МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РЕСПУБЛИКИ КАЗАХСТАН

САТБАЕВ УНИВЕРСИТЕТІ

Институт кибернетики и информационных технологий
Кафедра «Программной инженерии»

Бубликов Владислав Олегович

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

На соискание академической степени магистра

Название диссертации	Разработка системы по поиску тематической информации с использованием модели акторов для обеспечения масштабируемости, отказоустойчивости и параллельности вычислений
Направление подготовки	6М100200 – Системы информационной безопасности

Научный руководитель
канд. тех. наук
_____ А.А. Мухамедгалиев
«__» _____ 2020 г.

Оппонент
Доктор PhD, ассистент
профессор кафедры
_____ А. Богданчиков
«__» _____ 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 Бубликову Владиславу Олеговичу

Тема диссертации: «Разработка системы по поиску тематической информации с использованием модели акторов для обеспечения масштабируемости, отказоустойчивости и параллельности вычислений»

Срок сдачи законченной диссертации «___» _____

Исходные данные к магистерской диссертации. Даны перечень интернет-ресурсов и набор ключевых слов. Поставленные и цели задачи диссертационной работы направлены на разработку масштабируемой, отказоустойчивой и распределенной системы, основанной на модели акторов.

Перечень подлежащих разработке в магистерской диссертации вопросов или краткое содержание магистерской диссертации:

а) определение требований – анализ целей, задач, проблем и назначения разработки;

б) поиск и исследование возможных решений;

в) применение многопоточности и библиотеки TPL Dataflow к задачам системы;

г) проектирование и разработка системы с использованием модели акторов и фреймворка Akka.NET;

д) формирование сравнительного анализа между разработанными приложениями.

Рекомендуемая основная литература:

1 Клеппман М. Высоконагруженные приложения. Программирование, масштабирование, поддержка. – СПб.: Питер, 2018. – 640 с.: ил. – (Серия «Бестселлеры O’Reilly»).

2 Бёрнс Б. Распределенные системы. Паттерны проектирования. – СПб.: Питер, 2019. – 224 с.

3 Раймонд Рестенбург, Роб Баккер, Роб Уильямс. Akka в действии / пер. с англ. А.Н. Киселев – М.:ДМК Пресс,2018. – 522с.: ил.

ГРАФИК
подготовки магистерской диссертации

Наименование разделов, перечень разрабатываемых вопросов	Сроки представления научному руководителю	Примечание
Раздел 1. Постановка задачи системы и её проблемы		
Раздел 2. Решение задач и проблем системы		
Раздел 3. Разработка приложения. Применение многопоточности и библиотеки TPL Dataflow		
Раздел 4. Фреймворк Akka.NET. Применение Akka.NET в приложении		
Раздел 5. Сравнение разработанных приложений		

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 _____ А.А. Мухамедгалиев

Задание принял к исполнению магистрант _____ В.О. Бубликов

Дата "_____"_____ 2020 г.

АННОТАЦИЯ

В данной магистерской диссертации была разработана система по поиску тематической информации с использованием модели акторов, обеспечив при этом масштабируемость и отказоустойчивость. В задачи системы входили: поиск необходимой информации и файлов, формирование файлов и их обработка. В процессе работы были реализованы также другие приложения, с помощью которых удалось повысить производительность, но достаточно нужный уровень не был достигнут, поэтому использование модели акторов и инструментов, которые обеспечивают актороподобный стиль программирования, стало необходимым. Были рассмотрены подробно модель акторов и фреймворк Akka.NET, основанный на модели акторов.

В работе был использован фреймворк Akka.NET, библиотеки и модули которого позволили добиться успешного построения параллельной и распределенной системы. Производительность в сравнении с первым приложением выросла в разы. Добиться нужной производительности удалось с помощью использования нескольких серверов, также при необходимости существует возможность без особых усилий добавлять и другие сервера для совместной обработки данных.

В процессе работы были изучены вопросы конкурентности, многопоточности, параллельного программирования, распределенных систем. Было выполнено сравнение производительности работы всех приложений.

АҢДАТПА

Бұл магистрлік диссертацияда акторлар моделін қолдана отырып, тақырыптық ақпаратты іздеу жүйесі әзірленді. Жүйенің міндеттеріне: қажетті ақпарат пен файлдарды іздеу, файлдарды қалыптастыру және оларды өңдеу кіреді. Жұмыс барысында өнімділікті арттыра алатын басқа да қосымшалар іске асырылды, бірақ жеткілікті қажетті деңгейге қол жеткізілмеген, сондықтан акторлар мен құралдардың модельдерін қолдану қажет болды. Акторлар моделіне негізделген Akka.NET акторлар мен фреймворк моделіне толық қарастырылды.

Жұмыста Akka.NET фреймворк қолданылды, кітапханалар мен модульдер параллельді және бөлінген жүйені сәтті құруға мүмкіндік береді. Бірінші қосымшамен салыстырғанда өнімділік бірнеше есе өсті. Бірнеше серверлерді пайдалану арқылы қажетті өнімділікке қол жеткізу мүмкін болды, сондай-ақ, қажет болған жағдайда деректерді бірлескен өңдеу үшін басқа серверлерді қосуға болады.

Жұмыс барысында бәсекелестік, көп нүктелік, параллель бағдарламалау, бөлінген жүйелер мәселелері зерттелді. Барлық қосымшалардың жұмыс өнімділігін салыстыру жүргізілді.

ANNOTATION

In this master's thesis, a system for searching for thematic information using the actor model was developed, while ensuring scalability and fault tolerance. The system's tasks included searching for the necessary information and files, creating files and processing them. In the process, other applications were also implemented that improved performance, but the desired level was not reached, so the use of the actor model and tools that provide an actor-like programming style became necessary. The actor model and framework were discussed in detail Akka.NET, based on the actor model.

The framework was used in the work Akka.NET, whose libraries and modules allowed for the successful construction of a parallel and distributed system. Performance in comparison with the first application has increased significantly. You can achieve the desired performance by using multiple servers, and if necessary, you can easily add other servers for joint data processing.

In the course of work, the issues of concurrency, multithreading, parallel programming, and distributed systems were studied. The performance of all applications was compared.

СОДЕРЖАНИЕ

	ВВЕДЕНИЕ	8
1	Постановка задачи системы и её проблемы	10
1.1	Задачи системы	10
1.2	Проблемы системы	10
2	Решение задач и проблем системы	12
2.1	Масштабирование системы	12
2.2	Микросервисная архитектура	14
2.3	Распределённая система	16
2.4	Конкурентность. Параллельное и многопоточное программирование	17
3	Однопоточное приложение	20
4	Многопоточное приложение	22
5	Приложение с использованием TPL Dataflow	24
6	Модель акторов. Сравнение библиотек на платформе .NET	29
6.1	Модель акторов	29
6.2	Сравнение библиотек на платформе .NET для обеспечения актороподобного стиля программирования	31
7	Фреймворк Akka.NET	35
7.1	Введение в Akka.NET	35
7.2	Библиотеки и модули Akka.NET	35
7.3	Архитектура Akka.NET	37
7.4	Использование Akka.NET, сценарии развертывания	39
7.5	Системы акторов в Akka.NET	40
7.6	Акторы в Akka.NET	41
7.7	Ссылки акторов, пути и адреса	43
7.8	Почтовые ящики	45
7.9	Доставка сообщений	46
7.10	Диспетчеры	48
7.11	Маршрутизаторы	48
7.12	Надзор в Akka.NET	52
7.13	Конфигурация в Akka.NET и тестирование систем акторов	54
7.14	Akka.Remote	55
7.15	Akka.Cluster	58
8	Применение Akka.NET в приложении	61
9	Сравнение производительности приложений	63
	ЗАКЛЮЧЕНИЕ	65
	СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ	67

ВВЕДЕНИЕ

Целью магистерской диссертации является разработка системы по поиску тематической информации с использованием модели акторов, для того чтобы увеличить производительность за счёт масштабирования и параллельности вычислений, а также обеспечить отказоустойчивость. Главной проблемой системы является обработка большого объёма данных за небольшое количество времени.

Модель акторов, хоть она и была впервые опубликована несколько десятилетий назад, актуальности не теряет и сейчас. Актуальность использования модели акторов, обусловлена тем, что с каждым днём происходит стремительное увеличение данных, вместе с тем появляются мощные компьютеры, которые сочетают в себе возможности для решения множества задач, они имеют многоядерные процессоры, которые позволяют использовать многопоточность и обеспечивать параллельную обработку данных. При правильных настройках и оптимизации, это даёт увеличение производительности в разы. В мире огромное количество информации, чтобы обработать хотя бы маленькую часть из них даже мощных компьютеров недостаточно, нужно изменять концепцию программирования.

И в данном случае обычное однопоточное приложение не сможет решить текущую задачу, точнее сможет, но за огромное и недопустимое время. Поэтому необходимо изменять модель программирования в сторону конкурентных и распределенных приложений. Использование модели акторов необходимо в данном случае, чтобы система могла без каких-либо проблем увеличивать свой потенциал для разных объёмов задач, являлась отказоустойчивой системой, а также обеспечила грамотную последовательность взаимодействий между процессами, координацию ресурсов, которые делят процессы.

Существует множество языков программирования, которые ориентированы, поддерживают или используют модель акторов. Но также в мире существует множество реализаций в виде библиотек и фреймворков для обеспечения актороподобного стиля программирования на языках, у которых нет встроенных акторов. Так как в организации стеком разработки является .NET, необходимо было выбрать одну из библиотек или фреймворков, реализованных для данной платформы и выбор пал на фреймворк Akka.NET, который позволяет создавать параллельные, распределенные и отказоустойчивые приложения в .NET.

В работе будут рассмотрены различные понятия, такие как конкурентность, асинхронное, реактивное, параллельное и многопоточное программирование. Будет восстановлена поэтапно эволюция системы: как приложение от однопоточного переформировалось в многопоточное, а далее использовалась библиотека TPL Dataflow, но как оказалось и этого было недостаточно для решения задачи, поэтому было решено создать приложение

использующее фреймворк Akka.NET и его модуль Akka.Remote, для обеспечения работы нескольких серверов. С каждым разом процесс перестроения системы усложнялся, а вместе с ним и увеличивалась кодовая база приложения.

В процессе работы будут подробно рассмотрены модель акторов, фреймворк Akka.NET, который и основан на данной модели.

1 Постановка задачи системы и её проблемы

1.1 Задачи системы

Трудно представить, но всего лишь пару десятков лет назад, вычислительная мощность нынешних обычных компьютеров казалась нечто фантастическим для того времени, логично предположить, что и задачи, которые ставились раньше перед разработчиками, соответствовали техническому состоянию того времени. Если говорить о нынешнем времени, то сейчас даже смартфоны, различные девайсы наделены большой мощностью, процессорами, которые могут выполнять сложные вычислительные задачи и работы, не говоря уже и о компьютерах. Компьютеры эволюционировали очень быстро, вместе с этим изменялись, дополнялись концепции программирования, к примеру, компьютеры, в свою очередь, стали многопроцессорными, имея множество ядер, машины повлияли на саму разработку, построение архитектуры систем.

Итак, данное вступление было написано не просто так, данная тема будет рассматриваться и раскрываться дальше подробнее в следующих разделах.

Ниже представлены основные задачи системы:

– Поиск информации по определённым ключевым словам в нескольких интернет-ресурсах.

– Формирование файлов в нескольких форматах на базе найденной информации.

– Осуществление соответствующего перевода и сохранение файлов на нескольких языках.

– Выделение ключевых слов, которые определены в базе данных, в сформированных файлах.

Все файлы будут храниться на сервере, но также вся информация и непосредственно данные будут сохраняться в базу данных.

1.2 Проблемы системы

Задачи, которые были поставлены, осуществить реально, хоть и такая работа с файлами является трудоёмкой, но тут кроются нюансы, которые в корне меняют суть задачи и добавляют ей сложности, давайте рассмотрим подробнее и поймём, какие проблемы могут возникнуть в системе.

Во-первых, неизвестно точное количество файлов, которые будут сформированы, но в теории это десятки, а то и сотни тысяч файлов.

Во-вторых, существует несколько интернет-ресурсов, по которым будет выполняться поиск данных и будут формироваться файлы, и они соответственно разные, то есть поиск будет индивидуальным.

В-третьих, изначально обработать все файлы нужно за короткий промежуток времени, так как информации накоплено большое количество, это и актуальная информация, и архивные данные.

В-четвёртых, нужно отметить, что сравнительно все операции занимают небольшое количество времени, кроме самого сложного и затратного: непосредственного формирования файлов, переводе содержимого и выделения в них ключевых слов, а также необходимо учесть то, что файлы могут быть различных размеров и объёмов.

В-пятых, количество ключевых слов насчитывается более десятка тысяч, а теперь представим: мы должны за небольшое количество времени найти в файле, имеющем сотню страниц, и отметить десять тысяч слов. Задача, мягко говоря, непростая.

Все эти дополнения вносят кардинальные коррективы в разработке и проектировании, архитектуре системы. Почему данные нюансы так повлияют на систему?

Главная проблема в производительности, такие задачи в таких масштабах выполнить за небольшое количество времени очень нетривиальное задание, поэтому обычные подходы к программированию и разработке здесь не помогут, необходимо искать различного рода возможные решения, что и будет проводиться в следующих разделах.

Если же вернуться к самому приложению, забегая немного вперёд, то оно изначально было однопоточным неконкурентным, имея при этом проблемы в производительности, которые предполагались выше и конечно, было бы неверным оставлять всё в таком положении и не применять разные инструменты, которые бы позволили решить проблемы, также нужно отметить, что машина на которой развёртывается приложение является многоядерным, что уже говорит нам о дальнейшем использовании данного преимущества, и в данном случае это может дать в будущем большие положительные изменения, естественно при умелом и правильном использовании.

2 Решение задач и проблем системы

2.1 Масштабирование системы

Как уже писалось в предыдущем разделе, чтобы решить проблемы с производительностью необходимо по максимуму использовать ресурсы сервера, задействовать процессоры.

После всевозможных оптимизаций, которые повлияли на время обработки файлов, записи в базу данных и других затратных операций,

произошло сокращение времени обработки, но сути проблемы не изменило: до сих нужно обработать колоссальное количество файлов.

Какие существуют пути возможного решения? Пожалуй, главным ответом будет масштабирование системы.

Масштабируемость означает наличие стратегий поддержания хорошей производительности даже в случае роста нагрузки. В масштабируемой системе есть возможность добавлять вычислительные мощности в целях сохранения надежной работы системы под высокой нагрузкой [1].

Очевидно, то, что и в данной ситуации необходимо искать пути масштабирования системы. Но прежде, чем продолжить следует отметить, что раньше программисты использовали один компьютер как вычислительный процессор и этого на самом деле хватало, часто оптимизировать ничего не приходилось, потому что тактовая частота на процессорах увеличивалась вдвое, но в какой-то момент тактовая частота перестала увеличиваться, а в свою очередь удваиваться стало количество процессорных ядер. Закон Мура, который можно трактовать так, что мощность вычислительных процессоров увеличивается экспоненциально каждые два года, спустя длительное время перестал действовать. Это произошло в середине 2000-х, когда тактовая частота зависла на отметке в районе 3600 МГц, и после этого количество ядер стало удваиваться каждые два года. И теперь возникло явное понимание того, что писать код так как раньше нельзя, потому что, если оставлять всё также, то в данном случае, в процессе работы программы, составленной как раньше, один процессор будет работать, а другие попросту будут простаивать.

Существуют следующие виды масштабирования:

– Вертикальное масштабирование.

– Горизонтальное масштабирование.

Вертикальное масштабирование – это увеличение производительности сервера, с помощью замены или добавления более мощных и быстрых технологий в рамках одного сервера. Наращивание мощности системы происходит при использовании потоков. На рисунке 1 изображен сервер, в котором могут быть увеличены характеристики, представляющий вертикальное масштабирование.

Рисунок 1 – Вертикальное масштабирование

Инструменты, позволяющие масштабироваться вертикально:

– Parallel Linq – запросы Linq, которые проектируются для параллельного выполнения.

– TPL – библиотека параллельных задач.

– Использование потоков.

Но всё же главным недостатком данного вида масштабирования – это то, что один сервер может не справиться с нагрузкой, которая будет в системе. Данный недостаток можно исправить другим видом масштабирования.

Горизонтальное масштабирование – это разделение системы на компоненты, которые распределены на множестве серверов, а также увеличение количества серверов, которые выполняют одни функции для параллельной работы, но это очень трудоёмкие процессы: разделение приложения, настройка взаимодействия компонентов. На рисунке 2 изображены несколько серверов, которые представляют горизонтальное масштабирование за счёт увеличения количества серверов.

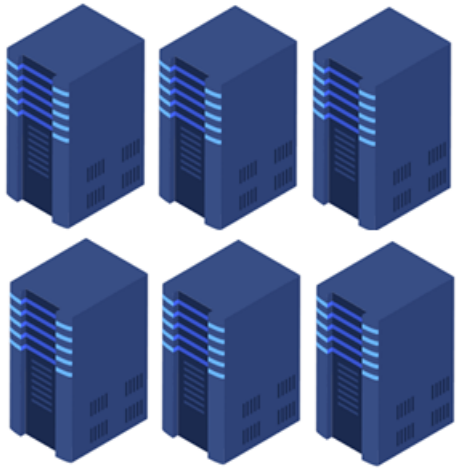

Рисунок 2 – Горизонтальное масштабирование

Средства, которые позволяют масштабироваться горизонтально:

– WCF – технология, которая позволяет разрабатывать распределенные (сервис-ориентированные) системы.

– Web API – инструмент, который позволяет создавать различные веб-службы.

– ServiceStack – фреймворк для создания веб-сервисов и веб-приложений.

– MSMQ – реализация очереди сообщений.

– RabbitMQ – программный брокер сообщений.

