Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

Факультет Информатика и системы управления» (ИУ)
Кафедра «Информационная безопасность» (ИУ8)

Отчет

по научно-исследовательской работе студента

на тему Разработка отказоустойчивой системы распределенных вычислений с использованием алгоритма консенсуса Raft

ФИО студента: Железцов Никита Владимирович

Группа: ИУ8-104-2025 Л.д.: 20У474

Специальность: 10.05.01 Компьютерная безопасность

Специализация: 10.05.01_01 Математические методы защиты информации

Научный руководитель НИРС доцент кафедры ИУ8 Колесников Александр Владимирович

Работа выполнена _____ Н.В.Железцов
дата подпись студента И.О.Фамилия студента

Допуск к защите _____ А.В.Колесников
дата подпись научного руководителя И.О.Фамилия научного руководителя

Отчет принят _____ Д.О.Левиев
дата подпись ответственного за НИРС И.О.Фамилия Ответственного за НИРС

Результаты защиты НИРС			
Дата	Балл	Подпись	ФИО

Москва
2025

РЕФЕРАТ

Отчёт содержит 50 стр., 29 рис., 7 источн., 3 прил.

Ключевые слова: распределённые вычисления, отказоустойчивость, алгоритм консенсуса, Raft, репликация, выбор лидера.

Основная цель работы — разработка отказоустойчивой системы распределённых вычислений с использованием алгоритма консенсуса Raft.

В рамках исследования изучены основы построения отказоустойчивых распределённых систем и подробно рассмотрены механизмы работы алгоритма Raft. На основе полученных знаний спроектирована архитектура системы, обеспечивающая согласованное выполнение вычислительных задач в условиях сетевых сбоев и отказов отдельных узлов.

Разработан и реализован прототип системы, включающий серверные и клиентские компоненты, а также механизм автоматического восстановления после отказа ведущего узла. Проведённое тестирование показало, что система сохраняет корректность работы и согласованность состояния при сбоях.

Результаты работы подтверждают, что использование Raft позволяет упростить построение надёжных распределённых систем и обеспечивает требуемый уровень отказоустойчивости без избыточной сложности реализации, а также закладывают основание для последующей разработки библиотеки, обеспечивающей непрерывную связь между формальной спецификацией TLA+ и реальным кодом.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Современные программные системы становятся всё более сложными и распределёнными. Увеличение количества пользователей, потребность в высокой доступности сервисов и минимальных задержках вынуждают разработчиков переходить от монолитных решений к распределённым архитектурам. Подобные системы позволяют масштабировать вычислительные ресурсы, распределять нагрузку между множеством узлов и сохранять работоспособность даже при частичных отказах оборудования.

Однако разработка распределённых систем связана с рядом фундаментальных проблем. Одной из ключевых задач является обеспечение согласованности состояния между узлами, которые могут работать в разных географических точках, испытывать сетевые задержки или полностью выходить из строя. Для решения этой задачи применяются алгоритмы консенсуса, гарантирующие, что все узлы системы приходят к единому решению, несмотря на наличие сбоев.

Алгоритм Raft является одним из наиболее популярных и практически применимых решений для построения отказоустойчивых систем. Он предлагает понятный механизм выбора лидера и репликации логов, что делает его удобным для реализации и интеграции в реальные проекты. Использование Raft позволяет создавать системы, способные обеспечивать согласованное выполнение вычислительных задач и сохранять данные даже при сбоях отдельных узлов.

Обоснование актуальности темы исследования

С каждым годом увеличивается зависимость бизнеса и общества от информационных систем, работающих круглосуточно и без сбоев. Даже кратковременная недоступность может привести к существенным финансовым потерям, снижению доверия пользователей и репутационным рискам. Поэтому разработка отказоустойчивых распределённых решений является одной из ключевых задач современной ИТ-индустрии.

Алгоритм Raft получил широкое распространение благодаря простоте понимания и доказанной надёжности. Его использование позволяет снизить вероятность критических ошибок и облегчить построение согласованных кластеров узлов. Разработка системы, использующей Raft, позволяет на практике изучить принципы построения отказоустойчивых систем и проверить их работу в реальных сценариях сбоев.

Дополнительно данная работа закладывает основу для следующего этапа исследования — разработки инструментария для тестирования покрытия спецификацией TLA+ реального кода. Созданная в рамках этой работы реализация алгоритма Raft будет использована в качестве экспериментальной базы для автоматической проверки соответствия кода его формальной спецификации. Формальная верификация будет взята из предыдущей научной работы. Такой подход позволит объединить инженерные и формальные методы верификации, повысив надёжность проектируемых распределённых систем.

Таким образом, выбранная тема является актуальной, так как направлена на исследование и практическую реализацию решений, повышающих надёжность и доступность распределённых вычислительных систем, а также создаёт основу для дальнейших исследований в области автоматизации верификации и ее поддерживаемости.

Цели и задачи НИРС

Целью работы является разработка и исследование отказоустойчивой системы распределённых вычислений, основанной на алгоритме консенсуса Raft, а также подготовка практической базы для будущей интеграции с инструментами формальной верификации.

Для достижения этой цели решаются следующие задачи:

- провести обзор принципов построения отказоустойчивых распределённых систем;
- изучить устройство алгоритма Raft и описать его ключевые механизмы;
- спроектировать архитектуру системы распределённых вычислений с использованием Raft;
- реализовать прототип системы и провести его ручное тестирование при сбоях узлов;

ОСНОВНАЯ ЧАСТЬ

1 Raft

В Raft каждый узел хранит локально журнал команд, исполняемых конечным автоматом. Так как все процессы получают одинаковые входные данные и применяют идентичные команды в одном и том же порядке, их конечные автоматы приходят к одинаковому состоянию. Одно из отличий Raft заключается в том, что роль лидера здесь вынесена на первый план: он координирует репликацию и манипуляции над конечным автоматом. С этой точки зрения Raft схож с Мульти-Паксосом и атомарной рассылкой: среди узлов выбирается лидер, который принимает решения и задаёт упорядочение сообщений.

Алгоритм Raft определяет три основные роли:

- Кандидат (candidate): Узел, который пытается стать лидером. Он набирает голоса большинства узлов. Если выборы не приводят к явному победителю, запускается новый период и процесс голосования повторяется.
- Лидер (leader): Временный управляющий кластером, обрабатывающий запросы клиентов и взаимодействующий с реплицируемым конечным автоматом. Лидер выбирается на определённый период, который идентифицируется возрастающим номером. Если лидер перестаёт отвечать или подозревается в отказе, начинается процедура переизбрания.
- Последователь (follower): Пассивный участник, хранящий записи журнала и реагирующий на запросы от лидера и кандидатов. По сути, в Raft он объединяет в себе функции акцептора и ученика из Паксоса. Каждый узел стартует в роли последователя.

Чтобы добиться упорядочения без жёсткой синхронизации часов, в Raft используются периоды (эпохи, термы), в течение которых существует только один лидер. Каждый период имеет уникальный номер, а команды внутри периода получают дополнительный индекс. Узлы могут по-разному воспринимать текущий период (например, если они пропустили этап выборов), но каждая отправляемая команда указывает номер периода [1]. Если узел видит период с более высоким номером, он обновляет своё значение периода.

Процесс выбора лидера инициируется, когда последователь не получает подтверждений от текущего лидера, полагая, что тот вышел из строя. В этом случае последователь переходит в состояние кандидата и собирает голоса большинства узлов, стремясь стать новым лидером.

На рис. 1 приведена схема раунда Raft.

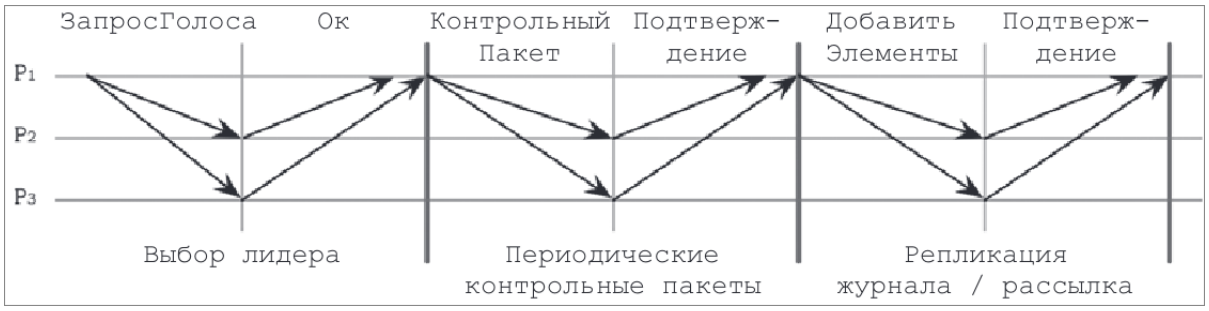

Рисунок 1 – Схема раунда Raft

- Выбор лидера. Когда узел-кандидат (P1 на рисунке) решает стать лидером, он рассылает остальным участникам сообщение , содержащее свой период, последнюю известную ему информацию о периоде, а также идентификатор самой свежей записи в журнале, которую он видел. Если кандидат получает большинство голосов, он становится лидером на текущий период. При этом каждый узел может отдать голос лишь одному кандидату.
- Периодические контрольные пакеты. Для поддержания жизнеспособности системы лидер с определённой периодичностью отправляет контрольные пакеты всем последователям, тем самым подтверждая своё лидерство. Если последователь не получает такие пакеты в течение «тайм-аута выборов», он предполагает сбой лидера и инициирует новый процесс голосования.
- Репликация. Лидер может неоднократно пополнять реплицируемый журнал, отправляя сообщение , где указывает период лидера, индекс и период последней зафиксированной записи, а также одну или несколько новых записей для сохранения.

1.1 Роль лидера в Raft

Лидер может быть выбран только среди узлов, содержащих все актуальные записи. Если в процессе выборов журнал последователя более актуальный, чем у кандидата, то голос за этого кандидата не отдается.

Для победы в голосовании кандидат должен получить большинство голосов. Поскольку записи реплицируются строго по порядку, достаточно сравнить идентификаторы последних записей. После избрания лидер начинает принимать запросы от клиентов и реплицирует их на своих последователей. Для этого он добавляет запись в свой журнал и одновременно отправляет её всем последователям.

Когда последователь получает сообщение о добавлении записей, он вносит эти записи в локальный журнал и отправляет подтверждение, сообщая лидеру, что данные сохранены. Как только лидер получает достаточное количество подтверждений, запись считается зафиксированной и помечается соответствующим образом в его журнале.

Поскольку лидером может стать только узел с наиболее актуальными данными, последователь не отправляет ему обновления. Записи журнала передаются только в одном направлении — от лидера к последователям.

На рисунке 2 представлен пример раунда достижения консенсуса, в котором узел P1 выступает в роли лидера с наиболее актуальной информацией. Лидер выполняет алгоритм, реплицируя записи на своих последователей и фиксируя их после получения подтверждений. Фиксация одной записи автоматически фиксирует все предшествующие записи в журнале. Решение о фиксации может принимать только лидер. Каждая запись в журнале имеет идентификатор периода (терма, указан в верхнем правом углу записи) и индекс, определяющий её позицию в журнале. Зафиксированные записи гарантированно реплицируются на кворум узлов, что позволяет безопасно применять их к конечному автомату в порядке их добавления.

