TCPConnection

Software Documentation

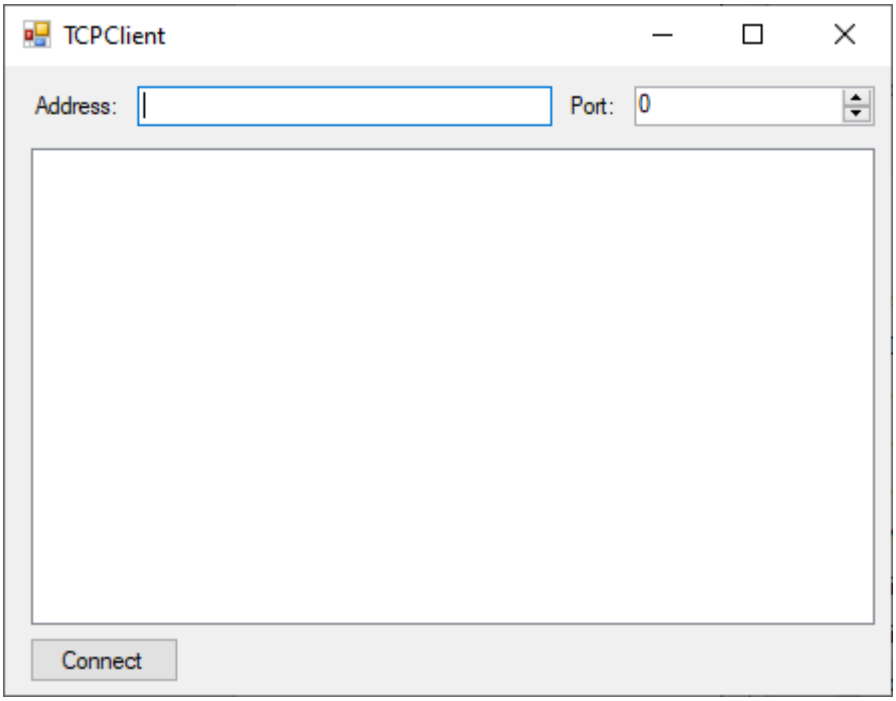
Author: matiwa

Table of contents

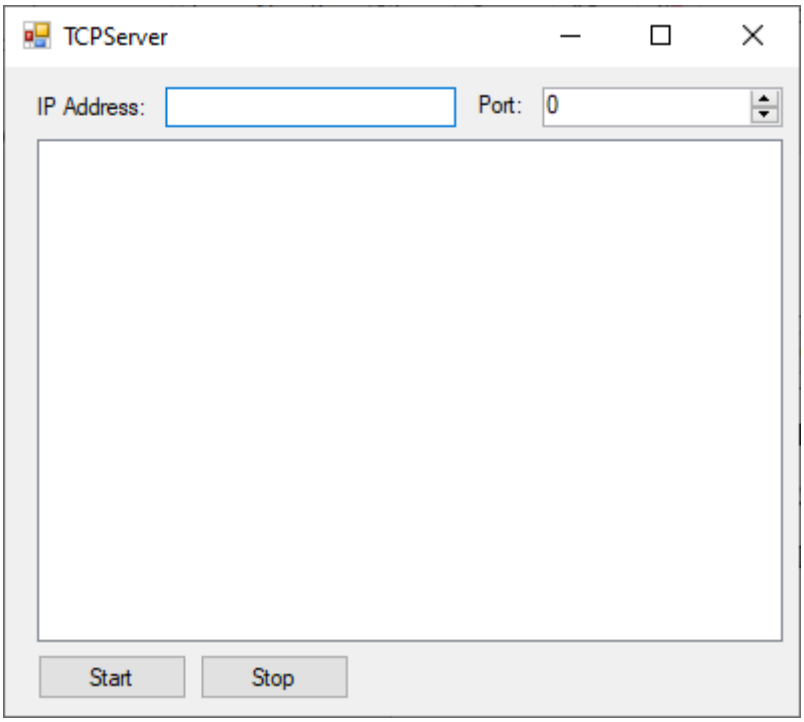
Introduction

This software documentation includes: description of the application's operation, what is needed for use, algorithms used, interface description and source code description. These applications are used to establish TCP connections with each other according to the principles of the client-server architecture. The concept consists of two projects, TCPClient and TCPServer, and additionally there is the AsynchronousTCPServer version.

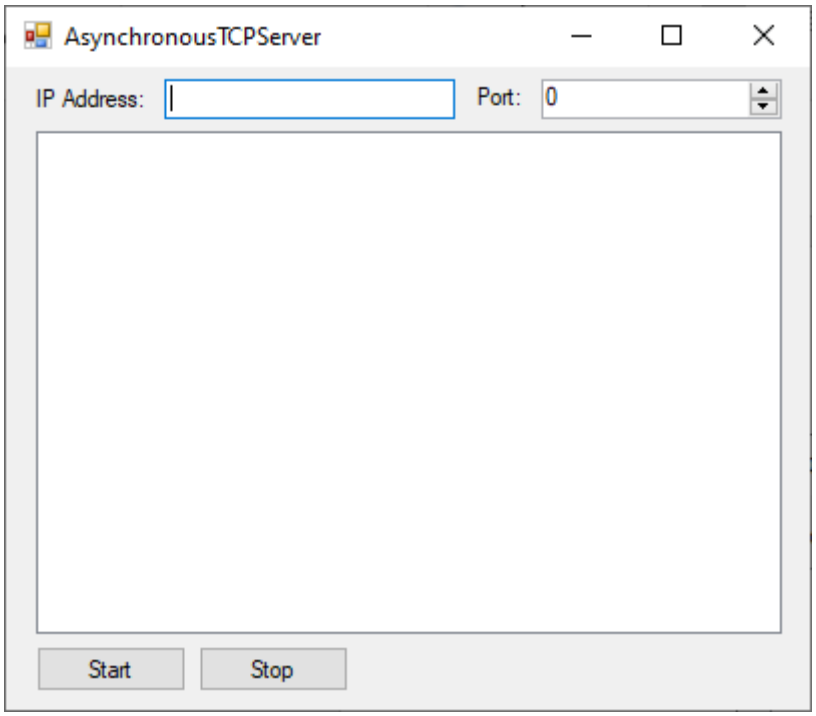
Describing of the application’s operation



