Выполнение первого задания

Выполнил Чайковский ПП

Краткое описание

Проект состоит из двух программ: Server.exe и Client.exe. Так же в проект присутствует библиотека классов TcpIpLib, в ней находятся класс для работы с сетевыми клиентами и класс для работы с файлами.

Принцип работы: Запускается сервер и ожидает подключения. При запуске, клиент, пытается соединиться с сервером и, в случае успеха, пересылает информацию, после завершает работу. Сервер, при получении информации, обрабатывает её и создаёт на её основе файл, после чего ожидает следующего подключения, его работа длится до выхода из программы. Работа сервера и клиента осуществляется в соответствии с ТЗ.

Листинг программы

Программа-сервер:

```
namespace Server
{
    — ссылки
    internal class Program
    {
        static bool flag = true;
        static string[] strCon = new string[5];
        static TcpIp tcpIp;

        — ссылки
        static int Main(string[] args)
        {
            //для дебага
            //args = new string[3];
            //args[0] = "127.0.0.1";
            //args[1] = "4000";
            //args[2] = @"Data\Files\";

            //проверка параметров
            if (args.Length != 3)
            {
                Console.WriteLine("Ошибка: Не верное колличество параметров !!!");
                return -1;
            }
            tcpIp = new TcpIp();
            //преобразование параметров в нужный тип
            try
            {
                tcpIp.ip = IPAddress.Parse(args[0]);
                tcpIp.port_tcp = Convert.ToInt32(args[1]);
            }
            catch
            {
                Console.WriteLine("Ошибка: Параметры введены неверно !!!");
                return -1;
            }
```

При запуске сервер проверяет введённые аргументы на их наличии и пытается их преобразовать к нужным типам и записать в соответствующие переменные. В случае ошибок, они выведутся на экран и сервер прекратит работу.

```
    //старт работы сервера
    ConsoleUpdate();
    strCon[0] = "Сервер запущен...";
    NetWork(args[2]);
    //команда на завершение работы
    string com = "";
    while (com!="exit")
    {
        strCon[4] = "Если вы хотите завершить работу сервера - введите exit: ";
        com = Console.ReadLine();
        if (com == "exit")
        {
            strCon[4] = "Вы уверены, что хотите завершить работу сервера? (yes/no): ";
            com = Console.ReadLine();
            if((string.Equals(com, "yes", StringComparison.OrdinalIgnoreCase))||(string.Equals(com, "y", StringComparison.OrdinalIgnoreCase)))
            {
                com = "exit";
            }
        }
    }
    flag = false;
    return 0;
}
```

Далее сервер создаёт две задачи: на обновлении и вывод информации в консоль и на приём и отправлении сообщений по сети. Сервер будет выполнять эти задачи и всю программу, пока не произойдёт ошибка, либо пользователь введёт команду на завершение работы.

```
//обновление текста консоли
Ссылок: 1
static async void ConsoleUpdate()
{
    bool flagCon = true;
    string[] strConBuf = new string[strCon.Length];
    await Task.Run(async () =>
    {
        while (flag)
        {
            //проверка на отличие от текущего
            for(int i = 0; i < strCon.Length; i++)
            {
                if (strConBuf[i] != strCon[i])
                {
                    flagCon = true;
                    break;
                }
            }
            //обновление текста
            if (flagCon)
            {
                flagCon = false;
                Console.Clear();
                for(int i = 0; i < strCon.Length; i++)
                {
                    strConBuf[i] = strCon[i];
                    Console.WriteLine(strCon[i]);
                }
            }
        }
    });
}
```

