

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования

«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет

вычислительной техники

Кафедра

«Вычислительная техника»

Направление подготовки: 09.04.01 «Информатика и вычислительная техника»
Магистерская программа: «Вычислительные машины, комплексы, системы и сети»

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

на тему

Система обеспечения отказоустойчивого

доступа

Студент


(подпись, дата)

Васильев Никита Николаевич
(ФИО полностью)

Руководитель


(подпись, дата)

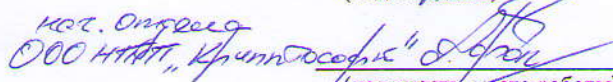
Дорошников И.В.
(фамилия, инициалы)

Нормоконтролёр


(подпись, дата)

А.В. Кучин
(фамилия, инициалы)

Рецензент

нач. отдела
ООО «ИТ-Системы»

(должность, место работы)
(фамилия, инициалы)


(подпись, дата)

Работа допущена к защите (протокол зас. кафедры от 03.06.2020 г № 12)

Заведующий кафедрой


(подпись)

М.А. Митрохин
(фамилия, инициалы)

Работа защищена с отметкой



(протокол заседания ГЭК от 15.06.2020 г № 1)

Секретарь ГЭК


(подпись)

Е.И. Гурин
(фамилия, инициалы)

Пенза 2020

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет
вычислительной техники

Кафедра
«Вычислительная
техника»

«Утверждаю»
Заведующий кафедрой
 М.А.
Митрохин
«08» ноября 2018 г.

**ЗАДАНИЕ
НА МАГИСТЕРСКУЮ ДИССЕРТАЦИЮ**

1. Студент Васильев Никита Николаевич гр. 18ВВм1 факультета вычислительной техники.
2. Тема работы: «Система обеспечения отказоустойчивого доступа». Тема утверждена приказом ПГУ № 1012а/о от 08.11.2018 г.
3. Руководитель работы к.т.н., доцент каф. ВТ ПГУ Дорошенко И.Н.
4. Задание на работу (назначение разработки, исходные данные и т.п.).
Назначение разработки – исследование современных методов обеспечения отказоустойчивости, проектирование и разработка отказоустойчивой системы, работающей в условиях сетевых задержек и разделения сети.
5. Перечень подлежащих разработке вопросов. Требуется изучить различные причины сбоев и отказов, возникающих в информационных системах. Исследовать различные стратегий поддержания работоспособности, противодействия отказов и повышения доступности. Разобрать теоретические основы этих стратегий. Предложить улучшения изученных методов. Спроектировать и реализовать отказоустойчивую систему с высоким уровнем доступности.

6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1.	Изучение общей информации об отказоустойчивости и надежности информационных систем	10	18.12.18	Торосенко
2.	Разбор моделей отказов	10	28.12.18	Торосенко
3.	Изучение причин возникновения сбоев	14	07.01.19	Торосенко
4.	Поиск метрик оценки доступности и отказоустойчивости	14	21.01.19	Торосенко
5.	Изучение основных метрик	14	05.02.19	Торосенко
6	Рассмотрение общих подходов обеспечения отказоустойчивости	12	19.02.19	Торосенко
7	Изучение общих сведений о распределенных системам	15	03.03.19	Торосенко
8	Рассмотрение CAP теорем	8	18.03.19	Торосенко
9	Исследование вопроса достижения консенсуса	18	26.03.19	Торосенко
10	Анализ проблем распределенных систем	18	13.03.19	Торосенко
11	Изучение методов повышения надежности	14	01.04.19	Торосенко
12	Изучение способов обнаружения сбоев в информационных системах	14	15.04.19	Торосенко
13	Исследование методов мониторинга	10	25.04.19	Торосенко
14	Сравнение различных способов мониторинга	10	05.05.19	Торосенко
15	Изучение подходов к тестированию	14	19.05.19	Торосенко
16	Разбор различных методов тестирования систем	14	02.06.19	Торосенко
17	Сравнение типов тестирования	10	16. 06.19	Торосенко

18	Изучение вопроса маскирования сбоев	16	26.07.19	Хороших
19	Рассмотрение алгоритма N-версионного программирования	8	12.07.19	Хороших
20	Изучение групп процессов, как способа маскирования сбоев	18	20.07.19	Хороших
21	Исследование вопроса восстановления систем	20	08.08.19	Хороших
22	Рассмотрение способов применения точек восстановления	22	28.08.19	Хороших
23	Анализ журналирования события	10	20.09.19	Хороших
24	Изучение общих сведений о алгоритме консенсуса Raft	24	30.09.19	Хороших
25	Рассмотрение основных фаз работы алгоритма Raft	14	24.10.19	Хороших
26	Исследование возможности модификации алгоритма Raft	49	07.11.19	Хороших
27	Изучение вопроса маскирования сбоев	14	26.12.19	Хороших
28	Исследование архитектур информационных систем с целью выявления свойств отказоустойчивости и высокой доступности	14	10.01.20	Хороших
29	Выбор архитектур	10	24.01.20	Хороших
30.	Изучение монолитной архитектуры	8	04.02.20	Хороших
31	Изучение плагиной архитектуры	8	12.02.20	Хороших
32.	Изучение сервисной архитектуры	14	20.02.20	Хороших
33	Выбор архитектуры для проектирования отказоустойчивого и высокодоступного приложения	10	03.03.20	Хороших
34	Проектирование архитектуры веб приложения	20	13.03.20	Хороших
35.	Разработка сервиса учетных записей	14	15.03.20	Хороших

36	Разработка сервиса каталогов	14	29.03.20	Дорошенко
37	Разработка сервиса заказов	14	11.04.20	Дорошенко
38	Разработка API шлюза	14	25.04.20	Дорошенко
39	Настройка сервисов для запуска в контейнерах	10	10.05.20	Дорошенко
40	Оформление пояснительной записки	10	20.05.20	Дорошенко

Дата выдачи задания "03" октября 2018 г.

Руководитель магистерской работы Дорошенко И.Н. 09.10.2018
(подпись, дата)

Задание к исполнению принял студент Васильев Н.Н. 09.10.2018
(подпись, дата)

Работу к защите допустить

Декан факультета

Фионова Л.Р.
(подпись, дата)

РЕФЕРАТ

Магистерская диссертация содержит 132 листов, 15 рисунков, 6 таблиц, 5 листингов, 6 приложений.

СИСТЕМА ОБЕСПЕЧЕНИЯ ОТКАЗОУСТОЙЧИВОГО ДОСТУПА, ОТКАЗОУСТОЙЧИВОСТЬ, РАСПРЕДЕЛЕННЫЕ СИСТЕМЫ, RAFT

Цель работы – провести анализ существующих методов и алгоритмов обеспечения отказоустойчивости и доступности информационных систем, выявить достоинства и недостатки. Изучить различные виды возможных сбоев, установить причины их возникновения, выстроить стратегии предотвращения их возникновения и пути восстановления. В итоге, на основе результатов исследования необходимо разработать отказоустойчивую систему с высоким уровнем доступности.

В результате проведенной работы были проанализированы алгоритмы отказоустойчивости на ряду с маскированием отказов, восстановлением из состояния ошибки, обеспечением отказоустойчивости групп. Отдельное внимание уделено проблеме достижения распределенного консенсуса, разобран алгоритм Raft, предложены его модификации для улучшения доступности операций чтения. В дополнение, реализован проект, обладающий устойчивостью к определенному набору сбоев и демонстрирующий свойство высокой доступности. В нем на практике демонстрируется применение результатов, полученных в результате исследования.

Полученные академические результаты могут выступать в качестве фундамента для дальнейших исследований по таким направлениям как: отказоустойчивые системы и модели, согласованность данных, отказоустойчивая природа распределённых систем. Получение выводы представляют практический интерес для реализации отказоустойчивых промышленных систем.

Содержание

Перечень сокращений и обозначений	9
Введение	11
1 Основные положения	13
1.1 Основная информация о надежности	14
1.2 Отказы и сбои	15
1.2.1 Модели отказов	16
1.2.2 Причины отказов и сбоев	18
1.3 Метрики отказоустойчивости	20
1.4 Методы обеспечения отказоустойчивости	21
1.5 Распределенные системы	23
1.5.1 Проблемы распределенных систем	24
1.5.2 CAP теорема	27
1.5.3 Распределенный консенсус	28
1.6 Восстановление	29
2 Обеспечение отказоустойчивости	32
2.1 Обнаружение сбоев	32
2.1.1 Мониторинг	34
2.1.2 Тестирование	36
2.2 Маскирование сбоев	39
2.2.1 N-версионное программирование	39
2.2.2 Группы процессов	40
2.3 Алгоритмы восстановления	42
2.3.1 Точки восстановления	42
2.3.2 Журналирование событий	44
2.4 Алгоритм консенсуса Raft	46
2.4.1 Основные понятия	47
2.4.2 Основные фазы работы	49

2.4.3	Работа с журналом	49
2.4.4	Доступность	50
2.4.5	Применение.....	51
2.5	Модификация алгоритма Raft	51
3	Проектирование архитектуры отказоустойчивой и высокодоступной системы.....	55
3.1	Общие принципы.....	55
3.2	Виды архитектур.....	57
3.2.1	Монолитная архитектура.....	58
3.2.2	Плагинная архитектура.....	59
3.2.3	Сервисная архитектура	60
3.3	Архитектура отказоустойчивой системы.....	63
4	Отказоустойчивое веб приложение	67
4.1	Архитектура	67
4.2	Сервис учетных записей	69
4.3	Сервис каталога	71
4.4	Сервис заказов.....	73
4.5	API шлюз	75
4.6	Инфраструктура	78
	Заключение	80
	Список используемых источников.....	82
	Приложение А Листинг сервиса “Каталог”.....	85
	Приложение В Листинг сервиса “Учетные записи”.....	92
	Приложение С Листинг сервиса “API шлюз ”	99
	Приложение D Листинг сервиса “Заказы”	113
	Приложение E Листинг конфигурации Docker compose.....	122
	Приложение F Презентация	124

Перечень сокращений и обозначений

GPS (англ. Global Positioning System) — система глобального позиционирования.

RAID (англ. Redundant Array of Independent Disks) — избыточный массив независимых (самостоятельных) дисков.

CPU (англ. central processing unit) — универсальное обрабатывающее устройство, часто просто процессор.

POFOD (англ. Probability of Failure on Demand) — Вероятность отказа по требованию.

ROCOF (англ. Rate of Occurrence of Fault) — Частота возникновения сбоев.

MTBF (англ. Mean Time Between Failures) — Среднее время между сбоями.

MTTF (англ. Mean Time to Failure) — Среднее время до сбоя.

MTTR (англ. Mean Time to Repair) — Среднее время восстановления.

ПО — программное обеспечение.

CAP (англ. Consistency, Availability, Partition tolerance) — теорема CAP, эвристическое утверждение о том, что в любой реализации распределённых вычислений возможно обеспечить не более двух из трёх следующих свойств: согласованность данных доступность устойчивость к разделению.

CP (англ. Consistency, Partition tolerance) — согласованность данных, устойчивость к разделению.

AP (англ. Availability, Partition tolerance) — доступность устойчивость к разделению.

ACID (англ. Atomicity, Consistency, Isolation, Durability) — атомарность, согласованность, изолированность, сохраняемость.

BASE (англ. Basically Available, Soft state, Eventual consistency) — доступность в большинстве случаев, неустойчивое состояние, согласованность в конечном счете.

TDD (англ. test-driven development) — разработка через тестирование.

RSM (англ. replicated state machine) — машина с реплицированным состоянием.

Raft — алгоритм для решения задач консенсуса в сети ненадёжных вычислений.

C# — язык программирования.

Golang — язык программирования.

API (англ. Application programming interface) — программный интерфейс приложения, интерфейс прикладного программирования.

JSON (англ. JavaScript Object Notation) — текстовый формат обмена данными.

JWT (англ. JSON Web Token) — это открытый стандарт (RFC 7519) для создания токенов доступа, основанный на формате JSON.

HTTP (англ. HyperText Transfer Protocol) — протокол передачи гипертекста.

REST (англ. Representational State Transfer) — передача состояния представления.

Куки (англ. cookie) — небольшой фрагмент данных, отправленный веб-сервером и хранимый на компьютере пользователя.

Введение

Современные информационные системы, могут работать с сотнями тысяч пользователей. Широкое распространение приобрело использование облачных технологий для высоконагруженных и распределённых систем, предоставляющих доступ к своим вычислительным ресурсам. С их помощью производится автоматическое масштабирование, за счет размещения данных на серверах в огромных дата центрах. Что позволяет предотвратить отказ в обслуживании при пиковых нагрузках.

Обеспечение позитивного опыта взаимодействия с системой требует поддержания высокого уровня доступности и производительности. Время отклика современных веб-приложений должно находиться в пределах от нескольких сотен миллисекунд до секунды. В ином случаи многие пользователи перестанут пользоваться им. Для соответствия этим требованиям необходимо поддерживать бесперебойную работу на протяжении всей эксплуатации. Обеспечение отказоустойчивости позволяет поддерживать работоспособность системы в случаи сбоя.

Многие существующие алгоритмы позволяют повысить надежность при решении отдельных задач, с которыми сталкиваются информационные системы: хранение, обработка, передача информации. Одни из них эффективны на практике, другие могут применяться только в теории. Однако рост числа высоконагруженных геораспределенных систем требует постоянного увеличения эффективности, как существующих подходов, так и разработку новых.

В данной магистерской диссертации объектом исследования являются свойства отказоустойчивости и высокой доступности информационных систем. Предметом исследования выступают способы достижения отказоустойчивости

и методы повышения надежности. В частности, за счет применения распределенных систем.

Целью магистерской диссертации является улучшение известных алгоритмов обеспечения отказоустойчивости и высокой доступности, а также проектирование архитектуры и реализация приложения, обладающего свойствами отказоустойчивости и высокой доступности.

1 Основные положения

Любая система будь то программная или аппаратная изначально проектируется и конструируется с целью эффективного решения ряда практических задач. Информационная система – это удобная абстракция для описания большого класса систем, обладающих типовыми характеристиками. Им свойственно решать задачи хранения и обработки информации, построения поисковых индексов и кэшей, организация связи. В зависимости от характера решаемой проблемы в условиях ограниченных ресурсов, к ним предъявляются определенные требования. К числу подобных требований могут относиться: производительность, масштабируемость, время реакции, стоимость, поддерживаемость, безопасность и отказоустойчивость [1]. Последнее может неявно и/или непосредственно влиять на остальные требования или на свойства системы.

Первая глава посвящена рассмотрению теоретических основ отказоустойчивости и смежных с ней понятиям. А именно определению понятия отказа и сбоя системы, обзор формальных методов отказоустойчивости, метрики, используемые для оценки или сравнения систем, общие подходы к повышению доступности. Затем проводится анализ распределенных систем и их пригодности в качестве основы для построения отказоустойчивой системы. Уделяется внимание проблеме надежной передачи данных. В заключении, затрагивается вопрос восстановления работоспособности системы после сбоя. Рассматриваются несколько методов восстановления. Внимание уделяется компромиссу между системами, претерпевающими частые смены фаз сбоя и восстановления. И системами с высоким уровнем безотказности, работающие на протяжении долгого промежутка времени, не испытывая отказов или сбоев.

1.1 Основная информация о надежности

Отказоустойчивость – это свойство системы, выражающиеся в её способности поддерживать выполнение работы в случае сбоев или неполадок, обеспечивая необходимый уровень производительности. Важность внедрения системы отказоустойчивости заключается в обеспечении непрерывного обслуживания и поддержании функциональности в случае сбоя системы. Основопологающей целью для обеспечения отказоустойчивости является повышение общей надежности системы [2]. Отказоустойчивость позволяет предотвратить следующие проблемы:

- Утеря или повреждение данных
- Недоступность сервиса
- Компрометация системы
- Нарушение производительности

Надежность тесно связана с такими понятиями как: доступность, безотказность, обслуживаемость и безопасность. Они оказывают большое влияние на общую надежность системы.

Доступность. Система, обладающая данным свойством, должна обеспечивать корректную работу в каждый отдельный момент времени. Высоко-доступная система с высокой вероятностью корректно ответит на запрос в любой момент времени.

Система способная работать в течение долгого промежутка времени, не испытывая при этом сбоев обладает свойством *безотказности*. Обеспечение высокого уровня безотказности позволяет работать системе без сбоев в течении долгого промежутка времени. В случае доступности, речь ведется о моменте во времени, безотказность рассматривается в течение некоторого промежутка времени. Система может обладать высокой доступностью и в тоже время не отвечать требованиям безотказности. Например, веб сервис, не

отвечающий на запросы одну секунду каждый день в течение года является высоко доступным, но не безотказным.

Создать 100% надежную и отказоустойчивую информационную систему практически невозможно. Существует множество методов, позволяющие уменьшить влияние сбоев, повысить устойчивость к ним или вовсе предотвратить возможность их возникновения, об этом далее. Тем не менее, если в системе все-таки произошел сбой, важно предусмотреть пути восстановления работоспособности. *Обслуживаемость* характеризует систему с точки зрения удобства ее восстановления после сбоя. Для поддержания высоких показателей по данному параметру, необходимо организовать непрерывный контроль и мониторинг состояния системы. С их помощью возможно установить факт наличия сбоя, локализовать его местоположения и выстроить грамотную стратегию его устранения и восстановлению от последствий сбоя. Особенно важно, чтобы восстановление производилось автоматически, без участия человека.

Существует класс систем, для которых отказ или остановка на короткий промежуток времени может привести к катастрофическим последствиям. Они должны обладать высоким уровнем *безопасности*. К ним относятся системы, от которых зависят жизни людей или состояние окружающей среды. Их разработка сложна и принципиально отличается от систем общего назначения.

1.2 Отказы и сбои

Перед началом рассмотрения вопросов и проблем, связанных с отказоустойчивостью необходимо дать определения понятиям отказа и сбоя. А также определить их отличия. В профессиональной литературе и других источниках такие понятия как: сбой и отказ, не имеют согласованности, что часто приводит к путанице. Для ее устранения, в данном разделе даны

определения этим понятиям, в соответствии с которыми они использованы далее по тексту.

Сбоем – это некорректное внешнее поведение системы, либо отклонение ее характеристик от “нормальных”, заранее определенных показателей.

Отказ – ситуация, в которой система перестает производить полезную работу. Это может проявляться, как аварийное завершение программы, выход из строя аппаратного обеспечения, либо прекращение предоставления некоторого сервиса. Все это ведет к одному – нарушение контракта системы с внешним пользователем или другой системой.

1.2.1 Модели отказов

Отдельная программа или сетевая служба, считается рабочей до тех пор, пока действует согласно своей спецификации. В ней могут быть описаны возможные состояния системы, протокол взаимодействия с ней, различные квоты, а также показатели ее производительности. Нарушение спецификации по любому из пунктов может считаться неадекватным поведением и возникать из-за различных причин, специфичных для конкретной системы. Так, например, выход из строя одного диска, являющегося частью распределенного хранилища, не нанесет вреда данным, так как происходит их репликация. Но может повредить доступности, или привести к снижению показателей производительности хранилища в целом, нарушая тем самым спецификацию.

Существует несколько классификационных схем отказов (Рисунок 1). С их помощью можно абстрагироваться от специфичных проблем конкретной системы, и использовать при описании эквивалентных проблем и поиски путей их решения [3].

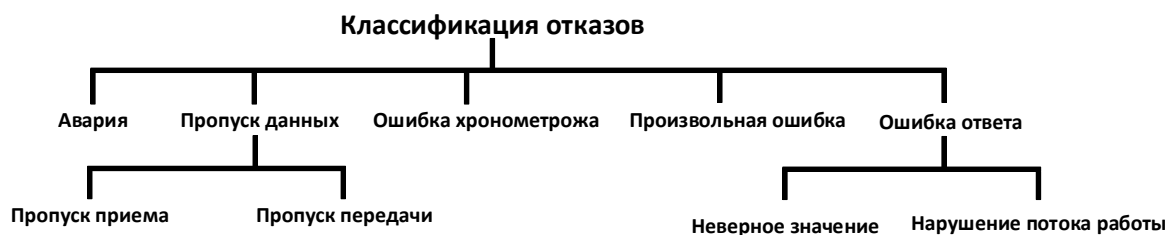


Рисунок 1 – Классификация отказов

Аварией считается, полная остановка работы системы, но при условии, что до этого ее работа протекала корректно. После остановки система не может произвести никаких действий, в том числе нанести вред себе и своим данным. Так аварийное завершение процесса базы данных, вследствие обращения по неверному адресу, может классифицироваться как авария. Напротив, если перед завершением работы, были повреждены данные, то такой сбой не является аварией, и несет более серьезные проблемы.

Передача данных – это процесс, при котором данные могут быть пропущены одной из сторон. Например, в клиент-серверном взаимодействии сервер может не получить данные из-за пропуска запроса или нахождения в состоянии, в котором не ожидается поступление данных от каких-либо клиентов, данная ситуация называется *пропуском приема*. Прямо противоположная ситуация – *пропуск передачи*, в этом случае сервер не может отправить ответ клиенту. Причиной может служить разрыв соединения, переполнение буфера отправки и т. д.

Нарушение хронометража происходит в случаи выхода за временные рамки при выполнении операции или обмене данными. Это касается не только операций, которые выполняются слишком долго, но также может относиться к процессам, которые протекают слишком быстро. В качестве примера, может служить сервис, предоставляющий видеопоток, выдача слишком большого объема данных в короткий промежуток времени, может вызвать проблемы на принимающей стороне в виду истощения буфера приема.

Ошибочный ответ является серьезным отказом, его наличие с высокой вероятностью свидетельствуют о проблемы с безопасностью системы. Существуют два подтипа. Первый – *неверное значение*, система выдает неверные данные, например, GPS навигатор выдает неверные координаты. Второй – *нарушение потока работы*, выражается нарушением протокола работы или взаимодействия. Примером может служить обработка некоторой последовательности запросов, ведущая к тому, что сервер предоставить доступ к данным минуя необходимую проверку прав пользователя (авторизацию).

Последний и самый опасный тип сбоев – *произвольные* или *византийские* сбои [4]. По сути, к ним относятся все остальные виды сбоев, которые не подпадают под выше обозначенные типы. Вся сложность заключается в числе возможных их проявлений и специфичности для конкретных систем, из-за чего невозможно выстроить эффективную стратегию их обнаружения и борьбы с ними.

1.2.2 Причины отказов и сбоев

Отказ или сбой может произойти из-за различных причин, важно понимать потенциальные источники их возникновения, с целью оценки ее вероятности. А также организации защитных мер, для их предотвращения [5]. Ниже приведена классификация сбоев и отказов, которая идет от наиболее предсказуемых и прогнозируемых причин к более случайным (рисунок 2).



Рисунок 2 – Схема классификация причин сбоев и отказов

Аппаратные сбои. Наиболее простой в обращении тип сбоев в информационных системах. Изначально может показаться, что данный тип сбоев ведет к сложным непоправимым последствиям для системы. Однако, они легко поддаются прогнозируемости так как, изготовителем изначально заявлено приблизительное время эксплуатации изделия. Отказы, вызванные из-за сбоев в аппаратуре часто (но не всегда) легко обнажаемы, и могут быть замаскированы путем обеспечения избыточности аппаратуры [6]. Например, использование RAID-массива накопителей, дублирование электропитания серверов и использование CPU с возможностью горячей замены.

