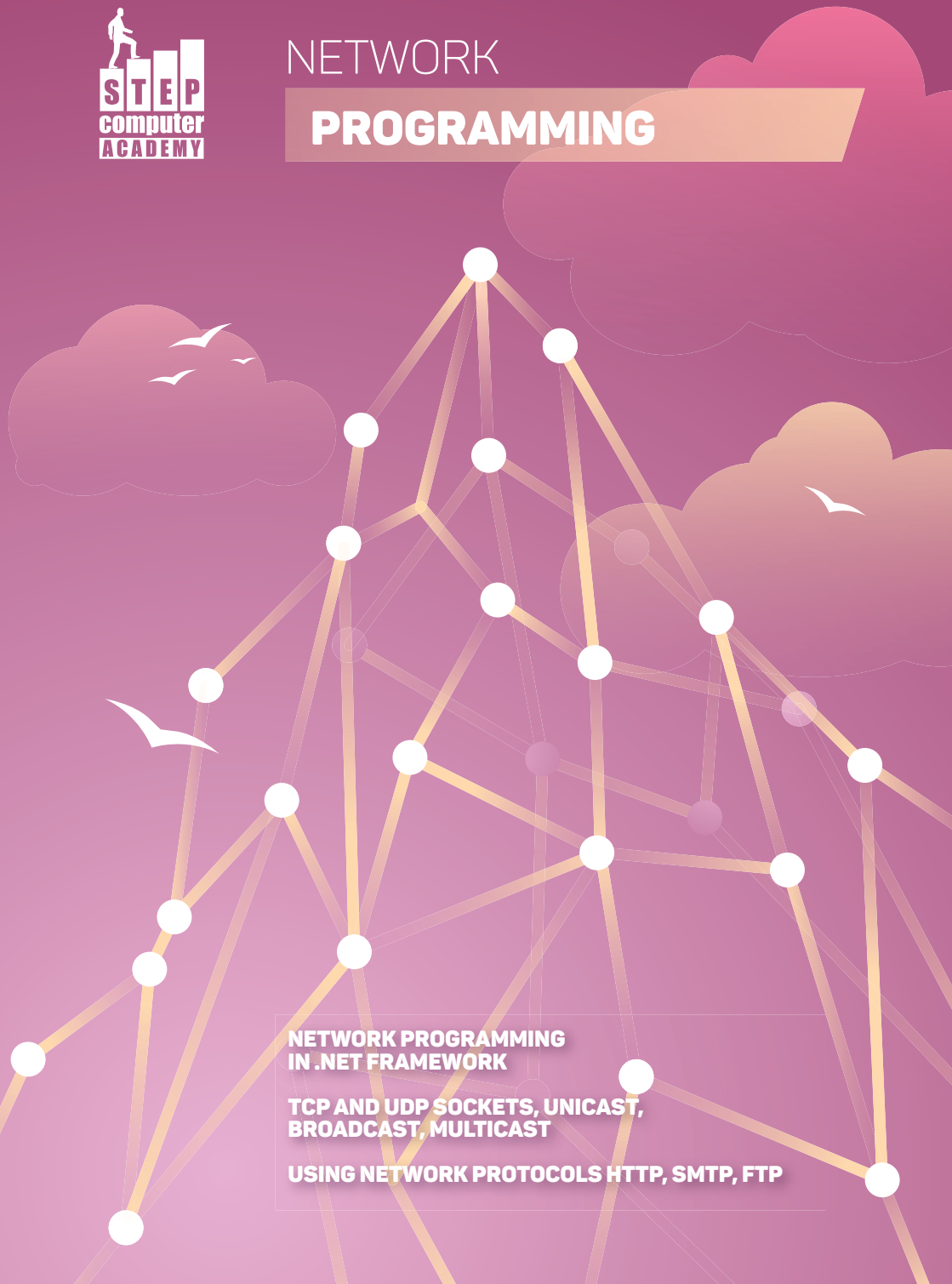
STEP computer ACADEMY

NETWORK PROGRAMMING



NETWORK PROGRAMMING IN .NET FRAMEWORK

TCP AND UDP SOCKETS, UNICAST, BROADCAST, MULTICAST

USING NETWORK PROTOCOLS HTTP, SMTP, FTP

Lesson 3

TCP and UDP sockets, unicast, broadcast, multicast

Contents

1. General Overview of TCP Protocol

TCP or *Transmission Control Protocol* is applied as a reliable providing interaction through the computer network. TCP tests whether the data is properly delivered to an addressee.

TCP is a protocol which is oriented on the connections. It means that two processes or applications should establish TCP connection before starting to exchange data. This is the main difference of TCP protocol from a UDP () protocol. The latter is a «connectionless» protocol, which allows exercising a broadcast data transmission to an indefinite client number. TCP protocol was specifically developed in order to provide a secure end-to-end byte streaming along an insecure internetwork. Such a network differs from a typical LAN by the fact that its various parts can have different topologies, bandwidth, delay time, package size, etc. Upon TCP development main attention was paid to the protocol capacity to adapt to the b internetwork properties and error tolerance in case any problems with data transmission arise.

TCP protocol is described in the RFC 793. In the course of time there were various errors and deficiencies found, and there were standard requirements on some articles were changed. A detailed description of these specifications is given in the standard RFC 793. Protocol extensions are described in the RFC 1323.

TCP is used in such application protocols as HTTP, FTP, SMTP and Telnet.

1.1. TCP Terminology

- ***Segment —*** a "portion" of data, which TCP sends to IP
- ***Datagram*** — a "portion" of data, which an IP send to the network interface layer.
- ***Number (No.)*** — each TCP segment, which is sent through the connection, is assigned with the index number so that a client application could eventually receive the data in a proper sequence.

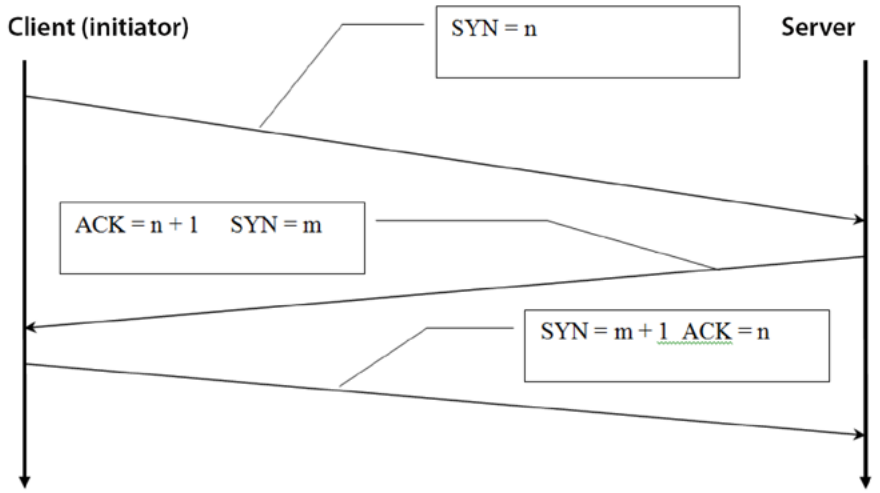
Each machine supporting TCP protocol has a so called TCP transport principle. It can be either a library procedure, or user interface, or part of the system core. In any case the transport principle manages TCP-threads and data exchange with an IP-layer (IP-layer within an OSI model is situated on the 3rd network layer). TCP-principle accepts users' data flows from the local processes, then splits them into the fragments, each 64 Kb maximum (on practice this number is usually equal to 1460 byte, which allows placing them within one Ethernet frame with the IP and TCP headings) and then sends them in a form of separate IP-datagrams. When IP-datagrams together with the TCP-data come onto the machine they are sent to the TCP-principle, which restores an initial byte flow. Sometimes TCP word is used to indicate the TCP principle, i.e. software part, and sometimes it is used to indicate a TCP protocol, i.e. set of the rules for data transmission.

