

**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБЩЕОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО
ОБРАЗОВАНИЯ
«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им В.Г.ШУХОВА»
(БГТУ им. В.Г.Шухова)**

Кафедра программного обеспечения вычислительной техники и
автоматизированных систем

КУРСОВОЙ ПРОЕКТ

по дисциплине: Компьютерные сети

тема: Анализ способов защиты от ошибок передаваемой информации по сети.

Автор работы _____ Гринева Марина Сергеевна, ПВ-222
(подпись)

Руководитель проекта _____ Федотов Евгений Александрович
(подпись)

Оценка _____

Белгород 2025

Содержание

Введение	3
Обзор предметной области	4
Цель и задачи	4
Объект и предмет исследования	4
Методы исследования	4
Научная новизна и практическая значимость	5
Теоретическая часть	6
Понятие и причины возникновения ошибок при передаче данных	6
Классификация методов защиты от ошибок	7
Контроль четности	8
Избыточные коды. Код Хэмминга	9
Циклический избыточный код (CRC)	10
Сравнительный анализ методов защиты от ошибок	11
Практическая часть	14
Разрабатываемая программа	14
Ознакомление со средой разработки и техническими средствами	15
Архитектура программного решения	17
Реализация метода CRC и проверка целостности	21
Формат заголовка прикладного протокола	22
Логика клиента	23
Логика сервера	25
Пользовательский интерфейс	27
Методика тестирования	32
Заключение	39
Список литературы	41
Приложение	42

Введение

В информационный век, где ключевым объектом деятельности человека и функционирования различных систем является информация и данные, современные сети передачи данных получили абсолютную значимость. Однако при передаче данных по сети могут возникать ошибки из-за помех, неисправностей оборудования или программных сбоев. Для обеспечения надежной передачи данных используются различные методы защиты от ошибок. Эти методы позволяют либо обнаружить повреждения информации, либо — в более продвинутых случаях — восстановить искаженную часть данных без необходимости повторной передачи.

Актуальность разработки инструментов защиты от ошибок при передаче данных по сети обусловлена стремительным ростом объема передаваемой информации и расширением применения сетевых технологий, в том числе в системах интернета вещей и распределённых вычислений. Современные сетевые протоколы, особенно такие как UDP, не гарантируют надежную доставку данных, что требует внедрения дополнительных механизмов контроля целостности.

Теоретическая значимость работы заключается в обобщении и систематизации подходов к обеспечению надежности сетевой передачи данных, что имеет важное значение при проектировании устойчивых коммуникационных систем. Практическая ценность определяется возможностью применения полученной реализации в учебных проектах, а также при разработке прикладных протоколов для нестабильных сетевых сред, таких как беспроводные соединения или IoT-устройства.

Обзор предметной области

Цель и задачи

Цель данной курсовой работы заключается в изучении и сравнительном анализе методов защиты от ошибок при передаче информации по сети, а также в разработке и практической реализации программной модели, демонстрирующей один из таких методов — контроль целостности с использованием CRC в сочетании с протоколом ARQ.

Конкретные задачи на разработку:

- 1) Теоретические задачи:
 - Провести обзор существующих методов защиты от ошибок при передаче информации по сети.
 - Выполнить сравнительный анализ этих методов.
 - Выбрать один из методов для реализации.
- 2) Практические задачи:
 - Разработать программу, моделирующую метод защиты от ошибок при передаче информации по сети.
 - Провести тестирование работоспособности программы при намеренном внесении ошибок в передаваемые фрагменты.

Объект и предмет исследования

Объектом исследования выступают алгоритмы обнаружения и коррекции ошибок при передаче данных.

Предметом исследования являются программные и теоретические принципы функционирования защитных механизмов, таких как циклический избыточный код (CRC), код Хэмминга и контроль четности.

Методы исследования

- 1) Анализ научной литературы и технической документации.
- 2) Программная реализация на языке Python с использованием библиотек:
 - Socket для реализации сетевого взаимодействия по протоколу UDP;
 - Select для контроля времени ожидания при получении данных и реализации ARQ (Stop & Wait);
 - Tkinter для создания графического интерфейса.
- 3) Экспериментальное тестирование функциональности приложения

Научная новизна и практическая значимость

Научная новизна работы заключается в:

1) Разработке собственного протокола поверх UDP, обеспечивающего отказоустойчивость передачи данных с помощью механизма ARQ (Stop & Wait) и встроенного контроля целостности на основе CRC

2) Интеграции симуляции ошибок в процессе передачи. Реализован механизм намеренного искажения фрагментов по выбору пользователя, что позволяет тестировать устойчивость протокола к ошибкам в контролируемых условиях.

3) Автоматической реконструкции передаваемых файлов и сообщений. Приёмник корректно собирает файл из фрагментов, а при потере данных отправляет запрос на повторную передачу. Протокол работает без необходимости синхронной сортировки — всё реализовано через диалоговое взаимодействие и управление состоянием соединения.

Практическая значимость работы проявляется в:

- Использовании в образовательных целях для демонстрации принципов работы ARQ-протоколов, CRC-контроля и передачи сообщений через нестабильные сети.
- Применении для диагностики и отладки приложений, использующих UDP.
- Возможности интеграции в системы передачи данных для локальных сетей.

Теоретическая часть

Понятие и причины возникновения ошибок при передаче данных

В процессе передачи данных по компьютерным сетям неизбежно возникают искажения, связанные с влиянием как физических, так и логических факторов. Ошибки могут проявляться в виде изменения, потери или дублирования битов в передаваемом потоке.

Основные причины возникновения ошибок:

- Электромагнитные помехи. Влияние внешних электромагнитных полей (например, от бытовых приборов или промышленного оборудования) способно вызвать искажение битов, особенно в неэкранированных кабелях или при передаче по воздуху.
- Затухание и отражения сигнала. При передаче на большие расстояния или по среде с неоднородностями сигнал может ослабевать или отражаться, вызывая интерференцию и ошибки в интерпретации данных.
- Сбой в аппаратном обеспечении. Некачественные сетевые адаптеры, повреждённые кабели, устаревшие коммутаторы или маршрутизаторы могут приводить к непредсказуемым ошибкам при передаче.
- Ошибки на программном уровне. Программные сбои или некорректная реализация протоколов обмена также способны исказить данные до момента их получения или обработки.
- Коллизии в сетях. Особенно актуальны в средах с разделяемым доступом (например, Ethernet без коммутации), где два узла могут передавать одновременно, вызывая повреждение кадров.
- Потери и задержки в передаче. В случае перегрузки сети или нестабильного соединения (например, в беспроводных сетях), пакеты могут быть утеряны, повреждены или доставлены с ошибками.

Ошибки могут носить случайный (одноразовое искажение одного или нескольких битов) или систематический (повторяющийся паттерн повреждений) характер. Их своевременное обнаружение и корректировка — ключевой элемент надёжной сетевой передачи.

Для решения этих задач применяются специализированные методы защиты, которые позволяют:

- обнаружить наличие ошибки (например, метод контрольной суммы или паритет);
- восстановить данные без повторной передачи (например, код Хэмминга);
- инициировать повторную отправку данных при обнаружении искажений (например, ARQ Stop & Wait).

Понимание природы ошибок при передаче данных помогает эффективно выбрать подходящий метод защиты, что напрямую влияет на надёжность коммуникационных систем.

Классификация методов защиты от ошибок

Существующие методы обеспечения надёжности передачи данных можно условно разделить на две основные группы:

- 1) Методы исправления ошибок, которые позволяют не только обнаружить, но и восстановить оригинальные данные. Пример: код Хэмминга.
- 2) Методы обнаружения ошибок, которые позволяют выявить факт искажения данных, но не восстанавливают их. Примеры: контроль четности, CRC.

Для первого подхода в передаваемый стек данных добавляется дополнительная информация. Такой подход позволяет выявить ошибки при передаче данных, а также исправить их. Второй подход, как и первый, заключается в использовании дополнительной информации, но отличается алгоритмом действий при обнаружении ошибок - при обнаружении ошибки данный метод запрашивает повторную передачу данных у источника. Э.С. Таненбаум в своей работе «Компьютерные сети» определил первый подход как «подход, который использует корректирующие коды (error-correcting codes)», второй — как «подход, который использует коды для обнаружения ошибок (error-detecting codes)»

Некоторые источники выделяют также протоколы с повторной передачей (ARQ — Automatic Repeat reQuest) как третью категорию. Однако в контексте данной работы ARQ не рассматривается как отдельный метод защиты, а трактуется как сопровождающий механизм, который дополняет основной метод контроля целостности данных, он не предотвращает ошибки и не обнаруживает

их самостоятельно, а лишь реагирует на сигналы об ошибках, полученные от механизмов контроля.

Каждый из подходов имеет свои преимущества и области применения. Например, в сетях с высокой степенью надежности, таких как оптоволоконные, более экономичным вариантом является использование кодов для выявления ошибок с последующей повторной передачей поврежденных данных. В беспроводных сетях, где ошибки возникают чаще, выгоднее использовать корректирующие коды, которые позволяют восстановить данные без повторной передачи.

К основным методам защиты от ошибок при передаче данных относят: циклический избыточный код (CRC), избыточные коды (например, код Хэмминга), коды с проверкой на чётность.

Контроль четности

Алгоритм проверки на чётность заключается в добавлении в каждый блок информации одного бита, который при дальнейшем декодировании указывает на четность или нечетность переданной единицы информации. Данный алгоритм является одним из самых простых и, как следствие, распространенных способов выявления ошибок.

Этот метод отличается легкостью реализации и требует минимальных вычислительных ресурсов, что делает его особенно удобным для применения в системах с ограниченными ресурсами, таких как микроконтроллеры и простые сенсорные сети.

Метод контроля четности не восстанавливает данные, но позволяет обнаружить одиночные ошибки. Механизм реализуется следующим образом: при записи или передаче данных в каждый блок информации добавляется специальный «бит четности» или же, по-другому, «паритетный бит». При чтении или получении данных данный бит пересчитывается, что дает информацию о целостности данных. Паритетный бит вычисляется как сумма всех бит данных по модулю 2. Если значение бита меняется при передаче данных (например, с 0 на 1), значит, возникли ошибки. Метод довольно прост, но имеет свой недостаток: если в ходе передачи данных изменилось два бита одновременно, то контрольная сумма битов при получении или чтении информации останется неизменной, что введет систему в заблуждение и не

позволит выявить ошибку. То есть метод обнаруживает лишь нечётное количество ошибок.

Несмотря на этот недостаток, около 90% случайных ошибок затрагивают лишь один бит, что делает проверку четности подходящим решением для большинства случаев.

Избыточные коды. Код Хэмминга

Избыточное кодирование (от англ. Redundant encoding — это метод кодирования, в основе которого лежит передача избыточного количества информации с целью дальнейшего контроля целостности данных при их получении или передачи. Код Хэмминга — один из самых известных и первых кодов, которые могут сами обнаруживать и исправлять ошибки. Они основаны на двоичной системе счисления. Код Хэмминга позволяет закодировать информационное сообщение таким образом, чтобы после передачи можно было определить наличие ошибок и, если возможно, восстановить сообщение. Алгоритм включает две части: кодирование сообщения с добавлением контрольных битов и декодирование с проверкой этих битов и исправлением ошибок.

Кодирование:

- Информационные и проверочные биты размещаются по определённым индексам (обычно степени двойки: 1, 2, 4, ...).
- Проверочные биты вычисляются как XOR определённых подмножеств битов.

Декодирование и проверка:

- На приёме производится повторный расчёт контрольных битов.
- Если возникает расхождение — его позиция указывает на ошибочный бит (позиция = сумма индексов «ошибочных» контрольных битов).
- При наличии ошибки, её можно исправить инвертированием соответствующего бита.

Первая часть предполагает использование на стороне отправителя специального кодера, который генерирует проверочную информацию и встраивает её в передаваемые данные. Вторая – предполагает использование декодера, который валидирует получаемые данные на стороне получателя,

восстанавливает исходное сообщение, исходя из того, какая информация была добавлена кодером, и выполняет поиск ошибок.

В основе кода Хэмминга лежит расчет избыточности кода, т.е. количества проверочной информации в сообщении. Показатель избыточности рассчитывается по формуле:

$$\frac{k}{(i + k)},$$

где k — количество проверочных бит, i — количество информационных бит.

Основное преимущество этого метода заключается в его способности исправлять ошибки, что значительно повышает надежность передачи данных. Однако использование избыточных кодов требует увеличения объема передаваемых данных и повышенных вычислительных затрат, что может быть значительным недостатком в условиях ограниченных ресурсов.

Циклический избыточный код (CRC)

Циклический избыточный код, который в специализированной литературе, определяют еще как «контрольный код», является собой в упрощенной интерпретации: «число, рассчитанное на основе передаваемого сообщения, которое добавляется к нему для проверки правильности передачи данных».

