Министерство образования и науки Российской Федерации
Московский физико-технический институт (государственный университет)
Факультет управления и прикладной математики
Кафедра информатики

Диссертация допущена к защите
зав. кафедрой

_____ Петров Игорь Борисович

«____» _____ 2016 г.

ДИССЕРТАЦИЯ
на соискание ученой степени
МАГИСТРА

Тема: **Разработка модели среды распределенных вычислений Evergrid для сравнения алгоритмов управления потоками задач и данных**

Направление: 03.04.01 – Прикладные математика и физика

Магистерская программа: 010956 – Математические и информационные технологии

Выполнил студент гр. 073а _____ Колеснев Роман Владимирович

Научный руководитель,
к. ф.-м. н., доцент _____ Устюжанин Андрей Евгеньевич

Рецензент,
к. ф. н., доцент _____ Рыков Владимир Васильевич

Москва – 2016

Оглавление

Введение

В последние несколько лет развитие информационных технологий и, в частности, Интернета крайне активно. Многие сервисы и инструменты, которые были актуальны несколько лет назад, сейчас уже считаются устаревшими. Это создает потребность в создании современных инструментов для решения различных задач. Таких инструментов, которые превзойдут своих предшественников по удобству, функциональности или даже принесут что-то качественно новое. Данная работа посвящена раннему этапу разработки одного из таких продуктов. И действительно, как будет далее видно, создание подобного продукта с подобной архитектурой несколько лет назад было бы существенно сложнее, нежели сейчас.

Если мы говорим о новом инструменте, то в первую очередь стоит рассказать о двух вещах: как он называется и какие задачи он решает. Называется он Evergrid. А решаемая задача - это предоставление web-сервиса для проведения вычислительных экспериментов, публикации их результатов вкупе с программной реализацией и данными, предоставление удобного интерфейса для их воспроизведения и модификации. Проект, несомненно, большой - поэтому этот диплом посвящен только его малой части, одной из самых первых и рассматривает решение более узкой задачи нежели создание этой системы целиком.

У проекта есть и другая сторона - вычислительные мощности тоже предоставляются пользователями. Причем необязательно бесплатно. То есть сервис, помимо прочего, является посредником между потребителями и провайдерами ресурсов.

На данный момент аналогичных решений на рынке нет. Как минимум потому, что Evergrid по своей архитектуре не является ни cloud-платформой, ни grid'ом в чистом виде.

В данном дипломе решаются две задачи:

- на основе общих принципов Evergrid придумать и описать подходящую для него архитектуру
- на основе описанной архитектуры реализовать симулятор для сравнения работы алгоритмов планировщиков

В виду ранней стадии разработки проекта описание архитектуры не является доскональным, а скорее задает общую структуру и критичные ограничения. Но, тем

не менее, описание достаточно подробно, чтобы на основе него можно было сделать симулятор.

Также рассмотрены существующие симуляторы для grid'ов и обоснована их неэффективность для использования в Evergrid. Показаны сильные стороны выбранного для реализации подхода.

Одной из ключевых особенностей работы является активное использование языка программирования go - именно на нем реализован симулятор.

Реализованный симулятор представляет из себя многоагентную систему. Причем, та его часть, которая реализует среду для выполнения агентов, независима и может использоваться отдельно для решения других задач. Выводом симулятора являются логи в формате JSON, что позволяет проводить произвольный анализ результатов. Код документирован, что облегчит его последующую разработку.

Глава 1

О системе Evergrid

В первую очередь стоит подробнее объяснить, что такое Evergrid, какие предпосылки обуславливают его актуальность и новизну, какие задачи он призван решать и каковы сценарии его использования. Все это важно еще и тем, что задает основу для разработки и описания подходящей архитектуры, а без нее невозможно говорить о написании симулятора. В конце главы будет пара слов о самом симуляторе, а начнем мы с предпосылок появления Evergrid.

Согласно Эрику Шмидту из Google, теперь каждые два дня человеческая раса создает столько информации, сколько мы производили от начала нашей цивилизации до 2003 года. Это что-то около пяти эксабайт информации в день. Если копнуть глубже, то не совсем понятно, откуда он взял эти цифры[1] , но сложно спорить с основной идеей - информации ежедневно производится беспрецедентно много. Эрик Шмидт озвучил свою мысль в 2010 году. Логично предположить, что спустя 6 лет информационные потоки стали только мощнее. Такое количество производимой информации делает более актуальной проблему ее обработки и экспериментов над ней. Есть даже такие на первый взгляд странные исследования, как определение мест землетрясений в реальном времени с помощью твиттера[1]. Разработка подобных методов зачастую связана с экспериментами над большими массивами данных (гигабайты информации) и, если когда-то работа с такими объемами была мало распространена, то сейчас этим уже никого не удивить. Соответсвенно, инструменты, которые помогают в этих задачах, востребованы на рынке. Как один из многочисленных примеров можно привести Apache Spark.

Помимо инструментов есть еще один не менее важный фактор - коммуникация. Эффективное развитие науки невозможно без эффективной коммуникации между отдельными учеными, университетами, исследовательскими центрами и т. д. Доступность информации - тоже, по сути, является элементом коммуникации. Написание многих современных научных работ было бы более трудоемким без таких инструментов, как Google Scholar, без отлаженной работы научных изданий и прочих факторов, связанных с эффективным обменом результатами исследований. Вывод из этого можно сделать такой: чем лучше коммуникация - тем эффективней развивается наука.

[1] http://readwrite.com/2011/02/07/are-we-really-creating-as-much/

Если говорить об экспериментах над данными, то в плане коммуникации возникает одна очень заметная проблема: воспроизведение экспериментов. Ведь, чтобы воспроизвести эксперимент, надо воссоздать инфраструктуру, в которой он был проведен, и уже на этом этапе порой начинаются проблемы. Не всегда инфраструктура достаточно подробно описана в самой работе. Не всегда с первого раза получается заставить работать описанную инфраструктуру. Не всегда ее можно воспроизвести на локальной машине с используемой исследователем операционной системой. Этот список возможных технических проблем можно продолжить на несколько страниц. И то он будет неполным. И, когда исследователь, желающий повторить оригинальный эксперимент, но с другими данными, сталкивается с такой массой проблем - он рискует потратить много времени на несущественные для его работы вещи.

Также важен открытый доступ к результатам эксперимента. На данный момент с этим нет особых проблем, но в контексте Evergrid стоит об этом упомянуть: результаты должны быть открыты и доступны. Думаю, можно даже не объяснять ценность этого фактора.

Помимо воспроизводимости эксперимента важной является и возможность его модификации. Я говорю о случаях, когда описан сам алгоритм, а его реализация либо не является частью работы, либо сложна в использовании. Evergrid должен учитывать этот фактор и делать модификацию эксперимента максимально доступной. Это означает, что мы должны ввести некоторые стандарты оформления для реализации. На самом деле это вытекает не только из идеологических соображений, но и из технических (сложно сделать систему, которая будет запускать что угодно и как угодно). Более того, введение подобных требований в первую очередь облегчит воспроизводимость. Итого: Evergrid должен задавать некоторый формат (или форматы) оформления программных реализаций ради удобства модификации и воспроизводимости.