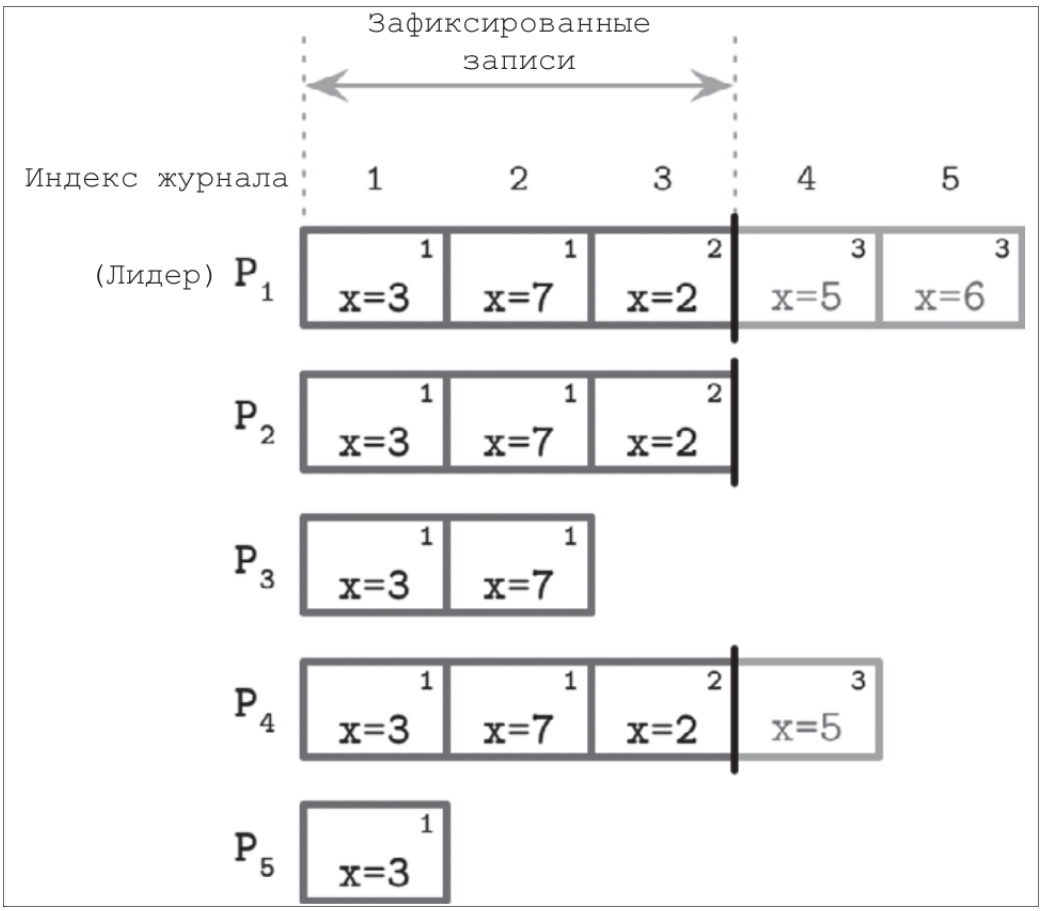

Рисунок 2 – Конечный автомат алгоритма Raft

1.2 Сценарии отказов

Когда несколько последователей решают стать кандидатами, но ни один из них не может набрать большинство голосов, такая ситуация называется "разделенным голосованием". Чтобы снизить вероятность таких случаев, алгоритм Raft применяет рандомизированные таймеры. Это позволяет одному из кандидатов начать следующий этап выборов раньше других, получить достаточное количество голосов и быть избранным, пока остальные кандидаты находятся в ожидании. Такой подход ускоряет процесс выборов, исключая необходимость дополнительной координации между кандидатами.

Если последователи отключаются или задерживают ответы, лидер обязан предпринимать дополнительные попытки доставки сообщений. Если подтверждение от узлов не поступает в ожидаемый срок, лидер повторно отправляет сообщения.

Благодаря уникальным идентификаторам, присваиваемым реплицируемым записям, порядок в журнале остается неизменным, даже при повторной доставке сообщений. Последователи устраняют дублирующие записи, основываясь на их порядковых номерах, что предотвращает нежелательные эффекты от повторных отправок. Также порядковые номера используются для соблюдения хронологии в журнале: последователь отклоняет записи с более высокими номерами, если предыдущие записи не совпадают с его журналом. Если две записи из разных журналов имеют одинаковые идентификаторы и индексы, то они хранят одну и ту же команду, а все предшествующие им записи идентичны.

Для обнаружения сбоев лидер отправляет последовательным узлам контрольные сообщения, подтверждая тем самым активность своего периода. Если один из узлов замечает, что текущий лидер перестал отвечать, он инициирует процедуру выборов. Новый лидер восстанавливает состояние кластера, определяя последнюю согласованную запись (то есть запись с наибольшим номером, которую разделяют лидер и последователь). Он приказывает узлам удалить все незафиксированные записи после этой точки и реплицирует актуальные записи из своего журнала. Лидер не удаляет и не перезаписывает собственные записи, а только добавляет новые.

Таким образом, Raft предоставляет следующие гарантии:

- Только один лидер может быть избран одновременно на заданный период (терм); в течение одного периода не может быть двух активных лидеров;
- Лидер не удаляет и не переупорядочивает содержимое журнала; он только добавляет новые сообщения к нему;
- Зафиксированные записи в журнале гарантированно присутствуют в журналах для последующих лидеров;
- Все сообщения однозначно идентифицируются по идентификаторам сообщений и периодов; ни текущий, ни последующие лидеры не могут повторно использовать один и тот же идентификатор для другой записи.

2 Проектирование архитектуры системы

2.1 Общая архитектура системы

2.1.1 Роли узлов

На верхнем уровне архитектуры выделяются два класса узлов: *узлы консенсуса (Raft-узлы)* и *вычислительные узлы (workers)*. Такое разделение соответствует распространённой практике отделения *control plane* (координация, согласование состояния, диспетчеризация) от *data/compute plane* (непосредственное выполнение пользовательских задач) [2].

Raft-узлы формируют кластер, поддерживающий реплицированную машину состояний, принимают клиентские запросы, выполняют проверку/валидацию, принимают решения о постановке задач в очередь и о распределении задач по воркерам, а также обеспечивают линейризуемую (или близкую к ней) семантику наблюдения состояния.

Воркеры, в свою очередь, реализуют только «полезную работу»: исполняют выданные задания в соответствии с политикой планировщика, периодически отчитываются о прогрессе и возвращают результаты.

Архитектура приложения приложения приведена на рис. 4.

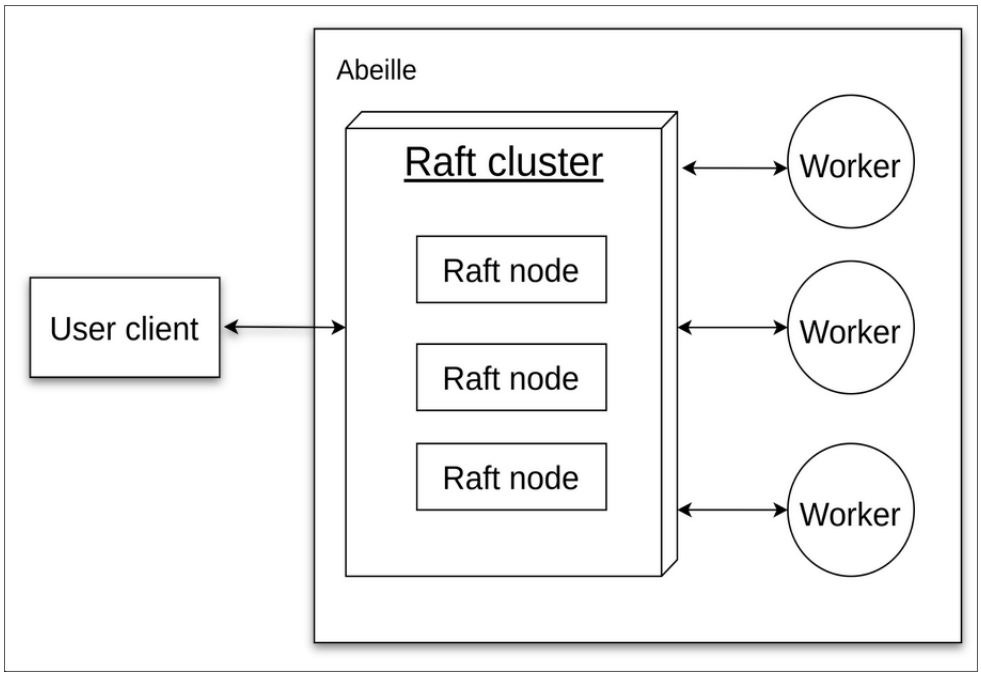

Рисунок 3 – Высокоуровневая архитектура приложения

Данный подход минимизирует связанность компонентов: все операции записи в «истину состояния» проходят через кластер Raft, а вычислительный контур остаётся тонким и легко масштабируемым.

Это упрощает развитие системы (изменения протокола, эволюцию форматов, миграции), повышает управляемость и упрощает доказательство корректности, так как инварианты безопасности сосредоточены в одном месте — в реплицированной машине состояний.

2.1.2 Потоки управления и данные

Клиент взаимодействует только с кластером Raft (запросы могут быть приняты любым узлом и переадресованы лидеру). Каждая операция, которая влияет на состояние системы (создание/отмена задания, изменение приоритета, отметка результата), сначала записывается в лог кластера, подтверждается кворумом узлов и *только затем* применяется к машине состояний.

Из машины состояний изменения транслируются в модуль диспетчеризации (Task manager), который назначает задания конкретным воркерам.

Обратный поток (результаты) также фиксируется через Raft перед тем, как станет видимым для клиента.

Такая схема обеспечивает:

- детерминированный порядок событий;
- устойчивость к повторам и сетевым артефактам за счёт идемпотентных идентификаторов задач;
- возможность повторного назначения задач при сбоях воркеров без нарушения согласованности.

2.1.3 Обоснование отсутствия прямых связей между клиентом и рабочим узлом

Исключение прямого клиентского трафика на воркеры и запрет горизонтальных коммуникаций между воркерами снижает сложность системы и радиус отказа:

- **Границы доверия и безопасность.** Все проверки аутентификации/авторизации, квоты, контроль нагрузки и согласование политик происходят в одном месте (в кластере Raft), что уменьшает поверхность атаки и упрощает аудит.

– **Единая точка сериализации.** Централизация записи в лог предотвращает расхождение состояний при разделениях сети и сбоях; воркеры не принимают решений, влияющих на глобальную консистентность.
– **Простота воркеров.** Воркеры могут оставаться без состояния (stateless) или слабосвязными по состоянию; замена/масштабирование происходит без координации между ними, что упрощает эксплуатацию.
– **Управляемое масштабирование.** Планирование, балансировка сосредоточены в кластере Raft; добавление воркеров линейно увеличивает пропускную способность без изменения протоколов согласования.

2.1.4 Отказоустойчивость и модель сбоев

Система проектируется под модель сбоев crash-stop/crash-recovery и частичной синхронности сети.

Кластер Raft обеспечивает безопасность (отсутствие противоречивых коммитов) при любой динамике и живучесть при выполнении допущений о времени [1]. Отказ отдельного воркера не влияет на целостность: незавершённые задания остаются зафиксированными в состоянии и могут быть переназначены. При отказе лидера выполняется переизбрание; после выбора нового лидера система продолжает обслуживание запросов, как только будет достигнут кворум. Такая конфигурация позволяет сохранять работоспособность при недоступности до $\lfloor (N-1)/2 \rfloor$ узлов кластера консенсуса (для N — числа Raft-узлов), что является стандартной кворумной гарантией [3].

2.1.5 Масштабирование и управление нагрузкой

Архитектура поддерживает независимое масштабирование плоскостей:
– **Горизонтальный рост вычислительных мощностей.** Добавление воркеров увеличивает суммарные вычислительные мощности без влияния на алгоритм консенсуса.
– **Стабильность.** Количество Raft-узлов остаётся небольшим (обычно 3–5) для минимизации латентности кворума и времени выборов [1].

2.1.6 Рассмотренные альтернативы

Были рассмотрены альтернативные топологии:
– прямое взаимодействие клиента с воркерами;

– полно-связанная p2p-сетка воркеров с децентрализованным согласованием;
– объединение ролей (каждый воркер — участник консенсуса).

От первого варианта отказались из-за роста поверхности ошибок и сложности обеспечения глобальной согласованности. Второй вариант усложняет доказательство корректности и эксплуатацию (динамика членства, маршрутизация, «горячие» ключи). Третий ухудшает масштабирование: увеличение количества участников консенсуса повышает латентность кворума и время выбора лидера.

Предложенная архитектура сохраняет консенсус компактным и управляемым, а вычисления — эластичными и дешёвыми в масштабировании.

2.2 Архитектура Raft узлов

Каждый узел кластера Raft содержит несколько функциональных модулей, взаимодействие которых обеспечивает корректную обработку запросов и поддержание согласованного состояния всей системы. Архитектура узла изображена на рис. 4.