Метод с задачей на обновление консоли выполняет циклом проверку на изменение 5 строк. Эти строки содержат в себе всю полезную информацию. Если хотя бы одна строка изменилась с последнего вывода в консоль, программа очищает консоль и выводит строки заново.

```
static async void NetWork(string temp)
{
    string message, filename;
    byte[] textMess;
    int num = 0; int counter = 0;
    byte[][] textBlock=new byte[1][];
    FileWork fileWork = new FileWork();
    bool flagConnection = false;
    await Task.Run(async () =>
    {
        while(flag)
        {
            //tcp прослушивание
            if (tcpIp.GetConnection())
            {
                strCon[1] = "Новое подключение...";
            }
            else
            {
                strCon[2] = "Ошибка подключения !!!";
                tcpIp.Close();
                break;
            }
            //получение имени файла и номера порта для udp
            if (tcpIp.ReadTcp(out message))
            {
                try
                {
                    filename = message.Substring(0, message.IndexOf(":"));
                    tcpIp.port_udp = Convert.ToInt32(message.Substring(message.IndexOf(":") + 1, message.Length - message.IndexOf(":") - 1));
                    flagConnection = true;
                    strCon[2] = "Имя файла и данные о UDP подключении получены...";

                }
                catch
                {
                    strCon[2] = "Ошибка: Данные о UDP подключении переданны не верно !!!";
                    tcpIp.Close();
                    break;
                }
            }
            else
            {
                strCon[2] = "Ошибка: Отсутствует соединение !!!";
                tcpIp.Close();
                break;
            }
```

Метод с задачей обработки работы с сетью так же работает циклом. Каждая итерация цикла - один раунд работы с клиентом. В начале сервер ожидает tcp подключения. После ждёт и обрабатывает информацию об имени файла и udp порте, получая их по tcp.

```
        //получение сообщения по UDP
        textMess = null; num = 1; counter = 0;
        while (counter < num)
        {
            tcpIp.ReadUdp(out textMess);
            if (textMess != null)
            {
                if (num == 1)
                {
                    //получение колличества датаграмм из первой
                    num = (int)textMess[1];
                    textBlock = new byte[num][];
                }
                textBlock[counter] = textMess;
                //отправляем сообщение о получении сообщения
                tcpIp.SendMessageTcp("ok");
                counter++;
                strCon[3] = $"Полученно данных {counter}/{num} ...";
            }
        }
        //формирование единого текста и удаление служебной информации
        textMess = fileWork.UnionBlock(textBlock);
        //создание и запись файла
        //создание директории
        if (fileWork.CreateDir(temp))
        {
            strCon[2] = "Директория создана...";
        }
        else
        {
            strCon[2] = "Ошибка: Директория не может быть создана по указанному пути !!!";
            break;
        }
        //создание файла
        if (fileWork.WriteFile(temp, filename, textMess))
        {
            strCon[2] = "Файл создан...";
        }
        else
        {
            strCon[2] = "Ошибка: Файл не может быть создан по указанному пути !!!";
            break;
        }
        strCon[1] = "Клиент отключился...";
        tcpIp.Close();
    }
});
```

После чего начинается процесс получения udp информации. Он длится, пока не будут получены все датаграммы с блоками для создания файла. Изначально количество датаграмм установлено на 1, но после получения первой - это число меняется на нужное. Если датаграмма получена, то сервер отправляет tcp сообщение "ok" - это подтверждение получения. После получения всех датаграмм, программ объединяет их в единый текст файла, удаляя все служебные данные. После чего программа проверяет на наличии директории и создаёт её, если она отсутствует. Далее программа создаёт файл и записывает в него текст файла.

Клиент

```
namespace Client
{
    — ссылки
    public class Program
    {
        static bool flag = false;

        — ссылки
        static int Main(string[] args)
        {
            //для дебага
            //args = new string[5];
            //args[0] = "127.0.0.1";
            //args[1] = "4000";
            //args[2] = "5000";
            //args[3] = "Client1.exe";
            //args[4] = "1000";

            //проверка параметров
            if (args.Length != 5)
            {
                Console.WriteLine("Ошибка: Не верное колличество параметров !!!");
                return -1;
            }
            //преобразование параметров в нужный тип
            int time=0; string filename; TcpIp tcpIp = new TcpIp();
            try
            {
                tcpIp.ip = IPAddress.Parse(args[0]);
                tcpIp.port_tcp = Convert.ToInt32(args[1]);
                tcpIp.port_udp = Convert.ToInt32(args[2]);
                filename = args[3];
                time = Convert.ToInt32(args[4]);
            }
            catch
            {
                Console.WriteLine("Ошибка: Параметры введены неверно !!!");
                return -1;
            }
            //проверка и считывание файла
            FileWork fileWork = new FileWork();
            byte[] fileText = fileWork.ReadFile(filename);
            if(fileText == null)
            {
                Console.WriteLine("Ошибка: Файл не был прочитан !!!");
                return -1;
            }
            byte[][] fileBlock = fileWork.SplitByte(tcpIp.mtu, fileText);
```