IP layer does not guarantee a proper datagram delivery and that is why the TCP monitors the expired expecting intervals and if it is needed to resend the packages. Very often datagrams remain in incorrect order. And TCP protocol also can restore the messages from such datagrams. In other words, TCP protocol is to provide reliability that modern applications imply and which cannot be guaranteed by IP protocol.

1.2. TCP Connections

There is a certain action sequence for establishing TCP connection, it is also called a "Three-phase Handshake". The sequence consists of the following steps:

1. A client initiates interaction with the server by means of sending a segment with an established SYN bit (synchronization). This data segment contains an initial client index number.
2. In response server sends a segment with installed SYN bits (synchronization) and ACK (acknowledgement). This segment contains an initial server index number (not bind to a client index number) and acknowledgement number, which is one unit more than client index number. In other words, acknowledgement number is equal to the next expected client number.
3. A client should verify the segment receipt by responding with a segment with an installed ACK bit (acknowledgement). Verification number will be one unit more than server index number, and the number will be equal to server acknowledgement index number.

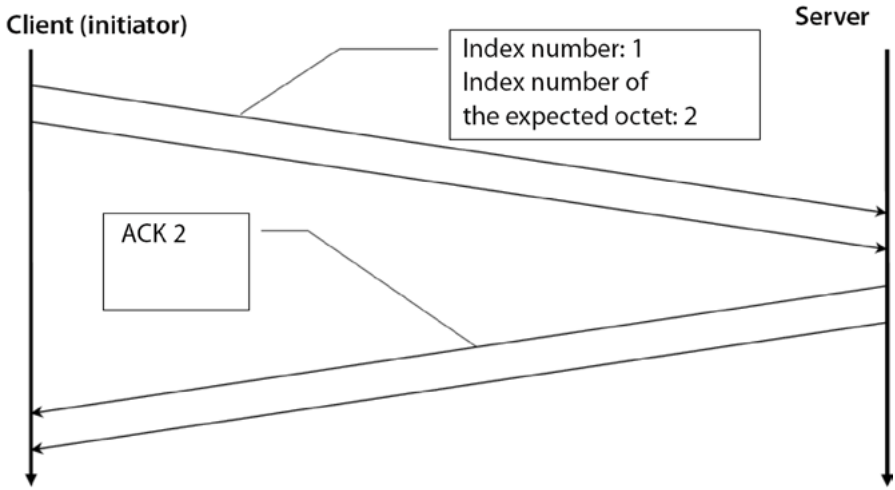


Data Streaming in TCP

TCP protocol transmits the data in portions, which are called segments. In order to guarantee the segment accept in a proper order, each of them is assigned with an index number. A receiver sends acknowledgement of segment receipt. If verification was not received before expiration of time-out interval, the data will be retransmitted. Every eight bit of data (so called octet) are assigned with a number. Segment number is equal to the number of the first data octet within a segment; this number is sent within a TCP heading of the segment. There could be a verification number within a segment, and it will be equal to the index number of the next expected data segment.

TCP uses index numbers to guarantee non transmission of the data duplicates to the application and also guarantees data delivery in a proper order. TCP heading contains a check sum to guarantee validity of the data delivered to an addressee.

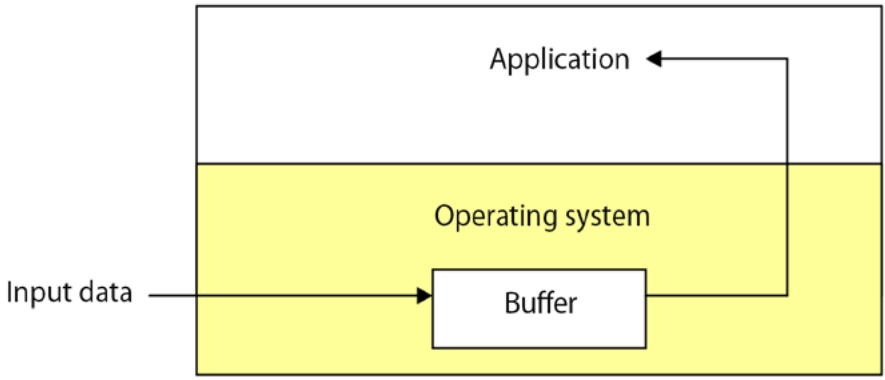
If a segment is delivered to an addressee with a wrong check sum, then such a segment is rejected and the acknowledgement about this segment receipt will not be sent. This will lead to the fact that after expiration of the time-out a sender will have to repeat transmission of a non-delivered segment.



Data Flow Management

TCP protocol manages the volume of the directed to him data using an indication about a so called frame (window) size in each confirmation. Frame size is volume of data, which an addressee can receive. Between an application and data flow in the network there is a special data buffer. Frame size is actually a difference between the buffer size and the volume of the data it stores. This number is sent within a heading in order to inform a remote host about the current window size. This technique is called a sliding window.

Sliding window algorithms manage the data flow transmitted by the network:



Received data is stored within the buffer, and upon this an application can address the buffer at a customary rate. This rate depends of various factors (CPU boot, application priority, flow priority where buffer reading takes place, etc.). With the course of reading of data from the buffer it опустошается and gets an opportunity to record new data coming from the network.

Multiplexing

TCP allows a few processes running on one machine simultaneously use TCP socket. TCP socket consists of host address and unique port number, and TCP-connection includes two sockets on different network sides. Port can be used for several connections simultaneously, meaning that the socket on one side of the connection can be used for the operating of several connections with different sockets at the other end. A web–server could serve as a typical example of the situation, listening on the 80th port and responding the queries of several machines.

1.3. TCP Headings

Structure of TCP heading is reviewed in the next scheme:

TCP heading

<table>
<tr><td colspan="8">0 bit</td><td>16 bit
32 bit</td></tr>
<tr><td colspan="8">Port-resource</td><td>Port-addressee</td></tr>
<tr><td colspan="9">Index number</td></tr>
<tr><td colspan="9">Verification number</td></tr>
<tr><td>Length of TCP heading</td><td>Reserve</td><td>URG</td><td>ACK</td><td>PSH</td><td>RST</td><td>SYN</td><td>FIN</td><td>Frame (window) size</td></tr>
<tr><td colspan="8">Check sum</td><td>Indicator to the string data</td></tr>
<tr><td colspan="8">Options</td><td>Filling</td></tr>
</table>

Index number and verification number are used by TCP in order to guarantee the proper order of the data.

In the fourth byte after TCP heading length and reserve bits there are managing bits (flags):

- ***URG*** — means that the segment contains string (urgent) data
- ***ACK*** — means that the segment contains acknowledgement number
- ***PSH*** — means that the segment should be pushed towards the addressee RST — connection reset
- ***SYN*** — is applied upon synchronization of index numbers
- ***FIN*** — end of data.

When an application transmits data using TCP, they are moving within them along the protocol stack. Data passes

through all layers and in the end they are passed through the network as a bit flow. And what signal corresponds either 0 or 1 is defined at the lowest layer — physical layer. Each layer in a set of the TCP/IP protocols adds additional information to the initial data in form of the headers or trailers. When a package comes to the end node of the network, it passes through all layers again from the bottom to the tops and reaching an application layer in the end. Each layer analyses data separating from the package from header and/or trailer information and after all transformations data reaches an application-server in the same form as they have left an application-client.

Example of data transmission between the OSI layers for TCP/IP

	User data				
Application layer	Application header	User data			
Transport layer	TCP header	Application data = application header + user data			
Network layer	IP header	TCP header	Applica-tion data		
Data link layer	Ethermet header	IP header	TCP header	Applica-tion data	Ethermet trailer

1.4. Advantages and Disadvantages of TCP

Advantages of TCP

TCP — is a complex, cost and time consuming protocol, which can be explained by its connection establishment mechanisms, but it also is responsible for secured packages delivery, saving the programmer from the necessity to switch on this functionality into an application protocol.

TCP exercises repeated data request in case of data loss and removed duplication in case two copies of one package come. In comparison with the UDP protocol, it guarantees that application will receive the data in the same sequence they were sent and without any losses.

