Лабораторная работа №1. Сокеты

Цель работы. Приобрести практические навыки разработки простейших сетевых приложений типа клиент-сервер, с применением протоколов TCP и UDP используя сокеты.

Задачи:

1. Изучить понятие сокетов.
2. Изучить механизм межсетевого взаимодействия на основе сокетов.
3. Рассмотреть классы в пространстве имен System.Net.Sockets обеспечивающих поддержку сокетов в .NET
4. Изучить пример клиент-серверного приложения на сокетах TCP.
5. Изучить пример сетевого приложения на сокетах UDP.
6. Разработать сетевое приложение типа клиент-сервер с использование протокола TCP.
7. Разработать сетевое приложение, используя протокол UDP.

Краткие теоретические сведения

Протокол UDP

Протокол UDP (User Datagram Protocol – протокол пользовательских дейтаграмм) используется реже, чем протокол TCP, но он проще для понимания, поэтому рассмотрим его первым. Коротко UDP можно описать как ненадежный протокол без соединения, основанный на дейтаграммах. Теперь рассмотрим каждую из этих характеристик подробнее.

UDP не имеет никаких дополнительных средств управления пакетами по сравнению с IP. Это значит, что пакеты, отправленные с помощью UDP, могут теряться, дублироваться и менять порядок следования. В сети без роутеров ничего этого с пакетами почти никогда не происходит, и UDP может условно считаться надежным протоколом. Сети с роутерами строятся, конечно же, таким образом, чтобы подобные случаи происходили как можно реже, но полностью исключить их, тем не менее, нельзя. Происходит это из-за того, что передача данных может идти несколькими путями через разные роутеры. Например, пакет может пропасть, если короткий путь к удаленному узлу временно недоступен, а в длинном приходится пройти больше роутеров, чем это разрешено. Дублироваться пакеты могут, если они ошибочно передаются двумя путями, а порядок следования может изменяться, если пакет, посланный первым, идет по более длинному пути, чем пакет, посланный вторым.

Все вышесказанное отнюдь не означает, что на основе UDP нельзя построить надежный обмен данными, просто заботу об этом должно взять на себя само приложение. Каждый исходящий пакет должен содержать порядковый номер, и в ответ на него должен приходить специальный пакет - квитанция, которая уведомляет отправителя, что пакет доставлен. При

отсутствии квитанции пакет высылается повторно (для этого необходимо ввести таймауты на получение квитанции). Принимающая сторона по номерам пакетов восстанавливает их исходный порядок.

UDP не поддерживает соединение. Это означает, что при использовании этого протокола можно в любой момент отправить данные по любому адресу без необходимости каких-либо предварительных действий, направленных на установление связи с адресатом. Это напоминает процесс отправки обычного письма: на нем пишется адрес, и оно опускается в почтовый ящик без каких-либо предварительных действий. Такой подход обеспечивает большую гибкость, но лишает систему возможности автоматической проверки исправности канала связи.

Дейтаграммами называются пакеты, которые передаются как единое целое. Каждый пакет, отправленный с помощью UDP, составляет одну дейтаграмму. Полученные дейтаграммы складываются в буфер принимающего сокета и могут быть получены только раздельно: за одну операцию чтения из буфера программа, использующая сокет, может получить только одну дейтаграмму. Если в буфере лежит несколько дейтаграмм, потребуется несколько операций чтения, чтобы прочитать все. Кроме того, одну дейтаграмму нельзя получить из буфера по частям: она должна быть прочитана целиком за одну операцию.

Чтобы данные, передаваемые разным сокетам, не перемешивались, каждый сокет должен получить уникальный в пределах узла номер от 0 до 65535, называемый номером порта. При отправке дейтаграммы отправитель указывает IP-адрес и порт получателя, и система принимающей стороны находит сокет, привязанный к указанному порту, и помещает данные в его буфер. По сути дела, UDP является очень простой надстройкой над IP, все функции которой заключаются в том, что физический поток разделяется на несколько логических с помощью портов, и добавляется проверка целостности данных с помощью контрольной суммы (сам по себе протокол IP не гарантирует отсутствия искажений данных при передаче).

