ЛАБОРАТОРНАЯ РАБОТА №5

Телекоммуникационные технологии.

СЕТЕВОЕ ПРОГРАММИРОВАНИЕ

Существует несколько различных способов передачи данных между программами:

- Обмен сообщениями
- Объекты синхронизации
- Механизмы разделения памяти
- Удалённые вызовы

В этой лабораторной работе мы рассмотрим механизм передачи данных под названием **сокеты**.

Сокет — это один конец двустороннего канала связи между двумя программами, работающими как на локальном компьютере, так и в сети. Соединяя вместе два **сокета**, можно передавать данные между разными процессами. Реализация **сокетов** осуществляется посредством протоколов сетевого и транспортного уровней.

Сокеты работают по алгоритму open/read/write/close. Прежде чем использовать **сокет**, его нужно открыть, задав соответствующие параметры. После открытия из него можно считывать или записывать в него данные. После использования **сокета** пользователь должен вызывать метод **Close**, чтобы завершить работу с этим ресурсом.

Существуют два основных типа **сокетов** — **потоковые** и **дейтаграммные**.

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, сдублированных 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.

Дейтаграммные сокеты 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**.

Использование дейтаграммных сокетов требует, чтобы передачей данных от клиента к серверу занимался **User Datagram Protocol** (**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.

Если данные должны гарантированно доставляться другой стороне или размер их велик, **потоковые сокеты** предпочтительнее дейтаграммных. Если надежность связи между двумя приложениями не имеет особого значения, выбирайте **дейтаграммные сокеты**.

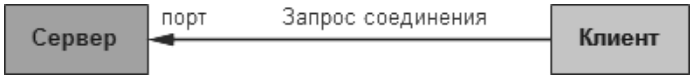
Сокет состоит из **IP-адреса** машины и номера **порта**, используемого приложением TCP.

Порт определен, чтобы разрешить задачу одновременного взаимодействия с несколькими приложениями. Компьютер, на котором одновременно выполняется несколько приложений, получая пакет из сети, может идентифицировать целевую программу, пользуясь уникальным номером порта, определенным при установлении соединения.

АРХИТЕКТУРА КЛИЕНТ-СЕРВЕР

Обычно приложение, использующее сокеты, состоит из двух разных приложений - **клиента**, инициирующего соединение с целью, и **сервера**, ожидающего соединения от клиента.

Например, на стороне клиента, приложение должно знать адрес цели и номер порта. Отправляя запрос на соединение, клиент пытается установить связь с сервером:



Если события развиваются удачно, при условии что сервер запущен прежде, чем клиент попытался с ним соединиться, сервер соглашается на соединение. Дав согласие, серверное приложение создает новый сокет для взаимодействия именно с установившим соединение клиентом:

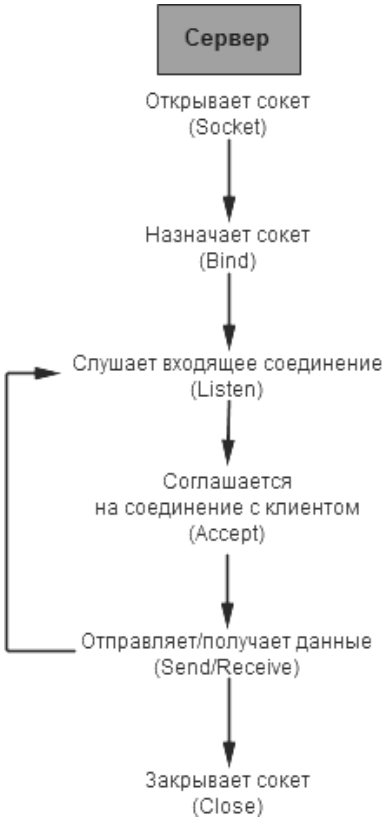
Теперь клиент и сервер могут взаимодействовать между собой, считывая сообщения каждый из своего сокета и, соответственно, записывая сообщения.

ПРОСТЕЙШЕЕ ПРИЛОЖЕНИЕ С ИСПОЛЬЗОВАНИЕМ СОКЕТОВ

Напишем приложение, состоящее из двух отдельных программ - **клиента** и **сервера**.

Сервер будет получать и выводить запросы от клиентов. Мы используем простейшую **синхронную** реализацию, поэтому выполнение программы будет **блокироваться**, пока сервер не даст согласия на соединение с клиентом. Клиент завершает соединение, отправляя серверу сообщение: «**off**».

Общий алгоритм работы сервера показан на следующем рисунке:



Откройте **Visual Studio** и создайте новое консольное приложение C#. Назовите его **Server**.

Для удобства пропишите в начале файла следующие пространства имен:

```
using System.Net;
using System.Net.Sockets;
using static System.Console;
```

Первый шаг заключается в установлении для **сокета** локальной конечной точки. Прежде чем открывать сокет для ожидания соединений, нужно подготовить для него **адрес локальной конечной точки**. Уникальный адрес для обслуживания TCP/IP определяется комбинацией **IP-адреса** хоста с номером **порта** обслуживания, которая создает конечную точку для обслуживания.