По аналогии с сервером клиент так же с начало проверяет и пытается преобразовать полученные аргументы, завершая работу в случае ошибки. Далее считывается файл и разбивается на блоки.

```
//старт работы клиента
//подключение TCP
if (tcpIp.Connection())
{
    Console.WriteLine("Подключено...");
}
else
{
    Console.WriteLine("Ошибка подключения !!!");
    tcpIp.Close();
    return -1;
}
//отправка имени файла и порта для udp
if (tcpIp.SendMessageTcp($"{filename}:{tcpIp.port_udp}"))
{
    Console.WriteLine("Имя файла и номер порта для UDP отправленны...");
}
else
{
    Console.WriteLine("Ошибка: Имя файла и номер порта для UDP не отправленны, возможно отсутствует соединение !!!");
    tcpIp.Close();
    return -1;
}

//udp
string answer;
Task sendMess;
Console.WriteLine("Передача файла...");
for (int i=0;i<fileBlock.Length;i++)
{
    flag = true;
    answer = "";
    sendMess = SendMess(fileBlock[i], time, tcpIp);
    if (tcpIp.ReadTcp(out answer))
    {
        if (answer == "ok")
        {
            flag = false;
        }
    }
    Task.WaitAll(sendMess);
}
```

После клиент пытается подключится по tcp соединению и отправить имя файла и udp порт. Далее идёт процесс отправки udp информации и вынесен в отдельную задачу. Эта задача выполняется, пока не будет получен подтверждающий ответ от сервера. После чего цикл переходит на следующий блок, для его отправления.

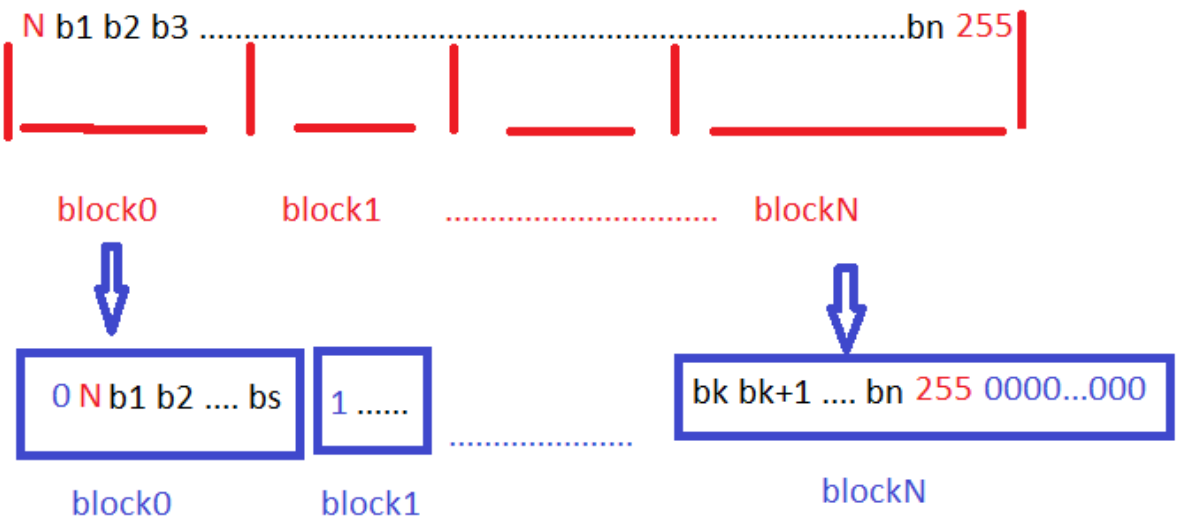
```
            //завершение передачи
            Console.WriteLine("Передача завершена...");
            Console.WriteLine("Работа завершена !");
            //конец работы клиента
            tcpIp.Close();
            return 0;
        }

        Ссылок: 1
        static async Task SendMess(byte[] message, int time, TcpIp tcpIp)
        {
            await Task.Run(async () =>
            {
                while (flag)
                {
                    //отправка сообщения udp
                    if (!tcpIp.SendMessageUdp(message))
                    {
                        Console.WriteLine("Ошибка: Отсутствует соединение !!!");
                        flag = false;
                    }
                    Thread.Sleep(time);
                }
            });
        }
    }
}
```

Далее происходит завершение работы программы. Так же на этом скриншоте изображён метод, который создаёт задачу на цикличную отправку udp с задержкой, переданной в аргументах. Цикл завершает свою работу, если ответ от сервера изменил значение flag.