Итак, Evergrid, очевидно, нацелен на большое количество пользователей. Это означает, что нужно много вычислительных мощностей для обеспечения адекватного выполнения задач. Часть мощностей можно получить бесплатно, но их почти наверняка не хватит. Соответсвенно, в системе будут присутствовать те мощности, за которые придется платить. И здесь есть два подхода: покупать эти мощности самим или быть посредником. Второй подход интереснее и более чем имеет право на жизнь в виду того факта, что зачастую кластеры различного размера порой просто простаивают. Если мы дадим возможность продавать эти мощности, то выгода будет

всем сторонам. Пользователям - т. к. будет ценовое разнообразие, команде Evergrid - т. к. это неплохая бизнес-модель, поставщикам мощностей - получить выгоду от простаивающих кластеров и просто сделать благое дело. Именно реализация этой схемы является одной из ключевых особенностей Evergrid как проекта.

Итак, мы развернуто сформулировали основные идеологические предпосылки для создания Evergrid. Теперь мы можем их выразить в виде лаконичного и короткого списка. Evergrid - это сервис:

- для выполнения экспериментов над данными
- для публикации результатов этих экспериментов
- с возможностью удобно модифицировать исходный эксперимент как в плане алгоритмов, так и в плане данных
- с возможностью опубликовать результат выполнения модификации
- являющийся посредником между поставщиками вычислительных мощностей и конечными пользователями
- задающий некий формат или форматы оформления программной части экспериментов ради удобства воспроизведения и модификации (в том числе и на локальных машинах пользователей)

Опираясь на этот список, мы теперь можем сформулировать основные сценарии использования системы. Но полноценные сценарии использования системы в виде диаграмм в данной работе будет делать опрометчиво. Во-первых, это работа для UX-специалиста, коим я сейчас не являюсь. Во-вторых, в рамках данной работы это неактуально. Важно выявить список *возможностей* системы, а архитектуру проектировать исходя из того, чтобы эти возможности в ее рамках эффективно реализовывались. Вот этот список, разбитый по группам пользователей:

- Исследователи:
  - Возможность загружать датасеты в систему
  - Возможность загружать процессоры (программные реализации экспериментов) в систему
  - Возможность запустить определенный процессор на определенном датасете

  - Возможность задать ограничения на выполнение (уложиться в указанную стоимость, предпочитать быстрое выполнение, несмотря на стоимость, начать выполнение строго до определенного времени и пр.)
  - Возможность получить результат выполнения
  - Возможность опубликовать связку датасет+процессор+результат
  - Возможность "склонировать"опубликованный эксперимент и вносить изменения

- Гости:
  - Возможность просматривать опубликованные эксперименты
  - Возможность скачивать результаты, датасеты и процессоры

- Поставщики мощностей:
  - Возможность видеть статистику использования ресурсов
  - Возможность модифицировать параметры использования ресурсов
  - Возможность предоставлять новые ресурсы, закрывать доступ к существующим.

Исследователи - это, очевидно, те, кому необходимо проводить эксперименты над данными. Гости - незарегистрированные пользователи. Их функционал доступен всем. Поставщики мощностей - понятно из наименования.

Это относительно грубый список, но достаточный для поставленных задач.

Отдельно упомяну про реализованный симулятор: полноценно говорить о его актуальности, новизне и практической значимости без описания архитектуры не получится, но, если забежать вперед и попробовать сформулировать краткий список, то он будет таким:

- нет симуляторов на go, и в то же время go популярен для микровервисов
- популярные симуляторы (simgrid, gridsim, alea 2, netlogo) не используют языки с акторной или CSP-моделью
- популярные симуляторы не поддерживают необходимую архитектуру без существенных модификаций

- модификация существующего решения не менее сложна, чем написание своего
- изоморфность (использование одного и того же кода как для симуляции, так и для реального окружения) дает существенные преимущества
- симуляция архитектуры на раннем этапе может заблаговременно выявить ее недостатки
- расширяемость и модифицируемость созданного симулятора
- готовая отправная точка для создания остальных компонентов системы

Глава 2

Постановка задач

Пусть список возможностей из предыдущей главы получился весьма лаконичным, его реализация - это огромная работа. Напомню, что данный диплом представляет собой лишь один из самых первых этапов - это имеет прямое влияние на те задачи, которые непосредственно решались, и на то, как они решались.

Краткий список того, над чем велась работа, выглядит так:

- Четко сформулировать базовые требования к системе Evergrid
- Разработать описание архитектуры, удовлетворяющее этим требованиям
- Для данной архитектуры реализовать среду моделирования для исследования эффективности планировщиков выполнения задач и распределения данных

Базовые требования были описаны и обоснованы в предыдущей главе.

2.1. Проектирование архитектуры

Наличие описания архитектуры является необходимым условием для создания среды моделирования, которая, в свою очередь, является темой диплома. Мало того, это описание с одной стороны должно быть достаточно подробным, с другой - максимально общим, чтобы не "замораживать"спецификацию тех аспектов системы, детали реализации которых несущественны для данной работы. Тем не менее, даже для таких аспектов будет не лишним дать некоторую обоснованную рекомендацию - она может послужить удачной точкой для принятия финального решения в будущем.

Итого, в плане проектирования архитектуры надо решить следующие задачи:

- из каких компонентов состоит система
- как эти компоненты распределены по физическим машинам
- как компоненты взаимодействуют друг с другом
- какие технологии подойдут лучше всего для их реализации

2.2. Реализация симулятора

Вторая и основная задача диплома - это реализация симулятора. Предметом симуляции является планировщик - та часть системы, которая управляет размещением данных и выполнением задач на подконтрольных ресурсах. Перед непосредственно реализацией симулятора надо ответить на следующие вопросы:

- каким требованиям должен отвечать симулятор
- есть ли готовые решения либо решения, которые удобны для модификации

Причем важно помнить о том, что какие-то аспекты архитектуры могут изменяться, и нужно будет изменять симулятор в соответствии с ними. Подобные корректировки симулятора не должны быть неоправданно сложными.

Глава 3

Спроектированная архитектура

3.1. Основные ограничения

Проектирование хорошей архитектуры основывается на:

- достаточно подробной постановке задачи
- выявлении ограничений
- использовании актуальных технологий как составных блоков
- максимальной простоте реализации не в ущерб необходимому уровню качества

Постановка задачи у нас есть, последние два пункта постараемся соблюдать. Осталось понять, какие у нас ограничения. Под ограничениями понимаются технические аспекты, которые напрямую следуют из постановки задачи и ограничивают нас в свободе выбора тех или иных технологий, тех или иных подходов. В этом разделе я приведу те ограничения, которые считаются неочевидными или заслуживающими отдельного упоминания.

