Project 1
A Simple & Reliable File Transfer Protocol over UDP using Winsock 2.2

Author: **Ioan Raicu**
SS#: **171-72-8942**
Email: **iraicu@cs.wayne.edu**
Course: **CSC6991**

Document Created: 2/22/2001
Last Date Modified: 9/17/2001

Table of Contents

Cover Page – A Simple & Reliable FTP over UDP using Winsock 2.2 1
Table of Contents 2
1.0 Proposal 3
2.0 Implementation 4
2.1 Client 4
2.2 Server 5
3.0 Performance Results 6
3.1 Hardware 6
3.2 Results 6
4.0 Conclusion 8
5.0 Glossary 9
6.0 Works Cited 10

1.0 Proposal

The goal of this project is to simulate a file transfer between a sender and a receiver over a reliable channel using a simple Stop-and-Wait Protocol. This reliable channel is actually an unreliable communication channel (UDP), but through retransmission and acknowledgements, we can make it a reliable communication channel.

In order to get a head start on what I will be doing in Project 3, I decided to implement the system in Windows 2000 using their experimental Ipv6 stack, known as the "Microsoft IPv6 Technology Preview for Windows 2000. The IPv6 stack requires an existing IPv4 stack to be previously installed, and thus retains all functionality between both IP versions.

The good part about the IPv6 implementation that Microsoft created is that they embedded the IPv6 socket creation in the Winsock2.2 API. That means that they added a few more functions when you create the sockets, however, the fundamentals remained the same, and thus a programmer that can make an IPv4 application can most likely learn how to make a simple IPv6 application as well. Microsoft even included a utility called "checkv4.exe" which will scan IPv4 source code and give recommendations on where the code should be changed in order to support IPv6, however, after thoroughly testing it, I came to the conclusion that it does not make enough recommendations in order for everything to just work; there is plenty of programming intervention in order to change an IPv4 application to an IPv6 one.

I will create two applications, a client and a server, using C++ and utilizing Winsock2.2 to send and receive IPv4/6 packets using UDP. I will explain both the sender (client) and receiver (server) in farther detail below.

The sender will read a file, which is specified at the command prompt, cache it, and send packets one at a time, in sequence. It will wait for an acknowledgement for each sent packet, and use an exponential back-off algorithm in order to retransmit. When it finished sending all the packets, it quits the application and presents the user with the transfer's statistics. The server will wait for packets to come in. As they come in, it will check weather or no they are in the right sequence, and the CRC for the payload; if these are OK, it will send back a positive ACK, otherwise, it sends a negative ACK. When it finished receiving all expected packets, it should exit the application.

I plan to do some small experiments in order to see the performance of my system. I will be performing my tests using varying packet sizes on both IPv4 and IPv6. Some of my performance metrics will be: throughput, round trip time (RTT), delay variation, and number of retransmissions.

Through this project, I hope to better my skills with IPv6 socket programming and gain valuable information regarding performance differences between IPv4 vs. IPv6 under Windows 2000. I am sure I will see the drawbacks of a single threaded design approach of the stop-and-wait protocol. This should also get me warmed up for the 3rd project, which I will be testing the difference between IPv4 and IPv6 in much more detail.

2.0 Implementation

2.1 Client

The sender will take various arguments from the command line. They are as follows:

▪ [-s server]	– server name or IP address
▪ [-f family]	– family type; PF_INET - IPv4 or PF_INET6 - IPv6 or PF_UNSPEC for undecided
▪ [-p port]	- port number where to send packets
▪ [-n file name(8.3 format)]	– file to send; the destination filename will be this
▪ [-d delay]	- delay between each packet send in microseconds
▪ [-o debug]	- debug option; 1 for yes; anything else for no.

The client will then read the file into memory. I have a static 2 dimensional character array defined, which has the dimensions of 3000 by the packet size in bytes; therefore, for a 64K packet, the maximum capacity of the array is 187.5 MB, which means it can hold a file up to this size in memory. The client will fragment the file in the specified size messages, and stored in the array for later use. When the client will send packets, it will send a portion of the big array, the i^{th} element in the array, in the order they are found in the array.

