Ministerul Educației al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare Informatică și Microelectronică

Departamentul Ingineria Software și Automatică

RAPORT

Lucrare de laborator nr. 6 la Programarea în Rețea

Tema: Aplicație client-server: Sockets API

„FileTransfer”

Efectuat st. gr. TI-142:

Chifa Vladislav

Verificat lect. asistent.:

Ostapenco Stepan.

Scopul lucrării:

Porturi şi socket-uri. Operații tip pentru conexiuni prin socket pentru client. Operații tip pentru conexiuni prin socket pentru server.

Obiectiv:

Proiectarea şi realizarea unui protocol de transfer date(mesaje) , utilizînd protocolul de nivel de transfer(TCP).

înțelegerea mecanismului de comunicare în rețea în prisma conceptului fundamental numit socket, studiul operațiilor primare ce compun un API bazat pe socket; obiectivul specific constă în elaborarea unei aplicații client-server şi descrierea structurată a protocolului ce defineşte interacțiunea dintre componentele distribuite ale sistemului.

Link la repozitoriu: https://github.com/tarakan-x/PR_Lab

Protocolul TCP

Transmission Control Protocol (sau **TCP**, în traducere liberă din engleză *Protocolul de Control al Transmisiei*) este un protocol folosit de obicei de aplicații care au nevoie de confirmare de primire a datelor. Efectuează o conectare virtuală full duplex între două puncte terminale, fiecare punct fiind definit de către o adresă IP și de către un port TCP.

Transmission Control Protocol (TCP) este unul dintre protocoalele de bază ale suitei de protocoale Internet. TCP este unul dintre cele două componente originale ale suitei (celalalt fiind Protocolul Internet, sau IP), astfel încât întreaga suita este frecvent menționată ca stiva TCP/IP. În special, TCP oferă încredere, asigura livrarea ordonata a unui flux de octeți de la un program de pe un computer la alt program de pe un alt computer aflat în rețea. Pe lângă sarcinile sale de gestionare a traficului, TCP controlează mărimea segmentului de date, debitul de informație, rata la care se face schimbul de date, precum și evitarea congestionării traficului de rețea. Printre aplicațiile cele mai uzuale ce utilizează TCP putem enumera World Wide Web (WWW), posta electronica (e-mail) și transferul de fișiere (FTP).

Comunicarea în rețea dintre aplicații este facilitată de existența unei interfețe programatice, numite socket. În perspectiva conceptuală, socketul reprezintă un capăt al unei conexiuni și definește utilizarea serviciilor de rețea oferite de sistemele de operare (14). Prin urmare orice socket are tip bine definit, dar și un proces asociat (figura 6.1)

