МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ
Кафедра компьютерных технологий и систем

ГАТАЛЬСКИЙ Антон Игоревич

БЛОКЧЕЙН КАК ИНСТРУМЕНТ ОПТИМИЗАЦИИ ЛОГИСТИЧЕСКИХ ПРОЦЕССОВ

Дипломная работа

Научный руководитель:
старший преподаватель
С.С. Соловей

Допущена к защите

«____»_____20___г.

Заведующий кафедрой компьютерных технологий и систем
доктор педагогических наук, кандидат физико-
математических наук, профессор В.В. Казачёнок

Минск, 2025

ОГЛАВЛЕНИЕ

РЕФЕРАТ

Дипломная работа, 73 страницы, 32 рисунка, 2 таблицы, 12 приложений, 13 источников

ТЕХНОЛОГИЯ БЛОКЧЕЙН, ДЕЦЕНТРАЛИЗАЦИЯ, СМАРТ-КОНТРАКТ, ЛОГИСТИКА, ЦЕПОЧКИ ПОСТАВОК, ЭЛЕКТРОННЫЙ ДОКУМЕНТООБОРОТ, ОПТИМИЗАЦИЯ

Объект исследования – логистические процессы и их оптимизация с использованием блокчейна.

Цель исследования – разработка децентрализованного приложения для оптимизации логистического документооборота и повышения прозрачности цепочек поставок.

Методы исследования – изучение литературы, посвященной технологии блокчейн и разработке децентрализованных веб приложений. Анализ существующих решений в этом направлении. Изучение рынка и проблем логистики.

Полученные результаты – децентрализованное веб приложение, позволяющее напрямую обмениваться логистическими документами, а также создавать, назначать, отслеживать и завершать перевозки.

Область возможного практического применения – разработанное приложение можно использовать в разных сферах логистики, но особо актуально для международных перевозок, с большим объемом документации, а также для компаний со скоропортящимися товарами, где важен вопрос прозрачности цепочки поставок.

РЭФЕРАТ

Дыпломная работа, 73 старонкі, 32 малюнка, 2 табліцы, 12 дадаткаў, 13 крыніц

ТЭХНАЛОГІЯ БЛАКЧЭЙН, ДЭЦЭНТРАЛІЗАЦЫЯ, СМАРТ-КАНТРАКТ, ЛАГІСТЫКА, ЛАНЦУЖКІ ПАСТАВАК, ЭЛЕКТРОННЫ ДАКУМЕНТАЗВАРОТ, АПТЫМІЗАЦЫЯ

Аб'ект даследавання – лагістычныя працэсы і іх аптымізацыя з выкарастаннем блокчэйна.

Мэта даследавання – распрацоўка дэцэнтралізаванага дадатку для аптымізацыі лагістычнага дакументазвароту і павышэння празрыстасці ланцужкоў паставак.

Метады даследавання – вывучэнне літаратуры, прысвечанай тэхналогіі блокчэйн і распрацоўцы дэцэнтралізаваных вэб прыкладанняў. Аналіз існуючых рашэнняў у гэтым напрамку. Вывучэнне рынку і праблем лагістыкі.

Атрыманыя вынікі – дэцэнтралізаваны вэб дадатак, які дазваляе напрамую абменьвацца лагістычнымі дакументамі, а таксама ствараць, прызначаць, адсочваць і завяршаць перавозкі.

Вобласць магчымага практычнага прымянення – распрацаванае прыкладанне можна выкарыстоўваць у розных сферах лагістыкі, але асабліва актуальна для міжнародных перавозак, з вялікім аб'ёмам дакументацыі, а таксама для кампаній са скорапсавальных таварамі, дзе важнае пытанне празрыстасці ланцужкі паставак.

SUMMARY

Diploma work, 73 pages, 32 figures, 2 tables, 12 appendixes, 13 references

BLOCKCHAIN TECHOLOGY, DECENTRALIZATION, SMART-CONTRACT, SUPPLY CHAINS, ELECTRONIC DOCUMENT MANAGMENT, OPTIMIZATION

The object of the research – logistics processes and their optimization through blockchain technology.

The aim of the research – development of a decentralized application to optimize logistics document workflows and enhance supply chain transparency.

Research methods – study of literature on blockchain technology and the development of decentralized web applications. Analysis of existing solutions in this domain. Examination of the market and logistics challenges.

The results of the work – a decentralized web application enabling direct exchange of logistics documents, as well as creation, assignment, tracking, and completion of shipments.

Recommendations on the usage – the developed application can be applied across various logistics sectors, but is especially valuable for international shipments with large documentation volumes and for companies handling perishable goods, where supply chain transparency is crucial.

ВВЕДЕНИЕ

Современный мир требует анализа новых подходов в оптимизации новых систем, включая логистику. Логистика является важной частью мировой экономики, она обеспечивает перемещение товаров, информации и ресурсов. Но, не смотря на важность сферы, существует ряд нерешенных проблем: недостаточная прозрачность, сложная коммуникация, избыточный документооборот и риски мошенничества. Одно из их решений – блокчейн.

Блокчейн – способ хранения данных, имеющий высокий уровень защиты и прозрачность. Технология имеет множество применений в разных сферах: финансы, недвижимость и логистика.

Данная работа посвящена использованию блокчейна для оптимизации логистических процессов.

Целью дипломной работы является изучение блокчейна для применения в логистике и разработка приложения для прямого взаимодействия перевозчиков и заказчиков.

Для достижения цели были поставлены следующие задачи:

1. Изучить архитектуру блокчейна, выделить преимущества и недостатки.
2. Выделить наиболее подходящие процессы для применения блокчейна.
3. Проанализировать существующие децентрализованные решения логистических компаний.
4. Разработать децентрализованное приложение для прямого взаимодействия заказчиков и перевозчиков.

ГЛАВА 1
ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

1.1 Принципы и механизмы работы технологии блокчейн

1.1.1 Понятие блокчейна, структура блока

Рассмотрим устройство и принципы работы технологии блокчейн для понимания ее возможностей в сфере логистики. Важно отметить, что блокчейн это технология, имеющая множество реализаций, в рамках дипломной работы будут исследованы самые известные сети – Bitcoin, Ethereum и Hyperledger Fabric.

Блокчейн – децентрализованная база данных, которая содержит цепочку блоков, содержащих транзакции, связанные между собой при помощи хеша, где корректность каждого блока проверяется всеми участниками сети [4].

Ключевым понятием блокчейна является блок – файл, содержащий транзакции и другие компоненты. Блок состоит из заголовка и тела блока. Тело блока содержит информацию о всех транзакциях. В зависимости от сети, содержимое заголовка блока может меняться. На рисунке 1.1 изображено содержимое блоков сети Ethereum. Но вне зависимости от реализации, следующие поля являются основой и присутствуют в любой сети:

- собственный хеш, вычисленный для транзакций внутри блока;
- хеш блока предшественника;
- nonce (код используемый для подбора хеша);
- время добавление блока в сеть (timestamp);
- номер и размер блока;

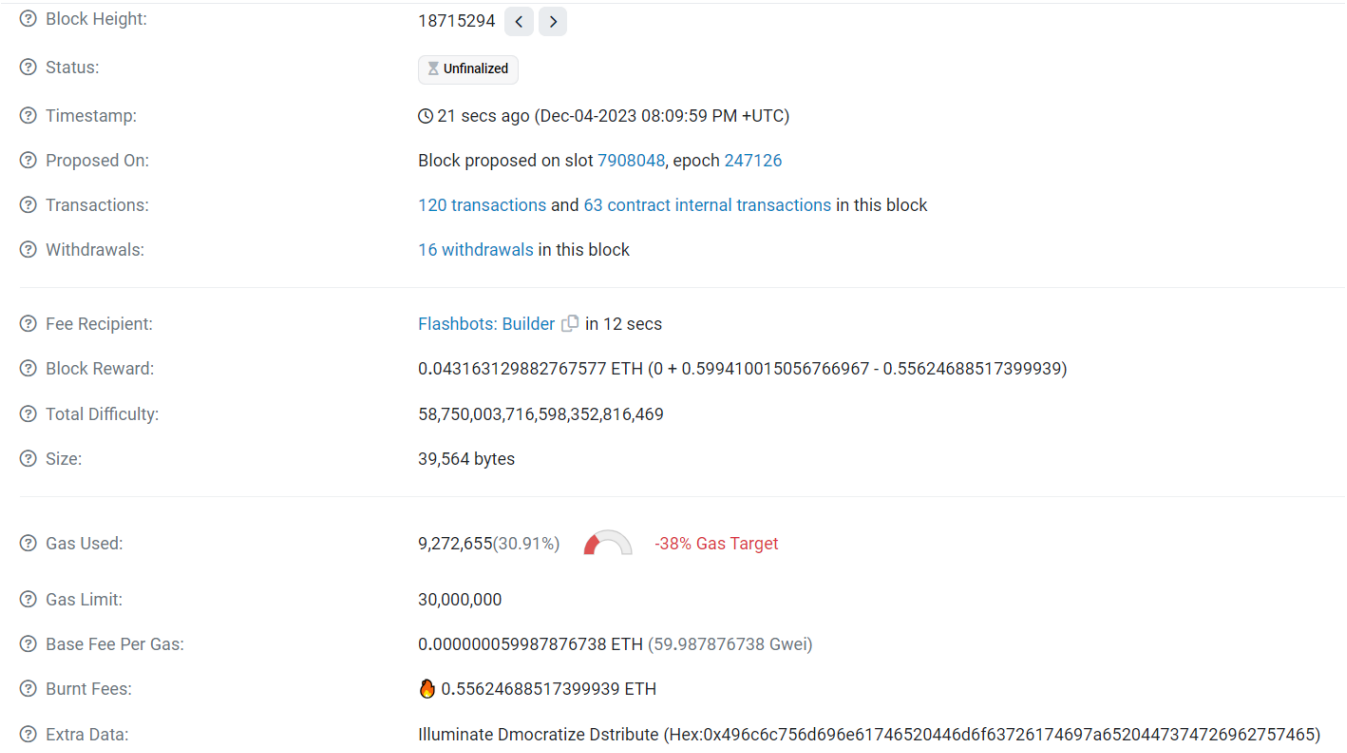

Рисунок 1.1 – Содержимое блока сети Ethereum

1.1.2 Цепь блоков

Блоки объединяются в непрерывную, упорядоченную цепочку, каждый блок которой хранит сведения о предыдущем блоке. На рисунке 1.2 представлена схема блоков в цепи.

Рисунок 1.2 – Схема блоков в цепи

Существует две модели появления новых блоков в сети:

1. Временная модель. При использовании такой модели блоки в сети появляются через определенный промежуток времени, такая модель реализована в Ethereum, где блоки выпускаются раз в 12 секунд.
2. Событийно-ориентированная модель. Данная модель производит блоки без привязки ко времени, а зависит от сложности и случайности

процесса обработки блока, такая модель реализована в Bitcoin, где среднее время выпуска блока 10 минут, но может и значительно отличаться.

Процесс добавления нового блока в цепь можно разделить на следующие шаги:

1. Генерация блока – новый блок генерируется в системе при наступлении определенных условий, например, при достижении консенсуса.
2. Подтверждение транзакций – этот этап может включать проверку подписей, балансов и других деталей.
3. Распространение блока по сети – после генерации блока и подтверждения транзакций, блок распространяется по сети, таким образом, чтобы каждый узел имел актуальную цепочку блоков.

Отметим, что цепочка блоков может ветвится, пример цепочки с ветвлением на рисунке 1.3. Это может быть сделано намеренно, например, для создания локальной цепи, либо случайно, из-за сбоев системы. Локальная сеть может быть полезна для корпоративных целей. Если создание блоков возобновляется, основным считается вариант цепи с большим количеством блоков.

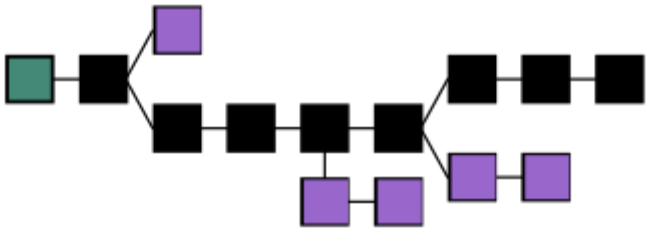

Рисунок 1.3 – Цепочка блоков с ветвлением

1.1.3 Децентрализация

Блокчейн состоит из сети «узлов» (компьютеров), каждый из которых содержит копию блокчейна. Из-за того, что блокчейн хранит всю историю транзакций, со временем, он становится достаточно громоздким (на конец 2022 года размер всей сети Биткоина составлял порядка 437 Гб), что делает поддержку всей сети на мобильных устройствах или слабых компьютерах проблемой. Для ее решения, существуют узлы двух видов: full nodes, узлы, которые содержат все транзакции и все заголовки блоков и lite nodes, облегченные узлы, которые не хранят самую затратную часть – тело блока.

Так как копии хранятся у каждого участника сети, реализуется принцип децентрализации – все участники равноправны в сети и изменить историю блокчейна единолично невозможно.

Важными участниками сети являются майнеры – узлы, выполняющие деятельность по созданию новых блоков. За успешное создание блока, майнер получает вознаграждение в виде криптовалюты. Майнинг призван быть децентрализованным, предотвращая возможность монополии в руках одного или нескольких участников, в обратном случае история блокчейна могла бы быть изменена.

1.1.4 Криптография

Архитектура блокчейна предполагает, что безопасность взаимодействия между участниками сети основывается на принципах математики. Основой безопасности является понятие криптографии – это метод защиты информации путем использования алгоритмов, хешей и подписей. Используя разнообразные средства и техники криптографии гарантируется неизменность истории цепочки блоков.

Существует множество криптографических алгоритмов, но основополагающее значение имеют два: RSA и SHA-256. Рассмотрим их подробнее.

RSA – криптографический алгоритм с открытым ключом, названный в честь создателей Rivest, Shamir и Adleman [13]. Основой алгоритма являются односторонние функции, обладающие следующими свойствами:

- если известно x, то вычисление $f(x)$ относительно просто;
- если известно $y = f(x)$, то вычисление x не эффективно;

Такие функции не имеют способа эффективного вычисления исходных значений, хоть математически они существуют.

RSA основан на вычислительной сложности задачи произведения двух достаточно больших простых чисел. Шифрование происходит путем возведения числа в степень. Обратная же операция требует вычисления функции Эйлера от заданного большого числа, что возможно лишь при знании его простых множителей, без знания которых, вычисление за разумное время невозможно.

В любой криптографической системе с открытым ключом, пользователь имеет пару ключей: закрытый – ключ который необходимо держать в секрете и не делится им и открытый – ключ который можно передавать куда угодно и не беспокоится о безопасности. Пользователь самостоятельно генерирует

свою пару ключей, в случае RSA это пара целых чисел. Это пара называется “согласованной”, это значит, что: ∀ допустимых пар открытого и закрытого ключей *(p,s)* ∃ соответствующие функции шифрования $E_p(x)$ и расшифрования $D_s(x)$ такие, что ∀ сообщение $m \in M$, где M – множество допустимых сообщений, $m = D_s\left(E_p(m)\right) = E_p(D_s)$.

