МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ М. В. ЛОМОНОСОВА
ФИЛИАЛ МГУ В ГОРОДЕ СЕВАСТОПОЛЕ

ФАКУЛЬТЕТ КОМПЬЮТЕРНОЙ МАТЕМАТИКИ

Направление подготовки 01.03.02
«Прикладная математика и информатика»
квалификация «бакалавр»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

РАЗРАБОТКА МЕТОДА РАЗДЕЛЕНИЯ ПОТОКА ДАННЫХ ДЛЯ ПЕРЕДАЧИ ФАЙЛОВ ОТ ПОЛЬЗОВАТЕЛЯ НА СУПЕРКОМПЬЮТЕР С УЧЕТОМ ПРИОРИТЕТОВ

Выполнил:
Осадчук Дмитрий Русланович
студент учебной группы ПМ – 401

Научный руководитель:
канд. физ. – мат. наук
Сальников Алексей Николаевич

Севастополь – 2018

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Развитие теоретической и практической базы некоторых современных приемов научных исследований достигло такого прогресса, что для их практического применения вычислительной мощности персонального компьютера становится недостаточно. Существует множество задач, которые для своего решения и последующего его анализа требуют вычислительные мощности совокупности сотен, или даже тысяч персональных компьютеров. В связи с этим, большое распространение и использование получили так называемые «суперкомпьютеры» или «кластеры», которые представляют собой множество серверов (компьютеров, не имеющих монитора, клавиатуры, мыши, устанавливающихся в специальные крепления), соединенных между собой высокоскоростными компьютерными сетями для обмена данными. Один суперкомпьютер может в сотни, или даже тысячи, раз превышает вычислительную мощность обычного ПК.

Благодаря этому факту, стало возможным написание и запуск программ, работающих и обрабатывающих большие объемы данных, с параллельными вычислениями (чтобы можно было задействовать как можно большее количество вычислительных узлов суперкомпьютера), что привело к сокращению времени работы таких программ и, соответственно, более быстрому получению результата. Иногда количество данных, которые требуют обработки, достигает нескольких сотен Гигабайт, а, в редких случаях, эта величина может превосходить даже несколько Терабайт памяти [2]. Естественно, совокупность таких программ, решающих задачу обработки и/или анализа большого объема данных, не образует все множество программ, запускаемых на суперкомпьютерах. Так же существуют задачи, где из, относительно, небольшого объема данных необходимо вычислить определенные его характеристики, или из небольшого объема данных

необходимо спрогнозировать поведение того или иного объекта при разных условиях путем ресурсоемких вычислений (имитационное моделирование поведения объектов, задачи биоинформатики) на суперкомпьютере [3]. Таким образом, при постановке задачи на выполнение на суперкомпьютере возникает необходимость передачи исходных данных для задач от администратора или пользователя на суперкомпьютер для последующей их обработки, а также при переносе результатов работы программ с суперкомпьютера на персональный компьютер пользователя. Для решения этой задачи актуально использование потоковых каналов связи между персональным компьютером и суперкомпьютером.

При передаче больших объёмов данных пользователем на суперкомпьютер важным критерием является удобство интерфейса для обмена данными с суперкомпьютером. Нетрудно догадаться, что при передаче файла большого размера, время передачи, особенно при относительно маленькой пропускной способности канала связи, может достигать больших значений. Например, для передачи файла размером 10 Гигабайт со скоростью 1 Мбит/сек потребуется около 22 часов времени. Существует вероятность того, что в этот временной промежуток может произойти некое событие (отключение электропитания, обрыв связи провайдером, экстренное завершение работы операционной системы и т. д.), которое приведёт к остановке или сбою передачи файла, причём, чем больше этот временной промежуток, тем больше эта вероятность. При возникновении сбоя в приёме/передаче файла, операцию, в которой произошёл сбой, необходимо будет повторить с начала, что требует дополнительных временных издержек и затрудняет запуск задач на суперкомпьютере.

Ещё одним критерием удобства программы является наличие пользовательского (графического) интерфейса. В программах, ориентированных на конечного пользователя всегда присутствует человеческий фактор, поэтому наличие графического интерфейса облегчает

работу с программой, делает её более понятной и структурированной для пользователя. Так же, одной из множества характеристик «удобной» для пользователя программы является её переносимость, то есть, платформо-независимость. Это качество позволяет одной и той же программе, с одним и тем же программным кодом, запускаться и корректно работать в распространённых операционных системах, например таких, как Linux и Windows. Таким образом, пользователю не придётся заботиться о том, чтобы загружать и устанавливать разные версии программы на компьютеры с разными операционными системами, что так же облегчает использование программы. К сожалению, на данный момент предоставляемый пользователю программный интерфейс для обмена файлами с суперкомпьютером не обладает подобными качествами («наличие графической оболочки» и «переносимостью»), что доставляет пользователю неудобства при его использовании.

Одним из немаловажных качеств программы, особенно при удалённой работе с суперкомпьютером, является идентификация пользователя и возможность программы различать злоумышленника. Для этого, чаще всего, в программах используются кодовые фразы, называемые логином и паролем, о которых знает только сам конечный пользователь. Так же допускается использование так называемых «ключей», которые представляют собой наборы символов для сравнения на суперкомпьютере и ПК, и автоматизируют процесс аутентификации пользователя на суперкомпьютере, либо экземпляра запущенной им программы.

В описанном контексте нередко возникают ситуации, когда требуется загрузить и/или передать сразу несколько файлов на суперкомпьютер, причем некоторые нужно принять/передать быстрее, чем другие, для планирования и запуска ряда задач. Для того чтобы какие-то файлы были загружены/выгружены быстрее, чем другие, возникает необходимость в разделении общего потока данных между ПК пользователя и суперкомпьютером на «подпотоки» для каждого из принимаемых и

передаваемых файлов. Текущая версия программного интерфейса не предоставляет возможности влиять на скорость приёма/передачи каждого отдельного файла.

Данная работа посвящена разработка программного интерфейса, обладающего вышеперечисленными качествами, который позволяет осуществлять передачу и загрузку файлов большого размера с суперкомпьютера.

ГЛАВА 1. ОСНОВНЫЕ ЦЕЛИ И ТРЕБОВАНИЯ

§1. Цель научной работы

Во введении был обозначен ряд следующих проблем, которые возникают при обмене файлов большого размера между ПК пользователя и суперкомпьютером:

- передача/загрузка файла с самого начала при сбое,
- аутентификация пользователя и/или его программного интерфейса на суперкомпьютере,
- отсутствие механизма «неравномерного» разделения потока данных между ПК и суперкомпьютером на «подпотоки» для каждого файла, участвующего в обмене,
- отсутствие и неполнота удобного для пользователя графического интерфейса для взаимодействия с программой,

Решением совокупности данных проблем может быть новый программный интерфейс для обмена файлами с суперкомпьютером, который обладает следующими качествами:

- при повторной попытке загрузить/передать файл после сбоя, программный интерфейс должен учитывать часть уже принятых/переданных данных. Это можно реализовать путём проверки размера файла на сервере, который был частично передан, либо, в случае если производилась загрузка файла, то проверкой размера части загруженного файла на персональном компьютере пользователя. После того, как программный интерфейс «узнал» количество принятой/переданной информации в прошлый раз, он может начать снова

принимать/передавать этот же файл, но уже с того места, на котором он остановился при сбое,

- при запуске и подключении к суперкомпьютеру, программный интерфейс должен запрашивать у пользователя имя его учётной записи и пароль, тем самым предотвращая доступ посторонних лиц к суперкомпьютеру (возможно, злоумышленников) и делая возможным идентификацию пользователя на суперкомпьютере для дальнейшей безопасной работы,
- программный интерфейс должен предоставлять возможность параллельной загрузки и передачи файлов, а так же влияния на скорость загрузки и передачи каждого из файлов. Для воплощения этого качества программного интерфейса рассмотрим 2 варианта его реализации: задание статического значения скорости приёма/передачи для каждого файла и введение системы приоритетов загрузки/передачи, так же для каждого из файлов. Первый вариант предполагает статическую пропускную способность канала связи, если пропускная способность каждого из «подпотоков» будет задаваться конкретным значением, но, зачастую, канал связи между ПК и суперкомпьютером имеет переменную пропускную способность, поэтому данный вариант не является подходящим в данном контексте. Так же, пользователь мог бы задавать пропускную способность каждого «подпотока» в процентах от общей доступной пропускной способности канала связи, однако данный подход перестаёт быть удобным для использования при увеличении количества одновременно принимаемых и передаваемых файлов, так как при добавлении каждой новой загрузки или передачи файла, необходимо будет освобождать «нужные проценты» ширины канала связи под каждую новую загрузку/передачу файла. Второй вариант – введение системы

приоритетов для всех «загрузок и выгрузок», которые так же будут задаваться пользователем. Это позволит разделять общий канал связи пропорционально заданным приоритетам для каждой загрузки или передачи файла, а, следовательно, избавит пользователя от необходимости в «выкраивании лишних» процентов пропускной способности канала связи для задания их новой загрузке или передаче файла. В качестве варианта для решения проблемы «неравномерной» параллельной передачи и загрузки файлов был выбран второй вариант, что обосновано его преимуществом перед первым вариантом, описанным выше,

- графический интерфейс программы должен быть реализован и интуитивно понятен для конечного пользователя. Он так же должен предоставлять возможность параллельной загрузки/передачи нескольких файлов и назначения приоритета этим «загрузкам/выгрузкам».

Главной целью данной работы является предоставление пользователю персонального компьютера возможности обмениваться файлами большого размера с другим компьютером (суперкомпьютером) параллельно, в обе стороны (прием/передача), с возможностью задания приоритета принимаемым и передаваемым файлам (загрузкам/выгрузкам) с помощью предоставления пользователю специального программного интерфейса.

§2. Обзор теоретической базы

Рассмотрим теоретическую сторону контекста требуемого программного обеспечения. Для обмена файлами с суперкомпьютером, чаще всего, между FTP–сервером (программе, работающей на суперкомпьютере) и FTP–клиентом (программе, работающей на персональном компьютере пользователя) устанавливается Интернет соединение по FTP–протоколу (File Transfer Protocol), через которое осуществляется идентификация

пользователя, загрузка его рабочей файловой директории (папки) на суперкомпьютере и, естественно, сам обмен файлами. Для лучшего понимания сути материала, излагаемого далее, необходимо ввести определение «FTP–протокола» и изложить основные принципы его работы, однако, перед тем, необходимо определить архитектуру «клиент-серверного» приложения.

Клиент–серверная архитектура приложения устроена таким образом, что приложение, обладающее данной архитектурой, состоит из двух частей: клиентской и серверной. Серверную часть приложения (программы) будем называть «сервером», а клиентскую – «клиентом». Обычно, каждая из этих частей представляет собой отдельную программу. Особенность данной архитектуры заключается в том, что, зачастую, сервер обменивается данными с несколькими клиентами одновременно, а клиент обменивается данными только с одним сервером. Так же, сервер может взаимодействовать с единым хранилищем данных, чтобы хранить и использовать в нем определенную информацию. Если представить эту архитектуру графически, то она будет иметь вид как на картинке ниже:

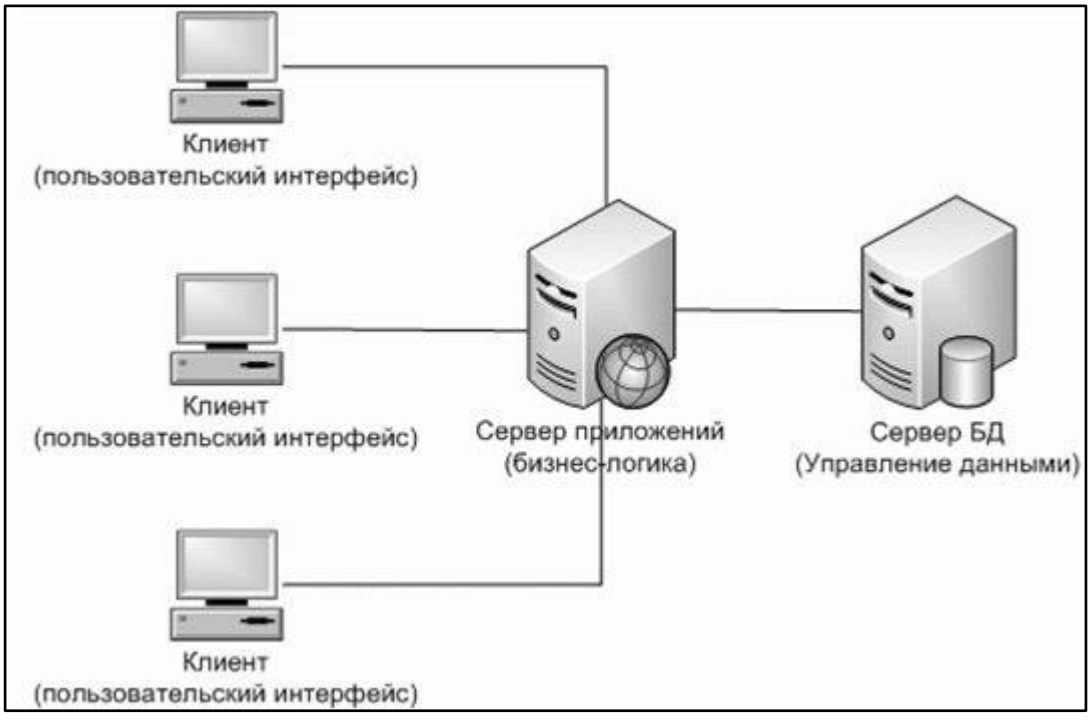


Рисунок 1: клиент-серверная архитектура приложений

Приложения, которые взаимодействуют через глобальную сеть, чаще всего, имеют именно такую архитектуру. То есть, используется один или несколько территориально–удаленных серверов, которые обрабатывают запросы клиентов, предоставляя им необходимый функционал через каналы связи (сеть Интернет).

Переходя к определению понятия «FTP–протокола», стоит упомянуть, что для сетевых приложений так же используется так называемая **«peer to peer»** архитектура, в которой отсутствуют выделенные серверы и каждый участок сети (клиент) равен по значимости и функционалу с другими, то есть функционал и роль каждого «клиента» одинаковы. Структура сети с такой архитектурой представлена на картинке ниже.

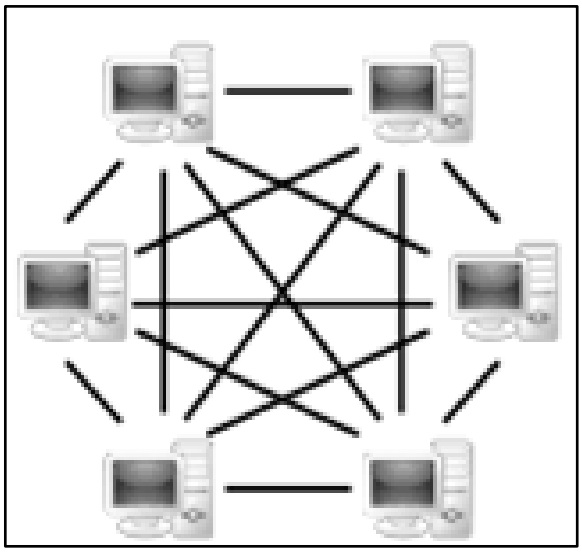


Рисунок 2: peer to peer сеть

Стоит отметить, что данный вид архитектуры не подходит для рассматриваемого программного обеспечения. Рассматривая совокупность узлов в образуемой при работе программы сети, можно выделить те, которые работают только с одним узлом (персональные компьютеры пользователей работают только с суперкомпьютером и не взаимодействуют между собой), и те, которые работают с множеством узлов (сами суперкомпьютеры, которые принимают и передают файлы из своей файловой системы). Таким образом,

возникает «неравенство» между двумя типами узлов в сети, что противоречит основной идее рассматриваемого типа архитектуры программного обеспечения.

«Протоколом» в компьютерных сетях называется набор правил, следуя которым, два или более компьютеров в сети могут корректно обмениваться данными друг между другом.

FTP (File Transfer Protocol) – стандартный протокол, предназначенный для передачи файлов по TCP – сетям (например, Интернет). FTP часто используется для обмена файлами между двумя устройствами в сети.

Протокол построен на архитектуре «клиент–сервер» и использует разные сетевые соединения для передачи команд и данных (файлов) между клиентом и сервером. Пользователи FTP могут пройти аутентификацию, передавая логин и пароль открытым или зашифрованным текстом, или же, если это разрешено на сервере, они могут подключиться анонимно.

Если попытаться графически представить взаимодействие FTP–сервера и FTP–клиента, то получим схему, представленную на картинке ниже.

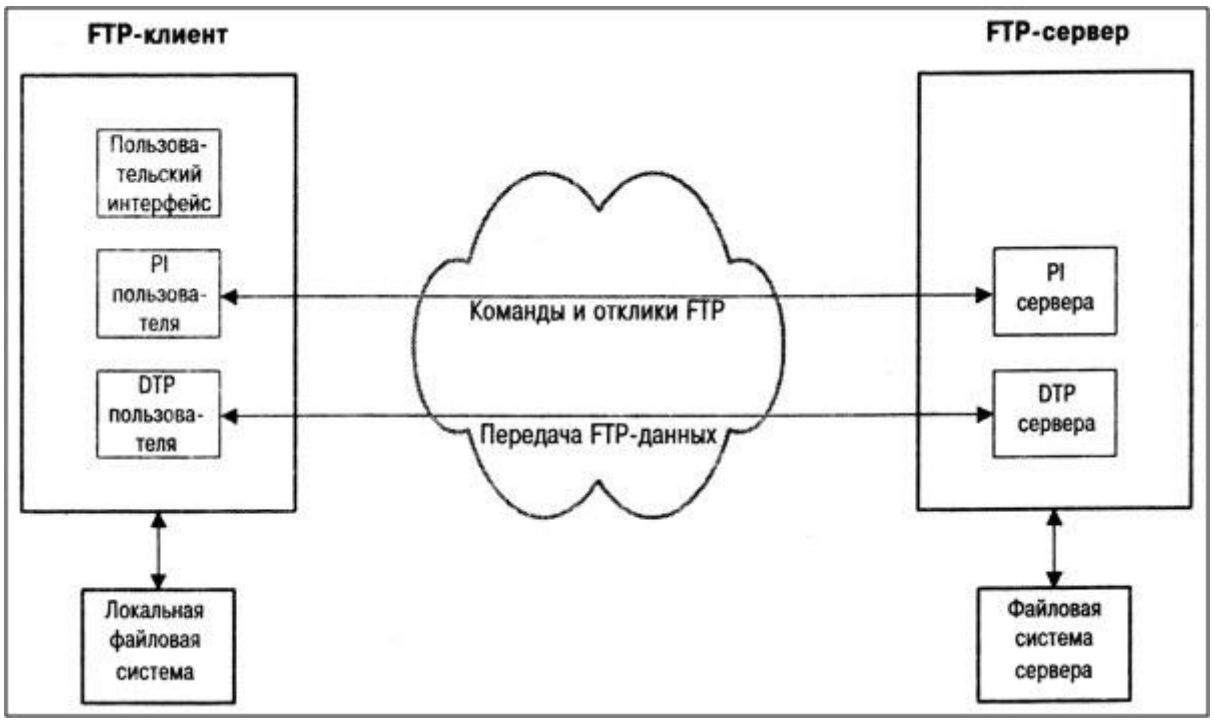


Рисунок 3: схема работа протокола FTP

То есть, для передачи одного файла (от персонального компьютера пользователя на сервер или наоборот) между клиентом и сервером создаются два соединения: первое – для обмена командами и ответами (на естественном языке это можно представить так: «Сервер, скажи, пожалуйста, какие файлы находятся у тебя в моей рабочей директории?», или «Сервер, прекрати передавать мне файл!»; второе – для обмена самими данными, то есть самим содержимым файлов (байтами информации, из которых состоит копируемый файл). Такое устройство протокола предоставляет разработчику инструменты для полного управления приемом и передачей файлов. Однако этот подход имеет один недостаток: для передачи на сервер или скачивания на персональный компьютер каждого из файлов требуется держать открытыми 2 соединения (командное и информационное) между сервером и клиентом, то есть, для одновременной передачи, например, 3 файлов, требуется 6 открытых соединений. Для лучшего понимания излагаемого, будет полезным дать определение «порту» в компьютерных сетях.

Порт – это некое целое число от 1 до 65 535, которое указывается в заголовках пакета данных протокола TCP/IP для указания того, какому из процессов на компьютере – получателе адресован данный пакет. Грубо говоря, если провести аналогию с обычным бумажным письмом, то IP-адрес – это страна, город, улица и номер дома, в котором живет получатель, а порт – это имя получателя (так как в одном доме могут жить несколько человек).

Если учитывать обстоятельство того, что для одного передаваемого файла требуется 2 соединения, а, следовательно, 2 порта, то со стороны клиента ситуация некритична, ведь среднее количество параллельных загрузок варьируется от 5ти до 20ти, что означает, что, в среднем, количество открытых соединений будет колебаться от 10ти до 40ка. Что же касается серверной части приложения – здесь ситуация обстоит несколько сложнее. Исходя из устройства компьютерных сетей [5] нам известно, что количество портов на один компьютер в сети равно 65535. Около 1000 из этих портов будем считать зарезервированными системой (то есть, некоторые порты

зарезервированы международными стандартами для корректной работы определенных сервисов, например, 80ый порт отведен для работы HTTP–протокола). Полный список зарезервированных портов можно посмотреть здесь [12]. Это значит, что, используя FTP–протокол, серверная часть приложения сможет одновременно принимать/передавать примерно (65 000 – 1 000) / 2 = 32 000 файлов. Если считать, что один подключенный клиент будет параллельно скачивать/закачивать в среднем 5 файлов, то максимальное количество клиентов, которое может обрабатывать сервер единовременно равно примерно 32 000 / 5 = 6 400 пользователей. На первый взгляд, это обстоятельство может показаться сильно ограничивающим серверную сторону программы, однако, давайте проведем простой расчет. Допустим, скорость Интернет соединения у компьютера (хоста), на котором работает серверная часть программы равна 1 Гбит/сек = 1024 Мбит/сек = 1 048 576 Кбит/сек = 1 073 741 824 бит/сек = 134 217 728 Байт/сек = 131 072 Кбайт/сек = 128 Мегабайт/сек. При максимальной загрузке сервер сможет отдавать/принимать файлы со скоростью 131 072 / 32 000 = 4.096 Кбайт/сек = 4.096 / 1024 Мегабайт/сек = 0.004 Мегабайт/сек. Файл, размером всего 2 Гигабайта будет передаваться с такой скоростью 142.2 часа, а это примерно 6 суток. Для сервера, в рассматриваемом примере, выделен достаточно широкий на сегодняшний день канал связи (по условию 1 Гбит/сек), но даже при такой ширине канала поддержка сервером более чем 32 000 одновременно передаваемых или принимаемых файлов просто нерациональна, поэтому подход к реализации клиент–серверного приложения с использованием протокола FTP является более чем оправданным.

Следует отметить, что существует еще один вариант для организации коммуникации между сервером и клиентом. Вместо использования протокола FTP, можно разработать собственный протокол, причем таким образом, что для передачи файлов и команд будет использоваться константное количество соединений (вероятнее всего это число равно двум – для приема и для отправки), вне зависимости от количества скачиваемых/передаваемых файлов

от одного клиента. Тогда количество клиентов, которых сервер сможет обрабатывать, единовременно вырастет до 32 000. Это кажется более выгодным вариантом по сравнению методом реализации, использующим FTP–протокол. Однако у данного способа есть «обратная сторона медали» – это избыточный трафик и сложность реализации. Чтобы организовать параллельную передачу файлов по одному каналу, необходимо раздробить файлы на множество маленьких кусочков. Будем называть такие «кусочки» (части) файлов **«кадрами»**. Затем, последовательно передавать кадры из разных файлов по каналу связи. При уменьшении размера кадра каждого из файлов и увеличении ширины (пропускной способности) канала связи, эффект «параллелизма» проявлялся бы все четче и сильнее. Избыточный трафик возникает тогда, когда у принимающей файлы стороны возникает вопрос: «От какого файла я только что получил кадр?». Для ответа на него отправляющей кадры файлов стороне приходится заключать каждый кадр от каждого файла в так называемые «теги», из которых в свою очередь получатель может извлечь полезную для него информацию, подобно той, которая сообщит, к какому файлу следует отнести только что принятый кадр файла. Как было упомянуто выше, файлы могут достигать размеров свыше 10 Гбайт, а для достижения эффекта «параллелизма» передачи нескольких файлов требуется «дробить» файлы на маленькие кадры, и чем их размер меньше – тем лучше. Из этого следует, что при уменьшении размера кадров передаваемого файла и увеличении размера передаваемых файлов, количество информации, содержащей в себе теги (то есть не полезной информации), которое необходимо передать вместе с кадрами файлов становится очень большим. Этот факт говорит о том, что при параллельной передаче нескольких файлов большого размера, канал связи будет использоваться неэффективно, и время передачи файлов будет увеличиваться.