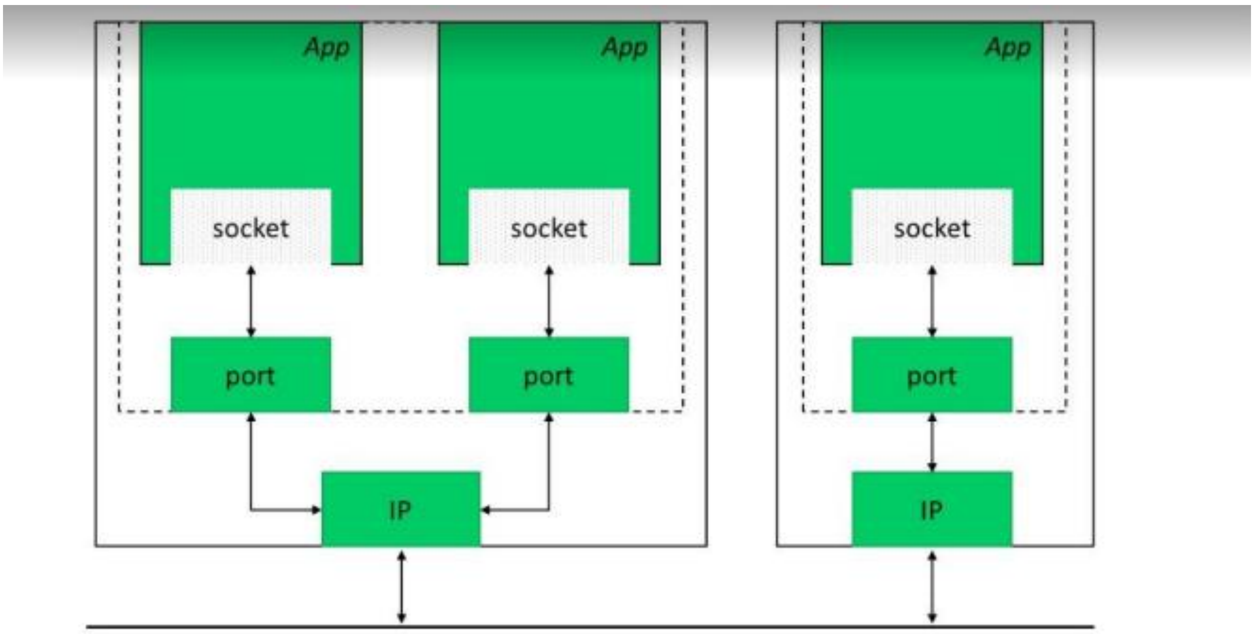

Fig. 1- Interfață socket API

Socketul fiind introdus în UNIX la implementarea unor mecanisme de comunicare inter-proces în rețea a preluat modelul existent deschide- citește/scrie-închide a datelor prin intermediul unui descriptor un socket reprezentă, de facto, o structură internă de date caracterizată univoc prin adresa IP și numărul portului fig. 1.

Portul, fiind identificator de sistem, permite comunicarea în rețea a mai multor aplicații simultan. Sistemul primind un pachet în corespundere cu numărul portului (specificat de emițător) redirecționează mesajul spre aplicație prin intermediul portului ascultat/atașat.

Comunicarea dintre aplicații este de două tipuri: orientată pe conexiune și fără conexiune. Un socket orientat pe conexiune (de tip SOCK_STREAM) intermediază o sesiune de lucru utilizând protocolul TCP. Deși stabilirea conexiunii implică costuri adiționale, totuși aceasta garantează transportul datelor între punctele terminale ale conexiunii.

În vederea aplicării socket-ului API-ul implică o succesiune specifică de funcții. Funcțiile pot lua diverse forme sintactice în limbajele utilizate, dar semantica acestora este una similară celei prezentate în figura 2.

