Лабораторная работа №8

Разработка сетевых приложений

Теоретическая часть

В языке C# поддерживается как стандартный интерфейс сокетов, описанный в лекции 8, так и упрощенный интерфейс значительно облегчающий разработку простых сетевых программ. Упрощенные классы сокетов помогают программисту создавать сетевые программы используя более простые конструкции и минимальный объем кода. В этой лабораторной работе рассмотрим создание сетевых приложений на языке C# с использованием классов UdpClient, TcpClient и TcpListener.

TcpClient

Методы класса TcpClient используются для создания клиентских сетевых программ, которые используют протокол с установкой соединения TCP.

Существует три способа создания объекта TcpClient и подключения его к удаленному хосту.

Конструктор по умолчанию

Формат конструктора по умолчанию создает сокет на любом доступном локальном порту. Затем вы можете использовать метод Connect () для подключения к указанному удаленному хосту:

```
TcpClient newclient = new TcpClient ();
newclient.Connect (“www.isp.net”, 8000);
```

Первый оператор создает новый объект TcpClient и связывает его с локальным адресом и портом. Вторая строка соединяет сокет с удаленным адресом хоста и номером порта. Обратите внимание, что адрес удаленного сервера может быть указан как в виде IP адреса, так и в виде доменного имени.

Конструктор с параметром EndPoint.

Второй вид конструктора позволяет указать конкретный объект EndPoint для использования при создании сокета:

```
IPAddress ia = Dns.GetHostByName(
Dns.GetHostName()).AddressList[0];
IPEndPoint iep = new IPEndPoint(ia, 10232);
```

```
TcpClient newclient2 = new TcpClient(iep);
newclient2.Connect("www.isp.net", 8000);
```

Конструктор с указанием удаленного сервера

Третий вид конструктора является наиболее часто используемым. Он позволяет указать адрес и порт удаленного сервера для подключения внутри конструктора, и таким образом избавляет от необходимости вызывать метод Connect().

```
TcpClient newclient3 = new TcpClient ("www.isp.net", 8000);
```

После создания экземпляра TcpClient можно отправлять и получать данные на удаленный сервер. Метод GetStream() используется для создания объекта NetworkStream, который позволяет отправлять и получать байты в сокет. После того, как получен экземпляр NetworkStream для сокета, можно использовать стандартные методы Read () и Write () для чтения и записи в сокет.

Важно! При завершении программы все активные подключения должны быть закрыты используя метод Close().

TcpListener

Так же как и класс TcpClient упрощает разработку клиентских приложений, класс TcpListener упрощает разработку серверных программ. Их конструкторы тоже похожи:

- Конструктор с указанием порта TcpListener(int port);
  Этот конструктор позволяет создать экземпляр класса TcpListener и сразу же подключить его к указанному локальному порту.
- Конструктор с указанием IPEndPoint: TcpListener(IPEndPoint ep);
  Позволяет подключиться к указанному интерфейсу
- Конструктор с указанием IP адреса и порта: TcpListener(IPAddress addr, int port);

После создания экземпляра класса TcpListerer необходимо вызвать метод AcceptTcpClient, который запустит процесс ожидания подключения клиента и в случае успеха вернет экземпляр класса TcpClient ассоциированный с подключенным клиентом.

Внимание! Метод AcceptTcpClient() является блокирующим.

После установления связи, сервер может отправлять и принимать данные подключенному клиенту используя полученный экземпляр TcpClient.

Важно! При завершении программы все активные подключения должны быть закрыты используя метод Close().

UdpClient

Для приложений, которым требуется передача данных без установки подключения использующая протокол UDP, C# предоставляет класс UdpClient. Класс UdpListener отсутствует, так как попросту не нужен, а UdpClient предоставляет все возможности необходимые для разработки как UDP клиента, так и UDP сервера.

Так как протокол UDP не подразумевает установки подключения, то весь процесс взаимодействия клиента и сервера сводятся к прослушиванию порта с одной стороны и передачи в него данных с другой. Для этого используются методы Receive и Send.

Передача данных в UDP

Для передачи данных в UPD сокет класс UdpClient предоставляет метод Send(). В качестве параметров в этот метод можно передать адрес и порт получателя, данные в виде массива байтов и размер отправляемых данных.

Прием данных по UDP

Для приема данных из UDP сокета класс UdpClient предоставляет метод Receive().

Внимание! Метод Receive является блокирующим.

Использование потоков выполнения

Поток — это по сути последовательность инструкций, которые выполняются параллельно с другими потоками. Каждая программа создает по меньшей мере один поток: основной, который запускает функцию main(). Программа, использующая только главный поток, является однопоточной; если добавить один или более потоков, она станет многопоточной.

Стандартное приложение C# является однопоточным. Внутренние расчеты, обработка действий пользователя, чтение файлов и сетевое взаимодействие выполняется в одном потоке, то есть друг за другом. Это работает хорошо до тех пор, пока в программе не появляются блокирующие вызовы, которые останавливают работающий поток.

Блокирующий вызов – это такой вызов метода, который может выполняться бесконечно долго, до тех пор, пока не произойдет событие, которое ожидает этот метод или не случится исключительная ситуация. В предыдущем разделе мы рассмотрели ряд классов для передачи данных по сети. К некоторым из методов этих классов была дана пометка, что данный метод является блокирующим. В частности, блокирующим является метод ожидания подключения клиента AcceptTcpClient. Действительно, после вызова

этого метода выполнение потока будет приостановлено до тех пор, пока к этому сокету не будет подключен клиент. Во время такой остановки потока программа будет выглядеть «зависшей» и не будет отвечать ни на какие действия пользователя.

Использование многопоточного подхода - это способ выполнять несколько действий одновременно. Это может быть полезно, для отображения анимации и обработки пользовательского ввода данных во время загрузки изображений или звуков.

Именно потоки широко используется в сетевом программировании, во время ожидания данных, приложение будет продолжать обновляться и отвечать на действия пользователя.

Для создания и управления потоками C# предоставляет множество различных методов. В данной работе рассмотрим класс Thread.

Важно! При завершении программы все потоки должны быть остановлены!

Создание потока

При создании потока необходимо вызвать конструктор класса Thread с параметром – функцией, которая будет выполняться в новом потоке.

```
Thread other = new Thread(func);
```

Если функции требуются какие-либо параметры, то такой вызов можно записать в лямбда-виде:

```
Thread thread = new Thread(() => func(params));
```

После создания потока, его можно запустить:

```
thread.Start();
```

В момент вызова метода Start, поток будет запущен на выполнение параллельно с основным потоком.

Остановка потока.

После того, как поток выполнил свою задачу, его необходимо остановить. Это может быть выполнено двумя способами:

- Вызвать метод Abort(). Поток будет аварийно завершен.
- Выйти из функции потока. Если во время работы потока произойдет выход из корневой функции, то такой поток будет автоматически остановлен

Теперь, используя полученную информацию, попробуем разработать приложение использующее сетевые технологии.

Практическая часть

Программа для обмена сообщениями точка-точка.