Первое из них - связанное с безопасностью. Мы используем вычислительные ресурсы, предоставляемые третьими лицами. И они имеют неограниченный (root) доступ к ним. Поскольку процесс предоставления этих ресурсов подразумевается достаточно свободным, будет правильно представить злоумышленника на месте поставщика мощностей. Какие потенциальные угрозы он может предоставлять?

- захват не принадлежащих ему ресурсов
- нарушение работы кластера (например, вмешательство в работу планировщика)
- нарушение корректной работы предоставленного вычислительного ресурса (фальсификация результатов, намеренное замедление скорости вычислений и т. п.)

Первые две угрозы нивелируются достаточно просто: *на арендуемых ресурсах не должен выполняться код, связанный с управлением кластером. Только выполнение задач и отправка результатов.*

Третья - наиболее сложная. Но риски можно свести к минимуму, если *вообще не запускать на арендуемых ресурсах компоненты системы*. Естественной реализацией этого принципа является удаленное управление по SSH. Тогда злоумышленник будет знать минимум о текущем состоянии системы. Прочие методы борьбы с этой угрозой уже выходят за рамки этой работы.

Следующее ограничение происходит из того, как мы будем выполнять задачи. Очевидно, что нам нужна виртуализация - иначе наш кластер быстро превратят в ботнет. Нам нужна универсальность - хочется покрыть как можно больше сценариев использования. Также нам нужна скорость - следовательно, нам нужна легковесная виртуализация. Третий критерий - технология должна быть простой в использовании. В идеале - проведение эксперимента на своей локальной машине не должно отличаться от выполнения его на наших мощностях. Сложив все эти требования воедино, мы получим ответ - Docker. И это решение имеет много преимуществ: Docker отлично отвечает требованиям к изоляции, удобству воспроизводимости и модификации. А его инструментарий органично вписывается в архитектуру. От предоставляемых мощностей нам в итоге требуется:

- свободное место на диске
- доступ по ssh
- корректно работающий docker
- возможность мониторинга нагрузки (чтобы регистрировать не относящуюся к нашему сервису нагрузку)

Даже неограниченный доступ в Интернет не является жестким требованием - мы можем собрать контейнер на наших машинах и готовый образ передать по ssh.

Возникает вопрос о том, как пользователь должен предоставлять контейнер со своим алгоритмом. Если следовать идеям доступности и простоты модификации, то наиболее очевидное решение - Github. Наиболее органично будет работать с специально оформленными github-репозиториями, которые содержат все необходимое для сборки контейнера.

Использование Docker на первый взгляд накладывает ограничение вида „одна задача = одна машина”, но это можно обойти - ведь мы можем активизировать несколько машин и разрешить контейнерам общаться между собой. Тем не менее,

в рамках данной работы акцент сделан именно на подходе "одна задача = одна машина". Несмотря на это, архитектура должна быть пригодна для обоих вариантов.

Если уж рассмотрели то, как загружать реализации алгоритмов (далее будем называть их контейнерами, раз имеем в виду Docker) - то стоит сказать пару слов о загрузке датасетов. Никаких особых ограничений здесь незаметно. Все, что нужно - это иметь системе возможность закачать собранный контейнер на свои машины. Путей достижения этого много, и в данной работе они не будут рассматриваться (за исключением перемещения датасетов внутри самой системы).

И последнее ограничение, о котором я хочу сказать, - это CAP-теорема. Мы имеем дело с распределенной в глобальной сети системой, а в таком случае нельзя забывать о CAP-теореме. Нарушение связи между элементами системы не должно приводить к ее некорректной работе.

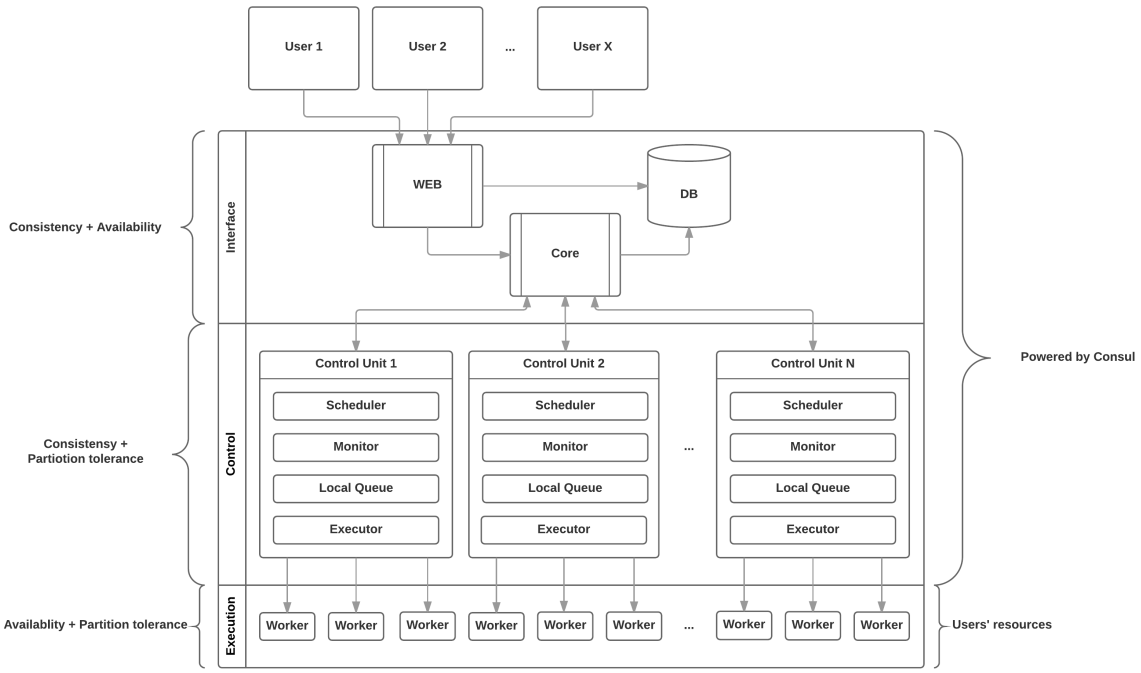

Рис. 3.1. Диаграмма архитектуры

3.2. Предлагаемая архитектура

Теперь можно рассмотреть конкретную реализацию, удовлетворяющую описанным выше принципам. Диаграмма предлагаемой архитектуры показана на рисунке 3.1. Далее мы рассмотрим в отдельности каждый из ее слоев и компонентов.

Для начала объясню общую структуру. Здесь видно разделение на три слоя плюс пользователи системы. User 1 – User X - это пользователи. Каждый слой состоит из компонентов. Распределение компонентов по машинам специфично для каждого слоя. Для слоя Interface оно явно не ограничивается предложенной архитектурой. В зависимости от нагрузки и прочих факторов все эти компоненты могут быть как на одной машине, так и разнесены по нескольким. Для слоя Control истинно правило "одна машина - один Control Unit". Для слоя Execution все просто - один Worker равен одной машине, предоставленной для вычислений.

Направление стрелок - это направление запросов между компонентами. В случае долговременного соединения - стрелка направлена от инициатора.