§3. Обзор готовых программных интерфейсов

Перейдём к обзору конкретных примеров реализации пользовательских (клиентских) программных интерфейсов, чьи алгоритмы работы основаны на вышеописанном теоретическом аспекте работы подобных программ.

WinSCP [23]

Свободный графический клиент протоколов SFTP и SCP, предназначенный для Windows. Распространяется по лицензии GNU GPL. Обеспечивает защищённое копирование файлов между компьютером и серверами, поддерживающими эти протоколы. К достоинствам данного программного обеспечения можно отнести удобство графического интерфейса, поддержка сразу нескольких протоколов, защищённость копируемых файлов. Однако данный программный интерфейс не предоставляет возможности разделения потока данных в пропорциональном отношении, а лишь делает возможным статически ограничивать скорость загрузки/передачи каждого файла. Так же, к недостаткам следует отнести и тот факт, что данное программное обеспечение работает только под управлением операционной системы Windows, что говорит о том, что данное ПО не обладает качеством «переносимости».

FileZilla [24]

Свободный многоязычный FTP-клиент с открытым исходным кодом для Microsoft Windows, Mac OS X и Linux. Он поддерживает FTP, SFTP, и FTPS (FTP через SSL/TLS) и имеет настраиваемый интерфейс с поддержкой смены тем оформления. Достоинства: возможность «перетаскивания» объектов мышью, синхронизации директории, поиска на удаленном сервере, поддержка многопоточной загрузки файлов, а также дозагрузки при обрыве (если поддерживается сервером) интернет-соединения. «Переносимостью» данное ПО так же не обладает, так как пользователю придётся позаботиться о том, чтобы загрузить и установить ту версию программы, которая соответствует версии ОС, в которой он хочет использовать данное ПО.

Данный программный интерфейс предоставляет возможность задания приоритетов копируемым файлам, однако набор таких приоритетов ограничен 5-ью значениями: «Самый низкий», «Низкий», «Средний», «Высокий» и «Высочайший». При приеме/передаче файлов (большого количества и размера каждого отдельного из них) на суперкомпьютер, часто требуются более тонкие настройки приоритетов копируемых файлов. В-частности, каждому из копируемых файлов необходимо ставить в соответствие целое положительное число от единицы до бесконечности, называемое «приоритетом», которое далее будет использоваться механизмом разделения и контроля потока данных для определения пропускной способности «подпотока» данных для каждого из копируемых файлов. Это позволило бы пользователю более точно распределять весь поток данных между всеми копируемыми файлами, и, как следствие, упростило бы процесс планирования и запуска задач на суперкомпьютере.

Были рассмотрены одни из наиболее распространенных программных интерфейсов для копирования файлов «с» и «на» удалённый компьютер, однако каждый из них лишь частично удовлетворяет потребностям пользователя, которому необходимо производить обмен файлами с суперкомпьютерами.

Таким образом, возникает необходимость в новом программном обеспечении, которое удовлетворяет полному перечню потребностей пользователя при обмене файлами большого размера с суперкомпьютерами, которые были описаны выше. Сформулируем основные требования к необходимому программному обеспечению.

§4. Требования к методу разделения потока данных

Как было упомянуто в задаче данной работы, необходимо разработать метод разделения потока данных, который при параллельной двунаправленной передаче файлов с помощью FTP–протокола учитывал бы

приоритет передаваемых и загружаемых файлов, которые задаются пользователем. Файлы, которые имеют высокий приоритет, с использованием разрабатываемого метода, должны передаваться быстрее, чем файлы, которым пользователь задал более низкий приоритет. Разработанный метод должен «справедливо» разделять поток данных в соответствии с расставленными приоритетами, то есть если, например, двум файлам заданы приоритеты 1 и 2 соответственно, то скорости передачи этих файлов должны соотноситься друг с другом именно в таком соотношении, а именно как 1:2, иначе введенная система приоритетов будет терять свой смысл.

Разрабатываемый метод сделает возможным отправку и получение файлов, приоритетно требуемых пользователю, быстрее, чем других, что сделает приём и передачу файлов и планирование задач на суперкомпьютере более удобными для пользователя, а также снизит время полного цикла выполнения приоритетных задач на суперкомпьютере.

§5. Требования к серверной части приложения

Предполагается, что серверная часть приложения будет работать на суперкомпьютере, а клиентская, соответственно, на персональных компьютерах пользователей.

Основные требования к серверу будут изложены ниже в виде перечня требований к его функционалу и особенностям его работы.

Необходимо, чтобы работающий сервер:

- Предоставлял возможность подключиться к нему с помощью клиента на протяжении всего периода своей работы;
- Вел учетную базу данных пользователей, которые могут подключиться к серверу;
- Каждому новому пользователю предоставлял доступ к своей и только своей рабочей директории;

- Вел учет операций, которые пользователи выполняют с помощью подключенного к серверу клиента;
- Позволял клиенту просматривать список файлов, скачивать/удалять/загружать файлы из рабочей директории пользователя в соответствии с заданными им правами;
- Функционировал на любой из UNIX платформ, то есть был кроссплатформенным;
- При разрыве соединения с клиентом и повторном его последующем подключении продолжал отдавать и принимать файлы от клиента с того же «места» (номера байта), на котором остановилась передача и загрузка этих файлов;
- Использовал «ширину» канала связи (потока данных) эффективно, то есть, чтобы ограничением скорости передачи данных от сервера к клиенту и обратно была не определенная в сервере константа, а максимальная производительность сервера, который обслуживает многих клиентов параллельно и пропускная способность канала связи.

§6. Основные требования к клиентской части приложения

Опишем основные положения и требования, которым должна соответствовать клиентская часть программы (клиент), то есть программа, которая будет работать на персональном компьютере пользователя.

Совокупность всех требований к клиенту можно разбить на несколько категорий: требования к особенностям работы и функционалу клиента и требования к его графическому интерфейсу.

Основные требования к особенностям работы и функционалу клиентской части приложения

Необходимо, чтобы работающий клиент:

- Осуществлял FTP–подключение к серверу по заданному IP–адресу и номеру порта;
- Предоставлял реализованные пользовательские функции в соответствии с FTP–протоколом, а именно: перемещаться по удаленной рабочей директории пользователя; получать список файлов и информации о них в текущем каталоге; перемещаться по каталогам, к которым у пользователя есть доступ;
- Предоставлял возможность скачивать файлы с сервера;
- Предоставлял возможность выгружать файлы на сервер;
- Предоставлял возможность обмениваться файлами с сервером параллельно;
- Позволял задать приоритет файлам, участвующим в обмене;
- Предоставлял возможность поставить загрузку/выгрузку файлов на паузу или полностью остановить передачу или прием конкретного файла;
- Предоставлял возможность создать каталог в удаленной рабочей директории;
- Предоставлял возможность переименовать файл, расположенный в удаленной рабочей директории;
- Функционировал на любой из UNIX–платформ, то есть был кроссплатформенным;
- Реализовывал метод разделения потока данных, требования к которому были описаны ранее, с использованием тех приоритетов файлов, что задал пользователь;

Основные требования к графическому интерфейсу клиентской части приложения

Необходимо, чтобы графический интерфейс клиентской части приложения:

- Предоставлял пользователю возможность ввести адрес сервера (IP–адрес и номер порта, на котором сервер будет принимать подключения) для подключения, а также данные своей учетной записи (логин и пароль) для предоставления доступа на сервер;
- При неуспешном подключении клиента к серверу уведомлял пользователя об ошибке подключения к серверу;
- Предоставлял возможность графического просмотра списка файлов на сервере в своей рабочей директории и основную информацию о них;
- Предоставлял возможность графического просмотра списка файлов в своей локальной рабочей директории и основную информацию о них;
- Отображал каждый принимаемый и передаваемый файл, а также основную информацию об этом файле и состоянии приема/передачи в понятном для пользователя формате;
- Позволял пользователю задать приоритет каждому из принимаемых и загружаемых файлов с помощью графического контекстного меню;
- Предоставлял пользователю возможность выбрать файл для загрузки на сервер или с сервера;
- Предоставлял пользователю возможность проводить операции с файлами в своей рабочей директории на сервере с помощью графического контекстного меню (изменение имени файла, удаление файла, создание новой папки);
- Предоставлял возможность переходить в другие папки своей рабочей директории на сервере и в локальной файловой системе;

§7. Постановка задачи

После обзора теоретической базы, готовых решений, анализа их функционала и основных качеств, а также формулировки требований к новому программному продукту, можно перейти к постановке задачи данной работы.

В ходе выполнения данной работы было поставлена следующая задача: разработать программное обеспечение, имеющее клиент–серверную архитектуру, работающее по FTP–протоколу, которое будет использовать разработанный метод для разделения потока данных между клиентом и сервером для параллельного двунаправленного обмена файлами большого размера с учетом пользовательских приоритетов. Разработанное программное обеспечение должно удовлетворять всем требованиям, описанным выше.

Выразив основные цели и требования, а также сформулировав задачу данной работы, целесообразно перейти к рассмотрению различных методологий, подходов и механизмов их достижения.

ГЛАВА 2. ОБЗОР И ВЫБОР ВНУТРЕННЕГО УСТРОЙСТВА СЕРВЕРА

Серверы, в программном их понимании, бывают разных типов. Они отличаются по способу обслуживания входящих подключений. Вообще говоря, чтобы разобраться с разными видами обслуживания соединений на сервере, необходимо понять основные принципы работы самих серверов, и общий принцип взаимодействия клиента с сервером, чему будет посвящен первый параграф этой главы.

В конце данной главы будет произведен выбор подхода для обслуживания соединений клиентов с сервером, взвешены все «за» и «против».

§1. Принцип работы серверного программного обеспечения

Описание основных принципов работы серверной части приложения требует введения нескольких определений, которые будут использоваться в дальнейшем.

Сокет – название программного интерфейса для обеспечения обмена данными между процессами. Процессы при таком обмене могут исполняться как на одной ЭВМ, так и на различных ЭВМ, связанных между собой сетью. Сокет – абстрактный объект, представляющий конечную точку соединения. При использовании сетевых сокетов в стеке протоколов TCP/IP, сокет «привязывается» к определенному порту устройства для того, чтобы другое устройство могло отправить пакеты данных по IP–адресу первого устройства и на определенный порт, а получивший эти данные сокет у первого устройства мог распознать, что данные адресовались именно ему. Следует отметить, что так же существуют и другие распространённые стеки сетевых протоколов, например UDP, однако данный стек не гарантирует целостность принятых данных. В связи с этим размер принятого или переданного файла

может отличаться от размера файла «оригинала», что является недопустимым в контексте рассматриваемой задачи.

При использовании стека протоколов TCP/IP, гарантирующего целостность принятых данных, следует различать клиентские и серверные сокеты. **Клиентские сокеты**, абстрагируясь от конкретной реализации, можно сравнить с конечными аппаратами телефонной сети, а серверные – с коммутаторами. Клиентское приложение, например браузер, использует только клиентские сокеты, а серверное, например веб–сервер, которому браузер посылает запросы, использует как клиентские, так и серверные сокеты.

Серверные сокеты (еще их иногда называют «слушающими») – это такие сокеты, которые открыты для подключения устройств извне по сети. Операционная система следит за изменением состояния слушающего сокета, к которому, потенциально, будут приходить попытки подключения и передачи информации от других устройств в сети. Когда «слушающий» сокет принял подключение, чаще всего, создается новый, клиентский, сокет и полученное подключение передается ему.

Таким образом, совмещая все вышеупомянутое, можно описать общую структуру взаимодействия сервера и клиента: пользователь на своем персональном компьютере запускает программу (клиент), в которой указаны или можно указать IP–адрес хоста и номер порта, к которому будет производиться подключение; при попытке подключения по указанному адресу (при условии, что этот адрес правильный и сервер на хосте работает в нормальном режиме) у сервера изменяется состояние слушающего сокета (сокет принимает подключение) и создается новый клиентский сокет для обмена данными с только что подключившимся клиентом, после чего «слушающий» сокет продолжает принимать запросы на создание соединения от других клиентов.

Как было сказано ранее, сервер может взаимодействовать сразу со многими клиентами, и была рассмотрена общая структура их

взаимодействия, поэтому далее будут рассмотрены основные подходы к обработке уже принятых подключений от клиентов на стороне сервера.

§2. Однопоточный подход

В первом параграфе был рассмотрен процесс подключения самого клиента к серверу, однако взаимодействие между ними на этом не прекращается. На этапе, когда соединение между сервером и клиентом установлено, и они могут обмениваться данными, возникает вопрос в том, как организовать обработку каждого из подключений на сервере.

Современные серверы могут обрабатывать большое количество соединений одновременно, благодаря особенностям организации их обработки. Рассмотрим простейший, однопоточный случай.

При использовании данного подхода, сервер принимает подключение на слушающем сокете и далее работает с этим подключением в основном потоке выполнения программы. Это говорит о том, что одно принятое соединение блокирует собой основной поток выполнения программы (сервера) до тех пор, пока данное соединение не будет разорвано. Только в таком случае сервер сможет перейти к принятию и обработке следующего соединения.

У такого подхода есть существенный минус – сервер не может обрабатывать больше одного соединения одновременно, поэтому данный подход для реализации требуемого серверного приложения не подходит.

Данный пример был рассмотрен лишь в качестве примитива (на сегодняшний день он практически не используется в разработке), чтобы после его рассмотрения была более понятна суть написанного ниже.

§3. Многопоточный подход

Из названия подхода можно догадаться, что при его использовании используется не один поток для обработки соединений, а сразу несколько. Чтобы описать алгоритм работы сервера, использующего такой подход, необходимо ввести определение «потока» в операционной системе.

Поток выполнения – наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы. Реализация потоков выполнения и процессов в разных операционных системах отличается друг от друга, но в большинстве случаев поток выполнения находится внутри процесса операционной системы. Несколько потоков выполнения могут существовать в рамках одного и того же процесса и совместно использовать аппаратные ресурсы, такие как память.

Алгоритм работы сервера, основанного на таком подходе довольно простой. При подключении клиента к серверу, слушающий сокет принимает запрос на подключение от клиента и передает дальнейшую работу с соединением в отдельный поток. В таком потоке происходит вся работа сервера с отдельным клиентом. Таким образом, можно выстроить цепочку последовательных действий и операций на сервере, которая будет являться неким условным протоколом работы сервера с отдельным клиентом. При этом, как было описано ранее, операционные системы поддерживают параллельную обработку нескольких потоков, что обуславливает работу сервера одновременно с несколькими клиентами.

На сегодняшний день многопоточный подход широко используется в серверах, основными требованиями к которым является параллельная обработка соединений и поддержание соединения с клиентом длительное время.

Известный пример сервера, в котором используется многопоточный подход – Apache 2.0 [20].

Данный подход широко используется и сегодня, ведь он имеет ряд достоинств, которые отлично сочетаются с прямым назначением сервера: параллельной обработкой большого количества клиентов. Однако, согласно

тестам проведенными независимыми разработчиками, данный подход перестает использовать ресурсы компьютера рационально при одновременной обработке более чем 10000 соединений [25]. Суть проблемы кроется в том, что при переходе от выполнения одного потока к другому, компьютер тратит на это определенное количество своих ресурсов. Поэтому был разработан еще один подход к обработке большого количества соединений, для которого не требуется большое количество ресурсов при переходе от обработки одного соединения к обработке другого.

§4. Асинхронный подход

Асинхронный подход к обработке большого количества соединений начал набирать свою популярность, когда количество пользователей, требующих параллельного взаимодействия с сервером стало превышать такое число, при котором даже довольно мощные компьютеры (хосты) начали не успевать обрабатывать все соединения. Этот период приходится на время, когда глобальная сеть Интернет начала широко распространяться среди населения и внедряться в повседневную жизнь человека. Количество пользователей таких гигантов как Facebook, Google и других стало увеличиваться по экспоненте.

Это подтолкнуло разработчиков на мысль о том, что нужно каким–то образом обрабатывать запросы к серверу только тогда, когда это требуется, а не порождать огромное множество потоков, требующих много ресурсов для своей работы и переключения от одного к другому.

Похожая технология уже была разработана и довольно широко применялась в операционных системах. Называется она **«функции обратного вызова»**. Функции обратного вызова – это такие функции, выполнение которых можно связать с происхождением какого–то события в операционной системе, например, удалением файла, истечением какого–то времени, нажатию на кнопку и так далее. То есть, формально эти функции

можно определить вот как: когда в системе произойдет вот такое событие, должна быть вызвана вот эта функция с такими аргументами. Таким образом, система находится все в том же потоке исполнения программы и не требует его смены, а, соответственно, ресурсов и времени на это.

Разработчикам стало понятно, что это может решить проблему большого количества соединений и асинхронный подход начали широко использовать при разработке высоконагруженных серверов.

Для каждого соединения создаётся контекст: некую структуру данных, которая будет хранить состояние соединения, информацию о том, что сейчас происходит в этом соединении и так далее. Действия же с соединением (чтение, запись, разрыв и так далее) происходят только тогда, когда соединение к этому готово, то есть когда происходит конкретное событие. На это событие назначается функция обратного вызова. Такой подход позволяет избежать ожидания, то есть не нужно постоянно «стоять и смотреть», произошло ли событие в этом соединении или еще нет.

Существует множество способов определить, к каким действиям готово соединение. К ним относятся такие методы как select, poll, kqueue, epoll и другие. Один из самых эффективных – linux epoll [21;22]. Его мы и будем использовать в дальнейшем при разработке сервера.

Формально, асинхронный подход позволяет основному исполняющему потоку программы сохранить свое состояние при выполнении долгосрочной операции, и пока она выполняется, основной поток занят выполнением других задач, а по окончанию выполнения долговременной операции, поток возвращается в то же место и продолжает выполнение конкретной задачи.

Таким образом, в 2004 году был разработан веб–сервер Nginx [19]. На тот момент, Nginx был единственным сервером, решившим «проблему 10 000 соединений» (обработка 10 000 соединений одновременно), благодаря реализации архитектуры, основанной на асинхронных вызовах. На сегодняшний день он широко применяется при разработке высоконагруженных веб–ресурсов, которые справляются со своей нагрузкой

более эффективно, чем серверы, в которых используется многопоточный подход к обработке соединений.

К сожалению, у такого подхода есть и минусы – сложность разработки и более сложный для понимания программный код. То есть, при разработке многопоточного сервера схема взаимодействия сервера с клиентом более очевидна и последовательна, чем при применении асинхронного подхода. Так же, применение асинхронного подхода требует использования особых конструкций, характерных для конкретного языка программирования, в программном коде, что усложняет читабельность и последовательность программного кода.

§5. Выбор подхода для разработки серверной части приложения и обоснование

В данной научной работе будет использоваться асинхронный подход для разработки серверной части приложения, так как этот подход, по определению цели своего создания, является более эффективным, не смотря на трудности его применения.

§6. Обзор системных вызовов для мультиплексированного ввода/вывода в операционных системах select, poll и epoll

Как было сказано выше, существует несколько способов наблюдения за состояниями сразу нескольких файловых дескрипторов (которые могут быть файлами, устройствами или сокетами, в том числе сетевыми), для того, чтобы узнать, готово ли устройство для продолжения ввода (вывода).

Select/Poll

Для использования select или poll приложение должно передать в ядро полный список всех файловых дескрипторов, в которых оно ожидает появление данных; а ядро, в свою очередь, должно для каждого из переданных элементов проверить состояние дескрипторов и сформировать структуру, описывающую состояние каждого переданного дескриптора. Такой подход не создаст много проблем в условиях десятков или сотен дескрипторов. Тем не менее, производительность в больших сетях заметно падает. Сложность данного алгоритма O(n).

Epoll в Linux (Kqueue в FreeBSD)

Linux предоставляет следующие вызовы в рамках API для использования epoll:

- **epoll_create() –** создаёт структуру данных (epoll instance), с которой в дальнейшем идёт работа. Структура одна для всех файловых дескрипторов, за которыми идёт наблюдение. Функция возвращает файловый дескриптор, который в дальнейшем передаётся во все остальные вызовы epoll API.
- **epoll_ctl() –** используется для управления epoll instance, в частности, позволяет выполнять операции EPOLL_CTL_ADD (добавление файлового дескриптора к наблюдению), EPOLL_CTL_DEL (удаление файлового дескриптора из наблюдения), EPOLL_CTL_MOD (изменение параметров наблюдения), EPOLL_CTL_DISABLE – для безопасного отключения наблюдения за файловым дескриптором в многопоточных приложениях
- **epoll_wait() –** возвращает количество (один или более) файловых дескрипторов из списка наблюдения, у которых поменялось состояние (которые готовы к вводу–выводу).

Принцип работы данного способа обнаружения изменений в файловых дескрипторов следующий: после того, как приложение добавляет дескрипторы к наблюдению и вызывает epoll_wait(), при готовности какого–либо дескриптора (появлению информации, опустошению буфера и т. д.) из epoll_wait ядро возвращает приложению список файловых дескрипторов, которые готовы к работе. Если какие–то дескрипторы становятся готовыми к работе до вызова epoll_wait, то они отмечаются соответствующим образом и при вызове epoll_wait возвращается список готовых файловых дескрипторов, включающий вышеупомянутые. Сложность данного алгоритма O(1), поэтому данный способ приобрел большую популярность при разработке высоконагруженных серверов.

Список событий, за которыми можно наблюдать с помощью epoll:

- **EPOLLIN** – новые данные (для чтения) в файловом дескрипторе
- **EPOLLOUT** – файловый дескриптор готов продолжить принимать данные (для записи)
- **EPOLLERR** – в файловом дескрипторе произошла ошибка
- **EPOLLHUP** – закрытие файлового дескриптора

ГЛАВА 3. СРЕДСТВА РЕАЛИЗАЦИИ СЕРВЕРНОЙ ЧАСТИ ПРИЛОЖЕНИЯ

В данной главе будет произведен обзор возможных средств реализации серверной части разрабатываемого приложения и, в конце главы, будет сделан выбор одного из них для дальнейшей разработки.

Существует множество языков программирования и дополнительных к ним библиотек, используя которые можно реализовать серверную часть приложения с необходимым функционалом. Рассмотрим несколько из них.

§1. C# и .NET Framework

C# – это C–подобный объектно–ориентированный язык программирования приложений, который использует программную платформу .NET Framework для разработки приложений, работающих под операционной системой Windows.

.NET Framework представляет собой сборку большого количества программных библиотек, разработанных и поддерживаемых компанией Microsoft, которые значительно расширяют возможности языка программирования, который ее использует, и упрощает написание программного кода.

В качестве примера будет разобран простой асинхронный «эхо» сервер на языке C#. Сервер будет построен с помощью асинхронного сокета, поэтому выполнение серверного приложения не приостанавливается, пока оно дожидается подключения клиента. Приложение получает строку от клиента, выводит ее на консоль и отправляет обратно клиенту. Строка от клиента должна содержать строку "<EOF>" для обозначения конца сообщения.