Максимальный размер одной дейтаграммы IP равен 65535 байтам. Из них не менее 20 байт занимает заголовок IP. Заголовок UDP имеет размер 8 байт. Таким образом, максимальный размер одной дейтаграммы UDP составляет 65507 байт.

Типичная область применения UDP – программы, для которых потеря пакетов некритична. Достоинствами UDP являются простота установления связи, возможность использования одного сокета для обмена данными с несколькими адресами и отсутствие необходимости возобновлять соединение после разрыва связи. В некоторых задачах также очень удобно то, что дейтаграммы не смешиваются, и получатель всегда знает, какие данные были отправлены одной дейтаграммой, а какие – разными.

Еще одним достоинством UDP является возможность отправки широковещательных дейтаграмм. Для этого нужно указать широковещательный IP-адрес, и такую дейтаграмму получат все сокеты в локальной сети, привязанные к заданному порту. Эту возможность нередко используют программы, которые заранее не знают, с какими компьютерами они должны связываться. Они посылают широковещательное сообщение и связываются со всеми узлами, которые распознали это сообщение и прислали на него соответствующий ответ. По умолчанию для широковещательных пакетов число роутеров, через которые они могут пройти, устанавливается равным нулю, поэтому такие пакеты не выходят за пределы подсети.

Протокол TCP

Протокол TCP (Transmission Control Protocol – протокол управления передачей) является надёжным потоковым протоколом с соединением, т.е. полной противоположностью UDP. Единственное, что у этих протоколов общее – это способ адресации: в TCP каждому сокету также назначается уникальный номер порта. Уникальность номера порта требуется только в пределах протокола: два сокета могут использовать одинаковые номера портов, если один из них работает через TCP, а другой – через UDP.

Для отправки пакета с помощью TCP отправителю необходимо сначала установить соединение с получателем. После выполнения этого действия соединенные таким образом сокеты могут использоваться только для отправки сообщений друг другу. Если соединение разрывается (самой программой или из-за проблем в сети), эти сокеты уже не могут быть использованы для установления нового соединения: они должны быть уничтожены, а вместо них созданы новые сокеты.

Механизм соединения, принятый в TCP, подразумевает разделение ролей соединяемых сторон: одна из них пассивно ждёт, когда кто-то установит с ней соединение, и называется сервером, другая самостоятельно устанавливает соединение и называется клиентом. Действия клиента по установлению связи заключаются в следующем: создать сокет, привязать его к адресу и порту, вызвать функцию для установления соединения, передав ей адрес сервера. Если все эти операции выполнены успешно, то связь установлена, и можно начинать обмен данными. Действия сервера выглядят следующим образом: создать сокет, привязать его к адресу и порту, перевести в режим ожидания соединения и дождаться соединения. При соединении система создаст на стороне сервера специальный сокет, который будет связан с соединившимся клиентом, и обмениваться данными с подключившимся клиентом сервер будет через этот новый сокет. Старый сокет останется в режиме ожидания соединения, и другой клиент сможет к нему подключиться. Для каждого нового подключения будет создаваться новый сокет, обслуживающий только данное соединение, а исходный будет по-прежнему ожидать соединения. Это позволяет нескольким клиентам одновременно соединяться с одним сервером, а серверу – не путаться в своих клиентах.

Установление такого соединения позволяет осуществлять дополнительный контроль прохождения пакетов. В рамках протокола TCP выполняется проверка доставки пакета, соблюдения очерёдности и отсутствия дублей. Механизмы обеспечения надежности достаточно сложны, и мы их здесь рассматривать не будем. Программисту для начала достаточно знать, что данные, переданные с помощью протокола TCP, не теряются, не дублируются и доставляются в том порядке, в каком были отправлены. В противном случае отправитель получает сообщение об ошибке. Соединенные сокеты время от времени обмениваются между собой специальными пакетами, чтобы проверить наличие соединения.