Realization of TCP, as a rule, is built in the core of the modern operating system, though there are TCP realizations within the application context.

Common Disadvantages of TCP

TCP requires a direct indication of maximum segment sizea (MSS) in case if virtual connection is exercised through a network segment where maximum unit size (MTU) is less than the standard MTU Ethernet (1500 byte).

Within tunneling protocols, such as GRE, IPIP maximum unit size is less than a standard one; that is why maximum TCP segment has a bigger package length than MTU. Though fragmentation in majority of the cases is forbidden, such packages are rejected.

This problem comes through the “hang-up” of the connections. While “hang-up” can happen any moment, and especially when sender had used the segments of the size longer that admissible.

To resolve the problem Firewall rules are applied at the routers, these rules add MSS parameter to all packages initiating connections in order a sender could use the segments of admissible size.

Error Diagnosis upon Data Transmission

Though the protocol exercises a check sum test, the applied algorithm is considered to be relatively weak. For example in 2008 an error that was not detected in one transmitted bit by networking tools led to shut-off of the Amazon Web Services system services.

In general distributed networking applications should be used together with the additional software tools in order to secure the integrity of transmitted data.

Attacks on the Protocol

Disadvantages of the protocol involve theoretical and practical attacks when an attacker can obtain an access to transmitted data, to pass himself for the other side or lead to system failure.

2. General Review of UDP Protocol

UDP or ***User Datagram Protocol*** (*user datagram protocol*) — is a simple datagram oriented protocol without connection establishment, providing fast and not necessarily secure data transmission method. It provides a "one-to-many" communication, that's why it is often used for broadcast and group datagram transmissions.

<table>
<tr><th colspan="2">OSI layers</th><th>TCP /IP</th><th colspan="5">Stack of protocols TCP /IP</th></tr>
<tr><td>7</td><td>Application</td><td rowspan="3">Application</td><td rowspan="3">HTTP</td><td rowspan="3">FTP</td><td rowspan="3">SMTP</td><td rowspan="3">RIP</td><td rowspan="3">DNS</td></tr>
<tr><td>6</td><td>Presentations</td></tr>
<tr><td>5</td><td>Session</td></tr>
<tr><td>4</td><td>Transport</td><td>Transport</td><td colspan="4">TCP</td><td>UDP</td></tr>
<tr><td>3</td><td>Network</td><td>Internet</td><td colspan="5">IP</td></tr>
<tr><td>2</td><td>Data link</td><td rowspan="2">Network</td><td colspan="5" rowspan="2">Ethernet, ATM, Frame Rela, etc.</td></tr>
<tr><td>1</td><td>Physical</td></tr>
</table>

UDP is situated on a transport layer above IP, i.e. network layer protocol. Transport layer provides interaction between the networks through the gateways. It uses IP–addresses for sending data packages through the Internet or any other network using devices drivers. TCP and UDP are included into a set of TCP/IP protocols, each of them has its advantages and disadvantages.

2.1. UDP Terminology

Packages — upon data sending a package is a sequence of binary numbers representing management data and signals, which are transmitted and commuted through the host. Inside the package all the information is arranged according to a special format.

Datagram is a separate independent package containing enough information for the transmission from the sender to an addressee. Datagram in this sense is "independent", there is no any additional exchange between the source, addressee and transport network required.

MTU is an abbreviation that means Maximum Transmission Unit. MTU characterizes a data link layer and corresponds to the maximum number of bytes, which can be sent within one package. In other words, MTU is the biggest package, which particular network medium can handle. For example, Ethernet has a fixed value of MTU, which is equal to 1500 bytes. Within a UDP protocol, if datagram size is bigger that the MTU's one, then protocol exercises a fragmentation splitting one datagram into several fragments so that each fragment were less than MTU.

Port — UDP uses ports in order to ensure any process running of a computer in accordance with the input data. UDP sends a package to an addressee using port number indicated in UDP — datagram header. Ports are represented by 16-bit numbers and consequently they accept values in a range from 0 to 65535. Ports, which are also terminals of logic connections, are divided into three categories:

- ***Common ports*** — from 0 to 1023
- ***Registered ports*** — from 1024 to 49 151
- ***Dynamic / frequency ports*** — from 49 152 to 65 535

UDP uses the following common ports

Number of UDP port	Description
15	NETSTAT — network status
53	DNS — server of domain names
69	TFTP — he simplest file transfer protocol
137	NetBIOS name service
138	NetBIOS datagram service
161	SNMP

It should be noted that UDP ports can receive more than one message at the same time. In some cases TCP and UDP services can use the same port numbers, for example 7 (Echo) and 23 (Telnet).

List of UDP and TCP ports you can find in the Internet ref. http://www.iana.org/assignments/port-numbers

IP-addresses

IP datagram consists of 32-bit IP addresses of the source and destination. Destination IP address knows a UDP datagram destination point, and source IP-address is used for obtaining information about a sender. At the destination point all the packages are filtered and the sources addresses which are not included within an admissible address set are automatically rejected without notifying the sender.

One-way IP-address precisely identifies a host within the network, while group IP identifies a particular group of addresses within the networkт. Broadcast IP addresses are

accepted and handled by all hosts within LAN and relevant sub-network.

IP addresses are divided into five classes

IP address	Range of IP addresses	Use
Class A	From 0.0.0.0 to 127.255.255.255	Within the networks with big number of hosts, for example big corporations networks
Class B	From 128.0.0.0 to 191.255.255.255	Networks with an average number of hosts
Class C	From 192.0.0.0 to 223.255.255.255	Networks with a small number of hosts
Class D	From 224.0.0.0 to 239.255.255.255	Multicasting networks
Class E	From 240.0.0.0 to 247.255.255.255	Reserved for experiments

Several IP addresses are reserved for special cases:

IP address	Use
0.0.0.0	IP address of default host
127.0.0.1	IP address of local host
255.255.255.255	Broadcast IP address for the whole LAN

TTL — (time to live) life-time value. It allows installing an upper limit of the routers number through which the datagram may pass. TTL use doesn't allow packages to be continuously sent by the network. It initializes by the sender and is reduced by one by each router processing datagram. When TTL value drops to zero the datagram is canceled.

Multicasting is an open standards-based method of simultaneous distribution of identical information to several

users. Multicasting is main tool of UDP protocol, and it is impossible with the TCP protocol. Multicasting allows reaching interaction of one with many, for example it makes possible the following services: news and mails sending to several addressees; Internet-radio or real time demonstration programs. Multicasting doesn't overload the network as much as a broadcast, because the data is sent immediately to particular users, other don't receive the data.

2.2. Principle o UDP Operation

When a UDP-based application sends data to other host within a network, UDP complements them with the 8 bit header containing the addressee and sender port numbers, general amount of data and check sum. An IP adds its heading above UDP datagram forming an IP datagram.

Format of UDP

<table>
<tr><td rowspan="8">Дейтаграмма IP</td><td rowspan="5">Header IP</td><td colspan="2"></td><td>Version (4 bit)</td><td>Header type (4 bit)</td><td>Service type (8 bit)</td><td colspan="2">General length (16 bit)</td></tr>
<tr><td colspan="2"></td><td colspan="3">Identification (16 bit)</td><td>Flag and (3 bit)</td><td>Fragment shift (13 bit)</td></tr>
<tr><td colspan="2"></td><td colspan="2">Life time (8 bit)</td><td>Protocol (8 bit)</td><td colspan="2">Header check sum (16 bit)</td></tr>
<tr><td colspan="2"></td><td colspan="5">IP-address of source (32 bit)</td></tr>
<tr><td colspan="2"></td><td colspan="5">IP-address of sender (32 bit)</td></tr>
<tr><td rowspan="3">UDP datagram</td><td rowspan="2">UDP header</td><td colspan="3">Number of source port (16 bit)</td><td colspan="3">Number of sender's port (16 bit)</td></tr>
<tr><td colspan="3">UDP length (16 bit)</td><td colspan="3">Check sum (16 bit)</td></tr>
<tr><td>Data</td><td colspan="6">Data + filler bytes</td></tr>
</table>

General length of UDP header has 8 bytes. Maximum possible size of IP datagram is equal to 65535 bytes. Considering twenty bytes of IP header and eight bytes of UDP the length of users' data can reach 65535 – 28 = 65507 bytes. Though the majority of the software operate data of smaller size. For example, for the majority of software the datagram length of 8192 bytes, because this is the default data volume which is read and record by network file system NFS.