Класс **Dns** предоставляет методы, возвращающие информацию о сетевых адресах, поддерживаемых устройством в локальной сети. Если у устройства локальной сети имеется более одного сетевого адреса, класс **Dns** возвращает информацию обо всех сетевых адресах, и приложение должно выбрать из массива подходящий адрес для обслуживания.

Создадим **IPEndPoint** для сервера, комбинируя первый IP-адрес хост-компьютера, полученный от метода **Dns.Resolve**(), с номером порта:

```
var host = Dns.GetHostEntry("localhost");
var addr = host.AddressList[0];
var endPoint = new IPEndPoint(addr, 11000);
```

Здесь класс **IPEndPoint** представляет **localhost** на **порте** 11000. Далее новым экземпляром класса **Socket** создаем потоковый сокет. Установив локальную конечную точку для ожидания соединений, можно создать сокет:

```
var socket = new Socket(addr.AddressFamily, SocketType.Stream, ProtocolType.Tcp);
```

Перечисление **AddressFamily** указывает схемы адресации, которые экземпляр класса **Socket** может использовать для разрешения адреса.

В параметре **SocketType** различаются сокеты **TCP** и **UDP**. В нем можно определить в том числе следующие значения:

- **Dgram** – Поддерживает дейтаграммы. Значение **Dgram** требует указать **Udp** для типа протокола и **InterNetwork** в параметре семейства адресов.

- **Raw** – Поддерживает доступ к базовому транспортному протоколу.
- **Stream** – Поддерживает **потоковые** сокеты. Значение **Stream** 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**(). Чтобы сокет клиента мог идентифицировать потоковый сокет **TCP**, серверная программа должна дать **имя** своему сокету:

```
socket.Bind(endPoint);
```

В параметре определяется задел (backlog), указывающий максимальное число соединений, ожидающих обработки в очереди. В приведенном коде значение параметра допускает накопление в очереди до десяти соединений.

```
socket.Listen(10);
```

В состоянии прослушивания надо быть готовым 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. Никакое серверное приложение не должно закрывать слушающий сокет. Он должен продолжать работать наряду с сокетами, созданными методом **Accept** для обработки входящих запросов клиентов.

```
while (true) {
    WriteLine("Ожидаем соединение через порт {0}", endPoint);

    var handler = socket.Accept();
```

Как только клиент и сервер установили между собой соединение, можно отправлять и получать сообщения, используя методы Send() и Receive() класса Socket.

```
string data = null;

var bytes = new byte[1024];
var size = handler.Receive(bytes);

data += Encoding.UTF8.GetString(bytes, 0, size);

Write("Полученный текст: " + data + "\n\n");

string reply = "Сервер принял " + data.Length + " символов";
var msg = Encoding.UTF8.GetBytes(reply);

handler.Send(msg);

if (data.IndexOf("off") > -1) {
    WriteLine("Соединение закрыто");
    break;
}
```

Метод **Send**() записывает исходящие данные сокету, с которым установлено соединение. Метод **Receive**() считывает входящие данные в потоковый сокет. При использовании системы, основанной на **TCP**, перед выполнением методов **Send**() и **Receive** () между сокетами должно быть установлено соединение. Точный протокол между двумя взаимодействующими сущностями должен быть определен заблаговременно, чтобы клиентское и серверное приложения не блокировали друг друга, не зная, кто должен отправить свои данные первым.

Данные между приложениями передаются в виде потока байт, поэтому перед отправкой строку с текстом необходимо преобразовать из формата **Unicode-16** в однобайтовый формат, например **UTF-8**. Для этого используется функция **Encoding.UTF8.GetString**.

Когда обмен данными между сервером и клиентом завершается, нужно закрыть соединение используя методы **Shutdown**() и **Close**():

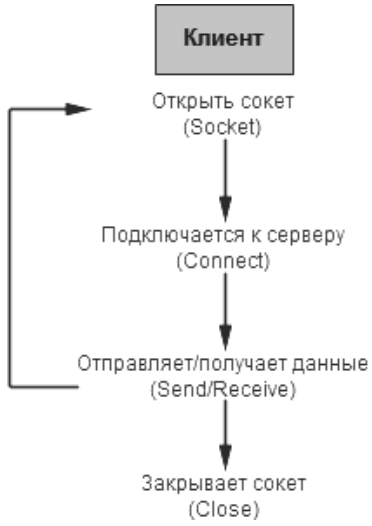
```
            handler.Shutdown(SocketShutdown.Both);
            handler.Close();
        } // while
```

SocketShutdown — это перечисление, содержащее три значения: **Both** - останавливает отправку и получение данных сокетом, **Receive** - останавливает получение данных сокетом и **Send** - останавливает отправку данных сокетом.