Drawing 1: The beginning of the application's operation TCPClient [own study]

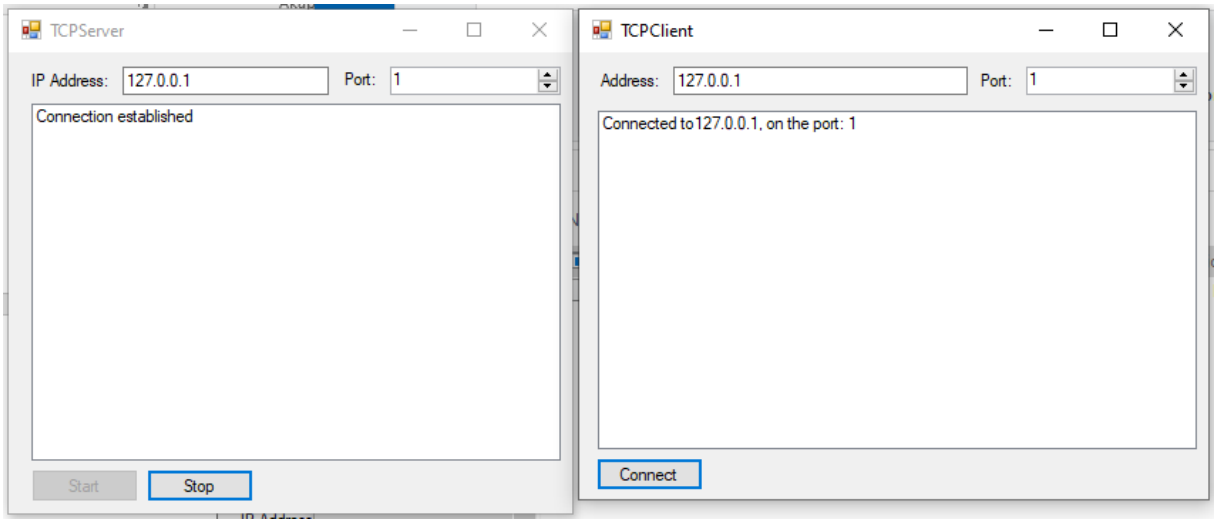


Drawing 2: The beginning of the application's operation TCPServer [own study]

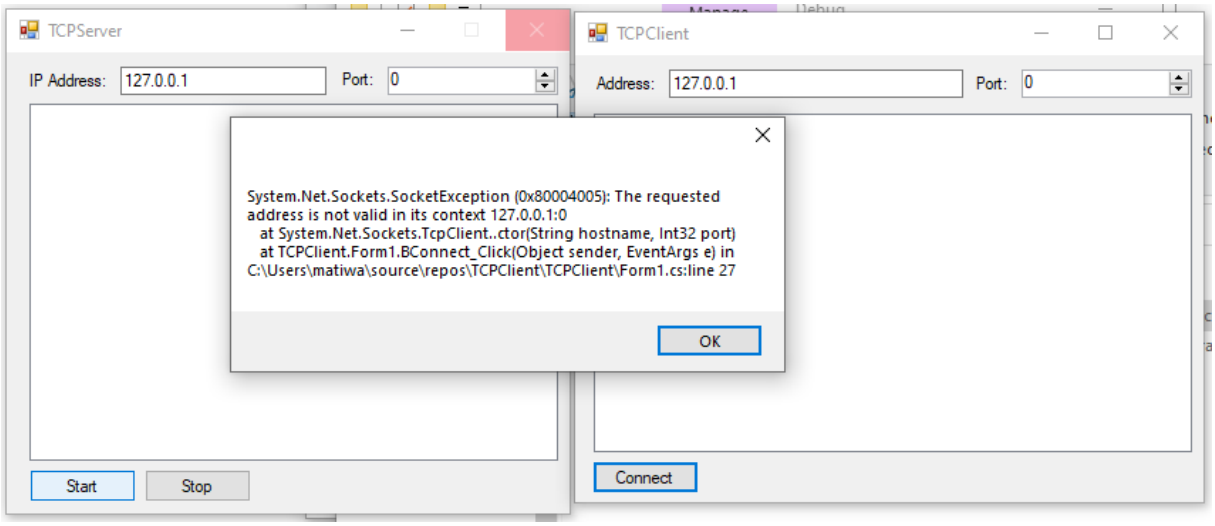


Drawing 3: The beginning of the application's AsynchronousTCPServer [own study]

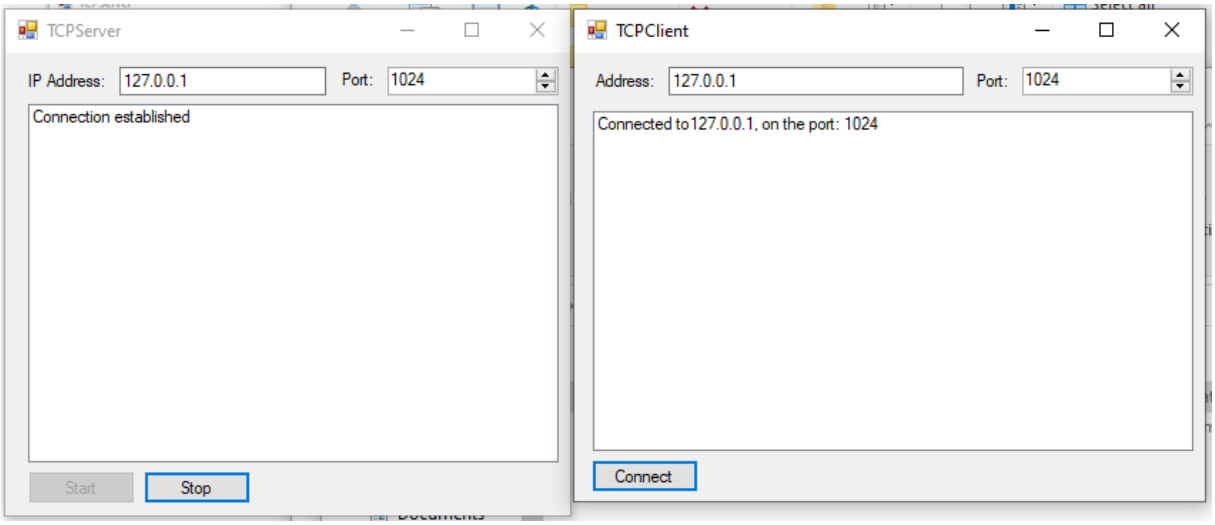
The operation of the structure for a synchronous server is shown in the figures below.