If the package length exceeds a preset default MTU, on the network layer (in this case IP) the package is split into the fragments. IP header contains all necessary information about the fragments.

When a software-sender transmits a datagram into the network, it directs to a designation IP address. This address is indicated within an IP header. Upon passing through the router the value of time (time to live, TTL) is reduced by one.

When datagram comes to a preset destination point and port, an IP header is checked on the network layer, and defined whether it is fragmented. If yes, then datagram is formed in accordance with the information recorded within a header. And at last at the application layer a header is deleted and application–addressee receives the transmitted data.

2.3. Advantages of UDP

- ***No connection establishment***. As UDP is a protocol without arrangement of the connections, it does not require applied expenditures connected with the setting and check of the connections. Consequently we can avoid connection establishment delays. This UDP peculiarity DNS service

uses, and working with connection establishment would slow its work considerably.

- Due to the absence of the connection, UDP works ***faster*** than TCP.
- ***Topologic diversity***. UDP supports possibilities of interaction "one with many" and "one with one", while TCP supports only interaction "один с одним".
- ***Header size***. For each package UDP header is 8 bytes long, while TCP has 12 byte long header.

2.4. Disadvantages of UDP

- ***Absence of handshaking signal***. As long as there is no connection establishment, a sender has no opportunity to find out if a datagram reached an addressee. As a result UDP cannot guarantee the delivery if compared with the TCP, which uses handshaking signals.
- ***Absence of sessions***. In comparison with TCP, UDP protocol cannot support sessions. TCP uses a special session identifier.
- ***Order of delivery***. UDP does not guarantee that an addressee will receive only one data copy. UDP does not guarantee that data will be received in the same order as they were created by a source. TCP protocol together with the port numbers uses index numbers and continuously sent confirmations, which guarantees an arranged data delivery.
- ***Security***. TCP is safer than UDP. Brandmauers and routers do not transmit UDP packages. This happens due security reasons, because hackers can use UDP ports without establishment of obvious connections.

- ***Flow management***. There is no flow management in UDP; that is why poorly designed UDP application can considerably reload a network.

Summary table of UDP and TCP protocols characteristics

Characteristic	UDP	TCP
Establishment of connections	No	Yes
Use of sessions	No	Yes
Availability of acknowledgments	No	Yes
Arranged transmission	No	Yes
Flow management	No	Yes
Security	Lower	Higher
Relative rate	Higher	Lower
Interaction	One with many, one with one	One with one
Header length	8 bytes	20 bytes

3. Working with TCP Protocol on the Basis of the platform .NET

Before platform .NET was created there were practically no tools for top level TCP programming. For example, in Visual C++ we had to use classes CSocket and CAsyncSocket. In the .NET classes for working with the sockets are placed within a separate name space System.Net.Sockets. This name space contains both низкоуровневые классы (for example, Socket) as well as TcpListener and TcpClient classes for arranging applications interaction using TCP protocol. These classes have simple interfaces if compared with the Socket class, which implies byte-by-byte approach for transmitting/receiving data, TcpListener and TcpClient classes use threading model. These classes imply that interaction between a client and server is based on the flow usinga NetworkStream class. If necessary it is possible to work with separate bytes.

3.1. TCPListener Class

Application — server starts its operation by means of installing the connection with a local terminal and being in the waiting state for incoming client queries. As soon as a server receives a client request connected with the relevant port an application-server is activated. Application-server creates a channel designated for interaction with the client.

Application-server waits for other income client requests within the main flow. TCPListener class in designated for listening to the client requests and creating a new pattern of Socket class or TCPClient class, which can be used for the further interaction with the client. Both TCPClient class and TCPListener class encapsulate within it a Socket class pattern in a form of the field m_Serversoket with protected access level, available for sub-classes only. Main class methods are represented within a table:

Return result	Method name	Description
Socket	AcceptSocket()	Allows creating a connection and returns a Socket object used for interaction with the client
TcpClient	AcceptTcpClient()	Allows creating a connection and returns a TcpClient object used for interaction with the client
bool	Pending()	Shows if there are requests waiting for connection
void	Start() / Start(int)	Makes the instance TCPListener starts listening the connection requests
void	Stop()	Closes the instance TCPListener that is in listening mode
IAsyncResult	BeginAcceptTcpClient (AsyncCallback *callback*, object *state*)	Starts asynchronous operation for accepting an income query
TcpClient	EndAcceptTcpClient (IAsyncResult *asyncResult*)	Asynchronously accepts an incoming connection query and creates a new instance of TcpClient class for working with a client

Main properties of TCPListener class:

Type	Property name	Description
IPEndpoint	LocalEndpoint	Returns an object realizing an IPEndpoint interface, which contains information about local network interface and port number used for waiting of incoming queries from clients
bool	Active	Shows whether the instance of TCPListener class is listening at to the connection queries at the moment
Server	Socket	Returns an object of basic class Socket used by TCP object for listening to the queries

Creating an Instance of TCPListener Class

In order to create an instance of TCPListener class the following constructors are used:

- public TcpLitener(int port) // is considered to be aged
- public TcpLitener(IPEndPoint endPoint)
- public TcpLitener(IPAddress ipAddr, int port)

Port number used for listening to the queries is sent to the first constructor. In this case IP address is equal to IPAddress.Any, i.e. server accepts client actions on all network interfaces, and this value is equivalent to IP-address 0.0.0.0. IPEndPoint object defining IP address and port at which listening is executed is sent to the second constructor:

```
IPAddress adr = IPAddress.Parse("127.0.0.1");
IPEndPoint endp = new IPEndPoint(adr, 11000);
TcpListener tcpListener = new TcpListener(endp);
```

The last overloaded constructor accepts IPAddress object and port number:

```
IPAddress adr = IPAddress.Parse("127.0.0.1");
int port = 11000;
TcpListener tcpListener = new TcpListener(adr, port);
```

Client Listening

At the next step after socket creation we can shift to listening client queries. Class TCPListener has Start() method, which executes the following steps upon its call:

1. At first it binds socket using an IP-address and port sent to the constructor TCPListener
2. Then after having called a Listen() method of basic class Socket, it starts to listen the connection requests from the clients:

```
TcpListener tcpListener = new TcpListener(adr, port);
tcpl.Start();
```

Having proceeded to socket listening, we can request Pending() method for testing if there are any pending connection requests in application queue. This method allows to test availability of clients waiting request of Accept() method, which blocks executed flow:

```
if (tcpListener.Pending())
{
    this.Title = "there are connection requests
                  in the software queue";
}
```

Establishing Connections with Clients

In common case a software-server operates with two sockets: one is used by TCPListener class, and the other is used for interaction with a separate client. In order to accept a pending query in the queue, we can use AcceptSocket() or AcceptTcpClient() methods. These methods return objects Socket or TCPListener respectively and give consent to clients' queries.

```
Socket socket = tcpListener.AcceptSocket();
```

or

```
TcpClient tcpClient = tcpListener.AcceptTcpClient();
```

Sending and Receiving Messages

Depending on the socket type created during connection establishment, real exchange between the clients and a socket is exercised by means of Send() and Receive() methods of Socket object. We can also use writing/reading methods of NetworkStream class.

Stop of Listening the Program-Server

After termination of interaction with the client we should call a method Stop() class TCPListener in order to stop socket listening.

