МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РЕСПУБЛИКИ КАЗАХСТАН

СӘТБАЕВ УНИВЕРСИТЕТІ

Институт кибернетики и информационных технологий
Кафедра «Программной инженерии»

Курманшады Фараби

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

На соискание академической степени магистра

Название диссертации	Разработка менеджера ресурсов на основе Torque с использованием Akka.NET и расширением Akka.Cluster
Направление подготовки	6М100200 – Системы информационной безопасности

Научный руководитель
доктор тех. наук, проф.

_____ Ч. Кеншимов
«__»_____ 2020 г.

Оппонент
Доктор. тех. наук, профессор
_____ Н.Т. Дузбаев
«__» _____ 2020 г.

Нормоконтроль
лектор
_____ Ж.М.Алибиева
«__» _____ 2020 г.

ДОПУЩЕН К ЗАЩИТЕ
Заведующий кафедрой ПИ
Доктор PhD
_____ М. Тұрдалыұлы
«__»_____ 2020 г.

Алматы 2020

Институт кибернетики и информационных технологий

Кафедра Программная инженерия

Специальность: 6М100200 – Системы информационной безопасности

УТВЕРЖДАЮ
Заведующий кафедрой ПИ
Доктор PhD
_____ М. Тұрдалыұлы
«___» _____ 2020 г.

ЗАДАНИЕ
на выполнение магистерской диссертации

магистранту Курманшады Фараби

Тема диссертации: «Разработка менеджера ресурсов на основе Torque с использованием Akka.NET и расширением Akka.Cluster»

Срок сдачи законченной диссертации «___» _____

Исходные данные к магистерской диссертации. Дана задача на исследование существующих решений в области кластеризации вычислительных ресурсов. Поставленные цели и задачи диссертационной работы направлены на создание прототипа менеджера ресурсов для обработки больших документов.

Перечень подлежащих разработке в магистерской диссертации вопросов или краткое содержание магистерской диссертации: а) определение требований – анализ целей, задач и назначения разработки; б) исследование архитектуры, реализуемого в кластеризации вычислительных ресурсов; в) анализ инструментов и технологий для разработки; г) проектирование способа взаимодействия вычислительных узлов; д) разработка системы на основе модели актора.

ГРАФИК
подготовки магистерской диссертации

Наименование разделов, перечень разрабатываемых вопросов	Сроки представления научному руководителю	Примечание
Раздел 1. Общее описание		
Раздел 2. Функциональность		
Раздел 3. Экспериментальная часть		

Консультации по проекту с указанием относящихся к ним разделов проекта

Раздел	Консультант, (уч. степень, звание)	Сроки	Подпись
Нормоконтроль	Ж.М.Алибиева, Лектор кафедры программная инженерия		

Дата выдачи задания "____"_____ 2020 г.

Заведующий кафедрой _____ М. Тұрдалыұлы

Научный руководитель _____ Ч. Кеншимов

Задание принял к исполнению магистрант _____ Ф. Курманшады

Дата "_____"_____ 2020 г.

АННОТАЦИЯ

Данная работа предназначена для создания базового прототипа менеджера ресурсов.

Разрабатываемое решение позволяет минимизировать время обработки определенных задач, таких как обработка больших файлов

В ходе работы была собрана информация о способе реализации классического менеджера ресурсов

Целью данной работы минимизация времени выполнения определенных задач и с дальнейшим использование его на существующем проекте.

Данный проект был реализован с помощью технологии Akka.NET на платформе .NET.

АҢДАТПА

Бұл жұмыс ресурстар менеджерінің базалық түрдегі прототипін құруға арналған.

Әзірленген шешім үлкен файлдарды өңдеудің уақытын азайту сияқты, белгілі бір тапсырмаларды өңдеу уақытын азайтуға мүмкіндік береді

Жұмыс барысында классикалық ресурстар менеджерін жүзеге асыру тәсілі туралы ақпарат жиналды

Бұл жұмыстың мақсаты белгілі бір тапсырмалардың орындау уақытын барынша азайту және оны қолданыстағы жобада одан әрі пайдалану.

Бұл жоба мынадай технологиялар арқылы жүзеге асырылды: Akka.NET және .NET платформасы.

SUMMARY

This work is intended for creating a basic prototype of the resource manager.

The developed solution allows you to minimize the processing time for certain tasks, such as processing large files

In the course of this work, we collected information about how to implement the classic resource Manager

The purpose of this work is to minimize the time required to complete certain tasks and then use it on an existing project.

This project was implemented with the help of technology Akka.NET on the platform .NET.

СОДЕРЖАНИЕ

Введение		7
1	Общее описание	8
1.1	Цель разработки приложения	8
1.2	Обоснование выбора программного обеспечения	9
1.3	Akka.NET	9
1.3.1	Модель актора	10
2	Функциональность	12
2.1	Функциональные части системы	12
2.2	Задание и ее жизненный цикл	14
2.2.1	Сценарий задания (создание задания)	15
2.3	Планировщик заданий изнутри	18
2.3.1	Алгоритм вычисления приоритетного номер задания	19
2.4	Вычисление в кластерной системе	19
2.5.	Рабочий процесс управления заданиями	22
2.6	Akka.NET как платформу для разработки менеджера ресурсов	23
2.6.1	Проблемы решаемые моделью актора	23
2.6.2	Иллюзия инкапсуляции	24
2.6.3	Иллюзия общей памяти на современных компьютерных архитектурах	27
2.6.4	Иллюзия стека вызовов	28
2.7	Модель актора как один из столпов параллельных распределенных систем	30
2.7.1	Использование передачи сообщений позволяет избежать блокировки и перекрывания	31
2.8	Обоснование выбора Akka.Cluster как библиотеку для решения сетевых проблем	34
2.8.1	Функционал Akka.Cluster	34
2.8.2	Отличие кластеризаций от простого удаленного взаимодействия	35
2.8.3	Организация общения в кластере	36
2.8.4	Формирование кластера	37
3	Экспериментальная часть	42
3.1	Создание архитектуры приложения	42
3.2	Этапы выполнения задачи	43
3.3	Взаимосвязь отдельных частей приложений	45
	Заключение	46
	Список использованной литературы	47

ВВЕДЕНИЕ

Разные IT-компании используют разные языки программирования, библиотеки и фреймворки для проектирования своих приложений. Все они имеют свои достоинства и недочеты. Управление активами, то есть ресурсами - это процесс первоначальный проектирования и распространения ресурсов для достижения пиковой производительности. Актив - это все, что вам требуется для выполнения проблемы или цели - это может быть набор возможностей компании (компьютеры, сеть).

Основное намерение контроля ресурсами - делать немалое с меньшими расходами. Никто не любит распутства, в частности в бизнесе. Контроль ресурсами сосредоточено на рационализации и действенности. Когда вы понимаете, что вам нужно, чтобы выполнить указание быстро, без потери свойства как качества, вы можете успешно намереваться, как лучше использовать эти ресурсы. Продуктивный контроль за ресурсами позволяет избегать «перераспределения» или «несамостоятельности» от ресурсов, получая видение о рабочем грузе сейчас. Это делает компьютерную сеть равномерно распределенной по задачам или заданиям. Построенный опытный образец контроллера ресурсов, позволяет быстро настроить стратегию помощника и его подчиненного «ведущий-ведомый», не прибегая к сторонним или проектным библиотекам. То есть это приложение для решения определенного типа списка задач.

1 Общее описание

1.1 Цель разработки приложения

Организации используют различные языки, библиотеки и фреймворки для разработки своих приложений. Все они имеют свои преимущества и недостатки. Управление ресурсами — это процесс предварительного планирования и распределения ресурсов для достижения максимальной эффективности. Ресурс — это все, что необходимо для выполнения задачи или задания — это может быть набор доступных мощностей компании (компьютеров, связанных между собой сетью) [1].

Основная цель управления ресурсами заключается в том, чтобы делать больше с меньшими затратами. Никто не любит расточительства, особенно в бизнесе. Управление ресурсами сосредоточено на оптимизации и эффективности. Когда вы знаете, что вам нужно, чтобы сделать задание быстро выполняемым и без потери качества, вы можете эффективно планировать оптимальный способ использования этих ресурсов. Эффективное управление ресурсами позволяет избежать "чрезмерного распределения” или "зависимости" ресурсов, получив представление о рабочей нагрузке в данный момент времени. Это делает сеть компьютеров равномерно распределенными по задачам или заданиям. Созданный прототип менеджера ресурсов позволяет быстро настроить паттерн мастера и его подчиненного “master-slave”, не прибегая к сторонним библиотекам или проектам. То есть это легковесное приложение для решения определенного типа списка задач.

Преимущества данного прототипа для распределения задач по кластеру заключается в том, чтобы максимально быстро получить результат от поставленных в очередь задач (job) при этом не теряя в качестве ее исполнения. Используя надстройку паттерна “master-slave”, и переписав под свои нужды малую часть (вид задачи для дробления на мелкие части) под предлогом command & conquer, мы можем получить более быстрый вариант выполнения задач, поставленных на выполнение в очередь. Нет необходимости с нуля писать распределение задач на доступные машины, требуется только написание дробления вашей задачи на мелкие части, чтобы достичь высокой скорости получения результата, от поставленной задачи.

Преимуществом данного приложения является то, что оно само по себе не требует процессорного времени, во время простоя. То есть ваше приложение будет занимать только малую часть памяти, а процессорного времени он не будет занимать вообще, так как простаивает. Это делает его эффективным при

создании менеджера ресурсов. Также в связи использованием технологии Akka.Cluster мы может достичь масштабируемости, не написав при этом дополнительных строк кода, что делает его прекрасным дополнением. Это дает преимущества при отлавливании ошибок, либо тестировании работоспособности задачи. Это дает разработчикам концентрацию на проблемной задаче, а не на проблемах связи в сети или масштабируемостью.

1.2 Обоснование выбора программного обеспечения

Во время разработки автор данного проекта столкнулся со следующими проблемами:

1) Создание библиотеки;
2) Масштабирование библиотеки;
3) Паттерн для обработки задач;
4) Мониторинг выполненных задач;
5) Создание распределенной задачи;

Чтобы решить все эти проблемы, автор данного проекта использовал следующее решение:

1) Akka.NET;
2) Akka.Cluster;
3) Akka.Remote;
4) .NET Core

1.3 Akka.NET

Akka.NET, набор библиотек с открытым начальным кодом для разработки масштабируемых, отказоустойчивых систем, обволакивающие процессорные ядра и сети. Akka позволяет нам концентрироваться на выполнении бизнес-требовании вместо записи низкоуровневого кода для предоставления достоверной манеры, отказоустойчивости и высокой эффективности.

Стандартные методы и общие образцы программирования не значимых важных задач, свойственных разработке систем для нынешних компьютерных архитектур. Чтобы быть удачными на рынке, разделяемые системы должны справляться в окружении, где элементы или составляющие части отказывают без ответа, информации теряются без следа на линии, а сетевое отставание меняется время от времени. Эти вопросы регулярно происходят в внимательно контролируемых средах внутри учреждении обработки записей - тем более в виртуализированных архитектурах.

Чтобы справиться с этими явлениями, Akka.NET обеспечивает:

- Многопоточная манера без использования низкоуровневых конструкций параллелизма, таких как атомары или замки. Вам даже не нужно думать о проблемах прозрачности памяти.
- Видимая дистанционная коммуникация между системами и их элементами. Вам не нужно писать или поддерживать трудный сетевой код.
- Кластеризованная архитектура высокой доступности, которая является эластичной, масштабируется или отключается по требованию.

Все эти особенности достижимый через единую модель программирования: Akka.NET использует модель актора для обеспечения такого уровня абстракции, который способствует написанию корректных параллельных и распределенных систем. Модель актера охватывает большую часть Akka.NET библиотеки, путем предоставления последовательного подхода осознования и использования их. Таким образом, Akka.NET предлагает такую глубину внедрения, которую вы не можете достичь, выбирая библиотеки для урегулирования отдельных трудностей и пытаясь объединять их вместе.

Изучая Akka.NET и его акторную модель, мы получаем доступ к огромному и большому набору инструментов, которые решают комплексные задачи распределенных/параллельных систем в одной модели программирования, где все хорошо и слаженно сочетается.

1.3.1 Модель актора

Параметры нынешних вычислительных сред существенно отличаются от тех, которые применялись при создании предыдущих моделей программирования. Акторы были придуманы полвека назад Карлом Хьюиттом. Но относительно недавно их применимость к задачам нынешних вычислительных систем была общепризнана и продемонстрировала свою оперативность.

Модель актора обеспечивает абстракцию, которая позволяет нам думать о своем коде в условиях коммуникации, не отличаясь от лиц в крупной организации. Ключевая характеристика акторов заключается в том, что они имитируют мир как сущности, имеющие состояние, общающиеся друг с другом с помощью явной передачи сообщений.