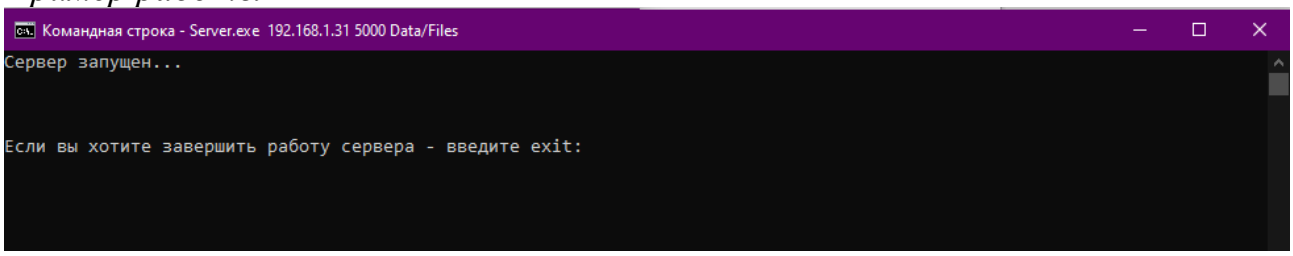
Листинг классов TcpIp и FileWork описан в комментариях к коду.

Принцип преобразования файла в блоки

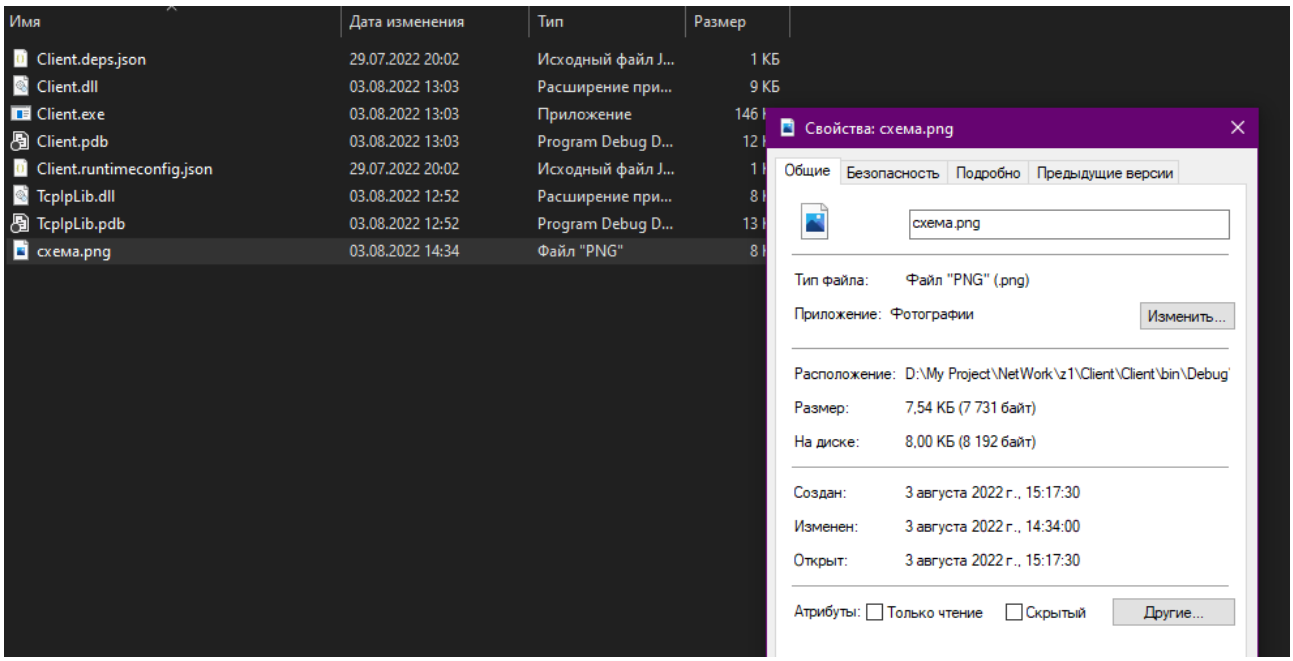


На данной не очень ровной схеме изображён принцип преобразования текста файла в блок. В начале к тексту добавляется в нулевой байт - количество блоков (так как в ТЗ указан файл не более 10Мб, а размер датаграммы 64511 байт, то одного байта хватит на хранение информации об количестве блоков), а в последний число 255, которое является конечным байтом. Далее происходит деление на блоки, в начало добавляется номер блока. Конечный байт нужен для отделении пустых 0 от исходной информации, при получении последнего блока и преобразования обратно в единый текст. Это может быть любое число, отличное от 0 и известное обеим программам. При отделении 0 формируется счётчик, который с конца считает позицию в блоке, пока идут 0, встретив число конечного байта, счётчик готов.

