Pavel Nosovskiy

Data transfer using the UDP protocol in the Gigabit Ethernet environment

Bachelor's Thesis
Information Technology

May 2015

MAMK
University of Applied Sciences

DESCRIPTION

MAMK
University of Applied Sciences

Date of the bachelor's thesis
08 May 2015

Author(s)
Pavel Nosovskiy

Degree programme and option
Information Technology

Name of the bachelor's thesis

Data transfer using the UDP protocol in the Gigabit Ethernet environment

Abstract

This thesis was written as a result of a case in the company Comin Ltd. The objective was set to implement a UDP communication setup to replace the existing TCP one. The focus of this thesis was on the problems that were faced during the process of writing a UDP client-server application, explaining the reasons behind the observed high rates of packet loss and low network utilization rates.

The main methods that were used for the measurements were network parameters analysis to see how much of the network channel was used, and packet capturing to analyse if there are any packet loss or corruption during transmission.

The results reached were sufficient for the particular case of local area networks. By implementing congestion control for UDP the utilization rates were successfully increased and the reliability of TCP was reached.

Subject headings, (keywords)
Gigabit Ethernet, UDP, Socket programming, Local area networks, Congestion Control

Pages	Language	URN
27 + 16	English	

Remarks, notes on appendices

Tutor	Bachelor's thesis arranged by
Reijo Vuohelainen	Comin Ltd, Russia

CONTENTS

1 INTRODUCTION

Computer networks have come a long way since their initial introduction in the middle of the 20th century. While still developing rapidly even today, we can already say that a computer network has become a highly effective technology for data transfer, especially over long distances. As the complexity of computers grew and their use in human activity increased, the need for higher data transfer rate and reliability kept pushing the network technologies to new levels. This growth made people understand that if the speeds are fast enough, a network can be an efficient replacement for the analog television and telephone lines.

Over the years the use of networking increased drastically, and the purpose of the networks shifted from being only used by universities and military for information exchange and retrieval to heavier tasks like video and audio streaming. The many separated local networks were not enough anymore, and the need to create a gateway that would connect many networks to each other forced the introduction of a network protocol that would support accurate delivery of data to the right place in a wide area network. Many protocols like IPX and ARPANET were developed to suit this needs. However, historically the TCP/IP suite became the most commonly used. In the TCP/IP suite the IP protocol is a network protocol that handles the delivery of data over the network to the right host, while TCP is a transport protocol that manages the reassembly and delivery to the right process on the host. Originally, TCP was supposed to incorporate IP functionality in it, but later a decision was made that IP and TCP should be separated. This decision gave TCP/IP suite the flexibility that the other protocols did not have, allowing the IP protocol to be used in conjunction with any other non-TCP protocol, and enabling the creation of UDP. (Leiner Barry, Cerf Vinton & Clark David 2015.)

The UDP (User Datagram Protocol) is an OSI transport layer protocol that had been defined in 1980 by RFC 768 as a response to the need for an alternative to the TCP protocol. It belongs to the TCP/IP family, and thus, it is commonly used in conjunction with the IP protocol, replacing the TCP features of the suite. It was obvious for the computer engineers of the time that the network bandwidth was the chokepoint of computing, and the easiest way to increase the bandwidth was to utilize the already available networks in a more optimal way. The TCP protocol allowed stability and had many mechanisms for ensuring data integrity, but the additional checks and communication rules added to the cost of bandwidth. On the other hand, the UDP protocol was designed to be a lot less reliable and relieved from most TCP communication rules which gave the software and hardware engineers more freedom with how they could approach their objectives. (Peter 2004.)

UDP is an unreliable protocol. This means that whatever fault happens in the network directly affects the UDP conversation between two hosts. The network errors can also happen with TCP, but the TCP protocol is designed in a way where the faults of the preceding protocols are not forwarded past the TCP, and therefore the underlying protocols will never know of the errors. Still, it does not mean that the transfer speed and the quality of TCP is impossible to obtain by using UDP. UDP is heavily reliant on the reliability of the protocols preceding it. TCP, on the other hand, provides independent reliability mechanisms. In the environment where all packets are to arrive, orderly and free of corruption due to underlying protocols, TCP reliability would be redundant, and thus, only impose unnecessary overhead on the communication. Although the real-life scenarios are never ideal, as the data link layer protocols are getting more and more refined, we actually see very little packet corruption in modern day networks. The Ethernet cables are already an example of a very reliable data link layer technology that has negligible fault likelihood. Therefore, while TCP will see no improvement in efficiency over time, UDP will become more and more efficient.

UDP is widely used when high performance is of importance. The UDP packet has a very simple structure compared to other protocols. This simplicity allows UDP to have a higher potential transfer rate than the reliable protocols such as TCP. However, in reality the possibility of packet loss and non-sequential arrival limits the use of UDP down to the applications, such as VoIP, that do not need data integrity as much as data freshness.

This work will describe in detail the drawbacks and benefits of using UDP against TCP. It introduces techniques of how the drawbacks can be minimized for the UDP packet transactions over the local networks. All the experiments that will be described in this bachelor's thesis were done as part of the work for the company Comin Ltd. Comin Ltd is a Russian satellite and telecommunications service provider in the North-West Russia region, operating since 1992. Among the many fields where this company specializes is digital video and audio transmission for both the television and the internet. The company has a Gigabit Ethernet network of a server and its clients. The server had been transferring a continuous high throughput stream of video and audio data to the clients. During my work I have noticed that some of the clients, on rare occasions, experienced stuttering, and that the clients that experienced this would fall a few milliseconds behind the rest of the clients in time. This would then result in an eventual jump in the video and audio, when the difference between the live stream and the client became too high. The software engineer of the company explained to me that the current implementation of the system was using TCP to transfer data, and that it is probably the TCP forcing acknowledgements that prevent the clients from catching up to the server. I was curious as to why they used TCP instead of UDP, because UDP seemed to fit the purpose better. I was told that, for various reasons, none of the UDP implementations the company had tried could reach the transfer speed of a TCP connection. Furthermore, a very high packet loss was observed with the UDP implementations. This was a very strange case, because the UDP is by default considered to have higher transfer speed than TCP.

2 THEORY

The theory chapter will cover the operating principles and logic behind relevant technologies. Because this thesis deals with the Gigabit Ethernet network environments, at first, the operating principle of the Gigabit Ethernet protocol will be explained. Next, the overview of the TCP/IP and UDP/IP stacks will show their major differences. Finally, the section on socket programming will show how to use UDP in one's own programs.

2.1 Gigabit Ethernet operation principles

Ethernet is a family of network technologies using Carrier Sense Multiple Access with Collision Detection. Most modern Ethernet connections use the twisted pair medium which has full duplex. The uplink and downlink channels are separate, meaning that the sender can only collide with himself, and the collision detection is less important. But, in coaxial cables and optic fiber, where both the sender and the receiver use the same medium, collision detection is the main mechanic for congestion control.

The Ethernet standard was first published in 1980 and initially used the coaxial cable and had the bandwidth of 3 Mb/s. Initially, it was intended that the Ethernet would work in an environment where all the devices had been connected via one coaxial cable. Later, the twisted pair and optic cables have been added to the standard, and the nodes are now connected one-to-one with the switches that connect to multiple nodes. The second generation of Ethernet had 10Mb/s speed. The latter generations had 100 Mb/s, 1 Gb/s and 10 Gb/s speeds. In our particular case, we are mostly interested in the Gigabit Ethernet technology.

The Gigabit Ethernet has multiple physical layer standards: optical fiber, coaxial cable and the twisted pair cable. Each physical layer standard encodes its signals in its own way, but the protocol stack is similar. The Gigabit Ethernet uses frames to transfer data. Table 1 shows the structure of the Gigabit Ethernet frame.

TABLE 1. Ethernet frame structure

7	1	6	6	4	2	1-9000	4	12
Preamble (optional)	Start of frame delimiter	Destination address	Source address	VLAN tag (optional)	Length or Ether Type	Payload	Frame Check Sequence	Inter packet gap

It uses destination and source MAC addresses, Protocol Type, Payload (the actual data), and a CRC checksum for error detection. Preamble is a sequence of alternating zeroes and ones that is used by the physical devices to detect the start of a new incoming frame. It is not present in all Ethernet realizations and its implementation depends highly on the medium.

Collision detection in Ethernet

The Ethernet network device listens to traffic on the wire. If there is no traffic currently, anyone is allowed to send the frames. Once a frame is sent, the device waits for a predefined time to avoid collisions. If there is already traffic on the wire, the device also waits until it gets an opening. If multiple devices start sending frames at the same time, all the traffic is disrupted and all the devices wait for their predefined times.

Ethernet MTU

The Ethernet introduces the concept of the maximum transmission unit – the biggest chunk size of data that can be transmitted at once without fragmentation. The MTU restricts sending chunks of data, bigger than the MTU, for several reasons. The first reason is to reduce the retransmission time in the case of an error in the packet. The second reason is to avoid unfair transmission monopoly in the half-duplex medium. The last reason is historical - the hardware imposed its restriction as the smaller hardware buffers were usually faster to write and read.

The MTU definition and implementation depends on the manufacturer, but the RFC 1191 standard for Ethernet II, published in November 1990, states that it should be no more than 1 500 bytes. In the high-throughput networks, the above explained reasons for the small MTU are less important than the performance needed, and because of that the Ethernet standard introduced the “jumbo frames”, which are the frames with MTU of proportionally larger size.

2.2 Brief overview of the IP protocol

The Internet protocol is used as an underlying protocol in both TCP/IP and UDP/IP, and because of that some overview of the Internet Protocol is useful. IP is operating on the Network level of the OSI model. It was introduced in RFC 791 in 1981. Its main duty is to route the network units called packets through the network to the final destination. One very important feature of the IP is that it is designed in a way that abstracts the next protocol from the network delivery. There are quite many details about the IP that are not relevant to this thesis, and because the IP is allowing us to abstract from its internal operation, only the features that are perceived as relevant will be explained.

The IP encapsulates data with a variable length header in the front. There are currently two versions of the IP that are widely used – IPv4 and IPv6. The header structures can be seen in Table 2a and Table 2b respectively.