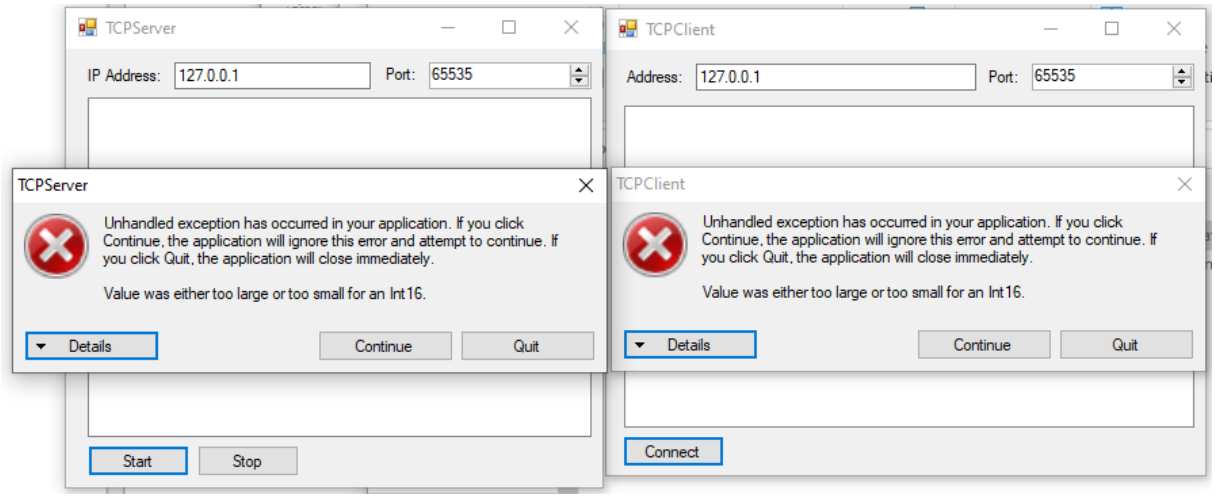
Drawing 4: Operation of the port structure for network services [own study]



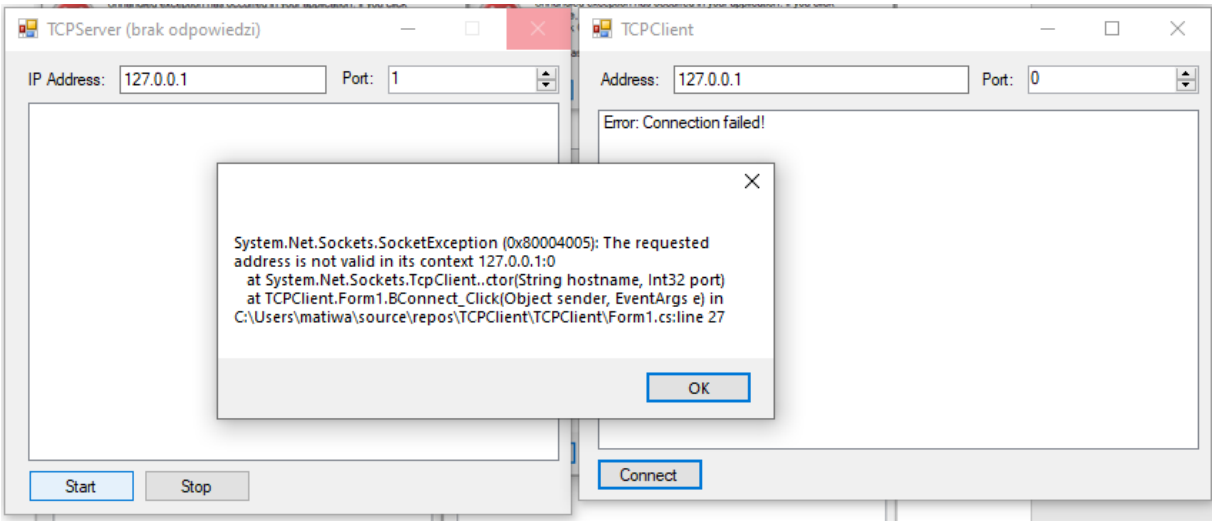
Drawing 5: The operation of the structure for port 0 [own study]



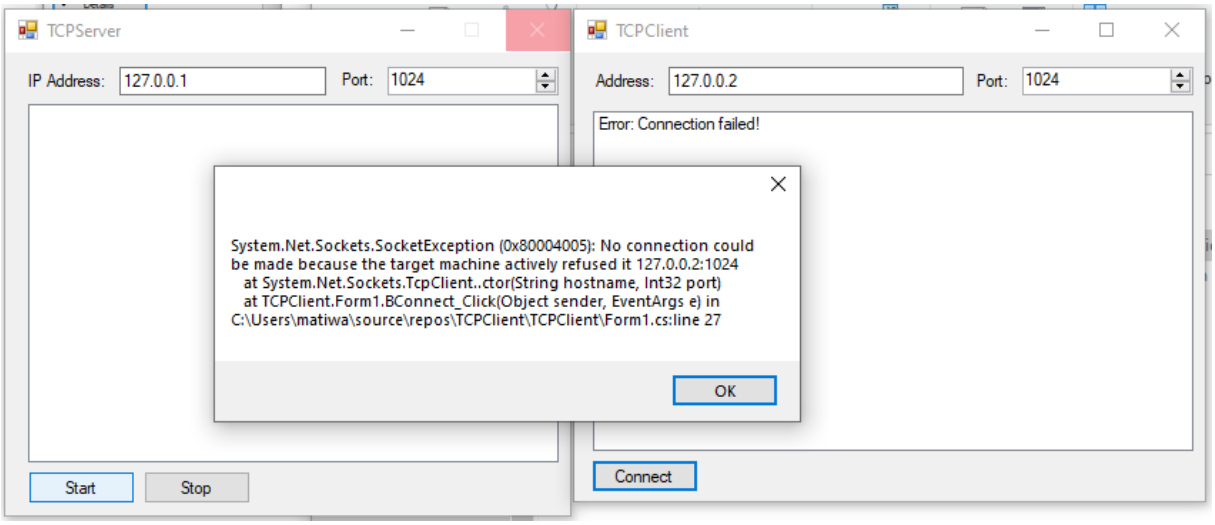
Drawing 6: Structure activity for the minimum port for the user app [own study]