TcpClient Class

TcpClient class provides TCP services for connections of the client side. This class encapsulates a template of Socket class and providesт TCP-services on the higher level if compared with the level of Socket. Template of Socket class — is

a private field m_ClientSoket used for interaction with TCP server. TcpClient class contains the following properties:

Type	Property name	Description
LingerOption	LingerState	Allows creating a connection and returns a Socket object used for interaction with a client
bool	NoDelay	Allows creating a connection and returns a TcpClient object used for interaction with a client
int	ReceiveBufferSize	Shows fi there any queries pending connection
int	ReceiveTimeout	Sets up waiting time (m/sec) that class awaits to receive data af-ter initiating this operation. When this time expires an exception SocketException is generated
int	SendBufferSize	Sets up a buffer size for outcome data
int	SendTimeout	Sets up waiting time (m/sec) that class awaits to get a data reception confirmation. When this time expires an exception SocketException is generated
bool	Active	Indicates if there is active connection with a remote host
Socket	Client	Sets up a Socket object encapsulated by TcpClient

Main methods of TcpClient class:

Возвращаемый результат	Method name	Description
void	Close()	Closes a TCP connection
void	Connect()	Establishes a connection with a remote TCP host
NetworkStream	GetStream()	Returns an object NetworkStream, which is used for data transmission between a client and remote host

Constructors of TcpClient Class

There are four constructors defined within a class

- TcpClient()
- TcpClient(IPEndPoint ipEnd)
- TcpClient(string hostName, int port)
- TcpClient(AddressFamily family)

Default constructor creates a TcpClient object so that an object receives an IP address and port number from a service-provider. Then a Connect() method must be called for establishment of the connection. This constructor operates IPv4 address types only.

Second reloaded constructor accepts one IPEndPoint parameter. It initiates a new экземпляр of TcpClient class linked with the indicated EndPoint. It should be noted that this is not a remote, but local EndPoint. If we try to transmit to a constructor a remote EndPoint, then an exception with a massage about a wrong IP is cast out within this context.

If to use a constructor with a template of IPEndPoint class, then after creating a template TcpClient we should call a method Connect():

```
//creating local end point
IPAddress adr = IPAddress.Parse("127.0.0.1");
//creating a template of IPEndPoint class
IPEndPoint endPoint = new IPEndPoint(adr, 9876);
//creating a template of TcpClient class
TcpClient client = new TcpClient(endPoint);
//attempt to establish connection with a server
client.Connect("192.168.1.52", 9877);
```

Parameter transmitted to a TcpClient constructor is a lovcal EndPoint, while Connect() method establishes a connection netween a client and server and that is why it accepts a remote EndPoint as a parameter.

He third overloaded constructor creates a new экземпляр класса TcpClient and establishes a connection using two parameters — DNS-name and port number, for example

```
TcpClient client = new TcpClient("192.168.1.52", 80);
```

In this case we indicate an IP address of a remote host and port on the remote host. Thus, in this case we cannot indicate a local port address. There is another constructor that allows transmitting a template AddressFamily of enumeration type in a capacity of a parameter, possible values are represented below:

Name	Description
Unknown	Family of unknown addresses
Unspecified	Family of non-indicated addresses
InterNetwork	Address for IP version 4
Iso	Address for ISO protocols
Osi	Address for OSI protocols
Ecma	Address for ECMA association
DataLink	Direct interface address of data-link interface
NetBios	NetBios address
FireFox	FireFox address
Banyan	Banyan address
InterNetworkV6	Address for IP version 6

Establishing a Connection with a Host

After creating a template of TcpClient class at each next step we should call a method Connect(). It has several overloaded variants:

Name	Description
Connect(IPEndPoint)	Establishes a connection with a remote TCP host using IPEndPoint.
Connect(IPAddress, Int32)	Establishes a connection of a client with a remote TCP host using an indicated IP address and port number
Connect(IPAddress[], Int32)	Establishes a connection between a client and a remote TCP using several indicated IP addresses (in an array) and a port number
Connect(String, Int32)	Establishes a connection between a client and a remote TCP using host name and port number, for example newClient.Coonect("127.0.0.1", 80)

Since connection establishment depends on many factors, including quality network functioning, availability of a running application — server, it is recommended to call this method in a block try/catch:

```
try
    {
        TcpClient client = new TcpClient("localhost", 80);
    }

    catch (SocketException sockEx)
    {
        MessageBox.Show("Socket error: " +
                        sockEx.Message);
    }

    catch (Exception Ex)
    {
        MessageBox.Show("Error : " + Ex.Message);
    }
```

Sending and Receiving Messages

NetworkStream type is used for arranging the interaction between two connected applications a template of. Before sending and receiving any messages we should define a basic flow. Class TcpClient gives us a GetStream() method for these purposes. Using a socket it creates a template of NetworkStream class and returns it to the calling program. After we have received the flow we should use methods of a flow class Write() and Read() for reading the host and record from an application. Example of Write() method application:

```
try
{
    TcpClient client = new TcpClient("localhost", 80);
    NetworkStream nwStream = client.GetStream();
    byte []bt = Encoding.ASCII.GetBytes("test
                of connection!");
    nwStream.Write(bt, 0, bt.Length);
}
catch (SocketException sockEx)
{
    MessageBox.Show("Socket error: " + sockEx.Message);
}
catch (ArgumentNullException nullEx)
{
    MessageBox.Show("Object is not created");
}
catch (IOException ioEx)
{
    MessageBox.Show("Read/write error" + ioEx.Message);
}
```

Method Write receives three parameters — byte array, position within an array, which a starting point for reading

the data and number of bytes that should be written in the flow. In the aforementioned example a whole byte array is written within the flow, that we obtained in the result of transforming an ASCII encoded string.

Method Read() also has three parameters — byte array for storing read data, position within an array, which a starting point for data placing and number of bytes a method has to read:

```
TcpClient client = new TcpClient("localhost", 80);
NetworkStream nwStream = client.GetStream();
client.ReceiveBufferSize = 1000;
byte []btRead = new byte[client.ReceiveBufferSize];
nwStream.Read(btRead, 0, client.ReceiveBufferSize);
```

Property ReceiveBufferSize of TcpClient class serves to set up buffer length for reading.

Closure of TCP Socket

After termination of interaction with a client we should call a Close() method. It allows releasing the resources, connected with the TCP socket.

Example of Classes Use for Working with TCP

In order to illustrate the manipulations with the TcpListener and TcpClient classes we can build two window applications — server and client. Server receives a message from a client and adds it to a special list of received messages.

Server application code:

```
using System;
using System.Threading;
using System.IO;
```

```
using System.Net.Sockets;
using System.Net;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace TCP_1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        TcpListener list;
        private void button1_Click(object sender,
                EventArgs e)
        {
            try
            {
                //creating a template of TcpListener
                //class data about a host and port
                //are read from text windows
                list = new TcpListener(
                    IPAddress.Parse(textBox1.Text),
                    Convert.ToInt32(textBox2.Text));
                //beginning of slistening
                //to the clients
                list.Start();
                //creating a separate flow for message
                //reading
                Thread thread = new Thread(
                        new ThreadStart(ThreadFun)
                        );
```

```
            thread.IsBackground = true;
            //startup of a flow
            thread.Start();
        }
        catch (SocketException sockEx)
        {
            MessageBox.Show("Socket error: " +
                        sockEx.Message);
        }
        catch (Exception Ex)
        {
            MessageBox.Show("Error : " +
                        Ex.Message);
        }
    }

    void ThreadFun()
    {
        while (true)
        {
            //server notifies a client about
            //the readiness
            //for the connection
            TcpClient cl = list.AcceptTcpClient();
            //data reading from a netork
            //in the Unicode format
            StreamReader sr = new StreamReader(
                cl.GetStream(), Encoding.Unicode
            );
            string s = sr.ReadLine();
            //adding of received message into a list
            messageList.Items.Add(s);
            cl.Close();
            //upon receiving a message EXIT we
            //should terminate an application
            if (s.ToUpper() == "EXIT")
            {
```

```
                    list.Stop();
                    this.Close();
                }
            }
        }

        private void Form1_FormClosed(object sender,
                FormClosedEventArgs e)
        {
            if (list != null)
                list.Stop();
        }
    }
}
```

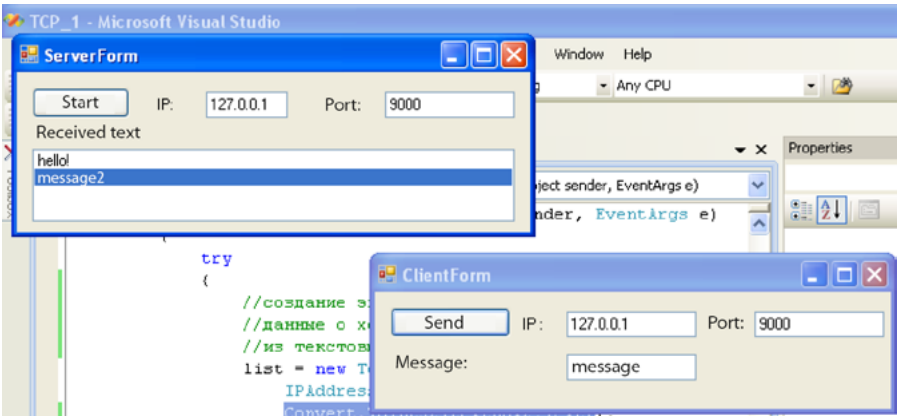
Client Application Code:

```
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace tcp_client
{
    public partial class Form1 : Form
    {
        TcpClient client;
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender,
                EventArgs e)
```

```
{
    try
    {
        //creating a template of IPEndPoint
        //class
        IPEndPoint endPoint = new IPEndPoint(
            IPAddress.Parse(textBox1.Text),
            Convert.ToInt32(textBox3.Text)
        );
        client = new TcpClient();
        //establishment of connection using
        //IP data and port number
        client.Connect(endPoint);

        //receiving a network flow
        NetworkStream nstream = client.
                                GetStream();
        //transformation of message string
        //into a byte array
        byte[] barray = Encoding.Unicode.
                        GetBytes(textBox2.Text);
        //record of the whole array within
        //a network flow
        nstream.Write(barray, 0, barray.Length);
        //client closure
        client.Close();
    }
    catch (SocketException sockEx)
    {
        MessageBox.Show("Socket error:" +
                    sockEx.Message);
    }
    catch (Exception Ex)
    {
        MessageBox.Show("Error :" +
                    Ex.Message);
    }
```

```
        }
        private void Form1_FormClosed(object sender,
                FormClosedEventArgs e)
        {
            if (client != null)
                client.Close();
        }
    }
}
```

Example of Applications Functioning in a Form of Server-Client Interaction



4. Working with UDP Protocol on the .NET Platform

In comparison with the TCP protocol that employs handshaking (data sending / receiving control), UDP protocol doesn't have such a mechanism, which allows working much faster. UDP is often used to transmit multimedia (for example, video), for an exact order of packages delivery to a destination point is often not important.

There is a special UdpClient class on the .NET platform for realizing the User datagramm Protocol (UDP). There are also other opportunities for realizing the UDP — Socket class, Winsock control element and API Winsock. The last two employ COM technology. UdpClient class is built above the Socket class, and when a programmer uses the UdpClient he in fact works with Socket's template.

In .NET classes for working with the protocols TCP and UDP are placed within System.Net.Sockets namespace.

4.1. UdpClient Class

Scheme of UdpClient class use is rather simple. At first you should create a UdpClien class sample. Then by means of calling Connect() method we should establish the connection with the remote host. And since UDP is not oriented for establishing the connections, method Connect() does not establish connection with the remote

host before sending or receiving the data. When we send a datagram, a destination point should be known, with this purpose we need to indicate an IP address and port number. The third step is sending and receiving data using the methods Send() and Receive(). In the end of the work we should call the method Close() for closure of the connection with UDP.

Main methods of the lass are represented within a table:

Return result	Method name	Description
void	Connect (IPEndPoint *endPoint*) Connect (IPAddress, int) Connect (string, int)	Sets up a remote host parameters
byte[]	Receive (ref IPEndPoint remoteEP)	Returns a UDP datagram, which was sent by a remote host
int	Send (array<Byte>[], Int32, IPEndPoint) Send (byte[] dgram, int bytes)	Sends a UDP datagram to a remote host
void	Close()	Closes the current UdpClient sample
void	JoinMulticastGroup (IPAddress) JoinMulticastGroup (Int32, IPAddress)	Joining a group of multi-address mailout (group mail-out), which is defined by the parameter IPAddress. Parameter Int32 defines a TTL (time-to-live) value
void	DropMulticastGroup (IPAddress) DropMulticastGroup (IPAddress, int)	Excluding from the group of multi-address mailout, which is defined by the parameter IPAddress

Main properties of UdpClient class:

Type	Property name	Description
bool	Active	Returns or establishes a variable that defines whether the parameters of a remote host were set up for the connection. This is a protected property
Socket	Client	Returns an encapsulated field of Socket type within UdpClient class. This is a protected property
Bool	DontFragment	Returns or establishes a variable that defines whether we can defragment a datagram IP

Creation of a UdpClient Class Sample

In order to create a sample of TCPListener class we use the following constructors:

Name	Secription
UdpClient()	Creates a sample of UdpClient class. Provider-service assigns it with an IP address and port number. Then we should either call a Connect() method or send information about the connection together with data transmission
UdpClient (AddressFamily)	Creates a sample of UdpClient class using AddressFamily enumeration
UdpClient (Int32)	Creates a sample of UdpClient class and binds it with the number of local port. Limits of admissible addresses are stored within the fields MaxPort and MinPort of IPEndPoint class. If an indicated port is busy, a SocketException exception is generated
UdpClient (IPEndPoint)	Creates a sample of UdpClient class on the basis of IPEndPoint. IPEndPoint encapsulates the data about an IP address and port number. In its turn an IP address can be classified as a IPAddress class sample.
UdpClient (Int32, AddressFamily)	Creates a sample of UdpClient class using AddressFamily enumeration and port number
UdpClient (String, Int32)	Creates a sample of UdpClient class on the basis of remote host name and port number. After using of this constructor we may not request a Connect() method, for it is called directly from the constructor.

Defining Information about the Connection

After creating an object UdpClient we can establish a connection for interacting with a remote host. In fact it is possible to define data about the remote host within some constructors UdpClient, or directly request a method Connect() or we can translate this data to the method Send() upon actual data transfer.

Overloaded methods Connect ():

Method signature	Description
Connect(IPEndPoint)	Sets up the data about a remote host using IPEndPoint class sample, which encapsulates information about an IP address and port number.
Connect(IPAddress, Int32)	Sets up the data about a remote host using an IP address and port number.
Connect(String, Int32)	Sets up the data about a remote host using host name and port number.

Example of establishing a connection using a IPEndPoint class sample:

```
UdpClient client = new UdpClient();
IPAddress address = IPAddress.Parse("192.168.1.51");
IPEndPoint endPoint = new IPEndPoint(address, 1234);
try
{
    client.Connect(endPoint);
}
catch(Exception ex)
{
    MessageBox.Show("Error" + ex.Message);
}
```

Example of establishing a connection using an IP address and port number:

```
UdpClient client = new UdpClient();
IPAddress address = IPAddress.Parse("192.168.1.51");
try
{
    client.Connect(address, 1234);
}
catch(Exception ex)
{
    MessageBox.Show("Error" + ex.Message);
}
```

Data Sending Using a UdpClient Object

Having created a UdpClient class sample and having set up the data about a remote host, we can proceed to data transmission. With this purpose we use a Send() method. It sends a datagram to a remote host. Since a UDP protocol doesn't not employ handshaking, there are no confirmations from the remote host. Just like Connect() method, Send() method exists in several overloaded variants, wherein all methods return a number of transmitted bytes:

Method signature	Description
Send (Byte[], Int32)	Sends UDP datagram to a remote host. First parameter is — a byte array defining a datagram, and second one is – a datagram length.
Send (Byte[], Int32, IPEndPoint)	Sends UDP datagram to a remote host. First parameter is – a byte array defining a datagram, second one is – a datagram length, and third one is – remote host parameters to which a datagram is sent
Send (Byte[], Int32, String, Int32)	Sends UDP datagram to a remote host. First parameter is – a byte array defining a datagram, second one is – a datagram length, and third one is – remote host name, fourth one is – port number.