Опишем алгоритм шифрования сообщения m от первого пользователя ко второму пользователю [13]:

1. Взять открытый ключ (*e,n*) пользователя 1.

2. Создать случайный сеансовый ключ m.

3. Зашифровать сеансовый ключ используя открытый ключ пользователя 1 используя формулу 1.1.

$$c = E(m) = m^e \ mod \ n \tag{1.1}$$

4. Зашифровать сообщение M_A используя сеансовый ключ по формуле 1.2.

$$C = E_m(M_A) \tag{1.2}$$

Тогда алгоритм расшифровывания:

1. Принять сеансовый ключ c.

2. Взять закрытый ключ пользователя 2 (*d,n*).

3. Применить закрытый ключ для расшифровывания сеансового ключа используя формулу 1.3.

$$m = D(c) = c^d \ mod \ n \tag{1.3}$$

4. Расшифровать сообщение C с помощью сеансового ключа по формуле 1.4.

$$M_A = D_m(C) \tag{1.4}$$

На рисунке 1.4 изображена схема, процесса, описанного выше.

Рисунок 1.4 – Схема алгоритма RSA с использованием сеансового ключа

SHA-256 – однонаправленная хеш-функция предназначенная для создания хеша сообщения произвольной длины [12]. Функция обладает следующими свойствами: collision-free, то есть никакие два разных входных набора данных не дадут один хеш, avalanche effect – незначительное изменение дочернего компонента полностью меняет результат и puzzle friendly, то есть не существует способа, за исключением перебора, изменить входные данные так, чтобы получить желаемый результат. Такое преобразование называется хешированием. Также выделяют криптографические хеш-функции, которые отвечают более строгим требованиям.

Работа алгоритма начинается с разбиения входящего сообщения на блоки размером 16 слов. Далее каждый блок пропускается через цикл с 64 либо 80 итерациями (одна итерация алгоритма изображена на рисунке 1.5), где 2 слова преобразуются по функции преобразования. Функция преобразования задаётся другими словами. После обработки каждого блока его результат складывается с предыдущим, и итоговая сумма является выходным значение хеш-функции. Отметим, что из-за связанности блоков их независимая обработка невозможна.

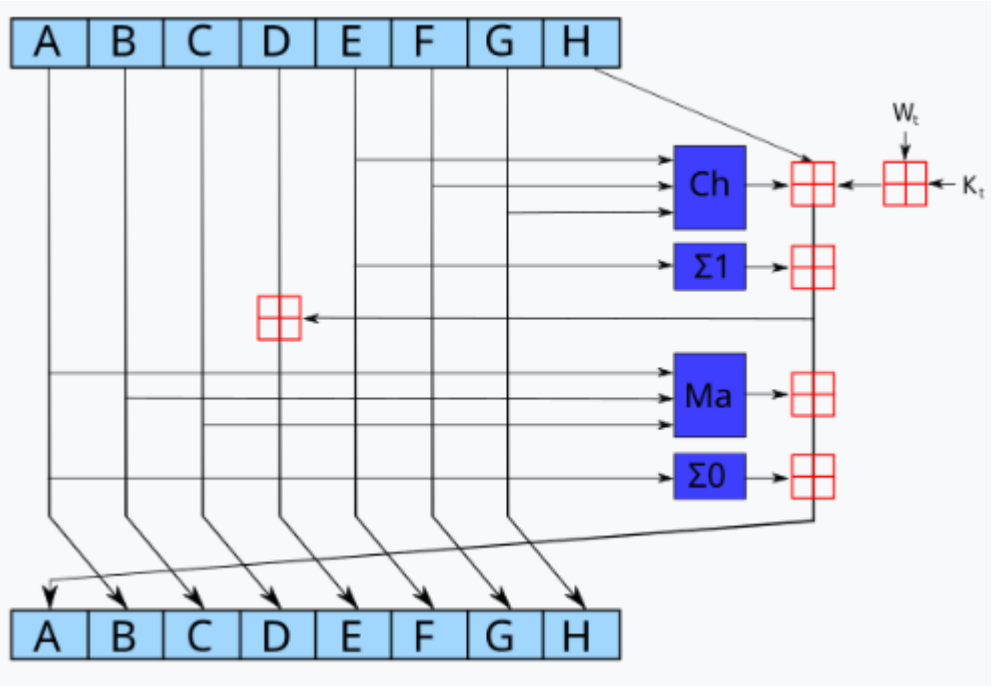

Рисунок 1.5 – Одна итерация алгоритма SHA-256

В блокчейне хеш-функции используются для вычисления хеша блока, а именно хеша заголовка блока, для этого необходим хеш корневого дерева транзакций. Это дерево представляет из себя полное двоичное дерево или дерево меркла, где листья – данные о транзакциях, а внутренние узлы содержат хеши от комбинации дочерних узлов. После того, как хеш вычислен, из-за свойства avalanche effect, какие-либо изменения в цепочке, даже самые малые, изменяют итоговый хеш.

1.1.5 Консенсус

Консенсус – это процедура, в которой одноранговые узлы сети достигают соглашения о текущем состоянии данных в сети. Благодаря этому алгоритмы консенсуса обеспечивают надежность и доверие к сети. Механизмы консенсуса составляют основу всех криптовалютных блокчейнов и делают их безопасными. Чтобы гарантировать, что все участники сети согласны с единой версией истории, сети, такие как Bitcoin и Ethereum, реализуют так называемые механизмы консенсуса (также известные как протоколы консенсуса или алгоритмы консенсуса). Эти механизмы направлены на то, чтобы сделать систему отказоустойчивой. Существует много видов механизмов консенсуса, но все они применяются для одного – обеспечить достоверность записей.

Выделим два основных механизма достижения консенсуса – Proof of Work и Proof of Stake.

Proof of Work (доказательство выполненной работы) – механизм консенсуса, требующий от участников цепи выполнить достаточно сложную и длительную задачу. PoW задачи всегда решаемы компьютером в конечный сроки, но требуют достаточно больших вычислительных мощностей. Более подробно рассмотрим механизм Proof of Work на примере блокчейн-сети Bitcoin. Прежде чем новый блок может быть добавлен в сеть, ему необходимо удовлетворять определенному условию, касающемуся хеша блока. Это условие требует наличия определенного количества нулей в начале строки хеша. Для достижения этого требования необходимо изменить хеш блока, что осуществляется с использованием nonce – строки, применяемой для модификации хеша блока. Из-за свойства puzzle friendly функции хеширования, решение, то есть нахождение подходящего nonce, осуществимо только методом перебора. Как только условие удовлетворено, блок добавляется в блокчейн и рассылается всем участникам. PoW имеет значительный недостаток, из-за роста вычислительной мощности компьютеров, сложность решения задачи тоже должна увеличиваться, что постоянно повышает затраты на электроэнергию и снижает доступность майнинга.

Proof of Stake (доказательство доли владения) – механизм консенсуса, который предложен как замена Proof of Work и решение проблемы с электроэнергий. Предложение заключается в том, что все участники децентрализованной сети должны иметь право выпускать блоки, пропорционально количеству монет, которыми они владеют от общего количества. Процесс производства блоков получил название Форжинг. А процесс хранения монет для производства блоков стейкинг. Этот механизм решил проблемы PoW, но имеет и собственные недостатки. Главной из которых является производительность сети. Многие сети, использующие PoW, предоставляют пропускную способность значительно выше.

1.1.6 Преимущества и недостатки технологии блокчейн

Выделим преимущества технологии блокчейн:

- повышенная прозрачность данных;
- защита от подделки;
- снижение количества ошибок;
- беспрерывная доступность;

- автоматизация процессов через умные контракты;

Но блокчейн имеет и ряд недостатков:

- сложность внедрения в существующие сферы;
- необходимость переквалификации сотрудников;
- затраты на электроэнергию;
- сложность отладки ошибок;

Технология является перспективной, но требует изучения целесообразности и применимость в логистики.

1.2 Роль логистики в современном мире

Логистика – направление хозяйственной деятельности по управлению материальными и информационными потоками [1]. Она связывает производителей, поставщиков и конечных потребителей. Благодаря логистике развиваются компании и страны. Рынок логистики активно увеличивается последние 10 лет, на рисунке 1.6 изображены объемы рынка с 2018 по 2022 год, а также спрогнозирован рост до 21.13 триллионов долларов в 2030 году. Прогнозируемый годовой процент роста – 4%.

Рисунок 1.6 – График объема рынка логистики и грузоперевозок

Выделим факторы роста рынка логистики [10]. Первым важным фактором является глобализация или мировая торговля. Рост логистического рынка подтверждают данные Организации сотрудничества железных дорог (ОСЖД). В 2011 году количество поездов из Китая в ЕС составило 117, спустя

6 лет уже 3673, а в 2023 году 17500 поездов, что на 18% выше предыдущего 2022 года. Таким образом можно прогнозировать и дальнейшее развитие этого направления. Третий фактор развития – расширение рынка благодаря продажам и покупкам товаров и услуг в интернете. В Республике Беларусь в 2024 году онлайн-коммерция заняла 8.1% от общего товарооборота.

На фоне роста рынка логистики, инвестиции в ее оптимизацию выглядят очень грамотным решением, что подтверждается активностью крупных компаний, стремящихся к повышению эффективности своих цепочек поставок. Исследование, разработка и внедрение решение по оптимизации логистики имеют следующие преимущества:

1. Снижение рисков. Правильно работающие цепочки поставок позволяют проще проходить сложные этапы, такие как экономические кризисы и пандемии.
2. Улучшение обслуживания клиентов. Грамотно настроенная логистика позволяет повысить точность и скорость доставки, что приводит к росту числа довольных и лояльных клиентов.
3. Снижение операционных затрат. Согласно данным McKinsey, компании вкладывающие деньги в оптимизацию логистических процессов сокращают операционные затраты на 15 - 20%.
4. Уменьшение вреда окружающей среде. Во всем мире 24% выбросов в атмосферу CO2 происходит из-за сжигания топлива в транспорте, 12% из которых относятся к сфере логистики, оптимизация по нахождению кратчайших путей или уменьшению времени простоя транспорта позволяет значительно уменьшить влияние на окружающую среду.

Таким образом, учитывая размеры рынка логистики и тенденции к его росту, оптимизация логистических процессов является не только актуальной задачей, но и стратегически важным направлением развития компаний и стран. Использование новых подходов, исследование новых технологий и инвестиции в этом направлении позволят развивать логистику, ускоряя доставки по всему миру, повышая качество обслуживания и снижая риски ошибок и издержек.

1.3 Роль блокчейна в оптимизации логистических процессов

Выделим основные логистические процессы:

- организация цепочек поставок;
- физическое перемещение товаров;
- складская логистика;

- контроль запасов;
- документооборот;
- оптимизация маршрутов доставки;

Каждый процесс имеет свои трудности и проблемы, блокчейн может эффективно решить некоторые из них. Внедрение технологии имеет потенциал изменить мировую экономику благодаря децентрализованному хранению данных и неизменности данных. Благодаря умным контрактам появляется возможность автоматизировать процессы, из-за чего компании должны заранее адаптироваться к данным нововведениям.

Результаты исследования показывают стремительное распространение данной технологии. Происходит активный рост средних и крупных компаний (более чем 20000 работников), внедряющих технологию блокчейн. Рисунок 1.7 отображает график восприятия данной технологии среди руководств компании:

- 82% руководителей считают технологию применимой в логистике;
- 39% опрошенных заявили, что хорошо понимают технологию блокчейн;
- 39 % готовы к развертыванию технологии;
- 76% считают технологию полезной;

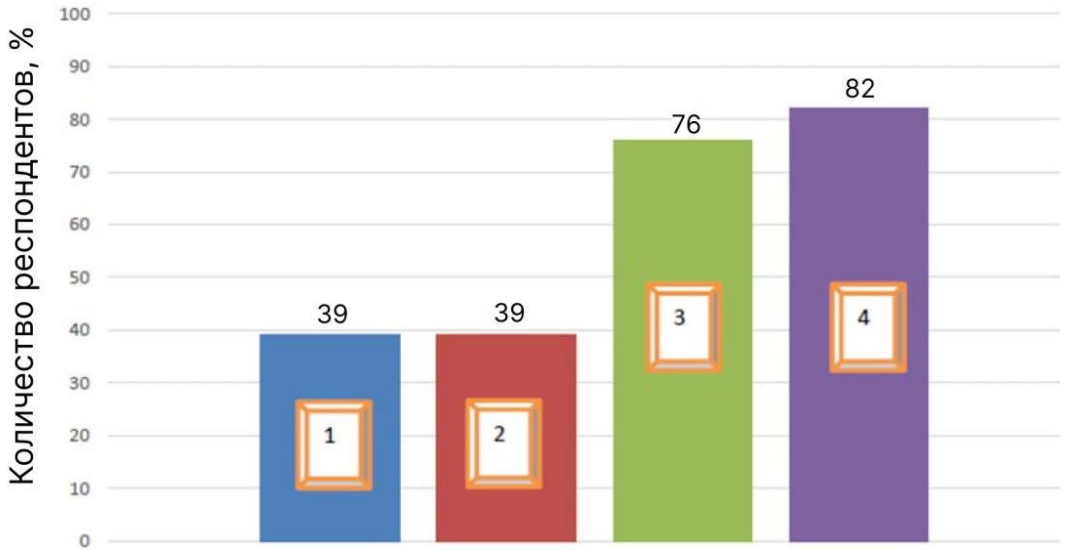

Рисунок 1.7 – Восприятие респондентами технологии Блокчейн (1 – готовы развертывать блокчейн; 2 – топ-менеджмент хорошо понимает возможности блокчейна; 3 – блокчейн полезен/очень полезен; 4 – понимают, что блокчейн дает несколько существенных преимуществ)

Опрос проводимый среди руководителей крупных компаний из списка Forbes 500 подтверждает полезность и применимость блокчейна в разных

сферах, включая логистику. На рисунке 1.8 изображены отмеченные варианты использования технологии. Большая часть рецензентов (около 30%) отмечают, что самой подходящей сферой применения технологии блокчейн являются платежи и финансы. Однако применение в интернете вещей (IoT) и в управлении цепочками поставок занимают значительную долю ответов. Это подчеркивает универсальность и применимость технологии.