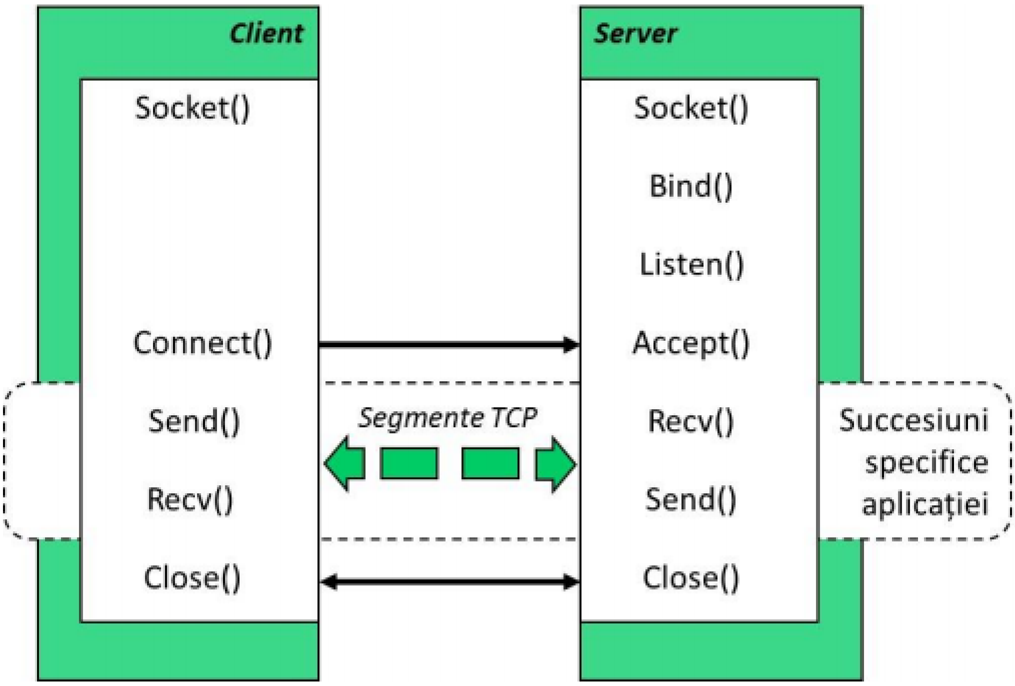

Fig. 2 - Operații primare asupra unui socket orientat pe conexiune

Astfel construim o aplicație clien-> server, server->client. Programul v-a pute fii rulat ca client sau ca server. App va face legătura intre două calculatoare prin intermediul protocolui TCP. Unde va fi pasate niște packet. Protocolul TCP face posibilă transmiterea packetelor intr-o anumită adresă ip, și intr-un port prestabilit. Un codul de mai jos este prezentat metoda de start a unui socker.

```
public void Start(int port)
        {
            if (_running)
                return;

            _port = port;
            _running = true;
            _socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);
            _socket.Bind(new IPEndPoint(IPAddress.Any, port));
            _socket.Listen(100);
            _socket.BeginAccept(acceptCallback, null);
               }
```

Deasemenea Pc trebuie să accepte un socker de apel de la partener. Modul de Callback a unui socker.

```
private void acceptCallback(IAsyncResult ar)
        {
            try
            {
```

```
                Socket sck = _socket.EndAccept(ar);

                if (Accepted != null)
                {
                    Accepted(this, new SocketAcceptedEventArgs(sck));
                }
            }
            catch
            {
            }

            if (_running)
                _socket.BeginAccept(acceptCallback, null);
               }
```

Modelul de intrare/ieșire este prezentat in forma următoare de cod. Pentru inscriere sau transmitere este necesar ca cpu să aloce `MemoryStream()`. Datele sunt convertate într-un masiv de date.

```
public class PacketReader : BinaryReader
{
    private BinaryFormatter _bf;
    public PacketReader(byte[] data)
        : base(new MemoryStream(data))
    {
        _bf = new BinaryFormatter();
    }

    public Image ReadImage()
    {
        int len = ReadInt32();

        byte[] bytes = ReadBytes(len);

        Image img;

        using (MemoryStream ms = new MemoryStream(bytes))
        {
            img = Image.FromStream(ms);
        }

        return img;
    }

    public T ReadObject<T>()
    {
        return (T)_bf.Deserialize(BaseStream);
           }
```

Funcția care determină adresa ip a utilizatorului `IPHostEntry`.

```
void getIP()
        {
            string localIP = string.Empty;
            IPHostEntry host;

            host = Dns.GetHostEntry(Dns.GetHostName());
            foreach (IPAddress ip in host.AddressList)
            {
```

```
        if (ip.AddressFamily.ToString() == "InterNetwork")
        {
            localIP = ip.ToString();
        }
    }
    textBox1 = localIP;

  }
```

Concluzie

În urma acestei lucrări de laborator au fost obținute abilități de lucru cu protocolul TCP in mediul C#. Au fost verificate mai multe metode socker pe care le aplicăm prin protocolul TCP. Am stabilit o conexiune dintre 2 PC și mai multe prin același protocol. Am obținult deprinderi practice la nivelul de transport. Am concluzionat că TCP este un protocol sigur de transmitere a datelor.

