is presented which has almost nothing of detected limitations. When using new simulator, researches would be able to run experiments from the environment favorable for them, automatically collect results and easily extend existing code if needed.

Список ключевых слов

облачные вычисления; симуляторы облаков; инфраструктура-как-сервис; алгоритмы распределения виртуальных машин по физическим машинам; оптимизация энергопотребления облачных дата-центров.

1 Введение

1.1 Актуальность

Облачные вычисления стали популярны по многим причинам — из-за выгоды в цене, простоты использования, гарантий надежности и других. Отсюда возникает спрос на развитие технологий в данной области. Многие исследователи занимаются разработкой новых подходов к построению облаков, а именно в части организации инфраструктуры и создания нового программного обеспечения.

Для тестирования новых разработок необходима система, которая обеспечивает воспроизводимость экспериментов, полный контроль за своим состоянием и, желательно, невысокую стоимость. Тестирование на реальных облачных системах не вполне отвечает данным требованиям — даже если речь идет о внутренних разработках компании-провайдера, то обеспечить воспроизводимость в системе, использующей физическую сеть и серверы, полностью невозможно. Для исследователей, не обладающих доступом к ресурсам, задача еще более осложняется — стоимость тестирования при высокой нагрузке или на больших кластерах может оказаться неподъёмной.

Для подобных задач разработаны *симуляторы облачных вычислений*, которые позволяют моделировать работу физического кластера и ПО поверх него и на этой основе запускать новые алгоритмы.

1.2 Постановка задачи

В облачных вычислениях существуют различные модели обслуживания — от IaaS[1], когда предоставляются исключительно вычислительные мощности в виде виртуальных машин, до SaaS[2] — облачного программного обеспечения. Для каждой модели необходимо свое ПО для управления облаком, и, соответственно, свои алгоритмы планирования. Подробнее типы задач для

[1] Infrastructure-as-a-Service — «инфраструктура как сервис»

[2] Software-as-a-Service — «программное обеспечение как сервис»

планировщиков описаны в статье [1]. Наиболее базовой является модель IaaS и планировщики, которые определяют, на какой сервер необходимо аллоцировать[3] вновь запрошенную виртуальную машину. Именно симуляторы такого уровня были рассмотрены в данной работе.

Опишем подробнее, какие требования предъявляются к подобному фреймворку. Неформально, он должен *достаточно точно* моделировать условия реальной системы облачных дата-центров. В общем случае такое моделирование состоит из следующих подзадач:

1. Моделирование «окружающего мира»: представления времени, происходящих событий и объектов, которые способны данные события создавать и обрабатывать, способ коммуникации данных объектов между собой;

2. Представление физических сущностей и компонентов облака (например, сервера и его ресурсов), их иерархии и их жизненных циклов;

3. Описание потребителей ресурсов (для симуляторов IaaS единственным типом потребителя является виртуальная машина) и их связка с сущностями этих ресурсов (модель потребления);

4. Наконец, как уже было упомянуто в аннотации, симулятор должен обладать понятным API[4], мощной системой мониторинга и анализа результатов для удобства проведения и воспроизведения экспериментов.

В дополнение к этим пунктам во многих симуляторах реализован функционал *измерения энергопотребления* [1, 2]. По оценкам на 2011 год [3] облачные дата-центры потребляют порядка 1.5% от всей используемой в мире электроэнергии. Учитывая рост популярности облачных технологий, сейчас эта цифра выше. В связи с этим количество потраченной энергии часто включается в оптимизируемый планировщиком функционал, разработано множество

[3] выделить на сервере мощности под данную ВМ, загрузить на него её образ и запустить

[4] application programming interface

соответствующих алгоритмов [4, 5]. В описываемом симуляторе возможность измерять энергию также присутствует.

1.3 Критерии корректности работы симулятора

Качество реализации симулятора определяется двумя критериями. Во-первых, упрощение реальности, неизбежное при симуляции, не должно приводить к получению качественно других результатов. То есть, выводы, полученные с помощью симулятора, должны быть объяснимыми, согласовываться с реальной жизнью и поведением известных аналогов.

Во-вторых, симулятор должен быть *достаточно удобным*. Несмотря на субъективность данного критерия, можно выделить ряд свойств, которые делают его использование более удобным:

- Поддержка принципа Infrastructure-as-Code (IaC), то есть возможность задания конфигурации облака в файлах специального вида;
- Гибкая система сбора и анализа информации. Данный пункт включает как систему логирования с поддержкой сериализации данных в текстовом виде для последующего анализа (в том числе автоматического), так и подробный вывод самого симулятора, позволяющий представлять полную картину происходящего в симулируемом облаке;
- Высокое качество кода симулятора, в том числе его логичное разбиение на «модули», наличие разумных абстракций, а также отделение слоя пользовательского кода от основной кодовой базы симулятора, чтобы сборка после изменения кода занимала меньшее время;
- Наличие документации и примеров использования.

Как будет показано в обзоре аналогов, существующие симуляторы не вполне удовлетворяют данным пунктам. Они сложны, требуют использования определенного языка программирования и не всегда обладают удачным интерфейсом.

1.4 Полученные результаты

В результате проведённой работы был разработан новый симулятор[5], который соответствует описанным выше требованиям, прост в использовании и при этом корректно эмулирует поведение реальной облачной системы.

В частности, дополнительной возможностью, отличающей описанный в данной работе симулятор, является наличие RPC[6]-интерфейса, то есть API, к которому можно подключаться из runtime[7] различных языков программирования. Это дает дополнительные возможности для исследователей — они могут создавать более гибкие тесты для алгоритмов.

2 Обзор существующих решений

В этом разделе проведён обзор основных существующих решений — симуляторов, которые позволяют тестировать алгоритмы планирования выполнения задач. В частности, будет описана система CloudSim [6], проведен краткий обзор различных надстроек над ней, а также описаны фреймворки DISSECT-CF[1] и SCORE [2], идеи из которых частично вошли в реализованный в рамках данной ВКР симулятор. Помимо статей от авторов данных продуктов, при анализе источников было использованы обзорные стат о сравнении симуляторов [7, 8].

2.1 CloudSim

Данный симулятор впервые был описан в 2010 году и является, по впечатлению автора, наиболее известным [6]. Статья о нём имеет более 2000 цитирований. На базе CloudSim создано более 10 других систем, которые так или иначе дополняют его функционал.