TABLE 2a. IPv4 header structure

<table>
<tr><td>Byte</td><td colspan="2">0</td><td colspan="2">1</td><td colspan="2">2</td><td>3</td></tr>
<tr><td>0</td><td>Version</td><td>IHL</td><td>DSCP</td><td>ECN</td><td colspan="3">Total Length</td></tr>
<tr><td>4</td><td colspan="4">Identification</td><td>Flags</td><td colspan="2">Fragment Offset</td></tr>
<tr><td>8</td><td colspan="2">TTL</td><td colspan="2">Protocol</td><td colspan="3">Header Checksum</td></tr>
<tr><td>12</td><td colspan="7">Source IP Address</td></tr>
<tr><td>16</td><td colspan="7">Destination IP Address</td></tr>
<tr><td>20</td><td colspan="7">Options</td></tr>
</table>

TABLE 2b. IPv6 header structure

<table>
<tr><td>Byte</td><td colspan="2">0</td><td colspan="2">1</td><td>2</td><td>3</td></tr>
<tr><td>0</td><td>Version</td><td colspan="2">Traffic Class</td><td colspan="3">Flow Label</td></tr>
<tr><td>4</td><td colspan="4">Payload Length</td><td>Next Header</td><td>Hop Limit</td></tr>
<tr><td>8</td><td colspan="6" rowspan="4">Source Address</td></tr>
<tr><td>12</td></tr>
<tr><td>16</td></tr>
<tr><td>20</td></tr>
<tr><td>24</td><td colspan="6" rowspan="4">Destination Address</td></tr>
<tr><td>28</td></tr>
<tr><td>32</td></tr>
<tr><td>36</td></tr>
</table>

The idea behind the Internet Protocol is that every network node is assigned a network address, and that address is used for packet delivery. The addressing schemes differ in IPv4 and IPv6, but the differences are only relevant to the Network layer. The IPv4 protocol has a packet size limit of 65 535 bytes, but if the IP suspects that the packet is too big for single transmission because of the Ethernet MTU, the packet can be split into fragments that should be reassembled by the receiver at the final destination. Although not necessarily bad, the IP fragmentation is generally avoided, because it creates additional overhead of encapsulating each fragment with an IP header. This is the main reason why the packet size on the Internet is not exceeding 1 500 bytes.

2.3 TCP protocol operation principles

In order to understand how to deploy reliability to UDP we should analyze how TCP differs from UDP. The TCP protocol data unit is a TCP header followed by the raw unaltered payload that can be fragmented. UDP has the same structure, and as such the main difference of TCP over UDP is the structure of the header. Let us look at the TCP header. Table 2 shows the structure of the TCP header. The purpose of the header fields will be explained in the order of their appearance in the list below Table 2.

TABLE 3. TCP header structure

<table>
<tr><td>Byte</td><td colspan="3">0</td><td>1</td><td>2</td><td>3</td></tr>
<tr><td>0</td><td colspan="4">Source port</td><td colspan="2">Destination port</td></tr>
<tr><td>4</td><td colspan="6">Sequence number</td></tr>
<tr><td>8</td><td colspan="6">Acknowledgement number</td></tr>
<tr><td>12</td><td>Data offset</td><td>Reserved</td><td colspan="2">Flags</td><td colspan="2">Window size</td></tr>
<tr><td>16</td><td colspan="4">Checksum</td><td colspan="2">Urgent pointer (optional)</td></tr>
<tr><td>20</td><td colspan="6">Optional data fields</td></tr>
</table>

Source port – a 16-bit field that identifies the process of the sender.
Destination port – a 16-bit field that identifies the process of the receiver.
Sequence number – a 32-bit field. If the SYN flag bit is set, this field contains the ISN (Initial Sequence Number). If the ACK flag bit is set, this field contains a number that represents the total amount of data sent by this host plus the ISN.
Acknowledgement number – a 32-bit field. If the ACK flag bit is set, this field contains a number that represents the total amount of data received by this host plus the ISN of the sender.
Data offset - a 4-bit field that shows the length of the TCP header in 4-byte words.
Reserved field – a 6-bit field consisting of zeroes reserved for future use.
Flags – a 6-bit field that contains the flag bits for URG, ACK, PSH, RST, SYN and FIN flags.
Window size – a 16-bit field that shows how many bytes of payload the host is ready to receive.

The fields in the TCP header can be divided by purpose to delivery, sequencing and connection management. The source and destination ports are ensuring delivery to the right process. The Sequence and Acknowledgement numbers are ensuring the proper delivery order (sequencing). The Checksum is for corruption detection, and the rest of the fields carry various data aimed to organize and manage the connection. A TCP connection starts with a three-way handshake. More about the three-way handshake can be seen in Figure 1. The first packet is a packet from the connection initiating side, which carries no payload, has a random sequence number, and has the SYN flag set to one and the ACK flag to zero. This is called a SYN packet, and it indicates an attempt to start a connection. The other side must reply with a packet that carries no payload, has a random sequence number, and has an Acknowledgement number of the Sequence number of the previously received packet incremented by one. The ACK flag means that the Acknowledgement number confirms the reception of the previous packet, and the SYN flag, again, attempts to start a connection. The last packet is the packet from the connection initiating side, which has no payload, has the Sequence number of the first packet incremented by one, has the Acknowledgement number of the received packets Sequence number incremented by one, and has the ACK flag set to one and SYN flag set to zero.

After the three-way handshake is complete, the Sequence number will be increased by the number of payload bytes sent with each packet. The visual scheme of the TCP three-way handshake can be seen in Figure 1. The acknowledgement number is increased with each received packet by the size in bytes of the payload of the newly received packet. By doing so, the receiver acknowledges the fact that in total there has been the following amount of data received.

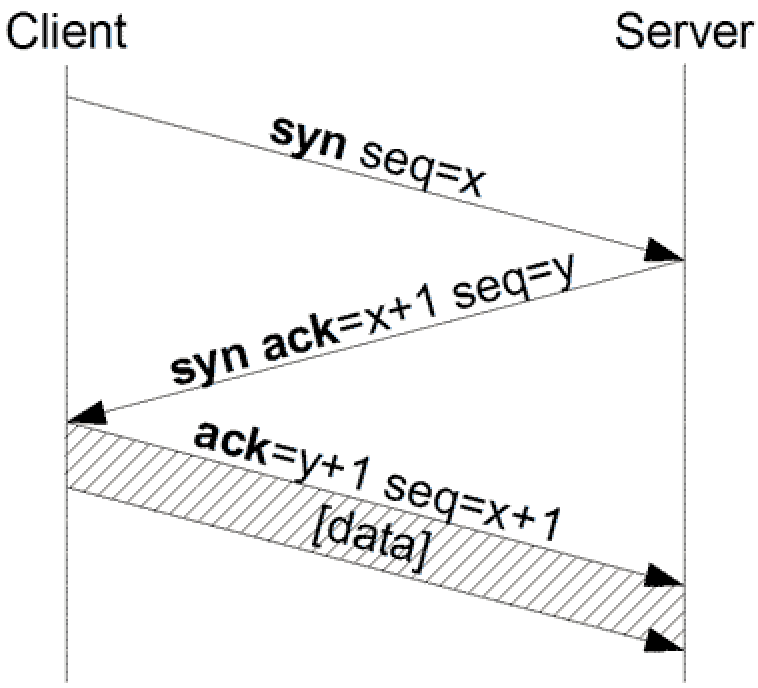


FIGURE 1. The TCP three-way handshake

TABLE 4. Structure of the TCP pseudo-header

byte	0	1	2	3
0	Source IP address			
4	Destination IP address			
8	Reserved	IP protocol	TCP segment length	

It is important to note that the checksum calculation of the TCP packet is different from the way the IP protocol does it. The Checksum is the ones' complement sum of the 16-bit words that compose the header and the payload, and a special nonexistent pseudo header. The pseudo header is never transmitted, but it is calculated from the arriving data, and added to the checksum in the same manner. The structure of the TCP pseudo-header can be seen in Table 3. The source and destination IP addresses, as well as the IP protocol number, are obtained from the preceding IP header. The TCP segment length is the total length of the TCP packet (header + payload) in bytes. The TCP segment length is calculated by subtracting the IP packet length and the IP header length. The introduction of pseudo headers increases the reliability of the error detection algorithm.

A collection of TCP packets is usually referred to as segments. Instead of waiting for Acknowledgements for each packet before sending the next one, the sender transmits data in segments in TCP. A segment's length is defined by the Receive Window, which is the Window Size that the host advertises in its packets. The frequency of sending segments is defined by the Congestion Window which is a figure that represents the currently perceived throughput of the network. Therefore, the segments are sent in bursts, and the all in all rules that govern how the segments are sent are referred to as the Sliding Window technique. After a segment has been transmitted, the receiver waits for a calculated amount of time for an acknowledgement of each of the packets in the segment. If the acknowledgement does not arrive this time, a retransmission happens.

The Sliding Window technique is used to achieve adaptive flow control and lessen the negative effects of acknowledgement overhead. The Sliding Window limits how many windows the sender can transmit at a time. There can only be one packet in a single window. Initially the Sliding Window is equal to one. When the receiver acknowledges the first packet, the sender increases the size of the Sliding Window two times. The Sliding Window will continue to increase in size until the calculated maximum is reached, based on the receivers advertised window, or until the perceived throughput of the network can no longer sustain the number of packets the sender generates, which usually results in packet loss. It should be noted that it takes several acknowledgements from the receiver before the Sliding Window can reach the calculated optimal value. This means that TCP communication is slower at the beginning than it is afterwards, which is regarded as the TCP slow start.

An important feature of TCP that should be mentioned is the retransmission. Since TCP is a connection-oriented and reliable protocol, it uses Timeouts and acknowledgements to detect retransmission necessity. If the receiver has not acknowledged packets for a specific amount of time, which is a function of the estimated round-trip delay time, the whole segment is retransmitted. This obviously creates overhead on the connection as the sender has to wait for the time to confirm the failure and then retransmit multiple packets, even in the case when only one packet from the segment is missing and a technique called fast retransmission had been developed to address the overhead of retransmission. The fast retransmission is able to recognize the loss of packets, because the receiver usually sends multiple acknowledgements with the same ACK number, since it cannot increment the number until the right packet arrives. Such packets are called Duplicate Acknowledgements, and once the sender receives a specific number of Duplicate Acknowledgements, this enables the Fast Retransmission to retransmit the missing packets immediately, avoiding the wait time and the need to retransmit the whole segment. (Snader 2009, 34-55.)