Протокол TCP называется потоковым потому, что он собирает входящие пакеты в один поток. В частности, если в буфере сокета лежат 30 байт, принятые по сети, не существует возможности определить, были ли эти 30 байт отправлены одним пакетом, 30-ю пакетами по 1 байту или ещё как-либо. Гарантируется только то, что порядок байт в буфере совпадает с тем порядком, в котором они были отправлены. Принимающая сторона также не ограничена в том, как она будет читать информацию из буфера: все сразу или по частям. Это существенно отличает TCP от UDP, в котором дейтаграммы не объединяются и не разбиваются на части.

Общие сведения о сокетах

Сокет (англ. *Socket* – разъём) – название программного интерфейса для обеспечения обмена данными между процессами (локальными или удаленными). Сокет – абстрактный объект, представляющий конечную точку соединения. Другими словами сокет – это комбинация IP-адреса и порта.

Первоначально сокеты были разработаны для UNIX в Калифорнийском университете в Беркли. Сокеты Беркли появились до появления компьютерных сетей. Изначально они предназначались для взаимодействия между процессами в системе и только позже были приспособлены для TCP/IP.

Чтобы две программы могли общаться друг с другом через сеть, каждая из них должна создать сокет. Каждый сокет обладает двумя основными характеристиками: протоколом и адресом, к которым он привязан. Протокол задаётся при создании сокета и не может быть изменён впоследствии. Адрес сокета задаётся позже, но обязательно до того, как через сокет пойдут данные. В некоторых случаях привязка сокета к адресу может быть неявной.

Формат адреса сокета определяется конкретным протоколом. В частности, для протоколов TCP и UDP адрес состоит из IP-адреса сетевого интерфейса и номера порта.

Работа с сокетами в .NET

Поддержку сокетов в .NET обеспечивают классы в пространстве имен System.Net.Sockets – начнем с их краткого описания.

Класс	**Описание**
MulticastOption	Класс *MulticastOption* устанавливает значение IP-адреса для присоединения к IP-группе или для выхода из нее.
NetworkStream	Класс *NetworkStream* реализует базовый класс потока, из которого данные отправляются и в котором они получаются. Это абстракция высокого уровня, представляющая соединение с каналом связи TCP/IP.
TcpClient	Класс *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	UDP – это протокол, не организующий соединение, следовательно, для реализации UDP-обслуживания в .NET требуется другая функциональность. Класс *UdpClient* предназначен для реализации UDP-обслуживания.
SocketException	Это исключение порождается, когда в сокете возникает ошибка.
Socket	Последний класс в пространстве имен System.Net.Sockets – это сам класс Socket. Он обеспечивает базовую функциональность приложения сокета.

Класс Socket

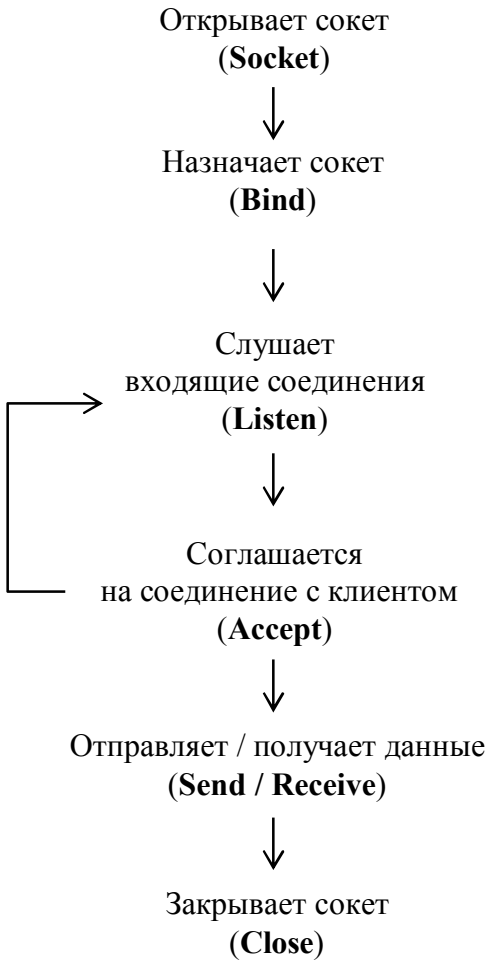
Класс Socket играет важную роль в сетевом программировании, обеспечивая функционирование как клиента, так и сервера. Главным образом, вызовы методов этого класса выполняют необходимые проверки, связанные с безопасностью, в том числе проверяют разрешения системы безопасности, после чего они переправляются к аналогам этих методов в Windows Sockets API.