Очень часто сравнивают вертикальное масштабирование – переход на более мощную машину – и горизонтальное масштабирование – распределение

нагрузки по нескольким меньшим машинам. Распределение нагрузки по нескольким машинам известно также под названием архитектуры, не предусматривающей разделения ресурсов. Системы, способные работать на отдельной машине, обычно проще, а высококлассные машины бывают весьма недешевые, так что при высокой рабочей нагрузке часто нельзя избежать горизонтального масштабирования [1].

Уровень надежности, гибкости и масштабируемости, который ожидается от современных программ, могут обеспечить распредёленные системы [2].

В процессе решения текущих задач было решено повысить производительность системы, сначала путём масштабирования вертикального, использовав многопоточность, работа с потоками проводилась самостоятельно без использования дополнительных инструментов, но сложность такой работы повлияла на стабильность и результат приложения, в связи с чем после пришлось использовать библиотеку TPL Dataflow. Но в данном случае использовалась мощность лишь одного компьютера, которой, как показала практика, не хватило.

То есть можно сказать, что многопоточность трудна, но даже если предположить, что всё получилось, удалось решить проблемы, то все равно возникают сложности, а именно это производительность одного сервера.

Но обычно горизонтальное масштабирование сложно в реализации и требует множества ресурсов.

2.2 Микросервисная архитектура

Рассмотрим микросервисную архитектуру, которая как раз-таки сулит горизонтальное масштабирование.

Стоит сразу сказать, что к текущей системе применение микросервисов будет неразумным, если назвать причину коротко и сжато, то потому что это будет очень затратно.

Говоря о применении данной архитектуре, то всегда нужно понимать, что данный переход может быть крайне дорогим. Микросервисы обычно применяются к очень большим и крупным проектам, которые обслуживают миллионы пользователей. А в данной работе приложение будет работать автономно без клиента.

Микросервисная архитектура – вариант сервис-ориентированной архитектуры программного обеспечения, ориентированный на взаимодействие насколько это возможно небольших, слабо связанных и легко изменяемых модулей – микросервисов, получивший распространение в середине 2010-х годов в связи с развитием практик гибкой разработки и DevOps [3].

Архитектурный стиль микросервисов – подход, при котором единое приложение разрабатывается как множество небольших сервисов, где каждый работает в своем процессе и общается с остальными, используя при этом легковесные механизмы, как правило HTTP [4]. Помимо того, что сервисы могут быть развертываемы и масштабируемы независимо, каждый сервис также обеспечивает надежную границу модуля, даже позволяя писать разные сервисы на разных языках программирования. Они также могут управляться разными командами [5]. В микросервисной архитектуре слабо связанные сервисы, взаимодействуя друг с другом, выполняют задачи, относящиеся к их бизнес-возможностям. Микросервисы в значительной степени получили свое название из-за того, что сервисы здесь меньше, чем в монолитной среде. Тем не менее, микро – о бизнес-возможностях, а не о размере [6].

Микросервисы архитектурно лучше организованы, т.к. каждый сервис занимается определенной задачей, и его не волнует работа других компонентов. Несвязанные сервисы также легче переделать и перестраивать для обслуживания различных приложений (например, обслуживать веб-клиентов и публичный API). Также могут быть преимущества в производительности в зависимости от того, как они организованы, потому что можно изолировать проблемные места и масштабировать их независимо от остального приложения.

Проблемы, которые возникают при переходе на микросервисную архитектуру: декомпозиция, приложение необходимо разбить таким образом, чтобы компоненты как можно меньше взаимодействовали, иначе сопровождать приложение будет сложно; транзакции, если монолит дает нам архитектурную целостность автоматически, то с микросервисами нужно реализовывать свой механизм или искать решения в библиотеках; построение отчетов, используя монолитную архитектуру с единой базой данных, построение отчетов не имеет особых сложностей, но в микросервисах эти данные могут находиться в разных базах, сервисах, что уже усложняет работу; высокая сложность разработки, сервисы могут разрабатываться различными командами, которые должны поддерживать документацию, актуальность данных, что добавляет сложности в работе; сложность тестирования, трассировки и отладки, чтобы протестировать какую-либо проблему, нужно скачать все задействованные микросервисы, а отладка становится нетривиальной задачей, и все логи нужно собирать где-то в одном месте, при этом логов нужно как можно больше, чтобы разобраться, что случилось [7]. Все эти проблемы необходимо решить для нормальной и стабильной работы системы.

Если говорить в общем, то микросервисы применяются, когда есть множество команд и разработчиков, ведь построение микросервисов нетривиальная задача, требующая больших ресурсов.

Поэтому необходимо искать иное решение. Нужно поменять подход к организации кода и проектированию самого приложения и решением текущей проблемы станет модель акторов.

И, как уже говорилось, может возникнуть ситуация, когда не будет хватать мощностей одного сервера, в этом случае появится необходимость изменить систему в распределённую систему. Чтобы была возможность добавления дополнительных серверов. В некоторых случаях может понадобиться при этом изменять саму архитектуру приложения и структуру программного кода. Это необходимо предусмотреть.

2.3 Распределённая система

В отличие от клиент-серверных архитектур распределенные приложения состоят либо из нескольких разных приложений, либо из нескольких копий одного приложения, работающих на разных машинах [2].

Их сложность в том, что очень трудно понять и оценить свойства распределенных систем в целом, их проектирование порой очень затрудняет процесс, а тестирование и обслуживание заставляет критически думать на протяжении всего проекта. На производительность распределенных систем влияет скорость сети.

Безопасность таких систем требует отдельной работы, когда мы имеем несколько машин и сообщения, которые передаются в сети, очевидно могут быть перехвачены извне, поэтому в данном случае поддержка безопасности является не самой тривиальной задачей для программистов.

Когда мы имеем несколько машин, ресурсы могут располагаться на разных компьютерах, поэтому необходимо создать продуманную идентификацию ресурсов

В распределённых системах для большинства случаев коммуникация будет основываться на протоколах TCP/IP в Internet, но иногда нужно будет применение иного подхода.

На производительность, работоспособность и надежность системы в целом может повлиять несколько факторов, одними из которых являются распределение процессов и ресурсов, аппаратные средства, и возможности адаптации системы.

Архитектура программного обеспечения практически во всех случаях является ключевым фактором правильности построения и вообще работоспособности приложения и системы, поэтому самый очевидный шаг – это использование как можно правильного подхода для какого-то конкретного случая.

Конечно же если мы говорим о наборе компьютеров в сети, которые являются некими функциональными элементами системы, то в конечном итоге для пользователя система выглядит в виде единого целого.

В таком случае, исходя из всех сложностей, действительно может возникнуть вопрос, а почему изначально не строить сразу распределенную систему?

Когда такая система создаётся сразу на начальном этапе, появляются большие сложности в разработке и тестировании, что в свою очередь оттягивает время окончания самой разработки приложения. Стоит упомянуть, что в данном случае написание программного кода не является простой задачей, множество операций сосредоточено на работе с файлами.

2.4 Конкурентность. Параллельное и многопоточное программирование

В своей книги «Конкурентность в C#. Асинхронное, параллельное и многопоточное программирование» Клири Стивен даёт определения для множества понятий, в том числе таким, как конкурентность, многопоточность, параллельная обработка, асинхронное, реактивное программирование. Конкурентность – выполнение сразу нескольких действий в одно и то же время; асинхронное программирование – разновидность конкурентности, использующая обещания или обратные вызовы для предотвращения создания лишних потоков; реактивное программирование – декларативный стиль программирования, при котором приложение реагирует на события; параллельная обработка – выполнение большого объема работы за счет распределения ее между несколькими потоками, выполняемыми одновременно; многопоточность – форма конкурентности, использующая несколько программных потоков выполнения [8].

Понимание данных определений даёт большое преимущество в организации кода, понимании фундаментальных понятий, тем более они очень употребительны в текущем контексте, а с помощью всего этого можно выгодно для себя решить, что и как применить в той или иной задаче и проблеме.

Итак, можно сделать вывод из этого, что конкурентность практически в каждом крупном проекте приносит свою пользу и наш случай не исключение. Вопрос скорее стоит другой, а именно какая форма конкурентности подходит для выполнения текущий задачи больше всего. В текущей ситуации, когда большой объем вычислительной работы в нашем приложении можно разделить на независимые блоки [8], отличным решением будет использование многопоточности и параллельного программирования, которое и является одной из разновидностей многопоточности.

Программирование с учетом множества ядер или процессоров называют параллельным программированием. Это подмножество более широкой концепции многопоточности [9].

Существуют две стратегии разбиения работы между потоками: параллелизм данных и параллелизм задач. Когда набор задач должен быть выполнен над множеством значений данных, мы можем распараллелить работу, заставив каждый поток выполнять (тот же самый) набор задач на

подмножестве значений. Такое выполнение называется параллелизмом данных, потому что мы распределяем данные между потоками. В другом случае при параллелизме задач мы распределяем задачи; другими словами, заставляем каждый поток выполнять свою задачу [9].

В нашем же случае, можно сказать, присутствует параллелизм данных, так как имеется множество значений данных, подмножества которых мы можем передать на параллельную работу потокам.

Таким образом изобразим на рисунке 3 схематично работу системы в режиме параллельности в будущем, как может она работать.

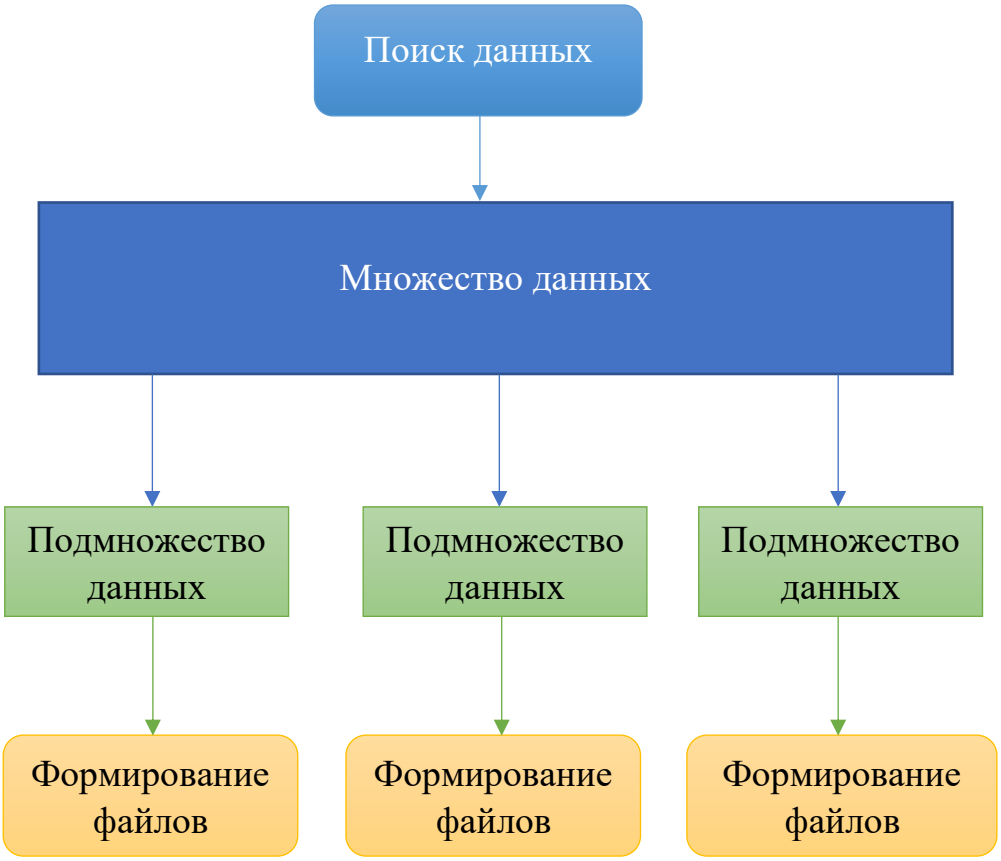

Рисунок 3 – Схема работа системы при параллельной обработке

В процессе работы в данном случае придётся решать несколько проблем: с одной стороны, необходимо загрузить процессоры равномерно, иначе может возникнуть ситуация, когда один процессор выполняет большую задачу, которая занимает основное время, а остальные после выполнения некоторых незначительных задач будут простаивать, здесь можно говорить об обычном последовательном вычислении, в этом случае использование параллельности не принесет выигрыша.

Стоит упомянуть о скорости обмена информации между всеми задействованными процессорами, если нам удалось достаточно равномерно загрузить процессоры, но скорость обмена низкая, здесь большое количество

времени будет уходить на ожидание информации, которая могла бы тратиться на выполнение работы процессора.

Но всё это в действительности будет описываться позже, следующие разделы расскажут о, так скажем, эволюции системы, каким оно было в первоначальном виде, во что оно превращалось, как развивалось и к чему пришло.

3 Однопоточное приложение

Если говорить о начальном приложении, то оно было синхронным однопоточным, в своем первозданном виде оно просуществовало недолго, потому что в таком виде приложение было по большей степени для того, чтобы реализовать саму функциональность приложения и тестировать непосредственно код, то есть плюсом можно посчитать то, что саму функциональность приложения можно было достаточно просто проверить.

На рисунке 4 схематично изображена работа приложения, все задачи выполнялись друг за другом, имея один поток, в котором последовательно выполняются несколько задач.

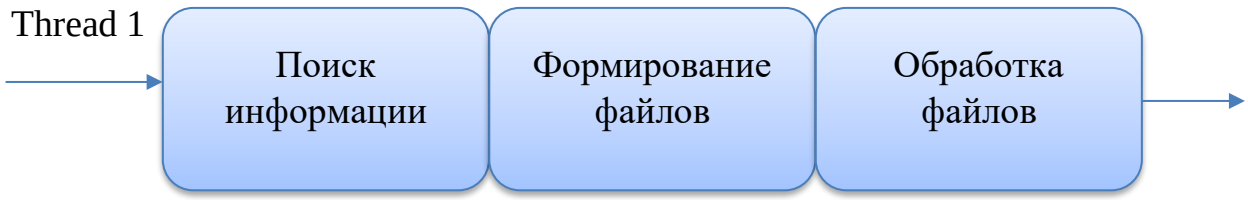

Рисунок 4 – Работа одного потока с несколькими задачами в приложении

Но нужно сказать, что данное приложение является тем самым рабочим приложением, которое позволило непосредственно создать рабочий программный код, оно позволяло во время написания кода быстро и без особых усилий, а главное верно, тестировать и отлаживать код. Только убедившись, что приложение соответствует требованиям и оно правильно работает можно было приступать к масштабированию, изменению архитектуры системы дабы улучшить производительность. В данном случае это не являлось чем-то принципиально недопустимым, так как основную логику и функциональность приложения не было необходимости менять, нужно было лишь продумать архитектурную логику и соответственно изменять по данной архитектуре, хотя нужно сказать, что это довольно непростая задача, которая требует множество времени на проектирование.

В связи с этим были проанализированы похожие ситуация, в различных аспектах, как они решаются и в дальнейшем используются.

Но нельзя недооценивать однопоточные приложения только из-за их простоты.

Иногда системы, которые нацелены на однопоточное выполнение, работают быстрее чем, поддерживающие конкурентность, потому что не требуют затрат на блокировки. Хотя их пропускная способность в одном ядре CPU [1].

Благодаря тому, что можно было установить последовательно выполнение функций, воспроизведение ошибок не было чем-то недостижимым, что позволяло и упрощало их устранению.

Таким образом, разработав приложение как однопоточное, была достигнута цель наполнения верной и правильно работающей кодовой базы, без применения многопоточности и других форм конкурентности, а значит не было препятствий и проблем, связанных с этими понятиями, которые бы препятствовали самой разработке.

4 Многопоточное приложение

Дальнейшее решение появилось практически сразу и это преобразование приложения из однопоточного в многопоточное. Соответственно следующим этапом было построение приложения с потоками, причем потоки реализовывались вручную, учитывая характеристики машины, на которой ведётся разработка, было создано 5 потоков, каждый выполнял свои задачи, схема данной работы изображена на рисунке 5.

Рисунок 5 – Многопоточное приложение с несколькими потоками, выполняющие несколько задач

Каждый из потоков выполнял поиск на своём интернет-ресурсе, но их объединяли одни функции: поиск информации, формирование файлов и различная обработка файлов.

Но работа с потоками сложна, с появлением в приложении больше одного потока появляется также и масса задач, которые необходимо решать: отладка, контроль взаимосвязей потоков, совместное использование данных, переменных и другое.

И этот случай не стал исключением, очень часто появлялись исключения, которые не позволяли дальнейшую работу с файлами, что вело к потере информации, также какой-либо процесс занимал определенный ресурс, а другие процессы пытались получить доступ к нему, соответственно им было отказано, что и приводило к очередным ошибкам, которые просто не позволяли дальнейшую обработку данных.

Эту проблему поднимают многие авторы различных книг.

Авторы книги «Язык программирования C# 7 и платформы .NET и .NET Core» Эндрю Троелсен и Филипп Джепикс пишут, что один из многих болезненных аспектов многопоточного программирования связан с ограниченным контролем над тем, как операционная система или среда CLR задействует потоки. Например, написав блок кода, который создает новый поток выполнения, нельзя гарантировать, что этот поток запустится немедленно [10].

