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.

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 приема данных Receive(). 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, иначе программа не сможет закрыться!

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:

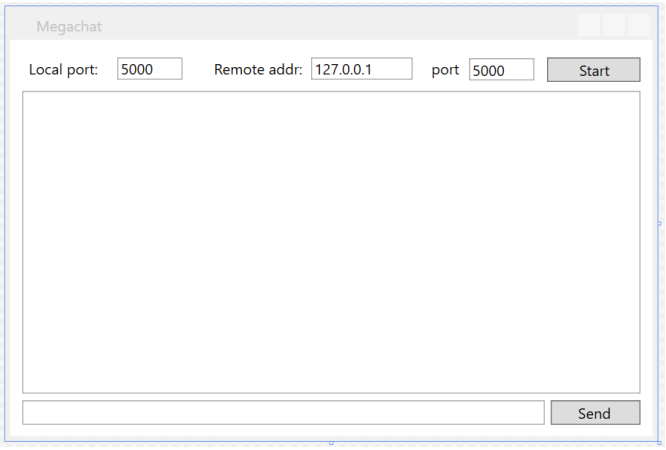

Рис. 1. Внешний вид программы в редакторе форм

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.

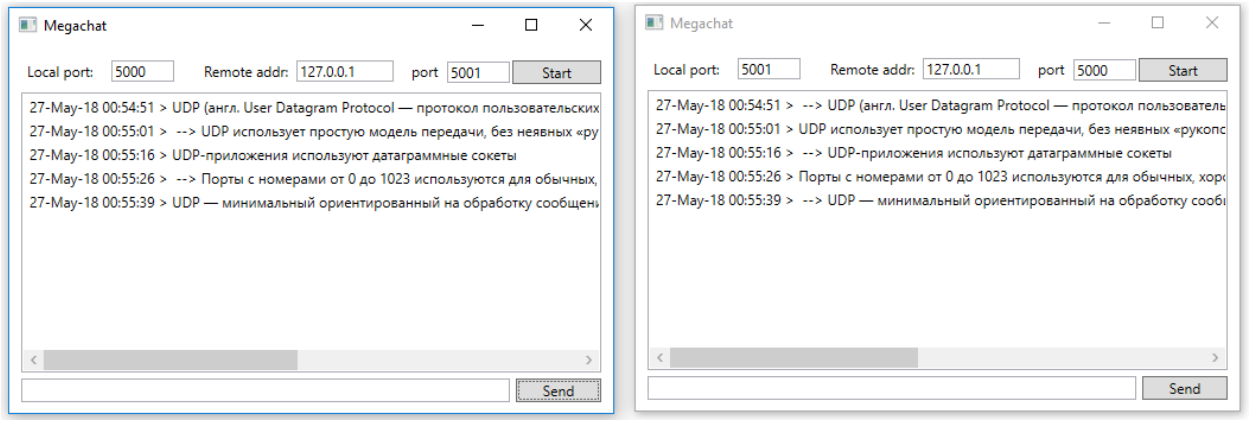

Рис. 2. Пример работы программы

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.

Задания

Вариант 1.

Разработать программу «Чат». Сделать возможным обмен сообщениями между тремя (или более) клиентами.

Вариант 2.

Разработать сервер позволяющий узнать точное время и дату. Для получения текущего времени используйте запрос DateTime.Now.

После запуска сервер должен ожидать подключение на порту 2023.

- При подключении клиент передает
- По запросу «date» - должна быть возвращена строка с датой в формате «ДД.ММ.ГГГГ».
- По запросу «time» - должна быть возвращена строка со временем в формате «ЧЧ:ММ:СС.ЗЗЗ», где ЗЗЗ – миллисекунды.
- По запросу «datetime» - должна быть возвращена строка с датой и временем в формате «ДД.ММ.ГГГГ ЧЧ:ММ:СС.ЗЗЗ»
- По любому другому запросу возвращается ответ «Specified command is not supported»

Вариант 3.

Разработать сервер, возвращающий вступительный текст эпизодов Звездных войн 1-6.

После запуска сервер должен ожидать подключение на порту 3200 для выдачи данных на русском языке, и на порту 3201 для выдачи данных на английском языке

- При подключении клиент отправляет номер эпизода саги 1, 2, 3, 4, 5 или 6.
- Ответ от сервера должен возвращаться построчно с паузами между строками 1 с.

Вариант 4.

Разработать сервер открывающий в браузере указанную клиентом страницу.

После запуска сервер должен ожидать подключение на порту 8081.

- При подключении клиент отправляет адрес страницы в интернет
- При получении адреса сервер должен открыть браузер с этим адресом. Для этого можно воспользоваться вызовом Diagnostics.Process.Start(), однако в этом случае нужно убедиться, что сервер не будет запускать ничего кроме браузера.