Сокет закрывается при вызове метода **Close**(), который также устанавливает в свойстве **Connected** сокета значение **false**.

После создания серверного приложения настало время написать программу-**клиент**. Откройте еще один экземпляр **Visual Studio** и создайте новое консольное приложение. Назовите его **Client**.

Функции, которые используются для создания приложения-клиента, более или менее напоминают серверное приложение. Как и для сервера, используются те же методы для определения конечной точки, создания экземпляра сокета, отправки и получения данных и закрытия сокета:



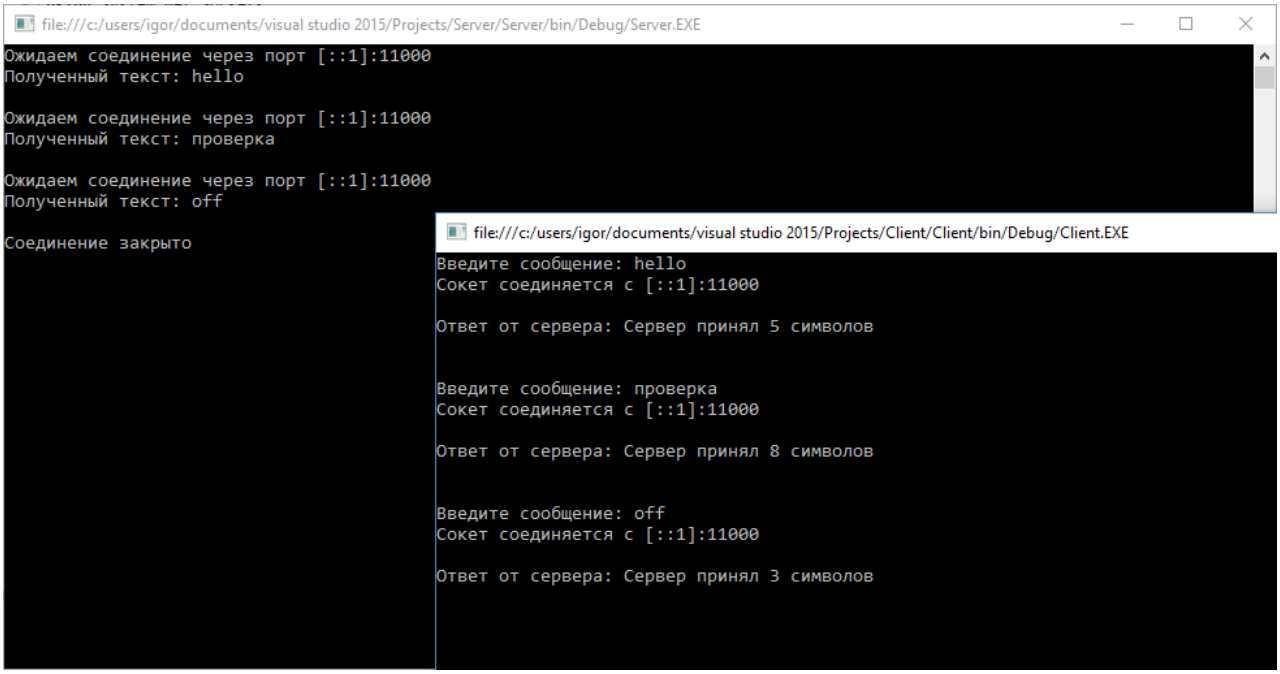
Единственный новый метод - метод **Connect**(), используется для соединения с удаленным сервером.

Код клиента выглядит следующим образом:

```
static void Main(string[] args) {
    Send(11000);
}

static void Send(int port) {
    var bytes = new byte[1024];

    var host = Dns.GetHostEntry("localhost");
    var addr = host.AddressList[0];
    var endPoint = new IPEndPoint(addr, port);

    var socket = new Socket(addr.AddressFamily, SocketType.Stream, ProtocolType.Tcp);

    socket.Connect(endPoint);

    Write("Введите сообщение: ");
```

```
            var message = ReadLine();

            WriteLine("Сокет соединяется с {0} ", socket.RemoteEndPoint.ToString());
            var textBytes = Encoding.UTF8.GetBytes(message);

            int bytesSent = socket.Send(textBytes);

            int bytesRec = socket.Receive(bytes);

            WriteLine("\nОтвет от сервера: {0}\n\n", Encoding.UTF8.GetString(bytes, 0, bytesRec));

            if (message.IndexOf("off") == -1) {
                Send(port);
            }

            socket.Shutdown(SocketShutdown.Both);
            socket.Close();
        }
```

Теперь необходимо запустить программу-сервер, а затем запустить клиент. Результат работы представлен на рисунке:



ДОПОЛНИТЕЛЬНОЕ ЗАДАНИЕ

Попробуйте изменить код клиента так, чтобы он устанавливал соединение с сервером, расположенным на другом компьютере в локальной сети.