Drawing 7: Structure activity for a port in scope for the application [own study]



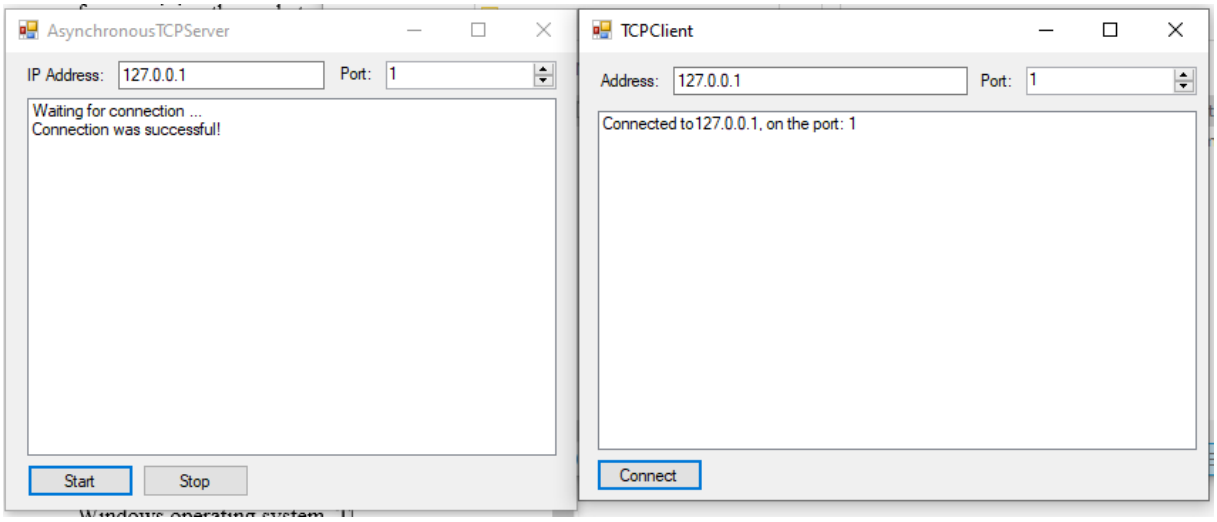
Drawing 8: Structure behavior for incompatible ports [own study]



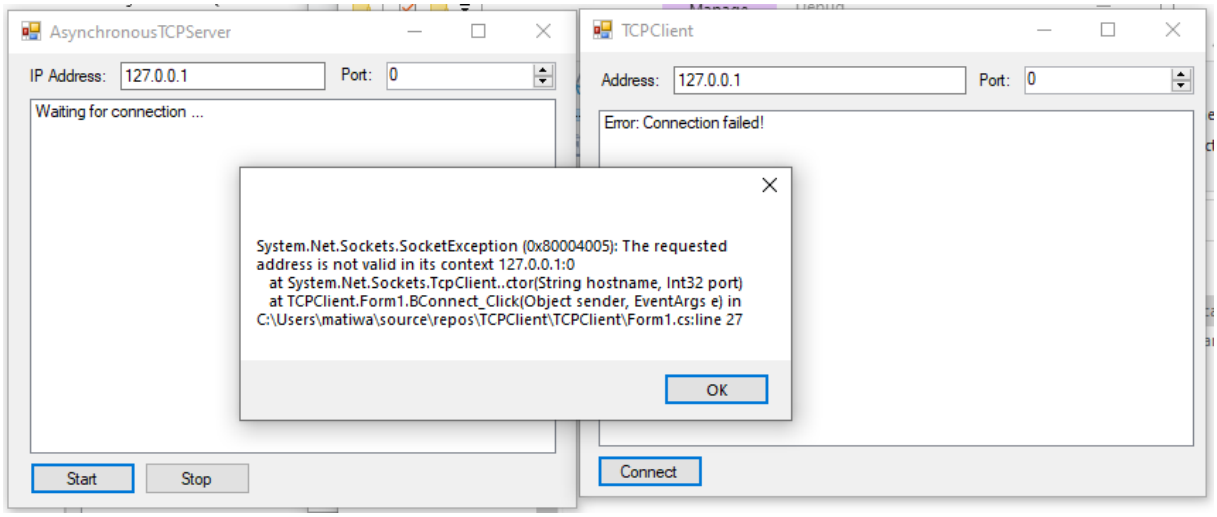
Drawing 9: Structure behavior for incompatible IP address [own study]

After starting the server program, the user enters the IP address and sets the port on which to listen. Then presses the Start button. Then the user goes to the client application. Enter the IP address (127.0.0.1 for localhost), set the same port that the server is listening on. Finally, he confirms that he wants to connect by pressing the Connect button. If everything is correct, the connection accepted message appears on the server side, and the client displays a message that it has connected to the given server's IP address on the specified port. There are comments about the port. Although ports 0-1023 are reserved for network services and ports 1024-65535 are reserved for user applications, a connection is usually made for a port greater than 1. In the event of an IP address, port, or port 0 conflict, a message from the client and the server application freezes. Unhandled exceptions appear on both sides in the case of port 65535 - lower ones can also have similar - it's hard to tell from which. The problem also arises for the maximum port. These shortcomings serve for possible work, but are not necessary to show the essence of the guiding structure. There are certainly some bugs that the developer didn't discover while working on the apps.