Избыточность аппаратных ресурсов может замаскировать множество проблем, но не все. Современные отказоустойчивые системы - облачные платформы, способны перенести отказ целых машин благодаря методам обеспечения отказоустойчивости, а аппаратная избыточность применяется как дополнение к ним.

Программные ошибки. Зачастую программные ошибки имеют систематическую природу, их сложнее обнаружить, чем аппаратные сбои [7]. Их проявление носит случайный характер, неравномерно распределённый во времени. Таким образом ошибки могут проявляться в необычных условиях эксплуатации, поэтому сложны в отладке и воспроизведении. Примеры программных сбоев:

- Некорректные входные данные, приводящие к отказу системы.
- Истощение ресурсов, останавливает продвижение системы.
- Каскадные сбои, сбой одного компонента ведет к отказу всей системы

Решение проблемы возникновения сбоев в программном обеспечении носит комплексный характер, не существует одного единственного способа. Необходимо применять широкий спектр организационных и технических мер для их предотвращения, обнаружения и последующего исправления:

- Составление спецификации требований к программному обеспечению.
- Использование формальных методов описания системы.
- Использование готовых, надежных и протестированных компонентов
- Комплексное тестирование
- Оценка, мониторинг и анализ системы

Человеческий фактор. Недавние исследования показывают, что основной причиной отказов крупных интернет-сервисов являются ошибки, допускаемые человеком при конфигурировании систем. Для уменьшения степени влияния данного фактора необходимо повсеместно автоматизировать процессы: тестирования, разворачивания, восстановления. Только сведя к минимуму взаимодействие человека с конечной системой можно предотвратить эти непредсказуемые сбои [8].

1.3 Метрики отказоустойчивости

Метрика – численная мера, позволяющая оценить определенные свойства работы системы. Для каждой метрики обычно существуют ее эталонные показатели, указывающие, при каких крайних значениях стоит обратить внимание на правильность функционирования системы. Метрики разделяются на категории и могут оценивать совершенно различные аспекты системы.

Метрики отказоустойчивости могут выражаться в единицах измерения отказоустойчивости системы. Отказоустойчивость системы определяется подсчетом числа оперативных отказов и при необходимости соотносится с требованиями предъявляемые к системе и ко времени, в течение которого система должна быть в рабочем состоянии. Основные метрики:

Вероятность отказа по требованию (POFOD). Вероятность того, что система откажет во время обращения к ней.

Частота возникновения сбоев (POCOF). Частота возникновения некорректного поведения. Например, частота возникновения сбоев равная 0.05 показывает, что 5 сбоя возникают в течении сотни условных временных периодов.

Среднее время до сбоя (MTTF). Среднее время между сбоями системы. Например, при MTTF равным 300 значит, что в среднем каждые 300 условных временных единиц система испытывает сбой.

Среднее время восстановления (MTTR). Среднее время между сбоем системы и восстановление ее нормальной работы.

Среднее время между сбоями (MTBF). Сумма среднего времени до возникновения сбоя и среднего времени восстановления системы.

$$MTBF = MTTF + MTTR \quad (1)$$

Доступность системы. Характеризует готовность системы к работе, предоставлению своего сервиса немедленно, по требованию. Показывает вероятность того, что система безошибочно осуществит работу в любой момент времени. Может быть выражена как отношение среднего времени до возникновения сбоя к среднему времени между сбоями.

$$A = \frac{MTTF}{MTBF} = \frac{MTTF}{MTTF + MTTR} \quad (2)$$

1.4 Методы обеспечения отказоустойчивости

Отказоустойчивость, можно представить, как комплекс мер и техник, направленных на поддержание работоспособности системы. Путем к получению высокого уровня отказоустойчивости служит возможность своевременного обнаружение сбоя. Без него невозможно своевременно

прекратить распространение негативных последствий во всей системе. После установления факта наличия сбоя, осуществляется процесс оценки ущерба: компонент, работа которых была нарушена; поврежденные данные. Исходя из этой информации, предпринимаются конкретные действия по восстановлению. Только проведение всех вышеперечисленных этапов позволяет обеспечить систему свойством отказоустойчивости.

Основным подходом для обеспечения отказоустойчивости является маскирование сбоев. Избыточность ресурсов является ключевой концепцией для увеличения надежности. Примерами ресурсов, для которых следует обеспечивать избыточность: временные, программные, аппаратные, информационные.

Аппаратная избыточность выражается в использовании дополнительного оборудования. В случае выхода одного из блоков, возможно, осуществить его замену и продолжать выполнение работы. Данная техника особенно эффективна в средах, для которых характерна параллельность протекания процессов. Из-за простоты использования, находит широкое применение.

Использование только аппаратной избыточности не позволяет системе быть устойчивой к отказам, возникающим по причине наличия неисправности в программном обеспечении. Программная избыточность состоит в использовании дополнительных программ, модулей или функций для обеспечения отказоустойчивости [9]. Зачастую ошибки в программном обеспечении не могут быть выявлены за счет использования простого дублирования. Обычное копирование с высокой вероятностью приведет к повторному возникновению ошибки в том же месте. Для обхода данной проблемы одновременно используют различные реализации ПО. Для этого, параллельно разрабатывают несколько вариантов программы по одной спецификации. Таким образом, функциональность будет совпадать, но вероятность возникновения одних и тех же ошибок уменьшается.

Временная избыточность заключается в использовании дополнительных временных затратах для осуществления функций обеспечения отказоустойчивости. Она характерна как для аппаратной, так и для программной части. Обычно заключается в повторном выполнении тех же действий, в которых произошла ошибка. Например, при использовании резервных копий, избыточность – затрачиваемое на восстановление время. Для некоторых типов приложений данный вид избыточности может стать главным фактором выбора того или иного метода отказоустойчивости. Временная избыточность особенно полезна, если система склонна испытывать проходные или перемежающиеся отказы.

1.5 Распределенные системы

Распределенная система - группа отдельных, несвязанных компьютеров (узлов) работающих таким образом, чтобы внешний наблюдатель (пользователь или другая система) имел о ней представление как о единой сущности. [3]. Например, почти все большие веб сервисы, такие как Amazon, Ebay, Facebook, каждый с точки зрения пользователя является единым продуктом. Но в действительности представляет собой распределенную систему, одновременно размещенную на десятках и даже сотнях тысяч компьютеров.

Многие из проблем, которые решены для систем, работающих на отдельном компьютере, сложны для решения в распределенных системах. Для всех систем такого класса характерны следующие свойства:

- Конкурентная работа. Процессы, протекающие на отдельных узлах (компьютерах, серверах), работают одновременно.
- Независимость сбоев. С каждым узлом может произойти сбой, и каждый из них происходят независимо друг от друга. Так, например, отказ одного компьютера из-за выхода из строя жесткого диска не влияет на работоспособность других узлов.

- Узлы не имеют глобальных часов для синхронизации.

Распределенные системы часто применяются для решения таких задач как: хранение данных, вычисления, обмен сообщениями. Выше описанные свойства вносят множество проблем при решении этих задач [10]. С другой стороны, они позволяют решать проблемы масштабируемости, отказоустойчивости и доступности, которые невозможно преодолеть с использованием стандартного подхода – одна программа, запущенная на отдельном сервере.

1.5.1 Проблемы распределенных систем

Распределенные системы как уже было сказано выше добавляют комплексности. Для более детального разбора возникающих трудностей рассмотрим проблемы, возникающие при решении типичной задачи хранения данных.

Большинству систем для хранения информации достаточно одного мощного сервера, на котором развернута реляционная база данных. Для нее характерна большая нагрузка по чтению, чем по записи. Существует множество систем, для которых такое поведение естественно, например, веб.

Рост количества пользователей приводит к нагрузке, с которой в определенный момент база данных, состоящая из одного мастер-сервера, перестает справляться. В результате возникает потребность в ее масштабировании. Так, если проблема заключается в нагрузке по чтению, решением может служить *репликация* (Рисунок 3).

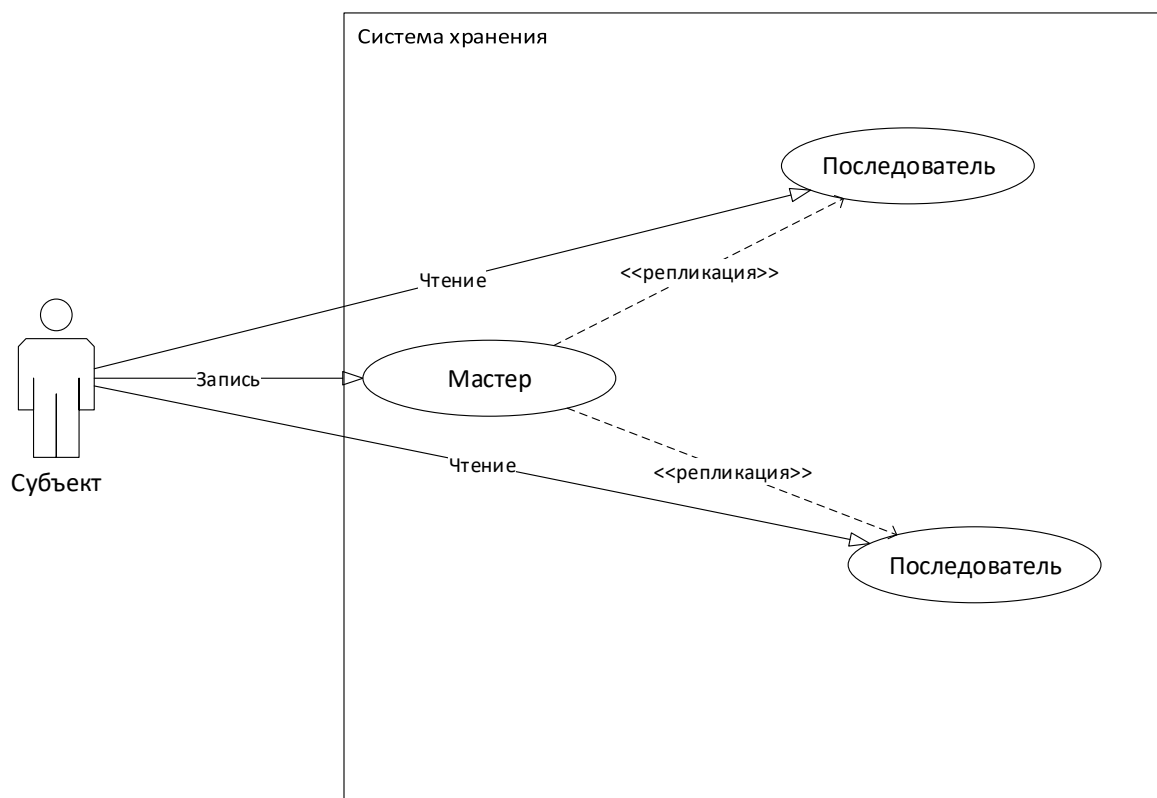


Рисунок 3 – Схема работы репликации

Репликация заключается в следующем. Все операции записи производятся мастер сервером и затем изменения распространяются на сервера последователи. В тоже время операции чтения могут быть осуществлены как к мастеру, так и к любому из последователей. Таким образом, получается распределенная система хранения данных. Но наряду с возможностью осуществления большего количества операций чтения, данная схема уменьшает согласованность данных. Вследствие конкурентного протекания операций на отдельных узлах, данные не согласованны в каждый момент времени. Может произойти так, что операция записи изменила данные на мастер узле, но еще не была воспроизведена на последователе, к которому в данный момент обращается пользователь.

Репликация позволяет добавлять новые сервера-последователи тем самым справляясь с более высоким уровнем нагрузками по чтению [11]. Но при росте нагрузки по записи, единственный мастер-сервер может переставать выдавать

приемлемые показатели производительности. Для решения этой проблемы необходимо использовать секционирование (шардинг) (Рисунок 4).



Рисунок 4 – Схема секционирования

Секционирование подразумевает разбиение некоторым образом всего набора данных на диапазоны. Каждому диапазону назначается сервер, который отвечает за него, то есть осуществляет операции записи. Репликация часто применяется совместно с секционированием. Таким образом, система хранения становится более распределенной, она может справляться с нагрузками по чтению и записи, с которыми не может справиться ни одно хранилище, расположенное на одном сервере. Ценой за секционирование служит наращение

модели данных. Так как данные хранятся на разрозненных серверах, нельзя эффективно осуществить слияние данных, поддерживаться целостность данных.

1.5.2 CAP теорема

Теорема CAP гласит, что распределенная система может иметь одновременно только два из трех свойств.

C - Consistency. Согласованность, все копии разделяемых, резервируемых копий одинаковы.

A - Availability. Доступность данных в каждом узле.

P - Устойчивость к нарушению связности сети.

Другими словами, в контексте, где сетевые ошибки являются неотъемлемой частью коммуникации, невозможно реализовать атомарное чтение/запись разделяемых данных, гарантированно для каждого запроса.

Проиллюстрировать работу данной теоремы можно в примере с двумя процессам, которые не могут коммуницировать вследствие неисправной сети. Если разрешить изменять данные одному из них, то такая система обладает согласованностью и устойчивостью к разделению (CP). Если же процессы могут взаимодействовать, тогда возможно обеспечить согласованность и доступность (AP). В сущности, вся CAP теорема говорит о компромиссах между безопасностью и живучестью.

На практике распределенные системы не надежны и никогда нельзя гарантировать их устойчивость к разделению. Из этого следует, что необходимо принять во внимание изначальное присутствие разделения. Тогда для конкретной системы следует решить с каким из остальных двух свойств (согласованностью, доступностью) она может нормально функционировать. Данный компромисс лежит в основе двух различных семантик, которые применяются при работе с хранилищами данных: ACID (Atomicity, Consistency,

Isolation, Durability – атомарность, согласованность, изолированность, сохраняемость) и BASE (Basically Available, Soft state, Eventual consistency – доступность в большинстве случаев, неустойчивое состояние, согласованность в конечном счете) [12]. Первая семантика соотносится с СР свойствами, вторая с АР. Это позволяет BASE-хранилищам на некоторых задачах, обрабатывать такие объемы данных, с которыми невозможно иметь дело при использовании ACID. Но есть проблемы, которые невозможно решить, используя BASE, или решения получаются ненадежными. Например, финансовые системы могут пожертвовать частью доступности, но не могут нарушать согласованности данных.

1.5.3 Распределенный консенсус

Распределенные системы могут испытывать множество различных проблем: потеря пакетов при передаче, неправильная последовательность входных данных или их дублирование, выход из строя отдельных узлов, неопределенные временные задержки. Есть системы, нестрогие требования которых позволяют вывести ошибку и остановить свою работу. С другой стороны, отказоустойчивые системы должны справляться со сбоями.

Подобно различным паттернам сложившиеся в разработке программного обеспечения, которые решают часто возникающие, типовые задачи. А также создают язык для размышления о системах в целом. В контексте распределенных систем также существуют полезные абстракции, предоставляющие ясные гарантии для построения отказоустойчивых систем. Ярким примером является консенсус – согласованность между всеми узлами по какому-либо вопросу. Он одновременно сложен в реализации, но также предоставляет понятные гарантии, на основе которых, становится возможным строить надежные, отказоустойчивые распределенные системы [13].

Алгоритмы консенсуса можно разделить на два вида:

- Crash-fail. Не позволяют вышедшему из строя узлу вернуться в работу.
- Crash-recovery. Узел может восстановиться и вернуться к работе.

Однако согласно теореме Фишера, Линча и Паттерсона о невозможности консенсуса в системе со сбоями, согласно которой в условиях ненадежной сети невозможно заставить работать ни один алгоритм консенсуса [14]. В действительности, решение задачи достижения консенсуса рассматривается в ограниченном промежутке времени, что позволяет преодолеть теоретическое ограничение и прогрессировать алгоритму большую часть времени. Существует множество алгоритмов достижения консенсуса: Paxos, Raft, Zab, Mencius. Raft будет рассмотрен во второй главе.

1.6 Восстановление

Система, неправильно работающая в отдельно взятый момент времени все еще может считаться отказоустойчивой. Для этого должен быть предусмотрен механизм, позволяющий нивелировать последствия сбоя и восстановить работоспособность системы. Существует два основных подхода к решению вопроса восстановления:

- Обратное восстановление
- Прямое восстановление

Первый способ опирается на тот факт, что до того, как система пришла в нерабочее состояние она работала исправно. Следовательно, откат к последнему исправному состоянию позволит вернуть работоспособность. Чтобы обеспечить возможность такого восстановления, через определенные промежутки времени производится создания слепка состояния системы.

Преимуществом такого подхода является его универсальность, нет необходимости учитывать логику работы системы. Для данного метода

достаточно факта обнаружения неисправности, после чего произвести откат к последнему слепку. Таким образом, данный способ может справляться с широким спектром ошибок.

Однако существует высокая вероятность повторного возникновения отказа. После восстановления в работоспособное состояние на короткий промежуток времени, система вновь испытывает сбой [15]. В итоге все сводится к замкнутому циклу восстановление-сбой. Следует также принимать во внимание, что сохранение состояния системы, а также восстановление являются, в большинстве случаев, дорогостоящими операциями, как в плане времени, так и производительности.

Второй способ прямо противоположен первому. Прямое восстановление, опираясь на семантику работы системы, пытается перейти из ошибочного в новое, корректное состояние системы. У него нет той универсальности, которой обладает первый способ. Определение факта наличия сбоя, также недостаточно. Необходимо классифицировать его тип, после чего предпринять конкретные действия к восстановлению.

Преимуществом этого способа является, как правило, меньшие накладные расходы по сравнению с обратным восстановлением, в котором приходится сохранять полное состояние системы. А также низкая вероятность повторного сбоя, так как система не откатывается в состояние, из которого произошел отказ.

Яркой иллюстрацией различий двух подходов может служить реализация надежной передачи данных. Так, повторная передача пакетов, в сущности, является способом привести систему в последнее корректное состояние с точки зрения отправителя (первый способ). Отражением прямого восстановления, является использование кодов коррекции. В нем, для восстановления потерянных пакетов используется информации из успешно пришедших, иными словами система постоянно продолжает прогрессировать.

Иногда, исходя из способа работы системы и требований, которые к ней предъявляются, целесообразнее после сбоя повторить весь процесс званого. Такой подход носит название: вычисления, ориентированные на восстановление. Преимуществом данного подхода является простота, например, если сервер перестал отвечать на запросы, следует просто завершить его процесс и запустить званого. С другой стороны, если система состоит из множества частей, ключевым становится момент локализации места сбоя.

Эффективность применения полного восстановления во многом зависит от системы. Многокомпонентная, слабосвязанная система позволяет просто локализовать ошибку, и осуществить покомпонентное восстановление, не затрагивая всю систему. В случае сильно связанных систем эффективнее применять стандартные техники восстановления. Данный подход, можно рассматривать как метод восстановления из снимка состояния, где за корректное состояние принимается начальная конфигурации системы.

2 Обеспечение отказоустойчивости

Вторая глава нацелена на предметное рассмотрение вопроса обеспечения отказоустойчивости. В ней дается обзор распространенных на практике методов, алгоритмов и способов повышения надежности и борьбы с различными сбоями. Оцениваются пути их эффективного применения, с учетом преимуществ и недостатков. Взятые алгоритмы и методики являются отражением теоретических положений, изложенных в первой главе. На основе материалов данной главы сформируется основа для реализации отказоустойчивой и высоко доступной системы.

2.1 Обнаружение сбоев

Эффективно противостоять тому или иному сбою можно только после того как, как он становится обнаружен. Исходя из определения сбоя, данное в разделе 1.2, следует, что любое некорректное поведение или состояние системы может быть признано как сбой. Таким образом, чтобы выявить его наличие необходимо описать, что для конкретной системы является неприемлемым. *Функция пригодности*, понятие, взятое из теории эволюционных алгоритмов, позволяет элегантно решить данный вопрос [16].

Системы являются сложными, комплексными сущностями. К каждой из них предъявляются свои требования, отличающиеся по разным параметрам. Некоторые системы нуждаются в эффективном масштабировании, другим важна безотказность работы и контроль объема потребляемых ресурсов.

Функции пригодности определяют рамки важных для нее характеристик. С их помощью можно оценивать целостность системы, где выход параметра за пределы допустимых значений принимается за сбой. В качестве примера можно рассмотреть требование, согласно которому сервер должен отвечать клиенту в

течение 500 миллисекунд. В данном случае функция пригодности оценивает время ответа всем (или части) клиентам. Нарушения временных рамок свидетельствует о сбое. В ответ на него система может сделать вывод о недостатке ресурсов и поднять дополнительный экземпляр сервера, тем самым маскируя сбой.

Функции пригодности обладают различными свойствами, исходя из направления их применения. Для их оценки можно выделить следующие параметры:

Комплексность. Существуют функции, которые затрагивают несколько компонентов системы и оценивают одновременно несколько параметров, они подходят для обнаружения широкого набора сбоев. Для их устранения используются алгоритмы общей направленности, такие как восстановление из контрольной точки. Свое применение находят и узко направленные функции пригодности, направленные на выявление специфичных проблем, для устранения которых могут применяться техники прямого восстановления.

Событийность. Большое число событий одновременно происходит в работающих системах. Наступление некоторых из них дает повод для применения функций пригодности. Функция пригодности, выполнение которой зависит от события, является событийной. Ее противоположность – это непрерывная функция пригодности. Выполнение последней не зависит от возникновения каких-либо событий. Она постоянно оценивает различные характеристики системы.

Динамичность. Функция пригодности может иметь жестко заданное значение, такая функция называется статической. Но иногда требуется учитывать контекст, в котором находится система в некоторый момент времени и согласно ему, определять возможные значения характеристик. Такая функция называется динамической.

Автоматизированность. Чтобы эффективно и своевременно обнаруживать сбои в системе необходимо, чтобы большинство функций системы должны быть автоматизированы. Но существуют такие сбои, которые невозможно обнаружить автоматически. Так многие проблемы с безопасностью, часто являются скрытым ошибками, которые не поддаются автоматизированному нахождению. Для борьбы с ними привлекают специалистов для проведения аудита системы безопасности.

2.1.1 Мониторинг

Одной из часто используемых техник для наблюдения за состоянием системы и ее функциями пригодности является мониторинг. Это обширное понятие, включающее в себя множество направлений оценки работы конечной системы и ее окружения. Мониторинг занимается сбором, обработкой и агрегированием информации о количественных показателях системы. К ним могут относиться: число запросов, количество и типы ошибок, время работы системы. Для отказоустойчивости важным механизмом мониторинга является оповещение. Оповещение – сигнализирует о возникновении исключительной ситуации [17].

Мониторинг и оповещения дают возможность не только реагировать на факт наличия сбоя в системе, но также на основе оценки текущих и ретроспективных данных предвидеть ожидаемые неисправности. И затем предпринять меры по их предотвращению. Например, наблюдение за состоянием диска и оповещение о его быстром заполнении позволяет заблаговременно увеличить пространство и предотвратить возможную остановку работы.