В качестве примера рассмотрим программу использующую протокол UDP для передачи текстовых сообщений между двумя компьютерами. Для того, чтобы компьютеры могли общаться друг с другом, требуется следующее:

- Открытый принимающий UDP сокет на доступном порту каждом из компьютеров.
- При отправке сообщения отправитель должен открыть подключение по адресу получателя, указав порт, который тот прослушивает.
- Отправляемые данные должны быть закодированы в последовательность байтов при отправке и декодированы по приему. Для передачи текстовых данных рекомендуется использовать стандартную и распространенную кодировку, например, UTF-8.
- Так как каждый компьютер должен постоянно находиться в состоянии приема данных, процесс приема и обработки входящих сообщений должен быть вынесен в отдельный поток.

Создадим новый проект WPF приложения. Для начала сделаем набросок пользовательского интерфейса:

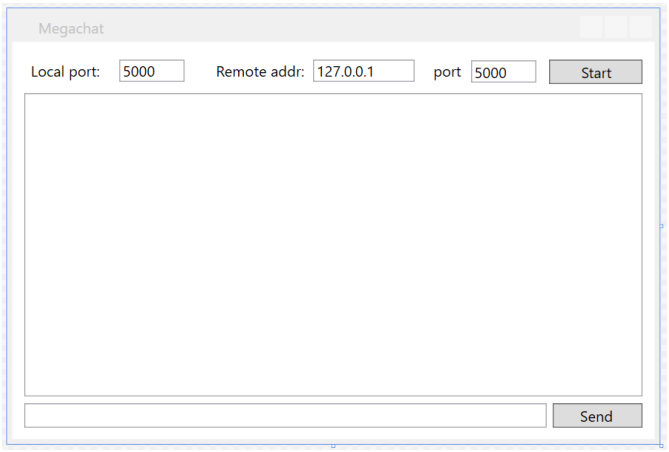

Рис. 1. Внешний вид программы в редакторе форм

На форме находятся следующие элементы:

- Поле ввода локального порта
- Поле ввода удаленного адреса и порта
- Кнопка начала чата
- Окно чата
- Поле ввода сообщения
- Кнопка отправки сообщения на удаленный компьютер

После этого перейдем к написанию кода. Начнем с кнопки начала чата. По нажатию этой кнопки, мы будем создавать новый поток, в котором будет открываться порт для приема данных от удаленного компьютера.

Листинг 1. Обработчик кнопки начала чата

```
private void pbStart_Click(object sender, RoutedEventArgs e)
{
    // Получаем номер порта который будет слушать наш клиент
    int port = int.Parse(tbLocalPort.Text);
    // Запускаем метод Receiver в отдельном потоке
    Thread tRec = new Thread(() => Receiver(port));
    tRec.Start();
}
```

Для использования класса Thread необходимо подключения библиотеки `using System.Threading;`.

После этого нам необходимо написать сам метод Receiver, в котором будет создаваться принимающий сокет.

Листинг 2. Метод открывающий принимающий сокет и вычитывающий входящие данные

```
private void Receiver(int port)
{
    // Создаем UdpClient для чтения входящих данных
    receivingUdpClient = new UdpClient(port);
    IPEndPoint remoteIpEndPoint = null;

    try
    {
        while (true)
        {
            // Ожидание дейтаграммы
            byte[] receiveBytes = receivingUdpClient.Receive(ref remoteIpEndPoint);

            // Преобразуем байтовые данные в строку
            string returnData = Encoding.UTF8.GetString(receiveBytes);

            // Выводим данные из нашего потока в основной
            lbLog.Dispatcher.BeginInvoke(new Action( () => addMessage(" --> " + returnData)));
        }
    }
    catch (Exception ex)
    {
        // В случае ошибки завершаем поток и выводим сообщение в лог чата
        lbLog.Dispatcher.BeginInvoke(new Action(() =>
            addMessage("Возникло исключение: " + ex.ToString() + "\n  " + ex.Message))
        );
    }
}
```

Основная часть этого метода – это бесконечный цикл, который выполняет три действия:

- Открывает порт `port` и ожидает входящих данных
- Декодирует входящие данные из кодировки UTF-8 в класс `string`
- Пересылает сообщение из потока чтения в поток графического интерфейса, используя метод `Dispatcher.BeginInvoke`.

В случае возникновения ошибок в окно чата так же отправляется сообщение о причине ошибки.

Затем переходим к действию по нажатию кнопки `Send`. В этом обработчике мы должны выполнить следующие действия:

- Создать сокет для подключения к удаленному компьютеру.
- Подготовить структуру данных с информацией о подключении. Она включает в себя удаленный адрес и порт.
- Закодировать сообщение, используя UTF-8.
- Отправить полученную дейтаграмму в созданный сокет.

Листинг 3. Обработчик нажатия на кнопку отправки сообщения

```
private void pbSend_Click(object sender, RoutedEventArgs e)
{
    // Создаем сокет для отправки данных
    UdpClient other = new UdpClient();

    // Создаем endPoint по информации об удаленном хосте
    IPEndPoint endPoint = new IPEndPoint(IPAddress.Parse(tbRemoteAddr.Text),
                                         int.Parse(tbRemotePort.Text));

    try
    {
        // Преобразуем данные в массив байтов, используя кодировку UTF-8
        byte[] bytes = Encoding.UTF8.GetBytes(tbMessage.Text);

        // Отправляем данные
        other.Send(bytes, bytes.Length, endPoint);
        // выводим наше сообщение в лог
        addMessage(tbMessage.Text);
        // очищаем поле ввода
        tbMessage.Clear();
    }
    catch (Exception ex)
    {
        // в случае ошибки выводим сообщение в лог
        addMessage("Возникло исключение: " + ex.ToString() + "\n  " + ex.Message);
    }
    finally
    {
        // Закрыть соединение
        other.Close();
    }
}
```

Так как протокол UDP не подразумевает установки подключения, созданный сокет необходимо закрыть сразу же после отправки сообщения.

В листингах выше для вывода сообщения в окно лога использовался метод `addMessage`. Вот его код:

Листинг 4. Метод для вывода сообщений в окно лога

```
private void addMessage(string message)
{
    // добавляем с сообщению метку времени и выводим в лог
    lbLog.Items.Add(DateTime.Now.ToString() + " > " + message);

    // прокручиваем лог до самого последнего сообщения
    var border = (Border)VisualTreeHelper.GetChild(lbLog, 0);
    var scrollViewer = (ScrollViewer)VisualTreeHelper.GetChild(border, 0);
    scrollViewer.ScrollToBottom();
}
```

Последним штрихом будет корректная обработка закрытия окна. Так как при запуске создается поток, выполняющийся бесконечно, наша программа будет продолжать свою работу даже после закрытия окна. Чтобы этого избежать, необходимо останавливать поток одновременно с закрытием окна. Для этого воспользуемся обработчиком события `Closed` окна.