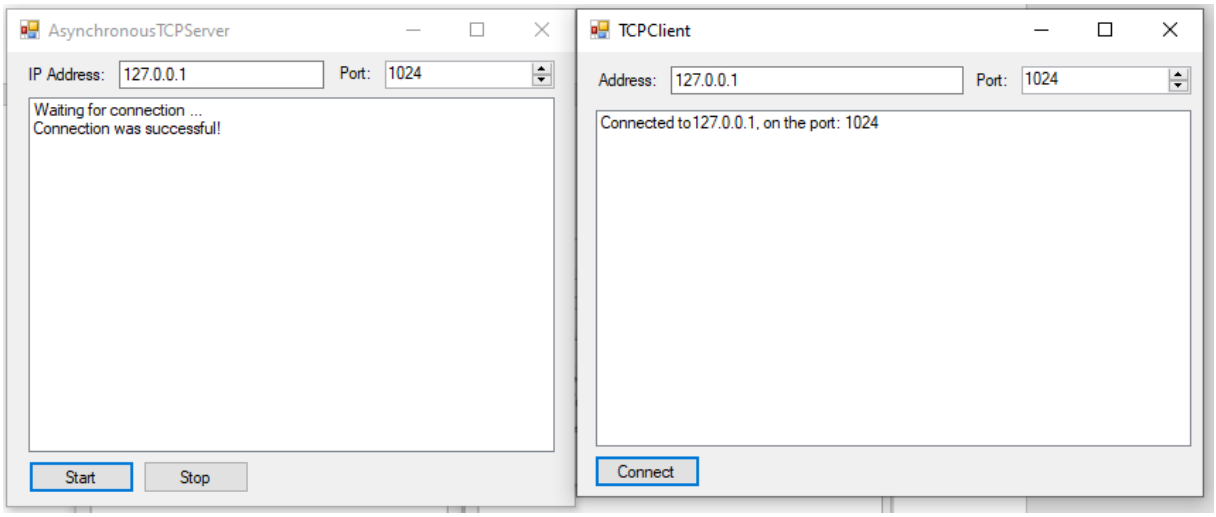
The operation of the structure for an asynchronous server is shown in the following figures.



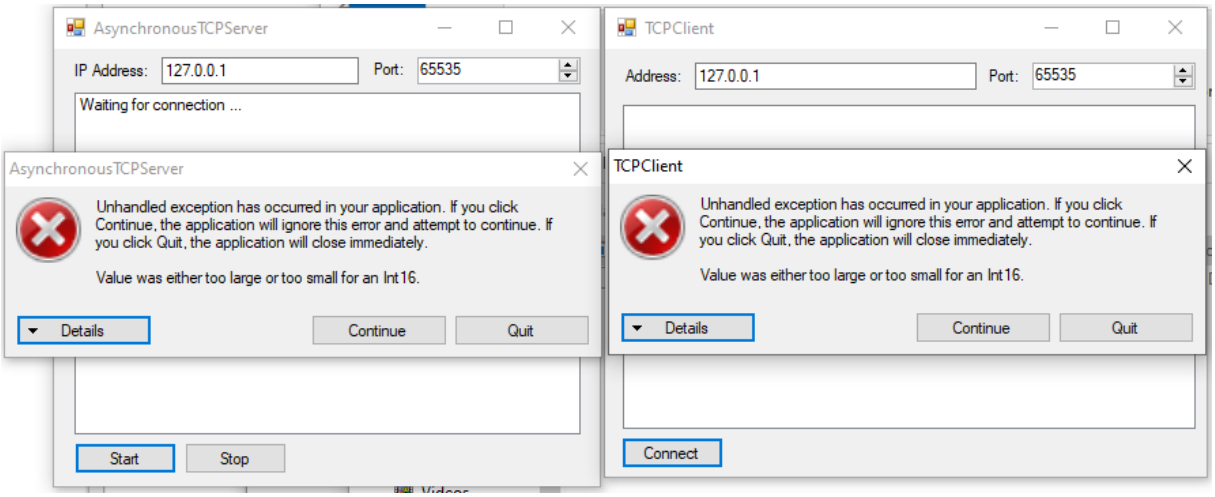
Drawing 10: Operation of the port structure for network services [own study]



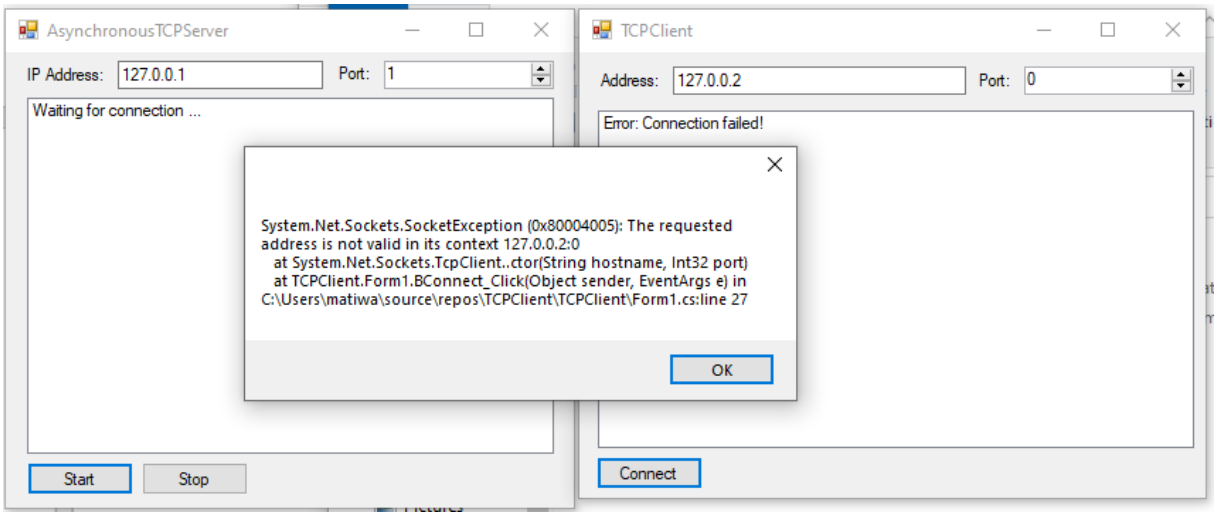
Drawing 11: The operation of the structure for port 0 [own study]