Первые два слоя - это “наши” машины, которым мы доверяем, и только у нас

есть к ним доступ.

Третий слой - машины, предоставленные извне, мы им “не доверяем”.

Слева показаны приоритеты в терминах CAP-теоремы для каждого слоя.

Справа показано, что первые два слоя используют Hashicorp Consul для service discovery и leader election.

3.2.1. Слой Execution

На этом слое находятся машины, предоставленные пользователями системы. Они частично конфигурируются нами и не содержат никакого кода, который бы управлял слоями выше (из соображений безопасности). Про специфику работы с этими ресурсами и требования к ним было написано выше.

Все запросы приходят со слоя Control. То есть машинам даже необязательно знать IP других компонентов системы. Отдельный случай - если машины не имеют ”белого” IP: в этом случае в их обязанность входит самостоятельно поддерживать соединение со слоем Control, и нам все же придется ставить на Worker’ы какое-нибудь приложение, которое будет обеспечивать соединение со слоем Core.

3.2.2. Слой Control

Состоит только из Control Unit’ов. Каждый Control Unit - это отдельная машина. Каждый Worker из слоя ниже принадлежит только одному Control Unit’у.

Control Unit состоит из нескольких логических компонент и представляет из себя монолитную программу.

Все Control Unit с этого слоя представляют из себя распределенную систему. То есть запросы с верхнего слоя Interface по своей сути направлены не к конкретному Control Unit’у, а ко всей системе целиком.

Смысл подобного разбиения - это большая стабильность системы. Т. е. эффективным подходом будет, например, расположить по одному Control Unit на географическую зону.

Смысл всей этой системы в следующем:

- Получать задания со слоя выше (от Core)
- Раскидывать датасеты по Worker’ам
- Запускать вычисления на Worker’ах

- Отправлять результаты вычислений на Worker'ах на слой выше (в Core).

Теперь конкретные примеры запросов, которые могут приходить в эту систему:

- Загрузи датасет
- Загрузи и собери этот docker-контейнер
- Обнови этот докер контейнер
- Обнови этот датасет
- Запусти этот контейнер с этим датасетом

Из подобных задач формируются локальные очереди (Local Queue). В какую локальную очередь и на какой Worker отправить задачу, определяет "распределенный" Scheduler. Также состояние, корректность и доступность Worker'ов, прочие глобальные характеристики системы отслеживаются Monitor'ом. Monitor может давать информацию о всех доступных Worker'ах системы. Monitor - основной инструмент Scheduler'а для получения текущего состояния системы. Executor отвечает за "опустошение"очереди и выполнение задач на Worker'ах.

Результат выполнения работы отправляется в Core, который, в свою очередь, решает, где хранить этот результат.

3.2.3. Слой Interface

Распределение по машинам в этом слое диктуется лишь нагрузкой. При малом количестве пользователей - можно все три компонента держать на одной машине. Теперь отдельно про каждый компонент.

WEB Веб-приложение, написанное на любом адекватном web-фреймворке. Именно через него происходит постановка задач, просмотр результатов и прочее. Является единой точкой управления системой как для рядовых пользователей, так и для администраторов. Важно понимать, что здесь используются не только HTML + JS, но и реализована вся бизнес-логика интерфейса.

DB Основная база данных. В ней хранится все относящееся к бизнес-логике приложения. По своему содержанию должна быть независима от нижестоящих слоев. Т. е. если конфигурация всего того, что есть ниже, станет иной - данные не должны стать некорректными.

Core WEB не общается напрямую со слоем Control. Он работает с Core через API, а оно в свою очередь контролирует слой ниже. Т. е. Core - это микросервис взаимодействия со слоем Control. Это важное разделение, поэтому подчеркну: WEB - это бизнес логика, Core - микросервис, который контролирует общение с распределенной вычислительной системой. Core не проверяет никаких параметров, связанных с бизнес-логикой: кто поставил задачу, сколько у него денег на счету и прочее.

3.3. Выбор технологий для реализаций компонентов

Теперь, когда есть представление о структуре, надо понять, какие есть эффективные способы реализации. Если не оговорено иного, предложенные способы несут рекомендательный характер.

3.3.1. WEB

Посколько Evergrid является по своей сути стартапом, то важным аспектом является скорость разработки. То есть надо уметь быстро создать прототип и иметь возможно быстро вносить изменения. На Java и ASP.NET подобное получается плохо, особенно у небольших команд. Наиболее успешно используемые технологии - это Ruby on Rails[1] , Node.js[2] фреймворки[3] , Django[4] и прочие python-фреймворки, PHP[5].

Есть также менее распространенные решения, но тоже заслуживающие внимания: Play Framework[6] , Phoenix Framework[7].

Веб-фреймворки можно разделить на два класса:

С "синхронной"архитектурой – когда запускается ровно N обработчиков запросов (в отдельных процессах или тредах). То есть система может одновременно обрабатывать не более чем N запросов. Выполнение запросов изолировано друг от друга. В данных ограничениях удобно писать код, но страдает масштабируемость и не стоит использовать продолжительные по времени подключения (websocket'ы, например). Синхронным такой подход называется т. к. в случае

[1] http://rubyonrails.org/
[2] https://nodejs.org/en/
[3] Например: http://expressjs.com/
[4] https://www.djangoproject.com/
[5] http://php.net/
[6] https://www.playframework.com/
[7] http://www.phoenixframework.org/

одного обработчика мы блокируем выполнение следующего запроса до завершения предыдущего.

С "асинхронной"архитектурой – в этом подходе для каждого входящего запроса на лету создается свой поток обработки. Т. е. мы, в отличие от предыдущего подхода, не вынуждены ждать завершения обработки предыдущего запроса для начала обработки нового. Для такой архитектуры сложнее писать код, но обычно решения на ней обладают лучшей производительностью и более широким спектром возможностей.

Это описание, достаточное для общего понимания. Еще один важный параметр - это используемый язык программирования. От него зависит как скорость продукта, так и, отчасти, легкость его сопровождения.

Сначала рассмотрим технологии, использование которых может оказаться неэффективным:

Java/ASP.NET Плохо годятся для небольших команд. ASP.NET плох своим акцентам на Windows-сервера. Java, на фоне прочих языков, слишком "многословна и быстро реализовать на ней прототип довольно тяжело.

Node.js Несмотря на асинхронность архитектуры, javasript работает в одном потоке. Но главный его недостаток - это дизайн языка. У JS крайне слабая система типов, запутанная спецификация и много неоправданно неочевидных моментов.

PHP Очень популярен в вебе, до сих пор развивается, множество фреймворков. Но дизайн языка хоть и лучше, чем у JS, все равно уступает ruby и python. Другой его недостаток - слабость как языка общего назначения - выражается в том, что на нем неудобно писать что-то кроме непосредственно генерации ответов на запросы к веб-серверу. А подобное может понадобиться - например, для организации выполнения внешних очередей задач или удобного использования websocket'ов. Кратко говоря, PHP неоправданно накладывает слишком много ограничений.