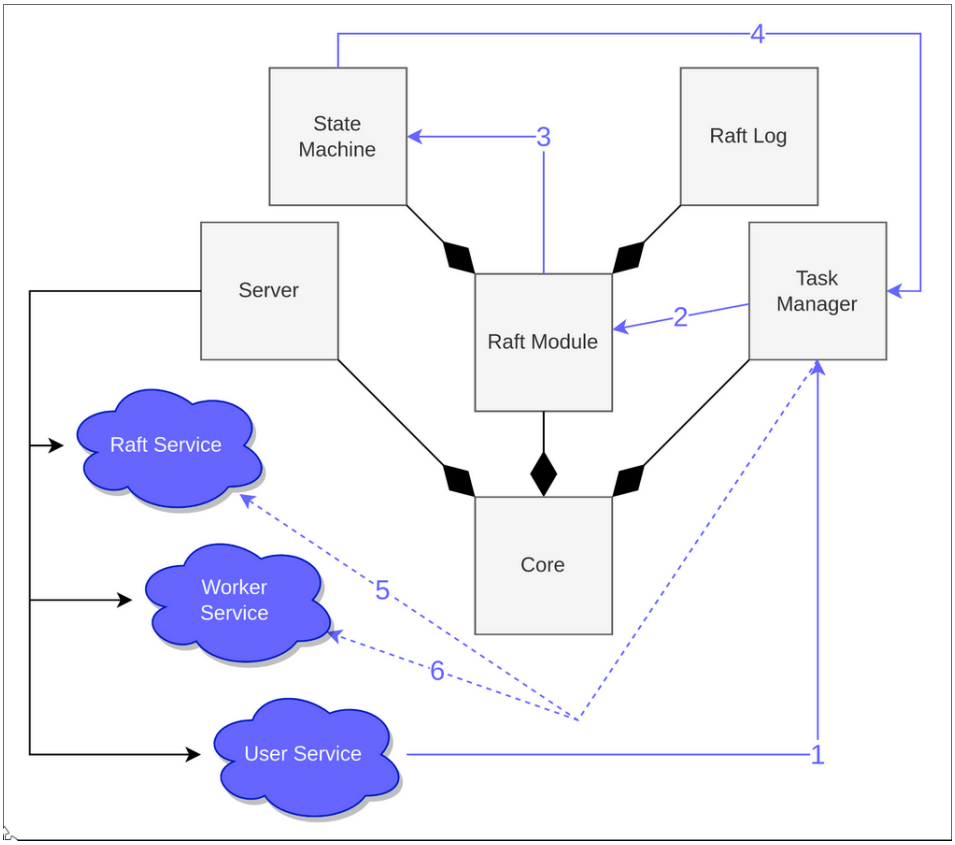

Рисунок 4 – Архитектура Raft узла

Поток обработки клиентского запроса начинается с поступления данных на **серверный модуль** узла-лидера. Сервер принимает запросы от клиентов и маршрутизирует их во внутренние сервисы. В частности, запросы, связанные с постановкой новых заданий на выполнение, направляются в **пользовательский сервис (User Service)**, зарегистрированный в сервере. Этот сервис выполняет первичную валидацию и подготовку данных, после чего передаёт их в **менеджер заданий (Task Manager)**. Задача менеджера — выбрать подходящий рабочий узел (worker) для выполнения поступившей задачи, исходя из состояния системы, доступности ресурсов и стратегий планирования.

Поскольку система должна сохранять корректность работы даже при сбоях отдельных узлов, Task Manager не отправляет задание воркеру напрямую. Вместо этого сформированная операция передаётся в **модуль Raft**, который отвечает за репликацию и согласование состояния между всеми узлами кластера. Raft-модуль добавляет операцию в **журнал Raft (Raft Log)**, который хранит все ещё не зафиксированные изменения состояния. В рамках механизма Raft эти записи реплицируются на все узлы кластера при очередных heartbeat-сообщениях лидера.

Когда большинство (кворум) узлов подтверждает успешную запись данных в свои локальные логи, лидер выполняет **коммит (commit)** — операцию переноса записи в **машину состояний (State Machine)**. Машина состояний представляет собой детерминированный автомат, который преобразует последовательность закоммиченных операций в текущее состояние системы. За счёт применения одной и той же последовательности команд на каждом узле достигается консенсус — все реплики приходят к одинаковому результату независимо от того, какой узел является лидером.

После того как операция фиксируется в машине состояний лидера, Task Manager получает сигнал о возможности безопасного выполнения задания. На этом этапе задание отправляется выбранному воркеру. Результат выполнения также проходит через Raft: при завершении работы воркер формирует ответ, который реплицируется и коммитится аналогично исходному запросу. Лишь после этого результат считается подтверждённым и возвращается клиенту, что гарантирует согласованность даже при отказах в момент ответа.

Ключевым элементом узла является **ядро (Core)**, которое выполняет роль интеграционного слоя между модулями. Core отвечает за инициализацию и завершение работы компонентов, координирует обмен сообщениями между ними и предоставляет единый интерфейс для внешнего управления узлом.

Такая архитектура обеспечивает согласованное и детерминированное поведение системы при сбоях, гарантирует, что любая операция либо применена на большинстве узлов, либо будет проигнорирована. Система остаётся работоспособной до тех пор, пока доступно более половины узлов кластера, что является базовым требованием алгоритмов кворумного консенсуса.

2.3 Архитектура клиентской части

Клиентская часть системы обеспечивает пользователю удобный интерфейс для постановки задач на выполнение и получения результатов. Основными компонентами клиентской стороны являются:

- **Конфигурационные данные задачи** — входные параметры, описывающие вычислительную задачу, хранятся в формате JSON (например, `data.json`). Такой формат выбран благодаря читаемости, простоте валидации и широкому распространению в экосистемах различных языков.
- **Библиотека задачи** — бинарный модуль (`libtask.so`), содержащий реализацию функции обработки данных. Такой подход позволяет передавать не только статические входные данные, но и сам алгоритм, который должен быть исполнен на воркере.
- **Клиентское приложение (CLI)** — консольная утилита, служащая интерфейсом между пользователем и распределённой системой. CLI принимает указанные входные файлы, сериализует их в формат, подходящий для сетевой передачи, и формирует запрос к кластеру Raft.

Процесс взаимодействия клиента с системой проходит следующие этапы (см. рис. 5):

а) Пользователь передаёт клиенту `data.json` и динамическую библиотеку `libtask.so`, после чего CLI формирует запрос, содержащий данные задачи и код её исполнения, и отправляет его в кластер Raft.

б) Лидер кластера принимает запрос, реплицирует его и передаёт в подсистему диспетчеризации, которая назначает задачу конкретному рабочему узлу. В запросе указывается уникальный идентификатор задачи для обеспечения идемпотентности.

в) Клиент воркера получает задание и передаёт его на выполнение в среду исполнения, где подгружается `libtask.so` и запускается соответствующая функция с параметрами из `data.json`. После завершения выполнения результат формируется в сериализованном виде и отправляется обратно в кластер Raft для фиксации в состоянии.

г) После коммита результата в реплицированной машине состояний лидер Raft уведомляет клиента о завершении работы. CLI принимает ответ, сохраняет результат локально либо выводит пользователю в удобной форме.

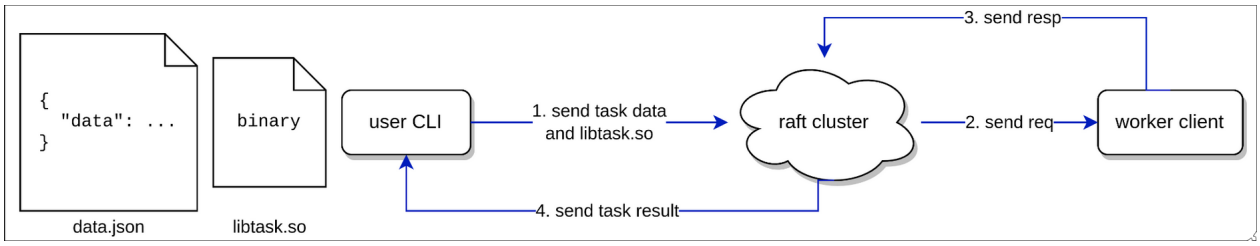

Рисунок 5 – Взаимодействие клиента с кластером

Данная архитектура клиентской части обладает рядом преимуществ:

– **Универсальность.** Клиент не привязан к конкретному типу задач — любое вычисление, представленное в виде бинарной библиотеки с единым интерфейсом, может быть передано в систему.

– **Минимальные зависимости.** Клиент реализован как лёгкая утилита, не требующая сложной конфигурации; это упрощает её развёртывание и интеграцию в CI/CD.

– **Надёжность.** Все этапы (от постановки задачи до получения ответа) проходят через механизм Raft, что гарантирует сохранность данных даже при сбоях сети или падении узлов.

Таким образом, клиентская часть системы выступает удобным и унифицированным интерфейсом, который инкапсулирует сложность взаимодействия с распределённой инфраструктурой и позволяет пользователю сосредоточиться на определении вычислительных задач и анализе результатов.

2.4 Архитектура вычислительного узла

Вычислительный узел предназначен для непосредственного выполнения задач, поступающих из кластера Raft, и возврата результатов их обработки. Архитектура воркера организована таким образом, чтобы обеспечить изоляцию вычислений, устойчивость к сбоям и минимальное влияние выполняемой задачи на стабильность основной службы.

Ключевые компоненты вычислительного узла:

- **Главный процесс (main)** — точка входа, отвечающая за инициализацию среды, запуск клиента воркера и менеджера процессов. Главный процесс контролирует жизненный цикл остальных компонентов и обеспечивает их перезапуск при сбоях.
- **Клиент воркера (worker client)** — процесс, который поддерживает постоянное соединение с кластером Raft, принимает задания на выполнение и уведомляет менеджер о необходимости запуска вычислительных контейнеров.
- **Менеджер (manager)** — отдельный процесс, отвечающий за создание изолированных рабочих процессов (**mayfly**), которым передаётся выполнение конкретных задач. Такой подход снижает риск повреждения памяти или зависания основного сервиса из-за некорректного кода задачи.
- **Mayfly-процесс** — лёгкий дочерний процесс, который выполняет задачу. На этапе инициализации загружает указанную динамическую библиотеку, выполняет функцию обработки данных и формирует результат.

Поток выполнения задачи организован следующим образом (см. рис 6):

а) **Приём запроса.** Worker client получает задание из кластера Raft и подтверждает его приём.

б) **Создание рабочего процесса.** Worker client отправляет запрос менеджеру на создание нового mayfly-процесса. Менеджер выполняет `fork()`, создавая изолированный процесс.

в) **Загрузка библиотеки.** Mayfly-процесс подгружает указанную пользователем библиотеку задачи (по имени, переданному в запросе).

г) **Выполнение задачи.** Обработка входных данных производится внутри mayfly-процесса. В случае ошибки или аварийного завершения процесс завершается без влияния на основной клиент воркера.

д) **Отправка результата.** После завершения выполнения результат возвращается менеджеру, который передаёт его обратно в worker client.

е) **Репликация результата.** Worker client формирует ответ и отправляет его в кластер Raft для фиксации в реплицированной машине состояний.