Drawing 12: Structure activity for the minimum port for the user app [own study]



Drawing 13: Structure activity for a port in scope for the application [own study]



Drawing 14: Structure behavior for incompatible ports and IP address [own study]

The work of an asynchronous server with a client differs from a synchronous server in that there are messages about waiting for a connection, and when it does, there is a success message. This is at first sight. The asynchronous version does not block the user interface. A synchronous TCP server that is waiting for a connection. Upon incoming call, disconnects the client and enters Stop mode. It works by locking the socket to a connection. The synchronous version runs on the main thread, while the asynchronous version creates a separate thread waiting for client connections. The asynchronous version immediately ends (returns control to the calling thread) and runs moves to a separate thread. Controls the execution of the asynchronous thread.

What is needed for use?

The application does not require installation. It only needs the Windows operating system.

Algorithm used

The basic form of the algorithm can be deduced from the previous section. It only needs the Windows operating system. These applications are used to establish TCP connections with each other in accordance with the principles of the client-server architecture.

Client / server, client-server model - the architecture of the computer system, in particular software, enabling the division of tasks (roles). This is done by establishing that the server is providing services to clients that make service requests to the server.

The basic, most common servers operating on the basis of the client-server architecture are: e-mail server, web server, file server, application server. Typically multiple clients can use one server. A single client, in general, may use multiple servers simultaneously. According to the client-server scheme, most of the currently used database management systems also operate.

In some simplification, without going into technical details of the implementation, the method of communication according to the client-server architecture can be characterized by defining tasks (assigning roles) to both parties and defining their modes of operation.

Client side - this is the party requesting access to the given service or resource.

Client work mode:

- active,
- sends the request to the server,
- waiting for a response from the server.

Server side - this is the site that provides a service or resource.

Server operation mode:

- passive,

- waiting for requests from customers,
- upon receiving the request, it processes it and then sends a response.

Due to the division of performed tasks, the following types of client-server architecture are distinguished:

- two-tier architecture - data processing and storage takes place in one module
- three-tier architecture - data processing and storage takes place in two separate modules
- multi-tier architecture - processing, storage and other data operations take place in many separate modules.

The connection between the client and the server is described using specific communication protocols. The most common is TCP / IP. In most cases, communication is based on a pattern where the client connects to the server. It then sends a request in the specified format to the server and waits for a response. The server is waiting for the clients all the time and when it receives the request, it processes them and sends a response. In the OSI model, communication between the parties takes place at the application layer.

P2P is a type of different architecture where each host can act as both a client and a server.

Benefits:

- All information is stored on the server, so better data security is possible. The server can decide who has the right to read and change the data.
- There are many advanced technologies supporting the operation, safety and usability of this type of solution.

Disadvantages:

- A large number of clients trying to receive data from one server causes various types of problems related to the bandwidth of the link and the technical ability to process customer requests.
- While the server is down, data access is completely impossible.
- To run a server unit capable of serving a large number of clients, special software and hardware are required, which are not found in most home computers.

The closest example is the organization of access to Internet resources, where:

- the role of the server is played by a web server,
- the role of the client is played by a web browser.

When browsing websites, the user's computer is the client, and the computers that run databases and other applications needed to handle the connection are the server. When the browser requests a page, the server searches for the relevant information in the database, converts it into a website, and then sends it to the client. [1]

TCP (Transmission Control Protocol) - a connection-based, reliable, streaming communication protocol used to transfer data between processes running on different machines, which is part of the currently widely used TCP / IP stack (it uses the IP protocol services to send and receive data as well as fragmentation when necessary).

TCP is a protocol that works in a client-server mode. The server waits for a connection on the specified port to be established. The client initiates a connection to the server.

Unlike UDP, TCP guarantees the higher communication layers to deliver all packets in full sequence and without duplicates. This provides a reliable connection at the cost of more header overhead and more packets transmitted. Although the protocol defines the TCP packet, from the point of view of the upper software layer, the data flowing over the TCP connection should be treated as a sequence of octets. In particular, one call to an application programming interface function (e.g., send ()) does not need to be answered with sending one packet. Data from one call can be split into several packets or vice versa - data from several calls can be combined and sent as one packet (thanks to Nagle's algorithm). Also functions receiving data (recv ()) in practice receive not specific packets, but the contents of the TCP / IP stack buffer, which is successively filled with data from incoming packets.

In the transmission control protocol, a procedure called a three-way handshake is used to establish a connection between two hosts. Normally, it is started when host A wants to establish a connection with host B. It looks like this:

- Host A sends the SYN segment to host B along with the information about the lower value of the sequence numbers used for the numbering of the segments sent by host A (e.g., 100) and then enters the SYN-SENT state.
- Host B, after receiving the SYN segment, goes into the SYN-RECEIVED state and, if it also wants to establish a connection, sends host A a SYN segment with information about the lower value of the sequence numbers used for numbering the segments sent by host B (e.g. 300) and the ACK segment with the sequence number field set to a value one greater than the sequence field value of Host A's first SYN segment, i.e. 101.
- After receiving the SYN and ACK segments from Host B, it enters the ESTABLISHED state and sends it an ACK segment acknowledging receipt of the SYN segment (sequence number set to 301).
- Host B receives the ACK segment and enters the ESTABLISHED state.
- Host A can now start transferring data.

If the receiving host is unwilling or unable to answer the connection, it should reply with a packet with the RST (reset) flag set.

TCP uses checksums and sequential packet numbers to verify sending and receiving. The receiver acknowledges receipt of packets with the specified sequence numbers by setting the ACK flag. Missing packets are retransmitted. The host that receives TCP packets defragments them and orders them by sequence number to pass the complete composite segment to the upper layers of the OSI model.