Код программы:

```
// Подключение необходимых библиотек из .NET Framework
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Threading;

// Объект состояния для асинхронного получения данных. В нем будем сохранять
состояние, чтобы по завершению считывания данных восстановить состояние
public class StateObject
{
   // Сокет клиента
   public Socket workSocket = null;
   // Размер буфера
   public const int BufferSize = 1024;
   // Получаем буфер
   public byte[] buffer = new byte[BufferSize];
   // Полученная строка
   public StringBuilder sb = new StringBuilder();
}

public class AsynchronousSocketListener
{
   // Сигнал для основного потока
   public static ManualResetEvent allDone = new ManualResetEvent(false);

   public AsynchronousSocketListener()
   {
   }

   public static void StartListening()
   {
      // Буер для входящих данных
      byte[] bytes = new Byte[1024];

      // Устанавливаем имя хоста. Устанавливаем IP–адрес. Создаем конечную точку для
подключения, дополнительно указав порт.
      IPHostEntry ipHostInfo = Dns.Resolve(Dns.GetHostName());
      IPAddress ipAddress = ipHostInfo.AddressList[0];
      IPEndPoint localEndPoint = new IPEndPoint(ipAddress, 11000);

      // Создание TCP/IP сокета
      Socket listener = new Socket(AddressFamily.InterNetwork,
         SocketType.Stream, ProtocolType.Tcp);

      // Привязка сокета к локальной конечной точке соединения и запуск «прослушки»
сокета
      try
      {
         listener.Bind(localEndPoint);
```

```
        listener.Listen(100);

        while (true)
        {
            // Установка события в начальное состояние
            allDone.Reset();

            // Запуск асинхронного слушающего сокета на прослушку соединений
            Console.WriteLine("Waiting for a connection...");
            listener.BeginAccept(
                new AsyncCallback(AcceptCallback),
                listener);

            // Ждем до того момента, когда соединение будет установлено
            allDone.WaitOne();
        }

    }
    catch (Exception e)
    {
        Console.WriteLine(e.ToString());
    }

    Console.WriteLine("/nPress ENTER to continue...");
    Console.Read();

}

public static void AcceptCallback(IAsyncResult ar)
{
    // Сигнал для продолжения главного потока программы
    allDone.Set();

    // Получаем сокет, который обработал запрос клиента
    Socket listener = (Socket)ar.AsyncState;
    Socket handler = listener.EndAccept(ar);

    // Создаем объект состояния и сохраняем в него состояие сокета
    StateObject state = new StateObject();
    state.workSocket = handler;
    handler.BeginReceive(state.buffer, 0, StateObject.BufferSize, 0,
        new AsyncCallback(ReadCallback), state);
}

public static void ReadCallback(IAsyncResult ar)
{
    String content = String.Empty;

    // Восстановление объекта состояния и сокета–обработчика
    // из асинхронного состояния объекта
    StateObject state = (StateObject)ar.AsyncState;
    Socket handler = state.workSocket;
```

```
        // Чтение данных из буфера
        int bytesRead = handler.EndReceive(ar);

        if (bytesRead > 0)
        {
            // Здесь может быть больше данных, чем предполагалось,
            // поэтому дозаписываем данные
            state.sb.Append(Encoding.ASCII.GetString(
                state.buffer, 0, bytesRead));

            // Проверяем, есть ли в полученных данных тэг конца файла. Если нет – читаем
данные дальше
            content = state.sb.ToString();
            if (content.IndexOf("<EOF>") > -1)
            {
                // Все данные были прочитаны от клиента. Выведем сообщение на консоль
                Console.WriteLine("Read {0} bytes from socket. /n Data : {1}",
                    content.Length, content);
                // Вернем полученные данные обратно клиенту
                Send(handler, content);
            }
            else
            {
                // Не все данные прочитанные. Считываем больше.
                handler.BeginReceive(state.buffer, 0, StateObject.BufferSize, 0,
                new AsyncCallback(ReadCallback), state);
            }
        }
    }

    private static void Send(Socket handler, String data)
    {
        // Преобразуем строку в массив байт
        byte[] byteData = Encoding.ASCII.GetBytes(data);

        // Начинаем отправку байт удаленному устройству
        handler.BeginSend(byteData, 0, byteData.Length, 0,
            new AsyncCallback(SendCallback), handler);
    }

    private static void SendCallback(IAsyncResult ar)
    {
        try
        {
            // Восстанавливаем сокет из обекта состояния
            Socket handler = (Socket)ar.AsyncState;

            // Передача данных удаленному устройству завершена
            int bytesSent = handler.EndSend(ar);
            Console.WriteLine("Sent {0} bytes to client.", bytesSent);
```

```
            handler.Shutdown(SocketShutdown.Both);
            handler.Close();

        }
        catch (Exception e)
        {
            Console.WriteLine(e.ToString());
        }
    }

    public static int Main(String[] args)
    {
        // Запуск сервера
        StartListening();
        return 0;
    }
}
```

Как можно видеть, архитектура программного кода простого асинхронного «эхо» сервера на языке C# довольна проста. Отсюда следует предполагать, что при правильном подходе к масштабированию приложения, программный код будет сохранять свою читабельность и простоту. Однако в Главе 1 к серверу было выдвинуто требование «кроссплатформенности», а, как известно, приложения, написанные на языке C# с использованием библиотек из .NET Framework способны работать только под управлением операционной системы Windows. Поэтому, не смотря на весомые достоинства данного языка программирования, мы не можем использовать его для разработки кроссплатформенного асинхронного FTP–сервера.

§2. C++ и Boost.Asio

Чтобы решить проблему «кроссплатформенности» сервера, можно использовать более низкий язык программирования **C++** и дополнительную к ней библиотеку Boost, которая имеет в себе сборку модулей, значительно расширяющих и упрощающих разработку приложений на C++. В данной библиотеке присутствует модуль **Asio (Asynchronous Input/Output)**, который позволяет разрабатывать асинхронные приложения.

Рассматривая данный вариант как средства реализации асинхронного FTP–сервера, мы не будем приводить здесь пример программного кода простейшего асинхронного сервера, так как он слишком громоздкий [13]. В указанном источнике рассматриваются синхронный и асинхронный подход при разработке приложений на C++, а также приводятся конкретные примеры программного кода.

Данное средство реализации обладает и качеством возможности создания асинхронного FTP–сервера, и качеством кроссплатформенности, однако, если читатель ознакомился с приведенным выше веб–ресурсом, то можно было заметить, что при использовании языка C++ в связке с библиотекой Boost и ее модулем Asio, запутанность и объем требуемого программного кода возрастает, что может плохо отразиться при масштабировании разрабатываемого приложения. По этим причинам, данное средство реализации при разработке FTP–сервера использоваться не будет.

§3. Python 3 и Asyncio

Python – высокоуровневый интерпретируемый язык программирования общего назначения, ориентированный на повышение производительности разработчика и читаемости кода. Синтаксис ядра Python минималистичен. В то же время стандартная библиотека включает большой объем полезных функций. Это активно развивающийся язык программирования, новые версии (с добавлением/изменением языковых свойств) выходят примерно раз в два с половиной года.

На сегодняшний день Python разработчики очень востребованы на рынке, что говорит о способности конкурировать с такими «гигантами», как C++, C#, Java и другими.

Стандартная библиотека включает в себя множество полезных модулей, в числе которых присутствует модуль Asyncio(Asynchronous Input/Output), который позволяет довольно просто разрабатывать асинхронные приложения

на языке Python. Для демонстрации простоты использования данного языка, ниже будет приведен пример программного кода все того же «эхо» сервера, только в этот раз на языке программирования Python.

Код программы:

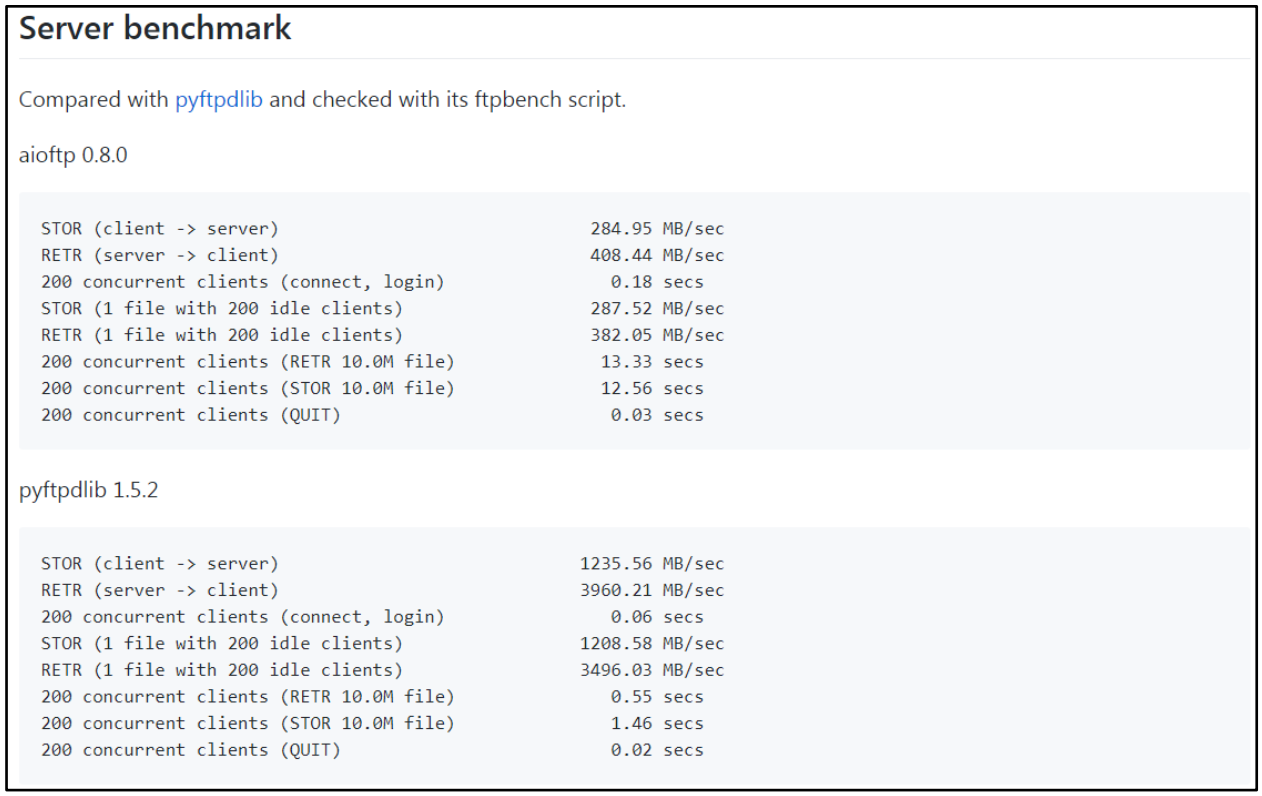
```
# Подключаем модуль asyncio
import asyncio
# Объявляем функцию, которая может работать асинхронно – сопрограмму(async)
# Это функция – обработчик подключения нового клиента
# Два параметра – поток считывания из буфера и поток ввода из буфера данных
async def handle_echo(reader, writer):
    # Асинхронно считываем данные из буфера в переменную data
    data = await reader.read(100)
    # Декодируем массив байт, чтобы получить строку
    message = data.decode()
    # Получаем адрес клиента
    addr = writer.get_extra_info('peername')
    # Выводим информацию о подключенном клиенте в консоль
    print("Received %r from %r" % (message, addr))

    print("Send: %r" % message)
    # Записываем данные обратно в буфер
    writer.write(data)
    # Если данных для буфера много – запись будет асинхронной
    await writer.drain()

    print("Close the client socket")
    # Закрываем поток записи
    writer.close()

# Создаем стандартный цикл событий
loop = asyncio.get_event_loop()
# Создаем специальную сопрограмму для работы сервера и задаем ему адрес
coro = asyncio.start_server(handle_echo, '127.0.0.1', 8888, loop=loop)
# Запускаем цикл событий
server = loop.run_until_complete(coro)

# Обрабатываем запросы, пока не получим сигнал завершения работы (Ctrl + C)
print('Serving on {}'.format(server.sockets[0].getsockname()))
try:
    loop.run_forever()
except KeyboardInterrupt:
    pass

# Закрытие сервера и слушающего сокета
server.close()
loop.run_until_complete(server.wait_closed())
loop.close()
```

Как можно видеть в примере выше, код простого асинхронного «эхо» сервера предельно наглядный и понятный. Если методология работы примера недостаточно ясна, то хорошим советом будет рекомендация к повторному прочтению введения данной научной работы, а также ознакомлению с различными примерами асинхронных серверов в глобальной сети Интернет [14;15].

Очевидными достоинствами данного средства реализации являются простота разработки, «кроссплатформенность» и конкурентоспособность разрабатываемого приложения среди схожих реализаций на других языках программирования.

Учитывая все написанное выше, было принято решение разрабатывать серверную часть приложения именно на языке Python с использованием модуля asyncio. Приняв это решение, были рассмотрены уже готовые различные реализации FTP–серверов на языке Python с использованием модуля asyncio в виде отдельных модулей, таких как: aioftp [10] , pyftpdlib [9].

По данным тестов разработчиков модуля aioftp, их реализация FTP–сервера уступает в производительности серверу, с использованием модуля pyftpdlib в несколько раз. Результаты тестов приведены на картинке ниже и доступны для сравнения:



Server benchmark

Compared with pyftpdlib and checked with its ftpbench script.

aioftp 0.8.0

```
STOR (client -> server)                             284.95 MB/sec
RETR (server -> client)                             408.44 MB/sec
200 concurrent clients (connect, login)               0.18 secs
STOR (1 file with 200 idle clients)                 287.52 MB/sec
RETR (1 file with 200 idle clients)                 382.05 MB/sec
200 concurrent clients (RETR 10.0M file)             13.33 secs
200 concurrent clients (STOR 10.0M file)             12.56 secs
200 concurrent clients (QUIT)                         0.03 secs
```

pyftpdlib 1.5.2

```
STOR (client -> server)                            1235.56 MB/sec
RETR (server -> client)                            3960.21 MB/sec
200 concurrent clients (connect, login)               0.06 secs
STOR (1 file with 200 idle clients)                1208.58 MB/sec
RETR (1 file with 200 idle clients)                3496.03 MB/sec
200 concurrent clients (RETR 10.0M file)              0.55 secs
200 concurrent clients (STOR 10.0M file)              1.46 secs
200 concurrent clients (QUIT)                         0.02 secs
```

Рисунок 4: сравнение производительности FTP-серверов на основе aioftp 0.8.0 и pyftpdlib 1.5.2

Согласно официальной документации модуля pyftpdlib [9], данный модуль обладает всеми необходимыми инструментами для создания высокопроизводительного асинхронного FTP–сервера, который можно конфигурировать необходимым для разработчика образом. Если же функционала данного модуля становится недостаточно, то эту проблему можно решить путем написания собственных классов, которые должны наследоваться от базовых классов. Методы новых классов будут обрабатывать события на сервере необходимым для разработчика образом.

Установить данный модуль можно используя утилиту pip, которая устанавливается вместе с интерпретатором Python командой: pip install pyftpdlib.

Данный модуль используется компанией Google в браузере Chromium для скачивания файлов из глобальной сети Интернет, а также платформой контроля версий Bazaar для обмена файлами с исходным кодом между

разработчиками и самой платформой, что говорит о хорошей репутации и надежности модуля.

В связи с вышеописанными фактами, было принято решение разрабатывать асинхронный FTP–сервер на базе данного модуля, корректируя и преобразуя его под требования, которые были выделены в Главе 1.

§4. Модуль Pyftpdlib. Краткий обзор

Как было сказано ранее, **pyftpdlib** – это модуль Python, который реализует очень быстрый асинхронный FTP–сервер на языке Python с использованием библиотеки ядра Python – asyncio. Скорость его работы обусловлена использованием вышеупомянутого способа обнаружения изменений в файловых дескрипторах, а именно epoll, а также простой и понятной структурой программного кода. Если в операционной системе не предоставляется возможным использование epoll или kqueue (в случае использования ОС FreeBSD), то используются такие способы как select и poll. Кратко рассмотрим структуру и функциональность модуля, так как именно он будет использоваться при разработке серверной части приложения, после чего перейдем к самой реализации FTP–сервера, требования к которому были представлены в Главе 2.

Иерархия классов в модуле pyftpdlib выглядит следующим образом:

```
pyftpdlib.authorizers
pyftpdlib.authorizers.AuthenticationFailed
pyftpdlib.authorizers.DummyAuthorizer
pyftpdlib.authorizers.UnixAuthorizer
pyftpdlib.authorizers.WindowsAuthorizer
pyftpdlib.handlers
pyftpdlib.handlers.FTPHandler
pyftpdlib.handlers.TLS_FTPHandler
pyftpdlib.handlers.DTPHandler
pyftpdlib.handlers.TLS_DTPHandler
pyftpdlib.handlers.ThrottledDTPHandler
pyftpdlib.filesystems
pyftpdlib.filesystems.FilesystemError
```

```
pyftpdlib.filesystems.AbstractedFS
pyftpdlib.filesystems.UnixFilesystem
pyftpdlib.servers
pyftpdlib.servers.FTPServer
pyftpdlib.servers.ThreadedFTPServer
pyftpdlib.servers.MultiprocessFTPServer
pyftpdlib.ioloop
pyftpdlib.ioloop.IOLoop
pyftpdlib.ioloop.Connector
pyftpdlib.ioloop.Acceptor
pyftpdlib.ioloop.AsyncChat
```

Как можно видеть, в иерархии присутствует 5 базовых классов, от которых наследуются классы, реализующие различный функционал. Чтобы суть излагаемого была лучше ясна, рассмотрим предназначение каждого из классов.

Класс **authorizers** – базовый класс, который содержит в себе методы для авторизации клиента при подключении. Он используется внутри класса pyftpdlib.handlers.FTPHandler для проверки пароля пользователя, получения домашней директории пользователей, проверки прав пользователя при возникновении события чтения / записи файловой системы и смены пользователя перед доступом к файловой системе.

AuthenticationFailed (не является классом – наследником authorizers) – исключение, возникающее при неправильном вводе логина и пароля при подключении.

Классы–наследники:

- DummyAuthorizer – является базовым «авторизатором», предоставляющим платформенно–независимый интерфейс для управления «виртуальными» пользователями на FTP–сервере. То есть пользователи операционной системы, на которой запущен сервер, никак не используются, а используются «свои» виртуальные пользователи на сервере, которым так же можно задавать необходимые права, добавлять их и удалять программно.

- UnixAuthorizer – авторизатор, который взаимодействует с базой данных паролей UNIX. Пользователи больше не должны быть явно добавлены, как при использовании pyftpdlib.authorizers.DummyAuthorizer. Все пользователи FTP такие же, как в системе UNIX, поэтому, если вы получаете доступ к своей системе, используя «john» в качестве имени пользователя и «12345» в качестве пароля, те же учетные данные могут использоваться для доступа к FTP–серверу. Домашние каталоги пользователей будут автоматически определяться при входе пользователя в систему.
- WindowsAuthorizer – аналогично предыдущему классу, только для ОС Windows.

Можно написать свой класс–наследник, который будет обрабатывать информацию и работу с пользователями надлежащим образом.

Далее следует базовый класс **handlers**, который обрабатывает события на сервере соответствующим методам класса–наследника образом. Объект данного класса связывается с классом authorizers следующим образом:

```
from pyftpdlib.handlers import FTPHandler
handler = FTPHandler
handler.authorizer = authorizer
```

Классы–наследники:

- FTPHandler – этот класс реализует интерпретатор протокола FTP–сервера (см. RFC–959), обрабатывая команды, полученные от клиента по каналу управления, вызывая соответствующий метод команды (например, для принятой команды «путь MKD», метод ftp_MKD () вызывается с именем пути в качестве аргумента). Вся соответствующая информация сеанса хранится в переменных экземпляра. conn – это базовый экземпляр объекта сокета только

что созданного соединения, сервер – экземпляр класса pyftpdlib.servers.FTPServer.

- DTPHandler – этот класс обрабатывает процесс передачи данных сервера (сервер–DTP, см. RFC–959), который управляет всеми процессами передачи в отношении канала данных. sock_obj – это основной экземпляр объекта сокета только что установленного соединения, cmd_channel – это экземпляр класса pyftpdlib.handlers.FTPHandler.
- ThrottledDTPHandler – класс–наследник pyftpdlib.handlers.DTPHandler, который обертывает отправку и прием в счетчике данных и временно «спит», чтобы вы передавалось не более чем X Кб в секунду. Его следует использовать как DTPHandler, только с ограничением скорости получения и отдачи информации.

Остальные классы–наследники используют криптографические алгоритмы для шифрования отправляемых данных. Так же можно реализовать свой класс–наследник.

Класс **filesystems** – базовый класс, который описывает работу с различными операционными системами (то есть, проверка существования файла, получение списка файлов в директории и так далее). В целом, написание своего класса–наследника необходимо только в том случае, если используется нестандартная операционная система.

Классы–наследники:

FilesystemError – исключение, которое возникает при некорректной работе с ОС. Используется для обертывания ошибок и удобного их представления пользователю.

AbstractedFS – класс–наследник, используемый для взаимодействия с файловой системой, обеспечивающий кроссплатформенный интерфейс, совместимый как с файловыми системами в стиле Windows, так и с UNIX, где

все пути используют разделитель «/». AbstractedFS различает «реальные» пути файловой системы и «виртуальные» пути FTP, имитирующие замкнутую директорию chroot UNIX, где пользователь не может покинуть свой домашний каталог (пример: реальный путь «/home/user» будет рассматриваться как «/» клиентом). Он также предоставляет некоторые методы утилиты и обертывает все вызовы os.*, включающие операции с файловой системой, такие как создание файлов или удаление каталогов. Конструктор принимает два аргумента: root, который является «реальным» домашним каталогом пользователя (например, «/home/user») и cmd_channel, который является экземпляром класса pyftpdlib.handlers.FTPHandler.

UnixFilesystem – представляет реальную файловую систему UNIX. В отличие от pyftpdlib.filesystems.AbstractedFS клиент войдет в систему /home/<имя_пользователя> и сможет покинуть свой домашний каталог и перейти к реальной файловой системе.

Базовый класс **server** реализует функционал серверного приложения, то есть прослушивание определенного порта, принятие и работа с подключениями.

Классы–наследники:

- FTPServer – класс–наследник, который создает сокет, прослушивающий по адресу (хост, порт) или использует уже существующий объект сокета, отправляя запросы обработчику (обычно это класс pyftpdlib.handlers.FTPHandler). Также запускается асинхронный цикл ввода–вывода. backlog – максимальное количество подключенных к очереди соединений, переданных в socket.listen (). Если запрос на соединение приходит, когда очередь заполнена, клиент может получить исключение ECONNRESET.

- ThreadedFTPServer – измененная версия базового класса pyftpdlib.servers.FTPServer, которая генерирует поток каждый раз, когда устанавливается новое соединение. В отличие от базового класса FTPServer, обработчик свободен от блокировки и не останавливает весь цикл ввода–вывода.
- MultiprocessFTPServer – измененная версия базового класса pyftpdlib.servers.FTPServer, которая запускает процесс каждый раз, когда устанавливается новое соединение. В отличие от базового класса FTPServer, обработчик свободен от блокировки и не останавливает весь цикл ввода–вывода.

Базовый класс **ioloop** и его наследники реализуют разные виды циклов событий. Они отличаются своей работой, однако разработчики рекомендуют использовать стандартный IOLoop, так как он подходит для решения большинства задач.