Углубясь внутрь теории, можно дать следующее определение: алгоритм CRC — это процесс, в котором применяется полиномиальное деление для вычисления контрольной суммы. Каждый бит данных рассматривается как коэффициент многочлена, и процедура вычисления сводится к определению остатка от деления этого многочлена на специальный генерирующий полином. Полученная сумма добавляется к передаваемым данным, что позволяет приемной стороне выполнить аналогичную процедуру и проверить соответствие.

В отличие от простого контроля чётности, CRC способен выявлять сложные комбинации ошибок, включая одиночные, двойные и ошибки в целых блоках битов., что позволяет определить CRC как высокоэффективный и

надежный метод обнаружения ошибок. Вместе с тем, данный метод имеет следующий недостаток: несмотря на высокую вероятность обнаружения ошибок, CRC не может исправлять найденные ошибки. Зона применения CRC довольно обширна, он применяется в сетевых протоколах – Ethernet, Bluetooth, а также в программах для сжатия данных - PKZIP и WinZIP».

CRC технически реализуется в двух различных алгоритмах: табличном и матричном. Табличный алгоритм вычисления CRC позволяет побайтно вычислять контрольную сумму, используя предвычисленные таблицы, что значительно ускоряет процесс. Матричный алгоритм также использует таблицы, но вместо этого применяет операцию умножения вектора на матрицу. Исследования показывают, что табличный алгоритм CRC превосходит матричные алгоритмы по скорости вычисления, несмотря на то, что последние требуют меньше памяти, что делает их более подходящими для систем с ограниченными ресурсами, таких как микроконтроллеры.

Сравнительный анализ методов защиты от ошибок

Для выбора оптимального метода защиты от ошибок в различных сетевых условиях важно провести сравнительный анализ основных подходов. В таблице ниже отражено сравнение рассмотренных методов по ряду ключевых параметров.

Критерий	Контроль четности	Код Хэмминга	CRC (циклический код)
Тип метода	Обнаружение ошибок	Обнаружение и исправление	Обнаружение ошибок
Выявление одиночной ошибки	Да	Да	Да
Выявление двойной ошибки	Нет	Частично	Да
Исправление ошибок	Нет	Да (1-битовая)	Нет

Вычислительная сложность	Низкая	Средняя	Средняя
Избыточность	Низкая (1 бит)	Средняя (несколько бит)	Средняя (зависит от полинома)
Применимость	Простые устройства, микроконтроллеры	Беспроводные сети, встроенные системы	Сетевые протоколы, архиваторы
Надежность выявления ошибок	Низкая	Средняя–высокая	Высокая

Анализируя эти методы, можно выделить их основные области применения. CRC (циклический избыточный код) представляет собой сбалансированный вариант, обеспечивающий высокую точность обнаружения ошибок при умеренных затратах. Этот метод широко применяется в сетевых протоколах и архиваторах данных, где важно быстро и точно выявлять ошибки.

Код Хэмминга, несмотря на свою сложность и высокие вычислительные затраты, способен не только обнаруживать, но и исправлять ошибки, что делает его эффективным средством в борьбе с частыми ошибками в условиях, когда повторная передача данных невозможна или затруднена, например, в беспроводных сетях.

Проверка на четность является самым простым и дешевым методом, однако может обнаруживать только одиночные ошибки. Благодаря своей простоте и минимальным вычислительным затратам, этот метод широко используется в системах с ограниченными ресурсами, таких как микроконтроллеры и простые сенсорные сети.

На основании проведенного анализа можно сделать следующие выводы:

1. Для задач, требующих простоты и быстроты, подходят методы с низкими вычислительными затратами, такие как проверка на четность.

2. В более критических системах, где важна надежность и возможность исправления ошибок, следует использовать избыточные коды, такие как код Хэмминга.

3. Метод CRC является универсальным решением, обеспечивающим высокую вероятность обнаружения ошибок при умеренных затратах на реализацию.

Выбор подходящего метода защиты от ошибок зависит от конкретных условий и требований эксплуатации сети. При правильном выборе метода можно значительно повысить надежность и эффективность передачи данных, что является ключевым фактором для стабильного функционирования современных информационных систем.

Практическая часть

Разрабатываемая программа

В практической части будет разработана программа, реализующая передачу данных по сети с защитой от ошибок на прикладном уровне. В её основе лежит собственный протокол, построенный поверх UDP, обеспечивающий контроль целостности с помощью CRC, фрагментацию сообщений и механизм повторной передачи при обнаружении ошибок.

Программа обеспечивает двухстороннее взаимодействие между узлами локальной сети в архитектуре клиент–сервер, при этом каждый участник может переключаться между режимами без перезапуска. Основное назначение — передача текстовых сообщений и файлов с защитой от искажений, возникающих в процессе передачи.

Ключевые возможности программы:

1. Передача файлов и сообщений
 - Поддерживается передача текстовых сообщений и файлов;
 - При превышении заданного размера фрагмента данные автоматически разбиваются.
2. Проверка целостности с помощью CRC
 - Для каждого фрагмента вычисляется контрольная сумма
 - На стороне получателя проверяется корректность фрагмента перед его принятием.
3. Поддержка механизма ARQ (Stop & Wait)
 - В случае обнаружения ошибки получатель отправляет отрицательное подтверждение (RST);
 - Отправитель повторно пересылает повреждённый фрагмент;
4. Симуляция ошибок
 - Возможность намеренного искажения фрагментов при передаче, что позволяет протестировать устойчивость протокола и эффективность защиты от ошибок.
5. Графический интерфейс (GUI)
 - Позволяет выбрать режим (клиент / сервер), IP-адрес, порты, путь к файлу, размер фрагмента и включить симуляцию ошибок;
6. Журналирование передачи

- Логи фиксируют путь, имя и размер передаваемого файла, количество фрагментов, количество ошибок и время передачи.

Ознакомление со средой разработки и техническими средствами

Разработка программного обеспечения велась с учётом требований к кроссплатформенности, доступности инструментов и гибкости настройки окружения. В качестве языка программирования был выбран Python, а для среды разработки — Visual Studio Code в связке с WSL (Windows Subsystem for Linux) и дистрибутивом Ubuntu.

Такой выбор обусловлен рядом технических и практических преимуществ. Python — это высокоуровневый язык с лаконичным синтаксисом, м и богатым набором библиотек, включая стандартные средства для работы с сетевыми соединениями (socket, select) и пользовательскими интерфейсами (tkinter). Позволяет быстро итеративно отлаживать сетевые сценарии, что особенно важно в условиях сложной логики обработки фрагментов и проверки целостности данных.

Среда Visual Studio Code, используемая в связке с WSL Ubuntu, обеспечила удобную и лёгкую настройку кода в Linux-среде непосредственно из Windows. Преимущества Linux (более предсказуемая работа с сетевыми сокетами и правами доступа) совмещены с привычной Windows-инфраструктурой. Код запускался и тестировался непосредственно через терминал WSL в окне VS Code, что позволяло отслеживать логи, отладочную информацию и сетевое поведение без переключения между средами.

Также код поддерживает запуск через стандартный терминал WSL Ubuntu, без использования графической среды. Это делает решение пригодным как для локального использования, так и для развертывания на удалённых Linux-машинах и тестирования в минимальных средах.

Комбинация Python, WSL и Visual Studio Code предоставила гибкое, надёжное и масштабируемое окружение, идеально подходящее для разработки сетевого протокола с контролем целостности и поддержкой пользовательского интерфейса.

Для реализации протокола передачи данных с защитой от ошибок использовались стандартные и широко поддерживаемые библиотеки Python, что обеспечило высокую портируемость решения и его независимость.

1. socket

Библиотека `socket` является стандартным модулем Python для работы с сетевыми соединениями. Она позволяет создавать и использовать сокеты как на уровне TCP, так и UDP. В проекте используется UDP-сокеты, что соответствует целям разработки: минимизация накладных расходов и явная реализация механизмов надёжности на прикладном уровне.

- Применение:

- создание серверного и клиентского сокета;
- отправка и приём фрагментов данных;
- управление адресами и портами.

2. select

Модуль `select` используется для реализации механизма ожидания событий на сокете с тайм-аутом. Это позволило реализовать модель ARQ Stop & Wait, при которой передатчик ждёт подтверждения (ACK) на каждый отправленный фрагмент, а при отсутствии ответа в заданное время — повторяет попытку.

- Применение:

- ожидание ответа от сервера;
- реализация тайм-аутов на чтение;
- контроль завершения сессии.

3. os и ntpath

Эти модули применяются для работы с файловой системой:

- определение абсолютных путей;
- извлечение имени файла из полного пути;
- проверка существования файла или директории;
- создание уникальных имён при конфликтах.

Особенно важны в серверной части при сохранении принятых файлов и в клиентской — при указании пути к файлу.

4. math и re

- math.ceil() используется при расчёте количества фрагментов, необходимых для передачи файла или сообщения;
- re (регулярные выражения) — для извлечения количества фрагментов из инициализационного сообщения.

5. logging

Библиотека logging позволяет вести журнал передачи данных: время начала и окончания, количество отправленных и повторно отправленных фрагментов, путь к сохранённому файлу. Это критично для отладки и последующего анализа работы системы.

6. tkinter

Модуль tkinter используется для создания графического интерфейса пользователя (GUI). Он значительно упрощает взаимодействие с приложением, особенно для непрофессиональных пользователей, позволяя запускать клиент и сервер, выбирать файлы, задавать параметры (IP, порт, размер фрагмента), а также включать симуляцию ошибок без необходимости работы в командной строке.

- Применение:

- переключение между режимами;
- настройка параметров;
- управление запуском клиента/сервера.

7. threading

Используется для разделения сетевой логики и интерфейса. Передача данных запускается в отдельном потоке, что позволяет избежать "замораживания" интерфейса при ожидании ответа от сокета и повысить отзывчивость GUI.

Архитектура программного решения

Передача данных с использованием протокола UDP реализована в 5 исходных файлах, при этом коммуникационный протокол и сам обмен данными между клиентом и сервером реализованы в следующих четырех исходных файлах:

- `client.py`
- `server.py`
- `protocol.py`
- `gui.py`

Клиент

Реализация коммуникации и поведения клиента определяется в файле `client.py`, который содержит следующее поведение клиента:

- пользовательский интерфейс
- настройка клиента
- отправка текстового сообщения
- отправка файла
- получение имени файла из пути к файлу

Сервер

Реализация связи и поведения сервера определяется в файле `server.py`, который содержит следующее поведение сервера:

- пользовательская среда
- настройка сервера
- прием данных
- запись текстового сообщения в консоль
- запись файла по заранее определенному пути с полученным именем файла в инициализационном пакете

Протокол

Реализация коммуникационного протокола и его поведение определяется в файле `protocol.py`, который содержит:

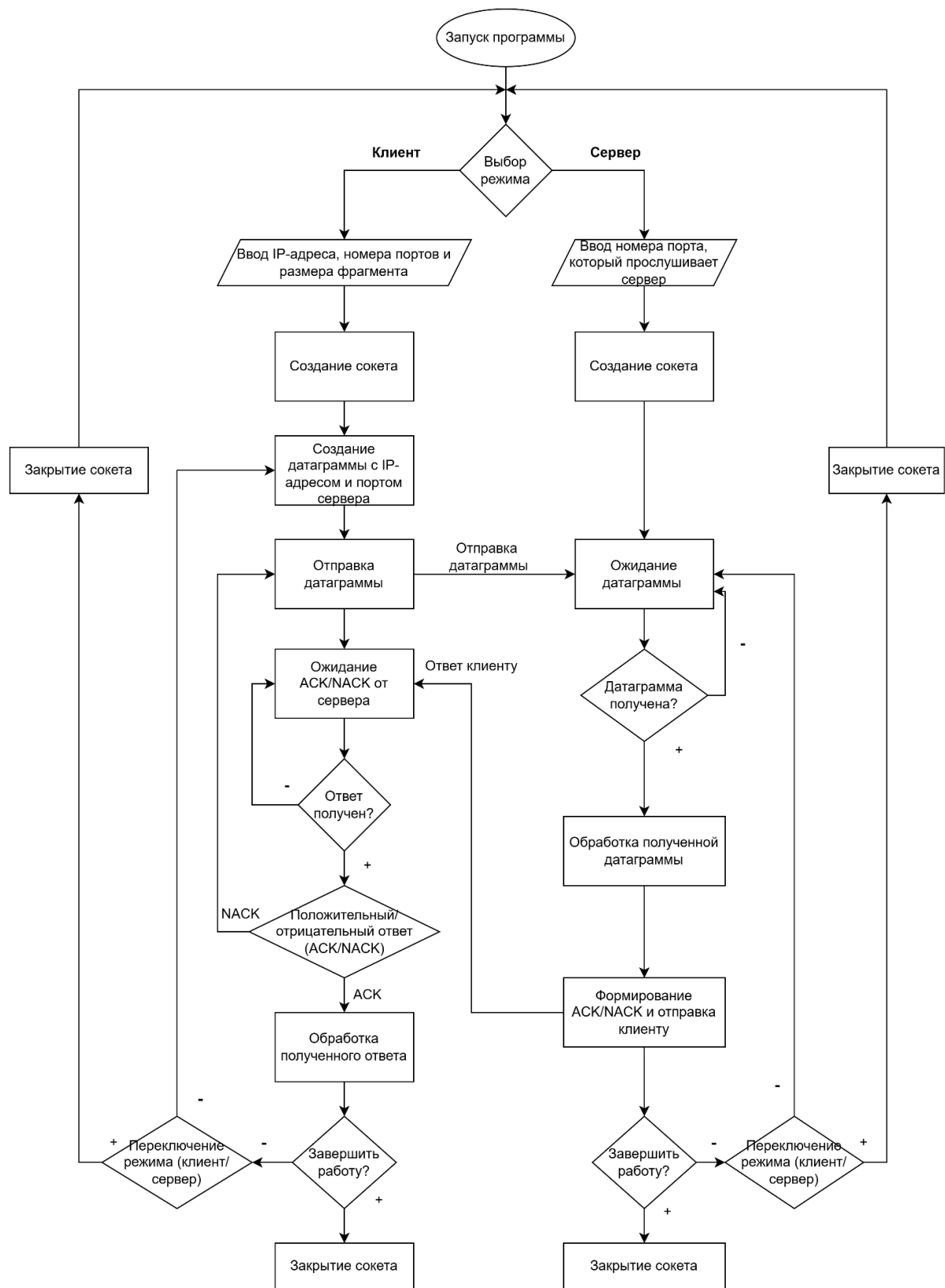
- контрольная сумма типа CRC
- создание заголовка при коммуникации
- создание заголовка при инициализации.