В свою очередь Мартин Клеппман в своей книге «Высоконагруженные приложения. Программирование, масштабирование, поддержка» описывает некоторые события, которые создают проблему, напоминающую обеспечение выполнения многопоточного кода на одной машине: нет никакой уверенности в хронометраже в силу параллелизма и возможности произвольных переключений контекста. Существует немало инструментов для обеспечения безопасного выполнения многопоточного кода на отдельной машине: взаимные исключения (mutex), семафоры, атомарные счетчики, безблокировочные структуры данных, блокирующие очереди и т. д. К сожалению, эти утилиты нельзя непосредственно использовать для распределенных систем, поскольку в последних нет разделяемой памяти, а есть только отправляемые по ненадежной сети сообщения. Узел в распределенной системе должен учитывать то, что могут произойти приостановки в любой момент на длительное время, даже когда выполняется функция. Но все остальные в это время продолжают работать, приостановленный узел даже может быть объявлен неработающим, поскольку не отвечает. В конце концов приостановленный узел возобновляет работу, вероятно даже не замечая данной паузы до момента проверки часов. [1]

В книге «C# для профессионалов: тонкости программирования» автором которой является Джон Скит, говорится, что все так же сложно изящно

обрабатывать ошибки во всех ситуациях. Все еще довольно нелегко писать корректные многопоточные приложения, хотя эта задача значительно упростилась за счет улучшений, годами вносимых в язык и библиотеки. [11]

Подводя итоги данного раздела, хочется сказать, что самостоятельная работа с потоками возможна при определённых случаях, когда наше приложение не является нечто большим и громоздким, когда мы контролируем все её части и непосредственно в наших силах должным образом протестировать данное приложение. Но всё же на текущий момент существует немало хороших библиотек, которые помогают разработчикам в этом нелёгкой области. И дальнейшей целью будет использование в данной задаче таких инструментов и библиотек.

5 Приложение с использованием TPL Dataflow

На основе предыдущих разделов, стало очевидно, что появилась необходимость в использовании уже готовых библиотек для успешного использования потоков, и отличным решением в данном случае стала библиотека – TPL Dataflow.

Task Parallel Library (TPL) – это набор открытых типов и API в пространствах имен System.Threading и System.Threading.Tasks. Цель TPL – сделать разработчиков более продуктивными, упрощая процесс добавления параллелизма и конкурентности к приложениям. TPL динамически масштабирует степень параллелизма, чтобы наиболее эффективно использовать все доступные процессоры [12].

TPL предоставляет компоненты потока данных, помогающие повысить надежность приложений с поддержкой параллелизма. Эти компоненты потока данных вместе называются TPL Dataflow Library [13].

TPL Dataflow обычно используется в качестве простого конвейера: данные входят с одного конца и перемещаются, пока не выйдут с другого конца [8]. В нашем случае также был создан своего рода конвейер. На входе подаётся информация о страницах для обработки, далее происходят различные действия и на выходе полная обработанная информация, созданные файлы.

Библиотека TPL будет автоматически распределять нагрузку приложения между доступными процессорами в динамическом режиме с применением пула потоков CLR. Библиотека TPL поддерживает разбиение работы на части, планирование потоков, управление состоянием и другие низкоуровневые детали [10].

Система, написанная с использованием TPL DataFlow, использует многоядерную систему, поскольку все блоки, составляющие рабочий процесс, могут работать параллельно. TPL Dataflow обеспечивает эффективные методы для выполнения чрезвычайно параллельных задач, когда многие независимые вычисления могут выполняться параллельно очевидным образом.

Выше упоминалось такое понятие как чрезвычайная параллельность (чрезвычайно параллельная задача, англ. embarrassingly parallel) – это такой тип задач в системах параллельных вычислений, где не нужно прилагать большие усилия при разделении на несколько отдельных параллельных задач. Обычно зависимости нет между этими параллельными задачами, а это значит их результаты не влияют друг на друга [14].

Библиотека TPL Dataflow обеспечивает фундамент для передачи сообщений в системе и распараллеливания приложений с использованием центрального процессора и ввода/вывода, имея при этом высокую пропускную способность и низкую задержку. Данная библиотека даёт возможность разработчикам контролировать буферные данные и их координацию в системе.

Источники и цели в TPL Dataflow.

Библиотека TPL Dataflow состоит из блоков потока данных, являющиеся структурами данных, которые записывают в буфер и обрабатывают данные. Библиотека определяет три вида блоков потока данных: исходные блоки – источники, целевые блоки – цели и распространяющие блоки. Исходные блоки действуют как источники данных и могут быть прочитаны. Целевые блоки действуют как приёмник данных и могут быть записаны. Распространяющие блоки действуют как исходные блоки и целевые, могут как записываться, так и считываться.

Соединительные блоки в TPL Dataflow.

Библиотека даёт возможность соединения блоков потока данных, чтобы сделать конвейеры, которые будут представлять собой линейные последовательности блоков потока данных, или сети, которые будут представлять собой графики блоков потока данных. Конвейер является одной из форм сети. В конвейере или в сети, источники асинхронно распространяют данные до цели, когда эти данные становятся доступными.

Фильтрация в TPL Dataflow.

При связке источника с целевым блоком, можно предоставить делегат, который будет определять, принимает ли целевой блок сообщение или же его отклоняет, решая это по значению принимаемого сообщения. Таким образом можно гарантировать, что блок получит только определённые сообщения. Для большинства предопределенных типов блоков потока данных, если исходный блок связан с несколькими целевыми блоками, когда целевой блок отказывается от сообщения, источник предлагает это сообщение другим целевым блокам.

Модель программирования потока данных связана с концепцией передачи сообщений, когда независимые компоненты программы обмениваются данными друг с другом посредством отправки сообщений [13].

Также блоки потока данных поддерживают концепцию завершения, когда блок находится в завершенном состоянии, не выполняет никакой дальнейшей работы

Библиотека TPL Dataflow предоставляет несколько предопределенных типов блоков потоков данных. Эти типы подразделяются на три категории: буферные блоки, исполнительные блоки и группирующие блоки. Ниже описываются типы блоков, которые составляют эти категории конструкций.

Буферные блоки содержат данные для использования их потребителями данных:

– Класс BufferBlock <T>. По сути, это буфер для хранения экземпляров T, структура асинхронного обмена сообщениями общего назначения. Данный класс хранит очередь сообщений по принципу FIFO – первый пришел, первый вышел.

– Класс BroadcastBlock <T>. Данный класс необходимо использовать, когда нужно передать несколько сообщений другому компоненту, но для этого компонента требуется только самое последнее значение. А также существует возможность передачи нескольким компонентам некоторое сообщение.

– Класс WriteOnceBlock <T>. Если BufferBlock самый фундаментальный блок в TPL Dataflow, то WriteOnceBlock самый простой блок. Хранит не более одного значения, и как только это значение будет установлено, оно никогда не будет заменено или перезаписано. Класс WriteOnceBlock <T> напоминает класс BroadcastBlock <T>, за исключением того, что объект WriteOnceBlock <T> может быть записан только один раз. Класс полезен, когда нужно распространять только первое из нескольких сообщений.

Блоки исполнения вызывают предоставленный пользователем делегат для каждой части полученных данных:

– Класс ActionBlock <T>. Здесь данные поступают в блок, которые должны быть обработаны. Класс ActionBlock <TInput> является целевым блоком, который вызывает делегат при получении некоторых данных, необходимо понимать это как делегат, который выполняется асинхронно, когда данные становятся доступными. Этот класс можно логически представить как буфер для обработки данных в сочетании с задачами для обработки этих данных [15].

– Класс TransformBlock <TInput, TOutput>. Данный класс также позволяет исполнению делегата выполнять некоторые действия для каждого входного элемента данных, но в отличие от ActionBlock, он имеет выходные данные. То есть он действует и как источник, и как цель.

– Класс TransformManyBlock <TInput, TOutput>. Данный класс создает ноль или более выходных значений для каждого входного значения, а не только одно выходное значение для каждого входного значения.

Группирующие блоки объединяют данные из одного или нескольких источников с различными ограничениями:

– Класс BatchBlock <T>. Здесь объединяются несколько отдельных элементов, наборы входных данных, называемыми пакетами, в массивы выходных данных. Существует два режима – жадный и нежадный. В жадном режиме принимается каждое предлагаемое сообщение и распространяется по массиву после получения некоторого количества элементов, которое будет

указано. В нежадном режиме откладываются все входящие сообщения до тех пор, пока достаточное количество источников не предложит сообщения блоку для формирования пакета.

– Класс JoinBlock(T1, T2, ...). Как и класс BatchBlock <T>, классы JoinBlock <T1, T2> и JoinBlock <T1, T2, T3> работают в жадном или нежадном режиме. В жадном режиме, который используется по умолчанию, объект JoinBlock <T1, T2> или JoinBlock <T1, T2, T3> принимает каждое предлагаемое им сообщение и распространяет кортеж после того, как каждая из его целей получит хотя бы одно сообщение. В нежадном режиме объект JoinBlock <T1, T2> или JoinBlock <T1, T2, T3> откладывает все входящие сообщения до тех пор, пока всем целям не будут предложены данные, необходимые для создания кортежа. Эти классы могут группировать данные из нескольких источников данных.

– BatchedJoinBlock (T1, T2, ...). BatchedJoinBlock <T1, T2,…> в некотором смысле является комбинацией BatchBlock и JoinBlock <T1, T2,…>. Указывать размер каждого пакета можно при создании объекта BatchedJoinBlock <T1, T2>. BatchBlock используется для объединения N входных данных в коллекцию, а BatchedJoinBlock <T1, T2,…> используется для сбора N входных данных со всех целей в кортежи коллекций [15].

В нашем же случае достаточно было использовать классы Buffer, который использовался для хранения данных потребительским блоком, и Action, целевой блок, который выполнял основную функцию потребления данных.

На рисунке 6 схематично изображена работа приложения с TPL Dataflow.

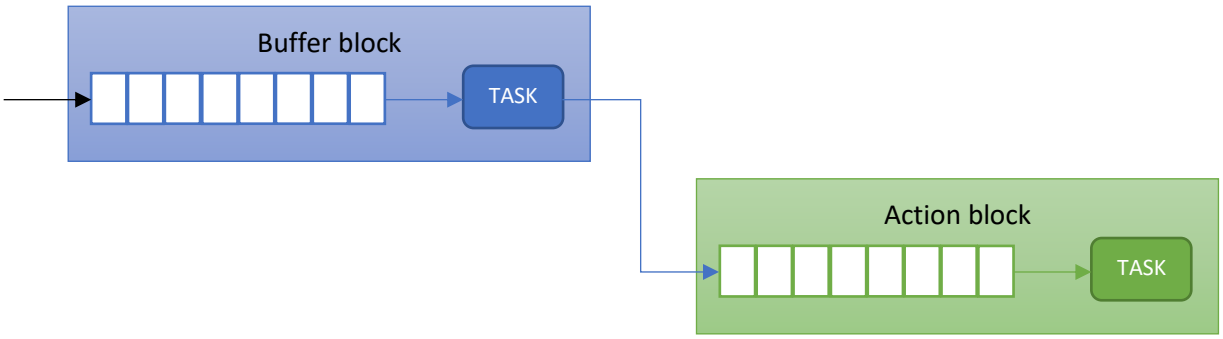

Рисунок 6 – Схема работы приложения с TPL Dataflow

На входе поступает найденная информация, далее buffer блок отправляет на обработку данные в action блок, а уже он формирует и записывает файлы на сервер и в базу данных.

Когда нужно использовать данную TPL Dataflow библиотеку:

– Если необходимо построить конвейер обработки данных.

– Когда есть необходимость в передачи данных и полезным это будет при существовании блока, который сможет воспроизводить данные для

других блоков, а также он должен иметь свой буфер для хранения данных во время ожидания, чтобы они оставались доступными.

Но TPL Dataflow в некоторых случаях не подходит для того, чтобы её применять в приложении:

– В первую очередь, когда приложение не является конкурентным.

– Когда разработчикам нужен детальный контроль над тем, что делает поток, как он это делает и т.п.

Но на текущий момент данная библиотека, хоть и намного увеличила производительность, но всё же этого недостаточно, поэтому решено было и дальше увеличивать производительност с помощью увеличения серверов, для этого нужно было менять парадигму программирования и разработки. Следующим этапом является использование модели акторов в приложении. И на самом деле TPL Dataflow схожа с данной моделью, но всё же существуют причины, по которым эту библиотеку нельзя назвать акторским фреймворком. Но общие и отличительные черты обсудим в следующем разделе.

6. Модель акторов. Сравнение библиотек на платформе .NET

6.1. Модель акторов

Модель акторов — математическая модель параллельных вычислений, строящаяся вокруг понятия «актора», на английском actor – актёр, действующий субъект, который считается универсальным примитивом параллельного исполнения. В данной модели взаимодействие акторов происходит через передачу сообщений, в ответ на сообщения акторы могут принимать локальные решения, создавать новые акторы, посылать свои сообщения, устанавливать, как следует реагировать на последующие сообщения. Модель акторов создана как теоретическая база для ряда практических реализаций параллельных систем [16].

Для начала стоит разобраться в некоторых связанных определениях, если модель акторов является одним из представителей параллельных вычислений приведем определение данному понятию.

Параллельные вычисления — способ организации компьютерных вычислений, при котором программы разрабатываются как набор взаимодействующих вычислительных процессов, работающих параллельно (одновременно) [17].

Вместе с тем, стоит определить и такое понятие, как распределённые вычисления. Потому что сразу, с первого взгляда трудно определить разницу между распределёнными и параллельными вычислениями, но разница существует.

Распределённые вычисления – способ решения трудоёмких вычислительных задач с использованием нескольких компьютеров, чаще всего объединённых в параллельную вычислительную систему [18].

Исходя из определений можно сказать, что распределенные вычисления не выполняются на одном компьютере, а параллельные вычисления могут производиться как на одном, с использованием многопоточности, так и на нескольких компьютерах.

В действительности выполнение параллельных вычислений в распределенных системах возможно, чтобы убедиться укажем определение параллельным вычислительным системам.

Параллельные вычислительные системы – это физические компьютерные, а также программные системы, реализующие тем или иным способом параллельную обработку данных на многих вычислительных узлах [19].

То есть под параллельной обработкой данных в данном случае подразумеваются параллельные вычисления.

В процессе разбора данной тематики могут возникнуть различные вопросы по поводу того, что с какой целью нам распараллеливать вычисления,

а дело в том, что большое количество задач можно разделить на набор меньших задач и решать их одновременно, но в данном случае необходима координация действий. Писать такие программы намного сложнее обычных последовательных, появляется конкуренция за ресурсы, а это большое количество потенциальных ошибок, само взаимодействие между процессорами – это большая проблема, которая требует верного решения иначе о высокой производительности параллельных систем не идёт и речи. Казалось бы, для чего так усложнять систему, раньше такие вычисления использовались не так часто, хватало ресурсов машин, но всё дело в том, что в какой-то момент тактовая частота перестала удваиваться, о чем шла речь в прошлых разделах. Сейчас же параллельные вычисления применяются в многоядерных процессорах.

Итак, вернёмся к модели акторов. И вкратце затронем историю данной модели.

Модель Акторов была опубликована в 1973-ем году, благодаря Карлу Хьюитту (Carl Hewitt), далее развивалась в 1981-ом году Уильямом Клингером (William Clinger) и Гулом Агха (Gul Agha) в 1985-ом году [20].

Что ж, концепция данной модели была опубликована сравнительно давно, почему же сейчас она является очень популярной и известной, и она не утратила своей актуальности?

Дело в том, что на это повлияли несколько факторов и на самом деле было несколько волн популярности модели, последний бум начался несколько лет назад и длится он до сих пор, этому предшествовало развитие многоядерных процессоров, они в свою очередь возобновили интерес к параллельному программированию.

А также развитие языка Erlang, который является функциональным языком программирования с динамической типизацией. Используется он при создании распределенных вычислительных систем. Порождает легковесные процессы, которые работают параллельно, при этом взаимодействия происходят через асинхронные сообщения, в соответствии с моделью акторов [21].

Его развитие и успешное применение в различных проектах внесла свой вклад в дальнейшую популярность модели акторов.

Затем в IT индустрии появляется фреймворк Akka, который также способствовал росту популярности модели акторов. Данный фреймворк для языков Java и Scala. Затем был создан порт в .NET известный как Akka.NET. Целью Akka является предоставление необходимого инструмента для простого создания приложений, которые развёртываются в облаке или запускаемых на многоядерных процессорах, которые эффективно используют все имеющиеся вычислительные ресурсы [22].

Чтобы не иметь N потоков, конкурирующих друг с другом за ресурсы, можно сделать M потоков, которые будут владеть собственными данными. Ни один из потоков не имеет доступа к данным других потоков [20].

Сравнить акторы и традиционную модель можно примером: телефон и отправление сообщений по почте. Когда несколько человек пытаются дозвониться на один телефонный номер, то начинается конкуренция за доступ к общему разделяемому ресурсу – адресату. А с почтой иначе, отправитель письма (актор) просто посылает письмо адресату без каких-либо задержек, при этом неизвестно, когда получатель прочтет письмо [23].

Как уже говорилось в прошлом разделе, TPL Dataflow имеет общие черты с моделью акторов. Каждый блок потока данных независим от других – он запускает задачи для выполнения работы по мере необходимости (например, выполнения преобразующего делегата или передачи вывода следующему блоку). Можно также настроить каждый блок для параллельного выполнения, чтобы он запускал несколько задач для обработки дополнительного ввода. Из-за этого поведения каждый блок отчасти напоминает актора в акторских фреймворках. Но TPL Dataflow не является полноценным акторским фреймворком; в частности, отсутствует встроенная поддержка корректного восстановления после ошибок или повторных попыток. TPL Dataflow – библиотека с функциональностью, сходной с функциональностью акторов, но не являющаяся полноценным акторским фреймворком [8]. Также модель акторов позволяет масштабировать как локально, так и не локально, поэтому переход в данном случае на модель акторов стал необходимым.

Основным мотивирующим фактором создания модели стала задача построения распределённых вычислительных систем на базе сотен и тысяч независимых компьютеров, оснащённых собственной локальной памятью и коммуникационными интерфейсами. С появлением многопроцессорных систем и многоядерных архитектур интерес к модели акторов возрос также вне контекста распределённых систем [16].

Акторная модель (actor model) — модель программирования для создания конкурентного доступа в пределах одного процесса. Вместо того чтобы иметь дело непосредственно с потоками выполнения, логика инкапсулируется в акторах. В распределенных акторных фреймворках эта модель программирования используется для масштабирования приложения на несколько узлов. Независимость от расположения лучше развита в акторной модели, чем RPC, поскольку эта модель изначально допускает возможность потери сообщений даже в пределах одного процесса. Хотя задержка передачи сообщений по сети, вероятно, выше, чем в пределах одного процесса, при использовании акторной модели различия между локальным и удаленным обменом сообщениями не столь велики [1].