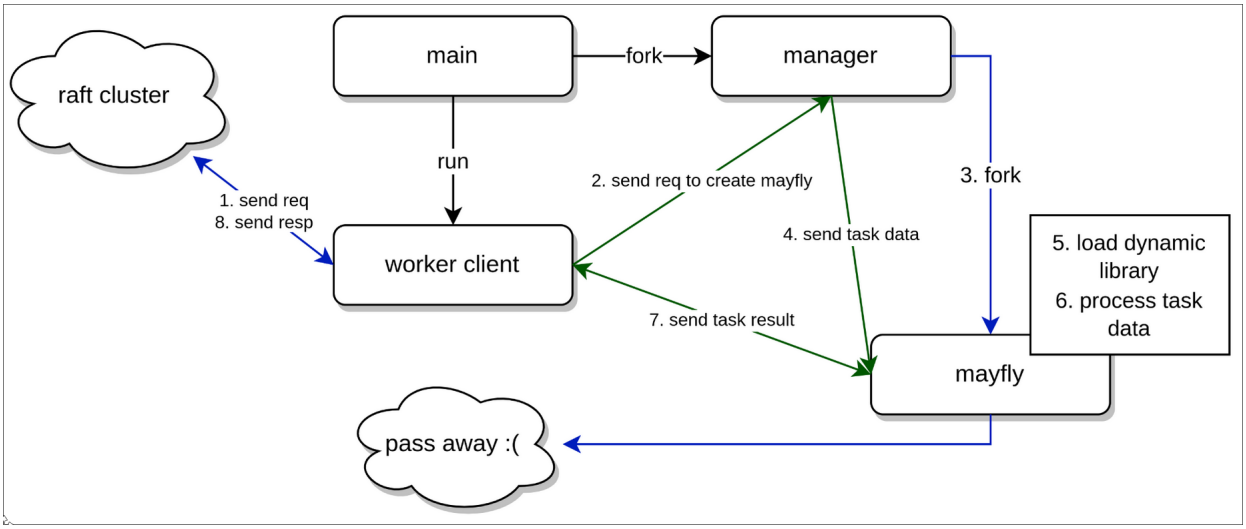

Рисунок 6 – Взаимодействие вычислительного узла с кластером

Архитектура поддерживает модель «одно задание — один процесс», что обеспечивает:

- **Изоляцию исполнения**: сбой в задаче не влияет на остальные процессы воркера;
- **Высокую отказоустойчивость**: при завершении дочернего процесса менеджер может корректно обработать сбой, зарегистрировать ошибку и при необходимости запросить повторное выполнение через кластер;
- **Простоту очистки ресурсов**: после завершения работы mayfly-процесс уничтожается вместе с выделенными ресурсами, исключая утечки памяти.

Такое построение вычислительного узла обеспечивает гибкость и устойчивость всей системы, позволяя безопасно выполнять пользовательский код, динамически масштабировать количество обрабатываемых задач и гарантировать возврат результата даже при частичных сбоях.

3 Реализация системы распределенных вычислений

В данном разделе описан выбор технологического стека и его роль в реализации распределённой системы. Подробно обоснован выбор gRPC и Protocol Buffers как основы сетевого взаимодействия, обеспечивающих строгую типизацию, производительность и удобство сопровождения кода. Далее рассмотрена реализация Raft-узла как центрального компонента кластера, включая структуру лога, машину состояний с механизмом снепшотов, серверный модуль и менеджер заданий. Приведены ключевые аспекты работы клиентского приложения, включая формат конфигурационных файлов и двунаправленный потоковый протокол взаимодействия, а также описана организация вычислительных узлов, их взаимодействие с кластером и обработка сбоев. В совокупности раздел демонстрирует, каким образом выбранные технологии и архитектурные решения обеспечивают согласованность состояния, отказоустойчивость и удобство масштабирования всей системы.

В приложении А приведен протокол всех частей распределенной вычислительной системы, в приложении Б находится полная UML-диаграмма классов всех частей. В проекте Abeille [4] можно найти полный исходный код.

3.1 Выбор технологий и инструментов

Для реализации сетевого взаимодействия в системе был выбран современный стек **gRPC + Protocol Buffers**, который обеспечивает эффективную и строго типизированную коммуникацию между узлами. gRPC представляет собой высокопроизводительный фреймворк для удалённых вызовов процедур (Remote Procedure Call, RPC), разработанный компанией Google. Он использует протокол HTTP/2 на прикладном уровне и TCP на транспортном, что позволяет реализовать двунаправленные потоки, мультиплексирование и сжатие заголовков, а также снижает накладные расходы на установку соединений.

Для описания интерфейсов gRPC использует **.proto**-файлы, обрабатываемые компилятором Protocol Buffers, который автоматически генерирует код клиентских и серверных заглушек на выбранных языках. В рамках данной системы .proto-файлы служат единым источником правды для определения всех

контрактов: как между клиентом и кластером Raft, так и между внутренними компонентами (сервер, менеджер заданий, воркеры). Это снижает вероятность ошибок согласования интерфейсов, упрощает сопровождение системы и повышает читаемость кода.

Использование данного стека технологий обеспечивает следующие преимущества:

- **Единая модель данных:** структуры описываются один раз в .proto-файлах и автоматически транслируются в типобезопасные объекты для всех участвующих компонентов.
- **Удобная сериализация:** поддержка как бинарного, так и текстового формата, что упрощает отладку и анализ сетевого трафика.
- **Эволюция протокола:** поддержка обратной и прямой совместимости при расширении .proto-схем, что позволяет постепенно обновлять компоненты системы без остановки кластера.
- **Высокая производительность:** низкие накладные расходы на передачу сообщений и эффективная поддержка параллельных соединений благодаря HTTP/2.

3.2 Реализация Raft-узла

Raft-узел является центральным элементом системы: он реализует алгоритм консенсуса, поддерживает реплицированную машину состояний и предоставляет интерфейсы для взаимодействия с пользователями и вычислительными узлами. Реализация узла включает несколько взаимосвязанных компонентов: Raft-лог, машину состояний с поддержкой снепшотов, серверный модуль, менеджер заданий и подсистему конфигурации.

3.2.1 Raft-лог

Лог Raft представляет собой основной журнал команд, которые подлежат репликации между всеми узлами кластера. После достижения кворума запись из лога коммитится и применяется к машине состояний. В данной реализации поддерживаются две команды:

- `RAFT_COMMAND_ADD` — добавление новой задачи в очередь назначенных;
- `RAFT_COMMAND_MOVE` — перевод задачи в состояние завершённых.

Структура записи лога описана с использованием Protocol Buffers (см. рис. 7). Каждая запись включает тип команды, номер терма, полезную нагрузку (упакованную в `TaskWrapper`), а также дополнительные поля, зависящие от команды (`AddRequest` или `MoveRequest`). Использование Proto-схемы обеспечивает строгую типизацию, возможность обратной совместимости при изменении формата и автоматическую генерацию сериализаторов/десериализаторов.

```
message Entry {
  RaftCommand command = 1;
  uint64 term = 2;
  TaskWrapper task_wrapper = 3;
  AddRequest add_request = 4;
  MoveRequest move_request = 5;
}
```

Рисунок 7 – Структура записи Raft-лога, определённая в формате Protocol Buffers.

С точки зрения реализации лог представляет собой потокобезопасную обёртку над `std::deque`. Такой выбор объясняется необходимостью обеспечения:

- **Быстрого случайного доступа** по индексу (для алгоритма согласования);
- **Эффективного удаления элементов** как из начала (очистка закоммиченных записей после создания снепшота), так и из конца (откат неконсистентных записей при пересинхронизации).

Индексация в логе начинается с 1; нулевой индекс считается невалидным и используется для упрощения проверок корректности ссылок на записи.

3.2.2 Машина состояний и механизм снепшотов

Машина состояний реализует детерминированный автомат, который постепенно применяет закоммиченные записи лога и формирует текущее состояние кластера. Она хранит:

- список назначенных задач;
- список завершённых задач;
- текущий номер терма и индекс последней применённой команды.

Для предотвращения бесконтрольного роста размера лога реализован механизм **снепшотов (snapshotting)**. При достижении порога, заданного пользователем в конфигурационном файле, машина состояний инициирует создание снепшота: сериализует своё текущее состояние и сохраняет его на диск. Данный процесс выполняется асинхронно, что позволяет продолжать применение новых команд без блокировки. После успешной записи снепшота лог очищается от записей, которые были зафиксированы в сохранённом состоянии, что ограничивает его размер сверху.

В случае, если лидер не находит в своём логе необходимой записи для синхронизации с отстающим узлом, он инициирует передачу ему актуального снепшота, после чего узел может восстановить своё состояние и продолжить репликацию с последнего зафиксированного индекса.

3.3 Серверный модуль

Серверный модуль инкапсулирует gRPC-сервер, принимающий запросы от:

- других узлов (сервис **RaftService**: `RequestVote()`, `AppendEntries()`, `InstallSnapshot()`);
- клиентов (сервис **UserService**);
- вычислительных узлов (сервис **WorkerService**).

Сервер маршрутизирует запросы: приём запросов на запись выполняется любым узлом, но при отсутствии лидерства происходит редирект клиента на лидера.

3.4 Менеджер заданий

Task Manager отвечает за подготовку записей для лога Raft при добавлении и завершении задач, а также за передачу результата пользователю. В реализации предусмотрены три основных метода:

- `UploadTaskData` — получает данные задачи, выбирает идентификатор воркера через `WorkerServiceImpl::AssignTask()` и формирует запись с командой `RAFT_COMMAND_ADD`. После успешного выбора воркера запись добавляется в лог Raft для репликации и последующего применения к машине состояний.
- `ProcessTask` — передаёт задачу на исполнение выбранному воркеру.

– `UploadTaskResult` — формирует команду `RAFT_COMMAND_MOVE`, переводящую задачу из статуса *assigned* в *completed*, и добавляет её в лог для репликации и коммита.
– `SendTaskResult` — отправляет готовый результат обратно в `UserServiceImpl` для передачи клиенту.

Следует отметить, что назначение задачи на воркера выполняется ровно один раз в момент вызова `UploadTaskData`. В текущей версии реализации не предусмотрен механизм автоматического перевыбора воркера или повторной выдачи задачи при сбое, однако согласованность и идемпотентность обеспечиваются за счёт самого алгоритма Raft: если запись не была закоммичена до отказа лидера, она не будет применена к машине состояний, и задача может быть отправлена повторно клиентом.

3.4.1 Конфигурация и инициализация узла

Перед запуском узла пользователь должен предоставить JSON-конфигурацию (см. рис. 8), в которой указываются:
– адреса сервисов (пользовательский, рафт-сервис, сервис воркера);
– список адресов всех узлов кластера (включая данный узел);
– параметр `snapshot_after` — количество коммитов до создания нового снепшота;
– путь к файлу, куда будет записываться снепшот.

```
{
  "user_address": "127.0.0.1:50050",
  "raft_address": "127.0.0.1:50050",
  "worker_address": "127.0.0.1:50050",
  "peers": [
    "127.0.0.1:50050",
    "127.0.0.1:50051",
    "127.0.0.1:50052",
  ],

  "snapshot_after": 10,
  "snapshot_to": "./build/snapshots/snapshot_0.sp"
}
```

Рисунок 8 – Пример конфигурационного файла Raft-узла.

В коде конфигурация инкапсулирована в синглтон-объект, который доступен всем компонентам узла, что упрощает инициализацию и исключает дублирование настроек. Приложение также поддерживает переопределение некоторых параметров через аргументы командной строки, что полезно для тестирования и развёртывания в разных окружениях.

3.5 Реализация клиентского приложения

Клиентское приложение предоставляет пользователю консольный интерфейс для постановки задач на выполнение в кластер и получения результатов. Его функциональность разбита на несколько подсистем: загрузка конфигураций, подготовка задания, взаимодействие с кластером, ожидание/получение результата и сохранение артефактов.

3.5.1 Конфигурация клиента

Клиентское приложение конфигурируется с помощью трех файлов. Первый из них содержит перечень известных адресов узлов Raft. Пример привден на рис 9.

Рисунок 9 – Пример клиентского конфигурационного файла

Клиент инициализирует пул соединений к указанным адресам и выбирает рабочую «точку входа» по порядку (или по доступности). При ошибке соединения выполняется следующий адрес из списка, что обеспечивает толерантность к отказу отдельного узла без участия пользователя. Запросы, изменяющие состояние (постановка задачи), отправляются на любой доступный узел; при необходимости выполняется прозрачный редирект на лидера.

Второй файл задаёт путь для сохранения результатов и имя динамической библиотеки задачи (см. рис. 10)

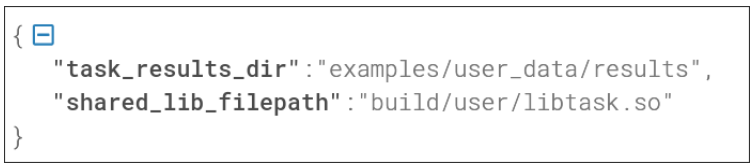

Рисунок 10 – Пример пользовательского конфигурационного файла

Третий же файл содержит данные для задания (см. рис. 11).