Whenever there is packet loss or corruption in the network, the sender has to retransmit packets or segments, which means that previously transmitted data must be buffered somewhere until there is certainty that it will no longer be needed. The retransmission buffer always needs to hold the last transmitted segment. The retransmission buffer is usually a dedicated NIC memory space, and buffering is implemented on the hardware side. This allows TCP buffering to be independent of the origin of data, whether it is generated dynamically or statically, as a copy is stored in the memory at all times.

2.4 UDP protocol operation principles

Now that we have an idea of what mechanics TCP uses to operate, we can take a look at UDP and note the differences. Much like TCP, UDP encapsulates the raw payload with its header. The structure of the header, however, is different and can be seen in Table 4. The purpose of the header fields will be explained in the order of their appearance in Table 4.

TABLE 5. UDP header structure

byte	0	1	2	3
0	Source port		Destination port	
4	Length		Checksum	

Source port – a 16-bit field that identifies the process of the sender.
Destination port – a 16-bit field that identifies the process of the receiver.
Length - a 4-bit field that shows the length of the UDP header in 4-byte words.

From the comparison of the two types of headers shown above, we can see that TCP and UDP have identical delivery fields. They both also have the Checksum, and the Checksum calculation algorithm itself is identical. However, in UDP the Checksum has an option to be disabled by being set to zeroes. We know that intermediate network devices only check the IP header for routing purposes, and if we compare how network devices treat the TCP and UDP packets, we will see that there is no actual difference between them. None of the additional data TCP carries in its header matters until the data arrives to the final destination and is analyzed there. And, it is only then that the discrepancy is detected and handled. This leads us to the realization that UDP has the same fault rate as TCP, but TCP handles it while UDP simply does not.

Unlike TCP, UDP does not have any congestion or flow control. There are no segments in UDP and that is why each packet is a standalone entity. This means that the way packets are sent depends on the application using the socket. In addition, there is no acknowledgement overhead, because UDP does not wait for acknowledgements. Since the concept of connections does not exist in the context of UDP protocol, there is no connection establishment procedure. UDP simply sends data straightaway avoiding the three-way handshake of TCP. Another obvious missing feature is the retransmission possibility. The protocol has no way of knowing whether the packets arrived or not, and therefore, the protocol does not make use of the retransmission buffer.

2.5 Network socket programming

A network socket is an API for communication between processes over network. It is an abstract object that represents an endpoint of a network connection. Each socket has a file descriptor which is an identification number of this socket in the operating system. Before a process can access the socket, it needs to obtain the file descriptor of that socket from the operating system. Depending on the operating system's state and the security policy it may, or may not, grant the file descriptor to the process. There are several types of network sockets. However, the type that is relevant to this case is the internet socket type. Named after the Internet Protocol it is the socket type that works with the IP-based networks. The internet sockets can be datagram sockets, stream sockets and raw sockets.

The stream sockets represent all the data as a continuous stream where the precise order of packets is crucial to the operation of the next protocol in the stack, meaning that they are connection-oriented. An example of a protocol that uses such a socket is the TCP. Stream sockets require a connection to be established before anything is sent, and the sender has to wait for a confirmation of each packet before sending the next one.

The datagram sockets, on the other hand, are connectionless. Each packet is considered logically independent. An example of a protocol that uses such a socket is the UDP. Datagram sockets do not impose any restrictions on the sender. In fact, on the network level the sender has no way of knowing if there is a receiver at all. There may not be any existing host at the destination of the datagram, and the sender is still allowed to send it.

Once the file descriptor for a socket is obtained, the process needs to link the newly created descriptor to the physical port of the host. This is called binding a socket, and it tells the operating system that packets arriving to this port are for this socket, and whichever process has the descriptor of this socket, is the process that should get the packet. Once the socket is bound to a port, the process can perform various actions on the socket. The full list of actions depends on the operating system. However, all operating systems generally have the following actions available: listening to a socket, connecting to a socket, accepting a connection, sending data and receiving data. The send and receive functions are self-explanatory. The send function puts the data into the operating system's send buffer, and the receive function is for attempting to fill the receive buffer with data from a socket. Both of these are operating system specific and will not be given much attention in this study, because their implementation is irrelevant.

Listening to a socket is waiting for a connection attempt from a remote host. In this mode the socket does nothing until it receives a connection attempt. Once the attempt is received, the socket creates an interrupt. Connecting to a socket is attempting to create a connection with a remote host. Accepting a connection refers to creating a new socket connection with a remote host who attempts to connect. It should be noted that accepting a connection creates a new file descriptor, and therefore, the descriptor that listens to the socket is not destroyed and can continue accepting new connections.

Stream socket communication begins with the server binding a socket to a specific port and listening to it. The client host binds its socket to its port as well, and attempts to connect to the server. The server accepts the connection and the data exchange begins.

A datagram socket communication begins with the server binding a socket to a specific port and receives packets on this port. The client host binds its socket to its port as well, and sends data to the server. The server receives the data and this acts as a signal that there is a client connected. The server then sends data to the client and the client sees this as a signal that the server has accepted the client. (Hall 2011, 16-26.)

3 CASE DESCRIPTION

The case of this study took place in an internal class C gigabit local area network of the company connected via an HP 1410-24G-R switch. This network was created to simulate the internet for testing new prototypes of software and services. The server was a 6-core Intel i7-5820K machine based on Haswell architecture with 12 hyper-threads. It was connected to the Comtech CDM-600L satellite modem, and it distributed network traffic from the modem to different hosts in a gigabit local area network via the switch that was using self-made TCP socket-based software. The server had the IP address 192.168.100.1 on an integrated Realtek PCIe Gigabit Ethernet network card that was connected to the switch by a standard crossover Gigabit Ethernet twisted pair with 8P8C connectors. The five computers that were assigned for my testing purposes had the IP addresses ranging from 192.168.100.38 to 192.168.100.42. The server and four of the clients were running Microsoft Windows 7, and the last one was running Microsoft Windows XP. The full topology of the network can be seen in Figure 2. The switch was connected to various additional hosts on the network, but these hosts were not included in the testing. They will be shown as the cloud in Figure 2. My task was to create a prototype of a program that would replace the TCP transmission with UDP and provide proof that it was possible to get UDP to function as effectively as TCP speed wise.