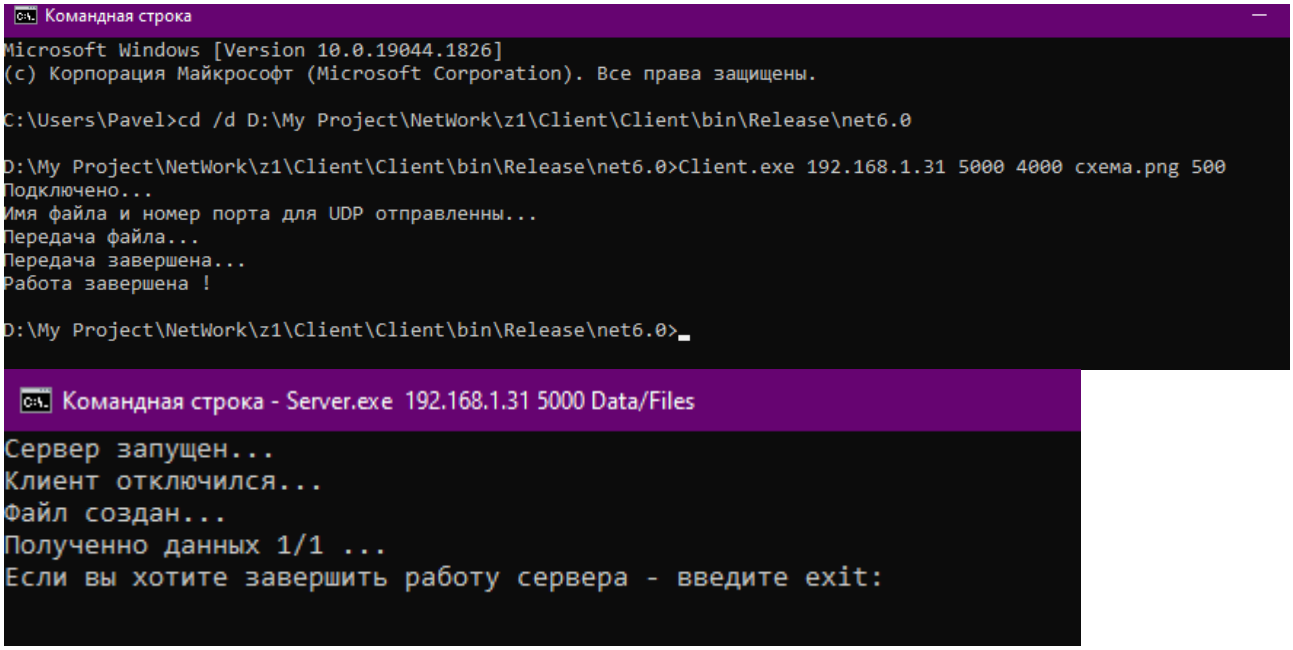
Пример работы

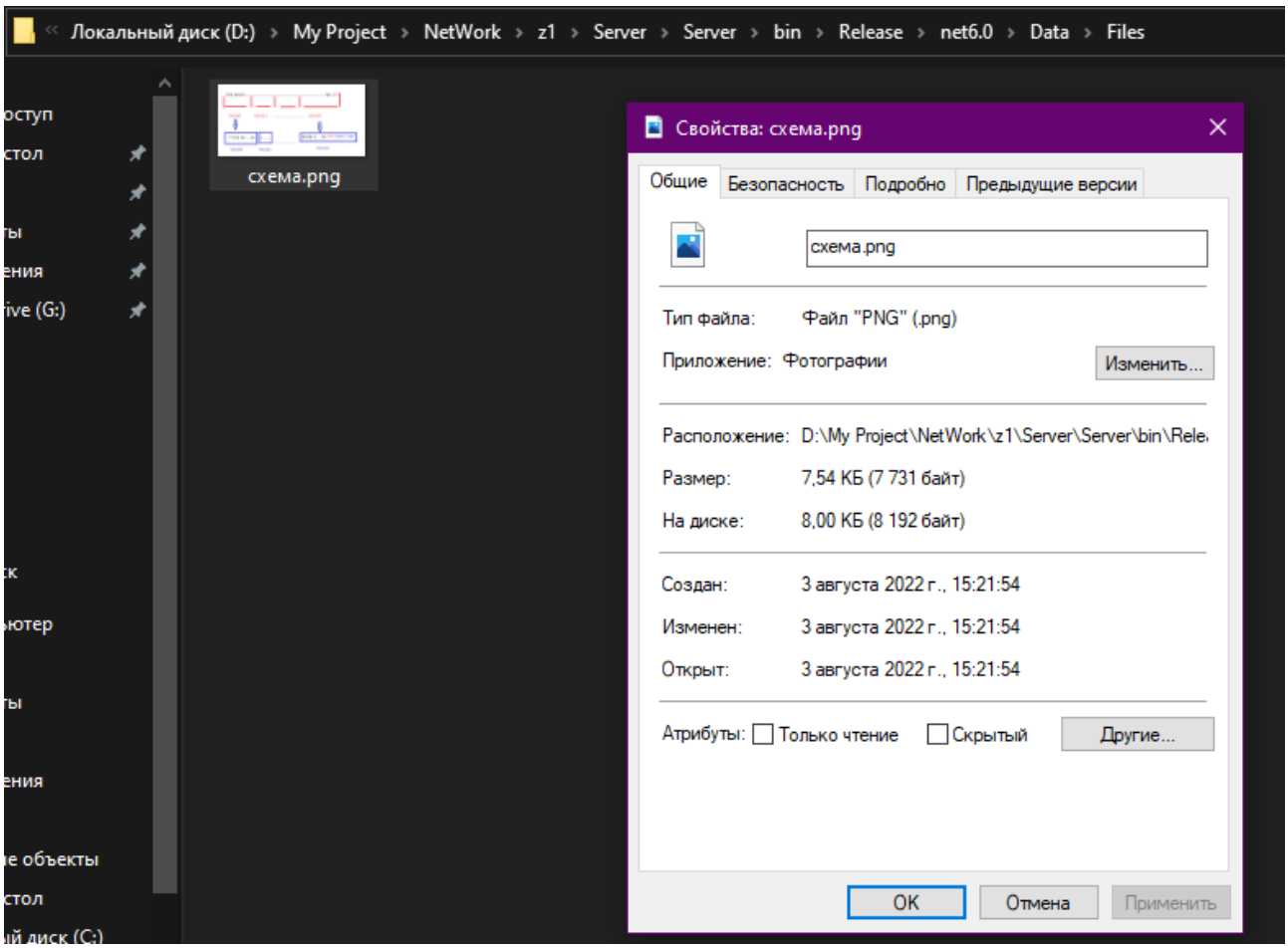


Запускаем сервер



Как вариант передадим изображение схемы из отчёта





Полученный итог

Выводы и замечания

К сожалению мне мало хватает опыта при работе с Task и Thread. Как следствие я не уверен, что они правильно формируется и функционируют с точки зрения оптимизации. Как следствие отправление одной датаграммы медленное по моему мнению (приблизительно равно 1 сек).

Так же при любой ошибке сервер необходимо перезапускать вручную, что не очень правильно.

Необходимо изменить подсчёт полученных блоков, не циклом, а исходя из записанного в него id. Так как ответ от сервера может исказиться и клиент пришлёт один и тот же блок несколько раз, хотя сервер уже получал их, это довольно большой недочёт, который я осознал уже в момент написания отчёта.

Ещё один недочёт - это отсутствие стоп крана для udp отправки сообщений от клиента. Если с сервером что то произойдёт, то клиент будет слать их бесконечно. Как самый простой вариант, это использовать конечное число, и если количество отправленных без ответа сообщений привысит его, то зафиксировать это и прекратить работу (примерно как в ping).

Орфографические ошибки - думаю в комментариях их особенно много, так что мои извинения, для читающего.