Листинг 5. Обработчик закрытия окна

```
private void Window_Closed(object sender, EventArgs e)
{
    // при закрытии окна обязательно закрываем принимающий сокет
    if (receivingUdpClient != null)
        receivingUdpClient.Close();
}
```

В этом обработчике мы вызываем метод `Close` у принимающего сокета. Это вызывает исключение System.Net.Sockets.SocketException в потоке чтения и приводит к завершению бесконечного цикла. После этого поток завершает свою работу.

Теперь можно проверить работу программы. Так как сетевые подключения можно устанавливать в том числе и внутри одного компьютера, то для проверки мы можем запустить два экземпляра программы и в качестве адреса удаленного компьютера указать loopback адрес 127.0.0.1. При этом важно, чтобы у каждого экземпляра различались порты, так как в каждый момент времени только одна программа может использовать определенный порт.

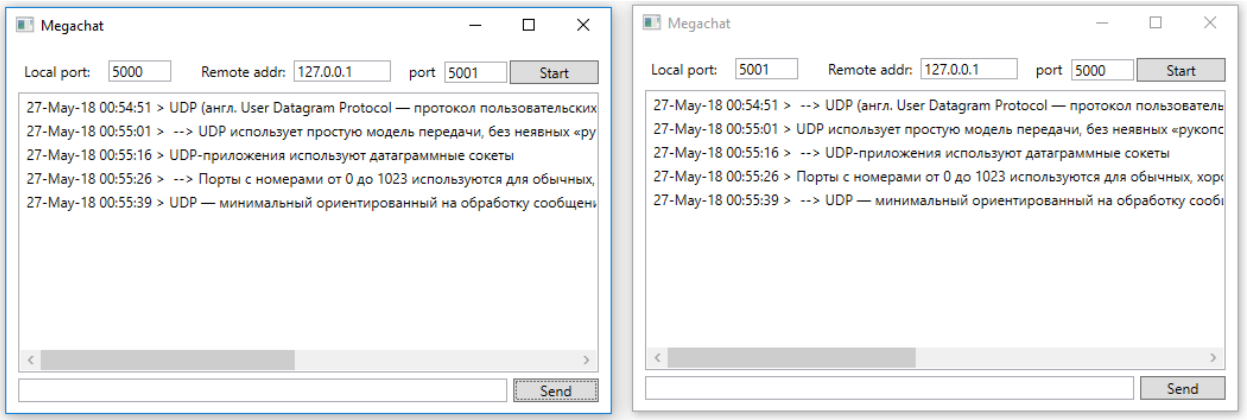

Рис. 2. Пример работы программы

Видим, что программа работает, и сообщения передаются из одного экземпляра в другой.

Данный подход к разработке сетевых приложений является бессерверным или peer-to-peer, так как каждый из экземпляров приложения является равнозначным по отношению к другим. У него есть как плюсы, такие как отсутствие посредника, более высокая скорость и безопасность передачи данных, так и минусы – сложности при организации работы более 2 участников, необходимость в прямой сетевой видимости и др. Альтернативным решением является использование клиент-серверной архитектуры.

Рис. 3. Peer to peer Client - Server (справа)

Если мы захотим расширить нашу программу для совместной работы трех и более человек, нам понадобится устанавливать соединение с каждым узлом, либо строить автоматическую систему трансляции сообщений к каждому из узлов. Однако, если мы сменим архитектуру на клиент-серверную, задача значительно упростится.

Разработка клиент-серверной программы обмена сообщениями

В клиент-серверной архитектуре, как и следует из названия, существует два типа узлов. Первый тип – это сервер, центральный узел осуществляющий прием данных и маршрутизацию сообщений. Сервер взаимодействует с каждым из узлов. Второй –

Клиент, пользовательское ПО обращающееся напрямую только к серверу.

Создадим новый проект WPF приложения. В качестве названия проекта укажем Server, а название решения – ClientServerChat, как указано на рисунке.

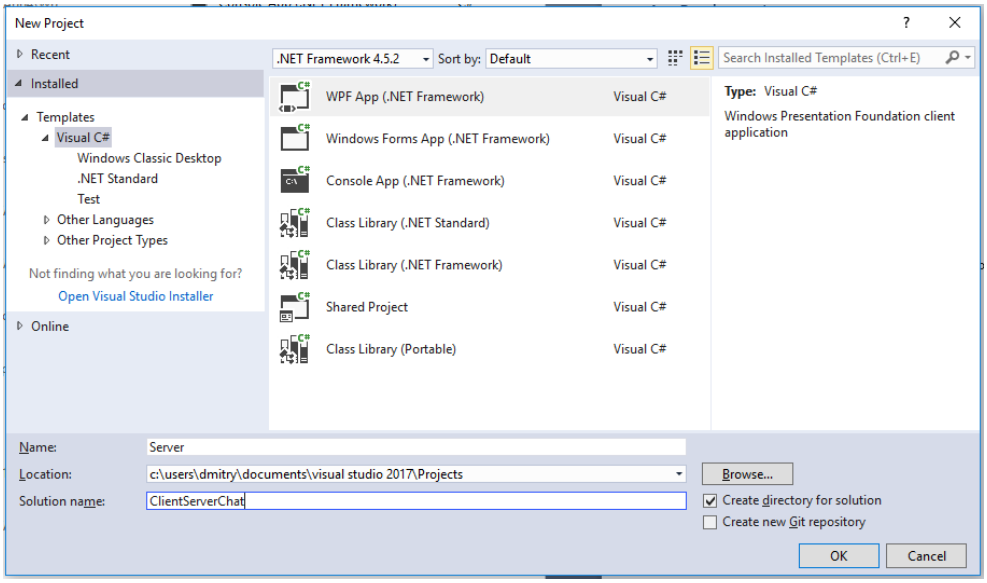

Рис. 4. Создание проекта

После создания решения и проекта, необходимо так же создать проект Клиента, так как это должны быть два отдельных приложения. Для этого выбираем «File\Add\New Project…» и в качестве названия указываем Client.

Обозреватель решений должен выглядеть следующим образом:

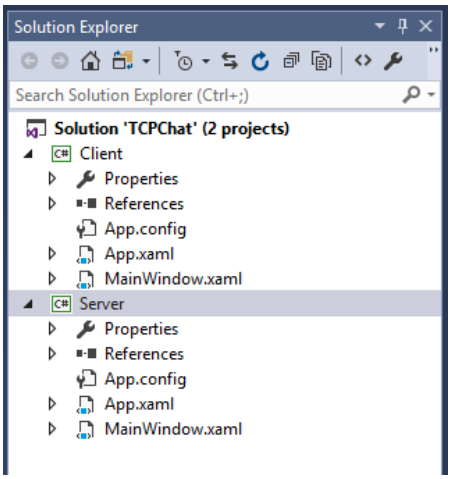

Рис. 5. Обозреватель решений с двумя проектами

Далее необходимо настроить решение таким образом, чтобы клиент и сервер запускались одновременно, что полезно для отладки. Переходим в настройки решения и указываем, что при запуске нужно запускать оба проекта, как показано на рисунке:

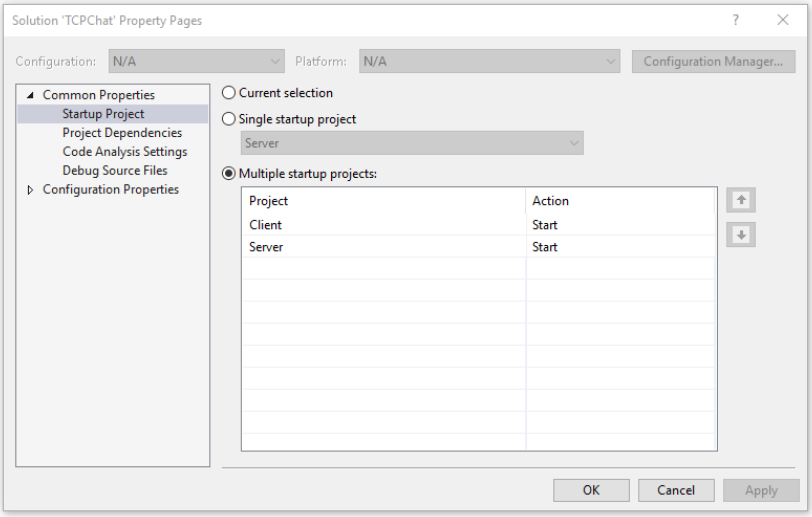

Рис. 6. Настройка режима запуска проектов

После завершения подготовительных работ, переходим к написанию приложений. Начнем с серверной части.

Разработка серверной части приложения

Добавим на форму приложения несколько элементов для управления сервером. В окне сервера будет одна кнопка для запуска сервера чата и окно для вывода событий, которые будут происходить во время работы.

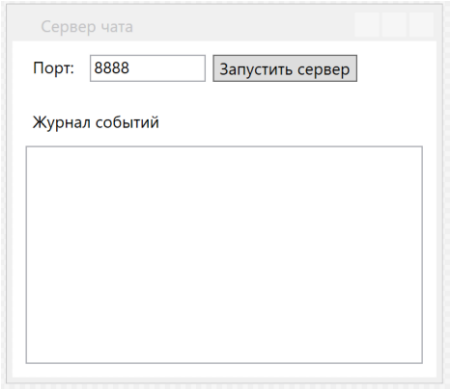

Рис. 7. Окно сервера чата

Запуск сервера.

Откроем обработчик нажатия кнопки и напишем следующий код:

Листинг 6. Запуск TCP сервера

```
private void Start_Click(object sender, RoutedEventArgs e)
{
    // Считываем номер порта
    int port = int.Parse(tbPort.Text);
    // Создаем слушающий сокет
    serverSocket = new TcpListener(IPAddress.Any, port);
    serverSocket.Start();
    // Создаем и запускаем поток ожидания нового клиента
    Thread thread = new Thread(waitForClient);
    thread.Start();
}
```

В этом обработчике происходит создание сокета и потока в котором сервер будет ожидать подключения внешних клиентов. Ожидание подключения в новом потоке необходимо для того, чтобы интерфейс программы оставался отзывчивым. Так же, по мере подключения новых клиентов, для каждого из них будет создаваться новый поток.

В этой функции мы запускаем процесс ожидания (прослушки) нашего порта и ждем, когда к серверу обратится клиент. Вызов AcceptTcpClient – блокирующий, это означает, при его вызове поток будет остановлен до тех пор, пока не подключится клиент или не произойдет ошибка. Если бы мы вызвали этот метод в основном потоке приложения, то программа бы «зависла» и перестала отвечать на действия пользователя. Но так как мы вызвали этот код в отдельно созданном потоке, программа продолжит работать как ни в чем не бывало.

Подключение клиента

После того как клиент подключится к серверу, будет создан новый сокет - **clientSocket**, соединяющий сервер с данным конкретным клиентом. Если наш сервер будет обрабатывать 100 клиентов, то в нашей программе одновременно будет создано 100 сокетов на каждого. Поэтому необходимо передать **clientSocket** в основной поток приложения для дальнейшей обработки. Для этого напишем функцию **addClient**, которая будет добавлять клиента в общий список и создать для него поток чтения\записи. Однако, эта функция должна быть вызвана из основного потока приложения. Для этого необходимо воспользоваться Диспетчером потоков, который позволяет вызвать требуемую функцию в нужном потоке. Для этого служит вызов **BeginInvoke**.

Листинг 7. Функция ожидания нового клиента

```
private void waitForClient()
{
    // Пока сокет подключен к порту
    while (serverSocket.Server.IsBound)
    {
        try
        {
            // Начинаем ожидание нового клиента
            // Эта функция блокирует поток и прервется
            // либо по исключению
            // либо по подключению нового клиента
            var clientSocket = serverSocket.AcceptTcpClient();
            // если мы попали сюда, значит к нам подключился клиент
            // передаем сокет клиента в основной поток
            SrvWindow.Dispatcher.BeginInvoke(
                new Action(() => addClient(clientSocket)));
        }
        catch (SocketException)
        {
            continue;
        }
    }
}
```

В функции addClient происходит создание и запуск потока для обработки ввода\вывода с новым клиентом.

Листинг 8. Функция добавления нового клиента

```
private void addClient(TcpClient client)
{
    // Добавляем клиента в список
    clients.Add(client);
    // Выводим сообщение
    lbLog.Items.Add("Client connected: " + client);
    // Создаем и запускаем новый поток
    Thread thread = new Thread(() => clientThread(client));
    thread.Start();
}
```

Обработка данных клиента

В созданном потоке clientThread будем принимать данные от клиента и ретранслировать его сообщения всем остальным. Таким образом получится, так что все что отправил один клиент будут получать все остальные. Прием данных от клиента осуществляется через поток (stream) ввода\вывода. При вызове метода Read() сервер будет ожидать данных от клиента. Затем эти данные необходимо преобразовать из бинарного вида в текст, используя кодировку UTF-8.

Листинг 9. Прием данных от клиента

```
private void clientThread(TcpClient client)
{
    // буфер чтения
    byte[] bytesFrom = new byte[10000];
    string dataFromClient = null;

    while (true)
    {
        if(!client.Connected) // если клиент отключился, то закрываем соединение
        {
            lbLog.Dispatcher.BeginInvoke(new Action(() => disconnect(client)));
            return;
        }
        // Получаем поток ввода\вывода клиента
        NetworkStream networkStream = client.GetStream();
        // Читаем данные пришедшие от клиента
        int read = networkStream.Read(bytesFrom, 0, 10000);
        if (read > 0)
        {   // Если клиент что-то прислал, то декодируем сообщение
            dataFromClient = Encoding.UTF8.GetString(bytesFrom, 0, read);
            // Ретранслируем сообщение остальным клиентам
            lbLog.Dispatcher.BeginInvoke(new Action(() => relayMessage(client, dataFromClient)));
        }
        else
        {
            Thread.Sleep(100);
        }
    }
}
```

В функции пересылки сообщений перебираем каждого клиента, и отправляем принятое сообщение в его поток ввода\вывода, предварительно закодировав его в UTF-8.