Как вычислительные субъекты, акторы обладают этими параметрами:

- Они сотрудничают с асинхронными посланиями вместо вызовов методов
- Отвечая на сообщение, они могут:
    - Творить других (дочерних) актеров
    - Направить сообщения другим актерам

- Приостановить (дочернего) актеров или самих себя

2 Функциональность

2.1 Функциональные части системы

Как известно контроллер ресурсов состоит из планировщика заданий и контроллера ресурсов. Он управляет непосредственным применением вычислительных ресурсов и обеспечивает пользовательский интерфейс для осуществления заданий в больших масштабах на единых средствах. С точки видения клиента, утилизация ресурсов для научных вычислений означает применение системы для презентации и выполнения вычислительных задач в виде заданий и приема результатов после завершения задания. Как говорилось ранее у меня было две части это планировщик заданий и менеджер ресурсов. Так как на предыдущем этапе мы разрабатывали систему только для нашей компании мы не стали разделять эти две понятии, а объединили в одно. Но в будущем если перенести планировщик заданий в отдельный проект или библиотеку это не станет преградой, так как код останется неизменным. На данном этапе я хотел бы остановиться на описании рабочего процесса менеджера ресурсов

s

Рисунок 2.1 Устройство менеджера ресурсов изнутри

Как показано на Рис. 2.1, актор планировщика и менеджера ресурсов (показаны зелеными блоками) имеют по 2 и 3 основных компонента, соответственно: (i) клиентские команды (в красных блоках), которые обеспечивают пользовательский интерфейс для постановки заданий, мониторинга и запуска на узлах. Подробное описание того, как использовать этот интерфейс, приведено в последующих разделах. Обратите внимание, что менеджер ресурсов предоставляет пользовательские команды для отправки заданий и управления ими уже после отправки заданий; (ii) компоненты в желтых блоках, которые установлены и запущены на узле инфраструктуры и недоступны пользователям; (iii) актор система узла (slave) (фиолетовый блок) запускаются на всех вычислительных узлах. Они запускают задания на назначенных узлах и контролируют выполнение заданий и использование ресурсов. В итоге, планировщик и менеджер ресурсов работают вместе для достижения ожидаемого результата

Рабочий процесс пакетной системы (batch system) для менеджмента заданиями и активами лаконично описан ниже в порядке номеров:
① Когда задание передается в систему с помощью интерфейса пользователя, система будет анализировать сценарий задания, чтобы убедиться, что ресурсы, подписанные заданием, нормальные и могут быть доступны. Если ошибки в сценарии задания будут найдены, задание не будет принято и выкинется. Идентификатор задания будет сгенерирован и возвращен клиенту в качестве стандартного вывода команды, если сценарий задания будет принят командой.
② Если задание успешно подано в систему, то запись задания (jobID и другие атрибуты задания) будет сгенерирована и предоставлен в планировщик задач для планирования. На этом этапе возникает задание, имеющее идентификатор и запись в системе для поиска.
③ Актор планировщика добавляет задание в очередь заданий и применяет алгоритм (scheduling algorithm) для определения назначения ресурсов и приоритета выполнения задания.
④ Как только задание было запланировано к запуску, актор планировщика передает информацию обратно главному актору, где происходит подготовка к запуску выполнения с зарезервированными ресурсами. На этом этапе актор планировщика помечает статус задания как “выполняющиеся”.
⑤ После того, как задание запланировано в акторе планировщика и передано главному актору, дальше он (master) даст ресурсы для задания и представит информацию о выполнении задания, а также управление заданием назначенному рабочему узлу (slave).
⑥ На вычислительном узле(узлах) будет управляться выполнение заданий, контролируя использование ресурсов. Когда задание будет успешно выполнено в течение приведенного времени, узел будет держать все выходные данные, направленные в stdout и stderr (стандартные точки направления ошибок), в выходной результат и передаст их главному актору, то есть серверу. Если выполнение задания перевеливает заданное время работы или применение

памяти, узел прекратит выполнение задания и уведомит об этом держателя задания. Когда задание завершено или завершено аварийно, актор узла возвращает статус задания обратно актору сервера, и информация о задании будет освежена.
⑦ Пользователи могут запрашивать информацию с помощью API интерфейса о своей работе и состоянии ресурсов.

2.2 Задание и ее жизненный цикл

В высокопроизводительных системах (HPC) большинство программ выполняется в виде пакетных задач. Задача в пакетной системе — это набор команд, которые будут выполняться в похожих назначенных активах (узлы, память, дисковое пространство, а также другие управляемые лицензии на программное обеспечение и т. д.).). Задача как правило, будет находиться в одном из следующих статусов в течение своего жизненного цикла. На рис. 2.2 показан жизненный цикл задания .

Создание: задания создаются клиентом. Чтобы создать задание, пользователю требуемо будет указать две части: (1) Какие активы требуемый для выполнения задания. Он состоит из спецификации, начинающийся хэштегом, которая сообщает контроллеру активов о ресурсах, запрашиваемых заданием; и (2) Какие команды (вычислительные задачи) будут выполняться в задании. Он состоит из последовательности команд windows command prompt, которые выполняются на запрошенном ресурсе. Эти спецификации остаются в файле, обычно называемом "сценарий задания”. На данном этапе задание находится в своей спецификации и известно только его держателю. Об этом не знает система.

Представление: пользователь может использовать команду из допустимых API, чтобы отправить задание, указанное в файле в пакетную систему. После успешного выполнения команды задание становится объектом в очереди заданий системы. В очереди заданий каждое задание будет иметь свой идентификационный номер, называемый "jobID", и статус. На этом этапе задание находится в пакетной системе, заметим то, что вычислительные задачи уже не могут быть изменены. Если задача должна быть изменена, то оно должно быть удалено и снова отправлена после принятия изменений.

Выполнение: когда ресурсы, запрошенные заданием (заданные в сценарии задания), доступны, задание запускается диспетчером ресурсов на вычислительных узлах. Команды (вычислительные) в спецификации сценария задания будут выполняться последовательном направлении, то есть по одному. Идентификатор JOBID можно использовать для получения информации о задании, такой как номера узлов выполнения, прошедшее время, использование памяти и т. д. На этом этапе пользователь не может изменить запрошенные ресурсы. Если выполнение команд в задании превышает запрошенный ресурс,

например при использовании большего объема памяти чем было указано, чем указано, или при выполнении более длительного времени, чем заданное время работы в спецификации, задание будет прекращено планировщиком и пользователю будет отправлено уведомление. На этом этапе держатель задания может удалить его.

Завершение: когда задачи задания завершаются либо из-за завершения выполнения, либо из-за аварийной ошибки, либо из-за того, что задание превышает запрошенное время работы или объем памяти, задание помечается как завершенное. Стандартный вывод и стандартная ошибка задания будут собраны и отправлены в интерфейс, в который задание отправляется по умолчанию. Контроллер ресурсов заберет ресурсы из выполненного задания и сделает их видимыми для других заданий планирования. На этом этапе задание больше не существует в очереди заданий, и идентификатор задания дальше не будет действительным.

Рисунок 2.2 Жизненный цикл задания

2.2.1 Сценарий задания (создание задания)

Чтобы отправить задание выполнение, держателю сначала необходимо создать файл сценария задания. Сценарий задания обычно может состоять из двух частей: (1) условии ресурсов и (2) условии команд. На рис. 2.3 показан пример файла сценария задания "My_job.qsub" [2].

В строке 1 говорится, что это задание будет обрабатываться в среде, где используется оболочка bash.

С 5 строки и далее, начинаются с хэштега “PBS” и за ними следуют нужные параметры, указывая пользовательские ресурсы. Это ясляется спецификацией ресурсов задания для запроса. Актор проанализирует эти строки и создаст объект с указанной информацией задания в очереди заданий планировщика системы.

Начиная с 12-16 строки — это команды bash для *NIX систем, которые можно воспринимать как сценарий выполнения последовательно с некоторыми объявленными переменными, назначенными самим рабочим узлом, такими как переменные HPC_JOBID и другие. Отметим, то что команды могут пониматься только интерпретатором команд bash, все другие пользовательские команды могут пузырить ошибку.

Myjob.qsub

```
1  #!/bin/bash
2
3  ## Часть 1: Спецификация ресурсов
4
5  #PBS -l nodes=1:ppn=1,walltime=04:00:00      #Запрашиваем 1 узел с 1
6                                               #ядром на 4 часа
7
8
9
10
11 ## Часть 2: Спецификация команд
12
13 cd  Директория                                #сменить директорию
14
15
16 filename.exe                                  #запустить команду
17
```

miro

Рисунок 2.3 Сценарий задания

Часть 1 сценария задания, в общем случае, содержит следующие параметры ресурсов:

- Базовые требования к активам вычислительных узлов: в задании должно указываться параметры нужных ресурсов. Если они не будут указаны, планировщик прибегнет к использованию настроек по умолчанию. Например,

```
#PBS -l nodes=2:ppn=2:gpus=1,mem=20GB,walltime=1:30:00
```

Рисунок 2.4 Требования сценария задания, заданные пользователем

- На примере рис. 2.4 выше запрашивается 2 узла с 2 ядрами и 1 видеокартой GPU на каждый узел, 20 Гб жесткой памяти узла и полтора часа времени выполнения. Значение по умолчанию на рис. 2.5 потребности в ресурсах эквивалентно рисунку ниже. Здесь можно увидеть то, что по умолчанию будет требоваться 1 узел с 1 незанятым ядром и минимум более 700 МБ жесткой памяти на узле. На выполнение заданию дается один час.

```
#PBS -l nodes=1:ppn=1,mem=750MB,walltime=00:01:00
```

Рисунок 2.5 Требования сценария задания по умолчанию

Со всем списком параметров спецификации ресурсов можно ознакомиться в Таблице 1 раздела "Приложение А. Команды программы".

Часть 2 сценария задания, обычно, содержит следующие команды:

- Установление опциональных переменных среды для задания : по умолчанию задание запускается в командной консоли, как указано в первой строке сценария задания. Задание, как указано в сценарии задания, иногда нуждается в дополнительных переменных для правильного запуска, либо нахождения папок и т.д. Некоторые переменные определяются каждым независимым узлом, например HOME, PATH и т. д.
- Команды на выполнения: это самая главная часть задания. Это может быть одна команда для запуска другой программы, или это может быть сценарий, при котором создается рабочий процесс из сложных задач. Эти сведения не входят в границы данного документа.
- Команды для отслеживания состоянии о задании: эта система позволяет отслеживать информацию о состоянии задании и выводить полную картину выполнения в интерфейс клиента. Если задание по какой-либо причине завершается аварийно, то клиент будет осведомлен об этом.

Предупреждение: очень важно, что все строки начинались с #HPS, помещались перед конкретными параметрами, предоставляющими команду для запуска. Любая строка без этого начала будут проигнорированы.

2.2.2 Подача задания на выполнение

Если вы создали сценарий задания, постановка задачи в очередь планировщика может быть выполнена с помощью простой командной строки:

```
$ qsub <your_job_script>
```

Рисунок 2.6 Способ подачи задания в менеджер ресурсов

Здесь на рис. 2.6 <your_job_script> - это имя файла вашего сценария задания.

Как можно наблюдать за своей работой?

При представлении многих задач к менеджеру ресурсов, могут возникнуть трудности при управлений этими заданиями. Здесь предлагается несколько подходов по управлению ими.

- Использовать индивидуальные имена заданий и файлов. При одновременном выполнении нескольких заданий важно установить разные имена заданий и связать имя задания или номер JOBID с выходными результатами, чтобы выходные ошибки не были перепутаны другими заданиями и имели четкое разделение. Использование опции #HPS-N <your_jobname> для назначения имени задания, служит для быстрого нахождения задания в клиентском интерфейсе.
- Если на данный момент выполняются задания, которые могут занять много часов или даже дней, можно рассматривать состояния задач с помощью доступных API системы (далее в документации).

2.3 Планировщик заданий изнутри

Система хранит все отправленные задания в очереди заданий у планировщика. Эти задания будут время от времени перетасовываться и конкурировать за использование ресурсов, управляемых системой. Как происходит сортировка в очереди? Планируются ли задания на основе принципа “первым зашел, первым вышел” или используются другие подходы? Что происходит, если задание находится в выходной части очереди, то есть к его началу, но активы еще не доступны, может ли планировщик отдать на выполнение другие задания, стоящие после него в очереди? Здесь попытаемся объяснить, как работает планировщик изнутри чтобы помочь пользователям быстрее внедриться в систему.

Система использует стандартный, как у многих систем, стратегию к планированию. Он часто будет проверять доступность активов для заданий в началной части очереди и планирует выполнение этих заданий, когда ресурсы будут доступны. В очередях есть такое важное свойство, как приоритетные номера. Приоритетные номера определяют порядок выполнения заданий, в отсортированном порядек по убыванию, в очереди. Когда у задания с наивысшим приоритетом будут доступны запрошенные ресурсы, это задание будет запущено по расписанию, и его статус станет “запущено”. Если ресурсы не могут быть для него доступными в этот момент, планировщик зарезервирует ресурсы для задания, то есть оставляет пустые лазейки и попытается заполнить

пустые, имеющиеся места меньшими заданиями, которые имеют высокий приоритет в очереди заданий. Это очень похоже как заполняется память с помощью планировщика ядра. Приоритетный номер задания часто проверяется и обновляется.