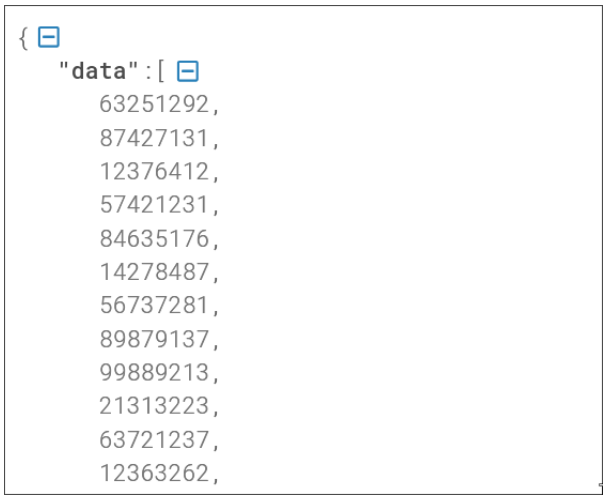

Рисунок 11 – Пример данных для задания

3.5.2 Протокол клиентского приложения

Взаимодействие клиента с кластером реализовано поверх двунаправленного потокового вызова `UserService.Connect`, определённого в gRPC (см. рис 12. Использование потокового RPC позволяет поддерживать длительное соединение, через которое клиент и сервер обмениваются сообщениями в обоих направлениях: клиент отправляет последовательность запросов, а сервер — последовательность ответов, не прерывая сессию. Такой подход позволяет объединить обнаружение лидера, постановку задачи и получение результата в единый непрерывный диалог.

```
// ──────────────────────────── User Service ──────────────────────────── //

enum UserStatus {
  USER_STATUS_DOWN = 0;
  USER_STATUS_IDLE = 1;
  USER_STATUS_UPLOAD_DATA = 2;
}

message UserConnectRequest {
  UserStatus status = 1;
  bytes task_data = 2;
  string filename = 3;
  bytes lib = 4;
}

enum UserCommand {
  USER_COMMAND_NONE = 0;
  USER_COMMAND_REDIRECT = 1;
  USER_COMMAND_ASSIGN = 2;
  USER_COMMAND_RESULT = 3;
}

message UserConnectResponse {
  UserCommand command = 1;
  uint64 leader_id = 2;
  TaskState task_state = 3;
}

service UserService {
  rpc Connect(stream UserConnectRequest) returns (stream UserConnectResponse) {}
}
```

Рисунок 12 – Протокол общения клиента с кластером Raft

Каждое сообщение клиента содержит поле `status`, отражающее фазу работы. На этапе инициализации клиент посылает кадр со статусом `IDLE`, сигнализируя о готовности к работе. Если узел, на который был направлен запрос, не является лидером кластера, сервер возвращает ответ с командой `REDIRECT` и идентификатором актуального лидера. Получив такую команду, клиент закрывает поток и устанавливает соединение с указанным узлом, гарантируя, что все последующие запросы пойдут через единственный согласованный канал.

После установления соединения с лидером клиент переходит к передаче данных задачи. Для этого используется статус `UPLOAD_DATA`, а в сообщении указываются полезная нагрузка (например, содержимое входного файла), логическое имя этой нагрузки и идентификатор динамической библиотеки, которая будет использована при выполнении задачи. Следует подчеркнуть, что по сети передаётся только имя библиотеки, а не её бинарный образ, поскольку сами модули заранее развёрнуты на вычислительных узлах. Сервер принимает эти

данные, реплицирует соответствующую команду в Raft-лог, а после достижения кворума отправляет клиенту ответ с командой `ASSIGN`, в котором содержится сгенерированный идентификатор задачи и подтверждение того, что она принята кластером.

Соединение не разрывается после постановки задачи, а остаётся открытым до получения финального результата. Когда задача завершается на воркере, её результат реплицируется через Raft и фиксируется в машине состояний. Лишь после этого сервер отправляет клиенту команду `RESULT`, содержащую окончательный статус выполнения и, при необходимости, сериализованный результат. Таким образом, клиент всегда получает согласованное состояние, подтверждённое кворумом узлов.

В совокупности такой протокол обеспечивает упорядоченный и надёжный обмен: постановка задачи и выдача результата проходят через единую двунаправленную сессию, которая переносит всю сложность работы с распределённым кластером на серверную сторону. Клиенту не требуется отслеживать состояние выборов или дублировать запросы вручную: редиректы, подтверждения и уведомления о результате получаются автоматически, что делает взаимодействие простым, но при этом линейризуемым и устойчивым к сбоям.

3.6 Реализация вычислительного узла

Вычислительный узел (worker) предназначен для исполнения задач, поступающих из кластера, и построен вокруг длительного двунаправленного потокового соединения `WorkerService.Connect`. При запуске узел считывает локальную конфигурацию, в которой указывается каталог для размещения динамических библиотек (`libs_dir`) (пример конфигурации приведен на рис. 13). Этот каталог используется как единое хранилище исполняемых модулей, на которые ссылаются задания. Клиентская часть передаёт библиотеку в кластер Raft; после репликации и коммита серверная сторона распределённой системы доставляет содержимое соответствующего артефакта на вычислительные узлы, где он сохраняется в `libs_dir` и становится доступным среде исполнения.

```
{
   "libs_dir":"worker/data/"
}
```

Рисунок 13 – Пример конфигурационного файла вычислительного узла

Протокол общения вычислительного узла с кластером представлен на рис. 14.

```
enum WorkerStatus {
  WORKER_STATUS_DOWN = 0;
  WORKER_STATUS_IDLE = 1;
  WORKER_STATUS_BUSY = 2;
  WORKER_STATUS_COMPLETED = 3;
}

message WorkerConnectRequest {
  WorkerStatus status = 1;
  TaskState task_state = 2;
}

enum WorkerCommand {
  WORKER_COMMAND_NONE = 0;
  WORKER_COMMAND_ASSIGN = 1;
  WORKER_COMMAND_PROCESS = 2;
  WORKER_COMMAND_REDIRECT = 3;
}

message WorkerConnectResponse {
  WorkerCommand command = 1;
  uint64 leader_id = 2;
  TaskID task_id = 3;
  bytes task_data = 4;
  bytes lib = 5;
}

service WorkerService {
  rpc Connect(stream WorkerConnectRequest) returns (stream
      WorkerConnectResponse) {}
}
```

Рисунок 14 – Протокол общения вычислительного узла с кластером Raft

Логика взаимодействия с кластером организована как непрерывная сессия. Сразу после установления соединения вычислительный узел отправляет кадр с собственным состоянием `IDLE`, тем самым сигнализируя о готовности к приёму работы. Если подключение установлено не с лидером, сервер немедленно возвращает команду перенаправления с указанием `leader_id`; воркер закрывает текущий поток и повторяет подключение к лидеру, после чего цикл продолжается без участия оператора. В штатном режиме лидирующий сервер выдаёт команду назначения, в которой указывает идентификатор задачи и, при необходимости, содержит полезную нагрузку и бинарные данные библиотеки. В момент получения такого ответа воркер выполняет атомарную подготовку окружения: проверяет наличие требуемого модуля в `libs_dir`, при отсутствии записывает поступивший образ на диск и синхронно fsync'ит его, формируя детерминированное имя файла (например, по хешу содержимого или по име-

ни, пришедшему от сервера), после чего сверяет права доступа и готовность к загрузке. Этим достигается инвариант, согласно которому перед запуском пользовательского кода соответствующая библиотека обязательно присутствует локально и доступна для динамического связывания.

Переход от состояния ожидания к выполнению инициируется командой `PROCESS`. Получив её, узел запускает изолированный дочерний процесс исполняющей среды (в проекте такой процесс может создаваться простым `fork()` без сложной контейнеризации), загружает динамическую библиотеку из `libs_dir` и вызывает заранее оговорённый символ интерфейса, передавая входные данные из `task_data`. Основная служба воркера при этом остаётся в состоянии «занят» и периодически отправляет в поток `WorkerConnectRequest` собственный статус и актуальную проекцию `TaskState`, обеспечивая наблюдаемость прогресса. В случае штатного завершения исполняющий процесс формирует результат и отдаёт его основному процессу узла; далее результат включается в `TaskState` и возвращается в кластер тем же потоковым соединением. После этого вычислительный узел переводит себя в состояние `COMPLETED`, что служит для серверной стороны сигналом о готовности зафиксировать завершение в реплицированной машине состояний и инициировать доставку результата клиенту.

Особое внимание уделено устойчивости к частичным сбоям. Если в ходе выполнения падает дочерний процесс или попытка динамической загрузки модуля завершается ошибкой, основная служба не теряет связь с кластером: воркер немедленно фиксирует ошибочное завершение в `TaskState` и отправляет его через активный поток, после чего возвращается в состояние `IDLE`. Повторы команд со стороны сервера не приводят к дублированию работы: задача идентифицируется по `task_id`, поэтому воркер либо игнорирует повторные `PROCESS` для уже завершённой задачи, либо корректно восстанавливает исполнение, если сбой произошёл до выдачи финального состояния. Аналогично сетевые разрывы и смена лидера обрабатываются на транспортном уровне: поток может быть закрыт с соответствующим статусом gRPC, после чего узел повторяет подключение с публикацией своего фактического состояния, а серверная сторона, опираясь на зафиксированные в Raft переходы, восстанавливает согласованную картину происходящего.

Таким образом, вычислительный узел реализует минимально необходимую инфраструктуру для безопасного исполнения пользовательского кода: чтение конфигурации и подготовка каталога библиотек; устойчивое потоковое соединение с лидером кластера; детерминированная подготовка исполняемого окружения и загрузка модулей из `libs_dir`; изоляция пользовательского исполнения в отдельном процессе; регулярная публикация статуса и финализация результата через `WorkerService.Connect`. Все переходы по состояниям воркера (`IDLE` $\rightarrow$ `BUSY` $\rightarrow$ `COMPLETED`) отражаются в потоке запросов, а сервер отвечает соответствующими командами управления и данными (`ASSIGN`, `PROCESS`, `REDIRECT`), сохраняя линейризуемую семантику наблюдения и единый порядок событий для всех участников системы.

4 Ручное тестирование системы

4.1 Тестирование отказоустойчивости

Проверка корректности реализации алгоритма Raft проводилась с использованием ручных интеграционных испытаний. Для этого на локальной машине было запущено три экземпляра Raft-узлов, каждый из которых инициализировался собственной конфигурацией и прослушивал уникальный TCP-порт. На этапе инициализации узлы формируют сетевые потоки (`peer threads`), регистрируют gRPC-сервисы и начинают обмен heartbeat-сообщениями.