Теперь рассмотрим список технологий, хорошо подходящих этому проекту:

Python Python активно используется как для веб-разработки, так и для научных целей. Веб-фреймворки на нем довольно качественные, и есть большое количество решений для типовых задач (например, регистрация пользователей и пр.). Поскольку Python скриптовый язык с GIL, большинство решений на нем реализуют синхронную архитектуру. Это не настолько большой недостаток, как может показаться, но, если высокая производительность или использование websocket'ов является критичным, - python может оказаться не лучшим выбором.

Ruby on Rails В основном ситуация, схожая с python за исключением следующих различий: редко используется в научной среде; язык более гибкий, но и более сложный, чем python; большее количество библиотек с готовыми решениями для веб-разработки. Отдельным преимуществом является и то, что Rails однозначный лидер среди Ruby-фреймворков, а это означает, что почти любая развитая библиотека умеет корректно с ним работать. В случае с python все более разобщено. То есть, если есть возможность эффективно использовать Rails - то это является удачным решением в рамках синхронной архитектуры.

Phoenix Framework (Elixir) Если важно использование асинхронной архитектуры, то стоит вспомнить об Erlang. Продукты, написанные на Erlang, в виду особенностей BEAM (Bogdan/Björn's Erlang Abstract Machine) и принципов OTP (Open Telecom Platform) отличаются высокой стабильностью и производительностью. Elixir - это молодой язык для BEAM, который более удобен в использовании, имеет больше возможностей, чем Erlang, и может без ограничений использовать его библиотеки. На этом языке реализован Phoenix Framework. Что язык, что сам фреймворк - еще молодые, используются в production, но пока еще не получили широкого распространения. Сам фреймворк удобен в использовании. Данный вариант хорош, если хочется получить преимущества Erlang/OTP-экосистемы: высокая стабильность, высокая производительность на IO задачах, дешевый и удобный параллелизм благодаря green threads и акторной модели, преимущества подходов из функциональной парадигмы программирования.

Play Framework (Scala) Еще одно возможное решение, если нужна асинхронность и производительность. Scala - преимущественно функциональный язык про-

граммирования поверх JVM, имеющий возможности интероперабельности с Java. Тоже использует акторную модель (Akka). Более зрелое решение, чем phoenix, и более широко распространено на данный момент. Из недостатков можно отметить то, что в Erlang более развитые средства для мониторинга и более стабильная реализация акторной модели и супервизоров. Также Scala является более сложным в изучении и использовании языком, нежели Elixir/Erlang. Из преимуществ можно отметить, что мы получаем доступ к внушительному парку Scala и Java библиотек.

3.3.2. Core и Control Unit

Эти два компонента формируют внутреннюю инфраструктуру сервиса. В данном случае корректным будет следующий список требований:

- высокая производительность (система должна адекватно вести себя под высокими нагрузками)
- выбранное решение должно использоваться для схожих известных продуктов
- иметь развитые библиотеки для сетевого взаимодействия
- чем проще будет инициализировать новые сервера - тем лучше
- язык должен быть популярен и иметь активное сообщество
- простота языка будет плюсом

Требование высокой производительности ставит под сомнение использование интерпретируемых языков. Из "быстрых"языков в первую очередь приходят на ум C/C++ и go. ASP.NET плохо дружит с Linux, а Java/Scala довольно прожорливы в плане памяти и не являются простыми в использовании языками. C++ - тоже довольно сложный в использовании язык. Если же внимательнее посмотреть на go, то становится видно, что он является хорошим решением:

- на нем написаны продукты hashicorp (nomad, consul), на нем написан docker
- скорость выполнения сравнима с C/C++
- прост в изучении: опытный программист может освоить все основные возможности языка за пару вечеров

- популярен для написания микросервисов
- встроенная в язык модель параллелизма CSP
- активное сообщество и большое количество готовых библиотек
- скомпилированное приложение для своей работы не требует установки самого go или каких-либо специфических пакетов

Поэтому для Core и Control Unit go является подходящим выбором. Причем данный выбор имеет жесткий характер, так как симулятор тоже написан на go, и часть его преимуществ основывается на том, что для реализации инфраструктуры тоже будет использоваться этот язык.

3.3.3. DB

На самом деле на данном этапе сложно предсказать все те требования, которые могут возникнуть для основного хранилища. Возможно, даже одного продукта не хватит, и будет использоваться некая комбинация (например: часто можно встретить связки Postgres + Redis).

Но, в качестве решения по умолчанию, стоит рассматривать именно Postgres. На данный момент это наиболее универсальное и активно используемое решение среди open source баз данных.

3.4. Ограничения, связанные с CAP-теоремой

Поскольку мы имеем дело с распределенной системой, то стоит явно указать приоритеты в терминах CAP-теоремы. Причем эти приоритеты отдельные для каждого слоя (именно из-за этого вообще появилось разделение на слои).

Также первый и второй слой используют consul - он предоставляет service discovery и leader election интерфейсы, к тому же он создан для решения этих задач как раз-таки в распределенных системах.

Слой Execution Для этого слоя подходит конфигурация [C]AP. Если машина потеряла связь с системой – перераспределяем нагрузку. Здесь важно в любой момент времени максимально использовать доступные ресурсы.

Слой Control При критичной сегментации сети в слое, Control Unit'ы перестают принимать запросы, но продолжают выполнение локальных очередей. Примерно так выглядит жертва availability во имя consistency и partition tolerance. Стоит сказать, что это довольно грубый подход. Но начать разработку проще именно с него. Как второй шаг может подойти weak consistency подход ("согласованность в итоге").

Слой Interface На этом слое крайне важна согласованность данных, а без доступности он не имеет смысла. Поэтому жертвуем partition tolerance.

3.5. Требования к симулятору

Имея описание архитектуры, можно сформулировать список требований к симулятору.

Первое из них связано с симуляцией сети. Для симуляции данной архитектуры нам не важен пинг между ее компонентами. Нам важна только скорость передачи данных между ними и сам факт наличия связи. Отдельный случай - это черезмерно большой пинг - это может повлиять на порядок запросов и подобное. Но по факту нам нужно симулировать не пинг, а факт поздней доставки сообщений. В итоге имеем следующие требования:

- симуляция видимости между машинами
- симуляция скорости передачи данных
- симуляция разрыва соединения
- симуляция поздно пришедших сообщений

Второе ограничение связанно с тем, что мы должны симулировать время. Т. к. нам важно понимать когда выполнялась задача и сколько времени она выполнялась. Наиболее гибким простым решением является "дискретная"природа симулируемого времени. То есть мы используем дискретный таймер, и каждый агент системы делает ту работу, которую теоретически может успеть за эту дискретную единицу времени.

И третье ограничение - симулятор должен быть достаточно гибким, чтобы при изменении принципов работы компонент можно было легко внести изменения в модель.

