

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет
вычислительной техники

Кафедра
МОиПЭВМ

Направление подготовки 09.03.04 «Программная инженерия»

БАКАЛАВРСКАЯ РАБОТА

на тему

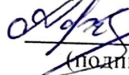
**Распределенная высоконагруженная система
хранения данных**

Студент

 21.06.21
(подпись, дата)

Чурсин Павел Алексеевич
(ФИО полностью)

Руководитель

 21.06.21
(подпись, дата)

Афонин А.Ю.
(фамилия, инициалы)

Нормоконтролёр

 22.06.21
(подпись, дата)

Такташкин Д. В.
(фамилия, инициалы)

Работа допущена к защите (протокол заседания кафедры от _____ № _____)

Заведующий кафедрой

(подпись)

Макарычев П. П.
(фамилия, инициалы)

Работа защищена с отметкой _____ (протокол заседания ГЭК от _____ № _____)

Секретарь ГЭК

(подпись)

Попова Н. А.
(фамилия, инициалы)

Пенза, 2021

Федеральное государственное бюджетное образовательное учреждение
высшего образования
ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет
Вычислительной техники

Кафедра
МО и ПЭВМ

«Утверждаю»
Заведующий кафедрой
П.П. Макарычев
«21» января 2021 г

**ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ
БАКАЛАВРА**

1. Студент Чурсин Павел Алексеевич гр. 17ВП1 факультета ВТ
<Фамилия, имя, отчество полностью>
2. Тема работы Распределенная высоконагруженная система хранения данных

Тема утверждена приказом ПГУ № 1007 от " 21 " января 2021

3. Руководитель работы к.т.н., доцент кафедры «МОиПЭВМ» Афонин А.Ю.

4. Задание на работу (назначение разработки, исходные данные и т.п.)

Назначение: разработать программные средства для отказоустойчивого, согласованного и высокопроизводительного хранения данных

Среда выполнения: операционная система Windows, архитектура x86-64

Среда разработки: операционная система Windows

Языковые средства разработки: C, python

Инструментальные средства разработки: Microsoft Visual Studio 2019, Draw.io, VS Code

Технологии разработки: Чистая архитектура

Основные функции:

а) хранение данных

б) обеспечение отказоустойчивости

в) обеспечение согласованности данных

г) высокая производительность

5. Перечень подлежащих разработке вопросов

1) Анализ предметной области и постановка задачи

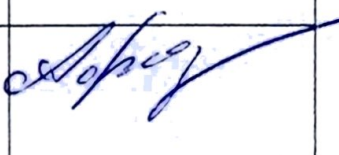
2) Проектирование и реализация

3) Формализация системы

4) Тестирование

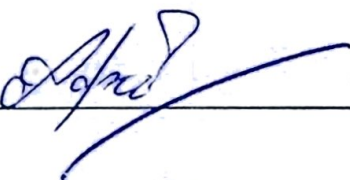
5) Оформление пояснительной записки

6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1	Анализ предметной области постановка задачи	15%	14.04.2021– 21.04.2021	
2	Проектирование и реализация	40%	22.04.2021– 05.05.2021	
3	Формализация системы	25%	06.05.2021– 30.05.2021	
4	Тестирование	5%	31.05.2021– 02.06.2021	
5	Оформление пояснительной записки	15%	03.06.2021– 14.06.2021	

Дата выдачи задания "28" _____ декабря _____ 2020

Руководитель бакалаврской работы
(подпись, дата)



Задание к исполнению принял студент
(подпись, дата)



Работу к защите допустить
Декан факультета
(подпись, дата)

Реферат

Пояснительная записка содержит 87 листов, 42 рисунка, 17 таблиц, 21 использованных источников, 2 приложения.

С, РАСПРЕДЕЛЕННЫЕ СИСТЕМЫ, РАСПРЕДЕЛЕННЫЙ КОНСЕСУС, ОДНОРАНГОВАЯ СЕТЬ, RPC, IPC, КОНЕЧНЫЕ АВТОМАТЫ, РЕПЛИКАЦИЯ, ВЕРОЯТНОСТНЫЕ СТРУКТУРЫ ДАННЫХ, ХРАНЕНИЕ ДАННЫХ

Объектом разработки являются программные средства отказоустойчивого и высоконагруженного хранилища данных.

Цель работы – построение системы хранения данных так, чтобы она могла терпеть отказы, обрабатывать большие объемы данных, а также быстро реагировать на запросы пользователей.

Результатом работы является отказоустойчивый кластер, а также повышение производительности некоторых компонентов системы по сравнению с другими подобными решениями.

Построение графиков производительности системы производилось с помощью языка программирования python и библиотеки matplotlib.

Разработка проводилась на языке программирования C. В качестве среды разработки использовалась IDE Visual Studio 2019. Разработка проводилась с использованием операционной системы Windows 10.

Был получен акт о внедрении данного продукта в состав программного комплекса, разрабатываемого ООО НТП «Криптософт».

					ПГУ 09.03.04 – 08ВР171.29 ПЗ						
Изм.	Лист	№ докум.	Подп.	Дата	Распределенная высоконагруженная система хранения данных Пояснительная записка	Лист		Лист		Листов	
Разраб.		Чурсин П.А.		21.06.21							
Пров.		Афопин А.Ю.		21.06.21					3		87
Реценз.						Группа 17ВР1					
И. контр.		Такташкин Д.В.		22.06.21							
Утв.		Макарычев П.П.									

Содержание

Введение	6
1 Анализ предметной области	8
1.1 Распределенные системы	8
1.1.1 CAP теорема	9
1.1.2 Модели согласованности	12
1.1.3 Репликация данных	13
1.1.4 Синхронизация времени	15
1.1.5 Распределенный консенсус	16
1.2 Системы хранения данных	19
1.2.1 Карты данных постоянного хранилища	19
1.2.2 Фильтр Блума	21
1.2.3 Восстановление после сбоев	22
1.3 Протокол консенсуса Raft	22
1.3.1 Основные понятия	23
1.3.2 Эпохи	24
1.3.3 Реплицированный журнал	24
1.3.4 Фазы работы	25
2 Проектирование и разработка	27
2.1 Архитектура системы	27
2.2 Модули ядра системы	29
2.2.1 Буферизованные каналы	29
2.2.2 Неблокируемые очереди сообщений	31
2.2.3 Пул регионов памяти	33
2.2.4 Асинхронный таймер	34
2.2.5 Обертка над сокетами	37
2.3 Модули бизнес-логики	38
2.3.1 Модуль распределенного консенсуса	38
2.3.2 Вызов удаленных процедур	39
2.3.3 Модуль одноранговой сети	44
2.3.4 Реплицированный журнал	49

3 Повышение производительности распределенного хранилища.....	53
3.1 Модификация реплицированного лога Raft.....	53
3.1.1 Вставка множества записей в конец лога.....	55
3.1.2 Удаление записей до и после индекса в логе.....	56
3.1.3 Результаты применения структуры	58
3.2 Модификация структуры хранения данных принятых конечным автоматом	63
3.2.1 Lock-Free структуры данных и CAS операция	64
3.2.2 Проблема распараллеливания красно-черных деревьев	64
3.2.3 Результаты применения структуры	65
4 Формализация системы.....	68
4.1 Существование одного лидера	68
4.2 Выборы лидера	68
4.3 Согласованность журнала.....	70
5 Тестирование системы	72
5.1 Метрики кода.....	75
5.1.1 Метрики Холстеда.....	76
5.1.2 Метрики Чепина	78
Заключение	80
Список использованных источников	81
Приложение А. Диаграммы модуля ядра	83
Приложение Б. Акт о внедрении	86

Введение

Современные системы, которые должны обслуживать миллионы пользователей одновременно, необходимо строить таким образом, чтобы они могли справляться с огромными нагрузками на сеть, максимально использовать данные им ресурсы CPU, а также исправно функционировали несмотря на аппаратные и другие отказы.

Большое количество систем относится к виду высоконагруженных данными (data-intensive). В отличие от высоконагруженных вычислений (compute-intensive) основной проблемой таких систем является не процессорная производительность, а объем, характер и скорость изменения данных. Обычно высоконагруженные данными системы создаются как композиция стандартных блоков, которые позволяют обеспечить необходимую функциональность. Например, большинство систем требует следующий функционал:

- хранить данные так, чтобы эти данные было возможно получить из других систем;
- хранить результаты ресурсоемких задач для увеличения скорости их чтения, а также, например, разгрузки операций аппаратурой постоянного хранения (кэши);
- составлять поисковые индексы для поиска и фильтрации данных различными способами;
- обеспечивать отказоустойчивость к потере данных с помощью репликации или аппаратных устройств;
- распределять данные для балансировки нагрузки на отдельные реплики данных.

Объединение всех этих функциональных качеств в системах несет за собой большой спектр проблем, решение которых является нетривиальным. Многие существующие алгоритмы и структуры данных позволяют обеспечить все перечисленные выше требования, но для этого необходима их композиция, а также модернизация.

Любой современный сервис боится потерять данные из-за какого-либо сбоя. Это, как если бы социальная сеть потеряла данные о своих пользователях из-за отказавшего жесткого диска или неверно написанной программы. Также бывает, что системный администратор может совершить ошибку и оборвать сеть между некоторыми серверами. Все эти проблемы могут привести к негативному опыту использования сервисов. Таким образом отказоустойчивость является актуальной темой для построения современных систем, а достичь ее можно с помощью распределения вычислений и построения кластера.

Также актуальна тема нагрузок на сервисы и их хранилища данных. Современные системы обрабатывают большое количество данных, и эта обработка должна осуществляться максимально производительно, ведь от скорости получения ответов пользователем зависит опыт работы с системой и не каждый захочет пользоваться тем, что долго обрабатывает запросы. Так у пользователей складывается негативный или позитивный опыт с системой.

Считается, что среднее время наработки на отказ жестких дисков составляет от 10 до 50 лет. Таким образом, в кластере с 10 тысячами жестких дисков следует ожидать в среднем одного отказа в день [1, 2].

Таким образом, планируется построить систему, которая способна выдерживать работу с большими объемами данных, обеспечивать их быструю обработку, а также справляться с различными отказами, как программными, так и аппаратными.

1 Анализ предметной области

1.1 Распределенные системы

Распределенная система, также известная как распределенные вычисления, представляет собой систему с несколькими компонентами, расположенными на разных машинах, которые обмениваются данными и координируют действия, чтобы выглядеть для конечного пользователя как единая согласованная система.

Машины, которые являются частью распределенной системы, могут быть компьютерами, физическими серверами, виртуальными машинами, контейнерами или любым другим узлом, который может подключаться к сети, иметь локальную память и обмениваться данными посредством передачи сообщений.

Распределенные системы обычно относятся к одной из четырех различных базовых архитектурных моделей [3].

Клиент-сервер. Клиенты обращаются к серверу за данными, затем форматируют их и отображают конечному пользователю. Конечный пользователь также может внести изменение на стороне клиента и передать его обратно на сервер, чтобы сделать его постоянным.

Трехуровневый – информация о клиенте хранится на среднем уровне, а не на клиенте, чтобы упростить развертывание приложения. Эта модель архитектуры наиболее распространена для веб-приложений.

N-уровневый – обычно используется, когда приложению или серверу необходимо перенаправить запросы к дополнительным корпоративным службам в сети.

Одноранговая сеть – нет дополнительных машин, используемых для предоставления услуг или управления ресурсами. Обязанности равномерно распределяются между машинами в системе, известными как одноранговые узлы, которые могут выступать в качестве клиентов или серверов.

1.1.1 CAP теорема

Теорема CAP (consistency, availability, partition-tolerance) была введена как компромисс между согласованностью, доступностью и устойчивостью к разделению в распределенных системах [4]. Теорема гласит о том, что одновременно можно выбрать только два из трех свойств для построения распределенной системы (рисунок 1).

Согласованность (consistency). Все данные хранимые на узлах должны быть одинаковы, а также система должна обрабатывать запросы в том порядке, в котором они в нее поступают.

Доступность (availability). Система должна быть доступна в любом случае и на каждый запрос к системе будет получен ответ.

Устойчивость к разделению (partition-tolerance). Потеря канала общения между узлами системы не должна влиять на работоспособность системы.



Рисунок 1 – CAP теорема

AC – MySQL, SQL Server, Postgres.

AP – Amazon Dynamo, Cassandra, Voldemort, CouchDB, Riak.

CP – Neo4J, Google Bigtable, MongoDB HBase, Hypertable, Redis.

Данная теорема является эвристической, но за долгое время ее использования на практике доказала свою пригодность при проектировании распределенных систем. Таким образом выбор компромисса при проектировании систем должен стоять первым пунктом.

Рассмотрим очень простую распределенную систему. Пусть система состоит из двух серверов, S_1 и S_2 . Оба этих сервера отслеживают одну и ту же переменную v , значение которой изначально v_0 . Пусть S_1 и S_2 могут общаться друг с другом, а также с внешними клиентами. На рисунке 2 показано как может выглядеть такая система.

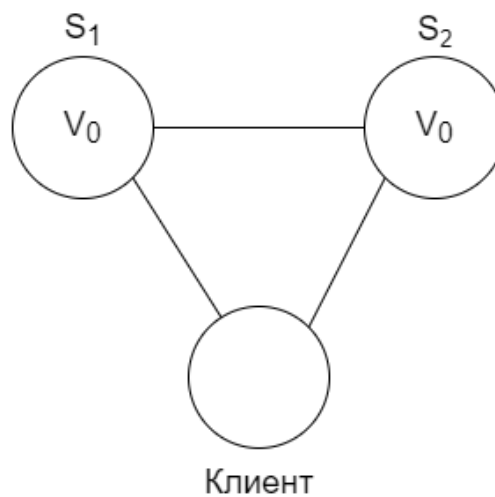


Рисунок 2 – Пример распределенной системы из двух узлов

Клиент может запросить запись и чтение переменной v с любого сервера. Когда сервер получает запрос, он выполняет вычисления, а затем отвечает клиенту. На рисунке 3 изображен пример записи значения v_1 на S_1 .

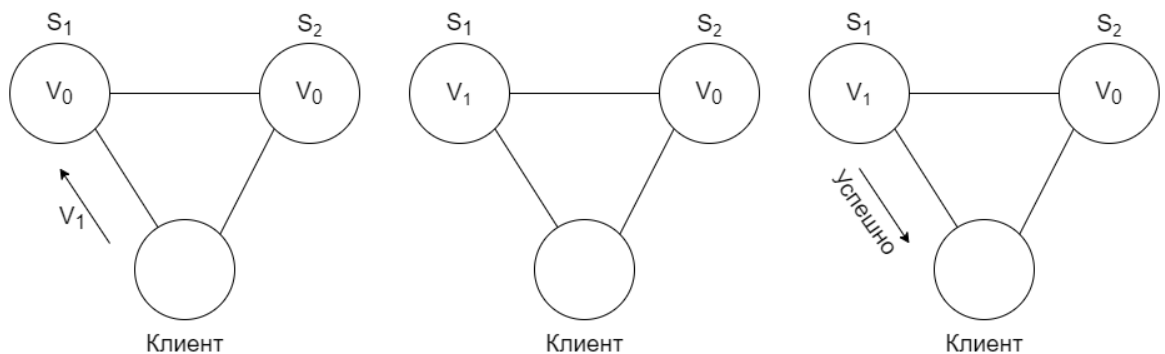


Рисунок 3 – Пример записи в переменную на одном из серверов

На рисунке 4 изображен пример чтения переменной v с сервера S_1 .

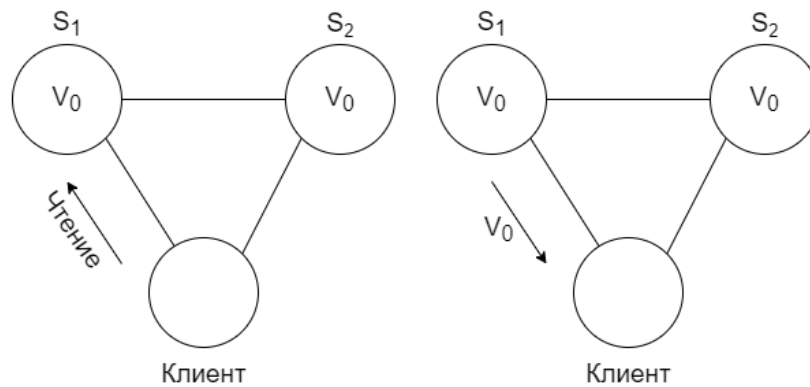


Рисунок 4 – Пример чтения переменной с одного из серверов

Предположим для противоречия, что действительно существует система, которая является согласованной, доступной и устойчивой к разделению. Первое, что мы делаем, это разбиваем нашу систему на два несвязных узла (рисунок 5). Теперь сервера S_1 и S_2 не имеют канала общения.

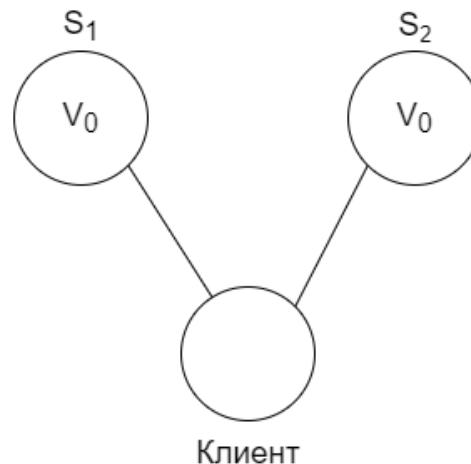


Рисунок 5 – Пример разделенной системы

Теперь когда клиент делает запрос на запись значения v_1 серверу S_1 . Сервер S_1 должен ответить так как система обладает свойством доступности по CAP теореме. Однако, поскольку сеть разделена, то S_1 не может реплицировать свои данные на S_2 и при попытке чтения переменной v с сервера S_2 клиент получит несогласованное значение (рисунок 6).

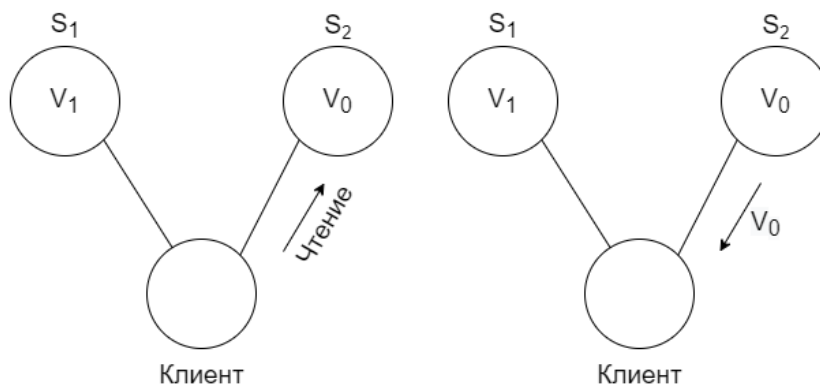


Рисунок 6 – Попытка чтения несогласованного значения из системы

Таким образом можно сделать вывод о том, что система не может быть согласованной, доступной и устойчивой к разделению одновременно. В рамках данной работы будет построена CP система.

1.1.2 Модели согласованности

Модели согласованности используются в распределенных системах, таких как системы распределенной общей памяти или распределенные хранилища данных (например, файловые системы, базы данных, системы репликации или веб-кэширование). Считается, что система поддерживает данную модель, если операции с памятью подчиняются определенным правилам. Модель согласованности данных определяет контракт между программистом и системой, в котором система гарантирует что, если программист следует правилам, память будет согласованной, а результаты чтения, записи или обновления памяти будут предсказуемыми [5].