§5. Простой пример асинхронного FTP–сервера pyftpdlib

Рассмотрим простой пример асинхронного FTP–сервера, код которого представлен ниже:

```
import os

from pyftpdlib.authorizers import DummyAuthorizer
from pyftpdlib.handlers import FTPHandler
from pyftpdlib.servers import FTPServer

def main():
    # Инициализируем объект класса ‘DummyAuthorizer’ для работы с виртуальными пользователями
    authorizer = DummyAuthorizer()

    # Определяем пользователя с полными правами и анонимного пользователя с доступом на чтение
    authorizer.add_user('user', '12345', '.', perm='elradfmwMT')
    authorizer.add_anonymous(os.getcwd())
```

```
    # Инициализируем объект класса FTPHandler и связываем его с объектом класса–
авторизатора
    handler = FTPHandler
    handler.authorizer = authorizer

    # Устанавливаем сообщение приветствия
    handler.banner = "pyftpdlib based ftpd ready."

    # Раскомментировать, если используется NAT(задание маски и диапазона портов)
    #handler.masquerade_address = '151.25.42.11'
    #handler.passive_ports = range(60000, 65535)

    # Задаем адрес конечной точки подключения к серверу
    address = ('', 2121)
    server = FTPServer(address, handler)

    # Устанавливаем лимит подключений
    server.max_cons = 256
    server.max_cons_per_ip = 5

    # Запускаем сервер
    server.serve_forever()

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

Если запустить данный код с помощью интерпретатора Python, то на компьютере будет запущен асинхронный FTP–сервер, который «слушает» подключения на порту 2121. Подключиться к данному серверу сможет анонимный пользователь с правами только на чтение, и пользователь «user» с паролем «12345» и полными правами на чтение и запись.

В следующей главе будет рассмотрена реализация асинхронной серверной части приложения с использованием модуля pyftpdlib, который соответствует требованиям, определенным в Главе 1, параграфе 2.

ГЛАВА 4. РЕАЛИЗАЦИЯ СЕРВЕРНОЙ ЧАСТИ ПРИЛОЖЕНИЯ

В настоящей главе будет приведено описание реализованной серверной части приложения (FTP–сервера), а также программный код с пояснениями.

Разработанный сервер должен принимать подключения от пользователей и устанавливать с ними командное соединение. Затем, сервер должен отправлять соответствующий ответ по данному соединению и, в зависимости от полученной по данному соединению команды от клиента, устанавливать или закрывать информационное соединение, по которому производится приём или передача файла. Список команд и ответов, реализуемых сервером, предусмотрен стандартом RFC-959 [26].

Как было сказано в предыдущей главе, для реализации серверной части приложения будет использоваться язык Python 3, а также модуль pyftpdlib.

§1. Выбор компонентов модуля pyftpdlib для реализации сервера

Необходимо, чтобы сервер вёл учёт пользователей, причём их данные не должны быть привязаны к пользователям операционной системы, поэтому компоненты UnixAuthorizer и WindowsAuthorizer не подходят. Для достижения данной цели использовался компонент DummyAuthorizer, позволяющий вести учёт пользователей в рамках работающего FTP–сервера.

Далее, обработчик принятых команд мы будем использовать стандартный для данного модуля, а именно FTPHandler, так как ThrottledDTPHandler используется для ограничения скорости передачи и отдачи файлов для всех пользователей, что не подходит для разрабатываемого приложения.

Для создания сервера будет использоваться компонент FTPServer модуля pyftpdlib, так как этот компонент создаёт FTP–сервер, использующий именно асинхронный подход для обработки соединений, а не многопоточный

или многопроцессный, как это делают компоненты ThreadedFTPServer и MultiprocessFTPServer соответственно.

Также необходимо предоставить администратору возможность добавлять и удалять пользователей на сервере. Для этого будет реализован бесконечный цикл, “вечно слушающий” ввод с терминала в отдельном потоке исполнения программы, который позволит администратору через консоль совершать подобные действия.

§2. Программный код сервера с пояснениями

Так как программный код данного сервера не занимает более 100 строчек кода (благодаря модулю pyftdlib), автор данной научной работы позволил себе вставить его в текст данной научной работы. Программный код сервера представлен ниже.

Код программы:

```
#!/usr/bin/env python
# -*--encoding:utf8 --*--
import os
from sys import stdin
import errno
from pyftpdlib.handlers import FTPHandler # подключение класса обработчиков
from pyftpdlib.servers import FTPServer # подключение класса сервера
from pyftpdlib.authorizers import DummyAuthorizer # подключение класса авторизации
import logging # подключение модуля для логирования
import threading

def listen_commands(authorizer, handler, server): # цикл, слушающий ввод с терминала для
#добавления и удаления пользователей администратором
    while True:
        print(" – – – ===MENU OF COMMANDS=== – – – ")
        print("1. Add user")
        print("2. Remove user")
        choice = str(input())
        if choice == "1":
            print(" – – Enter login of the new user:")
            login = str(input())
            # если такой логин уже есть, то при добавлении попадём в except
            print(" – – Enter password of the new user:")
            password = str(input())
            print(" – – Enter rights for a new user (example: elrad):")
            print(" – Read permissions – ")
```

```
            print("'e' = change directory (CWD, CDUP commands)")
            print("'l' = list files (LIST, NLST, STAT, MLSD, MLST, SIZE commands)")
            print("'r' = retrieve file from the server (RETR command)")
            print(" – Write permissions – ")
            print("'a' = append data to an existing file (APPE command)")
            print("'d' = delete file or directory (DELE, RMD commands)")
            print("'f' = rename file or directory (RNFR, RNTO commands)")
            print("'m' = create directory (MKD command)")
            print("'w' = store a file to the server (STOR, STOU commands)")
            print("'M' = change file mode / permission (SITE CHMOD command)")
            print("'T' = change file modification time (SITE MFMT command)")
            permissions = str(input())
            try:
                add_user(login, password, permissions, os.path.join(os.getcwd(),'users_data/'),
authorizer)
                logging.info("Administrator added a new user with (login, permissions, homedir) as ("
+ login + ", " + permissions +", " + home_directory + ")")
            except ValueError as err:
                print("Error! " + err + " Try again...")
        if choice == '2':
            print(" –  – Enter login of the user, you want to remove:")
            remove_login = str(input())
            try:
                authorizer.remove_user(remove_login)
                logging.info("Administrator removed user with login :" + remove_login)
            except:
                print("Error occured when trying to remove user. Try again!")

# «обёрточная» функция для добавления новых пользовтаелей
# (добавляет нового пользователя, создаёт рабочую директорию пользователя)
def add_user(login, password, permissions, path_of_data_directory, authorizer):
    path_to_user_directory = os.path.join(path_of_data_directory, login)
    try:
        os.makedirs(path_to_user_directory)
    except OSError as e:
        if e.errno != errno.EEXIST:
            logging.error("Error occured when trying to create a new directory for user. Err text:" +
e.args[0])
            raise
    authorizer.add_user(login, password, homedir=path_to_user_directory, perm=permissions)
    logging.info("New user was added a new user with (login, permissions, homedir) as (" + login
+ ", " + permissions +", " + path_to_user_directory + ")")
    pass

def main(): # точка входа в приложение

    logging.basicConfig(filename = "ftp_server.log", level = logging.DEBUG, format =
'%(asctime)s %(levelname)s: %(message)s', datefmt = '%Y – %m – %d %H:%M:%S')

    try:
        os.makedirs(os.path.join(os.getcwd(),'users_data'))
        os.makedirs(os.path.join(os.getcwd(),'users_data/anonim'))
```

```
    except OSError as e:
        if e.errno != errno.EEXIST:
            logging.error("Error occured when trying to create a new directory for user. Err text:" +
e.args[0])
            raise

    authorizer = DummyAuthorizer()
    # раскомментировать строку ниже при необходимости добавления анонимного
пользователя
    # authorizer.add_anonymous(homedir='users_data/anonim')

    # конфигурируем сервер
    handler = FTPHandler # создание объектра обработки событий
    handler.authorizer = authorizer # присваивание объектра авторизации
    handler.passive_ports = range(10000, 65535)
    handler.banner = "Hello! I am Asynchronious Supercomputer FTP – server."
    handler.timeout = None # максимальный промежуток времени между коммандами
клиента

    serv = FTPServer(('0.0.0.0', 3100), handler) # создание объектра сервера
    serv.max_cons = 55535
    # включим прослушку команд в отдельном потоке
    command_thread = threading.Thread()
    command_thread._target = listen_commands
    command_thread._args = (authorizer, handler, serv)
    command_thread.start()
    serv.serve_forever() # запуск прослушивания порта в бесконечном цикле

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

ГЛАВА 5. МЕТОД РАЗДЕЛЕНИЯ ПОТОКА ДАННЫХ

В настоящей главе будет описана основная идея метода разделения потока данных между клиентом и сервером для приоритетной передачи файлов. Описанный ниже метод не является единственным, однако, он соответствует требованиям, описанным в Главе 2, а также соответствует серверной части приложения, реализация которой была представлена в предыдущей главе.

§1. Описание программного окружения

В предыдущей главе мы рассмотрели реализацию серверной части приложения. Полученный в результате сервер работает по принципу «делаю столько, сколько успеваю сделать». То есть, количество данных, которое может быть передано за единицу времени от сервера к клиенту и обратно не ограничивается определенной величиной, а ограничивается лишь производительностью хоста (компьютера, на котором этот сервер работает). Так же, ограничением может являться максимальная пропускная способность канала связи, однако данной ограничение мы не учитываем, так как у разработчика программного обеспечения отсутствуют методы воздействия на него. Мы же должны уловить основную суть, которая заключается в вопросе: «как, при наличии FTP–сервера, который в пределе (при стремлении своей производительности к бесконечности) за определенное количество времени может передавать и принимать большое количество информации, и при бесконечно «широком» канале связи, обеспечить такой алгоритм работы FTP–клиента, который позволил бы принимать и отсылать какие–то данные быстрее, чем другие?». При этом возможна ситуация, когда сервер будет не успевать обрабатывать все соединения и скорость обмена файлами будет снижаться. Это не должно влиять на надежность передачи данных.

Ответом на написанный выше вопрос и будет являться метод разделения потока данных, устройство которого будет описано в следующем параграфе.

§2. Описание метода разделения потока данных

Для точного понимания сути методов, которые будут описаны ниже, необходимо помнить об упомянутой выше специфике FTP–протокола (на один передающийся файл – два отдельных подключения). Под одной и той же учетной записью на FTP–сервере можно работать в сразу нескольких клиентах, при этом каждый запущенный клиент может передавать на сервер или скачивать свой отдельный файл. Так же уже было упомянуто, что сам метод будет реализовываться не на серверной, а на клиентской стороне, чтобы не обременять сервер лишними функциями. Сервер будет лишь принимать и отдавать файлы настолько быстро, насколько он может. То есть, когда клиент считал данные из буфера, сервер, по мере своей производительности (почти мгновенно, при маленьком количестве соединений), высылает клиенту еще одну порцию данных, и так же программа должна работать в обратную сторону.

Итак, формализуем задачу, а затем идею решения попытаемся распространить на все случаи.

Задача: с одного компьютера на другой по FTP–протоколу нужно передавать несколько файлов. Требования:

a) параллельно
b) эффективно использовать канал связи
c) в зависимости от приоритета передавать какие–то файлы быстрее, а какие–то медленнее

В соответствии с FTP–протоколом, по одному FTP–подключению (2 соединения – для команд и для данных) можно передавать или принимать

только один файл. Это значит, что для удовлетворения требования а) необходимо будет использовать сразу несколько FTP–подключений. Далее, было продумано два алгоритма разделения одного канала связи для передачи нескольких файлов параллельно с приоритетом – «по времени передачи» и «по кадрам передачи».

Метод разделения потока данных «по времени передачи»

Для того чтобы было проще понять алгоритм работы данного метода, разберем пример. Допустим, 3 файла скачиваются с сервера. У первого файла приоритет равен единице, у второго – трем, у третьего – шести. Можно взять определенный интервал времени, например, 1 секунда. Этот интервал времени будет означать один **«цикл передачи»**, то есть, цикл, за который из всех загружаемых/скачиваемых файлов будет принят или отправлен как минимум 1 байт информации. Разделив интервал на сумму приоритетов, мы получим временной промежуток, который требуется уделить для принятия/передачи файла на один приоритет. В данном примере, сумма приоритетов равна 1 + 3 + 6 = 10. 1с/10пр = 0.1 с/пр. То есть, для «погашения» одной единицы приоритета у любого файла необходимо уделить принимаемому или передаваемому файлу 0.1 секунду. Вернемся к примеру. На один «цикл передачи» будет приходиться 0.1 секунда приема первого файла, 0.3 секунды второго файла и 0.6 секунды 3 файла. Допустим, что пропускная способность канала связи 1 Мбит/сек. Это означает, что за одну секунду (один «цикл передачи») от первого файла будет принято 0.1 Мбита информации, от второго файла 0.3 Мбита информации, а от третьего – 0.6 Мбита информации. Такой метод разделения потока данных удовлетворяет всем требованиям, причем, при стремлении величины «цикла передачи» к нулю – эффект «параллелизма» загрузки и выгрузки файлов будет проявляться все сильнее.

Стоит отметить, что существует задача равномерного распределения ресурсов сервера между клиентами. То есть, зачастую каждый из клиентов имеет «свою» ширину (пропускную способность) канала связи с сервером. По этой причине один клиент может произвести большее количество «циклов передачи», чем другой (принять или передать большее количество информации серверу, чем другой клиент), то есть сервер будет «занят» только «сильными» клиентами (теми, у которых ширина канала связи с сервером наибольшая).

Одним из критериев, который должен выполняться для равномерного распределения ресурсов сервера является равенство «циклов передачи» у каждого из клиентов, то есть у каждого клиента должен быть задан одинаковый размер «цикла передачи» файлов. Однако, выполнения одного этого критерия недостаточно, ведь даже при одинаковом размере «цикла передачи» файлов среди клиентов один клиент может произвести большее количество «циклов передачи» файлов, чем другой, а при большой разнице этих величин, зачастую, когда суммарная величина пропускной способности каналов связи у клиентов превышает величину пропускной способности канала связи у сервера, самые «сильные» клиенты, могут просто «занять» сервер только обработкой своих подключений, что является показателем неравномерного распределения ресурсов сервера между всеми клиентами. Эту проблему решает алгоритм работы «цикла событий» (см. Главу 2, параграф 4). Его внутреннее устройство таково, что если поставленная задача (экземпляр функции получения или отправки данных) не выполняется долго, то ее приоритет выполнения среди остальных задач, поставленных в очередь выполнения, возрастает до тех пор, пока данная задача (экземпляр функции) не будет выполнена. Таким образом, когда «слабый» клиент конкурирует за ресурсы сервера с «сильным» клиентом, сервер не позволяет «сильному» клиенту заполучить все свои производительные ресурсы и, на сколько бы величина пропускной способности «сильного» клиента не была большой, сервер все–равно будет уделять часть своих ресурсов подключению со

«слабым» клиентом, то есть будет принимать и отправлять информацию «слабому» клиенту. Это означает, что сколько бы файлов каждый отдельный клиент не принимал и не передавал параллельно и сколь «широким» не был бы его канал связи с сервером – это не будет влиять на справедливость распределения потока информации среди подключенных к серверу клиентов.

К сожалению, метод разделения потока данных «по времени» перестает быть «приоритетно справедливым» по отношению к принимаемым/передаваемым файлам, когда ширина канала связи перестает быть константной величиной, что является довольно частым явлением на пользовательских ПК. Например, ширина канала связи первые половину секунды равна 1 Мбит/сек, а во вторую половину секунды она равна 3 Мбит/сек. Тогда от первого файла будет передано 0.1 Мбита информации, от второго 0.3 Мбита информации, а от третьего 1*0.1 + 3*0.5 = 1.6 Мбита информации. Если сравнивать пропорции размеров принятых кадров информации от каждого файла к их приоритетам, то равенство между ними исчезнет, а это значит, что приоритет, установленный пользователем, учитывается не полностью, что есть неправильно.

Проблему «приоритетной несправедливости» в методе разделения потока данных «по времени передачи», был разработан еще один метод разделения потока данных «по кадрам передачи».

Метод разделения потока данных «по кадрам передачи»

Принцип данного метода разделения потока данных схож с предыдущим, однако теперь «цикл передачи» будет измеряться не во времени, а в количестве информации, которая должна быть передана.

Рассмотрим данный метод на примере, который был приведен выше, только теперь «цикл передачи» будет равен не одной секунде, а 64 000 Байт. Скачиваются 3 файла. У первого приоритет – единица, у второго – приоритет равен трем, а у третьего – шести. Сумма приоритетов равна 10. Это значит,

что для «погашения» одной единицы приоритета на любом из файлов требуется принять/отправить 6400 Байт. Итак, за один «цикл передачи» необходимо принять кадр от первого файла размером 6400 Байт, кадр от второго файла размером 6400*3 = 19 200 Байт, и кадр от третьего файла размером 6400*6 = 38 400 Байт. Таким образом, данный метод будет соответствовать всем требованиям и будет «справедлив» даже тогда, когда скорость соединения клиента с сервером меняется на протяжении всего обмена файлами.

Блок–схема данного метода (алгоритма) разделения потока данных представлена на рисунке ниже.

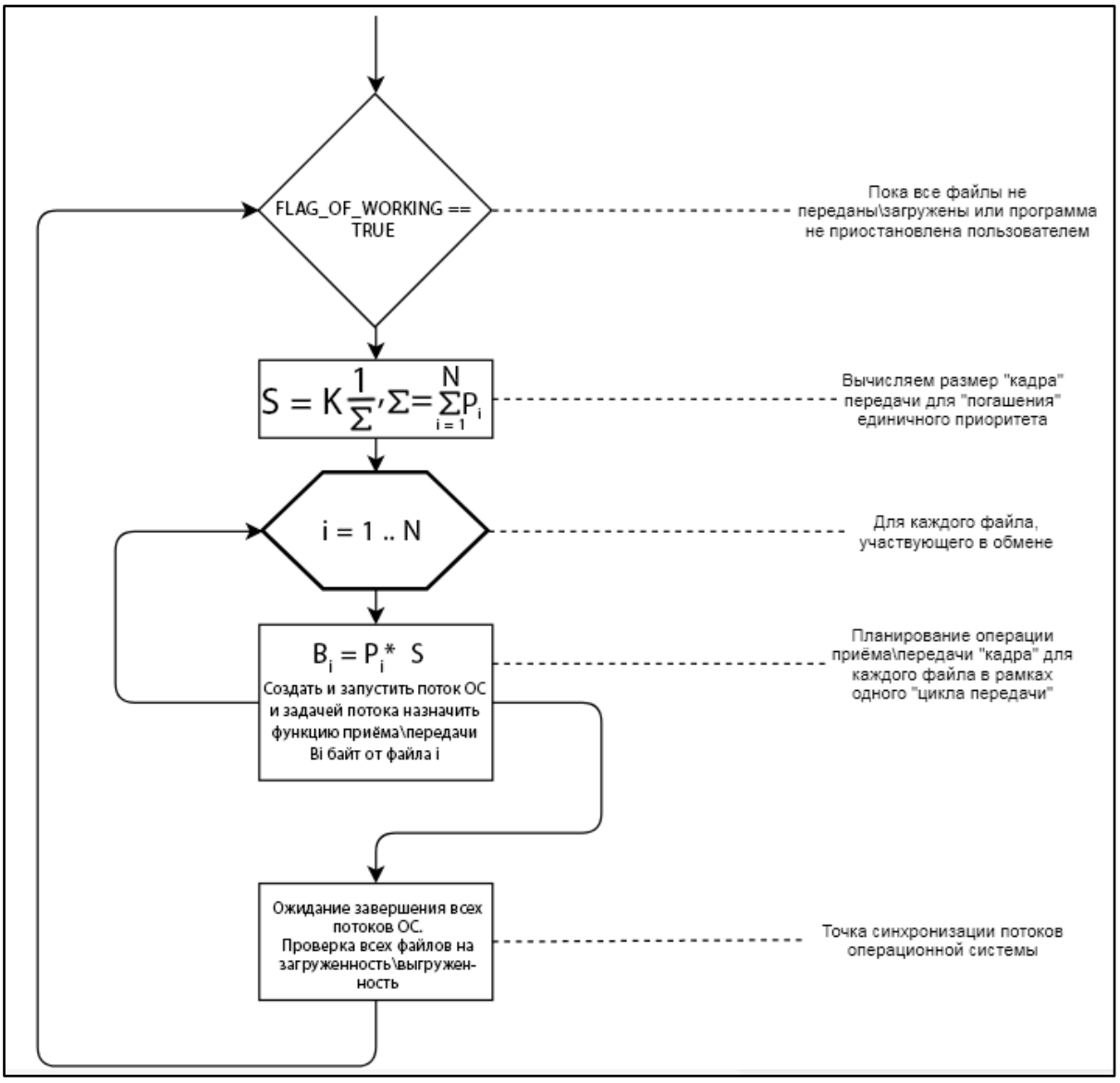


Рисунок 5: Блок-схема метода разделения потока «по блокам передачи»

Для лучшего понимания разберем еще один пример разделения потока данных с помощью метода разделения «по кадрам передачи» более подробно: Передается 3 файла, разных размеров. У первого файла приоритет – 3, у второго – 1, а у третьего – 4. Зададим размер цикла передачи 64000 байт. То есть, за один цикл передачи должно быть передано не более 64000 байт. Просуммируем приоритеты: 3+1+4 = 8. Разделим размер цикла передачи на сумму приоритетов, чтобы выяснить, сколько байт информации должно быть передано для «погашения» одной единицы приоритета. 64000/8 = 8000 байт.

Итак, 8000 байт требуется передать, чтобы «покрыть» для приоритета одну единицу приоритета. Следовательно:

- у первого файла приоритет равен 3, значит нужно в отдельном потоке (параллельно) передать 3*8000 = 24000 байт от первого файла;
- у второго файла приоритет равен 1, значит нужно в отдельном потоке (параллельно) передать 1*8000 = 8000 байт от второго файла;
- у третьего файла приоритет равен 4, значит нужно в отдельном потоке (параллельно) передать 4*8000 = 32000 байт от третьего файла.

Очевидно, что физически канал связи один (один провод или беспроводное соединение). Поэтому «разбиение» канала связи на несколько потоков происходит виртуально, то есть происходит имитация данного процесса операционной системой для разработчика. Операционная система в одном цикле передачи дает примерно одинаковый приоритет каждому из отдельных "файловых" потоков, следовательно, алгоритм передачи для случая выше можно описать следующим образом:

- Допустим, пропускная способность всего канала связи N байт/сек.
- Физический поток делится на 3 виртуальных файловых потока.

- Передались 8000 байт от каждого кадра файлов со скоростью N/3 байт/сек (скорость для каждого отдельного файлового потока), следовательно, передался полностью кадр от второго файла. Файловый поток №2 закрывается. Времени затрачено 8000/ (N/3) секунд.
- Теперь физический поток делиться на два файловых потока (для первого и третьего файла). Передалось еще 16000 байт от каждого кадра со скоростью N/2 байт/сек (скорость для каждого отдельного файлового потока), следовательно, передался полностью кадр от первого файла. Файловый поток №1 закрывается. Времени затрачено на это 16000/(N/2) секунд.
- Теперь физический поток не делится и его скорость N байт/сек. Осталось передать еще 8000 байт от кадра третьего файла. Файловый поток №3 закрывается. Времени затрачено на передачу 8000/N секунд.
- Цикл передачи завершен. Далее планируется новый цикл передачи, и если приоритеты у файлов не сменились и ни один файл не передан, то планируемый цикл передачи будет идентичен тому, что описан выше.

В итоге, за один кусок времени t были переданы кадры из разных файлов, разного размера. Следовательно, какие–то файлы будут загружаться быстрее, а какие–то медленнее, тем самым реализуя приоритет загрузок. Очевидно, что физический канал связи каждый момент времени используется на 100%, ведь никаких препятствий и ограничений не используется.

Вычислим время передачи кадры от каждого файла:

- Время, затраченное на передачу кадра от первого файла: 8000/(N/3) + 16000/(N/2) секунд;
- Время, затраченное на передачу кадра от второго файла: 8000/(N/3) секунд;

- Время, затраченное на передачу кадра от третьего файла: 8000/(N/3) + 16000/(N/2) + 8000/N секунд.

Допустим, N = 16 Мбит/сек = 2 Мбайт/сек = 2048 Кбайт/сек = 2 097 152 байт/сек. N/3 = 699 050 байт/сек. N/2 = 1 048 576 байт/сек. Подставим значение пропускной способности канала связи в формулу и вычислим конкретное время передачи кадра от каждого файла при заданной пропускной способности канала связи:

- Время передачи первого кадра: 0.0114 + 0.0152 = 0.0266 секунд;
- Время передачи второго кадра: 0.0114 секунд;
- Время передачи третьего кадра: 0.0114 + 0.0152 + 0.0038 = 0.030 секунд.

Отсюда можно посчитать, с какой скоростью передавался каждый из кадров:

- Скорость передачи кадра от 1 файла: 0.022888 Мегабайт/0.030 сек = 0.7629 Мегабайт/сек;
- Скорость передачи кадра от 2 файла: 0.0076 Мегабайт / 0.030 сек = 0.253 Мегабайт/сек;
- Скорость передачи кадра от 3 файла: 0.03051 Мегабайт / 0.030 сек = 1.017 Мегабайт/сек.

Сложим показатели скорости, чтобы проверить эффективность использования пропускной способности физического канала: 0.7629 + 0.253 + 1.017 = 2.0329 (с погрешностью вычислений) примерно равно 2 Мегабайта/сек, следовательно, канал используется на 100% и метод не является неэффективным.

ГЛАВА 6. СРЕДСТВА РЕАЛИЗАЦИИ КЛИЕНТСКОЙ ЧАСТИ ПРИЛОЖЕНИЯ

В данной главе будут рассмотрены возможные средства реализации клиентской части приложения, то есть реализации FTP–клиента, который будет использовать разработанный метод разделения потока данных, описанный в Главе 5, для обмена файлами с серверной частью приложения.

При реализации серверной части приложения нам не требовалось создавать графический интерфейс для пользователя, так как с сервером будет взаимодействовать не пользователь, а программы (клиенты), однако для клиентской части графический интерфейс необходим пользователю для удобного взаимодействия с программой–клиентом.

Существует множество языков программирования и библиотек, используя которые можно создать программу, выступающую в качестве FTP–клиента, имеющую графический интерфейс.