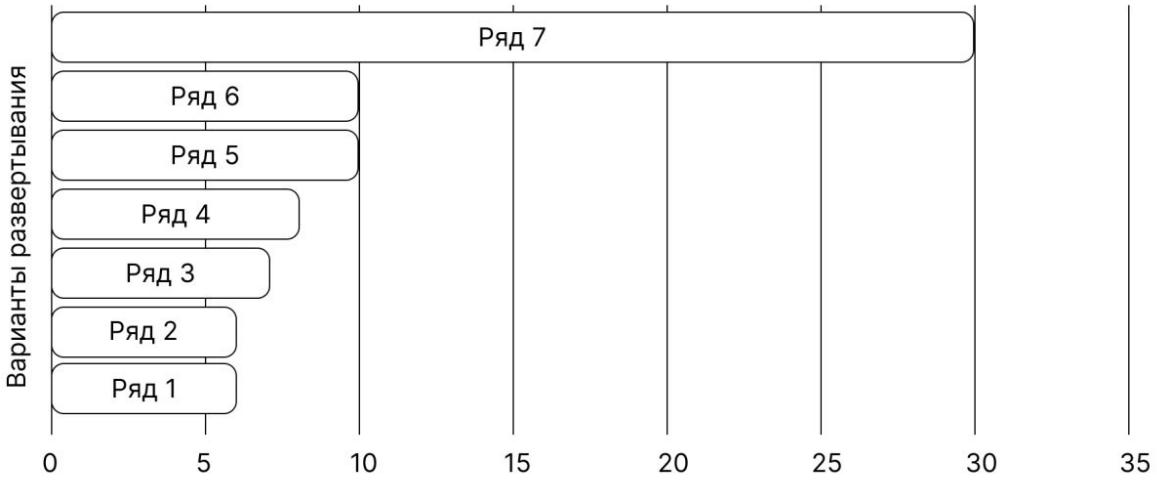

Рисунок 1.8 – Варианты, доминирующие в раннем развертывании технологии блокчейн (1 – здравоохранение; 2 – управление данными; 3 – идентификация; 4 – Интернет вещей; 5 – управление цепями поставок и отслеживание; 6 – Смарт-контракты; 7 – расчеты и платежи)

Сильными сторонами блокчейна при применении в сфере логистики можно считать: защищенность данных, прозрачность, внедрение смарт-контрактов для автоматизации, безопасное хранение документации и предотвращение мошенничества. Таким образом способов применения блокчейна в логистике множество, далее рассмотрим их подробнее.

Главной проблемой в управлении цепочками поставок является отсутствие прозрачности, низкая безопасность и необходимость доверия. Это связано с отсутствием системы с полной прозрачностью информации. Создание данной системы требует огромное количество ресурсов и времени. Технология блокчейн позволяет решить эту проблему, предлагая единую систему для обмена информацией, с равноправными пользователями, что обеспечивается за счет децентрализации.

Так как в современном мире информация не перемещается вместе с грузом появляется ряд проблем [10]. Основная – это обеспечить прозрачность обеим заинтересованным сторонам, что не позволяет планировать операции заранее. Технология блокчейн хорошо подходит для решения данной

проблемы. Ее использование упрощает отслеживание доставки. Схема такой интеграции блокчейна представлена на рисунке 1.9.

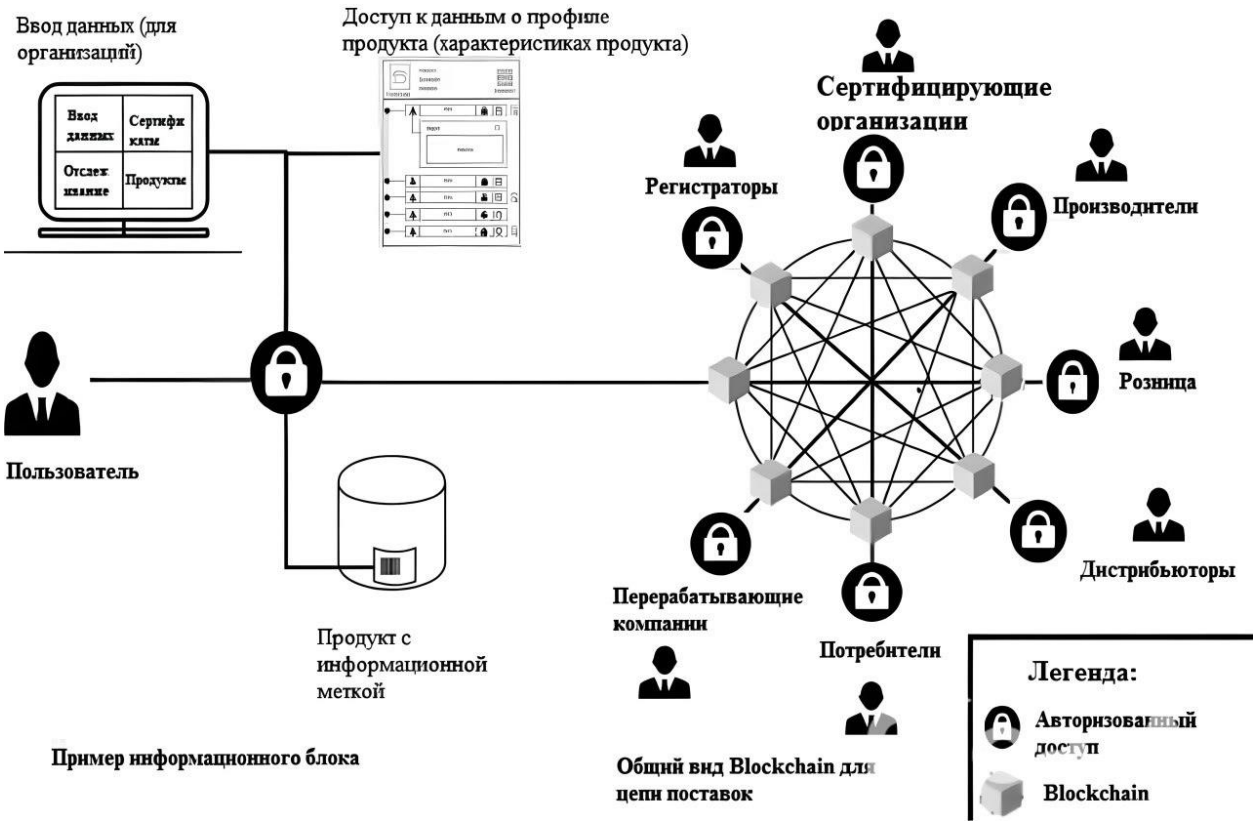

Рисунок 1.9 – Применение блокчейна в цепи поставок

Одной из причин использовать блокчейн так же является снижение контрафактной продукции [5]. Это достигается за счет прозрачности жизненного цикла продукта, который находится в общем доступе. Каждый этап – от производства до доставки конечному потребителю – фиксируется в распределенной базе данных. Это позволяет всем участникам цепочки поставок, включая производителей, дистрибьюторов, логистические компании и конечных потребителей, отслеживать происхождение и подлинность продукции в режиме реального времени. Например, каждый товар может быть снабжен уникальным идентификатором, который записывается в блокчейн, исключая возможность подделки данных.

Логистические процессы требуют хранения множества документации: транспортные накладные (crm), сертификаты происхождения, складская документация, карты маршрутов и так далее. Блокчейн может выступать в роли их надежного хранилища. Данные хранящиеся в блокчейне защищены от подделки, что особенно важно в международной логистике из-за повышенных требований к прозрачности процесса. Использование данной технологии позволяет делится электронными документами мгновенно, что ускорит

контроль на таможне или во время прохождения сертификации, существенно снижая затраты.

Важной частью блокчейна являются смарт-контракты, позволяющие избегать посредников. Благодаря интеграции контрактов повышается безопасность исполнения обязательств и перевода средств. Из-за их автоматического исполнения укрепляется надежность договоров, сокращаются случаи мошенничества.

1.4 Анализ примеров внедрения блокчейна в логистике

По мере того как технология блокчейн набирает популярность, многие традиционные сферы проявляют интерес к её использованию, и логистика не является исключением. Многие компании-лидеры, проводят исследования и внедряют блокчейн в свои процессы. Актуальность данной технологии особенно велика для Республики Беларусь, где одним из приоритетных направлений социально-экономического развития на 2021–2025 года является повышение эффективности логистических процессов [8]. Далее будут рассмотрены разработки, предлагаемые компаниями для повышения эффективности логистических процессов.

1.5 Отслеживание цепочки поставки манго Walmart

Компания Walmart Technology – крупнейший ритейлер в мире, его опыт применения блокчейна в логистике считается одним из самых успешных. Главной задачей компании стало повышение прозрачности и эффективности управления цепочками поставок, особенно в сфере отслеживания продуктов питания. Этот вопрос актуален при возникновении пищевых заболеваний, из-за чего встает вопрос поиска источника, на что может потребоваться от нескольких дней до недель.

Walmart посчитал, что блокчейн хорошо подойдет для решения этой проблемы. Компания рассматривала несколько сетей, включая Ethereum, Burrow и Hyperledger Fabric, но в конечном итоге остановились на последнем. Карл Бедвелл, старший директор, заявил, что Hyperledger Fabric отвечает большинству потребностей в технологии блокчейн. Данная система должна использоваться всеми участниками процессов, поэтому важно было выбрать технологию с открытым исходным кодом.

Hyperledger Fabric – это платформа с открытым исходным кодом для создания частных и корпоративных блокчейн-сетей, разработанная в рамках проекта Hyperledger, который курируется Linux Foundation. Разработка проводилась совместно с IBM и началась в 2016 году. Основные участники проекта Walmart и IBM анонсировали проект. Целью проекта было отслеживание происхождения манго, продаваемых в магазинах Walmart на территории США.

Первым этапом стало тестирование, бывший вице-президент компании, Франк Яннас, купил манго в случайном магазине Walmart. Отслеживание цепочки поставки данного манго заняло целых 7 дней, что хоть и считается приемлемым результатом, компания была нацелена на его улучшение. Таким образом они приступили к созданию системы отслеживания продуктов питания на основе блокчейна.

Используя Hyperledger Fabric, было создана частная сеть, куда допускались только одобренные участники. В систему были добавлены все участники цепочки поставок: фермеры, перевозчики и Walmart. Каждый этап транспортировки груза фиксировался в блокчейне с добавлением подробной информации с разнообразных датчиков (температура, влажность, время и так далее). Основой автоматизации выступили смарт-контракты проверяющие условия на каждом этапе. При нарушении условий отправлялось соответствующее оповещение. Система так же взаимодействовала с IoT, каждая партия имела QR-код и необходимые датчики.

Все данные хранились в зашифрованном виде для защиты информации, а архитектура блокчейна обеспечивала устойчивость к сбоям. Участники сети имели доступ только к необходимым данным, остальные же были изолированы от них.

Итогом внедрения этой системы стало снижение времени остлеживания цепочки поставок с 7 дней до 2.2 секунд. Это радикально изменило подход к управлению поставками, позволив Walmart быстро выявлять источники контрафактной продукции, минимизировать потери из-за задержек и гарантировать качество продуктов.

Для дальнейшего развития системы компания предложила другим крупным игрокам рынка присоединиться. Новыми участниками выступили Nestle и Universe, а система получила название "IBM Food Trust". Эксперимент закончился успехом и отметил возможность решения логистических задач с использованием блокчейна.

1.6 Maersk TradeLens

Maersk, крупнейшая в мире судоходная компания, активно исследует и внедряет блокчейн для оптимизации своих логистических процессов. Одним из ключевых проектов компании в этом направлении стала платформа TradeLens, разработанная совместно с IBM.

TradeLens – это блокчейн-система, предназначенная для повышения прозрачности, эффективности и безопасности в сфере морских перевозок. Она объединяет все задействованные в цепочке поставок стороны: грузовладельцев, экспедиторов, внутренних поставщиков транспорта (включая железнодорожный и автомобильный), морских перевозчиков, порты и терминалы, таможенные и другие государственные органы. Разработанная система позволяет быстро и прозрачно обмениваться информацией о грузе и транспорте, анализировать показания с датчиков и работать с документами.

Опишем работу TradeLens.

- Оборудование и груз снабжаются датчиками, регистрирующими множество показателей (температура, влажность, тряска и так далее). Информация из этих датчиков записывается в блокчейн сеть, что обеспечивает прозрачность. Это позволяет проверять состояние груза на протяжении всей доставки.
- Смарт-контракты проверяют данные с датчиков и передают информацию заинтересованным сторонам. Эти контракты заполняют документы, производят финансовые расчеты и так далее. Именно эта автоматизация ускоряет документооборот и повышает его надежность.

В этой схеме блокчейн выступает в роли базы данных, обеспечивающей надежное хранение, передачу и защиту информации.

Система разрабатывалась до 2018 года. Тестовым проектом выступила цепочка поставок из Африки в Европу. Диаграмма этой цепочки до внедрения технологии на рисунке 1.10. Как видно всего было 6 этапов: подготовка цветов, внесение информации в сеть, добавление соответствующей транзакции, погрузка на корабль, работа смарт-контрактов и получение груза.

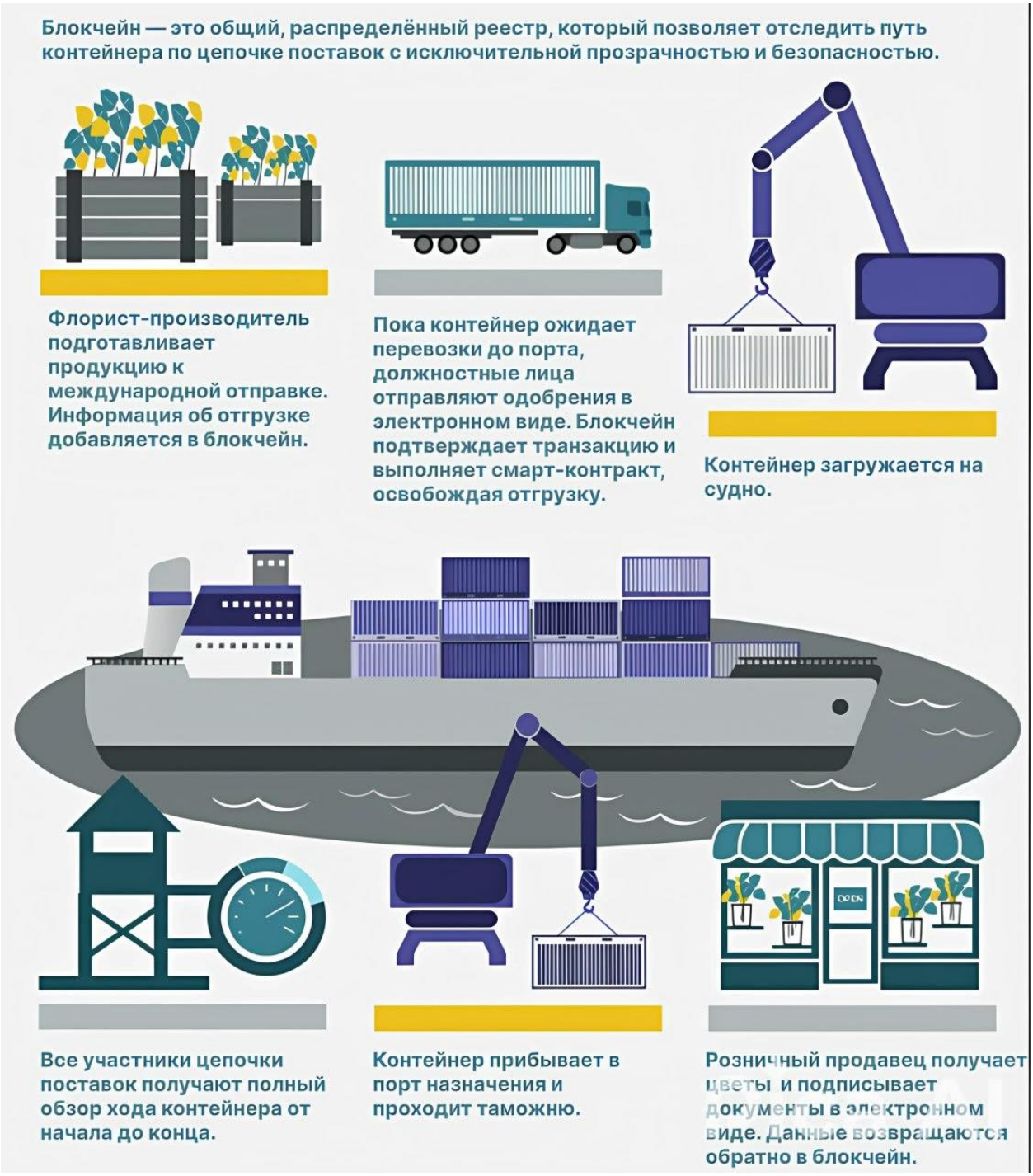

Рисунок 1.10 – Цепочка поставок цветов в пробном проекте TradeLens

Итогом тестирования стало более 200 сообщений (что создало 25 сантиметров документации) и 30 задействованных организаций (банки, порты, транспортные компании и так далее). Затем всех участников цепочки интегрировали в блокчейн и отправили новую партию цветов. Все действия были зафиксированы в смарт-контракте, который сопровождал груз по всей цепочке поставок, автоматически заполняя документы, проставляя штампы и делая финансовые расчеты.