Прежде чем обращаться к примеру использования класса Socket, рассмотрим некоторые важные свойства и методы этого класса:

Свойство	**Описание**
AddressFamily	Возвращает все адреса, используемые сокетом. Данное свойство представляет одно из значений, определенных в одноименном перечислении AddressFamily. Перечисление содержит 18 различных значений. Одно из наиболее используемых – InterNetwork (адрес по протоколу IPv4).
Available	Возвращает объем доступных для чтения данных.
LocalEndPoint	Возвращает локальную точку, по которой запущен сокет и по которой он принимает данные.
RemoteEndPoint	Возвращает адрес удаленного хоста, к которому подключен сокет.
ProtocolType	Возвращает одно из значений перечисления ProtocolType, представляющее используемый сокетом протокол. Наиболее используемыми являются Tcp и Udp.
SocketType	Возвращает тип сокета. Представляет одно из значений из перечисления SocketType.
Bind()	Связывает сокет с локальной конечной точкой для ожидания входящих запросов на соединение.
Listen()	Помещает сокет в режим прослушивания (ожидания). Этот метод предназначен только для серверных приложений.
Accept()	Создает новый сокет для обработки входящего запроса на соединение.
Connect()	Устанавливает соединение с удаленным хостом.
Receive()	Получает данные от соединенного сокета.
Send()	Отправляет данные соединенному сокету.
Shutdown()	Запрещает операции отправки и получения данных на сокете.
Close()	Заставляет сокет закрыться.

Клиент-серверное приложение на сокетах TCP. Сервер.

Рассмотрим пример, как создать сервер, работающий по протоколу TCP, с помощью сокетов. Структура сервера показана на следующей функциональной диаграмме:



Листинг 1.

```
using System;
using System.Text;
using System.Net;
using System.Net.Sockets;

namespace SocketTcpServer
{
   class Program
   {
      static void Main(string[] args)
      {
         // Создаем сокет (данный сокет используется только для установки соединения)
         Socket socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);
```

```
        // Устанавливаем для сокета локальную конечную точку
        IPEndPoint localEP = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 1111);

        try
        {
            // Связываем сокет с локальной точкой, по которой будем принимать данные
            socket.Bind(localEP);

            // Начинаем слушать (ожидать подключений со стороны клиентов)
            socket.Listen(5);

            Console.WriteLine("Сервер запущен. Ожидание подключений...");

            while (true)
            {
                // Метод Accept извлекает из очереди ожидающих запросов первый запрос
                // и создает для его обработки объект Socket.
                // Если очередь запросов пуста, то метод Accept
                // блокирует вызывающий поток до появления нового подключения
                Socket newSocket = socket.Accept();

                // Буфер для приема входящего сообщения
                byte[] data = new byte[1024];

                // Метод Receive() считывает данные в буфер
                // и возвращает число успешно прочитанных байтов
                int bytes = newSocket.Receive(data);

                // Преобразуем данные в строку
                string message = null;
                message += Encoding.UTF8.GetString(data, 0, bytes);

                // Отображаем полученное сообщение
                Console.Write(message);

                // Закрываем сокет созданный методом Accept()
                newSocket.Shutdown(SocketShutdown.Both);
                newSocket.Close();
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.ToString());
        }
        finally
        {
            Console.ReadLine();
        }
    }
}
```

Рассмотрим структуру данной программы.

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 (комбинация) создает конечную точку для обслуживания (сокет).