Bibliografie

1. https://drive.google.com/file/d/0B0vf11XUnLc2Q0NrLTk3czlOTWM/view
2. https://msdn.microsoft.com/en-us/library/btky721f.aspx
3. https://moodle.ati.utm.md/course/view.php?id=90
4. https://www.youtube.com/channel/UC5gufuYHPSsJA-jul-iwyXA

Anexă

Listener.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Net.Sockets;
using System.Net;
    internal delegate void SocketAcceptedHandler(object sender, SocketAcceptedEventArgs
e);

    internal class SocketAcceptedEventArgs : EventArgs
    {
        public Socket Accepted
        {
            get;
            private set;
        }

        public IPAddress Address
        {
            get;
            private set;
        }

        public IPEndPoint EndPoint
        {
            get;
            private set;
        }

        public SocketAcceptedEventArgs(Socket sck)
        {
```

```
            Accepted = sck;
            Address = ((IPEndPoint)sck.RemoteEndPoint).Address;
            EndPoint = (IPEndPoint)sck.RemoteEndPoint;
        }
    }

    internal class Listener
    {
        #region Variables
        private Socket _socket = null;
        private bool _running = false;
        private int _port = -1;
        #endregion

        #region Properties
        public Socket BaseSocket
        {
            get { return _socket; }
        }

        public bool Running
        {
            get { return _running; }
        }

        public int Port
        {
            get { return _port; }
        }
        #endregion

        public event SocketAcceptedHandler Accepted;

        public Listener()
        {

        }

        public void Start(int port)
        {
            if (_running)
                return;

            _port = port;
            _running = true;
            _socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);
            _socket.Bind(new IPEndPoint(IPAddress.Any, port));
            _socket.Listen(100);
            _socket.BeginAccept(acceptCallback, null);
        }

        public void Stop()
        {
            if (!_running)
                return;

            _running = false;
            _socket.Close();
        }

        private void acceptCallback(IAsyncResult ar)
        {
            try
            {
```

```
                Socket sck = _socket.EndAccept(ar);

                if (Accepted != null)
                {
                    Accepted(this, new SocketAcceptedEventArgs(sck));
                }
            }
            catch
            {
            }