Причина и следствие

Эффективность мониторинга зависит не только от возможности определения симптомов сбоя, но и от причин повлекшие его. Установка причинно-следственных связей типа: что и почему – это то, что позволяет

создать полезный мониторинг, с помощью которого возможно выявить критически важные точки функционирования системы на фоне всех событий, происходящих в ней.

Белый и черный ящик

Мониторинг методом черного ящика – это наблюдение за симптомами, которые система испытывает в данный момент. При его использовании система рассматривается со стороны, что позволяет опустить детали её внутреннего устройства и упростить сам процесс. Метод белого ящика учитывает внутреннее устройство системы, благодаря этому становится возможным обнаруживать не только фактические, но и потенциальные проблемы. Иногда различия между данными методами заключается лишь в точки зрения. Так в многоуровневых системах, проблема в одном из внутренних компонентов может служить причиной появления симптомов поломки одного из внешних. Важно чтобы мониторинг предоставлял информацию о первоочередной причине, для ее быстрого устранения.

Размерность измерений

Характер работы различных компонентов определяет приемлемый уровень детализацию их измерений. Неверно выбранный масштаб измерений, может вводить в заблуждение. Рассмотрим на примере наблюдения за состоянием устройств:

Загрузка центрального процессора является крайне изменчивым показателем. Его измерение с минутной периодичностью не позволяет выявить пики нагрузки. Секундный интервал позволяет получить репрезентативную оценку работы процессора, но сбор и анализ могут повредить чистоте измерений. Напротив, измерение степени загрузки дисков каждые несколько минут влечет чрезмерными накладными расходами.

Компромиссом является разделение процесса сбора и анализа данных, накопление происходит с масштабом необходимым для получения полезных

метрик. Анализ производится либо внешней системой, либо с периодом, который не повредит общей производительности. Следует иметь в виду, что такой подход снижает оперативность возможных оповещений.

Рассмотрев основные положения мониторинга можно сформировать правила для построения полезной системы мониторинга и оповещения:

- Выбирать разумную детализацию измерения параметров
- Проводить внешний и внутренний мониторинг
- Разделять аварийные и проходные оповещения
- Оповещение должно содержать информацию, как о симптомах, так и возможных причинах возникновения проблемы

2.1.2 Тестирование

Тестирование – это комплекс действий, который позволяют найти области эквивалентности системы в процессе ее изменения. Успешное выполнение тестов после внесения в нее изменений, увеличивает вероятность того, что поведение не изменилось. Следовательно, не была допущена ошибка, которая могла привести к сбою, при работе системы. Тесты являются одной из реализаций функций пригодности. Полное тестирование позволяет повысить общую надежность системы [18].

Уровень надежности, определяемый требованиями к системе, непосредственно влияет на объем необходимого тестирования. Однако тесты дают эвристическую оценку надежности системы, неисправные компоненты могут быть не учтены в тестовых сценариях. Для более точной оценки, тестирование необходимо дополнять данными мониторинга. Благодаря им можно не только удостовериться в корректной работе, но и определить области, для которых необходим больший объем тестирования.

Тестирование непосредственно влияет на среднее время восстановления (MTTR). Ошибки, обнаруженные с помощью мониторинга, в процессе эксплуатации системы являются дорогостоящим в плане времени

необходимого для восстановления. Устранение ошибки на этапе тестирования ликвидирует время, которое необходимо было бы затратить на ее восстановление. Так как показатель среднего времени между сбоями (MTBF) зависит от MTTR, значение первого увеличивается по мере увеличения покрытия системы тестами. Рост MTBF позволяет увеличить частоту внесения изменений.

Виды тестов ПО

По аналогии с тем, как системы могут рассматриваться с разным масштабом, так и тесты делаться на несколько видов в зависимости от размера компонента, к которому они применяются. Чем более крупную часть системы проверяет тест, тем более хрупким он становится. Его поддержание требует больших ресурсов (Рисунок 5).



Рисунок 5 – Зависимость количества тестов и устойчивости их к изменениям от их вида

Модульное тестирование. Наиболее обширная по количеству разновидность тестов. С их помощью проверяются мельчайший тестируемый компонент в системе. Модульные тесты содержат наименьшее количество зависимостей и связей с другими компонентами, поэтому они устойчивы к

внесению изменений. Данный вид тестирования является основой подхода разработки через тестирование (test driven development, TDD).

Интеграционное тестирование. Данный вид тестирования проверяет правильность взаимодействия между отдельными модулями. Отдельные компоненты, прошедшие модульное тестирование объединяются в группы. Работоспособность отдельных компонентов не гарантирует таковую у группы, в которую они входят. Целью интеграционного тестирования является проверка корректности поведения этих групп.

Системное тестирование. Наиболее масштабный вид тестов. Они проверяют работу всей системы. Она оценивается с разных точек зрения, в соответствии с ними выделяют подвиды системных тестов:

- Проверочные тесты. Проверяют базовую функциональность системы, ради которой она существует.
- Тесты производительности. Проверяют приемлемость основных показателей работы системы.
- Регрессионные тесты. Содержат описание всех ошибок, которые когда-либо происходили в системе и проверяют возможность их повторного возникновения.

Каждый вид тестов несет свои накладные расходы. Модульные тесты выполняются быстро и не требуют больших ресурсов. С другой стороны, системные тесты требуют наличия развернутой системы. Достижение компромисса относительно набора тестов различных видов позволяет эффективно использовать тестирование в качестве инструмента повышения надежности.

2.2 Маскирование сбоев

Маскирование – основная техника, используемая как основа в различных алгоритмах обеспечения отказоустойчивости. Ее идея заключается в сокрытии произошедшего сбоя и его последствий. Сбои, которые не могут нанести серьезного ущерба, маскируются путем их игнорирования. Для борьбы с критическими сбоями используют избыточность [3].

2.2.1 N-версионное программирование

Фундаментом данного метода является функциональная спецификация. На ее основе создаются несколько версий программы. В процессе реализации могут быть использованы различные технологии алгоритмы и языки программирования. Работа нескольких версий программы осуществляется в специальном окружении – N-версионной среде исполнения (СИ). В ее обязанности входит управление выполнением ПО, а также выбор одного результата, из результатов всех программ. С точки зрения среды каждая версия программы является черным ящиком. Входные данные поступают на вход программе, обрабатываются, на выход выводится результат работы [19].

Данный метод маскирует программные ошибки за счет избыточности версий программы. Все версии обеспечивают одну и ту же функциональность, так как были разработаны на основе одной спецификации. Сбой одной версии, может быть замаскирован результатами работы другой, так вероятность нахождения ошибки в одном и том же месте сразу в нескольких версиях мала. Учитывая тот факт, что версии ПО могут выполняться с разной степенью эффективности, n-версионная среда исполнения оценивает различные параметры работы программ (количество используемой памяти, время выполнение, точность результатов). Наиболее эффективные варианты могут обладать большим приоритетом при выборе в качестве источника результата.

На схеме представлен пример типовой организации n-версионного программирования (Рисунок 6). Она отражает следующую последовательность действий:

1. Поступление входных данных на несколько или все версии программы.
2. Программы выполняют необходимую работу.
3. Проверка результатов с помощью тестов (функций пригодности).
4. В случае успеха 3-го шага вернуть результат.
5. В случае провала 3-го шага, выбрать альтернативный блок для выполнения операции.
6. При отсутствии альтернатив вернуть ошибку, иначе перейти на шаг 2.

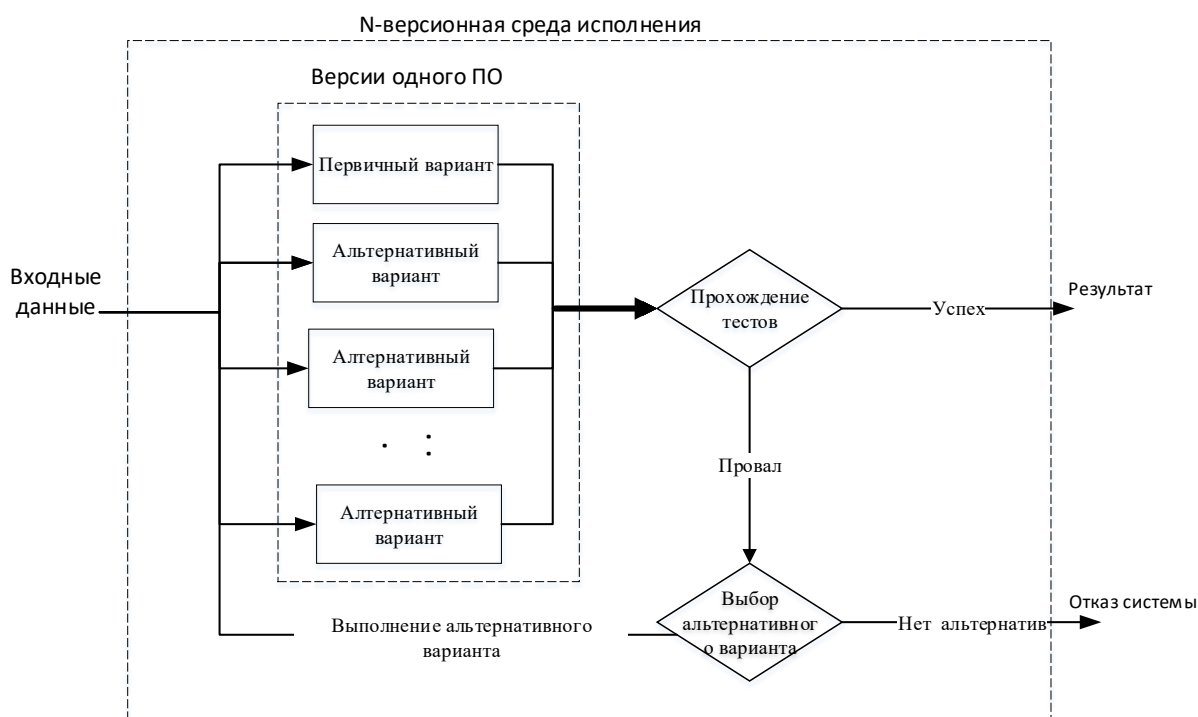


Рисунок 6 – Схема n-версионного программирования

2.2.2 Группы процессов

Альтернативный способ маскирования программных сбоев – обеспечения избыточности процессов. В отличие от n-версионного программирования, здесь необходимо написать только одну программу. Маскирование происходит путем запуска нескольких процессов. Если один из процессов претерпевает отказ, другие процессы обеспечивают прогресс системы. При n-версионном

программировании программа может ничего не знать о маскировании сбоя, вся работа ложится на среду исполнения. Однако, чтобы заставить эффективно и корректно работать данный подход необходимо особым образом организовать взаимодействие этих процессов.

Разновидности групп

Группы могут быть организованы различным образом, внутренняя организация влияет на способ их работы. Они могут быть *статическими* и *динамическими*. Состав статической группы никогда не изменяется. Динамические группы напротив могут менять состав процессов во время работы. С одной стороны, это позволяет выводить вышедшие из строя процессы из состава группы и добавлять новые. Но добиться правильной работы динамической группы является сложной задачей.

Другим критерием выступает права процессов, входящих в одну группу. В группах, где все процессы равноправны, решения принимаются сообща, кворумом. Кворум – это подмножество процессов, принимающих решение в группе. Преимуществом такой организации является отсутствие единой точки отказа, выход из строя одного процесса ведет лишь к уменьшению группы. Однако в данной схеме принятие решений требует слаженной координации всех процессов в группе (Рисунок 7), что требует дополнительного времени.

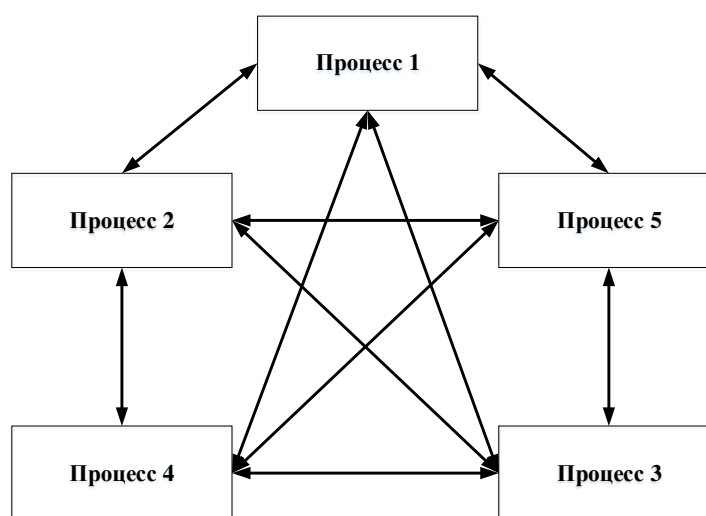


Рисунок 7 – Организация взаимодействия одноранговой группы

Противоположностью являются группы с иерархической организацией. В них задачи различных процессов отличаются, например, один процесс координирует работу остальных. В отличие от одноранговых групп, выход из строя определенных членов иерархической группы может привести к снижению производительности или вовсе вывести ее из строя. В тоже время принятие решений происходит быстрее за счет более упрощённой организации взаимодействия (Рисунок 8).

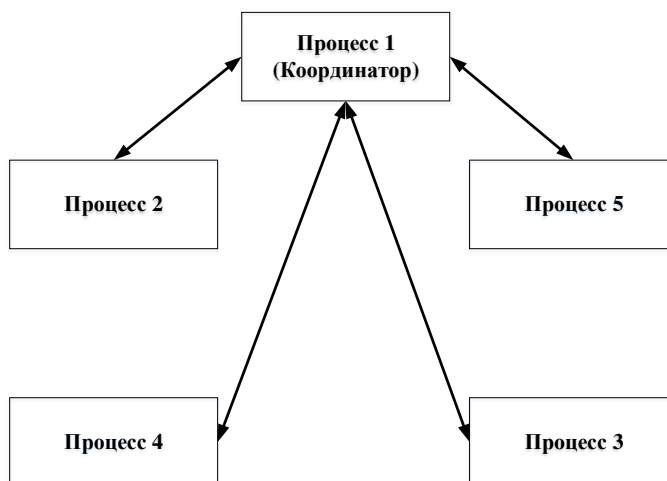


Рисунок 8 – Организация взаимодействия иерархической группы

2.3 Алгоритмы восстановления

Алгоритмы восстановления определяют конкретную последовательность действий предпринимаемые системой для перехода из ошибочного в корректное состояние. Рассматриваемые подходы к реализации восстановления относятся к методам обратного восстановления, общие сведения о которых рассмотрены в разделе 1.6.

2.3.1 Точки восстановления

Точки восстановления позволяют вернуть состояние системы к некоторому моменту в прошлом. Процесс восстановления использует их для замены ошибочного состояния на исправное. Информация, хранящаяся в точке восстановления специфична для каждой системы. К хранимой информации

может относиться: состояние переменных и регистров процесса, конфигурация и т.д. Точки восстановления должны сохраняться в надежное хранилище для предотвращения потери в случае выхода узла из строя. Процесс их создания и восстановления из них прост для систем, состоящих из одного процесса и работающих на одном узле. Однако их использование в распределенных системах является нетривиальной задачей.

Точки восстановления в распределенных системах

Глобальное согласованное состояние распределённой системы, которое необходимо сохранить носит название *распределенного снимка состояния*. Распределённый снимок состояния представляет собой коллекцию локальных точек восстановления, сделанных на каждом узле. Особым подмножеством является *линия восстановления*, в нее входят согласованные локальные точки, сделанные последними на узлах.

Задача создания распределенного снимка состояния является сложной задачей. Существует два подхода к ее решению:

Согласованное. Все узлы координируют момент начала создания локальных точек восстановления. Таким образом, созданные точки автоматически являются глобально согласованными. Например, можно использовать алгоритм двухфазной блокировки: координатор оповещает все узлы о начале создании снимка состояния, затем, когда все процессы оповестят координатора об окончании создания снимка, он рассылает сообщение об окончании, тем самым снимая блокировку. Недостатком данного подхода являются проблемы, унаследованные от двухфазной блокировки.

Улучшение данной схемы заключается в рассылке события создания точек только тем узлам, которые зависят от восстановления координатора. К зависимым относятся те узлы, которые непосредственно или косвенно связаны с сообщениями, от которых зависит координатор. Таким образом, система

создает *инкрементальные точки восстановления*, так как затрагивается только измененная часть системы.

Независимое. Узлы создают локальные точки от времени, не координируя данный процесс между собой. Это позволяет избавиться от негативных свойств захвата глобальной блокировки. Однако нельзя гарантировать, что последние точки восстановления образуют распределенный снимок состояния. Поиск линии восстановления требует сохранение в них дополнительной информации о зависимостях между процессами. Вычисление и хранение таких зависимостей являются комплексной задачей.

На практике накладные расходы на координацию оказываются предпочтительнее запутанных и сложных алгоритмов вычисления зависимостей. Поэтому даже в крупномасштабных системах большее применение находят методы согласованного создания точек восстановления.

2.3.2 Журналирование событий

Журналирование событий позволяет уменьшить число дорогих по производительности операций создания точек восстановления. Это осуществляется за счет повторного исполнения событий, сохранённых в нем. Для восстановления состояние системы необходимо выполнить заново те события, которые произошли с момента начала ее работы или с момента создания последней точки восстановления.

Откат к точке восстановления часто ведет к потере данных, которые были обработаны с момента создания последней точки восстановления до возникновения сбоя. Журналирование событий позволяет избавиться от этого недостатка. После восстановления из точки, система воспроизводит все события, произошедшие после нее, тем самым возвращаясь к актуальному состоянию.

Классификация методов журналирования событий

Согласованное состояние – это то, к чему должна прийти система после восстановления. Для того чтобы привесит классификацию необходимо ввести понятие *осиротевшего процесса* в контексте журналирования. Осиротевшим является процесс, который зависит от некоторого события, которым он не располагает и все процессы, владеющие им, дали сбой и не успели записать его в надежное хранилище. Таким образом, осиротевший процесс не сможет получить данное событие. Необходимо исключить возможность возникновения осиротевших процессов, чтобы гарантировать восстановление согласованного состояния при воспроизведении событий. Гарантия того, что процесс может становиться зависим от некоторого события, только если он обладает им, позволяет избежать появления осиротевших процессов. В основном существует два подхода к журналированию событий:

1. Пессимистичное журналирование
2. Оптимистичное журналирование

Пессимистичное журналирование требует обязательное сохранение события в надежное хранилище перед тем, как отправлять его другому процессу. Таким образом, даже после сбоя процесс сможет восстановиться и снова передать событие, избежав тем самым появления осиротевших процессов.

Оптимистичное журналирование не ставит целью предотвращение появления осиротевших процессов, а устанавливает факт их наличия и заставляет их откатиться из этого состояния. Их выявление требует наличия механизма отслеживания зависимостей.

Первый способ более прост в реализации и подходит для множества систем, но требование сохранения каждого события в надёжное хранилище при каждой передаче сказывается на пропускной способности событий. С другой стороны, оптимистичное журналирование более сложно в реализации из-за необходимости отслеживания зависимостей.

Машины с реплицированным состоянием

Журналирование событий тесно связано с понятием *машины с реплицированным состоянием* (*replicated state machine, RSM*). Основная его идея заключается в том, что выполнение одинаковой последовательности событий в нескольких процессах приводит их в конечном итоге к одному состоянию. RSM – это полезная абстракция, используемая для построения компонентов распределенных систем, например является частью маскирования сбоя группой процессов (раздел 2.2.2).

Машина с реплицированным состоянием является логическим уровнем поверх компонентов ответственных за прием и определение последовательности сообщений (Рисунок 9). Выполняемые операции отбираются алгоритмами достижения консенсуса.

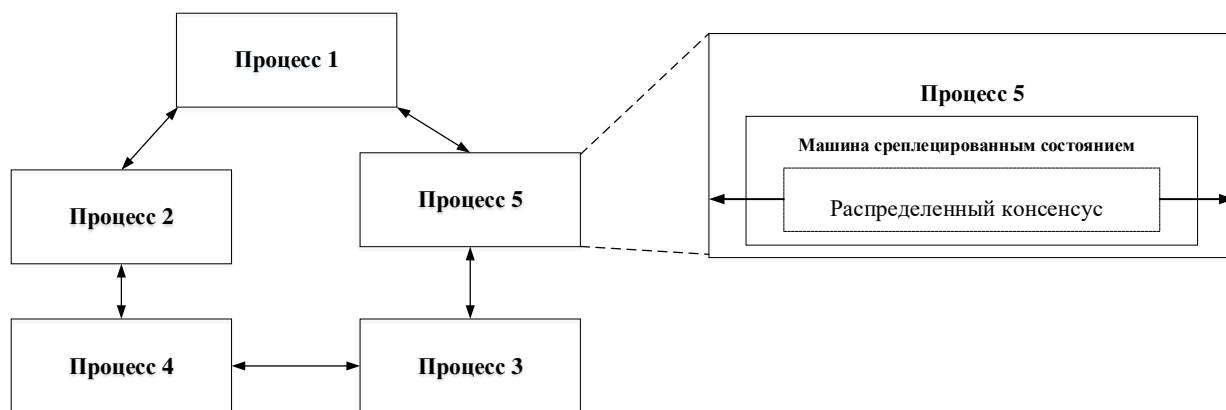


Рисунок 9 – Схема применения машины с реплицированным состоянием

2.4 Алгоритм консенсуса Raft

Raft – это алгоритм достижения распределенного консенсуса основанный на выборе лидера [13]. В сущности, Raft управляет работой группой процессов с целью ведения реплицированного журнала сообщений с линеаризуемой семантикой. Состав группы может динамически изменять. Все операции по изменению журнала проходят через один узел – лидер, выбираемый в процессе работы. Другими словами, в нем используется динамическая, иерархическая

группа процессов (раздел 2.2.2), в каждом из которых работает машина с реплицированным состоянием (раздел 2.3.2).

Практически все алгоритмы консенсуса представляют собой многоступенчатый набор действий, сложный для понимания и реализации. Однако, данный алгоритм ставит одной из своих целей – быть максимально простым для изучения. Что позволяет избавиться от неопределенности и неверных допущений при реализации алгоритма.

Raft может справляться со всеми видами отказов, за исключением произвольных, пока сохраняются нормальные условия работы. Алгоритм допускает выхода из строя N процессов (узлов) при наличии в группе $2N+1$ процессов.

2.4.1 Основные понятия

Процессы, работающие под управлением Raft, не являются равноправными. Существуют несколько ролей, в соответствии с которыми действует тот или иной процесс. Каждый из них может выполнять любую роль, но только одну в один момент времени.

- Лидер
- Кандидат
- Последователь

Лидер – это процесс, координирующий работу с реплицированным журналом. Его выбирают путем голосования все процессы. Побеждает тот узел, который заполучил большинство голосов. Каждая команда изначально поступает лидеру, затем лидер реплицирует ее на другие процессы, после чего осуществляет запись в журнал. Лидер, через определенные промежутки времени, сигнализирует остальным процессом о том, что он находится в работоспособном состоянии.