Графический интерфейс (GUI)

Реализация пользовательского интерфейса и логики управления передачей данных осуществляется в файле `gui.py`. Модуль обеспечивает визуальную обёртку над клиентской и серверной логикой. Реализует следующие функции:

- выбор режима работы (клиент или сервер) с автоматическим обновлением элементов интерфейса;
- ввод IP-адреса, портов, размера фрагмента и путей к файлу или директории;
- обработка нажатий кнопок, включая запуск клиента или сервера в отдельном потоке;
- отправка текстового сообщения или файла через клиентскую логику;
- отображение состояния и ошибок передачи через всплывающие окна;
- возможность включения симуляции ошибок, контролируемой пользователем через флажок;
- выбор файла или каталога через стандартные диалоговые окна.

Блок-схема приложения клиент - сервер с использованием UDP:



1. Блок-схема логики работы программы в режимах клиента и сервера

Реализация метода CRC и проверка целостности

Одним из ключевых компонентов разработанного протокола является механизм контроля целостности данных на основе циклического избыточного кода (CRC). Он используется как на стороне клиента, так и на стороне сервера для оценки корректности каждого передаваемого фрагмента. CRC обеспечивает надёжное обнаружение ошибок, возникших в процессе передачи, и служит триггером для повторной отправки при обнаружении искажения.

Передаваемые данные сначала преобразуются в двоичную строку, к которой добавляется хвост из нулей длиной на единицу меньше, чем степень полинома. Затем производится последовательная операция XOR между данным блоком и полиномом, пока не будет обработан весь поток.

В коде этот процесс реализуется следующим образом:

- Преобразование данных в двоичный вид (`format(ord(char), 'b')`);
- Пошаговое выполнение XOR с полиномом $x^3 + 1$ ('1001');
- Подсчёт суммы оставшихся битов (в виде итоговой контрольной суммы в 1 байт).

Контрольная сумма вставляется в заголовок каждого передаваемого фрагмента. На стороне приёмника выполняется та же операция, и полученная сумма сравнивается с переданной.

Контрольная сумма CRC формируется на этапе упаковки фрагмента с помощью функции `set_crc(data)`. Она добавляется в шестибайтный заголовок, занимающий один байт после поля длины, т.е. расположена в позиции №5.

Заголовок включает:

- 1 байт — тип сообщения (`MsgType`);
- 4 байта — размер фрагмента;
- 1 байт — контрольная сумма CRC;
- далее — данные фрагмента.

Каждый фрагмент, независимо от типа сообщения (текст или файл), защищён своей CRC-суммой, обеспечивающей проверку его целостности.

Проверка корректности при получении:

На стороне сервера (или клиента, в случае симметричной логики) при получении каждого фрагмента вызывается функция `check_crc(data)`, которая:

- извлекает контрольную сумму из заголовка;
- пересчитывает CRC для полученных данных;
- сравнивает оригинальную и пересчитанную сумму;
- возвращает сигнал подтверждения (ACK) или отказ (RST), в зависимости от результата.

Этот механизм напрямую связан с реализацией протокола ARQ Stop & Wait: в случае ошибки инициируется повторная передача соответствующего фрагмента. Выбранный полином позволяет сохранить баланс между вычислительной простотой и надёжностью обнаружения ошибок, что делает его оптимальным выбором для данной реализации.

Формат заголовка прикладного протокола

Для обеспечения надёжной передачи данных по протоколу UDP в рамках проекта был реализован прикладной протокол, который добавляет к каждому фрагменту специальный заголовок. Этот заголовок содержит минимально необходимую информацию для определения типа сообщения, длины данных и проверки целостности. Такая структура позволяет обеспечить отказоустойчивость передачи без избыточной нагрузки.

Формат заголовка был разработан с учётом простоты, компактности и достаточной информативности. Он имеет фиксированный размер — 6 байт, включает следующие поля:

Структура заголовка

Поле	Размер(байт)	Назначение
------	--------------	------------

Тип сообщения	1	Определяет роль фрагмента (данные, ACK, RST и т.д.)
Длина фрагмента	4	Указывает размер фрагмента, заданный пользователем
CRC	1	Контрольная сумма, вычисленная по алгоритму CRC

- Размер фрагмента указывается пользователем в интерфейсе или консоли, после чего сохраняется в поле заголовка при отправке каждого фрагмента.
- Нумерация фрагментов не используется явно — передатчик и приёмник работают в модели «один за другим», что позволяет отказаться от поля номера без потери логики передачи.

Добавление заголовка осуществляется функцией `add_header(msg_type, fragment_size, data)`:

- данные заполняются до нужной длины;
- к ним добавляется тип сообщения, длина и CRC;
- результат отправляется по сети.

На стороне приёмника из пакета извлекаются:

- тип сообщения (`data[:1]`);
- длина фрагмента (`data[1:5]`);
- CRC (`data[5:6]`);
- полезная нагрузка (`data[6:]`).

Такая структура позволяет обеспечить необходимую функциональность при минимуме байтов, что особенно важно при передаче по нестабильным или низкопроизводительным каналам.

Логика клиента

При реализации UDP-сокетов используется модуль Python `socket`, который позволяет в рамках программы создавать и работать с сетевыми сокетами. Введение модуля `socket` в программу осуществляется для обеих частей программы клиент – сервер:

```
import socket
```

Клиентская часть программы отвечает за подготовку и надёжную передачу данных на сервер. Логика клиента реализована в модуле `client.py` и отвечает за этапы: установку параметров соединения, инициализацию, разбиение на фрагменты, контроль целостности, отправку данных и обработку ответов. Поддерживается также режим симуляции ошибок, позволяющий проверить устойчивость протокола.

Установка соединения и создание сокета

Клиент создаёт сетевой сокет с помощью метода:

```
socket.socket(family=AF_INET, type=SOCK_DGRAM, proto=0, fileno=None) ,
```

где:

- `family=AF_INET` — указывает использование IPv4;
- `type=SOCK_DGRAM` — определяет, что используется протокол UDP;

Созданный сокет ведёт себя как объект и поддерживает методы для отправки и приёма данных. Передача осуществляется с помощью:

```
socket.sendto(bytes, address) ,
```

где `bytes` — упакованные данные с заголовком, а `address` — кортеж из IP-адреса и порта получателя. Источник (IP и порт клиента) присваивается автоматически операционной системой, если не задан вручную через `bind()`.

Принятие ответа от сервера выполняется через:

```
serverData, serverAddress = socket.recvfrom(bufsize) ,
```

где:

- `serverData` — полученные байты (фрагмент с заголовком),
- `serverAddress` — IP и порт отправителя (обычно не используются, так как адрес сервера известен заранее).

Инициализация передачи

Передача начинается с отправки инициализационного пакета, содержащего информацию о размере фрагмента, имени файла или типе сообщения и количестве фрагментов. Он формируется функцией `msg_initialization()` и отправляется на сервер с последующим ожиданием подтверждения. В случае отсутствия ответа или получения RST, клиент повторяет попытку.

Разбиение на фрагменты

Если передаваемый файл превышает заданный размер фрагмента, он делится на части:

$$N = \left\lceil \frac{a}{b-6} \right\rceil,$$

Где N – количество фрагментов, a – размер файла, b – размер фрагмента.

Передача реализована по принципу ARQ Stop & Wait:

- Отправляется один фрагмент;
- Клиент ожидает ACK или RST;
- Если получено ACK, передаётся следующий;
- Если получено RST или наступил тайм-аут, фрагмент отправляется повторно.

После каждой передачи клиент проверяет корректность получения. В случае ошибки повтор отправляется автоматически. По завершении всех фрагментов соединение закрывается:

```
socket.close()
```

Это освобождает ресурсы и завершает работу сокета.

Для тестирования устойчивости реализован механизм симуляции ошибок. Если пользователь в GUI отмечает соответствующий флажок, первый фрагмент намеренно искажается (например, удаляется часть полезных данных). Это приводит к ошибке CRC на стороне сервера, и клиент получает RST, после чего повторяет фрагмент уже без искажения.

Логика сервера

Серверная часть программы реализует приём, проверку и обработку входящих фрагментов от клиента. Её поведение определено в файле `server.py` и включает: запуск процесса, приём и проверку инициализационного сообщения, обработку фрагментов, сборку файла или вывод сообщения, логирование и возврат подтверждений. Также сервер реализует механизм повторного запроса фрагментов при обнаружении ошибок, в рамках протокола ARQ Stop & Wait.

Запуск и создание сокета

Сервер создаёт UDP-сокеты тем же методом, что и клиент:

```
socket.socket(family=AF_INET, type=SOCK_DGRAM)
```

Но в отличие от клиента, он привязывает сокет к порту, чтобы принимать входящие данные по заданному адресу. Это делается через:

```
socket.bind(address)
```

где `address` — кортеж из IP-адреса и порта, указанного пользователем.

После создания сокета сервер входит в бесконечный цикл, где ожидает данные от клиента и обрабатывает их, пока пользователь не завершит программу или не изменит режим работы.

Приём и обработка пакетов

Каждый входящий пакет сервер получает с помощью:

```
clientData, clientAddress = socket.recvfrom(buffer_size)
```

где:

- `clientData` — содержимое фрагмента (включая заголовок),
- `clientAddress` — кортеж из IP-адреса и порта клиента, необходимый для отправки ответа.

В отличие от клиента, сервер не знает заранее, кто отправитель, и использует `clientAddress` для ответа, делая взаимодействие универсальным.

Ответ формируется функцией `add_header(...)` с типом ACK или RST и возвращается через:

```
socket.sendto(response, clientAddress),
```

так каждый фрагмент получает обратную связь: был он принят или требует повторной отправки.

Перед основной передачей данных клиент отправляет инициализационное сообщение. После получения инициализационного сообщения от клиента сервер проверяет CRC, парсит заголовок и подготавливает буфер приёма. В случае успешной проверки отправляется ACK, иначе — RST. После этого начинается основной приём данных.

Обработка фрагментов и сборка

В зависимости от типа сообщения:

- если это текст — фрагменты выводятся в консоль;
- если это файл — они записываются в бинарный файл, имя которого извлекается из инициализационного сообщения.

Для каждого полученного фрагмента:

- 1) Вызывается `check_crc()` для проверки контрольной суммы;
- 2) Если сумма верна — данные сохраняются, и сервер отправляет ACK;
- 3) Если нет — сервер отправляет RST, инициируя повтор.

Также ведётся подсчёт полученных фрагментов и логирование, включая информацию о количестве попыток, времени и сохранённом пути.

После получения всех фрагментов (по количеству, переданному в инициализации), сервер отправляет финальное ACK, выводит сообщение об успешной передаче и завершает обработку текущего клиента.

Серверный сокет остаётся активным для новых соединений, и цикл продолжается.

Пользовательский интерфейс

Для взаимодействия пользователя с программой были реализованы два независимых интерфейса:

- консольный — предназначен для запуска и работы через терминал;
- графический — обеспечивает управление в визуальной среде, построен на библиотеке `tkinter`.