Всякий симулятор преследует цель проверки гипотез. В данном случае симулятор должен помогать в исследовании:

- эффективности алгоритмов планировщиков
- принципов взаимодействия компонентов системы

Глава 4

Сравнение с существующими решениями

В рамках работы была рассмотрена возможность использования или модификации существующего решения вместо написания своей реализации. Для корректного сравнения стоит сразу перечислить преимущества написания своей реализации на go. Если взять во внимание тот факт, что подобных приложений на go найдено не было, то преимущества таковы:

- возможность использовать один код планировщика как для симуляции, так и для реального окружения - это дает дополнительные гарантии корректности его реализации
- удобная возможность записи реальных сценариев работы и последующей симуляции их с различными вариантами планировщиков
- go - простой в использовании язык и имеет встроенную модель параллелизма CSP, это облегчает разработку и модификацию кода симуляции
- go - быстрый язык, скорость его работы сравнима с C/C++
- в go встроен автоматический детектор состояний гонки
- go популярен для микросервисов, но на данный момент нет ни одного симулятора для go

Следует понимать, что использование или модификация готового решения должны предоставлять преимущества, перевешивающие этот список.

4.1. NetLogo

NetLogo[2] - очень известное решение для моделирования и исследования работы многоагентных систем. Его преимущество состоит в развитых инструментах визуализации.

Но для решения конкретной задачи NetLogo подходит плохо, т. к.:

- NetLogo использует свой язык программирования. Это язык узкого назначения и писать на нем менее удобно, чем на популярных языках общего назначения

- мы симулируем не только scheduler, но и прочие компоненты системы. Нам важно, чтобы модель общения между ними была приближена к реальной. Это дает возможности для поиска состояний гонки в рамках архитектуры в целом. Реализация честного общения всех компонентов довольно сложна на NetLogo и потенциально сложнее написания своей реализации на go.

- у него низкая скорость работы

4.2. Узкоспециализированные симуляторы: SimGrid, GridSim, ALEA 2

Среди более узкоспециализированных решений было найдено три кандидата: SimGrid[3], GridSim[4] и ALEA 2[5].

Эти продукты реализованы на Java (GridSim, ALEA 2) и C (SimGrid). У всех трех есть один критический недостаток: без модификации исходного кода невозможно реализовать поставленную задачу - архитектура проекта слишком специфична. А сама задача их модификации получается сложнее, чем написание своего решения. Также усложняется и сопровождение получившегося симулятора - вносить правки тяжелее по двум причинами: Java и C менее удобны чем go, больший шанс неожиданных ошибок в виду возможных конфликтов с изначальной архитектурой симулятора.

Еще одна особенность всех трех симуляторов - они сверхсконцентрированы на анализе планировщика, в то время как одной из наших задач является и анализ архитектуры в целом.

Также есть проблемы специфичные для каждого из решений:

4.2.1. GridSim

Дата последнего обновления - 2010й год. Видимо, проект более не поддерживается, и при возникновении незадокументированных проблем трудно будет связаться с его разработчиком.

Также при использовании GridSim будет проблематично симулировать "динамическое"окружение - разрывы сети, меняющийся список воркеров и прочее.

Итого: суммарная сложность модификации и поддержки решения на GridSim потенциально выше сложности написания и поддержки своей реализации на go. При этом мы еще жертвуем вышеописанными преимуществами своей реализации.

4.2.2. ALEA 2

ALEA 2 является модификацией GridSim, и ее использование сопряжено с теми же проблемами. В отличие от GridSim, она еще поддерживается разработчиками.

4.2.3. SimGrid

Наиболее активно поддерживаемый продукт, но более узкоспециализирован, чем два других. Количество модификаций, которые необходимо будет внести, существенно больше - что усложняет его адаптацию.

К тому же, из соображений скорости он написан на C, а модифицировать код на C обычно сложнее, чем модифицировать код на Java.

Глава 5

Реализованная в симуляторе модель

В этой главе описана реализованная в симуляторе модель. Это даст представление о том, как проходит процесс симуляции и какова степень подробности и точности симулятора.

5.1. Многоагентное окружение с дискретным временем

Основу симулятора представляет из себя многоагентая среда. Она состоит из агентов и ядра, которые общаются друг с другом посредством сообщений.

Ядро системы обеспечивает синхронизацию работы агентов и управляет временем. Симуляция не является непрерывной и разбита на дискретные единицы времени (ticks). Каждый tick соответствует одной минуте. Подобный выбор масштаба обуславливается тем, что предполагается достаточная скорость работы планировщика, чтобы успеть обработать запросы пришедшие в рамках одного tick'а за минуту. А исходя из того, что предполагаются довольно большие объемы датасетов, погрешность времени порядка минуты на передачу данных является допустимой.

В процессе каждого тика агент меняет свое состояние. Данный процесс цикличен. Возможные состояния агента и переходы между ними показаны на рисунке 5.1. Прямоугольники - это состояния. Второй тип элементов - это синхронизации. Синхронизация означает, что невозможно пройти по данному ребру, пока все агенты не будут готовы.

Описание состояний:

Done Означает, что предыдущий тик завершен успешно. Является исходным состоянием агента.

Ready Означает, что агент готов начать работу в рамках нового тика.

Working Означает, что агент в данный момент совершает работу.

Idle Означает, что агент в данный момент бездействует и выполнил всю делегированную ему на данный момент работу.

Описание синхронизаций:

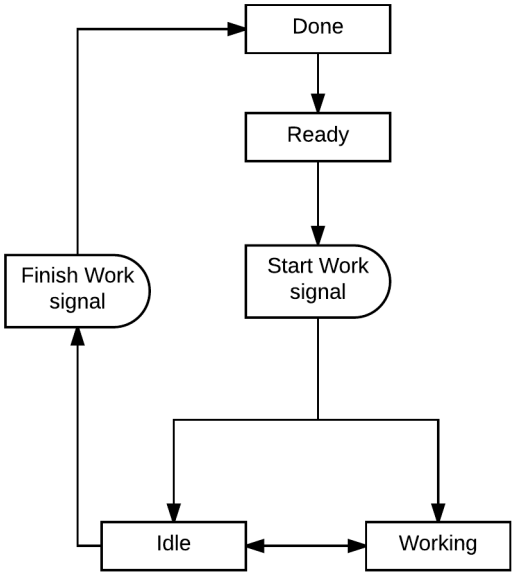

Рис. 5.1. Цикл работы агента

Start Work Дает гарантию, что перед началом работы все агенты достигли состояния Ready

Finish Work Дает гарантию, что перед завершением тика все агенты находятся в состоянии Idle.

Важным аспектом является то, что признак завершения тика - это статус Idle у всех агентов системы. Это нужно помнить и не позволять системе попадать в это состояние, когда не вся работа сделана.

В состоянии Idle агент ждет одного из двух событий: новое сообщение от другого агента либо сигнал Finish Work. В последнем случае состояние меняется на Done. Эта смена состояния единственная не контролируется агентом напрямую.

Помимо основного статуса каждый агент имеет булев маркер stopFlag. Если он равен true, значит агент не запланировал никакой работы на будущие тики, и, если симуляция завершится сейчас, то это будет корректный исход.