В поставке такого рода можно выделить 13 звеньев:

1. Внутренние перевозки.
2. Таможенный брокер.
3. Таможня.
4. Порты и терминалы.
5. Морские перевозки.
6. Экспедиторы.
7. Система управления портовыми операциями.
8. Страховые услуги.
9. Государственные операции.
10. Системы управления транспортом.
11. Торговые ассоциации.
12. Система отслеживания цепочек поставки.
13. Грузоотправители.

На рисунке 1.11 слева изображено взаимодействие этих звеньев в системе, не использующей блокчейн. Видно, что они переплетены из-за чего повышается сложность координации и снижается прозрачность цепочки поставок. Справа изображена система после внедрения технологии блокчейн, благодаря ей удалось избежать хаотичного переплетения, а информация постепенно переходит между всеми звеньями. Это делает процессы эффективнее и сокращает издержки.

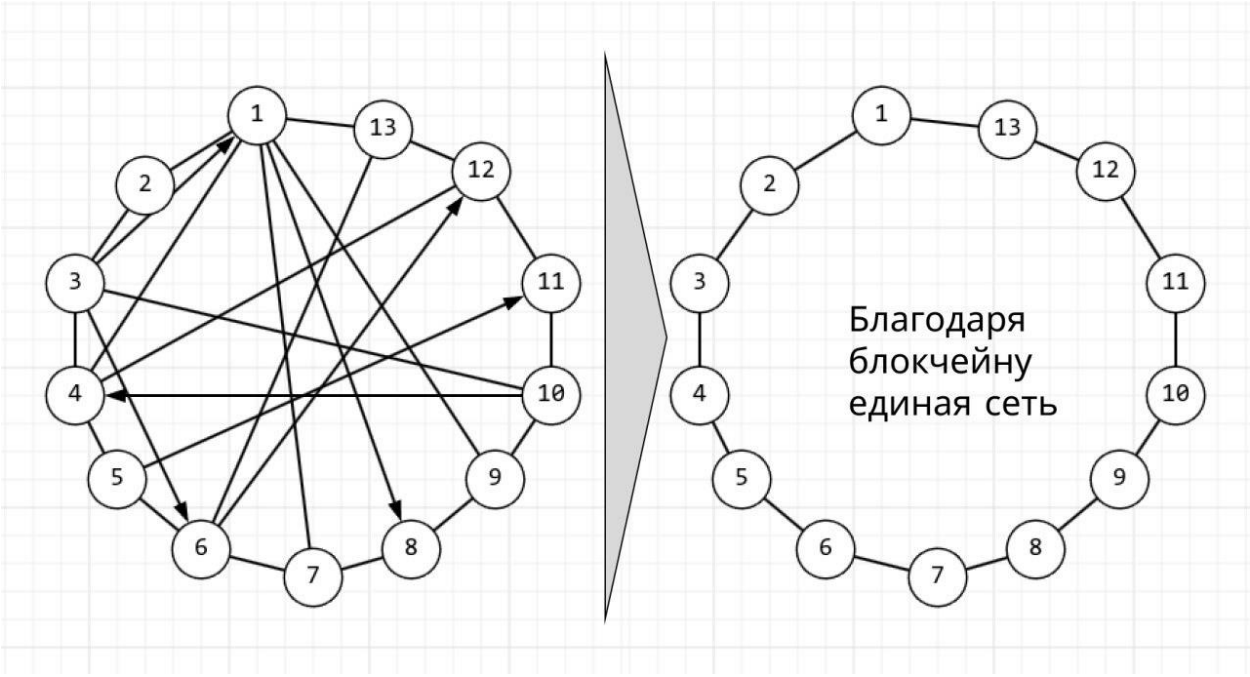

Рисунок 1.11 – Сравнение традиционной модели взаимодействия(слева) с интеграцией блокчейна(справа)

Тестовый проект дал подтверждение концепции, но выявил ряд проблем. Основная проблема – это отсутствие компетентности у персонала

компаний и госструктур, а также их сложная коммуникация. Решая проблемы тестирование продолжилось.

По итогам использования TradeLens в течении года показало:

- Повышение прозрачность цепочки поставок.
- Снижение документооборота. Например, при отправке авокадо из Момбасы в Роттердам затраты составляют 300 долларов или 15-20% от стоимости доставки. TradeLens уменьшила эти расходы на 70-90%
- Ускорение перевозок примерно на 40% на территории США.

Система TradeLens успешно выполнила поставленные задачи, продемонстрировав свою эффективность и став ярким примером применимости блокчейна для оптимизации логистических процессов, что способствует дальнейшему развитию этой технологии.

1.7 Выводы

В данной главе проведен анализ логистических процессов и изучены возможности применения технологии блокчейн в этой сфере. Рассмотрены основные логистические процессы, включая управление цепочками поставок, транспортировку, складскую логистику, управление запасами и документооборот. Также выявлены основные проблемы, с которыми сталкивается отрасль: отсутствие прозрачности, сложность координации между участниками цепочки поставок и высокий риск потерь.

Блокчейн обладает рядом преимуществ, которые позволяют решать многие из этих задач. В ходе изучения механизма работы блокчейна было выделено его ключевое преимущество – обеспечение децентрализованного и защищенного хранения данных, что повышает прозрачность всех этапов логистического процесса и не требует доверия между его участниками.

Проанализированы преимущества внедрения технологии в логистику: автоматизация процессов используя умные контракты, отслеживание грузов, борьба с контрафактной продукцией. Однако помимо преимуществ выделены и недостатки: высокая стоимость разработки подобных систем и лишние энергозатраты.

Изучены существующие подходы к решению логистических задач с применением технологии блокчейн. В рамках этой части были проанализированы два значимых примера: оптимизация отслеживания цепочки поставок манго компанией Walmart и внедрение блокчейна в рамках системы TradeLens от Maersk. По итогу рассмотрения данных примеров сделан вывод о возможности и обоснованности применения технологии в

сфере логистики. Данные разработки явно показали преимущества и недостатки использования блокчейна, выявили наиболее подходящие процессы для оптимизации, а также пояснили этапы разработки, тестирования и поддержки подобных систем.

ГЛАВА 2
ПРАКТИЧЕСКАЯ ЧАСТЬ

2.1 Приложение прямого взаимодействия заказчика и перевозчика

В практической части дипломной работы было реализовано децентрализованное приложение для управления поставками, позволяющее создавать, принимать, завершать доставки, а также сохранять и скачивать логистические документы.

Как было отмечено в первой главе, документооборот занимает значительную часть расходов всего логистического процесса. Целью данного приложения является обеспечение прямого взаимодействия заказчиков и перевозчиков.

Перед реализацией приложения были сформулированы следующие требования:

1. Хранение документов с использование распределенного реестра.
2. Автоматизация с использованием смарт-контрактов.
3. Кроссплатформенность приложения.
4. Наличие удобного пользовательского интерфейса.
5. Поддержка нескольких типов загружаемых документов (pdf и docx).

В качестве основы приложения были выбраны следующие инструменты. Фреймворк React js позволяет разрабатывать пользовательский интерфейс и интегрировать их с логикой приложения. В качестве языка бэкенда был выбран Golang за удобную и легкую стандартную библиотеку для работы с http. Для реализации смарт-контрактов на платформе Ethereum используется язык Solidity [3]. В качестве инструмента для работы с документами выбрано IPFS хранилище – решение с открытым исходным кодом, работающее по принципам децентрализации [2].

Перед разработкой приложения было сформулировано описание его работы: пользователь попадает на главную страницу с навигационной панелью. На данной панели он видит 4 опции: создание перевозки, список созданных перевозок, история всех перевозок и его перевозки. Для доступа к функционалу приложения пользователь должен авторизоваться на сайте используя криптовалютный адрес, для этого в правом углу навигационной панели есть кнопка “Подключить кошелек”. После подключения вместо кнопки отображается его адрес. В приложении предусмотрено 2 вида пользователей – заказчик и перевозчик. Основной функционал приложения

включает создание перевозки, принятие заявки перевозчиком, отображение перевозок для авторизованного пользователя и отображение всех созданных перевозок.

Для создания перевозки необходимо авторизоваться, перейти на страницу “Создать перевозку”, прикрепить необходимые документы, указать срок доставки. По нажатию кнопки “Загрузить файл в IPFS и создать перевозку” необходимо будет подписать транзакцию через ранее авторизованный кошелек, после чего созданная перевозка закрепляется в блокчейне, загруженные файлы попадают в IPFS хранилище.

На странице “Список созданных перевозок” отображаются все созданные и не принятые перевозки. Каждая перевозка содержит id, cid, дату доставки, адрес заказчика, кнопку “скачать” для загрузки прикрепленных документов, и кнопку “принять” для принятия данной перевозки. При нажатии кнопки “принять” пользователь должен подписать транзакцию в авторизованном кошельке. Данная транзакция закрепляет адрес пользователя в качестве перевозчика и переводит перевозку в статус “принято”.

Страница “История перевозок” содержит все созданные перевозки, отображая их id, cid, статус, адрес заказчика и перевозчика.

Страница “Мои перевозки” отображает перевозки для авторизованного пользователя, если пользователь является перевозчиком, то он может подтвердить доставку нажав на кнопку “завершить”. Нажатие на эту кнопку переводит перевозку в статус “завершено”.

2.2 Реализация приложения

2.2.1 Архитектура приложения

Одной из ключевых задач при разработке было обеспечить кроссплатформенность и доступность приложения максимальному числу пользователей, для этого было принято решение разрабатывать веб-приложение. В отличии от приложений, работающих вне браузера, оно не требует установки и совместимо со всеми современными браузерами.

К разработке предстояло выполнить 3 компонента приложения:

- клиентская часть;
- серверная часть;
- смарт-контракты;

На рисунке 2.1 изображены связи между этими компонентами.

Рисунок 2.1 – Архитектура приложения

2.2.2 Клиентская часть

Разработка началась с клиентской части с использованием фреймворка React js. Первым шагом стала реализация панели навигации, позволяющая пользователю перемещаться между 4 страницами, описанными ранее и авторизоваться через кошелек. На рисунке 2.2 изображена эта панель, ее реализация представлена в приложении А.

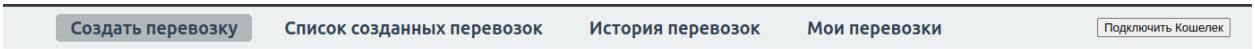

Рисунок 2.2 – Панель навигации

Страница создания перевозки.

Страница для создания перевозки представлена на рисунке 2.3. Она содержит следующие элементы:

- Форма для прикрепления логистических документов.
- Поле ввода срока доставки.

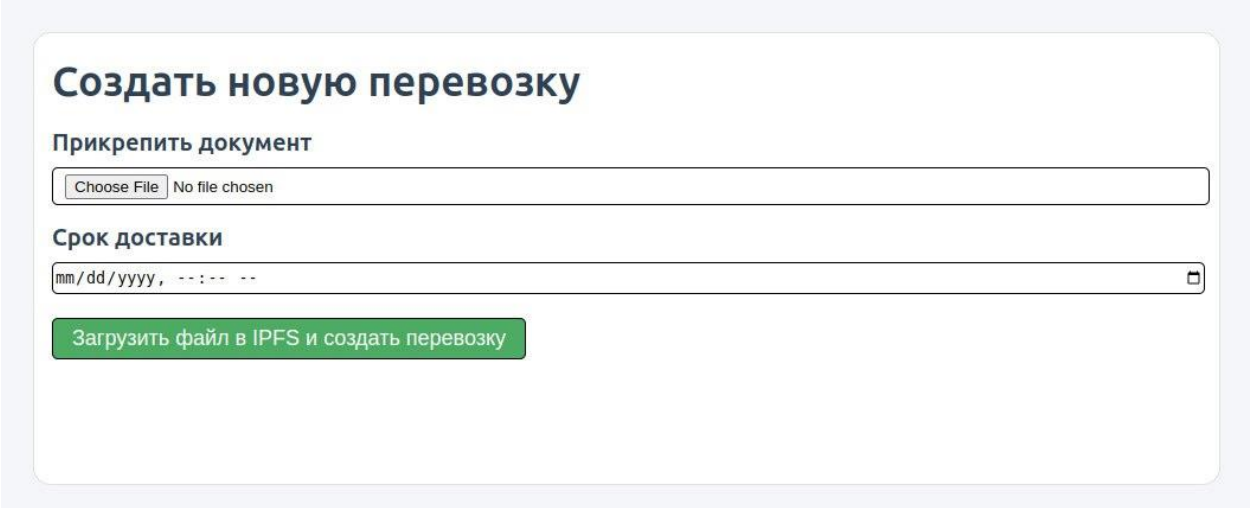

Рисунок 2.3 – Страница создания новой перевозки

После корректного заполнения всех полей, прикрепления файла и нажатия на кнопку “Загрузить файл в IPFS и создать перевозку” на экране всплывает окно с результатом этого действия, где-либо отображена ошибка, возникшая при создании новой перевозки, либо информация об успешно созданной.

Страница со списком созданных перевозок.

Данная страница, изображённая на рисунке 2.4, отображает список созданных непринятых перевозок. Каждая строка с перевозкой содержит:

- ID.
- CID.
- Дату доставки.
- Адрес заказчика.
- Кнопку “принять”.
- Кнопку “скачать”.

Принятая и выполненная перевозка переходит на страницу с историей перевозок.