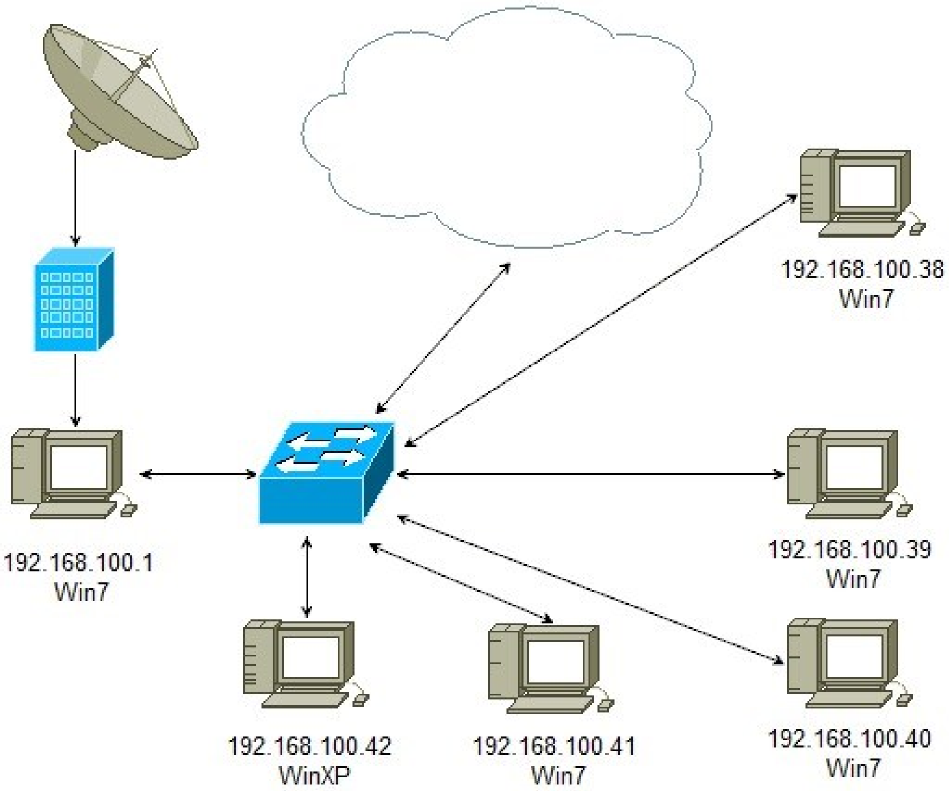


FIGURE 2. The topology of the testing network

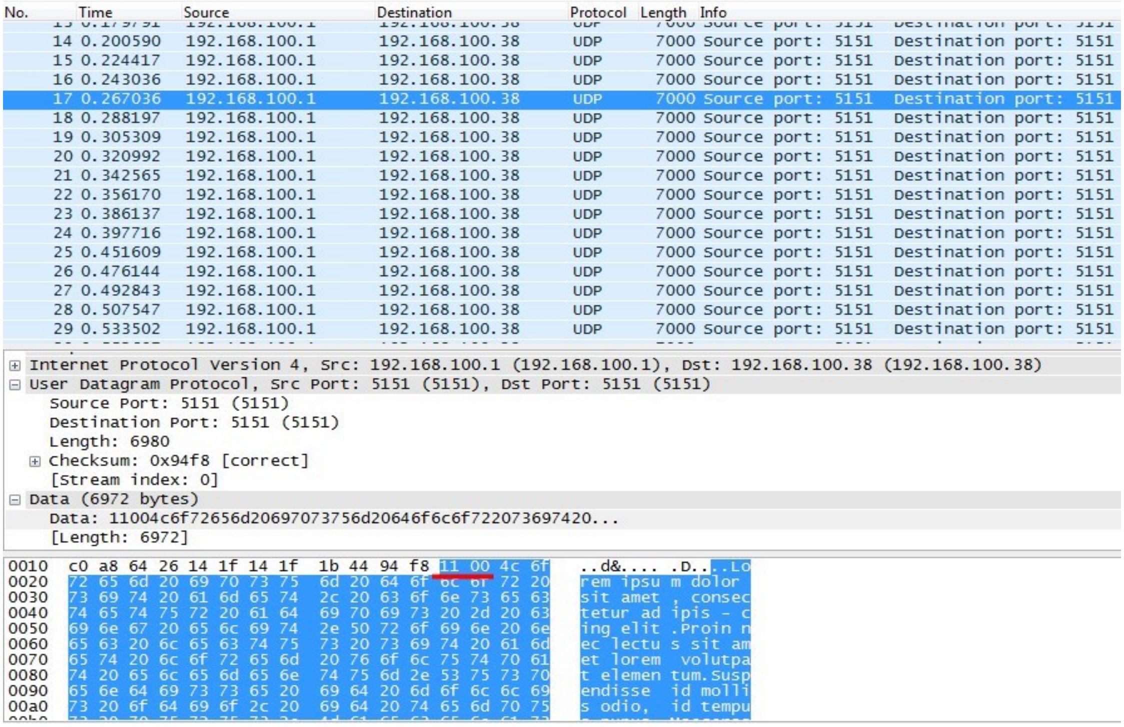
One of the first things I needed in order to draw conclusions was to be able to monitor the network channel utilization percentage. The basic network utilization can be seen in the Windows Task Manager, but I chose to use TechNet's Process Explorer. It has functions similar to the Task Manager present in the Windows operating systems, but it is greatly expanded and provides more in-depth information. Another tool used for minor monitoring of the network was the Wireshark free network protocol analyzer software. This software allowed monitoring the state of the packets. Before implementing my solution I took the readings of the channel utilization rates of the TCP solution. With four clients the channel was consistently used at the level of less than 3%. The TCP transactions were taking the majority of this number, the rest of the communication taking less than 0.01%.

After noting down the network utilization percentage I began examining the TCP environment. I ran a test to see how TCP behaved in the environment with different numbers of clients. For that, I ran 20 client applications configured with different TCP ports on each client host. The network utilization rose to 40%. Reasonably enough, this suggested that the server was capable of providing service to many more than three clients.

I began writing test UDP prototypes. The first prototype was simply broadcasting static UDP datagrams and displaying statistics in a console window. This is described more in Chapter 3.3. The network utilization percentage with such a prototype showed below 1% for five clients. The number, however, improved somewhat when one of the colleagues suggested getting rid of the graphical output for the program, as it turned out that the STL console output methods such as `cout` are blocking the execution of the code until the messages are displayed, and this slows down the execution greatly. After I removed the console output, the algorithm was able to load the network channel a little higher and the monitoring software showed 1% load. Although there was no way of knowing whether the data was arriving on the other end intact yet, this served as an alarming sign that something was going wrong in the setup.

After this, I had to come up with a lightweight and simple way to check the integrity of the data to be sure that it was not mangled. This time the prototype transmitted the same data, but added a 2-byte incremental counter to the front of the data. The client would read this counter upon connection establishment and expect each next packet to have this number incremented by one, keeping in mind that it must be reset after overflowing 65 535. If there were errors in the network, the client would report unexpected packet numbers. There is more information on the implementation of this test in Chapter 3.1 titled Sequence tracking.

The sequence tracking prototype showed major sequence de-synchronization even within the first few seconds of the client-server communication. This indicated a missing packet. Again, the Process Explorer showed very small actual network load. This gave me a suspicion that the packets were not transmitted. To test if the packets were ever actually transmitted by the server machine, I connected the server and one of the clients directly, bypassing the LAN, and ran Wireshark on the client machine. The Wireshark did not register any of the missing packets, proving that the traffic was not reaching the client.

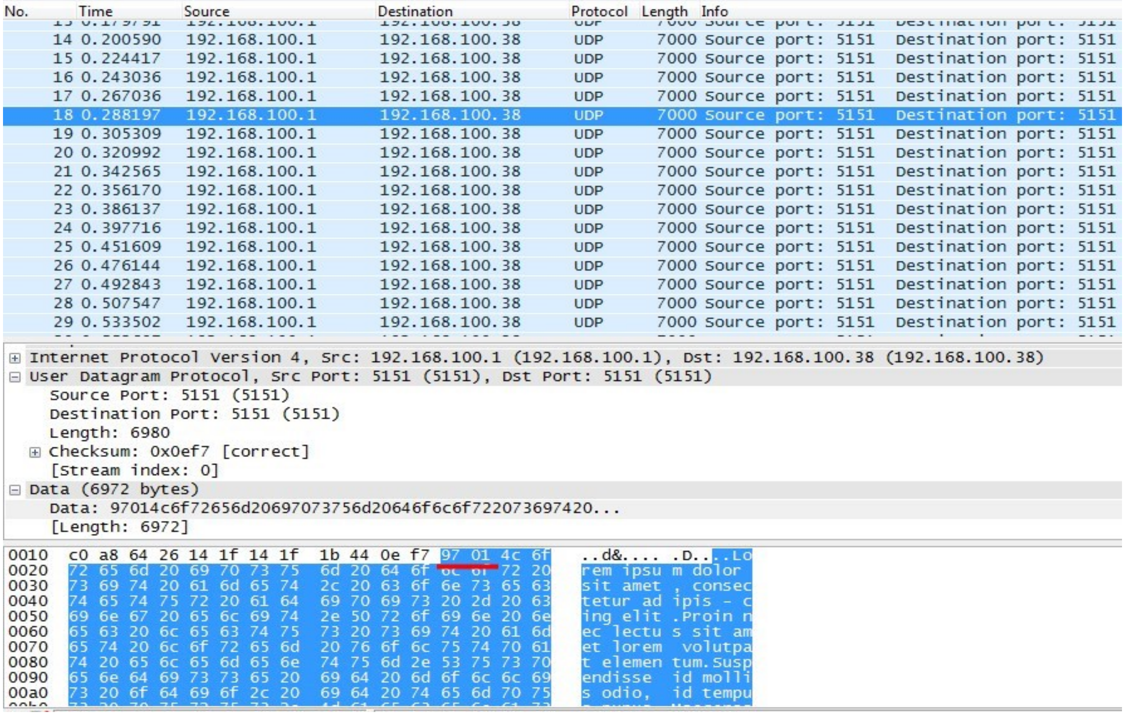


```
No.     Time       Source           Destination       Protocol Length Info
  14 0.200590  192.168.100.1    192.168.100.38    UDP     7000 Source port: 5151  Destination port: 5151
  15 0.224417  192.168.100.1    192.168.100.38    UDP     7000 Source port: 5151  Destination port: 5151
  16 0.243036  192.168.100.1    192.168.100.38    UDP     7000 Source port: 5151  Destination port: 5151
  17 0.267036  192.168.100.1    192.168.100.38    UDP     7000 Source port: 5151  Destination port: 5151
  18 0.288197  192.168.100.1    192.168.100.38    UDP     7000 Source port: 5151  Destination port: 5151
  19 0.305309  192.168.100.1    192.168.100.38    UDP     7000 Source port: 5151  Destination port: 5151
  20 0.320992  192.168.100.1    192.168.100.38    UDP     7000 Source port: 5151  Destination port: 5151
  21 0.342565  192.168.100.1    192.168.100.38    UDP     7000 Source port: 5151  Destination port: 5151
  22 0.356170  192.168.100.1    192.168.100.38    UDP     7000 Source port: 5151  Destination port: 5151
  23 0.386137  192.168.100.1    192.168.100.38    UDP     7000 Source port: 5151  Destination port: 5151
  24 0.397716  192.168.100.1    192.168.100.38    UDP     7000 Source port: 5151  Destination port: 5151
  25 0.451609  192.168.100.1    192.168.100.38    UDP     7000 Source port: 5151  Destination port: 5151
  26 0.476144  192.168.100.1    192.168.100.38    UDP     7000 Source port: 5151  Destination port: 5151
  27 0.492843  192.168.100.1    192.168.100.38    UDP     7000 Source port: 5151  Destination port: 5151
  28 0.507547  192.168.100.1    192.168.100.38    UDP     7000 Source port: 5151  Destination port: 5151
  29 0.533502  192.168.100.1    192.168.100.38    UDP     7000 Source port: 5151  Destination port: 5151

⊞ Internet Protocol Version 4, Src: 192.168.100.1 (192.168.100.1), Dst: 192.168.100.38 (192.168.100.38)
⊟ User Datagram Protocol, Src Port: 5151 (5151), Dst Port: 5151 (5151)
    Source Port: 5151 (5151)
    Destination Port: 5151 (5151)
    Length: 6980
  ⊞ Checksum: 0x94f8 [correct]
    [Stream index: 0]
⊟ Data (6972 bytes)
    Data: 11004c6f72656d20697073756d20646f6c6f722073697420...
    [Length: 6972]

0010  c0 a8 64 26 14 1f 14 1f  1b 44 94 f8 11 00 4c 6f   ..d&.... .D....Lo
0020  72 65 6d 20 69 70 73 75  6d 20 64 6f 6c 6f 72 20   rem ipsu m dolor 
0030  73 69 74 20 61 6d 65 74  2c 20 63 6f 6e 73 65 63   sit amet , consec
0040  74 65 74 75 72 20 61 64  69 70 69 73 20 2d 20 63   tetur ad ipis - c
0050  69 6e 67 20 65 6c 69 74  2e 50 72 6f 69 6e 20 6e   ing elit .Proin n
0060  65 63 20 6c 65 63 74 75  73 20 73 69 74 20 61 6d   ec lectu s sit am
0070  65 74 20 6c 6f 72 65 6d  20 76 6f 6c 75 74 70 61   et lorem  volutpa
0080  74 20 65 6c 65 6d 65 6e  74 75 6d 2e 53 75 73 70   t elemen tum.Susp
0090  65 6e 64 69 73 73 65 20  69 64 20 6d 6f 6c 6c 69   endisse  id molli
00a0  73 20 6f 64 69 6f 2c 20  69 64 20 74 65 6d 70 75   s odio,  id tempu
```