2.3.1 Алгоритм вычисления приоритетного номер задания

Главным и единственным критерием при задании приоритета для задания является запрашиваемые активы.

Запрашиваемые ресурсы: учитывается количество узлов, ядер и объем памяти. Чем больше запрашиваемых активов, тем и ясно будет приоритетный номер. Однако тут есть загвоздка. Более крупные запросы ресурсов также могут занять больше времени простаивания в очереди, так как быстро собрать все запрошенные ресурсы доступными может и не удосужится возможным. Кроме того, даже если задание имеет наивысший приоритет, оно может не быть запущен и выкинут, поскольку запрошенные ресурсы не могут суммарно соответствовать требованиям задания. На этом шаге некоторые более легковесные задания(короткое время работы и малое количество узлов) могут быть переданы к выполнению. Используются конкретные политики рассредотачивания ресурсов и приоритизации заданий, для рационализации использования ресурсов кластера сравнительно задании. Остановимся на одном из них по подробней.

Ценность задания определяет порядок, в котором будет запущено задание в кластере. В расчете приоритета учитывается планировщиком и такой фактор, как время ожидания задания, проведенное в очереди. Этот фактор добавит по одной единице приоритета к каждой задаче, ожидающим еще одну минуту в очереди. Стоит отметить, что задания которые требуют много вычислительных мощностей автоматически выбрасываются из очереди, заранее предупредив клиента.

2.4 Вычисление в кластерной системе

Кластер — это настроенная сеть высококлассных компьютеров (также известных как вычислительные узлы или ноды), контролируемых менеджером ресурсов и планировщиком задач [3]. Вместо того чтобы разрешить клиенту входить на один, заданный ранее, компьютер и свободно выполнять нужные ему задачи, клиенту будет предложено отправляет свои вычисления в виде заданий в кластер. Эскиз на рисунке ниже кратко описывает, как задания управляются главной частью приложения и планируются его товарищем планировщиком, для

выполнения вычислений на рабочих узлах кластера. Кластеры в основном пользуются популярностью в пакетных системах. Пакетные системы (batch system) — это множество компьютеров (сети, системы хранения и т. д.), которые считают что одно целое лучше сумм отдельных машин. Этот подход, проверенный годами пользуется нынче большой популярностью, так как некоторым организациям требуются много мощностей для их потребностей. Некоторые из систем состоят всего лишь из нескольких машин, выполняющих однопроцессорные задания и минимально возможными вариациями видов задач. Другие системы имеют тысячи и тысячи машин, выполняющих разный перечень задании пользователей одновременно. Объединение ресурсов в пакетной системе уменьшает техническое администрирование ресурсов, обеспечивая при этом единообразное представление для пользователей. После грамотной и правильной настройки такие системы удаляют многие неудобства и трудностей с плеч пользователей, связанные с запуском и управлением заданиями, что позволяет повысить производительность использования ресурсов и человеко часов. На рис. 2.7 показан упрощенный вариант менеджера ресурсов. Например, пользователям обычно требуется только указать минимальные требования задания и не нужно знать отдельные имена машин каждого узла, так как система изначально уже обладает такими знаниями. Посредством такого единого абстрактного представления пакетные системы могут выполнять тысячи и тысячи заданий одновременно.

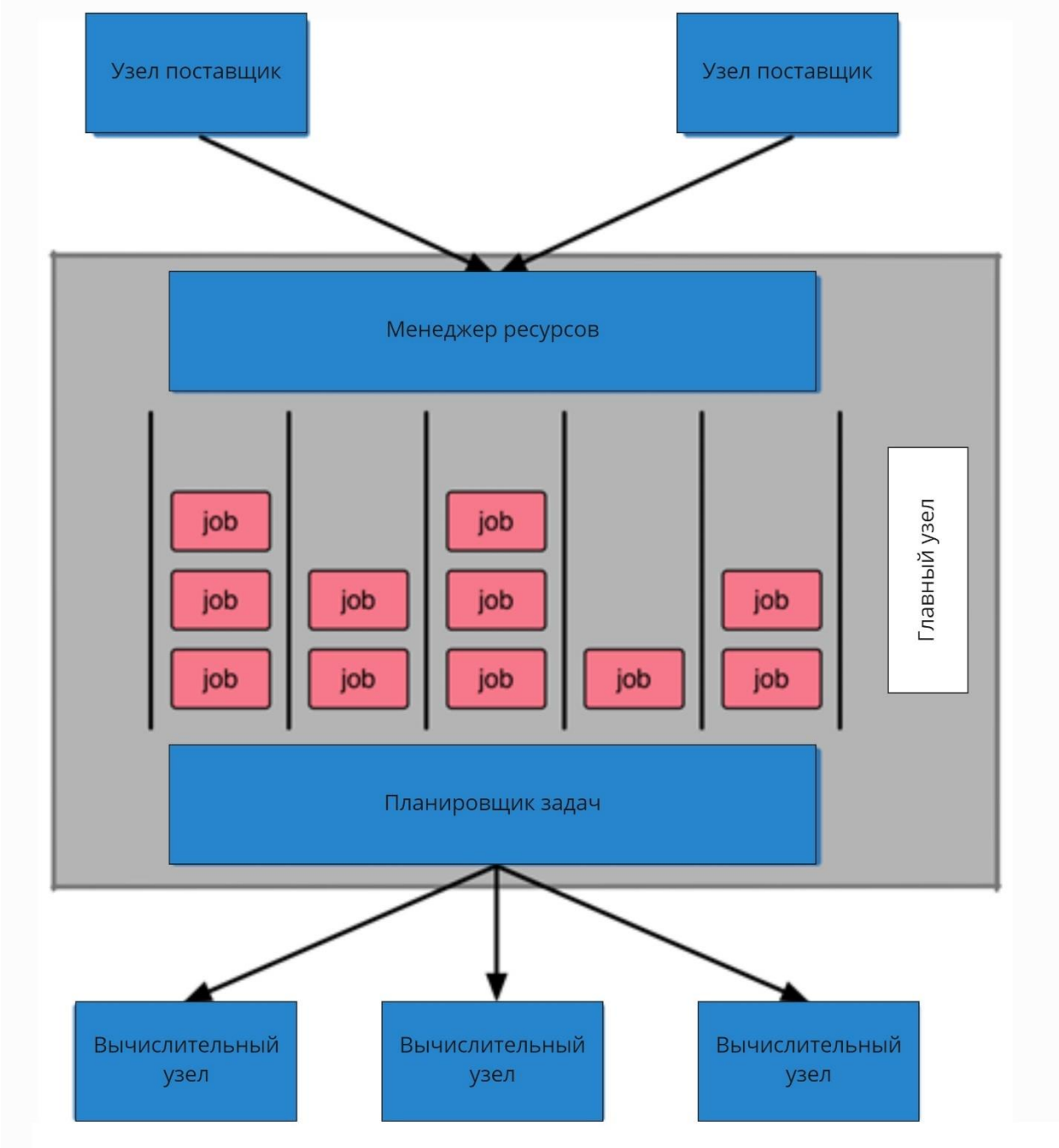

Рисунок 2.7 Упрощенный вид кластера менеджера ресурсов

Задания, передаваемые в кластер с набором условии к активам (например, продолжительность вычислений, количество ядер процессора, объем оперативной памяти и т. д.). Зависимо от условий задания размещаются по-различному в очереди. В обязанности планировщика входит определение приоритетов заданий и размещение их правильным образом вычислительным узлам, на которых выполняются условия заданий. Система также гарантирует отведенные ресурсы для проведения вычислений. Это приводит к тому, что меж разными вычислениями помехи минимизируются, что является залогом к более прогнозируемому времени окончания работы.

2.5. Рабочий процесс управления заданиями

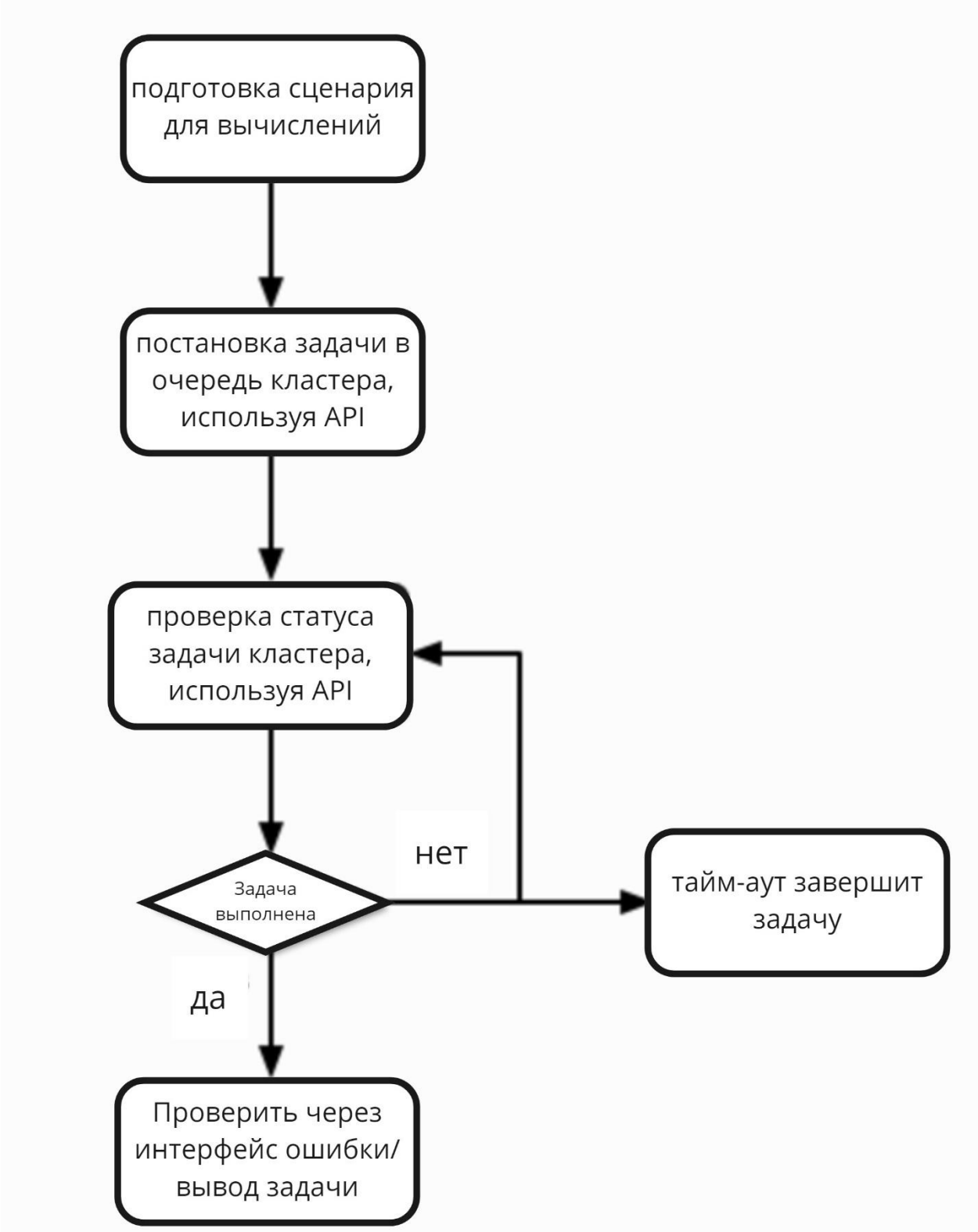

Рисунок 2.8 Общий рабочий процесс выполнения задания

Пакетные системы состоят из двух основных схожих компонентов: (1) главный узел, (2) вычислительные узлы и (3) ресурсы.

- Главный узел- система должна иметь главный узел, на котором выполняется главный актор. Главный узел может быть выделен только для координации и мониторинга задач, либо в зависимости от потребностей заказчиков, он может выполнять роли других компонентов также.
- Вычислительные узлы - вычислительные узлы являются рабочими силами системы. Их роль заключается только в выполнении представленных заданий, так как эти узлы не имеют никакого состояния. Эти узлы могут динамически добавляться, либо удаляться из кластера. На рис. 2.8. показано как происходит выполнение каждого отдельного задания. На каждом вычислительном узле рабочий актор (slave) запускается для выполнения, уничтожения и управления отданными в его распоряжение заданиями. Он взаимодействует с главным актором на главном узле. В зависимости от требований организации вычислительный узел может быть одновременно главным узлом (или более).