Оба интерфейса позволяют выполнять передачу сообщений и файлов, настраивать параметры соединения и включать симуляцию ошибок. Они используют одну и ту же сетевую и протокольную логику, что делает их функционально эквивалентными, но различающимися по удобству взаимодействия.

Консольный интерфейс

Клиент

Интерактивная среда пользователя клиента проходит взаимодействие с пользователем на уровне консоли. Пользователь настраивает целевой IP адрес сервера, также целевой порт сервера и максимальный размер фрагмента. Доступное консольное меню на терминале позволяет пользователю:

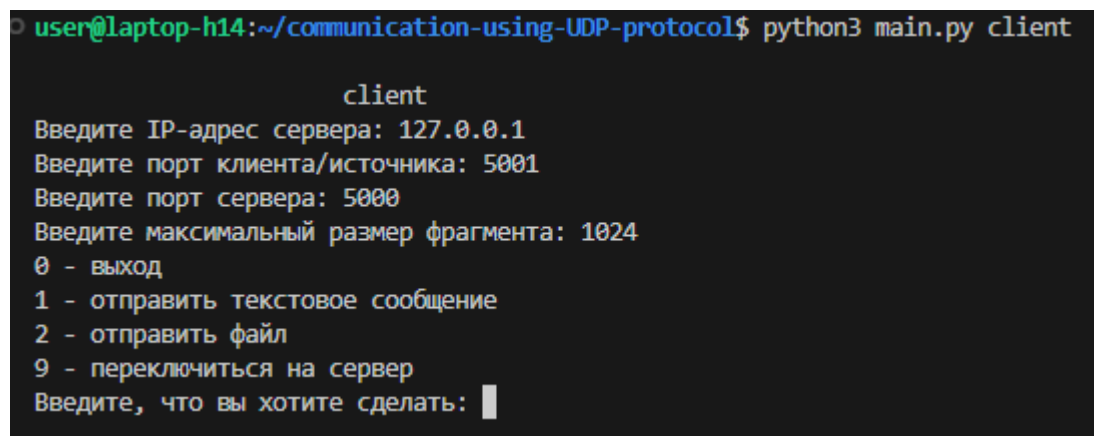
- завершить программу, введя число 0
- отправить текстовое сообщение, введя число 1
- отправить файл, введя число 2

- изменить функциональность с клиента на сервер, введя число 9

Перед отображением пользовательского меню клиента необходимо инициализировать, т.е. настроить клиента, и поэтому пользователь должен ввести по запросу программы:

- серверный, т.е. целевой, IP-адрес в формате IPv4
- исходный порт клиента в диапазоне от 1024 до 65535
- серверный, т.е. целевой, порт клиента в диапазоне от 1024 до 65535
- максимальный размер фрагмента, который не должен быть меньше 1

В случае неправильного ввода пользователя предотвращается не определенное поведение во время выполнения программы.



```
user@laptop-h14:~/communication-using-UDP-protocol$ python3 main.py client

client
Введите IP-адрес сервера: 127.0.0.1
Введите порт клиента/источника: 5001
Введите порт сервера: 5000
Введите максимальный размер фрагмента: 1024
0 - выход
1 - отправить текстовое сообщение
2 - отправить файл
9 - переключиться на сервер
Введите, что вы хотите сделать: █
```

Рис. 3 Интерфейс консольного клиента при запуске программы

Сервер

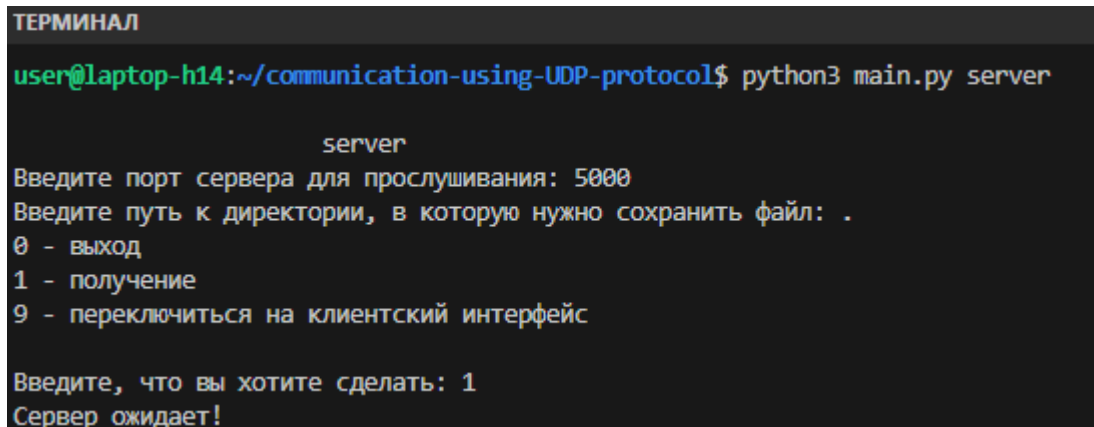
Интерактивная пользовательская среда доступна также для сервера для общения с пользователем на уровне консоли. Пользователь настраивает исходный порт сервера, на котором сервер «слушает» и путь расположения для сохранения полученных файлов. Доступное консольное меню на терминале позволяет пользователю:

- завершить программу, введя число 0
- принять данные, введя число 1
- изменить функциональность с сервера на клиента, введя число 9

Перед отображением пользовательского меню сервера необходимо инициализировать сервер, и поэтому пользователь вводит по запросу программы:

- исходный порт сервера в диапазоне от 1024 до 65535
- существующий путь к папке, где сервер будет хранить полученные файлы

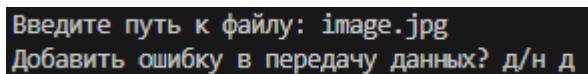
В случае неправильного ввода пользователя предотвращается не определенное поведение во время выполнения программы.



```
ТЕРМИНАЛ
user@laptop-h14:~/communication-using-UDP-protocol$ python3 main.py server
server
Введите порт сервера для прослушивания: 5000
Введите путь к директории, в которую нужно сохранить файл: .
0 - выход
1 - получение
9 - переключиться на клиентский интерфейс
Введите, что вы хотите сделать: 1
Сервер ожидает!
```

Рис. 4 Интерфейс консольного сервера при запуске программы

В консольной версии предусмотрено текстовое подтверждение успешной передачи, информации о фрагментах и времени передачи. При симуляции ошибок пользователю предлагается ввести 'д' или 'н' — в зависимости от того, нужно ли исказить данные.



```
Введите путь к файлу: image.jpg
Добавить ошибку в передачу данных? д/н д
```

Рис. 5 Симуляция ошибки при передаче файла в консольном режиме

Графический интерфейс

Для повышения удобства взаимодействия был реализован графический интерфейс на базе tkinter, описанный в модуле `gui.py`. GUI полностью повторяет возможности консольной версии, но предоставляет их в интуитивно понятной форме с кнопками, полями и флажками.

Основные возможности интерфейса:

Переключение между режимами:

- Кнопки позволяют выбрать режим: Клиент или Сервер.
- Интерфейс автоматически перестраивается в зависимости от выбранного режима, отображая только нужные поля.

Ввод параметров:

- IP-адрес сервера (для клиента);
- порт сервера (для клиента и сервера);
- порт клиента (только для клиента);
- размер фрагмента (в байтах);
- путь к файлу или директории (с выбором через диалоговое окно).

Управление передачей:

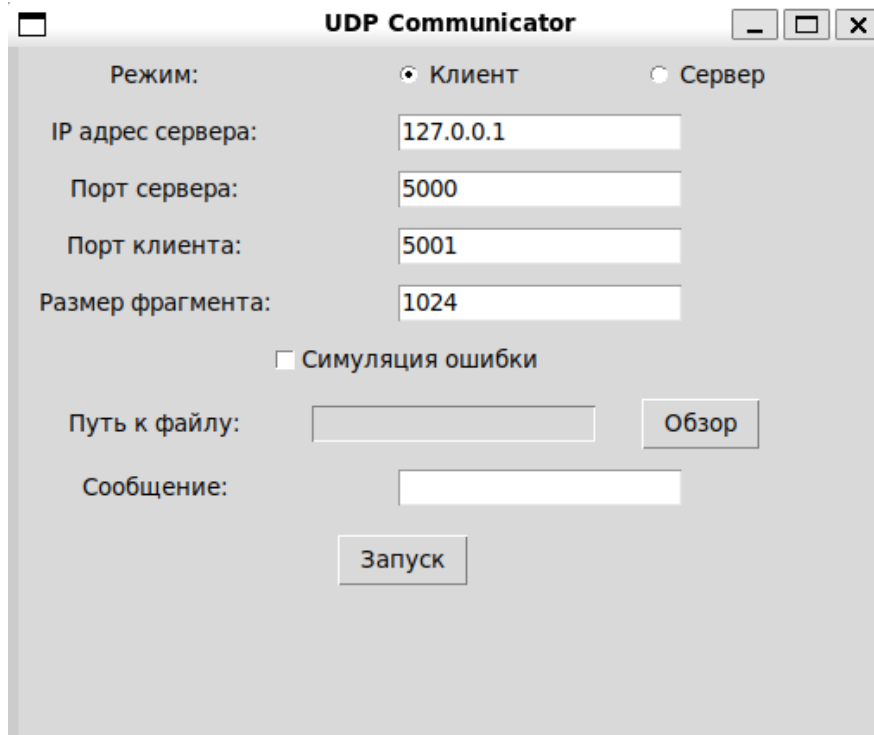
- Кнопка «Запуск» инициирует отправку или приём данных в зависимости от выбранного режима;
- При передаче сообщения достаточно ввести текст в отдельное поле.

Симуляция ошибок:

- Отдельный флажок «Симуляция ошибки» позволяет включить искажение первого фрагмента;
- Это позволяет на практике протестировать механизм ARQ и реакцию на CRC-сбой.

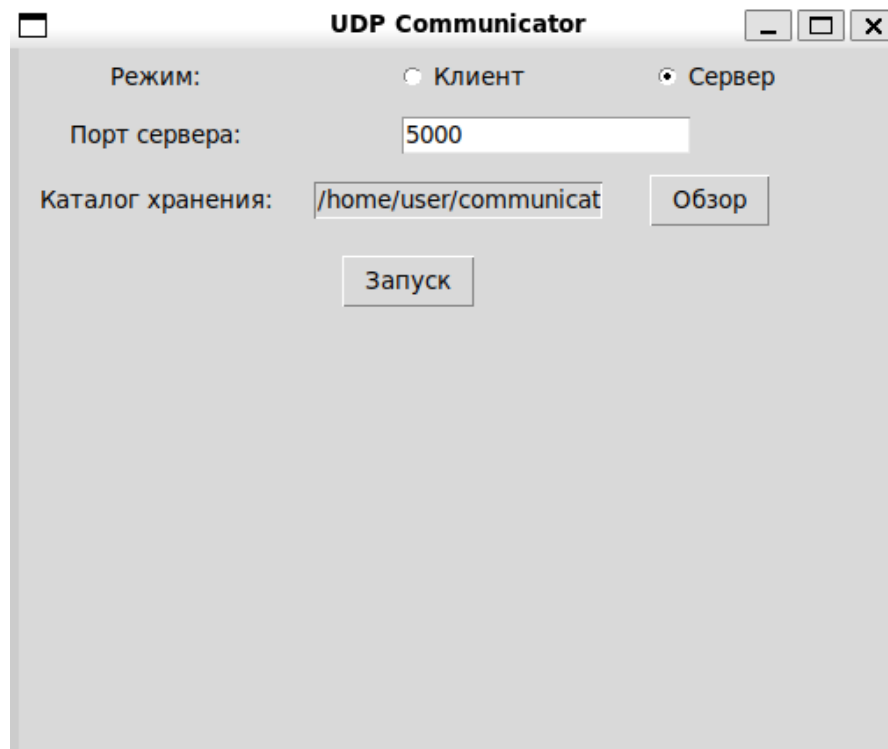
Всплывающие окна:

- При ошибках или успешной передаче данных отображаются всплывающие сообщения;
- Пользователь получает мгновенную обратную связь.



The image shows a window titled "UDP Communicator" with standard window controls (minimize, maximize, close). Inside, there are two radio buttons for "Режим:" (Mode), with "Клиент" (Client) selected. Below this are four text input fields: "IP адрес сервера:" (Server IP address) containing "127.0.0.1", "Порт сервера:" (Server port) containing "5000", "Порт клиента:" (Client port) containing "5001", and "Размер фрагмента:" (Fragment size) containing "1024". There is a checkbox for "Симуляция ошибки" (Error simulation) which is unchecked. Below these are two more input fields: "Путь к файлу:" (File path) and "Сообщение:" (Message). To the right of the "Путь к файлу:" field is a button labeled "Обзор" (Browse). Below the "Сообщение:" field is a button labeled "Запуск" (Start).