[5] https://github.com/kuskarov/cloud-modeling

[6] remote procedure call — удалённый вызов процедур

[7] среда исполнения программы

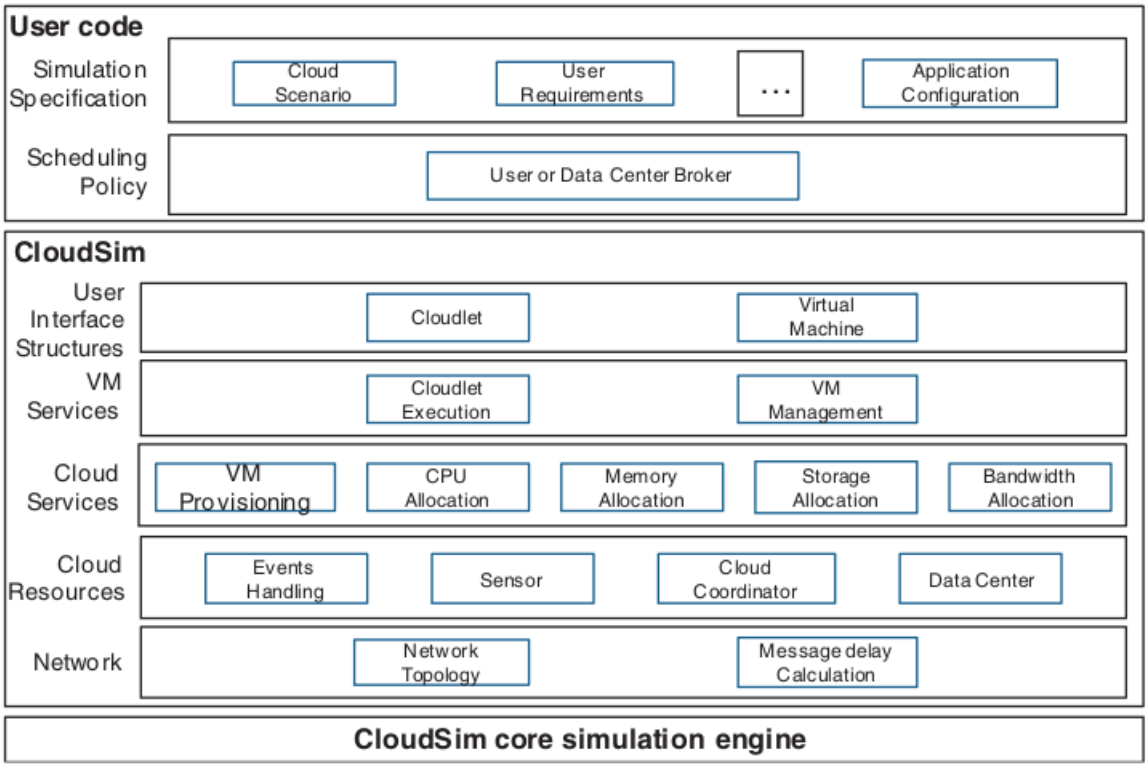

Рис. 2.1: Архитектура CloudSim

Система имеет многоуровневую архитектуру (см. рис. 2.1). В основе всего лежит *core simulation engine* — модуль, который отвечает за саму симуляцию. В CloudSim основным примитивом является событие — ими обмениваются дата-центры, серверы и виртуальные машины. Simulation engine хранит и обрабатывает очереди событий, генерируемых другими компонентами.

Над simulation engine расположен модуль с основной логикой CloudSim — классы, описывающие физические объекты (дата-центры, серверы, ВМ), сеть и правила их взаимодействия.

Над внутренними классами расположен слой API — средства задания конфигурации для приложений и прочие настройки. Кроме того, в API входит и возможность переопределять политики, например, принцип размещения виртуальных машин по серверам или распределение ресурсов на одном сервере по размещенным на нем виртуальным машинам. Переопределение реализовано с помощью Java ООП.

CloudSim предоставляет возможность динамически менять конфигурацию облака — добавлять и удалять серверы, реализуя таким образом эму-

ляцию отказов. Кроме того, присутствует важный функционал по автоматическому измерению энергопотребления оборудования на основе заранее заданных параметров.

Недостатками данного симулятора являются отсутствие сетевой топологии — сеть моделируется с помощью *latency matrix* (матрицы, где для каждой пары объектов указана задержка сети), а также единственный экземпляр планировщика. Это ограничивает масштабируемость, так как планирование является математически сложной задачей, и решение её одним процессом может быть неприемлемо долгим на больших кластерах.

С точки зрения API и удобства, части пользователей данный симулятор может не подойти из-за отсутствия графического интерфейса и необходимости использования Java.

2.2 CloudSim-based

В последующие годы исследователями было предпринято много попыток добавить в CloudSim некий функционал, который бы исправлял часть его недостатков [7].

Например, в NetworkCloudSim [9] полностью моделируется сеть, что важно при сравнении алгоритмов, использующих миграции (перемещение образов ВМ с сервера в хранилище занимает время, которым нельзя пренебречь). В симуляторе CloudReports [10] добавлен графический интерфейс, богатая система сбора логов и аналитики. В работе ElasticSim [11] моделируется переменное потребление ресурсов виртуальной машиной, что также присутствует в описываемом симуляторе.

Тем не менее, любые надстройки над и без того объемным фреймворком CloudSim делают результирующий продукт ещё более тяжеловесным (см. пример архитектуры на рис. 2.2). Кроме того, сохраняется зависимость от Java.