            if (_running)
                _socket.BeginAccept(acceptCallback, null);
        }
    }
```

TransferClient.cs

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Net;

using System.Net.Sockets;

using System.IO;

namespace file_transfer

{

  public delegate void TransferEventHandler(object sender, TransferQueue queue);

  public delegate void ConnectCallback(object sender, string error);


  public class TransferClient

  {

    string GetIPAdress()

    {

      IPHostEntry myHost;

      string ip = "?";

      myHost = Dns.GetHostEntry(Dns.GetHostName());

      foreach (IPAddress ips in myHost.AddressList)

      {
```

```
        if (ips.AddressFamily.ToString() == "InterNetwork") ip = ips.ToString();
    }
    return ip;

}
void getIP()
{
    string localIP = string.Empty;
    IPHostEntry host;

    host = Dns.GetHostEntry(Dns.GetHostName());
    foreach (IPAddress ip in host.AddressList)
    {
        if (ip.AddressFamily.ToString() == "InterNetwork")
        {
            localIP = ip.ToString();
        }
    }
    textBox1 = localIP;

}
private void Form1_Load(object sender, EventArgs e)
{
    getIP();
}



//This will hold our connected or connecting socket.
private Socket _baseSocket;

//This is our receive buffer.
```

```
private byte[] _buffer = new byte[8192];

//This is used for connecting.
private ConnectCallback _connectCallback;

//This stores all of our transfers. Download and upload.
private Dictionary<int, TransferQueue> _transfers = new Dictionary<int, TransferQueue>();
private object label2;
private object textBox1;

public Dictionary<int, TransferQueue> Transfers
{
   get { return _transfers; }
}

//We should of used IsDisposed, but eh; You get the point.
public bool Closed
{
   get;
   private set;
}

//The folder we will save the files too.
//By default, it will be "Transfers" which we will set.
public string OutputFolder
{
   get;
   set;
}

//The IPEndPoint (IP Address and Port) of the connected socket.
public IPEndPoint EndPoint
```

```
{
    get;
    private set;
}

public event TransferEventHandler Queued; //This will be called when a transfer is queued.
public event TransferEventHandler ProgressChanged; //This will be called when progres is made.
public event TransferEventHandler Stopped; //This will be called when a transfer is stopped.
public event TransferEventHandler Complete; //This will be called when a transfer is complete.
public event EventHandler Disconnected; //And as you can tell, it will be called upon disconnection.

//This will be the constructor for the client when we want to connect.
public TransferClient()
{
    _baseSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
}

//This is the constructor we will use once a connection is accepted by the listener.
public TransferClient(Socket sock)
{
    //Set the socket.
    _baseSocket = sock;
    //Grab the end point.
    EndPoint = (IPEndPoint)_baseSocket.RemoteEndPoint;
}

public void Connect(string hostName, int port, ConnectCallback callback)
{
    //Set the callback we set in the parameter to our local variable.
    //We could also use the state parameter with BeginConnect so we don't need a variable as well.
    _connectCallback = callback;
    //We will begin an async connect.
```

```
  _baseSocket.BeginConnect(hostName, port, connectCallback, null);
}

private void connectCallback(IAsyncResult ar)
{
  string error = null;
  try //.NET will throw an exception if a connection could not be made.
  {
    //Call EndConnect to finish the async operation.
    _baseSocket.EndConnect(ar);
    //Grab the end point like we did up top.
    EndPoint = (IPEndPoint)_baseSocket.RemoteEndPoint;
  }
  catch (Exception ex)
  {
    //If an exception is thrown, we will set the error to the message to inform the user.
    error = ex.Message;
  }

  //After everything is done, call the callback.
  _connectCallback(this, error);
}

public void Run()
{
  try
  {
    //Begin receiving the information.
    //.NET can throw an exception here as well if the socket disconnects.
    //Just as a precaution.

    /*Except this time, we will use the socket flag of Peek
```

```
             * We will use peek to see how much data is actually available to read
             * The data can be fragmented; Meaning 2 bytes might come through, but the other 2 might lag
for
             * a few milliseconds or so
             * We'll use Peek so we don't mis-read our size bytes and get off the wall sizes.*/
            _baseSocket.BeginReceive(_buffer, 0, _buffer.Length, SocketFlags.Peek, receiveCallback, null);
        }
        catch
        {
            //If an exception is thrown, close the client.
            Close();
        }
    }

    public void QueueTransfer(string fileName)
    {
        try
        {
            //We will create our upload queue.
            TransferQueue queue = TransferQueue.CreateUploadQueue(this, fileName);
            //Add the transfer to our transfer list.
            _transfers.Add(queue.ID, queue);
            //Now we will create and build our queue packet.
            PacketWriter pw = new PacketWriter();
            pw.Write((byte)Headers.Queue);
            pw.Write(queue.ID);
            pw.Write(queue.Filename);
            pw.Write(queue.Length);
            Send(pw.GetBytes());

            //Call queued
            if (Queued != null)
            {
```

```
            Queued(this, queue);
        }
    }
    catch
    {
    }
}

public void StartTransfer(TransferQueue queue)
{
    //We'll create our start packet.
    PacketWriter pw = new PacketWriter();
    pw.Write((byte)Headers.Start);
    pw.Write(queue.ID);
    Send(pw.GetBytes());
}

public void StopTransfer(TransferQueue queue)
{
    //If we're the uploading transfer, we'll just stop it.
    if (queue.Type == QueueType.Upload)
    {
        queue.Stop();
    }

    PacketWriter pw = new PacketWriter();
    pw.Write((byte)Headers.Stop);
    pw.Write(queue.ID);
    Send(pw.GetBytes());
    //Don't forget to close the queue.
    queue.Close();
}
```

```
public void PauseTransfer(TransferQueue queue)
{
    //Pause the queue.
    //This doesn't have to be done for the downloading queue, but its here for a reason.
    queue.Pause();

    PacketWriter pw = new PacketWriter();
    pw.Write((byte)Headers.Pause);
    pw.Write(queue.ID);
    Send(pw.GetBytes());
}

public int GetOverallProgress()
{
    int overall = 0;