Листинг 10. Пересылка сообщений клиентам

```
private void relayMessage(TcpClient client, String message)
{
    // Проходим по каждому из клиентов
    foreach (var c in clients)
    {
        // Получаем его поток ввода\вывода
        var clientStream = c.GetStream();
        // Кодируем сообщение в UTF-8
        byte[] outStream = System.Text.Encoding.UTF8.GetBytes(message);
        // Отправляем данные
        clientStream.Write(outStream, 0, outStream.Length);
        clientStream.Flush();
    }
    // Выводим сообщение в лог сервера для отладки
    addMessage(message);
}
```

После этого остались только мелочи:

- При отключении клиента его нужно удалить из списка, чтобы не пытаться пересылать ему сообщения в будущем
- При закрытии приложения необходимо останавливать все потоки и закрывать все

сокеты, иначе наша программа не закроется после закрытия окна и будет висеть в фоне.

Листинг 11. Отключение клиента и закрытие приложений

```
1 reference
private void disconnect(TcpClient client)
{
    addMessage("Client disconnected");
    clients.Remove(client);
}

1 reference
private void Window_Closed(object sender, EventArgs e)
{
    serverSocket.Stop();
    foreach (var c in clients)
    {
        c.Close();
    }
}
```

Это все что касается сервера. Перейдем к разработке клиента.

Разработка TCP клиента

Как и в случае с сервером, начнем с создания интерфейса. На клиенте интерфейс будет немного сложнее. Добавим на форму несколько полей ввода, пару кнопок и метку:

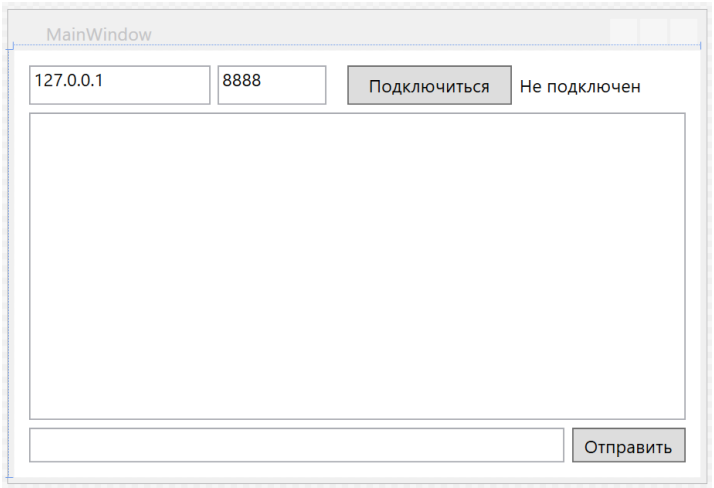

Рис. 8. Интерфейс TCP клиента

На форме должны быть расположены текстовые поля ввода адреса и порта сервера, кнопка подключения к серверу, поле ввода текстового сообщения и кнопка отправки сообщения. Основную часть приложения занимает ListBox, в который будут выводиться сообщения чата. В правом верхнем углу расположена метка (Label) отображающая статус подключения к серверу.

После расположения всех необходимых элементов, переходим к написанию кода.

Программа будет состоять из трех основных частей:

- Установка подключения к серверу
- Отправка сообщения на сервер
- Прием сообщений от сервера и вывод их в область чата

Рассмотрим их по отдельности.

Установка подключения к серверу

Подключение будет устанавливаться при нажатии пользователем кнопки «Подключить». Данные для подключения будут браться из полей адреса и порта сервера.

Листинг 12. Установка подключения к серверу

```
private void Start_Click(object sender, RoutedEventArgs e)
{
    // создаем новый сокет для подключения к серверу
    clientSocket = new TcpClient();
    // получаем адрес и порт
    string address = tbAddress.Text;
    int port = int.Parse(tbPort.Text);
    // пытаемся подключиться
    clientSocket.Connect(address, port);
    // все получилось
    lblStatus.Content = "Подключен";
    // создаем новый поток для получения сообщений от сервера
    Thread thread = new Thread(() => receiveMessages(clientSocket));
    thread.Start();
}
```

Этот код во многом аналогичен запуску самого сервера. Сперва создается объект сокета, затем программа пробует подключить этот сокет к серверу по указанному адресу и порту, и, в случае успеха, создается новый поток, в котором будет производиться прием данных пришедших от сервера.

Отправка сообщений на сервер

Для передачи данных на сервер, необходимо получить поток ввода-вывода сокета и передать в него требуемые данные. Данные для передачи через сеть должны быть представлены в виде массива байтов. В нашем случае мы будем передавать текстовое сообщение, чтобы преобразовать его в массив байтов, это сообщение необходимо закодировать, используя одну из доступных кодировок. Как следует из лекции 3, оптимальным будет использовать кодировку UTF-8

Листинг 13. Отправка данных на сервер

```
private void pbSend_Click(object sender, RoutedEventArgs e)
{
    // получаем поток ввода-вывода связанный с сокетом
    NetworkStream serverStream = clientSocket.GetStream();
    // преобразуем введенный текст в массив байтов
    byte[] outStream = System.Text.Encoding.UTF8.GetBytes(tbMessage.Text);
    // передаем этот массив в сокет
    serverStream.Write(outStream, 0, outStream.Length);
    serverStream.Flush();
}
```

Прием сообщений от сервера

Прием сообщений от сервера полностью аналогичен приему сообщений от клиентов сервером, за исключением того, что клиент не пересылает полученные сообщения.

Листинг 14. Прием сообщений от сервера

```
private void receiveMessages(TcpClient socket)
{
    byte[] bytesFrom = new byte[10025];
    string dataFromServer = null;
    while (true)
    {
        if (!socket.Connected)
        {
            lbLog.Dispatcher.BeginInvoke(new Action(() => addMessage("Отключен")));
            return;
        }
        NetworkStream networkStream = socket.GetStream();
        try
        {
            int read = networkStream.Read(bytesFrom, 0, 10025);
            if (read > 0)
            {
                dataFromServer = Encoding.UTF8.GetString(bytesFrom, 0, read);
                lbLog.Dispatcher.BeginInvoke(new Action(() => addMessage(dataFromServer)));
            }
            else
            {
                Thread.Sleep(100);
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine(" >> " + ex.ToString());
        }
    }
}
```

Разработка сетевой пошаговой игры

В этом примере разработаем игру в крестики-нолики, в которую можно будет играть по сети. Как и в предыдущем примере, разобьем задачу на две части: серверную и клиентскую. Для того, чтобы предотвратить возможность читинга, вся игровая логика будет находиться на стороне сервера, а на клиенте будет лишь отображение текущего состояния игры и обработка ввода пользователя.

Алгоритм работы сервера будет следующим:

Рис. 9. Схема алгоритма работы сервера

Создание игрового сервера

Код сервера во многом повторяет предыдущие примеры и так же состоит из трех основных частей:

- Создание сокета и ожидание клиента
- Прием данных от клиента
- Отправка данных клиенту

Однако принципиальным отличием в данной программе является то, что для работы необходимо подключение ровно двух клиентов-игроков. Так же нам понадобится серверный сокет и массив данных, в котором будет храниться состояние игрового поля.