В первом эксперименте система демонстрирует штатное избрание лидера: через заданный таймаут по истечении срока аренды один из узлов инициировал выборы (лог содержит сообщение `Starting election in term 1`) и, получив кворум голосов, объявил себя лидером (`I'm the leader now`). В таком состоянии кластер готов принимать клиентские запросы, а закоммиченные записи будут реплицироваться на оставшиеся узлы. Лог мастера изображен на рис. 15. Лог одной из реплик же представлен на рис. 16.

```
[INFO][2025-09-18T23:11:46][INFO][core.cpp:Init]: Core initializing for 127.0.0.1:50050
[INFO][2025-09-18T23:11:46][INFO][abeille_raft.hpp:run]: Launching abeille-raft ...
[INFO][2025-09-18T23:11:46][INFO][core.cpp:Run]: Launching raft module...
[INFO][2025-09-18T23:11:46][INFO][peer.cpp:Run]: Starting peer thread for [127.0.0.1:50050]
[INFO][2025-09-18T23:11:46][INFO][peer.cpp:Run]: Starting peer thread for [127.0.0.1:50051]
[INFO][2025-09-18T23:11:46][INFO][peer.cpp:Run]: Starting peer thread for [127.0.0.1:50052]
[INFO][2025-09-18T23:11:46][INFO][raft_consensus.cpp:Run]: RaftConsensus for server 127.0.0.1:50050 was started
[INFO][2025-09-18T23:11:46][INFO][server.cpp:init]: registering services...
[INFO][2025-09-18T23:11:46][INFO][server.cpp:Run]: launching the server...
[INFO][2025-09-18T23:11:46][INFO][raft_consensus.cpp:peerThreadMain]: Peer thread for [127.0.0.1:50050] was started
[INFO][2025-09-18T23:11:46][INFO][raft_consensus.cpp:peerThreadMain]: Peer thread for [127.0.0.1:50052] was started
[INFO][2025-09-18T23:11:46][INFO][raft_consensus.cpp:peerThreadMain]: Peer thread for [127.0.0.1:50051] was started
[INFO][2025-09-18T23:11:46][INFO][state_machine.cpp:snapshotThreadMain]: Node's snaphoting is enabled
[INFO][2025-09-18T23:11:46][INFO][server.cpp:Run]: server is running
[INFO][2025-09-18T23:11:56][INFO][raft_consensus.cpp:startNewElection]: Starting election in term 1
[WARN][2025-09-18T23:11:56][WARN][raft_consensus.cpp:requestVote]: requestVote failed: Unable to communicate with
127.0.0.1:50052
[INFO][2025-09-18T23:11:56][INFO][raft_consensus.cpp:becomeLeader]: I'm the leader now (term 1)
[WARN][2025-09-18T23:11:56][WARN][raft_consensus.cpp:requestVote]: requestVote failed: Unable to communicate with
127.0.0.1:50051
[INFO][2025-09-18T23:12:32][INFO][raft_consensus.cpp:HandleRequestVote]: Rejecting RequestVote from
140736872206192 ... Heard from a leader recently
[WARN][2025-09-18T23:12:49][WARN][raft_consensus.cpp:appendEntry]: appendEntry failed: Unable to communicate
with 127.0.0.1:50052
```

Рисунок 15 – Лог мастера при первом выборе лидера

```
[INFO][2025-09-18T23:11:59][INFO][core.cpp:Init]: Core initializing for 127.0.0.1:50051
[INFO][2025-09-18T23:11:59][INFO][abeille_raft.hpp:run]: Launching abeille-raft ...
[INFO][2025-09-18T23:11:59][INFO][core.cpp:Run]: Launching raft module...
[INFO][2025-09-18T23:11:59][INFO][peer.cpp:Run]: Starting peer thread for [127.0.0.1:50050]
[INFO][2025-09-18T23:11:59][INFO][peer.cpp:Run]: Starting peer thread for [127.0.0.1:50051]
[INFO][2025-09-18T23:11:59][INFO][peer.cpp:Run]: Starting peer thread for [127.0.0.1:50052]
[INFO][2025-09-18T23:11:59][INFO][raft_consensus.cpp:Run]: RaftConsensus for server 127.0.0.1:50051 was started
[INFO][2025-09-18T23:11:59][INFO][server.cpp:init]: registering services...
[INFO][2025-09-18T23:11:59][INFO][server.cpp:Run]: launching the server...
[INFO][2025-09-18T23:11:59][INFO][raft_consensus.cpp:peerThreadMain]: Peer thread for [127.0.0.1:50050] was started
[INFO][2025-09-18T23:11:59][INFO][state_machine.cpp:snapshotThreadMain]: Node's snaphoting is enabled
[INFO][2025-09-18T23:11:59][INFO][raft_consensus.cpp:peerThreadMain]: Peer thread for [127.0.0.1:50051] was started
[INFO][2025-09-18T23:11:59][INFO][raft_consensus.cpp:peerThreadMain]: Peer thread for [127.0.0.1:50052] was started
[INFO][2025-09-18T23:11:59][INFO][server.cpp:Run]: server is running
[INFO][2025-09-18T23:12:03][INFO][raft_consensus.cpp:stepDown]: Stepping down to 1
[INFO][2025-09-18T23:12:03][INFO][raft_consensus.cpp:HandleAppendEntry]: New leader is [127.0.0.1:50050]
[INFO][2025-09-18T23:12:32][INFO][raft_consensus.cpp:HandleRequestVote]: Rejecting RequestVote from
140736912150896 ... Heard from a leader recently
```

Рисунок 16 – Лог реплики при первом выборе лидера

Затем моделировался отказ лидера: первый узел был остановлен, что привело к прекращению рассылки heartbeat-пакетов (см. рис. 17).

```
[WARN][2025-09-18T23:12:49][WARN][raft_consensus.cpp:appendEntry]: appendEntry failed: Unable to communicate
with 127.0.0.1:50052
^C[INFO][2025-09-18T23:19:56][INFO][core.cpp:Shutdown]: Shutting down all instances...
[INFO][2025-09-18T23:19:56][INFO][server.cpp:Shutdown]: Shutting down the server...
[INFO][2025-09-18T23:19:56][INFO][server.cpp:Shutdown]: Server shutdown successfully...
[INFO][2025-09-18T23:19:56][INFO][raft_consensus.cpp:Shutdown]: Shutting down raft...
[INFO][2025-09-18T23:19:56][INFO][raft_consensus.cpp:Shutdown]: Raft pool was shutted down
[INFO][2025-09-18T23:19:56][INFO][state_machine.cpp:Shutdown]: State machine was shutted down successfully
[INFO][2025-09-18T23:19:56][INFO][raft_consensus.cpp:Shutdown]: State machine was shutted down
[INFO][2025-09-18T23:19:56][INFO][raft_consensus.cpp:Shutdown]: Timer thread was shutted down
[INFO][2025-09-18T23:19:56][INFO][raft_consensus.cpp:Shutdown]: Waiting for threads to finish...
[INFO][2025-09-18T23:19:56][INFO][raft_consensus.cpp:Shutdown]: All peer's threads have been shutted down
```

Рисунок 17 – Логи при остановке мастера

Спустя период `election timeout` один из оставшихся узлов инициировал новый раунд выборов (терм 2), получил поддержку большинства и стал новым лидером. В логах зафиксировано событие `becomeLeader` с указанием нового терма, что подтверждает корректную работу механизма смены лидерства и сохранение согласованности состояния кластера, это представлено на рис. 18.

```
[INFO][2025-09-18T23:12:52][INFO][raft_consensus.cpp:stepDown]: Stepping down to 1
[INFO][2025-09-18T23:12:52][INFO][raft_consensus.cpp:HandleAppendEntry]: New leader is [127.0.0.1:50050]
[INFO][2025-09-18T23:20:02][INFO][raft_consensus.cpp:startNewElection]: Starting election in term 2 (haven't heard
from leader 140736954430128 lately
[WARN][2025-09-18T23:20:02][WARN][raft_consensus.cpp:requestVote]: requestVote failed: Unable to communicate
with 127.0.0.1:50050
[INFO][2025-09-18T23:20:02][INFO][raft_consensus.cpp:becomeLeader]: I'm the leader now (term 2)
```

Рисунок 18 – Логи нового мастера после остановки старого

Процесс голосования реплики за нового лидера представлен на рис. 19.

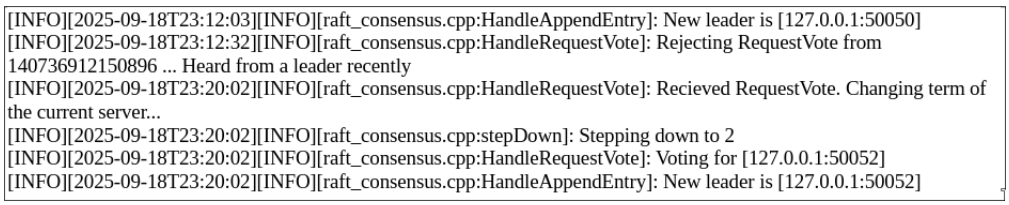

Рисунок 19 – Процесс голосования за нового лидера

Такие испытания позволяют убедиться, что система корректно реагирует на сбои: потеря лидера не приводит к недоступности сервиса, а лишь инициирует повторные выборы, после которых кластер восстанавливает способность обрабатывать запросы. В сочетании с механикой репликации и commit-индексов это гарантирует, что все закоммиченные записи остаются видимыми для клиента даже при перезапуске узлов.

4.2 Инициализация клиентского приложения и подключение к кластеру

Тестирование вычислительного процесса начинается с запуска консольного клиента с указанием конфигурационных файлов пользователя и кластера. На рис. 20 приведён фрагмент журнала запуска клиента. В ходе подключения клиент последовательно обходит адреса, указанные в `client_config.json`. Поскольку первый узел кластера (`127.0.0.1:50050`) в момент запуска был недоступен, попытка установления соединения завершилась ошибкой, что зафиксировано в логах с уровнем `ERROR`. После истечения интервала переподключения клиент автоматически переходит к следующему адресу и успешно устанавливает соединение с узлом `127.0.0.1:50051`, а затем и с `127.0.0.1:50052`. Такая стратегия повышает отказоустойчивость системы: пользователь не обязан вручную выбирать рабочий узел, так как клиент сам находит доступного участника кластера.

После установления соединения клиент переходит в интерактивный режим, предлагая пользователю список доступных команд. В частности, доступны функции загрузки данных из файла или директории (`ud`, `udd`), вывод справочной информации и завершение работы программы. На этом этапе система готова к приёму задания и дальнейшему тестированию логики репликации и обработки задач.

```
./build/bin/user_client examples/config/user_config.json examples/config/client_config.json
abeille> [ERROR][2025-09-19T00:24:11][ERROR][client.hpp:connect]: failed to connect to [127.0.0.1:50050]
[ERROR][2025-09-19T00:24:12][ERROR][client.hpp:connect]: failed to connect to [127.0.0.1:50050]
[INFO][2025-09-19T00:24:13][INFO][client.hpp:connect]: successfully connected to [127.0.0.1:50051]
[INFO][2025-09-19T00:24:13][INFO][client.hpp:reconnect]: reconnecting to [127.0.0.1:50052]
[INFO][2025-09-19T00:24:13][INFO][client.hpp:connect]: successfully connected to [127.0.0.1:50052]
help
-- ud <filepath>      uploads data from <filepath> to the raft cluster
-- udd <dirpath>      uploads data from <dirpath> to the raft cluster
-- help               lists available functions
-- exit               terminates the programm
abeille>
```

Рисунок 20 – Журнал запуска клиентского приложения: обработка недоступности узла и успешное подключение к кластеру.

4.3 Запуск вычислительных узлов и их подключение к кластеру

После инициализации клиентской части производится запуск вычислительных узлов, которые будут непосредственно выполнять задачи.Логи на рис. 21 демонстрируют, что, как и в случае с клиентом, воркер поочерёдно пытается подключиться к указанным в `client_config.json` узлам. Первая пара попыток к `127.0.0.1:50050` завершилась ошибкой соединения, после чего воркер переключился на следующий адрес и успешно установил потоковое RPC-соединение с `127.0.0.1:50051`, а затем и с лидером `127.0.0.1:50052`.

```
./build/bin/worker_client examples/config/worker_config.json examples/config/client_config.json
[ERROR][2025-09-19T00:29:33][ERROR][client.hpp:connect]: failed to connect to [127.0.0.1:50050]
[ERROR][2025-09-19T00:29:34][ERROR][client.hpp:connect]: failed to connect to [127.0.0.1:50050]
[INFO][2025-09-19T00:29:35][INFO][client.hpp:connect]: successfully connected to [127.0.0.1:50051]
[INFO][2025-09-19T00:29:35][INFO][client.hpp:reconnect]: reconnecting to [127.0.0.1:50052]
[INFO][2025-09-19T00:29:35][INFO][client.hpp:connect]: successfully connected to [127.0.0.1:50052]
```

Рисунок 21 – Логи запуска вычислительного узла: автоматический обход узлов и успешное подключение.

После установления соединения воркер переходит в состояние `IDLE`, публикуя свой статус в поток `WorkerService.Connect`, что позволяет кластеру учитывать его при назначении новых задач. Таким образом, уже на этом этапе проверяется важная функциональность системы — автоматический выбор доступного узла и готовность воркера принимать задания от лидера кластера.

4.4 Тестирование выполнения задания

Для проверки корректности работы системы был проведён end-to-end тест с использованием простого вычислительного задания. В качестве тестовой функции был выбран алгоритм факторизации числа на простые множители, реализованный на C++. Такая задача является хорошим примером для функционального тестирования: она не требует сложной подготовки данных, но даёт детерминированный и легко проверяемый результат. Код задания представлен в приложении В. В качестве входных данных подавались 10000 чисел.