Рис. 6 Графический интерфейс клиента



The image shows a window titled "UDP Communicator" with standard window controls (minimize, maximize, close). Inside, there are two radio buttons for "Режим:" (Mode), with "Сервер" (Server) selected. Below this are two text input fields: "Порт сервера:" (Server port) containing "5000" and "Каталог хранения:" (Storage directory) containing "/home/user/communicat". To the right of the "Каталог хранения:" field is a button labeled "Обзор" (Browse). Below these fields is a button labeled "Запуск" (Start).

Рис. 7 Графический интерфейс сервера

Наличие двух интерфейсов делает программу универсальной. Консольная версия подходит для быстрой работы в терминале и тестирования с использованием логов, тогда как графический интерфейс облегчает работу конечному пользователю, скрывая детали реализации и предоставляя простой способ управления процессом передачи.

Методика тестирования

Для проверки надёжности и устойчивости реализованного протокола передачи данных был разработан набор тестов, имитирующих реальные условия работы в нестабильной сетевой среде. Особое внимание было уделено способности системы обнаруживать и обрабатывать ошибки, возникающие в процессе передачи фрагментов. Тестирование включало как автоматические реакции протокола на сбои, так и анализ сетевого трафика с помощью специализированного инструментария.

Одной из особенностей реализованной программы является возможность намеренно искажать передаваемые данные, чтобы протестировать механизм контроля целостности и повторной отправки.

- В GUI-интерфейсе реализован флажок «Симуляция ошибки» — при его активации первый фрагмент передаётся в искажённом виде (например, с обрезанными байтами);
- В консольной версии пользователь вводит 'д' (да) или 'н' (нет) в ответ на вопрос: «Внести ошибку при передаче?»;
- После отправки сервер выявляет несоответствие CRC, отправляет RST, а клиент повторяет фрагмент уже в корректной форме.

Результаты таких тестов отображаются в логах, а также — визуально в интерфейсе (сообщения об ошибке и успешной повторной передаче).

В ходе тестирования лог-файлы (`transfer_YYYYMMDD_HHMM.log`) автоматически создавались при запуске клиента и сервера. В логах фиксировались:

- имя отправленного файла;
- общее количество фрагментов;

- количество повторно отправленных фрагментов;
- общее время передачи (в секундах);
- путь сохранённого файла.

```

2025-06-09 09:31:31,829 [INFO] Получен фрагмент 1267/1269
2025-06-09 09:31:31,981 [INFO] Получен фрагмент 1268/1269
2025-06-09 09:31:32,074 [INFO] Получен фрагмент 1269/1269
2025-06-09 09:31:35,079 [INFO] Файл сохранен как: /home/user/communication-using-UDP-protocol/image(1).jpg
2025-06-09 09:31:35,079 [INFO] Получен файл: image.jpg
2025-06-09 09:31:35,080 [INFO] Ожидаемые фрагменты: 1269
2025-06-09 09:31:35,080 [INFO] Отправленный файл: /home/user/projects/file-transfer/network-error-correction-main/ima
2025-06-09 09:31:35,080 [INFO] Всего фрагментов: 1269, Повторные передачи (NACK): 0
2025-06-09 09:31:35,081 [INFO] Время передачи: 203.72 секунд

```

Рис. 8 Пример логов успешной передачи файла

Методика тестирования включает в себя проверку связи на одном конечном устройстве в одной локальной сети, так называемом loopback, с захваченным сетевым трафиком с помощью программы Wireshark, отправляя текстовые сообщения и выбранные файлы.

В ходе тестирования функциональности задания были отправлены по локальной сети текстовые сообщения и следующие файлы:

- Исходные Python файлы задания
- PDF файлы
- Изображения в формате PNG

Связь была захвачена программой для анализа сетевого трафика - Wireshark, а также была проверена связь с симуляцией ошибки.

Отправка файла без ошибок:

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help						
udp.port == 5000 udp.port == 5001						
No.	Time	Source	Destination	Protocol	Length	Info
4	0.003687477	127.0.0.1	127.0.0.1	UDP	48	5000 → 5001 Len=6
6	0.011697723	127.0.0.1	127.0.0.1	UDP	1072	5001 → 5000 Len=1030
7	0.016680474	127.0.0.1	127.0.0.1	UDP	48	5000 → 5001 Len=6
8	0.021887407	127.0.0.1	127.0.0.1	UDP	1072	5001 → 5000 Len=1030
9	0.027956689	127.0.0.1	127.0.0.1	UDP	48	5000 → 5001 Len=6
10	0.035728607	127.0.0.1	127.0.0.1	UDP	1072	5001 → 5000 Len=1030
11	0.042012997	127.0.0.1	127.0.0.1	UDP	48	5000 → 5001 Len=6
12	0.048863157	127.0.0.1	127.0.0.1	UDP	1072	5001 → 5000 Len=1030
13	0.055557394	127.0.0.1	127.0.0.1	UDP	48	5000 → 5001 Len=6
15	0.062251410	127.0.0.1	127.0.0.1	UDP	1072	5001 → 5000 Len=1030
16	0.068626987	127.0.0.1	127.0.0.1	UDP	48	5000 → 5001 Len=6
17	0.074906236	127.0.0.1	127.0.0.1	UDP	1072	5001 → 5000 Len=1030
18	0.081290341	127.0.0.1	127.0.0.1	UDP	48	5000 → 5001 Len=6
19	0.088706625	127.0.0.1	127.0.0.1	UDP	1072	5001 → 5000 Len=1030
20	0.095540714	127.0.0.1	127.0.0.1	UDP	48	5000 → 5001 Len=6
21	0.101812320	127.0.0.1	127.0.0.1	UDP	1072	5001 → 5000 Len=1030
22	0.108070168	127.0.0.1	127.0.0.1	UDP	48	5000 → 5001 Len=6
23	0.113918717	127.0.0.1	127.0.0.1	UDP	1072	5001 → 5000 Len=1030
24	0.119635740	127.0.0.1	127.0.0.1	UDP	48	5000 → 5001 Len=6
25	0.126183653	127.0.0.1	127.0.0.1	UDP	1072	5001 → 5000 Len=1030
26	0.131410734	127.0.0.1	127.0.0.1	UDP	48	5000 → 5001 Len=6
27	0.139380893	127.0.0.1	127.0.0.1	UDP	1072	5001 → 5000 Len=1030
28	0.144908335	127.0.0.1	127.0.0.1	UDP	48	5000 → 5001 Len=6
29	0.150947330	127.0.0.1	127.0.0.1	UDP	1072	5001 → 5000 Len=1030
30	0.156669166	127.0.0.1	127.0.0.1	UDP	48	5000 → 5001 Len=6
31	0.161744953	127.0.0.1	127.0.0.1	UDP	1072	5001 → 5000 Len=1030
32	0.168375346	127.0.0.1	127.0.0.1	UDP	48	5000 → 5001 Len=6
33	0.174394432	127.0.0.1	127.0.0.1	UDP	1072	5001 → 5000 Len=1030
34	0.180184076	127.0.0.1	127.0.0.1	UDP	48	5000 → 5001 Len=6
35	0.185790241	127.0.0.1	127.0.0.1	UDP	1072	5001 → 5000 Len=1030
36	0.192176060	127.0.0.1	127.0.0.1	UDP	48	5000 → 5001 Len=6
▶ Frame 4: 48 bytes on wire (384 bits), 48 bytes captured (384 bits) on interface lo, id 0 ▶ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00) ▶ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1 ▶ User Datagram Protocol, Src Port: 5000, Dst Port: 5001 ▶ Data (6 bytes)						

Рис. 9 Анализ UDP-трафика в программе Wireshark

Отправка текстового файла с ошибкой:

3949	580.508061024	127.0.0.1	127.0.0.1	UDP	55 5001 → 5000	Len=13
3950	580.508432698	127.0.0.1	127.0.0.1	UDP	48 5000 → 5001	Len=6
3951	580.510360671	127.0.0.1	127.0.0.1	UDP	61 5001 → 5000	Len=19
3952	580.510895453	127.0.0.1	127.0.0.1	UDP	48 5000 → 5001	Len=6
3953	580.511839823	127.0.0.1	127.0.0.1	UDP	75 5001 → 5000	Len=33
3954	580.512783625	127.0.0.1	127.0.0.1	UDP	48 5000 → 5001	Len=6
3997	583.516907583	127.0.0.1	127.0.0.1	UDP	43 5000 → 5001	Len=1[Malformed Packet]

▶ Frame 3952: 48 bytes on wire (384 bits), 48 bytes captured (384 bits) on interface lo, id 0
 ▼ Ethernet II, Src: 00:00:00 00:00:00 (00:00:00:00:00:00), Dst: 00:00:00 00:00:00 (00:00:00:00:00:00)
 ▶ Destination: 00:00:00 00:00:00 (00:00:00:00:00:00)
 ▶ Source: 00:00:00 00:00:00 (00:00:00:00:00:00)
 Type: IPv4 (0x0800)
 ▼ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
 0100 = Version: 4
 0101 = Header Length: 20 bytes (5)
 ▶ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
 Total Length: 34
 Identification: 0xaa49 (43593)
 010. = Flags: 0x2, Don't fragment
 ...0 0000 0000 0000 = Fragment Offset: 0
 Time to Live: 64
 Protocol: UDP (17)
 Header Checksum: 0x927f [validation disabled]
 [Header checksum status: Unverified]
 Source Address: 127.0.0.1
 Destination Address: 127.0.0.1
 ▶ User Datagram Protocol, Src Port: 5000, Dst Port: 5001
 ▼ Data (6 bytes)
 Data: 333130343333
 [Length: 6]

Рис. 10 Структура UDP-пакета с заголовком пользовательского протокола при добавлении ошибки при передаче

0000	00 00 00 00 00 00 00 00	00 00 00 00 08 00 45 00E.
0010	00 22 aa 49 40 00 40 11	92 7f 7f 00 00 01 7f 00	..".I@.@.....
0020	00 01 13 88 13 89 00 0e	fe 21 33 31 30 34 33 33!310433

Рис. 11 Представление UDP-пакета в шестнадцатеричном виде

На скриншоте видно, что сервер обнаружил ошибку и сгенерировал отрицательное подтверждение (RST, MessageType = 3), соответствующее повторному запросу фрагмента.

Данные: 33 31 30 34 33 33

ASCII: 3 1 0 4 3 3

‘3’- MessageType

‘1043’- размер фрагмента,

‘3’ Контрольная сумма (в виде ASCII)

Сервер, получив повреждённый фрагмент, обнаружил рассогласование контрольной суммы CRC и отправил отрицательное подтверждение (NACK).

Отображение обработки этого файла в консоли:

```
negative acknowledgment msg. Ошибка обработки сообщения  
  
Полученные фрагменты: 1   учтенные фрагменты: 1  
  
Передача прошла успешно, файл находится /home/user/communication-using-UDP-protocol/ex(1).txt  
  
Сообщение получено!  
Время: 3.007850408554077  
Сохранено в /home/user/communication-using-UDP-protocol/ex.txt  
Отправлено фрагментов: 1   всего фрагментов: 1  
Отправлено фрагментов: 2   NACK фрагментов: 1
```

Рис. 12 Вывод программы при обработке ошибки

Лог передачи:

```
1  
2   2025-06-09 10:28:40,702 [INFO] Ожидаемые фрагменты: 1  
3   2025-06-09 10:28:40,703 [INFO] Отправленный файл: /home/user/communication-using-UDP-protocol/ex.txt  
4   2025-06-09 10:28:40,703 [INFO] Всего фрагментов: 2, Повторные передачи (NACK): 1  
5   2025-06-09 10:28:40,703 [INFO] Время передачи: 3.01 секунд  
6   2025-06-09 10:28:40,703 [INFO] Сохранено как: /home/user/communication-using-UDP-protocol/ex.txt  
7
```

Рис. 13 Логгирование передачи при ошибке

Предусмотрена обработка ошибок ввода для графического и консольного интерфейса:

```
user@laptop-h14:~/communication-using-UDP-protocol$ python3 main.py server  
  
server  
Введите порт сервера для прослушивания: 1  
Этот порт зарезервирован. Введите другой.  
Введите порт сервера для прослушивания: █
```

Рис. 14 Обработка ошибки при вводе пользователем номера занятого порта

```
Введите путь к директории, в которую нужно сохранить файл: dd  
ERROR 01: Путь dd не существует!  
Введите путь к директории, в которую нужно сохранить файл: █
```

Рис. 15 Ввод пользователем некорректного пути

```
Введите путь к директории, в которую нужно сохранить файл: .
0 - выход
1 - получение
9 - переключиться на клиентский интерфейс

Введите, что вы хотите сделать: 3

Некорректный ввод! Введите еще раз.

0 - выход
1 - получение
9 - переключиться на клиентский интерфейс

Введите, что вы хотите сделать: █
```

Рис. 16 Ошибка ввода при взаимодействии с пользовательским меню для сервера

```
Введите, что вы хотите сделать: 1
Сервер ожидает!
Timeout
0 - выход
1 - получение
9 - переключиться на клиентский интерфейс

Введите, что вы хотите сделать: █
```

Рис. 17 Время прослушивания сервера ограничено

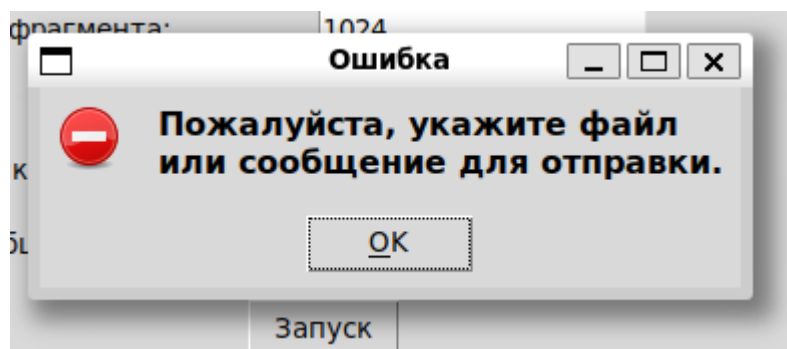


Рис. 18 Ошибка в графическом интерфейсе при отсутствии ввода данных для передачи

Заключение

В ходе выполнения курсовой работы была разработана прикладная система надёжной передачи данных с использованием пользовательского протокола на базе UDP. Разработка, реализованная на языке Python и дополненная графическим интерфейсом на библиотеке Tkinter, продемонстрировала эффективность подхода: передаваемые данные контролируются на целостность, а ошибки устраняются автоматически посредством механизма повторной отправки.