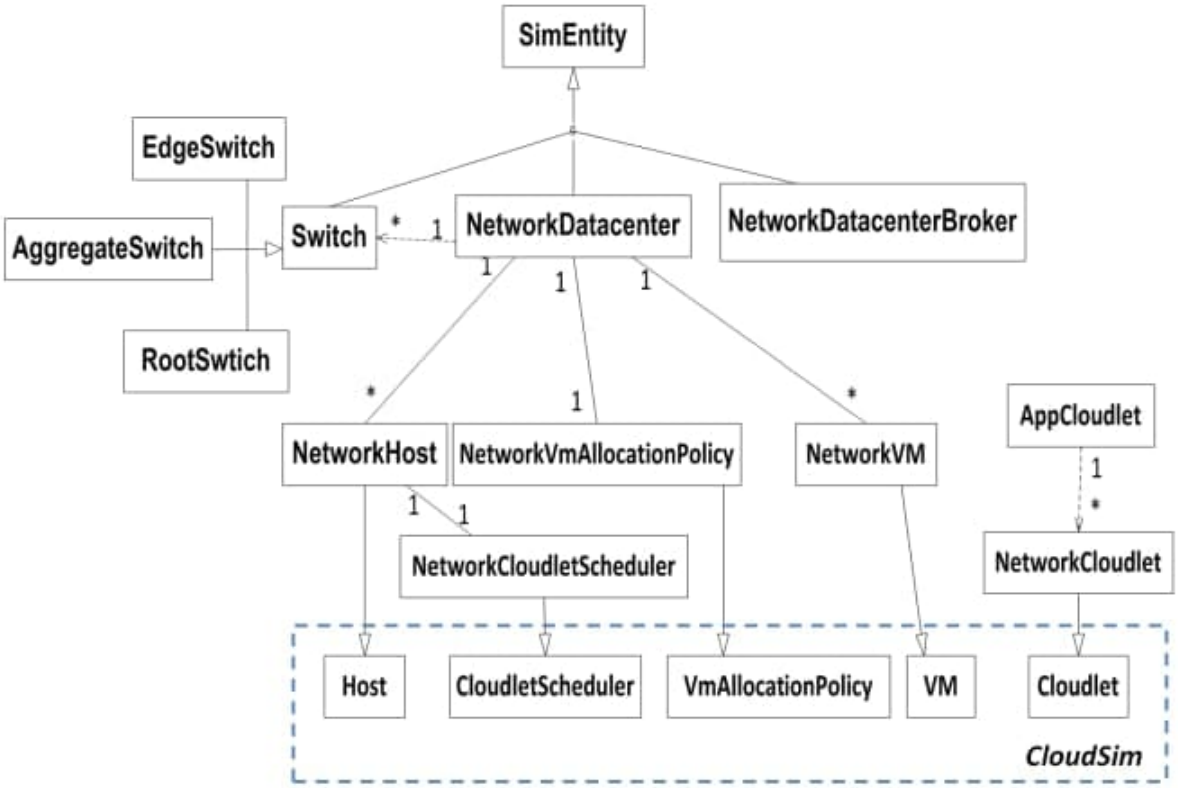

Рис. 2.2: NetworkCloudSim как надстройка над CloudSim

2.3 DISSECT-CF

Статья о данном фреймворке отличается подробностью и глубоким описанием исследуемой области. Название расшифровывается как «DIScrete event baSed Energy Consumption simulaTor for Clouds and Federations». В данном симуляторе реализована абстрактная модель распределения и потребления ресурсов (unified resource sharing model), которая позволяет также измерять энергопотребление.

В отличие от CloudSim, здесь присутствует определённая симуляция сети в виде объектов `NetworkNode`. Архитектурно (см. рис. 2.3) данное ПО похоже на аналоги — и на реализованный в рамках данной ВКР симулятор, что будет видно в дальнейшем.

Данный симулятор написан на Java и, как и предыдущие решения, требует от разработчиков написания алгоритмов исключительно на данном языке. Кроме того, наличие сильных абстракций требует вдумчивого прочтения статьи перед началом использования.

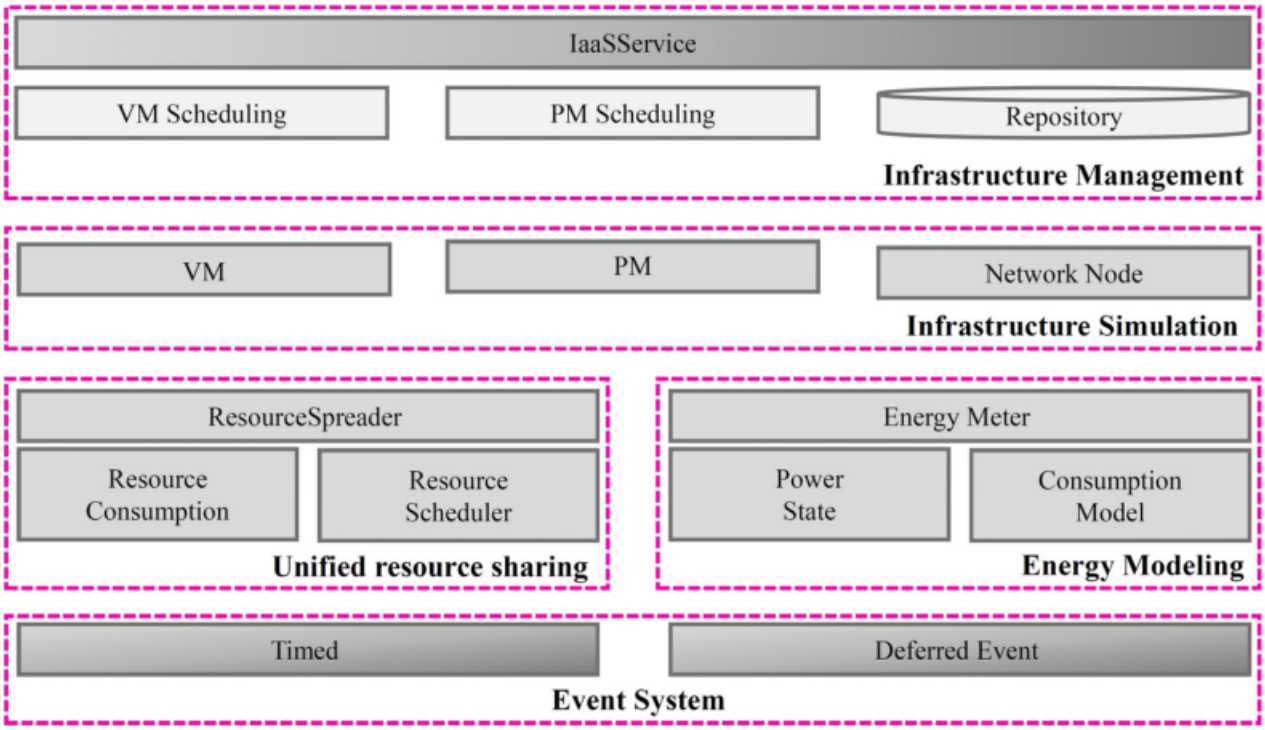

Рис. 2.3: Архитектура DISSECT-CF

2.4 SCORE

Симулятор SCORE [2] также не является надстройкой над CloudSim. Его авторы задавались целью создать систему для сравнения различных стратегий построения расписания задач на очень больших кластерах, что недоступно с CloudSim. Кроме того, данный симулятор содержит функционал для измерения энергопотребления.

Существует несколько подходов к тому, как нужно проектировать модули для составления расписания задач в облаке. В простейшей модели есть единый сервис, который статически или динамически генерирует время и место запуска для каждой задачи, однако это решение не является масштабируемым.

В двухслойной модели есть модуль, который выделяет ресурсы блоками и передает управления малым узлам, которые делят выделенные им ресурсы между задачами. Существуют и более сложные распределенные системы планировщиков, которые также могут быть исследованы с помощью SCORE.