Соответственно, симуляция завершается, когда у всех агентов stopFlag маркер равен true.

5.2. Сетевое окружение

В терминах описанной многоагентной среды симуляция сетевого окружения является заботой самих агентов. Причем это несложно реализуется: если агент знает

свое место и места прочих агентов внутри сети, то он сам может накладывать ограничения на общения с этими агентами. То есть симуляция сети сводится к предоставлению конкретным агентам информации о конфигурации сети.

Структура сети в текущей версии симулятора представляет из себя набор из сегментов. Каждый сегмент содержит несколько машин. К одной машине могут быть привязаны один и более агентов.

Скорость передачи данных внутри сегмента и между сегментами может быть различной и является частью его описания.

5.3. Параллелизм и коммуникация

Агенты должны исполняться параллельно друг другу и ядру. Конечно, они могут блокировать друг друга - это соответствует ситуации, когда сервис не может ответить на новый запрос, пока не завершена обработка предыдущего. Но в рамках независимых участков кода мы получаем прирост производительности симуляции благодаря параллелизму. А в виду недетерминированного порядка выполнения этих участков - можем заметить состояния гонки вызванные недостаточно качественной проработкой архитектуры.

5.4. Изоляция агентов

В большинстве случаев агенты должны быть изолированы друг от друга. Это означает, что весь обмен информацией между ними должен происходить посредством сообщений.

В порядке исключения, конечно, можно использовать разделяемую память и мьютексы. Например, это годится для тех случаев, когда мы симулируем распределенное хранилище, а его конкретная реализация и возможность сбоев выходят за рамки эксперимента.

5.5. Генерация сценариев

Генерация сценариев, которые воспроизводятся в симуляции, должна происходить отдельно. Это позволит сравнивать поведение планировщиков и архитектуры в максимально приближенных условиях. Следовательно, симулятор должен предоставлять два инструмента: для генерации сценариев и для запуска симуляции.

Сценарий состоит из трех частей: конфигурация сети, описание датасетов и контейнеров, последовательность запросов в систему.

Сгенерированный сценарий представляет из себя набор YAML-файлов (вышеуказанные части лежат каждая в своем файле). При желании вместо генерации можно составить сценарий вручную или вносить правки в уже сгенерированный.

5.6. Сбор статистики

Нужно предоставить максимальную свободу в обработке данных. Это приводит к тому, что правильно сделать результатом работы симулятора логи, причем в удобном формате. В логи надо писать, что и когда произошло, а с помощью любого внешнего инструмента можно на основе логов выстроить произвольные метрики.

В данной работе в качестве формата логов был выбран JSON в виду хорошего баланса простоты и гибкости.

5.7. Детали реализации компонентов системы

Теперь подробнее рассмотрим принципы реализации конкретных компонентов архитектуры. Всего есть три типа агентов: Core, Control Unit, Worker

5.7.1. Core

Данный агент при инициализации получает список запросов. Каждому запросу из списка соответсвует порядковый номер тика, когда он должен быть послан в систему. Каждый отдельный запрос посылается случайному Control Unit'у.

5.7.2. Control Unit

К каждому Control Unit'у привязана группа Worker'ов. С данными Worker'ами может общаться только этот Control Unit.

В каждый момент времени ровно один Control Unit является лидером.

Control Unit содержит в себе следующий подкомпоненты:

scheduler - сам планировщик. Критически важна возможность использования одной и той же реализации планировщика как для симуляции, так и для реальных условий. Может либо пополнять очереди выполнения произвольных воркеров, либо делегировать запрос лидеру.

monitor - компонент, который используется планировщиком, чтобы узнать глобальное состояние системы

local queue - содержит очередь задач для подконтрольных воркеров. Компонент Executor из описания архитектуры в виду простоты своего устройства встроен в этот подкомпонент модели Control Unit.

Подкомпоненты работают в отдельных потоках выполнения.

Control Unit может принимать два типа запросов:

Запросы на загрузку датасета или запуск контейнера с указанным датасетом - данные запросы могут приходить от Core или быть делегированы другим Control Unit'ом. Обрабатываются scheduler'ом.

Запросы на добавление задачи в очередь подконтрольного Worker'а - данные запросы приходят от других Control Unit'ов. Причина их появления в том, что при своей работе планировщик рассматривает все Worker'ы системы, а не только принадлежащие его Control Unit'у.

5.7.3. Worker

В один момент времени Worker может исполнять только один запрос. Запросы к нему могут приходить только от связанного с ним Control Unit'а.

Worker может принимать три вида запросов:

Загрузка датасета Время, необходимое на это, вычисляется на основе скорости передачи данных, полученной из описания сети.

Сборка контейнера - в данный момент считается, что контейнер собирается за один тик. При желании этот параметр можно изменить.

Выполнение контейнера с заданным датасетом - датасет и контейнер должны быть уже загружены на Worker. Время, необходимое на данную задачу, вычисляется по упрощенной модели: воркер имеет параметр производительности во флопсах, а контейнер имеет параметр, сколько флопс нужно на каждый мегабайт данных. Разделив одно на другое, получаем искомое время.

Глава 6

Результаты

Результатами работы является описание архитектуры (приведенное в тексте диплома) и реализация симулятора доступная на github: https://github.com/ffloyd/evergrid-go.

Код симулятора продокументирован для облегчения его последующей разработки и модификации. Технические тонкости реализации и более подробное рассмотрение принципов работы отдельных компонентов вынесены за рамки текста диплома и являются частью документации.

Удобная он-лайн версия документации доступна по адресу: https://godoc.org/github.com/ffloyd/evergrid-go.

Помимо описания работы компонентов, в документации даны рекомендации по путям эффективной модификации и расширения системы.

При реализации описанной в предыдущей главе модели go показал себя с сильной стороны. Модель параллелизма CSP оказалась крайне удобной для реализации многоагентной системы как с точки зрения внутренней архитектуры, так и с точки зрения получившегося API.

Реализованная как часть симулятора многоагентная среда получилась независимой от остального кода и может быть использована как база для написания других симуляторов.

При реализации для симулятора компонентов CoreUnit, Core и Worker было выявлено множество небольших технических аспектов, касающихся синхронизации работы компонентов. Причем эти аспекты будут актуальны и при написании реализаций этих компонентов для реальных условий. Этот факт подтверждает оправданность идеи, что необходимо тестировать не только сам планировщик, но и принципы взаимодействия остальных частей системы.

Эти тонкие моменты выражены в документации, которой сопровожден код и самом коде. В качестве примера приведу один из таких моментов: "если два последовательных запроса в систему оперируют одним датасетом или контейнером, то второй запрос должен быть послан в систему строго после окончания обработки первого Control Unit'ами". Конкретно этот пример связан с тем, что глобальное состояние системы должно полностью отражать изменения, привнесенные обработкой первого

запроса. Иначе может оказаться, что мы, например, дважды запланируем загрузку одного и того же датасета на один Worker.