Постановка задачи выполнялась из интерактивной консоли клиентского приложения с помощью команды `ud`, передающей содержимое `data.json` в кластер. Это представлено на рис. 22. Клиент подтвердил успешную загрузку данных, а после обработки задачи вывел сообщение о получении результата.

```
abeille> ud examples/user_data/data.json
successfully uploaded task data
abeille> [INFO][2025-09-19T00:47:21][INFO][user_client.cpp:handleCommandResult]: received result for [data.json]
```

Рисунок 22 – Процесс пересылки данных на клиенте

На стороне лидера Raft логи показан полный путь задачи: назначение её конкретному воркеру, репликацию соответствующей команды в Raft-лог, достижение кворума и коммит, а также последующую фиксацию результата (см. рис. 23). При этом реплика синхронизировалась с лидером, что видно по сообщениям `HandleAppendEntry` и обновлению индексов коммита (см. рис. 24).

```
[INFO][2025-09-18T23:20:02][INFO][raft_consensus.cpp:becomeLeader]: I'm the leader now (term 2)
[DEBUG][2025-09-19T00:47:17][DEBUG][task_manager.cpp:UploadTaskData:29]: assigned task#[data.json] to
[127.0.0.1:60420]
[DEBUG][2025-09-19T00:47:19][DEBUG][worker_service.cpp:ProcessTask:133]: successfully sent [data.json] task to
[127.0.0.1:60420]
[INFO][2025-09-19T00:47:19][INFO][state_machine.cpp:Commit]: Updating commit index from 0 to 1
[INFO][2025-09-19T00:47:20][INFO][worker_service.cpp:handleStatusCompleted]: worker has finished task#[data.json]
[DEBUG][2025-09-19T00:47:20][DEBUG][task_manager.cpp:UploadTaskResult:51]: uploading task result...
[INFO][2025-09-19T00:47:20][INFO][state_machine.cpp:Commit]: Updating commit index from 1 to 2
```

Рисунок 23 – Лог мастера при назначении задания

Логи вычислительного узла подтвердили успешное получение задания: воркер перешёл в состояние `BUSY`, инициировал локальный процесс обработки данных через IPC, корректно завершил вычисление и вернул результат обратно в кластер (см. рис. 25).

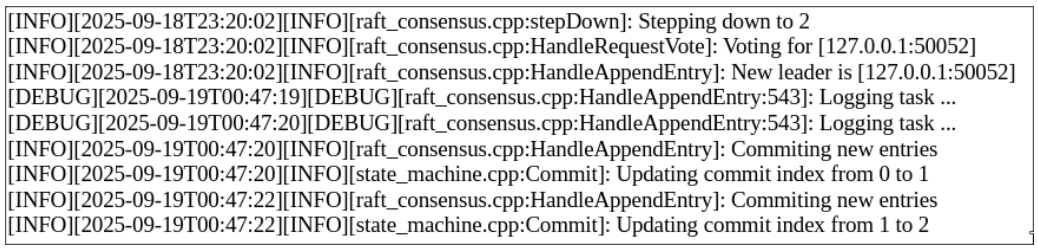

Рисунок 24 – Лог реплики при назначении задания

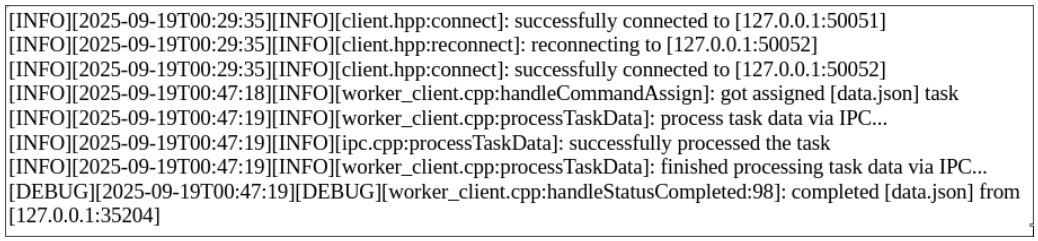

Рисунок 25 – Лог вычислительного узла при выполнении работы

На стороне мастера была зафиксирована команда `RAFT_COMMAND_MOVE`, переведшая задачу в статус завершённой, после чего результат был передан пользовательскому клиенту. Таким образом, в рамках теста была проверена вся цепочка обработки: от приёма входных данных до финализации состояния в машине состояний и возврата результата.

Проведённый эксперимент подтверждает корректность реализации основных механизмов системы: клиентские запросы реплицируются и коммитятся только при достижении кворума, воркеры получают задачи и публикуют статус их выполнения, а пользователь всегда наблюдает согласованное состояние кластера, даже при ранее проведённых сменах лидера или мертвых узлах.

4.5 Нагрузочный тест с 50 параллельными клиентами

Для проверки поведения системы при высокой конкурентной нагрузке был организован сценарий с одновременной работой пятидесяти клиентских процессов. Каждый клиент открывал двунаправленное gRPC-соединение с кластером, отправлял одну команду `ud` для постановки задачи в очередь и завершал работу, освобождая слот для следующих запусков. Параллельный запуск был реализован средствами утилиты `GNU parallel` (см. рис. 26), обеспечивающей одновременный запуск 50 процессов с минимальной задержкой между стартами.

```
yes "examples/user_data/data.json" | head -n 50 | \
  parallel -j50 --delay 0.02 --line-buffer "
./build/bin/user_client examples/config/user_config.json examples/config/client_config.json <<'EOF'
ud {}
EOF
```

Рисунок 26 – Нагрузочный тест

В данном тесте использовались два рабочих узла (воркера). Кластер Raft выбрал лидера в начале прогона, после чего последовательно обрабатывал поступающие задания. В журнале лидера наблюдается монотонное увеличение индекса коммита каждой новой записи лога (рис. 27), что подтверждает корректную репликацию команд и достижение кворума. Каждое успешно завершённое задание сопровождается сообщением `worker_service.cpp:handleStatusCompleted`, после чего в машине состояний фиксируется переход задачи из состояния `assigned` в `completed`.

```
[INFO][2025-09-19T11:51:32][INFO][raft_consensus.cpp:becomeLeader]: I'm the leader now (term 1)
[INFO][2025-09-19T11:51:58][INFO][state_machine.cpp:Commit]: Updating commit index from 0 to 1
[INFO][2025-09-19T11:51:59][INFO][worker_service.cpp:handleStatusCompleted]: worker has finished task#[data.json]
[INFO][2025-09-19T11:52:00][INFO][state_machine.cpp:Commit]: Updating commit index from 1 to 2
[INFO][2025-09-19T11:52:01][INFO][worker_service.cpp:handleStatusCompleted]: worker has finished task#[data.json]
[INFO][2025-09-19T11:52:02][INFO][state_machine.cpp:Commit]: Updating commit index from 2 to 3
<...>
[INFO][2025-09-19T12:03:20][INFO][state_machine.cpp:Commit]: Updating commit index from 96 to 97
[INFO][2025-09-19T12:03:22][INFO][state_machine.cpp:Commit]: Updating commit index from 97 to 98
[INFO][2025-09-19T12:03:23][INFO][state_machine.cpp:Commit]: Updating commit index from 98 to 99
[INFO][2025-09-19T12:03:25][INFO][state_machine.cpp:Commit]: Updating commit index from 99 to 100
[INFO][2025-09-19T12:03:25][INFO][state_machine.cpp:snapshot]: Snapshotting...
[INFO][2025-09-19T12:03:25][INFO][state_machine.cpp:updateLog]: Snapshotted successfully. Dropped 10 entries from log
```

Рисунок 27 – Фрагмент лога лидера во время нагрузочного теста с 50 клиентами

При достижении сотого коммита сработал механизм создания снепшота, что подтверждается сообщением `Snapshotting...` в логах. После успешного сохранения состояния машины состояний на диск (`Snapshotted successfully`) из лога были удалены 10 записей, как и предусмотрено параметром `snapshot_after` в конфигурации. Продолжение работы кластера после создания снепшота не вызвало увеличения задержек или ошибок репликации, что свидетельствует о корректности реализации механизма архивации и его прозрачности для клиентских приложений.

Таким образом, тест показал, что система способна обрабатывать десятки одновременных подключений, сохраняя линейризуемость состояния и выполняя своевременное архивирование лога без нарушения доступности сервиса.

4.6 Тестирование на задаче с большим объёмом данных

Для проверки корректности работы системы при выполнении ресурсоёмких задач был проведён эксперимент с входными данными увеличенного размера (около двух миллионов элементов). Задача была отправлена в кластер с использованием стандартного клиента и поступила на один из доступных рабочих узлов. Фрагмент лога воркера приведён на рис. 28.

```
[INFO][2025-09-19T12:16:52][INFO][worker_client.cpp:handleCommandAssign]: got assigned [big_data.json] task
[INFO][2025-09-19T12:16:53][INFO][worker_client.cpp:processTaskData]: process task data via IPC…
[INFO][2025-09-19T12:17:48][INFO][ipc.cpp:processTaskData]: successfully processed the task
[INFO][2025-09-19T12:17:48][INFO][worker_client.cpp:processTaskData]: finished processing task data via IPC..
```

Рисунок 28 – Лог вычислительного узла при обработке задачи с 2 млн элементов

Как видно из лога, узел получил команду назначения задачи (`handleCommandAssign`), после чего перешёл к фазе обработки данных через IPC-механизм (`processTaskData`). Общая длительность обработки составила около 56 секунд, что соответствует ожидаемому времени выполнения для задачи данного объёма. По завершении вычислений задача была успешно отмечена как выполненная (`finished processing task data via IPC`), а результат был передан обратно в кластер и зафиксирован в машине состояний.

Данный эксперимент подтвердил, что система корректно обрабатывает задачи увеличенной сложности и объёма, при этом не теряя связи с кластером и сохраняя линейризуемость состояния. В течение всего времени выполнения других ошибок или повторных назначений задачи не наблюдалось, что демонстрирует устойчивость и надёжность реализации даже при продолжительных вычислениях.

ЗАКЛЮЧЕНИЕ

В рамках проведённого исследования была разработана и реализована распределённая система вычислений, использующая алгоритм консенсуса Raft для обеспечения согласованности состояния между узлами. Работа стала логическим продолжением предыдущего научно-исследовательского проекта, в котором была построена и формально проверена спецификация алгоритма Raft на языке TLA+. В настоящей работе сделан следующий шаг — разработана практическая реализация системы, включающая модуль Raft, реплицированную машину состояний с поддержкой снепшотов, клиентскую часть для постановки задач и воркеры для их выполнения. Проведено ручное тестирование кластера, включающее сценарии смены лидера, репликации команд и обработку пользовательских задач, что подтвердило корректность реализации и её устойчивость к отказам.

При этом стратегическая цель проекта выходит за рамки одной реализации: создать фреймворк для C/C++, ориентированный на промышленную интеграцию (в первую очередь в СУБД *Tarantool*), который обеспечит непрерывную связь между формальной спецификацией и реальным кодом. Предыдущая НИРС дала нам спецификацию Raft на TLA+; текущая работа — рабочую систему. Следующий шаг — соединить эти два мира инженерным инструментом, чтобы спецификация не «устаревала» по мере эволюции кода и чтобы расхождения детектировались автоматически на уровне трасс исполнения.

Планируемый фреймворк будет собирать во время тестов трассы состояний реальной системы (в debug-сборках и при управляемом однонитевом исполнении): выбранные переменные и внутренние маркеры будут помечаться препроцессором, а их значения фиксироваться между *yield*/точками квазидетерминизма в лог-файл. Затем трасса преобразуется в абстракцию, сопоставимую со стейтами TLA+, и сравнивается с поведением модели; отсутствие соответствующей траектории в спецификации трактуется как сигнал к обновлению модели или кодификации недостающих предпосылок. Для тяжёлых эффектов (например, запись на диск) допускается замещение «сложных» функций упрощёнными переходами состояния — это позволит отделить функциональную корректность от особенностей подсистем ввода-вывода и повысить наблюдаемость. Концептуально подход опирается на идеи трасс/проекций состояний и синхронизации модели с реализацией [5—7].

Интеграция в *Tarantool* предполагает два направления: во-первых, инжектируемые C/C++–хуки и лёгкие макросы для маркировки наблюдаемых переменных и точек переходов в существующем коде ядра; во-вторых, конвейер тестирования (unit, интеграционные, стохастические тесты), который автоматически собирает и верифицирует трассы в CI. Такой процесс создаёт контур *continuous verification*: каждая регрессия и каждое изменение протокола отражаются одновременно в кодовой базе и в TLA+-спецификации, снижая стоимость расхождений «модель–код» и ускоряя обратную связь для разработчиков.