2.6 Akka.NET как платформа для разработки менеджера ресурсов

Всем хорошо известно, что традиционные методы программивания не совсем подходят под реалии новых нужд и потребностей людей на сегодняшний день. Ибо разработчиком приходится разрабатывать все более ухищренные методы реализации, спроса заказчиков, так как каждый потребитель на сегодняшний день желает, чтобы приложение было быстрым, надежным и безопасным для клиентских данных. Зачастую разработчики ссылаются на конкретные технологии, которые упрощают заготовку распределенных приложении, но загромождают приложение не нужными для конечной цели объектами. Для разработки таких приложений от разработчика требуется наличие определенных квалификаций и понимании, что налогает его еще больше ответственностью. Решением или замещением этого является модель актора, придуманная еще в 70-х годах прошлого века. Модель актора решает многие недостатки многопоточного программирования и создает хорошие условия разрабатывания распределенных систем. Остановимся по подробней на проблемах, которые решает модель актора.

2.6.1 Проблемы решаемые моделью актора

Akka.NET использует модель актора для преодоления ограничений традиционных моделей объектно-ориентированного программирования и решения уникальных задач высоко-распределенных систем. Чтобы полностью понять, почему необходима модель актора, нужно понять что она помогает

выявить несоответствия между традиционными подходами к программированию и реалиями параллельных и распределенных вычислений. Ниже представлены возникающие в традиционном программировании проблемы [4].

2.6.2 Иллюзия инкапсуляции

Объектно-ориентированное программирование (ООП)-это широко используемая во многих организациях, хорошо знакомая модель программирования. Одним из ее основных столпов является инкапсуляция. Инкапсуляция следует протоколу такому, чтобы внутренние переменные данные объекта не были доступны непосредственно извне; они могут быть изменены только путем вызова набора определенных публичных методов, выставленных на показ вызывающему. Объект отвечает за предоставление безопасных операций, которые обеспечивают инвариантную природу его инкапсулированных данных.

Например, если мы имеем двоичное дерево операции над его реализацией, мы не должны допускать нарушения инварианта упорядочения дерева. Вызывающие потоки будут ожидать, что порядок будет сохранен, и при запросе дерева для определенного куска данных они должны полагаться на это ограничение.

Когда мы анализируем поведение объектов среды выполнения ООП, мы рисуем диаграмму последовательности вызовов, показывающую взаимодействие вызовов методов. Например, как показано на рис. 2.9 (а):

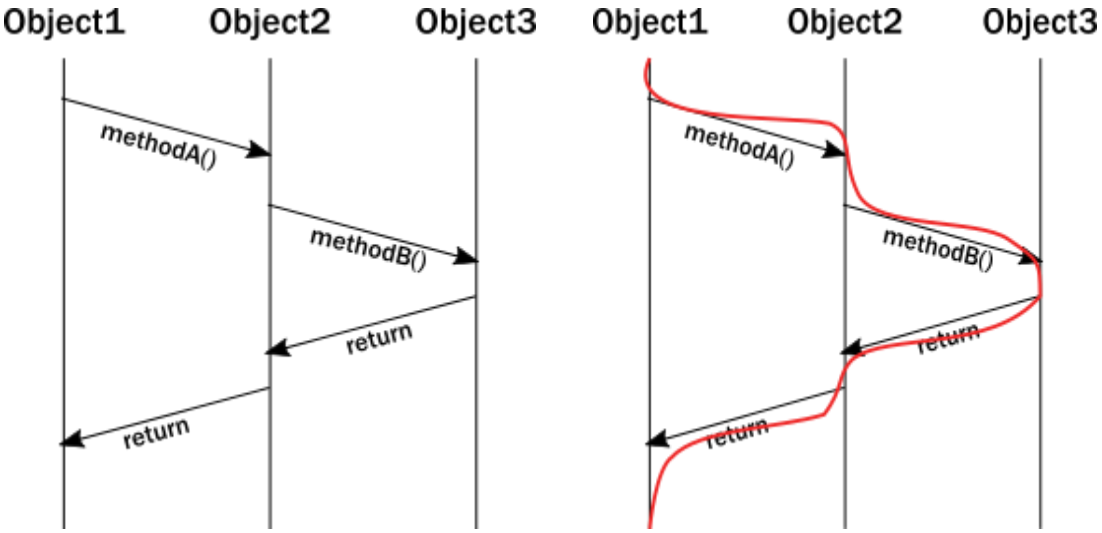

Рисунок 2.9 (а) диаграмма последовательности, (б) выполнения потока вызовов функции

К сожалению, приведенная выше диаграмма не дает точного представления о жизненных линиях, протекающих во время выполнения экземпляров. В действительности поток выполняет все эти вызовы, как показано на рис. 2.9 (б), и чтобы сохранить инвариантность мы должны выполнять остальные вызовы методов в том же потоке, из которого был вызван метод. Дорисовывая диаграмму с потоком выполнения, она выглядит следующим образом (рис. выше). Значение этого уточнения становится ясным, когда вы пытаетесь смоделировать то, что происходит с несколькими потоками. Внезапно наша понятно нарисованная схема становится неадекватной. Проиллюстрируем несколько потоков, обращающихся к одному и тому же экземпляру:

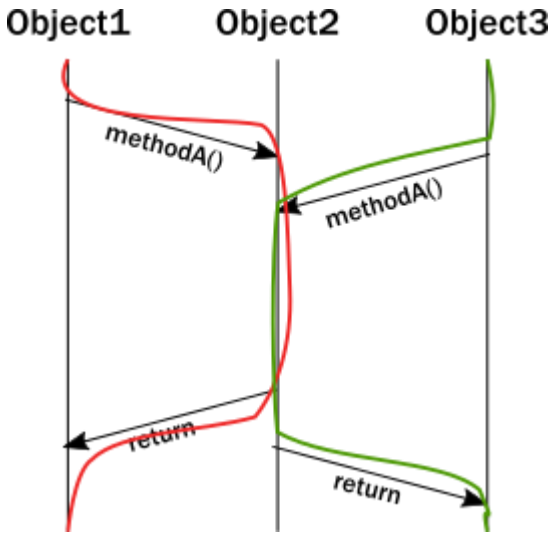

Рисунок 2.10 Обращение двух потоков к одному методу

Существует вариант выполнения, где два потока обращаются к одному и тому же методу (рис. 2.10). К сожалению, в модели инкапсуляции объектов ничего не говорится относительно того, что происходит в этом разделе. Результаты этих двух вызовов могут быть переплетены разными способами, которые исключают сохранность инвариантности переменных или состояния объекта нетронутыми без какой-либо координации, взаимообщения между двумя потоками. А когда этих потоков сотни, представляется что эта проблема усугубляется еще больше.

Традиционный подход к решению этой проблемы состоит в добавлении блокировок или семафоров вокруг этих методов. Это гарантирует, что не более одного потока войдет в метод в любой момент времени. Но этот подход плох тем, что стратегия очень дорогостоящая:

- Блокировки (lock) всерьез ограничивают параллелизм, они довольно дорогостоящи на передовых архитектурах, требуя большущих усилий от операционной системы, для приостановки потока и восстановления его позднее.
- Вызывающий поток ныне заблокирован, в следствие этого он не имеет возможность исполнять практически никакой иной важной работы. В том

числе и в десктопных приложениях это неприемлемо, мы желаем, дабы обращенные к юзеру части приложений (их пользовательский интерфейс) были отзывчивыми, в том числе и когда производится долговременное фоновое поручение. В бэкэнде блокирование-это откровенная расточительность производительности. Можно было бы подумать, что это можно компенсировать запуском новых потоков, но потоки также являются дорогостоящей абстракцией.

- Замки приводят за собой еще одну угрозу: такие как тупики (deadlock).

Эти ситуации приводят к безвыигрышной ситуации:

- Без достаточно большого количества блокировок состояние объекта становится искаженным.
- При наличии большого количества блокировок производительность страдает и очень легко приводит к тупикам (deadlock).

А еще замки (locks) действительно хорошо работают только локально. Когда дело доходит до координации между несколькими машинами, единственным вариантом становится распределенные блокировки (distributed locks). К огорчению, распределенные блокировки на некоторое количество величин наименее эффективны, чем локальные, и как правило накладывают строгие лимитирования на масштабирование. Протоколы распределенных блокировок настоятельно просят нескольких циклов обмена данными по сети меж несколькими машинами, в следствие этого заминка имеет возможность зашкаливает.

В объектно-ориентированных языках мы изредка полагаем о потоках или же линейных путях выполнения в целом. Мы нередко даем для себя систему как сеть экземпляров объектов, которые откликаются на вызовы способов (рис. 2.10 (а)), изменяют свое внутреннее положение, а вслед за тем ведут взаимодействие друг с другом сквозь вызовы способов, продвигая вперед все положение приложения:

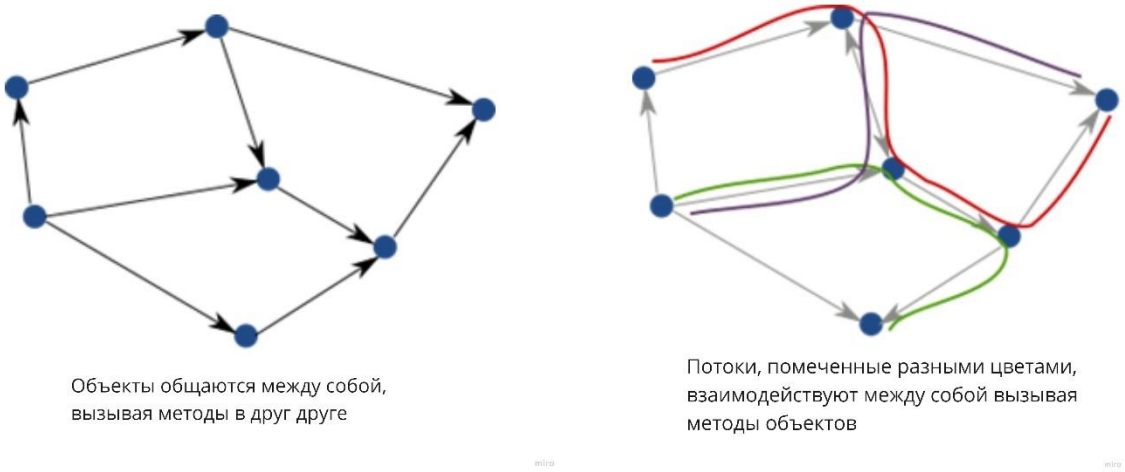

Рисунок 2.10 (а) Общение объектов между собой (б) Выполнение общения с помощью потоков

Впрочем, в многопоточной распределенной среде на самом деле случается то, что потоки "пересекают" данную сеть экземпляров объектов (рис. 2.10 (б)), следуя вызовам методов. И получается потоки — это то, что реально управляет выполнением (рис. выше).

В заключение:

- Объекты имеют все шансы обеспечивать лишь только инкапсуляцию (защиту инвариантности) перед лицом однопоточного доступа, многопоточное выполнение практически всякий раз приводит к повреждению внутреннего состояния. Любой инвариант имеет возможность быть нарушен наличием 2-ух создающих конкуренцию потоков в одном и что же сегменте/разделе кода.
- Хотя блокировки кажутся натуральным средством для поддержания инкапсуляции с несколькими потоками, на практике они неэффективны и проворно приводят к тупикам в всяком использовании реального масштаба.
- Блокировки трудятся локально, поползновения устроить их распределенными есть, но предлагается узкий потенциал для масштабирования.

2.6.3 Иллюзия общей памяти на современных компьютерных архитектурах

Способы программирования 80-90-х лет резюмируют, то что запись во переменную обозначает запись напрямую во ячейку памяти (то что ряд мутит воду, то что локальные переменные имеют все шансы действовать только лишь

во регистрах). На современных архитектурах все немного не так. Упрощенно говоря процессоры пишут переменные в кэш-строки, а не напрямую в память. И большинство этих переменных локальны для данного процессора, то есть ядра. Это означает что записи видны только этому ядру, а всем остальным нет. Нужно сделать усилия, чтобы сделать локальные изменения видимыми для другого ядра и, следовательно, для другого потока, кэш-строка должна быть отправлена в кэш другого ядра.

На примере языке C#, используются маркеры изменчивости (volatile) или атомарные (atomic) оболочки. Это все для того чтобы разработчики учитывали это, в CLR (common language runtime) явно обозначаются места памяти, которые будут совместно использоваться меж потоками. Почему не стоит отмечать все переменные как изменчивые? Это операция по доставке кэш-строк из одного ядра в другую очень дорогостоящая. Это приводит к тому что ядра будут отвлекаться на допольнительную работу по транспортировке этих строк, что неимоверно приводит к появлению узких мест. В конечном результате получается заметное замедление. Для разработчиков, знающих об этой ситуации, выяснение того, какие места памяти должны быть помечены как изменчивые, или какие атомарные структуры использовать, является темным искусством.

В заключение:

- Общей памяти как в прошлых архитектурах не существует. Ядра процессора явно передают друг другу данных, как это бы происходило в сетевых операциях.
- Межпроцессорная коммуникация и сетевая коммуникация имеют почти одинаковую архитектуру. Передача информации через сообщения теперь является нормой, будь то через процессоры ядра в одном компьютере или между несколькими сетевыми компьютерами.
- Лучше сохранить локальное состояние переменных, и вместо его ясно общаться через копии этого же сообщения между параллельными сущностями. Это является более принципиальным и дисциплинированным подходом, нежели скрывать аспект передачи сообщения через переменные, помеченные как общие, или использовать атомарные структуры данных.