FIGURE 3a. The packet Sequence is as expected. The Sequence bytes in Little-Endian order are underlined with red



```
No.     Time       Source           Destination       Protocol Length Info
  14 0.200590  192.168.100.1    192.168.100.38    UDP     7000 Source port: 5151  Destination port: 5151
  15 0.224417  192.168.100.1    192.168.100.38    UDP     7000 Source port: 5151  Destination port: 5151
  16 0.243036  192.168.100.1    192.168.100.38    UDP     7000 Source port: 5151  Destination port: 5151
  17 0.267036  192.168.100.1    192.168.100.38    UDP     7000 Source port: 5151  Destination port: 5151
  18 0.288197  192.168.100.1    192.168.100.38    UDP     7000 Source port: 5151  Destination port: 5151
  19 0.305309  192.168.100.1    192.168.100.38    UDP     7000 Source port: 5151  Destination port: 5151
  20 0.320992  192.168.100.1    192.168.100.38    UDP     7000 Source port: 5151  Destination port: 5151
  21 0.342565  192.168.100.1    192.168.100.38    UDP     7000 Source port: 5151  Destination port: 5151
  22 0.356170  192.168.100.1    192.168.100.38    UDP     7000 Source port: 5151  Destination port: 5151
  23 0.386137  192.168.100.1    192.168.100.38    UDP     7000 Source port: 5151  Destination port: 5151
  24 0.397716  192.168.100.1    192.168.100.38    UDP     7000 Source port: 5151  Destination port: 5151
  25 0.451609  192.168.100.1    192.168.100.38    UDP     7000 Source port: 5151  Destination port: 5151
  26 0.476144  192.168.100.1    192.168.100.38    UDP     7000 Source port: 5151  Destination port: 5151
  27 0.492843  192.168.100.1    192.168.100.38    UDP     7000 Source port: 5151  Destination port: 5151
  28 0.507547  192.168.100.1    192.168.100.38    UDP     7000 Source port: 5151  Destination port: 5151
  29 0.533502  192.168.100.1    192.168.100.38    UDP     7000 Source port: 5151  Destination port: 5151

⊞ Internet Protocol Version 4, Src: 192.168.100.1 (192.168.100.1), Dst: 192.168.100.38 (192.168.100.38)
⊟ User Datagram Protocol, Src Port: 5151 (5151), Dst Port: 5151 (5151)
    Source Port: 5151 (5151)
    Destination Port: 5151 (5151)
    Length: 6980
  ⊞ Checksum: 0x0ef7 [correct]
    [Stream index: 0]
⊟ Data (6972 bytes)
    Data: 97014c6f72656d20697073756d20646f6c6f722073697420...
    [Length: 6972]

0010  c0 a8 64 26 14 1f 14 1f  1b 44 0e f7 97 01 4c 6f   ..d&.... .D....Lo
0020  72 65 6d 20 69 70 73 75  6d 20 64 6f 6c 6f 72 20   rem ipsu m dolor 
0030  73 69 74 20 61 6d 65 74  2c 20 63 6f 6e 73 65 63   sit amet , consec
0040  74 65 74 75 72 20 61 64  69 70 69 73 20 2d 20 63   tetur ad ipis - c
0050  69 6e 67 20 65 6c 69 74  2e 50 72 6f 69 6e 20 6e   ing elit .Proin n
0060  65 63 20 6c 65 63 74 75  73 20 73 69 74 20 61 6d   ec lectu s sit am
0070  65 74 20 6c 6f 72 65 6d  20 76 6f 6c 75 74 70 61   et lorem  volutpa
0080  74 20 65 6c 65 6d 65 6e  74 75 6d 2e 53 75 73 70   t elemen tum.Susp
0090  65 6e 64 69 73 73 65 20  69 64 20 6d 6f 6c 6c 69   endisse  id molli
00a0  73 20 6f 64 69 6f 2c 20  69 64 20 74 65 6d 70 75   s odio,  id tempu
```

FIGURE 3b. The packet Sequence is wrong. The Sequence bytes in Little-Endian order are underlined with red

Figures 3a and 3b show the first part of the Wireshark capture file, recorded on the client machine during the tests. The time field shows the time of packets' arrival in microseconds, relative to the time of the first packet. The source address, destination address, ports and protocol fields show that the server (192.168.100.1) is sending packets to one of the clients (192.168.100.38) using the UDP protocol from port 5 151 to port 5 151. The length shows that the total size of the frames is 7 000 bytes, because jumbo frames of respective size are activated. Both images show that Wireshark validated UDP CRC checksum. The fact that the transmission took place in a local network means that errors have a low probability of being introduced to the packet payload during transmission. The bottom section shows the beginning of the selected packet's bytes in hexadecimal values. The selected blue bytes are the UDP payload. The first two bytes of the payload are the sequence counter, and are underlined with a red line.

In Figure 3a the packet number 17 is selected, and the sequence counter value underlined is 0x11, which corresponds to the decimal value 17, and is the expected Sequence value for this packet. In Figure 3b the packet number 18 is selected, and the Sequence counter value underlined is 0x197, which corresponds to the decimal value 407 and is 389 packets bigger than the expected value of 18. The Sequence numbers 18 to 406 are not in the capture file. In an environment where packets have a single path to travel this means that these packets never reached the destination.

In a local area network there are no intermediate devices that could introduce packet loss, suggesting that the packets were dropped by the sender without entering the network. Overall, the packet gaps followed a recurring pattern - all the test runs resulted in packet gaps with similar timing on the same machines, while having constant trip time, and reproducing symptoms on multiple runs. The out of sequence packets arrived in waves with big gaps of missing packet counts, and the missing packets would never arrive. The average time before the first packet gap would be ~250 milliseconds.

The UDP packet gaps carried some noticeable characteristics. The gaps were more pronounced in situations when there was a presence of multiple TCP communications on the same network alongside UDP. The TCP connections themselves suffered periodic nearly synchronous error occasions, followed by the reduction of the sliding window. Presumably, TCP sliding window was able to adjust to the congestion almost instantly.

Another parameter that affected the error rate of UDP was the packet size. The safe minimum size datagrams of 512 KB proved to have much more errors than the jumbo frames of 5 000 MB. This was due to overhead introduced by the Ethernet technology explained above, and the UDP/IP header overhead being bigger with more packets. With the header overhead being ten times higher the setup was generating more packets, hence missing more packets. The fact that the packets never reached the client side made me suspect that there is some sort of collision or transmission limitation on the hardware part that prevents the data from being transmitted. I therefore decided to try and add a simple congestion avoidance system, and Chapter 3.2 titled Congestion control contains the results. The new prototype used a predefined speed value that would prevent it from generating more packets than it should. This time around the network load showed as high as 50% of the network channel with five clients and no errors in sequencing.

After achieving these results I only needed to make the code such that it would transmit actual modem data instead of static data. The part of the code that was reading data from the modem was already written before me, and I only needed to make a modular dynamic library that could be called upon when the data needed transmission or reception. The final prototype implementation received data from the modem and transmitted it by the use of the `send` call to any host that tried connecting to the server via port 5 151. The code was written in object-oriented style. The main reason for that was to ensure that the code related to the network is logically separated from the rest of the program. Such separation increases the granularity of a program, and reduces the chances of a new feature introducing errors into a previously error-free part of the code. If there are any errors, they are localized in one class, and as such, can be discovered faster.

Another feature of object-oriented programming that is especially helpful in the context of dynamic libraries is the abstraction of the internal methods from the external interface. A program that is going to use a dynamic library does not need to know or care about how the library works – it only needs to know the interface. This independent approach is encouraged by class structure that already suggests a separation between an internal and an external scope. This lets multiple people work on a project and expand it without invoking any changes outside their part of the project, as long as the interfaces stay consistent with the newer versions of libraries. The details of the implementation can be seen in Chapter 3.4. The final solution showed 4% network load with five clients, and 50% load with 100 clients, which was higher than the TCP solution. The reliability of the packet delivery was comparable with the TCP solution, as the packet gaps were very hard to find and only appeared after three days of testing.

3.1 Sequence tracking

The first thing that has to be addressed is packet sequencing. In an environment where there are multiple routes to the same network the packets can arrive out of original order. In a fixed path network (e.g. local area network) the sequencing errors have not been observed. To deal with the unorderly packet arrival the TCP protocol makes use of two 32-bit long fields called the SEQ and ACK fields. These fields are incremented with the number of bytes sent and received in the packet.

In the UDP solution I provided I suggest using one 16-bit field that would simply store the packet number (corresponding to C++ unsigned short data type). The packet number would reasonably enough start from one with the first packets, and each subsequent packet would have a packet number of the previous packet number plus one. Once the number of packets becomes greater than 65 535(which is the maximum value of an unsigned short), the number becomes zero again. This approach to sequencing was partially inspired by the similar technique used in several other protocols, such as the Frame Relay and the Unidirectional Lightweight Encapsulation. Both of the mentioned protocols use simple 8-bit and 16-bit counters for their fragmentation.

Originally, I tried using 8-bits to track packet numbers, but during field testing it was soon discovered that in a faulty environment it is quite common to have more than 255 packets lost, resulting in confusion about whether the next packet is from far ahead future or from the past. 65 535 is an optimal number of packets for local area networks, as the possibility of packet losses of such scale is small. However, for the internet communication it may be the case that 16-bits are not enough, and at least 24-bits are recommended for numbering to have a number of more than 16 million packets.

While evaluating the benefits and drawbacks of these two approaches I’ve come to the conclusion that while the TCP approach seemed smart and complicated, it really did not grant substantial benefits compared to the more simple one I offer in the reality of

UDP, where there are no ACK and SYN packets. Thus, I've decided to stick with a more compact one to minimize the overhead. Once the sequence of the packets is properly tracked, we have means of detecting, if the packets come out of order, or are lost.

3.2 Congestion control

The second important feature of TCP that is not present in UDP is the congestion control. TCP uses window size to ensure that the channel is not overloaded beyond its physical capacities, but in a raw implementation of UDP, the only limiting factor to sending packets was the CPU frequency. Let us assume that the CPU has 4 GHz frequency, if we multiply that by the size of the IP packet we can get a very rough number of how much traffic the setup can generate. "The Ethernet standard uses an MTU of 1 500 bytes, although a lot of Ethernet hardware can support more, such as 9 000-byte jumbo frames. Wi-Fi also uses 1 500 bytes to be compatible with Ethernet". (Beijum, 2014.) If we do not resort to jumbo frames, our biggest non-segmental IP packet size would be 1 500 bytes. The minimum UDP/IP header overhead is 28 bytes, so the data payload in the packet is going to be 1 472 bytes.