Для повышения читаемости кода опишем несколько перечисляемых типов, которые будут кодировать команды, пересылаемые между сервером и клиентами, и состояния игрового поля:

```
// Перечисление описывающее набор команд
// пересылаемых между сервером и клиентами
13 references | 0 changes | 0 authors, 0 changes
enum CMD
{
    NewGame, // Начало новой игры
    Waiting, // Ожидание второго игрока
    Move,    // Ход игрока
    EndGame  // Игра завершена
}

// Перечисление описывающее состояние
// ячейки поля
16 references | 0 changes | 0 authors, 0 changes
enum Type
{
    None,   // Пустая ячейка
    Zero,   // Нолик
    Cross   // Крестик
}
```

Листинг 15. Вспомогательные перечисляемые типы

После этого объявим глобальные переменные для хранения данных о клиентах и игровом поле:

```
TcpListener server;
TcpClient player1;
TcpClient player2;

// ссылка на игрока, который будет ходить
// следующим
TcpClient nextTurn;

// Состояние игрового поля
Type[] field = new Type[9];
```

Листинг 16. Глобальные переменные

Игровое поле представляет собой массив из 9 ячеек, которые представляют собой поле 3 x 3. В каждой ячейке этого поля будет храниться отметка, о том какой символ был поставлен в это поле.

0	1	2
3	4	5
6	7	8

Рис. 10. Нумерация ячеек в игровом поле

Интерфейс пользователя сервера

На форме Сервера разместим поле ввода порта, на котором будет запущен сервер, кнопку запуска и окно логов:

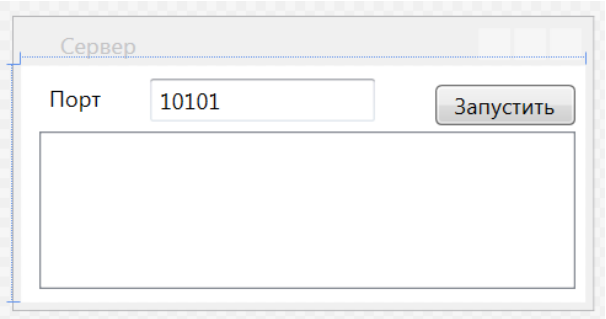

Рис. 11. Окно сервера

Запуск сервера

Запуск сервера полностью аналогичен предыдущим примерам

```
1 reference | 0 changes | 0 authors, 0 changes
private void btnStart_Click(object sender, RoutedEventArgs e)
{
    int port = int.Parse(tbPort.Text);
    server = new TcpListener(IPAddress.Any, port);
    Thread thread = new Thread(waitingClients);
    thread.Start();
}
```

Листинг 17. Запуск сервера

При запуске создается новый поток в котором будет происходить ожидание подключения клиентов.

Подключение клиентов

Обработчик подключения клиентов так же остается похож на предыдущие примеры. В бесконечном цикле мы ожидаем в методе AcceptTcpClient. И как только подключается новый клиент, мы передаем его в обработку.

```
1 reference | 0 changes | 0 authors, 0 changes
private void waitingClients()
{
    server.Start();
    serverWindow.Dispatcher.BeginInvoke(new Action(() => addMessage("Сервер запущен")));
    while (true)
    {
        var client = server.AcceptTcpClient();
        serverWindow.Dispatcher.BeginInvoke(new Action(() => addMessage("Клиент подключен")));
        acceptClient(client);
    }
}
```

Листинг 18. Поток ожидания подключения клиентов

Так как в этой программе мы должны дождаться подключения двух клиентов перед запуском основной игры, код обработки нового клиента немного усложнится. В нем нам нужно определить какой по счету игрок подключился в данный момент и в зависимости от этого перейти к ожиданию первого игрока или начать игру.

Для общения с клиентами мы будем использовать команды, которые были определены

в начале программы. Если к нам подключается первый игрок, то мы должны сообщить ему, что игра не может начаться прямо сейчас и нужно подождать. Для этого нужно отправить команду WAITING:

```
stream.Write(new byte[] { (byte)CMD.Waiting }, 0, 1);
```

Эта команда не имеет параметров, поэтому передаем только один байт.

При подключении второго игрока, мы уже готовы начать игру, поэтому отправляем каждому из игроков команду на начало новой игры. В этом сообщении мы передаем игрокам символ, которым они будут играть, крестик или нолик. То есть, команда имеет один параметр, соответственно передается 2 байта данных.

```
var stream = player1.GetStream();
stream.Write(new byte[] { (byte)CMD.NewGame,(byte) Type.Zero }, 0, 2);
stream = player2.GetStream();
stream.Write(new byte[] { (byte)CMD.NewGame, (byte)Type.Cross }, 0, 2);
```

В этом примере первый игрок всегда играет ноликами, а второй – крестиками.

Полный код метода:

```
private void acceptClient(TcpClient client)
{
    // Если не подключен ни один игрок, то этот клиент будет первым
    if (player1 == null)
        player1 = client;
    else //иначе - вторым
        player2 = client;

    // создаем поток для обмена данными с клиентом
    Thread thread = new Thread(() => listenClient(client));
    thread.Start();

    // если подключился первый игрок
    if(client == player1)
    {
        // то сообщаем ему, что нужно долждаться второго
        serverWindow.Dispatcher.BeginInvoke(new Action(() => addMessage("Waiting for second player")));
        NetworkStream stream = client.GetStream();
        stream.Write(new byte[] { (byte)CMD.Waiting }, 0, 1);
    }
    else
    {   // если подключился второй игрок, то начинаем новую игру
        serverWindow.Dispatcher.BeginInvoke(new Action(() => addMessage("Starting new game")));
        // Очищаем поле
        for (int i = 0; i < field.Length; i++)
            field[i] = Type.None;
        // Сообщаем игрокам о начале игры
        var stream = player1.GetStream();
        stream.Write(new byte[] { (byte)CMD.NewGame,(byte) Type.Zero }, 0, 2);
        stream = player2.GetStream();
        stream.Write(new byte[] { (byte)CMD.NewGame, (byte)Type.Cross }, 0, 2);
        // первым ходит игрок подключившийся первым
        nextTurn = player1;
    }
}
```

Листинг 19. Обработка подключения нового игрока

Прием данных от игрока

Обработчик входящих команд от игрока не отличается от предыдущих примеров. С интервалом 100 мс мы опрашиваем сокет клиента на наличие сообщений. При поступлении новых команд они передаются метод парсинга команд.