Страница с историей перевозок.

Эта страница изображена на рисунке 2.5, она содержит всю историю выполненных перевозок, отображая про нее следующую информацию:

- ID.

- CID.
- Статус (доставлено в срок, просрочено, утеряно и так далее).
- Адрес заказчика.
- Адрес перевозчика.

Цель данной страницы повысить прозрачность перевозок и позволить обеим сторонам (заказчикам и перевозчикам) иметь полное представление друг о друге.

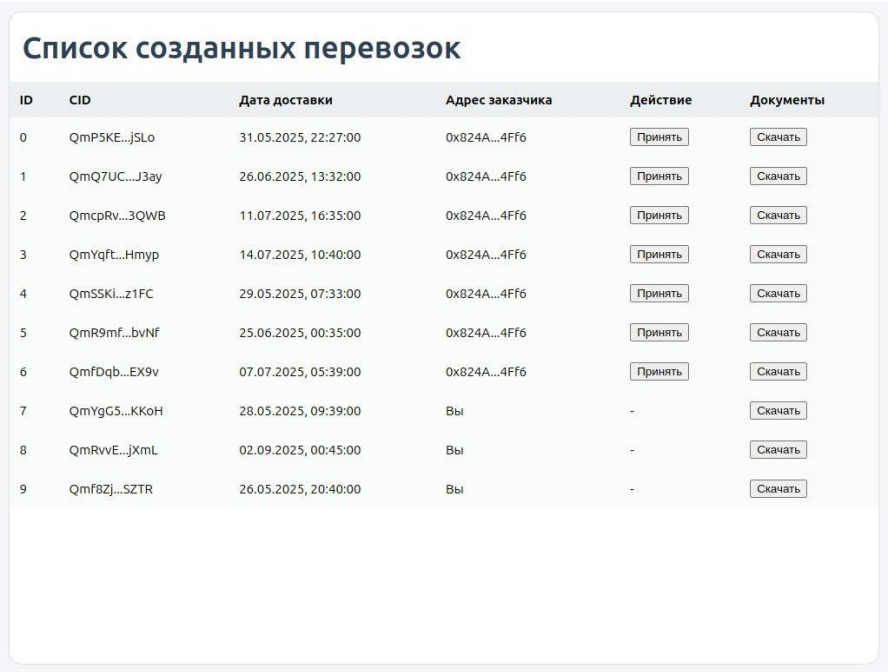

Рисунок 2.4 – Страница активных перевозок

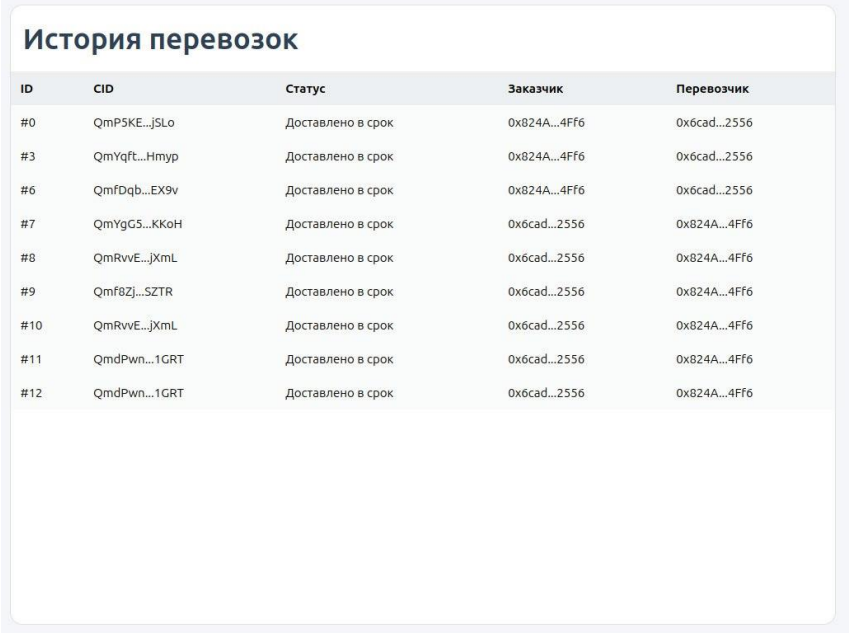

Рисунок 2.5 – Страница с историей перевозок

Страница “мои перевозки”.

Данная страница изображена на рисунке 2.6. Она доступна только после авторизации. Страница отображает все перевозки для авторизованного пользователя и позволяет завершить активные перевозки. Каждая перевозка имеет следующую информацию:

- ID.
- CID.
- Адрес заказчика.
- Адрес перевозчика.
- Действие (если доступно).
- Кнопка для загрузки документов.

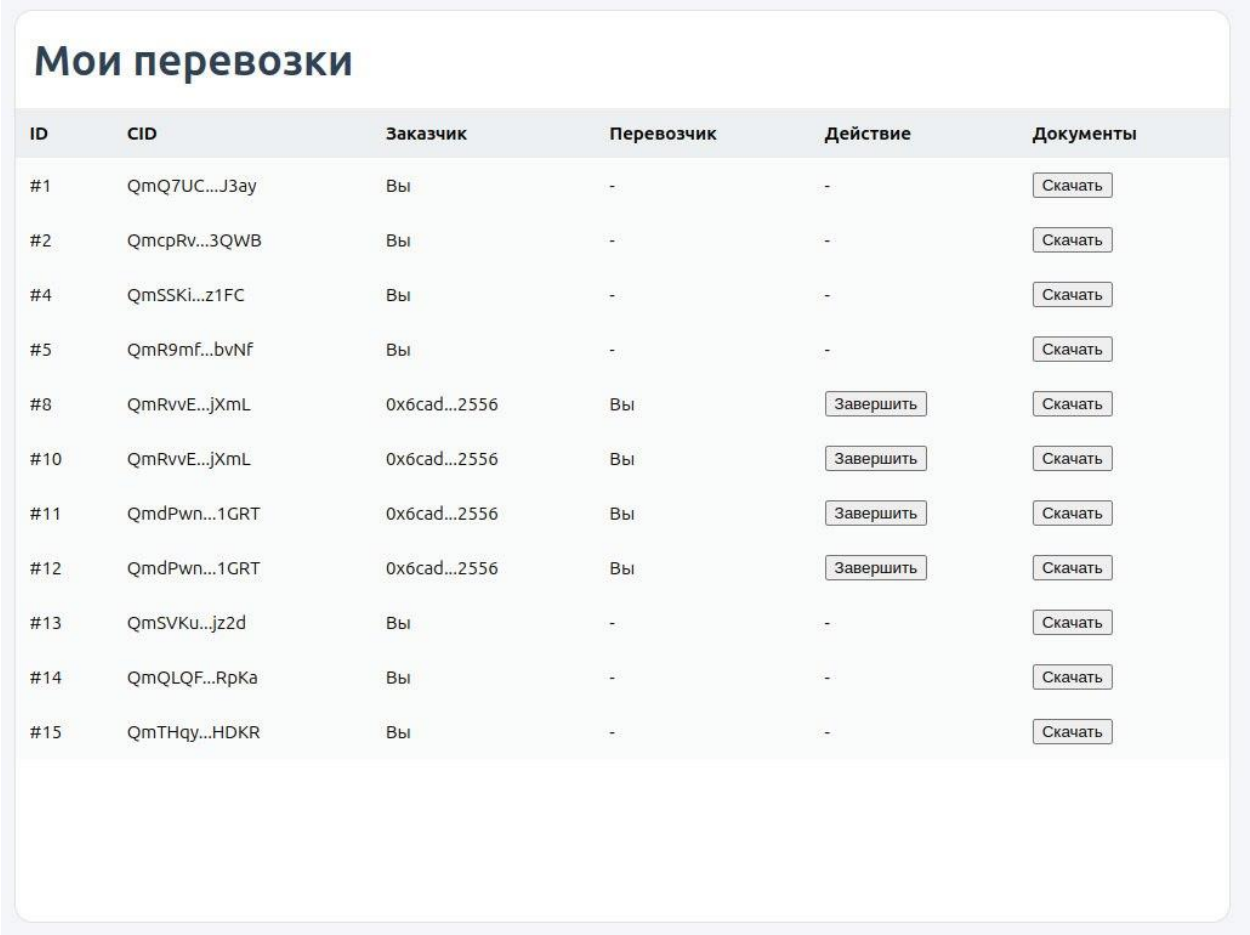

Рисунок 2.6 – Страница перевозок пользователя

2.2.3 Серверная часть

Для обеспечения взаимодействия клиентской части и блокчейна была разработана серверная часть приложения, выполняющая роль

промежуточного звена между веб-интерфейсом и технологией распределенного реестра.

Файловая структура серверной части (рисунок 2.7) состоит из папки http и service. В http реализовано 6 обработчиков:

- /api/acceptShipment – post обработчик, отвечающий за принятие созданной перевозки. Сервер принимает id перевозки и адрес перевозчика после чего подготавливает транзакцию с фиксацией состояния “accepted” для перевозки. Реализация данного обработчика представлена в приложении Б.
- /api/createShipment – post обработчик, позволяющий создать новую перевозку в блокчейне. Сервер получает загруженные файлы, адрес перевозчика и заказчика, после чего сохраняет файл в IPFS, принимает cid (идентификатор данных) и упаковывает эти данные в транзакцию. Реализация данного обработчика представлена в приложении В.
- /api/download/{cid}– get обработчик, отвечает за скачивание файлов, добавленных заказчиком при создании перевозки. Реализация данного обработчика представлена в приложении Г.
- /api/finisShipment – post обработчик, принимает id перевозки и изменяет ее состояние на “finished”, путем отправления транзакции в сеть. Реализация данного обработчика представлена в приложении Д.
- /api/getHistoryShipment - get обработчик. Сервер запрашивает у смарт-контракта все перевозки, фильтрует завершенные и возвращает их. Реализация данного обработчика представлена в приложении Е.
- /api/getShipments – get обработчик. Сервер запрашивает у смарт-контракта все перевозки, фильтрует не принятые и их возвращает. Реализация данного обработчика представлена в приложении Ж.
- /api/getMyShipments – post обработчик. Сервер запрашивает у смарт-контракта все перевозки, фильтрует их по заданному пользователю и возвращает. Реализация данного обработчика в приложении З.

Рисунок 2.7 – Файловая структура серверной части

Для реализации требования об децентрализованном хранении загруженных файлов было выбрано IPFS хранилище. IPFS (InterPlanetary File System) – это распределенный протокол хранения и передачи данных. В отличии от других протоколов, основой протокола ipfs выступают принципы децентрализации, что обеспечивает надежность хранения, повышает доступность и снижает нагрузку на инфраструктуру. Для разработки и тестирования приложения был развернут локальный кластер ipfs. Для работы с этим кластером из Go использовалась официальная библиотека go-ipfs-api, предоставляющая удобное api для работы. Были реализованы две функции для работы с ipfs:

- UploadFileToIPFS(filepath string) (string,error) – функция используемая для загрузки файла в распределенную файловую систему ipfs. Она принимает путь, по которому сохранять файл, для этого он открывает соединение с локальным узлом ipfs и отправляет данные. В случае успешной загрузки метод возвращает cid – уникальный идентификатор, который в будущем используется для фиксации созданной перевозки в блокчейне. В случае ошибки функция возвращает пустой cid и ошибку. Реализация данного метода представлена на рисунке 2.8.

```go
func UploadFileToIPFS(filePath string) (string, error) {  1 usage
    sh := shell.NewShell(url)

    f, err := os.Open(filePath)
    if err != nil { return "", fmt.Errorf( format: "cannot open file: %w", err) }
    defer f.Close()

    cid, err := sh.Add(f)
    if err != nil { return "", fmt.Errorf( format: "caannot upload: %w", err) }

    return cid, nil
}
```

Рисунок 2.8 – Реализация сервиса UploadFileToIPFS

- DownloadFileFromIPFS(cid string) ([]byte,error) – функция позволяющая получить содержимое ранее загруженного файла по его cid. Для загрузки файла формируется http-запрос к локальному узлу ipfs, загружаются данные и выдаются в виде массива байтов. В случае ошибки, массив остается пустым, а вторым параметром возвращается ошибка. Реализация данного метода на рисунке 2.9.

```go
func DownloadFileFromIPFS(cid string) ([]byte, error) {  1 usage
    url := fmt.Sprintf( format: "http://localhost:8080/ipfs/%s", cid)

    resp, err := http.Get(url)
    if err != nil { return nil, fmt.Errorf( format: "cannot download file: %w", err) }
    defer resp.Body.Close()

    if resp.StatusCode != http.StatusOK { return nil, fmt.Errorf( format: "status not ok: %s", resp.Status) }

    return io.ReadAll(resp.Body)
}
```

Рисунок 2.9 – Реализация сервиса DownloadFromIPFS

Как отмечалось ранее, сервер является связующей частью между контрактом в блокчейн сети и клиентской частью, для этого в папке contract реализованы следующие функции:

- func CreateShipmentData(client, parsedABI, cid, customerAddress common.Address, timestamp string,) (string, uint64, error) – функция генерирующая данные для транзакции вызывающей метод createaShipment в смарт-контракте и nonce для этой транзакции. В случае возникновения ошибки функция возвращает ее, а данные и nonce оставляет пустыми. Реализация этого сервиса на рисунке 2.10.

```go
func CreateShipmentData(
    client *ethclient.Client,
    parsedABI abi.ABI,
    cid string,
    customerAddress common.Address,
    timestamp string,
) (string, uint64, error) {
    timestampBigInt := new(big.Int)

    if _, ok := timestampBigInt.SetString(timestamp, base: 10); !ok {
        fmt.Println( a...: "Cannot convert timestamp to *big.Int")
    }

    data, err := parsedABI.Pack( name: "createShipment", cid, customerAddress, timestampBigInt)
    if err != nil { return "", 0, fmt.Errorf( format: "failed to pack arguments: %w", err) }

    nonce, err := client.PendingNonceAt(context.Background(), customerAddress)
    if err != nil { return "", 0, fmt.Errorf( format: "failed to get nonce: %w", err) }

    return "0x" + hex.EncodeToString(data), nonce, nil
}
```

Рисунок 2.10 – Реализация сервиса CreateShipmentData

- func AcceptShipmentData(client,parsedABI,carrierAddress, shipmentId) (string, uint64, error) – функция генерирующая данные для транзакции вызывающей метод acceptShipment в смарт-контракте и необходимый nonce для этой транзакции. В случае возникновения ошибки функция возвращает ее, а данные и nonce оставляет пустыми. Реализация данного сервиса на рисунке 2.11.

```go
func AcceptShipmentData(
    client *ethclient.Client,
    parsedABI abi.ABI,
    carrierAddress common.Address,
    shipmentId string,
) (string, uint64, error) {
    nonce, err := client.PendingNonceAt(context.Background(), carrierAddress)
    if err != nil { panic(err) }

    shipmentIdBigInt := new(big.Int)

    if _, ok := shipmentIdBigInt.SetString(shipmentId, base: 10); !ok {
        fmt.Println( a...: "Cannot convert timestamp to *big.Int")
    }

    data, err := parsedABI.Pack( name: "acceptShipment", shipmentIdBigInt)
    if err != nil { panic(err) }

    return "0x" + hex.EncodeToString(data), nonce, nil
}
```

Рисунок 2.11 – Реализация сервиса AcceptShipmentData

- func FinishShipmentData(client,parsedABI, contractAddress,shipmentId) (string, uint64, error) – функция генерирующая данные для транзакции вызывающей метод finishShipment в смарт-контракте и необходимый nonce для этой транзакции. В случае возникновения ошибки функция возвращает ее, а данные и nonce оставляет пустыми. Реализация данного сервиса на рисунке 2.12.

```
func FinishShipmentData(
    client *ethclient.Client,
    parsedABI abi.ABI,
    contractAddress common.Address,
    shipmentId string,
) (string, uint64, error) {
    nonce, err := client.PendingNonceAt(context.Background(), contractAddress)
    if err != nil { panic(err) }

    shipmentIdBigInt := new(big.Int)

    if _, ok := shipmentIdBigInt.SetString(shipmentId, base: 10); !ok {
        fmt.Println( a...: "Cannot convert shipmentId to *big.Int")
    }

    data, err := parsedABI.Pack( name: "finishShipment", shipmentIdBigInt)
    if err != nil { panic(err) }

    return "0x" + hex.EncodeToString(data), nonce, nil
}
```

Рисунок 2.12 – Реализация сервиса FinishShipmentData

- func GetHistoryShipments(client,parsedABI, contractAddress) ([]entity.Shipment, error) – запрашивает у смарт-контракта массив завершенных перевозок, в случае неудачи возвращает пустой массив с ошибкой. Реализация данного сервиса в приложении И.
- func GetShipments(client,parsedABI, contractAddress) ([]entity.Shipment, error) – запрашивает у смарт-контракта массив активных перевозок, работа с ошибками происходит по аналогии с сервисом GetHistoryShipment. Реализация данного сервиса в приложении К.
- func GetMyShipments(client,parsedABI, contractAddress, address) ([]entity.Shipment, error) – возвращает массив перевозок для заданного адреса. В случае возникновения сбоев возвращает ошибку с пустым массивом. Реализация данного сервиса в приложении Л.