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* хоста, полученный от метода *Dns.Resolve()*, с *номером порта:*

```
IPHostEntry ipHost = Dns.GetHostEntry("localhost");
IPAddress ipAddr = ipHost.AddressList[0];
IPEndPoint ipEndPoint = new IPEndPoint(ipAddr, 1111);
```

Здесь класс *IPEndPoint* представляет *localhost* на порте 1111.

Далее новым экземпляром класса *Socket* создаем потоковый сокет:

```
Socket socket = new Socket(ipAddr.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(localEP);
```

Метод Bind() 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:

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

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.

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.

```
Socket newSocket = socket.Accept();
```

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

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.

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

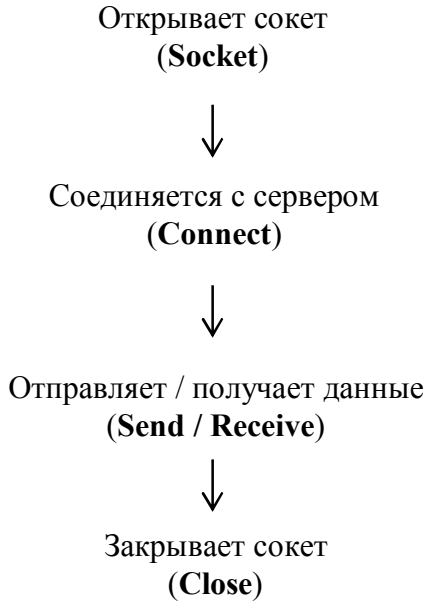
```
newSocket.Shutdown(SocketShutdown.Both);
newSocket.Close();
```

SocketShutdown – это перечисление, содержащее три значения для остановки: *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.

Клиент

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. Структура клиента показано на следующей функциональной диаграмме:



Листинг 2.

```
using System;
using System.Text;
using System.Net;
using System.Net.Sockets;

namespace SocketTcpClient
{
   class Program
   {
      static void Main(string[] args)
      {
         try
         {
            Socket socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);

            IPEndPoint remoteEP = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 1111);

            // Подключаемся к серверу, используя метод Connect() и конечную удаленную точку
            socket.Connect(remoteEP);
```

```
            // Формируем и отправляем сообщение
            byte[] data = Encoding.UTF8.GetBytes("Запрос клиента");
            socket.Send(data);

            // Закрываем сокет
            socket.Shutdown(SocketShutdown.Both);
            socket.Close();
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.ToString());
        }
        finally
        {
            //Console.ReadLine();
        }
    }
  }
}
```

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

Использование сокетов для работы с UDP

Протокол UDP не требует установки постоянного подключения, и, возможно, многим покажется легче работать с UDP, чем с TCP. Большинство принципов при работе с UDP те же, что и с TCP.

Вначале создается сокет. Вызов метода Socket() выглядит следующим образом:

```
Socket socket = new Socket(AddressFamily.InterNetwork, SocketType.Dgram,
ProtocolType.Udp);
```

Так как мы создаем сокет дейтаграмм, использующий протокол UDP, вторым аргументом, задающим тип создаваемого сокета, должен быть SocketType.Dgram.

Если сокет должен получать сообщения, то надо привязать его к локальному адресу и одному из портов с помощью метода Bind:

```
IPEndPoint localEP = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 1111);
socket.Bind(localEP);
```

После создания сокета в приложении-получателе и привязки его к определенному адресу и порту можно принимать данные от приложения-отправителя. Для получения сообщений используется метод **ReceiveFrom()**:

```
byte[] data = new byte[256]; // буфер для приема данных
 // адрес, с которого пришли данные
EndPoint remoteEP = new IPEndPoint(IPAddress.Any, 0);
int bytes = socket.ReceiveFrom(data, ref remoteEP);
```

В качестве первого параметра в метод ReceiveFrom() передается массив байтов, в который надо считать данные. Метод ReceiveFrom() считывает данные в массив, возвращает значение, указывающее число успешно прочитанных данных, и фиксирует ту конечную точку удаленного узла, с которой были посланы данные (по этому адресу можно послать ответ).