Таким образом, проделанная работа сформировала инженерный фундамент (реализация Raft) и одновременно задала методологический вектор: переход от разовой формальной проверки к систематической, автоматизированной и интегрированной в жизненный цикл разработки. Разработка C/C++-фреймворка для синхронизации спецификации и кода и его последующая интеграция в *Tarantool* является естественным и практически значимым продолжением проекта, направленным на повышение надёжности, воспроизводимости и эволюционной управляемости распределённых компонентов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Ongaro, Diego, and John Ousterhout. In search of an understandable consensus algorithm // In Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference. — 2006. — С. 305—320.

2. Distributed Systems: Concepts and Design / G. Coulouris [и др.]. — 5-е изд. — Addison-Wesley/Pearson, 2012. — ISBN 9780137521081. — URL: https://www.pearson.com/en-us/subject-catalog/p/distributed-systems-concepts-and-design/P200000003160/9780137521081.

3. Lynch N. Distributed Algorithms. — Morgan Kaufmann, 1996. — ISBN 9781558603486. — URL: https://shop.elsevier.com/books/distributed-algorithms/lynch/978-1-55860-348-6.

4. Железцов Н.В. М. О. abeille: система распределенных вычислений с Raft. — 2025. — URL: https://github.com/Serpentian/abeille/tree/master (дата обращения 15.09.2025) ; Исходный код проекта.

5. Pron A. Trace Checking for Distributed Systems. — 2023. — URL: https://pron.github.io/files/Trace.pdf ; дата обращения: 2025-05-14.

6. From Code to Specs: Assessing TLA+ Coverage / S. Chatterjee [и др.]. — 2024. — arXiv: 2404.16075 [cs.SE]. — URL: https://arxiv.org/abs/2404.16075 ; дата обращения: 2025-05-14.

7. Merz S. Bridging TLA+ and Implementations. — 2024. — URL: https://conf.tlapl.us/2024-fm/slides-merz.pdf ; дата обращения: 2025-05-14.

ПРИЛОЖЕНИЕ А

Протокол распределенной вычислительной системы

Листинг А.1 – Protocol Buffers

```
syntax = "proto3";

// ---------------------------- Common ---------------------------- //

message TaskID {
  string filename = 1;
  uint64 client_id = 2;
}

message TaskState {
  TaskID task_id = 1;
  bytes task_result = 2;
}

// ---------------------------- Raft Entry ---------------------------- //

message TaskWrapper {
  TaskID task_id = 1;
  uint64 worker_id = 2;
  bytes task_data = 3;
  bytes task_result = 4;
  bytes lib = 5;
}

enum TaskStatus {
  TASK_STATUS_ASSIGNED = 0;
  TASK_STATUS_COMPLETED = 1;
}

message AddRequest { TaskStatus to = 1; }

message MoveRequest {
  TaskStatus to = 1;
  TaskStatus from = 2;
}

enum RaftCommand {
  RAFT_COMMAND_ADD = 0;
  RAFT_COMMAND_MOVE = 1;
}

// Log entry to store
message Entry {
  RaftCommand command = 1;
  uint64 term = 2;
  TaskWrapper task_wrapper = 3;
  AddRequest add_request = 4;
  MoveRequest move_request = 5;
}

// ---------------------------- Raft Service ---------------------------- //

// Main raft servers service
service RaftService {
  rpc AppendEntry(AppendEntryRequest) returns (AppendEntryResponse) {}
  rpc RequestVote(RequestVoteRequest) returns (RequestVoteResponse) {}
  rpc InstallSnapshot(InstallSnapshotRequest)
                                      returns (InstallSnapshotResponse) {}
}

// Invoked by leader to replicate log entries
```

```
// Also used as heartbeat
message AppendEntryRequest {
  uint64 term = 1;           // leader's term
  uint64 leader_id = 2;      // so follower can redirect clients
  uint64 prev_log_index = 3; // index of log entry preceding new ones
  uint64 prev_log_term = 4;  // term of prev_log_index
  Entry entry = 5;           // log entry to store (empty for heartbeat)
  uint64 leader_commit = 6;  // leader's commit index
}

message AppendEntryResponse {
  uint64 term = 1;           // current term for leader to update itself
  bool success = 2;          // true if follower contained entry
                             // matching prev_log_index and prev_log_term
}

message RequestVoteRequest {
  uint64 term = 1;           // candidate's term
  uint64 candidate_id = 2;   // candidate requesting vote
  uint64 last_log_entry = 3; // index of candidate's log entry
  uint64 last_log_term = 4;  // term of candidate's log entry
}

message RequestVoteResponse {
  uint64 term = 1;           // current term for candidate to update itself
  bool vote_granted = 2;     // true if candidate recieved vote
}

message InstallSnapshotRequest {
  uint64 term = 1;
  uint64 leader_id = 2;
  uint64 last_index = 3;
  uint64 last_term = 4;
  Snapshot snapshot = 5;
}

message InstallSnapshotResponse {
  uint64 term = 1;
  bool success = 2;
}

// ------------------------------ User Service ------------------------------ //

enum UserStatus {
  USER_STATUS_DOWN = 0;
  USER_STATUS_IDLE = 1;
  USER_STATUS_UPLOAD_DATA = 2;
}

message UserConnectRequest {
  UserStatus status = 1;
  bytes task_data = 2;
  string filename = 3;
  bytes lib = 4;
}

enum UserCommand {
  USER_COMMAND_NONE = 0;
  USER_COMMAND_REDIRECT = 1;
  USER_COMMAND_ASSIGN = 2;
  USER_COMMAND_RESULT = 3;
}

message UserConnectResponse {
  UserCommand command = 1;
  uint64 leader_id = 2;
  TaskState task_state = 3;
}
```

```
service UserService {
  rpc Connect(stream UserConnectRequest) returns (stream UserConnectResponse) {}
}

// ---------------------------- Worker Service ---------------------------- //

enum WorkerStatus {
  WORKER_STATUS_DOWN = 0;
  WORKER_STATUS_IDLE = 1;
  WORKER_STATUS_BUSY = 2;
  WORKER_STATUS_COMPLETED = 3;
}

message WorkerConnectRequest {
  WorkerStatus status = 1;
  TaskState task_state = 2;
}

enum WorkerCommand {
  WORKER_COMMAND_NONE = 0;
  WORKER_COMMAND_ASSIGN = 1;
  WORKER_COMMAND_PROCESS = 2;
  WORKER_COMMAND_REDIRECT = 3;
}

message WorkerConnectResponse {
  WorkerCommand command = 1;
  uint64 leader_id = 2;
  TaskID task_id = 3;
  bytes task_data = 4;
  bytes lib = 5;
}

service WorkerService {
  rpc Connect(stream WorkerConnectRequest) returns (stream WorkerConnectResponse) {}
}

// ---------------------------- Snapshot ---------------------------- //

message Snapshot {
  uint64 last_index = 1;
  uint64 last_term = 2;
  // Crutch: key in map cannot be msg type
  repeated TaskID assigned_ids = 3;
  repeated TaskWrapper assigned_tasks = 4;
  repeated TaskID completed_ids = 5;
  repeated TaskWrapper completed_tasks = 6;
}
```

ПРИЛОЖЕНИЕ Б

Полная UML диаграмма классов проекта

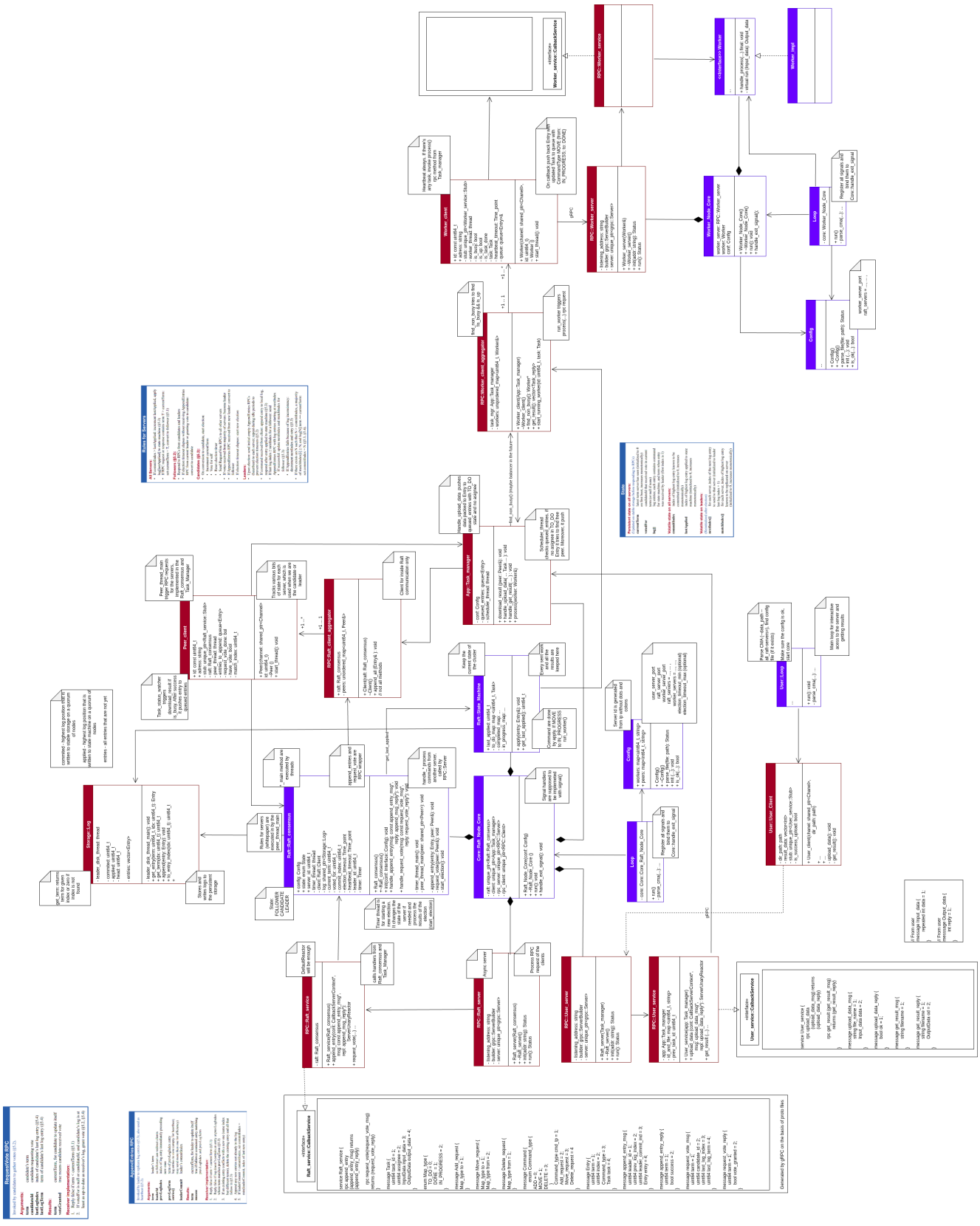

Рисунок Б.1 – UML-диаграмма классов

ПРИЛОЖЕНИЕ В

Тестовое задание.

Листинг В.2 – Разложение чисел на простые множители

```
#include "user/include/task.hpp"

#include <cmath>
#include <sstream>
#include <vector>

namespace abeille {
namespace user {

std::vector<int> primeFactors(int n) {
  std::vector<int> prime_factors;
  while (n % 2 == 0) {
    n /= 2;
    prime_factors.push_back(2);
  }

  for (int i = 3; i <= std::sqrt(n); i = i + 2) {
    while (n % i == 0) {
      n = n / i;
      prime_factors.push_back(i);
    }
  }

  if (n > 2) {
    prime_factors.push_back(n);
  }

  return prime_factors;
}

error ProcessTaskData(const Task::Data &task_data, Task::Result &task_result) {
  for (int elem : task_data.data()) {
    auto result = task_result.add_result();
    result->set_number(elem);

    for (int factor : primeFactors(elem)) {
      result->add_prime_factors(factor);
    }
  }

  return error();
}

}  // namespace user
}  // namespace abeille
```