6.1. Установка и использование evergrid-go

Реализованный программный продукт называется evergrid-go и представляет из себя CLI-приложение (Command Line Interface).

Ниже описан наиболее простой сценарий для установки разработанного симулятора и примеры его использования.

Для начала нам надо установить go и настроить его окружение. Инструкции есть на сайте языка программирования go: https://golang.org/doc/install. Упрощенный способ для установки на Ubuntu: https://github.com/golang/go/wiki/Ubuntu

Когда окружение настроено скачиваем и устанавлваем актуальную версию evergrid-go следующей командой:

```
go get github.com/ffloyd/evergid-go
```

После этого генерируем сценарий работы (со стандартными настройками) в папке simdata:

```
$GOPATH/bin/evergrid-go gendata test simdata
```

Поменять параметры генерации можно через опции, список которых можно увидеть с помощью команды:

```
$GOPATH/bin/evergrid-go gendata -h
```

Далее можно запустить симуляции для различных типов планировщика:

```
$GOPATH/bin/evergrid-go simulator simdata/test.yaml -s random
$GOPATH/bin/evergrid-go simulator simdata/test.yaml -s naivefast
$GOPATH/bin/evergrid-go simulator simdata/test.yaml -s naivecheap
```

6.2. Пример работы

Ниже приведены примеры результатов работы симуляции на одинаковом сценарии для всех трех тривиальных реализаций планировщика.

Показаны только последние строчки логов, которые содержат статистику использования воркеров.

Как будет видно, планировщики дают результаты соответсвующие их целям. Naive Fast имеет наименьшее значение ”total calculating ticks”, Naive Cheap имеет наименьшее значение ”total money spent”, а все три запуска random scheduler оказались существенно хуже по этим параметрам.

Данные результаты представлены в ознакомительных целях, а сами планировщики представляют из себя довольно наивные реализации, которые не предназначены для работы в реальных условиях.

6.2.1. Random Scheduler

При запросе на загрузку датасета выбирается один случайный воркер и датасет загружается на него.

При запросе на выполнение эксперимента - эксперимент запускается на воркере с уже загруженным датасетом.

Так как работа планировщика рандомизирована приведены результаты трех запусков. Остальные два планировщика выдают одинаковый результат при условии одинакового сценария.

Листинг 6.1. Random Scheduler: запуск 1

```
Total uploading ticks          simulation=big value=269
Total building ticks           simulation=big value=82
Total calculating ticks        simulation=big value=5336
Total money spent              simulation=big value=4158.3458
```

Листинг 6.2. Random Scheduler: запуск 2

```
Total uploading ticks          simulation=big value=269
Total building ticks           simulation=big value=83
Total calculating ticks        simulation=big value=4450
Total money spent              simulation=big value=2670.4092
```

Листинг 6.3. Random Scheduler: запуск 3

```
Total uploading ticks          simulation=big value=269
Total building ticks           simulation=big value=85
Total calculating ticks        simulation=big value=5353
Total money spent              simulation=big value=3992.3984
```

6.2.2. Naive Fast Scheduler

При запросе на загрузку датасета выбираются три наиболее производительных воркера с размером очереди меньше пяти, либо просто три наиболее производительных воркера.

При запросе на выполнение эксперимента - среди воркеров с загруженным (или с запланированным для загрузки) датасетом выбирается наиболее быстрый с размером очереди меньше 5-и, либо просто наиболее быстрый из воркеров с минимальной очередью, если это невозможно.

Листинг 6.4. Naive Fast Scheduler

```
Total uploading ticks          simulation=big value=807
Total building ticks           simulation=big value=55
Total calculating ticks        simulation=big value=1611
Total money spent              simulation=big value=1361.8439
```

6.2.3. Naive Cheap Scheduler

При запросе на загрузку датасета выбираются три наиболее дешевых воркера с размером очереди меньше пяти, либо просто три наиболее дешевых воркера.

Сравнивается цена за одну минуту работы.

При запросе на выполнение эксперимента - среди воркеров с загруженным (или с запланированным для загрузки) датасетом выбирается наиболее дешевый с размером очереди меньше 5-и, либо просто наиболее дешевый из воркеров с минимальной очередью, если это невозможно.

Листинг 6.5. Naive Cheap Scheduler

```
Total uploading ticks          simulation=big value=807
Total building ticks           simulation=big value=63
Total calculating ticks        simulation=big value=2574
Total money spent              simulation=big value=497.4759
```

Заключение

В данной работе на основе общего описания системы Evergrid была сформулирована ее архитектура. На основе этой архитектуры была реализована среда ее моделирования на языке программирования go. Соответсвенно, результатами работы являются:

- описание архитектуры
- рекомендации по ее реализации
- симулятор этой архитектуры для анализа работы планировщиков
- подробная техническая документация этого симулятора, посвященная его использованию и модификации

Архитектура приведена в тексте диплома, а продокументированный исходный код симулятора доступен на github: https://github.com/ffloyd/evergrid-go.

Сформулированная архитектура призвана облегчить разработку системы в будущем, а написание своего симулятора в противовес использованию готовых решений обосновано совокупностью следующих факторов:

- ни одно из популярных решений в своем исходном виде не поддерживает необходимую архитектуру, а их модификация оказывается не менее сложной задачей чем написание своей реализации
- малоизвестные решения рискованно использовать
- не требуется столь сложный механизм симуляции, который используется в популярных решениях
- go удобнее для написания и поддержки подобных систем, чем C/C++ или Java
- отсутствие подобных симуляторов реализованных на go
- возможность интероперабельности с реальной системой увеличивает качество симуляции и самого продукта
- важность проверки работоспособности принципов работы предложенной архитектуры

При написании симулятора учитывалось, что его будут расширять и модифицировать, поэтому особое внимание уделялось изолированности и заменяемости его компонентов.

Список литературы

1. Sakaki T., Okazaki M., Matsuo Y. Earthquake Shakes Twitter Users: Real-time Event Detection by Social Sensors // Proceedings of the 19th International Conference on World Wide Web. WWW '10. New York, NY, USA: ACM, 2010. P. 851–860. URL: http://doi.acm.org/10.1145/1772690.1772777.
2. Tisue S., Wilensky U. Netlogo: A simple environment for modeling complexity // International conference on complex systems / Boston, MA. Vol. 21. 2004.
3. Casanova H. Simgrid: A toolkit for the simulation of application scheduling // Cluster computing and the grid, 2001. proceedings. first ieee/acm international symposium on / IEEE. 2001. P. 430–437.
4. Buyya R., Murshed M. GridSim: a toolkit for the modeling and simulation of distributed resource management and scheduling for Grid computing // Concurrency and Computation: Practice and Experience. 2002. Vol. 14, no. 13-15. P. 1175–1220. URL: http://dx.doi.org/10.1002/cpe.710.
5. Klusáček D., Rudová H. Alea 2: Job Scheduling Simulator // Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques. SIMUTools '10. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2010. P. 61:1–61:10. URL: http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8722.