На рисунке 7 представлена схема существующих моделей согласованности и их связей. Чем выше находится модель, тем более сильные ограничения на операции с данными она накладывает.

В случае с хранилищами данных, все данные поступающие в систему должны подчиняться правилам выбранной модели согласованности. Данная работа делает упор на линеаризуемость поступающих данных.

Линеаризуемость (linearizable) – одна из самых сильных моделей согласованности, и она подразумевает, что каждая операция, по-видимому,

происходит атомарно, в некотором порядке, в соответствии с порядком этих операций в реальном времени: например, если операция А завершается до начала операции В, то В по логике должен вступить в силу после А [6].

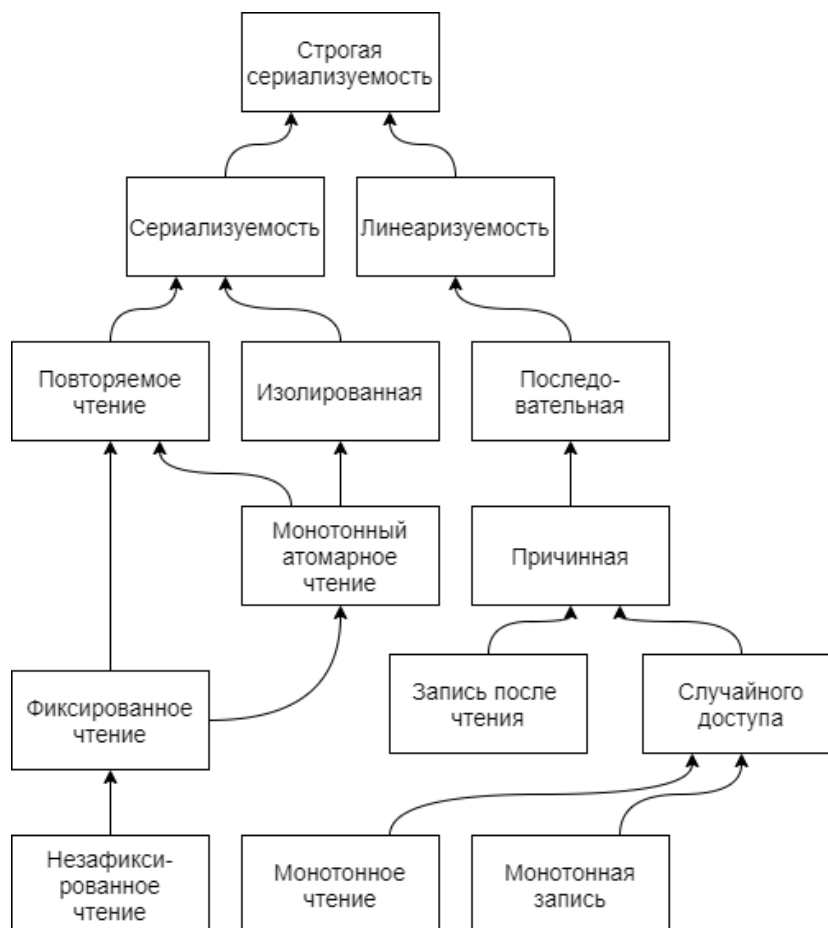


Рисунок 7 – Модели согласованности

Линеаризуемость была выбрана в качестве модели согласованности, из-за компромисса *CP* системы. Система жертвует доступностью в пользу согласованности, поэтому должна быть выбрана достаточно сильная модель.

1.1.3 Репликация данных

Копирование одних и тех же данных на несколько узлов в распределенных системах критически важно для поддержки базы данных и продолжения обслуживания запросов даже во время сбоев [7]. Такое копирования называется репликацией данных.

Такой подход к организации данных между узлами распределенной системы позволяет обеспечить следующие качества для системы:

- географическое распределение данных, для уменьшения задержек при работе с системой из разных регионов;
- способность продолжать работу при отказе некоторых узлов;
- повышение пропускной способности.

Существует несколько способов репликации данных между узлами распределенной системы:

- репликация с единым лидером;
- репликация с множеством лидеров;
- репликация без лидера.

При репликации с единым лидером, система имеет один узел, на который поступают все запросы, и он реплицирует их на все остальные узлы.

В репликации без лидера существует промежуточный слой между системой и клиентами, который реплицирует данные на узлы.

В рамках данной работы выбрана репликация с единым лидером в кластере (рисунок 8). Все остальные узлы в такой системе называются ведомыми (follower). При таком подходе в системе всегда должен существовать лидер.

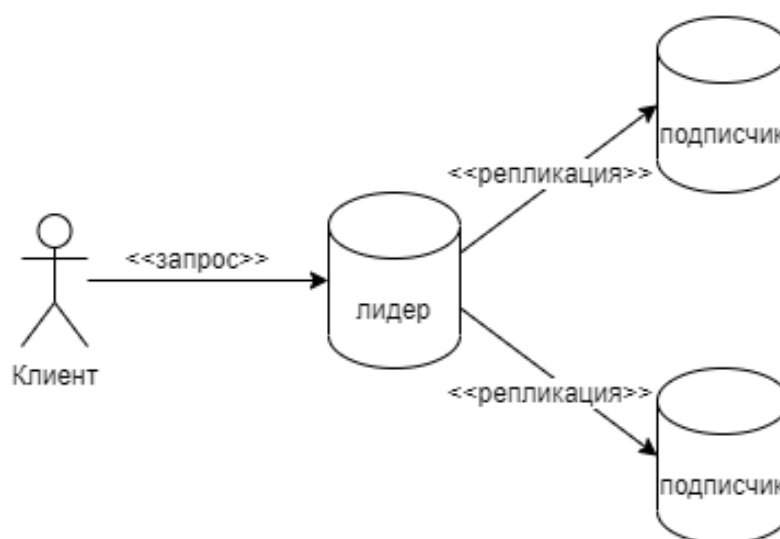


Рисунок 8 – Репликация данных с одним лидером

Если лидер выходит из строя, то ведомые узлы инициируют голосование и на основе полноты реплицированных данных каждого из узлов выбирают нового лидера. Процесс голосования будет подробно разобран в главе посвященной реализации системы.

1.1.4 Синхронизация времени

В распределенных системах аппаратные и программные компоненты взаимодействуют и координируют свои действия посредством передачи сообщений. Каждый узел в распределенных системах может делиться своими данными с другими узлами. Таким образом, необходимо правильное распределение данных для сохранения их согласованности и помощи в координации между несколькими процессами. Для разрешения таких конфликтов используется синхронизация. Синхронизация в распределенных системах достигается с помощью часов.

Использование реальных часов в таких системах невозможно, потому что синхронизировать их на всех узлах распределенной системы невозможно, так как все они могут иметь разную точность.

Так, например, GPS (global position system) использует квантовые часы для синхронизации. Также в таких часах стоят аппаратные устройства для обнаружения ошибок и все часы между собой высчитывают погрешность от релятивистских эффектов. В любом случае даже очень точные часы с течением времени начнут дрейфовать относительно друг друга.

Для решения проблемы синхронизации часов в распределенных системах существуют логические часы. Для построения логических часов можно использовать следующие алгоритмы:

- скалярное время Лэмпорта [8];
- векторное время;
- матричное время.

В данной работе будет использоваться скалярное время Лэмпорта. Для линеаризации событий в системе Лэмпорт предложил использовать функцию, которая отображает множество событий на множество положительных целых чисел. Таким образом локальное время каждого процесса P_i и его представление о глобальном логическом времени системы выражаются скалярной величиной L_i .

Перед выполнением каждого события в системе процесс P_i увеличивает показания своих часов:

$$L_i = L_i + d, \text{ где } d > 0.$$

В каждое сообщение, передаваемое между узлами системы добавляется L_i , так каждый узел может понять порядок событий поступающих в систему.

Такой подход был выбран из-за репликации с помощью единого лидера.

1.1.5 Распределенный консенсус

Простейший способ реализовать распределённую систему – создать набор клиентов, которые посылают команды избранному центральному узлу-серверу. Сервер должен быть описан как детерминированный конечный автомат, который выполняет в определенном порядке команд от клиентов. Конечный автомат имеет собственное текущее состояние. Шаг работы автомата состоит в обработке очередной команды, в результате автомат возвращает ответ клиенту и меняет своё текущее состояние.

Реализация, использующая избранный центральный сервер, не является отказоустойчивой, так как в случае отказа центрального сервера система перестанет обрабатывать команды клиентов. Таким образом, для того, чтобы обеспечить отказоустойчивость, системе следует использовать набор серверов, каждый из которых представляет собой конечный автомат. Так как конечный автомат – детерминированный, все серверы обязаны выполнять одинаковые последовательности переходов состояний при

условии одинаковых входных данных. Клиент, для которого выполняется команда, может использовать любой сервер.

Алгоритмы принятия консенсуса, как правило, возникают в контексте применения распределенных конечных автоматов. При таком подходе, конечные автоматы, расположенные на множестве серверов кластера, выполняют идентичную команду, находясь в одном и том же состоянии, таким образом кластер может продолжать работу, даже если некоторые из его серверов отказывают. Таким образом, распределенные автоматы используются для решения разнообразных проблем отказоустойчивости.

Распределенные автоматы, как правило, реализуется с помощью реплицированного журнала, как показано на рисунке 9. На каждом сервере хранится реплицируемый журнал, содержащий последовательность команд, которые его собственный конечный автомат выполняет в фиксированном порядке. Каждый журнал содержит одинаковые команды.

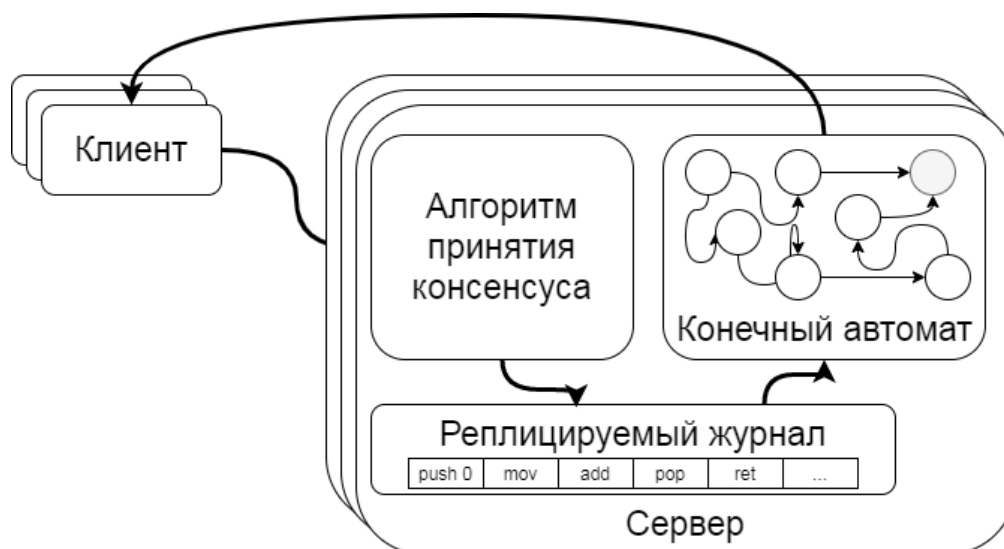


Рисунок 9 – Распределенная система, использующая распределенный конечный автомат

Поддержание реплицированного журнала согласованным является непосредственной задачей алгоритма принятия консенсуса. Модуль алгоритма принятия консенсуса, на сервере получает команды от клиентов и добавляет их в свой журнал. Он общается с модулями алгоритма принятия консенсуса на других серверах, чтобы гарантировать, что каждый журнал в

конечном итоге содержит одни и те же команды в одинаковом порядке, даже если некоторые серверы отказали. После того, как команда клиента успешно реплицируется, конечный автомат каждого сервера обрабатывает их согласно журналу, результаты работы возвращаются клиентам.

Один из подходов к достижению консенсуса заключается в том, чтобы все узлы договорились о значении большинства. В этом контексте большинству требуется, по крайней мере, один более половины доступных голосов (где каждый получает голос). Однако один или несколько узлов могут исказить результирующий результат, так что консенсус не может быть достигнут или достигнут неправильно (рисунок 10).

Существует два основных вида запроса:

- prepare (предложение);
- assert (принятие решения).

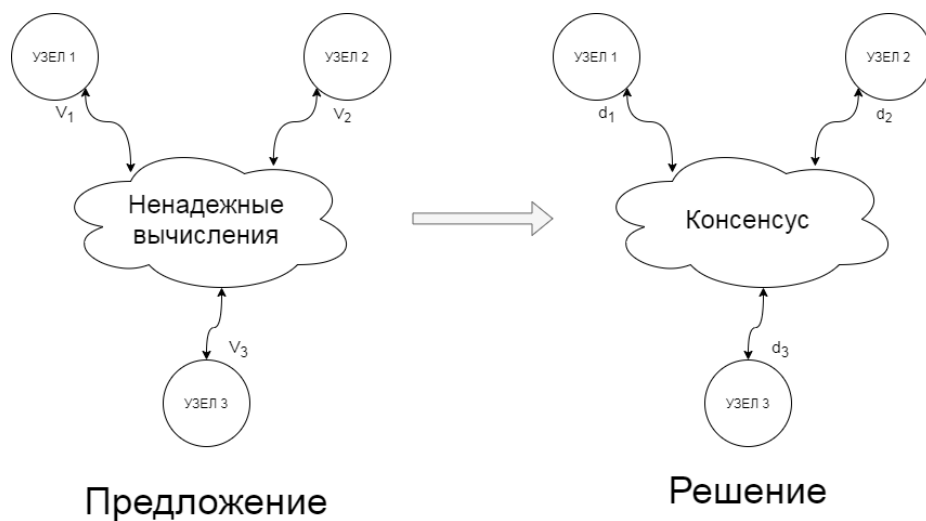


Рисунок 10 – Принятие решения узлами распределенной системы

Свойства корректного консенсуса:

- все правильные узлы в конечном итоге принимают решение v_i ;
- все правильные узлы выбирают одно и тоже решение даже если они потом упадут;
- если все правильные узлы предложили одно и тоже v_i , то каждый правильный узел должен решить v_i .

1.2 Системы хранения данных

Основной задачей систем хранения данных является сохранение и предоставление данных. Несмотря на одинаковые на одинаковые задачи, каждое хранилище может быть оптимизировано под конкретные цели.

Существует большое количество структур данных и алгоритмов с помощью которых можно оптимизировать хранилища под определенные нужды.

Среди всех существующих самыми популярными являются следующие виды баз данных:

- реляционные;
- объектно-ориентированные;
- документоориентированные;
- нереляционные.

В данной работе будет реализовано хранилище типа «ключ-значение». Такие базы данных относятся к нереляционным и могут использовать модели согласованности, в частности линейизуемость.

1.2.1 Карты данных постоянного хранилища

Данные в хранилище могут храниться как в оперативной памяти, так на жестком диске. Для хранения данных типа «ключ-значение» в оперативной памяти можно использовать большое количество различных структур: деревья, хэш-таблицы и другие, тогда как для хранения данных в постоянном хранилище часто лучшим вариантом является бинарный упорядоченный формат. Так как поиск по хранилищу на жестком диске достаточно дорогостоящая операция, необходимо как-то ускорить этот процесс.

Для таких задач используются хеш-индексы. Хеш-индексы хранятся в оперативной памяти и содержат смещение по каждому ключу в файле с данными (рисунок 11).

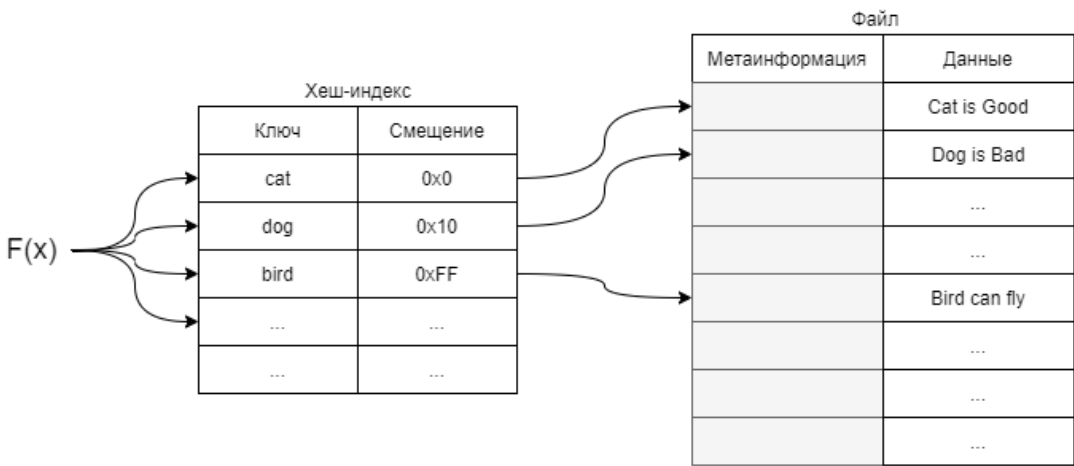


Рисунок 11 – Организация хеш-индекса

Конечный файл может содержать некоторую метаинформацию, необходимую для обеспечения согласованности и других целей, что будет рассмотрено далее.

Для получения смещения в файле необходимо вычислить значение хэш-функции по заданному ключу. Каждая запись в хеш-индексы имеет фиксированный размер, поэтому по результату хеш-функции можно получить смещение в оперативной памяти до индекса необходимо ключа.

Представленный способ используется в системе Vitcask.

В данной работе предлагается использовать индекс и сводку раздела. Индекс раздела представляет из себя файл, который содержит смещения для каждого ключа хранилища. Сводка раздела представляет из себя множество записей, каждая из которых указывает на записи индекса раздела с шагом в d . Учитывая тот факт, что записей могут быть миллиарды, разместить ключи записи и указатели на них полностью в оперативной памяти становится проблематично, а без сводки раздела пришлось бы считывать последовательно весь индекс для поиска необходимого значения. В таком случае использование одного указателя в оперативной памяти на каждые d записей индекса выглядит очень логично. Такой подход используется в системе Cassandra.

1.2.2 Фильтр Блума

Со временем на жестком диске может разместиться большое количество SS-Table, тогда придется производить поиск по каждой из них, что может занимать большое количество времени. Для решения этой проблемы будет использоваться фильтр Блума.

Фильтр Блума – вероятностная структура данных, позволяющая определить с заданной вероятностью принадлежность элемента множеству. Использование такой структуры данных может уменьшить количество обращений к жесткому диску, но потребляет большое количество оперативной памяти (рисунок 12).

Фильтр Блума представляет собой битовый массив из m бит. Изначально, когда структура данных хранит пустое множество, все m бит обнулены. Пользователь должен определить k независимых хеш-функций $h_1, \dots, h_k$ каждая из которых отображает множество элементов в множество мощностью m . Иными словами, каждому элементу хеш-функция сопоставляет число от 1 до m . Для каждого элемента e биты массива с номерами $h_1(x), \dots, h_k(x)$ равными значениям хеш-функций устанавливаются в 1.