2.2.4 Смарт-контракт ShipmentContract

В рамках реализации децентрализованной логистической системы ключевым элементов является интеграция с блокчейн сетью. Для этого был разработан смарт-контракт ShipmentContract используя язык Solidity. Для разработки и тестирования была выбрана одна из тестовых сетей. Код данного контракта находится в приложении М.

Контракт ShipmentContract обеспечивает надежную и прозрачную регистрацию перевозок. Основой контракта выступает структура данных Shipment, содержащая следующие поля:

- uint256 id – уникальный идентификатор перевозки;
- string cid – cid полученный от ipfs кластера;
- address customer – адрес заказчика;
- address carrier – адрес перевозчика;
- uint256 timestamp – временная метка;
- bool accepted – флаг отвечающий за принятие перевозки;
- bool finished – флаг отвечающий за завершение перевозки;

Контракт использует переменную shipmentCount в качестве счётчика количества созданных перевозок, для доступа к ним поддерживается массив shipmentIds.

Контракт генерирует следующие события:

- ShipmentCreated – при создании новой перевозки.
- ShipmentAccepted – при подтверждении перевозки.
- ShipmentFinished – при завершении перевозки.

Логика работы с перевозками заключена в следующих методах:

- createShipment(string calldata cid, address customer, address carrier) – позволяет создать новую перевозку. В параметрах указывается cid, адрес заказчика и перевозчика.
- acceptShipment(uint256 shipmentId) – предназначен для подтверждения перевозчиком, что он принял доставку с заданным id. Повторное подтверждение запрещено.
- finishShipment(uint256 shipmentId) – завершает доставку. Вызов возможен только для перевозок в состоянии “принято”.
- getShipment(uint256 id) – возвращает cid, адрес заказчика и перевозчика, временную метку timestamp и два флага accepted и finished для заданной перевозки через id.
- getLatestShipmentId() – вспомогательная функция возвращающая id последней созданной перевозки.
- getAllShipmentIds() – возвращает массив всех существующих id,

используется для получения подробной информации о всех перевозках.

2.3 Тестирование разработанного приложения

Важной частью разработки любого приложения является тестирование. Тестирование проверяет корректность его работы, выявляет ошибки, а также проверяет соответствие реализованного приложения его требованиям. Существует много видов тестирования: ручное тестирование, модульное тестирование, системное тестирование и интеграционное тестирование. Для тестирования реализованного приложения было проведено ручное тестирование, что позволяет оценить корректность его работы в реальных условиях.

Целью тестирования является проверка основных функций приложения:

- правильность создания перевозки с заданными параметрами;
- корректное принятие и завершение перевозки, а также закрепление за ней перевозчика;
- правильное отображение всех созданных перевозок;
- устойчивость к введенным пользовательским данным;

Тестирование проводилось вручную путем работы с интерфейсом приложения. Процесс тестирование включал:

Тестирование интерфейса приложения. Этот этап включает тестирование корректной работы полей ввода, корректного отображения информации о созданных перевозках, проверку работы всплывающих окон, предупреждающих об ошибках. Так же проверялась передача данных на сервер и верное отображение ответа него.

Тестирование серверной части включало проверку обработки запросов, проверку корректности загрузки файлов и получение данных от клиента. Особое внимание уделялось отладке формирования данных для смарт-контракта и интеграции с ipfs хранилищем. Для этого проверялось наличие данных по заданным cid, а также их соответствие.

Финальным этапом стала проверка корректности сохранения и обновления данных в смарт-контракте. Проверялось создание перевозок в сети, обновление их статусов, а также принятие и завершение. Отдельно было проверено, что в системе не создается противоречащих состояний (например, не назначенная, но завершенная перевозка).

Таким образом, тестирование позволило выявить и исправить ошибки и недоработки на раннем этапе. Благодаря этому этапу приложение стало стабильнее и надежнее.

Важной частью разработки приложения является не только реализация функционала, но и обеспечение хорошего качества кода. Для оценки сложности отдельно взятых модулей используется цикломатическая сложность. Эта метрика позволяет определить, насколько тяжело будет сопровождать и понимать код. Использование данной метрики позволяет выявить сложные участки кода, где могут быть скрыты потенциальные ошибки. Цикломатическая сложность разработана Томасом МакКейбом. Для ее вычисления строится граф потока управления. Узлами данного графа являются задачи, а ребрами пути выполнения. Обычно граф строится для отдельных частей программы, иначе бы он становился перегруженным и запутанным. На рисунке 2.13 изображены графы потоков управления операторов sequence, while, if-then-else, until.

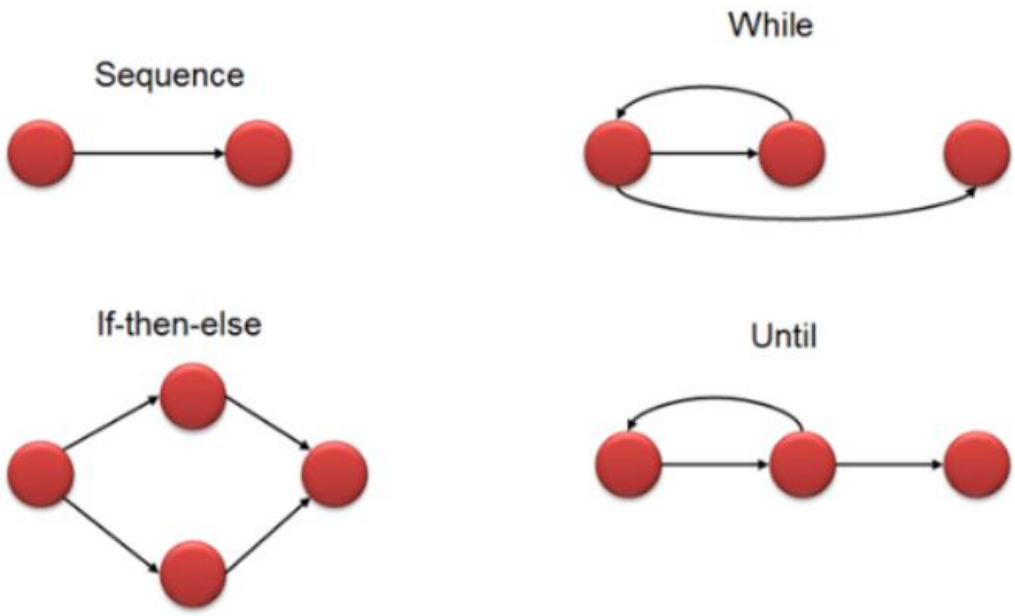

Рисунок 2.13 – Графы потоков управления операторов sequence, while, if-then-else, until

После построения графа для вычисления сложности используется формула 2.1:

$$M = E - N + 2P, \tag{2.1}$$

где M – цикломатическая сложность,

E – количество ребер (переходов между блоками),
N – количество узлов (блоков кода),
P – количество компонент связности.

Для оценки сложности разработанного приложения были посчитаны цикломатические сложности обработчиков, сервисов и методов смарт-контракта. Рассмотрим расчет на примере обработчика acceptShipment. Код этого обработчика на рисунке 2.14. Изначально тело функции было разбито на базовые блоки, которые являются узлами графа:

- базовый блок 1 – чтение и установка параметров;
- базовый блок 2 – вызов AcceptShipmentData;
- базовый блок 3 – обработка ошибки вызова AcceptShipmentData;
- базовый блок 4 – создание Response и установка заголовков ответа;
- базовый блок 5 – кодирование ответа в json формат;
- базовый блок 6 – обработка ошибки кодирования ответа;
- базовый блок 7 – выход из функции;

```
func New(client *ethclient.Client, parsedABI abi.ABI, contractAddress common.Address) http.HandlerFunc {  1 usage
    return func(w http.ResponseWriter, r *http.Request) {
        w.Header().Set( key: "Access-Control-Allow-Origin",  value: "*")
        w.Header().Set( key: "Access-Control-Allow-Methods",  value: "POST, GET, OPTIONS, PUT, DELETE")
        w.Header().Set( key: "Access-Control-Allow-Headers",  value: "Content-Type")

        carrier := r.FormValue( key: "carrier")
        shipmentId := r.FormValue( key: "shipmentId")

        data, nonce, err := contract.AcceptShipmentData(client, parsedABI, common.HexToAddress(carrier), shipmentId)
        if err != nil {
            http.Error(w, "AcceptShipmentData error: "+err.Error(), http.StatusInternalServerError)
            return
        }

        response := Response{
            To:       contractAddress.Hex(),
            Value:    "0",
            Data:     data,
            GasPrice: "60000000000",
            Nonce:    nonce,
            ChainID:  "10143",
        }

        w.Header().Set( key: "Content-Type",  value: "application/json")
        w.WriteHeader(http.StatusOK)
        if err := json.NewEncoder(w).Encode(response); err != nil {
            http.Error(w, err.Error(), http.StatusInternalServerError)
        }
    }
}
```

Рисунок 2.14 – Код обработчика acceptShipment

Далее были выделены переходы блоков, которые являются ребрами графа:

- базовый блок 1 в базовый блок 2;
- базовый блок 2 в базовый блок 3;
- базовый блок 2 в базовый блок 4;
- базовый блок 3 в базовый блок 7;
- базовый блок 4 в базовый блок 5;
- базовый блок 5 в базовый блок 6;
- базовый блок 6 в базовый блок 7;
- базовый блок 6 в базовый блок 7;

Полученный граф изображен на рисунке 2.15.

Рисунок 2.15 – Граф потоков управления обработчика acceptShipment

Применяя формулу 2.1 получаем, что цикломатическая сложность обработчика acceptShipment равна $M = E - N + 2P = 8 - 7 + 2 * 1 = 3$. Используя аналогичные вычисления были рассчитаны цикломатические сложности остальных модулей приложения (некоторые графы изображены на рисунках 2.16 – 2.21). Таблица 2.1 содержит эти данные.

Таблица 2.1 – Цикломатическая сложность основных модулей программы

Название модуля	Тип модуля	Цикломатическая сложность
AcceptShipment	Обработчик	3
CreateShipment	Обработчик	8
Download	Обработчик	3
FinishShipment	Обработчик	3
GetHistoryShipments	Обработчик	3
GetMyShipments	Обработчик	3
GetShipments	Обработчик	3
CreateShipmentData	Сервис	3
AcceptShipmentData	Сервис	4
FinishShipmentData	Сервис	4
GetHistoryShipments	Сервис	9
GetShipments	Сервис	11
GetMyShipments	Сервис	10
UploadFileToIpfs	Сервис	3
DownloadFileFromIpfs	Сервис	3

CreateShipment	Метод контракта	1
AcceptShipment	Метод контракта	3
finishShipment	Метод контракта	3
getShipment	Метод контракта	1
getLatestShipmentId	Метод контракта	2
getAllShipmentIds	Метод контракта	1