6.2 Сравнение библиотек на платформе .NET для обеспечения актороподобного стиля программирования

Так как в компании используется стек технологий .NET, то выбирать реализацию необходимо было в данном стеке. В рамках .NET существует несколько библиотек, которые обеспечивают актороподобный стиль программирования основными из них являются Akka.NET и Orleans.

Akka.NET – это порт оригинальной Akka framework Java/Scala, библиотека с исходным кодом, для создания параллельных, распределенных и отказоустойчивых приложений.

Orleans – это кроссплатформенный фреймворк компании Microsoft для создания надёжных, масштабируемых распределённых приложений на платформе .NET. Концепт данного фреймворка построен на виртуальных акторов, что немного отличает его от подобного фреймворка Akka.NET. Акторы в Orleans называются Grains, то есть зёрна, а сервера, которые участвуют в кластере – Silo.

Orleans был создан Microsoft Research и представил виртуальную модель акторов, как новый подход к созданию распределенных систем нового поколения для эпохи облачных вычислений. Основным вкладом Orleans является его модель программирования, которая устраняет сложность, присущую высоко параллельным распределенным системам, не ограничивая возможности и не накладывая обременительных ограничений на разработчика [24].

Немного о происхождении Orleans. Orleans был создан в Microsoft Research и предназначен для использования в облаке. С 2011 года он широко использовался в облаке и локально несколькими группами продуктов Microsoft, прежде всего игровыми студиями. Orleans стал с открытым исходным кодом в январе 2015 года и привлек многих разработчиков, которые сформировали одно из самых активных сообществ с открытым исходным кодом в экосистеме .NET. В активном сотрудничестве между сообществом разработчиков и командой Orleans в Microsoft, функции добавляются и улучшаются ежедневно. Microsoft Research продолжает сотрудничать с командой Orleans, предлагая новые важные функции, такие как гео-распределение, индексация и распределенные транзакции, которые продвигают современное состояние. Orleans стал основой выбора для построения распределенных систем и облачных сервисов для многих разработчиков .NET [25].

Зёрна являются фундаментальным понятием в Orleans, это сущности, которые содержат пользовательскую идентичность, поведение и состояние. Идентификаторы зерна – ключи, которые должны быть определены пользователем, они делают зёрна доступными. Зёрна могут быть вызваны другими зёрнами, клиентами через строго типизированные интерфейсы. Каждое зерно является экземпляром класса, который реализует один или несколько из этих интерфейсов [25].

Зерно изображено на рисунке 7.

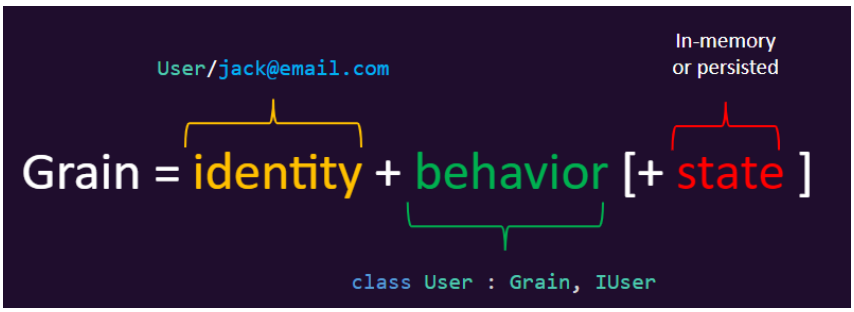

Рисунок 7 – Зерно в Orleans [25]

Зёрна могут иметь изменчивое, постоянное состояние, которое может храниться в любой системе хранения, значит, что зерна неявно разделяют состояние приложения, а это обеспечивает автоматическую масштабируемость и восстановление после ошибок. Состояние зерна сохраняется в памяти, пока оно активно, а это приводит к тому, что снижаются задержки и снижаются нагрузки на хранилища данных. Обработка зерна автоматически выполняется по требованию во время выполнения в Orleans. Зерна, которые не используются некоторое время, автоматически удаляются из памяти для того, чтобы ресурсы не были попросту заняты. Это возможно благодаря их стабильной идентичности, которая позволяет вызывать зерна независимо от того, загружены они уже в память или нет. Это также обеспечивает прозрачное восстановление после сбоя, поскольку вызывающей стороне не нужно знать, на каком сервере создается зерно в любой момент времени. На рисунке 8 изображён жизненный цикл зёрен, зёрна имеют управляемый жизненный цикл, при этом среда выполнения Orleans отвечает за активацию/деактивацию и размещение/установление зёрен по мере необходимости. Это позволяет разработчику писать код, как если бы все зёрна всегда были в памяти [25].

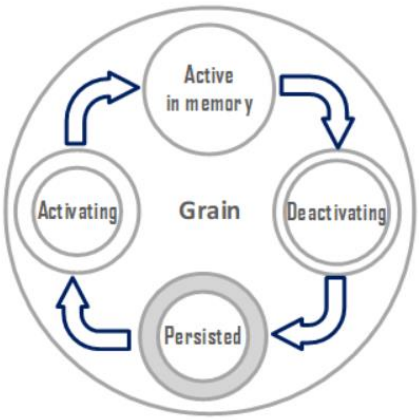

Рисунок 8 – Жизненный цикл зёрен [25]

В Orleans зёрна нельзя ни запускать, ни останавливать, потому что тут акторы не могут динамически создавать других акторов – все зерна как бы уже «существуют».

Вообще основной задачей Orleans является упрощение модели программирования, основанного на акторах, поднимая уровень абстракции в модели акторов, где даже разработчики, не являющиеся специалистами в распределенном программировании, смогут построить распределенную систему.

В нашем случае была выбрана библиотека Akka.NET, которая будет подробно описываться в следующих разделах.

Потому при сравнении этих библиотек получается, что Orleans упрощает распределенные вычисления, даже не эксперты в этой области могут создавать эффективные и масштабируемые системы. А фреймворк Akka позволяет строить распределенные системы, при этом предлагает полную мощность и внутреннюю при этом сложность этой области.

7. Фреймворк Akka.NET

7.1 Введение в Akka.NET

Akka.NET представляет набор библиотек с открытым исходным кодом для разработки масштабируемых, отказоустойчивых систем, охватывающих процессорные ядра и сети. Akka позволяет вам сосредоточиться на удовлетворении бизнес-потребностей, а не писать низкоуровневый код для обеспечения надежного поведения, отказоустойчивости и высокой производительности [26].

Akka.NET является портом оригинальной Akka на платформе JVM.

Akka.NET предоставляет: многопоточное поведение без использования низкоуровневых параллельных конструкций, не нужно думать о проблемах видимости памяти; прозрачная удаленная связь между системами и их компонентами, не нужно писать или поддерживать сложный сетевой код; кластерная архитектура высокой доступности, которая является гибкой, масштабируется по требованию [26].

Все функции, которые Akka.NET объединяет в своей модели программирования, обеспечивают уровень абстракции, который позволяет правильно проектировать конкурентные, параллельные и распределенные системы. Akka.NET предлагает глубину интеграции, которую достичь сложно, выбирая библиотеки для решения отдельных проблем и пытаясь их соединить вместе для единой системы.

7.2 Библиотеки и модули Akka.NET

На самом деле Akka.NET состоит из нескольких библиотек и модулей, различные функции которых позволяют использовать разработчикам в своих системах.

Акторы (Akka Library – библиотека Akka, the Core – Ядро).

Использование акторов в библиотеках Akka.NET обеспечивает согласованную, интегрированную модель разработки, при которой программисту нет необходимости индивидуально решать проблемы, которые возникают при параллельном или распределенном проектировании системы в целом. Если взглянуть издалека, акторы – это такая парадигма программирования, которая доводит инкапсуляцию, одну из основ ООП, до крайности. В отличие от объектов, акторы инкапсулируют не только своё состояние, но и исполнение. Акторы общаются не через вызовы методов, а через сообщения. Это позволяет избавиться от ограничений ООП, когда речь идёт о параллелизме и удаленном взаимодействии в системе.

Задачи, которые решают акторы:

– Создание и проектирование высокопроизводительных параллельных приложений.

– Обработка ошибок в многопоточном программировании.

– Защита проекта от ловушек параллелизма.

Модуль Remoting.

Remoting позволяет удалённым акторам, которые работают на разных компьютерах, беспрепятственно обмениваться сообщениями. Он может быть включен в основном с конфигурацией, у него всего несколько API. Благодаря модели акторов локальная и удаленная отправка сообщений выглядит одинаково. Шаблоны, которые используются в локальных системах, транслируются непосредственно в удаленные системы. Данный модуль обеспечивает основу, на которой построена подсистема Cluster.

Проблемы, которые решает Remoting:

– Как обращаться действующим системам, которые существуют на удалённых хостах.

– Как обращаться к отдельным акторам, которые находятся на удалённых системах акторов.

– Перевод сообщений в байты при передаче по сети.

– Управление низкоуровневыми сетевыми соединениями между хостами, обнаружение акторных систем и хостов, у которых произошли сбои.

– Как мультиплексировать связь от несвязанного набора акторов в одном сетевом соединении.

Модуль Cluster.

Если существует набор систем акторов, которые взаимодействуют для решения некоторой бизнес-задачи, наверняка нужно этим управлять дисциплинированно. В то время как Remoting решает проблему адресации и связи с компонентами удаленных систем, кластеризация даёт возможность организовать их в «мета-систему» связанную протоколом членства.

В большинстве случаев использование напрямую Remoting не понадобится, за место этого необходимо будет использование модуля Cluster. Кластеризация предоставляет дополнительный набор сервисов поверх модуля Remoting, он необходим многим реальным приложениям.

Задачи, которые решает модуль Cluster:

– Как поддерживать набор систем акторов, то есть кластер, которые могут взаимодействовать друг с другом и рассматривать друг друга как часть кластера.

– Безопасное внедрение новый систему в уже существующею.

– Как надежно обнаруживать системы, которые стали недоступными.

– Удаление неисправных частей системы, чтобы вся остальная система успешно продолжила работу.

– Как распределить вычисления среди всех частей системы.

– Как назначить членов кластера на определенную роль, чтобы они предоставляли определённые услуги, а не другие.

Cluster Sharding.

Sharding помогает решить проблему распределения набора акторов среди участников кластера Akka.NET. Sharding – это шаблон, который в основном используется вместе с Persistance, чтобы сбалансировать большой набор постоянных объектов.

Cluster Singleton.

Распространённый вариант использования в распределенных системах, состоит в том, чтобы иметь один объект, отвечающий за данную задачу, который совместно используется другими членами кластера и переносится в случае сбоя в работе хост-системы.

Cluster Publish-Subscribe.

Для координации между системами необходимо распространять сообщения для всех или для одной системы из набора заинтересованных систем в кластере.

Persistence.

Как только система выключается в нормальном режиме или из-за сбоя, все данные, которые были в памяти, теряются. Persistence обеспечивает шаблоны, позволяющие акторам сохранять события, которые приводят к их текущему состоянию, и после их можно воспроизвести для восстановления состояния объекта, размещённого актором.

Модуль Distributed Data.

Модуль распределенных данных обеспечивает инфраструктуру для обмена данными и ряд полезных типов данных.

Streams.

Streams или потоки предоставляют высокоуровневую абстракцию поверх акторов, что упрощает написание сетей обработки, обрабатывает все мелкие детали в фоновом режиме и обеспечивает безопасную, типизированную, компонуемую модель программирования.

Всё это неполный список доступных модулей, но он даёт хорошую оценку того, что можно сделать, чего достичь, используя Akka.NET в разрабатываемых системах. Все эти модули хорошо интегрируются друг с другом, смоделировав объекты и использовав разные модули, начать разработку, и можно быть готовым к масштабированию, доступности.

7.3 Архитектура Akka.NET

При построении новых систем иногда возникает вполне логичный вопрос, а с чего начать, что будет первым написано? Так и в случае с Akka.NET может возникнуть вопрос, а какой актор будет первым? Помочь на таких первых этапах жизни системы могут существующие лучшие практики.

При создании актора, он будет относиться к родителю, который его и создал, что в общем-то говорит о том, что акторы образуют дерево: есть родители и есть дочерние акторы.

Чтобы создать актор верхнего уровня, необходимо инициализировать систему акторов – actor system [27]. На рисунке 9 изображена архитектура верхнего уровня.

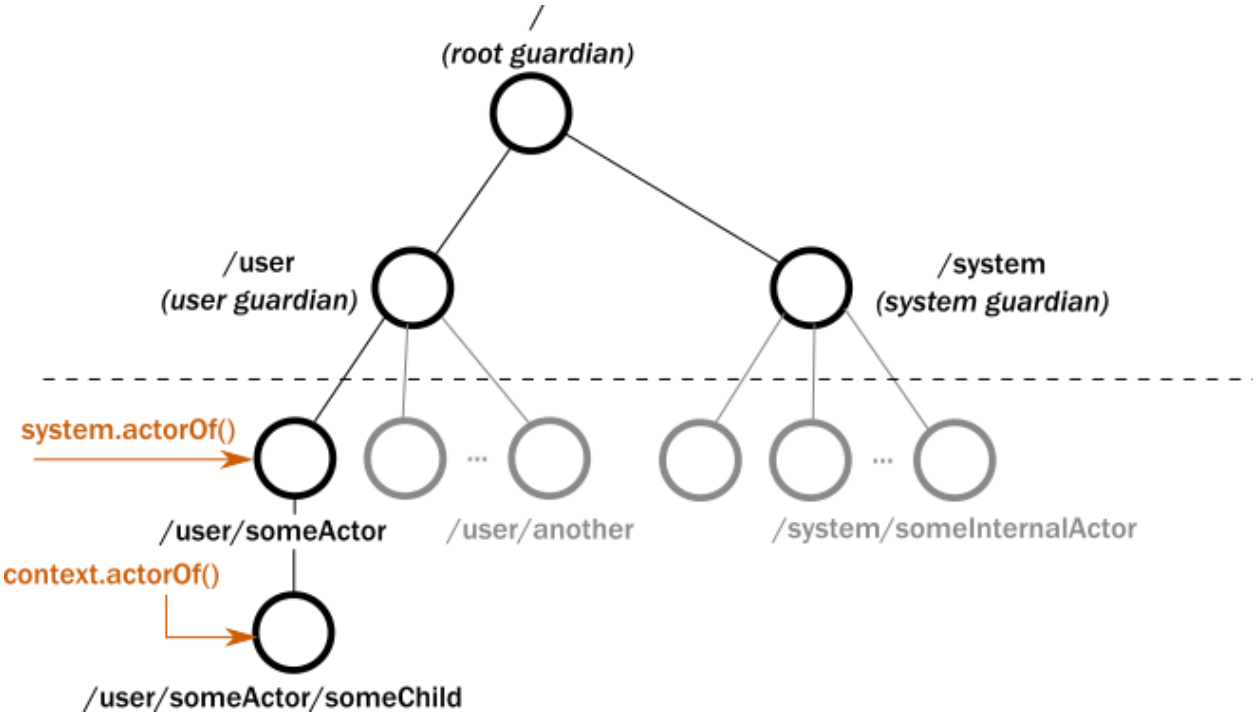

Рисунок 9 – Архитектура верхнего уровня [27]

Здесь необходимо определить некоторые акторы:

/ − так называемый корневой хранитель. Это родитель всех акторов в системе и последний, кто остановится, когда сама система будет остановлена.

/user – пользователь хранитель. Это родительский актер для всех пользовательских актеров. Имя user в данном случае не имеет ничего общего ни с вошедшим в систему пользователем, ни с обработкой пользователя в целом. Это имя на самом деле означает пространство пользователей, поскольку именно здесь живут акторы, не имеющие доступа к внутренним компонентам Akka.NET, т.е. все акторы, созданные пользователями библиотеки Akka.NET. Каждый актор, который будет создан, будет иметь постоянный путь /user/ к нему.

/system – хранитель системы.

Также может возникнуть вопрос о том, что есть ли у корневого хранителя родитель? На самом деле есть, эта особая сущность, которая называется «Bubble-Walker». Специальная сущность, которая невидима для пользователя и используется только внутри [27].

Поскольку все ссылки на акторы являются действительными URL-адресами, необходимо поле протокола, например, akka: // для акторов. Затем, как и во Всемирной паутине, система идентифицируется [27].

После появления акторов в системе, по запросу пользователя они могут быть остановлены, при этом каждый раз, когда актор останавливается, все его дочерние акторы рекурсивно также останавливаются. Что очень удобно, имея при этом положительный эффект, ведь тем самым удаётся предотвратить утечку ресурсов.

Родительские и дочерние акторы связаны не только жизненными циклами. Когда актор терпит неудачу, это могут быть исключения, он временно приостанавливается. И отправляется информация об ошибке родителю, который решает, как будет обрабатываться исключение. Стратегию супервизора разработчик может настроить самостоятельно, но по умолчанию дочерний актор остановится и перезапустится.

7.4 Использование Akka.NET, сценарии развертывания

Вариантов использования и сценариев развертывания Akka.NET множество: консольное приложение, платформа для веб разработки ASP.NET, служба Windows, Azure PaaS Worker Role.

В нашем случае использовалось консольное приложение, которое может развертываться как служба Windows, используя TopShelf, который упрощает хостинг Windows Service.

Прежде чем работать с Akka.NET, необходимо установить пакеты, к примеру, Akka и Akka.Remote. На рисунке 10 изображены команды установки пакетов Akka и Akka.Remote.

```
PM> install-package Akka
PM> install-package Akka.Remote
```

Рисунок 10 – Установка пакетов Akka и Akka.Remote

Пример консольного приложения с двумя сервисами изображен на рисунке 11.

```
using Akka;
using Akka.Actor;
using Akka.Configuration;
using System;

namespace TestAkkaNET
{
    Ссылок: 0
    class Program
    {
        Ссылок: 0
        static void Main(string[] args)
        {
            using (var system = ActorSystem.Create("my-actor-server"))
            {
                var service1 = system.ActorOf<Service1>("service1");
                var service2 = system.ActorOf<Service2>("service2");
                Console.ReadKey();
            }
        }
    }
}
```

Рисунок 11 – Пример консольного приложения с двумя сервисами

Akka.NET используется во многих крупных организациях и в таких отраслях, как инвестиционный и коммерческий банкинг, розничная торговля и социальные сети, симуляторы, игры и ставки, автомобильные и транспортные системы, здравоохранение, анализ данных и многое другое. Любая система, которая нуждается в высокой пропускной способности и низкой задержке, является хорошим кандидатом для использования Akka.NET [28].

7.5 Системы акторов в Akka.NET

Акторы – это объекты, которые инкапсулируют состояние и поведение, они общаются исключительно путем обмена сообщениями, которые помещаются в почтовый ящик получателя. ActorSystem – это тяжелая структура, которая будет выделять от 1 до N потоков, поэтому создавать необходимо по одной системе акторов для каждого логического приложения [29].

Акторы образуют иерархии. Один актор, который должен наблюдать за определенной функцией в программе, может разделить свою задачу на более мелкие, более управляемые части программы, для этого он запускает дочерних акторов, которыми он руководит. И у актора может быть один супервизор, если точнее тот, кто его создал.

Существенной особенностью систем акторов состоит в том, что задачи разделяются и делегируются до достаточно небольших размеров, чтобы их можно было обработать одним куском.

Несколько систем акторов с различной конфигурацией могут сосуществовать в одной и той же среде выполнения без проблем, глобального общего состояния нет в Akka.NET. Соединив это с прозрачной связью между системами акторов – в пределах одного узла или через сетевое соединение, чтобы увидеть, что сами системы акторов могут использоваться в качестве строительных блоков в функциональной иерархии [29].

Изначально в приложении на основе Akka создаётся система акторов – это экземпляр ActorSystem. Он может создавать акторы верхнего уровня, обычно им создаётся единственный актор верхнего уровня и далее все остальные акторы в приложении. Акка в действии

7.6 Акторы в Akka.NET

Устройство актора в Akka.NET изображено на рисунке 4, он состоит из поведения – behavior, «почтового ящика» – mailbox, состояния – state, его «детей» – childern и стратегии руководителя или супервизора – supervisorstrategy. Всё это заключено в ссылке актора – actor reference. Все акторы живут в контексте системы акторов. На рисунке 12 изображено устройство актора в Akka.NET.

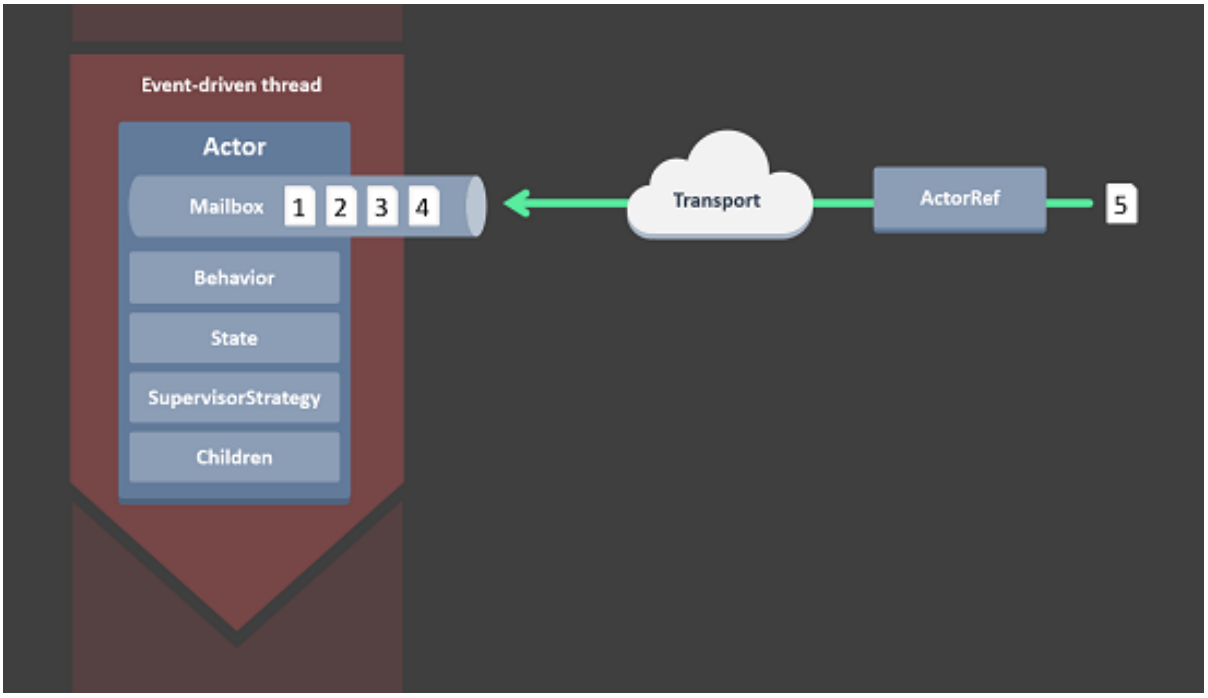

Рисунок 12 – Устройство актора в Akka.NET [30]

Ссылка актора.

Объект актора должен быть защищен снаружи, чтобы извлечь выгоду из модели субъекта. Следовательно, акторы представляются снаружи с использованием ссылок на акторы, являющиеся объектами, которые можно передавать свободно и без ограничений. Это разделение на внутренний и внешний объект обеспечивает прозрачность для всех требуемых операций: перезапуск актора без необходимости обновлять ссылки в другом месте, размещение фактического объекта актора на удаленных хостах, отправка сообщений акторам в совершенно разных приложениях. Но самый важный аспект заключается в том, что невозможно заглянуть внутрь актора и получить его состояние извне, если только актор неразумно не публикует эту информацию сам [30].

Состояние.

Объекты актора обычно содержат некоторые переменные, отражающие возможные состояния в которых может быть актор. Внутреннее состояние жизненно важно для действий актора, но наличие противоречивого состояния проблематично. Таким образом, когда в акторе происходят серьёзные противоречия и проблемы и он перезапускается его родителем, состояние будет создаваться с нуля, как при первом создании актора.

Поведение.

Поведение включает в себя функцию, которая определяет действия, необходимые предпринять в ответ на сообщение в данный момент времени, например, переслать запрос, если клиент авторизован, отклонить его в противном случае. Это поведение может со временем меняться [30].

Почтовый ящик.

Цель актора – это обработка сообщений, которые были отправлены другими акторами. И элементом между отправителем и получателем является почтовый ящик актора, он в единственном экземпляре у каждого актора. И в этот почтовый ящик отправители помещают свои сообщения в очередь. Причём порядок сообщений в очереди происходит во временном порядке операций отправки, что означает, что сообщения отправленные разными акторами, могут не иметь определенного порядка во время выполнения, потому что участники распределены случайно по потокам, но при этом отправка нескольких сообщений одному и тому же получателю от одного и того же актора ставит их в очередь по порядку, как они отправлялись. На выбор предлагаются разные реализации почтовых ящиков, по умолчанию FIFO: порядок сообщений, которые обрабатываются актором, будет соответствовать порядку, в котором они были поставлены в очередь. Но возможно также и реализовать иной порядок, отличный от FIFO.

Акторы.

Каждый участник потенциально является супервизором, если им будут созданы дочерние акторы для делегирования подзадач, он начинает их контролировать. Список дочерних акторов поддерживается в контексте актора, соответственно он имеет к нему доступ, измениться список может

после создания или остановки дочерних акторов, эти действия отражаются немедленно, они не блокируют родительский актор, фактически происходят асинхронно.

Стратегии супервизора.

Последняя часть актора – это стратегии для обработки ошибок его дочерних акторов. Akka прозрачно обрабатывает ошибки, применяя одну из стратегий для каждого входящего отказа. Поскольку эта стратегия является фундаментальной, она не изменяется после создания актора.

Завершение работы актора.

Как только актор завершает работу, то есть происходит сбой, который не обрабатывается при перезапуске, останавливается сам или останавливается своим супервизором, он освобождает свои ресурсы, сливая все оставшиеся сообщения из своего почтового ящика в системный «dead letter mailbox», который перешлет их EventStream как DeadLetters. Почтовый ящик затем заменяется в справочнике субъекта системным почтовым ящиком, перенаправляя все новые сообщения в EventStream как DeadLetters.

7.7 Ссылки акторов, пути и адреса

На рисунке 13 изображены отношения между наиболее важными объектами в системе акторов. Системы акторов – actor systems, формируют внутренние иерархии надзора, а связь между акторами прозрачна и понятна в отношении их размещения в узлах сети.

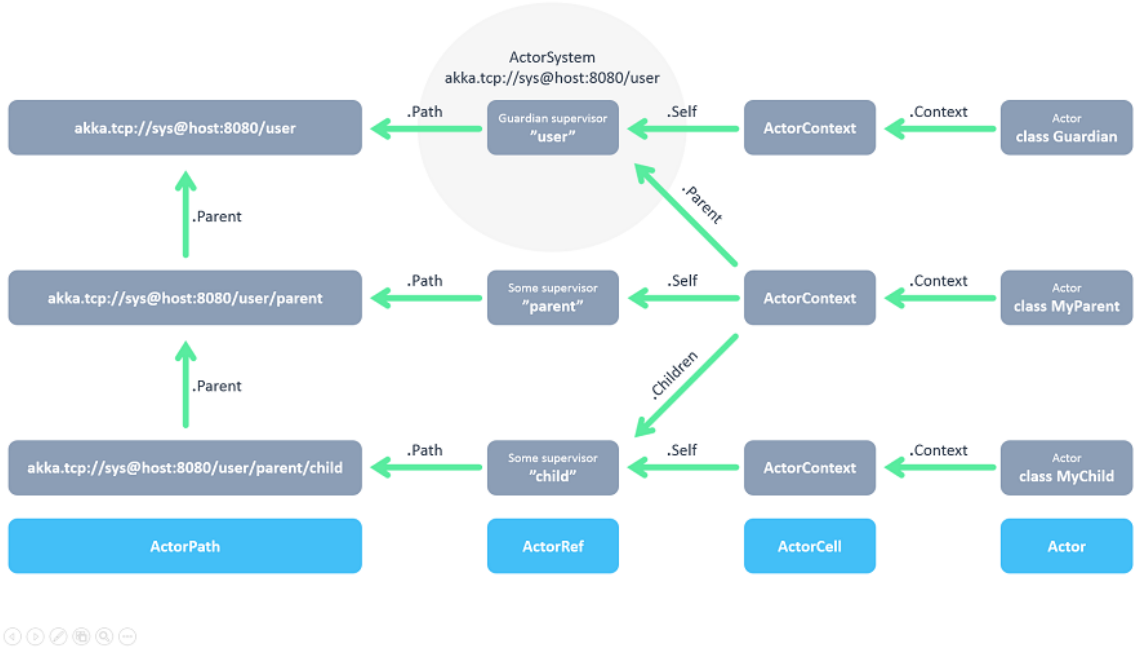

Рисунок 13 – Отношения между наиболее важными объектами в системе акторов [31]

Ссылка на актор – это подтип ActorRef, основной целью которого является поддержка отправки сообщений актору. Каждый актор имеет доступ к своей канонической (локальной) ссылке через свойство Self; эта ссылка также включена в качестве ссылки отправителя по умолчанию для всех сообщений, отправляемых другим акторам. Во время обработки сообщения актор имеет доступ к ссылке, представляющей отправителя текущего сообщения через метод отправителя [31].

Путь актора.

Акторы создаются иерархическим образом, существует некая уникальная последовательность имен акторов, которые следуют рекурсивно по связям между дочерними и родительскими объектами вплоть до корня системы. Данную последовательность можно сравнить с вложенными папками в файловой системе.

Элементы пути актора являются именами пройденных акторов, разделенные косой чертой.

Ссылка на актор обозначает одного актора, и жизненный цикл ссылки соответствует жизненному циклу этого актора; путь актора представляет собой имя, которое может быть заселено или не заселено актором, а сам путь не имеет жизненного цикла, он никогда не становится недействительным. Можно создать путь актора без создания актора, но нельзя создать ссылку на актор без создания соответствующего актора.

Возможно создать актор, завершить его, а затем создать новый актор с таким же путем. Недавно созданный актор – это новое воплощение актора, это не тот же актор. Ссылка актора на старый актор недействительна на новый актор, сообщения, отправленные по старой ссылке актора, не будут доставлены новому актору, даже если они имеют одинаковый путь.

Путь актора имеет компонент адреса, описывающий протокол и месторасположение, по которому достижим соответствующий актор, за которым следует имена акторов в иерархии от корня вверх, например, akka://my-system/user/service-a/worker1.

Логические пути акторов.

Уникальный путь, полученный по ссылкам родительского контроля к корневому хранителю, называется логическим путем актора. Данный путь совпадает с созданием предков актора, поэтому он становится полностью детерминированным, после того как будет определена настройка удаленного взаимодействия системы акторов, а также компонент адреса пути.

Физические пути акторов.

Хотя логический путь актора описывает функциональное расположение в одной системе акторов, удаленное развертывание на основе конфигурации означает, что актор может быть создан на другом хосте сети чем его родительский актор, то есть в другой системе. В таком случае путь актора проходит от корневого хранителя и влечет за собой обход сети, такая операция дорогостоящая. Таким образом, каждый актор имеет физический путь, начиная от корневого хранителя системы акторов, где находится

действующий объект актора. Использование данного пути в качестве ссылки на отправителя, когда запрашиваются другие акторы, позволит им отвечать непосредственно этому актору.

Также существует такая особенность, что физический путь актора никогда не охватывает несколько систем акторов или CLR. Это значит, что логический путь и физический путь актора могут отличаться, если один из предков находится под удаленном контролем.

7.8 Почтовые ящики

В Akka.NET почтовые ящики содержат сообщения, предназначенные для актора. Когда сообщение посылается актору, оно не отправляется непосредственно актору, а отправляется в почтовый ящик актора, до тех пор, пока актор его не обработает.

По сути, почтовый ящик можно описать как очередь сообщений. Затем сообщения как правило доставляются из почтового ящика по одному в том порядке, в котором они были получены, но существует реализации, которые могут изменить порядок доставки.

Обычно каждый актор имеет свой собственный почтовый ящик, но это не является обязательным требованием. Существуют реализации маршрутизаторов, в которых все маршруты совместно используют один почтовый ящик [32].

Можно настроить актор, чтобы он использовал определенный почтовый ящик, в коде или конфигурации. На рисунке 14 изображен пример определения почтового ящика в коде.

```
Props.Create<ActorType>().WithMailbox("my-custom-mailbox");
```

Рисунок 14 – Определение почтового ящика в коде [32]

На рисунке 15 изображен пример определение почтового ящика в конфигурации.

```
akka.actor.deployment {
   /my-actor-path {
       mailbox = my-custom-mailbox
   }
}
```

Рисунок 15 – Определение почтового ящика в конфигурации [32]

Чтобы использовать настраиваемый почтовый ящик, он должен быть сначала настроен с ключом, который система поиска может найти. Сделать это можно с помощью пользовательской настройки HOCON [32].

7.9 Доставка сообщений

Есть несколько правил по отправке сообщений в Akka.NET:

– Доставка осуществляется максимум один раз, т.е. доставка не гарантируется.

– Порядок сообщений поддерживается для пары отправитель и получатель.

Разберём данные правила подробнее.

Доставка осуществляется максимум один раз, также можно сказать не более одного раза, означает, что сообщение доставляется либо один раз, либо ни разу, если сказать иначе, то сообщения могут быть потеряны, но никогда не будут дублироваться. Такой вариант самый дешевый, с самой высокой производительностью и наименьшими накладными расходами на реализацию, потому что мы отправили сообщения и забыли о них, без сохранения состояния на передающем конце или в транспортном механизме.

Если взять другие сценарии, когда попыток доставки сообщения может быть несколько, сообщения могут дублироваться, в данном случае необходимо сохранять состояния на передающей стороне, а также нужно будет реализовать механизм подтверждения на принимающей стороне. Возможен также и другой механизм, когда мы определяем только однократную доставку, то есть сообщение не может быть потеряно или продублировано. Такой сценарий самый дорогой, имея при этом худшую производительность из данных, в дополнение ко второму сценарию необходимо разработать также и сохранение состояния на принимающей стороне, чтобы отфильтровать дублирующиеся доставки.

На самом деле определить, когда сообщение является гарантированно доставленным, сложно и тут нужно разбираться. Можно по-разному считать доставку гарантированной: когда сообщение отправлено по сети, когда оно получено, когда поместилось в почтовый ящик принимающего актора, когда началась обработка сообщения актором, когда сообщение успешно обработалось целевым актором. Поэтому Akka.NET не гарантирует доставку сообщений, разработчик должен сам определить гарантию доставки, и он должен знать и понимать, что в данной области считается успехом, поскольку он понимает специфику конкретной области.

Следующим правилом является порядок сообщений. Оно состоит в том, что для пары участников сообщения, отправленные непосредственно от первого ко второму не будут получены не по порядку. Эта гарантия применяется только при отправке с оператором tell напрямую в конечный

пункт назначения, но данная гарантия не относится к случаям, когда используется посредник.

Если:

– Актор A3 посылает сообщения M4, M5, M6 к A2.

– Актор A1 посылает сообщения M1, M2, M3 к A2.

Это значит, что:

– Если M1 доставлено, оно должно быть доставлено до M2 и M3.

– Если M2 доставлено, оно должно быть доставлено раньше M3.

– Если M4 доставлено, оно должно быть доставлено до M5 и M6.

– Если M5 доставлено, оно должно быть доставлено раньше M6.

– A2 можно увидеть сообщения от A1 чередуя с сообщениями от A3.

– Поскольку нет гарантированной доставки, любое из сообщений может быть отброшено, т.е. не доставлено к A2 [33].

Стоит упомянуть, что сообщения могут быть объектами любого типа, но они должны быть неизменными. Акка не может обеспечить неизменность (пока), так что это должно быть по соглашению [34]. Пример неизменяемого сообщения изображен на рисунке 16. Нужно помнить, что порядок уведомлений о сбоях относительно пользовательских сообщений не является детерминированным. Родитель может перезапустить своего потомка, прежде чем он обработает последние сообщения, отправленные потомком перед ошибкой [35].

```
public class ImmutableMessage
{
    Ссылок: 0
    public ImmutableMessage(int number, List<string> values)
    {
        Number = number;
        Values = values.AsReadOnly();
    }

    ссылка: 1
    public int Number { get; }
    ссылка: 1
    public IReadOnlyCollection<string> Values { get; }
}
```

Рисунок 16 – Пример неизменяемого сообщения

7.10 Диспетчеры

Диспетчеры несут ответственность за планирование всего кода, который выполняется внутри ActorSystem. Диспетчеры являются одной из наиболее важных частей Akka.NET, поскольку они контролируют пропускную способность и долю времени для каждого из участников, предоставляя каждому справедливую долю ресурсов [36].

Все акторы по умолчанию имеют общий Global Dispatcher. Если не конфигурацию, этот диспетчер будет использовать .NET Thread Pool, который оптимизирован для большинства распространенных сценариев, а это в принципе означает, что по умолчанию такая настройка будет достаточно хорошей для большинства случаев.

Диспетчер вызывают акторы, вталкивает сообщения в почтовый ящик актора. Тип диспетчера определяется моделью потоков, использующаяся для передачи сообщений. Многие акторы могут получать сообщения, которые поступают из различных потоков. При отправке сообщения актору, оно отправляется в диспетчер, который спустя какое-то время поместит сообщение в почтовый ящик актора [22].

7.11 Маршрутизаторы

Маршрутизатор (router) представляет собой особый тип актора, чья работа состоит в том, чтобы направлять сообщения на другие актеры, называемых маршрутами. Различные маршрутизаторы используют разные стратегии для эффективной маршрутизации сообщений [37].

Маршрутизаторы можно использовать как внутри, так вне актора, можно самостоятельно управлять маршрутами или использовать маршрутизатор с возможностями конфигурации, а также динамически изменять размер под нагрузкой.

Как правило, любое сообщение, отправленное на маршрутизатор, будет перенаправлено на один из его маршрутов, но есть одно исключение. Специальное широковещательное сообщение будет отправлено на все маршруты [37].

Маршрутизаторы можно развернуть несколькими способами, используя код или конфигурацию.

Пример развертывание кода изображен на рисунке 17, создаётся 5 рабочих (workers), используя маршрутизатор round robin.

```
var props = Props.Create<Worker>().WithRouter(new RoundRobinPool(5));
var actor = system.ActorOf(props, "worker");
```

Рисунок 17 – Пример развертывания кода [37]

На рисунке 18 изображен пример развертывания конфигурации, определив секцию в HOCON, создаём актор используя FromConfig.Instance

```
akka.actor.deployment {
  /workers {
    router = round-robin-group
    routees.paths = ["/user/workers/w1", "/user/workers/w2", "/user/workers/w3"]
  }
}
```

```
var props = Props.Create<Worker>().WithRouter(FromConfig.Instance);
var actor = system.ActorOf(props, "workers");
```

Рисунок 18 – Пример развертывания конфигурации

Существует два типа маршрутизаторов:

– Pools или пулы. Данные маршрутизаторы создают своих собственных рабочих акторов, то есть указывается количество экземпляров в качестве параметра для маршрутизатора, и он будет сам обрабатывать маршрут.

– Groups или группы. Иногда нужно, чтобы актор маршрутизатор не создавал свои маршруты, а создавать маршруты самостоятельно и предоставлять их маршрутизатору для его использования. Это можно сделать, передав пути маршрутов в конфигурацию маршрутизатора. Сообщения будут отправлены с ActorSelection этими путями.

Маршрутизаторы были реализованы как акторы, таким образом маршрутизаторы контролируются родителями, и они могут контролировать дочерних акторов.

Группы используют маршруты, которые были созданы где-то ещё, у него нет собственных дочерних элементов. Если маршрутизатор умирает, group router не будет знать об этом.

С другой стороны, пулы создают своих собственных детей. Таким образом маршрутизатор становится супервизором.

Существует несколько стратегий маршрутизации:

– RoundRobin изображен на рисунке 19. RoundRobinPool и RoundRobinGroup являются маршрутизаторами, которые отправляют сообщения на маршруты в циклическом, по круговому порядку. Это самый простой способ распространять сообщения среди нескольких акторов с наилучшими усилиями.

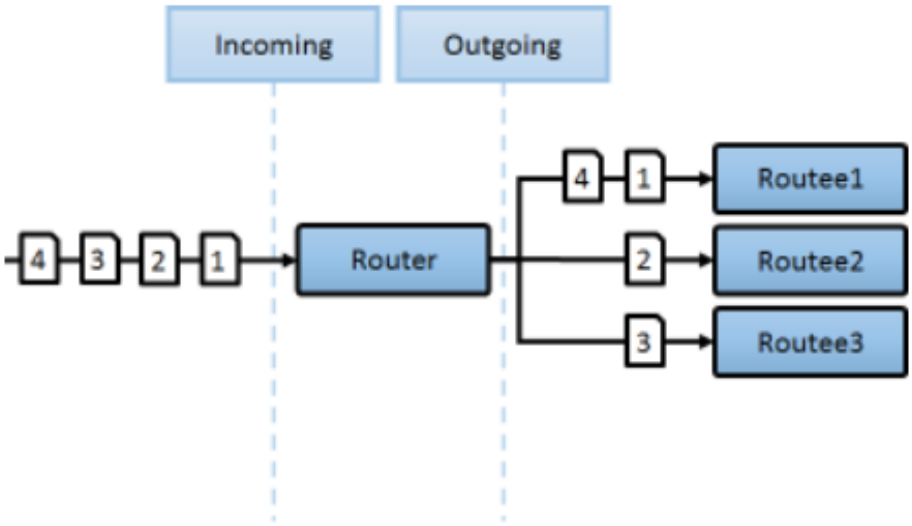

Рисунок 19 – Маршрутзатор RoundRobin [37]

– Broadcast изображен на рисунке 20. Маршрутизаторы BroadcastPool и BroadcastGroup вещают любые сообщения для всех своих маршрутов. На рисунке изображен маршрутизатор Broadcast.

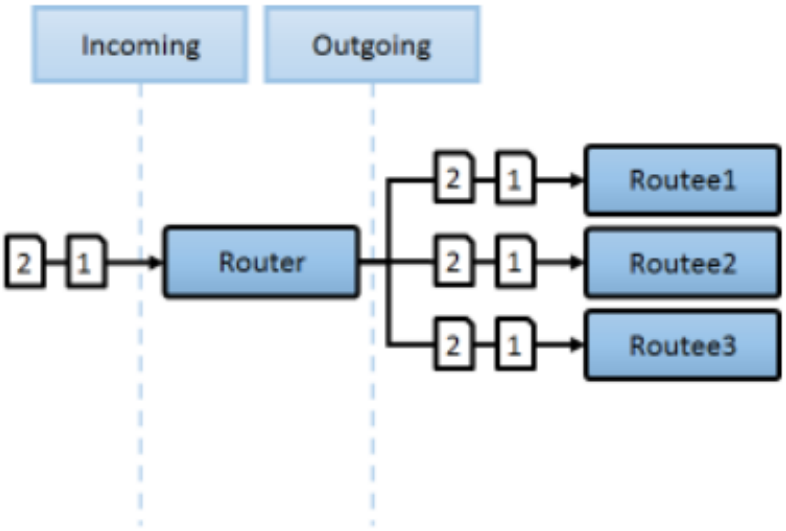

Рисунок 20 – Маршрутизатор Broadcast [37]

– Random. Маршрутизаторы RandomPool и RandomGroup будут пересылать сообщения к маршрутам в случайном порядке.

– ConsistentHashing изображен на рисунке 21. Маршрутизаторы ConsistentHashingPool и ConsistentHashingGroup используют алгоритм согласованного хеширования для того, чтобы выбрать маршрут для отправки сообщения. Идея состоит в том, что сообщения с одним и тем же ключом,

пересылаются по тому маршруту у которого будет тот же ключ. Сам ключ может быть любым объектом .NET, хотя обычно это число, строка или Guid.

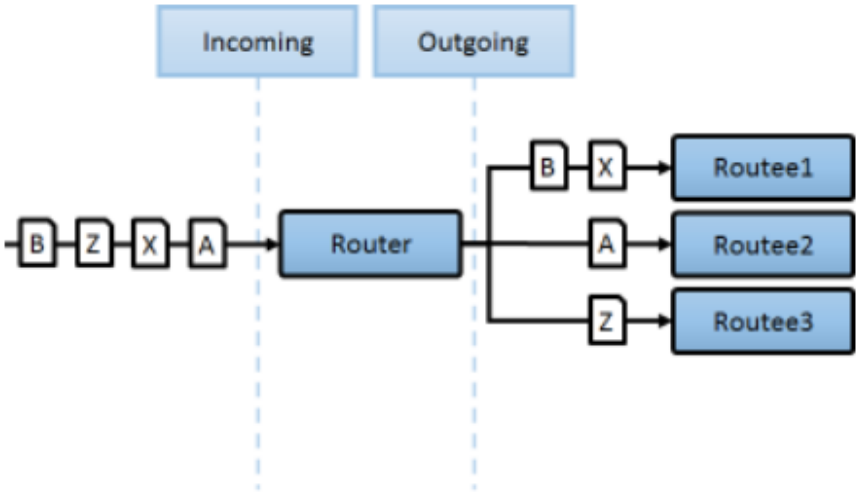

Рисунок 21 – Маршрутизатор ConsistentHashing [37]

– TailChopping. Маршрутизаторы TailChoppingPool и TailChoppingGroup передают сообщения случайному маршруту, и, если ответ не будет получен после заданной задержки, отправят сообщение в другой случайно выбранный маршрут. Ожидается первый ответ от любого из маршрутов и пересылается обратно исходному отправителю, другие же ответы будут отбрасываться. Если по истечению указанного интервала ответ не будет получен, сгенерируется ошибка тайм-аута.

– ScatterGatherFirstCompleted изображен на рисунке 22. ScatterGatherFirstCompletedPool и ScatterGatherFirstCompletedGroup маршрутизаторы транслируют сообщения всем маршрутам и получает обратно первый ответ. Все остальные ответы отбрасываются. На рисунке изображен маршрутизатор ScatterGatherFirstCompleted.

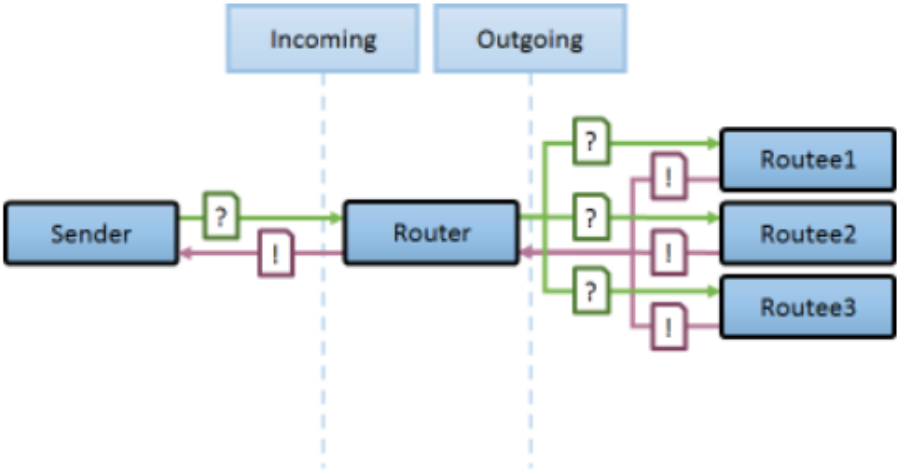

Рисунок 22 – Маршрутизатор ScatterGatherFirstCompleted [37]

– SmallestMailbox изображен на рисунке 23. Маршрутизатор SmallestMailbox отправляет сообщение маршруту с наименьшим количеством сообщений в почтовом ящике. Выбор осуществляется в следующем порядке:

1. Выбирается любой маршрут, который не обрабатывает сообщения с пустым почтовым ящиком.
2. Выбирается любой маршрут с пустым почтовым ящиком.
3. Выбирается маршрут с наименьшим количеством ожидающих обработки сообщений в почтовом ящике.
4. Выбирается любой удалённый маршрут, удалённые акторы считаются с самым низким приоритетом, так как их размер почтового ящика неизвестен.

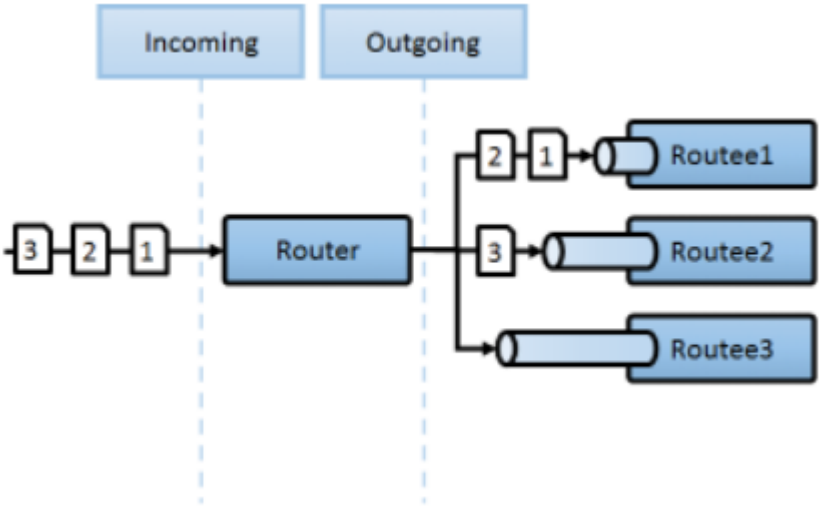

Рисунок 23 – Маршрутизатор SmallestMailbox [37]

Если рассматривать маршрутизаторы поверхностно, то они выглядят как обычные акторы, но на самом деле они реализованы по-другому. Маршрутизаторы призваны быть очень эффективными при получении сообщений и их быстрой передаче по маршрутам.

7.12 Надзор в Akka.NET

Надзор описывает отношения зависимости между субъектами: руководитель делегирует задачи подчиненным и, следовательно, должен реагировать на их ошибки. Когда подчиненный обнаруживает сбой (то есть происходит исключение), он приостанавливает себя и всех своих подчиненных и отправляет сообщение своему руководителю, сигнализируя о сбое. В зависимости от характера работы, подлежащей контролю, и характера отказа, руководитель имеет выбор из следующих четырех вариантов:

– продолжить работу подчиненного, сохраняя его накопленное внутреннее состояние;

– перезапустить подчиненного, очистив его накопленное внутреннее состояние;

– остановить подчиненного навсегда;

– эскалация сбоя к следующему родительскому элементу в иерархии, тем самым сообщить о сбое [38].

Существует две стратегии наблюдения:

– OneForOneStrategy, которая изображена на рисунке 5.

– AllForOneStrategy, которая изображена на рисунке 6

Разница между ними в том, что первая применяет полученную директиву только к дочернему актору, тогда как вторая применяет ее также к братьям и сестрам. Обычно используется OneForOneStrategy, которая является стратегией по умолчанию. Стратегия OneForOneStrategy изображена на рисунке 24.

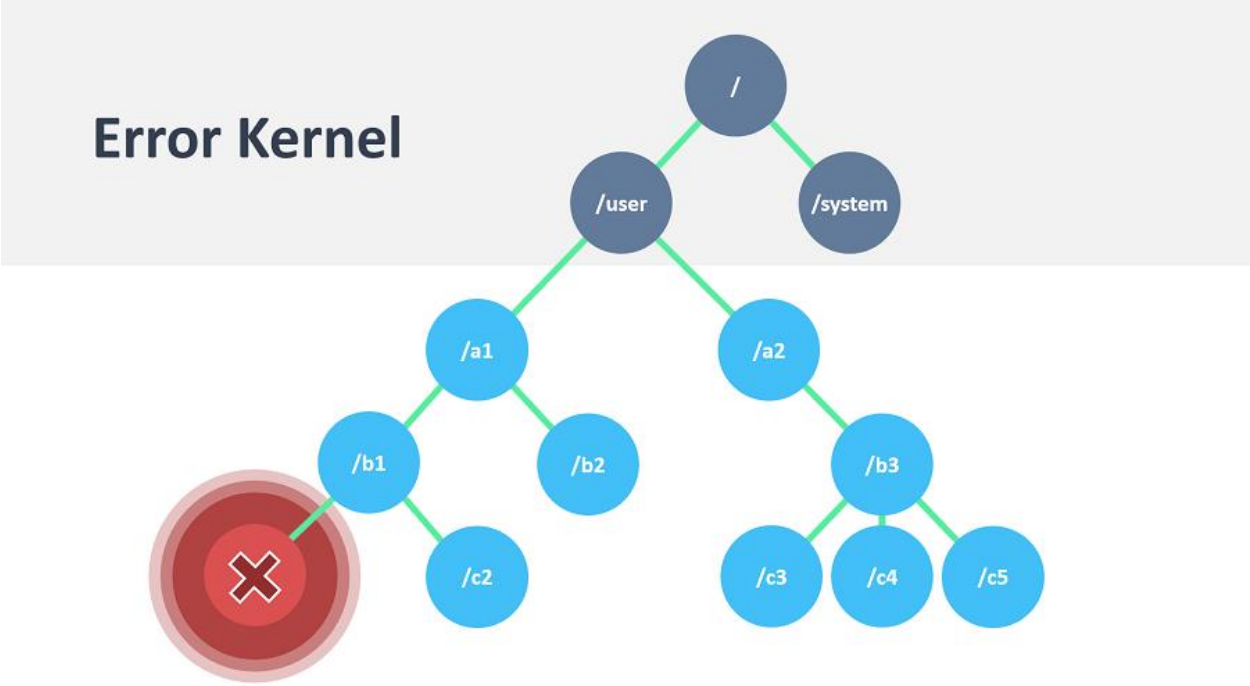

Рисунок 24 – Стратегия OneForOneStrategy [38]

Стратегия AllForOneStrategy применяется в тех случаях, когда дочерние акторы очень взаимосвязаны, все зависят друг от друга. Стратегия AllForOneStrategy изображена на рисунке 25.

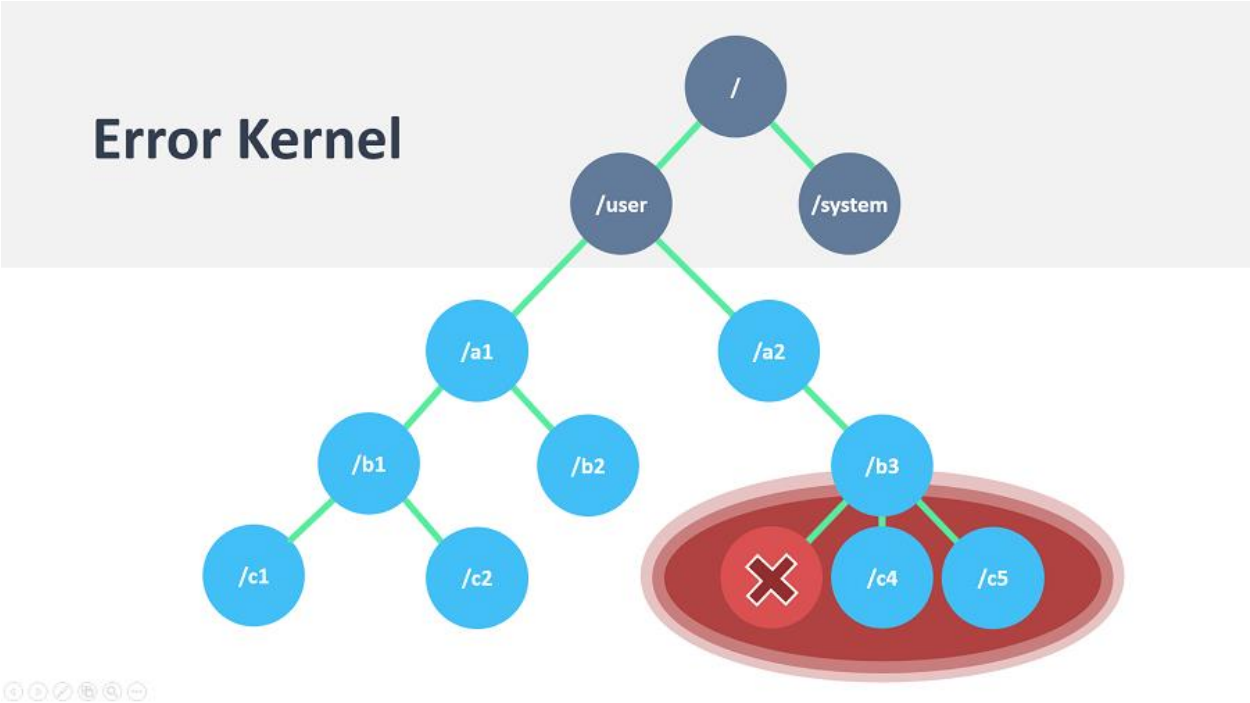

Рисунок 25 – Стратегия AllForOneStrategy [38]

7.13 Конфигурация в Akka.NET и тестирование систем акторов

Akka.NET использует формат конфигурации, называемый HOCON, чтобы позволить настраивать приложения Akka.NET с любым уровнем детализации.

HOCON (Human-Optimized Config Object Notation) – это гибкий и расширяемый формат конфигурации. Он позволяет настраивать: ведение журналов, сетевые транспорты и (как правило) развертывание отдельных акторов [39].

HOCON позволяет встраивать легко читаемую конфигурацию в App.config и Web.config. HOCON также позволяет запрашивать конфигурации по путям их разделов.

HOCON также позволяет вкладывать или объединять разделы конфигурации, создавая при этом слои детализации.

HOCON обычно используется для настройки параметров ведения журнала, включения специальных модулей, таких как Akka.Remote, или настройки развертываний, таких как Dispatcher или Router используемый для конкретного субъекта [39].

HOCON можно использовать в коде как string, но для динамического изменения HOCON можно использовать внутри App.config и Web.config.

Как и в другом программном обеспечении, автоматизированные тесты составляют важный элемент всей разработки. У Akka.NET существует выделенный модуль Akka.TestKit для поддержки тестов на разных уровнях.

Данный модуль позволяет тестировать акторы в среде, которая контролируема и реалистична. Модуль состоит из набора инструментов, которые делают задачу тестирования легкой для разработчиков.

7.14 Akka.Remote

Модуль Akka.Remote, пожалуй, является одним из самых мощных из дополнительных модулей, которые содержатся в Akka.NET. Именно этот модуль предоставляет возможность создавать системы акторов

Некоторые из возможностей Akka.Remote:

– Прозрачное расположение с RemoteActorRef – можно написать код, который будет выглядеть так, как будто он взаимодействует с локальными акторами, но, чтобы акторы начали взаимодействовать с удаленными акторами достаточно всего лишь нескольких настроек в конфигурации.

– Удаленная адресация – Akka.Remote расширяет Address и ActorPath компонента Akka.NET, включая информацию о том, как подключиться к удаленным процессам через ActorSelection.

– Удаленное развертывание – удаленное развертывание акторов с помощью метода ActorOf на удаленных экземплярах системы акторов в любой точке сети.

– Удаленный обмен сообщениями – отправление сообщений акторам, которые находятся на удаленных серверах.

– Несколько сетевых транспортов – Akka.Remote поддерживает не только TCP, но и другие сторонние транспорты.

Примеры использования Akka.Remote:

– Клиентские приложения WPF, WinForms с требованиями к двусторонней связи с удаленными серверами.

– Межсерверные приложения.

– И другие.

В основном данный модуль служит каналом для Akka.Cluster и других модулей высокой доступности.

Транспорт – относится к фактическому сетевому транспорту, такому как TCP или UDP. По умолчанию Akka.Remote использует транспорт TCP DotNetty, но также можно написать свой собственный транспорт и использовать. Адрес – это понятие относится к комбинации IP-адреса и порта, как и любой другой протокол с поддержкой IP. Также можно использовать имя хоста вместо IP-адреса, но имя хоста должно быть сначала преобразовано в IP-адрес. Конечная точка – это конкретная привязка адреса для транспорта. Если открыть транспорт TCP в localhost:8080 то, создастся конечная точка для этого транспорта по этому адресу. Ассоциация – это связь между двумя конечными точками, каждая из которых принадлежит к разным системам акторов.

Необходимо иметь действительную исходящую конечную точку и действительную входящую конечную точку для создания ассоциации [40].

Для того, чтобы работать с удаленной системой акторов, необходимо знать адрес для этой системы акторов.

У всех локальных акторов есть адрес, как часть пути актора. На рисунке 26 изображен локальный путь актора.

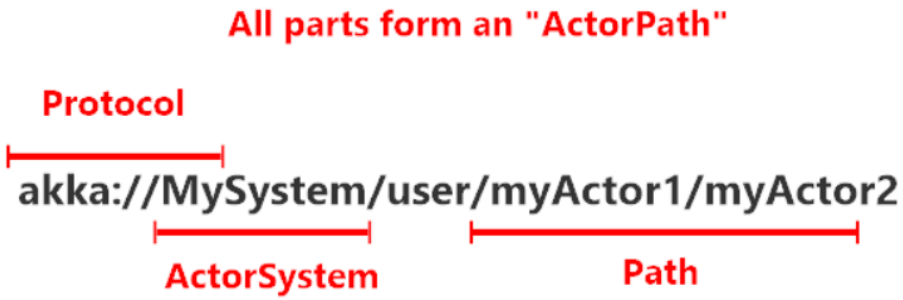

Рисунок 26 – Локальный путь актора [40]

На рисунке 27 изображен удаленный путь актора.

Рисунок 27 – Удалённый путь актора [40]

Как уже говорилось выше, в Akka.Remote существует такое понятие как транспорт. Транспорты в Akka.Remote являются абстракциями поверх реальных сетевых транспортов, таких как сокеты TCP и UDP, а на самом деле транспорты имеют довольно простые требования [41].

На рисунке 28 изображен пример того, как работает удаленное развертывание.

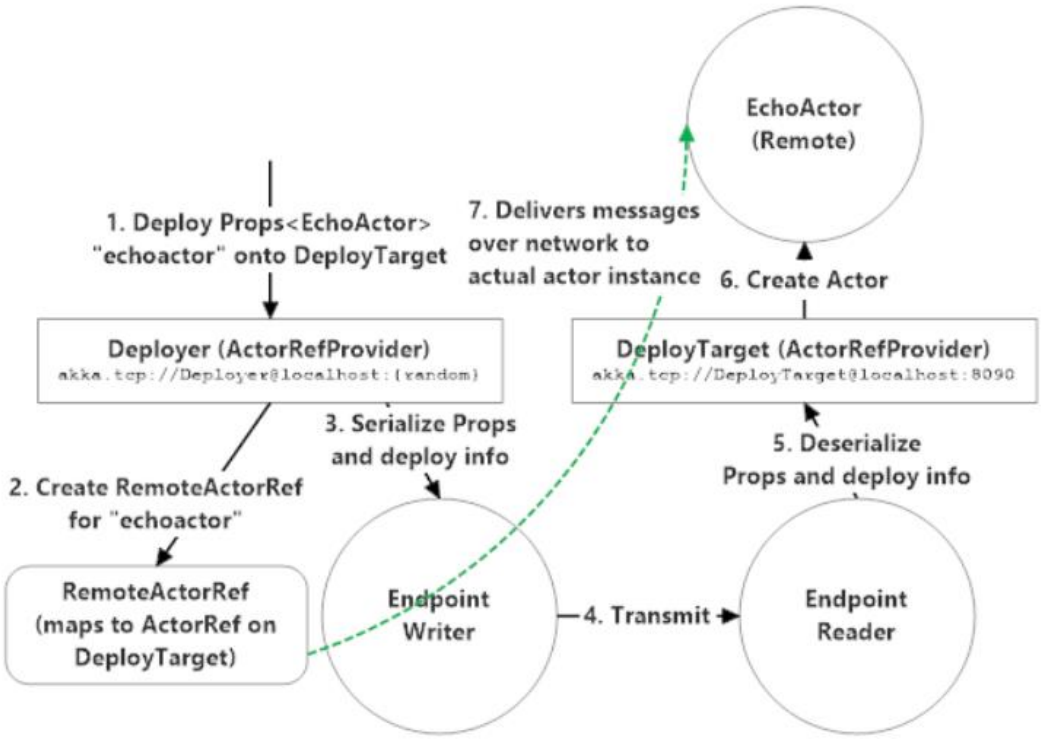

Рисунок 28 – Пример удаленного развёртывания [42]

При успешном установлении связи между удаленными машинами будут выполнены действия: локальное имя для актора, который развертывается удаленно, будет зарезервировано в Deployer, поскольку у всех акторов должно быть уникальное имя, далее объект Props для «EchoActor» будет сериализован, включая все аргументы конструктора для «EchoActor» класса, а также все остальное в развертывании актора, например сведения о маршрутизаторе, конфигурацию диспетчера и т.д. Сериализованный объект Props и вся соответствующая информация о пути и имени актера передается по сети к Deployer EndpointWriter и принимается DeployTarget EndpointWriter. EndpointWriter определяет, что это специальное сообщение удаленно развернутого актора, и сообщает специальной системе актеров, чтобы создать новый EchoActor экземпляр. В дальнейшем все сообщения, отправленные на RemoteActorRef, автоматически отправляются на EchoActor [42].

Существуют два распространенных сценария, когда используется удаленное развертывание:

– Распределенные работы – перенос работы на удаленные машины, используя удаленный или кластерный маршрутизатор.

– Доступ к ресурсам, относящимся к конкретной машине – если необходимо собрать данные из массива удаленных машин, чтобы получить конкретные метрики всех машин.

Создавая крупномасштабную систему с использованием Akka.NET, производительность Akka.Remote станет одним из важнейших факторов данного приложения, поскольку этот модуль непосредственно отвечает за

управление скоростью, с которой сообщения перемещаются по одному соединению между двумя узлами [43].

7.15 Akka.Cluster

Кластер представляет собой отказоустойчивую, эластичную, децентрализованную одноранговую сеть приложений Akka.NET без единой точки отказа или узкого места. Akka.Cluster – это модуль, который дает вам возможность создавать эти приложения [44].

Akka.Cluster – пакет, который обеспечивает поддержку кластеризации в Akka.NET, выполняя следующее:

– Упрощение создания одноранговых сетей приложений Akka.NET.

– Поддержка пиров в автоматическом нахождении новых узлов и удалении мертвых узлов без изменений в конфигурации.

– Подписка пользовательских классов на уведомления об изменениях доступности узлов в кластере.

– Вводит понятие «роли» для выделения различных приложений Akka.NET в кластере

– Позволяет создавать кластерные маршрутизаторы, которые представляют собой расширение встроенных маршрутизаторов Akka.NET, но данные маршрутизаторы автоматически корректируют свой список маршрутов в зависимости от доступности узлов в кластере.

Преимущества Akka.Cluster:

– Отказоустойчивость: кластеры изящно восстанавливаются после сбоев

– Эластичность: кластеры по своей природе эластичны и могут масштабироваться по мере необходимости

– Децентрализованный: возможно иметь несколько равных реплик данного микросервиса или части состояния приложения, которые работают одновременно в кластере.

– Одноранговый: новые узлы могут связываться с существующими одноранговыми узлами, получать уведомления о других одноранговых узлах и полностью интегрироваться в сеть без каких-либо изменений конфигурации.

– Нет единой точки отказа или узкого места: несколько узлов могут обслуживать запросы, увеличивая при этом пропускную способность и отказоустойчивость всего приложения.

Когда использовать Akka.Cluster:

– Значительная нагрузка трафика.

– Нетривиально выполнить.

– Ожидание быстрого времени отклика.

– Потребность в эластичном масштабировании, когда имеется большая нагрузка.

– Архитектура микросервисов.

Где используется Akka.Cluster:

– Аналитика.

– Многопользовательские игры.

– Системы оповещения и мониторинг.

– Отслеживание устройств, интернет вещей.

– Динамическое ценообразование.

– И многое другое.

Node – узел: логический член кластера. Узел определяется адресом, по которому он доступен (hostname:port tuple). Поэтому одновременно может существовать несколько узлов на одном компьютере. Также существует такое понятие, как Seed Nodes или начальные узлы. Начальный узел представляет собой общеизвестную точку контакта, к которому новый узел должен обратиться, чтобы присоединиться к кластеру. Начальные узлы функционируют как механизм обнаружения сервисов Akka.Cluster.

Cluster – кластер: набор, которые соединены через службу членства. Несколько приложений Akka.NET могут быть частью одного кластера.

Gossip – сплетня: основные сообщения, питающие сам кластер.

Leader – лидер: один узел в кластере, который добавляет или удаляет узлы из кластера.

Role – роль: именованное приложение в кластере. В кластере может быть несколько приложений Akka.NET и каждое со своей ролью, узел может быть в нескольких ролях сразу.

Convergence – конвергенция: когда кворум (простое большинство) сообщений о сплетнях соглашается на изменение состояния члена кластера в системе.

Cluster Gossip.

Эта одна из самых важных концепций Akka.Cluster. Именно этому узлы могут присоединяться и выходить из кластера без каких-либо изменений в конфигурации.

Сплетни или gossip – это непрерывный поток сообщений, которые передаются между узлами в кластере, обновляя элементы кластера в состоянии каждого члена кластера.

Когда узел хочет присоединиться к кластеру, он должен сначала связаться с одним из сконфигурированных начальных узлов. Как только узел сможет подключиться к одному начальному узлу, он начнет получать сообщения о сплетнях, содержащие информацию о других членах кластера [44]. На рисунке 29 изображены кластерные сплетни.

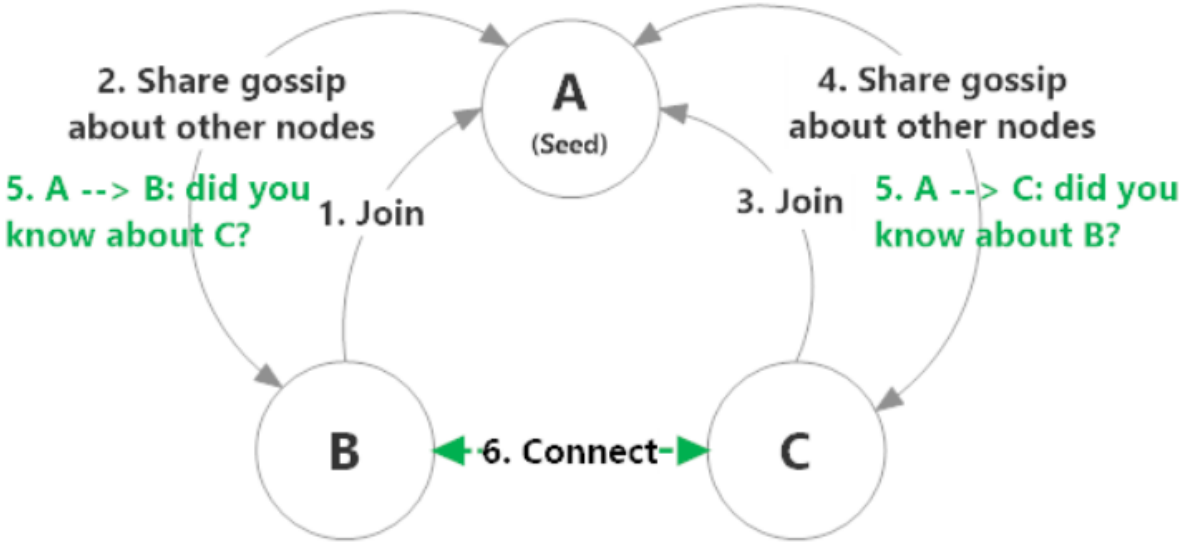

Рисунок 29 – Кластерные сплетни [44]

В приведенном выше примере происходят следующие события:

1. Узел B связывается со своим сконфигурированным начальным узлом A (seed) и даёт запрос на присоединение к кластеру.
2. Узел A помечает узел B как принятый в кластер и начинает обмениваться сплетнями с узлом B, а также о других узлах в кластере, но пока больше других узлов нет.
3. Узел C контактирует с узлом A и даёт запрос на присоединение в кластер.
4. Узел A приветствует узел C в кластере и начинает обмениваться слухами с узлом C.
5. Узел B и узел C оба получают информацию о каждом из них узлом A.
6. Узел B и узел C присоединяются к друг другу налаживают связь.

Сообщения о сплетнях будут регулярно появляться в кластере, каждый раз при изменениях узлов в кластере.

На самом деле разработчик не будет взаимодействовать с сообщениями о сплетнях на уровне приложения. Но это необходимо знать о них и знать, что они питают кластер.

Akka.Cluster расширяет возможности обоих Pool и Group маршрутизаторов для работы во всех кластерах приложений Akka.NET и может автоматически добавлять или удалять маршруты, когда новые узлы присоединяются и выходят из кластера [45].

8 Применение Akka.NET в приложении

Заключительным этап в разработке системы было применение фреймворка Akka.NET и его модуля Akka.Remote. Причем очень успешно, удалось использовать концепцию модели акторов и построить конкурентную и распределённую систему. За счёт этого повысилось отказоустойчивость, надёжность и главная цель работы – производительность.

На рисунке 30 изображена схема общей работы приложения с Akka.NET.

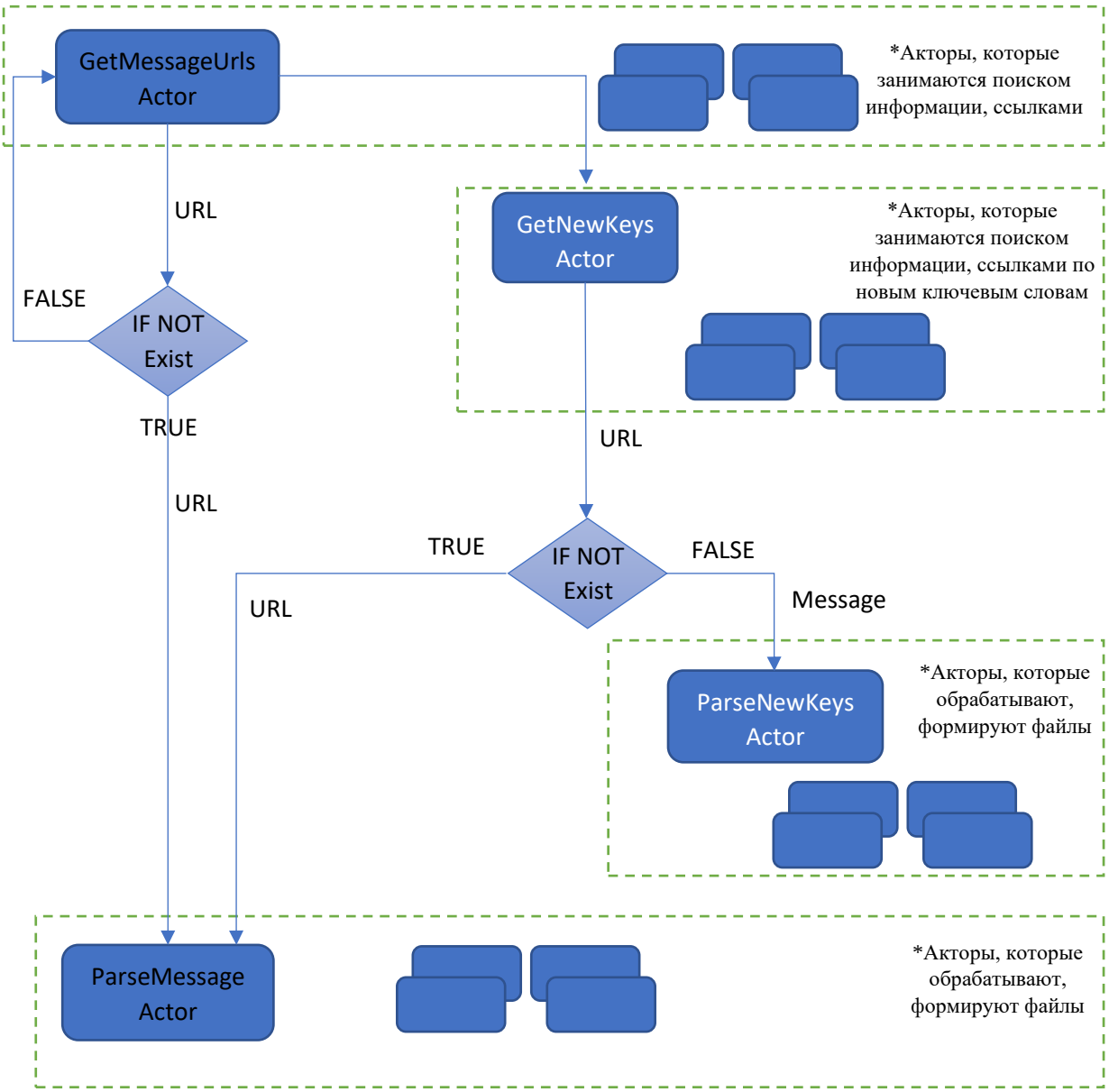

Рисунок 30 – Схема работы приложения с Akka.NET

На данной схеме изображена работа акторов которые участвуют в поиске и обработке данных и файлов. Стоит отметить, так как для поиска есть

пять интернет-ресурсов, то получается и реализовано по пять акторов в каждой функции, соответственно, что в каждой подсхеме присутствуют пять акторов.

Первыми начинают работать акторы, которые на схеме обозначены как GetMessageUrlsActor, после поиска они передают акторам ParseMessageActor – URL, если он не существует в текущей базе данных. Далее акторы обрабатывают файлы и данные. После отправки всех ссылок акторы GetMessageUrlsActor запускают акторы GetNewKeysActor, они также ищут данные на тех же ресурсов, но их задача делать это по новым ключевым словами, если URL не существует в базе, то URL обрабатывается акторами под названием ParseMessageActor, иначе формируется объект Message, который содержит информацию о файлах, новом ключевом слове и другой информации, и отправляется ParseNewKeysActor акторам для того, чтобы они отметили в созданных файлах новые ключевые слова.

В Akka.NET существует два базовых типа акторов: ReceiveActor и UntypedActor. В данной разработке использовался тип ReceiveActor, так как он позволяет легче обрабатывать сложные сопоставления и делать обработку сообщений.

9 Сравнение производительности приложений

Подводя итоги данной работы, изобразим схему эволюции развития приложения на рисунке 31.

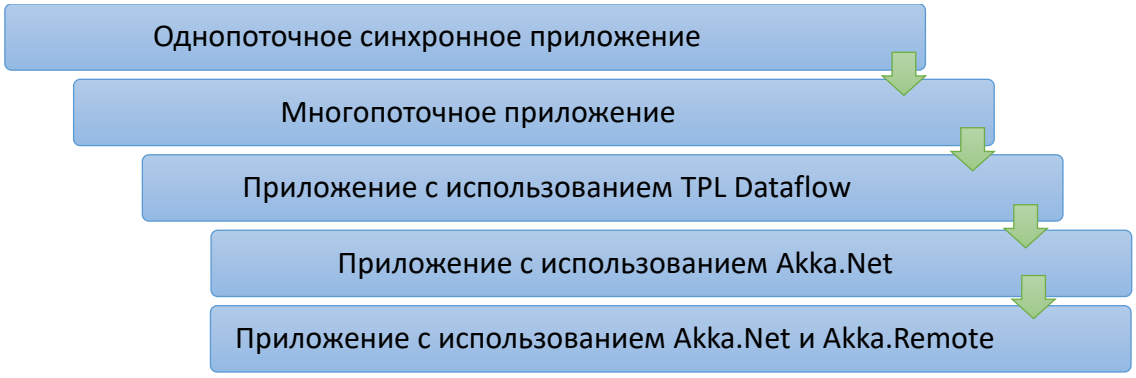

Рисунок 31 – Эволюция развития приложения

Пройдя большой путь от однопоточного приложения до приложения, основанного на модели акторов, с помощью Akka.NET и Akka.Remote.

Для сравнения были запущены приложения в рамках одинакового задания и объеме данных, средний результат изображен на рисунке 32, построена диаграмма сравнения производительности приложений, исходя из того за какое время справились приложения с поставленными целями.

Необходимо отметить, что данные примерные, они приблизительны, потому что файлы бывают различных размеров, что конечно же влияет на количество затраченного времени на обработку файла. Но данная диаграмма даёт необходимое представление о том, как увеличилась производительность.

Рисунок 32 – Диаграмма сравнения производительности

Проанализировав данную диаграмму, можно сделать выводы: многопоточное приложение увеличило производительность, но так как это приложение не было стабильным, потому что работа с потоками велась вручную, происходило множество ошибок и исключений, приложение с использованием TPL Dataflow дало отличный результат, прирост по сравнению с начальным приложением почти в 4 раза, но так как продолжение работы приложения в рамках одного сервера было сдерживающим фактором пришлось перепроектировать далее, производительность приложения с Akka.NET получилась почти такой же как и в приложении TPL Dataflow, но главное было впереди, когда был использован модуль Akka.Remote, что позволило подключить ещё один сервер, а это позволило стать данному приложению ожидаемо самым лучшим по производительности приложению, использовав мощности двух машин, была увеличена производительность в 6 раз.

То есть если допустим предположить, что изначальное однопоточное приложение обработает все файлы за 6 месяцев, то приложение с Akka.Remote обработает примерно за 1 месяц, что значительно сокращает время затраченное на обработку, но если даже и этой мощности не хватит, то без особых усилий можно добавлять другие машины для ускорения процесса, а значит задачи, которые были поставлены в самом начале работы выполнены успешно.

ЗАКЛЮЧЕНИЕ

Результатом данной магистерской диссертации является разработанная система, которая в процессе работы изменялась и дополнялась. Были выполнены все поставленные задачи, удалось решить проблемы с производительностью, увеличить отказоустойчивость системы. Система основана на модели акторов, разработана с помощью фреймворка Akka.NET и его модуля Akka.Remote, теперь при необходимости дальнейшего масштабирования приложения, нет необходимости изменять код, в созданном приложении, нужно лишь настроить конфигурацию. Теперь с лёгкостью можно увеличивать количество компьютеров в системе, чтобы увеличивать производительность.

Были разработаны приложения: однопоточное приложение, многопоточное приложение, приложение с использованием TPL Dataflow, приложение с использованием Akka.NET и Akka.Remote. Проанализированы работа и производительность, было выполнено сравнение производительности данных приложений.

В данной работе было рассмотрены множество аспектов конкурентности, её основные формы и разновидности. Разобраны такие понятия, как многопоточность, параллельная обработка. Была исследована модель акторов, изучен, основанный на данной модели, фреймворк Akka.NET.

Модель акторов является отличной концепцией программирования, когда необходима параллельной обработка. А такой фреймворк как Akka.NET предоставляет данную модель, позволяя разработчикам сосредоточиться на решении задач в концепции акторов. При этом фреймворк обеспечивает высокую производительность: 50 миллионов сообщений в секунду на одной машине, небольшой объем памяти – 2,5 миллиона акторов на гигабайт памяти.

Можно сказать, что все в Akka.NET предназначено для работы в распределённой среде: все взаимодействия акторов происходят через передачу сообщений, при том, что всё является асинхронным в системе. Данные возможности позволяют обеспечивать одинаковую доступность всех функций при работе как на одной машине, так и в кластере из сотен таких машин.

По итогам работы очевидным стало, что использование модели акторов в данном приложении было уместным и просто необходимым. На практике убедившись, что Akka.NET является мощным инструментом в создании таких масштабируемых параллельных систем, которые позволяют добиться отказоустойчивости.

Таким образом, при проведении работы было изучено множество материалов и литературы для исследования данной темы, в частности речь идёт о конкурентности, тема довольно непростая, вокруг неё множество споров, некоторые программисты избегают эту тему, а многие используют с огромным энтузиазмом и радуются успешным результатом. Благо на данный момент существует множество современных инструментов и библиотек, которые помогают в данном вопросе разработчикам. Использовав в работе

современные инструменты были выполнены все задачи. Итогом работы стала успешно работающая система, которая развернута на нескольких машинах.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1 Клеппман М. Высоконагруженные приложения. Программирование, масштабирование, поддержка. – СПб.: Питер, 2018. – 640 с.: ил. – (Серия «Бестселлеры O'Reilly»).

2 Бёрнс Б. Распределенные системы. Паттерны проектирования. – СПб.: Питер, 2019. – 224 с.

3 Микросервисная архитектура. URL: https://ru.wikipedia.org/wiki/Микросервисная_архитектура.

4 Микросервисы (Microservices). URL: https://habr.com/ru/post/249183/.

5 Microservices. URL: https://martinfowler.com/articles/microservices.html.

6 Монолитная vs Микросервисная архитектура. URL: https://proglib.io/p/monolitnaya-vs-mikroservisnaya-arhitektura-2019-09-16.

7 Пара слов в защиту монолита. URL: https://habr.com/ru/ocompany/simbirsoft/blog/453932/.

8 Клири Стивен, Конкурентность в C#. Асинхронное, параллельное и многопоточное программирование.2-е межд.изд. — СПб.: Питер, 2020. – 272 с.: ил. – (Серия «Для профессионалов»).

9 Албахари Джозеф, Албахари Бен. C# 6.0. Справочник. Полное описание языка, 6-е изд. : Пер. с англ. - М. : ООО "И.Д. Вильямс", 2016. – 1040 с.: ил. – Парал. тит. англ.

10 Троелсен Эндрю, Джепикс Филипп. Язык программирования C# 7 и платформы .NET и .NET Core, 8-е изд. : Пер. с англ. — СПб. : ООО "Диалектика", 2018 – 1328 с. : ил. — Парал. тит. англ.

11 Скит, Джон. C42 C# для профессионалов: тонкости программирования, 3-е изд. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2014. – 608 с.

12 Task Parallel Library (TPL). URL: https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl.

13 Dataflow (Task Parallel Library). URL: https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/dataflow-task-parallel-library.

14 Чрезвычайная параллельность. URL: https://ru.wikipedia.org/wiki/Чрезвычайная_параллельность.

15 Task Parallel Library (TPL) Dataflow Constructs. URL: https://csharppedia.com/en/tutorial/3110/task-parallel-library--tpl--dataflow-constructs.

16 Модель акторов. URL: https://ru.wikipedia.org/wiki/Модель_акторов.

17 Параллельные вычисления. URL: https://ru.wikipedia.org/wiki/Параллельные_вычисления.

18 Распределённые вычисления. URL: https://ru.wikipedia.org/wiki/Распределённые_вычисления.

19 Параллельные вычислительные системы. URL: https://ru.wikipedia.org/wiki/Параллельные_вычислительные_системы.

20 Модель Акторов и C++: что, зачем и как? URL: https://habr.com/ru/post/322250/.
21 Erlang. URL: https://ru.wikipedia.org/wiki/Erlang.
22 Раймонд Рестенбург, Роб Баккер, Роб Уильямс. Akka в действии / пер. с англ. А.Н. Киселев – М.:ДМК Пресс,2018. – 522с.: ил.
23 Модели акторов 40 лет. URL: https://habr.com/ru/company/tiktokcoach/blog/206300/.
24 Orleans. URL: https://dotnet.github.io/orleans/.
25 Orleans is a cross-platform framework for building robust, scalable distributed applications. URL: https://dotnet.github.io/orleans/Documentation/index.html.
26 What is Akka.NET? URL: https://getakka.net/articles/intro/what-is-akka.html.
27 Part 1: Top-level Architecture. URL: https://getakka.net/articles/intro/tutorial-1.html.
28 Akka.NET Users and Use Cases. URL: https://getakka.net/articles/intro/akka-users.html.
29 Actor Systems. URL: https://getakka.net/articles/concepts/actor-systems.html.
30 Actors. URL: https://getakka.net/articles/concepts/actors.html.
31 Actor References, Paths and Addresses. URL: https://getakka.net/articles/concepts/addressing.html.
32 Mailboxes. URL: https://getakka.net/articles/actors/mailboxes.html.
33 Part 2: The Device Actor. URL: https://getakka.net/articles/intro/tutorial-2.html.
34 ReceiveActor API. URL: https://getakka.net/articles/actors/receive-actor-api.html.
35 UntypedActor API. URL: https://getakka.net/articles/actors/untyped-actor-api.html.
36 Dispatchers. URL: https://getakka.net/articles/actors/dispatchers.html.
37 Routers. URL: https://getakka.net/articles/actors/routers.html.
38 Supervision. URL: https://getakka.net/articles/concepts/supervision.html.
39 Akka.NET Configuration. URL: https://getakka.net/articles/concepts/configuration.html.
40 Akka.Remote Overview. URL: https://getakka.net/articles/remoting/index.html.
41 Akka.Remote Transports. URL: https://getakka.net/articles/remoting/transports.html.
42 Remotely Deploying Actors. URL: https://getakka.net/articles/remoting/deployment.html.
43 Akka.Remote Performance Optimization. URL: https://getakka.net/articles/remoting/performance.html.
44 Akka.Cluster Overview. URL: https://getakka.net/articles/clustering/cluster-overview.html.

45 Akka.Cluster Routing. URL: https://getakka.net/articles/clustering/cluster-routing.html.