На первом этапе был проведён теоретический анализ современных методов защиты от ошибок: контроль чётности, код Хэмминга и CRC. По результатам сравнительного анализа для реализации был выбран метод циклического избыточного кода (CRC) в сочетании с протоколом ARQ Stop & Wait. Такое решение позволило достичь высокого уровня надёжности при минимальных накладных расходах и сохранить гибкость UDP.

Разработка включала проектирование структуры заголовка, реализацию вычисления контрольной суммы, организацию фрагментации данных и механизм подтверждений. Особое внимание было уделено упрощению протокола без потери его функциональности. Заголовок был минимизирован до 6 байт, что позволило сохранить пропускную способность и упростить обработку.

Функциональная часть проекта реализована в четырёх модулях: `client.py`, `server.py`, `protocol.py` и `gui.py`. Клиентская и серверная стороны могут динамически переключаться между режимами. Перед отправкой выполняется инициализация передачи, после чего данные передаются пофрагментно с контролем целостности. В случае обнаружения ошибки осуществляется повторная отправка конкретного фрагмента.

Графический интерфейс, построенный на Tkinter, обеспечивает интуитивно понятное управление: выбор режима, IP-адресов, портов, файлов и активацию симуляции ошибок. Встроенная система логирования позволяет фиксировать все этапы передачи, включая количество фрагментов, повторы и итоговое время передачи. Также предусмотрена возможность вручную исказить данные для проверки реакции протокола.

Тестирование проводилось как вручную, так и с использованием Wireshark для анализа UDP-трафика. Результаты подтвердили устойчивость работы, корректную реализацию механизма CRC и надёжность схемы ARQ.

Повреждённые фрагменты успешно обнаруживались и переотправлялись, что доказывает работоспособность протокола.

Практическая значимость работы заключается в возможности использования программы для образовательных целей: демонстрации принципов контроля целостности, надёжной передачи и проектирования прикладных протоколов. Решение также может применяться как основа для разработки отказоустойчивых приложений передачи данных в нестабильных сетях, встраиваемых системах и системах с ограниченными ресурсами.

Проект имеет перспективы развития: внедрение поддержки других кодов (например, Рида-Соломона), реализация параллельной передачи нескольких фрагментов, а также интеграция с внешними средствами визуализации для анализа статистики передачи. Разработанная программа может быть расширена для применения в лабораторных работах, практикумах и исследовательских проектах.

Важно отметить, что разработка велась исключительно в учебных целях. Все механизмы передачи и обработки данных выполняются в пределах локальной сети, без нарушения принципов безопасности и конфиденциальности. Программа не содержит функций шифрования и не предназначена для передачи чувствительной информации во внешние среды без дополнительных мер защиты.

Список литературы

1. **Таненбаум, Э. С., Уэзеролл, Д.** — "Компьютерные сети". — СПб.: Питер, 2012. — 960 с. (Фундаментальный источник по архитектуре сетей, видам кодирования и моделям ARQ)
2. **Сидоренко Д.А., Федотов Е.А.** — "Анализ способов защиты от ошибок передаваемой информации по сети" // Вестник науки №12 (81), 2024.
3. **Вахрушев, А.** — "Передача файлов по UDP и защита от ошибок: реализация на Python" // Habr, 2023. <https://habr.com/ru/articles/> (Практическое руководство по реализации пользовательского протокола передачи)
4. **RFC 768 — User Datagram Protocol (UDP).** — <https://www.rfc-editor.org/rfc/rfc768.html> (Официальная спецификация UDP)
5. **RFC 1662 — PPP in HDLC-like Framing** (описание CRC-алгоритма) — <https://www.rfc-editor.org/rfc/rfc1662> (Формальное описание алгоритма CRC и принципов контроля ошибок)
6. **Хабр — "Коды Хэмминга: теория и практика"** // <https://habr.com/ru/articles/140611/> (Доступное объяснение механизма кодов Хэмминга с примерами)
7. **Документация Python: модуль socket** — <https://docs.python.org/3/library/socket.html> (Официальная документация по работе с сетевыми сокетами в Python)
8. **Wireshark User Guide** — https://www.wireshark.org/docs/wsug_html_chunked/ (Официальное руководство по анализу сетевого трафика)

Приложение

protocol.py

```
import enum
import re

HEADER_SIZE = 6
DEFAULT_BUFF = 4096
DEFAULT_FRAGMENT_LEN = 4
FRAGMENT_MAX = 1466      # max_fragment = данные(1500) - UDP заголовок(8) - IP
                           # заголовок(20) - новый заголовок(6) = 1466
FRAGMENT_MIN = 1         # min_fragment = данные(46) - UDP заголовок(8) - IP
                           # заголовок(20) - новый заголовок(6) = 12
CRC_KEY = '1001'         #  $x^3 + 1$ 

class MsgType(enum.Enum):
    """ Сигнальные сообщения протокола в виде перечисления констант. """

    SET = '0'             # константа для заголовка при инициализации передачи файла
    PSH = '1'             # константа для заголовка при отправке данных файла
    ACK = '2'             # константа для заголовка при положительном ответе
    RST = '3'             # константа для заголовка при отрицательном ответе

    SET_MSG = '8'         # константа для заголовка при инициализации передачи сообщения

class MsgReply(enum.Enum):
    """ Перечисление констант для взаимодействия клиент-сервер """

    SET = 'Успешная инициализация передачи'
    ACK = 'Сообщение получено!'
    RST = 'Сообщение повреждено'
    KAP = 'Подключено'

def zero_fill(data):
    """ Заполнение фрагмента нулями, если он меньше длины по умолчанию(4).

    data: Данные, которые нужно заполнить нулями. """

    if DEFAULT_FRAGMENT_LEN == len(data):
        return data
    count = DEFAULT_FRAGMENT_LEN - len(data) - 1
    new_data = b'0'
    while count > 0:
```

```

        new_data += b'0'
        count -= 1
    new_data += data
    return new_data

def xor(a, b):
    """ Логический метод XOR для CRC. Возвращает строку битов.
    a XOR b

    a: Номер бита для XOR.
    b: Номер бита для XOR. """

    result = [] # список результатов
    for i in range(1, len(b)): # проходим по всем битам
        if a[i] == b[i]: # если одинаковые - XOR
            result.append('0')
            continue
        result.append('1') # если разные - XOR 1
    return ''.join(result)

def sum_checksum(checksum):
    """ Подсчитывает биты заданной контрольной суммы. Возвращает строку.

    checksum: Вычисляется контрольная сумма для суммирования всех битов. """

    sum_digit = 0
    for char in checksum:
        if char.isdigit():
            sum_digit += int(char)
    return str(sum_digit)

def set_crc(data):
    """ Получает контрольную сумму на основе CRC. Возвращает сводку всех битов
    контрольной суммы в виде строки.

    data: Данные для CRC. """

    crc_key_len = len(CRC_KEY)
    bin_data = (''.join(format(ord(char), 'b') for char in str(data))) + ('0' *
(crc_key_len - 1))
    checksum = bin_data[:crc_key_len]
    while crc_key_len < len(bin_data):

```

```

        if checksum[0] == '1':
            checksum = xor(CRC_KEY, checksum) + bin_data[crc_key_len]
        else:
            checksum = xor('0' * crc_key_len, checksum) + bin_data[crc_key_len]
        crc_key_len += 1
    if checksum[0] == '1':
        checksum = xor(CRC_KEY, checksum)
    else:
        checksum = xor('0' * crc_key_len, checksum)
    return sum_checksum(checksum)

def check_crc(data):
    """ Проверяет, была ли передана полученная информация корректно. Возвращает
    положительное подтверждение, если полученные данные
    корректны, в противном случае возвращает отрицательное подтверждение.

    data: Данные содержат заголовок протокола с информацией о первоначально
    вычисленной контрольной сумме. Вычисляется контрольная сумма для
    полученных данных и сравнивается с оригинальной контрольной суммой. """

    if data[5:6].decode('utf-8') != set_crc(data[6:]):
        return MsgType.RST
    return MsgType.ACK

def get_fragment_size(data):
    """ Получение размера фрагмента из заголовка протокола. Возвращает размер
    фрагмента в виде int.

    data: содержит заголовок протокола с информацией о размере фрагмента, введенном
    пользователем. """
    return int(data[1:5].decode('utf-8'))

def get_file_name(data, fragment_count):
    """ Получение имени файла из заголовка протокола. Возвращает имя файла в виде
    строки.

    data: содержит данные из инициализации передачи с информацией о имени файла,
    введенном пользователем. """
    return data[6:-len(fragment_count)].decode('utf-8')

def get_data(data):

```

```

    """ Получение данных без заголовка протокола. Возвращает полученные данные в
    виде байтов.
    """
    return data[6:]

def get_fragment_count(data, msg_type=None):
    """ Получение количества фрагментов из данных. Возвращает количество фрагментов
    в виде строки.
    Для текстовой передачи сообщения просто возвращает данные с индекса 7(6-битный
    заголовок + 1-битные данные).
    Для передачи файлов возвращает последнее число из данных, потому что данные
    также содержат имя файла, возможно, данные могут быть
    name1.pdf720, где последнее число 720 - это количество фрагментов.

    data: полученные данные с заголовком протокола. """

    if msg_type == MessageType.SET_MSG.value:
        return data.decode('utf-8')[7:]
    return re.compile(r'\d+').findall(data.decode('utf-8'))[-1:][0] # получение
    последнего числа из имени файла

def get_msg_type(data):
    """ Получение типа сигнального сообщения.

    data: полученные данные с заголовком протокола. """
    return data[:1]

def add_header(msg_type, fragment_size, data):
    """ Добавление заголовка протокола к данным. Возвращает новые данные с
    заголовком протокола в виде байтов.

    msg_type: Тип сообщения в виде MessageType.
    fragment_size: размер фрагмента данных. Введен пользователем.
    data: полученные данные без заголовка протокола. """

    fragment_size_bytes = zero_fill(bytes(str(fragment_size), 'utf-8'))
    new_data = bytes(msg_type.value, 'utf-8') + fragment_size_bytes
    checksum = set_crc(data)
    new_data += bytes(checksum, 'utf-8') + data
    return new_data

def msg_initialization(fragment_size, data):

```

```

""" Добавляет заголовок протокола к данным для инициализации.

fragment_size: размер фрагмента данных. Введен пользователем.
data: полученные данные без заголовка протокола. """

new_data = bytes(MsgType.SET.value, 'utf-8') if data[:1] !=
bytes(MsgType.SET_MSG.value, 'utf-8') else \
    bytes(MsgType.SET_MSG.value, 'utf-8') # тип
сообщения
fragment_size_bytes = zero_fill(bytes(str(fragment_size), 'utf-8')) #
заполнение нулями до 4 байт
checksum = set_crc(data) #
получение контрольной суммы
new_data += fragment_size_bytes + bytes(checksum, 'utf-8') #
добавление размера фрагмента и контрольной суммы
new_data += data #
добавление данных
return new_data #
возвращение заголовка + данных как новых данных