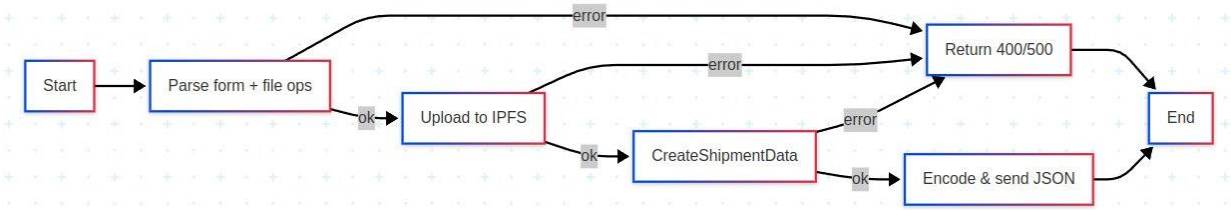

Рисунок 2.16 – Граф потоков управления обработчика createShipment

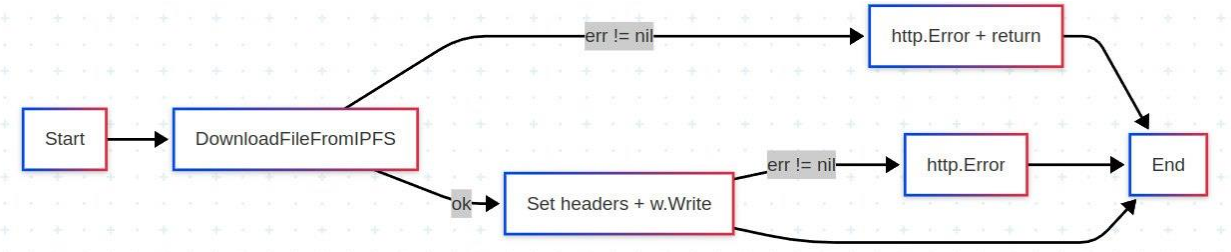

Рисунок 2.17 – Граф потоков управления обработчика download

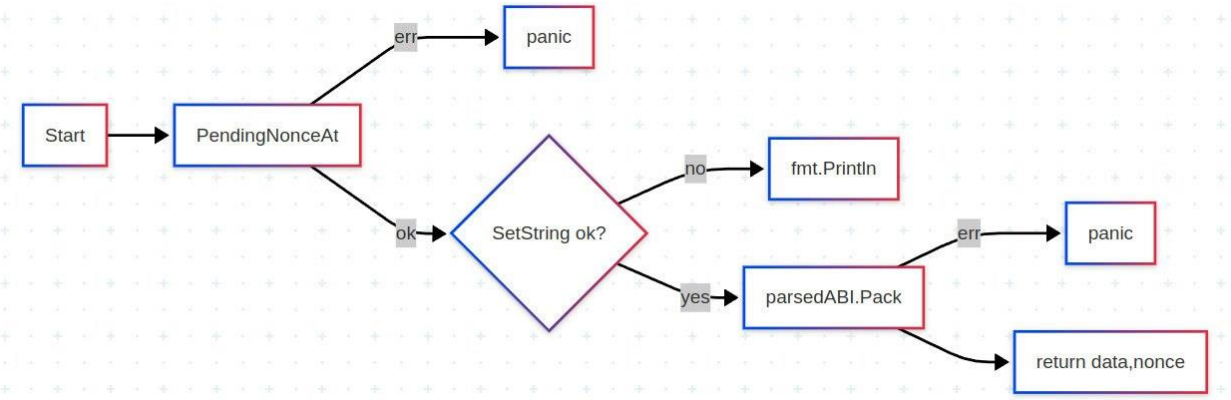

Рисунок 2.18 – Граф потоков управления сервиса createShipmentData

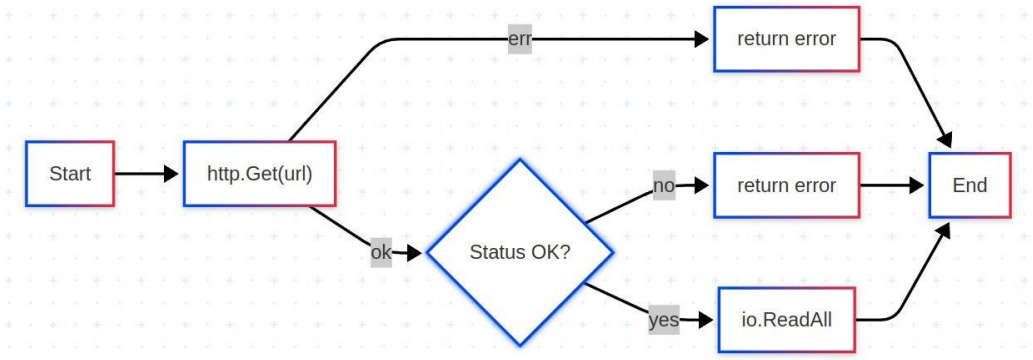

Рисунок 2.19 – Граф потоков управления сервиса DownloadFromIPFS

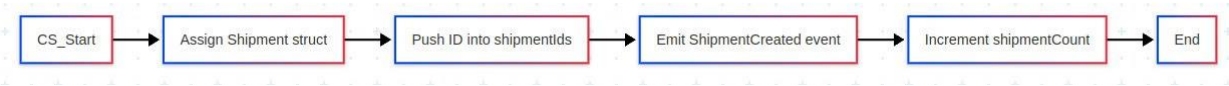

Рисунок 2.20 – Граф потоков управления метода createShipment

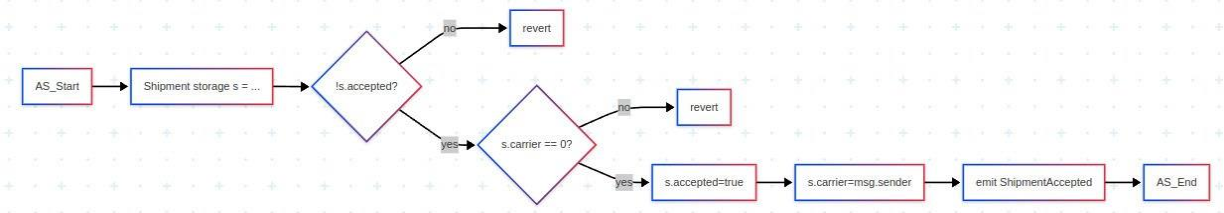

Рисунок 2.21 – Граф потоков управления метода acceptShipment

Проанализировав сложности компонентов системы видно, что обработчики имеют низкую сложность (средняя $M = 3.7$) за исключением $M = 8$ для createShipment, что связано с работой с ipfs. Сервисные функции имеют среднюю сложность $M = 6.9$ из-за множества проверок и валидаций. Методы контракта имеют предельно низкие сложности (от $M = 1$ до $M = 3$), что говорит об их простоте и надежности. Общая картина показывает, что все модули программы, в среднем, имеют хорошие показатели цикломатической сложности.

2.4 Выводы

Использование блокчейна в реализованном приложении позволило существенно оптимизировать электронный документооборот. Благодаря децентрализованному хранению логистических документов возможно прямое

взаимодействие заказчиков и перевозчиков. Применение ipfs хранилища повышает безопасность и надежность системы, а также снижает риск мошенничества и подделки документов.

Вместе с этим возросла прозрачность логистических цепочек. Каждое действие (создание перевозки, принятие перевозки, завершение перевозки) фиксируется в блокчейне и доступно для проверки всем заинтересованным сторонам в любое время. Это снижает количество ошибок, а при их возникновении позволяет быстрее на них реагировать.

Классическими системами, используемыми в логистике, являются EDI и TMS. EDI – система обмена бизнес-документами позволяющая автоматизировать их обработку. TMS – программное обеспечение для работы с транспортными операциями. Операции включают в себя оптимизацию путей, расчет стоимости перевозки, учет автопарка и так далее. В таблице 2.2 сравнение этих систем с реализованным приложением.

Таблица 2.2 – Сравнение EDI, TMS и реализованного приложения

Критерий сравнения	EDI-системы	TMS-системы	Реализация с использованием блокчейна
Документооборот	Централизованно хранится у каждого участника, из-за чего может возникать расхождение версий	Проблематичный внешний доступ	Децентрализованное хранение, легкая доступность заинтересованным сторонам.
Прозрачность цепочки	Ограничена для внешних участников	Прозрачность внутри компании	Полная прозрачность
Сложность интеграции	Низкая	Средняя	Высокая
Безопасность	Централизованная, зависит от реализации системы		Неизменяемые записи, криптографическая защита
Скорость	Медленно	Медленно для внешнего доступа	Мгновенно

Таким образом, классические решения позволяют начать быстро использовать необходимый функционал, но имеют ряд недостатков, связанных с прозрачностью и безопасностью. Реализованное приложение решает эти проблемы, но является дорогостоящим и сложным для внедрения.

ЗАКЛЮЧЕНИЕ

В данной работе рассмотрено применение технологии блокчейн для оптимизации логистических процессов, определены её преимущества и ограничения, а также перспективы дальнейшего развития.

В первой главе проанализированы ключевые аспекты блокчейн-технологии: её архитектура, преимущества и недостатки. Среди основных плюсов отмечены прозрачность, неизменность данных и автоматизация процессов с помощью смарт-контрактов. Недостатками остаются высокая стоимость внедрения, потребность в масштабируемости и сложность интеграции с существующими системами. Также проведён анализ логистического рынка, где выделены ключевые проблемы: недостаточная прозрачность, сложная координация участников и низкий уровень доверия между сторонами. Изучены примеры применения блокчейна на практике на примере компаний Walmart и Maersk. Опыт показал, что технология позволяет существенно повысить прозрачность цепочек поставок, сократить документооборот и минимизировать ошибки при передаче данных. Однако реализация подобных решений требует значительных затрат и времени на внедрение.

Во второй главе было разработано приложение, обеспечивающие прямое взаимодействие заказчика с перевозчиком. Основой приложения выступает смарт-контракт ShipmentContract, позволяющий пользователям создавать, принимать и завершать перевозки. Приложение демонстрирует применимость блокчейн в этой сфере и обладает рядом преимуществ – надежностью, безопасностью, прозрачностью и автоматизацией.

Таким образом, блокчейн имеет значительный потенциал для оптимизации логистических процессов, особенно в вопросах прозрачности, доверия и автоматизации. Однако его внедрение требует преодоления ряда технических и организационных барьеров, что делает дальнейшие исследования в этом направлении особенно актуальными.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Введение в логистику [Электронный ресурс]. – Режим доступа http://www.kgau.ru/distance/fub_03/eldeshtein/logistika/01_01.html – Дата доступа: 10.05.2025
2. Документация IPFS хранилища [Электронный ресурс]. – Режим доступа https://docs.ipfs.tech/ – Дата доступа: 10.05.2025.
3. Документация языка Solidity [Электронный ресурс]. – Режим доступа https://docs.soliditylang.org/en/v0.8.30/. – Дата доступа: 13.05.2025.
4. Дрешер, Д. Основы блокчейна / Д. Дрешер. — М.: ДМК Пресс, 2018. — 125 с.
5. Зиборев А.В. Использование цепочек blockchain и искусственного интеллекта в сфере логистики и автоперевозок / А.В. Зиборев.
6. Куприяновский В.П., Синягов С.А., Климов А.А., Петров А.В., Намиот Д.Е. Цифровые цепи поставок и технологии на базе блокчейн в совместной экономике» //International Journal of Open Information Technologies. – 2017. – № 8. – с. 80–95.
7. Левкин, Г.Г. Логистика: теория и практика / Г.Г. Левкин. — Москва: Юрайт, 2024. — 187 с.
8. Национальный Интернет-портал Республики Беларусь [Электронный ресурс] / Нац. Центр правовой информ. Респ. Беларусь. — Минск 2024. Режим доступа: https://pravo.by/document/?guid=3871&p0=P32100292. — Дата доступа: 01.12.2024.
9. Сергеев Е.В. Применение инновационной технологии «Блокчейн» в логистике и управлении цепями поставок / Е.В. Сергеев. — 140 с.
10. Сергеев В.И. Управление цепями поставок. Учебник для бакалавров и магистров. — Москва: Юрайт. – 479 с.
11. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems [Electronic Resource]. – Mode of access: https://people.csail.mit.edu/rivest/Rsapaper.pdf. Date of access: 20.05.2025
12. Deterministic Constructions of 21-Step Collisions for the SHA-2 Hash Family [Electronic Resource]. – Mode of access: https://link.springer.com/chapter/10.1007/978-3-540-85886-7_17. Date of access: 20.05.2025
13. Diffie W., New Directions in Cryptography / W. Diffie. – Boston: Addison Wesley, 2005. – 644 p.

ПРИЛОЖЕНИЕ А

РЕАЛИЗАЦИЯ НАВИГАЦИОННОЙ ПАНЕЛИ

```
import React, {useState} from 'react';
import classes from './Navbar.module.css';
import NavbarItem from "./NavbarItem";
import {ethers} from "ethers";
import {shortenAddress} from "../utils";
const Navbar = ({navbarItems, navbarClickHandler, address, setAddress, signer, setSigner}) => {
    const connectWallet = async () => {
        if (window.ethereum) {
            try {
                await window.ethereum.request({method: "eth_requestAccounts"});
                const provider = new ethers.BrowserProvider(window.ethereum);
                const signer = await provider.getSigner();
                setSigner(signer);
                setAddress(await signer.getAddress());
            } catch (error) {
                console.error("Ошибка подключения кошелька:", error);
            }
        } else {
            alert("Не найдено");
        }
    };

    return (
        <div className={classes.container}>
            {
                navbarItems.map(navbarItem => (
                    <NavbarItem navbarItem={navbarItem} navbarClickHandler={navbarClickHandler}></NavbarItem>
                ))
            }


            <div className={classes.wallet_connect_container}>
                <button onClick={connectWallet}>
```

```
                {
                    address ? <>{shortenAddress(address)}</> : <>Подключить
Кошелек</>
                }
            </button>
        </div>
    </div>
  );
};

export default Navbar;
```

ПРИЛОЖЕНИЕ Б

РЕАЛИЗАЦИЯ ОБРАБОТЧИКА /API/ACCEPTSHIPMENT

```
type Response struct {
	To       string `json:"to"`
	Value    string `json:"value"`
	Data     string `json:"data"`
	GasPrice string `json:"gasPrice"`
	Nonce    uint64 `json:"nonce"`
	ChainID  string `json:"chainId"`
}

func New(client *ethclient.Client, parsedABI abi.ABI, contractAddress common.Address) http.HandlerFunc {
	return func(w http.ResponseWriter, r *http.Request) {
		w.Header().Set("Access-Control-Allow-Origin", "*")
		w.Header().Set("Access-Control-Allow-Methods", "POST, GET, OPTIONS, PUT, DELETE")
		w.Header().Set("Access-Control-Allow-Headers", "Content-Type")

		carrier := r.FormValue("carrier")
		shipmentId := r.FormValue("shipmentId")

		data, nonce, err := contract.AcceptShipmentData(client, parsedABI, common.HexToAddress(carrier), shipmentId)
		if err != nil {
			http.Error(w, "AcceptShipmentData error: "+err.Error(), http.StatusInternalServerError)
			return
		}

		response := Response{
			To:       contractAddress.Hex(),
			Value:    "0",
			Data:     data,
			GasPrice: "60000000000",
			Nonce:    nonce,
			ChainID:  "10143",
```

```
                }

                w.Header().Set("Content-Type", "application/json")
                w.WriteHeader(http.StatusOK)
                if err := json.NewEncoder(w).Encode(response); err != nil {
                        http.Error(w, err.Error(), http.StatusInternalServerError)
                }
        }
}
```