2.6.4 Иллюзия стека вызовов

Стек вызовов метода сегодня является обыденным явлением. Но они были изобретены в эпоху, когда параллельное программирование не было столь важным. В то время мультипроцессорные системы не были как таковы распространены. Стеки вызовов не взаимодействуют с несколькими потоками и, следовательно, не моделируют асинхронные цепочки вызовов.

Представим себе такую обычную ситуацию, когда главному потоку нужно делегировать задачу другому потоку, более известному как “фоновый режим”. В этом моменте возникает проблема делегирования, так как не понятно что делать основному потоку, остановиться и сдать завершения, или забыть и идти по своим делам. На практике это действительно означает делегирование задачи совсем другому потоку. Список вызовов строго локальны для только и только одного потока. Традиционно происходит так, что "вызывающий" помещает объект в место памяти, разделяемое рабочим потоком ("вызываемым"), который, в свою очередь, берет его в каком-то периоде событий. Это позволяет потоку "вызывающему" продвигаться дальше и выполнять другие задачи.

Возникает резонный вопрос о том, как "вызывающий" поток может быть уведомлен о завершении задачи? Но самая большая проблема — это когда, задача терпит неудачу, вызывая исключение/ошибку. Как можно распространить ошибку “вызывающему” потоку? К сожалению, никак. Так как инструментов для его уведомления у нас нет. А ошибка будет распространяться на обработчика исключений рабочего потока, полностью игнорируя, кто был фактическим "вызывающим":

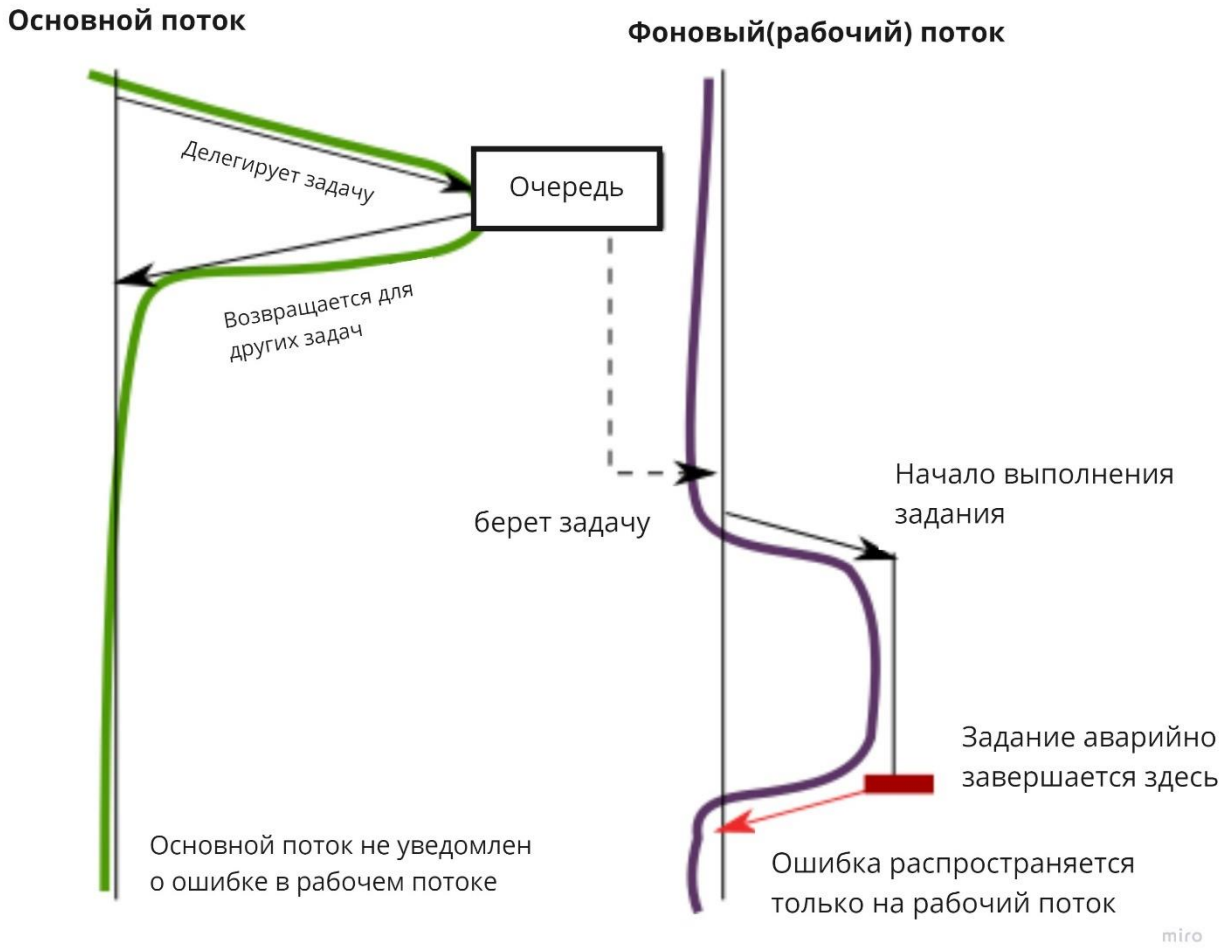

Рисунок 2.11 Делегирование задачи одним потоком другому

Это реальная проблема, с которой каждый разработчик в период своей профессиональной карьеры сталкивался. Как рабочий поток справляется с этой ситуацией, которая показано на рис. 2.11? Рабочий поток не знает, как исправить эту ситуацию, так как он не имеет состояния, либо информации зачем, это задание нужно. Обычно происходит то, что рабочий поток забывает об этой проблеме, и устремляется забрать другую задачу из очереди, то есть из общей памяти. Поток "вызывающий " должен быть каким-то образом уведомлен, но нет никакого стека вызовов, чтобы рассказать ему об исключении. Уведомление о сбое может быть сделано только через боковой канал, например, вводя код ошибки, где "вызывающий" поток ожидает результат, как только он будет готов. Если этого уведомления не будет на месте, то "вызывающий" никогда не получит уведомления о сбое и задача оценится как потерянной! Это очень схоже, когда в сетевых системах теряются пакеты. И могут быть случаи, когда сообщения/запросы могут потеряться/выйти из строя без какого-либо уведомления.

Представим себе еще худшую ситуацию когда рабочий поток столкнулся с ошибкой и оказался в неисправимой ситуации. Например, внутреннее исключение, вызванное ошибкой, пузырится, то есть поднимается до самого начала стека вызовов или до самого корня потока и приводит к его аварийному завершению. Возникает вопрос, кто должен перезапустить обычную работу службы, размещенную в потоке, и как она должна быть восстановлена в привычно хорошее состояние? После восстановления мы столкнемся с неожиданной ситуацией: фактическая задача, над которой в данный момент работал поток, больше не находится в общей памяти, откуда берутся задачи (обычно это очередь). Из-за того, что исключение достигает до самой вершины, состояние задачи полностью теряется. Мы потеряли сообщение, даже несмотря на то, что это локальная связь без участия сети (где потери сообщений следует ожидать всегда).

В заключение:

- Чтобы достигнуть высокого уровня параллелизма и производительности потоки должны делегировать задачи друг другу без блокировки. При таком раскладе дел, параллелизме с помощью делегирования задач (и при этом использовать сетевые/ распределенные вычисления) обработка ошибок требует ввести новые механизмы сигнализации ошибок. Неудачи становятся частью модели этой исследуемой области.
- Параллельные системы с делегированием работы должны обслуживать ошибки и иметь выделенные средства для их устранения. Клиенты таких служб должны помнить, что задачи/сообщения могут быть потеряны во время перезапуска. Даже если потери не произойдет, ответ может быть задержан на неопределенное время из-за ранее поставленных в очередь задач (длинная очередь), задержек из-за сборщика мусора (garbage collector) и т. д. В связи с этим параллельные системы должны

соответствовать на крайние сроки реагирования, ответа в виде тайм-аутов, как и сетевые системы.

2.7 Модель актора как один из столпов параллельных распределенных систем

Как было написано ранее, традиционные методы программирования не могут должным образом удовлетворить потребности современных параллельных и распределенных систем. Хорошие новости заключаются в том что, нам не нужно отбрасывать все, что мы знаем или знали до этого времени. Модель актора устраняет эти недостатки принципиальным образом, позволяя системам вести себя таким образом, который лучше соответствует нашей ментальной, воспринимаемой модели.

В частности, мы хотели бы:

- Обеспечить инкапсуляцию данных, не завися от блокировки (lock).
- Использовать модель кооперативных, общающихся сущностей, реагирующих на сигналы, изменяющих свое внутренне состояние и посылающих друг другу сигналы, чтобы оповещать о нынешних изменениях и вовлекать к работе все приложение.
- Не беспокоиться о выполняемом механизме, который не соответствует нашему мировоззрению.

Модель актера выполняет все выше перечисленные задачи. В следующих разделах описывается, как это сделать.

2.7.1 Использование передачи сообщений позволяет избежать блокировки (locking) и перекрывания (blocking)

Вместо вызова методов объекта актеры посылают друг другу сообщения, в виде копии объектов. При отправке сообщения невозможна передача потока выполнения от отправителя к получателю. Актор может отправить сообщение и продолжить работу без проверки доставления сообщения и его ожидания, то есть без блокировки. Поэтому он может выполнять больше работы.

Актор освобождает контроль над своим исполняемым потоком, когда выполнение метода завершается. Они этом отношении во многом похожи на субъекты, они реагируют на сообщения и возвращают ответ, когда заканчивают обработку текущего сообщения, а в другое время они отдыхают. Таким образом, актеры действительно достигают того уровня исполнения, которое мы себе представляли для объектов:

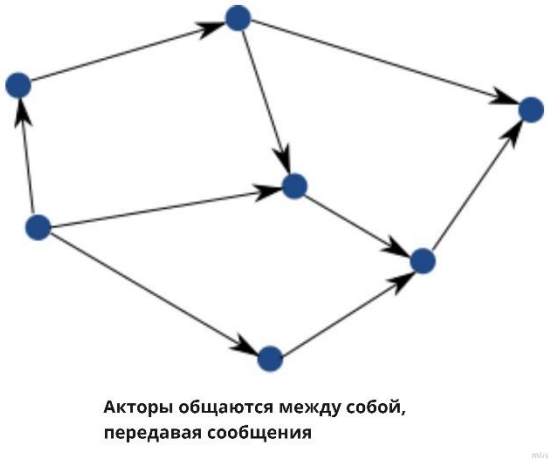

Рисунок 2.12 Общение акторов между собой, похоже на предыдущий рисунок

Один из отличительных свойств передачи сообщений в отличие от вызова метода является то, что сообщения не ожидаются возвращаемые значения. Можно сказать, что сообщения только односторонние. При отправке сообщения, актер делегирует работу другому актеру, и забывает про него, как на рис. 2.12. Как можно было увидеть в разделе “иллюзии стека вызовов”, если отправляющий поток будет ожидать возвращаемое значение, отправляющему субъекту нужно будет либо заблокироваться и ждать, либо выполнить работу другого субъекта в том же потоке. И чтобы это ограничение обойти принимающий субъект передает результаты в ответном сообщении.

Вторая проблема, которую решает модель актора – это восстановление инкапсуляции. Акторы реагируют на сообщения точно так же, как люди "реагируют" на сигналы, посылы других людей, вызвавшие их. Только разница в том что акторы обрабатывают только по одному сообщению, никаких множеств протягивающихся к нему других акторов и ломающих его внутреннее состояние. Акторы действуют независимо от отправителей сообщения и реагируют на входящие сообщения последовательно, по одному. Все как у людей. В то время как каждый актор (субъект) обрабатывает сообщения, отправленные ему последовательно, другие акторы (субъекты) работают одновременно взаимодействуя друг с другом, так что система акторов (субъектов) может обрабатывать столько сообщений одновременно, сколько ядер доступно на машине или на кластере машин. Инвариантность актора может сохраняться без надобности в синхронизации, таких как блокировки. Это происходит автоматически без использования замков (locks):

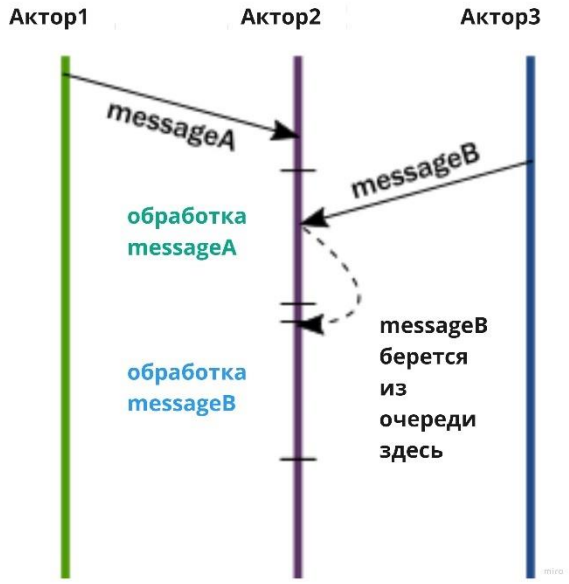

Рисунок 2.13 Обработка актором сообщений

Ниже представлен что происходит, когда актер получает сообщение (рис.2.13):

1. Добавляет сообщение в конец очереди.
2. Актор помечается как готовый к исполнению и планируется процессором, на котором он выполняется. Это все происходит если актор не был запланирован к исполнению процессором.
3. Процессорный планировщик принимает актора и начинает его выполнение.
4. Актор берет сообщение из передней части очереди.
5. Если сообщение требует изменения внутреннего состояния актера, то оно меняется и последующие сообщения будут обрабатываться по другому поведению.
6. У актора нет внеплановых занятий, в то время, когда на почтовом ящике нет сообщений.