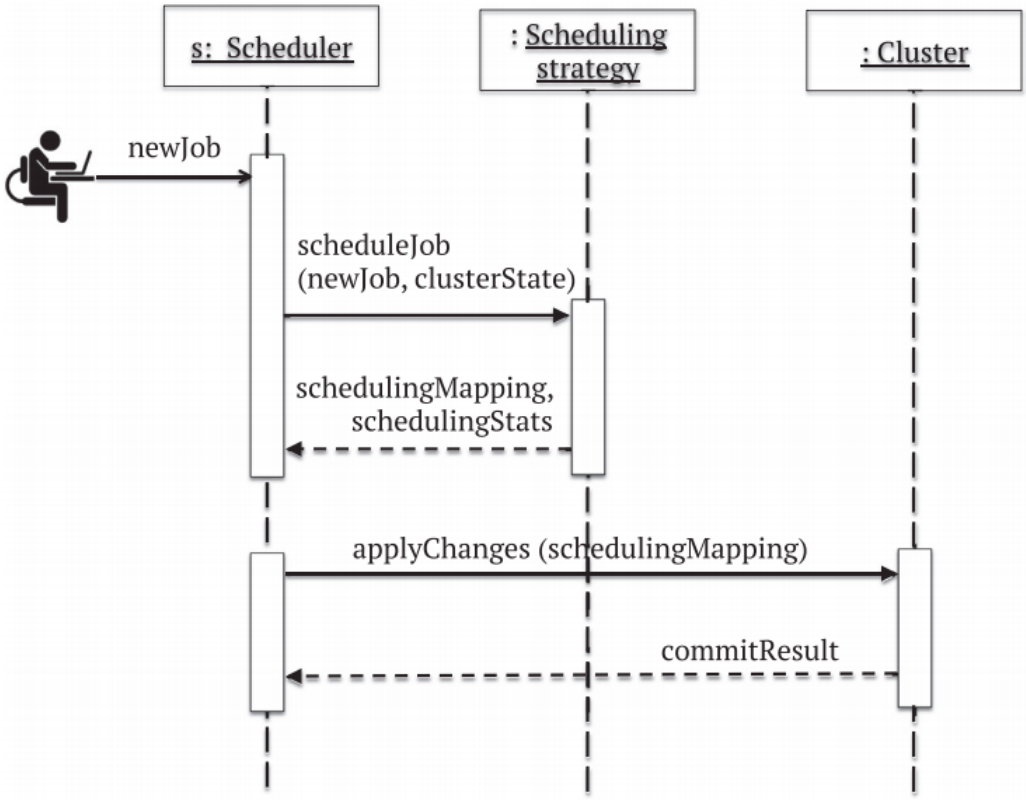

Рис. 2.4: Процесс планирования в SCORE

SCORE основан на легковесном симуляторе Google Omega, который сам по себе не адаптирован для облачной инфраструктуры. Поверх него на языке Scala реализованы классы непосредственно SCORE, отвечающие за стратегии расписания, а также для измерения потребления электроэнергии — то, чего нет в Omega.

В данном симуляторе отдельно исследуются модели, оптимизирующие энергопотребление, например, предлагающие отключать простаивающие (idle) серверы. Вообще говоря, такой подход сильно снижает производительность, так как отключенный сервер не способен быстро взять на исполнение новую задачу, однако оптимальное составление расписания позволяет это оптимизировать.

Дополнительно, в реализованном автором симулятором реализован процесс планирования, похожий на SCORE (см. схему на рис. 2.4) — применение изменений к облачному кластеру через последовательность команд.

2.5 Выводы

Всем изученным симуляторам характерна привязка к конкретному языку программирования — любые расширения и имплементации алгоритмов планирования нужно реализовывать на фиксированном ЯП, как правило, Java или Scala.

Подходы, описанные в разобранных статьях, по субъективному мнению автора, часто довольно сложны и требуют изучения статьи перед началом использования. Многие репозитории с кодовой базой описанных проектов долгое время не обновляются и содержат недостаточно документации и примеров.

Следовательно, имеется необходимость в создании продукта, который бы совмещал лаконичность кодовой базы, гибкость абстракций и низкий порог входа для начала использования.

3 Описание разработанного симулятора

3.1 Общая архитектура

В данном разделе будет описано, из каких частей состоит симулятор и какова их иерархия. Как уже было отмечено в обзоре аналогов, симуляторам характерна многослойная архитектура, где разные компоненты, насколько это возможно, отделены друг от друга. Такой подход позволяет гибко расширять функционал, а также оптимизировать внутреннюю реализацию отдельных частей без необходимости масштабного изменения кодовой базы.

При проектировании данного симулятора было решено действовать аналогично. В проекте можно выделить 4 основных слоя, не считая вспомогательных функций, которые на момент написания данного отчёта являются полностью `header-only` и могут напрямую включаться в нужные места основного кода.