    try
    {
        foreach (KeyValuePair<int, TransferQueue> pair in _transfers)
        {
            //Add the progress of each transfer to our variable for calculation
            overall += pair.Value.Progress;
        }

        if (overall > 0)
        {
            //We'll use the formula of
            //(OVERALL_PROGRESS * 100) / (PROGRESS_COUNT * 100)
            //To gather the overall progess of every transfer.
            overall = (overall * 100) / (_transfers.Count * 100);
        }
    }
```

```
    catch { overall = 0; /*If there was an issue, just return 0*/ }

    return overall;
}

public void Send(byte[] data)
{
    //If our client is disposed, just return.
    if (Closed)
        return;

    //Use a lock of this instance so we can't send multiple things at a time.
    lock (this)
    {
        try
        {
            //Send the size of the packet.
            _baseSocket.Send(BitConverter.GetBytes(data.Length), 0, 4, SocketFlags.None);
            //And then the actual packet.
            _baseSocket.Send(data, 0, data.Length, SocketFlags.None);
        }
        catch
        {
            Close();
        }
    }
}

public void Close()
{
    //INSERTED - NOT IN TUTORIAL
    if (Closed)
```

```
            return;
        //
        Closed = true;
        _baseSocket.Close(); //Close the socket
        _transfers.Clear(); //Clear the transfers
        _transfers = null;
        _buffer = null;
        OutputFolder = null;

        //Call disconnected
        if (Disconnected != null)
            Disconnected(this, EventArgs.Empty);
    }

    private void process()
    {
        PacketReader pr = new PacketReader(_buffer); //Create our packet reader.

        Headers header = (Headers)pr.ReadByte(); //Read and cast our header.

        switch (header)
        {
            case Headers.Queue:
                {
                    //Read the ID, Filename and length of the file (For progress) from the packet.
                    int id = pr.ReadInt32();
                    string fileName = pr.ReadString();
                    long length = pr.ReadInt64();

                    //Create our download queue.
                    TransferQueue queue = TransferQueue.CreateDownloadQueue(this, id,
Path.Combine(OutputFolder,
                        Path.GetFileName(fileName)), length);
```

```
            //Add it to our transfer list.
            _transfers.Add(id, queue);

            //Call queued.
            if (Queued != null)
            {
                Queued(this, queue);
            }
        }
        break;
    case Headers.Start:
        {
            //Read the ID
            int id = pr.ReadInt32();

            //Start the upload.
            if (_transfers.ContainsKey(id))
            {
                _transfers[id].Start();
            }
        }
        break;
    case Headers.Stop:
        {
            //Read the ID
            int id = pr.ReadInt32();

            if (_transfers.ContainsKey(id))
            {
                //Get the queue.
                TransferQueue queue = _transfers[id];
```

```
            //Stop and close the queue
            queue.Stop();
            queue.Close();

            //Call the stopped event.
            if (Stopped != null)
                Stopped(this, queue);

            //Remove the queue
            _transfers.Remove(id);
        }
    }
    break;
case Headers.Pause:
    {
        int id = pr.ReadInt32();

        //Pause the upload.
        if (_transfers.ContainsKey(id))
        {
            _transfers[id].Pause();
        }
    }
    break;
case Headers.Chunk:
    {
        //Read the ID, index, size and buffer from the packet.
        int id = pr.ReadInt32();
        long index = pr.ReadInt64();
        int size = pr.ReadInt32();
        byte[] buffer = pr.ReadBytes(size);
```

```
//Get the queue.
TransferQueue queue = _transfers[id];

//Write the newly transferred bytes to the queue based on the write index.
queue.Write(buffer, index);

//Get the progress of the current transfer with the formula
//(AMOUNT_TRANSFERRED * 100) / COMPLETE SIZE
queue.Progress = (int)((queue.Transferred * 100) / queue.Length);

//This will prevent the us from calling progress changed multiple times.
/* Such as
 * 2, 2, 2, 2, 2, 2 (Since the actual progress minus the decimals will be the same for a bit
 * It will be
 * 1, 2, 3, 4, 5, 6
 * Instead*/
if (queue.LastProgress < queue.Progress)
{
   queue.LastProgress = queue.Progress;

   if (ProgressChanged != null)
   {
      ProgressChanged(this, queue);
   }

   //If the transfer is complete, call the event.
   if (queue.Progress == 100)
   {
      queue.Close();

      if (Complete != null)
```

```
                    {
                        Complete(this, queue);
                    }
                }
            }
        }
        break;
    }
    pr.Dispose(); //Dispose the reader.
}

private void receiveCallback(IAsyncResult ar)
{
    try
    {
        //Call EndReceive to get the amount available.
        int found = _baseSocket.EndReceive(ar);

        //If found is or is greater than 4 (Meaning our size bytes are there)
        //We will actually read it from our buffer.
        //If its less than 4, Run will be called again.
        if (found >= 4)
        {
            //We will receive our size bytes
            _baseSocket.Receive(_buffer, 0, 4, SocketFlags.None);

            //Get the int value.
            int size = BitConverter.ToInt32(_buffer, 0);

            //And attempt to read our
            int read = _baseSocket.Receive(_buffer, 0, size, SocketFlags.None);
```

```
            /*Data could still be fragmented, so we'll check our read size against the actual size.
             * If read is less than size, we'll keep receiving until we have the full packet.
             * It will only take a few milliseconds or a second (In most cases), so we can use a sync-
             * receive*/
            while (read < size)
            {
                read += _baseSocket.Receive(_buffer, read, size - read, SocketFlags.None);
            }

            //We'll call process to handle the data we received.
            process();
        }

        Run();
    }
    catch
    {
        Close();
    }
}

internal void callProgressChanged(TransferQueue queue)
{
    if (ProgressChanged != null)
    {
        ProgressChanged(this, queue);
    }
}
}
}
```

