**BC/NW 2021№ 2 (37):7.1**

**ИССЛЕДОВАНИЕ ЗАВИСИМОСТИ ВРЕМЕНИ ПЕРЕДАЧИ ДАННЫХ ПО СЕТИ ОТ ИСПОЛЬЗУЕМЫХ ПРОТОКОЛОВ И ПРОГРАММНЫХ ИНТЕРФЕЙСОВ С ПРИМЕНЕНИЕМ ТЕХНОЛОГИЙ**

**БИБЛИОТЕК .NET**

Абросимов Л.И., Стеклов В.В.

В настоящее время огромное количество устройств общаются между собой, используя сеть передачи данных. Участники сетевого взаимодействия обязаны согласовать друг с другом множество правил, которые объединяются в протоколы передачи данных. Протоколы нижних уровней сетевой модели OSI часто реализуются с использованием программных и аппаратных средств.[1] Все протоколы, начиная с транспортного уровня и выше, реализуются программными средствами.[2]

В наше время очень актуальна проблема быстрой и надежной передачи информации.

Повысить эффективность транспорта данных достаточно сложно. В зависимости от используемых транспортных протоколов время и качество доставки информации может сильно отличаться.

Таким образом, в научной работе В настоящей статье определяются зависимости времени передачи данных от используемых протоколов транспортного и прикладного уровней. Эффективность передачи информации по сети во многом зависит от класса передаваемой информации. В исследовательской работе статье рассматривается осуществляется передача графических изображений между двумя компьютерами, находящимися в одной локальной сети.

К В разрабатываемой модели передачи данных для определениях качества обслуживания определяются качественные оценки обслуживания . Чаще всего параметры, фигурирующие в разнообразных определениях качества обслуживания, регламентируют используются следующие показатели работы сети:[1]

— скорость передачи данных;

— задержки передачи пакетов;

— уровень потерь и искажений пакетов.

Относительно этих оценок На основании этих показателей осуществляется выбор наилучшего рассмотренного метода передачи информации.

Простейшая модель передачи данных разрабатывается реализуется на языке программирования C# с использованием технологий библиотек .NET. В работе также исследуется роль сжатия графической информации для её эффективной передачи по каналам связи.

Выбранная тема исследовательской работы: «Исследование зависимости времени передачи данных по сети от используемых протоколов и программных интерфейсов с применением технологий библиотек .NET».

В предыдущей части исследовательской процессе работы были рассмотрены основные протоколы передачи данных по сети, входящие в библиотеку .NET. содержат в себе большое количество различных классов, реализующих сетевое взаимодействие. В основном эти классы располагаются в пространстве имен System.Net.Sockets. Помимо этого были выделены основные отличия, преимущества и недостатки каждого протокола передачи данных. Была описана модель передачи данных, на которой в будущем будут устанавливаться различные эксперименты, связанные с определением времени передачи.

В настоящей главе статье приводится пример программы, которая реализует сетевое взаимодействие двух компьютеров, находящихся в одной локальной сети. Работа программы не зависит от типа передаваемых файлов. Интернет – взаимодействие строится с использованием реализаций TCP и UDP протоколов в .NET.

Таким образом, в результате на данном этапе мы получаем программу, которая реализует взаимодействие нескольких компьютеров, используя через протоколы TCP и/или UDP, позволяющую передавать разнотипные файлы по локальной сети. При тестировании программы будем передавать разные файлы и фиксировать время их доставки.

**Разработка модели передачи данных**

В процессе разработки модели передачи данных создаются две программы, которые располагаются на разных компьютерах. Программы создаются в виде консольного приложения, причем при запуске программ появляется специальное меню, в котором можно выбрать тип подключения. Для правильного взаимодействия между устройствами требуется выбирать согласованный тип протокола: или UDP, или TCP.

Программа клиента осуществляет считывание файла из файловой системы и отправку его в сеть. Сервер ожидает подключение от клиента, после этого принимает файл и сохраняет его у себя в дисковом пространстве. Алгоритмы методов приема и передачи файлов у клиента и сервера представлены ниже.

Программы разрабатываются на языке C#. В программе клиента определим базовый абстрактный класс Client, который определяет интерфейс для будущих наследников, собственных реализаций TcpClient и UdpClient. Создание базового класса вместо интерфейса в данном случае оправдано тем, что сокращает дублирование повторяющегося кода.

using System;

using System.Net;

namespace Client.Clients

{

public abstract class Client

{

protected readonly IPAddress serverAddress;

protected readonly int serverPort;

protected int bufferSize = 512;

public Client(string serverAddress, int serverPort)

{

if (!IPAddress.TryParse(serverAddress, out var iPAddress))

{

throw new InvalidOperationException(

$"String {serverAddress} is not ip address");

}

this.serverAddress = iPAddress;

this.serverPort = serverPort;

}

public void SetBufferSize(int bufferSize) =>

this.bufferSize = bufferSize;

public abstract void SendFile(string fileName);

}

}

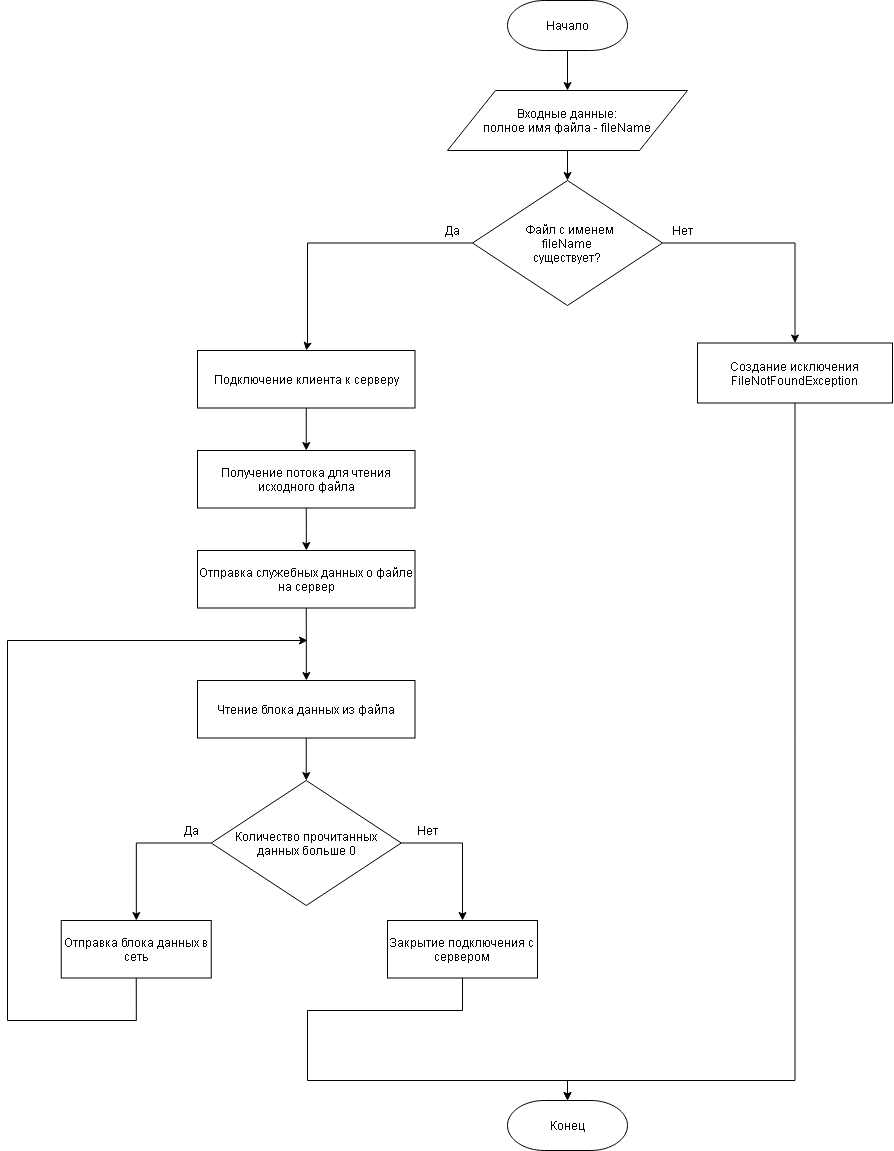


Рис 1. Алгоритм передачи файла для клиента

Клиент обязательно должен иметь адрес и номер порта сервера, к которому он хочет подключиться.[1] Эти поля устанавливаются в конструкторе базового класса клиента. Предложенный алгоритм передачи данных подразумевает разбиение исходного файла на части. Программа считывает часть файла в специальный буфер, размер которого определяется переменной bufferSize. Далее происходит отправка этого буфера в сеть. Данный процесс происходит в цикле, пока весь файл не будет полностью считан. Соответствующий алгоритм считывания и передачи файла в программе клиента приведен выше. Причем данный алгоритм разрабатывается и для TcpClient, и для UdpClient.

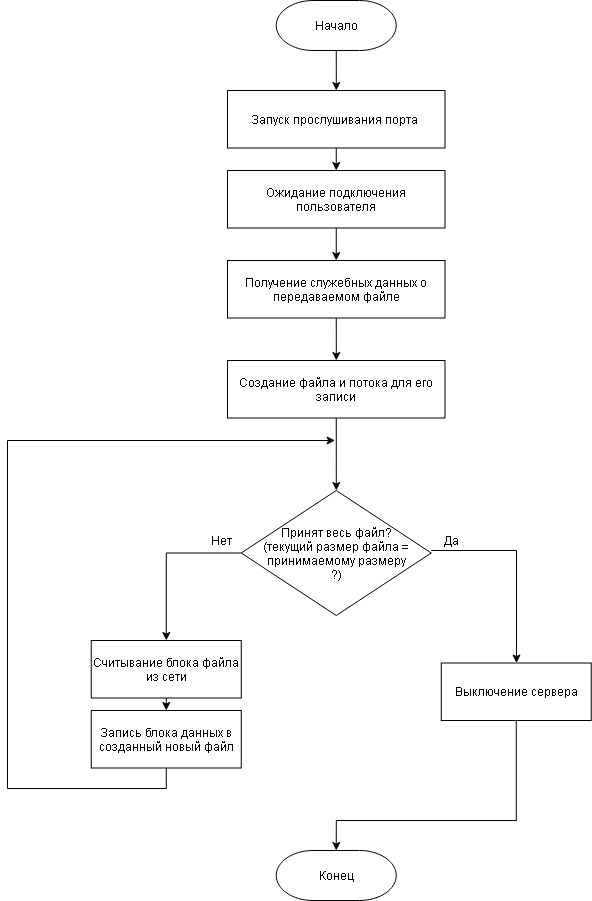


Рис 2. Алгоритм приема файла для сервера

Алгоритм приема файла для сервера диаметрально противоположен с алгоритмом клиента. Вначале происходит считывание потока байтов в класс FileInfo. Этот класс содержит в себе информацию о передаваемом файле (его имя и размер). Далее происходит поблочное считывание файла и загрузка блоков данных в файловый поток, при этом файловый поток создает новый файл на диске сервера.

**1.1 Реализация UDP - взаимодействия**

UDP — это протокол эффективной передачи данных в удаленный узел. UDP не предусматривает установление соединения. Доставка пакетов, отправляемых в удаленную конечную точку, не гарантируется так же, как не гарантируется их очередность при приёме. Обычно приложения, использующие протокол UDP, должны самостоятельно обрабатывать отсутствующие, повторяющиеся и идущие не по порядку пакеты, т.к. реализация протокола не подразумевает использование сигналов подтверждения как в TCP.[2] Но в разрабатываемой программе мы не будем реализовывать данную функциональность.

Класс UdpClient предоставляет простые методы для отправки и получения датаграмм UDP без подключения в блокирующем синхронном режиме. Так как UDP является транспортным протоколом без подключения, устанавливать подключение к удаленному узлу перед отправкой и получением данных не требуется.[3] То есть вызов метода Connect для UDP не происходит, как это делается, например, в TCP. Приведем пример реализации метода SendFile. Реализация метода удовлетворяет алгоритму клиента, который был представлен выше.

public override void SendFile(string fileName)

{

using (var inputFileStream = new FileStream(fileName, FileMode.Open))

{

SendFileInfo(inputFileStream);

SendFile(inputFileStream);

}

udpClient.Close();

}

private void SendFileInfo(FileStream fileStream)

{

var binaryFormatter = new BinaryFormatter();

var fileInfo = new WebProtocolsModel.FileInfo

{

FileName = fileStream.Name,

FileSize = fileStream.Length

};

var stream = new MemoryStream();

binaryFormatter.Serialize(stream, fileInfo);

stream.Position = 0;

var bytes = new byte[stream.Length];

stream.Read(bytes, 0, Convert.ToInt32(stream.Length));

udpClient.Send(bytes, bytes.Length, iPEndPoint);

}

private void SendFile(FileStream fileStream)

{

using (var binaryReader = new BinaryReader(fileStream))

{

byte[] buffer = new byte[bufferSize];

int count;

while ((count = binaryReader.Read(buffer, 0, bufferSize)) > 0)

{

udpClient.Send(buffer, count, iPEndPoint);

}

}

}

В начале метода происходит получение читающего потока для файла с полным именем filename, которое передается в параметрах. Далее происходит отправка служебной информации о файле. Информация о файле сериализуется в промежуточный поток с помощью двоичного преобразователя. Для отправки данных в сеть используется метод Send, который принимает массив байтов, полученный из промежуточного потока.

После передачи служебной информации о файле происходит отправка самого файла. Для этого из файла с помощью экземпляра класса BinaryReader происходит последовательное считывание блоков байтов и передача их в сеть.

На стороне сервера используется также экземпляр класса UdpClient. Считывание данных осуществляется с помощью метода Receive, который блокирует поток программы. Блокировка происходит пока следующий блок данных не придет на сервер. Реализация класса UdpServer представлена в приложении А.

***1.2 Реализация TCP - взаимодействия***

Протокол TCP – это сетевой протокол транспортного уровня, ориентированный на соединение между двумя хостами. Для установки соединения TCP использует механизм “тройного рукопожатия”, который заключается в последовательном обмене между устройствами специальных пакетов. Протокол TCP гарантирует целостность передаваемых данных, устраняя дублирование пакетов и сохраняя порядок их отправки.[2]

Чтобы передать данные по сети с использованием протокола TCP, в .NET используется класс TcpClient. Данный класс также требует адрес сетевого устройства, в котором размещается нужная служба, и порт, который служба использует для обмена данными.[3]

Сервер в свою очередь использует класс TcpListener. Данный класс используется для отслеживания входящих запросов. Метод  Start  включает прослушивание порта, а метод  Stop  отключает его. Метод  AcceptTcpClient  блокирует поток исполнения, принимает входящие запросы на подключение и создает экземпляр TcpClient для их обработки.

private WebProtocolsModel.FileInfo GetFileInfo(

NetworkStream networkStream)

{

var binaryFormatter = new BinaryFormatter();

var fileInfo = (WebProtocolsModel.FileInfo)binaryFormatter.Deserialize(networkStream);

fileInfo.FileName = fileInfo.FileName.Replace(

oldValue: "Client",

newValue: "Server");

return fileInfo;

}

private void SaveFile(

NetworkStream networkStream,

WebProtocolsModel.FileInfo fileInfo)

{

using (var binaryWritter = new BinaryWriter(

newFileStream(fileInfo.FileName, FileMode.Create)))

{

byte[] buffer = new byte[bufferSize];

for (int i = 0; i < fileInfo.FileSize; i += bufferSize)

{

networkStream.Read(buffer, 0, bufferSize);

binaryWritter.Write(buffer);

}

}

}

Как видно из примера, принцип работы с протоколом TCP несколько отличается от UDP в .NET. Реализация TCP подразумевает работу с непрерывным потоком байтов. Для TcpClient мы получаем соответствующий ему сетевой поток (в коде это объект класса NetworkStream) и с помощью методов Read и Send, характерных для потоков, осуществляем чтение и запись данных. Полный код клиента и сервера также представлен в приложении А.

**2. Результаты работы и тестирование программы**

Программы клиента и сервера выполнены в виде консольного приложения. Интерфейсы этих программ представлены на рисунке 3 и 4 соответственно. Как было сказано ранее, для правильного взаимодействия требуется выбирать один и тот же тип подключения у сервера и у клиента.

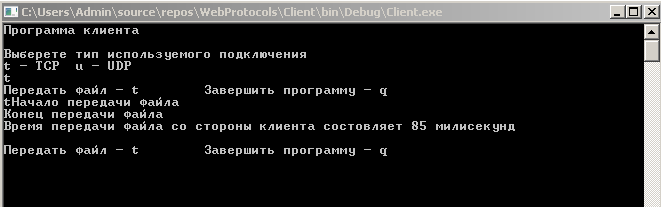


Рис 3. Пользовательский интерфейс программы клиента

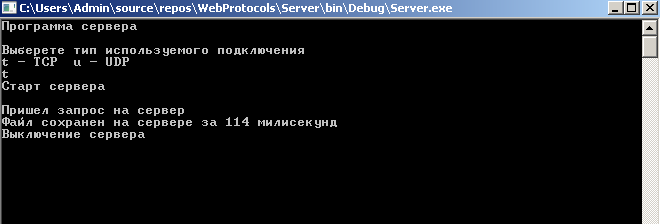


Рис 4. Пользовательский интерфейс программы сервера

В предложенном алгоритме данные считываются в виде массива байт. Работа с файлами и с сетью осуществляется с помощью вспомогательных классов-преобразователей, которые форматируют исходные данные в байты информации. Таким образом, программа не зависит от типа передаваемого файла. Одинаковым образом будут передаваться как изображения, так и pdf, doc – документы.

В ходе тестирования будем передавать разные типы файлов: текстовые документы в форматах txt, doc, а также изображения в форматах png и jpg. Будем фиксировать время передачи данных у клиента и у сервера. На стороне сервера также будем проверять целостность передаваемых данных. Результаты тестирования представлены в таблице 1.

Таблица 1. Результаты тестирования программы

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Тип файла | Размер файла  (байты) | Тип подключения | Время клиента  (мс) | Время сервера  (мс) |
| Текстовый документ в формате txt | 3 605 | TCP | 6 | 12 |
| Текстовый документ в формате txt | 3 605 | UDP | 3 | 7 |
| Текстовый документ в формате doc | 147 773 | TCP | 22 | 30 |
| Текстовый документ в формате doc | 147 773 | UDP | 6 | - |
| Изображение png | 621 004 | TCP | 45 | 48 |
| Изображение png | 621 004 | UDP | 20 | - |
| Изображение jpg | 755 637 | TCP | 51 | 60 |
| Изображение jpg | 755 637 | UDP | 33 | - |
| Изображение jpg | 1 102 805 | TCP | 58 | 65 |
| Изображение jpg | 1 102 805 | UDP | 32 | - |

**3. Заключение**

Таким образом, в данной практической работе была создана программа, реализующая сетевое взаимодействие между двумя устройствами через протоколы TCP и UDP. Для сервера и клиента были предложены и реализованы алгоритмы передачи данных, которые подразумевают чтение, запись, передачу байтовых блоков.

Были показаны сходства и различия в реализации TCP и UDP протоколов в .NET. Например, передача данных по TCP протоколу осуществляется с помощью объекта сетевого потока. Для UDP взаимодействия в .NET отсутствуют многие классы и методы, которые есть в TCP (например, класс TcpListener).

В ходе тестирования был получен ожидаемый результат – с увеличением размера передаваемого файла, происходит увеличение времени его передачи. Передача данных через протокол UDP осуществляется значительно быстрее, чем через TCP.

При тестировании также проверялась целостность переданных файлов на сервере. Было замечено, что целостность файла при его передаче через протокол UDP теряется. Ниже представлено изображение исходного файла, изначально находящегося на клиенте, который впоследствии передается на сервер.



Рис 5. Пример исходного передаваемого файла.

Использование TCP гарантирует целостность передаваемых данных. Полученный результат на сервере соответствует исходному файлу. Но, как можно заметить, переданный файл через UDP протокол является искаженным. Таким образом, несмотря на скорость передачи, для UDP соединения требуется дополнительные методы, позволяющие достигать целостности передаваемых данных.[3] Из таблицы 1 также видно, что время работы сервера для UDP соединения не определено для больших файлов. Это связано с тем, что сервер со временем блокируется в ожидании потерянных датаграмм.

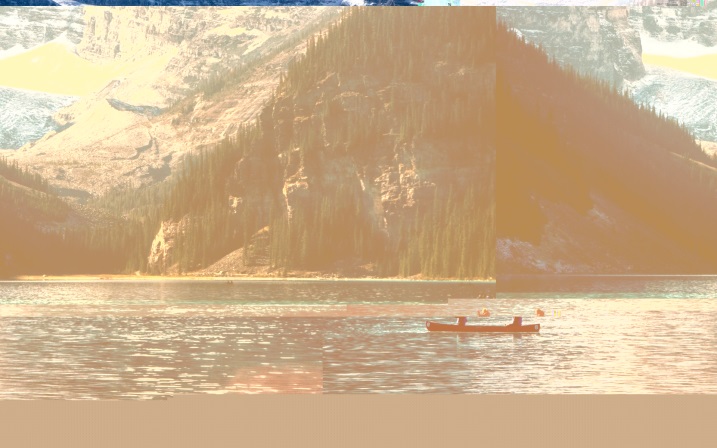


Рис 6. Переданные изображения (слева для TCP, справа для UDP подключений).

Можно заметить, что при увеличении размера считывающего/записывающего буфера время работы алгоритмов может быть изменено. Для больших файлов ожидается меньшее количество итераций считывания данных, вследствие чего время алгоритма будет уменьшаться. Для малых файлов изменения не будут заметны. Однако создание слишком большого буфера может повлечь за собой то, что избыточные байты в пакете будут отбрасываться.

Таким образом, несмотря на то, что целостность небольших файлов при передаче через UDP протокол не терялась, для описанных типов файлов рекомендуется использовать все же протокол TCP.

=============================

В предыдущей главе были продемонстрированы преимущества рассмотренных транспортных протоколов для передачи изображений по сети. В ходе тестирования были показаны ожидаемые результаты – качество изображения теряется при передачах по протоколу UDP. Большинство пакетов отправленных клиентом серверу были искажены в процессе передачи или вовсе потеряны.

В настоящей главе происходит разработка собственного потокового протокола передачи информации, <новую разработку нельзя называть известным именем> который представляет собой комбинацию существующих протоколов UDP и TCP. Описываются алгоритмы разрабатываемого протокола, оценивается время, затрачиваемое клиентом и сервером для передачи и приема соответственно. Временные характеристики протокола сравниваются с уже полученными результатами из предыдущей главы.

Помимо этого в данной главе отдельно рассматриваются сокеты. Изучается принцип разработки программы с использованием сокетов, отдельно оценивается роль сжатия информации на время передачи.

**1. Разработка надежного протокола UDP <**новую разработку нельзя называть известным именем>

Главное преимущество протокола TCP состоит в его надежности. Надежность данного протокола транспортного уровня обеспечивается, например, за счет установки соединения между устройствами, отправки подтверждений приема данных, доставки данных в определенном порядке. В это же время протокол UDP обеспечивает большую скорость, однако данное преимущество обесценивается тем фактом, что данные могут быть потеряны во время передачи. Стандарт НТТР не требует использования именно протокола TCP как основного. Мы можем передавать НТТР через датаграммный сокет или через любой другой. Но на практике весь НТТР трафик передается через TCP, благодаря удобству последнего.

В наше время протокол UDP используется приложениями не только для доставки датаграмм и широковещательных рассылок, но и для реализации одноранговой сети, в которой каждое устройство может быть представлено в виде клиента и сервера одновременно. Таким образом, у данного протокола появилось множество не запланированных ранее способов применения, но, недостатки протокола, такие как отсутствие гарантированной доставки, никуда при этом не исчезли.

Разрабатываемому приложению требуется гарантированная доставка изображений. В таком случае, для обеспечения гарантированной доставки пакетов, требуется реализовать протокол прикладного уровня, обеспечивающий необходимую функциональность и работающий поверх UDP.

Основная рассматриваемая задача в рамках выпускной квалификационной работы - это передача изображений по сети. Предлагается воспользоваться преимуществами каждого рассматриваемого протокола транспортного уровня - создать новый протокол, основанный на TCP и UDP и оценить результаты. Для этого рассмотрим более подробно преимущества каждого существующего протокола. Для разрабатываемого протокола предложим собственные варианты решения поступающих проблем.

Ниже представлены основные принципы и требования к разрабатываемому протоколу:

1. Надежная доставка пакетов

2. Передача изображений осуществляется поверх UDP

3. Передача подтверждений фактов приемки сообщений сервером осуществляется по протоколу TCP

4. Увеличенная пропускная способность канала

Гарантия доставки протоколом TCP обеспечивается за счёт механизма подтверждения полученных сервером сообщений.



Рис 1. Гарантия доставки

Для разрабатываемой программы введем еще один класс, который будет представлять собой подтверждающее сообщение (специальный пакет подтверждения). В сообщении сервер будет передавать номер пришедшего блока.

namespace WebProtocolsModel

{

[Serializable]

public class ConfirmMessage

{

public int BatchOrder { get; set; }

}

}

Очевидно, что нумерацию блоков необходимо производить на клиенте перед самой отправкой. Нумерация клиентом сегментов необходима не только для того, чтобы в будущем по пришедшему номеру определять факты приемки блока сервером. Стоит помнить, что для отправки изображений мы используем протокол UDP. Порядок следования сообщений в протоколе UDP не детерминирован. Сегменты изображений будут приходить в неопределенной последовательности. Несмотря на появление в разрабатываемом протоколе некоторой избыточности, введение нумерации пакетов оправдано. Таким образом, для сбора результирующей картинки серверу достаточно просто отсортировать пришедшие буферы, состоящие из байтов, в порядке их нумерации клиентом.

namespace WebProtocolsModel

{

[Serializable]

public class FileBatch

{

public byte[] Bytes { get; set; }

public int Order { get; set; }

}

}

Ниже, на рисунке 2 представлен алгоритм сервера. Видно, что сервер имеет стандартный алгоритм для UDP взаимодействия за исключением того, что к нему были добавлены несколько дополнительных блоков: блок подтверждения приема сообщения, блок сбора конечной информации. Отправка подтверждающих сообщений происходит после приема очередного сегмента. Из пришедшего сегмента извлекается его номер, создается подтверждающее сообщение с этим номером и отправляется клиенту именно по протоколу TCP. Применение протокола TCP в данный момент оправдано тем, что подтверждения не будут потеряны в процессе их передачи от сервера клиенту, а объём передаваемой информации по TCP не велик, что не приведет к значительному увеличению трафика.

private void SaveFile(

WebProtocolsModel.FileInfo fileInfo,

NetworkStream confirmingNetworkStream)

{

var remoteEndPoint = new IPEndPoint(IPAddress.Any, 0);

var fileBatches = new Dictionary<int, FileBatch>();

using (var binaryWritter = new BinaryWriter(new

FileStream(fileInfo.FileName, FileMode.Create)))

{

for (int i = 0; i < fileInfo.FileSize; i += bufferSize)

{

var data = server.Receive(ref remoteEndPoint);

var fileBatch = GetFileBatch(data);

fileBatches.Add(fileBatch.Order, fileBatch);

ConfirmReceiving(

fileBatch,

confirmingNetworkStream,

remoteEndPoint);

}

binaryWritter.Write(

fileBatches

.OrderBy(b => b.Key)

.SelectMany(b => b.Value.Bytes)

.ToArray());

}

}

Каким образом происходит сбор конечной картинки на сервере в условиях недетерминированного приема? Для того чтобы собрать картинку необходимо просто отсортировать пришедшие данные по их порядковому номеру, выданному клиенту перед отправкой.

Теперь подробнее рассмотрим роль и функции клиента в разрабатываемом взаимодействии. Как было сказано ранее, клиент должен уметь нумеровать исходные данные для отправки их серверу. Нумерация сегментов с байтами выполняется клиентом по мере их считывания из файловой системы. Но помимо этого у клиента при данном разрабатываемом взаимодействии появляются еще ряд сложных функций:

1. Прием подтверждений от сервера по протоколу TCP;

2. Обработка подтверждений;

3. Отправка потерянных сегментов заново по протоколу UDP.

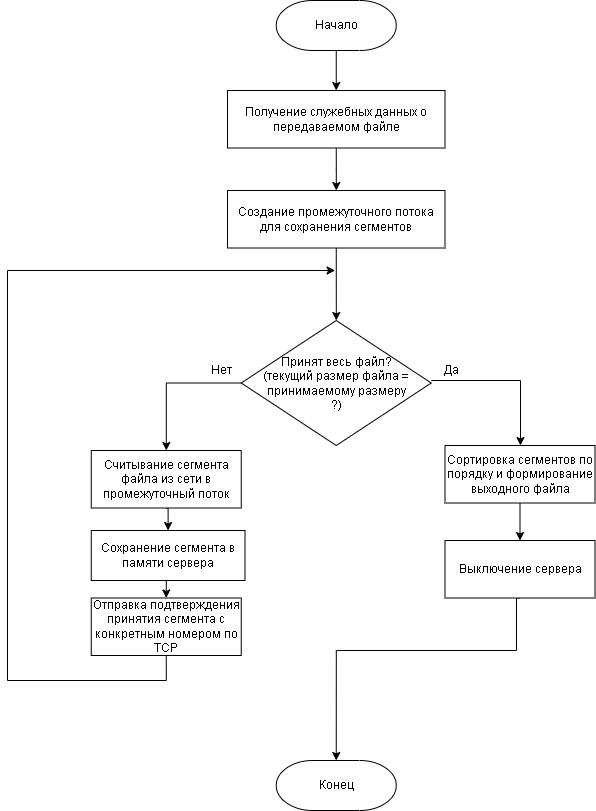


Рис 2. Алгоритм сервера

Принцип приема данных по протоколу TCP был подробно объяснен в предыдущих главах. Очевидно, что для принятия подтверждающих сообщений на клиенте необходимо завести экземпляр класса TcpListener. Данный класс представляет собой TCP сервер. Он будет ожидать и обрабатывать входящие подключения. Вся описанная логика должна быть вынесена в отдельный поток. Поток — это основная единица, которой операционная система выделяет время процессора. Если это не сделать, в момент, когда TCP сервер будет ожидать входящие подключения и подтверждающие сообщения, он заблокирует передачу сегментов с изображениями по UDP.

private void ReadConfirmations()

{

WaitBatches();

var confirmationsCounter = 0;

var client = confirmingTcpListener.AcceptTcpClient();

var networkStream = client.GetStream();

while (confirmationsCounter < fileBatches.Count)

{

var binaryFormatter = new BinaryFormatter();

if (networkStream.DataAvailable)

{

var confirmMessage =

binaryFormatter.Deserialize(networkStream) as

ConfirmMessage;

fileBatches[confirmMessage.BatchOrder].IsConfirmed =

true;

confirmationsCounter++;

}

else

{

RetrySendingBatches();

}

}

}

Рассмотрим случай, когда клиент принял сообщение о подтверждении сервером факта приемки конкретного блока. Тогда необходимо извлечь номер подтвержденного блока данных из сообщения и пометить его как подтвержденный уже на клиенте. Клиент гарантирует, что подтвержденные сервером сегменты не будут повторно отправлены ему по UDP еще раз.

Что происходит, когда клиент перестает принимать подтверждения от сервера? В этом случае начинает работать функция повторной отправки потерянных сегментов изображения.

private void RetrySendingBatches()

{

var unconfirmedBatches = fileBatches.Values.Where(b =>

!b.IsConfirmed).ToArray();

foreach (var batchInfo in unconfirmedBatches)

{

if (DateTime.UtcNow - batchInfo.LastRetryDateTimeUtc >

retryTimeSpan)

{

retryCounter++;

RetrySendingBatch(batchInfo, iPEndPoint);

}

}

}

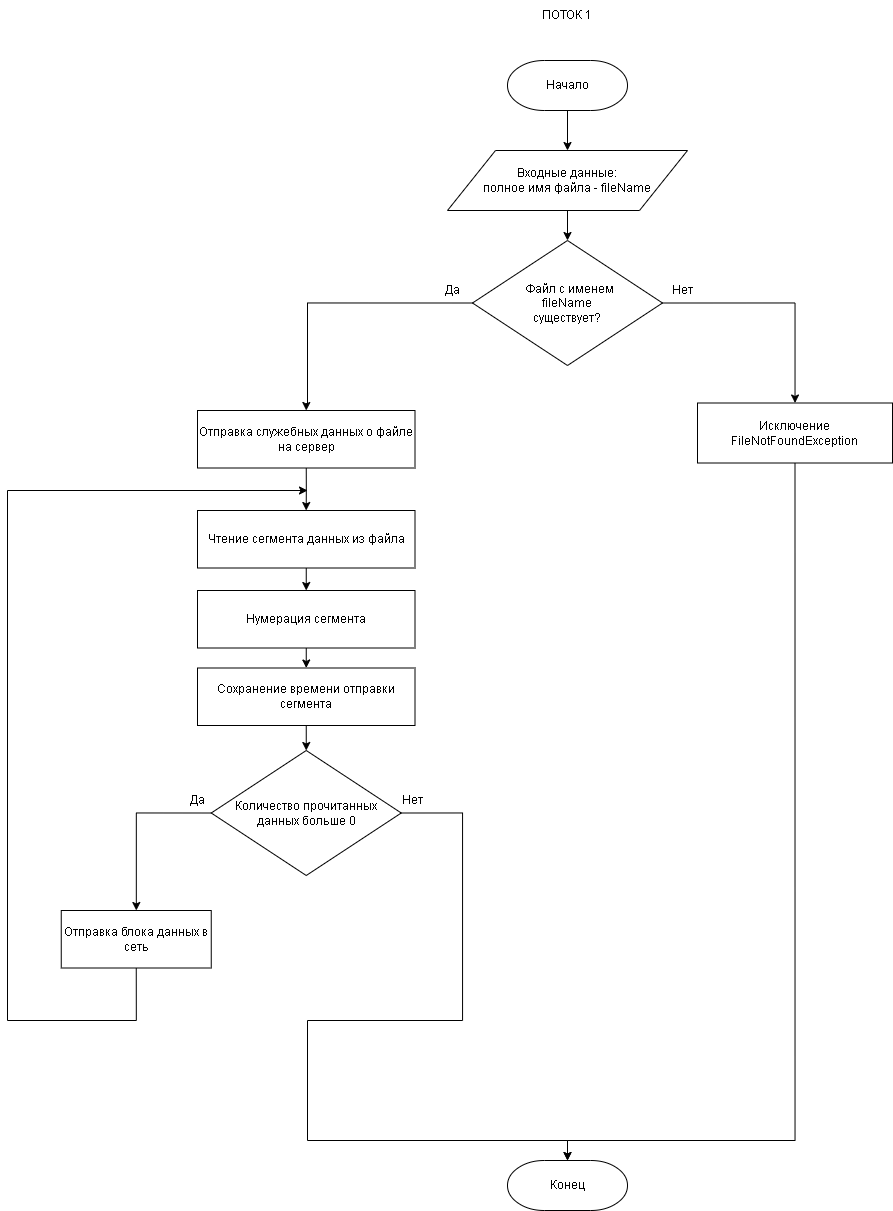


Рис 3. Алгоритм клиента. Поток 1.

Алгоритм поправить «спрямить» все соединительные линии алгоритма

Протокол TCP в качестве критерия необходимости повторной отправки данных использует механизм таймеров – для каждого сообщения задается время ожидания подтверждения, если в течение этого времени подтверждение не пришло, таймер срабатывает и потерянный сегмент отправляется еще раз.

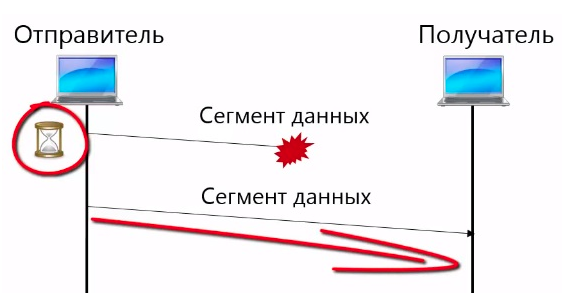


Рис 4. Потеря данных

Реализуем похожий механизм для нашего клиента. Для каждого сегмента будем сохранять время его отправки по протоколу UDP. В момент определения необходимости повторной отправки сегмента, будем сравнивать сохраненное время с текущим временем. Если оно больше порогового, то требуется повторная отправка сегмента по протоколу UDP. Если нет, то считаем, что время таймаута не прошло, а повторная отправка просто откладывается. Описанный алгоритм представлен на рисунках 3, 8 для потока 1 и потока 2 соответственно.

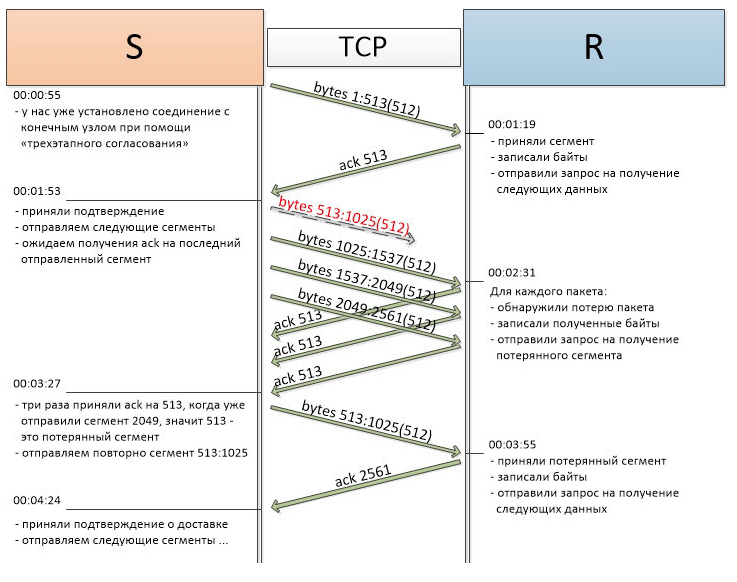


Рис 5. Механизм окна TCP

TCP для увеличения пропускной способности использует механизм скользящего окна. Отправитель передает сразу несколько порций данных, не дожидаясь подтверждения. Получатель отправляет одно общее подтверждение сразу на несколько сегментов.

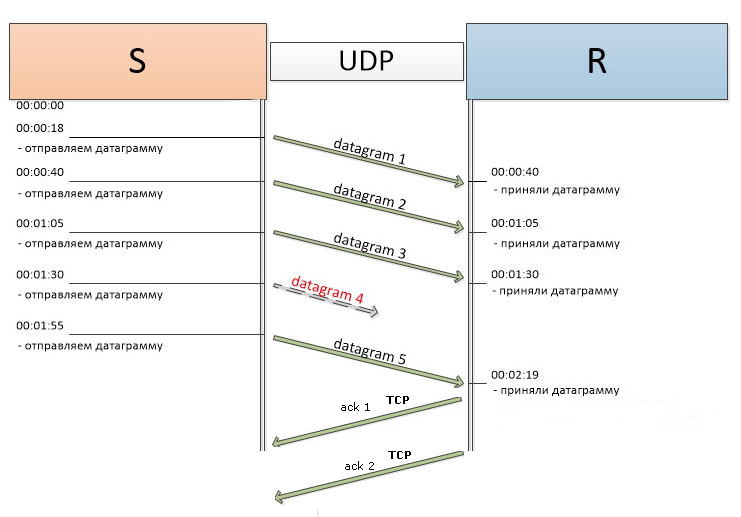


Рис 6. Диаграмма взаимодействия клиента и сервера разработанного протокола

В нашей реализации взаимодействия клиента и сервера окно не используется. Таким образом, теоретически клиент может отправить все датаграммы c изображением по протоколу UDP еще до того, как по TCP начнут приходить подтверждения.

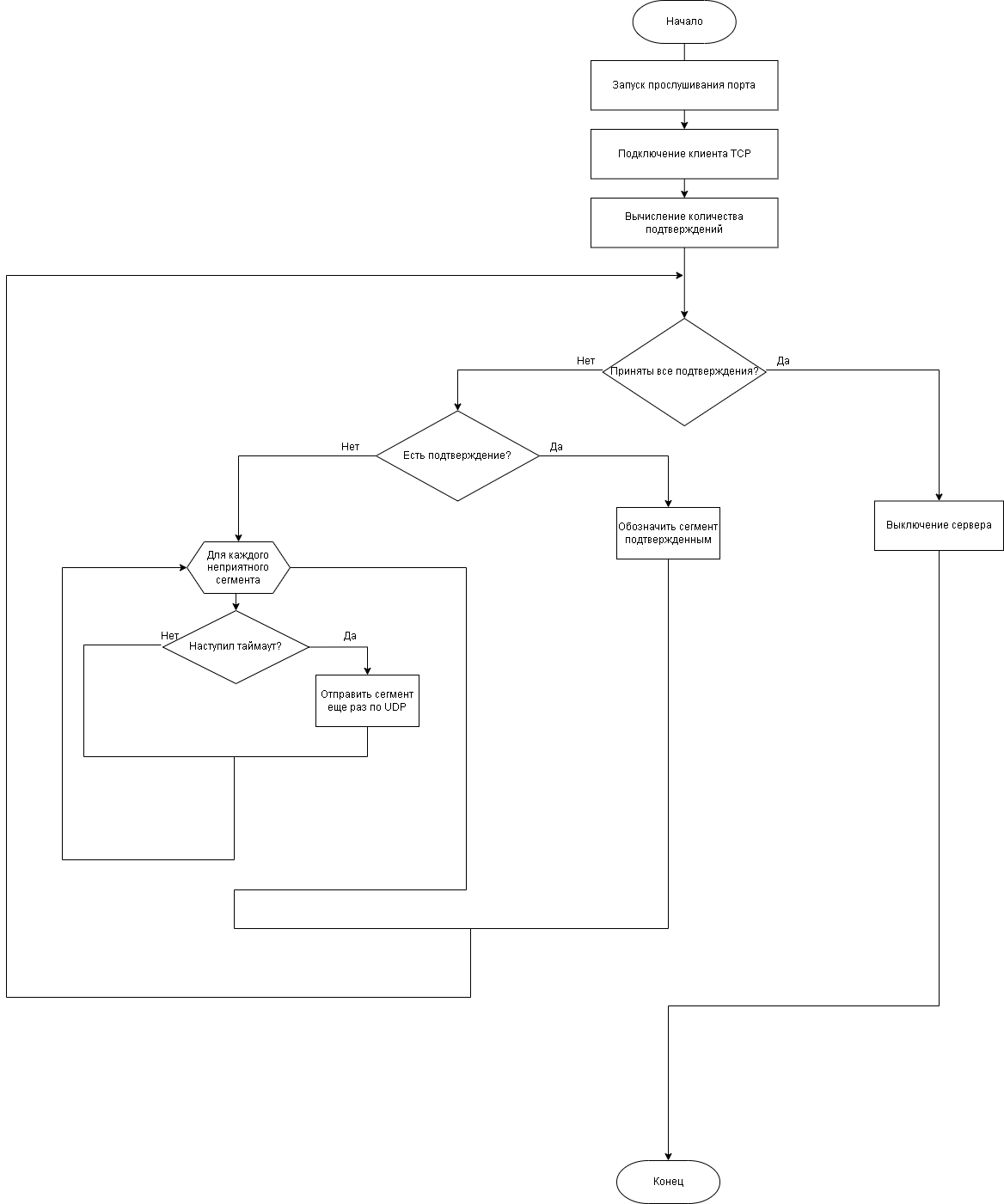


Рис 7. Алгоритм клиента. Поток 2 (Обработка подтверждений).

Алгоритм поправить: 1)Во **всех логических блоках необходимо указать условия перехлдов**

2) «спрямить» все соединительные линии алгоритма

**2. Тестирование и результаты работы протокола**

Программы клиента и сервера выполнены в виде консольного приложения. Интерфейсы этих программ представлены на рисунке 8. Передадим изображение по разработанному протоколу. Убедимся, что изображение успешно пришло на сервер.

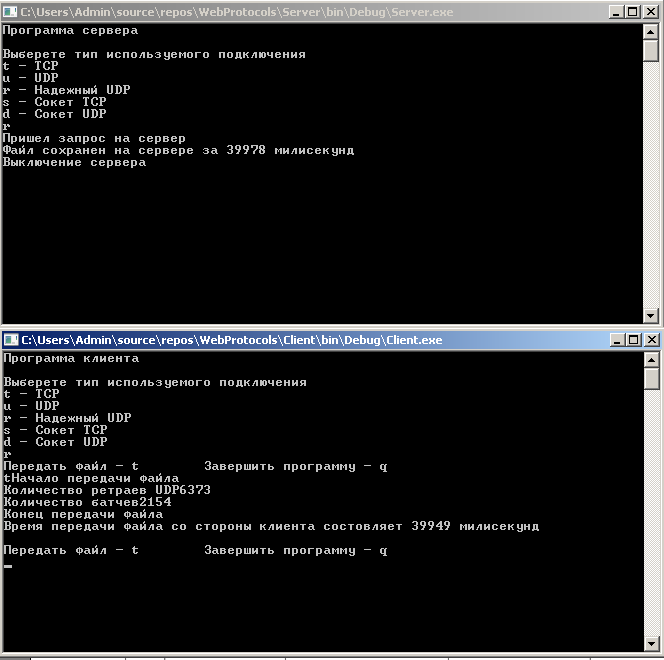


Рис 9. Интерфейс программы

Будем передавать различные изображения с разным размером и смотреть, как ведет себя протокол.

Таблица 1. Разный размер изображений

|  |  |  |  |
| --- | --- | --- | --- |
| Тип файла | Размер файла  (байты) | Время клиента  (мс) | Время сервера  (мс) |
| Изображение jpg | 147 773 | 10193 | 10198 |
| Изображение jpg | 621 004 | 26800 | 26813 |
| Изображение jpg | 755 637 | 31421 | 31500 |
| Изображение jpg | 1 102 805 | 39949 | 39978 |

Как было уже много раз сказано, изображение передается сегментами определенной длины. Попробуем изменить размеры входных и выходных буферов при фиксированной длине изображения **1 102 805 байт**.

Таблица 2. Разный размер буфера

|  |  |  |  |
| --- | --- | --- | --- |
| Количество сегментов в изображении/ Количество повторных отправок | Размер буфера  (байты) | Время клиента  (мс) | Время сервера  (мс) |
| 2154/4055 | 512 | 39949 | 39978 |
| 1077/2171 | 1024 | 28051 | 28082 |
| 539/625 | 2048 | 16674 | 16698 |
| 270/169 | 4096 | 11079 | 11069 |
| 135/97 | 8192 | 10500 | 10528 |

Необходимость повторной отправки сегмента клиентом на сервер определяется с помощью механизма таймаутов. Если подтверждение для сегмента с номером N не приходит в течении определенного времени, сегмент с номером N отправляется еще раз на сервер. Попробуем изменить время таймаута при фиксированной длине изображения **1 102 805 байт и буфере 2Кбайт.**

Таблица 3. Разное время таймаута

|  |  |  |
| --- | --- | --- |
| Время таймаута  (сек) | Время клиента  (мс) | Время сервера  (мс) |
| 1 | 4629 | 4619 |
| 2 | 9498 | 9200 |
| 5 | 16674 | 16698 |
| 8 | 28543 | 28557 |
| 10 | 31608 | 31115 |

**3. Принцип передачи информации по сокетам**

Сокет – некоторый абстрактный объект, представляющий конечную точку соединения. Реализация сокетов в .NET сделана поверх уже рассмотренных протоколов транспортного уровня и проткала IP сетевого уровня.

Потоковый сокет — это сокет с установленным соединением, состоящий из потока байтов, который может быть двунаправленным, т, е. через эту конечную точку приложение может и передавать, и получать данные. Данный тип сокета гарантирует исправление ошибок, обеспечивает порядок проходимых данных. Потоковой сокет базируется на протоколе TCP и подходит для задачи передачи изображений. Таким образом, если данные должны гарантировано доставляться другой стороне, и объем передаваемых данных достаточно велик (требуется установка постоянного соединения между узлами), то потоковые сокеты являются более предпочтительными. Принципы работы с сокетом изображены на рисунках 10 и 11 для клиента и сервера соответственно.



Рис 10. Общая схема работы клиентского сокета

Сокет клиента необходимо подключить к серверу. Для этого на клиенте вызывается метод Connect, в который передается информация об удаленной конечной точке, представляющая собой комбинацию IP – адреса и порта. Для передачи данных на сервер применяется метод Send. В нашем случае, вначале передается информация о файле (его название и размер), потом само изображение. Ниже представим код клиента:

public override void SendFile(string fileName)

{

if (!File.Exists(fileName))

{

throw new FileNotFoundException(fileName);

}

Console.Write("Подключение к удаленной точке\n");

socket.Connect(remoteIpPoint);

using (var inputFileStream = new FileStream(fileName,

FileMode.Open))

{

SendFileInfo(inputFileStream);

SendFile(inputFileStream);

//SendFileWithCompress(inputFileStream);

}

socket.Shutdown(SocketShutdown.Send);

socket.Close();

}

private void SendFileInfo(FileStream fileStream)

{

var fileInfo = new WebProtocolsModel.FileInfo

{

FileName = fileStream.Name,

FileSize = fileStream.Length

};

var bytes = SendingDataHelper.PrepareData(fileInfo);

socket.Send(bytes);

}

private void SendFile(FileStream fileStream)

{

using (var binaryReader = new BinaryReader(fileStream))

{

byte[] buffer = new byte[bufferSize];

while (binaryReader.Read(buffer, 0, bufferSize) > 0)

{

socket.Send(buffer);

}

}

}

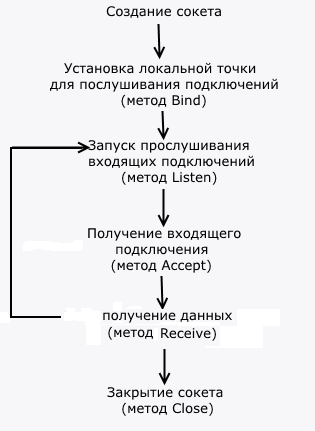


Рис 11. Общая схема работы серверного сокета

Серверный сокет должен вызвать метод Bind для установки точки для прослушивания входящих подключений и затем запустить прослушивание подключений с помощью метода Listen. Далее с помощью метода Accept можно получить входящие запросы на подключение в виде объекта Socket, из которого происходит считывание информации об изображении и само изображение. Соответствующий код сервера представлен в приложении А.

Сравним временные показатели клиента и сервера с полученными из предыдущей главы данными.

Таблица 4. Результаты программы с сокетным взаимодействием

|  |  |  |  |
| --- | --- | --- | --- |
| Тип файла | Размер файла  (байты) | Время клиента  (мс) | Время сервера  (мс) |
| Изображение jpg | 147 773 | 34 | 38 |
| Изображение jpg | 621 004 | 48 | 60 |
| Изображение jpg | 755 637 | 60 | 64 |
| Изображение jpg | 1 102 805 | 67 | 72 |

**4. Влияния сжатия информации**

Сжатие информации – это процесс преобразования информации, хранящейся в файле, путем уменьшения избыточности данных. Целью этого процесса является уменьшения объема, занимаемого данными. Степень сжатия зависит от используемого метода сжатия, типа исходного файла: текстового, графического, видео, звукового и т.д.

Существующие алгоритмы сжатия данных можно разделить на два больших класса – с потерями и без. Алгоритмы с потерями обычно применяются для сжатия изображений и аудио. Например, мы в качестве тестирования постоянно передаем файлы в формате jpeg. Данный алгоритм позволяет достичь больших степеней сжатия благодаря избирательной потере качества. Однако, по определению, восстановить первоначальные данные из сжатого результата невозможно.

Алгоритмы сжатия без потерь применяются для уменьшения размера данных, и работают таким образом, что возможно восстановить данные в точности такими, какие они были до сжатия. В нашей программе мы будем применять алгоритмы сжатия изображения без потерь DEFLATE и GZIP.

Основной принцип алгоритмов сжатия базируется на том, что в любом файле информация частично повторяется. Таким образом, можно определить вероятность повторения определённой комбинации символов и самые повторяющиеся фразы закодировать самыми короткими кодами. При этом необходимо обеспечить уникальную идентификацию последовательности символов кодами.

Алгоритм GZIP основан на алгоритме DEFLATE, который в свою очередь использует комбинацию алгоритма LZ77 и алгоритма Хаффмана.

Алгоритм LZ77 заменяет повторные вхождения данных на ссылки. Т.е. если в имеющихся данных какая-то цепочка элементов встречается более одного раза, то все последующие её вхождения заменяются «ссылками» на её первый экземпляр. Кодирование Хаффмана является методом кодирования с переменной длиной, которая назначает более короткие коды к более частым комбинациям символов.

Ниже представлены функции сжатия и восстановления изображений. Сжатие применяется на клиенте. По сети передается сжатый поток информации. Восстановление изображения происходит на сервере. Методы сжатия на узлах должны быть согласованы. Функции сжатия и восстановления были внедрены в выше представленную программу по взаимодействию сокетов.

public static MemoryStream CompressDeflate(FileStream fileStream)

{

var memoryStream = new MemoryStream();

var compressionStream = new DeflateStream(memoryStream,

CompressionMode.Compress);

fileStream.CopyTo(compressionStream);

return memoryStream;

}

public static void SaveAndDecompressDeflate(

string fileName, MemoryStream memoryStream)

{

memoryStream.Position = 0;

using (var targetStream = File.Create(fileName))

{

using (var decompressionStream = new

DeflateStream(

memoryStream,

CompressionMode.Decompress))

{

decompressionStream.CopyTo(targetStream);

}

}

}

Таблица 5. Влияние сжатия информации

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Тип файла | коэффициент сжатия/  Размер файла / Сжатый файл(байты) | Тип сжатия | Время клиента  (мс) | Время сервера  (мс) |
| Изображение jpg | 98,0/147 773 / 131072 | DEFLATE | 48 | 59 |
| Изображение jpg | 98,0/147 773 / 131 082 | GZIP | 28 | 31 |
| Изображение jpg | 93,5/621 004 / 581 632 | DEFLATE | 98 | 140 |
| Изображение jpg | 93,5/621 004 / 581 642 | GZIP | 70 | 72 |
| Изображение jpg | 97,2/1102 805/ 1073152 | DEFLATE | 144 | 178 |
| Изображение jpg | 0,974/1 102 805/  1073162 | GZIP | 119 | 125 |

Степень сжатия для каждого метода будет определена как: Kc=Vc/Vo\*100%

где Kc – коэффициент сжатия, Vc– объем сжатого файла, Vo – исходный объем файла.

**5. Заключение**

**Заключение**

Таким образом, в первой части настоящей главы был разработан собственный протокол передачи информации на основе уже рассмотренных протоколов TCP и UDP. Были продемонстрированы основные принципы надежности протокола TCP, после чего данные методы были внедрены в собственный протокол для надежной передачи изображений. Было показано, что протокол обеспечивает надежность и отвечает тем требованиям, которые были предъявлены ему. Насколько повысилась надежность НОВОГО алгоритма ?

Насколько повысилось время доставки дайтаграммы НОВОГО алгоритма ?

В отчете был отдельно рассмотрен принцип разработки программы с использованием сокетов. Отдельно оценивалась роль сжатия информации на время передачи. Подробное описание, сравнение и анализ полученных результатов будет рассмотрен в следующей главе.

=============

**Список использованных источников**

1. Руководство по сетевому программированию в С# и .NET. [Электронный ресурс]. – Режим доступа: https://metanit.com/sharp/net/, свободный – (10.06.2020).

2. Эндрю Кровчик, Винод Кумар. .Net. Сетевое программирование для профессионалов. : Пер. с англ. — СПб. : ООО “Издательство Лори”, 2007 — 417с.

3. Сетевое программирование в .NET Framework. [Электронный ресурс]. – Режим доступа: https://docs.microsoft.com/ru-ru/dotnet/framework/network-programming/, свободный – (10.06.2020).

1. Олифер В. Г., Олифер Н. А. Компьютерные сети. Принципы, технологии, протоколы: Учебник для вузов. 4-е изд. — СПб.: Питер, 2010. — 944 е.: ил.

2. Руденков Н.А., Долинер Л.И. Основы сетевых технологий: Учебник для вузов. Екатеринбург: Изд-во Уральского. Федерального ун-та, 2011. – 300 с.

3. Руководство по сетевому программированию в С# и .NET. [Электронный ресурс]. – Режим доступа: https://metanit.com/sharp/net/,  свободный – (10.06.2021).

4. Эндрю Кровчик, Винод Кумар. .Net. Сетевое программирование для профессионалов. : Пер. с англ. — СПб. : ООО “Издательство Лори”, 2007 — 417с.

5. Сетевое программирование в .NET Framework. [Электронный ресурс]. – Режим доступа: https://docs.microsoft.com/ru-ru/dotnet/framework/network-programming/,  свободный – (10.06.2021).

6. Реализация Reliable Udp протокола для .Net. [Электронный ресурс]. – Режим доступа: https://habr.com/ru/post/250227/,  свободный – (20.12.2021).

7. Как работает сжатие GZIP. [Электронный ресурс]. – Режим доступа: https://habr.com/ru/post/221849/,  свободный – (20.12.2021).

Введение. 3

1. Разработка надежного протокола UDP 4

2. Тестирование и результаты работы протокола 14

3. Принцип передачи информации по сокетам 17

4. Влияние сжатия информации 20

5. Заключение 23

Список литературы 24

===================

Заключение

Список использованных источников

**Введение**

В предыдущей главе были продемонстрированы преимущества рассмотренных транспортных протоколов для передачи изображений по сети. В ходе тестирования были показаны ожидаемые результаты – качество изображения теряется при передачах по протоколу UDP. Большинство пакетов отправленных клиентом серверу были искажены в процессе передачи или вовсе потеряны.

В настоящей главе происходит разработка собственного потокового протокола передачи информации, <новую разработку нельзя называть известным именем> который представляет собой комбинацию существующих протоколов UDP и TCP. Описываются алгоритмы разрабатываемого протокола, оценивается время, затрачиваемое клиентом и сервером для передачи и приема соответственно. Временные характеристики протокола сравниваются с уже полученными результатами из предыдущей главы.

Помимо этого в данной главе отдельно рассматриваются сокеты. Изучается принцип разработки программы с использованием сокетов, отдельно оценивается роль сжатия информации на время передачи.

**1. Разработка надежного протокола UDP <**новую разработку нельзя называть известным именем>

Главное преимущество протокола TCP состоит в его надежности. Надежность данного протокола транспортного уровня обеспечивается, например, за счет установки соединения между устройствами, отправки подтверждений приема данных, доставки данных в определенном порядке. В это же время протокол UDP обеспечивает большую скорость, однако данное преимущество обесценивается тем фактом, что данные могут быть потеряны во время передачи. Стандарт НТТР не требует использования именно протокола TCP как основного. Мы можем передавать НТТР через датаграммный сокет или через любой другой. Но на практике весь НТТР трафик передается через TCP, благодаря удобству последнего.

В наше время протокол UDP используется приложениями не только для доставки датаграмм и широковещательных рассылок, но и для реализации одноранговой сети, в которой каждое устройство может быть представлено в виде клиента и сервера одновременно. Таким образом, у данного протокола появилось множество не запланированных ранее способов применения, но, недостатки протокола, такие как отсутствие гарантированной доставки, никуда при этом не исчезли.

Разрабатываемому приложению требуется гарантированная доставка изображений. В таком случае, для обеспечения гарантированной доставки пакетов, требуется реализовать протокол прикладного уровня, обеспечивающий необходимую функциональность и работающий поверх UDP.

Основная рассматриваемая задача в рамках выпускной квалификационной работы - это передача изображений по сети. Предлагается воспользоваться преимуществами каждого рассматриваемого протокола транспортного уровня - создать новый протокол, основанный на TCP и UDP и оценить результаты. Для этого рассмотрим более подробно преимущества каждого существующего протокола. Для разрабатываемого протокола предложим собственные варианты решения поступающих проблем.

Ниже представлены основные принципы и требования к разрабатываемому протоколу:

1. Надежная доставка пакетов

2. Передача изображений осуществляется поверх UDP

3. Передача подтверждений фактов приемки сообщений сервером осуществляется по протоколу TCP

4. Увеличенная пропускная способность канала

Гарантия доставки протоколом TCP обеспечивается за счёт механизма подтверждения полученных сервером сообщений.



Рис 1. Гарантия доставки

Для разрабатываемой программы введем еще один класс, который будет представлять собой подтверждающее сообщение (специальный пакет подтверждения). В сообщении сервер будет передавать номер пришедшего блока.

namespace WebProtocolsModel

{

[Serializable]

public class ConfirmMessage

{

public int BatchOrder { get; set; }

}

}

Очевидно, что нумерацию блоков необходимо производить на клиенте перед самой отправкой. Нумерация клиентом сегментов необходима не только для того, чтобы в будущем по пришедшему номеру определять факты приемки блока сервером. Стоит помнить, что для отправки изображений мы используем протокол UDP. Порядок следования сообщений в протоколе UDP не детерминирован. Сегменты изображений будут приходить в неопределенной последовательности. Несмотря на появление в разрабатываемом протоколе некоторой избыточности, введение нумерации пакетов оправдано. Таким образом, для сбора результирующей картинки серверу достаточно просто отсортировать пришедшие буферы, состоящие из байтов, в порядке их нумерации клиентом.

namespace WebProtocolsModel

{

[Serializable]

public class FileBatch

{

public byte[] Bytes { get; set; }

public int Order { get; set; }

}

}

Ниже, на рисунке 2 представлен алгоритм сервера. Видно, что сервер имеет стандартный алгоритм для UDP взаимодействия за исключением того, что к нему были добавлены несколько дополнительных блоков: блок подтверждения приема сообщения, блок сбора конечной информации. Отправка подтверждающих сообщений происходит после приема очередного сегмента. Из пришедшего сегмента извлекается его номер, создается подтверждающее сообщение с этим номером и отправляется клиенту именно по протоколу TCP. Применение протокола TCP в данный момент оправдано тем, что подтверждения не будут потеряны в процессе их передачи от сервера клиенту, а объём передаваемой информации по TCP не велик, что не приведет к значительному увеличению трафика.

private void SaveFile(

WebProtocolsModel.FileInfo fileInfo,

NetworkStream confirmingNetworkStream)

{

var remoteEndPoint = new IPEndPoint(IPAddress.Any, 0);

var fileBatches = new Dictionary<int, FileBatch>();

using (var binaryWritter = new BinaryWriter(new

FileStream(fileInfo.FileName, FileMode.Create)))

{

for (int i = 0; i < fileInfo.FileSize; i += bufferSize)

{

var data = server.Receive(ref remoteEndPoint);

var fileBatch = GetFileBatch(data);

fileBatches.Add(fileBatch.Order, fileBatch);

ConfirmReceiving(

fileBatch,

confirmingNetworkStream,

remoteEndPoint);

}

binaryWritter.Write(

fileBatches

.OrderBy(b => b.Key)

.SelectMany(b => b.Value.Bytes)

.ToArray());

}

}

Каким образом происходит сбор конечной картинки на сервере в условиях недетерминированного приема? Для того чтобы собрать картинку необходимо просто отсортировать пришедшие данные по их порядковому номеру, выданному клиенту перед отправкой.

Теперь подробнее рассмотрим роль и функции клиента в разрабатываемом взаимодействии. Как было сказано ранее, клиент должен уметь нумеровать исходные данные для отправки их серверу. Нумерация сегментов с байтами выполняется клиентом по мере их считывания из файловой системы. Но помимо этого у клиента при данном разрабатываемом взаимодействии появляются еще ряд сложных функций:

1. Прием подтверждений от сервера по протоколу TCP;

2. Обработка подтверждений;

3. Отправка потерянных сегментов заново по протоколу UDP.

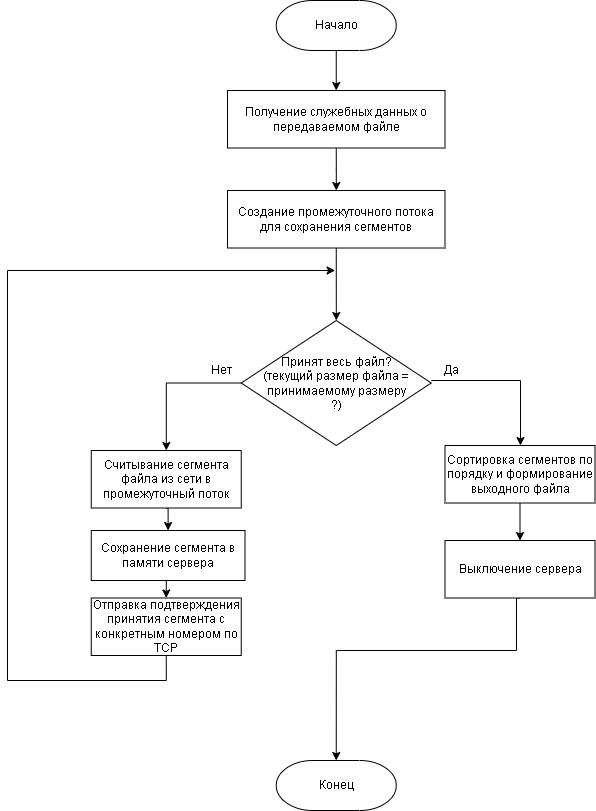


Рис 2. Алгоритм сервера

Принцип приема данных по протоколу TCP был подробно объяснен в предыдущих главах. Очевидно, что для принятия подтверждающих сообщений на клиенте необходимо завести экземпляр класса TcpListener. Данный класс представляет собой TCP сервер. Он будет ожидать и обрабатывать входящие подключения. Вся описанная логика должна быть вынесена в отдельный поток. Поток — это основная единица, которой операционная система выделяет время процессора. Если это не сделать, в момент, когда TCP сервер будет ожидать входящие подключения и подтверждающие сообщения, он заблокирует передачу сегментов с изображениями по UDP.

private void ReadConfirmations()

{

WaitBatches();

var confirmationsCounter = 0;

var client = confirmingTcpListener.AcceptTcpClient();

var networkStream = client.GetStream();

while (confirmationsCounter < fileBatches.Count)

{

var binaryFormatter = new BinaryFormatter();

if (networkStream.DataAvailable)

{

var confirmMessage =

binaryFormatter.Deserialize(networkStream) as

ConfirmMessage;

fileBatches[confirmMessage.BatchOrder].IsConfirmed =

true;

confirmationsCounter++;

}

else

{

RetrySendingBatches();

}

}

}

Рассмотрим случай, когда клиент принял сообщение о подтверждении сервером факта приемки конкретного блока. Тогда необходимо извлечь номер подтвержденного блока данных из сообщения и пометить его как подтвержденный уже на клиенте. Клиент гарантирует, что подтвержденные сервером сегменты не будут повторно отправлены ему по UDP еще раз.

Что происходит, когда клиент перестает принимать подтверждения от сервера? В этом случае начинает работать функция повторной отправки потерянных сегментов изображения.

private void RetrySendingBatches()

{

var unconfirmedBatches = fileBatches.Values.Where(b =>

!b.IsConfirmed).ToArray();

foreach (var batchInfo in unconfirmedBatches)

{

if (DateTime.UtcNow - batchInfo.LastRetryDateTimeUtc >

retryTimeSpan)

{

retryCounter++;

RetrySendingBatch(batchInfo, iPEndPoint);

}

}

}

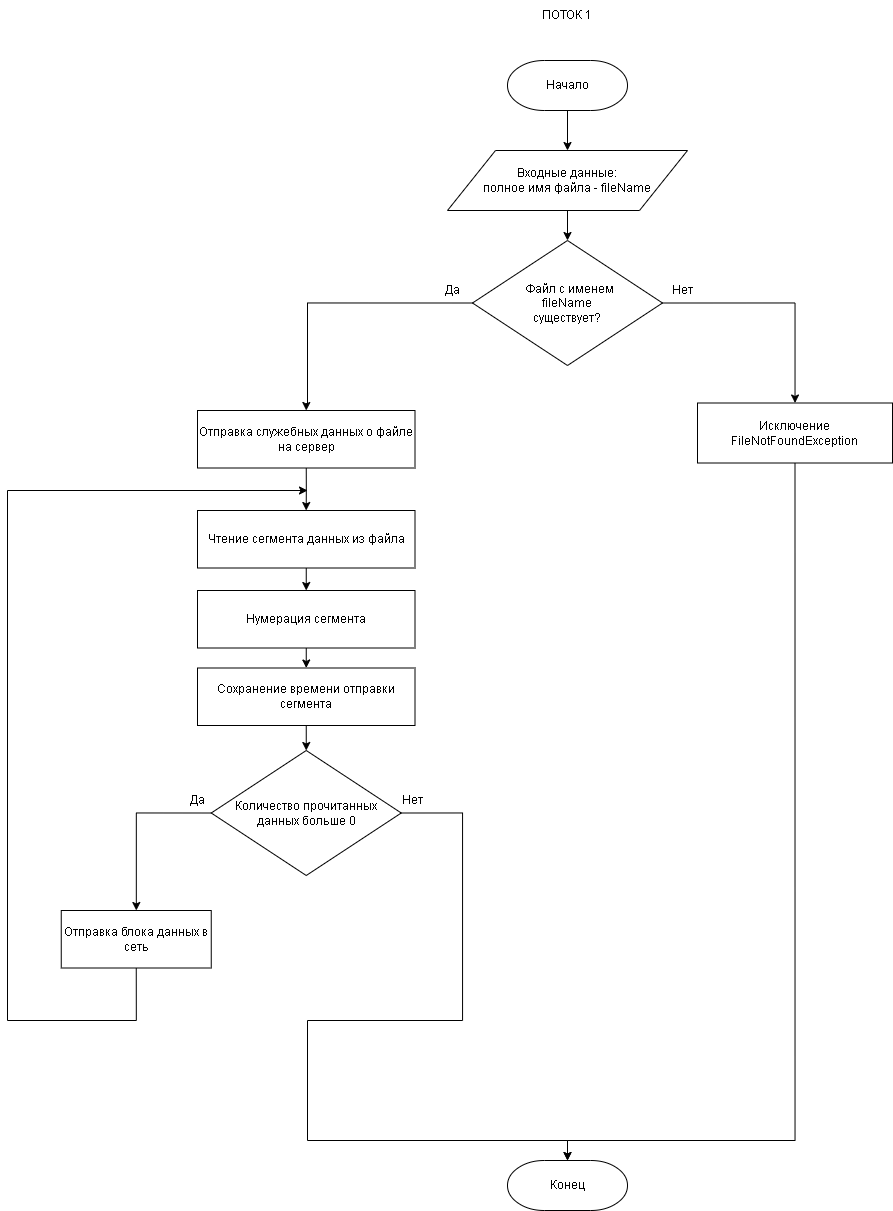


Рис 3. Алгоритм клиента. Поток 1.

Алгоритм поправить «спрямить» все соединительные линии алгоритма

Протокол TCP в качестве критерия необходимости повторной отправки данных использует механизм таймеров – для каждого сообщения задается время ожидания подтверждения, если в течение этого времени подтверждение не пришло, таймер срабатывает и потерянный сегмент отправляется еще раз.

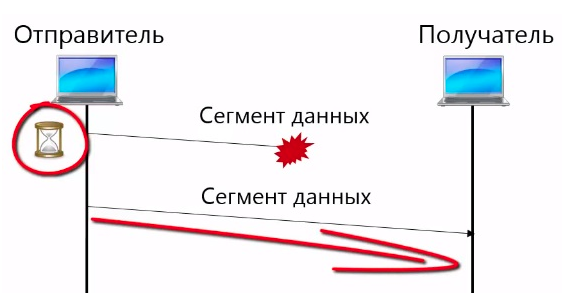


Рис 4. Потеря данных

Реализуем похожий механизм для нашего клиента. Для каждого сегмента будем сохранять время его отправки по протоколу UDP. В момент определения необходимости повторной отправки сегмента, будем сравнивать сохраненное время с текущим временем. Если оно больше порогового, то требуется повторная отправка сегмента по протоколу UDP. Если нет, то считаем, что время таймаута не прошло, а повторная отправка просто откладывается. Описанный алгоритм представлен на рисунках 3, 8 для потока 1 и потока 2 соответственно.

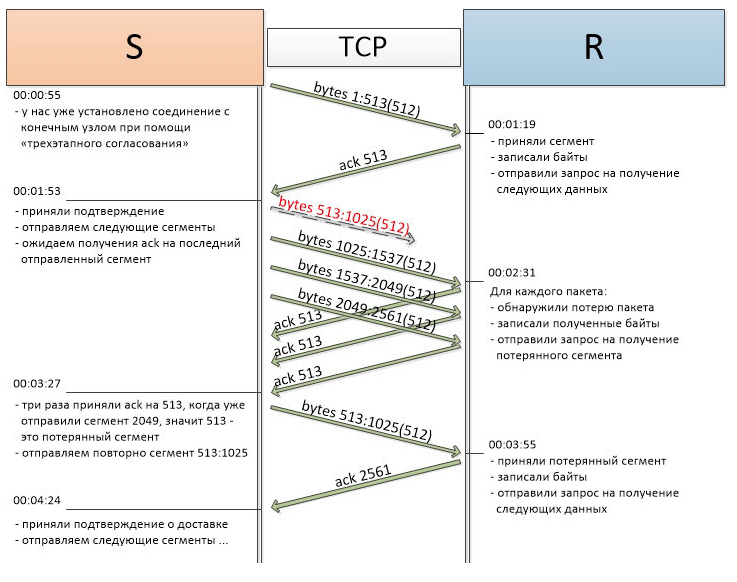


Рис 5. Механизм окна TCP

TCP для увеличения пропускной способности использует механизм скользящего окна. Отправитель передает сразу несколько порций данных, не дожидаясь подтверждения. Получатель отправляет одно общее подтверждение сразу на несколько сегментов.

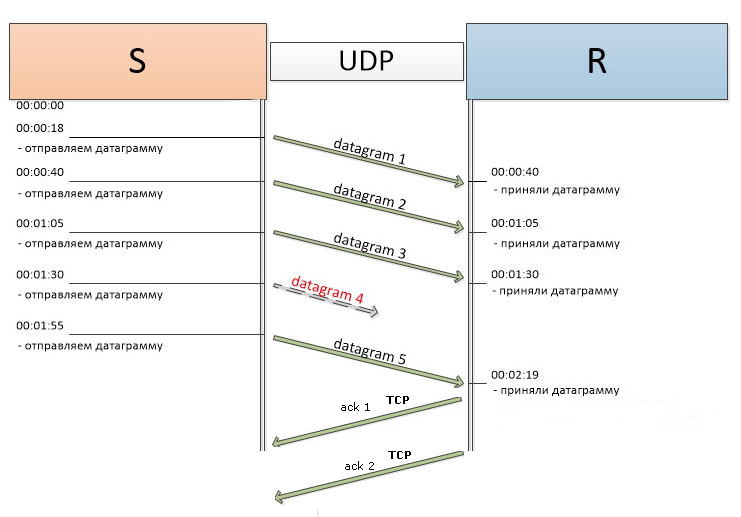


Рис 6. Диаграмма взаимодействия клиента и сервера разработанного протокола

В нашей реализации взаимодействия клиента и сервера окно не используется. Таким образом, теоретически клиент может отправить все датаграммы c изображением по протоколу UDP еще до того, как по TCP начнут приходить подтверждения.

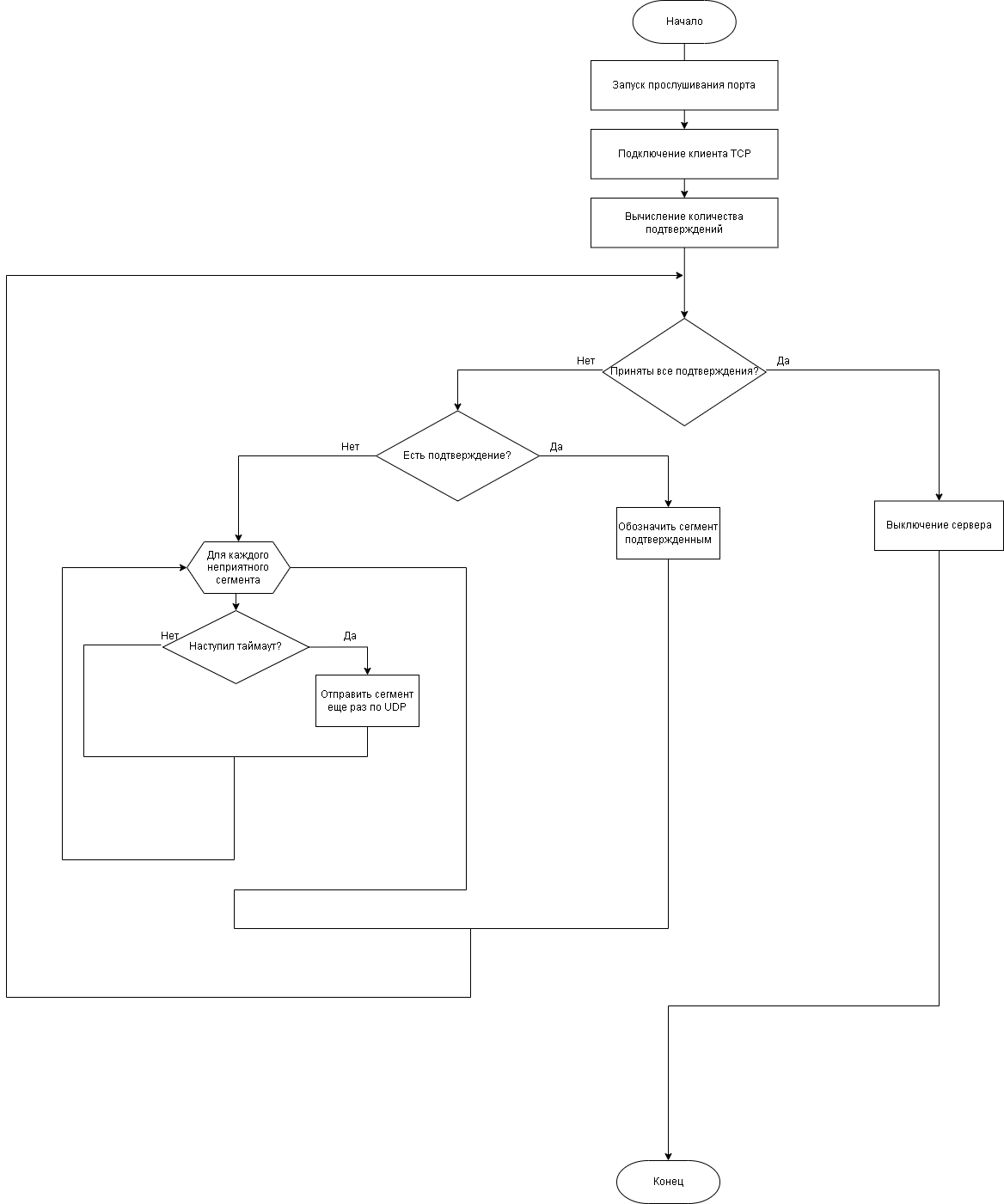


Рис 7. Алгоритм клиента. Поток 2 (Обработка подтверждений).

Алгоритм поправить: 1)Во **всех логических блоках необходимо указать условия перехлдов**

2) «спрямить» все соединительные линии алгоритма

**2. Тестирование и результаты работы протокола**

Программы клиента и сервера выполнены в виде консольного приложения. Интерфейсы этих программ представлены на рисунке 8. Передадим изображение по разработанному протоколу. Убедимся, что изображение успешно пришло на сервер.

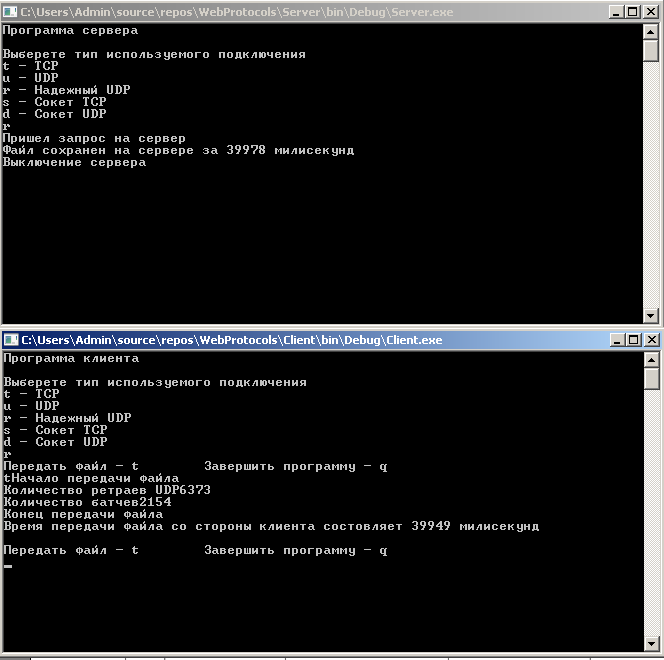


Рис 9. Интерфейс программы

Будем передавать различные изображения с разным размером и смотреть, как ведет себя протокол.

Таблица 1. Разный размер изображений

|  |  |  |  |
| --- | --- | --- | --- |
| Тип файла | Размер файла  (байты) | Время клиента  (мс) | Время сервера  (мс) |
| Изображение jpg | 147 773 | 10193 | 10198 |
| Изображение jpg | 621 004 | 26800 | 26813 |
| Изображение jpg | 755 637 | 31421 | 31500 |
| Изображение jpg | 1 102 805 | 39949 | 39978 |

Как было уже много раз сказано, изображение передается сегментами определенной длины. Попробуем изменить размеры входных и выходных буферов при фиксированной длине изображения **1 102 805 байт**.

Таблица 2. Разный размер буфера

|  |  |  |  |
| --- | --- | --- | --- |
| Количество сегментов в изображении/ Количество повторных отправок | Размер буфера  (байты) | Время клиента  (мс) | Время сервера  (мс) |
| 2154/4055 | 512 | 39949 | 39978 |
| 1077/2171 | 1024 | 28051 | 28082 |
| 539/625 | 2048 | 16674 | 16698 |
| 270/169 | 4096 | 11079 | 11069 |
| 135/97 | 8192 | 10500 | 10528 |

Необходимость повторной отправки сегмента клиентом на сервер определяется с помощью механизма таймаутов. Если подтверждение для сегмента с номером N не приходит в течении определенного времени, сегмент с номером N отправляется еще раз на сервер. Попробуем изменить время таймаута при фиксированной длине изображения **1 102 805 байт и буфере 2Кбайт.**

Таблица 3. Разное время таймаута

|  |  |  |
| --- | --- | --- |
| Время таймаута  (сек) | Время клиента  (мс) | Время сервера  (мс) |
| 1 | 4629 | 4619 |
| 2 | 9498 | 9200 |
| 5 | 16674 | 16698 |
| 8 | 28543 | 28557 |
| 10 | 31608 | 31115 |

**3. Принцип передачи информации по сокетам**

Сокет – некоторый абстрактный объект, представляющий конечную точку соединения. Реализация сокетов в .NET сделана поверх уже рассмотренных протоколов транспортного уровня и проткала IP сетевого уровня.

Потоковый сокет — это сокет с установленным соединением, состоящий из потока байтов, который может быть двунаправленным, т, е. через эту конечную точку приложение может и передавать, и получать данные. Данный тип сокета гарантирует исправление ошибок, обеспечивает порядок проходимых данных. Потоковой сокет базируется на протоколе TCP и подходит для задачи передачи изображений. Таким образом, если данные должны гарантировано доставляться другой стороне, и объем передаваемых данных достаточно велик (требуется установка постоянного соединения между узлами), то потоковые сокеты являются более предпочтительными. Принципы работы с сокетом изображены на рисунках 10 и 11 для клиента и сервера соответственно.



Рис 10. Общая схема работы клиентского сокета

Сокет клиента необходимо подключить к серверу. Для этого на клиенте вызывается метод Connect, в который передается информация об удаленной конечной точке, представляющая собой комбинацию IP – адреса и порта. Для передачи данных на сервер применяется метод Send. В нашем случае, вначале передается информация о файле (его название и размер), потом само изображение. Ниже представим код клиента:

public override void SendFile(string fileName)

{

if (!File.Exists(fileName))

{

throw new FileNotFoundException(fileName);

}

Console.Write("Подключение к удаленной точке\n");

socket.Connect(remoteIpPoint);

using (var inputFileStream = new FileStream(fileName,

FileMode.Open))

{

SendFileInfo(inputFileStream);

SendFile(inputFileStream);

//SendFileWithCompress(inputFileStream);

}

socket.Shutdown(SocketShutdown.Send);

socket.Close();

}

private void SendFileInfo(FileStream fileStream)

{

var fileInfo = new WebProtocolsModel.FileInfo

{

FileName = fileStream.Name,

FileSize = fileStream.Length

};

var bytes = SendingDataHelper.PrepareData(fileInfo);

socket.Send(bytes);

}

private void SendFile(FileStream fileStream)

{

using (var binaryReader = new BinaryReader(fileStream))

{

byte[] buffer = new byte[bufferSize];

while (binaryReader.Read(buffer, 0, bufferSize) > 0)

{

socket.Send(buffer);

}

}

}

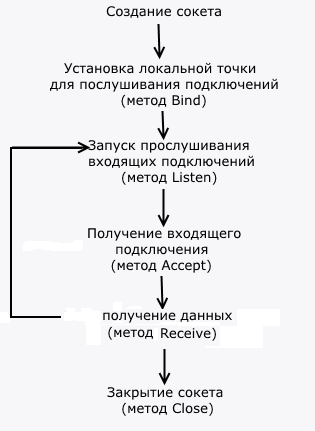


Рис 11. Общая схема работы серверного сокета

Серверный сокет должен вызвать метод Bind для установки точки для прослушивания входящих подключений и затем запустить прослушивание подключений с помощью метода Listen. Далее с помощью метода Accept можно получить входящие запросы на подключение в виде объекта Socket, из которого происходит считывание информации об изображении и само изображение. Соответствующий код сервера представлен в приложении А.

Сравним временные показатели клиента и сервера с полученными из предыдущей главы данными.

Таблица 4. Результаты программы с сокетным взаимодействием

|  |  |  |  |
| --- | --- | --- | --- |
| Тип файла | Размер файла  (байты) | Время клиента  (мс) | Время сервера  (мс) |
| Изображение jpg | 147 773 | 34 | 38 |
| Изображение jpg | 621 004 | 48 | 60 |
| Изображение jpg | 755 637 | 60 | 64 |
| Изображение jpg | 1 102 805 | 67 | 72 |

**4. Влияния сжатия информации**

Сжатие информации – это процесс преобразования информации, хранящейся в файле, путем уменьшения избыточности данных. Целью этого процесса является уменьшения объема, занимаемого данными. Степень сжатия зависит от используемого метода сжатия, типа исходного файла: текстового, графического, видео, звукового и т.д.

Существующие алгоритмы сжатия данных можно разделить на два больших класса – с потерями и без. Алгоритмы с потерями обычно применяются для сжатия изображений и аудио. Например, мы в качестве тестирования постоянно передаем файлы в формате jpeg. Данный алгоритм позволяет достичь больших степеней сжатия благодаря избирательной потере качества. Однако, по определению, восстановить первоначальные данные из сжатого результата невозможно.

Алгоритмы сжатия без потерь применяются для уменьшения размера данных, и работают таким образом, что возможно восстановить данные в точности такими, какие они были до сжатия. В нашей программе мы будем применять алгоритмы сжатия изображения без потерь DEFLATE и GZIP.

Основной принцип алгоритмов сжатия базируется на том, что в любом файле информация частично повторяется. Таким образом, можно определить вероятность повторения определённой комбинации символов и самые повторяющиеся фразы закодировать самыми короткими кодами. При этом необходимо обеспечить уникальную идентификацию последовательности символов кодами.

Алгоритм GZIP основан на алгоритме DEFLATE, который в свою очередь использует комбинацию алгоритма LZ77 и алгоритма Хаффмана.

Алгоритм LZ77 заменяет повторные вхождения данных на ссылки. Т.е. если в имеющихся данных какая-то цепочка элементов встречается более одного раза, то все последующие её вхождения заменяются «ссылками» на её первый экземпляр. Кодирование Хаффмана является методом кодирования с переменной длиной, которая назначает более короткие коды к более частым комбинациям символов.

Ниже представлены функции сжатия и восстановления изображений. Сжатие применяется на клиенте. По сети передается сжатый поток информации. Восстановление изображения происходит на сервере. Методы сжатия на узлах должны быть согласованы. Функции сжатия и восстановления были внедрены в выше представленную программу по взаимодействию сокетов.

public static MemoryStream CompressDeflate(FileStream fileStream)

{

var memoryStream = new MemoryStream();

var compressionStream = new DeflateStream(memoryStream,

CompressionMode.Compress);

fileStream.CopyTo(compressionStream);

return memoryStream;

}

public static void SaveAndDecompressDeflate(

string fileName, MemoryStream memoryStream)

{

memoryStream.Position = 0;

using (var targetStream = File.Create(fileName))

{

using (var decompressionStream = new

DeflateStream(

memoryStream,

CompressionMode.Decompress))

{

decompressionStream.CopyTo(targetStream);

}

}

}

Таблица 5. Влияние сжатия информации

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Тип файла | Размер файла / Сжатый файл  (байты) | Тип сжатия | Время клиента  (мс) | Время сервера  (мс) |
| Изображение jpg | 147 773 / 131072 | DEFLATE | 48 | 59 |
| Изображение jpg | 147 773 / 131082 | GZIP | 28 | 31 |
| Изображение jpg | 621 004 / 581632 | DEFLATE | 98 | 140 |
| Изображение jpg | 621 004 / 581642 | GZIP | 70 | 72 |
| Изображение jpg | 1 102 805/ 1073152 | DEFLATE | 144 | 178 |
| Изображение jpg | 1 102 805 /  1073162 | GZIP | 119 | 125 |

Степень сжатия для каждого метода будет определена как: Kc=Vc/Vo\*100%

Kc – коэффициент сжатия, Vc– объем сжатого файла, Vo – исходный объем файла.

Заключение

Таким образом, в первой части настоящей статье главы был разработан собственный протокол ТСР-м передачи информации на основе уже рассмотренных протоколов TCP и UDP. Были продемонстрированы основные принципы надежности протокола TCP, после чего данные методы были внедрены в собственный протокол ТСР-м для надежной передачи изображений. Было показано, что протокол обеспечивает надежность и отвечает тем требованиям, которые были предъявлены ему. Насколько повысилась надежность НОВОГО алгоритма ?

Насколько повысилось время доставки дайтаграммы НОВОГО алгоритма ?

В отчете был отдельно рассмотрен принцип разработки программы с использованием сокетов. Отдельно оценивалась роль сжатия информации на время передачи. Подробное описание, сравнение и анализ полученных результатов будет рассмотрен в следующей главе.

**Список использованных источников**

1. Олифер В. Г., Олифер Н. А. Компьютерные сети. Принципы, технологии, протоколы: Учебник для вузов. 4-е изд. — СПб.: Питер, 2010. — 944 е.: ил.

2. Руденков Н.А., Долинер Л.И. Основы сетевых технологий: Учебник для вузов. Екатеринбург: Изд-во Уральского. Федерального ун-та, 2011. – 300 с.

3. Руководство по сетевому программированию в С# и .NET. [Электронный ресурс]. – Режим доступа: https://metanit.com/sharp/net/,  свободный – (10.06.2021).

4. Эндрю Кровчик, Винод Кумар. .Net. Сетевое программирование для профессионалов. : Пер. с англ. — СПб. : ООО “Издательство Лори”, 2007 — 417с.

5. Сетевое программирование в .NET Framework. [Электронный ресурс]. – Режим доступа: https://docs.microsoft.com/ru-ru/dotnet/framework/network-programming/,  свободный – (10.06.2021).

6. Реализация Reliable Udp протокола для .Net. [Электронный ресурс]. – Режим доступа: https://habr.com/ru/post/250227/,  свободный – (20.12.2021).

7. Как работает сжатие GZIP. [Электронный ресурс]. – Режим доступа: https://habr.com/ru/post/221849/,  свободный – (20.12.2021).