Example of message sending using a Send() method without a Connect()method:

```
UdpClient client = new UdpClient();
byte[] bArray = Encoding.ASCII.GetBytes("test message");
try
{
    client.Send(bArray, bArray.Length,
           "remoHostNumberOne", 9001);
}

catch(Exception ex)
{
    MessageBox.Show("Error" + ex.Message);
}
```

Data Receiving Using a UdpClient Class

To receive data from a remote host we apply a Receive() method. This method has the following signature:

```
byte[] Receive(ref IPEndPoint remoteEP)
```

this method returns received values in a form of byte array. IPEndPoint class sample is used as an only parameter. It encapsulates the data about an IP address and remote host port. This method should be applied in a separate flow, for it requests availability of datagrams from a basic socket and blocks the flow until the program is received. If we call this method within the main flow, then program execution will be suspended.

After receiving the datagram this method returns the data into byte array after it deletes information and fills in a IPEndPoint class sample, which is transmitted as an only parameter.

Example of Receive() method use:

```
//variable LocalPort is preliminary initiated
UdpClient uClient = new UdpClient(LocalPort);
//variable ipEnd will be transmitted into
//a Receive method
IPEndPoint ipEnd = null;
byte[] responce = uClient.Receive(ref ipEnd);
//transformation into a string
string strResult = Encoding.Unicode.GetString(responce);
Console.WriteLine(strResult);
```

Closure of the Connection

While working with UdpClient class our last step is to call a method without parameters Close(). If there are any errors upon connection closure then we should generate a SocketException.

Use Case: Writing of a Chat Application

In order to illustrate a use of UdpClient class we can create a console application, which upon the startup requests the parameters for setting up a socket, and then it allows sending test data to another sample of this application, as a result working as the simplest chat. There are two flows within the application — one is for data reading, and another one is for sending. Data is received as a chain of bytes and then it transforms into a string, which is displayed on a screen. For a user's convenience the dialog partner's messages are displayed in other color.

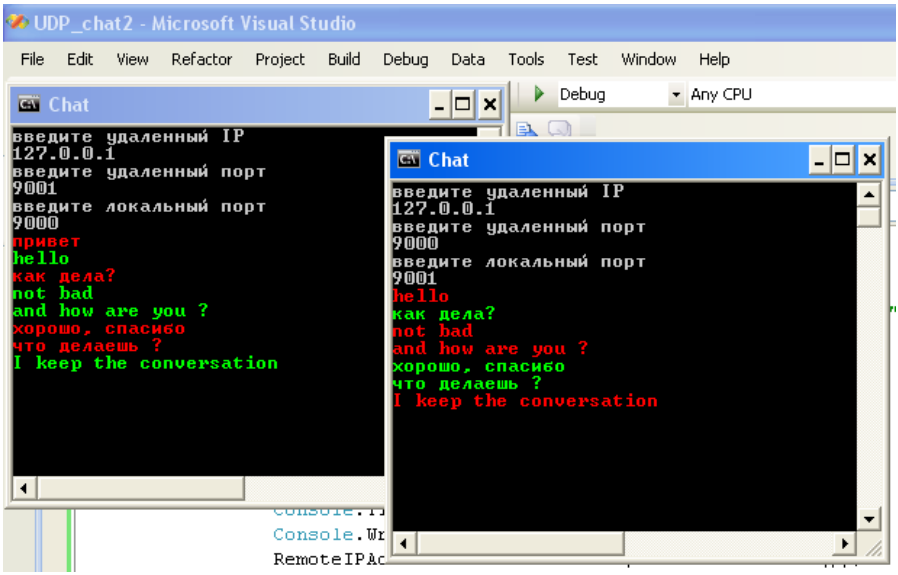
```
using System;
using System.Text;
using System.Net;
using System.Net.Sockets;
using System.Threading;
```

```
class Program
{
    //variables necessary for estabiliching
    //a connection:
    //remote host and ports - remote and local
    static int RemotePort;
    static int LocalPort;
    static IPAddress RemoteIPAddr;

    [STAThread]
    static void Main(string[] args)
    {
       try
       {
          Console.SetWindowSize(40, 20);
          Console.Title = "Chat";
          Console.WriteLine("input a remote IP");
          RemoteIPAddr = IPAddress.Parse(Console.
                  ReadLine());
          Console.WriteLine("input a remote port");
          RemotePort = Convert.ToInt32(Console.
                  ReadLine());
          Console.WriteLine("input a local port ");
          LocalPort = Convert.ToInt32(Console.ReadLine());
          //separate flow for reading in
          //ThreadFuncReceive method
          //this method calls Receive()UdpClient class,
          //which blocks the current flow, that is why
          //a separate flow is needed
          Thread thread = new Thread(
             new ThreadStart(ThreadFuncReceive)
          );
          //creating a background thread
          thread.IsBackground = true;
          //starting up a thread
          thread.Start();
          Console.ForegroundColor = ConsoleColor.Red;
```

```
                while (true)
            {
                SendData(Console.ReadLine());
            }
        }
        catch (FormatException formExc)
        {
            Console.WriteLine("Transformation is
                    impossible:" + formExc);
        }
        catch (Exception exc)
        {
            Console.WriteLine("Error : " + exc.Message);
        }
    }

static void ThreadFuncReceive()
{
    try
    {
        while (true)
        {
            //conencting a colac host
            UdpClient uClient = new UdpClient(LocalPort);
            IPEndPoint ipEnd = null;
            //receiving a datagram
            byte[] responce = uClient.Receive(ref ipEnd);
            //transformation into a string
            string strResult = Encoding.Unicode.
                    GetString(responce);
            Console.ForegroundColor = ConsoleColor.Green;
            //display on a screen
            Console.WriteLine(strResult);
            Console.ForegroundColor = ConsoleColor.Red;
                    uClient.Close();
        }
    }
```

```
    catch (SocketException sockEx)
    {
       Console.WriteLine("Socket error: " +
                sockEx.Message);
    }
    catch (Exception ex)
    {
       Console.WriteLine("Error : " + ex.Message);
    }
}

static void SendData(string datagramm)
{
    UdpClient uClient = new UdpClient();
    //connecting a remote host
    IPEndPoint ipEnd = new IPEndPoint(RemoteIPAddr,
                      RemotePort);
    try
    {
       byte[] bytes = Encoding.Unicode.GetBytes(datagramm);
       uClient.Send(bytes, bytes.Length, ipEnd);
    }
    catch(SocketException sockEx)
    {
       Console.WriteLine("Socket error: " +
                     sockEx.Message);
    }
    catch(Exception ex)
    {
        Console.WriteLine("Error: " + ex.Message);
    }
    finally
    {
       //closure of a UdpClient class sample
       uClient.Close();
    }
  }
}
```

Example of Chat Application Functioning



5. What is Unicast?

Routing scheme unicast implies that a package is directly sent to only one network device from the other network device.

This routing scheme can be used either through TCP, or UDP-protocol.

The sense is that the data is transmitted to one recipient. But in its turn it does not mean that they are always sent to the same computer, because one computer can have several network devices and it means that it can have several IP addresses and even be within different sub-networks. It can also change an IP address in different moments.

We should also understand that using a routing scheme unicast while sending the same package to several recipients you will have to manually send the data to each recipient separately.

6. What is Broadcast?

Broadcast (type of broadcasting channel) routing scheme is a special subtype of network interaction that implies that after package receipt a router transmits it to all computers that correspond some address space (subcollection). For example, to all computers of the sub-network to all available computers.