В этом методе мы проверяем

- Что за команду прислал игрок
- Чья очередь была ходить
- Каким символом походил игрок
- Куда походил игрок
- Мог ли игрок сделать такой ход

Если все условия выполнены, то мы обновляем состояние поля и уведомляем каждого из игроков о том, что состояние поля изменилось.

```
private void listenClient(TcpClient client)
{
    NetworkStream stream = client.GetStream();
    byte[] buf = new byte[1024];
    while (true)
    {
        int count = stream.Read(buf, 0, 1024);
        if (count > 0)
        {
            serverWindow.Dispatcher.BeginInvoke(new Action(() => parseClient(client, buf)));
        }
        else
        {
            Thread.Sleep(100);
        }
    }
}
```

Листинг 20. Прием данных от клиента

После того как ход был сделан мы должны уведомить игроков об этом. Затем выполняется проверка на завершение игры. Мы должны выполнить проверку с небольшой задержкой, для того, чтобы все игроки получили последнее обновление поля перед командой завершения игры.

```
private void parseClient(TcpClient client, byte[] response)
{
    // от клиента мы принимаем только одну команду - ход
    if ((CMD) response[0] == CMD.Move)
    {
        // если команда пришла не от того игрока
        // который должен ходить, то игнорируем
        if (client != nextTurn)
            return;

        // Следующим будет ходить другой игрок
        if (client == player1)
            nextTurn = player2;
        else
            nextTurn = player1;

        // Получаем номер поля в которе походил игрок
        int move = response[1];
        // Получаем тип символа - O или X
        Type type = (Type) response[2];

        // Здесь должна быть проверка на правильность хода!

        // Ставим символ в это поле
        field[move] = type;

        // Отправляем игрокам сообщение о том что был сделан ход
        var stream = player1.GetStream();
        stream.Write(new byte[] { (byte)CMD.Move, (byte) move, (byte)type }, 0 , 3);
        stream = player2.GetStream();
        stream.Write(new byte[] { (byte)CMD.Move, (byte)move, (byte)type }, 0, 3);

        // с небольшой задержкой проверяем не закончилась ли игра
        Task.Delay(101).ContinueWith(t => checkForWinner());
    }
}
```

Листинг 21. Обработка хода игрока

Проверка завершения игры

Для того, чтобы узнать, что игра завершена мы должны проверить каждую диагональ, строку и столбец – не заполнены ли они символами одного типа. Если же ни один игрок не составил три символа в ряд, а места на поле уже не осталось, то игра заканчивается ничьей.

```
private void checkForWinner()
{
    bool gameEnded = false;
    TcpClient winner = null;
    for(int i = 0; i < 3; i++)
    {   // Горизонтальные линии
        if (field[i * 3] != Type.None && (field[i * 3 + 0] & field[i * 3 + 1] & field[i * 3 + 2]) == field[i * 3])
        {
            gameEnded = true;
            winner = field[i * 3] == Type.Zero ? player1 : player2;
        }
        // Вертикальные линии
        if (field[i] != Type.None && (field[i + 0] & field[i + 3] & field[i + 6]) == field[i])
        {
            gameEnded = true;
            winner = field[i] == Type.Zero ? player1 : player2;
        }
    }   // Диагональ 1
    if(field[0] != Type.None && (field[0] & field[4] & field[8]) == field[0])
    {
        gameEnded = true;
        winner = field[0] == Type.Zero ? player1 : player2;
    }   // Диагональ 2
    if (field[2] != Type.None && (field[2] & field[4] & field[6]) == field[2])
    {
        gameEnded = true;
        winner = field[2] == Type.Zero ? player1 : player2;
    }
    // Осталось ли место на поле
    if (!field.Any(f => f == Type.None))
    {
        gameEnded = true;
    }
    if (gameEnded)
    {
        gameOver(winner);
    }
}
```

Листинг 22. Проверка завершения игры

Если одно из условий было выполнено, вызываем метод gameOver.

Завершение игры

В методе завершения игры мы должны сообщить каждому игроку результат. Это может быть один из трех вариантов:

- Победа первого игрока
- Победа второго игрока
- Ничья

Для этого мы отправляем команду EndGame с кодом

- 0 – если игрок проиграл
- 1 – если игрок победил
- 2 – если случилась ничья

После отправки этого сообщения завершаем игру и закрываем подключения к игрокам.

Листинг 23. Метод завершения игры

```
private void gameOver(TcpClient winner)
{
    if (winner == player1)
    {
        var stream = player1.GetStream();
        stream.Write(new byte[] { (byte)CMD.EndGame, 1 }, 0, 2);
        stream = player2.GetStream();
        stream.Write(new byte[] { (byte)CMD.EndGame, 0 }, 0, 2);
    }
    else if (winner == player2)
    {
        var stream = player1.GetStream();
        stream.Write(new byte[] { (byte)CMD.EndGame, 0 }, 0, 2);
        stream = player2.GetStream();
        stream.Write(new byte[] { (byte)CMD.EndGame, 1 }, 0, 2);
    }
    else
    {   // если ничья
        var stream = player1.GetStream();
        stream.Write(new byte[] { (byte)CMD.EndGame, 2 }, 0, 2);
        stream = player2.GetStream();
        stream.Write(new byte[] { (byte)CMD.EndGame, 2 }, 0, 2);
    }
    // Закрываем подключение
    player1.Close();
    player2.Close();
    player1 = null;
    player2 = null;
}
```

После завершения игры сервер готов к подключению новых игроков и проведению нового матча.

Создание игрового клиента

Приложение игрового клиента должно подключаться к нашему серверу и поддерживать его протокол сетевого обмена. Так как все проверки выполняются на стороне сервера, то единственной командой, которую сможет отправлять клиент является команда запроса на выполнение хода.

Начнем, как обычно, с интерфейса.

Интерфейс пользователя

Интерфейс будет состоять из двух основных частей:

- Панели с параметрами подключения к серверу
- Игрового поля, на котором будет происходить игра

Для создания игрового поля, разместим на форме 9 кнопок внутри контейнера Grid в

виде таблицы 3 х 3 кнопки. Должно получиться похоже на рисунок ниже:

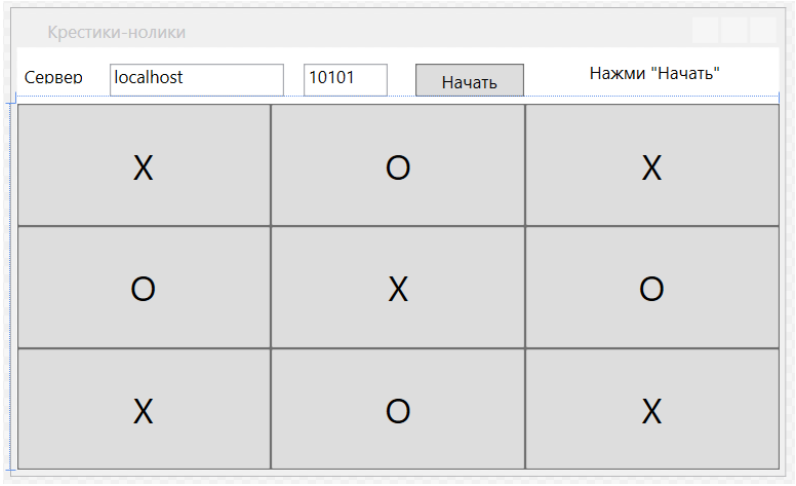

Рис. 12. Интерфейс игрового клиента

Игрок будет осуществлять ход путем нажатия на одну из кнопок, а при выполнении хода второго игрока на соответствующей кнопке будет отмечаться его ход.