§1. C# и .NET Framework

Возможности языка программирования C# и программной платформой .NET Framework уже были рассмотрены в Главе 3 в параграфе 1, однако, следует отметить, что программная платформа .NET Framework так же представляет для разработчика возможным создание графического интерфейса для разрабатываемого приложения в удобном виде. То есть, например, в среде разработки Visual Studio даже присутствует графический редактор, как раз–таки решающий вопрос удобной разработки приложений с графическим интерфейсом.

.NET Framework представляет в себе несколько технологий для работы с оконными приложениями. Туда входят Widows Forms [16], Windows Presentation Foundation [17], Universal Windows Platform и другие.

Однако, вышеперечисленные технологии создания оконных приложений, какими бы удобными они не были, не подходят для реализации поставленной задачи. Дело в том, что итоговое приложение, разработанное с использованием этих технологий, будет работать только под операционной системой Windows, что не соответствует требованию кроссплатформенности к клиентской части приложения, описанному в Главе 2. Поэтому данное средство реализации в нашем случае не подходит для использования.

§2. Платформа Qt

Qt – кроссплатформенный фреймворк для разработки программного обеспечения на языке программирования C++. Чаще всего он используется для создания графического интерфейса к программе. Кроссплатформенность и удобство использования являются большими достоинствами данной платформы, поэтому фреймворк Qt, помимо языка C++, прижился еще и в других языках, таких как Python – PyQt, Ruby – QtRuby, Java – Qt Jambi, PHP – PHP–Qt. Для платформы PyQt версии 5 была написана полная документация [11], что упрощает процесс создания пользовательских интерфейсов к программам.

Qt позволяет запускать написанное с его помощью программное обеспечение в большинстве современных операционных системе путем простой компиляции программы отдельно для каждой системы без изменения исходного программного кода. Платформа включает в себя все основные классы, которые могут потребоваться при создании графического интерфейса программы, а также множество других классов, которые могут использоваться для работы с сетью, базами данных и так далее.

Мы будем использовать данную платформу для разработки клиентской части приложения, так как это позволит запускать приложение с графическим интерфейсом на разных операционных системах, не меняя исходного кода. Однако, если реализовывать приложение на языке C++, то его исходный код

получится довольно громоздким и будет иметь сложную структуру. Поэтому для реализации функционала приложения будет использоваться язык программирования Python (как и в случае с сервером). Так же, программы, написанные на языке Python, как уже оговаривалось в Главе 3, тоже (как и на C++) являются кроссплатформенными.

Выбор языка программирования Python и платформы Qt для разработки клиентской части приложения позволяет сократить сложности и объем требуемого исходного кода, а, следовательно, снизить время разработки и упростить структуру исходного кода, при этом приложение будет являться кроссплатформенным.

Таким образом, разрабатываемое клиентское приложение будет реализовано на языке программирования Python с использованием модуля PyQt (набор «привязок» графического фреймворка Qt для языка программирования Python выполненный в виде модуля Python). Что же касается внутренней логики приложения, а именно работу по FTP–протоколу с сервером и работу с несколькими потоками одновременно – это позволяют сделать два модуля ядра Python, а именно модуль ftplib и модуль threading. Далее будет приведен краткий обзор этих модулей с простыми примерами.

§3. PyQt. Простой пример

Как уже говорилось ранее, PyQt – это некий модуль для языка Python, практически полностью реализующий возможности Qt. Если быть точнее, то это около 600 классов и более 6000 методов и функций в них для работы с готовыми виджетами, их стилями, поддержку воспроизведения аудио и видео и так далее.

Так как модуль PyQt версии 5 (последняя его версия) не является модулем ядра языка Python, его можно установить командой «sudo apt-get install python3 -pyqt5 pyqt5-dev-tools», если предполагается работа на операционной системе Linux.

Обзор классов и методов данной программной платформы PyQt 5 [11] не будет рассмотрен, так как обзор данного модуля может занять несколько сотен страниц, что не являлось бы рациональным.

Рассмотрим пример программного кода, который просто выводит пустое окно размером 250 на 150 пикселей и имеет название «Simple».

Код программы:

```
import sys
from PyQt5.QtWidgets import QApplication, Qwidget

if __name__ == '__main__':
    app = QApplication(sys.argv)
    w = QWidget()
    w.resize(250, 150)
    w.move(300, 300)
    w.setWindowTitle('Simple')
    w.show()

    sys.exit(app.exec_())
```

Результат будет выглядеть как на картинке ниже (запущено на Ubuntu 16.04):

Рисунок 6: пример простого окна с использованием PyQt5

Как видно выше, в приведенном примере, разрабатывать оконные приложения на Python с использованием PyQt довольно просто, поэтому данный модуль пользуется хорошей популярностью среди разработчиков ПО на Python.

§4. Модули ftplib и threading

Переходя к внутренней логике работы приложения, уже оговаривалось, что для ее реализации будут использоваться модули ftplib и threading, которые будут кратко рассмотрены ниже.

ftplib

Этот модуль языка Python определяет класс FTP и несколько дополнительных элементов. Класс FTP реализует клиентскую сторону протокола FTP. Модуль можно использовать для написания программ Python, в которых уже автоматизирована большая часть работы FTP–протокола.

Рассмотрим использования данного модуля на простом примере ниже.

Код программы:

```
from ftplib import FTP
ftp = FTP(„ftp.debian.org“)
ftp.login()
ftp.cwd(„debian“)
ftp.retrlines(„LIST“)
ftp.retrbinary(„RETR README“, open(„README“, „wb“).write)
ftp.quit()
```

Подключается библиотека ftplib, далее создается объект ftp, которому передается url – адрес хоста для подключения к нему, после чего мы пытаемся авторизоваться с помощью метода .login(). Далее, если авторизация прошла успешно, то в консоль будет выведено сообщение: «230 Login successful.». Далее, происходит смена рабочей директории на директории «debian». После этого происходит прием списка файлов и скачивание одного из них (в данном случае файла «README»). После того, как файл получен, закрываем соединение и завершаем работу FTP–клиента вызовом метода .quit().

Конечно, данный модуль реализует гораздо больше функционала, ознакомиться с которым можно изучив документацию к модулю [18].

threading

Threading – это модуль Python, который предоставляет высоко–уровневый программный интерфейс для работы с потоками, то есть данный модуль предоставляет разработчику возможность удобно взаимодействовать с потоками. Сам по себе поток представляет собой набор программных конструкций (функции, операции и так далее), которые могут выполняться параллельно на одном устройстве.

Рассмотрим простейший пример работы с модулем threading:

```
import threading

def doubler(number):
        """ A function that can be used by a thread """
        print(threading.currentThread().getName() + '/n')
        print(number * 2) print()

if __name__ == '__main__':
        for i in range(5):
                my_thread = threading.Thread(target=doubler, args=(i,)
                my_thread.start()
```

В примере выше импортируется модуль threading и создается обычная функция под названием doubler. Эта функция принимает значение и удваивает его. Также она выводит название потока, который вызывает функцию и выводит бланк–строчку в конце. Далее, в последнем блоке кода, мы создаем пять потоков, и запускаем каждый из них по очереди. Таким образом, будет запущено 5 экземпляров функции doubler с разными аргументами, которые будут выполняться параллельно.

В данной главе было рассмотрено несколько средств реализации клиентской части разрабатываемого приложения. Самыми оптимальными из

рассмотренных были определены язык программирования Python, платформа для создания кроссплатформенного графического интерфейса PyQt, модуль для разработки функционала FTP–клиента ftplib и модуль, для удобной работы с потоками операционной системы – threading.

В следующей главе будет рассматриваться конкретная реализация клиентской части приложения с приведением частей исходных кодов, на которые нужно обратить внимание, и последующем их разборе.

ГЛАВА 7. РЕАЛИЗАЦИЯ КЛИЕНТСКОЙ ЧАСТИ ПРИЛОЖЕНИЯ

В предыдущей главе были рассмотрены возможные средства реализации клиентской части программы, а также выбрано одно из них – это язык программирования Python версии 3, с использованием модулей ftplib и threading. В настоящей главе будет приведена идейная реализация клиентской части приложения, а также освещены технические моменты, на которые стоит обратить внимание для понимания того, как именно функционирует разработанная программа и как устроена ее внутренняя структура.

§1. Общая структура работы программы

Как было сказано, для передачи каждого отдельного файла требуется одно FTP–подключение к FTP–серверу (2 соединения – командное и информационное), поэтому клиентская часть программы для передачи нескольких файлов параллельно должна будет создать и использовать столько FTP–подключений, сколько файлов будут участвовать в обмене между сервером и клиентом. Так же, будет создано еще одно дополнительное FTP–подключение для организационной работы пользователя с FTP–сервером, а именно для создания папок, перехода по рабочим поддиректориям, получения списка файлов на сервере в текущей рабочей директории, удаления и изменение имени файлов и так далее. В предыдущей главе было оговорено, что FTP–подключения будут создаваться с помощью модуля ftplib, то есть, для каждого принимаемого или получаемого файла будет создаваться новый экземпляр класса, который предоставляет данный модуль для работы с одним FTP–подключением. Для «параллелизма» можно было бы хранить все эти подключения (экземпляры класса) в памяти и последовательно управлять каждым из них из основного управляющего потока программы (принимать или отправлять необходимое количество байт из того FTP–подключения

(экземпляра класса), которое соответствует тому файлу, часть которого необходимо принять или отослать в конкретный момент времени). Такой алгоритм работы выглядел бы примерно следующим образом: «Начало нового «цикла передали». FTP–подключение номер 1 (экземпляр класса номер 1), считай у себя из информационного соединения 6 400 Байт. Ждем завершения выполнения операции. Подключение номер 2, считай у себя из информационного соединения 19 200 Байт. Ждем завершение операции. Подключение номер 3, считай у себя из информационного потока 38 400 Байт. Ждем завершения операции. Конец «цикла передачи». Начинается новый «цикл передачи».

Однако такой подход тяжело считать параллельным, так как при реализации данного подхода программный код будет выполняться последовательно. Было принято решение о том, что настоящий эффект параллелизма будет достигнут тогда, когда все созданные FTP–подключения будут работать (отсылать и принимать части файлов) параллельно. При разработке программ, когда речь заходит о параллельности, с ней сразу же ассоциируются такие понятия, как «потоки» и «процессы» – сущности в операционных системах, которые работают параллельно (то есть, сама операционная система организовывает параллельную работу созданных потоков и процессов). Переключение между процессами для операционной системы является более ресурсоемкой операцией, чем переключение между потоками (так как происходит смена переменных окружения и смена адресного пространства в памяти). Для параллелизма было принято решение использовать потоки, так как в программе будут находиться некие переменные и их значения, которые необходимо будет использовать в разных частях кода, который выполняется параллельно. Таким образом, было определено, что в рамках одного «цикла передачи» необходимо, чтобы кадры файлов определенных размеров принимались или передавались из соответствующих FTP–подключений (экземпляров классов) в разных потоках

операционной системы, что позволит достигнуть эффекта «параллелизма» передачи и отправки файлов на уровне операционной системы.

Так как при реализации метода будут использоваться потоки операционной системы, необходимо, чтобы они между собой каким–то образом синхронизировались, то есть нужна, так называемая, «точка синхронизации». В данном случае, точкой синхронизации будет выступать окончание «цикла передачи», то есть тот момент, когда все кадры от каждого файла, участвующего в обмене, были приняты/переданы. Это гарантирует, что на каждой новой итерации планирования «цикла передачи» кадры от обмениваемых файлов были полностью переданы и прошлый цикл не будет «накладываться» на новый цикл (планируемый).

Для того, чтобы программа не блокировалась при планировании и исполнении «циклов передачи», данный функционал был так же в отдельный поток, что позволяет продолжать пользователю работу с программой даже когда новые «циклы передачи» блокируются, или исполняются. В таком случае, графически все потоки операционной системы выполнения программы можно представить так, как это сделано на рисунке ниже (см. Рисунок 7).

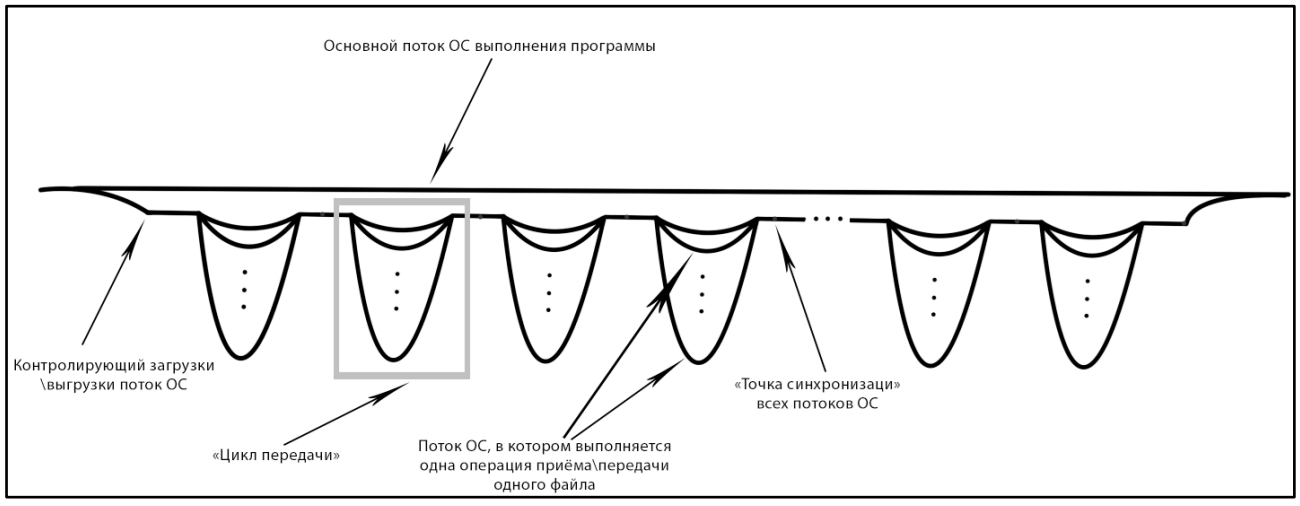


Рисунок 7: структурная схема потоков (нитей) ОС при работе программы (течение времени работы программы – слева направо)

§2. Технические моменты

При реализации программы возникали некие технические трудности, которые необходимо было как–то решать. Данный параграф посвящен таким трудностям и путям их решения.

Добавление и удаление загрузок и выгрузок файлов

Как можно видеть на структурной схеме потоков ОС в программе, которая приведена выше – контролирующий загрузки и выгрузки поток постоянно планирует и осуществляет операции по загрузке и выгрузке кадров файлов, участвующих в обмене. Программно это происходит в цикле, который «пробегаясь» по каждой загрузке/выгрузке в специальном контейнере (списке) планирует для нее новую операцию. При добавлении новой загрузки//выгрузки из основного потока, размер списка изменяется, что влечет за собой возникновение исключения при выполнении программы в потоке, который отвечает за контроль всех загрузок/выгрузок. Данная проблема была решена следующим образом: при добавлении/удалении новой загрузки или выгрузки файла в процессе работы программы, данная загрузка/выгрузка файла добавляется в отдельный специальный временный программный контейнер (структура данных, которая содержит в себе множество элементов одинаковой природы (типа)), а когда текущий «цикл передачи» завершен и программа должна перейти к планированию нового «цикла передачи», она предварительно «просматривает» временные контейнеры добавления/удаления загрузок и выгрузок и осуществляет их добавление или удаление из основного контейнера, в котором хранятся все загрузки/выгрузки.

Таким образом, происходит синхронизация добавленных операций по добавлению и удалению новых загрузок и выгрузок с основным контейнером, в котором хранятся все загрузки/выгрузки, что решает проблему изменения

размера списка всех загрузок и выгрузок в процессе работе с данным списком.

Работа контролирующего загрузки/выгрузки потока

Контролирующий загрузки и выгрузки поток является параллельным потоком по отношению к основному потоку ОС выполнения программы. При разработке программы возникла необходимость его запускать и останавливать. Для этого в программе был создан флаг, проверяя который контролирующий загрузки и выгрузки поток либо продолжал свою работу, либо заканчивал последние запланированные операции (последний «цикл передачи») и приостанавливал свою работу.

Таким способом была решена проблема запуска, приостановки и остановки работы контролирующего потока. Стоит отметить, что контролирующий поток, в ходе своей работы, изменяет данные в общем контейнере загрузок и выгрузок (такие параметры, как: сколько байт было загружено/выгружено от каждого файла, статус загрузки/выгрузки, время загрузки/выгрузки и так далее), что, впоследствии, становится доступным в основном исполняющем потоке и готово к выводу этих данных о загрузках/выгрузках пользователю.

Передача функций графического интерфейса в качестве аргументов

Структура программного кода клиентской части приложения реализована так, что в нем присутствует основной класс, отвечающий за всю функциональную часть приложения (создание подключения, обмен информацией с сервером и т. д.) и класс графического интерфейса, который инкапсулирует в себе объект основного класса.

Класс графического интерфейса передает в основной класс некоторые функции для работы с графическим интерфейсом, а именно его обновлением. То есть, например, когда в функциональном классе файл был принят/передан полностью, то обязательство вызова функции для удаления строки состояния обмена данного файла берет на себя не класс графического интерфейса, а именно объект основного класса.

Таким образом, некоторые функции графического интерфейса работают как функции обратного вызова, которые срабатывают по происхождению какого–то события в основном классе, отвечающем за функциональность программы.

§3. Графический интерфейс

Данный параграф посвящен рассмотрению графического интерфейса разработанного программного обеспечения, а именно клиентской его части. Условимся называть данную часть приложения «клиентом».

Главное окно

Главное окно программы имеет 3 кнопки: «Подключиться» – вызывает окно ввода данных для подключения к FTP–серверу, «Отключиться» – производит «закрытие» текущего подключения, и «Восстановить работу из истории» – при открытом подключении к тому серверу, с которым производилась работа в прошлый сеанс работы с программой, клик по данной кнопке восстановит все загрузки и выгрузки, которые были не закончены в прошлый сеанс работы с программой.

Так же, графический интерфейс главного окна содержит три области для отображения локальной файловой системы, всех загрузок/выгрузок и удаленной файловой системы (рабочей директории пользователя на сервере) соответственно. Как можно видеть, области по бокам имеют по три колонки:

«Тип» – Файл или Папка, «Имя» – имя Файла или Папки, и размер – размер файла. Таким образом, пользователь может видеть основную информацию о файлах для совершения обмена файлами. Центральная область отображает процесс обмена файлами с сервером и будет рассмотрена детально ниже (см. Рисунок 8).

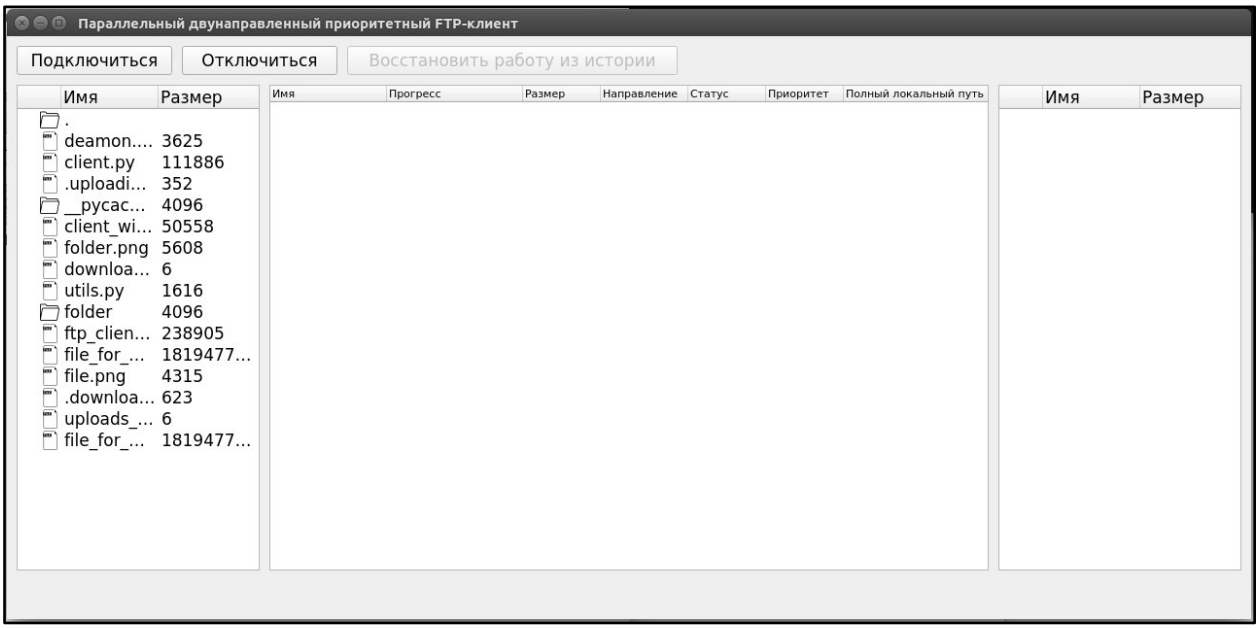


Рисунок 8: главное окно клиентской части приложения

Окно подключения

Окно подключения к FTP–серверу представляет собой модальное окно с полями для ввода. Введенные данные используются для подключения к удаленному FTP–серверу.

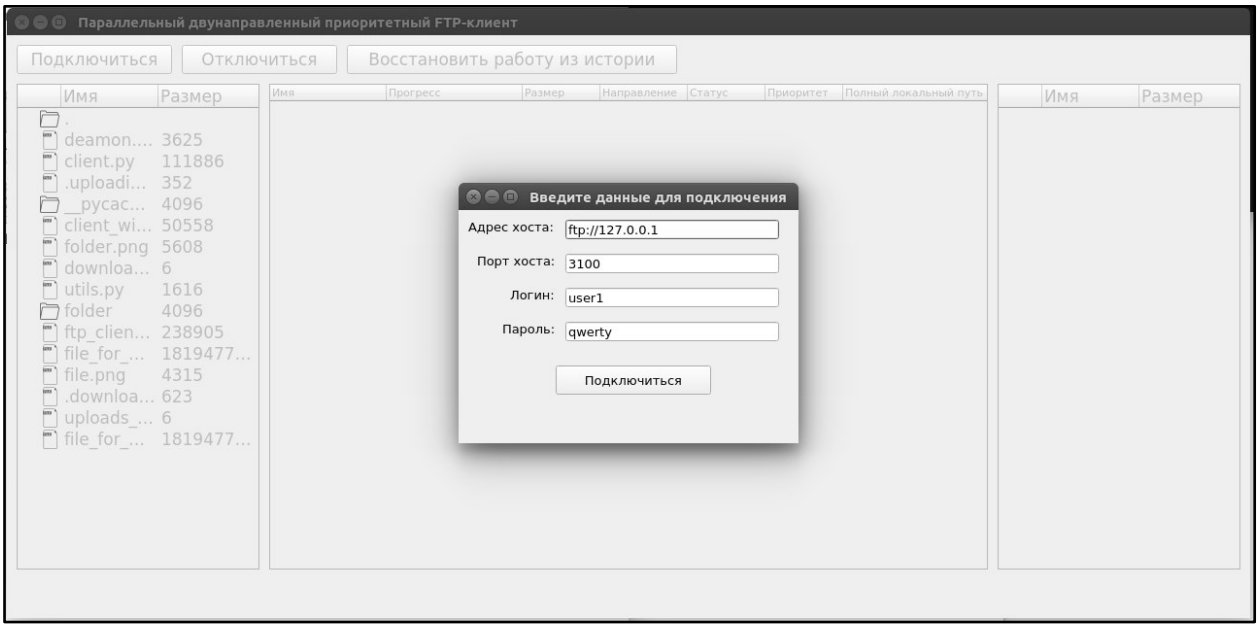


Рисунок 9: модальное окно подключения к FTP-серверу

После успешного подключения, данное окно закрывается, в главном окне заполняется список файлов в удаленной рабочей папке пользователя. Можно производить обмен файлами.