PacketIO.cs

```
using System;
using System.Collections.Generic;
```

```csharp
using System.Linq;
using System.Text;
using System.IO;
using System.Drawing;
using System.Drawing.Imaging;
using System.Runtime.Serialization.Formatters.Binary;
public class PacketWriter : BinaryWriter
{
    private MemoryStream _ms;
    private BinaryFormatter _bf;

    public PacketWriter()
        : base()
    {
        _ms = new MemoryStream();
        _bf = new BinaryFormatter();
        OutStream = _ms;
    }


    public void Write(Image image)
    {
        var ms = new MemoryStream();

        image.Save(ms, ImageFormat.Png);

        ms.Close();

        byte[] imageBytes = ms.ToArray();

        Write(imageBytes.Length);
        Write(imageBytes);
    }

    public void WriteT(object obj)
    {
        _bf.Serialize(_ms, obj);
    }

    public byte[] GetBytes()
    {
        Close();

        byte[] data = _ms.ToArray();

        return data;
    }
}

public class PacketReader : BinaryReader
{
    private BinaryFormatter _bf;
    public PacketReader(byte[] data)
        : base(new MemoryStream(data))
    {
        _bf = new BinaryFormatter();
    }

    public Image ReadImage()
    {
        int len = ReadInt32();

        byte[] bytes = ReadBytes(len);

        Image img;
```

```
        using (MemoryStream ms = new MemoryStream(bytes))
        {
            img = Image.FromStream(ms);
        }

        return img;
    }

    public T ReadObject<T>()
    {
        return (T)_bf.Deserialize(BaseStream);
    }
   }
```