This gave me the idea to test how many send attempts can a CPU generates. I ran the code in Appendix 1 with /O2 optimization settings, and it reported an average of 292 161 send calls in a second, based on 100 runs. Multiplied by the packet size of 1 500 the traffic generation per second would be 483 241 500 (~500 MB) bytes of traffic per second. These calculations represent how many times the CPU pushes a new outbound packet to the network interface card buffer. Ideally, this is also how much data would exit the network card of the host, if the hardware allowed for that kind of throughput. Still, if we multiply that by 8 to translate this number to bits, instead of bytes, we would get 3 505 932 000 (~3.5 Gb) bits of traffic, which is at least three times as much as the Gigabit Ethernet standard in question can support. These numbers do indicate that the networks are still the bottlenecks of the data transfer in the computing world, and that there is immense potential for improvement in the field. This is also proven by the fact that many hardware manufacturers have started using protocol compression algorithms like IPComp, introduced in RFC 3173, to increase the throughput for networks by spending additional CPU cycles compressing and decompressing data.

The fact that the sender would attempt to send more data than was possible resulted in the network card of the sending computer buffering the send queue. The network card model used by both computers was the Realtek PCIe GBE. This model has a maximum send buffer size of 128*64 KB or ~8,2MB, and a maximum receive buffer of 512*64 KB or ~32,7MB. Once the send buffer was overflown, the network card discarded the data without ever sending it. To deal with the above problem we will need to limit the packet sending attempts by UDP based on time.

3.3 Bare minimum UDP implementation in C++

To find the source of the problem I had to use a very simple console program and to see how the traffic behaves.

A bare minimum UDP implementation should carry out the following steps:

- Initialize an UDP socket. A socket can be initialized with the `socket (int address_family, int socket_type, int protocol)` function which takes an address family argument (which, in our case can be either 2 or 23 representing IPv4 and IPv6 respectively), a socket type argument (which is 2 for UDP datagrams), and a protocol argument (which can be left zero if the socket type is 2, or explicitly specified as 17 for UDP). This function has an implementation in both Windows and Linux, and it either returns a descriptor for the new socket, or -1, if there was an error.
- Bind the socket to a specific port. To assign a port to the socket the `bind(int socket, (struct sockaddr_in *) self, size_t size)` method can be used. The first argument is the descriptor of the created socket. The second argument is the pointer to a SOCKADDR_IN structure describing the address and port of operation. It should be noted that if the device has multiple network interfaces, SOCKADDR_IN can specify which one should be used via the `sin_addr.s_addr` parameter. The last argument is the size of the sockaddr_in structure.
- Configure the destination host addresses to send data to. The destination addresses are using the same structure SOCKADDR_IN, but the address and port are going to belong to the destination host.
- Send data. UDP sockets use the `send(int socket, const void * data_pointer, int data_length, int flags)` method to send datagrams. Similarly, the first

parameter is the descriptor of the created socket. The second parameter is a pointer to the data to be sent, and the third parameter is the size in bytes of the data block to be sent. The last parameter is a set of special flags that can influence the way the function behaves. For example, the MSG_OOB flag allows sending out-of-bound datagrams which are a special type of datagrams that can be used to signal events outside the data transmission channel. This flag could be useful for implementing congestion notifications and retransmission requests without interfering with the actual data flow. The default flag value is zero, and means that no special actions are performed.

- Receive data. To receive data on the UDP sockets we can use the `recv(int socket, const void * buffer_pointer, int buffer_length, int flags)` method. The first parameter is the socket descriptor for the created socket. The second parameter is the pointer to the receive buffer that will hold the received data, if there is any. The third parameter is the size of the receive buffer. It should be noted that the size of the receive buffer, for obvious reasons, should be at least the size of the MTU we chose to send, but it is not necessarily limited to that and can be bigger. The last parameter is a set of special flags that can influence the way the function behaves. For example, the MSG_PEEK flag allows the function to get the size of the incoming data without actually reading it. The default flag value is zero, and means that no special actions are performed.

The UDP sender would simply send non-stop 998 bytes of constant data, and the 2 byte number of the packet added in front. The actual codes for the UDP sender and receiver are listed in Appendices 1 and 2.

The receiver would receive the data on the socket and verify that the first two bytes of the message correspond to the number of the packets received. Once the testing programs were launched on both machines, the transmission would begin, and almost immediately, massive packet losses were observed. Upon investigating, it was uncovered that this was due to buffer overflow on the sender side.

4 FINAL UDP IMPLEMENTATION

Taking into account the necessity of congestion control on the sender side, we come to this solution. This time around, for the purpose of modularity and usefulness for other people this code is written as a class.

The server listens to connections, and once some client attempts a connection, the server starts sending data that is buffered. To use this class we will need to register an instance in the code like this:

UDPServer Udpserver;

Then we call the send method when we need to send a chunk of data.

Udpserver.send(data);

The data should be stored in a std::vector format, which is a standard C++ implementation, supported by Visual C++ and G++, and it should be available in most C++ compilers since 1998.

In order to organize congestion control we will need to call the send function in reasonable intervals. To do so we will need to use time-bound code like the following:

```
#include <chrono>
ULONG timeOld=std::chrono::high_resolution_clock::now();
ULONG timeNow;
…
timeNow = std::chrono::high_resolution_clock::now();
if(timeNow – timeOld > interval)
{
  Udpserver.send(data);
  timeOld = timeNow;
}
```

The interval would be the desired time difference in milliseconds. If the total amount of data sent in a second should be 900 000 000 (reserve 10% for the other purposes and packet overhead) bits, and the packet payload is 1 300 bytes (10 400 bits), we can send roughly 86 538 packets in a second. The interval therefore would be ~ 0.011 milliseconds.

The layout and usage of the UDP receiver is similar. You initialize an instance of the class and call the recv method to get the data from it. It should be noted that this time around a vector container is required for the output. The actual codes of the UDP receiver and sender can be seen in Appendices 3 and 4.

5 RESULTS

Once the congestion control system was introduced to the UDP, its stability increased to an approximate of one corrupted packet per three days of transmission. The network utilization rate became 10% higher, which was a success. This has proven that the UDP-based transmission can be considered as a viable alternative to the current TCP solution, and showed that TCP buffering is detrimental to the transfer rates. It is very important to notice that the transmission unit size is important for the network stability. The size of the packets is directly related to the throughput of the network and inversely related to the number of network errors. As displayed in the theory sections of Ethernet and UDP, the protocols have a mandatory packet overhead. The overhead of any single protocol may not seem significant, but together the Ethernet, IP, UDP and any underlying protocols create an overhead enough to degrade the network performance noticeably. The influence is particularly strong, if the transmission unit is below the 500 byte size, because then the overhead is comparable in size with 12% and above of the data payload, while being 10 times less with the jumbo frames of 8 000 bytes. The overall stability gained was sufficient to discard the necessity to implement a retransmission technique, because such a packet loss in the video and audio stream is almost unnoticeable. However, it can still be implemented based on the sequencing mechanism provided.

6 CONCLUSION

Most tests run during this study conclude that the UDP protocol has potential to be used in the future. This is also confirmed by the fact that in recent years the newly emerging applications show a tendency to choose UDP over TCP even for file transfer (see TFTP protocol). Over the years, the Link layer technologies have seen improvement in transfer speeds and reliability, and new link layer media have been discovered. Furthermore, the routing technologies nowadays have mechanisms to avoid out of order delivery through link distance evaluation (see OSPF protocol). Since the reliability of UDP is dependent on the transfer media, a conclusion can be made that in the future UDP may be used more liberally with the reliability not being as much of an issue. The main problem of UDP on the Internet is that the IP protocol allows multiple routes of unequal trip time to the same destination, which means that packets can arrive in arbitrary order. If there was not alternative routing, or the load balancing was not in use, we could see that UDP as being not too far away from the required reliability to carry the packets in the right order.

Although the internal difference between TCP and UDP is very big, the resulting output is comparable and many software designers have made wide use of UDP on the internal satellite links where the lower level protocols ensure that the sequencing is achieved by the use of other mechanisms. All in all, these conclusions do not undermine the main value of TCP, which is reliability. Both TCP and UDP have their purpose, and TCP will always be preferred to UDP in unstable environments. But, it is important to recognize the opportunities for using faster solutions with little to no drawbacks of being used, such as in this case where the local area network eliminates most of the problems TCP aims to solve.

The results enabled the software engineers of the company to reconsider investing time into a UDP-based prototype. At the moment the local testing environment UDP setup has passed the requirements and it is now being decided if it is worth implementing the UDP system to the real clients. However, the company feels reluctant to do it yet due to the lack of real internet environment testing. One place where the UDP solution is already running is in the internal links between the servers of the company. There is no device uncertainty and multipath delivery. Therefore, the efficiency of UDP can be further maximized on the local connection by using jumbo frames.

This case confirmed that UDP can achieve higher throughput than TCP, and that it was possible to implement it effectively in the particular environment of Comin Ltd. The main and unexpected problem to consider in local area networks with UDP was the congestion control, as it is surprisingly rarely touched upon by the literature describing the UDP and the programming techniques for it. The only clues as to what was happening in the network with the packet loss were drawn from the several similar discussions on the Internet where the network experts pointed out that network congestion was a possibility. The network congestion should be discussed more in the context of UDP, as it can definitely create unexpected problems for software developers.

7 SOURCES

Beijum, Iljutsch 2002. Building Reliable Networks with the Border Gateway Protocol. O'Reilly Media.

Cerf Vinton & Kahn Bob 1981. WWW document.
http://tools.ietf.org/html/rfc791
Updated September 1981. Referred May 1, 2015.

Hall Brian 2011. Beej's Guide to Network Programming Using Internet Sockets. Jorgensen Publishing.

Leiner Barry, Cerf Vinton, Clark David, Kahn Robert, Kleinrock Leonard, Lynch Daniel, Postel Jon, Roberts Larry & Wolff Stephen 2015. WWW page.
http://www.internetsociety.org/internet/what-internet/history-internet/brief-history-internet
Updated 2015. Referred May 1, 2015.

Mogul Jeffrey & Deering Steve. WWW document.
https://tools.ietf.org/html/rfc1191
Updated November 1990. Referred May 1, 2015.