Once the client sent the 1st packet, it will start a timer based on the round trip time (RTT) of packets. The initial RTT is calculated using the packet size and the network's capacity. Usually, this is an underestimation due to its optimistic nature. Once a packet has been sent successful, it will adjust the timer's timeout value according to the RTT. If the timer expires before the positive ACKs are received, it retransmits the same message again. At every retransmission, a counter increments, and is used to determine how long it should wait before the next retransmission; this is can be classified as an implementation of an exponential back-off algorithm. After 15 retransmissions, the system will give up and return back to the user. Take note that after every successful send, and of course successful ACK, there is an overall RTT calculated. This RTT is then used to set the timer; this approach allows for varying network conditions, since it will readjust the initial timeout value according to the RTT of previous packets. If the ACK is received before the timeout occurs, the client sends the next packet and starts the whole process all over again with the timer, etc… However, if the ACK was not received before the timeout, the client will retransmit the same packet again.

Each message, before it is sent, is exposed to a CRC algorithm, which is stored in the header. This will be used by the receiver to inspect the payload of the packet and verify that the message is in tact and it is as it was intended by the sender. [20] When the client finished sending all expected packets successfully, it returns back to the user with various transfer statistics.

This client has a debug option, in which it runs the client with the same functionality, but it also outputs a message for every packet sent, ACK received, and retransmissions. The performance of the client will most likely be degraded under this mode due to the blocking nature of displaying to the screen, but is a crucial tool in learning how the system operates.

2.2 Server

The receiver will take various arguments from the command line. They are as follows:

- [-f family] – family type; PF_INET for IPv4 or PF_INET6 for IPv6 or PF_UNSPEC
- [-p port] – port number where to receive packets
- [-o debug] – debug option; 1 for yes; anything else for no.

The server will wait for packets to come in on a specific port. When it gets a packet, it first checks weather it is the expected packet (in sequence), then checks the checkSum from the header using cyclic redundancy check (CRC). If both of these pass as good, then it send back a positive ACK to indicate it has that message. If either of the tests fail, it sends back a negative ACK indicating the packet was either the wrong one or a corrupt packet. The server exits when it received all expected packets.

This also has a debug option, in which it runs the server with the same functionality, but it also outputs a message for every packet received, and a message for every ACK sent. The performance of the server will most likely be degraded under this mode due to the blocking nature of displaying to the screen, but is a crucial tool in learning how the system operates.

3.0 Performance Results

3.1 Hardware

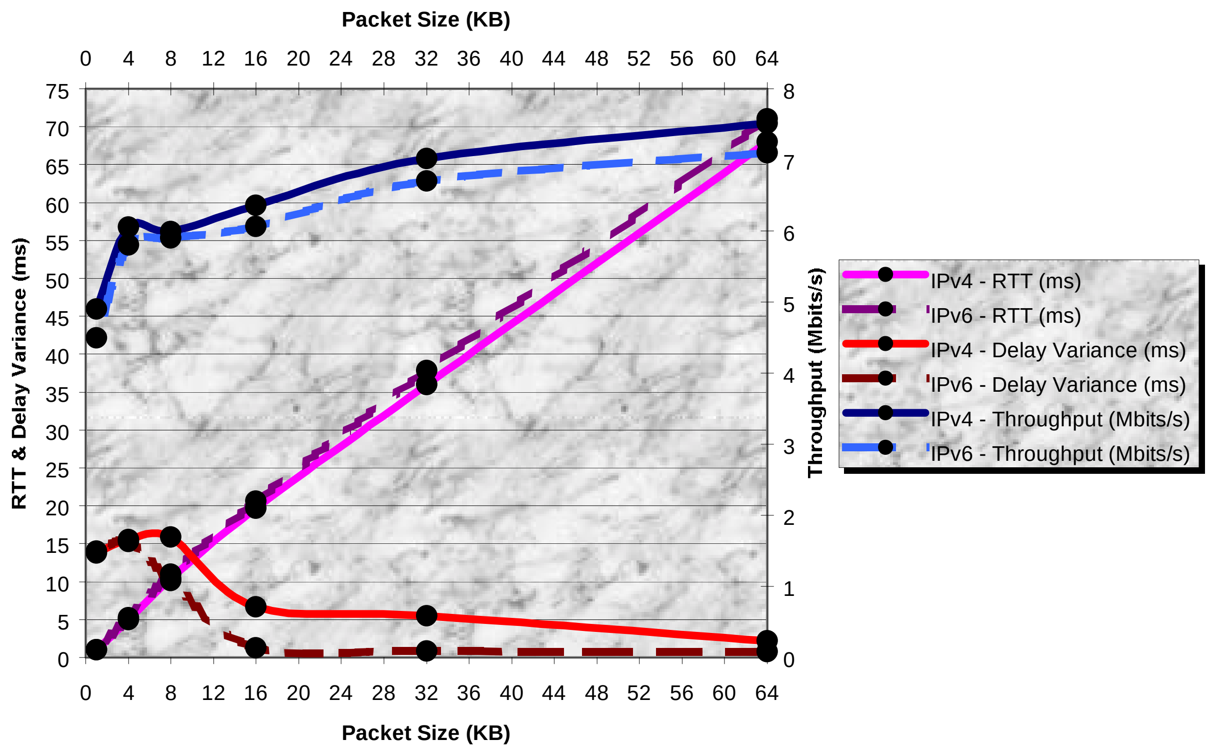
- Workstations
  - Name: sz06, sz07
  - IP addresses: 192.168.1.10, 192.168.2.10
  - OS: Windows 2000 Proffesional, Service Pack 1, IPv6 Stack
  - CPU: PIII 500
  - RAM: 256 MB SDRAM PC100
  - HD: 25 GB, UDMA/66, 5400 RPM
  - NIC: 3COM PCI Etherlink 10/100
- IBM 2216 Nways Multiaccess Connector Model 400

3.2 Results

I ran some experimental transfers, between two computers, utilizing a 26 MB file. Most of these tests took between 30 to 60 seconds to complete, depending on how well it performed. As you can see, the best performance was achieved using 64KB packets under IPv4. Doing a little bit of simple math, we can see that IPv6 is pretty consistently a worse performer by about 5%. It is true that the RTT and throughput are probably related, but the fact remains that IPv6 seems to perform a little worse when it comes to throughput. However, as you can clearly see, the Delay variance is much better on IPv6 than on IPv4. As long as there were no retransmission, IPv6's delay variance was usually within 1 ms, while IPv4's delay variance was usually about 5 ms. I calculated the delay variance by subtracting the actual RTT of each packet to the overall RTT of the system. At the end, I add up all these differences, and divide by the number of packets sent and therefore I get the delay variance. The number of retransmissions were usually 0 for packets greater than or equal to 16KB. The smaller packet transmissions had some retransmissions; that was to be expected, because, the smaller the packet size, the more the software has to work and thus it software becomes a computational bottleneck in terms of how many packets it can process in a very short amount of time.

Family	IPv4	IPv6	IPv4	IPv6	IPv4	IPv6	Family
Packet Size (KB)	**64**	**64**	**32**	**32**	**16**	**16**	**Packet Size (KB)**
Throughput (Mbits/s)	7.51	7.1	7.02	6.7	6.36	6.06	**Throughput (Mbits/s)**
RTT (ms)	68	71	36	37.8	19.7	20.59	**RTT (ms)**
Delay Variance (ms)	2.2	0.794	5.521	0.847	6.7	1.3	**Delay Variance (ms)**
Number of retransmisions	0	0	0	0	0	0	**Number of retransmisions**
Family	**IPv4**	**IPv6**	**IPv4**	**IPv6**	**IPv4**	**IPv6**	**Family**
Packet Size (KB)	**8**	**8**	**4**	**4**	**1**	**1**	**Packet Size (KB)**
Throughput (Mbits/s)	5.99	5.9	6.05	5.8	4.9	4.5	**Throughput (Mbits/s)**
RTT (ms)	10.37	11.01	4.99	5.2	1.008	0.994	**RTT (ms)**
Delay Variance (ms)	15.84	10.13	15.57	15.28	13.77	13.98	**Delay Variance (ms)**
Number of retransmisions	2	5	17	32	172	130	**Number of retransmisions**

IPv4 vs. IPv6



Based on what I have been reading about IPv6, and my performance results, I have concluded that IPv6 has a considerable and noticeable overhead that is greater when compared to IPv4. However, it seems as if the delay variance is much lower, which might indicate that it might be more suitable for delay sensitive applications. There are many things that decide weather a protocol is suited for specific kinds of applications, however, I believe my results a first peek in the performance analysis of IPv6.

4.0 Conclusion

I learned a lot while implementing this system. I acquainted myself even more with Winsock programming. I also got a first hand look at how difficult it is to write some of the low level functionality; in essence, I used an unreliable protocol (UDP) to send a file over the network just as if I was using a reliable connection oriented TCP, but with a little worse performance. When I ran my program using TCP, I believe I was getting about 9.8 Mbits/s for IPv4 and about 9.2 Mbits/s for IPv6. Once again, the 5% difference in performance shows up like usual.

The biggest lesson I learned was assembling the packet, sending it, and disassemble it. I must have spent about 10 hours without exaggeration, manually making the header. By manually, I mean by placing the sequence number of the packet from byte 0 to byte 3, then the total number of packets from byte 4 to byte 7, then the check sum value from byte 8 to byte 12, and so on… All these, including the sending message were placed in a string, which were the ultimate packet to be sent. On the other end, I had to decompose the header once again manually. I ran into many problems, such as having the right data where I was expecting it. I don't believe it was transmission errors, but more likely programming errors, since a design such as the one I was trying to implement was really prone to programming errors. I gave up on working for the night, and went home. I then picked up a UNIX socket programming book, and started reading on how to send messages, just like I was a beginner. I then realized that the sendto() and recvfrom() are not required to have string as its data to be sent or received. I then realized that if I create a structure, lets call it Packet, and I define all the header components including the message, I can send the pointer to this structure over the socket. As long as the same exact structure exists on the other end, all the members of original structure will be sent and reassembled on the other side. When I saw how easy that was, I was ready to shoot myself, but then decided not to since the rest of the project seemed like a piece of cake (but it was still very time consuming).

I spent about 40 hours on this project, which included things like coding, reading (finding solutions to weird problems), etc… It was a very worthwhile experience, and I believe it set me on a very solid path for project 3. I noticed that the new router from Ericsson came in, so I guess I can continue with project 3 ASAP and the rest of my research.

5.0 Glossary

1.	**IPv4**	– Internet Protocol version 4; current version of IP, which was finally revised in 1981; it has a 32 bit address looking like 255.255.255.255, and it supports up to 4,294,967,296 addresses.
2.	**IPv6**	– Internet Protocol version 6; IPv6 is designed as an evolutionary upgrade to the Internet Protocol and will, in fact, coexist with the older IPv4 for some time. IPv6 is designed to allow the Internet to grow steadily, both in terms of the number of hosts connected and the total amount of data traffic transmitted; it will have a 128 bit address looking like FFFF:FFFF:FFFF:FFFF, and it will support up to 340,282,366,920938,463,463,374,607,431,768,211,456 unique addresses.
3.	**IP**	– Internet Protocol; specifies the format of packets, also called datagrams, and the addressing scheme.
4.	**UDP**	– User Datagram Protocol, a connectionless protocol that, like TCP, runs on top of IP networks. Unlike TCP/IP, UDP/IP provides very few error recovery services, offering instead a direct way to send and receive datagrams over an IP network. It's used primarily for broadcasting messages over a network.
5.	**Connectionless**	– Refers to network protocols in which a host can send a message without establishing a connection with the recipient. That is, the host simply puts the message onto the network with the destination address and hopes that it arrives.
6.	**Connection-oriented**	– require a channel to be established between the sender and receiver before any messages are transmitted.
7.	**Unicast**	– sending a message to a specific recipient on a network.
8.	**Bandwidth**	– The amount of data that can be transmitted in a fixed amount of time. For digital devices, the bandwidth is usually expressed in bits per second(bps) or bytes per second. For analog devices, the bandwidth is expressed in cycles per second, or Hertz (Hz).
9.	**Throughput**	– The speed with which data can be transmitted from one device to another. Data rates are often measured in megabits (million bits) or megabytes (million bytes) per second. These are usually abbreviated as Mbps and MBps, respectively.
10.	**Latency**	– the amount of time it takes a packet to travel from source to destination.
11.	**Synchronous**	– processes where data is transmitted at regular intervals; most rigid.
12.	**Asynchronous**	– processes where data can be transmitted intermittently rather than in a steady stream; each party would be required to wait a specified interval before transmitting; most lenient.
13.	**Network**	– a collection of interconnected autonomous computers.

6.0 Works Cited

[1] William Stallings. *High Speed Networks, TCP/IP and ATM Design Principles.*

[2] Andrew S. Tanenbaum. *Computer Networks, 3rd Edition.*

[12] Hinden, Robert M. “IP Next Generation Overview.” May 14, 1995.

[13] Microsoft Corporation. “Introduction to IP Version 6, White Paper”. Pages 1 ~ 38.

[14] Microsoft Corporation. “Microsoft Research IPv6 Release 1.4”. January 13, 2000.

[15] Microsoft Corporation. “Microsoft IPv6 Technology Preview for Windows 2000”. December 12, 2000.

[20] Jenkins, Bob. http://burtleburtle.net/bob/hash/evahash.html 1996