ПРИЛОЖЕНИЕ В

РЕАЛИЗАЦИЯ ОБРАБОТЧИКА /API/ CREATESHIPMENT

```
type Response struct {
	To       string `json:"to"`
	Value    string `json:"value"`
	Data     string `json:"data"`
	GasPrice string `json:"gasPrice"`
	Nonce    uint64 `json:"nonce"`
	ChainID  string `json:"chainId"`
}

func New(client *ethclient.Client, parseABI abi.ABI, contractAddress common.Address) http.HandlerFunc {
	return func(w http.ResponseWriter, r *http.Request) {
		w.Header().Set("Access-Control-Allow-Origin", "*")
		w.Header().Set("Access-Control-Allow-Methods", "POST, GET, OPTIONS, PUT, DELETE")
		w.Header().Set("Access-Control-Allow-Headers", "Content-Type")

		err := r.ParseMultipartForm(20 << 20)
		if err != nil {
			http.Error(w, "File parse error: "+err.Error(), http.StatusBadRequest)
			return
		}

		file, handler, err := r.FormFile("file")
		if err != nil {
			http.Error(w, "File read error: "+err.Error(), http.StatusBadRequest)
			return
		}
		defer file.Close()

		customer := r.FormValue("customer")
		timestamp := r.FormValue("timestamp")
```

```
		os.MkdirAll("uploads", os.ModePerm)

		dst, err := os.Create("uploads/" + handler.Filename)
		if err != nil {
			http.Error(w, "File save error: "+err.Error(),
http.StatusInternalServerError)
			return
		}
		defer dst.Close()

		_, err = io.Copy(dst, file)
		if err != nil {
			http.Error(w, "File save error: "+err.Error(),
http.StatusInternalServerError)
			return
		}

		cid, err := ipfs.UploadFileToIPFS("uploads/" + handler.Filename)
		if err != nil {
			http.Error(w, "IPFS error: "+err.Error(),
http.StatusInternalServerError)
			return
		}

		w.WriteHeader(http.StatusOK)

		data, nonce, err := contract.CreateShipmentData(client, parseABI, cid,
common.HexToAddress(customer), timestamp)
		if err != nil {
			http.Error(w, "CreateShipmentData error: "+err.Error(),
http.StatusInternalServerError)
			return
		}

		response := Response{
			To:       contractAddress.Hex(),
			Value:    "0",
			Data:     data,
			GasPrice: "60000000000",
			Nonce:    nonce,
```

```
                ChainID:  "10143",
            }

            w.Header().Set("Content-Type", "application/json")
            w.WriteHeader(http.StatusOK)
            if err := json.NewEncoder(w).Encode(response); err != nil {
                http.Error(w, err.Error(), http.StatusInternalServerError)
            }
        }
    }
```

ПРИЛОЖЕНИЕ Г

РЕАЛИЗАЦИЯ ОБРАБОТЧИКА /API/DOWNLOAD/{CID}

```
func New() http.HandlerFunc {
	return func(w http.ResponseWriter, r *http.Request) {
		w.Header().Set("Access-Control-Allow-Origin", "*")
		w.Header().Set("Access-Control-Allow-Methods", "POST, GET, OPTIONS, PUT, DELETE")
		w.Header().Set("Access-Control-Allow-Headers", "Content-Type")

		vars := mux.Vars(r)
		cid := vars["cid"]

		fileContent, err := ipfs.DownloadFileFromIPFS(cid)
		if err != nil {
			http.Error(w, "File ipfs download error: "+err.Error(), http.StatusInternalServerError)
			return
		}

		w.Header().Set("Content-Type", "application/octet-stream")
		w.Header().Set("Content-Disposition", fmt.Sprintf("attachment; filename=\"%s.pdf\"", cid))
		w.WriteHeader(http.StatusOK)

		_, err = w.Write(fileContent)
		if err != nil {
			http.Error(w, "File write error: "+err.Error(), http.StatusInternalServerError)
		}
	}
}
```

ПРИЛОЖЕНИЕ Д

РЕАЛИЗАЦИЯ ОБРАБОТЧИКА /API/FINISHSHIPMENT

```
type Response struct {
	To       string `json:"to"`
	Value    string `json:"value"`
	Data     string `json:"data"`
	GasPrice string `json:"gasPrice"`
	Nonce    uint64 `json:"nonce"`
	ChainID  string `json:"chainId"`
}

func New(client *ethclient.Client, parsedABI abi.ABI, contractAddress common.Address) http.HandlerFunc {
	return func(w http.ResponseWriter, r *http.Request) {
		w.Header().Set("Access-Control-Allow-Origin", "*")
		w.Header().Set("Access-Control-Allow-Methods", "POST, GET, OPTIONS, PUT, DELETE")
		w.Header().Set("Access-Control-Allow-Headers", "Content-Type")

		shipmentId := r.FormValue("shipmentId")

		data, nonce, err := contract.FinishShipmentData(client, parsedABI, contractAddress, shipmentId)
		if err != nil {
			http.Error(w, "FinishShipmentData error: "+err.Error(), http.StatusBadRequest)
			return
		}

		response := Response{
			To:       contractAddress.Hex(),
			Value:    "0",
			Data:     data,
			GasPrice: "60000000000",
			Nonce:    nonce,
			ChainID:  "10143",
		}
```

```
                w.Header().Set("Content-Type", "application/json")
                w.WriteHeader(http.StatusOK)
                if err := json.NewEncoder(w).Encode(response); err != nil {
                        http.Error(w, err.Error(), http.StatusInternalServerError)
                }
        }
}
```

ПРИЛОЖЕНИЕ Е

РЕАЛИЗАЦИЯ ОБРАБОТЧИКА /API/GETHISTORYSHIPMENT

```
func New(client *ethclient.Client, parsedABI abi.ABI, contractAddress common.Address) http.HandlerFunc {
	return func(w http.ResponseWriter, r *http.Request) {
		w.Header().Set("Access-Control-Allow-Origin", "*")
		w.Header().Set("Content-Type", "application/json")

		shipments, err := contract.GetHistoryShipments(client, parsedABI, contractAddress)
		if err != nil {
			panic(err)
		}
		err = json.NewEncoder(w).Encode(shipments)
		if err != nil {
			http.Error(w, "Shipments writing error: "+err.Error(), http.StatusBadRequest)
			return
		}
	}
}
```

ПРИЛОЖЕНИЕ Ж

РЕАЛИЗАЦИЯ ОБРАБОТЧИКА /API/GETSHIPMENTS

```
func New(client *ethclient.Client, parsedABI abi.ABI, contractAddress common.Address) http.HandlerFunc {
	return func(w http.ResponseWriter, r *http.Request) {
		w.Header().Set("Access-Control-Allow-Origin", "*")
		w.Header().Set("Content-Type", "application/json")

		shipments, err := contract.GetShipments(client, parsedABI, contractAddress)
		if err != nil {
			panic(err)
		}
		err = json.NewEncoder(w).Encode(shipments)
		if err != nil {
			http.Error(w, "Shipments writing error: "+err.Error(), http.StatusBadRequest)
			return
		}
	}
}
```

ПРИЛОЖЕНИЕ 3

РЕАЛИЗАЦИЯ ОБРАБОТЧИКА /API/GETMYSHIPMENTS

```
func New(client *ethclient.Client, parsedABI abi.ABI, contractAddress common.Address) http.HandlerFunc {
	return func(w http.ResponseWriter, r *http.Request) {
		w.Header().Set("Access-Control-Allow-Origin", "*")
		w.Header().Set("Content-Type", "application/json")

		address := r.FormValue("address")

		shipments, err := contract.GetMyShipments(client, parsedABI, contractAddress, common.HexToAddress(address))
		if err != nil {
			panic(err)
		}
		err = json.NewEncoder(w).Encode(shipments)
		if err != nil {
			http.Error(w, "Shipments writing error: "+err.Error(), http.StatusBadRequest)
			return
		}
	}
}
```

ПРИЛОЖЕНИЕ И

РЕАЛИЗАЦИЯ СЕРВИСА GETHISTORYSHIPMENTS

```
func GetHistoryShipments(client *ethclient.Client,
	parsedABI abi.ABI,
	contractAddress common.Address) ([]entity.Shipment, error) {

	data, err := parsedABI.Pack("getLatestShipmentId")
	if err != nil {
		return nil, err
	}

	callMsg := ethereum.CallMsg{
		To:   &contractAddress,
		Data: data,
	}

	result, err := client.CallContract(context.Background(), callMsg, nil)
	if err != nil {
		return nil, err

	}

	var shipmentCount *big.Int
	if err := parsedABI.UnpackIntoInterface(&shipmentCount,
"getLatestShipmentId", result); err != nil {
		return nil, err

	}
	shipments := []entity.Shipment{}

	for i := int64(0); i <= shipmentCount.Int64(); i++ {
		data, err := parsedABI.Pack("getShipment", big.NewInt(i))
		if err != nil {
			log.Println("Pack error:", err)
			continue
		}
```

```
            callMsg.Data = data

            result, err := client.CallContract(context.Background(), callMsg, nil)
            if err != nil {
                continue
            }

            var out struct {
                Cid       string
                Customer  common.Address
                Carrier   common.Address
                Timestamp *big.Int
                Accepted  bool
                Finished  bool
            }
            if err := parsedABI.UnpackIntoInterface(&out, "getShipment", result);
err != nil {
                continue
            }

            if len(out.Cid) == 0 || !out.Finished {
                continue
            }

            shipments = append(shipments, entity.Shipment{
                ID:        uint64(i),
                CID:       out.Cid,
                Customer:  out.Customer.Hex(),
                Carrier:   out.Carrier.Hex(),
                Timestamp: out.Timestamp.Uint64(),
                Accepted:  out.Accepted,
                Finished:  out.Finished,
            })

            time.Sleep(time.Millisecond * 250)
        }

        return shipments, nil}
```

ПРИЛОЖЕНИЕ К

РЕАЛИЗАЦИЯ СЕРВИСА GETSHIPMENTS

```
func GetShipments(client *ethclient.Client,
	parsedABI abi.ABI,
	contractAddress common.Address) ([]entity.Shipment, error) {

	data, err := parsedABI.Pack("getLatestShipmentId")
	if err != nil {
		return nil, err
	}

	callMsg := ethereum.CallMsg{
		To:   &contractAddress,
		Data: data,
	}

	result, err := client.CallContract(context.Background(), callMsg, nil)
	if err != nil {
		return nil, err
	}

	var shipmentCount *big.Int
	if err := parsedABI.UnpackIntoInterface(&shipmentCount,
"getLatestShipmentId", result); err != nil {
		return nil, err
	}

	shipments := []entity.Shipment{}

	if shipmentCount == nil {
		return nil, nil
	}

	for i := int64(0); i <= shipmentCount.Int64(); i++ {
		data, err := parsedABI.Pack("getShipment", big.NewInt(i))
		if err != nil {
			log.Println("Pack error:", err)
```

```
                continue
            }

            callMsg.Data = data

            result, err := client.CallContract(context.Background(), callMsg, nil)
            if err != nil {
                log.Println("Ошибка вызова getShipment:", err)
                continue
            }

            var out struct {
                Cid       string
                Customer  common.Address
                Carrier   common.Address
                Timestamp *big.Int
                Accepted  bool
                Finished  bool
            }
            if err := parsedABI.UnpackIntoInterface(&out, "getShipment", result);
err != nil {
                log.Println("Unpack error:", err)
                continue
            }

            if out.Finished {
                continue
            }

            if len(out.Cid) == 0 {
                continue
            }

            shipments = append(shipments, entity.Shipment{
                ID:        uint64(i),
                CID:       out.Cid,
                Customer:  out.Customer.Hex(),
                Carrier:   out.Carrier.Hex(),
                Timestamp: out.Timestamp.Uint64(),
                Accepted:  out.Accepted,
```

```
                Finished:  out.Finished,
            })

            time.Sleep(time.Millisecond * 250)
        }

        return shipments, nil
}
```

ПРИЛОЖЕНИЕ Л

РЕАЛИЗАЦИЯ СЕРВИСА GETMYSHIPMENTS

```
func GetMyShipments(client *ethclient.Client,
	parsedABI abi.ABI,
	contractAddress common.Address, address common.Address)
([]entity.Shipment, error) {

	data, err := parsedABI.Pack("getLatestShipmentId")
	if err != nil {
		return nil, err
	}

	callMsg := ethereum.CallMsg{
		To:   &contractAddress,
		Data: data,
	}

	result, err := client.CallContract(context.Background(), callMsg, nil)
	if err != nil {
		return nil, err
	}

	var shipmentCount *big.Int
	if err := parsedABI.UnpackIntoInterface(&shipmentCount,
"getLatestShipmentId", result); err != nil {
		return nil, err
	}

	shipments := []entity.Shipment{}

	if shipmentCount == nil {
		return nil, nil
	}

	for i := int64(0); i <= shipmentCount.Int64(); i++ {
		data, err := parsedABI.Pack("getShipment", big.NewInt(i))
		if err != nil {
```

```
                log.Println("Pack error:", err)
                continue
            }

            callMsg.Data = data

            result, err := client.CallContract(context.Background(), callMsg, nil)
            if err != nil {
                log.Println("Ошибка вызова getShipment:", err)
                continue
            }

            var out struct {
                Cid       string
                Customer  common.Address
                Carrier   common.Address
                Timestamp *big.Int
                Accepted  bool
                Finished  bool
            }
            if err := parsedABI.UnpackIntoInterface(&out, "getShipment", result);
err != nil {
                log.Println("Unpack error:", err)
                continue
            }

            if (out.Carrier == address || out.Customer == address) && !out.Finished
{
                shipments = append(shipments, entity.Shipment{
                    ID:        uint64(i),
                    CID:       out.Cid,
                    Customer:  out.Customer.Hex(),
                    Carrier:   out.Carrier.Hex(),
                    Timestamp: out.Timestamp.Uint64(),
                    Accepted:  out.Accepted,
                    Finished:  out.Finished,
                })
            }

            time.Sleep(time.Millisecond * 250)
```

```
        }

        return shipments, nil
}
```

ПРИЛОЖЕНИЕ М

КОД СМАРТ-КОНТАРКТА SHIPMENTCONTRACT

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

contract ShipmentContract {
    struct Shipment {
        uint256 id;
        string cid;
        address customer;
        address carrier;
        uint256 timestamp;
        bool accepted;
        bool finished;
    }

    uint256 public shipmentCount;
    mapping(uint256 => Shipment) public shipments;
    uint256[] private shipmentIds;

    event ShipmentCreated(
        uint256 indexed id,
        string cid,
        address indexed customer,
        address indexed carrier,
        uint256 timestamp
    );

    event ShipmentAccepted(
        uint256 indexed id,
        address indexed carrier
    );

    event ShipmentFinished(
        uint256 indexed id
    );
```

```
    function createShipment(string calldata cid, address customer, address carrier)
external {
        shipments[shipmentCount] = Shipment({
            id: shipmentCount,
            cid: cid,
            customer: customer,
            carrier: carrier,
            timestamp: block.timestamp,
            accepted: false,
            finished: false
        });

        shipmentIds.push(shipmentCount);

        emit ShipmentCreated(
            shipmentCount,
            cid,
            customer,
            carrier,
            block.timestamp
        );

        shipmentCount++;
    }

    function acceptShipment(uint256 shipmentId) external {
        Shipment storage s = shipments[shipmentId];
        require(!s.accepted, "Shipment already accepted");

        s.accepted = true;

        emit ShipmentAccepted(shipmentId, msg.sender);
    }

    function finishShipment(uint256 shipmentId) external {
        Shipment storage s = shipments[shipmentId];
        require(s.accepted, "Shipment not accepted yet");
        require(!s.finished, "Shipment already finished");

        s.finished = true;
```

```
        emit ShipmentFinished(shipmentId);
    }

    function getShipment(uint256 id) external view returns (
        string memory cid,
        address customer,
        address carrier,
        uint256 timestamp,
        bool accepted,
        bool finished
    ) {
        Shipment storage s = shipments[id];
        return (s.cid, s.customer, s.carrier, s.timestamp, s.accepted, s.finished);
    }

    function getLatestShipmentId() external view returns (uint256) {
        return shipmentCount == 0 ? 0 : shipmentCount - 1;
    }

    function getAllShipmentIds() external view returns (uint256[] memory) {
        return shipmentIds;
    }
}
```