Peter Ian 2004. WWW-pages.
http://www.nethistory.info
Updated 2004. Referred May 1, 2015.

Postel J 1980. WWW document.
http://tools.ietf.org/html/rfc768
Updated September 28 ,1980. Referred May 1, 2015.

Snader Jon 2009. Effective TCP/IP Programming. Moscow: DMK Press.

APPENDICES

Appendix 1

```
#pragma comment(lib,"Ws2_32.lib")
#include <Winsock2.h>
```

```
#define BUF_SIZE 1024

int main(int argc, char* argv[]) {
   long long ttlpackets=0;
   int packets=0;
   const char handshake[]="hello";
   int clientsNum=0;
   unsigned short packnum=1;
   char recvline[sizeof(handshake)];
   const char buf[]="Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Proin nec lectus sit amet lorem volutpat elementum. Suspendisse id mollis odio,
id tempus purus. Maecenas non molestie augue. Nulla in semper odio, sed facilisis
nunc. Integer vitae massa nec enim vulputate bibendum. Mauris eget leo nec
diam fringilla congue nec id metus. Phasellus efficitur ultrices ipsum, in hendrerit
nibh aliquet ut. Mauris accumsan odio in magna imperdiet finibus. Aenean
non tortor vel diam suscipit tincidunt. Aenean sed posuere erat. Fusce neque
purus, tincidunt eget eleifend sed, posuere a leo. Maecenas ac neque ut tortor
rutrum imperdiet. Vivamus tempus purus sed orci aliquam, ac venenatis est
feugiat. In lorem augue, lacinia eget suscipit ut, rhoncus sit amet elit. Sed ac
facilisis nunc, non eleifend posuere.";";
   int t=sizeof(buf);
   char buffinal[8000];
   struct sockaddr_in self, other;
   int len = sizeof(struct sockaddr_in);
   int len2;
   int n, s, port;

   s=INVALID_SOCKET;

   WORD versionRequested = MAKEWORD (2, 2);
   WSADATA wsaData;

   int res = WSAStartup (versionRequested, &wsaData);
   if (res != 0)
   {
      throw "class WSInit: WSAStartup ()";
   }

   /* initialize socket */
   if ((s = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
   {
      perror("cannot create socket");
      return 0;
   }

   u_long yes=1;
   if(ioctlsocket(s, FIONBIO, &yes)!=0)
   {
      closesocket(s);
      s=INVALID_SOCKET;
      return 0;
   }

   const char recv_buf[]="900000000";
   if(setsockopt(s,SOL_SOCKET, SO_SNDBUF,recv_buf,sizeof(recv_buf))<0)
   {
      return 0;
   }

   /* bind to server port */
   port = 5151;
   memset((char *) &self, 0, sizeof(struct sockaddr_in));
   self.sin_family = AF_INET;
   self.sin_port = htons(port);
   self.sin_addr.s_addr = htonl(INADDR_ANY);
   if(bind(s, (struct sockaddr *) &self, sizeof(self))<0)
   {
      perror("cannot bind socket");
      return 0;
   }

   memset((char *) &other, 0, sizeof(struct sockaddr_in));
   other.sin_port=htons(port);
   other.sin_family = AF_INET;
   other.sin_addr.s_addr=inet_addr("192.168.100.35");
   std::cout<<"entering main loop";

   ULONG m_timeNow;
   ULONG m_timeBefore=GetTickCount();
   while (true)
   {
      packets++;
      m_timeNow=GetTickCount();
      int senddiff=m_timeNow-m_timeBefore;

      if(senddiff>=1000)
      {
         m_timeBefore=m_timeNow;
         packets=0;
      }

      if(packnum==65535)
      {
         packnum=0;
      }

      packnum++;
      memcpy(&buffinal[0],&packnum,2);
      memcpy(&buffinal[2],buf,510);
      int n=0;

      while(n!=8000)
      {
         n=sendto(s,buffinal, 8000, 0, (struct sockaddr *) &other, sizeof(struct
sockaddr));
      }

      ttlpackets++;
      Sleep(0);
      long long test=ttlpackets*8000;

      if(packets%1000==0)
      {
         system("cls");
         std::cout<<"\nBytes sent: "<<test/(1<<20)<<"MB";
      }
   }

   closesocket(s);
   return 0;
}
```

Appendix 2

```
#pragma comment(lib,"Ws2_32.lib")
#include <stdio.h>
#include <Winsock2.h>
```

```
int main(int argc, char**argv)
{
   long long ttlpackets=0;
   int errors=0;
   int BufLen = 83886080;
   int sockfd,n;
   struct sockaddr_in servaddr,cliaddr;
   char recvline[1400];

   sockfd=INVALID_SOCKET;

   WORD versionRequested = MAKEWORD (2, 2);
   WSADATA wsaData;

   int res = WSAStartup (versionRequested, &wsaData);

   if (res != 0)
   {
      sockfd=INVALID_SOCKET;
      throw "class WSInit: WSAStartup ()";
   }

   sockfd=socket(2,2,0);

   if (sockfd==-1)
   {
      closesocket(sockfd);
      sockfd=INVALID_SOCKET;
      return 0;
   }

   u_long yes=1;

   if(n=ioctlsocket(sockfd, FIONBIO, &yes))
   {
      if(n!=0)
      {
         closesocket(sockfd);
         sockfd=INVALID_SOCKET;
         n=WSAGetLastError();
         n=n;
      }
   }

   ZeroMemory(&servaddr,sizeof(servaddr));
   servaddr.sin_family = AF_INET;
   servaddr.sin_addr.s_addr=inet_addr("192.168.100.1");
   servaddr.sin_port=htons(5151);

   ZeroMemory(&cliaddr,sizeof(cliaddr));
   cliaddr.sin_family = AF_INET;
   cliaddr.sin_addr.s_addr=htonl(INADDR_ANY);
   cliaddr.sin_port=htons(5151);

   if (bind(sockfd,(SOCKADDR*)&cliaddr,sizeof(cliaddr))==-1)
   {
      return 0;
   }

   ULONG m_timeNow;
   ULONG m_timeBefore=GetTickCount();
   unsigned short packnum=0;
   bool firstPack=true;
   int fps=0;
   long long bytecounter=0;
   unsigned short middlecalc;

   bool m_kill=false;

   while (!m_kill)
   {
      if(recv(sockfd,recvline,1400,0)==1400)
      {
         memcpy(&middlecalc,&recvline[0],2);

         if(packnum==65535)
         {
            packnum=0;
         }

         fps++;
         packnum++;

         if(firstPack)
         {
            packnum=middlecalc;
            firstPack=false;
         }

         if(packnum!=middlecalc)
         {
            m_kill=true;
         }

         if(recvline[2]!=-53||recvline[3]!=-2)
         {
            m_kill=true;
         }

      }
   }

   closesocket(sockfd);
   sockfd=INVALID_SOCKET;
}
```