The correct termination of the connection can be initiated by either party. It consists in sending the package with the FIN (finished) flag set. Such a packet requires confirmation with

the ACK flag. Most often, after receiving a packet with the FIN flag, the other party also ends the communication by sending a packet with the FIN and ACK flags. Such a package also requires confirmation by sending ACK.

It is also allowed to break the connection in an emergency by sending the packet with the RST flag (reset). Such a package does not require confirmation.

A TCP connection can be in one of the following states:

LISTEN

Ready for the server to accept a connection on the specified port.

SYN-SENT

The first phase of establishing a connection by the client. A packet with the SYN flag was sent. Waiting for SYN + ACK packet.

SYN-RECEIVED

SYN packet received, SYN + ACK sent. Waiting for ACK. The connection is half-open.

ESTABLISHED

The connection has been successfully established. The transmission is probably in progress.

FIN-WAIT-1

FIN packet sent. Data can still be received, but sending is no longer possible.

FIN-WAIT-2

Own FIN package confirmation received. It is waiting for a FIN from the server.

CLOSE-WAIT

FIN packet received, ACK sent. Waiting to submit your own FIN packet (when the application has finished broadcasting).

CLOSING

The connection is closed.

LAST-ACK

FIN received and sent. Waiting for the last ACK packet.

TIME-WAIT

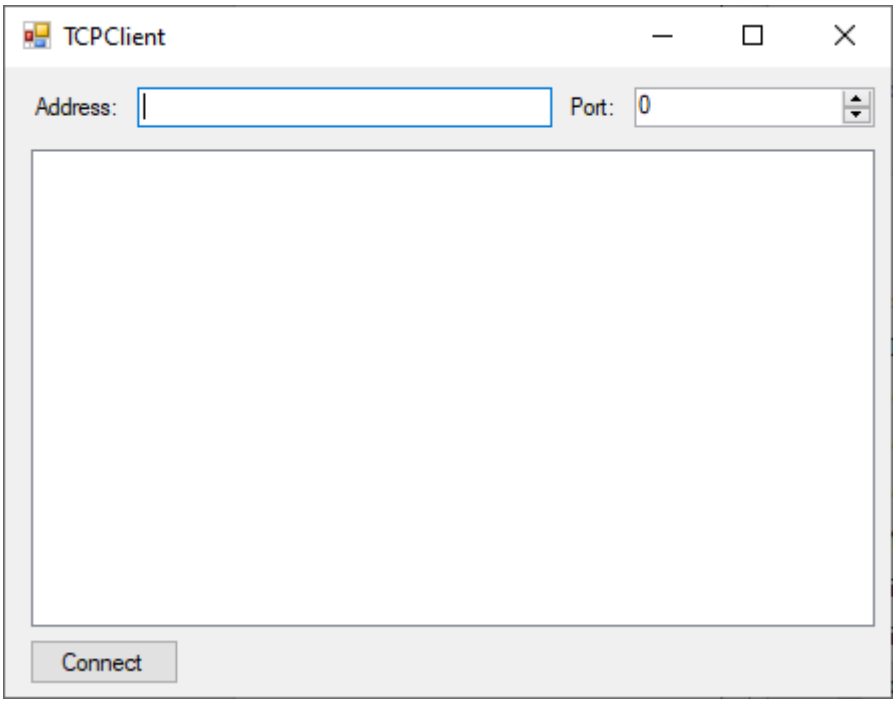
Waiting to make sure the other party has received a disconnect confirmation. In accordance with RFC 793 ↓, the connection may be in the TIME-WAIT state for a maximum of 4 minutes.

CLOSED

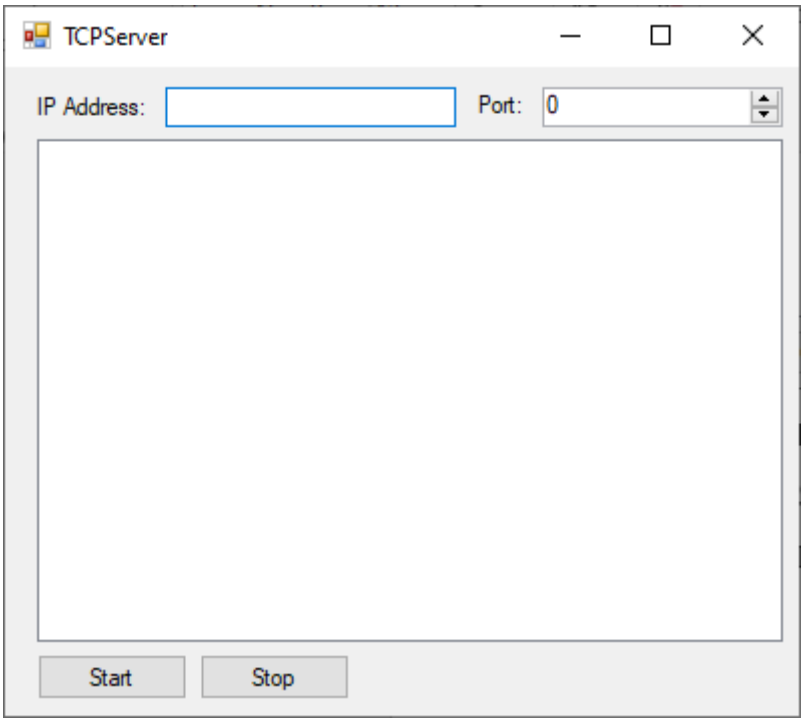
The connection is closed.