Для такого поведения, акторы должны иметь:

- Почтовый ящик (очередь, в которую выстраиваются сообщения).
- Поведение (состояние актора (субъекта), внутренние переменные, разные подходы обработки поведении и т. д.).
- Сообщения (фрагменты данных, представляющие сигнал, аналогичные вызовам методов и их параметрам).
- Среда выполнения, в нашем случае CLR (механизм, который принимает акторов, у которых есть сообщения, чтобы реагировать на них).
- Адрес, по которому другие акторы будут обращаться к этому актору.

Сообщения помещаются в так называемые почтовые ящики акторов, то есть мы дергаем не самих акторов, а только почтовые ящики. Поведение актора описывает, как он будет реагировать на стоящие в очереди сообщения (например, отправляет больше сообщений, замкнуться в себе, пока что-либо е станет доступным и/или изменяет состояние). В обязанности среды выполнения организация пула потоков, чтобы управлять всеми этими действиями полностью прозрачно.

Эта модель решает все перечисленные ранее проблемы:

- Инкапсуляция сохраняется путем сигнализации. То есть нет выше оговоренного стека вызова потоков (метод вызывается самим актором, а не извне).
- Отпадает необходимость в замках (locks). Изменение внутреннего состояния актора возможно только с помощью сообщений, которые обрабатываются по одному, устраняя гонку потоков при попытке сохранить инвариантность.
- Когда не используются замки (locks), и отправители не блокируются, что очень хорошо. Миллионы акторов могут спокойно сосуществовать и быть эффективно запланированы на дюжине потоков, достигая полного потенциала современных процессоров, то есть полностью напрягая процессор от простаивания. Делегирование задач-это естественный режим работы акторов.
- Состояние акторов локально, а не публично, изменения и данные распространяются через сообщения, что подходит тому, как на самом деле работает современная иерархия памяти. Это означает то что происходит передача только тех кэш-строк, которые содержат данные с сообщении. При этом сохраняется локальное состояние и данные, кэшированные в исходном ядре. Та же самая модель точно соответствует удаленной связи, где состояние хранится в оперативной памяти машин и изменения / данные распространяются по сети в виде пакетов.

2.8 Обоснование выбора Akka.Cluster как библиотеку для решения сетевых проблем

Далее мы будем пользоваться словом Akka.Cluster как кластер. Под словом кластер подразумеватся отказоустойчивая эластичная, децентрализованная сеть, состоящая из многих Akka.NET приложении без единой точки отказа или какого-либо узкого места. Akka.Cluster — это модуль, который дает нам возможность создавать такие приложения.

2.8.1 Функционал Akka.Cluster

Кластер добавляет ниже описанные возможности в систему, (таб. 1).

- Позволяет легко создавать и организовать сеть компьютеров для .NET приложении.
- Позволяет узлам автоматически обнаруживать новые подключенные узлы и автоматически удалять недоступные без каких-либо изменений конфигурации, повторяемся, без изменений конфигурации
- Отдельные узлы могут подписаться на уведомления об изменениях доступности узлов в кластере
- Вводит понятие "ролей", чтобы отличить различные NET приложения внутри кластера

Таблица 1 – Преимущества кластера

Преимущества	Определение
Отказоустойчивость	кластеры восстанавливаются после сбоев, то есть ошибок (особенно сетевые сбои) на ура.
Эластичность	Так как кластеры по своей природе похожи на человеческую природу, они эластичны и могут масштабироваться вверх/вниз по мере необходимости
Децентрализованность	возможно одновременное выполнение нескольких одинаковых реплик/повторов заданного состояния микросервиса или части приложения во всем кластере
Отсутствие единой точки отказа / узкого места	несколько узлов могут обслуживать запросы, увеличивая пропускную способность и отказоустойчивость

2.8.2 Отличие кластеризаций от простого удаленного взаимодействия

Кластер — это слой абстракции, который стоит над Akka.Remote далее как транспорт, что позволяет использовать удаленное взаимодействие для каждой определенной разработчиками структуры: а именно в нашем случае кластеров приложений, в таблице 2 представлены часто используемые термины. Ниже. Под капотом, транспорт используется кластером. Обычно транспорт используется

само по себе в сценариях, которые не требуют эластичности и отказоустойчивости.

Таблица 2 – Термины и их определения

Термины	Определение
Узел (node)	логический элемент кластера. Он определяется адресом, по которому он доступен (имя хоста:порт)
Кластер	набор узлов, присоединенных через службу членства, образующую ячеистую сеть. Несколько .NET приложении могут быть частью, например, одного кластера
Сплетни (gossip)	базовые сообщения, строящие сам кластер.
Лидер (leader)	один узел в кластере, который решает добавление / удаление узлов из кластера
Роль (role)	именованная ответственность или одно целое приложение в кластере. Кластер может иметь несколько .NET приложения в нем, каждый соответственно со своей ролью. Узел может существовать одновременно в ноль или больше ролях.
Сходимость (convergence)	когда кворум (простое большинство) сообщений сплетен согласует изменение состояния члена кластера (например, выбирает лидера кластера)
Сеяный узел (seed node)	это хорошо известная точка контакта, с которой новый узел должен связаться, чтобы присоединиться к кластеру. Сеяные узлы функционируют как механизм обнаружения сервиса кластера

2.8.3 Организация общения в кластере

Это самая важная концепция внутри Akka кластере. Именно так, как показано ниже на рисунке узлы присоединяются к кластеру и покидать их без каких-либо изменений конфигурации. Существует в Akka кластере понятие как сплетни. Хотя на первый взгляд это может послышаться как нечто плохая вещь, но она помогает системе надежно и бесперебойно работать. Сплетни — это непрерывный поток сообщений, которые передаются между узлами кластера, обновляя информационный список о состоянии у каждого члена кластера. Когда узел начнет присоединятся к кластеру, он должен сначала связаться с одним из

своих ранее указанных в настройках сеяных узлов. В начале когда узел только поднимается могут приходить ошибочные сообщения, но это нормально. После того, как узел сможет подключиться к одному сеяному узлу, он начнет получать понемногу сообщения сплетен, содержащие информацию о других членах кластера. В конце концов узел будет видеть полную картину о кластере.

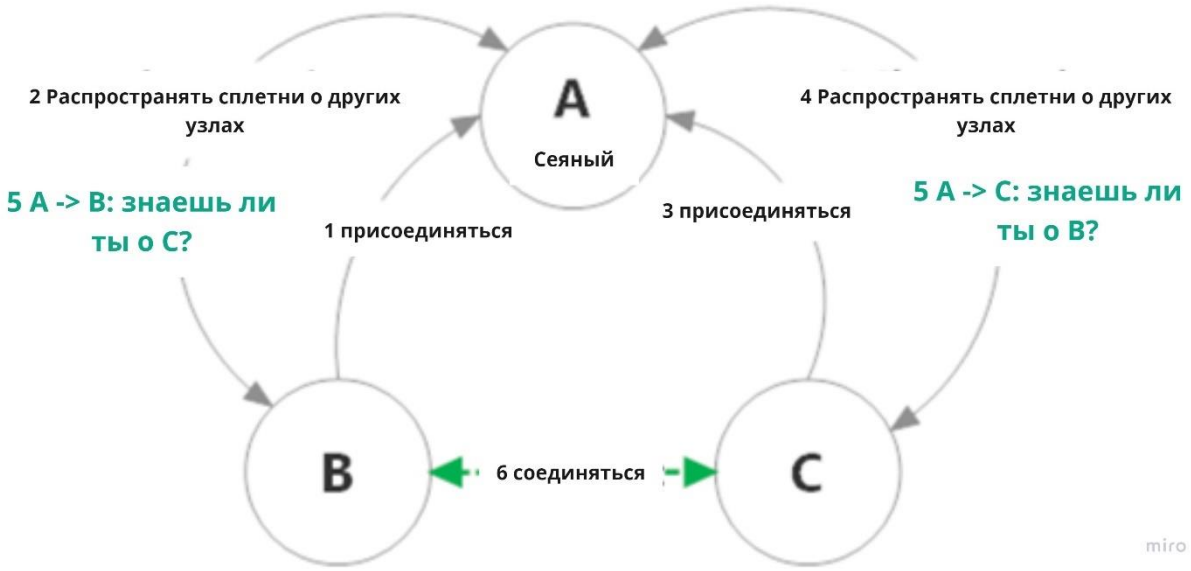

Рисунок 2.14 Организация общения в кластере

В приведенном выше примере (рис. 2.14):

1. B связывается со своим настроенным начальным заданным узлом A и просит присоединиться к кластеру.
2. A помечает B как члена кластера и начинает делиться сплетнями с B о других узлах в кластере, но в данный момент нет никаких других узлов, связанных между собой.
3. C контактирует с A и просит присоединиться к кластеру.
4. A приветствует C в кластере и начинает делиться информацией о сплетнях с узлом C.
5. B и C уведомляются друг о друге узлом A.
6. B и c соединяются друг с другом и устанавливают связь.

Сообщения о сплетнях будут регулярно появляться во время жизни кластера с течением времени всякий раз, когда происходит какое-либо изменение статуса члена кластера, например, когда узел присоединяется к кластеру, покидает кластер, становится недоступным для других узлов и т. д.

2.8.4 Формирование кластера

Вот как выглядит процесс присоединения узла к кластеру:

Кластеры изначально состоят из двух отдельных частей:

1. **Сеяные узлы** - узлы, которые находятся в хорошо известных местах в сети
2. **Обычные узлы** - узлы, начальные местоположения которых неизвестны, и эти узлы контактируют с сеяными узлами, чтобы сформировать кластер.

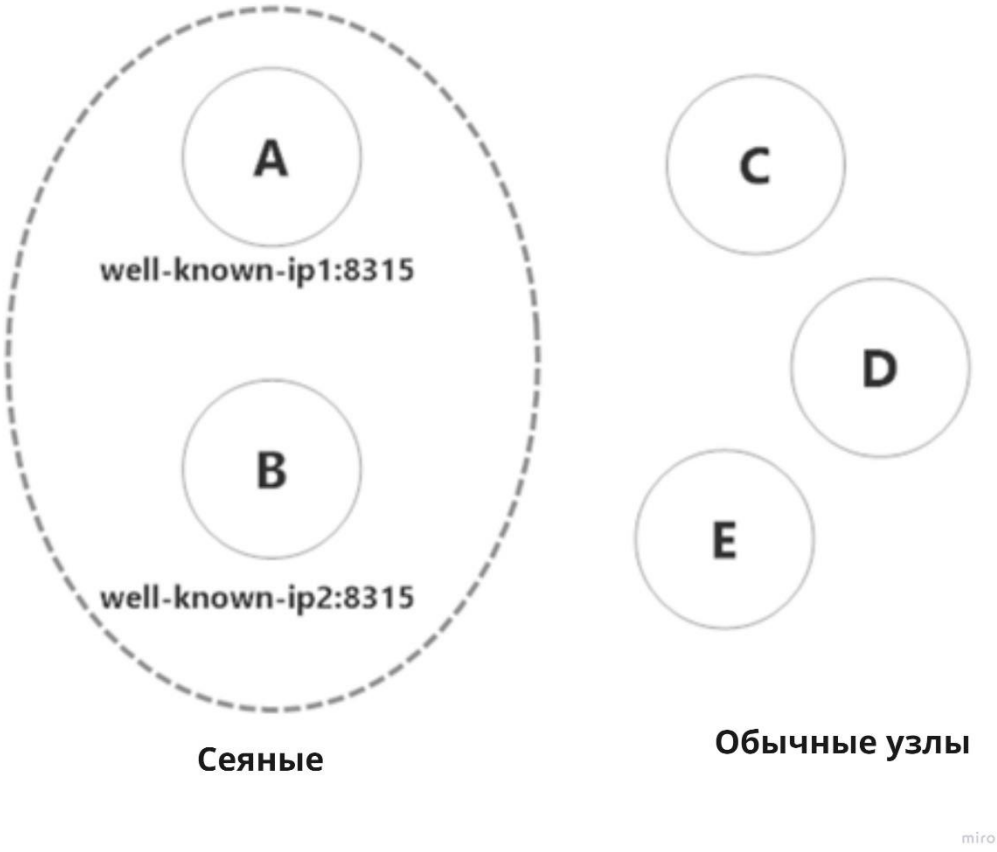

Рисунок 2.15 Начальное состояние кластера (Развертывание 5 узлов)

На рис. 2.15 A и B оба являются сеяными узлами и прослушивают комбинацию IP-адреса:порта, которые изначально задаются в конфигурации C, D и E. A и B уже изначально знают местоположение друг друга, поэтому они могут уже могут общаться друг с другом изначально.