```

client.py

```

import server
import protocol
import select
import socket
import sys
import os
import ntpath
import math
import time
import logging
from datetime import datetime
import matplotlib.pyplot as plt

# Настройка логгирования
log_filename = f"transfer_{datetime.now().strftime('%Y%m%d_%H%M%S')}.log"
logging.basicConfig(
    filename=log_filename,
    filemode='w',
    level=logging.INFO,
    format='%(asctime)s [%(levelname)s] %(message)s'
)

```

```

)
logger = logging.getLogger()

def get_file_name(file_path):
    """ Получает имя файла из пути к файлу. Возвращает имя файла.

    file_path: Заданный пользователем путь к передаваемому файлу. """
    return ntpath.split(file_path.decode('utf-8'))[1]

def initialization(client_socket, server_ip, server_port, fragment_size, data):
    """ Инициализация передачи текста или файла.

    client_socket: Клиентский сокет содержит адрес источника и метод sendto.
    server_ip: Одна часть целевого адреса в сокете
    server_port: Вторая часть целевого адреса в сокете
    fragment_size: Максимальный размер одного фрагмента, заданный пользователем
    data: данные для передачи текста содержат один байт, указывающий протоколу
        на добавление типа msg в качестве текстовой передачи.
        Данные для передачи файла содержат имя файла, который будет создан на
сервере.

        Остальное - количество фрагментов как для передачи текста, так и для
передачи файла. """
    try:
        while True:
            client_socket.sendto( # инициализация передачи данных
                                protocol.msg_initialization(fragment_size, data), #
инициализационный msg
                                (server_ip, server_port)) # адрес сервера
            ready = select.select([client_socket], [], [], 5)
            if ready[0]:
                new_data, server_address = client_socket.recvfrom(fragment_size)
            else:
                print('Соединение не установлено')
                return 0

            if new_data[:1].decode('utf-8') == protocol.MsgType.ACK.value:
                break
            print(protocol.MsgReply.SET.value)

    except ConnectionResetError:
        print('Соединение потеряно. Включите сервер.')
    return 1

def send_file(server_ip, client_socket, server_port, fragment_size, file_path):

```

```

""" Передача файла с логированием. """

user_input_mistake = globals().get("user_input_mistake", "n")
# user_input_mistake = input('Добавить ошибку в передачу данных? д/н ')

if user_input_mistake.lower() not in ['д', 'н']:
    print('Неверный ввод!')
    return

file_name = bytes(get_file_name(file_path), 'utf-8')
file_size = os.path.getsize(file_path)
num_of_fragment = math.ceil(file_size / (fragment_size - protocol.HEADER_SIZE))

nack_fragment, all_fragment = 0, 0
status_log = []

try:
    if initialization(client_socket, server_ip, server_port, fragment_size,
                      file_name + bytes(str(num_of_fragment), 'utf-8')) == 0:
        return

    start_time = time.time()
    with open(file_path, 'rb') as file:
        data = file.read(fragment_size - protocol.HEADER_SIZE)
        fragment_count = 0

        while data:
            new_data = protocol.add_header(protocol.MsgType.PSH, fragment_size,
data)

            if user_input_mistake == 'д':
                header = new_data[:6]
                new_data = header + new_data[20:]
                user_input_mistake = 'н'

            if client_socket.sendto(new_data, (server_ip, server_port)):
                reply, _ = client_socket.recvfrom(fragment_size)

                if reply[:1].decode('utf-8') != protocol.MsgType.ACK.value:
                    print('negative acknowledgment msg. Ошибка обработки
сообщения')

                    nack_fragment += 1
                    all_fragment += 1
                    status_log.append(0) # Повторная передача
                else:
                    data = file.read(fragment_size - protocol.HEADER_SIZE)
                    fragment_count += 1

```



```

        all_fragment += 1
        status_log.append(1) # Успешная передача

    ready = select.select([client_socket], [], [], 5)
    if ready[0]:
        _ = client_socket.recvfrom(fragment_size)
        end_time = time.time()
        print(protocol.MsgReply.ACK.value)
        print('Время:', end_time - start_time)
        print('Сохранено в', os.path.abspath(file_path.decode('utf-8')))
        print('Отправлено фрагментов:', fragment_count, ' всего фрагментов:',
num_of_fragment)
        print('Отправлено фрагментов:', all_fragment, ' NACK фрагментов:',
nack_fragment)

        # Логгирование
        logger.info(f"Отправленный файл: {file_path.decode('utf-8')}")
        logger.info(f"Всего фрагментов: {all_fragment}, Повторные передачи
(NACK): {nack_fragment}")
        logger.info(f"Время передачи: {end_time - start_time:.2f} секунд")
        logger.info(f"Сохранено как: {os.path.abspath(file_path.decode('utf-
8'))}")

    else:
        print('Соединение не установлено')

except ConnectionResetError:
    print('Соединение потеряно. Включите сервер.')

def send_message(server_ip, client_socket, server_port, fragment_size, message):
    """ Передача текстовых сообщений.

    client_socket: Клиентский сокет содержит адрес источника и метод sendto.
    server_ip: Одна часть целевого адреса в сокете
    server_port: Вторая часть целевого адреса в сокете
    fragment_size: Максимальный размер одного фрагмента, заданный пользователем
    Message: Данные, которые необходимо передать. """

    fragment_count = 0
    num_of_fragment = math.ceil(len(message.decode('utf-8')) / fragment_size)

    nack_fragment, all_fragment = 0, 0
    try:
        if initialization(client_socket, server_ip, server_port, fragment_size +
protocol.HEADER_SIZE,

```

```

        bytes(protocol.MsgType.SET_MSG.value, 'utf-8') +
bytes(str(num_of_fragment), 'utf-8')) == 0:
    return
    lenght = math.ceil(len(message) / fragment_size)
    index1, index2 = 0, fragment_size

    while lenght > 0:
        data = protocol.add_header(protocol.MsgType.PSH, fragment_size,
message[index1:index2])
        client_socket.sendto(data, (server_ip, server_port))
        ready = select.select([client_socket], [], [], 5)
        if ready[0]:
            reply, server_address = client_socket.recvfrom(fragment_size +
protocol.HEADER_SIZE)
            if reply[:1].decode('utf-8') != protocol.MsgType.ACK.value: #
проверка на ACK
                print('negative acknowledgment msg. Ошибка обработки
сообщения')
                nack_fragment += 1
                all_fragment += 1
            else:
                index1 += fragment_size
                index2 += fragment_size
                lenght -= 1
                fragment_count += 1
                all_fragment += 1
        else:
            print('Соединение не установлено')
            return
        logger.info(f"Отправлено текстовое сообщение: {message.decode('utf-8')}")
        logger.info(f"Всего фрагментов: {all_fragment}, Повторные передачи (NACK):
{nack_fragment}")

    except ConnectionResetError:
        print('Соединение потеряно. Включите сервер.')
        print('Отправлено фрагментов:', fragment_count, ' всего фрагментов:',
num_of_fragment, '\n')

def set_client():
    """ Инициализация настроек клиента. Возвращает server_ip,
        client_port, server_port, fragmentation, введенные пользователем. """

    server_ip, client_port, server_port, fragmentation = '', '', '', ''
    while server_ip == '' or server_port == '' or fragmentation == '': #
считывание значений для установки клиентом

```

```

try:
    if server_ip == '':
        server_ip = input('Введите IP-адрес сервера: ')
        socket.inet_aton(server_ip)

    if client_port == '':
        client_port = int(input('Введите порт клиента/источника: '))

    if client_port < 1024 or client_port > 65535: #
        print('Этот порт зарезервирован. Введите другой.')
        client_port = ''
        continue

    if server_port == '':
        server_port = int(input('Введите порт сервера: '))

    if server_port < 1024 or server_port > 65535: #
        print('Этот порт зарезервирован. Введите другой.')
        server_port = ''
        continue

    fragmentation = int(input('Введите максимальный размер фрагмента: '))

    if fragmentation < protocol.FRAGMENT_MIN: # проверка заданного
значения фрагмента
        fragmentation = protocol.FRAGMENT_MIN

    if fragmentation > protocol.FRAGMENT_MAX: # проверка максимального
значения фрагмента
        fragmentation = protocol.FRAGMENT_MAX

except ValueError: # введен неправильный тип данных
    print('Неверный ввод! Попробуйте снова')
    server_ip, server_port, fragmentation = '', '', ''
    continue

except OSError: # введен неправильный IP-адрес
    print('Неверный IP-адрес! Попробуйте снова')
    server_ip = ''
    continue

return server_ip, client_port, server_port, fragmentation

def user_interface():
    """ Интерфейс пользователя клиента. """

```

```

print('\n{:^50}'.format('client'))
server_ip, client_port, server_port, fragmentation = set_client()
client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) # установить
сокет с IPv4 и UDP
client_socket.bind(('', client_port)) # установить порт источника

while True:
    print('0 - выход\n'
          '1 - отправить текстовое сообщение\n'
          '2 - отправить файл\n'
          '9 - переключиться на сервер')
    user_input = input('Введите, что вы хотите сделать: ')

    if user_input == '0':
        print('Выход')
        client_socket.close()
        sys.exit(0)

    if user_input == '1':
        message = bytes(input('Введите сообщение: '), 'utf-8')
        send_message(server_ip, client_socket, server_port, fragmentation,
message)

    elif user_input == '2':
        file = bytes(input('Введите путь к файлу: '), 'utf-8')
        if not os.path.isfile(file): # check if given path is valid
            print('ERROR 01: Путь', file.decode('utf-8'), 'не существует!')
            continue
        send_file(server_ip, client_socket, server_port, fragmentation +
protocol.HEADER_SIZE, file)

    elif user_input == '9':
        client_socket.close()
        server.user_interface()

    else:
        print('\n{:^30}'.format('Неверный ввод! Попробуйте снова.\n'))

```

server.py

```

import client
import protocol
import socket

```

```

import sys
import os
import select
import logging
from datetime import datetime

log_filename = f"transfer_{datetime.now().strftime('%Y%m%d_%H%M')}.log"
logging.basicConfig(
    filename=log_filename,
    filemode='w',
    level=logging.INFO,
    format='%(asctime)s [%(levelname)s] %(message)s'
)

def write_msg(server_socket, fragment_size, fragment_no):
    """ Выводит полученное текстовое сообщение в консоль.

    server_socket: Серверный сокет содержит адрес источника и метод sendto.
    fragment_size: Максимальный размер полученного фрагмента данных. Вводится
пользователем.
    fragment_no: Общее количество фрагментов. """

    fragment_count = 0
    while True:
        ready = select.select([server_socket], [], [], 3)
        if ready[0]:
            message, client_address = server_socket.recvfrom(protocol.DEFAULT_BUFF)
            reply_crc = protocol.check_crc(message)
            reply = protocol.add_header(reply_crc, fragment_size, b'')
            server_socket.sendto(reply, client_address)
            if reply_crc.value !=
protocol.MsgType.ACK.value:                # ARQ Stop & Wait
                continue
            message = protocol.get_data(message)
            print(message.decode('utf-8'), end='')
            fragment_count += 1
        else:
            break
    print('\nПолученные фрагменты:', fragment_count, ' учтенные фрагменты:',
fragment_no, '\n')

def resolve_filename_collision(path):

```

```

        """Автоматически добавляет (1), (2), ... если файл с таким именем уже
        существует."""
        if not os.path.exists(path):
            return path

        base, ext = os.path.splitext(path)
        counter = 1
        new_path = f"{base}({counter}){ext}"
        while os.path.exists(new_path):
            counter += 1
            new_path = f"{base}({counter}){ext}"
        return new_path

def write_file(path, server_socket, fragment_size, fragment_no):
    """ Запись полученного файла в указанную директорию.
    Возвращает количество полученных фрагментов и общее количество фрагментов.

    server_socket: Сокет сервера содержит адрес источника и sendto метод.
    path: Информация, где хранить полученный файл.
    fragment_size: Максимальный размер полученного фрагмента данных. Вводится
    пользователем.
    fragment_no: Общее количество фрагментов. """

    fragment_count = 0
    with open(path, 'wb+') as file:
        while True:
            ready = select.select([server_socket], [], [], 3)
            if ready[0]:
                data, client_address = server_socket.recvfrom(fragment_size)
                reply_crc = protocol.check_crc(data)
                reply = protocol.add_header(reply_crc, fragment_size, b'')
                server_socket.sendto(reply, client_address)

                if reply_crc.value != protocol.MsgType.ACK.value:
                    continue
                data = protocol.get_data(data)
                fragment_count += 1
                file.write(data)
                logging.info(f"Получен фрагмент {fragment_count}/{fragment_no}")
            else:
                break

    print('\nПолученные фрагменты:', fragment_count, ' учтенные фрагменты:',
    fragment_no, '\n')