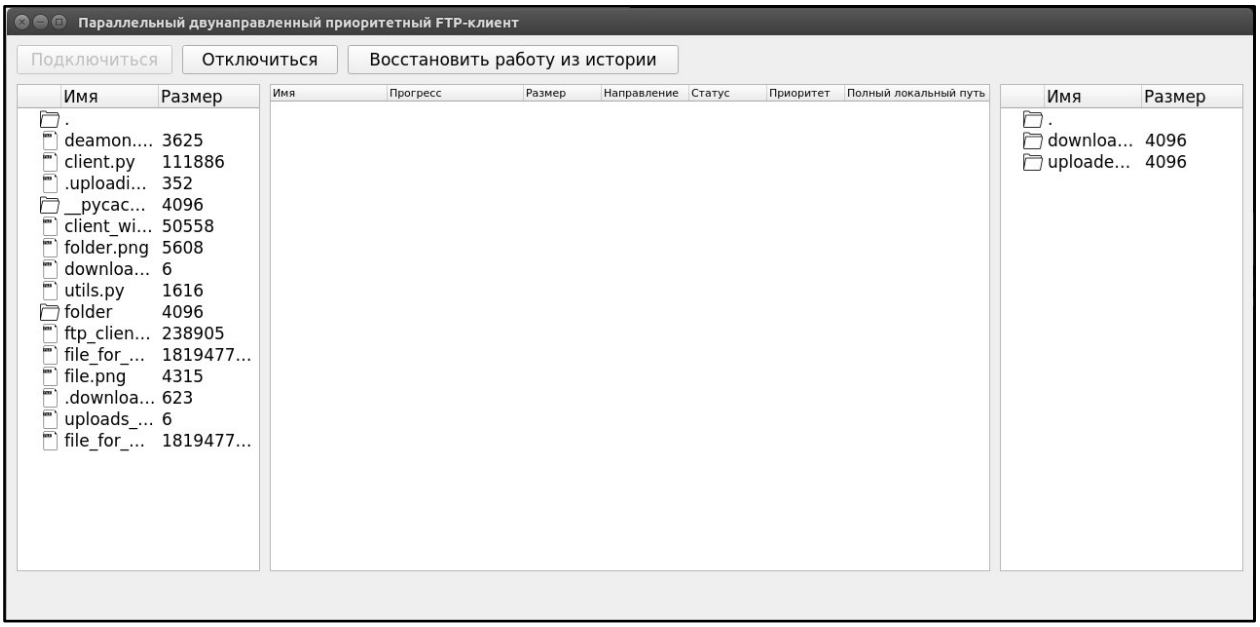


Рисунок 10: главное окно после успешного подключения к серверу

Переход в доступные директории осуществляется путем двойного клика мыши по необходимой директории. При двойном клике по файлу

ничего не происходит. При нажатии правой кнопкой мыши по локальному файлу открывается контекстное меню, как на рисунке ниже (см. Рисунок 11).

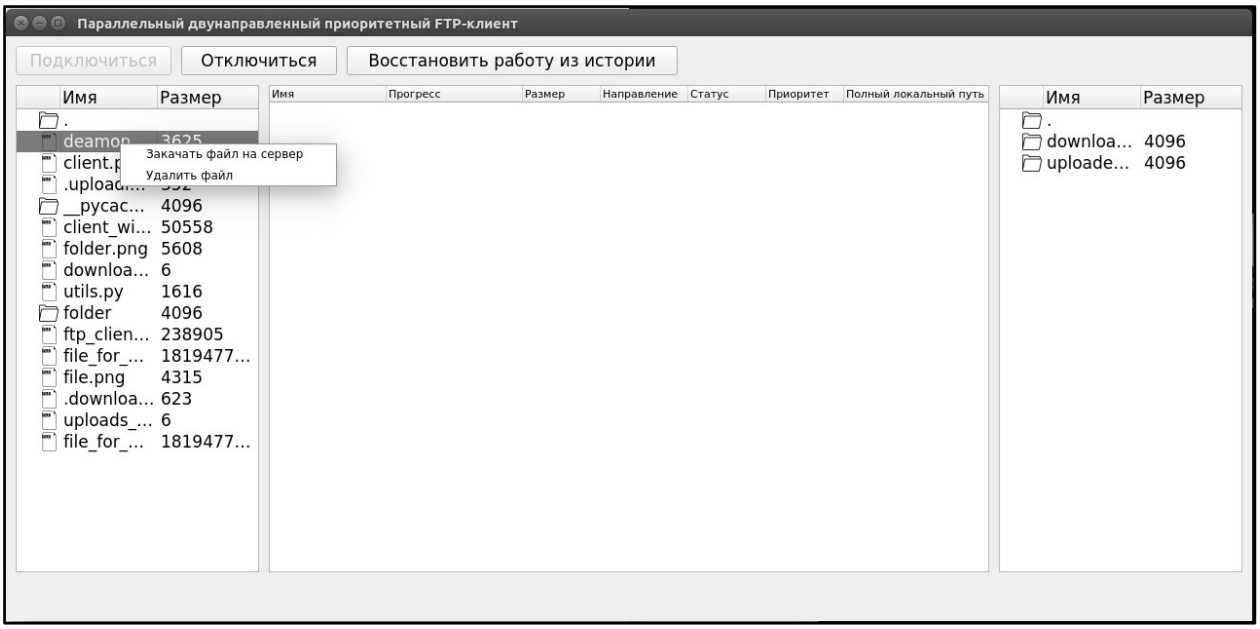


Рисунок 11: контекстное меню локального файла

Так же, при нажатии правой кнопкой мыши в области отражения локальной файловой системы на пустом месте, появляется возможность обновить данные о локальных файлах в данной рабочей директории.

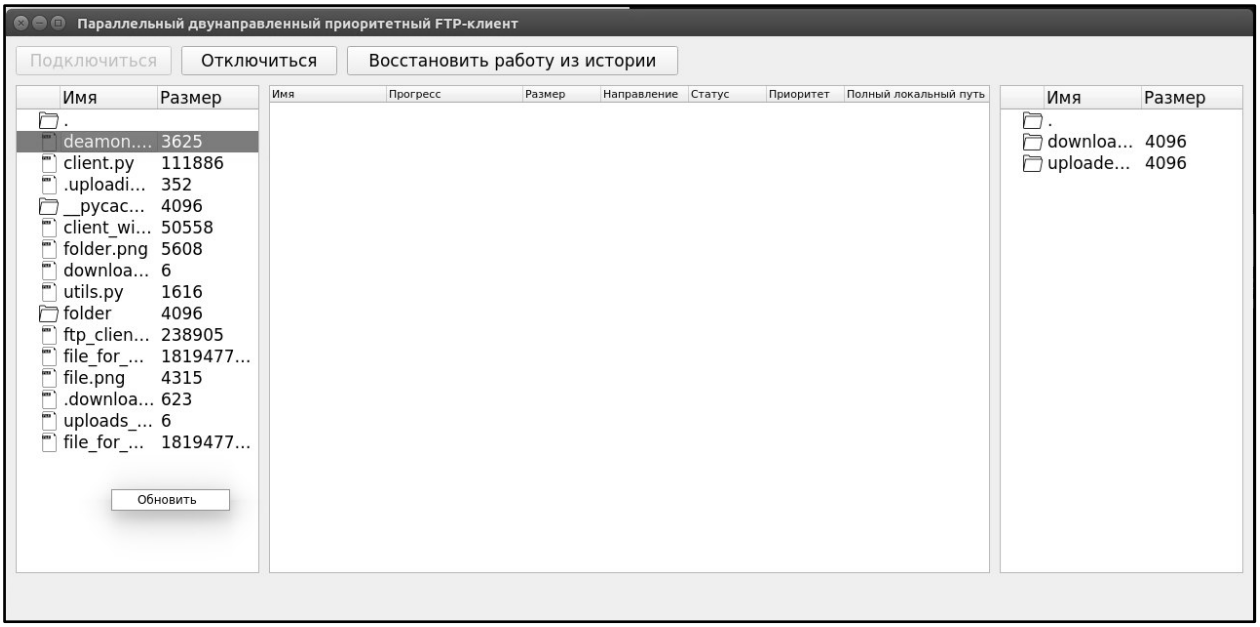


Рисунок 12: контекстное меню пустой области локальной файловой системы

Что же касается удаленной рабочей директории, то в ней все устроено таким же образом с некоторыми дополнениями, а именно добавлена возможность создания новой папки и изменения имени файла.

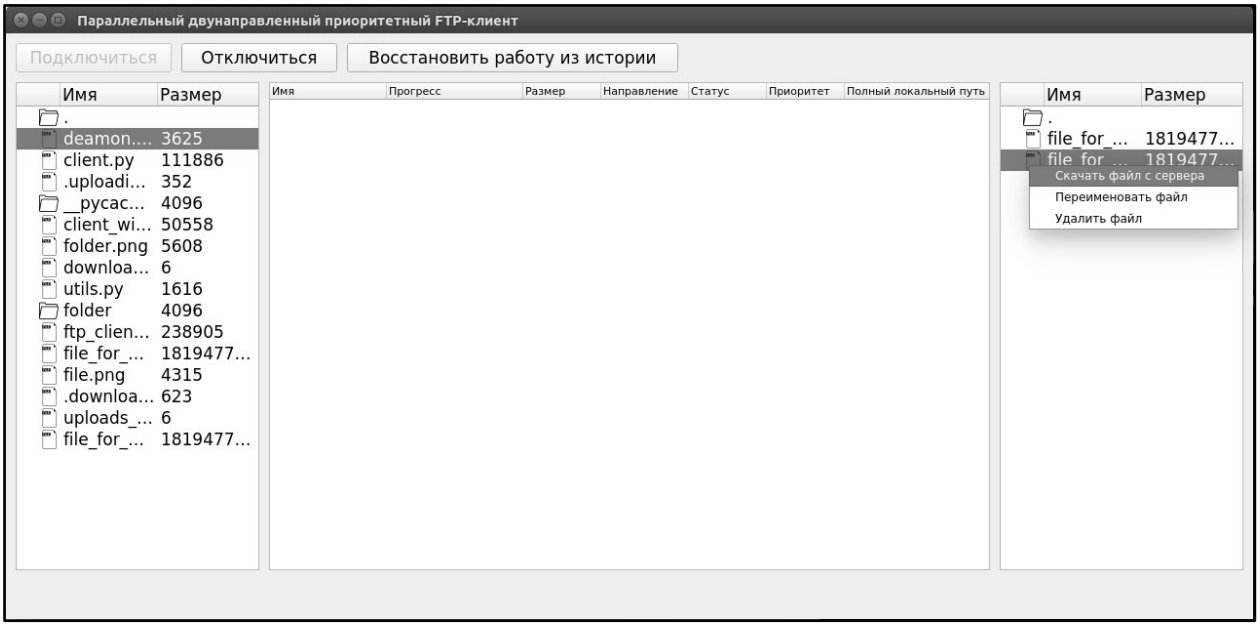


Рисунок 13: контекстное меню удаленного файла

После того, как какие–то файлы были задействованы в обмене, центральная область главного окна клиента отображает состояние обменов файлами так, как показано на рисунке ниже (см. Рисунок 14).

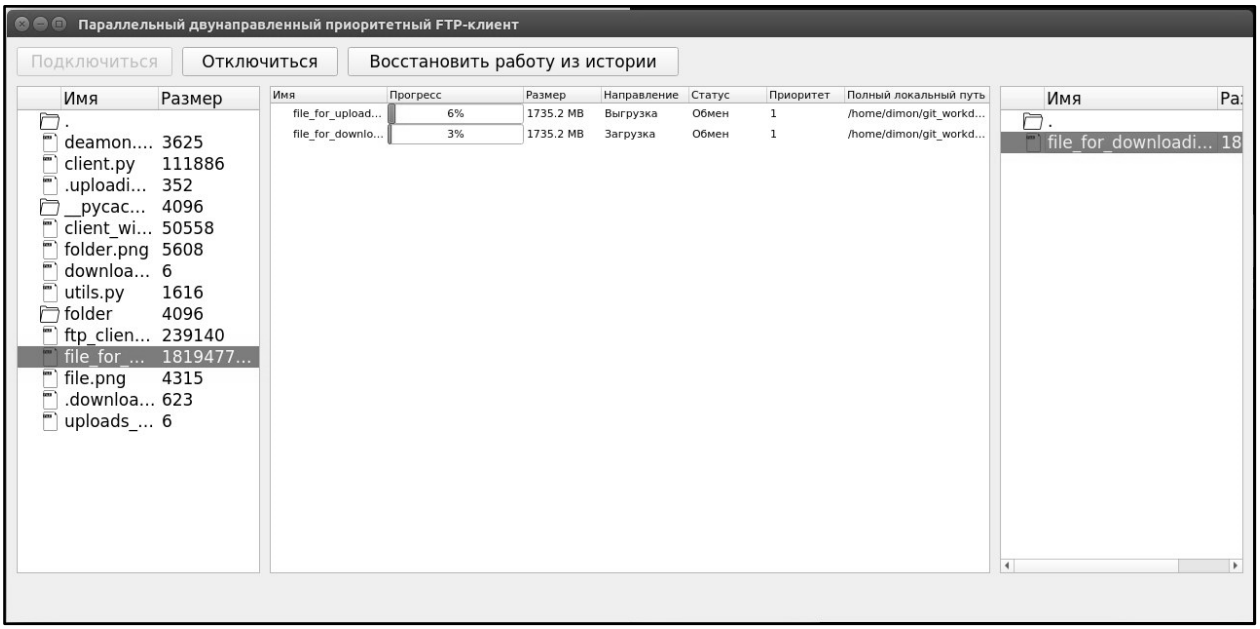


Рисунок 14: состояние главного окна в процессе обмена файлами с FTP-сервером

Каждый обмен файлами позволяет вызвать для себя контекстное меню, как на рисунке ниже (см. Рисунок 15).

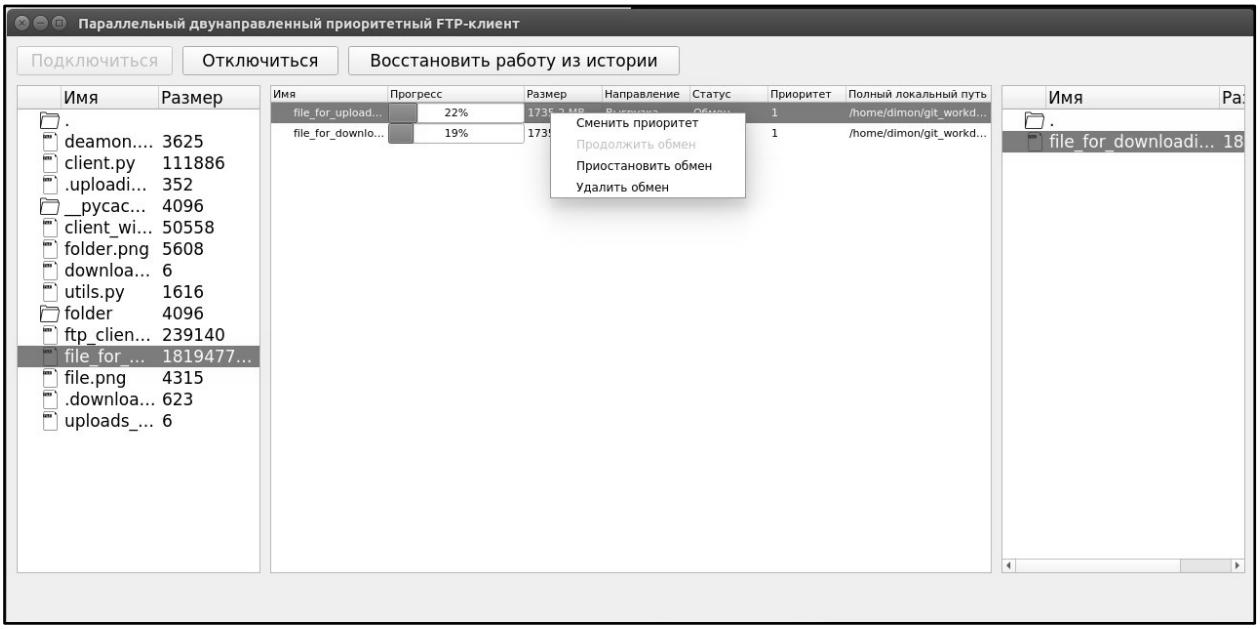


Рисунок 15: контекстное меню загрузок и выгрузок файлов

Данное контекстное меню позволяет сменить приоритет загрузки/выгрузки файла, приостановить прием/передачу, продолжить

прием/передачу файла, а также удалить (остановить) данную загрузку или выгрузку файла.

Окно смены приоритета загрузки/выгрузки

Окно для смены приоритета загрузки/выгрузки файла вызывается при попытке смены приоритета через контекстное меню обмена файлом. Данное окно выглядит так, как показано на рисунке ниже (см. Рисунок 16).

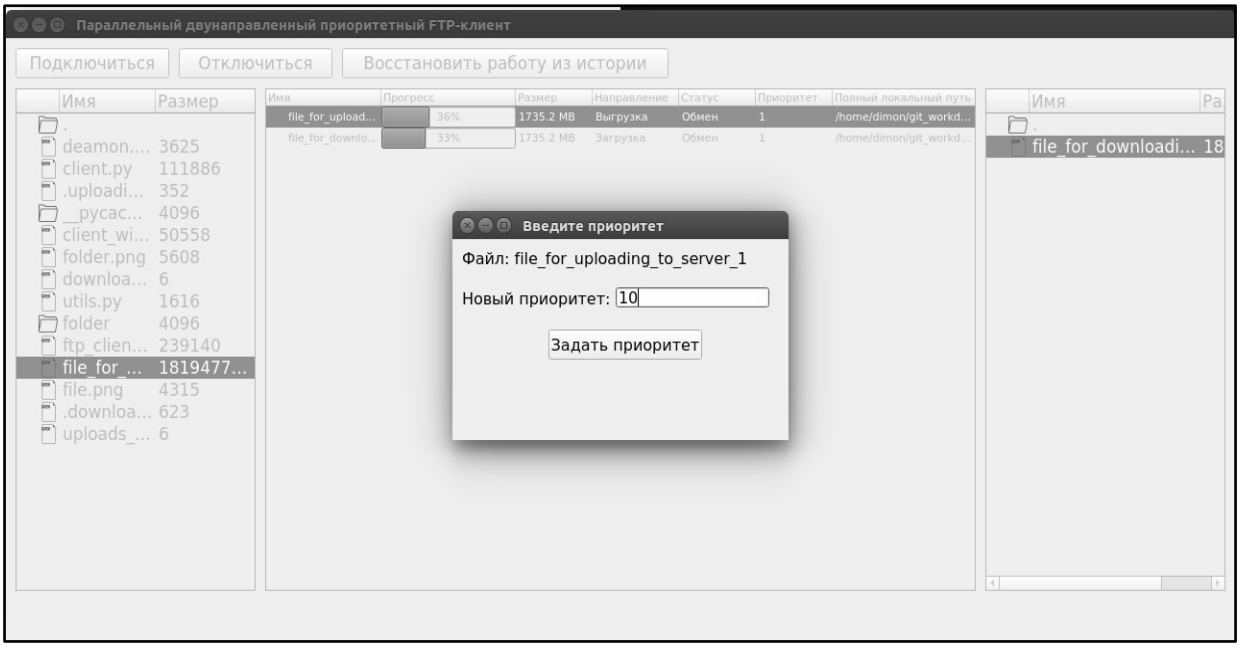


Рисунок 16: окно для смены приоритета загрузки/выгрузки файла

Несложно догадаться, что данное окно позволяет задать приоритет определенной загрузке/выгрузке файла, что влияет на скорость приёма или передачи соответствующего файла.

Другие вспомогательные элементы графического интерфейса (окно создания новой удаленной директории, окно изменения имени файла) не будут рассмотрены, так как их графический интерфейс интуитивно прост и понятен.

ГЛАВА 8. ТЕСТИРОВАНИЕ, СБОР И АНАЛИЗ РЕЗУЛЬТАТОВ

Настоящая глава посвящена описанию методов тестирования разработанного клиент-серверного приложения (работающего сервера и множества запущенных активных клиентов, подключенных к данному серверу), технического обеспечения, на котором производилось тестирование, характерных моментов, на которые стоит обратить внимание, а также анализу полученных данных.

§1. Методика тестирования и техническое обеспечение

Для проведения тестов разработанного клиент–серверного приложения были использованы 14 компьютеров, выступающих в роли «клиентов» (на каждом из них была запущена клиентская часть приложения с соответствующими параметрами) и один вычислительный узел («angel»), расположенный на факультете ВМК МГУ, который использовался в качестве «сервера». Обмен файлами происходил параллельно (одновременно) со всеми клиентами.

Так как необходимо было имитировать различных пользователей (клиентов с разной пропускной способностью канала связи с сервером). Для этого у 14ти компьютеров была ограничена пропускная способность канала связи в соответствии с законом 2*N Мбит/сек, где N – это номер компьютера (клиента) от 1 до 14ти соответственно.

Передача файлов тестировалась в обе стороны, то есть файлы передавались как от сервера к отдельному клиенту, так и от этого же клиента к серверу. Размер файлов был выбран одним из 16, 32 и 64 Мегабайт. Такой объем был выбран потому, что для обмена файлами такого размера один тест занимал не более 20ти минут времени, что облегчило сам процесс тестирования. Стоит отметить, что при обмене файлами меньшего размера в

результатах тестирования характерные моменты становятся менее заметны, поэтому для проведения тестирования использовать файлы меньшего размера было нерационально.

В ходе тестирования замерялось время загрузки и передачи каждого из файлов. То есть, отсчёт времени начинался тогда, когда файл добавлялся в список всех загрузок и выгрузок, и заканчивался, когда файл был полностью принят или передан, в зависимости от направления обмена.

§2. Тесты, описание и анализ результатов

В первую очередь было принято решение о проведении тестов, иллюстрирующих приоритетный обмен файлами.

Всего было произведено 10 тестов, однако рассмотрены будут только первые 6 из них, так как первые 4 теста были произведены с использованием метода разделения потока данных «по кадрам передачи», 5ый тест произведён без разделения потока данных (файлы передавались последовательно), и 6ой с использованием метода разделения потока данных «по времени передачи». При проведении 6го и последующих тестов целью тестирования было показать, что данный способ разделения потока данных не является «справедливым» по отношению к приоритетам, которые были заданы загрузкам/выгрузкам файлов. То есть, скорости передачи данных для каждой загрузки/выгрузки в результате проведения 6го и последующих тестов соотносились между собой не пропорционально расставленным приоритетам. Так как метод разделения потока данных «по времени передачи» не является основным в контексте данной работы и последующие результаты тестов показали схожую с 6ым тестом тенденцию, в качестве примера был рассмотрен лишь один из них.

Тест №1

Был произведен обмен несколькими файлами одинакового размера с разными приоритетами. Используемый метод разделения потока данных – «по кадрам передачи». Предполагалось, что файлы, имеющие приоритет выше, чем другие будут передаваться быстрее, чем те, что имеют более низкий приоритет.

Для этого на сервере в папке каждого пользователя были сгенерированы файлы со случайным содержимым объемом 32 Мбайт, причем для каждого пользователя (клиента/компьютера) с номером N было сгенерировано N файлов. Эти файлы должны были загружаться каждым соответствующим клиентом параллельно. Каждому файлу, поставленному на загрузку клиентом, был задан приоритет на 1 больше, чем предыдущему, начиная с единицы. Пример: поставить на загрузку с сервера первый файл размером 32 Мбайт с приоритетом 1, второй файл размером 32 Мбайт с приоритетом 2, третий файл размером 32 Мбайт с приоритетом 3, и так далее до N, где N – номер компьютера (клиента). Аналогичные действия были проделаны для файлов, которые должны были быть выгружены от клиента на сервер. То есть, каждый клиент (компьютер) с номером N должен выгрузить на сервер N файлов размером 32 Мбайт параллельно с загрузками. Приоритет задавался аналогично загрузкам.

Для проведения анализа результатов будут представлены результаты тестирования от двух случайно выбранных клиентов (компьютеров): 4го и 7го. Результаты работы остальных клиентов имеют ту же тенденцию, поэтому рассматривать их не имеет смысла.

Результаты работы 4го клиента представлены на диаграмме ниже (зависимость времени параллельной загрузки/выгрузки файлов от их приоритетов).