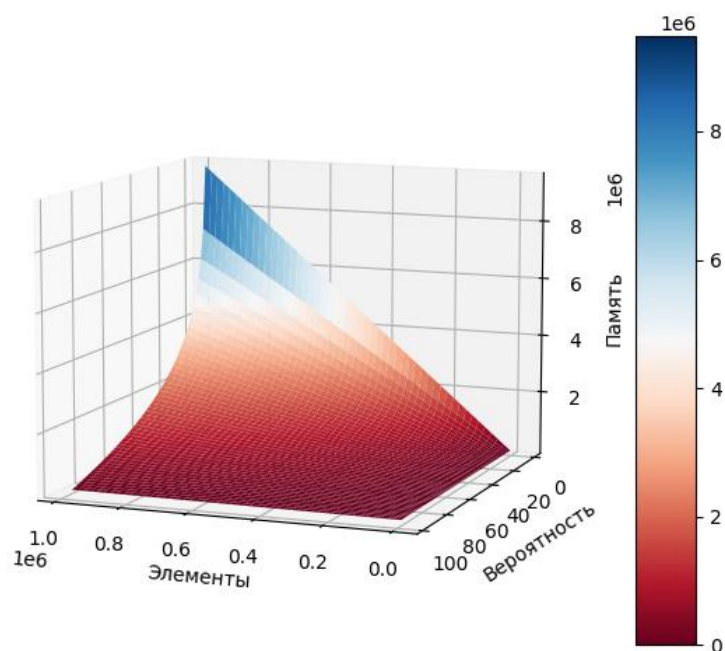


Рисунок 12 – График потребления памяти фильтром Блума

Фильтр Блума может потреблять большое количество оперативной памяти, поэтому процент ложноположительных срабатываний не должен стремиться к нулю.

1.2.3 Восстановление после сбоев

После сбоев система должна восстанавливать свое состояние так, будто она и не прекращала работать, но некоторые данные эксклюзивно могли храниться только в оперативной памяти.

Для решения таких проблем в системах хранения данных используется журналирование. Каждый раз, когда записи поступают в базу данных, прежде чем попасть в таблицу оперативной памяти они записываются на жесткий диск в специальный файл. Такой файл называется журналом.

Все поступающие записи попадают в конец журнала, так как такая запись является самой быстрой, а также нам не обязательно соблюдать порядок записей и выстраивать их организацию.

После каждого сбоя, система читает этот файл и заново заполняет таблицы в оперативной памяти, таким образом мы можем восстанавливаться, не теряя данные.

1.3 Протокол консенсуса Raft

Raft – это алгоритм распределенного консенсуса основанный на реплицированном конечном автомате. 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, в каждом из которых работает машина с реплицированным состоянием.

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.

1.3.1 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.

1.3.2 Эпохи

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.

1.3.3 Реплицированный журнал

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.

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.

1.3.4 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 в кластере (рисунок 13).

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.

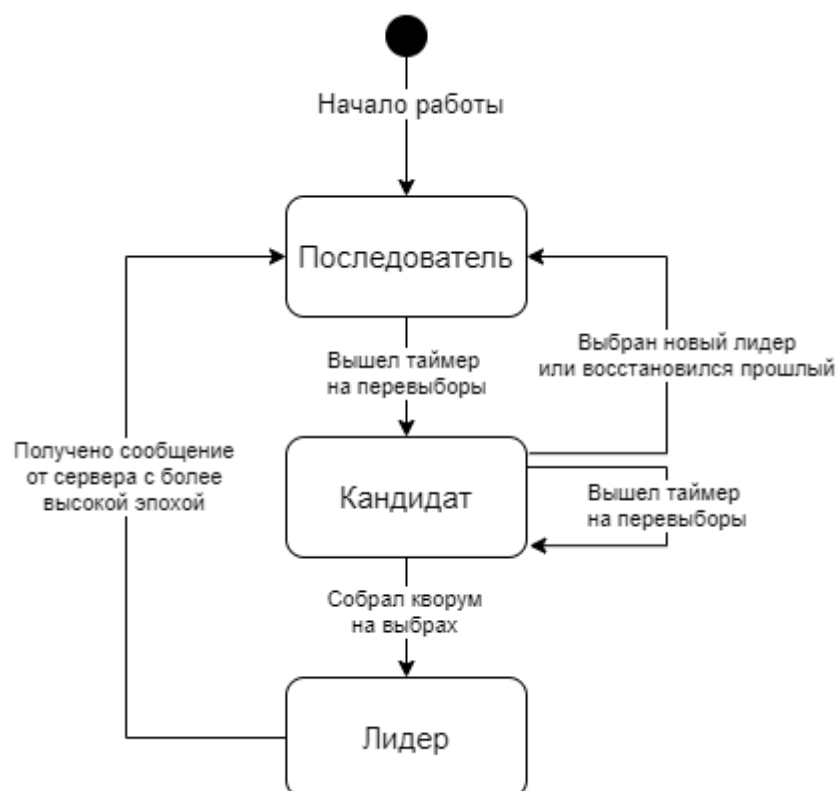


Рисунок 13 – Диаграмма состояний для конечного автомата Raft

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 Проектирование и разработка

В качестве языка программирования для разработки системы был выбран C в сочетании с интегрированной средой разработки Microsoft Visual Studio 2019. Язык C был выбран из-за максимального приближения к аппаратным средствам, прямым использованием возможностей операционных систем, а также из-за гибкого управления памятью, что в свою очередь позволяет построить наиболее высокопроизводительную систему.

Программное обеспечение будет исполняться под операционной системой Windows NT. В Windows есть большой набор инструментов для разработки программного обеспечения, поддерживается POSIX, а также его API имеет широкий набор функционала на уровне системного программирования. Также с недавнего времени в Windows есть подсистема Linux, которая позволяет исполнять программы, написанные для операционной системы Linux.

В качестве аппаратной архитектуры для выполнения системы используется Intel x86. Система прямо ориентирована под эту архитектуру и использует некоторый набор операций доступный только для данной архитектуры. Например, используются атомарные процессорные операции для сокращения задержек на планирование потоков операционной системой, вместо использования блокировок, также системой используются некоторые AVX и SSE операции доступные только на процессорах Intel.

2.1 Архитектура системы

В ходе проектирования системы было решено разбить ее на модули, каждый из которых может существовать независимо от других. Такая абстракция позволит построить масштабируемый код, где из модулей может выстраиваться некоторая иерархия, где каждый другой модуль может зависеть от другого. Предлагается максимально абстрагировать каждый модуль на нижнем уровне иерархии, эти модули будут составлять ядро

системы, таким образом можно добиться максимальной поддержки и расширяемости кода системы.

Каждый модуль системы принадлежит своему классу в иерархии. Таким образом было выделено три уровня иерархии: ядро, бизнес-логика и пользовательский уровень (рисунок 14).



Рисунок 14 – Иерархия модулей системы

Модули ядра – модули системы, которые являются неделимыми и используются для построения всех выше стоящих в иерархии модулей. Модули ядра делятся на два вида: компоненты и коннекторы.

Компоненты ядра – модули ядра, реализующие логику и обрабатывающие вызовы операционной системы.

Коннекторы ядра – один из важнейших видов модулей заданной архитектуры, позволяющий модулям на первых двух уровнях общаться между собой.

Модули бизнес-логики – реализующий функционал обозначений задачами, которые должна решать система.

Пользовательские модули – модули, реализующие логику для работы с системой извне. Эти модули необходимо программистами, которые могут

использовать данную систему для своих нужд. Эти модули полностью отделены от ядра системы и не имеют к нему доступа.

Коннекторы в системе очень важны из-за того, что для построения системы используются большое число потоков, каждый из которых имеет свой контекст исполнения. Некоторые потоки системы должны коммуницировать между собой, для этого и используются модули коннекторы.

2.2 Модули ядра системы

Ядро системы представляет собой неделимую сущность, на основе которой строится вся бизнес-логика системы, поэтому необходимо построить его так, чтобы все потребности были удовлетворены. Модули ядра делятся на компоненты ядра и коннекторы ядра.

В приложении А представлены диаграммы последовательности модулей ядра.

2.2.1 Буферизованные каналы

Буферизованные каналы (MessageChannel) – коннекторы входящие в состав ядра системы, которые представляют из себя разделяемую область память. Каналы реализуют две основные функции: запись и чтение в разделяемую память. Все операции над каналами являются потокобезопасными.

Каждый канал имеет буфер заранее установленного размера, этот буфер является разделяемым для потоков, которые работают с каналом.

При записи в канал поток блокируется до того момента, пока другой поток не прочитает передаваемые данные. Также и при попытке чтения, поток будет заблокирован до того, как другой поток не запишет данные в этот канал (рисунок 15).

Буферизованные каналы построены на событиях предоставляемых операционной системой Windows. Структура каналов представлена в таблице 1.

Дополнительно с передаваемыми данными канал позволяет передавать номер сигнала как дополнительное поле. Сигнал в каналах можно использовать для выбора обработчика входящих данных.

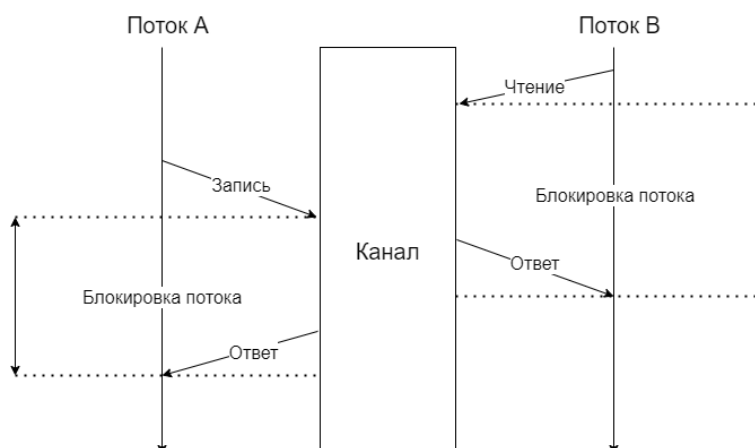


Рисунок 15 – Принцип работы буферизованного канала

Таблица 1 – Структура буферизованных каналов

Имя поля	Тип данных	Описание	Положение в иерархии
bIsWork	volatile BOOL	Флаг указывающий на то, что канал сейчас обрабатывает запросы. Запрещен для кэширования на процессоре.	Интегральный тип
dwChannelSize	DWORD	Размер канала в байтах	Интегральный тип
dwLocked	volatile DWORD	Количество потоков, пытающихся занять критическую секцию.	Интегральный тип

Продолжение таблицы 1

Имя поля	Тип данных	Описание	Положение в иерархии
hWaitRead	HANDLE	Событие чтения из канала	Тип предоставляемый WINAPI
hWaitWrite	HANDLE	Событие записи в канал	Тип предоставляемый WINAPI
hStopped	HANDLE	Событие прекращения работы канала	Тип предоставляемый WINAPI
pbChannelBuffer	BYTE	Буфер канала	Интегральный тип

Похожий подход реализован в каналах языка программирования Go. Главным отличием является то, что каналы языка Go работают с сопрограммами, а не с потоками.

2.2.2 Неблокируемые очереди сообщений

Некоторые компоненты могут не требовать блокировки до чтения из каналов, что могло бы повысить производительность для связи некоторых компонентов. Из-за того, что запись не является блокирующей все данные записанные в коннектор необходимо выстраивать в очередь, чтобы потоки читатели в правильном порядке могли исполнять поступающие в очередь данные. Таким образом для решения данной задачи были построены неблокируемые очереди сообщений (MessageQueue).

В целом очередь схожа с буферизованными каналами, но вместо одного буфера используется целый ряд заранее выделенных областей памяти, куда можно осуществлять запись.

В ходе работы каждая область памяти, выделенная под нужды передачи информации, делится на два типа: свободная и занятая. Такое деление необходимо чтобы каждый раз не перевыделять память под использованный буфер, он просто перейдет в разряд свободных и его можно будет использовать вновь.

Такая очередь может быть использована для построения пулов потоков, а также создания очередей задач.

Принцип работы очереди представлен на рисунке 16. Структура и организация очереди сообщений представлена в таблице 2.

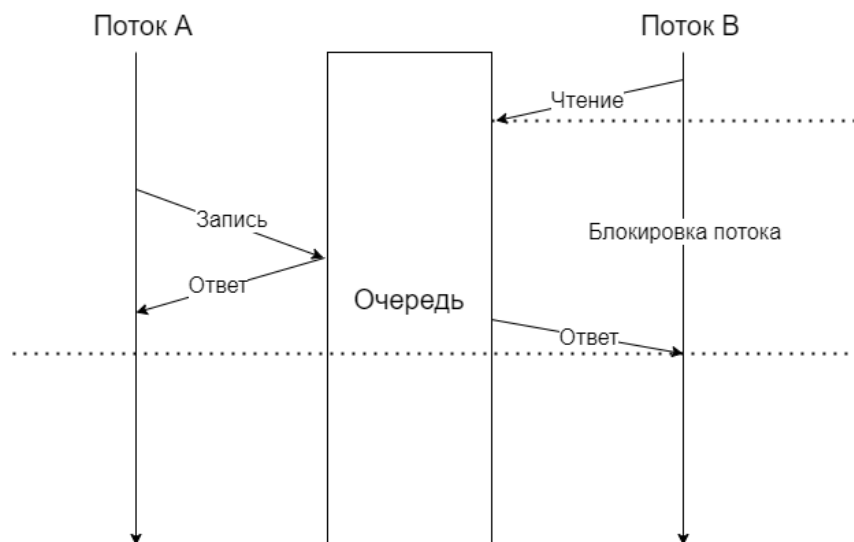


Рисунок 16 – Принцип работы неблокируемой очереди сообщений

Таблица 2 – Структура очереди сообщений

Имя поля	Тип данных	Описание	Положение в иерархии
bIsWork	volatile BOOL	Флаг указывающий на то, что канал сейчас обрабатывает запросы. Запрещен для кэширования на процессоре.	Интегральный тип
dwEntrySize	DWORD	Максимальный размер записи передаваемой в очередь	Интегральный тип
crLock	CRITICAL_SECTION	Блокировка критической секции на работу с очередью	Тип предоставляемый WINAPI
hEvent	HANDLE	Событие поступления новой записи в очередь	Тип предоставляемый WINAPI
sFreeList	SList	Список свободных областей памяти	Модуль ядра

Такие очереди можно использовать в различных системах массового обслуживания, например, в данной системе такая очередь позволяет распределять соединения по потокам из пула.

2.2.3 Пул регионов памяти

Пул регионов памяти (ArenaPool) – потокобезопасный аллокатор памяти, который располагает регионы памяти в отсортированном виде по их размеру и выдает их по запросу. Необходимость написания такого аллокатора пришла с тем, что в системе происходит очень много выделений памяти, а это не дешевая операция, поэтому было принято решение хранить заранее выделенные регионы памяти, так чтобы их можно было быстро получать.

Данный аллокатор умеет расширять количество регионов в зависимости от потребностей системы, а также реаллоцировать их размеры. Структура аллокатора памяти и выделяемых регионов представлены в таблицах 3 и 4 соответственно.

Таблица 3 – Структура аллокатора памяти

Имя поля	Тип данных	Описание	Положение в иерархии
dwArenasCount	volatile DWORD	Количество регионов памяти в пуле	Интегральный тип
dwArenasAvailible	volatile DWORD	Количество доступных регионов в пуле	Интегральный тип
dwWaitThreadCount	volatile DWORD	Количество потоков ожидающих регион	Интегральный тип
dwArenaMaxMemory	DWORD	Максимальный размер региона	Интегральный тип
psArenaList	SList	Список регионов памяти	Модуль ядра

Таблица 4 – Структура региона памяти

Имя поля	Тип данных	Описание	Положение в иерархии
pOwner	PVOID	Указатель на владельца региона (пул)	Интегральный тип
dwCapacity	DWORD	Вместимость региона	Интегральный тип
dwUsed	DWORD	Количество использованной памяти региона	Интегральный тип
pbMemory	BYTE	Буфер региона	Интегральный тип

Данный компонент выделяет буферы таким образом, чтобы их выравнивание позволяло выполнять над ними AVX и SSE операции, предоставляемые процессорами Intel.

2.2.4 Асинхронный таймер

В системе есть несколько таймеров, которые необходимо для поддержания распределенного консенсуса, поэтому необходимо сделать их максимально производительными. Также для поддержания архитектуры таймер будет использовать канал для сигнализации об его истечении. Таким образом один из сигналов в каналах будет обработка таймера.

Для реализации данного компонента используется аппаратный таймер, который по истечении вызывает прерывание таймера.

Для того, чтобы таймер можно было взводить вновь пересоздания было решено выделить поток, который принимал бы сообщения через канал и взводил таймер, а в оставшееся время поток выгружался бы с исполнения операционной системой. Таким образом можно добавить полной асинхронности при работе с таймером.

Таймер можно сбрасывать, например, если узел перешел в другой режим работы и таймер больше не нужен. Также есть возможность перезаведения уже заведенного таймера, это необходимо для таймера перевыборов.

Принцип работы асинхронного таймера представлен на рисунке 17.

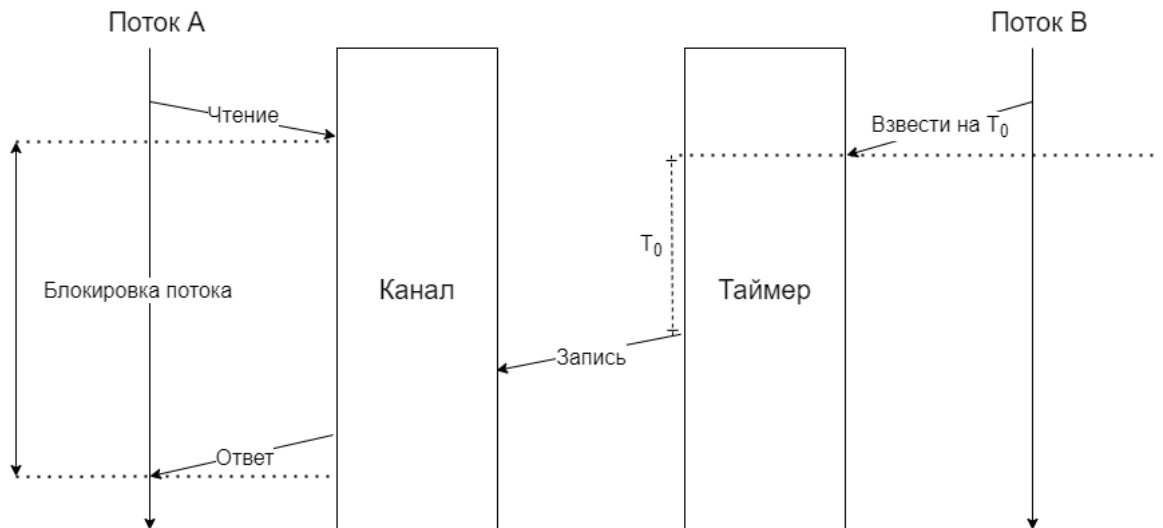


Рисунок 17 – Принцип работы асинхронного таймера

За счет того, что окончание работы таймера сигнализируется через канал, в это время поток читающий канал может принимать сообщения и от других потоков (рисунок 18).