Если размер буфера, переданный в метод ReceiveFrom(), слишком мал для приема всей дейтаграммы целиком, буфер заполняется теми данными, которые в него помещаются, а избыточные данные теряются.

Для отправки данных используется метод **SendTo()**:

```
string message = Console.ReadLine();
byte[] data = Encoding.Unicode.GetBytes(message);
EndPoint remoteEP = new IPEndPoint(IPAddress.Parse(remoteAddr), remotePort);
Socket.SendTo(data, remoteEP);
```

В метод передается массив отправляемых данных, а также адрес, по которому эти данные нужно отправить.

Листинг 3.

```
using System;
using System.Text;
using System.Net;
```

```
using System.Net.Sockets;

namespace SocketSender
{
   class Program
   {
      static void Main(string[] args)
      {
         Socket socket = new Socket(AddressFamily.InterNetwork,
                              SocketType.Dgram, ProtocolType.Udp);

         EndPoint remoteEP = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 1111);

         // формируем сообщение
         Console.Write("Введите сообщение: ");
         string message = Console.ReadLine();

         byte[] data = Encoding.Unicode.GetBytes(message);

         // отправляем сообщение
         socket.SendTo(data, remoteEP); // в метод передается массив отправляемых данных, а также адрес, по которому эти данные необходимо отправить

         // закрываем сокет
         socket.Shutdown(SocketShutdown.Both);
         socket.Close();
      }
   }
}
```

Листинг 4.

```
using System;
using System.Text;
using System.Net;
using System.Net.Sockets;

namespace SocketRecipient
{
   class Program
   {
      static void Main(string[] args)
      {
         Socket socket = new Socket(AddressFamily.InterNetwork, SocketType.Dgram, ProtocolType.Udp);

         IPEndPoint localEP = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 1111);

         socket.Bind(localEP);

         Console.Write("Ожидание подключений:");

         // получаем сообщение
```

```
                string message = null;
                byte[] data = new byte[256]; // буфер для приема данных

                // адрес, с которого пришли данные
                EndPoint remoteEP = new IPEndPoint(IPAddress.Any, 0);

                int bytes = socket.ReceiveFrom(data, ref remoteEP);
                message += Encoding.UTF8.GetString(data, 0, bytes);

                // получаем данные о подключении
                IPEndPoint fullRemoteEP = remoteEP as IPEndPoint;

                // выводим сообщение
                Console.WriteLine("{0}:{1}-{2}", fullRemoteEP.Address.ToString(),
fullRemoteEP.Port, message);

                // закрываем сокет
                socket.Shutdown(SocketShutdown.Both);
                socket.Close();

                Console.ReadLine();
            }
        }
    }
```

Задание на лабораторную работу

1. Изучите пример простейшего клиент-серверного приложения на потоковом сокете TCP. Доработайте приложение следующим образом.

   − На сервере добавьте возможность автоответа, то есть автоматическое отправление клиентскому приложению ответа на принятое сообщение.

   − Организуйте взаимодействие типа клиент-сервер. Клиент отправляет запрос серверу на выполнение какой-либо команды. Сервер выполняет эту команду и возвращает результаты клиенту.

2. Изучите код приложений «Отправитель» и «Получатель».

   − Добавьте возможность ответа на сообщение, используя данные полученной удаленной точки (адрес и порт).

   − Объедините функции отправки и получения сообщений в одном приложении. Запустите две копии приложения. Протестируйте работу.

КОНТРОЛЬНЫЕ ВОПРОСЫ:

1. Что такое сокет?
2. Какие бывают сокеты, в чем их особенности?
3. Особенность приложения клиент – сервер, основанного на потоковом сокете?

4. Алгоритм установления связи между клиентом и сервером для взаимодействия на основе потокового сокета.
5. Методы и свойства класса Socket.
6. Как осуществляется прием и отправка данных между клиентом и сервером?

Литература

1. Кумар В., Кровчик Э и др. .NET. Сетевое программирование / Пер. с англ. –М.: «Лори», 2007, стр.112-137.