Подключение к серверу

Подключение к серверу осуществляется по уже знакомой схеме. В случае успешного подключения создается новый поток в котором будет осуществляться прием данных от сервера. В случае неуспеха – сообщаем об этом пользователю.

Листинг 24. Метод подключения к серверу

```
private void connect(object sender, RoutedEventArgs e)
{
    try
    {
        string server = tbServer.Text;
        int port = int.Parse(tbPort.Text);
        socket = new TcpClient(server, port);

        Thread thread = new Thread(receiveServer);
        thread.Start();
    }
    catch
    {
        lblStatus.Content = "Сервер недоступен";
    }
}
```

Прием данных от сервера

После установки соединения с сервером, клиент ожидает от него одной из двух

команд:

- Если это первый подключившийся клиент, то сервер отправит ему сообщение «Ожидаем второго игрока»
- Если это второй подключившийся клиент, то сервер отправит ему сообщение «Начинаем игру»

Поток чтения данных от сервера аналогичен всем другим примерам: в бесконечном цикле с некоторым периодом производим прием данных от сервера. Если какие-то данные были получены, то передаем их в основной поток выполнения для дальнейшего анализа.

Листинг 25. Прием данных от сервера

```
private void receiveServer()
{
    byte[] buf = new byte[1024];
    var stream = socket.GetStream();
    while(true)
    {
        try
        {
            int count = stream.Read(buf, 0, 1024);
            if (count > 0)
            {
                gameWindow.Dispatcher.BeginInvoke(new Action(() => parseMessage(buf)));
            }
            else
            {
                Thread.Sleep(100);
            }
        }
        catch
        {
            break;
        }
    }
}
```

Разбор команд от сервера

При приходе команд от сервера, клиент должен их соответствующим образом обработать. Всего сервер может прислать 4 команды:

- CMD.Waiting - Ожидаем второго игрока
- CMD.NewGame - Начинаем игру
- CMD.Move - Был произведен ход
- CMD.EndGame - Игра закончена

Команда Waiting не имеет параметров и служит только для уведомления пользователя о том, что игра не может быть начата прямо сейчас. При поступлении этой команды будем выводить соответствующее сообщение в строку статуса.

Команда NewGame имеет один параметр – символ которым будет играть текущий игрок. Получив эту команду будет нужно подготовить игровое поле и проинициализировать некоторые переменные.

Команда Move имеет два параметра. Первый параметр – это поле в которое был произведен ход. Второй параметр – это тип символа, который сейчас был поставлен на поле: нолик или крестик. Так как сервер осуществляет проверку каждого хода, то даже собственный ход клиент будет отмечать на игровом поле только по команде от сервера.

Команда EndGame имеет один параметр – код завершения игры. Для клиента игра может завершиться с тремя возможными результатами:

- Победа – код 1
- Поражение – код 0
- Ничья – код 2.

В зависимости от того какой код будет получен, мы должны вывести соответствующее сообщение в поле статуса.

Листинг 26. Анализ команд сервера

```
private void parseMessage(byte[] buf)
{
    switch ((CMD)buf[0])
    {
        case CMD.NewGame:
            lblStatus.Content = "Начинаем игру!";
            newGame((Type) buf[1]);
            break;
        case CMD.Waiting:
            lblStatus.Content = "Ожидаем второго игрока";
            break;
        case CMD.Move:
            int position = buf[1];
            Type type = (Type) buf[2];
            move(position, type);
            break;
        case CMD.EndGame:
            endGame(buf[1]);
            break;
    }
}
```

Начало игры

При начале игры необходимо очистить каждую ячейку игрового поля, и проинициализировать переменную хранящую информацию о том, каким символом будет играть пользователь.

Листинг 27. Метод начала игры

```
private void newGame(Type type)
{
    myType = type;
    foreach(var b in Container.Children.Cast<Button>())
    {
        b.Content = "";
        b.Background = Brushes.White;
        b.Foreground = Brushes.Blue;
    }
}
```

Обработка хода пользователя

Для того, чтобы пользователь мог «походить», нужно обработать клик по одной из кнопок игрового поля. Затем нужно вычислить номер этого поля и отправить на сервер запрос на выполнение хода. При этом мы не будем отображать на игровом поле никаких изменений, так как сервер может не принять данный ход. Все отметки на поле должны производиться только по команде от сервера.

После вычисления всех необходимых параметров, отправляем команду на сервер в соответствии с протоколом:

```
stream.Write(new byte[] { (byte)CMD.Move,
                          (byte)index,
                          (byte)myType }, 0, 3);
```

0	1	2
3	4	5
6	7	8

Рис. 13. Нумерация ячеек в игровом поле

Здесь index – это номер ячейки игрового поля, а myType – это тип символа, который должен быть поставлен в эту ячейку.

Листинг 28. Отправка серверу сообщения о выполненном ходе

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    try
    {
        var button = (Button)sender;
        var column = Grid.GetColumn(button);
        var row = Grid.GetRow(button);

        var index = column + (row * 3);

        var stream = socket.GetStream();
        stream.Write(new byte[] { (byte)CMD.Move,
                                  (byte)index,
                                  (byte)myType }, 0, 3);

    }
    catch
    {
        lblStatus.Content = "Сервер недоступен";
    }

}
```

Сообщение о выполненном ходе

Как только сервер проверил правильность выполненного хода игрока, он пересылает каждому игроку сообщение о необходимости поставить отметку на игровом поле. Все кнопки пронумерованы как ячейки игрового поля, поэтому мы можем просто взять присланный сервером номер ячейки и с его помощью обратиться к кнопке нашего игрового поля.

Листинг 29. Обработка сообщения о произведенном ходе

```
private void move(int position, Type type)
{
    var button = Container.Children.Cast<Button>().ToList()[position];
    if (type == Type.Cross)
    {
        button.Content = "X";
    }
    else
    {
        button.Content = "O";
    }
}
```

Окончание игры

Когда один из игроков выстроил свои символы в ряд, или на поле закончились свободные ячейки сервер присылает сообщение об окончании игры.

Единственным параметром этой команды является код результата завершения. Он может быть одним из трех:

- Игрок проиграл – 0
- Игрок победил – 1
- Ничья – 2

В соответствии с этим кодом выводим сообщение в строку статуса.

Листинг 30. Окончание игры

```
private void endGame(byte result)
{
    if (result == 0)
        lblStatus.Content = "Вы проиграли!";
    else if (result == 1)
        lblStatus.Content = "Вы победили!";
    else
        lblStatus.Content = "Ничья!";
    socket.Close();
}
```

Задания

Задание для приложения «UDP чат»

- Сделать возможным обмен сообщениями между тремя (или более) клиентами

Задания для приложения «TCP чат»

- Добавить цензурирование (или автозамену) некоторых слов по словарю на стороне сервера
- Добавить возможность установки имени пользователя при подключении и отображение этого имени в логе чата

Задания для приложения «Крестики-Нолики»

- Добавить проверку на правильность хода на стороне сервера
- Добавить подсветку победной комбинации у игроков по окончанию игры
- Реализовать игру на бесконечном поле