Appendix 3

```
// UDPserver.h
//
#pragma once

class UDPserver
{
public:
   UDPserver();

   ~UDPserver();

   int __cdecl send( const std::vector<unsigned char> & data_in );

   void send_raw(char m)
   {
      m_sendRaw=m;
      heartbeat();
   }

   void set_port(int m)
   {
      m_port_num=m;
      heartbeat();
   }

   void set_bufSize(int m)
   {
      m_storageCap=m;
      heartbeat();
   }

   void set_transmit_unit(short m)
   {
      m_tu=m;
      heartbeat();
   }
private:
   static void callListenThread(void* p)
   {
      reinterpret_cast<UDPserver *>(p)->listenUDP();
   }

   static void callWorkThread(void* p)
   {
      reinterpret_cast<UDPserver *>(p)->workUDP();
   }

   HANDLE m_workerThread,m_listenerThread;
   CRITICAL_SECTION m_criticalSection;

   void workUDP();
   void listenUDP();
   void heartbeat();

   unsigned short m_packnum;
   int m_listSock;
   int m_port_num;
   bool m_kill;
   std::vector<unsigned char> m_sendData;

   //buffer size in bytes
   unsigned int m_storageCap;

   //vectors containing data about the clients
   std::vector<sockaddr_in> m_clientsaddr;
   std::vector<unsigned int> m_clientsAlive;

   //transmission unit in bytes
   unsigned short m_tu;
   unsigned char m_sendRaw;
};

// UDPserver.cpp : Defines the exported functions for the DLL application.
//
#pragma comment(lib,"Ws2_32.lib")
#include "Winsock2.h"

int __cdecl UDPserver::send( const std::vector<unsigned char> & data_in )
{
   unsigned int incomDataSize=0;
   EnterCriticalSection(&m_criticalSection);
   incomDataSize+=data_in[i].size();

   if((m_sendData.size()+incomDataSize)<m_storageCap)
   {
      m_sendData.inset(data_in.begin(),data_in.end());
   } else {
      m_sendData.resize(0);
   }

   LeaveCriticalSection(&m_criticalSection);

   return 0;
}

UDPserver::~UDPserver()
{
   m_kill=true;
   TerminateThread(m_listenerThread,0);
   TerminateThread(m_workerThread,0);
   DeleteCriticalSection(&m_criticalSection);
   closesocket(m_listSock);
}

void UDPserver::heartbeat()
{
   if(m_sendRaw!=999&&m_port_num!=0&&m_tu!=0)
   {
      struct sockaddr_in self;
      int len=sizeof(struct sockaddr_in);

      if ((m_listSock = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
      {
         // Failed to configure connection
         m_kill=true;
      }

      u_long yes = 1;

      memset((char *) &self, 0, len);
      self.sin_family = AF_INET;
      self.sin_port = htons(m_port_num);
      self.sin_addr.s_addr = htonl(INADDR_ANY);

      if(bind(m_listSock, (struct sockaddr *) &self, len)<0)
      {
         // Failed to configure connection
         m_kill=true;
         closesocket(m_listSock);
      }

      if(ioctlsocket(m_listSock, FIONBIO, &yes)!=0)
      {
         // Failed to configure connection
         m_kill=true;
         closesocket(m_listSock);
      }

      const char recv_buf[]="900000000";


if(setsockopt(m_listSock,SOL_SOCKET,SO_SNDBUF,recv_buf,sizeof(recv_buf))<0)
      {
         //memory allocation failed
         m_kill=true;
         closesocket(m_listSock);
      }

      m_listenerThread=(HANDLE)_beginthread(callListenThread,0,this);
      m_workerThread=(HANDLE)_beginthread(callWorkThread,0,this);
   }
}
```

```
void UDPserver::listenUDP()
{
   char recvline[6];
   struct sockaddr_in other;
   int len=sizeof(struct sockaddr_in);

   ZeroMemory(&other,len);

   while(!m_kill)
   {
      EnterCriticalSection(&m_criticalSection);
      int r=recvfrom(m_listSock,recvline,6,0,(struct sockaddr *)&other,&len);
      LeaveCriticalSection(&m_criticalSection);

      if(r==6)
      {

         if(strcmp(recvline,"hello")==0)
         {
            //check if the host is already registered
            bool alreadyWelcomed=false;

            for(unsigned int i=0;i<m_clientsaddr.size();i++)
            {

               if(m_clientsaddr[i].sin_addr.s_addr==other.sin_addr.s_addr)
               {
                  alreadyWelcomed=true;

                  EnterCriticalSection(&m_criticalSection);
                  m_clientsAlive[i]=0;
                  LeaveCriticalSection(&m_criticalSection);
                  break;
               }
            }

            if(!alreadyWelcomed)
            {
               EnterCriticalSection(&m_criticalSection);
               m_clientsaddr.push_back(other);
               m_clientsAlive.push_back(0);
               LeaveCriticalSection(&m_criticalSection);
            }
         }
      }else{
         //this can be more than 1, depending on how often you wish to check for
conections
         Sleep(1);
      }
   }

   _endthread();
}

void UDPserver::workUDP()
{
   char* buffinal=new char[m_tu+2];
   m_packnum=0;

   while(!m_kill)
   {

      if(m_clientsaddr.size()>0)
      {
         int sz;
         EnterCriticalSection(&m_criticalSection);
         sz=m_sendData.size();
         LeaveCriticalSection(&m_criticalSection);

         if(sz>=m_tu)
         {

            if(m_packnum==65535)
            {
               m_packnum=0;
            }
            m_packnum++;

            EnterCriticalSection(&m_criticalSection);

            memcpy(&buffinal[0],&m_packnum,2);
            memcpy(&buffinal[2],m_sendData.data(),m_tu);

            if(m_sendRaw>0)
            {
               m_sendData.erase_front(m_tu);
            }else{
               m_sendData.erase_front(m_tu-2);
            }

            LeaveCriticalSection(&m_criticalSection);

            //send the packets
            if(m_sendRaw>0)
            {
               //send without sequencing
               for(unsigned int i=m_clientsaddr.size();i>0;i--)
               {
                  //check if the client has disconnected
                  //disconnection is detected if the client
                  //has not sent any messages per N packets sent
                  //the smaller the m_tu the higher the N should be
                  int q=m_clientsAlive[i-1];

                  if(q>60000*(10000/m_tu))
                  {
                     m_clientsAlive.erase(m_clientsAlive.begin()+(i-1));
                     m_clientsaddr.erase(m_clientsaddr.begin()+(i-1));
                     continue;
                  }

                  int r=0;

                  EnterCriticalSection(&m_criticalSection);

                  while(r!=m_tu)
                  {
                     r=sendto(m_listSock,&buffinal[2], m_tu, 0, (struct sockaddr
*) &m_clientsaddr[i-1], sizeof(struct sockaddr_in));
                  }

                  m_clientsAlive[i-1]++;
                  LeaveCriticalSection(&m_criticalSection);
               }
            }else{
               //send with sequencing
               for(unsigned int i=m_clientsaddr.size();i>0;i--)
               {
                  int q=m_clientsAlive[i-1];
                  if(q>60000*(10000/m_tu))
                  {
                     m_clientsAlive.erase(m_clientsAlive.begin()+(i-1));
                     m_clientsaddr.erase(m_clientsaddr.begin()+(i-1));
                     continue;
                  }

                  int r=0;

                  EnterCriticalSection(&m_criticalSection);
                  while(r!=m_tu)
                  {
                     r=sendto(m_listSock,buffinal, m_tu, 0, (struct sockaddr *)
&m_clientsaddr[i-1], sizeof(struct sockaddr_in));
                  }

                  m_clientsAlive[i-1]++;
                  LeaveCriticalSection(&m_criticalSection);
               }
            }
         }
      }
   }
   _endthread();
}

UDPserver::UDPserver()
{
   m_tu=0;
   m_sendRaw=255;
   m_port_num=0;
   m_storageCap=16000000;
   m_kill=false;

   InitializeCriticalSectionAndSpinCount(&m_criticalSection, 0x1000);

   WORD versionRequested = MAKEWORD (2, 2);
   WSADATA wsaData;

   int res = WSAStartup (versionRequested, &wsaData);

   if (res != 0)
   {
      //failed to configure connection
      m_kill=true;
   }

   m_listSock=INVALID_SOCKET;
}
```

Appendix 4

```
// UDPclient.h
//
#pragma once

class UDPclient
{
public:
   UDPclient();

   ~UDPclient();

   static void callListenThread(void * ptr)
   {
      reinterpret_cast<UDPclient *>(ptr)->listenUDP();
   }

   int __cdecl recv(std::vector<unsigned char>& data_out);

   void set_port(int m)
   {
      m_port_num=m;
   }

   void set_serv(const char m[])
   {
      m_servaddr.sin_addr.s_addr=inet_addr(m);
      heartbeat();
   }

   void set_bufSize(int m)
   {
      m_storageCap=m;
   }
private:
   HANDLE m_workerThread;
   CRITICAL_SECTION m_criticalSection;
   void heartbeat();
   char * m_servAddr;
   void listenUDP();
   int m_sock,m_port_num;
   bool m_kill;
   std::vector<unsigned char> m_finalData;

   //recieve buffer size in bytes
   unsigned int m_storageCap;
   struct sockaddr_in m_servaddr,m_servaddr2;

   //transmission unit in bytes
   static const unsigned short tu=8802;
};


// UDPclient.cpp : Defines the exported functions for the DLL application.
//
#pragma comment(lib,"Ws2_32.lib")
#include "Winsock2.h"

void UDPclient::listenUDP()
{
   u_long yes=1;
   char yes_c=1;
   unsigned short currpack=0;
   const char handshake[]="hello";
   struct sockaddr_in cliaddr;
   char recvline[tu];
   int len=sizeof(sockaddr_in);
   bool firstConnect=true;
   int n;

   WORD versionRequested = MAKEWORD (2, 2);
   WSADATA wsaData;

   if (WSAStartup (versionRequested, &wsaData) != 0)
   {
      //could not initialize network drivers
      closesocket(m_sock);
      m_kill=true;
```

```
}

m_sock=socket(2,2,0);

if (m_sock==-1)
{
   //could not establish a connection
   closesocket(m_sock);
   m_kill=true;
}

if(n=ioctlsocket(m_sock, FIONBIO, &yes))
{
   if(n!=0)
   {
      //could not establish a connection
      n=WSAGetLastError();
      closesocket(m_sock);
      m_kill=true;
   }
}

if(setsockopt(m_sock, SOL_SOCKET, SO_REUSEADDR,&yes_c, 1)<0)
{
   //could not establish a connection
   n=WSAGetLastError();
   closesocket(m_sock);
   m_kill=true;
}

//windows buffer size
const char recv_buf[]="9000000";

if(n=setsockopt(m_sock,SOL_SOCKET, SO_RCVBUF,recv_buf,sizeof(recv_buf))<0)
{
   if(n!=0)
   {
      //could not allocate memory
      n=WSAGetLastError();
      closesocket(m_sock);
      m_kill=true;
   }
}

m_servaddr.sin_family = AF_INET;
m_servaddr.sin_port=htons(m_port_num);

ZeroMemory(&cliaddr,len);
cliaddr.sin_family = AF_INET;
cliaddr.sin_addr.s_addr=htonl(INADDR_ANY);
cliaddr.sin_port=htons(m_port_num);

if (bind(m_sock,(SOCKADDR*)&cliaddr,len)==-1)
{
   //could not establish a connection
   closesocket(m_sock);
   m_kill=true;
}

unsigned short middlecalc;

while (!m_kill)
{
   //send lifesigns to the server so he knows we are alive
   if(firstConnect||currpack%20000)
```

```
      {
         n=0;

         while(n!=sizeof(handshake))
         {
            n=sendto(m_sock,handshake,sizeof(handshake),0,(struct sockaddr
*)&m_servaddr,len);
         }
      }

      if(recv(m_sock,recvline,tu,0)==tu)
      {
         memcpy(&middlecalc,&recvline[0],2);

         if(currpack==65535)
         {
            currpack=0;
         }

         currpack++;

         if(firstConnect)
         {
            firstConnect=false;
            currpack=middlecalc;
         }

         if(currpack!=middlecalc)
         {
            //packet mismatch
            if(currpack<middlecalc)
            {
               currpack=middlecalc;
            }
         }

         EnterCriticalSection(&m_criticalSection);

         if(m_finalData.size()<m_storageCap)
         {
            m_finalData.push_back(&recvline[2],&recvline[tu-2]);
         }else{
            //overflow handle
            m_finalData.resize(0);
         }

         LeaveCriticalSection(&m_criticalSection);
      }else{
         Sleep(1);
      }
   }
   _endthread();
}

int __cdecl UDPclient::recv(std::vector<unsigned char>& data_out)
{
   if(m_finalData.size()>0)
   {
      EnterCriticalSection(&m_criticalSection);
      data_out.insert(m_finalData.begin(),m_finalData.end());
      m_finalData.resize(0);
      LeaveCriticalSection(&m_criticalSection);
   }else{
      Sleep(1);
   }
   return 0;
}

UDPclient::~UDPclient()
{
   m_kill=true;
   TerminateThread(m_workerThread,0);
   DeleteCriticalSection(&m_criticalSection);
   closesocket(m_sock);
}

void UDPclient::heartbeat()
{
   m_workerThread=(HANDLE)_beginthread(UDPclient::callListenThread,0,this);
}

UDPclient::UDPclient()
{
   InitializeCriticalSectionAndSpinCount(&m_criticalSection, 0x00000400);
   ZeroMemory(&m_servaddr,sizeof(struct sockaddr_in));
   m_kill=false;
   m_port_num=5000;
   m_storageCap=80;
   m_sock=INVALID_SOCKET;
}
```