Кандидатом выступает тот процесс, который пришел к выводу, что лидер вышел из строя, и выдвигает свою кандидатуру в качестве нового. После того, как выборы нового лидера прошли, если кандидат выиграл их, он становится новым лидером, иначе переходит в состояние последователь. При нормальных условиях процессы редко находятся в данном состоянии.

Последователь является пассивным участником, его задача заключается в поддержании работы локального журнала, то есть включение в него событий, приходящих от лидера. Каждый последователь ожидает уведомления от лидера, о работоспособности последнего, в течение определенного промежутка времени. После его истечения, последователь переходит в состояние кандидата.

Эпохи (термы)

Организация работы алгоритма не требует наличия глобальной синхронизации. Для того чтобы определять отставание процессов и сравнивать журналы на предмет новизны, вводится понятие эпохи. Эпоха – это логическая единица, определяющая период работы одного лидера. Она представляет собой монотонное значение. Каждая эпоха начинается с выборов, инициируемых кандидатом.

Каждый процесс хранит локальное представление о текущей эпохе. Оно используется при обмене информацией с другими процессами. Если эпоха одного из процессов меньше другого, он переходит в более новую эпоху. Если *кандидат* или *лидер* выясняет, что он находится в отстающей эпохе, то автоматически переходит в состояние *последователя*.

Журналы и записи

Каждый процесс хранит свою копию реплицируемого журнала. Журнал состоит из последовательности записей. Запись журнала включает в себя: индекс в журнале, эпоху, в которой она была добавлена, и команду. Эпоха используется для обнаружения несогласованности журналов различных процессов. Запись может быть зафиксированной и не зафиксированной.

Гарантируется, что зафиксированные записи не будут удалены из журнала. Они могут быть безопасно использованы, компонентами, работавшими поверх Raft. Незафиксированные записи могут быть отброшены и заменены другими записями.

2.4.2 Основные фазы работы

Все процессы начинают свою работу из состояния последователь. Из-за отсутствия лидера, через определенный промежуток времени последователи начнут переходить в состояние кандидат. Как только последователь становится кандидатом, последний переходит в новую эпоху и инициирует процесс выборов. Кандидат осуществляет сбор голосов от остальных процессов и в случае получения большинства переходит в состояние лидера. Став лидером, он начинает рассылать уведомления о своей работе, тем самым предотвращая появление новых кандидатов.

Для выбора одного лидера в одну эпоху, процесс может проголосовать в течение эпохи только один раз. Избиратель отдает голос кандидату лишь в том случае, если журнал последнего новее. Это необходимо, чтобы в качестве лидера мог быть выбран процесс с актуальным журналом.

В процессе выборов голоса могут быть распределены между несколькими кандидатами. Для выхода из данной ситуации вводится понятие времени проведения выборов. Если в течение данного промежутка времени не был выбран лидер, кандидат начинает новые выборы. Для уменьшения вероятности равномерного распределения голосов величина времени проведения выборов выбирается случайно (150-300 мс).

2.4.3 Работа с журналом

Лидер обрабатывает запросы клиентов для изменения журнала. Каждый запрос содержит команду. Полученная команда сохраняется в ново созданной, незафиксированной записи и рассылается остальным процессам, чтобы они

включили ее в свои журналы. В случае успешного включения записи на большинстве процессов, лидер фиксирует ее и рассылает уведомление об этом.

Raft поддерживает следующие свойства, которые вместе обеспечивают свойство согласованности журналов:

- Если две записи в разных журналах имеют одинаковый индекс и номер эпохи, из этого следует, что они содержат одну и ту же команду.
- Если две записи в разных журналах имеют идентичный индекс и номер эпохи, то все предыдущие записи идентичны.

Первое свойство следует из того, что лидер создает только одну запись с одним индексом в одну эпоху, и записи не меняют позицию в журнале. Второе свойство гарантируется за счет проверки согласованности, выполняемой при каждой репликации записей. Если последователь не найдет запись с предыдущей эпохой и индексом, то он не включит новую запись в свой журнал. Поэтому после каждой успешной репликации можно быть уверенным, что журналы лидера и последователя идентичны.

2.4.4 Доступность

Доступность Raft неизбежно зависит от временных интервалов. Выбор лидера – это тот аспект, где временные интервалы наиболее важны. Алгоритм способен выбирать и поддерживать стабильного лидера, пока сохраняются следующие временные соотношения.

$$broadcastTime \ll electionTimeout \ll MTBF$$

- `broadcastTime` – среднее время, которое процесс тратит для отправки запроса параллельно к другим и получает ответ.
- `electionTimeout` – время проведения выборов.
- `MTBF` – среднее время между сбоями для одного процесса (раздел 1.3).

2.4.5 Применение

Raft может быть использован как основа для написания обширного набора распределённых систем. Его гарантии позволяют создавать строго согласованные системы, позволяя справляться с широким спектром отказов. С его помощью можно реализовать:

- Распределённые транзакции в базах данных
- Рассылку общей последовательности в системе обмена сообщениями
- Службы распределенной блокировки

Реализации данного алгоритма используются в таких продуктах как: Kubernetes, RabbitMQ, CockroachDB.

2.5 Модификация алгоритма Raft

В ходе выполнения магистерской диссертации была произведена модификация исходный алгоритм Raft. Она позволит улучшить показатели доступности алгоритма на операциях чтения.

Оригинальный алгоритм достижения распределенного консенсуса Raft в контексте CAP теоремы относится к CP системе, акцент сделан на согласованности данных (раздел 1.5.2). Все операции проходят через единственный узел – лидер. Алгоритмом гарантируется следующее, если клиент получил уведомление о применении команды, то все последующие запросы будут видеть результат ее применения. При таком положении дел доступность системы во много зависит от производительности, выбранного в данный момент, лидера. Операции чтения и записи могут осуществляться до тех пор, пока сохраняются условия работоспособности алгоритма (функционирование более половины узлов). Для многих задач вышеописанные гарантии излишне строги. Путем ослабления некоторых из них и нарушая

абсолютную согласованность, можно добиться увеличения показателя доступности.

Для начала рассмотрим более подробно последовательность действий, выполняемых оригинальным алгоритмом.

1. Лидер получает команду от клиента.
2. Лидер добавляет новую запись, в локальный журнал, сохраняя команду, индекс записи и номер текущей эпохи.
3. Лидер реплицирует запись другим узлам – последователям.
4. Последователь принимает запись от лидера, добавляет ее в свой журнал, после чего возвращает ответ лидеру о том, что команда успешно добавлена.
5. Как только лидер успешно реплицировал запись на большинство узлов (включая себя), лидер фиксирует команду.
6. Лидер рассылает уведомление о том, что команда является зафиксированной.
7. Лидер выполняет зафиксированную команду.
8. Лидер уведомляет клиента о выполнении команды.

Такая последовательность действий позволяет всегда поддерживать актуальное состояние лидера. Операции чтения осуществляются за счет чтения состояния лидера. Основная проблема заключается в необходимости постоянного доступа к нему. Из-за особенностей работы сети, нарушений в работе оборудования, клиент и часть последователей могут оказаться отделены от лидера. Также новый лидер может оказаться недоступен из-за настроек политик доступа. Таким образом, алгоритм не сможет обслужить запросы. С целью обхода данной проблемы, предлагается следующие модификацию исходного алгоритма:

Обеспечить возможность выполнения операций чтения не только за счет лидера. А также последователей, с указанием размера кворума на операцию и

временем ожидания ответов. Таким образом, выполнение чтения выглядит следующим образом:

1. Узел, получивший запрос предпринимает попытку чтения с других узлов, их количество определяется клиентом.
2. Узел ожидает ответ в течение ограниченного времени, которое указывается клиентом в запросе.
3. Получив необходимое количество ответов, узел сравнивает полученные результаты на основе номера последней примененной записи и эпохи, таким образом, возможно, установить наиболее актуальный результат.
4. Отправка ответа клиенту с самыми актуальными данными.

Наглядно принцип взаимодействия демонстрирует диаграмма последовательности (Рисунок 10). Клиент не может получить доступ к лидеру, поэтому запрос отправляется одному из последователей. Чтобы применить операцию, она должна быть реплицирована на большинство узлов. Из-за этого велика вероятность, что клиент получит самые актуальные данные.

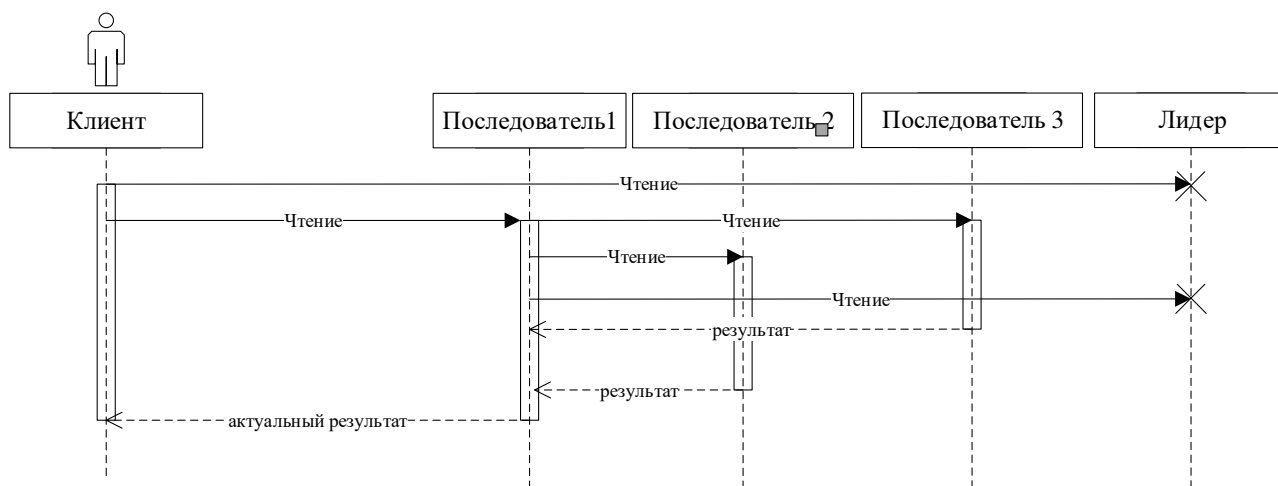


Рисунок 10 – Диаграмма последовательности работы модифицированного алгоритма

Следующая диаграмма демонстрирует, как за счет ограничения времени работы можно добиться увеличения времени реакции системы (Рисунок 11). Операция чтения успешно выполняется на двух узлах, лидер в виду

загруженности не может ответить в установленный срок. Клиенту отправляется ответ, выбранный из результатов успешного чтения.

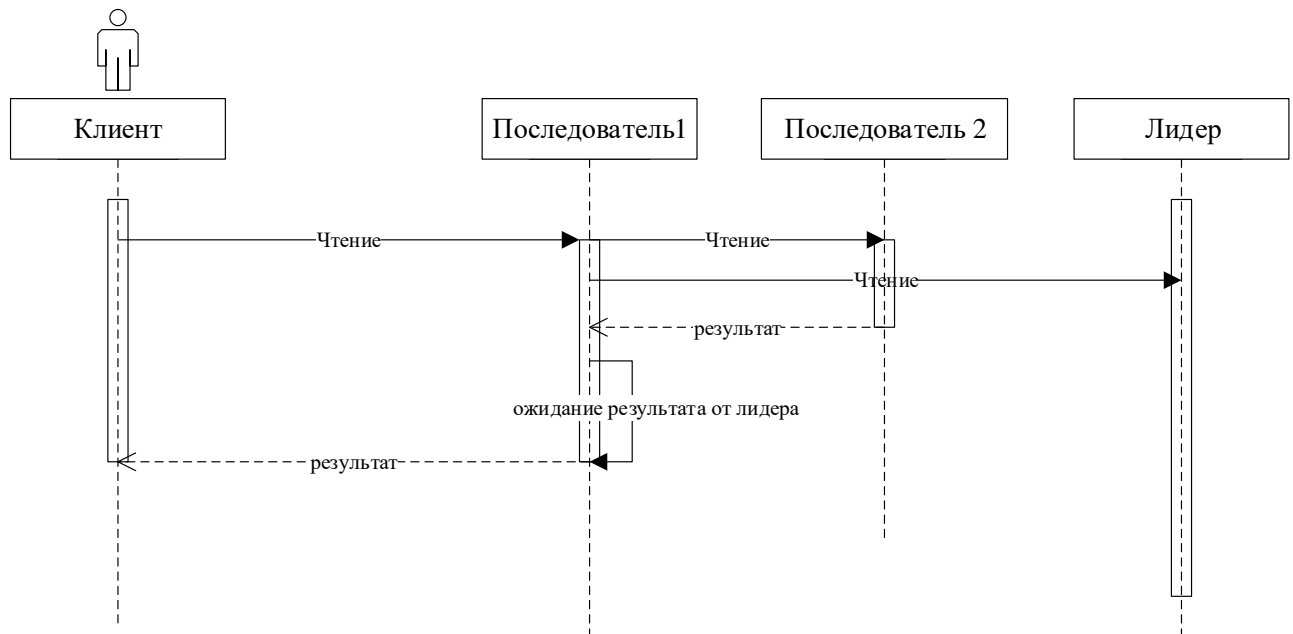


Рисунок 11 - Диаграмма последовательности работы модифицированного алгоритма с лимитом времени ожидания результата

Таким образом, изначальная линейаризуемая согласованность, уменьшается лишь до последовательной согласованности. Ее достаточно для широкого круга приложений. Появляется возможность осуществить чтение даже при наличии доступа только к одному узлу. Указание количества узлов для чтения и времени ожидания результата позволяет балансировать между ожидаемой актуальностью результата и необходимой доступностью со стороны клиента. В итоге, операции чтения могут выполняться в условиях отсутствия доступа к лидеру, с настраиваемым уровнем доступности и актуальности.

3 Проектирование архитектуры отказоустойчивой и высокодоступной системы

Существует множество определений понятию архитектуры. Но нет одного, полностью отражающего все его смыслы. Ральф Джонсон дал простое определение, описывающее главную суть: “Архитектура – это все что важно в системе, чем бы оно ни было”. Архитектура не представляет собой конечную законченную сущность, это процесс движения к цели, формирующейся и изменяющейся под влиянием требований. Для системы с минимальной функциональностью, эксплуатируемой в течение короткого промежутка времени, эффективно использовать одну архитектуру. Для масштабной, отказоустойчивой системы, с долгим сроком использования, необходимо совсем другая архитектура.

Целью данной главы является обзор различных архитектурных подходов к построению информационных систем и их особенностей. Выбрать наиболее эффективную архитектуру для построения на её основе высокодоступной и отказоустойчивой системы.

3.1 Общие принципы

Основная ценность системы с точки зрения ее использования – это ее функциональность, в ней кроется основная причина ее создания. Скрытой ценностью является структура системы – ее внутреннее устройство, которая во много определяет надежность и отказоустойчивость, а также влияет на корректность ее поведения. С одной стороны, факт, что правильность поведения никак не зависит от структуры системы. Однако неразборчивая структура часто является источником ошибок при реализации

функциональности. Что в конечном итоге приводит к нарушению безопасности системы, порче данных и ее полному отказу.

Эффективная архитектура позволяет выстроить правильное взаимодействие компонентов, определить четкие границы ответственности [20]. Появляется возможность вносить новые изменения в одни части системы, не нарушая работу других. Структура, архитектуры системы прямо или косвенно влияет на весь ее жизненный цикл: проектирование, разработку, функционирование, развёртывание, сопровождение.

Требования оказывают большое влияние не только на функциональную составляющую системы, но также формируют ее внутреннюю структуру. Они работают в качестве мотивации для принятия тех или иных решений. Одни требования могут быть явными, в них определяются набор компонентов системы и способы их взаимодействия. Неявные, оказывают косвенное воздействие, архитектура меняется с целью обеспечения возможности их выполнения. Например, требование к модульности явно влияет на состав и размер компонентов системы. С другой стороны, требование отказоустойчивости предполагает отсутствие единой точки отказа, то есть система не может состоять из одного компонента. Требование поддержания определенного уровня производительности диктует применение горизонтального масштабирования, что в свою очередь подразумевает одновременный запуск нескольких компонентов и организацию их совместной работы.

Некоторые требования часто противоречат друг другу. Так, может быть невозможно, создать отказоустойчивую систему с высоким уровнем производительности. При проектировании архитектуры необходимо учитывать допустимые границы параметров условий и прийти к некоторому балансу. Нарушение требований может считаться сбоем в системе. Использование

функций пригодности (раздел 2.1) позволяет выявить и предупредить выход показателей за заданные границы.

3.2 Виды архитектур

Выбор той или иной архитектуры для конечной системы зависит от задач и требований, с которыми ей придется столкнуться. Для того чтобы взвесить все за и против, необходимо ввести параметры, в соответствии с которым можно провести сравнение различных архитектур.

Любая достаточно крупная система состоит из множества частей. Она может представлять собой логическую структуру – *модуль*, формирующий новую функциональную или структурную сущность. Физический способ формирования связей образует *компонент* системы, например, пространство имен, объединяющее несколько классов. Компоненты и модули являются одним из видов воплощения инкапсуляции. Их применение позволяет повторно использовать уже написанные компоненты. Они могут иметь различный масштаб в зависимости от уровня рассмотрения системы, а также от вида архитектуры. Сравнение архитектур может строиться на основе соотнесения компонентов и модулей, выделяемых в них.

Связность – это ключевая характеристика, для описания набора компонентов. Она определяет, как различные компоненты зависят друг от друга, и взаимодействуют между собой. Возможность развертывания отдельных компонентов независимо друг от друга доступна слабосвязанным частям системы. В свою очередь, их внутренние элементы должны обладать сильной связью. Разделение одного модуля на несколько компонентов, не позволяет разорвать их, но приводит к усложнению их взаимодействия. В дополнение, становится невозможно, использовать преимущества, которые возникают при использовании слабосвязанных компонентов: тестируемость, отдельная

разработка, возможность внутренней реорганизации, независимость развертывания.

Задача хорошей архитектуры выявить компоненты системы, которые должны обладать сильной связностью для обеспечения максимальной выгоды при минимальных накладных расходах.

3.2.1 Монолитная архитектура

Монолитная архитектура на самом высоком уровне представляет собой единый компонент. Таким образом, система – это одна программа. Ее внутренние компоненты обладают сильной связанностью и их невозможно развернуть независимо друг от друга. Распространённой разновидностью является слоистая монолитная архитектура (Рисунок 12). Каждый слой отвечает за определенные технические возможности. Изолированные слои взаимодействуют друг с другом, используя некоторый интерфейс.

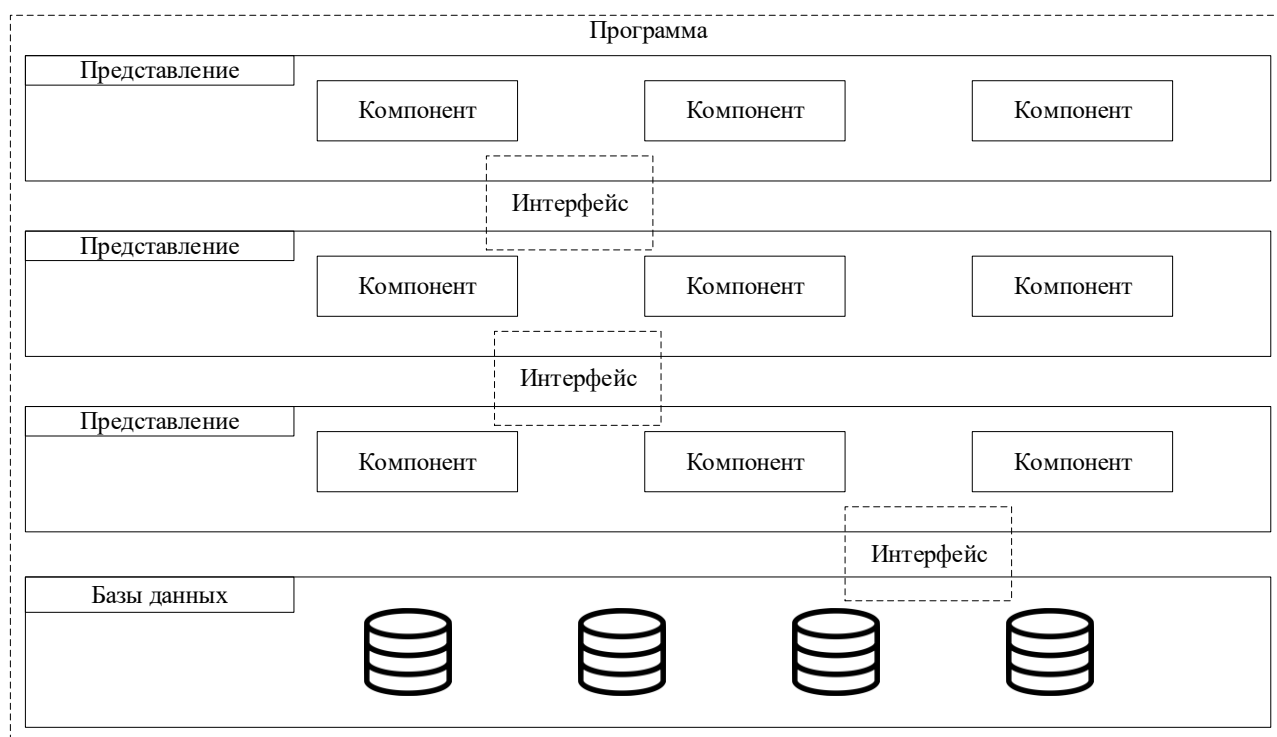


Рисунок 12 – Слоистая монолитная архитектура

Монолитная архитектура эффективна при создании небольших систем [21]. Наличие единой кодовой базы позволяет эффективно использовать

инструменты разработки, с их помощью можно легко вносить изменения во всю систему. Внутренние компоненты взаимодействуют с помощью простых локальных вызовов. Наличие одной единственной программы упрощает инфраструктуру развертывания и эксплуатации.

С точки зрения отказоустойчивости монолитная архитектура всегда имеет единую точку отказа. Все внутренние компоненты работают в едином адресном пространстве. Из-за ошибки, возникшей в любом внутреннем компоненте, даже самом незначительном, может произойти нарушение работы всей системы. Их локализация требует проведения комплексной отладки. Монолитная система не может масштабироваться горизонтально.

3.2.2 Плагиновая архитектура

Система, реализованная с применением данной архитектуры, содержит два основных типа компонентов: ядро и плагины. Плагин – это независимый компилируемый программный модуль, динамически подключаемый к основной программе – ядру, предназначенный для расширения или изменения ее возможностей. Ядро является сложным компонентом, реализующим основные алгоритмы работы системы. При разработке ядра создаются точки привязки для расширения или изменения исходной функциональности за счет плагинов [22]. Плагины независимы между собой, за счет этого они могут разрабатываться отдельно, легко тестироваться и развертываться. Связность образуется только между ядром и отдельными плагинами, на основе контрактов предоставляемых первым.

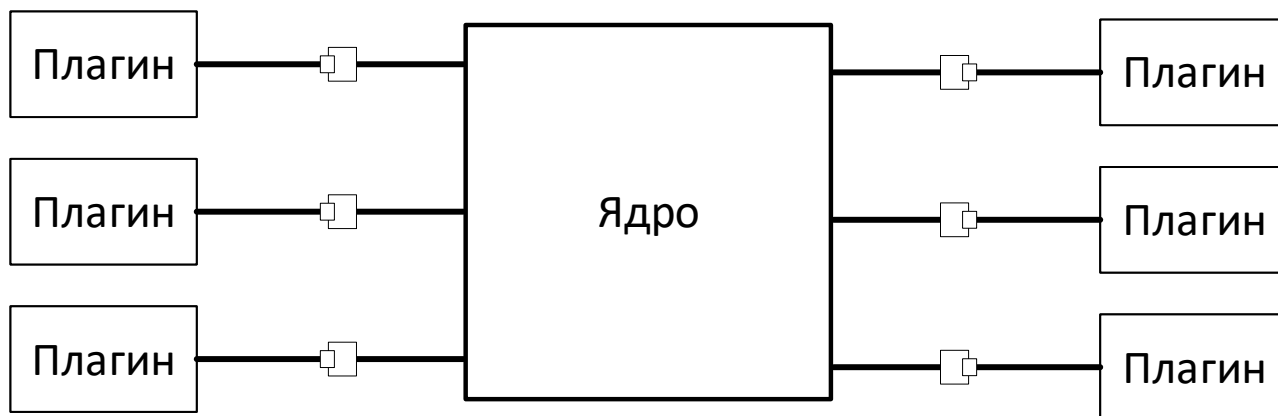


Рисунок 13 – Плагинная архитектура

Плагинная архитектура, в отличие от монолитной, подходит для разработки комплексных, многокомпонентных систем. Она обеспечивает достаточный уровень изоляции изменений, а также возможность независимой разработки, за счет применения версионирования. Современные браузеры, среды интегрированной разработки, компиляторы, а также некоторые компоненты операционных систем разрабатываются в соответствии с данной архитектурой.

Однако система, выполненная с использованием данного подхода, как и при монолитном, является одной программой, являющейся единой точкой отказа. Ее применение не позволяет добиться отказоустойчивости и высокого уровня доступности.

3.2.3 Сервисная архитектура

Выше описанные архитектуры рассматривают вопрос декомпозиции с точки зрения технических особенностей. Так, в монолитной архитектуре со слоистой организации, слои образуются вокруг таких задач как: пользовательский интерфейс, нормализация и хранение данных и т. д. В итоге многие логически связанные задачи, разбросаны по разным компонентам, например, валидация может осуществляться на нескольких слоях одновременно. Также и в плагинной архитектуре, ответственность за выполнение определенной функции может, возложена на группу плагинов. Это

создает неявную логическую связность, что требует усложнение координации между разработчиками отдельных частей.

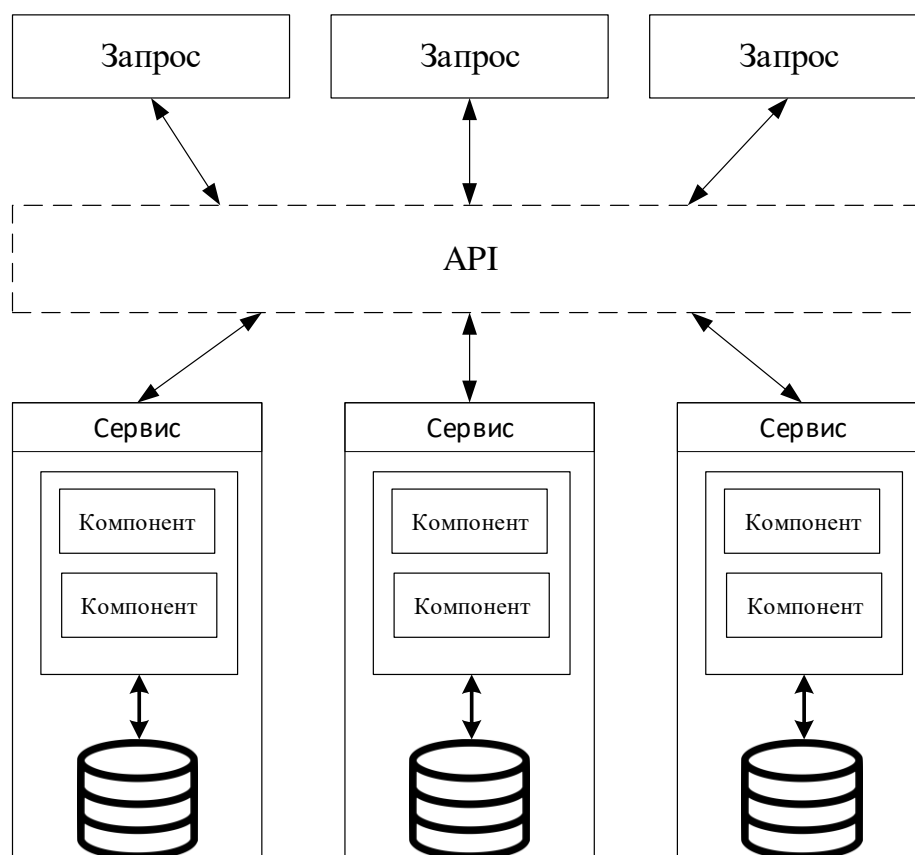


Рисунок 14 – Сервисная архитектура

Сервисная архитектура, на самом высоком уровне, разделяет всю в соответствии с предметной областью. Самым высокоуровневым компонентом является сервис – отдельная программа, в которой реализованы полезные функции. В свою очередь, внутри него разделение может строиться по техническим особенностям. Для доступа к его функциям необходимо задействовать четко определенный минималистичный API, который инкапсулирует внутреннюю реализацию. Это позволяет создать полностью независимый, ограниченный контекст, пригодный для решения определенной задачи. Каждый сервис самодостаточен, он управляет хранением и обработкой своих данных. Выполнение высокоуровневых задач на уровне системы осуществляется путем координации работы отдельных сервисов. Взаимодействие может протекать как синхронно – удаленный вызов процедур,

так и асинхронно – очереди сообщений. Данная архитектура позволяет достигнуть низкой связности отдельных сервисов.

Основной целью данного подхода добиться максимально возможной независимости работы отдельного сервиса, за счет чего достигается сохранность работоспособности всей системы при отказе некоторых из них. Техническая архитектура формируется внутри границ сервиса, на основе предметной области. Слабая связность позволяет вести независимую разработку каждого сервиса, упрощает процессы тестирования, обеспечивает возможность автоматизирования многих процессов в процессе эксплуатации системы. Интеграция происходит максимально прозрачным образом за счет обмена сообщениями, такие низкоуровневые детали как схема базы данных скрыта от других сервисов [24].

У сервисной архитектуры есть свои недостатки, и трудности при ее использовании. Для ее эффективного применения необходимо провести тщательный процесс декомпозиции предметной области. Неверно выделенные сервисы могут образовываться скрытые связи между собой, тем самым снижая эффективность архитектуры в целом. Децентрализованная природа вносит проблемы присущие распределенным системам. Приходится прилагать дополнительные усилия для сохранения согласованности данных. Необходимо учитывать то, что взаимодействие осуществляет не с помощью локальных вызовов, а за счет механизмов межпроцессорного взаимодействия, часто с использованием сети. Вдобавок наличие множества программ усложняет администрирование системы.

Данная архитектура рассматривает всю систему как набор отдельных программ. Каждая из них может развертываться на независимых серверах. При такой организации отсутствует единая точка отказа. Выход из строя отдельного сервера не приводит к отказу всей системы. Ошибки, допущенные при программировании одного сервиса, не могут повлиять на правильность

функционирования другого. Принцип изолированности в сервисном подходе позволяет справляться с частичными отказами. Слабая связность допускает дублирование сервисов с целью применения горизонтального масштабирования. Суммируя все выше изложенное можно сделать вывод о пригодности сервисной архитектуры в качестве основы при разработке отказоустойчивой и высоко доступной системы.

3.3 Архитектура отказоустойчивой системы

Приведенный выше обзор дает общее представление об архитектурах, их достоинствах и недостатках. Однако только сервисная архитектура обладает тем набором свойств, необходимых для создания отказоустойчивой и высоко доступной системы. В данном разделе предметно разбирается вопрос применения сервисно-ориентированного подхода в контексте отказоустойчивости. Представлены конкретные методики и подходы, увеличивающие надежность сервисов. Описываются часто возникающие проблемы и способы их решения.

Декомпозиция определяет, то настолько продуктивна будет конечная система, состоящая из множества сервисов. Неверное разделение делает систему неустойчивой для внесения в нее изменений, как следствие, уменьшая ее надежность. Таким образом, необходимо проводить глубокий анализ предметной области и придерживаться известных практик, для определения четких границ между сервисами. Это позволяет повысить надежность всей системы. Следует придерживаться следующих рекомендаций:

Применение *принципа единой ответственности*. Согласно ему, у сервиса должна быть только одна причина для изменения. Необходимо определить ту единственно сферу, за которую отвечает каждый отдельный компонент, а также агентов, которые взаимодействуют с ним. Понимание потребностей агентов позволяет разгрузить сервис от лишних деталей,

упростив тем самым его интерфейс. Сервисы, разработанные с использованием данного подхода, получаются компактными и устойчивыми к изменениям. Принцип *согласованного изменения*: причины изменения отдельных компонентов, включенных в один сервис должны быть одинаковы и затрагивать каждый из них. Следование ему помогает минимизировать количество сервисов, затрагиваемых отдельными изменениями, и значительно упрощает поддержку системы [20].

Сервисы взаимодействуют между собой через сеть. Следует учитывать наличие задержки в ней при проектировании взаимодействия между ними. Неэффективная стратегия извлечения данных может привести к уменьшению общей производительности. Нужно учитывать природу каждой операции и отдавать предпочтение асинхронному взаимодействию над синхронным. Это увеличивает реакцию системы и ее общую доступность. Этой же цели служит шаблон *предохранитель*, он выступает в качестве прокси для всех запросов. В нем определяется допустимый порог последовательных отказов. Его превышение дает право отклонять последующие запросы в течение определенного промежутка времени. Его применение позволяет достигнуть следующих преимуществ:

- Устранение длительных каскадных блокировок.
- Обеспечение приемлемого времени обработки запроса при наличии высокой нагрузки.

Шаблон *перемычка* нацелен на увеличение отказоустойчивости. Основная идея заключается в объединении экземпляров сервисов в изолированные группы, исходя из требований к их доступности. Такой подход позволяет изолировать отказы и сохранять функциональность, предотвращая распространение сбоя за границы группы.

Основной проблемой данной архитектуры является поддержание согласованности данных между сервисами. Некоторые операции могут

требовать одновременного изменения данных в нескольких сервисах. Однако, в виду отсутствия единой базы данных, нет возможности выполнить изменение в рамках одной ACID-транзакции. Одним из решений является, примените двухфазной блокировки и реализации на ее основе распределенной транзакции. Однако данный подход сложен в применении и существуют сценарии, в которых двухфазная блокировка приводит к возникновению взаимной блокировки, что приводит нарушению отказоустойчивости. Другим подходом является *повествование* – это последовательность локальных транзакций, выполняемых в отдельных сервисах, которые координируются путем обмена сообщений [25]. Благодаря ему обеспечивается возможность достижения конечной согласованности данных в отдельных сервисах. Данный подход больше соотносится с BASE-семантикой, чем с традиционными ACID-транзакциями. Главной проблемой повествования является недостаточная изолированность. Одновременно в системе могут протекать множество операций и повествований, связанных с ними. Каждая локальная транзакция, выполняемая в отдельном сервисе, изначально не учитывает, что параллельно может протекать повествование, которое опирается на изменяемые данные. Этот факт может привести к аномалиям, ведущих несогласованности. Распространёнными аномалиями являются:

- Потеря обновлений – одно из повествований перезаписывает изменения, внесенные другим, не считывая их.
- “Грязное” чтение – одно повествование считывает промежуточное состояние, обновляемое другим повествованием.
- Неповторяемое чтение – два разных этапа повествования читают одинаковые данные, но получают разные результаты, из-за вмешательства другого повествования.

Для противодействия им требуется применять следующие контрмеры:

- Семантическая блокировка – семантика выполнения операции исключает появление аномалий.
- Коммутативные обновления – свойство операций, очередность выполнения которых не влияет на конечный результат.
- Повторное чтение – чтение данных перед их изменением, для проверки их согласованности, предохраняет от “грязного” чтения.
- Версионирование операций – позволяет переупорядочивать выполнение локальных операций, с целью достижения глобальной упорядоченности повествований.

Применение вышеописанных подходов позволяет приблизиться к гарантиям обеспечиваемых стандартными транзакциями, при этом, не опираясь на взятие глобальных блокировок, ведущих к уменьшению доступности.

4 Отказоустойчивое веб приложение

Разработанное веб приложение позволяет продемонстрировать работоспособность алгоритмов, рассмотренных в главе 2 и теоретических положений, изложенных в главе 1. А также проверить эффективность применения сервисной архитектуры, обзор которой приведен в главе 3, для построения отказоустойчивого приложения. При разработке решались практические задачи присущие многим информационным системам: хранение, обработка информации, коммуникация. Использовались актуальные технологии, языки программирования: C# и Golang; базы данных; средства управления контейнерами.

Предметной областью приложения, выбран интернет магазин. Эта задача хорошо изучена и проработана, что позволяет сосредотачивать внимание не на проблеме декомпозиции, а на обеспечении отказоустойчивости и повышении доступности конечной системы.

4.1 Архитектура

Создание отказоустойчивого веб приложения, требует использование соответствующей архитектуры. Для его разработки выбрана сервисная архитектура. Данный выбор основан на анализе, который проведен в 3-й главе. Согласно ему, сервисная архитектура обладает полезными свойствами, необходимыми для создания отказоустойчивого приложения. Система разделена на следующие сервисы (Рисунок 15):

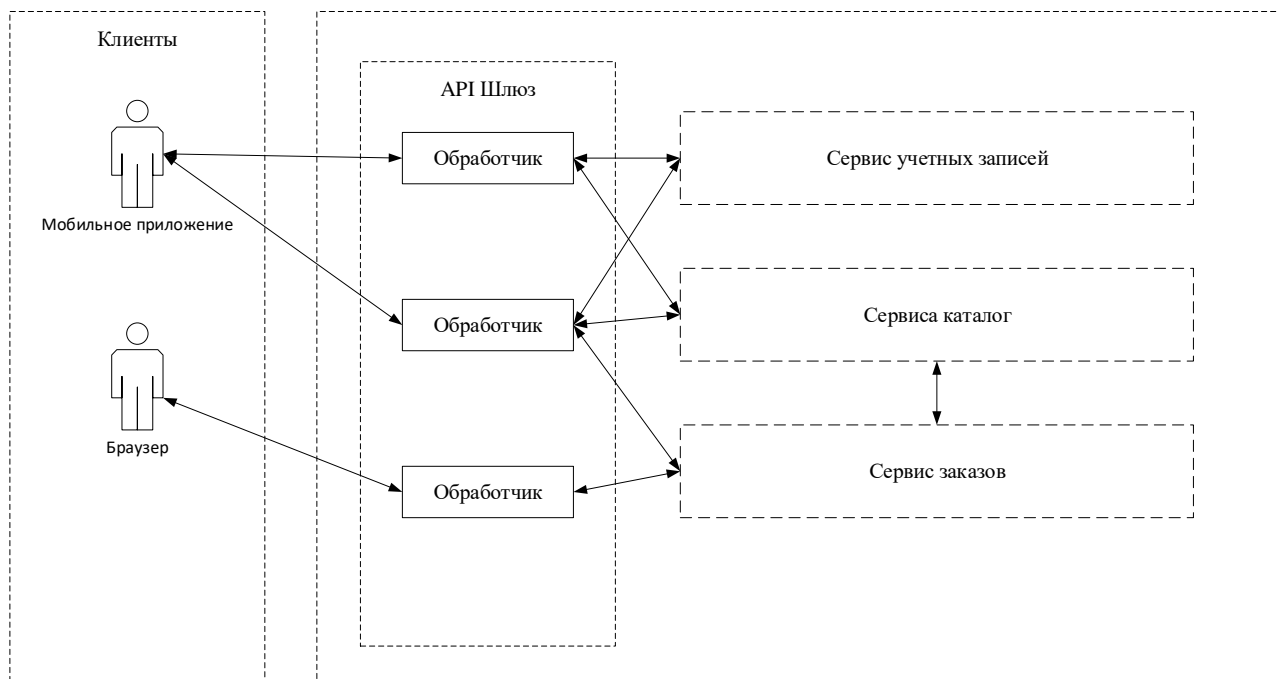


Рисунок 15 – Архитектура приложения

1) API шлюз. Сервис, с которым непосредственно взаимодействуют клиенты. Все запросы от них перенаправляются другим сервисам, либо выполняются самим шлюзом. Он объединяет результаты работы других сервисов. Шлюз инкапсулирует комплексность внутреннего взаимодействия между сервисами. В нем реализованы пограничные функции: аутентификации и авторизация, журналирование, политики повторных попыток и времени выполнения запросов. API-шлюз не хранит состояние, необходимое для обработки запросов, он может быть реплицирован и поставлен за балансировщиком нагрузки.

2) Сервис каталога. В нем хранятся данные о товарах: наименование, описание, тип, количество, цена и т. д. За сохранность отвечает база данных развернутая на кластере из нескольких узлов. Таким образом, данный сервис может пережить отказ нескольких узлов. Некоторые из функций предоставляются только авторизованным пользователем.

3) Сервис учетных записей пользователей. Хранит информацию о пользователях системы. Выход из строя данного сервиса, приведет к

отключению функций, завязанных на авторизации, но не приведет к прекращению работы всего приложения.

4) Сервис заказов. Занимается хранением и модифицированием заказов пользователей, а также хранит их историю. За сохранность заказов отвечает база данные, с высоким уровнем согласованности. Отключение данного сервиса не позволяет пользователем просматривать и создавать новые заказы, но не вредит остальным функциям приложения.

4.2 Сервис учетных записей

Основная задача данного сервиса – это хранение данных учетных записей пользователей, а также проведение аутентификации. Для поддержания сессии пользователя используется JWT. JWT (JSON Web Token) – представляет собой набор подписанных данных в формате JSON. Им можно обмениваться как внутри системы, так и через посредников. Факт модификации данных не может быть скрыт. Данный механизм подходит для поддержания сессий пользователей в распределенной среде, такой как сервисы. Сервис учетных записей написан на языке программирования Golang и предоставляет REST API по протоколу HTTP.

Аутентификация пользователя осуществляется отдельным обработчиком (Листинг 1). На вход ему поступает HTTP запрос, в теле которого содержатся данные учетной записи пользователя в формате JSON. По полученным данным производится поиск пользователя, если пользователь с указанными данными не найден в системе, возвращается ошибка. Если пользователь найден, его данные кодируются в JWT, который возвращается в ответе и сохраняется в cookie.

Листинг 1 Обработчик запроса аутентификации пользователя

```
// signin авторизует клиента
func signin(rs http.ResponseWriter, rq *http.Request) {

    var creds Credentials

    // Получаем данные для ауторизации
    err := json.NewDecoder(rq.Body).Decode(&creds)
    if err != nil {
        log.Println(err)
        // Не подходящий запрос
        rs.WriteHeader(http.StatusBadRequest)
        return
    }

    u, err := userStore.Get(creds.Password)
    if err != nil {
        rs.WriteHeader(http.StatusUnauthorized)
        return
    }

    if u.Name != creds.Username {
        rs.WriteHeader(http.StatusUnauthorized)
        return
    }

    expire, token, err := jwt.Create(u)
    if err != nil {
        rs.WriteHeader(http.StatusUnauthorized)
        return
    }

    // Сохраняем токен в куках
    http.SetCookie(rs, &http.Cookie{
        Name:     "session_token",
        Value:     token,
        Expires:  expire,
    })
}
```

Для взаимодействия с сервисом доступы, следующие конечные точки:

<https://<host>/user> – доступ к ресурсу пользователь (user). Таблица 5 содержит описание методов взаимодействия с данным ресурсом.

Таблица 1 – Методы взаимодействия с ресурсом пользователь

Метод	Параметры	Описание	Пример
POST	username — имя пользователя password — пароль пользователя	Аутентификация пользователя, возвращает JWT	https://shop.ru/user?username=Nikita&password=password1
POST	token — токен доступа	Обновляет токен пользователя	https://shop.ru/user?token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1bm90dXN0LmlpOpDAsKw9EBHPmwN1Fa9PxVwB70s3nx8y4D4BsY5lg

4.3 Сервис каталога

Сервис каталога предоставляет функциональность, связанную с хранением и управлением товарами. Благодаря ему пользователи могут просматривать товары, которые находятся в наличии. Сервис позволяет производить сортировку по типам и брендам. Данные выдаются постранично, это снижает количество передаваемой информации в каждом запросе. Товар содержит в себе следующую информацию: наименование, описание, тип, бренд, количество и цену. Пользователи могут просматривать информацию о товарах. Администраторам доступны функции управления каталогом: добавление новых товаров, удаление и изменение текущих.

Связь с сервисом осуществляется по REST API поверх HTTP. Для его реализации использован язык программирования C# и ASP NET CORE Framework. Для взаимодействия с сервисом доступны, следующие конечные точки:

https://<host>/products - доступ к ресурсу товары (products). Таблица 2 содержит описание методов взаимодействия с данным ресурсом.

Таблица 2 – Методы взаимодействия с ресурсом товар

Метод	Параметры	Описание	Пример
GET	pageSize – количество заказов, pageIndex – номер страницы	Получить страницу с товарами	https://shop.ru/products?pageSize=10&pageIndex=4
GET	IDs – идентификатор товаров	Получить заказы по их идентификаторам	https://shop.ru/products?IDs=19d5ff73-b121-4192-8e6c-9ba707b9c494,4fb8b7b2-ef67-4d10-b01c-7f91559001dd
GET	type – тип товаров brand – бренд товара	Получить товары с определенным типом и брендом	https://shop.ru/products?type=”одежда”&brand=”adidas”

https://<host>/products/types - доступ к ресурсу типы (types). Таблица 3 содержит описание методов взаимодействия с данным ресурсом.

Таблица 3 – Методы взаимодействия с ресурсом типы

Метод	Параметры	Описание	Пример
GET	Не имеет параметров	Получить типы товаров	https://shop.ru/products/types
POST	Name – название типа	Добавить новый тип	https://shop.ru/products/types?name=”компьютер”
DELETE	ID – идентификатор типа	Удалить тип	https://shop.ru/products/types?ID=4fb8b7b2-ef67-4d10-b01c-7f91559001dd

https://<host>/products/brands - доступ к ресурсу бренды (brands). Таблица 4 содержит описание методов взаимодействия с данным ресурсом.

Таблица 4 – Методы взаимодействия с ресурсом бренды

Метод	Параметры	Описание	Пример
GET	Не имеет параметров	Получить бренды товаров	https://shop.ru/products/brands
POST	Name – название бренда	Добавить новый бренд	https://shop.ru/products/brands?name=”apple”
DELETE	ID – идентификатор бренда	Удалить бренд	https://shop.ru/products/brands?ID=4fb8b7b2-ef67-4d10-b01c-7f91559001dd

Выдача товара осуществляется постранично, это позволяет снизить нагрузку на канал передачи данных. Для извлечения сведений о товарах обработчик запроса обращается к хранилищу данных (Листинг 2). В обращении

указывается количество необходимых заказов и место, начиная с которого обеспечивается выдача. Запросы выполняются в асинхронном стиле, что позволяет увеличить общую пропускную способность.

Листинг 2. Обработчик запроса на получение товаров

```
[HttpGet]
[Route("products")]
public async Task<IActionResult> ItemsAsync([FromQuery]int pageSize = 10, [FromQuery]int pageIndex = 0, string ids = null)
{
    if (!string.IsNullOrEmpty(ids))
    {
        var items = await GetItemsByIdsAsync(ids);

        if (!items.Any())
        {
            return BadRequest("invalid ids [id,id,id,...]");
        }

        ChangeUriPlaceholder(items);
        return Json(items);
    }

    var randomItems = await repo.GetItemsWithOptionsAsync(pageSize * pageIndex, pageSize);
    var count = await repo.GetItemscount();
    var model = new PaginatedItemsViewModel(pageIndex, pageSize, count, randomItems);
    ChangeUriPlaceholder(randomItems);

    return Json(model);
}
```

4.4 Сервис заказов

Данный сервис отвечает за хранение и обработку заказов. С его помощью пользователи могут создавать новые заказы в приложении. Заказ включает в себе информацию о текущем статусе, товарах и их количестве, цене и пользователе, который его совершил. Администраторы, могут просматривать заказы всех пользователей, изменять их статусы или удалять. Обычные пользователи могут просматривать историю своих заказов.

Данный сервис не является узким местом в системе. В случае его недоступности или отказа, это не затронет всю функциональность приложения. Недоступными станут функции, связанные с управлением заказами. Однако многие сбои могут быть замаскированы путем создания нескольких реплик данного сервиса.

Сервис заказов написан с использованием языка программирования C#. Взаимодействие с ним осуществляется по REST API поверх HTTP. Доступны следующие конечные точки:

<https://<host>/order/<ID>> - доступ к ресурсу заказ (order). Таблица 5 содержит описание методов взаимодействия с данным ресурсом.

Таблица 5 – Методы взаимодействия с ресурсом заказ

Метод	Параметры	Описание	Пример
GET	Не имеет параметров	Получить информацию о заказе с конкретным идентификатором	https://shop.ru/order/0152ea6b-9f05-4b98-bda2-958bc40cfa2d
PUT	userID – идентификатор пользователя, который создает данный заказ; catalogItems – список идентификаторов товаров	Создает новый заказ с указанным идентификатором	https://shop.ru/order/0152ea6b-9f05-4b98-bda2-958bc40cfa2d?userID=19d5ff73-b121-4192-8e6c-9ba707b9c494&catalogItems=4fb8b7b2-ef67-4d10-b01c-7f91559001dd
DELETE	Не имеет параметров	Удалить заказ по идентификатору	https://shop.ru/order/0152ea6b-9f05-4b98-bda2-958bc40cfa2d
POST	Status – статус заказ	Изменить статус заказа	https://shop.ru/order/0152ea6b-9f05-4b98-bda2-958bc40cfa2d?status=’Выполнен’

<https://<host>/orders> - доступ к ресурсу заказы (orders). Таблица 6 содержит описание методов взаимодействия с данным ресурсом.

Таблица 6 – Методы взаимодействия с ресурсом заказы

Метод	Параметры	Описание	Пример
GET	userID – идентификатор пользователя,	Получить все заказы определенного пользователя	https://shop.ru/orders?userID=19d5ff73-b121-4192-8e6c-9ba707b9c494
GET	pageSize – количество заказов, pageIndex – номер страницы	Получить страницу заказов	https://shop.ru/orders?pageSize=10&pageIndex=4

Постраничное получение заказов позволяет практически использовать канал передачи данных, передавая данные порциями, по мере возникновения необходимости в них. Обработка данного запроса заключается в обращении к хранилищу заказов (Листинг 3). В нем указывается порядковый номер, начиная с которого необходимо считать заказы и их количество. Запросы выполняются в асинхронном стиле, что позволяет увеличить количество обрабатываемых запросов в секунду.

Листинг 3. Обработчик запроса на получение страницы заказов

```
[HttpGet]
[Route("orders")]
public async Task<IActionResult> GetOrdersAsync([FromQuery]int pageSize = 10, [FromQuery]int pageIndex = 0)
{
    var orders = await repo.GetOrdersWithOptionsAsync(pageSize *
pageIndex, pageSize);
    var count = await repo.GetOrdersCount();
    var model = new PaginatedOrdersViewModel(pageIndex, pageSize
, count, orders);

    return Json(model);
}
```

4.5 API шлюз

Шлюз является точкой объединения функциональности всех сервисов, с его помощью выполняются комплексные операции и обеспечиваются некоторые политики. Он реализован на языке программирования C# и ASP NET

CORE Framework. Шлюз принимает запросы от клиентов, исполняет их, используя другие сервисы, и возвращает ответ в виде веб-страницы.

Большое количество запросов может привести к перегрузке данного сервиса, это в свою очередь приведёт к увеличению времени обработки каждого отдельного запроса. Для поддержания приемлемого времени обработки используется политика ограниченного времени выполнения запросов, а также применен шаблон *предохранитель* (раздел 3.3).

Шлюз определяет доступ, к каким ресурсам и операциям над ними есть, у каждого клиента. Перед тем как предоставить доступ к ресурсу или перенаправить запрос какому-либо внутреннему сервису производится проверка прав пользователя. Аутентифицированные пользователи хранят JWT в “куках” (cookie), таким образом, он содержится в каждом запросе, которые направлен шлюзу. Если токен присутствует и корректен, шлюз авторизует пользователя в соответствии с хранящимися в нем данными. Если клиент не авторизован для выполнения той или иной операции он будет перенаправлен на страницу аутентификации. Процесс аутентификации работает по следующей схеме. Клиент предоставляет аутентификационные данные в запросе к шлюзу. Он в свою очередь перенаправляет запрос к сервису учетных записей. Если данные корректны, шлюз получает токен пользователя, в нем содержится вся необходимая информация для авторизации клиента. Токен сохраняется в “куках” клиента, для авторизации последующих запросов (Листинг 4).

Листинг 4. Обработчик запроса аутентификации на стороне API шлюза

```
public async Task<IActionResult> SignIn(string username, string
password)
{
    string token = null;
    try
    {
        token = await _authSvc.Login(username, password);
        HttpContext.Response.Cookies.Append("access_token",
RecreateToken(token));
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex);
        ViewBag["Error"] = "Не удалось аутентифицировать
пользователя";
    }

    return RedirectToAction(nameof(CatalogController.Index),
"Catalog");
}

private string RecreateToken(string originalToken)
{
    var person = ParseToken(originalToken);
    var identity = GetIdentity(person.Username, person.Role);

    var now = DateTime.UtcNow;
    // создаем JWT-токен
    var jwt = new JwtSecurityToken(
        issuer: AuthOptions.ISSUER,
        audience: AuthOptions.AUDIENCE,
        notBefore: now,
        claims: identity.Claims,
        expires:
now.Add(TimeSpan.FromMinutes(AuthOptions.LIFETIME)),
        signingCredentials: new
SigningCredentials(AuthOptions.GetSymmetricSecurityKey(),
SecurityAlgorithms.HmacSha256));
    var encodedJwt = new
JwtSecurityTokenHandler().WriteToken(jwt);

    return encodedJwt;
}

private Person ParseToken(string token)
{
    var handler = new JwtSecurityTokenHandler();
    var jsonToken = handler.ReadToken(token);
    var tokenS = handler.ReadToken(token) as JwtSecurityToken;
```

```

        var person = new Person();

        person.Username = tokenS.Claims.First(claim => claim.Type ==
"username").Value;
        person.Role = tokenS.Claims.First(claim => claim.Type ==
"role").Value;

        return person;
    }

    private ClaimsIdentity GetIdentity(string username, string role)
    {
        var claims = new List<Claim>
        {
            new Claim(ClaimsIdentity.DefaultNameClaimType,
username),
            new Claim(ClaimsIdentity.DefaultRoleClaimType, role )
        };
        ClaimsIdentity claimsIdentity =
new ClaimsIdentity(claims, "Token",
ClaimsIdentity.DefaultNameClaimType,
ClaimsIdentity.DefaultRoleClaimType);
        return claimsIdentity;
    }

```

4.6 Инфраструктура

Создание приложения на основе сервисов является современной тенденцией в разработке. Отдельный сервис – это автономный прикладной модуль, который выполняют группу связанных предметной областью функций. По сути, он представляет собой небольшое приложение. Однако при увеличении их количества, заметно усложняется процесс управления ими. Особенно это проявляется, если используются разные технологические стеки. Решением этой проблемы является использование контейнеров для инкапсуляции в нем отдельных сервисов. Docker позволяет управлять контейнерами [26].

Docker – это технология контейнеризации. Он предоставляет разработчикам простой способ управления сервисом как отдельным контейнером. Docker решает проблемы, связанные со сборкой и

развертыванием приложений, а также обеспечивает изолированность окружения. С его помощью можно конфигурировать сеть, ограничивать использование ресурсов, обеспечивать разграничение доступа.

Docker-compose – это встроенная в Docker утилита, которая позволяет управлять группами контейнеров. С ее помощью можно удобно описывать конфигурацию нескольких сервисов и одновременно управлять процессами сборки и развертывания целого приложения. Конфигурация описывается YAML файлом.

Листинг 5 описывает конфигурацию сервиса каталога. Настраиваются необходимые переменные окружения, указывается ip-адрес контейнера, в котором будет запущен сервис. Связываются порты, для доступа к контейнеру из внешней сети хоста.

Листинг 5. Конфигурация контейнера сервиса каталога

```
version: '2.2'

services:
  catalog:
    environment:
      - ${JWT_KEY}
      - IMAGE_URL=http://localhost:5003/items/[0]/pic/
    image: catalog
    ports:
      - "5003:5002"
    build:
      context: ./CatalogService
      dockerfile: Dockerfile
    networks:
      p2p:
        ipv4_address: 172.28.5.1
```

Приложение Е содержит листинг полной конфигурации разработанного приложения. В нем описаны настройки каждого сервиса, сети, заданы переменны окружения. Сборка всего приложения осуществляется одной командой: `docker-compose build`. Разворачивание и запуск приложения выполняется с помощью команды: `docker-compose up`.

Заключение

Итогом проведенной работы являются следующие результаты: исследован материал, касающийся надежности функционирования информационных систем, а именно:

- отказы и сбои, причины их возникновения и классификация;
- отказоустойчивость, алгоритмы и пути ее достижения, метрики, использующиеся для оценки отказоустойчивости;
- восстановление систем после сбоев;
- общие методы повышения надежности.

Исследован вопрос отказоустойчивости распределенных систем, трудности, которые возникают при их использовании, а также получаемые выгоды. Отдельно изучена проблема достижения консенсуса в распределенных системах. С целью усвоения методов достижения отказоустойчивости при решении проблемы достижения консенсуса рассмотрен алгоритм Raft. В итоге, на основе изученного материала, предложены модификации исходного алгоритма Raft, призванные улучшить его доступность на операциях чтения.

Дополнительно, выполнен анализ различных архитектур информационных систем, выявлены их достоинства и недостатки. На его основе был сделан вывод об эффективности сервисной архитектуры для реализации отказоустойчивого и высокодоступного приложения. Разработано веб-приложение с использованием сервисной архитектуры. В нем применены различные методы противодействия и маскирования сбоев, изученные в процессе подготовки данной работы.

Полученные результаты обладают практической и научной ценностью. Они могут быть применены при проектировании и разработки современных масштабируемых и высоконагруженных систем. Дальнейшим развитием

данной работы может заключаться в создании нового алгоритма достижения распределенного консенсуса, нацеленного на работу с большими массивами данных и разработка на его основе распределенной базы данных.

Список используемых источников

1. Stonebraker M., Çetintemel U. ‘One Size Fits All’: An Idea Whose Time Has Come and Gone // 21st International Conference on Data Engineering (ICDE), URL: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.68.9136&rep=rep1&type=pdf>. (дата обращения: 13.11.2018).
2. Kopetz, H. and Verissimo, P.: Real Time and Dependability Concepts. In Mullender, S, с. 411-446. Wokingham: Addison-Wesley, 2nd ed., 1993.
3. Maarten van Steen, Andrew S. Tanenbaum. Distributed Systems — Published by Maarten van Steen, 2017 – 582 с. ISBN 978-15-43-0573-8-6
4. Pease, M., Shostak, R., and Lamport, L.: Reaching Agreement in the Presence of Faults. J. ACM, том 27, №. 2, с. 228-234, Апрель. 1980.
5. Yuan D., Luo Y., Zhuang X., et al. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems // 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI), URL: <https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-yuan.pdf>. (дата обращения: 10.04.2019).
6. Ford D., Labelle F., Popovici F. I., et al. Availability in Globally Distributed Storage Systems // 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI), URL: <https://static.googleusercontent.com/media/research.google.com/ru//pubs/archive/36737.pdf>. (дата обращения: 04.08.2019).
7. Gunawi H. S., Hao M., Leesatapornwongsa T., et al. What Bugs Live in the Cloud? // 5th ACM Symposium on Cloud Computing (SoCC),. — URL: <http://ucare.cs.uchicago.edu/pdf/socc14-cbs.pdf>. (дата обращения: 14.10.2019).
8. Oppenheimer D., Ganapathi A., Patterson D. A. Why Do Internet Services Fail, and What Can Be Done About It? // 4th USENIX Symposium on Internet Technologies and Systems (USITS),

URL:http://static.usenix.org/legacy/events/usits03/tech/full_papers/oppenheimer/oppenheimer.pdf. (дата обращения: 24.11.2019).

9. Guerraoui, R. and Schiper, A.: «Software-Based Replication for Fault Tolerance.» IEEE Computer, том. 30, №. 4, с. 68-74, Апрель. 1997.

10. Kreps J. Getting Real About Distributed System Reliability. URL: <http://blog.empathybox.com/post/19574936361/getting-real-about-distributed-system-reliability>. (дата обращения: 25.11.2019).

11. Клеппман М., К48 Высоконагруженные приложения. Программирование, масштабирование, поддержка. — СПб.: Питер, 2019. — 640 с. ISBN 978-5-4461-0512-0

12. Cook J. D. ACID Versus BASE for Database Transactions. URL: <https://www.johndcook.com/blog/2009/07/06/brewer-cap-theorem-base/>. (дата обращения: 04.05.2019).

13. Ongaro D., Ousterhout J. K. In Search of an Understandable Consensus Algorithm (Extended Version) // USENIX Annual Technical Conference (ATC). — URL: <https://pdfs.semanticscholar.org/b02c/6b00bd5dbdbd951fddb00b906c82fa80f0b3.pdf>. (дата обращения: 15.02.2019).

14. Fischer M.J., Lynch N. A., Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. JACM, 1985 URL: <http://dl.acm.org/citation.cfm?id=214121> (дата обращения: 14.04.2019).

15. Singhal, M. and Shivaratri, N.: Advanced Concepts in Operating Systems: Distributed, Database, and Multiprocessor Operating Systems. New York: McGraw-Hill, 1994.

16. Форд Нил, Парсонс Ребекка, Куа Патрик, Ф79 Эволюционная архитектура. Поддержка непрерывных изменений. — СПб.: Питер, 2019 — 272с ISBN 978-5-4461-0995-1

17. Бетси Бейер, Крис Джоунс, Дженнифер Петофф, Нейл Ричард Мерфи, С14 Site Reliability Engineering. Надежность и безотказность как в Google. — СПб.: Питер, 2019. — 592 с. —ISBN 978-5-4461-0976-0

18. Whittaker J. A., Arbon J., Carollo J. How Goggle Tests Software: Addison-Wesley, 2012
19. Algirdas Avizenis, The N-Version Approach to Fault-Tolerant Software, IEEE Computer, том. 30, №. 4, с.51-58, 1997
20. Мартин Р., М29 Чистая архитектура. Искусство разработки программного обеспечения. — СПб.: Питер, 2019. — 352 с. ISBN 978-5-4461-0772-8
21. Martin Fowler. MonolithFirst. URL: MonolithFirst.html (дата обращения: 08.05.2016).
22. Martin Fowler. Plugin. URL: <https://www.martinfowler.com/eaCatalog/plugin.html> (дата обращения: 11.02.2020).
23. Alessandro Nadalin. On monoliths, service-oriented architectures and microservices. URL: <http://odino.org/on-monolithsservice-oriented-architectures-and-microservices/> (дата обращения: 13.03.2020).
24. Thomas Erl. Service-Oriented Architecture (SOA): Concepts, Technology, and Design. Prentice Hall, 2005. 792 с. ISBN: 978-0131858589.
25. Ричардсон Крис, Р56 Микросервисы. Паттерны разработки и рефакторинга. — СПб.: Питер, 2020. — 544 с. ISBN 978-5-4461-0996-8
26. Моуэт Э., М89 Использование Docker / пер. с англ. А. В. Снастина; науч. ред. А. А. Маркелов. — М.: ДМК Пресс, 2017. — 354 с. ISBN 978-5-97060-426-7

Приложение А

Листинг сервиса “Каталог”

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using CatalogService.Models.Entity;
using CatalogService.Models.Repository;
using Microsoft.Extensions.Options;

namespace CatalogService.Controllers
{
    public class CatalogController : Controller
    {
        private IItemRepository repo;
        private CatalogSettings settings;

        public CatalogController(IItemRepository repository, IOp
tionsSnapshot<CatalogSettings> opt)
        {
            repo = repository;
            settings = opt.Value;
        }

        [HttpGet]
        [Route("products")]
        public async Task<IActionResult> ItemsAsync([FromQuery]i
nt pageSize = 10, [FromQuery]int pageIndex = 0, string ids = null)
        {
            if (!string.IsNullOrEmpty(ids))
            {
                var items = await GetItemsByIdsAsync(ids);

                if (!items.Any())
                {
                    return BadRequest("invalid ids [id,id,id,...
]");
                }

                ChangeUriPlaceholder(items);
                return Json(items);
            }

            var randomItems = await repo.GetItemsWithOptionsAsyn
c(pageSize * pageIndex, pageSize);
```

```

        var count = await repo.GetItemsCount();
        var model = new PaginatedItemsViewModel(pageIndex, pageSize, count, randomItems);
        ChangeUriPlaceholder(randomItems);

        return Json(model);
    }

    private async Task<IEnumerable<Item>> GetItemsByIdsAsync([FromQuery]string ids)
    {
        var numIds = ids.Split(',').Select(id => (Ok: Guid.TryParse(id, out Guid x), Value: x));

        if (!numIds.All(nid => nid.Ok))
        {
            return new List<Item>();
        }

        var idsToSelect = numIds
            .Select(id => id.Value);

        var items = await repo.GetByIdsAsync(idsToSelect);
        ChangeUriPlaceholder(items);

        return items;
    }

    private async Task<IEnumerable<Item>> GetRandoItemsAsync(int count = 10)
    {
        return await repo.GetRandomItemsAsync(count);
    }

    [HttpGet]
    [Route("items/type/{catalogTypeId}/brand/{catalogBrandId:Guid?}")]

    public async Task<ActionResult> ItemsByIdAndBrandIdAsync(Guid catalogTypeId, Guid? catalogBrandId, [FromQuery]int pageSize = 10, [FromQuery]int pageIndex = 0)
    {
        var items = await repo.GetByTypeIdAndTypeAsyncWithOpt(catalogTypeId, catalogBrandId, pageSize * pageIndex, pageSize);

        var totalItems = items.Count();
        ChangeUriPlaceholder(items);

        return Json(new PaginatedItemsViewModel(pageIndex, pageSize, totalItems, items));
    }

```

```

        [HttpGet]
        [Route("product/types")]
        public async Task<ActionResult<List<Models.Entity.Type>>
> CatalogTypesAsync()
        {
            return await repo.GetTypesAsync();
        }

        [HttpGet]
        [Route("product/brands")]
        public async Task<ActionResult<List<Brand>>> CatalogBrandsAsync()
        {
            return await repo.GetBrandsAsync();
        }

        private void ChangeUriPlaceholder(IEnumerable<Item> items)
        {
            var baseUrl = settings.ImageUrl;

            foreach (var item in items)
            {
                FillImages(item, baseUrl);
            }
        }

        private void FillImages(Item item, string picBaseUrl)
        {
            if (item != null)
            {
                item.ImageUri = picBaseUrl.Replace("[0]", item.Id.ToString());
            }
        }
    }
}
namespace CatalogService.Controllers
{
    public class ImageController : Controller
    {
        [HttpGet]
        public async Task<IActionResult> Get(Guid imageId)
        {
            var random = new Random();
            var number = random.Next(1, 12);
            var avatarPath = Path.Combine( ".", "Images", $"{number}.png");

```

```

        var stream = new FileStream/avatarPath, FileMode.Ope
n);
        return new FileStreamResult(stream, "image/png");
    }
}
namespace CatalogService.Exceptions
{
    public class DomainException : Exception
    {
        public DomainException()
        { }

        public DomainException(string message)
            : base(message)
        { }

        public DomainException(string message, Exception
innerException)
            : base(message, innerException)
        { }
    }
}
namespace CatalogService.Models.Repository
{
    public interface IItemRepository
    {
        void Save(Item item);
        Task SaveAsync(Item item);

        Item Get(Guid ItemID);
        Task<Item> GetAsync(Guid ItemID);

        Task<IEnumerable<Item>> GetByIdsAsync(IEnumerable<Guid>
ItemID);
        Task<IEnumerable<Item>> GetRandomItemsAsync(int count);

        Task<IEnumerable<Item>> GetItemsWithOptionsAsync(int
skip, int take);
        Task<int> GetItemsCount();
        Task<List<Brand>> GetBrandsAsync();
        Task<List<Entity.Type>> GetTypesAsync();

        Task<IEnumerable<Item>>
GetByTypeIdAndTypeAsyncWithOpt(Guid typeId, Guid? BrandID, int
skip, int take);
    }
}

namespace CatalogService.Models.Entity

```

```

{
    public class Brand
    {
        public Guid ID { get; set; }
        public string Name { get; set; }
    }
}

namespace CatalogService.Models.Entity
{
    public class PaginatedItemsViewModel
    {
        public int PageIndex { get; private set; }

        public int PageSize { get; private set; }

        public long Count { get; private set; }

        public IEnumerable<Item> Data { get; private set; }

        public PaginatedItemsViewModel(int pageIndex, int
pageSize, long count, IEnumerable<Item> data)
        {
            this.PageIndex = pageIndex;
            this.PageSize = pageSize;
            this.Count = count;
            this.Data = data;
        }
    }
}

namespace CatalogService.Models.Entity
{
    public class Item
    {
        public Guid ID { get; set; }

        public string Name { get; set; }

        public string Description { get; set; }

        public decimal Price { get; set; }

        public string PictureFileName { get; set; }

        public string ImageUri { get; set; }

        public Type Type { get; set; }

        public Brand Brand { get; set; }

        public UInt64 Quantity { get; set; }
    }
}

```

```

    /// <summary>
    /// Нужно ли заказать?
    /// </summary>
    public bool OnReorder { get; set; }

    public Item() { }

    public UInt64 RemoveStock(UInt64 quantityDesired)
    {
        if (Quantity == 0)
        {
            throw new Exceptions.DomainException($"Empty
stock, product item {Name} is sold out");
        }

        if (quantityDesired <= 0)
        {
            throw new Exceptions.DomainException($"Item
units desired should be greater than zero");
        }

        UInt64 removed = Math.Min(quantityDesired,
Quantity);

        this.Quantity -= removed;

        return removed;
    }

    /// <summary>
    /// Увеличиваем количество товара.ы
    /// </summary>
    public UInt64 AddStock(UInt64 quantity)
    {
        this.Quantity += quantity;
        this.OnReorder = false;

        return quantity;
    }
}

namespace CatalogService.Models.Entity
{
    public class Type
    {
        public Guid ID { get; set; }
        public string Name { get; set; }
    }
}

```

```

namespace CatalogService.Controllers
{
    public class CatalogSettings
    {
        public string ImageUrl { get; set; }
    }
}

FROM mcr.microsoft.com/dotnet/core/aspnet:2.2 AS base
EXPOSE 80
EXPOSE 443

FROM mcr.microsoft.com/dotnet/core/sdk:2.2 AS build
COPY . .
WORKDIR /CatalogService
COPY ./CatalogService/Images /app/Images
RUN dotnet publish -c Release -o /app

FROM base AS final
WORKDIR /app
COPY --from=build /app .
ENTRYPOINT ["dotnet", "CatalogService.dll", "--urls",
"http://0.0.0.0:5002/"]

```

Приложение В

Листинг сервиса “Учетные записи”

```
package communication

// Credentials содержит данные для аутентификации пользователя
type Credentials struct {
    Password string `json:"password"`
    Username string `json:"username"`
}

package communication

import (
    "log"
    "net/http"
    "os"

    "services.acount.v1/repo"
)

var userStore repo.UserStore
var address string

func init() {
    address = os.Getenv("ADDRESS")
}

func Start(us repo.UserStore) {

    userStore = us

    http.HandleFunc("/signin", signin)
    //http.HandleFunc("/welcome", welcome)
    http.HandleFunc("/refresh", refresh)

    bindAddress := ":8000"
    if len(address) != 0 {
        bindAddress = address
    }
    log.Fatal(http.ListenAndServe(bindAddress, nil))
}

package communication

import (
    "net/http"
```

```

    "services.acount.v1/jwt"
)

func refresh(w http.ResponseWriter, r *http.Request) {

    // Получаем токен
    c, err := r.Cookie("token")
    if err != nil {
        if err == http.ErrNoCookie {
            w.WriteHeader(http.StatusUnauthorized)
            return
        }
        w.WriteHeader(http.StatusBadRequest)
        return
    }

    expirationTime, token, err := jwt.Refresh(c.Value)
    if err != nil {
        w.WriteHeader(http.StatusUnauthorized)
    }

    // Set the new token as the users `session_token` cookie
    http.SetCookie(w, &http.Cookie{
        Name:     "session_token",
        Value:     token,
        Expires:   expirationTime,
    })
}

package communication

import (
    "encoding/json"
    "log"
    "net/http"

    "services.acount.v1/jwt"
)

// signin авторизует клиента
func signin(rs http.ResponseWriter, rq *http.Request) {

    var creds Credentials

    // Получаем данные для ауторизации
    err := json.NewDecoder(rq.Body).Decode(&creds)
    if err != nil {
        log.Println(err)
        // Не подходящий запрос
        rs.WriteHeader(http.StatusBadRequest)
        return
    }

```

```

    }

    u, err := userStore.Get(creds.Password)
    if err != nil {
        rs.WriteHeader(http.StatusUnauthorized)
        return
    }

    if u.Name != creds.Username {
        rs.WriteHeader(http.StatusUnauthorized)
        return
    }

    expire, token, err := jwt.Create(u)
    if err != nil {
        rs.WriteHeader(http.StatusUnauthorized)
        return
    }

    // Сохраняем токен в куках
    http.SetCookie(rs, &http.Cookie{
        Name:     "session_token",
        Value:    token,
        Expires:  expire,
    })
}
package entity

type Role string

var (
    SimpleUser Role = "User"
    Admin      Role = "Admin"
)

type User struct {
    Name      string
    Role      Role
    Password  string
}

package jwt

import (
    "errors"
    "os"
    "time"

    "github.com/dgrijalva/jwt-go"

    "services.acount.v1/entity"

```

```

)

// Данные хранимые в
type Claims struct {
    Username string `json:"username"`
    Role      string `json:"role"`
    jwt.StandardClaims
}

var (
    InvalidTokenErr          = errors.New("invalid token")
    NotExpiredInvalidTokenErr = errors.New("token is not expired")
)
var (
    jwtSecret []byte
)

func init() {
    jk := os.Getenv("JWT_KEY")

    if len(jk) == 0 {
        panic("jwt key must not be empty")
    }

    jwtSecret = []byte(jk)
}

func Create(u *entity.User) (time.Time, string, error) {
    expirationTime := time.Now().Add(5 * time.Minute)
    // Создаем наполнение токена
    claims := &Claims{
        Username: u.Name,
        Role:      string(u.Role),
        StandardClaims: jwt.StandardClaims{
            // Время протухание токена
            ExpiresAt: expirationTime.Unix(),
        },
    }

    // Создаем токен
    token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)
    tokenString, err := token.SignedString(jwtSecret)
    if err != nil {
        return expirationTime, "", err
    }

    return expirationTime, tokenString, nil
}

func validate(token string) (*Claims, error) {

```

```

    claims := &Claims{}
    tkn, err := jwt.ParseWithClaims(token, claims, func(token
*jwt.Token) (interface{}, error) {
        return jwtSecret, nil
    })

    if err != nil {
        if err == jwt.ErrSignatureInvalid {
            return claims, InvalidTokenErr
        }
        return claims, err
    }
    if !tkn.Valid {
        return claims, InvalidTokenErr
    }

    return claims, nil
}
func Refresh(token string) (time.Time, string, error) {
    expirationTime := time.Now().Add(5 * time.Minute)
    // проверяем токен
    claims, err := validate(token)
    if err != nil {
        return expirationTime, "", err
    }

    // Проверяем вышло ли время на самом деле
    if time.Unix(claims.ExpiresAt, 0).Sub(time.Now()) >
30*time.Second {
        return expirationTime, "", NotExpiredInvalidTokenErr
    }

    // Обновляем токен
    claims.ExpiresAt = expirationTime.Unix()

    t := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)
    tokenString, err := t.SignedString(jwtSecret)
    if err != nil {
        return expirationTime, "", err
    }

    return expirationTime, tokenString, nil
}
package memoryStore

import (
    "services.acount.v1/entity"
    "services.acount.v1/repo"
)

type User struct {

```

```

    Users []entity.User
}

func (ms *User) Save(u *entity.User) error {
    ms.Users = append(ms.Users, *u)

    return nil
}

func (ms *User) Get(password string) (*entity.User, error) {
    for _, u := range ms.Users {
        if u.Password == password {
            return &u, nil
        }
    }

    return nil, repo.NoUserErr
}

package repo

import (
    "errors"

    "services.acount.v1/entity"
)

var (
    NoUserErr = errors.New("there is no such a user")
)

type UserStore interface {
    Save(u *entity.User) error
    Get(password string) (*entity.User, error)
}

FROM golang:1.13 as build

ENV GO111MODULE=on
WORKDIR /app
COPY . .
RUN go build -o /app/bin/auth

FROM build as final
WORKDIR /app/bin
ENTRYPOINT ["/auth"]

module services.acount.v1

go 1.13

```

```
require github.com/dgrijalva/jwt-go v3.2.0+incompatible
package main

import (
    "services.acount.v1/communication"
    "services.acount.v1/entity"
    "services.acount.v1/memoryStore"
)

func main() {
    var us memoryStore.User
    communication.Start(&us)
}
```

Приложение С

Листинг сервиса “API шлюз ”

```
namespace Gateway.API
{
    public class AuthAPI
    {
        public static string Login(string baseUri)
        {
            return $"{baseUri}signin";
        }

        public static string Refresh(string baseUri)
        {
            return $"{baseUri}refresh";
        }
    }
}
namespace Gateway.API
{
    public class CatalogAPI
    {
        public static string GetAllCatalogItems(string baseUri,
int page, int take, Guid? brand, Guid? type)
        {
            var filterQs = "";

            if (type.HasValue)
            {
                var brandQs = (brand.HasValue) ?
brand.Value.ToString() : string.Empty;
                filterQs =
$"/type/{type.Value}/brand/{brandQs}";
            }
            else if (brand.HasValue)
            {
                var brandQs = (brand.HasValue) ?
brand.Value.ToString() : string.Empty;
                filterQs = $"/type/all/brand/{brandQs}";
            }
            else
            {
                filterQs = string.Empty;
            }
        }
    }
}
```

```

        return
        $"{baseUri}items{filterQs}?pageIndex={page}&pageSize={take}";
    }

    public static string GetAllBrands(string baseUri)
    {
        return $"{baseUri}catalogBrands";
    }

    public static string GetAllTypes(string baseUri)
    {
        return $"{baseUri}catalogTypes";
    }
}
namespace Gateway.API
{
    public class OrderAPI
    {
        public static string GetAllOrders(string baseUri, int
page, int take)
        {
            return
            $"{baseUri}orders?pageIndex={page}&pageSize={take}";
        }
    }
}
namespace Gateway.Controllers
{
    public class AccountController : Controller
    {
        private IAuthService _authSvc;
        private object _appSettings;

        public AccountController(IAuthService authSvc)
        {
            _authSvc = authSvc;
        }

        public async Task<IActionResult> Index()
        {
            return View();
        }

        public async Task<IActionResult> SignIn(string username,
string password)
        {
            string token = null;
            try
            {

```

```

        token = await _authSvc.Login(username,
password);

HttpContext.Response.Cookies.Append("access_token",
RecreateToken(token));
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex);
        ViewBag["Error"] = "Не удалось аутентифицировать
пользователя";
    }

    return
RedirectToAction(nameof(CatalogController.Index), "Catalog");
}

private string RecreateToken(string originalToken)
{
    var person = ParseToken(originalToken);
    var identity = GetIdentity(person.Username,
person.Role);

    var now = DateTime.UtcNow;
    // создаем JWT-токен
    var jwt = new JwtSecurityToken(
        issuer: AuthOptions.ISSUER,
        audience: AuthOptions.AUDIENCE,
        notBefore: now,
        claims: identity.Claims,
        expires:
now.Add(TimeSpan.FromMinutes(AuthOptions.LIFETIME)),
        signingCredentials: new
SigningCredentials(AuthOptions.GetSymmetricSecurityKey(),
SecurityAlgorithms.HmacSha256));
    var encodedJwt = new
JwtSecurityTokenHandler().WriteToken(jwt);

    return encodedJwt;
}

private Person ParseToken(string token)
{
    var handler = new JwtSecurityTokenHandler();
    var jsonToken = handler.ReadToken(token);
    var tokenS = handler.ReadToken(token) as
JwtSecurityToken;
    var person = new Person();

    person.Username = tokenS.Claims.First(claim =>
claim.Type == "username").Value;

```

```

        person.Role = tokenS.Claims.First(claim =>
claim.Type == "role").Value;

        return person;
    }

    private ClaimsIdentity GetIdentity(string username,
string role)
    {
        var claims = new List<Claim>
        {
            new Claim(ClaimsIdentity.DefaultNameClaimType,
username),
            new Claim(ClaimsIdentity.DefaultRoleClaimType,
role )
        };
        ClaimsIdentity claimsIdentity =
new ClaimsIdentity(claims, "Token",
ClaimsIdentity.DefaultNameClaimType,
ClaimsIdentity.DefaultRoleClaimType);
        return claimsIdentity;
    }

    public async Task<IActionResult> Signout()
    {
        await
HttpContext.SignOutAsync(CookieAuthenticationDefaults.Authenticati
onScheme);
        await
HttpContext.SignOutAsync(OpenIdConnectDefaults.AuthenticationSchem
e);

        var homeUrl =
Url.Action(nameof(CatalogController.Index), "Catalog");

        return
RedirectToAction(nameof(CatalogController.Index), "Catalog");
    }
}

namespace Gateway.Controllers
{
    public class CatalogController : Controller
    {
        const int itemsPage = 10;

        private ICatalogService _catalogSvc;

        public CatalogController(ICatalogService catalogSvc) =>
        _catalogSvc = catalogSvc;
    }
}

```

```

        [Authorize(Roles = "Admin,User")]
        public async Task<IActionResult> Index(Guid?
BrandFilterApplied, Guid? TypesFilterApplied, int? page,
[FromQuery]string errorMsg)
        {
            var catalog = await GetCatalog(BrandFilterApplied,
TypesFilterApplied, page);
            var vm = new IndexViewModel()
            {
                CatalogItems = catalog.Data,
                Brands = await _catalogSvc.GetBrands(),
                Types = await _catalogSvc.GetTypes(),
                BrandFilterApplied = BrandFilterApplied,
                TypesFilterApplied = TypesFilterApplied,
                PaginationInfo = new PaginationInfo()
                {
                    ActualPage = page ?? 0,
                    ItemsPerPage = catalog.Data.Count,
                    TotalItems = catalog.Count,
                    TotalPages =
(int)Math.Ceiling(((decimal)catalog.Count / itemsPage))
                }
            };

            vm.PaginationInfo.Next =
(vm.PaginationInfo.ActualPage == vm.PaginationInfo.TotalPages - 1)
? "is-disabled" : "";
            vm.PaginationInfo.Previous =
(vm.PaginationInfo.ActualPage == 0) ? "is-disabled" : "";

            ViewBag.BasketInoperativeMsg = errorMsg;

            return View(vm);
        }

        private async Task<Catalog> GetCatalog(Guid?
BrandFilterApplied, Guid? TypesFilterApplied, int? page)
        {
            try
            {
                return await _catalogSvc.GetCatalogItems(page ??
0, itemsPage, BrandFilterApplied, TypesFilterApplied);
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex);
                return new Catalog()
                {
                    Count = 0,
                    Data = new List<CatalogItem>(),

```

```

        };
    }
}

namespace Gateway.Controllers
{
    public class OrderController : Controller
    {
        const int itemsPage = 10;

        private IOrderService _ordergSvc;

        public OrderController(IOrderService ordergSvc) =>
            _ordergSvc = ordergSvc;

        [Authorize(Roles = "Admin")]
        [HttpGet]
        [Route("orders/{page:int?}")]
        public async Task<IActionResult> GetAll(int? page)
        {
            var orders = await GetAllUsersOrders(page);
            var vm = new OrderViewModel()
            {
                Orders = orders.Data,
                PaginationInfo = new PaginationInfo()
                {
                    ActualPage = page ?? 0,
                    ItemsPerPage = orders.Data.Count,
                    TotalItems = orders.Count,
                    TotalPages =
                        (int)Math.Ceiling(((decimal)orders.Count / itemsPage))
                }
            };

            vm.PaginationInfo.Next =
                (vm.PaginationInfo.ActualPage == vm.PaginationInfo.TotalPages - 1)
                ? "is-disabled" : "";
            vm.PaginationInfo.Previous =
                (vm.PaginationInfo.ActualPage == 0) ? "is-disabled" : "";

            return View(vm);
        }

        private async Task<PaginatedOrdersViewModel>
        GetAllUsersOrders(int? page)
        {

```

```

        try
        {
            return await _ordergSvc.GetAllOrdersAsync(page
?? 0, itemsPage);
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex);
            return new PaginatedOrdersViewModel()
            {
                Count = 0,
                Data = new List<Order>(),
            };
        }
    }

}

namespace Gateway.Infrostructure
{
    public class AuthOptions
    {
        public const int LIFETIME = 1; // время жизни токена - 1
минута

        public static string KEY = ""; // ключ для шифрации
        public static SymmetricSecurityKey
GetSymmetricSecurityKey()
        {
            return
new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(KEY));
        }
    }
}

namespace Gateway.Model
{
    public class Person
    {
        public string Username { get; set; }
        public string Role { get; set; }
    }
}

namespace Gateway.Services
{
    public class AuthService : IAuthService
    {
        private readonly IOptions<AppSettings> _settings;
        private readonly HttpClient _httpClient;

```

```

        private readonly string _remoteServiceBaseUrl;

        public AuthService(HttpClient httpClient,
            IOptions<AppSettings> settings)
        {
            _httpClient = httpClient;
            _settings = settings;
            _remoteServiceBaseUrl =
                $"({_settings.Value.AuthUrl})/";
        }

        public async Task<string> Login(string username, string
            password)
        {
            var user = new UserCreds()
            {
                Username = username,
                Password = password
            };
            var json = JsonConvert.SerializeObject(user);
            var data = new StringContent(json, Encoding.UTF8,
                "application/json");

            var uri = API.AuthAPI.Login(_remoteServiceBaseUrl);

            Console.WriteLine(uri);
            var r = await _httpClient.PostAsync(uri, data);
            IEnumerable<string> auth;
            r.Headers.TryGetValues("Set-Cookie", out auth);
            var token =
                auth.First().Substring("session_token=".Length);
            return token.Substring(0, token.IndexOf(';'));
        }

        public async Task<string> Refresh(string token)
        {
            var uri =
                API.AuthAPI.Refresh(_remoteServiceBaseUrl);
            return await _httpClient.GetStringAsync(uri);
        }

        public Task<bool> Verify(string token)
        {
            var task = new Task<bool>(() => true);
            task.RunSynchronously();
            return task;
        }
    }
}

```

```

namespace Gateway.Services
{
    public class CatalogService : ICatalogService
    {
        private readonly IOptions<AppSettings> _settings;
        private readonly HttpClient _httpClient;

        private readonly string _remoteServiceBaseUrl;

        public CatalogService(HttpClient httpClient,
            IOptions<AppSettings> settings)
        {
            _httpClient = httpClient;
            _settings = settings;

            _remoteServiceBaseUrl =
                $"({_settings.Value.CatalogUrl})/";
        }

        public async Task<Catalog> GetCatalogItems(int page, int
            take, Guid? brand, Guid? type)
        {
            var uri =
                API.CatalogAPI.GetAllCatalogItems(_remoteServiceBaseUrl, page,
                take, brand, type);

            var responseString = await
                _httpClient.GetStringAsync(uri);

            return
                JsonConvert.DeserializeObject<Catalog>(responseString);
        }

        public async Task<IEnumerable<SelectListItem>>
            GetBrands()
        {
            var uri =
                API.CatalogAPI.GetAllBrands(_remoteServiceBaseUrl);

            var responseString = await
                _httpClient.GetStringAsync(uri);

            var items = new List<SelectListItem>();

            items.Add(new SelectListItem() { Value = null, Text
                = "Bce", Selected = true });

            var brands = JArray.Parse(responseString);

            foreach (var brand in brands.Children<JObject>())
            {

```

```

        items.Add(new SelectListItem()
        {
            Value = brand.Value<string>("id"),
            Text = brand.Value<string>("name")
        });
    }

    return items;
}

public async Task<IEnumerable<SelectListItem>>
GetTypes()
{
    var uri =
API.CatalogAPI.GetAllTypes(_remoteServiceBaseUrl);

    var responseString = await
_httpClient.GetStringAsync(uri);

    var items = new List<SelectListItem>();
    items.Add(new SelectListItem() { Value = null, Text
= "Bce", Selected = true });

    var types = JObject.Parse(responseString);
    foreach (var t in types.Children<JObject>())
    {
        items.Add(new SelectListItem()
        {
            Value = t.Value<string>("id"),
            Text = t.Value<string>("name")
        });
    }

    return items;
}
}

namespace Gateway.Services
{
    public interface IAuthService
    {
        Task<bool> Verify(string token);
        Task<string> Login(string username, string password);
        Task<string> Refresh(string token);
    }
}

namespace Gateway.Services
{
    public interface ICatalogService
    {

```

```

        Task<Catalog> GetCatalogItems(int page, int take, Guid?
brand, Guid? type);
        Task<IEnumerable<SelectListItem>> GetBrands();
        Task<IEnumerable<SelectListItem>> GetTypes();
    }
}
namespace Gateway.Services
{
    public interface IOrderService
    {
        Task<PaginatedOrdersViewModel> GetAllOrdersAsync(int
page, int take);
    }
}

namespace Gateway.Services
{
    public class OrderService : IOrderService
    {
        private readonly IOptions<AppSettings> _settings;
        private readonly HttpClient _httpClient;

        private readonly string _remoteServiceBaseUrl;

        public OrderService(HttpClient httpClient,
IOptions<AppSettings> settings)
        {
            _httpClient = httpClient;
            _settings = settings;

            _remoteServiceBaseUrl =
$" {_settings.Value.OrderUrl} / ";
        }

        public async Task<PaginatedOrdersViewModel>
GetAllOrdersAsync(int page, int take)
        {
            var uri =
API.OrderAPI.GetAllOrders(_remoteServiceBaseUrl, page, take);

            var responseString = await
_httpClient.GetStringAsync(uri);

            return
JsonConvert.DeserializeObject<PaginatedOrdersViewModel>(responseSt
ring);
        }
    }
}
namespace Gateway.ViewModels.CatalogViewModels
{

```

```

public class IndexViewModel
{
    public IEnumerable<CatalogItem> CatalogItems { get; set; }
}

    public IEnumerable<SelectListItem> Brands { get; set; }
    public IEnumerable<SelectListItem> Types { get; set; }
    public Guid? BrandFilterApplied { get; set; }
    public Guid? TypesFilterApplied { get; set; }
    public PaginationInfo PaginationInfo { get; set; }
}
}
namespace Gateway.ViewModels
{
    public class Catalog
    {
        public int PageIndex { get; set; }
        public int PageSize { get; set; }
        public int Count { get; set; }
        public List<CatalogItem> Data { get; set; }
    }
}

namespace Gateway.ViewModels
{
    public class CatalogItem
    {
        public Guid Id { get; set; }
        public string Name { get; set; }
        public string Description { get; set; }
        public decimal Price { get; set; }
        public string ImageUri { get; set; }

        public CatalogBrand Brand { get; set; }
        public CatalogType Type { get; set; }
    }

    public class CatalogBrand
    {
        public Guid CatalogBrandId { get; set; }
        public string Name { get; set; }
    }

    public class CatalogType
    {
        public Guid CatalogTypeId { get; set; }
        public string Name { get; set; }
    }
}

namespace Gateway.ViewModels.Order

```

```

{
    public class Orderitem
    {
        public Guid      ID { get; set; }
        public string    productname { get; set; }
        public int       units { get; set; }
        public double    unitprice { get; set; }
        public string    pictureurl { get; set; }
    }

    public class Order
    {
        public Guid ID{ get; set; }
        public Guid userID { get; set; }
        public DateTime date { get; set; }
        public string status { get; set; }
        public List<Orderitem> orderitems { get; set; }
    }

    public class OrderSummary
    {
        public Guid ID { get; set; }
        public DateTime date { get; set; }
        public string status { get; set; }
        public double total { get; set; }
    }
}
namespace Gateway.ViewModels.Order
{
    public class OrderViewModel
    {
        public IEnumerable<Order> Orders { get; set; }
        public PaginationInfo PaginationInfo { get; set; }
    }
}

namespace Gateway.ViewModels.Order
{
    public class PaginatedOrdersViewModel
    {
        public int PageIndex { get; set; }
        public int PageSize { get; set; }
        public int Count { get; set; }
        public List<Order> Data { get; set; }
    }
}
namespace Gateway.ViewModels.Pagination
{
    public class PaginationInfo
    {
        public int TotalItems { get; set; }

```

```

        public int ItemsPerPage { get; set; }
        public int ActualPage { get; set; }
        public int TotalPages { get; set; }
        public string Previous { get; set; }
        public string Next { get; set; }
    }
}
namespace Gateway.ViewModels
{
    public class UserCreds
    {
        public UserCreds()
        {}
        public string Username { get; set; }
        public string Password { get; set; }
    }
}
FROM mcr.microsoft.com/dotnet/core/aspnet:2.2 AS base
EXPOSE 80
EXPOSE 443

FROM mcr.microsoft.com/dotnet/core/sdk:2.2 AS build
COPY . .
WORKDIR /Gateway
RUN dotnet publish -c Release -o /app

FROM base AS final
WORKDIR /app
COPY --from=build /app .
ENTRYPOINT ["dotnet", "Gateway.dll", "--urls",
"http://0.0.0.0:5102/"]

```

Приложение D

Листинг сервиса “Заказы”

```
namespace OrderService.Controllers
{
    public class OrderController : Controller
    {
        private IOrderRepository repo;
        public OrderController(IOrderRepository repository)
        {
            repo = repository;
        }

        [HttpGet]
        [Route("order/{orderId:Guid?}")]
        public async Task<ActionResult> GetOrderAsync(Guid? orde
rId)
        {
            if (orderId == null)
            {
                return NotFound();
            }

            try
            {
                var order = await repo.GetOrderAsync((Guid) order
Id);
                return Ok(order);
            }
            catch
            {
                return NotFound();
            }
        }

        [HttpGet]
        [Route("orders/summary")]
        public async Task<ActionResult<IEnumerable<OrderSummary>
>> GetOrdersSummaryAsync([FromQuery]Guid? userID)
        {
            if (userID == null)
            {
                return NotFound();
            }
        }
    }
}
```

```

        var orders = await repo.GetOrdersByUserSummaryAsync(
(Guid)userID);
        return Ok(orders);
    }

    [HttpGet]
    [Route("orders")]
    public async Task<ActionResult<IEnumerable<Order>>> GetO
rdersAsync([FromQuery]Guid? userID)
    {

        if (userID == null)
        {
            return NotFound();
        }

        var orders = await repo.GetOrdersByUserAsync((Guid)u
serID);
        return Ok(orders);
    }

    [HttpGet]
    [Route("orders")]
    public async Task<IActionResult> GetOrdersAsync([FromQue
ry]int pageSize = 10, [FromQuery]int pageIndex = 0)
    {
        var orders = await repo.GetOrdersWithOptionsAsync(pa
geSize * pageIndex, pageSize);
        var count = await repo.GetOrdersCount();
        var model = new PaginatedOrdersViewModel(pageIndex,
pageSize, count, orders);

        return Json(model);
    }

    [HttpPut]
    [Route("order/{orderId:Guid?}")]
    public async Task<IActionResult> CreateOrder(Guid? order
ID, [FromQuery]Guid userID, [FromQuery] Guid[] catalogItems = null
)
    {
        if (orderId == null)
        {
            return NotFound();
        }

        var order = new Order()
        {
            ID = (Guid)orderId,
            userID = userID,
            date = DateTime.Now,

```

```

        status = "в работе",
    };

    await repo.Save(order);

    return Ok();
}

[HttpDelete]
[Route("order/{orderId:Guid?}")]
public async Task<IActionResult> DeleteOrder(Guid? order
ID)
{
    if (orderId == null)
    {
        return NotFound();
    }

    await repo.Delete((Guid)orderId);
    return Ok();
}
}

namespace OrderService.Models.Repository
{
    public interface IOrderRepository
    {
        Task<Order> GetOrderAsync(Guid ID);

        Task<IEnumerable<Order>> GetOrdersByUserAsync(Guid userI
D);

        Task<IEnumerable<OrderSummary>> GetOrdersByUserSummaryAs
ync(Guid userID);

        Task<IEnumerable<Order>> GetOrdersWithOptionsAsync(int s
kip, int take);
        Task<int> GetOrdersCount();

        Task Save(Order order);

        Task Delete(Guid orderId);
    }
}

namespace OrderService.Models.Repository
{
    public class OrderInMemoryRepository : IOrderRepository
    {
        private static Dictionary<Guid, Order> orders;

```

```

static OrderInMemoryRepository()
{
    orders = new Dictionary<Guid, Order>();
    var random = new Random();
    var items = new List<Orderitem>()
    {
        new Orderitem{ ID = Guid.NewGuid(), productname
= "Item - 1", pictureurl = "", unitprice = 105, units= 1},
        new Orderitem{ ID = Guid.NewGuid(), productname
= "Item - 2", pictureurl = "", unitprice = 105, units= 1},
        new Orderitem{ ID = Guid.NewGuid(), productname
= "Item - 3", pictureurl = "", unitprice = 105, units= 1},
    };
    for (int i = 0; i < 100; i++)
    {
        var ID = Guid.NewGuid();
        var order = new Order()
        {
            userID = Guid.NewGuid(),
            date = DateTime.Now,
            ID = ID,
            status = "В процесі",
            orderitems = items
        };

        orders.Add(ID, order);
    }
}

public Task Delete(Guid orderID)
{
    var task = new Task(() =>
    {
        lock (orders)
        {
            orders.Remove(orderID);
        }
    })
    ;

    task.RunSynchronously();
    return task;
}

public Task<Order> GetOrderAsync(Guid guid)
{
    var task = new Task<Order>(() =>
    {
        lock (orders)
        {
            return TryGetValue(guid);
        }
    })
    ;
}

```

```

        }
    }
};

task.RunSynchronously();
return task;
}

private Order TryGetValue(Guid orderId)
{
    Order item = null;
    lock (orders)
    {
        var result = orders.TryGetValue(orderID, out item);
        if (result)
        {
            return item;
        }
        throw new Exception($"Repository: there is no items
with ID: {orderId}");
    }
}

public Task<IEnumerable<Order>> GetOrdersByUserAsync(Guid userID)
{
    var task = new Task<IEnumerable<Order>>(() =>
    {
        lock (orders)
        {
            return orders.Values.Where<Order>((v) => { return v.userID == userID; });
        }
    });

    task.RunSynchronously();
    return task;
}

public Task<IEnumerable<OrderSummary>> GetOrdersByUserSummaryAsync(Guid userID)
{
    var task = new Task<IEnumerable<OrderSummary>>(() =>
    {
        lock (orders)
        {
            var userOrders = orders.Values.Where<Order>((v) => { return v.userID == userID; });
            return CreateSummary(userOrders);
        }
    });
}

```

```

        });

        task.RunSynchronously();
        return task;
    }

    private IEnumerable<OrderSummary> CreateSummary(IEnumerable<Order> orders)
    {
        var summary = new List<OrderSummary>();
        foreach (var order in orders)
        {
            var sum = new OrderSummary();
            sum.ID = order.ID;
            sum.date = order.date;
            sum.status = order.status;
            sum.total = CalculateTotal(order.orderitems);
        }

        return summary;
    }

    private double CalculateTotal(List<Orderitem> items)
    {
        double total = 0;
        foreach (var item in items)
        {
            total += item.unitprice * item.units;
        }
        return total;
    }

    public Task<int> GetOrdersCount()
    {
        var task = new Task<int>(() =>
        {
            lock (orders)
            {
                return orders.Count;
            }
        });

        task.RunSynchronously();
        return task;
    }

    public Task<IEnumerable<Order>> GetOrdersWithOptionsAsync(int skip, int take)
    {
        var task = new Task<IEnumerable<Order>>(() =>

```

```

        {
            lock (orders)
            {
                return orders.Values
                    .OrderBy(c => c.ID)
                    .Skip(skip)
                    .Take(take);
            }
        });

        task.RunSynchronously();
        return task;
    }

    public Task Save(Order order)
    {
        var task = new Task(() =>
        {
            lock (orders)
            {
                orders.Add(order.ID, order);
            }
        });

        task.RunSynchronously();
        return task;
    }
}

namespace OrderService.Models.ViewModel
{
    public class PaginatedOrdersViewModel
    {
        public int PageIndex { get; private set; }

        public int PageSize { get; private set; }

        public long Count { get; private set; }

        public IEnumerable<Order> Data { get; private set; }

        public PaginatedOrdersViewModel(int pageIndex, int
        pageSize, long count, IEnumerable<Order> data)
        {
            this.PageIndex = pageIndex;
            this.PageSize = pageSize;
            this.Count = count;
            this.Data = data;
        }
    }
}

```

```

}

namespace OrderService.Models
{
    public class Orderitem
    {
        public Guid ID { get; set; }
        public string productname { get; set; }
        public int units { get; set; }
        public double unitprice { get; set; }
        public string pictureurl { get; set; }
    }

    public class Order
    {
        public Guid ID { get; set; }
        public Guid userID { get; set; }
        public DateTime date { get; set; }
        public string status { get; set; }
        public List<Orderitem> orderitems { get; set; }
    }

    public class OrderSummary
    {
        public Guid ID { get; set; }
        public DateTime date { get; set; }
        public string status { get; set; }
        public double total { get; set; }
    }
}

namespace OrderService
{
    public class OrderSettings
    {
        public string CatalogUrl { get; set; }
    }
}

FROM mcr.microsoft.com/dotnet/core/aspnet:2.2 AS base
EXPOSE 80
EXPOSE 443

FROM mcr.microsoft.com/dotnet/core/sdk:2.2 AS build
COPY . .
WORKDIR /OrderService
RUN dotnet publish -c Release -o /app

FROM base AS final
WORKDIR /app

```

```
COPY --from=build /app .  
ENTRYPOINT ["dotnet", "OrderService.dll", "--urls",  
"http://0.0.0.0:5103/"]
```

Приложение Е

Листинг конфигурации Docker compose

```
version: '2.2'

services:
  catalog:
    environment:
      - ${JWT_KEY}
      - IMAGE_URL=http://localhost:5003/items/[0]/pic/
    image: catalog
    ports:
      - "5003:5002"
    build:
      context: ./CatalogService
      dockerfile: Dockerfile
    networks:
      p2p:
        ipv4_address: 172.28.5.1
  auth:
    image: auth
    environment:
      - JWT_KEY=${JWT_KEY}
      - ADDRESS=172.28.5.2:8000
    build:
      context: ./authservice
      dockerfile: Dockerfile
    networks:
      p2p:
        ipv4_address: 172.28.5.2
  gateway:
    image: gateway
    environment:
      - JWT_KEY=${JWT_KEY}
      - SERVICE_CATALOG=http://172.28.5.1:5002
      - SERVICE_AUTH=http://172.28.5.2:8000
      - SERVICE_ORDER=http://172.28.5.4:5003
    ports:
      - "5002:5102"
    build:
      context: ./Gateway
      dockerfile: Dockerfile
    networks:
      p2p:
        ipv4_address: 172.28.5.3
  order:
```

```
environment:
image: order
ports:
  - "5003:5002"
build:
  context: ./OrderService
  dockerfile: Dockerfile
networks:
  p2p:
    ipv4_address: 172.28.5.4
networks:
  p2p:
    driver: bridge
    ipam:
      config:
        - subnet: 172.28.0.0/16
          ip_range: 172.28.5.0/24
          gateway: 172.28.5.254
```

Приложение F

Презентация

Тема работы: «Система отказоустойчивого доступа»

Автор работы: студент группы 18ВВм1 Васильев Н.Н.
Научный руководитель: к.т.н., доцент Дорошенко И.Н

1

Цели работы

- Изучить современные способы обеспечения отказоустойчивости информационных систем.
- Улучшить существующие методы.
- Разработать отказоустойчивую системы с высоким уровнем доступности

2

Отказы и сбои



3

Причины возникновения сбоев



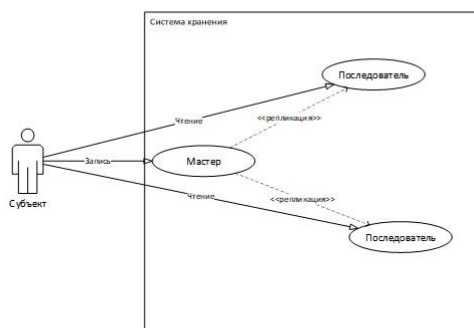
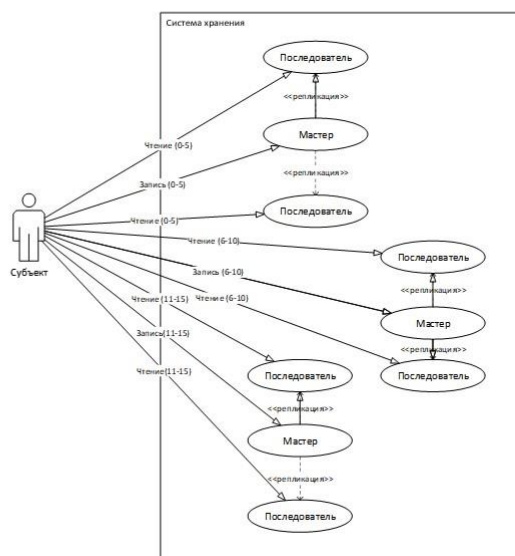
4

Метрики отказоустойчивости

- Вероятность отказа по требованию (POFOD)
- Частота возникновения сбоев (POCOF)
- Среднее время до сбоя (MTTF)
- Среднее время восстановления (MTTR)
- Среднее время между сбоями (MTBF)
- Доступность системы

5

Распределенные системы



6

Предотвращение сбоев

Мониторинг:

- Оповещения
- Черный и белый ящик

Тестирование:

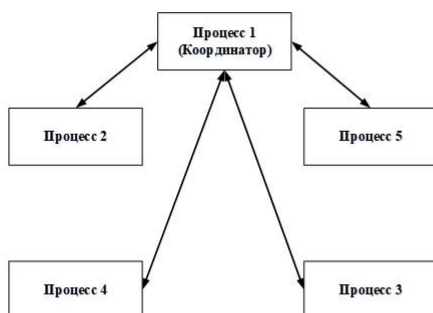
- Модульное
- Интеграционное
- Системное



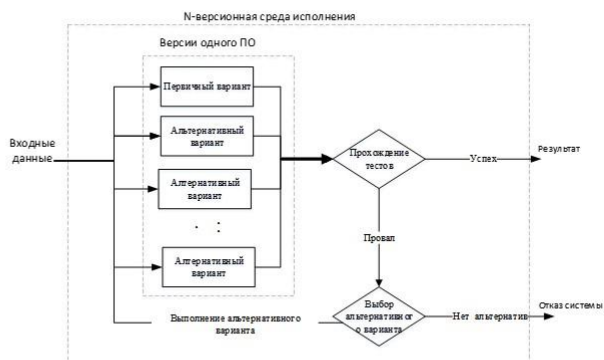
7

Маскирование сбоев

Группы процессов



N-версионное программирование



8

Восстановление

Точки восстановления

Журналирование

- Пессимистичное
- Оптимистичное

Консенсус

Типы:

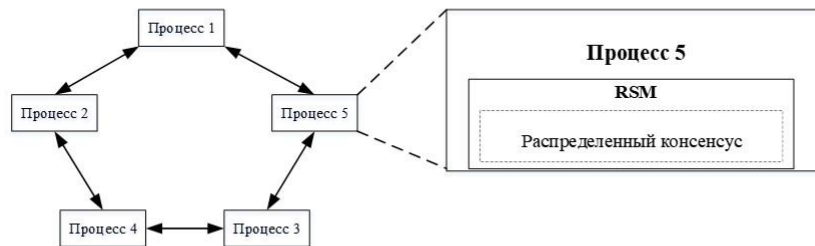
- Асинхронный распределенный
- Синхронный

Raft



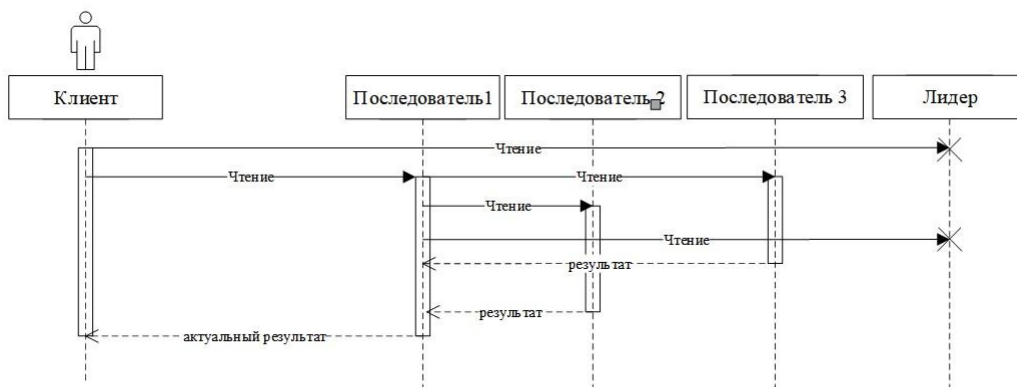
Машины с реплицированным состоянием

- Машина с реплицированным состоянием — это система, реализованная на логическом уровне, над алгоритмом достижения консенсуса.



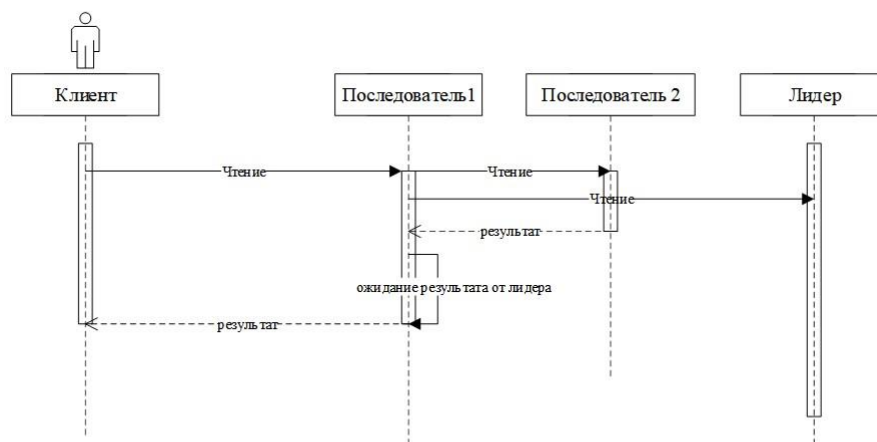
11

Модификация алгоритма Raft



12

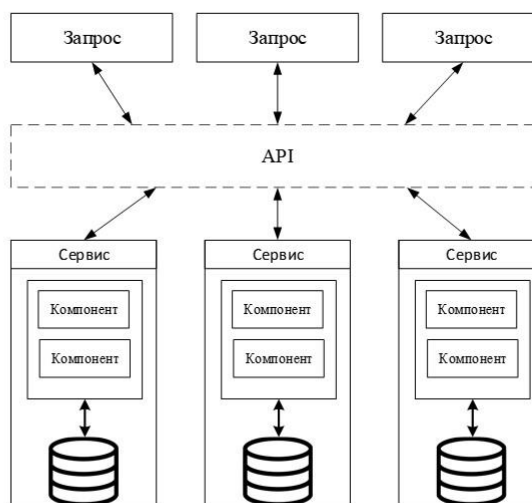
Модификация алгоритма Raft. Продолжение



13

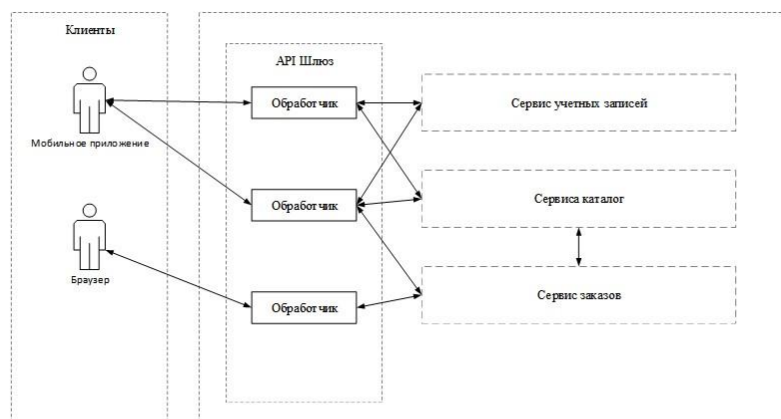
Выбор архитектуры отказоустойчивого приложения

- Монолитная
- Плагиная
- Сервисная



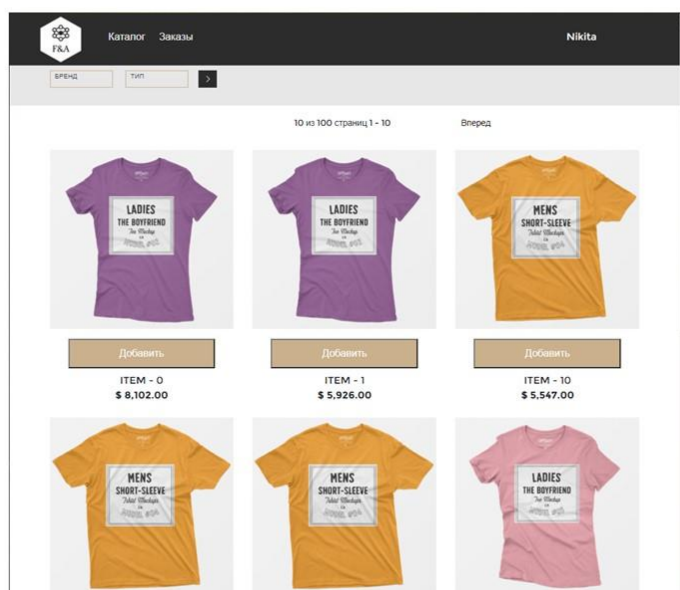
14

Архитектура



15

Демонстрация работы программы



16

Заключение

В итоге были получены следующие результаты:

- Изучены различные методы повышения надежности и обеспечения отказоустойчивости, основы распределенных систем;
- Предложены модификации алгоритма консенсуса Raft с целью увеличения показателя доступности.
- Разработано отказоустойчивое приложение с сервисной архитектурой.

17

Спасибо за внимание!

18