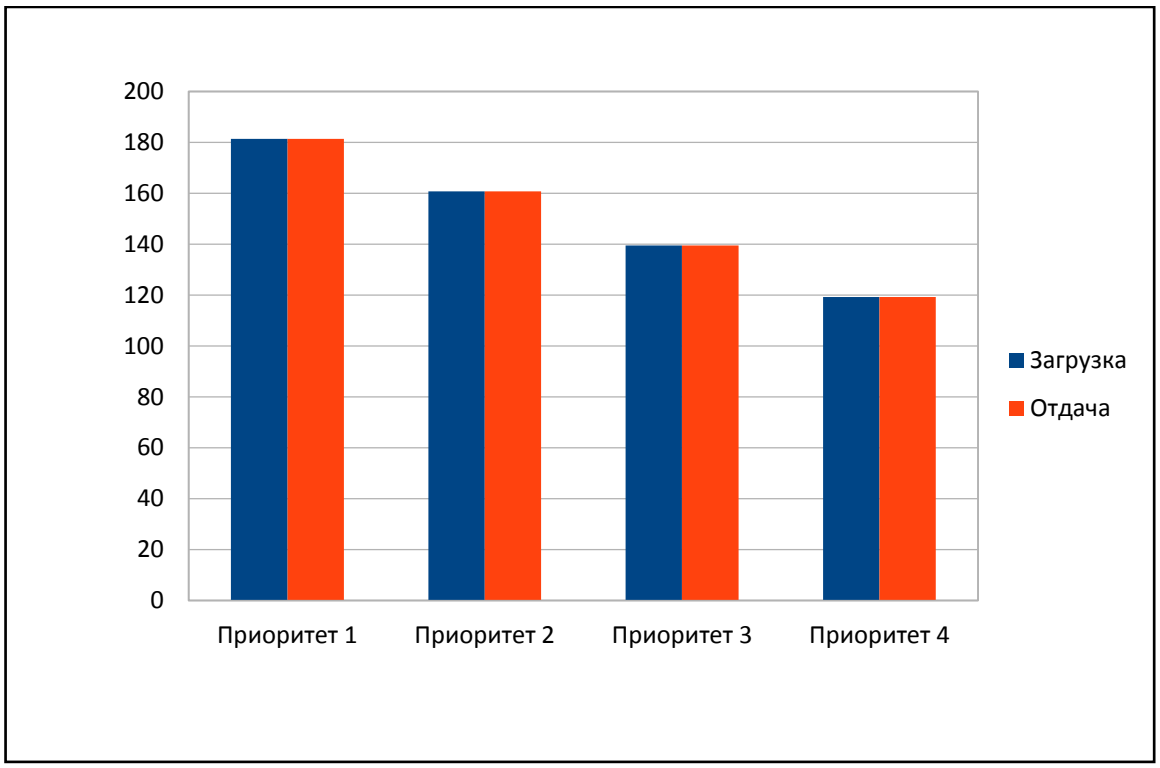


Рисунок 17: результат работы клиента №4 (пропускная способность канала связи 8 Мбит/сек)

Результаты работы 7го клиента представлены на диаграмме ниже (зависимость времени параллельной загрузки/выгрузки файлов от их приоритетов).

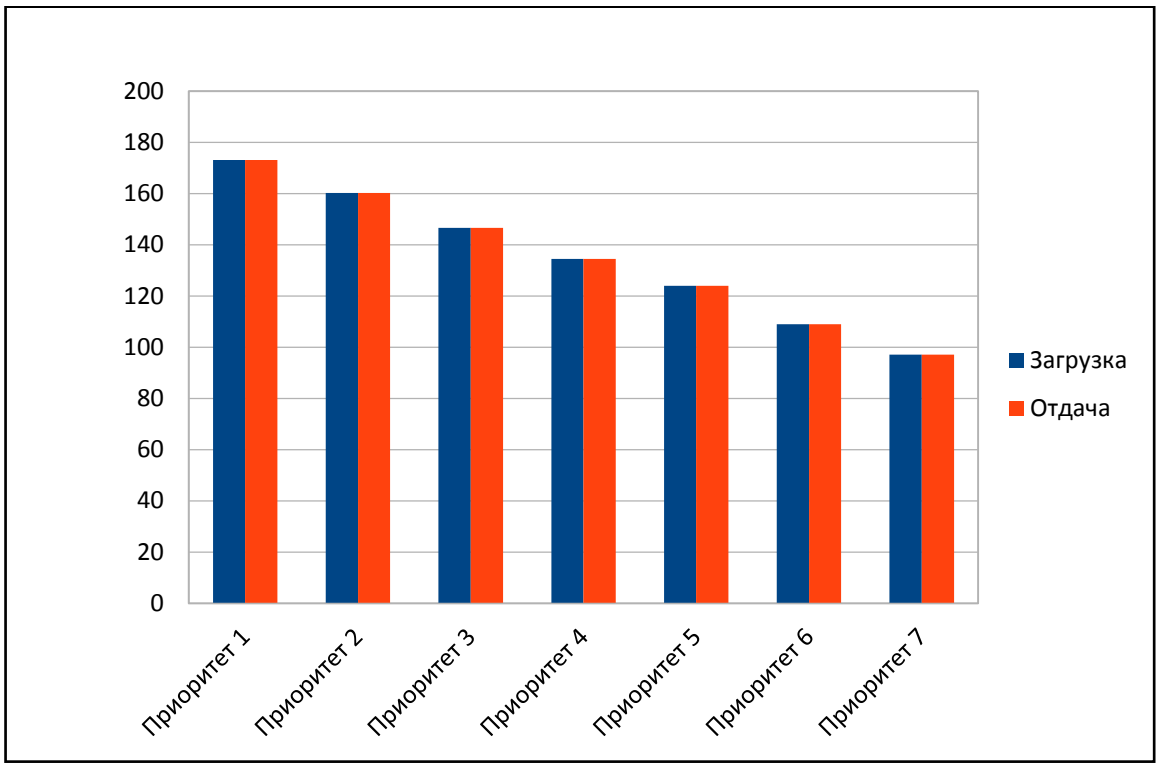


Рисунок 18: результат работы клиента №7 (пропускная способность канала связи 14 Мбит/сек)

Анализируя приведенный выше диаграммы, можно сделать вывод о том, что клиент–серверное приложение работает так, как предполагалось, а именно, файлы, участвующие в обмене с одинаковым приоритетом, имеют примерно одинаковое время загрузки/выгрузки (видно в рамках одного столбца), а файлы, имеющие разные приоритет имеют разное время загрузки/выгрузки (видно при сравнении двух или более столбцов). С результатами работы других клиентов можно ознакомиться в приложениях к данной научной работе.

Тест №2

Был произведен двусторонний параллельный обмен тремя файлами разного размера (16, 32 и 64 Мбайт) с разными приоритетами, причем приоритеты задавались таким образом, чтобы компенсировать разницу в размере файлов и время их обмена было одинаковым (1, 2 и 4 приоритет

соответственно). Используемый метод разделения потока данных – «по кадрам передачи».

Для проведения анализа результатов так же рассмотрим результаты работы двух случайно выбранных клиентов: 3го и 6го.

Результаты работы клиента №3 представлены на диаграмме ниже.

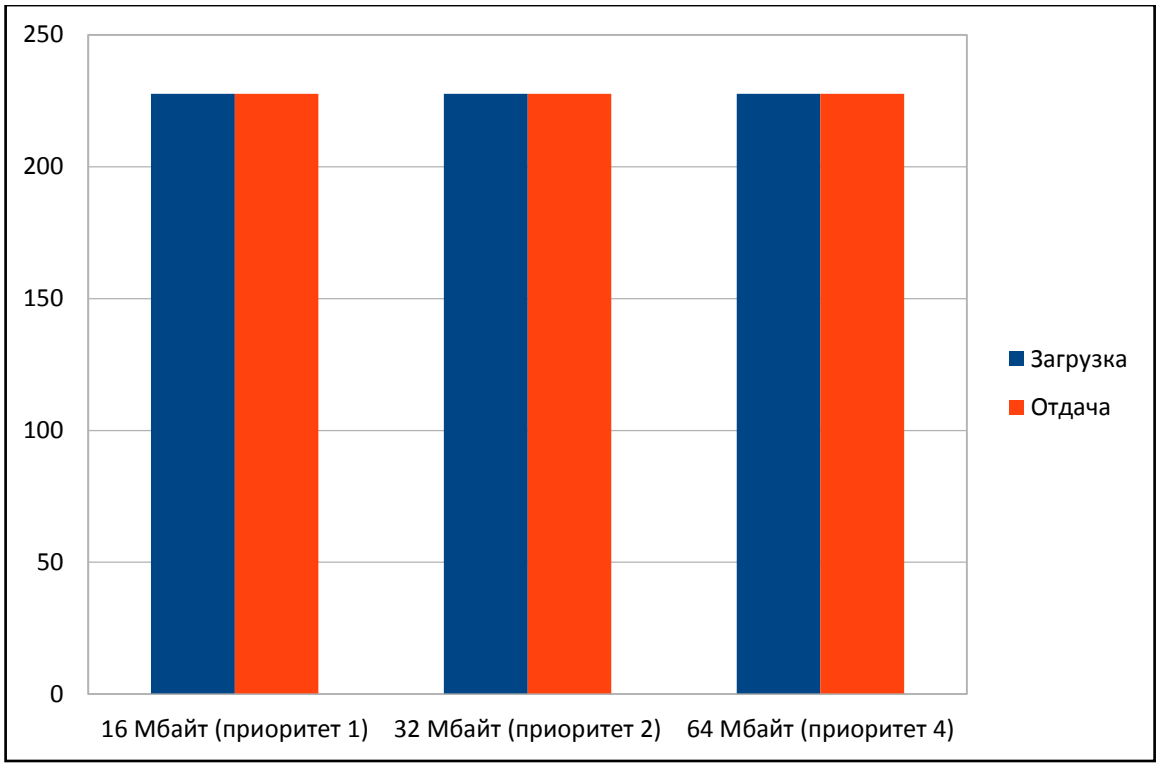


Рисунок 19 результат работы клиента №3 (пропускная способность канала связи 6 Мбит/сек)

Результаты работы клиента №6 представлены на диаграмме ниже.

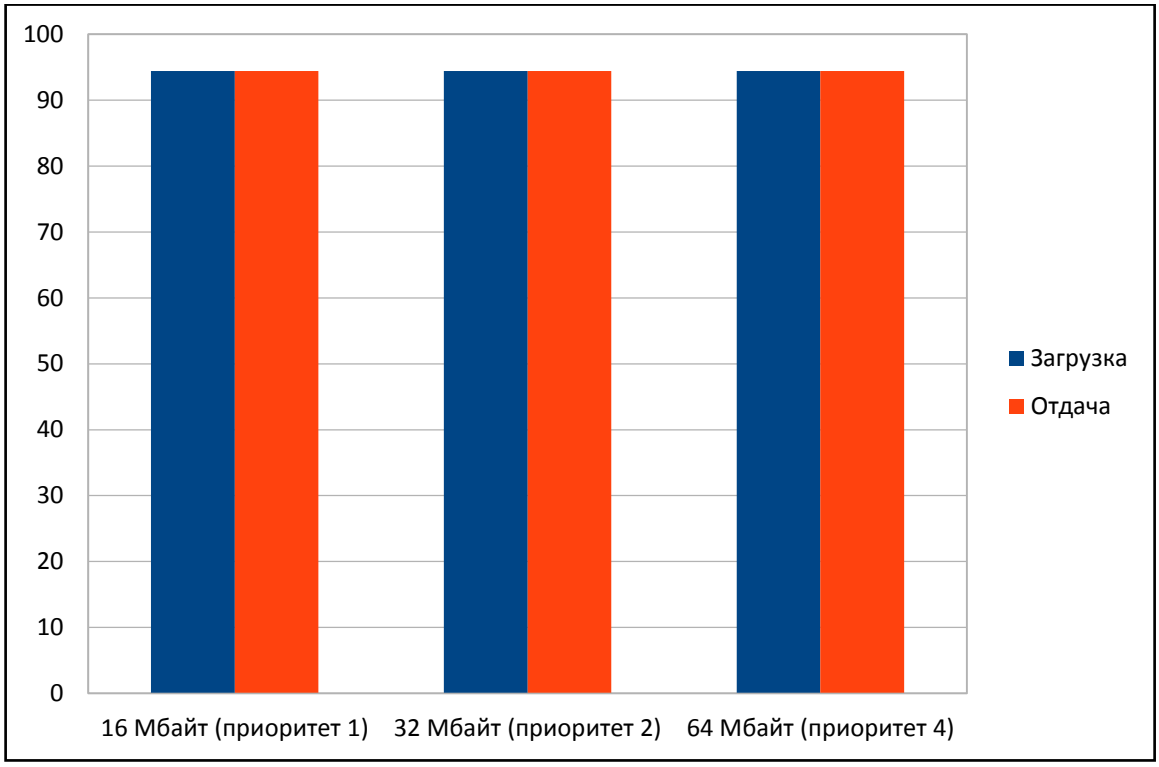


Рисунок 20: результат работы клиента №6 (пропускная способность канала связи 12 Мбит/сек)

Проанализировав диаграммы, можно заметить, что файлы, пусть и разного размера, были загружены и выгружены за одно и то же время. Такой эффект был достигнут за счет разницы в приоритетах. Это говорит о том, что программа отработала так, как и предполагалось перед проведением теста.

Тест №3

Данный тест – аналогия Теста №1, только приоритет у всех загружаемых и выгружаемых файлов одинаковый. Данный тест предполагает, что файлы, которые участвуют в обмене, должны загрузиться и выгрузиться примерно за одинаковое время.

Рассмотрим результат работы двух случайно выбранных клиентов: 2го и 5го.

Результаты работы клиента №2 представлены на диаграмме ниже.

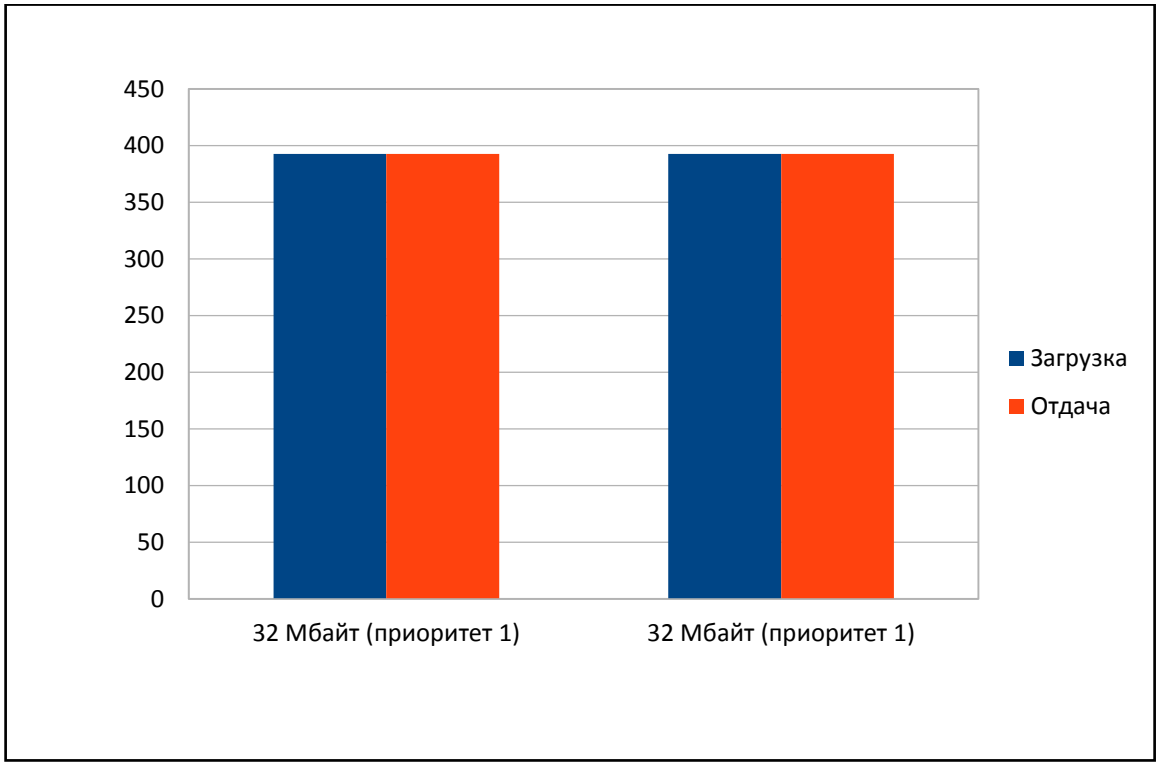


Рисунок 21: результат работы клиента №2 (пропускная способность канала связи 4 Мбит/сек)

Результаты работы клиента №5 представлены на диаграмме ниже.

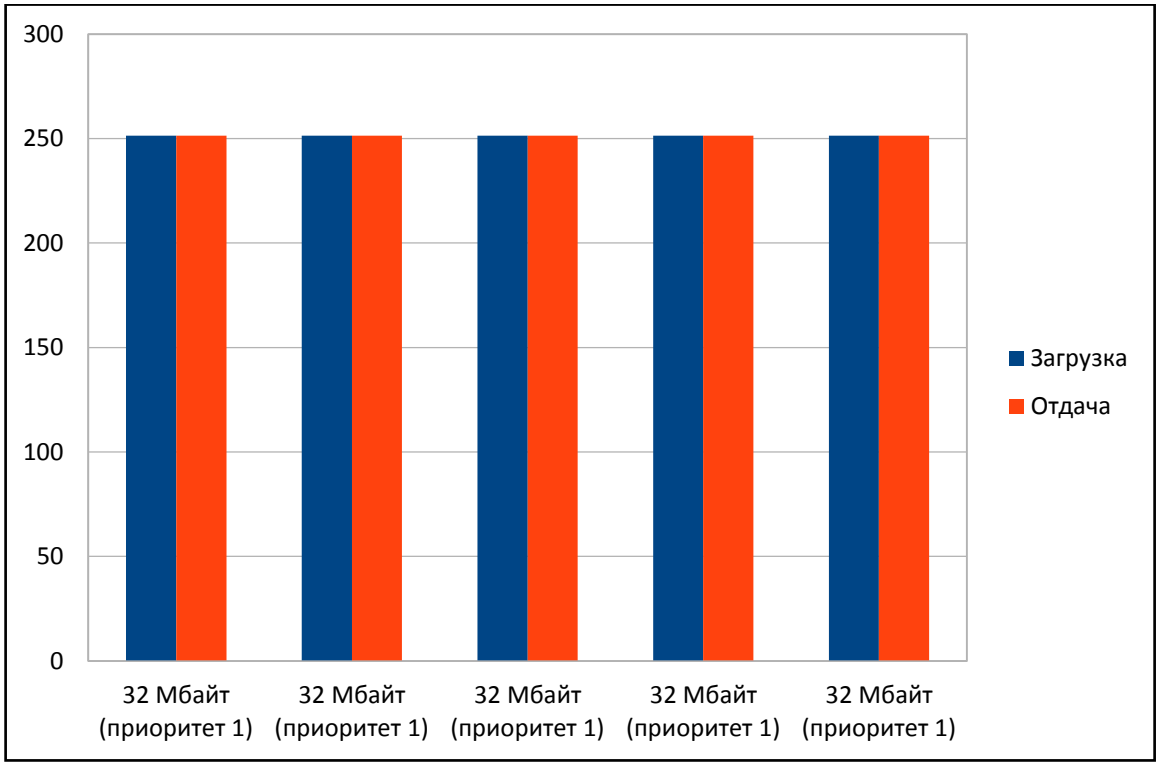


Рисунок 22: результат работы клиента №5 (пропускная способность канала связи 10 Мбит/сек)

Проанализировав диаграммы, можно заметить, что файлы все файлы были загружены и выгружены примерно за одинаковое время. Это значит, что обмен файлами происходил параллельно с одинаковым приоритетом.

Тест №4

Данный тест – аналогия Теста №2, за исключением того, что приоритет у всех загружаемых и выгружаемых файлов одинаковый. Данный тест предполагает, что файлы, которые участвуют в обмене, должны загрузиться и выгрузиться за разное время.

Рассмотрим результат работы двух случайно выбранных клиентов: 1го и 3го.

Результаты работы клиента №1 представлены на диаграмме ниже.

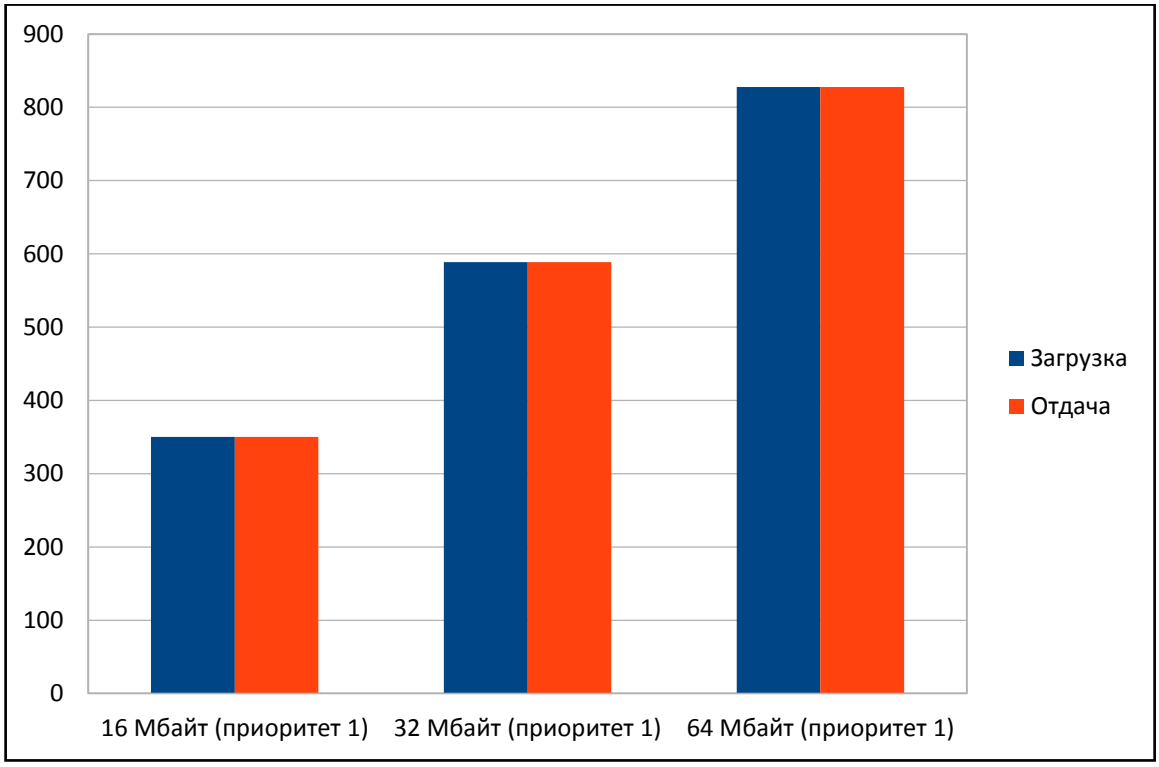


Рисунок 23: результат работы клиента №1 (пропускная способность канала связи 2 Мбит/сек)

Результаты работы клиента №3 представлены на диаграмме ниже.

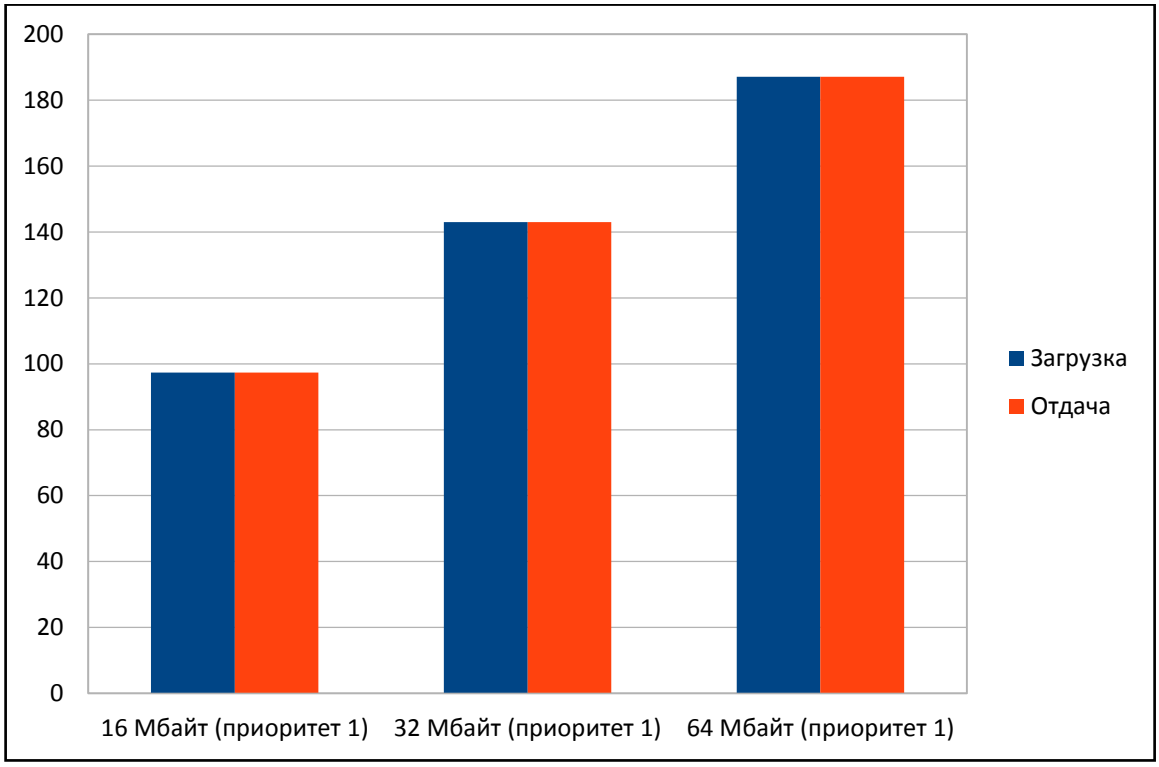


Рисунок 24: результат работы клиента №3 (пропускная способность канала связи 6 Мбит/сек)

Проанализировав диаграммы, можно заметить, что все файлы, имея разный приоритет, были загружены и выгружены за разное время, как и предполагалось. Это говорит о том, что они были разного размера. Программа отработала корректно.

Тест №5

Этот тест – подобен Тесту №4, однако, в данном тесте обмен файлами будет производиться не параллельно, а последовательно, причем сначала загрузки, а затем выгрузки. Предполагается, что на прием и передачу файлов большего размера будет затрачиваться пропорционально больше времени, чем на прием и передачу меньших файлов.

Рассмотрим и проанализируем результаты работы двух случайно выбранных клиентов: 4го и 6го.

Результаты работы клиента №4 представлены на диаграмме ниже.

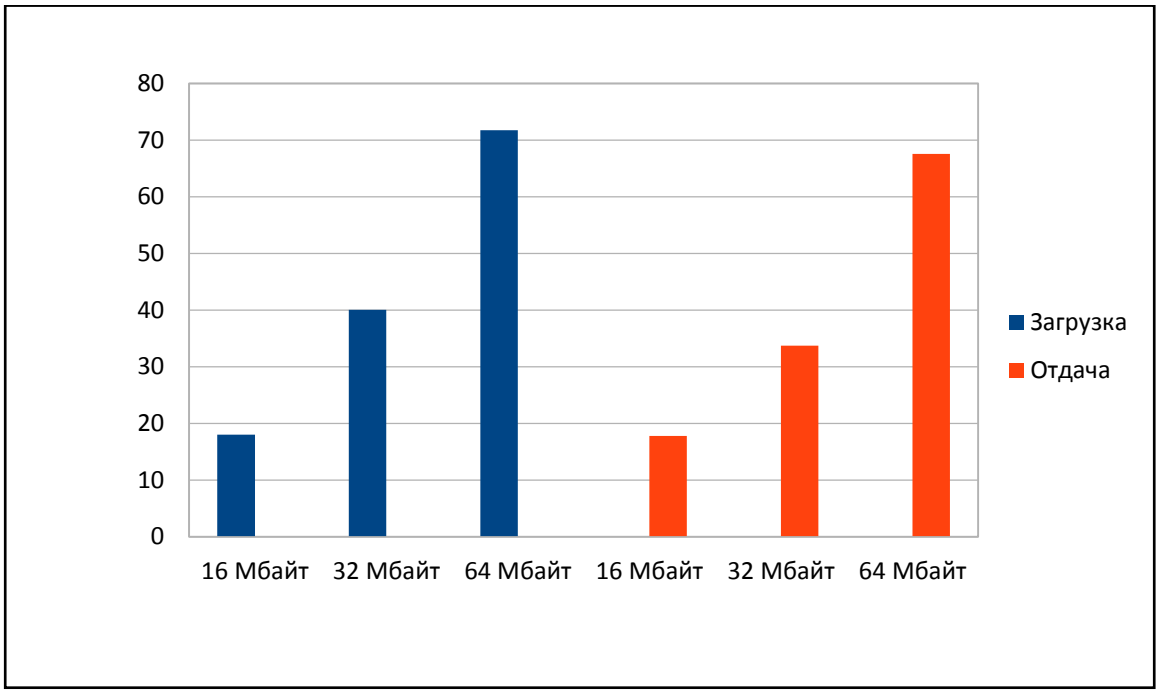


Рисунок 25: результат работы клиента №4 (пропускная способность канала связи 8 Мбит/сек)

Результаты работы клиента №6 представлены на диаграмме ниже.

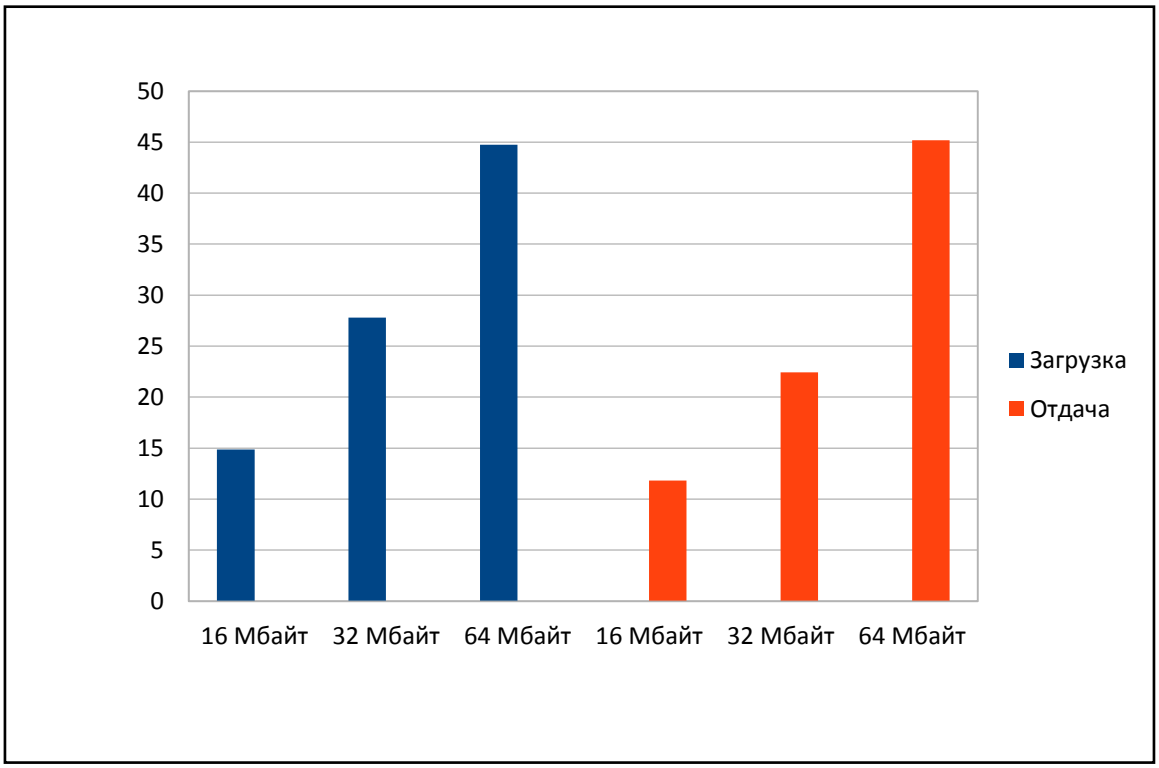


Рисунок 26: результат работы клиента №6 (пропускная способность канала связи 12 Мбит/сек)

На приведенных выше диаграммах видно, что файлы, которые больше, чем какие–то иные в N раз потребовали для своей пересылки примерно в N раз больше времени (пропорциональная зависимость), что является явным признаком последовательной передачи файлов. Программа отработала корректно, и обмен необходимыми файлами был произведен успешно.

Последующие тесты проводились с использованием метода разделения потока данных «по времени передачи» (см. Глава 5, пар. 2) с переменной величиной пропускной способности канала связи. Это позволит наглядно продемонстрировать тот факт, что данный метод не является «приоритетно справедливым» при параллельном приоритетном обмене файлами с переменной пропускной способностью.

Тест №6

Этот тест – подобен Тесту №1, однако, в данном тесте обмен файлами будет производиться параллельно с методом разделения потока данных «по времени передачи». Предполагается, что файлы, имеющие больший приоритет, не всегда будут загружаться быстрее, чем те файлы, что имеют меньший приоритет.

Рассмотрим и проанализируем результаты работы двух случайно выбранных клиентов: 3го и 7го.

Результаты работы клиента №3 представлены на диаграмме ниже.

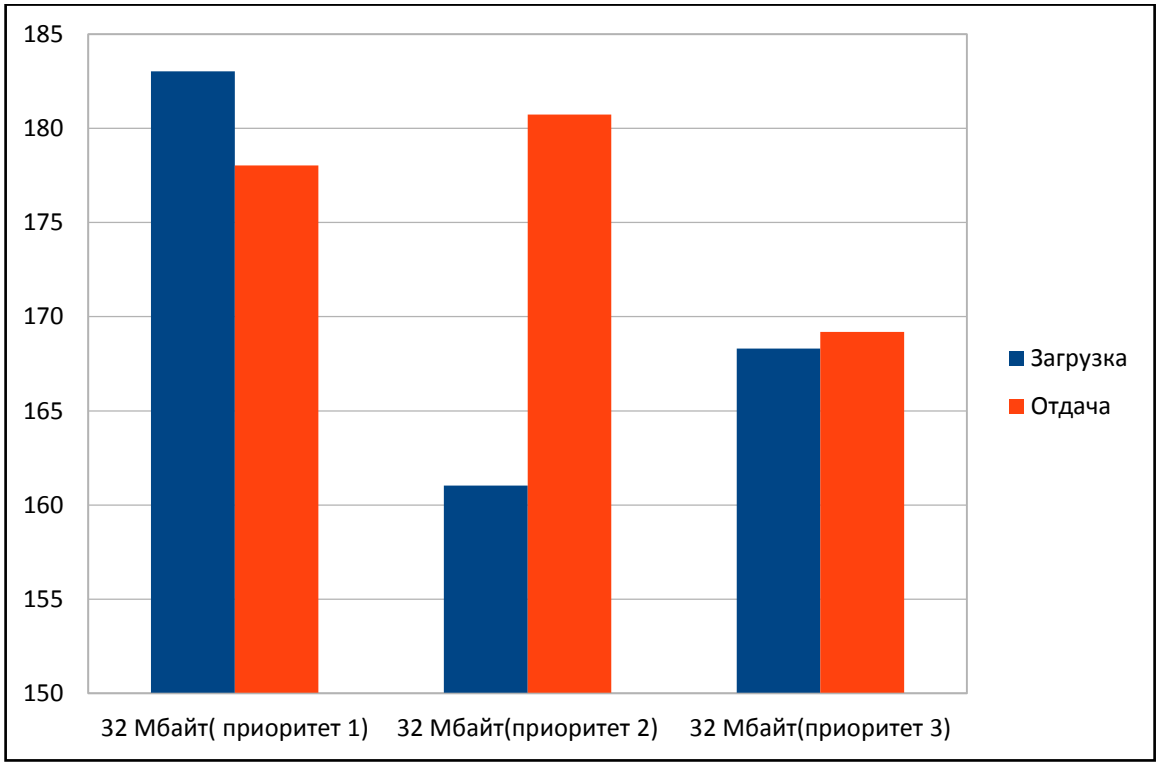


Рисунок 27: результат работы клиента №3 (пропускная способность канала связи 6 Мбит/сек)

Как можно видеть на диаграмме, при загрузке файл из третьей колонки имел больший приоритет, чем файл из второй колонки, однако, на его загрузку потребовалось больше времени, чем для загрузки файла с меньшим приоритетом. Аналогично произошло с первым и вторым файлом, которые были выгружены на сервер.

Результаты работы клиента №5 представлены на диаграмме ниже.

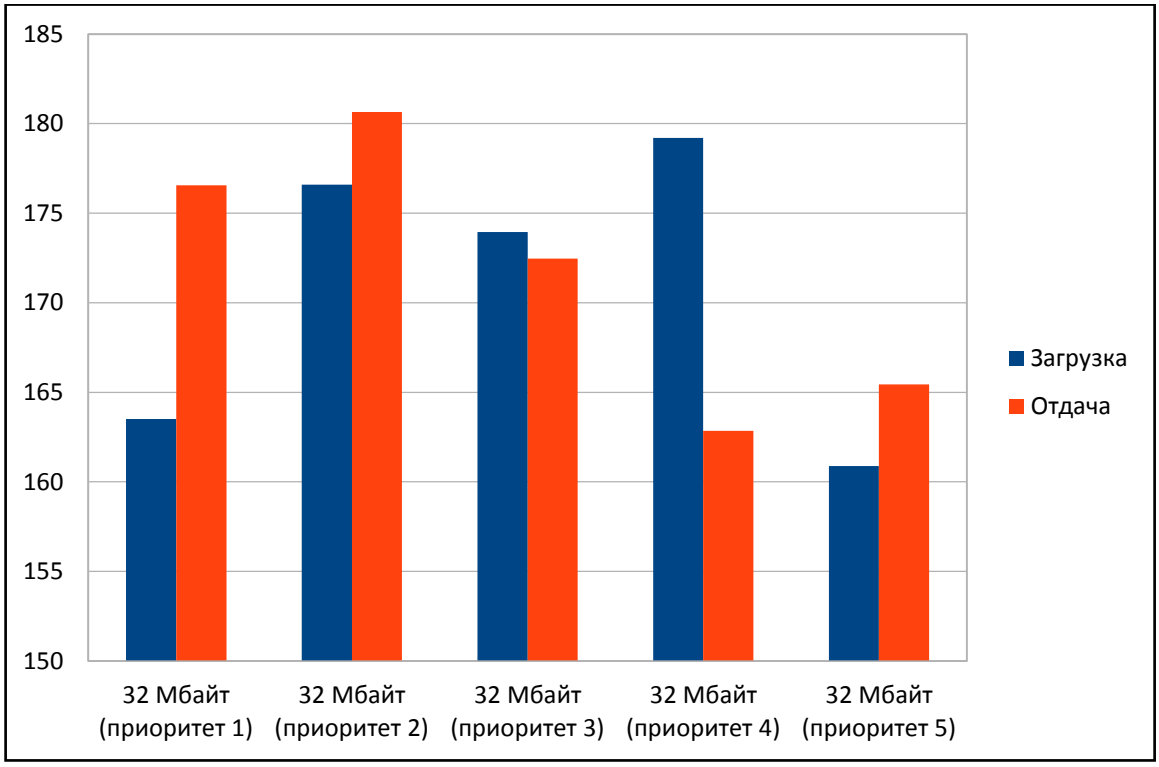


Рисунок 28: результаты работы клиента №5 (пропускная способность канала связи 10 Мбит/сек)

На диаграмме выше видно, что вне зависимости от направления обмена файлами, установленный файлам приоритет учитывается, но не так, как нужно. Так, например, при загрузке, файл, имеющий приоритет 4 загрузился позднее, чем файл, имеющий приоритет 1, хотя файлы имеют одинаковый размер.

Данный тест позволил увидеть «приоритетную несправедливость» при использовании метода разделения потока данных не «по кадрам передачи», а «по времени передачи».

Так же было проведено еще несколько тестов с использованием метода разделения потока «по времени передачи» и все они подтвердили то, что данный метод является некорректным при приоритетной передаче файлов, в чем мы убедились при рассмотрении результатов предыдущего теста, поэтому их результаты приведены не будут, однако с ними можно ознакомиться в приложениях к данной научной работе.

§3. Вывод на основе проведенного тестирования

Как и предполагалось, результаты проведенного тестирования подтвердили, что:

- метод разделения потока данных «по кадрам передачи» удовлетворяет всем предъявленным к методу требованиям в Главе 1, пар. 1, а именно: позволяет параллельно обмениваться файлами; при приоритетном обмене обеспечивает соответствующую разницу во времени передачи файлов; «справедливо» разделяет поток относительно расставленных пользователем приоритетов.
- метод разделения потока данных «по времени передачи» удовлетворяет НЕ всем предъявленным к методу требованиям в Главе 1, пар. 1, а именно: позволяет параллельно обмениваться файлами; при приоритетном обмене обеспечивает соответствующую разницу во времени передачи файлов. Однако данный метод «несправедливо» разделяет поток относительно расставленных пользователем приоритетов, то есть скорость передачи файлов с расставленными приоритетами не соотносится так, как соотносятся между собой приоритеты, следовательно, данный метод не будет использоваться в разрабатываемом приложении.
- клиентская сторона разработанного приложения, при использовании им метода разделения потока данных «по кадрам передачи», соответствует всем предъявленным функциональным требованиям к ней в Главе 1, пар. 3.
- серверная сторона разработанного приложения соответствует всем предъявленным функциональным требованиям к ней в Главе 1, пар. 2.

Результаты проведенного тестирования показали, что разработанное клиент–серверное приложение и разработанный метод разделения потока данных соответствуют всем предъявленным к ним требованиям в Главе 2. Так же, было выявлено, что использование метода разделения потока данных «по времени передачи» действительно является нерациональным в контексте поставленной задачи, в то время как способ разделения потока данных «по кадрам передачи» проявил себя как один из возможных методов решения проблемы приоритетного обмена файлами.

РЕЗУЛЬТАТЫ

При выполнении данной научной работы была поставлена цель, суть которой заключается в предоставлении пользователю персонального компьютера возможности обмениваться файлами большого размера с другим компьютером (суперкомпьютером) параллельно, в обе стороны (прием/передача), с возможностью задания приоритетов принимаемым и передаваемым файлам (загрузкам/выгрузкам). В связи с этим была поставлена задача разработки метода разделения потока данных при параллельном двунаправленном приоритетном обмене файлами по FTP–протоколу, а также задача разработки программного обеспечения клиент–серверной архитектуры, позволяющего осуществлять параллельный двунаправленный приоритетный обмен файлами большого размера, в основу работы которого заложен разработанный метод разделения потока данных.

Для выполнения данной задачи были рассмотрены два основных метода разделения потока данных («по времени передачи» и «по кадрам передачи»), после чего было выявлено, что метод разделения данных «по времени передачи» не обладает необходимым качеством «приоритетной справедливости» по отношению к файлам при переменной пропускной способности канала связи, участвующим в обмене. Было принято решение использовать метод разделения потока данных «по кадрам передачи», с некоторой доработкой, удовлетворяющий требованиям, которые были к нему выдвинуты.

После того, как был разработан метод разделения потока данных, началась разработка программного обеспечения, реализующего данный метод. Суммарно, для реализации функциональной стороны программного обеспечения, было написано около 1800 строк кода на языке Python 3, и около 700 строк кода на этом же языке с использованием платформы Qt 5 для реализации графического интерфейса.

Разработанное программное обеспечение было протестировано с использованием различных параметров (подробнее см. Глава 8). В ходе тестирования было подтверждено, что метод разделения потока данных «по времени передачи» не является «приоритетно справедливым» по отношению к передаваемым/принимаемым файлам при переменной пропускной способности канала связи, а также были получены результаты работы программного обеспечения с использованием метода разделения потока данных «по кадрам передачи». Проведя их анализ, было выявлено, что разработанное программное обеспечение и реализованный в нем метод разделения потока данных работают так, как было задумано при их проектировании, а также разработанное программное обеспечение удовлетворяет выдвинутым к нему требованиям.

В связи со всем вышеописанным можно сделать вывод о том, что цель данной работы была достигнута, то есть пользователю персонального компьютера предоставляется возможность обмениваться файлами с другим компьютером (суперкомпьютером) параллельно, двунаправленно, с возможностью задания приоритетов в виде использования разработанного программного обеспечения.

ЗАКЛЮЧЕНИЕ

Разработанное в ходе данной работы программное обеспечение можно развивать дальше, добавляя в него различный функционал, например, поиск файлов в удалённой рабочей директории пользователя, загрузка/выгрузка целых каталогов с файлами, постановка на паузу/возобновление приёма/передачи сразу нескольких файлов, присвоение пользовательских меток загружаемым и передаваемым файлам и т. д. Также, возможно добавление и доработка визуального оформления и дополнения к текущему графическому интерфейсу.

Стоить отметить, что при разработке программного обеспечения не рассматривался аспект безопасности передаваемых данных, поэтому в качестве дальнейшей доработки данного программного обеспечения можно рассматривать применение криптографического протокола SSL для шифрования передаваемых данных.

Также хотелось бы выразить благодарность сотруднику лаборатории программного оборудования факультета вычислительной математики и кибернетики (ВМК) МГУ имени М. В. Ломоносова, Масленникову Арсению Андреевичу, за помощь в проведении тестирования разработанного приложения, и, научному руководителю, кандидату физико-математических наук, ведущему научному сотруднику лаборатории вычислительной электродинамики ВМК МГУ имени М. В. Ломоносова, Сальникову Алексею Николаевичу, за своевременное консультирование и значительное расширение кругозора в области проводимого исследования.

СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

1. Александров Д. Е. Многопоточные сервера, использующие обработчики событий // Интеллектуальные системы. – 2013. – Т. 17, № 1 – 4, Секция «Защита информации». – С. 219 – 262.
2. Курин Е. А. Сейсморазведка и суперкомпьютеры // Вычислительные методы и программирование. – 2011. – Т. 12, № 1, Раздел «Вычислительные методы и приложения». – С. 34 – 39.
3. Назипова Н. Н., Исаев Е. А., Корнилов В. В., Первухин Д. В., Морозова А. А., Горбунов А. А., Устинин М. Н. Большие данные в биоинформатике // Информационные и вычислительные технологии в биологии и медицине. – 2017. – Т. 12, № 1, Раздел «Математическая биология и биоинформатика». – С. 102 – 119.
4. Лутц М. Изучаем Python, 4 – е издание. – Пер. С англ. – Санкт – Петербург: Символ – Плюс, 2011. – 1280 с.
5. Таненбаум Э., Уэзеролл Д. Классика Computer Science. Компьютерные сети, 5–ое издание – Сакт – Петербург: Питер, 2012. – 960 с.
6. Шлее М. Qt 5.3 Профессиональное программирование на C++, в подлиннике. – Санкт – Петербург: БХВ – Петербург, 2015. – 928 с.
7. Эндрюс Г. Р. Основы многопоточного, параллельного и распределенного программирования: Пер. с англ. – М.: Изд – во "Вильямс", 2003. – 512 с.
8. Веб–ресурс официальной документации языка программирования Python 3. – [https://docs.python.org/3/]
9. Веб–ресурс официальной документации модуля Python pyftpdlib для разработки асинхронного FTP – сервера. – [https://pyftpdlib.readthedocs.io/en/latest/]

10. Веб–ресурс официальной документации модуля aioftp. – [http://aioftp.readthedocs.io/]
11. Веб–ресурс официальной документации программной платформы для создания пользовательского интерфейса PyQt 5. – [http://pyqt.sourceforge.net/Docs/PyQt5/]
12. Веб–ресурс: «Список всех зарезервированных TCP и UDP портов». – [https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers]
13. Веб–ресурс: «Boost.Asio C++ Network Programming. Глава 1: Приступая к работе с Boost.Asio. Пример простейшего эхо – сервера». – [https://habrahabr.ru/post/192284/]
14. Веб–ресурс: «Разработка асинхронного HTTP сервера на Python с использованием asyncio». – [https://habrahabr.ru/post/217143/]
15. Веб–ресурс: «Пример асинхронного чат – сервера на Python с использованием Asyncio». – [https://gist.github.com/gregvish/7665915]
16. Веб–ресурс официальной документации Windows Forms. – [https://docs.microsoft.com/ru – ru/dotnet/framework/winforms/]
17. Веб–ресурс официальной документации Windows Presentation Foundation. – [https://docs.microsoft.com/ru – ru/dotnet/framework/wpf/]
18. Веб–ресурс официальной документации модуля ftplib. – [https://docs.python.org/3/library/ftplib.html]
19. Веб–ресурс русскоязычной официальной документации веб-сервера Nginx. – [https://nginx.org/ru/]
20. Официальный веб–ресурс веб-сервера Apache 2.0. – [https://httpd.apache.org/]
21. Веб–ресурс «Man-page for Linux epoll». – [http://man7.org/linux/man-pages/man7/epoll.7.html]
22. Веб–ресурс «epoll Scalability web page». – [http://lse.sourceforge.net/epoll/index.html]
23. Веб–ресурс «About WinSCP». – [https://winscp.net/eng/docs/introduction]

24. Веб–ресурс «FileZilla – The free FTP solution». – [https://filezilla-project.org/]
25. Веб–ресурс «Web server performance comparison». – [https://help.dreamhost.com/hc/en-us/articles/215945987-Web-server-performance-comparison]
26. Веб–ресурс «RFC-959 – File Transfer Protocol». – [https://tools.ietf.org/html/rfc959]