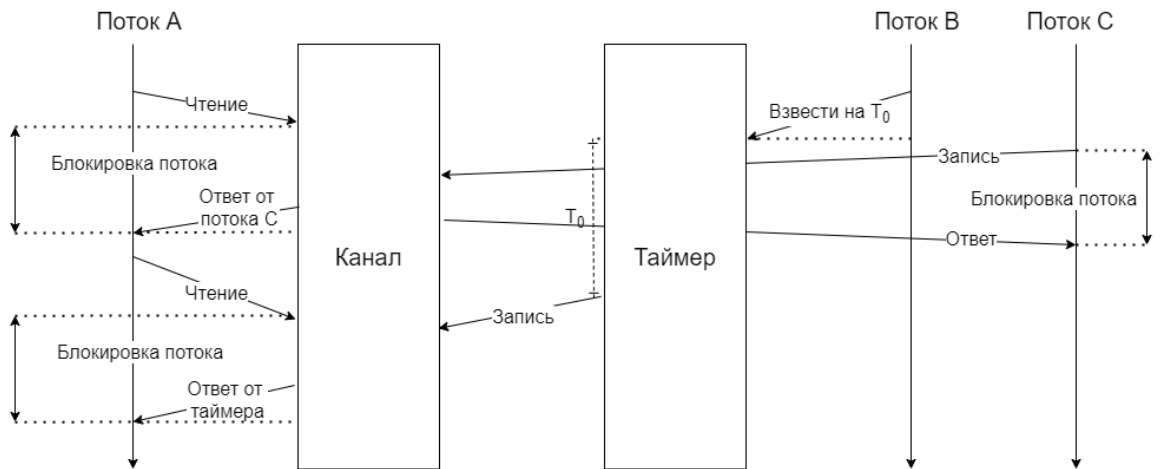


Рисунок 18 – Запись в канал разными потоками

Поток взводящий таймер не требует ответа, чтобы разблокироваться, так как таймер имеет свой поток, поэтому поток продолжает свое исполнение. Такой асинхронный подход взведения таймера накладывает некоторые расходы на реализацию, для этого внутри таймера создается свой поток, слушающий команды, но большую часть времени он снят с выполнения операционной системой, поэтому эти затраты не значительны.

Структура и организация таймера представлены в таблице 5.

Таблица 5 – Структура асинхронного таймера

Имя поля	Тип данных	Описание	Положение в иерархии
hThread	HANDLE	Поток для управления таймером	Тип предоставляемый WINAPI
hTimeoutChannel	MESSAGE_CHANNEL	Внутренний канал компонента для передачи сообщений в управляющий таймером поток	Модуль ядра
hWaitableTimer	HANDLE	Таймер предоставляемый операционной системой Windows	Тип предоставляемый WINAPI
bIsWork	volatile BOOL	Флаг указывающий на то, что таймер работает. Запрещен для кэширования на процессоре.	Интегральный тип
hMessageChannel	MESSAGE_CHANNEL	Внешний канал для передачи сигнала об истечении времени	Модуль ядра
dwSignalId	DWORD	Номер сигнала, указывается при инициализации таймера.	Интегральный тип

Такой таймер используется для определения выхода сроков на перевыборы лидера или для отправки сердцебиения.

2.2.5 Обертка над сокетами

Разрабатываемая система имеет большое количество взаимодействий между элементами по сети, поэтому было принято решение написать удобный для использования интерфейс сокетов (SocketWrapper).

Модуль должен передавать только заявленное количество данных, поэтому сначала передается заголовок, который содержит размер передаваемых данных, хэш-сумму и метаданные размером 128 байт. После того как конечное устройство получает заголовок выделяется буфер необходимого размера или берется регион из пула ядра системы. Таким образом не происходит лишних аллокаций в памяти, а также передачи лишних данных по сети, что позволяет снизить нагрузку на сеть.

Структура модуля SocketWrapper представлена в таблицах 6 и 7.

Таблица 6 – Структура заголовка сокетов

Имя поля	Тип данных	Описание	Положение в иерархии
dwSize	DWORD	Размер передаваемых данных	Интегральный тип
dwHash	DWORD	Хэш-сумма передаваемых данных	Интегральный тип
bMeta	BYTE[128]	Метаинформация	Интегральный тип

Таблица 7 – Структура обертки сокетов

Имя поля	Тип данных	Описание	Положение в иерархии
hSocket	SOCKET	Сокет	Тип предоставляемый WINAPI
pAddrInfo	PADDRINFOA	Адресная информация	Тип предоставляемый WINAPI

В работе используются только TCP сокет, так как необходимы гарантии доставки пакетов. Помимо гарантии доставки за счет протокола TCP/IP реализовано вычисление хэш-суммы передаваемых данных, для того чтобы сверить их на конечном устройстве. В качестве хэш-функции используется CRC32.

2.3 Модули бизнес-логики

Вся бизнес-логика системы состоит из двух головных модулей, которые разрешают задачи распределенной и высоконагруженной системы: блок распределенного консенсуса и блок хранилища данных.

2.3.1 Модуль распределенного консенсуса

Модуль распределенного консенсуса отвечает за поддержание отказоустойчивости и согласованности кластера, в его состав входят следующие компоненты:

- координатор распределенного консенсуса;
- peer-to-peer;
- реплицированный журнал.

Каждый из этих компонентов будет общаться между собой с помощью модулей коннекторов, которые будут описаны в разделе модулей ядра системы.

Координатор распределенного консенсуса принимает записи от конечных пользователей системы и записывает их в реплицированный журнал. После эти записи реплицируются на остальные узлы кластера с помощью peer-to-peer системы. Все конечные запросы, выходящие от пиров кластера, обрабатываются именно в этом модуле, здесь принимаются решения о выборах и разрешения несогласованной записей.

Peer-to-peer взаимодействие в кластере будет осуществляться через RPC (remote procedure call). Каждый узел кластера представляет из себя сервер и общается со всеми другими узлами. Через это взаимодействие

происходят репликация, голосование, установка снимков состояния кластера.

Реплицированный конечный автомат обрабатывает принятые записи и отправляет их в хранилище.

Также система в качестве отдельного модуля реализует RPC клиентское и серверное взаимодействие.

2.3.2 Вызов удаленных процедур

Вызов удаленных процедур (RPC) – механизм распределенных вычислений, позволяющий выполнять процедуры в другом адресном пространстве. В данном случае мы будем выполнять процедуры на других узлах кластера распределенной системы. Механизм должен обладать прозрачностью и делать вызовы так, будто они выполнялись там же где и вызывались.

Модуль распределенного консенсуса реализует следующие распределенные процедуры, которые необходимо вызывать как на своем узле, так и на других:

- репликация записей;
- инициирование голосования;
- отправка сердцебиения;
- установка снимков состояния.

Все RPC взаимодействие происходит через TCP сокет, предоставляемые операционной системой. В ходе проектирования было решено отказаться от сетевых протоколов выше транспортных по сетевой модели OSI в пользу получения производительности и компактной упаковки передаваемых данных.

В качестве сокетов используется обертка, описанная в модуле ядра системы «SocketWrapper».

Сам модуль вызова удаленных процедур делится на клиентский и серверный.

Серверный модуль должен принимать клиентские сокеты, читать с них данные и отдавать ответ. Для повышения производительности было решено использовать пул потоков для обработки клиентских соединений. Таким образом выделим основные потоки серверного модуля RPC:

- поток получения клиентских сокетов;
- пул потоков для обработки клиентских соединений.

После получения клиентского сокета поток А отправляет экземпляр соединения коннектору ядра – очередь сообщений. В очередь записывается экземпляр соединения и свободный поток из пула обработчиков вытягивает его.

В качестве данных клиент отправляет идентификатор вызываемой функции и все необходимые для вызова аргументы. Для хранения этих данных на время обработки запроса выбирается буфер из модуля ядра «ArenaPool».

Реализация обработчиков RPC функций устроена следующих образом:

- выбирается необходимая функция;
- разбираются аргументы функции;
- совершается вызов функции;
- создается канал обратной связи;
- аргументы отправляются в канал конечного реплицированного автомата;
- ожидание записи в канал обратной связи о завершении выполнения функции;
- отправка ответа RPC клиенту.

В данной реализации все функции создают канал обратной связи и передают его вместе с аргументами функции. Если в ходе выполнения функции произошли какие-либо ошибки, то при записи в канал обратной связи будет установлен флаг о неуспешном выполнении функции.

Устройство модуля вызова удаленных процедур представлено на рисунке 19 и таблице 8

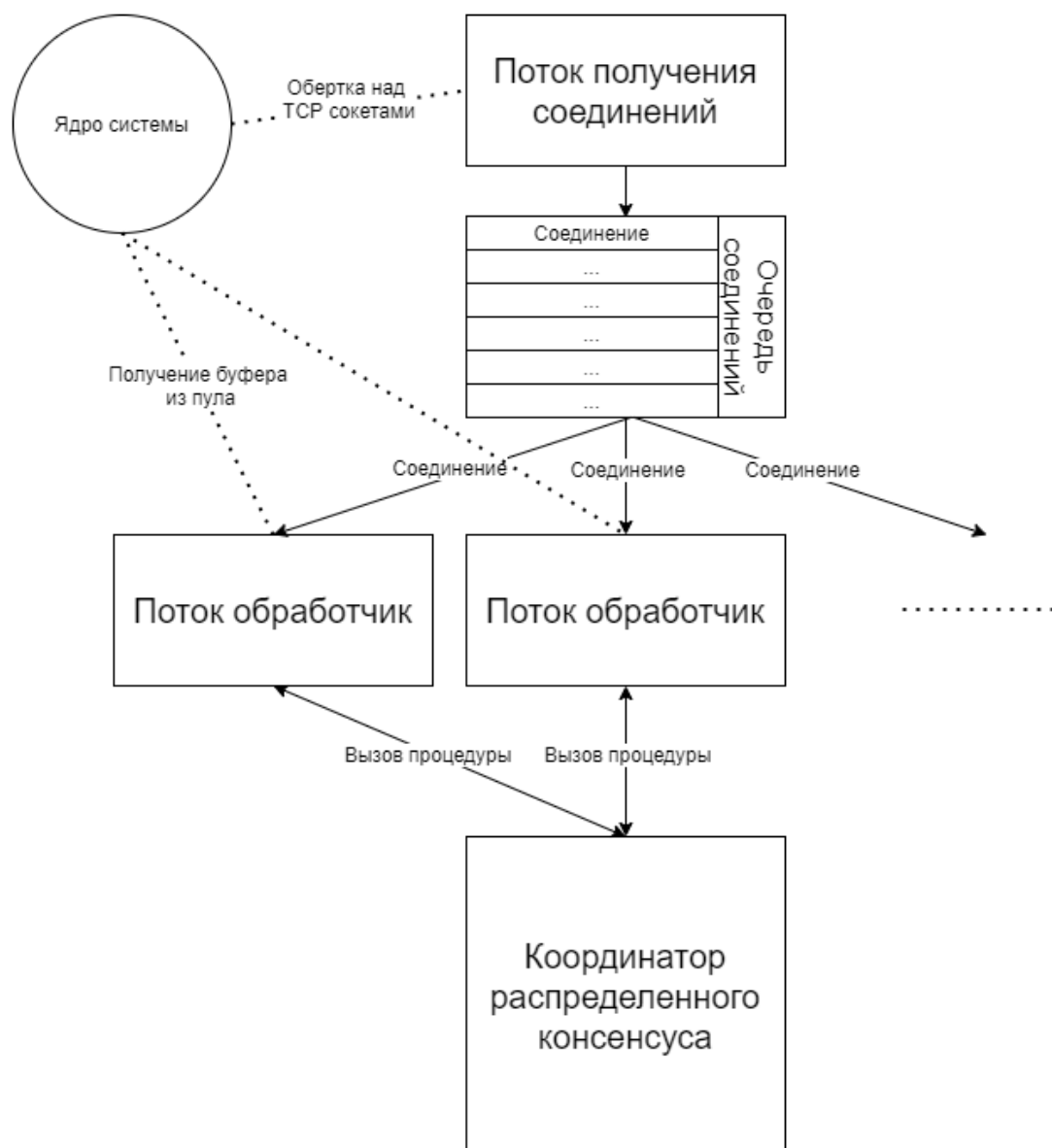


Рисунок 19 – Схема работы RPC сервера

Таблица 8 – Структура RPC сервера

Имя поля	Тип данных	Описание	Положение в иерархии
sSocket	SSocket	Сокет сервера для получения клиентских соединений	Модуль ядра
szAddress	CHAR[16]	Адрес сервера	Интегральный тип

Продолжение таблицы 8

Имя поля	Тип данных	Описание	Положение в иерархии
szPort	CHAR[6]	Порт сервера	Интегральный тип
hThread	HANDLE	Дескриптор потока приема соединений	Тип предоставляемый WINAPI
cbRpcHandler	RPCRequestHandler	Обработчик RPC вызовов	Компонент бизнес-логики
bIsWork	volatile BOOL	Флаг указывающий на состояния сервера (работает/не работает). Запрещен для кэширования на процессоре для изменения значения через атомарные операции	Интегральный тип
dwClientThreadCount	DWORD	Количество потоков обработчиков	Интегральный тип
dwSendTimeout	DWORD	Таймаут на отправку ответа клиенту в миллисекундах	Интегральный тип
dwRecvTimeout	DWORD	Таймаут на получение запроса от клиента в миллисекундах	Интегральный тип
dwMaxConnections	DWORD	Максимальное количество подключений, к серверу	Интегральный тип

Окончание таблицы 8

Имя поля	Тип данных	Описание	Положение в иерархии
hMessageQueue	MESSAGE_QUEUE	Коннектор обеспечивающий очередь между потоком получения соединений и потоками обработчиками	Модуль ядра
hClientThreads	HANDLE[20]	Список дескрипторов потоков обрабатывающих клиентские соединения	Тип предоставляемый WINAPI

Структура клиентского модуля RPC представлена в таблице 9.

Таблица 9 – Структура клиентского RPC модуля

Имя поля	Тип данных	Описание	Положение в иерархии
sSocket	SSocket	Сокет сервера для получения клиентских соединений	Модуль ядра
szAddress	CHAR[16]	Адрес сервера	Интегральный тип
szPort	CHAR[6]	Порт сервера	Интегральный тип
dwSendTimeout	DWORD	Таймаут на отправку ответа клиенту в миллисекундах	Интегральный тип
dwRecvTimeout	DWORD	Таймаут на получение запроса от клиента в миллисекундах	Интегральный тип

Клиентский модуль вызова удаленных процедур устроен куда проще, так как вызов выполняется в том же потоке где и был сделан. Каждый вызов оборачивается обработчиком вызова, который упаковывает аргументы процедуры и метаданные необходимую для идентификации RPC вызова.

2.3.3 Модуль одноранговой сети

Для того чтобы все узлы кластера могли общаться с друг с другом необходимо выстроить одноранговую сеть между ними. Каждый участник в такой сети является равноправным по отношению к другим. В данной реализации общение между участниками сети происходит с помощью механизма вызова удаленных процедур (рисунок 20).

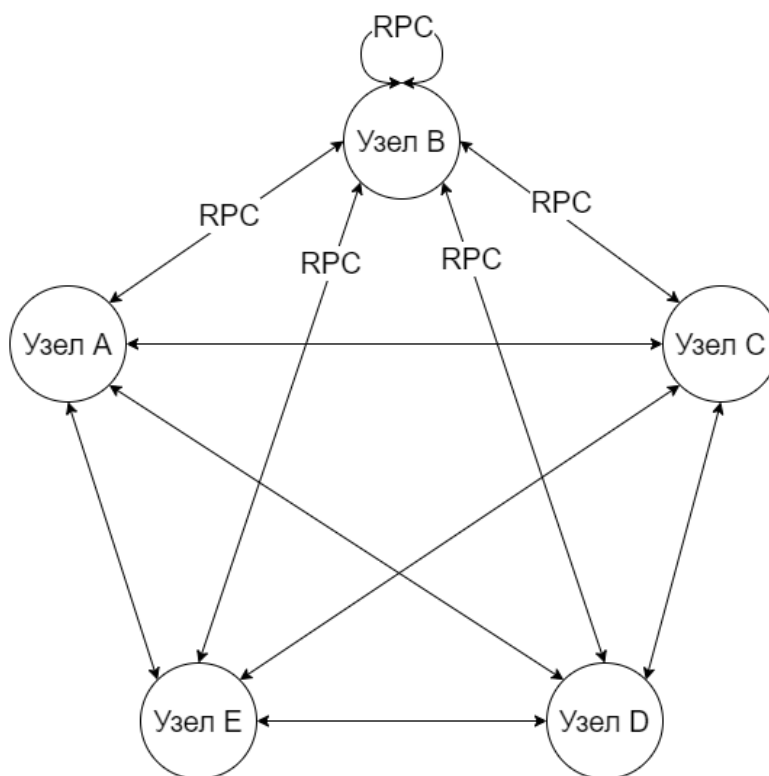


Рисунок 20 – Устройство одноранговой сети

Каждый узел для реализации одноранговой сети создает такое количество потоков, которому соответствует количество узлов в сети, кроме себя. Обычно для распределенного консенсуса с единым лидером и кворумным протоколом принятия записей используется не более 5 узлов,

так как большее количество потребляет много ресурсов и не имеет много смысла, уже при пяти узлах система может терпеть два отказа, чего вполне достаточно.

В системе обозначены роли для каждого узла, поэтому каждый узел одноранговой сети работает в своем режиме. Рассмотрим некоторые возможные случаи состояний узлов в системы.

Когда кластер только начал свою работу все узлы работают в режиме последователей, в этот момент система еще не имеет лидера, таким образом связи между узлами полностью отсутствуют (рисунок 21).



Рисунок 21 – Состояния кластера на начало работы

Как только у одного из узлов истекает таймаут на перевыборы он инициирует голосование и отправляет другим узлам запрос на получение голоса, а другие участники принимают решение о возможности кандидата стать лидером (рисунок 22). Кандидат может не удовлетворить другие узлы

в роли лидера, тогда голосование будут инициировать другие узлы, пока не будет избран лидер.

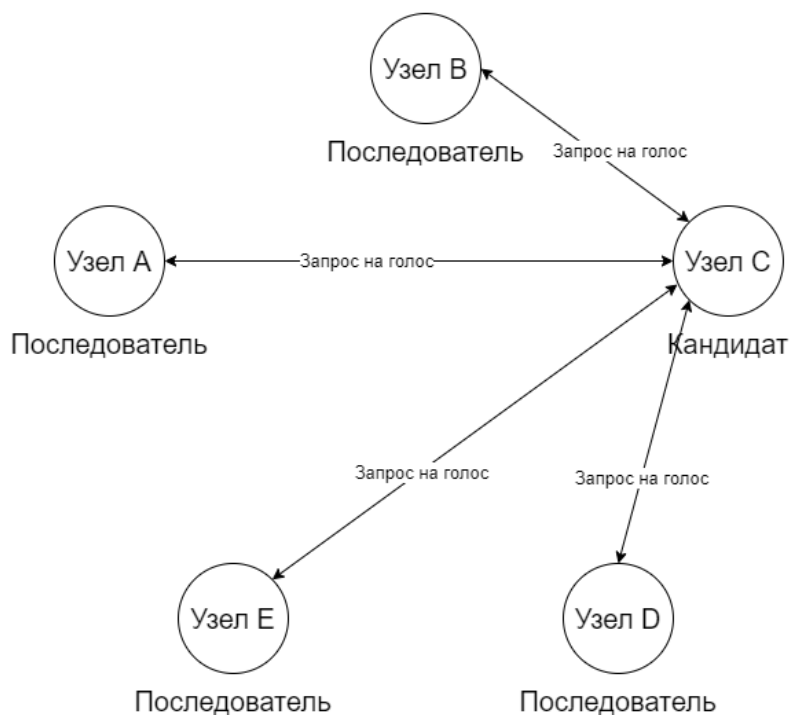


Рисунок 22 – Выборы в кластере

После того как в системы был избран лидер, он начинает реплицировать данные на другие узлы (рисунок 23).

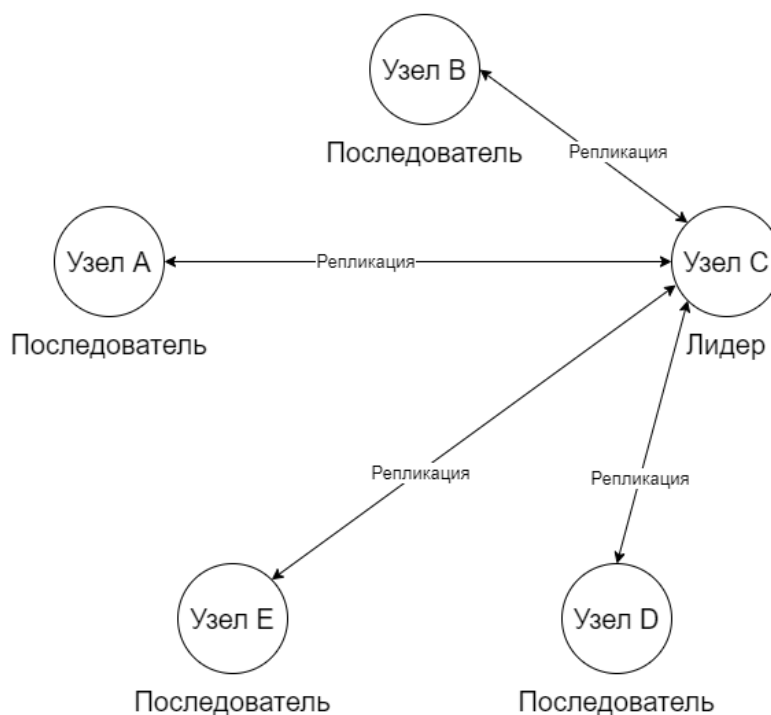


Рисунок 23 – Репликация данных в кластере

Возможны такие ситуации, что текущий лидер С (рисунок 22) может отключиться от сети и потерять соединение с остальными узлами кластера. Тогда он будет бесконечно пытаться реплицировать свои данные на другие узлы. За это время в системе будет избран новый лидер, например, узел Е. Тогда в системе будет существовать два лидера, но один из них будет уже недействительным и лидером он считает себя только сам, когда другие узлы уже принимают записи от другого узла (рисунок 24). Теперь, когда связь узла С с остальными будет восстановлена, он попытается реплицировать свои записи на остальные узлы, но они отвергнут репликацию исходя из эпохи, к которой привязан узел С, так как с каждым переизбором в системе эпоха кластера инкрементируется. После того, как истинный лидер Е начнет свою репликацию на отсталый узел С, он откатится к последователю. Таким образом в кластере разрешается проблема разделения на уровне одноранговой сети.

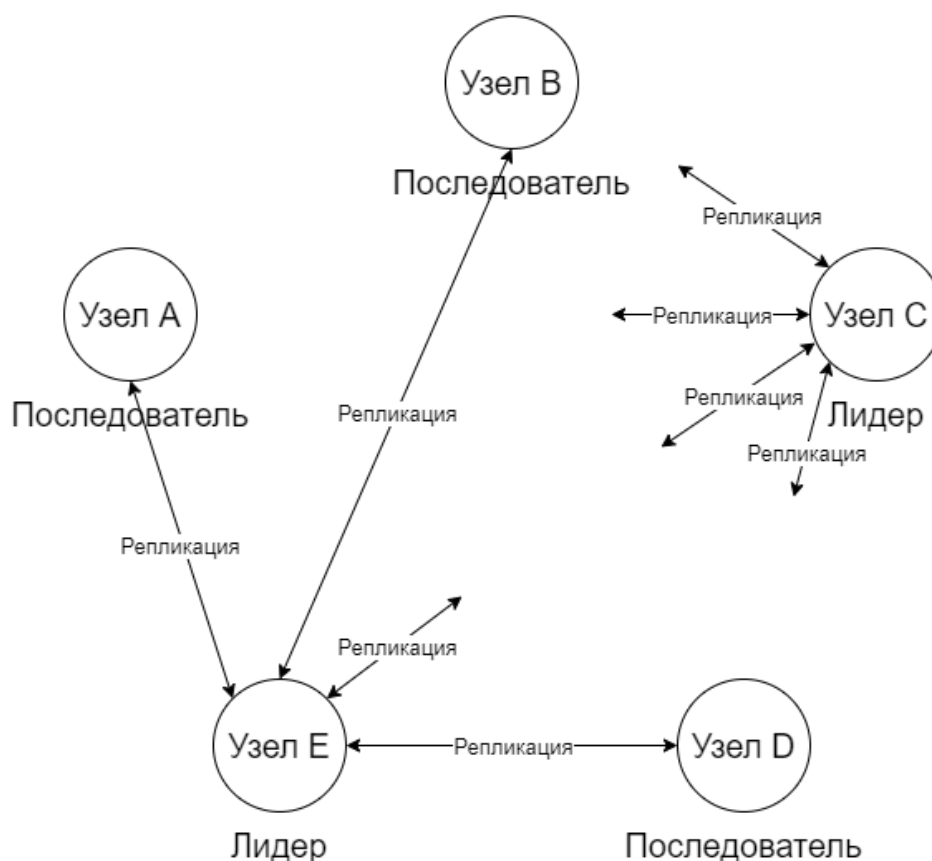


Рисунок 24 – Ситуация существования двух лидеров в системе

Режим работы узла в одноранговой сети устанавливает координатор распределенного консенсуса. Таким образом существует четыре режима работы узлов в кластере:

- сон в режиме последователя;
- репликация записей;
- установка снимков состояния;
- голосование.

Принцип работы и структура узла одноранговой сети представлены на рисунке 25 и таблице 10.

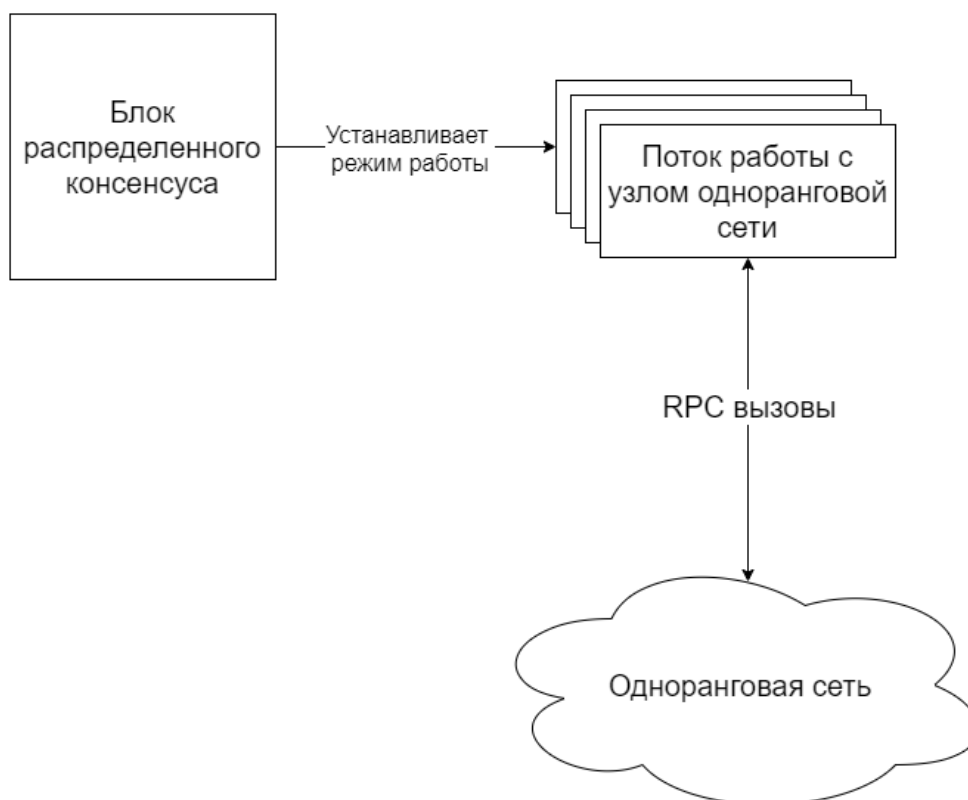


Рисунок 25 – Работа одного узла в одноранговой сети

Таблица 10 – Структура узла в одноранговой сети

Имя поля	Тип данных	Описание	Положение в иерархии
psReplicatedLog	PSReplicatedLog	Реплицированный журнал	Компонент бизнес-логики
dwIndex	DWORD	Индекс узла,	Интегральный тип
szAddress	CHAR[16]	Адрес сервера	Интегральный тип

Продолжение таблицы 10

Имя поля	Тип данных	Описание	Положение в иерархии
szPort	CHAR[6]	Порт сервера	Интегральный тип
dwHearbeatTimeout	DWORD	Таймаут для отправки узлу сердцебиения	Интегральный тип
sRpcClient	SRpcClient	Экземпляр RPC соединения с узлом	Компонент бизнес-логики
hThread	HANDLE	Дескриптор потока работающего с узлом	Тип предоставляемый WINAPI
ulNextIndex	UINT64	Индекс следующей для репликации записи	Интегральный тип
ulMatchIndex	UINT64	Индекс предыдущей успешно реплицированной записи	Интегральный тип

Данный компонент работает в связке с RPC и оба могут быть переиспользованы для построения любой одноранговой сети.

2.3.4 Реплицированный журнал

Реплицированный журнал хранит все поступающие в систему записи и делится на две основных составляющих: журнал, хранимый на энергонезависимом устройстве (HDD, SSD) и журнал хранимый в оперативной памяти.

Журнал, хранящий записи персистентном устройстве необходим для восстановления состояния после сбоев и перезапуска.

Каждый раз, когда в систему поступает новая запись ей назначается индекс и номер эпохи в которую она появилась в системе, это необходимо для поддержания согласованности записей между всеми узлами. С помощью индексов и эпох последователи принимающие реплицируемые записи принимают решение о добавлении записей в свой журнал. Также на основе индексов и эпох производятся выборы в системе.

Журнал расположенный в оперативной памяти необходим для быстрого обращения к записям, так как постоянное чтение записей с персистентного устройства могло бы сильно замедлить работу системы. Все записи в журнале хранятся последовательно и могут быть отброшены из-за несогласованности с журналом лидера при репликации. Данная система гарантирует свойство того, что если у двух записей совпадают индекс и эпоха, то все записи, предшествующие этой, являются согласованными. Журнал помимо записей также хранит информацию о последней принятой конечным автоматом записи и последней зафиксированной записи.

Запись считается зафиксированной если она была реплицирована на более чем $\left\lceil \frac{N}{2} \right\rceil$ узлов, где N – количество узлов в системе. Такой подход называется кворумным протоколом фиксации (A Quorum-Based Commit Protocol) [11].

Кворум – минимальное число голосов, которое должна получить распределенная транзакция, чтобы иметь возможность выполнить эту транзакцию. Таким образом в кворум в данной системе равен $\left\lceil \frac{N}{2} \right\rceil$.

Запись считается принятой конечным автоматом, если она уже зафиксирована и вышла за пределы блока распределенного консенсуса, то есть видна конечному пользователю.

Таким образом реплицированный журнал решает следующие задачи:

- хранение записей;
- принятие решений о добавлении новых записей;
- принятие решение о фиксации записей;

- разрешение несогласованностей;
- восстановление после сбоев.

В данной работе был предложен способ повышения производительности в работе с журналом оперативной памяти, который рассмотрен в 3 главе.

Реализацию журнала было решено разбить на 4 модуля.

Метаинформация журнала – модуль, отвечающий за хранение информации о текущей зафиксированной записи, а также именах файлов, содержащих записи. Записи о именах файлов необходимы для корректного сброса записей информация о которых была зафиксирована в снимках состояния системы.

Хранилище записей – модуль, хранящий непосредственно сами записи, поступающие в систему.

Таблица индексов журнала – модуль, содержащий информацию о всех записях без самих записей. В качестве информации здесь хранятся: индексы, эпохи, размеры, хэш-суммы, а также смещения записей в файле хранящим сами записи.

Структуры реплицированного журнала описаны в таблицах 11 – 13.

Таблица 11 – Структура модуля метаинформации журнала

Имя поля	Тип данных	Описание	Положение в иерархии
ulLastCommitIndex	UINT64	Номер последней зафиксированной системой записи	Интегральный тип
wszRawCommandDataFile	CHAR[260]	Путь к файлу с командами	Интегральный тип
wszIndexTableFile	CHAR[260]	Путь к файлу таблицы индексов	Интегральный тип

Таблица 12 – Структура таблицы индексов журнала

Имя поля	Тип данных	Описание	Положение в иерархии
ulTerm	UINT64	Эпоха записи	Интегральный тип
ulIndex	UINT64	Индекс записи	Интегральный тип
ulOffset	UINT64	Смещение в файле с записями	Интегральный тип
dwCommandSize	DWORD	Размер записи	Интегральный тип
dwHash	DWORD	Хэш-сумма записи	Интегральный тип

Таблица 13 – Структура персистентного журнала

Имя поля	Тип данных	Описание	Положение в иерархии
hDataset	HANDLE	Дескриптор файла содержащего записи	Тип предоставляемый WINAPI
hOffsetTable	HANDLE	Дескриптор файла содержащего таблицу индексов	Тип предоставляемый WINAPI
ulDatasetSize	UINT64	Размер всех записей	Интегральный тип
ulTableSize	UINT64	Размер таблицы индексов	Интегральный тип
szLogFileName	CHAR[260]	Путь к файлу с записями	Интегральный тип
hSrwLock	SRWLOCK	Тонкая блокировка	Тип предоставляемый WINAPI

В 3 главе описан способ модификации реплицированного журнала в оперативной памяти, поэтому его реализация здесь не описана.

3 Повышение производительности распределенного хранилища

Отказоустойчивые и высокопроизводительные системы являются важным компонентом любого крупного современного сервиса. Для обеспечения отказоустойчивости системы могут поддерживать несколько реплик, которые имеют одинаковые данные в одном и том же порядке. Такое решение приводит к возникновению различных проблем при реализации, например, поддержание согласованности между репликами после сбоя системы. Протоколы консенсуса на основе реплицированного конечного автомата являются наиболее подходящими кандидатами для решения этих проблем, но поддержание согласованности в таких системах требует больших затрат по исполнительному времени. Особенно характерно проявление таких затрат на журналах, расположенных в оперативной памяти, и имеющих большое количество записей.

Также на принятые протоколом консенсуса записи после обработки конечным автоматом добавляются в согласованное хранилище, которое является независимым от протокола.

3.1 Модификация реплицированного лога Raft

Обычно для реализации журнала используют интрузивный список, сложность операций поиска записей и их несогласованностей для которого имеет линейную сложность. В этой работе предлагается структура данных, которая сохраняет полезные свойства списка для поставленной задачи и при этом позволяет получить более высокопроизводительный по времени журнал в оперативной памяти для протокола консенсуса Raft.

За основу реализации структуры данных был взят список с пропусками (Skip List) [12]. Список с пропусками – вероятностная структура данных, основанная на связном списке и позволяющая выполнять операции вставки, поиска и удаления в среднем за $O(\log_2 n)$.

Список с пропусками состоит из множества уровней, на каждом из которых расположен односвязный список (рисунок 26).

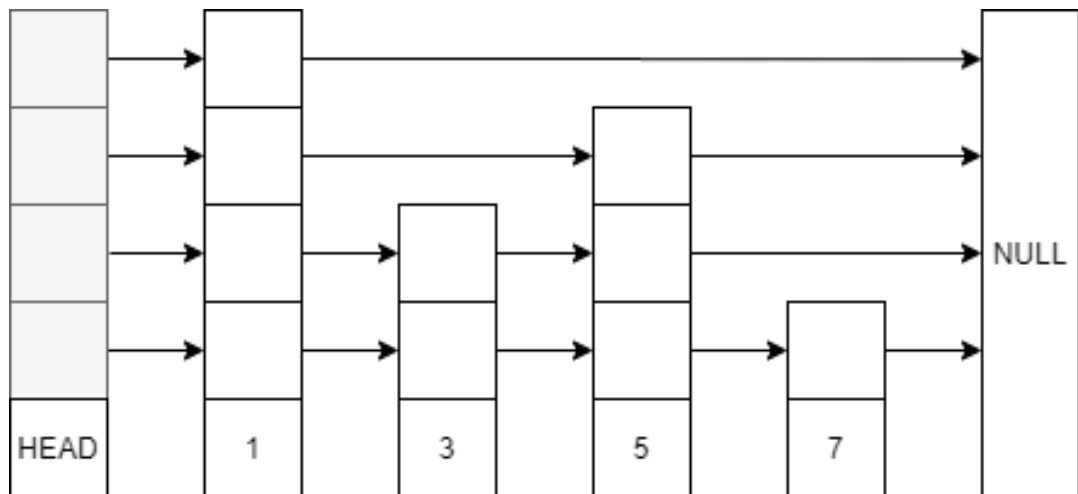


Рисунок 26 – Организация списка с пропусками

Использование деревьев для журнала Raft невозможно, так как они хранят данные не последовательно и никак не позволяют выполнять операции сразу над множеством элементов. Тогда как список с пропусками хранит элементы последовательно.

Основными операциями над журналом Raft являются:

- поиск записи по индексу;
- вставка n записей в конец;
- удаление n записей до индекса;
- удаление n записей после индекса.

Классическая реализация списка с пропусками удовлетворяет только поиску записи по индексу и не имеет операций для работы над n элементами.

Для решения этой проблемы был построен кольцевой двусторонний список с пропусками (RSL - Ring Skip List), где в основе каждого уровня лежит двусторонний кольцевой список. Такая реализация позволила выполнять операций сразу над множеством элементов структуры данных (рисунок 27).

Найденные открытие решения используют обычные массивы или списки в качестве структуры для организации реплицированного журнала в распределенных системах.

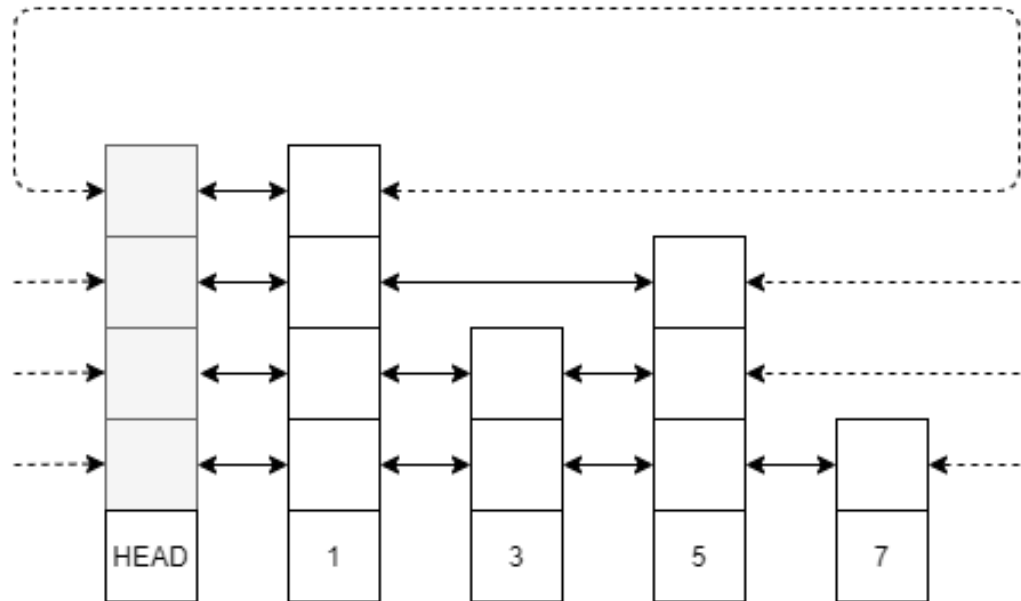


Рисунок 27 – Организация предлагаемого списка с пропусками

Основная цель использования двухсвязных списков на каждом уровне заключается в том, чтобы можно было быстро удалять элементы до и после индекса.

3.1.1 Вставка множества записей в конец лога

Для вставки n записей в конец структуры необходимо сначала заполнить временный список с пропусками. Вставлять записи во временный список с пропусками необходимо в конец, так как мы уверены, основываясь на политике Raft, что лидер реплицирует записи последовательно из своего журнала.

При создании новых узлов для вставки во временный список с пропусками необходимо использовать случайное число задающее высоту узла из той же последовательности, которое использовалось при создании последнего узла в основном списке.

Идея использования временного списка с пропусками состоит в том, что записи поступают по RPC (remote procedure call) в упакованном виде, а упаковка и распаковка может происходить на уровне реализации структуры.

После проверки записей на согласованность добавление записей из временного списка с пропусками происходит по алгоритму, приведенному в листинге 1.

Листинг 1 – Алгоритм вставки множества записей в конец лога

```
// Определение новой максимальной высоты для структуры
log.height = max(log.height, temp.height)
height = 0
// Цикл по высоте добавляемого списка
while(height < temp.height)
    // Голова исходного списка
    baseHead = log.head[height]
    // Последний элемент исходного списка
    successor = baseHead.prev

    // Первый элемент добавляемого списка
    tempFirst = temp.head[height].next
    // Последний элемент добавляемого списка

    tempLast = temp.head[height].prev

    successor.next = tempFirst
    baseHead.prev = tempEnd

    tempFirst.prev = successor
    tempEnd.next = baseHead

    height++
```

Такой алгоритм позволяет получить асимптотическую сложность для добавления записей в конец аналогичную обычному односвязному списку и не имеет преимуществ. Основное назначение этого алгоритма в том, чтобы не получить ухудшение в производительности, так как классический список с пропусками делает эту операцию с логарифмической сложностью для каждого элемента, вместо константной как у обычного списка.

3.1.2 Удаление записей до и после индекса в логе

Операции удаления n записей до и после индекса похожи, поэтому рассмотрим удаление до индекса.

Для удаления записей до индекса необходимо найти эту запись. Поиск должен осуществляться с конца списка, это нужно для того, чтобы заменять указатели назад на узлах с высотой выше искомого. Таким образом, каждый

раз, когда во время поиска встречается узел с индексом меньше текущего, происходит замена указателя на предыдущий элемент у узла, из которого попали в текущий, на головной элемент списка. Указатель на последующий элемент «головы» списка корректируется, с целью указания на исходный. Тем самым убираем из списка элементы, высота которых превосходит искомый. Когда искомый узел найден и по пути следствия все ненужные ссылки удалены, необходимо у данного узла и в «голове» структуры по всей высоте узла изменить указатель на следующий и предыдущий узлы. Для «головы» по этой высоте все указатели на следующий элемент будут ссылаться на узел, следующий после найденного, а для следующего узла все предыдущие будут указывать на «голову» (рисунок 28). Такое преобразование становится возможным из-за того, что каждый последующий индекс отличается только на единицу. Таким образом, поиск элемента происходит за $O(\log_2 n)$, а удаление за $O(1)$.

Алгоритм удаления записей после индекса аналогичен удалению записей до индекса, но обход списка необходимо осуществлять слева-направо. Таким образом, он выглядит полностью симметрично удалению до индекса и представлен в листинге 2.

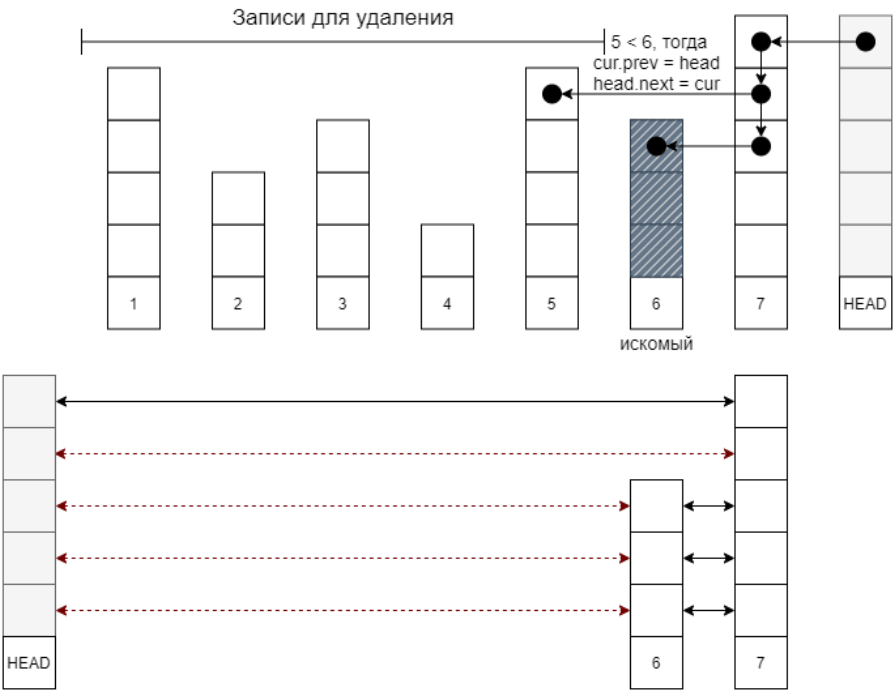


Рисунок 28 – Схема алгоритма удаления записей до индекса

Листинг 2 – Алгоритм удаления записей до индекса

```

height = log.height - 1
curNode = log.head[height]
lastNode = log.head[height]

// Спуск по высотам
while (height >= 0)
    curNode = curNode.prev

    // Продвижение справа-налево по высоте
    while (curNode != lastNode)
        // Найденный элемент будет удален
        if (curNode.index < index)
            // Сохраняем узел
            lastNode = curNode.link[height]
            break;

        // Искомый узел найден
        else if (curNode.index == index)
            // Удаление всех узлов до него
            for (i = 0; i <= height; i++)
                curNode.link[i].prev.next = log.head[i]
                log.head[i].next = curNode.link[i].next

            if (listIsEmpty(log.head[i]))
                log.height--

        curNode = curNode.prev

    // Найден узел который будет удален
    curNode.next.prev = log.head[height]

    // Замыкаем связи с головным узлом
    log.head[height].next = curNode.next
    curNode = lastNode.next

    // Спуск по высоте
    height--
    curNode = curNode.link[height]
    lastNode = lastNode.link[height]

```

Такой результат для удаления множества записей до и после индекса был получен благодаря использованию двухсвязных списков и закольцованности структуры.

3.1.3 Результаты применения структуры

С использованием предлагаемой структуры -кольцевого списка с пропусками получилось улучшить асимптотическую сложность

выполнения некоторых операций Raft над журналом. Сравнение приведено в таблице 14.

Таблица 14 – Сравнение асимптотик структур для операций Raft

Действие Raft над журналом	Односвязный список	Кольцевой список с пропусками
Поиск записи для репликации	$O(n)$	$O(\log_2 n)$
Поиск записи для принятия реплицированным автоматом	$O(n)$	$O(\log_2 n)$
Добавление новых записей при репликации	$O(n)$	$O(n)$
Отбрасывание несогласованных записей	$O(n)$	$O(\log_2 n)$
Отбрасывание принятых записей после создания снимка состояния	$O(n)$	$O(\log_2 n)$
Добавление новой записи в конец	$O(1)$	$O(n)$

Согласно рисункам 29 – 31, предложенная реализация имеет преимущество, над обычным списком в качестве журнала Raft.

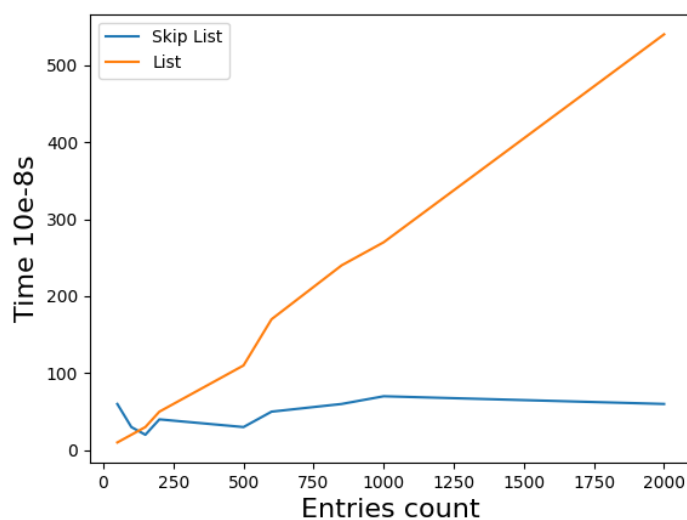


Рисунок 29 – График времени исполнения удаления записей до индекса

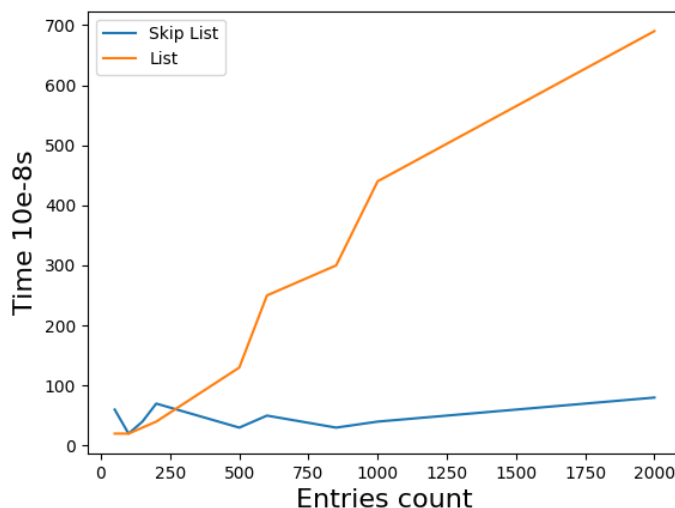


Рисунок 30 – График времени выполнения удаления записей после индекса

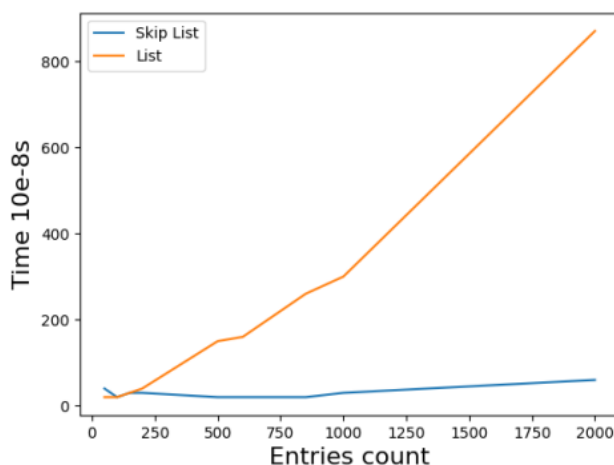


Рисунок 31 – График зависимости времени от количества записей на поиск

Таким образом, скорость выполнения следующих операций над журналом возрастает:

- принятие записей;
- репликация записей;
- решение несогласованности в журналах;
- отбрасывание записей при создании снимка состояния.

Данную реализацию можно использовать, если Raft принимает команды такого размера, чтобы можно было расположить их в большом

количестве в оперативной памяти, иначе прироста производительности в скорости работы не будет, а, наоборот, производительность может упасть.

Для того, чтобы повысить скорость освобождения памяти сразу для множества записей используется кастомный аллокатор, который позволяет выделять память сразу под несколько узлов списка. Также аллокатор обладает свойством последовательности, что обеспечивает возможность сброса данных на диск сразу большими чанками. Когда из аллоцированного чанка была удалена запись, аллокатор сдвигает смещение назад для заданного узла.

Также такой подход к организации журнала требует больше памяти, в среднем этот размер может требовать объема памяти V для n элементов в соответствии с формулой:

$$V = 2n - 2.$$

Докажем количество требуемого пространства в памяти для структуры. Пусть список с пропусками имеет n элементов, тогда по условию сбалансированности высота h должна быть равна:

$$h = \lceil \log_2 n \rceil.$$

Пусть p – вероятность, с которой элемент, находящийся на высоте h_i может присутствовать на высоте h_{i+1} , а на нижнем уровне элемент должен быть с вероятностью 1, тогда на каждом следующем уровне количество элементов в среднем должно быть:

$$f(h_{i+1}) = f(h_i) \cdot p,$$

$$f(h_1) = n.$$

Так как известно начальное значение можно вывести элементарную функцию вычисления количества элементов на высоте из рекуррентной функции:

$$f(h_{i+1}) = n \cdot p^i, \text{ где } f(h_i) = n \cdot p^{i-1}.$$

Подсчитаем количество элементов на всех высотах:

$$\sum_{i=1}^{\lfloor \log_2 n \rfloor} n \cdot p^{i-1} = n \cdot p^0 + n \cdot p^1 + \dots + n \cdot p^h.$$

Для получения логарифмической сложности операций над структурой необходимо использовать $p = \frac{1}{2}$, подставим его в предыдущую формулу:

$$\sum_{i=1}^{\lfloor \log_2 n \rfloor} \frac{n}{2^{i-1}} = 2 \cdot \sum_{i=1}^{\lfloor \log_2 n \rfloor} \frac{n}{2^i}.$$

Полученное выражение является суммой геометрической прогрессии S , вычислим ее:

$$\begin{aligned} S &= \frac{n \cdot \left(\frac{1}{2^h} - 1 \right)}{\frac{1}{2} - 1} = -2n \cdot \left(\frac{1}{2^{\log_2 n}} - 1 \right) = \\ &= -2n \cdot \left(\frac{1}{n} - 1 \right) = 2n - 2. \end{aligned}$$

Такое количество узлов будет использоваться структурой, теперь посчитаем, сколько указателей она будет требовать. Каждый узел имеет два указателя на следующий и предыдущий узлы, тогда получим следующее число указателей k :

$$k = 4n - 4.$$

Для подсчета количества требуемого количества оперативной памяти m необходимо умножить количество указателей на размер, занимаемый одним указателем ptr :

$$m = k * ptr = (4n - 4) * ptr.$$

Оценка производилась с учетом усреднения вероятностного распределения, используемого структурой данных.

Предложенная структура данных дает прирост в производительности для задач Raft, требующих операций над множеством записей, но потребляет больше памяти.

3.2 Модификация структуры хранения данных принятым конечным автоматом

Распараллеливание обеспечивает большой прирост производительности в работе программ, но и этот подход имеет очевидный недостаток: части кода, требующие синхронизации при работе с разделяемыми данными, не могут выполняться одновременно. Эффект, который оказывает сокращение времени исполнения за счет распараллеливания описывается универсальным законом масштабируемости (рисунок 32):

$$S(n) = \frac{N}{1 + \alpha(N - 1) + \beta N(N - 1)},$$

где N – количество потоков;

α – процент кода требующего синхронизации;

β – когерентность (задержки, чтобы данные стали согласованными).

В рамках работы был реализован конкурентный список с пропусками, который является свободным от блокировок при взаимодействии с ним одновременно нескольких потоков.

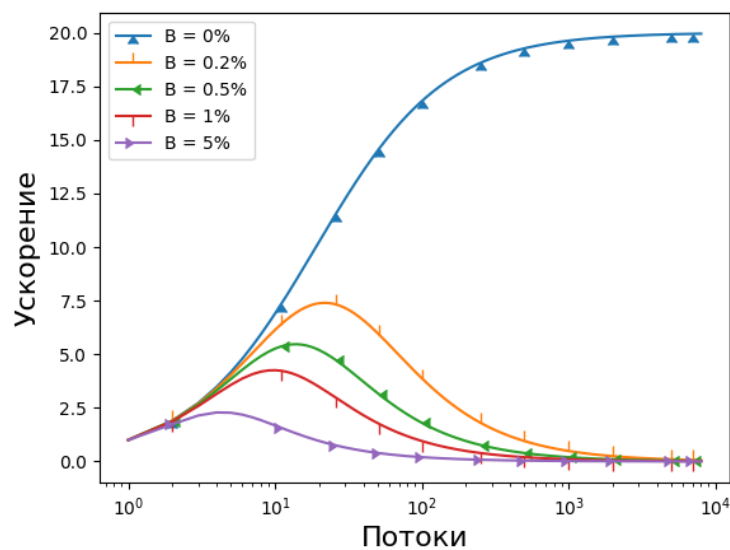


Рисунок 32 – Закон масштабируемости параллельных вычислений

Как видно из рисунка 32 даже небольшое значение когерентности сильно снижает производительность при росте числа потоков, поэтому необходимо понизить затраты на когерентность при параллельном доступе к памяти.

3.2.1 Lock-Free структуры данных и CAS операция

Для понижения когерентности используются lock-free структуры данных. Это такие структуры данных, реализация которых позволяет взаимодействовать с критическими секциями множества потоков без взаимного исключения, так чтобы не было утечек памяти, порчи данных и неопределенного поведения программы.

Для достижения такого рода структур данных используются атомарные операции «compare and swap». CAS – атомарная инструкция, которая выполняет сравнение с обменом [13]. Эта инструкция принимает три аргумента: ячейку памяти, значение для сравнения и новое значение. Новое значение будет записано в ячейку памяти только в случае если текущее значение равно значению, переданному для сравнения. Для разных архитектур операция может иметь различные реализации.

3.2.2 Проблема распараллеливания красно-черных деревьев

Красно-черные деревья являются одной из самых популярных структур данных для хранения в оперативной памяти компьютера это обусловлено логарифмической сложностью операций над ним. Но вот при работе со структурой в параллельной среде, возникают проблемы описанные в первой части работы. Из-за сложного механизма перебалансировки дерева нет возможности избежать взаимного исключения при работе с ним. Поэтому при росте числа потоков система может понизить эффективность работы [14].

Такого недостатка лишена структура данных «Список с пропусками». Эта структура позволяет добиться логарифмической сложности и при этом работать без взаимных исключений [15].

3.2.3 Результаты применения структуры

В ходе работы были написаны тесты производительности для различных структур на различных операциях и проведен их сравнительный анализ. Были реализованы следующие структуры:

- список с пропусками;
- lock-free список с пропусками;
- красно-черное дерево.

Над всеми реализациями были произведены тесты производительности. Результаты тестирования структур в одном потоке по времени представлены на рисунках 33 – 35.

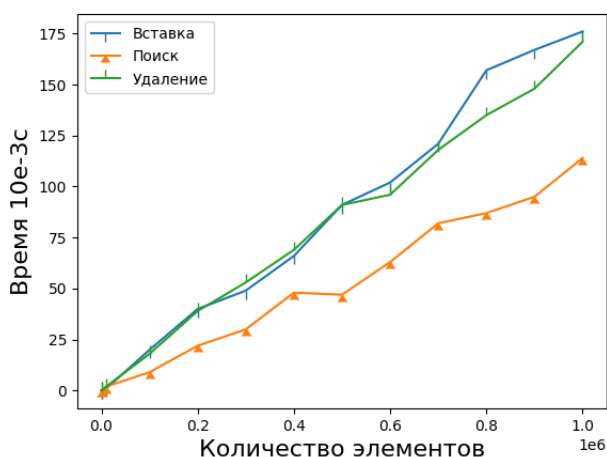


Рисунок 33 – График производительности красно-черных деревьев

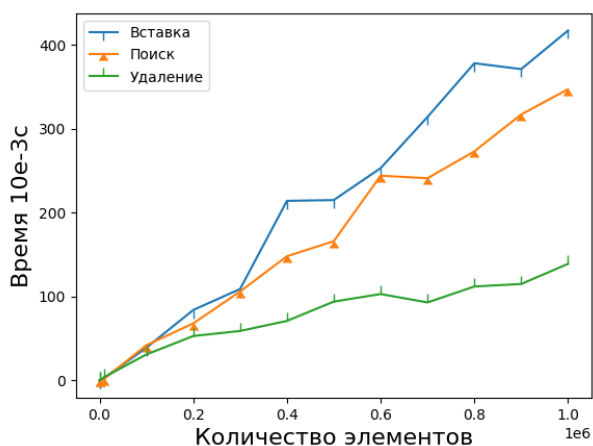


Рисунок 34 – График производительности списка с пропусками

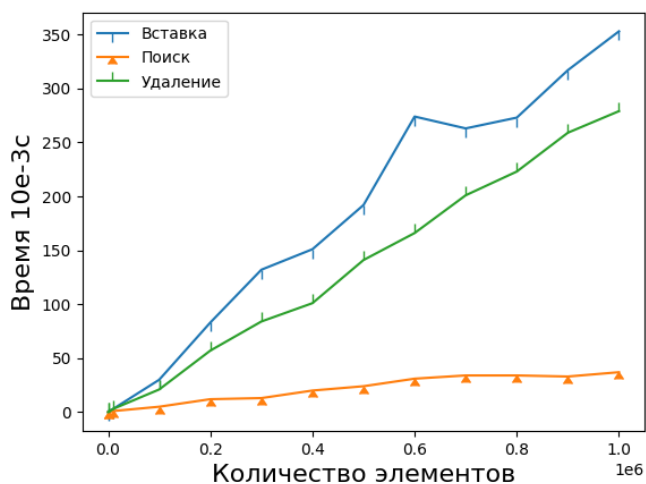


Рисунок 35 – График производительности свободного от блокировок списка с пропусками

На графиках однопоточного сравнения видно, что красно-черное дерево превосходит остальные по времени вставки и удаления в 2 раза, но поиск осуществляется быстрее в свободном от блокировок списке с пропусками, он в 2,5 раза быстрее, чем в красно-черном дереве.

На рисунках 36 – 38 приведены сравнения производительности структур для различных операций на множестве потоков (все тесты проводились для $2 * 10^6$ элементов).

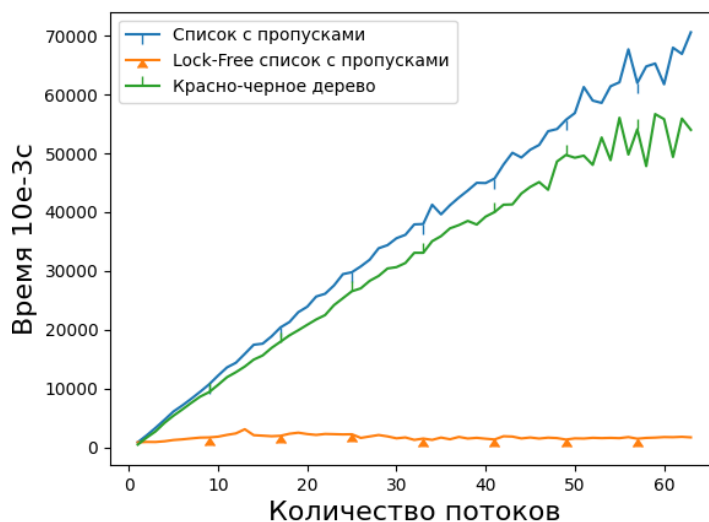


Рисунок 36 – График производительности структур для операции вставки на разном количестве потоков

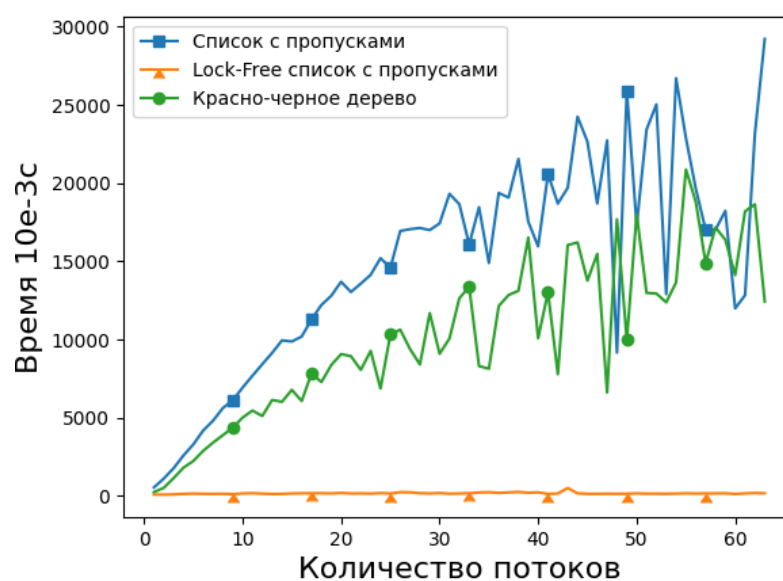


Рисунок 37 – График производительности структур для операции поиска на разном количестве потоков

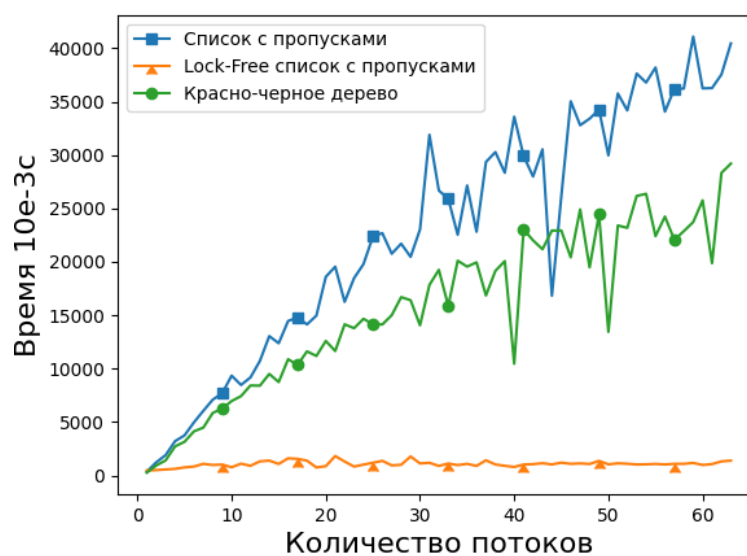


Рисунок 38 – График производительности структур для операции удаления на разном количестве потоков

Из графиков сравнения структур на множестве потоков видно, что свободный от блокировок список с пропусками. превосходит остальные во много раз на каждой из операций.

4 Формализация системы

Данный раздел посвящен доказательству работоспособности и гарантий заявленных в анализе требований к ней.

4.1 Существование одного лидера

Реализация системы обеспечивает наличие одного лидера в системе, докажем, что в каждой эпохе не может существовать более одного лидера.

Чтобы стать лидером узел должен получить голоса более чем от половины узлов согласно кворуму, каждый избиратель может проголосовать только один раз в течении одной эпохи:

$$\forall a, b \in \text{elections}: a.\text{term} = b.\text{term} \Rightarrow a.\text{leader} = b.\text{leader}.$$

Рассмотрим два голосования a и b в кластере после сбоя лидера, значит их эпохи равны:

$$a.\text{term} = b.\text{term}.$$

Поскольку для победы необходимо собрать кворум, то должно выполняться следующее:

$$a.\text{votes} = b.\text{votes}.$$

Пусть избираемый узел является произвольным из состава кластера, такой узел должен существовать так как любые два кворума перекрываются.

Когда избиратель проголосует за $a.\text{leader}$ в $a.\text{term}$, он не сможет отдать голос за другой узел в $b.\text{term}$, так как за эпоху возможно проголосовать лишь однажды.

Из выше сказанного следует, что $a.\text{leader}$ равен $b.\text{leader}$, таким образом в одну эпоху может существовать только один лидер.

4.2 Выборы лидера

Наиболее распространенный случай выборов лидера – это отсутствие разделения голосов, и в этом разделе анализируется вероятность выбора определенного узла в качестве лидера, а также оценка времени голосования.

Для формализации процесса выборов в кластере определим инварианты в таблице 15.

Таблица 15 – Инварианты для исследования скорости выборов лидера

№	Инварианта	Тип	Описание
1	s	Конфигурационный	Количество доступных узлов
2	n	Конфигурационный	Общее число узлов кластера
3	c	Конфигурационный	Количество разделений в сети
4	l	Время	Средняя задержка сети на запрос и ответ
5	W	Случайное время	Время на принятие решения о голосовании
6	T_i	Случайное время	Таймаут узла i
7	M_s	Случайное время	Минимальный таймаут
8	E	Случайное время	Время завершения голосования

Предположим, что каждый сервер выбирает время ожидания случайно из равномерного распределения на отрезке $[0; 1]$. Пусть $T_0, \dots, T_s$ – случайные величины, представляющие время ожидания каждого из s серверов. Тогда определим M_s как минимальный таймаут из всех серверов.

Функцией распределения в данном случае определяет, что M_s не больше определенного времени t , а это эквивалентно единице минус вероятность того, что время ожидания всех серверов истечет после t :

$$\begin{aligned}
 P(M_s \leq t) &= 1 - P(M_s > t) = \\
 &= 1 - \prod_{i=1}^s P(T_i > t) \\
 &= 1 - \prod_{i=1}^s (1 - t) \\
 &= 1 - (1 - t)^s.
 \end{aligned}$$

Например, рассмотрим кластер с пятью узлами, в котором отказал лидер. Вероятность того, что узел с самым ранним таймаутом инициирует выборы в первой четверти всех таумаутов на выборы равна:

$$1 - \left(1 - \frac{1}{4}\right)^4 \approx 0.68.$$

Вероятность выбора в качестве лидера узла с таймаутом M_s высока даже для первой четверти перевыборов, таким образом будет отталкиваться от того, что выбран будет он.

Общее время голосования складывается из задержки сети, времени на принятие решения. Таким образом среднее время голосования будет равно:

$$E = W + l + M_s.$$

Задержка сети имеет большое значение в подобных системах, поэтому необходимо правильно выбирать таймаут на перевыборы, иначе лидер может сменяться даже без сбоя лидера.

4.3 Согласованность журнала

Для поддержания линейризуемости системы необходимо разрешать возникающие рассогласованности в реплицированных журналах. Система опирается на тот факт, что если две записи имеют одинаковый индекс и эпоху, то все им предшествующие записи согласованы (утверждение 2). На основе такого положения система понимает, когда необходимо отбросить записи в журнале.

Подряд согласованные записи в двух журналах будем называть префиксом журнала. Таким образом, если p – префикс журнала $l \in L$, где L – множество всех журналов, тогда инварианта:

$$L' \in L \cup \{p\}.$$

Докажем данное утверждение с помощью математической индукции.

Изначально журналы на всех узлах пусты $L = \emptyset$ поэтому инвариант L' выполняется заведомо, до поступления записей в систему.

Рассмотрим возможные индуктивные шаги:

- а) лидер добавляет одну записи по запросу клиента:
 - 1) по предположению индукции $l_{leader} \in L$;
 - 2) пара $\langle \text{индекс, эпоха} \rangle$ для новой записи не может существовать в любом другом журнале, поскольку есть только один лидер (утверждение 1);

- 3) тогда инварианта $L' \in L \cup \{l_{leader}\}$ выполняется;
- б) последователь удаляет одну запись:
 - 1) инвариант сохраняется так как $l'_{follower}$ является префиксом $l_{follower}$;
- в) последователь добавляет одну запись:
 - 1) пусть журнал лидера l_{leader} не пуст, и он реплицирует записи последователям;
 - 2) $l_{follower} \in \emptyset$ до принятия записей, отправленных лидером, тогда $l_{leader} \equiv p$ для этих двух журналов, значит инвариант сохраняется.

Таким образом все утверждения построенные на базе индукции верны, значит журнал разрешает все виды несогласованностей.

5 Тестирование системы

Модульное тестирование (юнит-тестирование) позволяет проверить на корректность каждый модуль исходного кода программного продукта отдельно. Юнит-тесты пишутся на этапе кодирования приложения для избавления от дефектов и ошибок в будущем.

Так как модульные тесты прописаны практически на каждый файл приложения, составим таблицу, в которой опишем каждый тест и полученный результат. Описания юнит-тестов показаны в таблице 15.

Все тесты представляют из себя проверку свойств компонентов и операций над ними и, если поведение не удовлетворяет желаемому система вызывает отладочное прерывание.

Таблица 15 – Описание юнит-тестов

№	Название тестового метода	Описание теста	Результат
1	TimerSetTest	Тестирование создания, установки и удаления таймера после окончания времени срабатывания	Успешно
2	TimerCancelTest	Тестирование создания, установки, сброса и удаления таймера	Успешно
3	MCTimeReceiveSendTest	Тестирование отправки данных по каналу в другой поток	Успешно
4	MCTimeSendReceiveTest	Тестирование принятия данных по каналу из другого потока	Успешно
5	MCTimeMultiThreadTest	Тестирование принятия данных по каналу в одном потоке, отправляются данные в канал из 4 других потоков	Успешно
6	RpcTest	Тестирование вызова удаленных процедур	Успешно
7	RpcMultiThreadTest	Тестирование вызова удаленных процедур в нескольких потоках	Успешно
8	SocketWrapperTest	Тестирование функций отправки и принятия	Успешно
9	SocketWrapperTime outTest	Тестирование работы таймаутов на сокета из обертки	Успешно

Продолжение таблицы 15

№	Название тестового метода	Описание теста	Результат
10	SocketWrapperWithChecksumTest	Тестирование работы отправки и принятия данных по сокету с последующей проверкой хэш-суммы	Успешно
11	BloomFilterTest	Тестирование работы фильтра Блума	Успешно
12	BloomFilterMultiThreadTest	Тестирование работы фильтра Блума на нескольких потоках	Успешно
13	BitsetSetBitTest	Тестирование установки бита в битовое множество	Успешно
14	BitsetSet8MaskTest	Тестирование установки 8 бит в битовое множество	Успешно
15	BitsetSet16MaskTest	Тестирование установки 16 бит в битовое множество	Успешно
16	BitsetSet64Mask_x86Test	Тестирование установки 64 бит в битовое множество, оптимизированного под архитектуру x86	Успешно
17	BitsetSet64Mask_x64Test	Тестирование установки 64 бит в битовое множество, оптимизированного под архитектуру x64	Успешно
18	BitsetSet128Mask_x86Test	Тестирование установки 128 бит в битовое множество, оптимизированного под архитектуру x86	Успешно
19	BitsetSet128Mask_x64Test	Тестирование установки 128 бит в битовое множество, оптимизированного под архитектуру x64	Успешно
20	BitsetCheckBitTest	Тестирование функции принадлежности бита в битовом множество	Успешно
21	BitsetCheck8MaskTest	Тестирование функции принадлежности 8 бит в битовое множество	Успешно
22	BitsetCheck64Mask_x86Test	Тестирование функции принадлежности 64 бит оптимизированной под архитектуру x86	Успешно
23	BitsetCheck64Mask_x64Test	Тестирование функции принадлежности 64 бит оптимизированной под архитектуру x64	Успешно

Продолжение таблицы 15

№	Название тестового метода	Описание теста	Результат
24	BitsetCheck128Mask_x86Test	Тестирование функции принадлежности 128 бит оптимизированной под архитектуру x64	Успешно
25	BitsetCheck128Mask_x64Test	Тестирование функции принадлежности 128 бит оптимизированной под архитектуру x64	Успешно
26	BitsetCheck128MaskAligned_x86Test	Тестирование функции принадлежности 64 бит оптимизированной под архитектуру x86 и выровненного по 128 бит	Успешно
27	BitsetCheck128MaskAligned_x64Test	Тестирование функции принадлежности 64 бит оптимизированной под архитектуру x64 и выровненного по 128 бит	Успешно
28	LockFreeSkipListAddTest	Тестирование добавления узлов в конкурентный список с пропусками	Успешно
29	LockFreeSkipListFindTest	Тестирование поиска узлов в конкурентном списке с пропусками	Успешно
30	LockFreeSkipListMultiThreadAddTest	Тестирование добавления узлов в конкурентный список с пропусками из нескольких потоков	Успешно
31	LockFreeSkipListMultiThreadFind	Тестирование поиска узлов в конкурентном списке с пропусками из нескольких потоков	Успешно
32	LockFreeSkipListMultiThreadAddAndFind	Тестирование одновременного поиска и добавления узлов в конкурентный список с пропусками из нескольких потоков	Успешно
33	LockFreeSkipListResetTest	Тестирование переустановки значений по ключу в конкурентном списке с пропусками	Успешно
34	SkipListAddTest	Тестирование добавления значений в список с пропусками	Успешно
35	SkipListSetTest	Тестирование установки значений в список с пропусками	Успешно
36	SkipListFindTest	Тестирование поиска значений в списке с пропусками	Успешно
37	SkipListPrintTest	Тестирование прохода и вывода в списке с пропусками	Успешно
38	SkipListRemoveTest	Тестирование удаления узлов в списке с пропусками	Успешно

Окончание таблицы 15

№	Название тестового метода	Описание теста	Результат
39	LogEnumFnvTest	Тестирование соответствий конфигураций с уровнями логирования через FNV-1A	Успешно
40	ParseLogTest_1	Тестирование парсинга конфигурации лога с верными данными	Успешно
41	ParseLogTest_2	Тестирование парсинга конфигурации лога с неверными данными	Успешно
42	LogWriteTest_1	Тестирование записи в лог	Успешно
43	LogMultiThreadWriteTest_1	Тестирование записи в лог из нескольких потоков	Успешно

Все юнит-тесты были успешно пройдены. Это означает, что все компоненты приложения работают корректно.

5.1 Метрики кода

Метрикой программного кода называется количественная мера, позволяющая определить, в какой степени система, компоненты или процесс, обладают заданным атрибутом.

Для программного кода, представленного в листинге 3 рассчитаем метрики программного кода. В листинге представлена функция создания асинхронного таймера.

Листинг 3 – Код метода создания таймера канала

```

BOOL
CreateChannelTimer(
    CHANNEL_TIMER    *hTimer,
    MESSAGE_CHANNEL  hMessageChannel,
    DWORD            dwSignalId
)
{
    PSTimer pSTimer = malloc(sizeof(STimer));
    if (!pSTimer)
        return FALSE;
    BOOL bResult = CreateMessageChannel(&pSTimer->hTimeoutChannel);

    if (!bResult)
    {
        free(pSTimer);
        return FALSE;
    }
}

```

```

psTimer->sData.hMessageChannel = hMessageChannel;
psTimer->sData.dwSignalId = dwSignalId;
psTimer->hWaitableTimer = CreateWaitableTimer(
    NULL, FALSE, NULL);

if (!psTimer->hWaitableTimer)
{
    CloseMessageChannel(psTimer->hTimeoutChannel);
    free(psTimer);
    return FALSE;
}

psTimer->hThread = CreateThread(
    NULL, 0,
    TimerRoutine,
    psTimer, 0,
    NULL);

if (!psTimer->hThread)
{
    CloseHandle(psTimer->hWaitableTimer);
    CloseMessageChannel(psTimer->hTimeoutChannel);
    free(psTimer);
    return FALSE;
}
*hTimer = psTimer;
return TRUE;
}

```

В качестве параметров функция принимает: указатель на таймер, канал для записи сигнала и номер сигнала о срабатывании таймера.

5.1.1 Метрики Холстеда

Метрики Холстеда вычисляются на основании анализа числа строк и синтаксических элементов исходного кода программы.

Метрики Холстеда включают четыре измеряемые характеристики программы:

- число уникальных операторов программы;
- число уникальных операндов программы;
- общее число операторов программы;
- общее число операндов программы.

На основе метрик Холстеда можно получить следующие количественные оценки:

- длина программы = число операторов + число операндов;
- объем программы = длина программы * $\log_2$ словарь программы;
- сложность программы = (число уникальных операторов / 2) * (общее число операндов/число уникальных операндов).

Рассчитаем метрики Холстеда для программного кода, представленного в листинге 3. Уникальные операторы и операнды представлены в таблицах 16 и 17 соответственно.

Таблица 16 – Уникальные операторы

Номер	Оператор	Количество
1	=	7
2	free	3
3	;	18
4	(	16
5	)	16
6	sizeof	1
7	malloc	1
8	->	10
9	!	4
10	*	2
11	&	1
12	{	4
13	}	4

Таблица 17 – Уникальные операнды

Номер	Оператор	Количество
1	CreateMessageChannel	1
2	hTimer	2
3	hMessageChannel	3
4	dwSignalId	3

Продолжение таблицы 17

Номер	Оператор	Количество
5	psTimer	17
6	PSTimer	1
7	bResult	2
8	CreateWaitableTimer	1
9	hThread	2
10	CreateThread	1
11	CloseHandle	1
12	CloseMessageChannel	2
13	CHANNEL_TIMER	1
14	MESSAGE_CHANNEL	1
15	STimer	1
16	hTimeoutChannel	3
17	TimerRoutine	1

Рассчитаем словарь программы, который вычисляется как сумма уникальных операторов и уникальных операндов программы: словарь программы = $13 + 17 = 30$.

Рассчитаем длину программы, равную сумме всех операторов программы и всех операндов: длина программы = 130.

Рассчитаем объем программы: $130 * \log_2 30 = 637,91$.

Рассчитаем сложность программы: $(13 / 2) * (43 / 17) = 16,445$.

Полученные характеристики показывают, что программный код имеет высокий уровень качества и позволяет с ним работать в дальнейшем. Показатель сложности программы находится в допустимых границах, что означает, что программный код имеет среднюю сложность поддержки.

5.1.2 Метрики Чепина

Суть метода состоит в оценке информационной прочности отдельно взятого программного модуля с помощью анализа характера использования переменных из списка ввода-вывода.

Все множество переменных, составляющих список ввода-вывода, разбивается на 4 функциональные группы:

- вводимые переменные для расчетов и обеспечения вывода (P);
- модифицируемые или создаваемые внутри программы (M);
- управляющие переменные (C);
- не используемые в программе переменные (T).

Вводимыми переменными являются: {hTimer, hMessageChannel, dwSignalId}.

Модифицируемыми переменными являются: {psTimer, bResult, hMessageChannel, dwSignalId, hWaitableTimer, hThread, hTimer}.

Управляющими переменными являются: {psTimer, bResult, hWaitableTimer, hThread}.

Не используемыми переменными являются: {};

Вводим значение метрики Чепина: $Q = a_1 * P + a_2 * M + a_3 * C + a_4 * T$, где a_1, a_2, a_3, a_4 - весовые коэффициенты.

С учетом весовых коэффициентов выражение принимает вид: $Q = P + 2M + 3C + 0.5T = 3 + 2 * 7 + 4 * 3 + 0 = 29$.

На основе полученных значений метрики Чепина уровень сложности данной программы считаем немного выше низкого, как так количество переменных не слишком велико и не сильно препятствует пониманию программы.

Заключение

В ходе выполнения выпускной квалификационной работы была поставлена задача на разработку распределенной системы хранения данных. Проведен анализ предметной области и подобрана научная литература, рассмотрены особенности распределенных систем и устройство систем хранения данных.

Инкрементальным путем были выделены основные компоненты и характеристики системы, такие чтобы получить некоторые преимущества по сравнению с другими подобными системами.

Проведен функциональный анализ некоторых других подобных систем, которые имеют сходные или отличные от поставленной реализации в построении хранилищ данных.

Разработана, отлажена и протестирована распределенная система хранения данных на основе пар ключ-значение. Предложены модификации реализации, которые показали преимущество по сравнению с некоторыми другими системами.

Разработанная система способна хранить большие объемы данных, терпит аппаратные и программные сбои, а также умеет восстанавливаться после них.

Список использованных источников

1. Ford D., Labelle F., Popovici F. I., et al. Availability in Globally Distributed Storage Systems. USENIX Symposium on Operating Systems Design and Implementation (OSDI), October 2010 [Электронный ресурс]. – Режим доступа: <https://static.googleusercontent.com/media/research.google.com/ru//pubs/archive/36737.pdf> (дата обращения 25.04.2021).
2. Beach B. Hard Drive Reliability Update Sep 2014. — September 23, 2014 [Электронный ресурс]. – Режим доступа: <https://www.backblaze.com/blog/hard-drive-reliability-update-september-2014/> (дата обращения 25.04.2021).
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. Gilber S., A. Lynch. Perspectives on the CAP Theorem [Электронный ресурс]. – Режим доступа: <https://groups.csail.mit.edu/tds/papers/Gilbert/Brewer2.pdf> (дата обращения 25.04.2021).
5. Bosneag M. and Brockmeyer, M. (2002). A formal model for eventual consistency semantics. In IASTED PDCS, 204–209.
6. Maurice P. and Jeannette M. Linearizability: A Correctness Condition for Concurrent Objects. ACM Transactions on Programming Languages and Systems, Vol. 12, No. 3, July 1990, 463-492.
7. D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases. In Proceedings of EuroPar (EuroPar'97), Passau (Germany), 1997.
8. Lamport, L. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM 21, 7 (July 1978), 558–565.
9. Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. (1990) [Электронный ресурс]. – Режим доступа: <https://15721.courses.cs.cmu.edu/spring2018/papers/08-oltpindexes1/pugh-skiplists-cacm1990.pdf> (дата обращения 25.04.2021).

10. Клеппман М., К48 Высоконагруженные приложения. Программирование, масштабирование, поддержка. — СПб.: Питер, 2019. — 640 с. ISBN 978-5-4461-0512-0.
11. Skeen, Dale. A Quorum-based Commit Protocol. Cornell University ECommons Library, 10 February 2013.
12. Pugh, W. Skip lists: a probabilistic alternative to balanced trees. 1990 [Электронный ресурс]. — Режим доступа: <https://15721.courses.cs.cmu.edu/spring2018/papers/08-oltpindexes1/pugh-skiplists-cacm1990.pdf> (дата обращения 25.04.2021).
13. S.V. Adve, K. Gharachorloo. Shared memory consistency models: a tutorial. IEEE Computer, December 1996.
14. M. Greenwald. Non-blocking synchronization and system design. PhD thesis, Stanford University, August 1999.
15. A.D. Birrell, B.D. Levin. An exercise in distributed computing. Commun. AMC 25, 4, April 1982.
16. Lamport L. The part-time parliament. ACM Transactions on Computer Systems, 16 May 1998.
17. Lamport L. Paxos made simple. ACM SIGACT News 32, 4 December 2001
18. Lamport L. Fast paxos. Distributed Computing 19, 2006.
19. Reed, B. Personal communications, 17 May 2013.
20. Van Renesse, R. Paxos made moderately complex. Tech. rep., Cornell University, 2012.
21. Neul P., Cheng E., Gawlick D. The log-structured merge-tree (LSM-tree). Acta Informatica 33, 1996.

Приложение А
ДИАГРАММЫ МОДУЛЯ ЯДРА
(рекомендуемое)

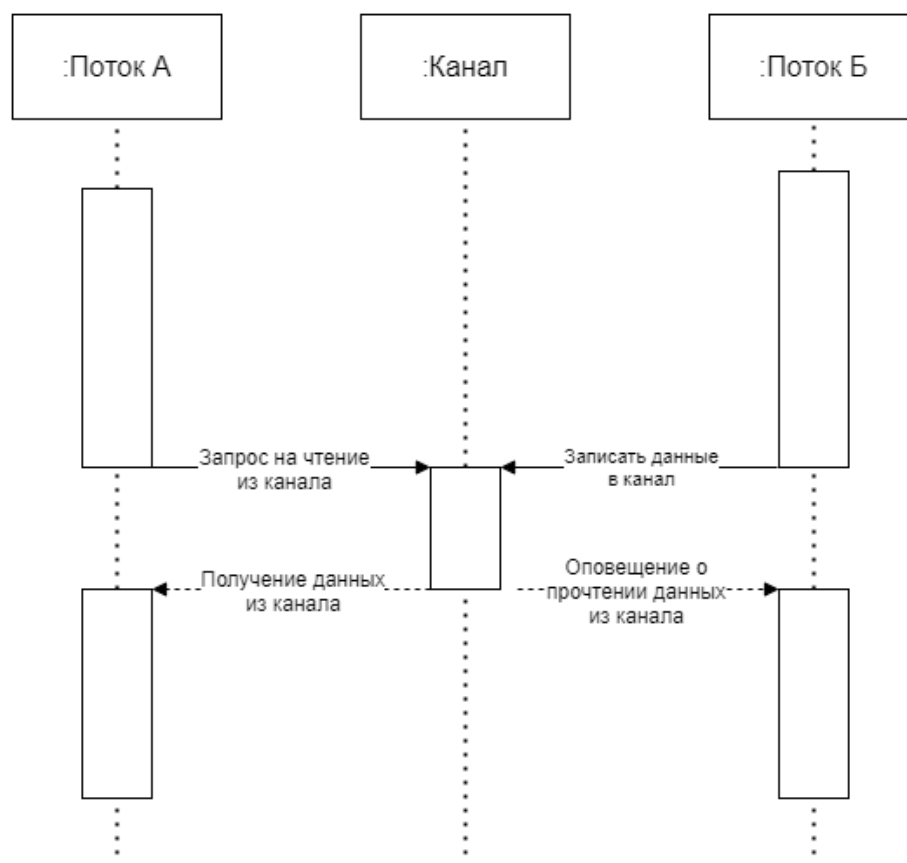


Рисунок А.1 – Диаграмма последовательности для канала сообщений

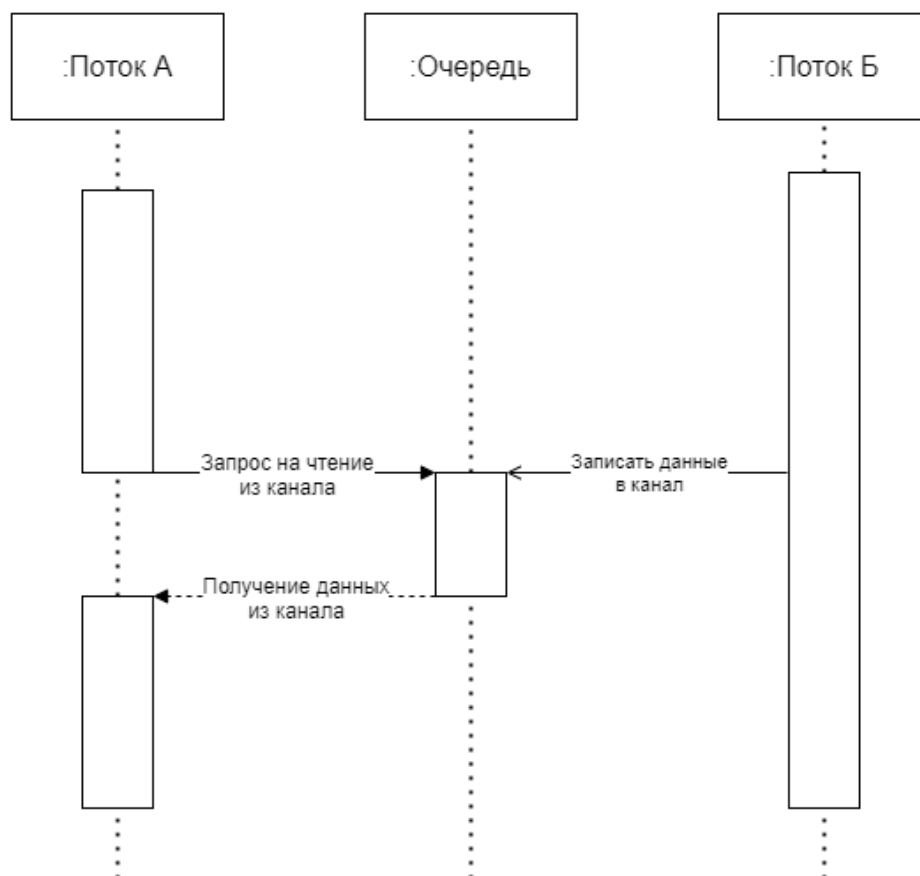


Рисунок А.2 – Диаграмма последовательности для очереди сообщений

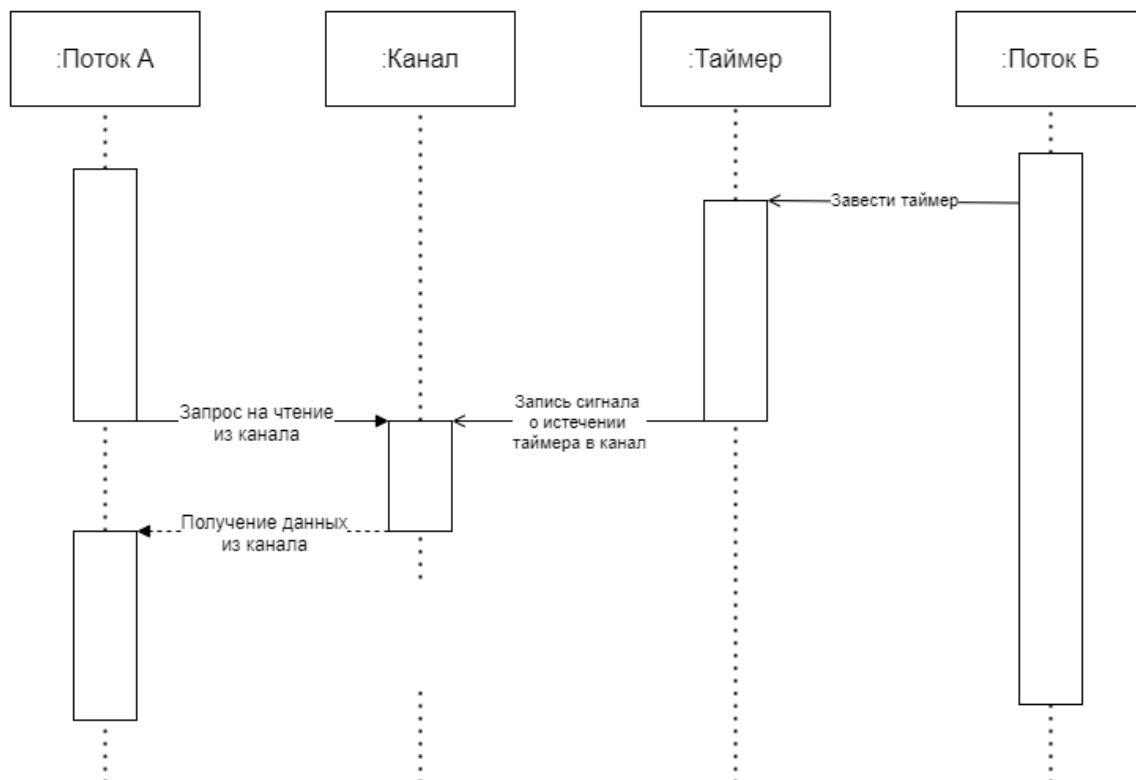


Рисунок А.3 – Диаграмма последовательности для асинхронного таймера

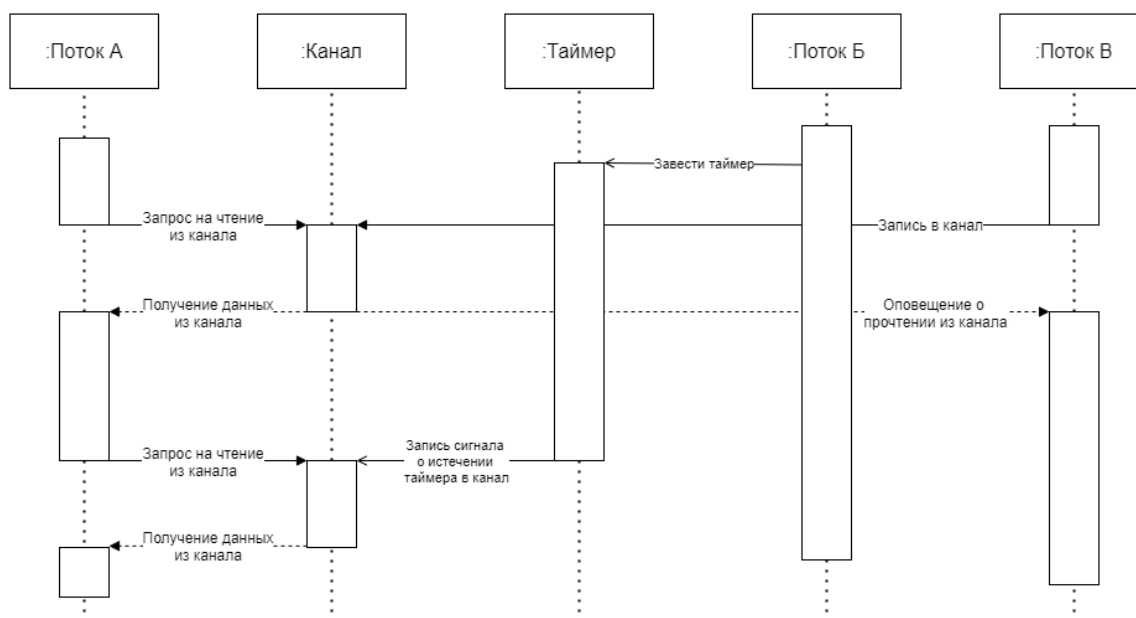


Рисунок А.4 – Диаграмма последовательности для асинхронного таймера и канала сообщений

Приложение Б
АКТ О ВНЕДРЕНИИ
(рекомендуемое)

СПРАВКА
о внедрении результатов выпускной квалификационной работы
Чурсина Павла Алексеевича

В процессе работы над выпускной квалификационной работой бакалавром 4 курса по направлению подготовки «Программная инженерия» Пензенского государственного университета Чурсиным Павлом Алексеевичем по теме: «Распределенные высоконагруженные системы хранения данных» получены конкретные результаты.

Чурсиным Павлом Алексеевичем предложена модификация протокола распределенного консенсуса Raft, с целью увеличения производительности операций, предусмотренных протоколом. В результате внедрения, улучшена производительность разрешения несогласованностей для обеспечения линеаризуемости системы и создания снимков состояний.

В настоящее время практические разработки, включающие результаты данной выпускной квалификационной работы используются в проектах, в которых участвует ООО НТП «Криптософт».

Начальник управления системных исследований
ООО НТП "Криптософт"

Дорошенко И.Н.



Дорошенко И.Н.