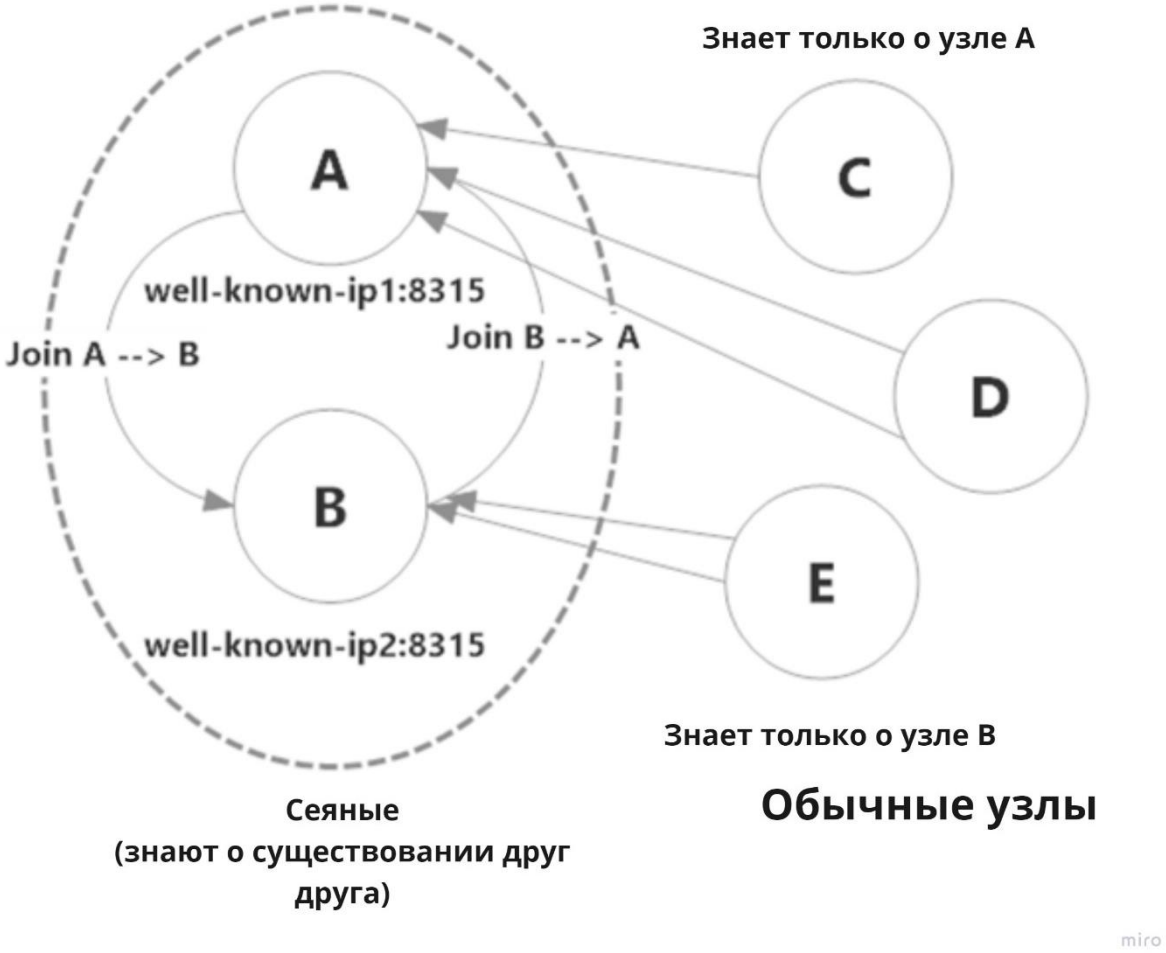

Рисунок 2.16 Присоединение к кластеру (Каждый стремится присоединиться к сеяному узлу)

Как показано на рис. 2.16 все узлы изначально пытаются подключиться к исходному узлу - в этом случае узлы будут настроены следующим образом:

- E знает, как связаться с B.
- C и D знают, как связаться с A.
- A и Б знают, как связаться друг с другом.

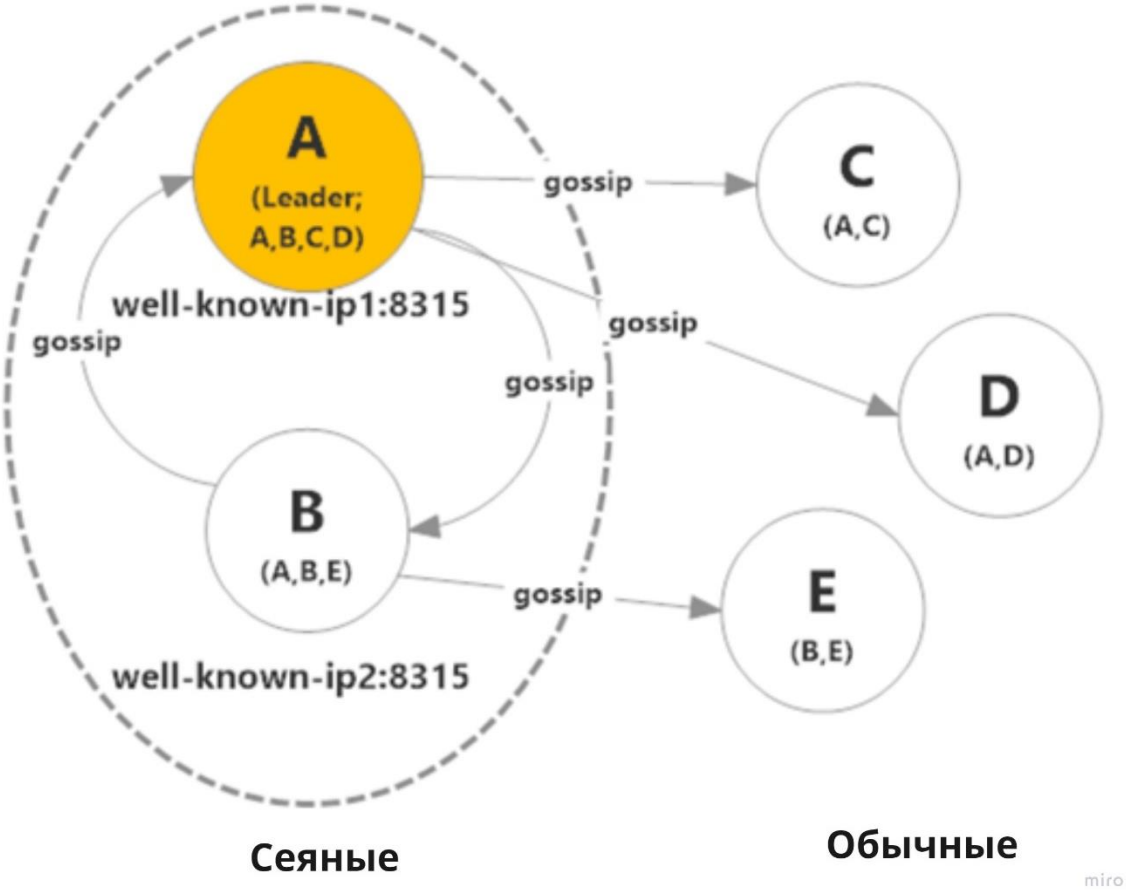

Рисунок 2.17 Лидер избран, помечание узлов как доступные (сплетни начинаются)

В процессе первоначального контакта между узлами внутри кластера будет избран лидер (рис. 2.17). В этом случае узел A выбирается в качестве лидера кластера. A начнет помечать узлы как доступные “поднявшиеся”, начиная с узлов, о которых он знает: A,B, C и D. A еще не знает о узле E, поэтому он не помечен как существующий “поднявшийся”. Сплетничающая информация о принадлежности к кластеру начнет распространяться на все доступные узлы, и все узлы начнут соединяться друг с другом, образуя ячеистую сеть.

Рисунок 2.18 Сплетни распространяются, кольцо сформировано (Коммуникация установлена между обычными узлами)

После того как сплетня успеет распространиться по всем узлам и лидер отметит всех как "работающими", каждый узел будет соединен с каждым другим узлом, и сформируется полноценный рабочий кластер (рис. 2.18). Теперь каждый узел может участвовать в любых определенных пользователем ранее межкластерных операциях.

3 Экспериментальная часть

3.1 Создание архитектуры приложения

Как уже писалось ранее, для создания менеджера ресурсов необходимо некая архитектура, которая максимизирует пропускную способность и количество выполняемых задач. Для решения этой проблемы, мною было выбрано классическая архитектура по образцу “master-slave”. Этот архитектурный стиль, проверенный годами использования, гарантирует прочную взаимосвязь между главным и дочерними узлами. Но отличие данного приложения от других, то что этот подход реализованный с использованием актор модели упрощает разработку данного типа приложения. Разработчику больше не надо писать много кода для того, чтобы построить распределенную систему. Стиль, который вносит модель актора как раз и предназначен для разработки таких систем.

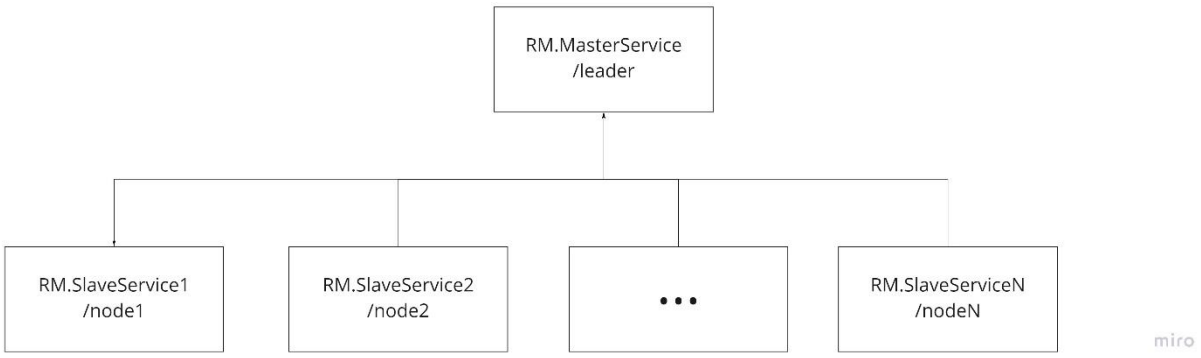

Рисунок 3.1 Архитектура менеджера ресурсов на основе Akka.NET

На первом рисунке 3.1 можно увидеть, как устроен “главный” узел, который управляет “подчиненными” узлами с помощью расширения Akka.Remote. Акка.Remote — это самый мощный из всех этих дополнительных библиотек, так как именно он дает возможность создавать систему актора через многих процессов в компьютерной сети. Таким образом наши главные и дочерние узлы без проблем могут общаться через интернет узел. На рисунке показано что главный узел может общаться со многими подчиненными узлами. И если какой-нибудь узел начнет аварийно завершаться, то главный узел об этом узнает и скорректирует свой список подчиненных узлов. Таким образом мы достигаем того что мы можем эластично расширяться или уменьшаться при необходимости увеличить или ограничить используемые ресурсы. Еще один хороший функционал который представлен в Akka.Remote из коробки, это то что оно дает нам возможность общаться с другими акторами локально так и удаленно. То есть нам не нужно городить много кода чтобы общаться с удаленными узлами, мы

будем обращаться к ним как к локальным переменным, хотя они и будут исполняться в другом процессе на другом узле. Этот хороший функционал называется прозрачностью местоположения.