However, in spite of the fact that broadcast routing scheme can seem attractive it greatly consumes network resources, which means it should be used as narrow as possible, because a datagram is sent not only to the computers that wait for the message, but to the rest of them as well. Thus, many types of network attacks, for example a broadcast storm uses a broadcast channel. That is why at present many routers block broadcast queries for security reasons. And if you might decide to develop a chat using a broadcast channel it will not work in the framework of LAN, most probably the package will not be sent.

In order to send a broadcast-package we should create a UDP-socket and send a datagram using a broadcast-address. To define a broadcast-address we should apply a bitwise AND operation to the first sub-network address (for example, the first address in the sub-network with a mask 255.255.0.0 would be 192.168.0.0) and to the supplement subnet mask (0.0.255.255). Thus, we have that 192.168.0.0 | 0.0.255.255 = 192.168.255.255.

Or in the binary way it looks as follows:

```
11000000101010000000000000000000
OR
00000000000000001111111111111111
-----------------------------------------------
= 11000000101010001111111111111111
```

After we have defined a broadcast address we create a socket and send a query.

```
Socket soc = new Socket(
    AddressFamily.InterNetwork,
    SocketType.Dgram,
    ProtocolType.Udp);
IPEndPoint ipep = new IPEndPoint(IPAddress.
    Parse("192.168.255.255"), 1234);
soc.Connect(ipep);
string message = "Hello network!!!";
soc.Send(Encoding.Default.GetBytes(message));
```

7. What is Multicast?

Multicast — is a network addressing technology upon which messages are sent to a group of recipients employing the strategy of the most efficient way of data delivery: message is only once sent through one network segment; data is copied when a direction to the recipients splits.

As a rule, the word multicast is used as a reference to IP Multicast — method of sending IP datagrams («popularly» called «packages») to a group recipients per one data transmission session.

According to the standard RFC 3171 addresses within the range 224.0.0.0-239.255.255.255 are used by the IPv4 protocol as multicast-addresses and form a «D» class of addresses. As soon as a sender transmits a datagram to some multicast-address, router immediately redirects it to all the recipients that have registered for receiving information from this multicast-address.

In order to fulfill data transmission for the multicast-address by means of .NET Framework, at first we should create a common UDP-socket.

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

The we need to install an option MulticastTimeTolive, which influences a package lifetime. If we set it up into a value

1, the package will not exceed LAN limits. If we set it up into a value other than 1, the datagram will several routers.

Installing of this option is set up by means of calling a component method SetSocketOption of Socket class.

```
someMulticastSocket.SetSocketOption(
    SocketOptionLevel.IP,
    SocketOptionName.MulticastTimeToLive, 2);
```

The next step is creating an IPAddress class object describing some multicast-address we have chosen. We should register this address for a newly created socket by means of calling a method SetSocketOption, create a connection endpoint on the basis of this address and connect our socket with the endpoint.

```
IPAddress dest = IPAddress.Parse("224.5.5.5");
someMulticastSocket.SetSocketOption(
    SocketOptionLevel.IP,
    SocketOptionName.AddMembership,
    new MulticastOption(dest));
IPEndPoint ipep = new IPEndPoint(dest, 4567);
someMulticastSocket.Connect(ipep);
```

Now we can realize transmitting multicast-message. Therefore, we can proceed to calling the method Send for the socket.

```
string message = "Hello network!!!";
someMulticastSocket.Send(Encoding.Default.
    GetBytes(message));
```

8. Example of Realizing a Musticast Application?

We shall study use of the routing scheme multicast on the example of window application that realizes mailing of information input by a user to all registered users (meaning those users who have a running client application) and represents a message board. We realize a client-server model with the view of unification of user interface.

Client application interface will be represented in a form of a window containing a text field in multiline mode and its enabled property is set up into a value false, so that a user couldn't edit incoming messages.

Multicast — is a network addressing technology upon which messages are sent to a group of recipients employing the strategy of the most efficient way of data delivery: message is only once sent through one network segment; data is copied when a direction to the recipients splits.

Server part interface will look as follows with the only exception — a field should allow editing so that a user could change messages on a message board.

Create two window projects in one window solution and edit their interface as it is shown above.

Let a text field in server application be called textBox1 (default name), and the name outputData for a client one.

Server Part of an Application

We should declare a static variable that is to store a sent message as well as integer variable, which should define an in-

terval that will be used for actual data that is sent to recipients. Further it is necessary to describe a thread method that is to deal with information sending.

```
static string message = "Hello network!!!";
static int Interval = 1000;
static void multicastSend()
{
    while (true)
    {
        Thread.Sleep(Interval);
        Socket soc = new Socket(
               AddressFamily.InterNetwork,
               SocketType.Dgram,
               ProtocolType.Udp);
        soc.SetSocketOption(
               SocketOptionLevel.IP,
               SocketOptionName.MulticastTimeToLive, 2);
        IPAddress dest = IPAddress.Parse("224.5.5.5");
        soc.SetSocketOption(
               SocketOptionLevel.IP,
               SocketOptionName.AddMembership,
               new MulticastOption(dest));
        IPEndPoint ipep = new IPEndPoint(dest, 4567);
        soc.Connect(ipep);
        soc.Send(Encoding.Default.GetBytes(message));
        soc.Close();
    }
}
```

As you can see a flow method we have revealed an endless loop in which we at first stop thread execution for a particular period of time that was defined by Interval variable, and then we create a socket and register a broadcasting address within it. After that we initiate the connection with the broadcast

multicast-channel, and then send information. After that we close the socket.

After declaration of a thread method we should create an object of thread class and run in within the constructor. Do not forget to set up a field IsBackground of a thread class true, so that there were no problems with its closure in future.

```
Thread Sender = new Thread(new ThreadStart(multicastSend));
public Form1()
{
    InitializeComponent();
    Sender.IsBackground = true;
    Sender.Start();
}
```

The last step is handling an event TextChanged of a text field to update information stored within a variable "message".

```
private void textBox1_TextChanged(object sender, EventArgs e)
{
    message = textBox1.Text;
}
```

Client Part

At first it is necessary to declare a delegate to have an opportunity to update information within the text field without an error of inter-thread access. Since a text field was created within a thread different from the one it is updated in, there are might be errors related to resource blocking.

```
delegate void AppendText(string text);
void AppendTextProc(string text)
```

```
{
    dataOutput.Text = text;
}

void Listner()
{
    while (true)
    {
        Socket soc = new Socket(
        AddressFamily.InterNetwork,
        SocketType.Dgram, ProtocolType.Udp);
        IPEndPoint ipep = new IPEndPoint(IPAddress.
                Any, 4567);
        soc.Bind(ipep);
        IPAddress ip = IPAddress.Parse("224.5.5.5");
        soc.SetSocketOption(
                SocketOptionLevel.IP,
                SocketOptionName.AddMembership,
                new MulticastOption(ip,
                IPAddress.Any));
        byte[] buff = new byte[1024];
        soc.Receive(buff);
        this.Invoke(
                new AppendText(AppendTextProc),
                Encoding.Default.GetString(buff));
        soc.Close();
    }
}
```

As you can see in a thread method that will execute information receipt along the multicast-channel, we have initiated an endless loop. We create a common UDP socket within this loop for which we register a multicast ip-address. After that we put a socket into a pending information mode, and as soon as it was obtained we call the method that will update information in a text field.

Within a class we should declare an object of a thread class, initialize it and run a thread by means of requesting a Start method.

```
Thread listen;
public Form1()
{
    InitializeComponent();
    listen = new Thread(new ThreadStart(Listner));
    listen.IsBackground = true;
    listen.Start();
}
```

Upon execution of the above described actions we can run bot projects and look at the run application.

9. Home Assignment

Using a routing scheme multicast realize a serverless chat graphic user interface. Meaning the chat that does not need a server and which would send messages to those computers that await it. You should also foresee же a possibility of online users monitoring.