Applications in which the advantages of the transmission control protocol outweigh the disadvantages (higher cost of maintaining a TCP session by the network stack) include programs using application layer protocols: HTTP, SSH, FTP, SMTP / POP3 and IMAP4.[2]

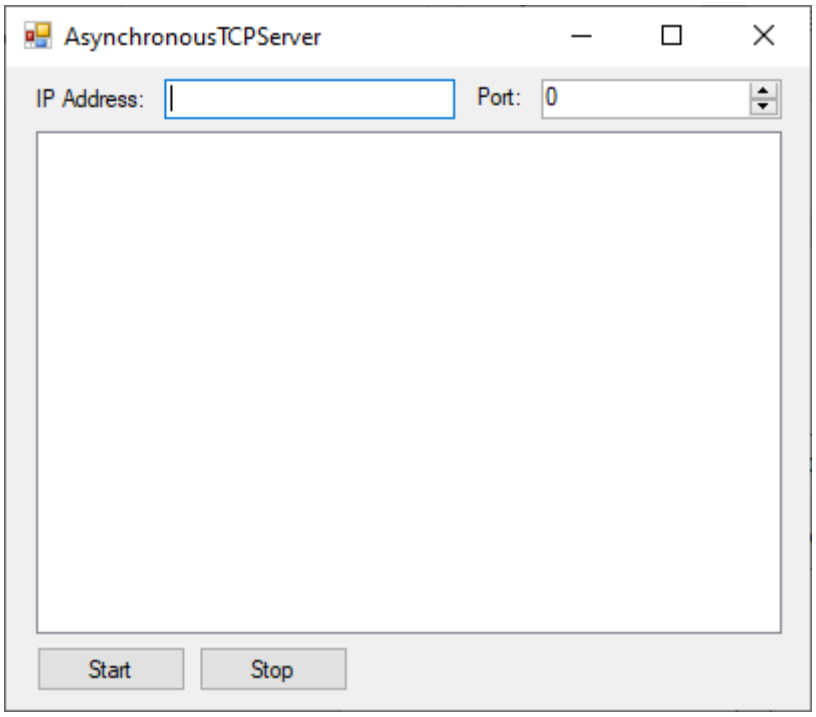
Interface description



Drawing 15: TCPClient graphical interface [own study]



Drawing 16: TCPServer graphical interface [own study]



Drawing 17: AsynchronousTCPServer graphical interface [own study]

The interfaces are typical for a Windows Forms Application. There are essential components: listbox, textbox, labels, numericupdowns and buttons.

Source code description

The projects were made in the C# programming language, in the Visual Studio Community 2017 programming environment. All work was done on the Windows 10 operating system. The TCPClient applications source code looks like this.

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

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

        private void BConnect_Click(object sender, EventArgs e)
        {
            listBox1.Items.Clear();
            string host = textBox1.Text;
            int port = System.Convert.ToInt16(numericUpDown1.Value);
            try
            {
                TcpClient klient = new TcpClient(host, port);
                listBox1.Items.Add("Connected to" + host + ", on the port: " + port);
                klient.Close();
            }
            catch (Exception ex)
            {
                listBox1.Items.Add("Error: Connection failed!");
                MessageBox.Show(ex.ToString());
            }
        }
    }
}
                                            }
```

Listing 1: TCPClient source code [own study]

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

```
namespace TCPServer
{
    public partial class Form1 : Form
    {
        private TcpListener serwer;
        private TcpClient klient;

        public Form1()
        {
            InitializeComponent();
        }

        private void BStart_Click(object sender, EventArgs e)
        {
            IPAddress adresIP;
            try
            {
                adresIP = IPAddress.Parse(textBox1.Text);
            }
            catch
            {
                MessageBox.Show("Wrong IP address format!", "Error"); textBox1.Text =
String.Empty;
                return;
            }
            int port = System.Convert.ToInt16(numericUpDown1.Value);
            try
            {
                serwer = new TcpListener(adresIP, port);
                serwer.Start();
                klient = serwer.AcceptTcpClient();
                listBox1.Items.Add("Connection established");
                BStart.Enabled = false;
                BStop.Enabled = true;
                klient.Close();
                serwer.Stop();
            }
            catch (Exception ex)
            {
                listBox1.Items.Add("Server initialization error!");
                MessageBox.Show(ex.ToString(), "Error");
            }
        }

        private void BStop_Click(object sender, EventArgs e)
        {
            serwer.Stop();
            klient.Close();
            listBox1.Items.Add("The server has ended ...");
            BStart.Enabled = true;
            BStop.Enabled = false;
        }
    }
}
```

Listing 2: TCPServer source code [own study]

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
```

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

namespace AsynchronousTCPServer
{
    public partial class Form1 : Form
    {
        private TcpListener serwer;
        private TcpClient klient;

        public Form1()
        {
            InitializeComponent();
        }

        private void BStart_Click(object sender, EventArgs e)
        {
            listBox1.Items.Add("Waiting for connection ...");
            IPAddress adresIP;
            try
            {
                adresIP = IPAddress.Parse(textBox1.Text);
            }
            catch
            {
                MessageBox.Show("Wrong IP address format!", "Error");
                textBox1.Text = String.Empty;
                return;
            }
            int port = System.Convert.ToInt16(numericUpDown1.Value);
            try
            {
                serwer = new TcpListener(adresIP, port);
                serwer.Start();
                serwer.BeginAcceptTcpClient(new
AsyncCallback(AcceptTcpClientCallback),
                serwer);
            }
            catch (Exception ex)
            {
                listBox1.Items.Add("Error: " + ex.Message);
            }
        }

        private void BStop_Click(object sender, EventArgs e)
        {
            if(serwer != null) serwer.Stop();
        }

        private void AcceptTcpClientCallback(IAsyncResult asyncResult)
        {
            TcpListener s = (TcpListener)asyncResult.AsyncState;
            klient = s.EndAcceptTcpClient(asyncResult);
            SetListBoxText("Connection was successful!");
            klient.Close();
            serwer.Stop();
        }

        private delegate void SetTextCallBack(string tekst);
        private void SetListBoxText(string tekst)
```

```
        {
            if (listBox1.InvokeRequired)
            {
                SetTextCallBack f = new SetTextCallBack(SetListBoxText);
                this.Invoke(f, new object[] { tekst });
            }
            else
            {
                listBox1.Items.Add(tekst);
            }
        }
    }
}
```

Listing 3: AsynchronousTCPServer source code [own study]

List of drawings

List of listings

Bibliography

[1] https://pl.wikipedia.org/wiki/Klient-serwer
[2] https://pl.wikipedia.org/wiki/Protok%C3%B3%C5%82_sterowania_transmisj%C4%85