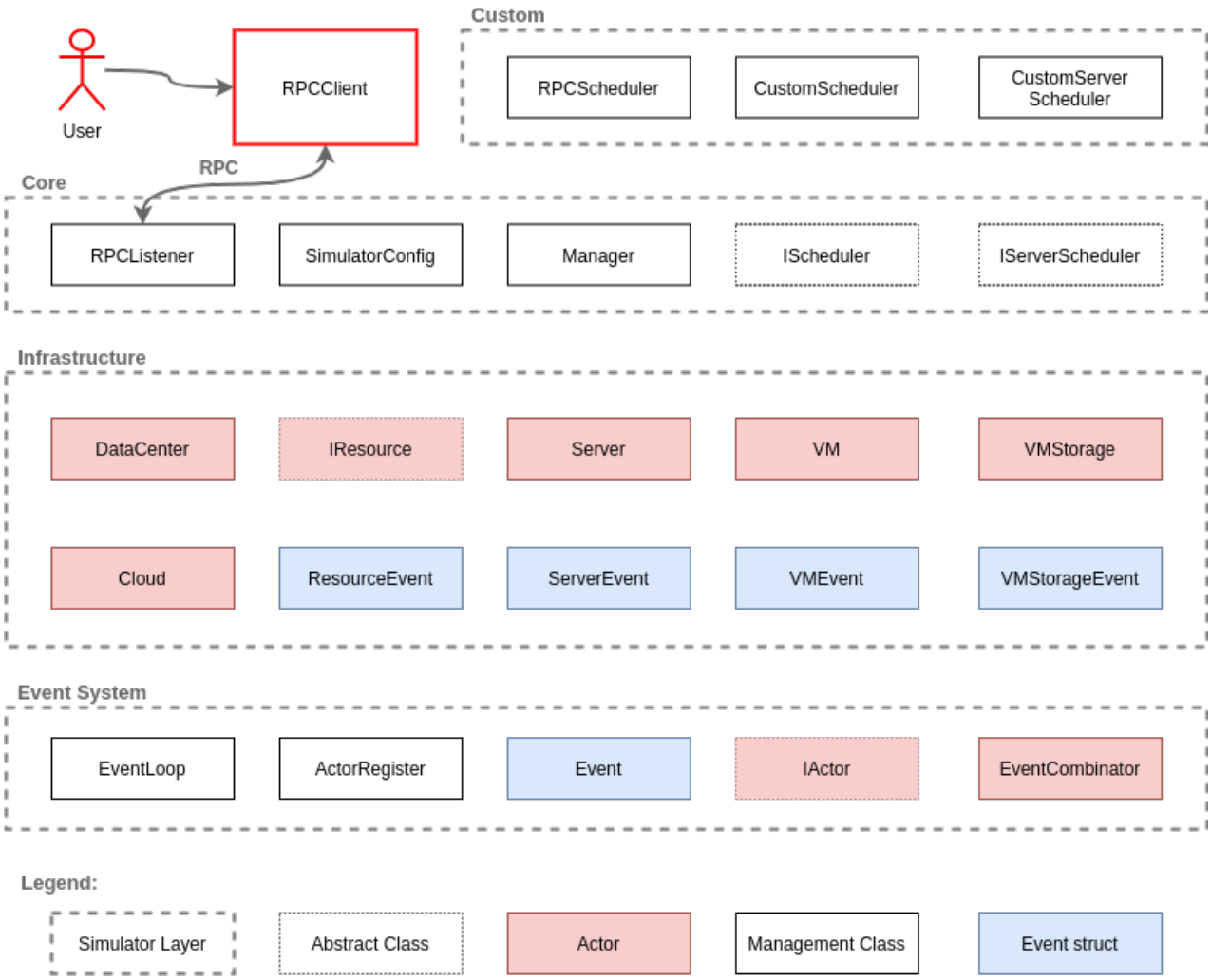

Рис. 3.1: Архитектура симулятора

Иерархия слоёв представлена на рис. 3.1. В основе лежит модуль для создания, хранения и вызова событий (`Event System`), в котором каждому событию приписано определенное дискретное время. Непосредственно использует систему событий следующий слой — `Infrastructure`. Как следует из названия, в нём реализованы представления для деталей облачной инфраструктуры. Они построены на общей для всех абстракции *ресурса*, которая будет подробнее описана в соответствующем разделе далее. Кроме того, там же описан автомат состояний виртуальной машины. Далее, поверх инфраструктуры расположен слой `Core` — классы, управляющие симуляцией и API симулятора, а именно парсер конфигураций, интерфейс командной строки и абстрактный класс `IScheduler` для реализации политик планирования задач в облаке.

Отдельным слоем `Custom` поверх `Core` вынесён пользовательский код — переопределенные политики планирования ВМ на кластере и планировщики виртуальных машин на одном сервере. В текущей версии в качестве примера реализован простейший планировщик, жадным образом расставляющий виртуальные машины по свободным серверам.

3.2 Система событий с дискретным временем

В основе симулятора лежит средство, которое позволяет эмулировать проведение ресурсоёмкой работы, занимающей время, без её реального запуска. Для этого была написана простая модель событий с дискретным «временем». Под термином «дискретное» понимается отсутствие привязки временных отметок к каким-либо физическим единицам и отсутствие в симуляторе системных вызовов, использующих счётчик времени, встроенный в компьютер. Данный выбор был сделан по нескольким причинам [12]:

1. **Универсальность:** время в симуляторе измеряется в условных единицах, которые можно при надобности ассоциировать с любой физической величиной;

2. **Детерминированность:** симулятор не зависит от погрешностей в работе компьютерного счётчика времени;

3. **Сжатое представление очереди задач:** события хранятся плотно, даже если в расписании имеются большие промежутки между соседними временными слотами;

4. **Удобство использования:** возможность пошаговой симуляции с остановками в необходимые пользователю моменты.

Реализация системы событий в данном симуляторе состоит из структуры `Event`, абстрактного класса актора `IActor` и класса `EventLoop`, исполняющего симуляции. Кроме того, реализован служебный класс `ActorRegister`,

который владеет всеми созданными экземплярами `IActor` и предоставляет к ним доступ по `UUID`. Опишем их подробнее.

Структура `Event` содержит поля `TimeStamp happen_time` — момент времени, когда событие должно быть отправлено, и `UUID addressee` — его адресат. Возможно наследование от данной базовой структуры для добавления контекста в событие. Кроме того, в `Event` имеется поле `Event* notificator` — указатель на событие, которое нужно отправить в `EventLoop` после завершения обработки данного. Такой функционал бывает нужен, если добавивший событие актор хочет получить уведомление о его успешной обработке.

Абстрактный класс `IActor` является базовым для всех классов, которые могут получать события. Он требует реализации метода `HandleEvent(const Event* event)`, который будет вызван `EventLoop`-ом при получении события с указателем на экземпляр данного класса. Внутри этого метода, как правило, производится попытка преобразования указателя к ожидаемому типу-наследнику структуры `Event` и затем обработка полученного события.

Наконец, класс `EventLoop` хранит внутри себя очередь событий и реализует их последовательное исполнение, то есть получает по `event->addressee` указатель на актора и вызывает его метод `HandleEvent`, передавая в него запланированное событие. для каждого события `event` из очереди. При этом предоставляется способ как для планирования события на определённое время, так и для его «немедленной» обработки — вставки в очередь непосредственно после обрабатываемого в данный момент.

В текущей реализации события хранятся в виде словаря на сбалансированном дереве, где в качестве ключа выступает временная отметка, а в виде значения — дек[8] из элементов типа `Event`. Соответственно, добавить или извлечь событие из очереди можно за $O(\log \mathcal{T})$, где $\mathcal{T}$ — число различных временных отметок, на которые запланировано хотя бы одно событие в данный момент.

[8] от `deque` — двусторонняя очередь, поддерживает вставку и удаление из конца и начала за $O(1)$

Класс `EventLoop` позволяет запускать обработку всех событий до момента опустошения очереди, так и симулировать определенное число шагов. Класс `ActorRegister` предоставляет шаблонный метод создания актора определённого типа. Далее в рамках облака к актору можно обращаться, используя его `UUID`. «Разыменование» актора доступно лишь управляющим классам симулятора, между собой акторы взаимодействуют исключительно с помощью системы событий.

Функция для планирования события `schedule_event()` передается акторам при их создании в виде замыкания, так как она обращается к экземпляру класса `EventLoop`, непосредственный доступ к которому акторы не должны иметь для сохранения абстракции.

3.3 Инфраструктура облака

3.3.1 Ресурсы

В симуляторе присутствуют следующие элементы инфраструктуры: облако (`Cloud`), дата-центр (`DataCenter`), сервер (`Server`) и абстрактное хранилище виртуальных машин (`VMStorage`). Первые три класса являются наследниками абстрактного класса `IResource`. Данный класс реализует автомат состояний ресурса — `off`, `turning_on`, `running`, `turning_off` и `failure`. Кроме того, имеется тип события `ResourceEvent`, который используется для переключения между этими состояниями. Автомат приведен на рис. 3.2.

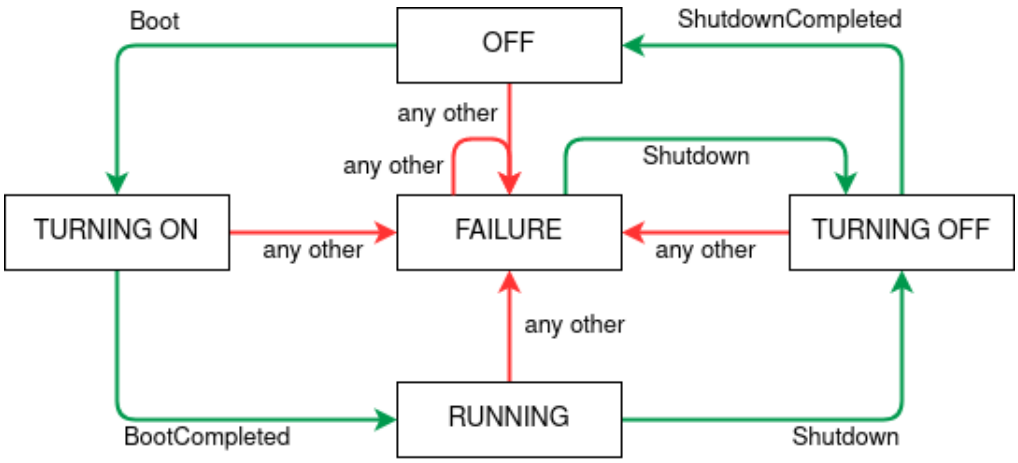

Рис. 3.2: Автомат состояний ресурса

Каждый ресурс помимо автомата состояний содержит ссылки на компоненты — ресурсы, которые находятся внутри него, как, например, серверы внутри дата-центра. Кроме того, у ресурса можно вычислить потраченную им энергию в абстрактных единицах `EnergyCount` с помощью метода `SpentPower()`. По умолчанию, вычисление производится рекурсивно: энергопотребление ресурса складывается из суммы потреблений его компонентов и некой дополнительной величины, вычисление которой разное у разных типов ресурсов.

Хранилище виртуальных машин необходимо, чтобы объекты ВМ переживали остановку — то есть были сохранены в облаке, но не аллоцированы на каком-либо физическом сервере. В текущей реализации оно не является ресурсом, то есть имеет неограниченную ёмкость, поэтому его стоит считать абстрактным.

3.3.2 Потребители — виртуальные машины

В симуляторе IaaS единственным типом потребителя ресурсов является виртуальная машина. Она обладает своей спецификацией нагрузки, которая хранится в структуре `VMWorkLoadSpec`. В данной структуре описано, как со временем меняются необходимые виртуальной машине память, CPU и другие характеристики. Поддерживаются как постоянные нагрузки, так и переменные, использующие распространенные распределения.

Так как виртуальная машина не является ресурсом, класс `VM` не наследует `IResource`. В нём реализован свой, более подробный автомат состояний [13], который учитывает возможную остановку без удаления и процесс ожидания размещения ВМ на каком-либо сервере.

3.3.3 Модель потребления и измерения ресурсов

Для принятия решения о выборе сервера для размещения виртуальной машины планировщику необходимо иметь доступ к текущему состоянию об-

лака. Помимо статических характеристик (конфигурации дата-центров, параметров серверов) состояние включает уровень загруженности.

Загруженность ресурса зависит от потребностей использующих его виртуальных машин. В простейшем случае потребности виртуальной машины не меняются со временем, и поэтому загруженность может меняться только в результате команд пользователя. Продвинутые же техники планирования учитывают историю реального потребления ресурсов виртуальной машиной, регулируя их распределение для оптимального расхода.

3.3.4 Эмуляция сети

На данный момент сеть в симуляторе не моделируется. Однако её добавление не потребует радикальных изменений в существующем коде. Опишем шаги, которые потребуется выполнить, чтобы симулятор поддерживал топологию сети:

- Топология будет задаваться в конфигурационных файлах как список маршрутизаторов и список ребёр графа сети;
- Все запросы к серверу будут проходить через цепочку маршрутизаторов. Каждый маршрутизатор при получении события будет исходя из своих правил определять, куда дальше пересылать запрос. Непосредственно команда серверу будет храниться в поле `notificator` события (см. пункт 3.2);
- Маршрутизатор будет являться ресурсом, потреблять энергию, иметь ограниченную пропускную способность и являться частью дата-центра.

3.4 API

Как уже было сказано ранее, в рамках облака части его инфраструктуры (акторы) взаимодействуют между собой исключительно с помощью системы

событий. Акторы не имеют доступа к объектам друг друга, для именования используются `UUID`.

В отличие от акторов, управляющие классы симулятора имеют доступ к объектам. Для получения указателя на чтение по `UUID` используется специальный шаблонный метод `ActorRegister::GetActor()`.

3.4.1 Конфигурация

Конфигурация облака задается в конфигурационных файлах формата `YAML`, удобного как для восприятия человеком, так и для чтения из кода. Файл `specs.yaml` содержит список доступных моделей серверов с их характеристиками и уникальными именами. На основе данного списка пользователь создает конфигурацию в файле `cloud.yaml` — собирает облако из дата-центров, а дата-центры — из серверов, указывая список пар «модель сервера — количество».

3.4.2 Логирование

Важнейшей частью симулятора является система логирования — необходимо обеспечить возможность как визуального, так и автоматического анализа результатов работы сценариев. В связи с этим на данный момент поддерживаются два выходных потока логов — в консоль симулятора в виде таблицы и в файл формата `csv` для удобного анализа средствами популярных программ. В обоих случаях логируется текущее дискретное время, тип и имя актора, записавшего лог, уровень важности лога и непосредственно сообщение. Кроме того, все логи отправляются в виде потока сообщений типа `LogMessage` вызывающей стороне через RPC, про который будет рассказано ниже.

3.4.3 RPC и интерфейс командной строки

Основное API симулятора реализовано в виде RPC с помощью фреймворка `gRPC`. Данный фреймворк реализован для множества языков программирования [14]. С помощью специального генератора `protoc` на основе файла с протоколом (описанием сообщений и RPC-методов) генерируются базовые классы для сервера и клиента, которые дополняются требуемой логикой. Это позволяет обращаться к симулятору из программного кода любого языка программирования, поддерживаемого `gRPC`.

В качестве примера клиентского приложения реализован интерфейс командной строки в отдельном процессе, который использует вышеупомянутый RPC-протокол. Данный интерфейс поддерживает все операции, доступные в симуляторе (включение и отключение ресурсов, управление виртуальными машинами и т.д.) и выводит логи на экран.

3.4.4 Реализация политик планирования

Предназначение симулятора — тестирование новых политик планирования запуска виртуальных машин на физических. Как уже было описано выше, планировщику нужен доступ к полному состоянию облака — конфигурации и загруженности кластера. Это реализовано с помощью интерфейса `IScheduler`, в котором имеется доступ на чтение к экземпляру класса `Cloud`. Через данный класс можно получить доступ ко всем ресурсам облака и виртуальным машинам, ожидающим размещения.

Любой наследник данного класса должен реализовывать метод `UpdateSchedule()`. В ходе своей работы планировщик добавляет в расписание (`EventLoop`) события, связанные с изменением состояния кластера (перемещение ВМ, включение и отключение серверов и т.д.).

Кроме того, есть возможность переопределения политики разделения ресурса физического сервера между аллоцированными на нём виртуальными машинами. В данном симуляторе такие планировщики реализуются как на-

следники абстрактного класса `IServerScheduler`. Им доступны на чтение состояния сервера и аллоцированных на нём ВМ, их спецификации и текущее требуемое число ресурсов.

3.4.5 RPC-планировщики

Одним из наследников класса `IScheduler` является `RPCScheduler` — планировщик, который для принятия решения делает вызов удалённой процедуры, дожидается ответа в виде последовательности сообщений и затем добавляет их в систему событий. Такой способ очень универсален, однако у удалённого планировщика по умолчанию нет доступа к состоянию облака во внутренних C++-классах. В текущей реализации при вызове удалённого планировщика производится сериализация полного состояния облака и отправка его по сети. Это является приемлемым вариантом для небольших кластеров, однако дамп состояния облака, в котором тысячи серверов, может серьёзно сказаться на производительности симулятора.

Одним из вариантов решения такой проблемы может являться отсылка по сети не полного состояния облака, а лога его изменений. То есть, на стороне удалённого планировщика должно быть своё представление облака, к которому применяется та же последовательность команд, что и к облаку в симуляторе. Стоит отметить, что такое представление может быть организовано проще, чем было описано выше и может быть достигнуто средствами стандартных контейнеров используемого языка программирования.

Использование удалённого планировщика серьёзно расширяет возможности симулятора. В обзоре аналогов было отмечено, что монолитный планировщик не всегда способен быстро давать ответ на кластерах больших размеров, поэтому в симуляторе SCORE авторы реализовали двухступенчатую модель, когда отдельные «зоны» облака обслуживаются разные планировщиками. Предложенный же в данном симуляторе метод позволяет реализовать произвольную модель распределения ресурсов, например, использующую модели машинного и глубинного обучения.

3.5 Технологии

Для реализации был выбран язык C++. Он предоставляет широкие возможности для наследования классов, использования полиморфизма, а также эффективной работы с памятью. Каждый упомянутый ранее «слой» выглядит как статическая библиотека и впоследствии включается сборщиком в общий бинарный файл вместе со своими зависимостями. В исполняемом файле также присутствует точка входа в программу `main()`, реализованная отдельно от слоёв. Сборка и управление зависимостями осуществляются с помощью системы `CMake`. Исходный код размещён в репозитории на хостинге `GitHub`, при каждом обновлении репозитория автоматически производится валидация форматирования кода с помощью утилиты `clang-format`, а также проверка собираемости.

При реализации помимо стандартной библиотеки C++ были использованы следующие библиотеки с открытым исходным кодом: `fmtlib` для удобного форматирования строк при логировании, `argparse` для парсинга аргументов командной строки, `yaml-cpp` для парсинга[9] конфигурационных `YAML`-файлов, клиент командной строки `replxx`, а также `gRPC` и генератор кода `protoc` для реализации RPC.

3.6 Сценарии использования

Перечислим ожидаемые автором сценарии использования данного симулятора, а также их ограничения. Как было описано ранее, у симулятора есть 3 источника получения пользовательской логики:

1. Команды пользователя (см. пункт 3.4.3). Принимаются исключительно по RPC. Вызов по RPC возможен либо через реализованный клиент с интерфейсом командной строки, либо через собственный RPC-клиент, который можно создать на любом ЯП, поддерживаемом `gRPC`;

[9] парсинг — синтаксический разбор строки и конвертация её в некоторую структуру

2. Переопределённые политики планирования ВМ в облаке (см. пункт 3.4.4). Возможны два варианта — либо реализация политики на C++ как класса-наследника `IScheduler`, либо использование RPC-планировщика (см. пункт 3.4.5). Во втором случае необходимо на стороне удалённого планировщика реализовать логику RPC-сервера в соответствии с протоколом, который приведён в документации. Базовые и служебные классы для сервера генерируются автоматически с помощью `protoc`;

3. Переопределённые модели потребления ресурсов виртуальной машины (`VMWorkLoadSpec`, см. пункт 3.3.2). По умолчанию доступно два типа моделей — постоянная, а также случайная нагрузка в рамках заданных ограничений с разными видами распределений. В текущей версии создавать собственные `VMWorkLoadSpec` можно только на C++ в рамках слоя `Custom`. При этом возможности исследователя не ограничены — допустимо моделирование любого характера нагрузки.

Таким образом, исследователь, работающий с симулятором, может комбинировать различные способы работы, сочетая удобство RPC-интерфейса и гибкость, которую даёт доступ к внутренним структурам симулятора в рамках C++ runtime.

4 Тестирование и валидация

4.1 Производительность

Скорость работы симулятора имеет большое значение, так как для получения состоятельных выводов при сравнении алгоритмов планирования необходимо запускать их на больших экспериментах, что в случае симулирования модели IaaS может означать:

- Использование конфигураций с большим числом физических машин;

- Запуск большого числа задач (в модели IaaS задачи связаны с жизненным циклом виртуальных машин — их создание, перемещение, удаление);

- Использование сложных моделей потребления ресурсов (в терминах из прошлого раздела — `VMWorkLoadSpec`), которые требуют вычислений при каждом вызове из `ServerScheduler`, например, вычисляющие необходимый объем ресурсов по формуле (предполагается, что данная функция имеет временную сложность $O(1)$ по всем её аргументам).

4.1.1 Сравнение с аналогами

Сравним, сколько физического времени занимает работа описанного в данной работе симулятора и его аналогов CloudSim и DISSECT-CF. Стоит отметить, что описанный ниже эксперимент не может считаться в полной мере репрезентативным по двум причинам: во-первых, имеются неустранимые отличия в особенностях runtime языков Java и C++; во-вторых, интерфейсы симуляторов отличаются и предоставляют разные опции для запуска.

Использованные в эксперименте конфигурации симуляторов приведены в таблице 4.1. Были выбраны простейшие виды планировщиком, чтобы минимизировать влияние их сложности на время работы. Эксперимент состоял в выполнении симуляции полного жизненного цикла N виртуальных машин для различных N в дата-центре из 1000 серверов.

Результаты сравнения представлены в таблице 4.2. Они позволяют су-

Таблица 4.1: Конфигурации симуляторов

Симулятор	Планировщик ВМ	Модель потребления ресурсов
CloudSim	`VmAllocationPolicySimple`	`UtilizationModelFull`
DISSECT-CF	`FirstFitScheduler`	`ConstantConstraints`
Новый	`FirstAvailableScheduler`	`VMConstantWorkLoad`

Таблица 4.2: Время, затраченное на симуляции (мс)

	N, число виртуальных машин				
	1000	2000	5000	10000	20000
CloudSim	706	1082	5003	20653	43673
DISSECT-CF	179	329	789	1101	2037
Новый	3179	6956	16945	69324	159883

дить о том, что реализованный симулятор уступает по производительности DISSECT-CF, но при этом отрабатывает примерно в 3 раза медленнее, чем CloudSim, для большинства N, что можно назвать неплохим результатом, учитывая отсутствие оптимизаций и специальных структур в данных в новом симуляторе.

4.1.2 Влияние RPC

Рассмотрим, как использование RPC влияет на производительность симулятора. Так как RPC предполагает сериализацию и десериализацию данных, разумно было бы предположить, что произойдёт определённое замедление работы. Заметим, что ожидаемым сценарием использования является запуск симулятора и клиента к нему на одной физической машине, поэтому эксперименты не учитывают возможные при иных сценариях сетевые задержки.