```

```

def initialization(server_socket):
    """ Получение инициализационного сообщения. Возвращает полученные данные, адрес
    клиента, размер фрагмента,
    количество фрагментов и имя файла для приёма передаваемых данных.

    server_socket: Серверный сокет, содержащий адрес источника и метод sendto. """

    while True:
        ready = select.select([server_socket], [], [], 20)
        if ready[0]:
            data, client_address = server_socket.recvfrom(protocol.DEFAULT_BUFF)
        else:
            return None
        reply_crc = protocol.check_crc(data)
        reply = protocol.add_header(reply_crc, 0, b'')
        server_socket.sendto(reply, client_address)
        if reply_crc.value == protocol.MsgType.ACK.value:
            fragment_size = protocol.get_fragment_size(data) + len(data)

            if data.decode('utf-8')[:1] == protocol.MsgType.SET_MSG.value:
                fragment_count = protocol.get_fragment_count(data,
protocol.MsgType.SET_MSG.value)
            else:
                fragment_count = protocol.get_fragment_count(data)

            file_name = protocol.get_file_name(data, fragment_count)
            break
        return data, client_address, fragment_size, fragment_count, file_name

def receive(server_socket, dir_path):
    """ Получает данные от клиента и решает, основываясь на заголовке протокола,
    является ли это передача текстовых данных или файловых """

    print('Сервер ожидает!')
    try:
        data, client_address, fragment_size, fragment_count, file_name =
initialization(server_socket)
    except TypeError:
        print('Timeout')
        return

    if protocol.get_msg_type(data).decode('utf-8') == protocol.MsgType.SET.value:
        save_path = resolve_filename_collision(dir_path + file_name)
        write_file(save_path, server_socket, fragment_size, fragment_count)

```

```

        server_socket.sendto(bytes(protocol.MsgType.ACK.value, 'utf-8'),
client_address)
        print('Передача прошла успешно, файл находится',
os.path.abspath(save_path), '\n')
        logging.info(f"Файл сохранен как: {os.path.abspath(save_path)}")
        logging.info(f"Получен файл: {file_name}")
        logging.info(f"Сохранено в: {os.path.abspath(dir_path + file_name)}")
        logging.info(f"Ожидаемые фрагменты: {fragment_count}")

    else:
        write_msg(server_socket, fragment_size, fragment_count)

def set_server():
    """ Начальные настройки сервера. Возвращает порт сервера и путь к директории
для сохранения полученных файлов.
        И порт сервера, и путь к директории вводятся пользователем. """

    server_port = 0
    dir_path = ''
    while server_port == 0 or dir_path == '':
        try:
            if server_port == 0:
                server_port = int(input('Введите порт сервера для прослушивания:
'))

                if server_port < 1024 or server_port > 65535:
#
проверка порта сервера
                    print('Этот порт зарезервирован. Введите другой.')
                    server_port = 0
                    continue

                dir_path = input('Введите путь к директории, в которую нужно сохранить
файл: ')
                if not os.path.isdir(dir_path):
#
проверка пути
                    print('ERROR 01: Путь', dir_path, 'не существует!')
                    dir_path = ''
                    continue

                if dir_path[-1] != '/':
#
проверка, заканчивается ли имя директории на /
                    dir_path += '/'

            except ValueError:
#
неверный введенный тип данных

```



```

        print('Неверный порт сервера!')
        server_port = 0
        continue

    return server_port, dir_path

def user_interface():
    """ Интерфейс пользователя сервера. """

    print('\n{:^50}'.format('server'))
    server_port, dir_path = set_server()
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    server_socket.bind(('', server_port))

    while True:
        print('0 - выход\n'
              '1 - получение\n'
              '9 - переключиться на клиентский интерфейс\n')
        user_input = input('Введите, что вы хотите сделать: ')
        if user_input == '0':
            print('Завершение работы')
            server_socket.close()
            sys.exit(0)
        elif user_input == '1':
            receive(server_socket, dir_path)
        elif user_input == '9':
            client.user_interface()
        else:
            print('\n{:^30}'.format('Некорректный ввод! Введите еще раз.\n'))

```

gui.py

```

import tkinter as tk
from tkinter import ttk, filedialog, messagebox
import threading
import socket
import protocol
import client
import server
import os
import matplotlib.pyplot as plt

```

```

class UDPCommunicatorApp:
    def __init__(self, root):
        self.root = root
        self.root.title("UDP Communicator")
        self.root.geometry("500x400")

        # Переменные для полей ввода
        self.mode_var = tk.StringVar(value="client") # "клиент" или "сервер"
        self.ip_var = tk.StringVar(value="127.0.0.1")
        self.port_var = tk.IntVar(value=5000)
        self.client_port_var = tk.IntVar(value=5001)
        self.fragment_size_var = tk.IntVar(value=1024)
        self.file_path_var = tk.StringVar()
        self.message_var = tk.StringVar()
        self.dir_path_var = tk.StringVar(value=os.getcwd() + "/") # По умолчанию
текущий каталог
        self.simulate_error_var = tk.BooleanVar(value=False)

        self._build_ui()

    def _build_ui(self):
        # Выбор режима(Клиент/Сервер)
        tk.Label(self.root, text="Режим:").grid(row=0, column=0, padx=10, pady=5)
        tk.Radiobutton(self.root, text="Клиент", variable=self.mode_var,
value="client", command=self.update_ui).grid(
            row=0, column=1, padx=10, pady=5)
        tk.Radiobutton(self.root, text="Сервер", variable=self.mode_var,
value="server", command=self.update_ui).grid(
            row=0, column=2, padx=10, pady=5)

        # IP адрес (для клиента)
        self.ip_label = tk.Label(self.root, text="IP адрес сервера:")
        self.ip_entry = tk.Entry(self.root, textvariable=self.ip_var)

        # Порт сервера (для клиента и сервера)
        self.port_label = tk.Label(self.root, text="Порт сервера:")
        self.port_entry = tk.Entry(self.root, textvariable=self.port_var)

        # Порт клиента (для клиента)
        self.client_port_label = tk.Label(self.root, text="Порт клиента:")
        self.client_port_entry = tk.Entry(self.root,
textvariable=self.client_port_var)

        # Размер фрагмента (для клиента)
        self.fragment_size_label = tk.Label(self.root, text="Размер фрагмента:")

```

```

        self.fragment_size_entry = tk.Entry(self.root,
textvariable=self.fragment_size_var)
        self.simulate_error_check = tk.Checkbutton(self.root, text="Симуляция
ошибки", variable=self.simulate_error_var)

        # Путь к файлу (для клиента)
        self.file_path_label = tk.Label(self.root, text="Путь к файлу:")
        self.file_path_entry = tk.Entry(self.root, textvariable=self.file_path_var,
state="readonly")
        self.file_browse_button = tk.Button(self.root, text="Обзор",
command=self.choose_file)

        # Сообщение (для клиента)
        self.message_label = tk.Label(self.root, text="Сообщение:")
        self.message_entry = tk.Entry(self.root, textvariable=self.message_var)

        # Путь к каталогу (для сервера)
        self.dir_path_label = tk.Label(self.root, text="Каталог хранения:")
        self.dir_path_entry = tk.Entry(self.root, textvariable=self.dir_path_var,
state="readonly")
        self.dir_browse_button = tk.Button(self.root, text="Обзор",
command=self.choose_directory)

        # Кнопка запуска
        self.start_button = tk.Button(self.root, text="Запуск",
command=self.start_process)

        self.update_ui()

def update_ui(self):
    """ Обновление UI в зависимости от выбранного режима. """
    mode = self.mode_var.get()

    self.ip_label.grid_forget()
    self.ip_entry.grid_forget()
    self.port_label.grid_forget()
    self.port_entry.grid_forget()
    self.client_port_label.grid_forget()
    self.client_port_entry.grid_forget()
    self.fragment_size_label.grid_forget()
    self.fragment_size_entry.grid_forget()
    self.simulate_error_check.grid_forget()
    self.file_path_label.grid_forget()
    self.file_path_entry.grid_forget()
    self.file_browse_button.grid_forget()
    self.message_label.grid_forget()

```

```

self.message_entry.grid_forget()
self.dir_path_label.grid_forget()
self.dir_path_entry.grid_forget()
self.dir_browse_button.grid_forget()

# Отображение элементов интерфейса в зависимости от режима
if mode == "client":
    self.ip_label.grid(row=1, column=0, padx=10, pady=5)
    self.ip_entry.grid(row=1, column=1, columnspan=2, padx=10, pady=5)
    self.port_label.grid(row=2, column=0, padx=10, pady=5)
    self.port_entry.grid(row=2, column=1, columnspan=2, padx=10, pady=5)
    self.client_port_label.grid(row=3, column=0, padx=10, pady=5)
    self.client_port_entry.grid(row=3, column=1, columnspan=2, padx=10,
pady=5)
    self.fragment_size_label.grid(row=4, column=0, padx=10, pady=5)
    self.fragment_size_entry.grid(row=4, column=1, columnspan=2, padx=10,
pady=5)
    self.simulate_error_check.grid(row=5, column=0, columnspan=3, padx=10,
pady=5)
    self.file_path_label.grid(row=6, column=0, padx=10, pady=5)
    self.file_path_entry.grid(row=6, column=1, padx=10, pady=5)
    self.file_browse_button.grid(row=6, column=2, padx=10, pady=5)
    self.message_label.grid(row=7, column=0, padx=10, pady=5)
    self.message_entry.grid(row=7, column=1, columnspan=2, padx=10, pady=5)
else: # для сервера
    self.port_label.grid(row=1, column=0, padx=10, pady=5)
    self.port_entry.grid(row=1, column=1, columnspan=2, padx=10, pady=5)
    self.dir_path_label.grid(row=2, column=0, padx=10, pady=5)
    self.dir_path_entry.grid(row=2, column=1, padx=10, pady=5)
    self.dir_browse_button.grid(row=2, column=2, padx=10, pady=5)

self.start_button.grid(row=10, column=0, columnspan=3, pady=10)

def choose_file(self):
    """ Диалоговое окно для выбора файла """
    file_path = filedialog.askopenfilename()
    if file_path:
        self.file_path_var.set(file_path)

def choose_directory(self):
    """ Диалоговое окно для выбора каталога сохранения. """
    dir_path = filedialog.askdirectory()
    if dir_path:
        self.dir_path_var.set(dir_path + "/")

def start_process(self):

```

```

        """ Запуск процесса в зависимости от выбранного режима (клиент или сервер).
        """

        mode = self.mode_var.get()
        if mode == "client":
            self.start_client()
        else:
            self.start_server()

    def start_client(self):
        file_path = self.file_path_var.get()
        message = self.message_var.get()

        if not file_path and not message:
            messagebox.showerror("Ошибка", "Пожалуйста, укажите файл или сообщение для отправки.")
            return

        # симуляция ошибки
        client.user_input_mistake = 'д' if self.simulate_error_var.get() else 'н'

        # Запуск клиента в отдельном потоке
        thread = threading.Thread(target=self.run_client, args=(file_path, message))
        thread.start()

    def run_client(self, file_path, message):
        try:
            sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
            sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
            sock.bind(("", self.client_port_var.get()))

            if file_path:
                client.send_file(
                    server_ip=self.ip_var.get(),
                    client_socket=sock,
                    server_port=self.port_var.get(),
                    fragment_size=self.fragment_size_var.get() + protocol.HEADER_SIZE,
                    file_path=file_path.encode('utf-8')
                )
            elif message:
                client.send_message(
                    server_ip=self.ip_var.get(),
                    client_socket=sock,
                    server_port=self.port_var.get(),
                    fragment_size=self.fragment_size_var.get(),

```

```

        message=message.encode('utf-8')
    )

    except Exception as e:
        messagebox.showerror("Ошибка", str(e))
    finally:
        sock.close()

def start_server(self):
    """ Запуск процесса сервера. """
    thread = threading.Thread(target=self.run_server)
    thread.start()

def run_server(self):
    try:
        server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        server_socket.bind("", self.port_var.get())

        server.receive(server_socket, self.dir_path_var.get())
    except Exception as e:
        messagebox.showerror("Ошибка", str(e))
    finally:
        server_socket.close()

if __name__ == "__main__":
    root = tk.Tk()
    app = UDPCommunicatorApp(root)
    root.mainloop()

```

main.py

```

import client
import server
import sys

def main():
    if len(sys.argv) != 2:
        print("ERROR 00: Некорректный ввод!")
        sys.exit(-1)

```

```
if sys.argv[1] == 'client':
    client.user_interface()
elif sys.argv[1] == 'server':
    server.user_interface()
else:
    print("ERROR 00: Некорректный ввод!")
    sys.exit(-1)

if __name__ == '__main__':
    main()
```

photoСсылка на репозиторий github: