**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

Механико-математический факультет

Кафедра веб-технологий и компьютерного моделирования

**Иванов Тимофей Владимирович**

# РАЗРАБОТКА ПРИЛОЖЕНИЙ ТИПА КЛИЕНТ/СЕРВЕР.

Курсовая работа

Студента II курса

Руководитель **Люлькин Аркадий Ефимович**

Доцент кафедры численных методов и программирования

Минск 2020

**Содержание**

## Введение

Миллионы людей каждый день используют Интернет, чтобы почитать новости, пообщаться с друзьями, отправить сообщения и почту, получить полезную информацию, совершить покупку или оплатить счет. Подавляющая часть этого взаимодействия происходит на сайтах или программном обеспечении, использующем систему “Клиент/Сервер”.

Концепция Клиент/Сервер, как понятно из названия, реализует две стороны. Клиент – это заказчик той или иной услуги, а сервер – поставщик услуг. Клиент и сервер физически представляют собой программы, например, типичным клиентом является браузер или любая программа для отправки сообщений. В качестве сервера можно привести, например, любые сервера на которых работают сайты в сети интернет, например, Apache, Open Server, NodeJS и т.д.

Клиент и сервер взаимодействую друг с другом в сети Интернет или в любой другой компьютерной сети при помощи различных сетевых протоколов, например, IP/TCP протокол, HTTP протокол, FTP и другие. При помощи HTTP протокола браузер отправляет специальное HTTP сообщение, в котором указано какую информацию и в каком виде он хочет получить от сервера. Сервер, получив такое сообщение, отсылает браузеру в ответ похожее по структуре сообщение, в котором содержится нужная информация, например, HTML документ.

Стоит заметить, что зачастую в основе взаимодействия Клиент/Сервер лежит принцип того, что такое взаимодействие начинает клиент, сервер лишь отвечает клиенту, а не наоборот. Клиентское программное обеспечение и серверное программное обеспечение обычно установлено на разных машинах, но также они могут работать и на одном компьютере.

Одним из способов реализации приложений типа Клиент/Сервер является использование сокетов (socket — разъём). Сокет — это двунаправленный канал между двумя компьютерами в сети, который обеспечивает конечную точку соединения. Данные в сокетах могут передаваться в двух направлениях — от клиента к серверу и наоборот. Сокет-интерфейс используется для получения доступа к транспортному уровню протокола TCP/IP и представляет собой набор системных вызовов операционной системы и библиотечных функций.

TCP, используемый в большинстве сокетов, гарантирует доставку пакетов, их очередность, автоматически разбивает данные на пакеты и контролирует их передачу. Но при этом TCP работает несколько медленнее протоколов без таких

проверок за счет повторной передачи потерянных пакетов и большему количеству выполняемых операций над пакетами.

В данной работе будет рассмотрена разработка приложений типа Клиент/Сервер на основе сокетов. Целью работы является изучение этого процесса и изучение проектирования Клиент/Серверного приложения. Поставлены следующие задачи:

- Изучить средства Microsoft Visual Studio 2017 для разработки приложений типа Клиент/Сервер
- Разработать чат на основе протокола TCP/IP с использованием сокетов.

## Средства Microsoft Visual Studio 2017 для разработки приложений типа Клиент/Сервер

Среда Visual Studio предоставляет возможности разработки на нескольких языках. В данном случае будет рассмотрена разработка на C++/CLI, языка для среды программирования Microsoft .NET, который интегрирует C++ с общеязыковой инфраструктурой (Common Language Infrastructure - CLI).

Сокет в .NET это один конец двустороннего канала связи между двумя программами, работающими в сети. Соединяя вместе два сокета, можно передавать данные между разными процессами, как локальными, так и удаленными. Реализация сокетов обеспечивает инкапсуляцию протоколов сетевого и транспортного уровней.

Первоначально сокеты были разработаны для UNIX, где обеспечивающий связь метод ввода-вывода следует алгоритму open/read/write/close. Прежде чем ресурс использовать, его нужно открыть, задав соответствующие разрешения и другие параметры. Как только ресурс открыт, из него можно считывать или в него записывать данные. После использования ресурса пользователь должен вызвать метод Close(), чтобы подать сигнал операционной системе о завершении его работы с этим ресурсом.

В общем жизнь сокетов можно разделить на три фазы: открыть (создать) сокет, получить из сокета или отправить сокету и в, конце концов, закрыть сокет.

Интерфейс для взаимодействия между разными процессами построен поверх методов ввода-вывода. Они облегчают для сокетов отправку и получение данных. Каждый целевой объект задается адресом сокета, следовательно, этот адрес можно указать в клиенте, чтобы установить соединение с целью.

Рассмотрим **потоковые сокеты**. 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.

Потоковый сокет гарантирует исправление ошибок, обрабатывает доставку и сохраняет последовательность данных. На него можно положиться в доставке упорядоченных, не дублированных данных. Потоковый сокет также подходит для передачи больших объемов данных, поскольку накладные расходы, связанные с установлением отдельного соединения для каждого отправляемого сообщения, может оказаться неприемлемым для небольших объемов данных. Потоковые сокеты достигают этого уровня качества за счет использования протокола Transmission Control Protocol (TCP). TCP обеспечивает

поступление данных на другую сторону в нужной последовательности и без ошибок.

Для этого типа сокетов путь формируется до начала передачи сообщений. Тем самым гарантируется, что обе участвующие во взаимодействии стороны принимают и отвечают. Если приложение отправляет получателю два сообщения, то гарантируется, что эти сообщения будут получены в той же последовательности.

Однако, отдельные сообщения могут дробиться на пакеты, и способа определить границы записей не существует. При использовании TCP этот протокол берет на себя разбиение передаваемых данных на пакеты соответствующего размера, отправку их в сеть и сборку их на другой стороне. Приложение знает только, что оно отправляет на уровень TCP определенное число байтов и другая сторона получает эти байты. В свою очередь TCP эффективно разбивает эти данные на пакеты подходящего размера, получает эти пакеты на другой стороне, выделяет из них данные и объединяет их вместе.

Потоки базируются на явных соединениях: один сокет запрашивает соединение с другим сокетом, и тот либо соглашается с запросом на установление соединения, либо отвергает его.

Сервер чата или, например, почты представляет пример приложения, которое должно доставлять содержание в правильном порядке, без дублирования и пропусков. Потоковый сокет рассчитывает, что TCP обеспечит доставку сообщений по их назначениям.

Кроме того, существуют **дейтаграммные сокеты**. Их иногда называют сокетами без организации соединений, т. е. никакого явного соединения между ними не устанавливается — сообщение отправляется указанному сокету и, соответственно, может получаться от указанного сокета.

Потоковые сокеты по сравнению с дейтаграммными действительно дают более надежный метод, но для некоторых приложений накладные расходы, связанные с установкой явного соединения, неприемлемы. В конце концов на установление надежного соединения с сервером требуется время, которое просто вносит задержки в обслуживание, и задача серверного приложения не выполняется. Такие сокеты используются для сокращения накладных расходов.

Использование таких сокетов требует, чтобы передачей данных от клиента к серверу занимался протокол UDP. В этом протоколе на размер сообщений налагаются некоторые ограничения, и в отличие от потоковых сокетов, умеющих надежно отправлять сообщения серверу-адресату, дейтаграммные сокеты

надежность не обеспечивают. Если данные затерялись где-то в сети, сервер не сообщит об ошибках.

**Порт** нужен, чтобы разрешить задачу одновременного взаимодействия с несколькими приложениями. По существу, с его помощью расширяется понятие IP-адреса. Компьютер, на котором в одно время выполняется несколько приложений, получая пакет из сети, может идентифицировать целевой процесс, пользуясь уникальным номером порта, определенным при установлении соединения.

Для обеспечения соединения сокетов используются порты. Сокет состоит из IP-адреса машины и номера порта, используемого приложением TCP. Поскольку IP-адрес уникален в Интернете, а номера портов уникальны на отдельной машине, номера сокетов также уникальны во всем Интернете. Эта характеристика позволяет процессу общаться через сеть с другим процессом исключительно на основании номера сокета.

За определенными службами номера портов зарезервированы. Ваше приложение может пользоваться любым номером порта, который не был зарезервирован и пока не занят.

Поддержку сокетов в .NET обеспечивают классы в пространстве имен **System::Net::Sockets**. Такие, как:

- TcpClient – строится на классе Socket, чтобы обеспечить TCP-обслуживание на более высоком уровне. TcpClient предоставляет несколько методов для отправки и получения данных через сеть.
- TcpListener – этот класс также построен на низкоуровневом классе Socket. Его основное назначение — серверные приложения. Он ожидает входящие запросы на соединения от клиентов и уведомляет приложение о любых соединениях.
- UdpClient – это класс использующий протокол, не организующий соединение, следовательно, для реализации UDP-обслуживания в .NET требуется другая функциональность.
- Socket – базовый класс в пространстве имен System::Net::Sockets. Он обеспечивает базовую функциональность приложений, использующих сокеты.

Рассмотрим важные свойства и методы класса Socket, который будет использоваться для последующей разработки:

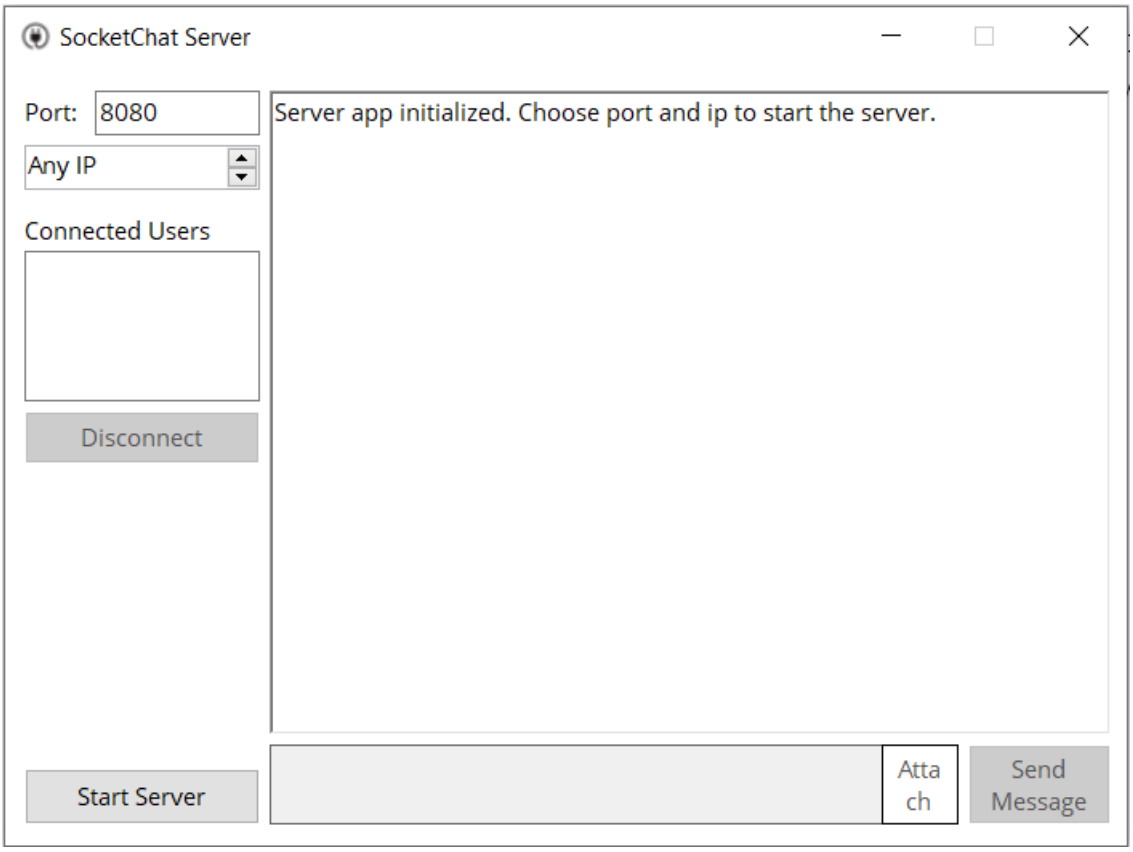
- AddressFamily – возвращает семейство адресов сокета, значение из перечисления Socket::AddressFamily, перечисляет семейства адресов (IPv4, IPv6 и т.д.)
- Available – возвращает объем доступных для чтения данных.
- Blocking – возвращает или устанавливает значение, показывающее, находится ли сокет в блокирующем режиме.
- Connected – возвращает значение, информирующее, соединен ли сокет с удаленным хостом.
- LocalEndPoint – возвращает локальную конечную точку.
- ProtocolType – возвращает тип протокола сокета.
- RemoteEndPoint – возвращает удаленную конечную точку сокета, с которой он соединен.
- SocketType –возвращает тип сокета.
- Accept() – ожидает подключение и создает новый сокет для обработки этого подключения.
- Bind() – связывает сокет с локальной конечной точкой для ожидания входящих запросов на соединение.
- Close() – заставляет сокет закрыться.
- Connect() – устанавливает соединение с удаленным хостом.
- Listen() – помещает сокет в режим прослушивания (ожидания).
- Receive() – получает данные от соединенного сокета.
- Send() – отправляет данные соединенному сокету.
- Shutdown() – запрещает операции отправки и получения данных на сокете.

## Разработка приложения типа Клиент/Сервер – чат

Рассмотрим приложение **Сервер**, который получает запросы на соединение и занимается получением-отправкой сообщений от клиентских приложений.

Программа ожидает настройки порта и IP, затем запускается и начинает ожидать подключения клиентов, после подключения начинает синхронизировать общий чат с клиентами. Реализуется отправка обычных текстовых и, также, графических сообщений, с использованием сетевых потоков, задач и сокетов.

Для начала определим графический интерфейс и возможности чата. Для этого используем возможности Windows Forms и C++/CLI. Пример графического интерфейса чата сервера:




Левая часть, сверху вниз:

- Создаем label указывающий, что нужно ввести, и textBox для ввода порта.
- Создаем domainUpDown объект, позволяющий выбрать один элемент из списка, для того чтобы пользователь мог выбрать IP на котором будет работать сервер.

- Создаем, label, checkedListBox и кнопку, по нажатию которой администратор сервера сможешь отключить с сервера пользователей, выбранных в окне с помощью чекбоксов.
- Затем создаем кнопку запуска сервера, по нажатию которой будет создаваться сокет, прослушивающий запросы на подключения.

Затем справа сверху вниз:

- Добавим richTextBox, в котором будут отображаться полученные сообщения.
- Снизу создадим textBox поле для отправки сообщения от имени сервера, кнопку “Attach”, по нажатию которой будет открываться toolStrip с опцией отправки изображения в чат, и кнопку отправки сообщения.

Был реализован интерфейс и теперь необходим создать основной код приложения. Опишем метод, необходимый для запуска сервера.

- Для начала работы создадим список доступных IPv4 адресов. И затем добавим их в список для выбора. «Any IP» – 0.0.0.0

```
domainUpDownIPs->Items->Add("Any IP"); // possibility to connect from any ip
//getting PC address
IPHostEntry^ host = Dns::GetHostEntry(Dns::GetHostName());
for (int i = 0; i < host->AddressList->Length; i++) {
 if (host->AddressList[i]->AddressFamily == AddressFamily::InterNetwork) {
  domainUpDownIPs->Items->Add(host->AddressList[i]);
 }
}
```

- Получим порт и IP из интерфейса программы.

```
int port = 8080; // standard port
try {
 port = Convert::ToInt32(textBoxPort->Text); // parse port from textbox
 if (port < 1024 || port > 65535) {
  throw gcnew Exception();
 }
}
catch (...) {
 MessageBox::Show("Incorrect port entered. Will be used port 8080", "Error",
   MessageBoxButtons::OK, MessageBoxIcon::Error);
}

Object^ item = domainUpDownIPs->Items[selectedIP];
IPAddress^ ip = (item->Equals("Any IP")) ? IPAddress::Any : (IPAddress^)item; // get
chosen ip
```

- Далее нам необходимо создать новую локальную IP точку, создать сам socket, привязать точку у нему и поставить в режим прослушивания.

Установив локальную конечную точку для ожидания соединений, можно создать сокет: gcnew Socket(AddressFamily::, SocketType::, ProtocolType::);

```
IPEndPoint^ ipPoint = gcnew IPEndPoint(ip, port); // get address to start the server
Socket^ listenSocket = gcnew Socket(AddressFamily::InterNetwork, SocketType::Stream,
ProtocolType::Tcp);
listenSocket->Bind(ipPoint); // bind socket to chosen ip
listenSocket->Listen(25); // allow socket to listen connections
```

Перечисление AddressFamily указывает типы адресации, которые Socket может использовать для адресов (IPv4, IPv6 и т.д.).

В параметре SocketType различаются сокеты TCP и UDP. В нем можно определить в том числе: дейтаграммы(требует указать Udp для типа протокола и InterNetwork в параметре семейства адресов), базовый транспортный протокол, потоковые сокеты (Требует указать Tcp для типа протокола).

Третий и последний параметр определяет тип протокола, требуемый для сокета. В параметре ProtocolType можно указать следующие наиболее важные значения - Tcp, Udp, Ip, Raw.

Следующим шагом должно быть назначение сокета с помощью метода Bind(). Когда сокет открывается конструктором, ему не назначается имя, а только резервируется дескриптор. Для назначения имени сокету сервера вызывается метод Bind(). Этот метод связывает сокет с локальной конечной точкой. Вызывать метод Bind() надо до любых попыток обращения к методам Listen() и Accept().

Теперь, создав сокет и связав с ним имя, можно слушать входящие сообщения, воспользовавшись методом Listen(). В состоянии прослушивания сокет будет ожидать входящие попытки соединения.

В параметре метода определяется число соединений, ожидающих обработки в очереди.

- После этого нам необходимо создать цикл, который будет ожидать нового подключения до тех пор, пока сервер запущен.

```
while (isStarted) { // isStarted can be set out of function
 try {
  Socket^ handler = listenSocket->Accept(); // waiting for connection
  connected->Add(handler); // adding new socket to list of connected
  Task^ messageTransfer = gcnew Task((Action<Object^>^)(gcnew Action<Object^>(this,
       &ServerWindow::startMessageTransfering)), handler);
  messageTransfer->Start(); // start message transferring for connected socket
 }
 catch (SocketException^ ex) {
  if (ex->ErrorCode != 10054 && ex->ErrorCode != 10004) {
   MessageBox::Show("Error code: " + ex->ErrorCode + ". " + ex->Message, "Error",
       MessageBoxButtons::OK, MessageBoxIcon::Error);
  }
 }
}
```

Для прерывания цикла создаём глобальную переменную isStarted, которая показывает, запущен ли сервер. Затем мы пытаемся получить соединение с другим сокетом с помощью метода Accept().

После начала прослушивания надо дать согласие на соединение с клиентом, для чего используется метод Accept(). С помощью этого метода получается соединение клиента и завершается установление связи имен клиента и сервера. Метод Accept() блокирует поток вызывающей программы до поступления соединения.

Метод Accept() извлекает из очереди ожидающих запросов первый запрос на соединение и создает для его обработки новый сокет. Хотя новый сокет создан, первоначальный сокет продолжает слушать и может использоваться с многопоточной обработкой для приема нескольких запросов на соединение от клиентов. Он будет продолжать работать наряду с сокетами, созданными методом Accept для обработки входящих запросов клиентов.

Затем добавляем полученный сокет этого клиента в заранее созданный массив, чтобы мы могли получить доступ к нему из-за пределов метода и отключить от сервера из любого другого места.

Затем мы создаем новую задачу, которая будет выполняться параллельно остальной программе и будет обрабатывать получение сообщений клиента, добавляя их в общий чат.

В случае ошибки, если это не потеря соединения с клиентом (это ошибка будет обрабатываться отдельно и будет выведена в чат), выводиться сообщение для администратора.

- Теперь опишем метод, который будем заниматься обработкой сообщений и который вызывается из предыдущего метода.

```
void ServerWindow::startMessageTransfering(Object^ handler) {
 Socket^ localHandler = (Socket^)handler; // casting object to socket
 // creating delegates to change out processes data
 MessageDelegate^ msg = gcnew MessageDelegate(this, &ServerWindow::serverMessage);
 ImageMessageDelegate^ imageMsg = gcnew ImageMessageDelegate(this,
       &ServerWindow::InsertChatImage);
 String^ clientIP = addUser(localHandler); // get username and ip from socket
 int startMessage = chatPool->Count; // start point of message getting
 while (true) { // while will be stopped only by catch or user disconnect
  // wait and getting message from client
  String^ messageContent = getStringMessage(localHandler);
  if (messageContent->Contains("&disconnect")) { // if user want to disconnect
   break;
  }
  if (messageContent->Contains("&get_message=")) { // get request from client
   // get message that client wants to get
   int messageNum = Convert::ToInt32(messageContent->Replace("&get_message=", ""));
   messageNum += startMessage; // client send message num from start point
   sendChatMessage(localHandler, messageNum); // send this message to client
   continue;
  }
  if (messageContent->Contains("&get_num_of_messages")) { // send num of messageы
   int deltaNum = chatPool->Count - startMessage; // count num of message
   sendMessagesNum(localHandler, deltaNum);
   continue;
  }
  if (messageContent->Contains("&image")) { // if user send an image
   Image^ image = getImageMessage(localHandler);
   saveImage(image); // save image to disk
   // insert image in chat
   this->BeginInvoke(imageMsg, (String^)userDB[clientIP], imagePathes->Count - 1);
   continue;
  }
  //this will only be called if none of ifs above worked
  this->BeginInvoke(msg, userDB[clientIP], messageContent);
 }
 try {
  localHandler->Shutdown(SocketShutdown::Both);
  localHandler->Close();
 }
 catch (...) {} // ignore all exceptions
 // insert user left message in chat
 this->BeginInvoke(msg, "SYSTEM", userDB[clientIP] + " left the chat.");
 removeUser(clientIP); // remove user from userDB
 connected->Remove(localHandler);
}
```

Так как метод запускается как отдельная задача и работает параллельно другим процессам, то параметр содержащий сокет передается в виде объекта. Мы явно приводим его к классу сокета. Затем создаем делегатов, которые будут использоваться для вызова методов, изменяющих содержимое интерфейса.

После этого получаем имя клиента с помощью специального метода, который, кроме того, возвращает IP клиента.

Переменнаяя startMessage будет определять начиная с какого сообщения в чате отправлять сообщения клиенту.

Затем, в основном цикле, мы получаем сообщение клиента с помощью метода getStringMessage, проверяем, является ли сообщение запросом и выполняем одно из действий: отправляем запрошенное сообщений, отправляем количество сообщений, обрабатываем получение изображения. Если сообщение не является запросом, то воспринимаем его как простую отправку сообщения пользователем и добавляем в чат.

И в конце, после отключения пользователя или потери соединения с ним, закрываем сокет и удаляем его из списка подключенных пользователей.

- Разберем методы, реализующие получение сообщений.

```
String^ ServerWindow::getStringMessage(Socket^ handler) {
 try {
  NetworkStream^ netStream = gcnew NetworkStream(handler);
  BinaryReader^ reader = gcnew BinaryReader(netStream);
  String^ message = reader->ReadString(); // download message from socket

  reader->Close();
  netStream->Close();
  return message;
 }
 catch (...) { // if there are problem with message getting
  return "&disconnect"; // disconnect user
 }
}
```

```
Image^ ServerWindow::getImageMessage(Socket^ handler) {
 try {
  NetworkStream^ netStream = gcnew NetworkStream(handler);
  BinaryFormatter^ formatter = gcnew BinaryFormatter();

  // deserializaing image from socket data
  Image^ image = (Image^)formatter->Deserialize(netStream);

  // set image max size
  int maxWidth = richTextBoxChat->Width * 9 / 10,
      maxHeight = richTextBoxChat->Height * 9 / 10;
  image = resizeImageWithRatioSave(image, maxWidth, maxHeight);
  saveImage(image); // save image to disk
  return image;
 }
 catch (...) {
  return gcnew Bitmap(1, 1);
 }
}
```

Для получения и отправки данных с помощью сокетов в .NET используется класс потоков NetworkStream из пространства имен System::Net::Sockets. Он не является буферизованным и не поддерживает перемещение в произвольную

позицию. Для удобства работы с таким потоком будем использовать другой класс.

Для записи данных обертываем поток в BinaryReader. Этот класс предназначен для работы с бинарными данными, а именно бинарными потоками.

Получаем сообщение с помощью метода ReadString(). В случае ошибки при получении, возвращаем команду о том, что пользователь отключился и не может отправить сообщение.

В случае получения картинки (класс Bitmap) пользователь должен сериализовать изображение в сетевой поток и отправить серверу.

Кроме сетевых потоков здесь используется еще и десериализация с использование класса BinaryFormatter

Мы напрямую десериализуем картинку из сетевого потока и затем меняем её размер, чтобы она подходила под размеры чата. В случае ошибки возвращаем пустое изображение.

После окончания работы с потоками, они обязательно должны быть закрыты.

(Реализация методов saveImage, resizeImageWithRatioSave одинакова для клиента и сервера и будет приведена позже)

- Разберем методы, реализующие отправку сообщений.

```
void ServerWindow::sendTextMessage(Socket^ handler, String^ message) {
 try {
  NetworkStream^ netStream = gcnew NetworkStream(handler); // make stream of socket
  BinaryWriter^ writer = gcnew BinaryWriter(netStream); // prepare write to net
  writer->Write(message); // send message to client
  writer->Flush(); // close streams
  writer->Close();
  netStream->Close();
 }
 catch (...) {}
}
```

```
void ServerWindow::sendImageMessage(Socket^ handler, String^ imagePath) {
 try {
  BinaryFormatter^ formatter = gcnew BinaryFormatter();
  NetworkStream^ netStream = gcnew NetworkStream(handler);
  Formatter->Serialize(netStream, Image::FromFile(imagePath)); // serializing of image
  netStream->Close();
 }
 catch (...) {}
```

Класс BinaryWriter, как и BinaryReader, служит оболочкой, в которую заключается байтовый поток, управляющий выводом двоичных данных. И затем этот класс используется для записи текста в поток.

Структура метода отправки сообщения аналогична структуре метода получения сообщения. Используются  те же сетевые потоки и их обертки.

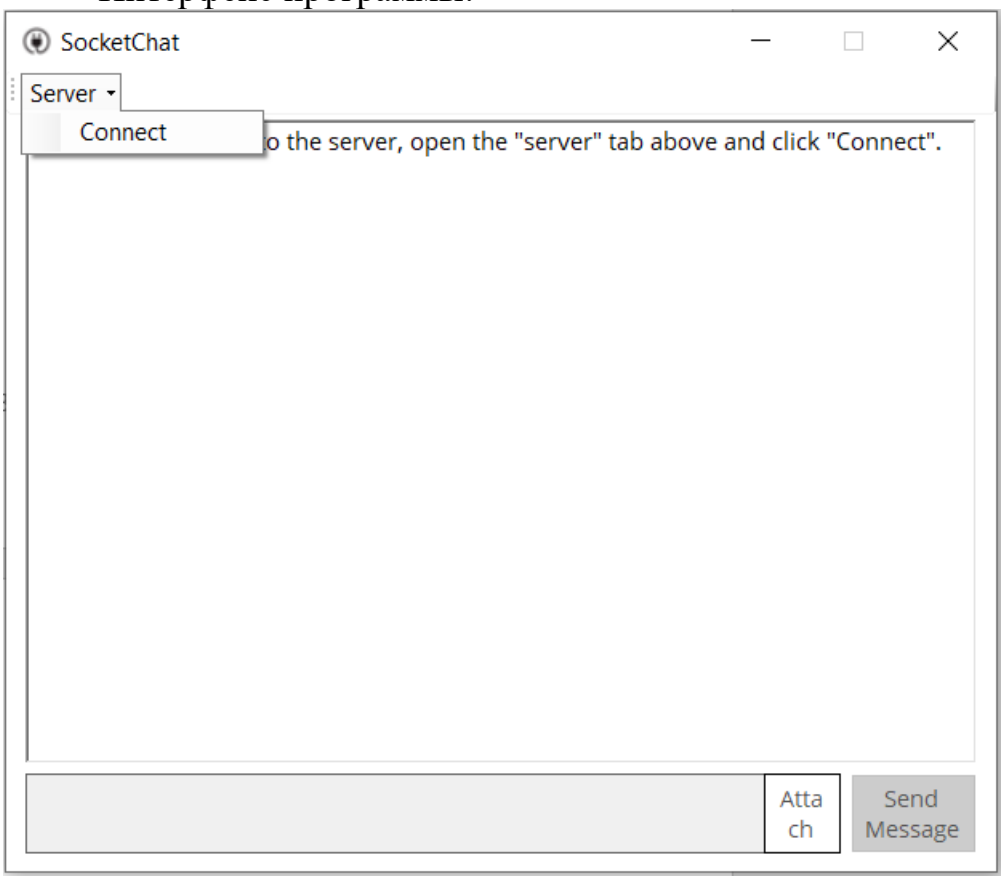
В случае с сериализацией картинки, мы напрямую сериализуем ее в сетевой поток.

После отправки сообщения все открытые потоки должны быть закрыты.

Рассмотрим приложение **Клиент**.

Следующая программа подключается к серверу, данные которого мы вводим в специальную форму, затем начинает отправлять запросы на сервер, ожидая момента, пока на нем не появятся новые сообщения, и затем получает их с помощью сокета. Кроме того, может отправлять сообщения, как текстовые, так и графические.

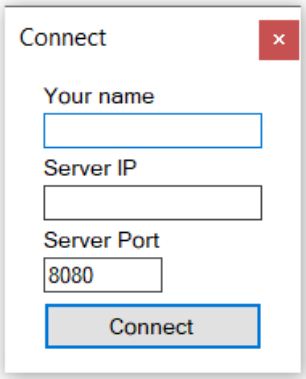
Интерфейс программы:




Интерфейс имеет:

- richTextBox, в котором будет отображаться получаемая информация.
- textBox для ввода сообщений.
- Кнопку для отправки сообщения.
- Кнопку для отправки изображения.
- toolStrip с кнопкой присоединения.

При нажатии на кнопку connect вызывается новая форма, в которую вводится информация о сервере. Она взаимодействует с основной формой посредством созданных в основной форме методов get-set.




Алгоритм работы с клиентским сокетом такой же, как и с серверным, за исключением использования метода подключения к серверу. Используются те же методы для определения конечной точки, создания экземпляра сокета, отправки и получения данных и закрытия сокета.

Рассмотрим основные методы клиентского приложения:

- Для того, чтобы начать отправку сообщений, сначала необходимо подключиться к серверу.

```
void ClientWindow::connectToTheServer() {
 try {
  IPEndPoint^ ipPoint = gcnew IPEndPoint(ip, port);
  messageSocket = gcnew Socket(AddressFamily::InterNetwork, SocketType::Stream,
    ProtocolType::Tcp);
  messageSocket->Connect(ipPoint);
  sendSystemMessage(name); // send user name to server

  Task^ messageTransfering = gcnew Task(gcnew Action(this,
    &ClientWindow::startMessageTransfering));
  messageTransfering->Start();

  connectSuccess = true;
 }
 catch (Exception^ e) {
  MessageBox::Show("Connection error!\n" + e->Message, "Error", MessageBoxButtons::OK,
    MessageBoxIcon::Error);
 }
}
```

Для подключения мы создаем точку IP с помощью введенных ранее ip и port. А затем подключаемся к серверу.

В случае успешного подключения отправляем с помощью метода, использующего тот же алгоритм, что и отправка сообщения на сервере, введенное ранее имя пользователя для идентификации на сервере.

Затем запускаем задачу (Task), которая начинает общение с сервером. И, после успешного выполнения, устанавливаем флаг, обозначающий подключение к серверу на true.

- Теперь рассмотрим метод, занимающийся обработкой сообщений.

```
void ClientWindow::startMessageTransfering() {
 MessageDelegate^ msg = gcnew MessageDelegate(this, &ClientWindow::addChatMessage);
 ImagePathDelegate^ imageMsg = gcnew ImagePathDelegate(this,
   &ClientWindow::insertChatImage);
 int currentMessage = 0; // indicate how many messages already downloaded
 while (true) {
  System::Threading::Thread::Sleep(250); // wait for message asking
  sendSystemMessage("&get_num_of_messages"); // request for messages num

  String^ toConvert = getStringMessage(); // get asked messages num
  if (toConvert->Contains("&disconnect")) { // if server don't response
   this->BeginInvoke(gcnew EventDelegate(this, &ClientWindow::itemDisconnect_Click),
     gcnew Object, gcnew EventArgs);
   break;
  }
  int numOfMessages = Convert::ToInt32(toConvert);
  if (numOfMessages == currentMessage) {
   continue; // wait for new messages
  }
  while (currentMessage != numOfMessages) { // update messages pool
   // send request for message
   sendSystemMessage("&get_message=" + currentMessage++);
   String^ messageContent = getStringMessage();
   if (messageContent->Contains("&image")) { // if next message is img
    Image^ image = getImageMessage();
    saveImage(image); // save image to disk
    String^ path = imagePathes[imagePathes->Count - 1];
    this->BeginInvoke(imageMsg, path);
    continue; // do not add "&image" to the chat
   }
   if (messageContent->Contains("&disconnect")) {
    break;
   }
   this->BeginInvoke(msg, messageContent);
  }
 }
}
```

Так как метод запускается как отдельная задача и работает параллельно другим процессам, то мы создаем делегатов, которые будут использоваться для вызова методов, изменяющих содержимое интерфейса.

Переменная currentMessage будет использоваться для проверки, появились ли новые сообщения на сервере.

Затем, в основном цикле, мы отправляем на сервер запрос на количество сообщений с помощью метода sendSystemMessage, который реализует отправку сообщений (описанную в приложении сервера), и если это количество изменилось с прошлого раза, то мы отправляем запрос на получение этих сообщений и ожидаем их получения.

Таким образом, сервер всегда только отвечает на запросы, а не оповещает клиентов самостоятельно. Это помогает в какой-то степени разгрузить сервер и используется большинством приложений типа Клиент/Сервер.

В случае, если потеряно соединение или сервер разорвал соединение, выводиться сообщение с помощью MessageBox о получении ошибки и работа цикла оканчивается.

Рассмотрим работу с классом **Bitmap** (Изображениями):

- Для сохранения изображений используется следующий метод:

```
void ClientWindow::saveImage(Image^ image) {
 // forming image path to download directory
 String^ imagePath = Application::StartupPath + "/downloads/img" + (imagePathes->Count
   + 1) + ".jpg";
 (gcnew FileInfo(imagePath))->Directory->Create();
 image->Save(imagePath, Imaging::ImageFormat::Jpeg);
 imagePathes->Add(imagePath);
}
```

Сохранять изображения будем в папку downloads, которая будет размещаться в папке с исполняемым файлом. Путь к исполняемому файлу можно узнать с помощью обращения к переменной Application::StartupPath.

Затем если папка downloads не существует, то она создается. И по составленному пути сохраняется изображение с помощью метода Save. Путь к этому файлу сохраняется в List<String^>^ imagePathes для дальнейшего использования.

- Для изменения размера используется следующий метод:

```
Image^ resizeImage(Image^ image, int width, int height) {
 Bitmap^ resizedImage = gcnew Bitmap(width, height);
 Graphics^ g = Graphics::FromImage(resizedImage);
 g->InterpolationMode = Drawing2D::InterpolationMode::High;
 g->DrawImage(image, 0, 0, width, height);
 return resizedImage;
}
```

Мы создаем новое изображения с новым размером (он указывается в пикселях) и начинаем работать с графикой этого изображения.

Устанавливаем качество интерполирования новых пикселей и рисуем старое изображение поверх нового используя интерполирование.

Возвращаем новое изображение с измененным размером.

Но для изображений зачастую необходимо оставить исходное соотношение сторон (16:9, 4:3, 1:1 и т.д.), чтобы не разрушить ее наполнение. Это требует некоторых расчетов для новых сторон и использование resizeImage метода.

- В случае, если надо сохранить соотношение сторон изображения используется следующий метод:

```
Image^ resizeImageWithRatioSave(Image^ image, int maxWidth, int maxHeight) {
 int width = image->Width,
  height = image->Height;
 if (width > maxWidth) {
  width = maxWidth;
  height = maxWidth * image->Height / image->Width;
 }
 if (height > maxHeight) {
  height = maxHeight;
  width = image->Width  * maxHeight / image->Height, maxHeight;
 }
 if (image->Width != width || image->Height != height) {
  image = resizeImage(image, width, height);
 }
 return image;
}
```

Этот метод принимает изображения, и максимально допустимый размер в ширину и высоту. В случае если изображение не соответствует ширине или высоте, то оно приводится к максимально допустимой величине с сохранением соотношения сторон с помощью метода resizeImage.

Затем возвращается новое изображение.

- Для вставки изображения в чат используется следующий метод:

```
void ClientWindow::insertChatImage(String^ imagePath) {
 DataFormats::Format^ imageFormat = DataFormats::GetFormat(DataFormats::Bitmap);
 String^ oldClipData = Clipboard::GetText();
 Clipboard::SetImage(Image::FromFile(imagePath));
 richTextBoxChat->ReadOnly = false;
 richTextBoxChat->Focus();
 if (richTextBoxChat->CanPaste(imageFormat)) {
  richTextBoxChat->Paste(imageFormat);
 }
 richTextBoxChat->ReadOnly = true;
 Clipboard::SetText(oldClipData);
}
```

Сначала мы получаем формат изображения, затем старое текстовое содержимое из буфера обмена.

Для вставки изображения мы перемещаем изображение в буфер обмена, делаем richTextBox (специальное поле Windows Forms позволяющее отображения изображений) доступным для записи и фокусируемся на нем. Если изображение можно вставить в richTextBox, то вставляем его. Опять делаем richTextBox недоступным для записи и возвращаем старый буфер обмена.

## Инструкция по использованию

**Сервер:**

1. Запустить исполняемый файл “SocketChat-Server.exe”.
2. Ввести необходимый порт.
3. Выбрать IP на котором будет запущен сервер.
4. Нажать кнопку “Start Server”
5. Использование во время работы сервера:
    - Для отправки сообщений используется нижнее текстовое поле. После ввода сообщения необходимо нажать “enter” на клавиатуре или кнопку “send”.
    - Чтобы отправить изображение необходимо нажать кнопку “attach” и выбрать “attach image”, затем в специальной форме выбрать изображение и нажать “ok”.
    - В левой части интерфейса есть окно выбора пользователей, там можно выбрать несколько пользователей и по нажатию кнопки “disconnect” они будут отключены от сервера.
6. Для отключения сервера достаточно нажать кнопку “Shutdown Server”.

**Клиент:**

1. Запустить исполняемый файл “SocketChat-Client.exe”.
2. Нажать в меню сверху на “Server”, затем на “Connect”.
3. Заполнить появившуюся форму, введя port, ip и username. Затем нажать “Connect”. При успешном подключении появиться возможность отправки сообщений.
4. Использование во время работы:
    - Для отправки сообщений используется нижнее текстовое поле. После ввода сообщения необходимо нажать “enter” на клавиатуре или кнопку “send”.
    - Чтобы отправить изображение необходимо нажать кнопку “attach” и выбрать “attach image”, затем в специальной форме выбрать изображение и нажать “ok”.
5. Чтобы отключиться от сервера, нужно нажать в меню сверху на “Server”, затем на “Disconnect”.

## Заключение

Сокет — это один конец двустороннего канала связи между двумя программами, работающими в сети. Соединяя вместе два сокета, можно передавать данные между разными процессами.

Сокет играет важную роль в сетевом программировании, обеспечивая функционирование как клиента, так и сервера. Сокеты используются в приложениях, в которых необходима передача данных через протокол TCP или UDP.

Для выполнения курсовой работы были изучены возможности Visual Studio 2017 и платформы .NET для разработки приложений типа Клиент/Сервер с использование сокетов с возможностями параллельной обработки данных.

Практическую часть курсовой работы составляет разработка чата на основе протокола TCP/IP типа Клиент/Сервер и рассмотрение программных возможностей для разработки приложений такого типа.

## Список Литературы


1. Архангельский А.Я., Тагин М.А. Приемы программирования в C++ Builder 6. – «Издательство Бином», 2004.
2. https://metanit.com/sharp/net/3.1.php, 14.05.2020.
3. https://professorweb.ru/my/csharp/web/level3/3_1.php, 14.05.2020.
4. https://lecturesnet.readthedocs.io/net/low-level/ipc/socket/intro.html, 15.05.2020.