Таблица 4.3: Время работы в различных режимах (мс)

	N, число виртуальных машин				
	1000	2000	5000	10000	20000
No RPC	2497	5188	13351	56973	136008
RPC Client, Std Scheduler	3179	6956	16945	69324	159883
RPC Client, RPC Scheduler	3751	7843	18981	72064	164279

В таблице 4.3 представлены результаты измерения времени, затраченного симулятором в трёх конфигурациях:

1. `No RPC`: Модифицированная версия симулятора, в которой RPC-интерфейс заменён на вызовы функций в рамках одного исполняемого файла.

2. `RPC Client, Std Scheduler`: Стандартная версия с RPC-интерфейсом и стандартный клиент, реализованный на C++, при этом планировщик определен в `Custom`-слое (конфигурация из прошлого пункта);

3. `RPC Client, RPC Scheduler`: То же, что выше, но используется RPC-планировщик на Python.

Порядок проведения экспериментов такой же, как и предыдущем пункте. Из полученных результатов можно сделать вывод о том, что использование RPC оказывает влияние на производительность, однако замедление не является серьёзным. При использовании RPC-планировщика время работы замедляется также из-за того, что производится дамп состояния облака и его отправка по RPC. Возможные способы ускорения данной конфигурации были описаны в разделе 3.4.5.

4.2 Функциональность

В данной секции проведём сводный анализ функционала, реализованного в данном симуляторе и двух его аналогах, рассмотренных ранее в обзоре, а именно CloudSim и DISSECT-CF.

Анализ представлен в таблице 4.4. В нём перечислены основные свойства, наличие которых требуется от симулятора облака, а также дополнительные возможности, реализованные в новом продукте. Из таблицы можно сделать вывод о том, что он поддерживает основной функционал IaaS-симулятора, при этом улучшая user experience исследователей.

Таблица 4.4: Сравнение функционала симуляторов

Свойство	CloudSim	DISSECT-CF	Новый симулятор
Симуляция жизненного цикла ВМ и ресурсов	есть	есть	есть
Симуляция сети	есть (в последних версиях)	есть	нет, но архитектура позволяет добавить
Симуляция переменной нагрузки ВМ	есть	есть	есть (ограниченная поддержка в RPC-режиме)
Способ определения алгоритмов планирования	с помощью Java ООП	с помощью Java ООП	с помощью C++ ООП, с использованием RPC-планировщика
Измерение энергопотребления	есть	есть	есть
Интерфейс управления	реализация на Java	реализация на Java	реализация RPC-клиента, интерфейс командной строки
Использование распределённых планировщиков	нет	нет	есть, в RPC-режиме

5 Заключение

В рамках данной выпускной квалификационной работы были проанализированы существующие симуляторы облачных вычислений, определены их основные свойства и недостатки. Был реализован новый симулятор, который не содержит большинства найденных проблем, прост в использовании и снабжён подробной документацией.

Помимо базовой функциональности, присущей большинству аналогов, в реализованном симуляторе имеются новые функции, создающие новые сце-

нарии его использования — RPC-интерфейс доступен из runtime, удобного исследователю, кроме того, у пользователей данного симулятора не будет необходимости генерировать файлы с конфигурацией для тестирования — запросы можно отправлять напрямую из программного кода.

Вместе с тем, остаются и пути развития данной работы. Многие части симулятора реализованы «наивно» и могут быть оптимизированы — например, имеет смысл применить более оптимальную структуру данных для очереди событий, учитывающую специфику поступающих к ней запросов. Также всегда остаётся поле для дальнейших абстракций, в частности, можно перейти от симулятора исключительно IaaS к более общим моделям, если разработать интерфейс для сущности «потребителя ресурсов», который сейчас тождественен понятию «виртуальная машина».

Некоторые планы были описаны и в тексте работы. Все они выполнимы, так как симулятор проектировался в достаточной степени абстрактным и расширяемым — основа для новых функций уже реализована и протестирована.

Список источников

1. Kecskemeti Gabor. DISSECT-CF: A simulator to foster energy-aware scheduling in infrastructure clouds // Simulation Modelling Practice and Theory. — 2015. — Vol. 58. — P. 188–218. — Special issue on Cloud Simulation. Access mode: https://www.sciencedirect.com/science/article/pii/S1569190X15000842.

2. SCORE: Simulator for cloud optimization of resources and energy consumption / Damián Fernández-Cerero, Alejandro Fernández-Montes, Agnieszka Jakóbik et al. // Simulation Modelling Practice and Theory. — 2018. — Vol. 82. — P. 160–173. — Access mode: https://www.sciencedirect.com/science/article/pii/S1569190X18300030.

3. Koomey Jonathan et al. Growth in data center electricity use 2005 to 2010 // A report by Analytical Press, completed at the request of The New York Times. — 2011. — Vol. 9, no. 2011. — P. 161.

4. Akhter Nasrin, Othman Mohamed. Energy aware resource allocation of cloud data center: review and open issues // Cluster computing. — 2016. — Vol. 19, no. 3. — P. 1163–1182.

5. Kar Ipsita, Parida R.N. Ramakant, Das Himansu. Energy aware scheduling using genetic algorithm in cloud data centers. — 2016. — P. 3545–3550.

6. CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms / Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov et al. // Software: Practice and Experience. — 2011. — Vol. 41, no. 1. — P. 23–50. — https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.995.

7. Mansouri N., Ghafari R., Zade B. Mohammad Hasani. Cloud computing simulators: A comprehensive review // Simulation Modelling Practice and

Theory. — 2020. — Vol. 104. — P. 102144. — Access mode: https://www.sciencedirect.com/science/article/pii/S1569190X20300836.

8. Sinha Utkal, Shekhar Mayank. Comparison of various cloud simulation tools available in cloud computing // International Journal of Advanced Research in Computer and Communication Engineering. — 2015. — Vol. 4, no. 3. — P. 171–176.

9. Garg Saurabh Kumar, Buyya Rajkumar. NetworkCloudSim: Modelling Parallel Applications in Cloud Simulations. — 2011. — P. 105–113.

10. Pakize Seyed Reza, Khademi Seyed Masood, G Abolfazl. Comparison Of CloudSim, CloudAnalyst And CloudReports Simulator in Cloud Computing.

11. Cai Zhicheng, Li Qianmu, Li Xiaoping. Elasticsim: A toolkit for simulating workflows with cloud resource runtime auto-scaling and stochastic task execution times // Journal of Grid Computing. — 2017. — Vol. 15, no. 2. — P. 257–272.

12. Writing a Discrete Event Simulation. — https://users.cs.northwestern.edu/~agupta/_projects/networking/QueueSimulation/mm1.html. — (Accessed on 05/17/2021).

13. VM statuses | Yandex.Cloud - Documentation. — https://cloud.yandex.com/en-ru/docs/compute/concepts/vm-statuses. — (Accessed on 05/17/2021).

14. Supported languages | gRPC. — https://grpc.io/docs/languages/. — (Accessed on 05/17/2021).