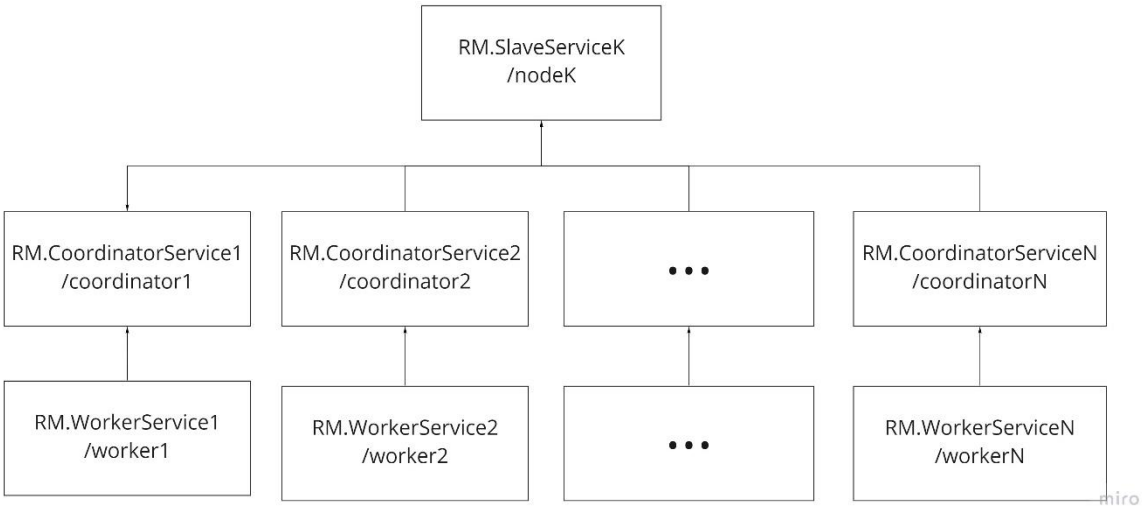

Рисунок 3.2 Архитектура вычислительного узла на основе Akka.NET

На втором рисунке 3.2 виден как устроена архитектура одного подчиненного узла. Здесь можно увидеть, что у каждого “рабочего” актора имеет надзиратель в виде “координирующего” актора. Задается резонный вопрос почему создается надзиратель для каждого рабочего актора, который запускает непосредственно задачу на выполнение, ведь мы не даем приложению упасть. Дело в том что не всегда бывает возможным предвидеть все возможные ситуации, обстоятельства, да и через координатор можно в будущем распараллеливать задачу на малые куски. Достигается это с помощью так называемых в Akka.NET роутеров. Роутер – это так называемый канал который перенапляет задачу на выполнение актором по некоторым критериям которые могут быть описаны разработчиком или можно воспользоваться уже имеющими решениями. Здесь вы можете заметить что на одном узле могут создаваться много акторов координаторов и рабочих. Дело в том что акторы сами по себе легковесные и не занимают процессорного времени если на почте у актора не имеют сообщения. Так что узел на котором производятся такие операции не страшны.

3.2 Этапы выполнения задачи

В этой системе я хотел добиться устойчивости и надежности выполнения задач таким образом, чтобы они не могли навредить всей системе таким образом, что система могла бы отключиться, либо пополниться до грани задачами и упасть. Для этого я применяю подход тайм-аутов (рис. 3.3). Тайм-ауты хороши тем, что, когда задача выполняется слишком долго, чем на самом деле ей надо, либо использует очень много ресурсов узла, мы можем высвободить узел от этой задачи, не вовлекая в этот процесс человеческий ресурс. Таким образом система у нас становится самовосстанавливающейся. Это у нас первая сторона выполнения задачи. На второй стороне выполнения задач могут возникнуть неожиданные ошибки, не предусмотренные пользователем. Эти ошибки мы собираем через канал, который перенаправит ошибки и выходные данные в результат выполнения задачи. Подведя итоги, можно сказать, что мы элегантно собираем ошибки и выходные данные и подносим его для дальнейших решений путем сообщения пользователю. И так, мы разобрались что у нас задача проходит такие этапы, которые описаны на рисунке ниже. Здесь есть нюанс в том что приложение было сделано чтобы не завершаться аварийно, так как не было видно при каких ситуациях могут возникнуть непредсказуемые обстоятельства. Но модель актора разрешает такое поведение.

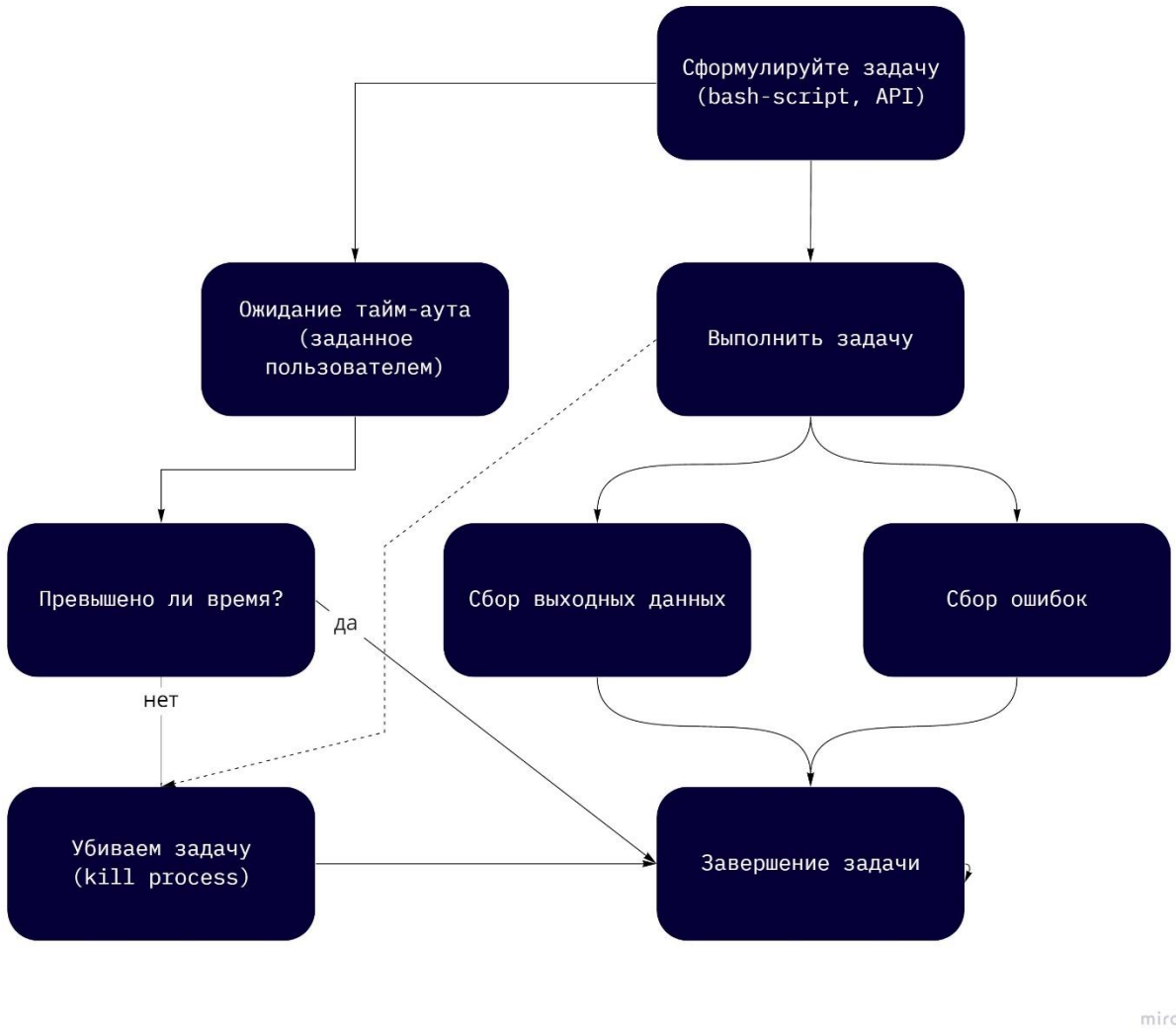

Рисунок 3.3 Рабочий процесс задания на основе Akka.NET

3.3 Взаимосвязь отдельных частей приложений

Так как было решение создать кластер, а как известно оно состоит из нескольких или многих частей. Это приводит к тому, что каждый узел имеет свой адрес и роль в решении общей проблемы. Вдобавок к этому все узлы нужно обслуживать, эксплуатировать, смотреть за состоянием отдельных узлов. На рисунке 3.4 ниже можно увидеть взаимосвязь отдельных частей системы. Роль веба состоит в том, чтобы мониторить и ставить задачи для координаторов. Так как “вэб” является одновременно и планировщиком задач, то в его обязанности входит и балансировка нагрузки на подчиненных узлов, а также осмотр на существование конкретного узла. В свою очередь в обязанность подчиненного

узла входит мониторинг выполнения задачи и его отмены при превышении тайм-аута, заданного пользователем системы.

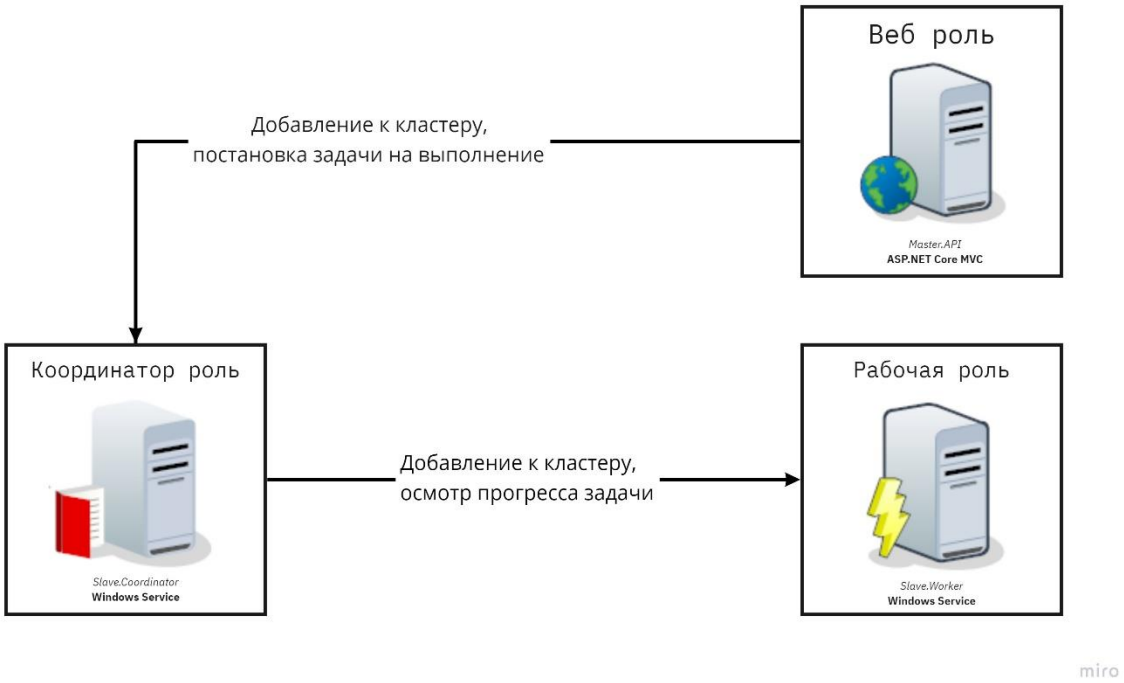

Рисунок 3.4 Архитектура взаимодействии разных узлов на основе Akka.NET

ЗАКЛЮЧЕНИЕ

В ходе выполнения данной работы был рассмотрен один из вариации реализации менеджера ресурсов на основе модели актора и технологии Akka.NET. Опубликовали базовую версию работы в репозитории на github.com. В общем и целом, можно сказать, что модель актора хорошо справляется с задачей менеджера ресурсов. Грамотная и эластичная настройка узлов позволяет удовлетворять нужды потребителей. В неиспользуемое время эти же самые узлы можно отключить в целях экономии ресурсов.

Создание менеджера ресурсов на основе Akka.NET, смотрится очень вдохновляющим. Akka.NET актеры чрезвычайно недороги в создании и уничтожении, поэтому для разработчиков обычно сотни тысяч или миллионы из них работают одновременно в каждом процессе. Akka.NET акторы отвечают за управление бизнес-состоянием или выполнение команд, которые возникают, когда субъект получает сообщение. Одним из основных атрибутов актора в модели актора является то, что он может изменять свое поведение между сообщениями, которые он обрабатывает. Эта возможность позволяет вам делать всевозможные интересные вещи, такие как создание конечных автоматов или изменение того, как ваши актеры обрабатывают сообщения, основанные на других сообщениях, которые они получили.

С помощью модели актора можно достичь полноценного использования потенциала, то есть полностью загрузить полезной нагрузкой процессорное время всех доступных ядер одного рассматриваемого узла. А дополнительная масштабируемость и эластичность системы может удовлетворять нужды потребителей, не написав при этом ни одной дополнительной строчки кода.

Актуальность исследования состоит в том чтобы применить новые подходы для решения старых проблем, как бы смотря в новой точки зрения.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1.Управление ресурсами и планировщик заданий https://wiki.hpcc.msu.edu/display/hpccdocs/Resource+Managment+and+Job+Scheduler?src=search

2. Планирование заданий:

https://wiki.hpcc.msu.edu/display/hpccdocs/Scheduling+Jobs?src=search

3. TORQUE Resource Manager Adminstrator Guide 5.0.1:

http://docs.adaptivecomputing.com/torque/5-0-1/torqueAdminGuide-5.0.1.pdf

4. Akka.NET против Kafka, RabbitMQ и другие системы обмена сообщениями. Аарон Станнард, Февраль 25, 2019