TEL-AVIV UNIVERSITY

FACULTY OF ENGINEERING

School of Electrical Engineering

אוניברסיטת תל-אביב

הפקולטה להנדסה

בי"ס להנדסת חשמל

TCP/IP IMPLEMENTATION

PROJECT NUMBER: 14-1-1-816

FINAL REPORT

AUTHOR

TOM MAHLER 204639926

ADVISOR:

PROF. PATT-SHAMIR TEL AVIV UNIVERSITY

LOCATION IN WHICH THE PROJECT WAS DONE:

ALGORITHMS LAB, ROOM 210, SOFTWARE BUILDING, ENGINEERING FACULTY, TEL AVIV UNIVERSITY

Abstract

The purpose of this project is to provide a simple C++ implemented of a full TCP/IP (layer 3 and layer 4 of the OSI model) stack for educational purposes. The project expanded to also include layer 2, layer 1 and layer 5 thus resulting in the creation of a full networking operating system that I call "`inet_os`".

The implementation is based on the 4.4BSD-Lite2 [1] distribution, a UNIX based operating system, which is implemented in C. Using C++ instead of C was a huge step, however necessary in order to better implement the code, and provide a cleaner implementation.

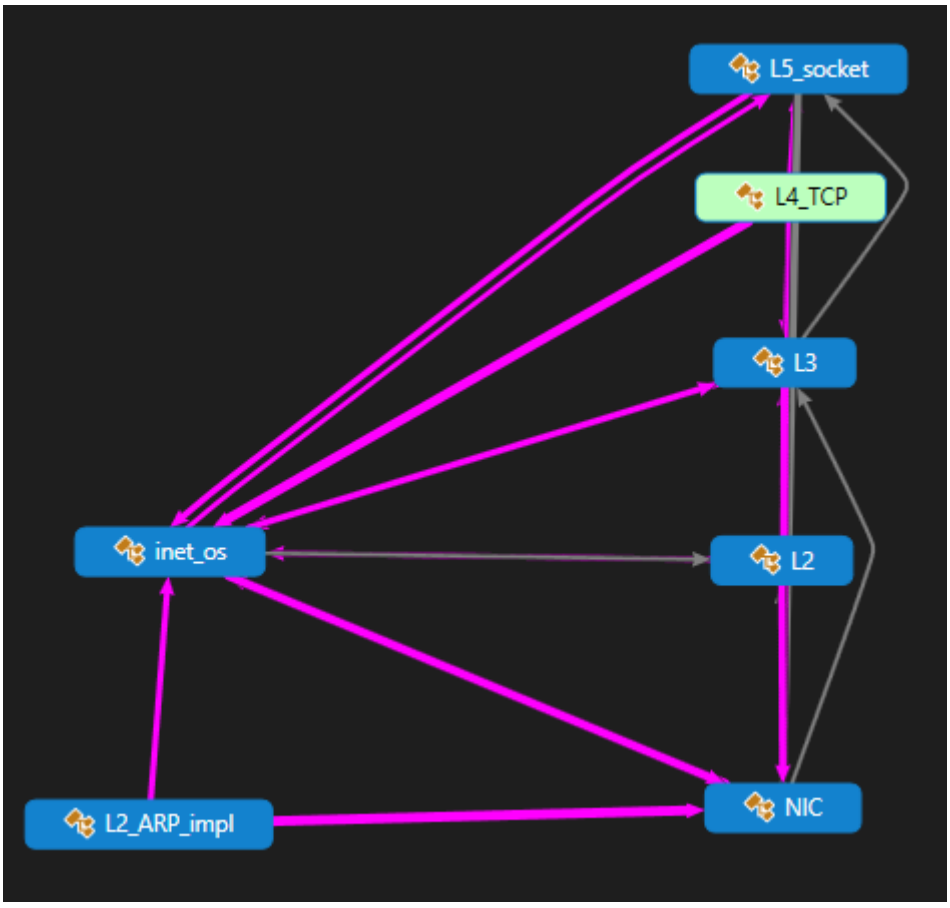


Figure 0.1: inet_os, top view

This figure shows the relations between the layers, under `inet_os`. Incoming data arrives from the NIC module, which pass it upwards to Layer 2 (Ethernet), Layer 3 (IP), Layer 4 (TCP), Layer 5 (sockets) which can pass the data to an application.

Note that the ARP module is not part of the OSI module and acts as a side layer for a specific purpose: resolving network address.

Table of Contents

Table of Figures

1 Introduction

In this project I show the implementation of an entire network OS. I start with the motivation of taking this project, and this document's structure.

I begin explaining the general motivation for taking this project, and my personal motivation as well. For the past year I have been the instructor of the "Advanced Lab in Computer Networks" [2] under the supervising of Prof. Patt-Shamir. The lab was written years ago and was not updated much ever since then, therefore Prof. Patt-Shamir offered me this project.

The goal was to completely rebuild the lab, giving the students the chance to implement the TCP/IP stack themselves. Therefore, the entire design of the project is based on abstraction between the layers and separation into independent layers while breaking down each part so that it can be handed in as a lab for the students.

My personal motivation, other than the satisfaction in building an entire academic course in Tel Aviv University, was the chance to deeply learn and understand the OSI model. The entire internet, which we all use on a daily bases, is very complicated. I believe that understanding how it works, is a major asset that any computer scientists should have. Furthermore, the project gave me hands-on experience in a vast number of topics which I wanted to evolve in, such as programing, OOP, network programing, system design, OS/kernel programing, network traffic analysis and witting an academic document, such as this.

The document is begins with a code review, which is similar to a literature review. I then give the theoretical background which cover the bases of this project. Afterwards, I continue explaining about the implementation, in which I provide a top view of the implementation itself. The 3rd part shows a number of tests and use cases for the code. In the final part, the epilogue, I summarize and discuss future work.

In this document I will be using class names directly. All reference can be found in the appendix [Appendix], or in the attached site. I also assume the reader has knowledge in computer network, and in the OSI model in particular, thus, in some parts, the bases won't be explained in details.

2 Code Review of Existing Designs

There exists various designs and implementation of Network Operating Systems (NOS) and of the TCP/IP stack in particular. In this chapter I review the ones that inspired this project's design and explain why were they chosen and how did they influence the design.

2.1 4.4BSD Lite [3]

The implementation itself, including the functions structure, function names, structs etc. follows a common reference implementation of TCP/IP: the implementation from tile Computer Systems Research Group (CSRG) at the University of California at Berkeley [1]. Historically this has been distributed with the 4.x BSD system (Berkeley Software Distribution). This implementation was first released in 1982 and has survived many significant changes, much fine tuning, and numerous ports to other UNIX and non-UNIX systems. This is not a toy implementation, but the foundation for TCP/IP implementations that are run daily on hundreds of thousands of systems worldwide. This implementation also provides preparation for support to a router functionality expansion module, which will complete the system into a host implementation of TCP/IP and a router.

Specifically, the version of the Berkeley code on which this project is based on is the 4.4BSD-Lite release 2 [1], which source code for the kernel implementation of TCP/IP is approximately 15,000 lines of C code. This code was made publicly available in April 1994, and it contains numerous networking enhancements that were added to the 4.3BSD Tahoe release in 1988, the 4.3BSD Reno release in 1990, and the 4.4BSD release in 1993.

The following figure provides additional details of the various releases of the Berkeley networking code:

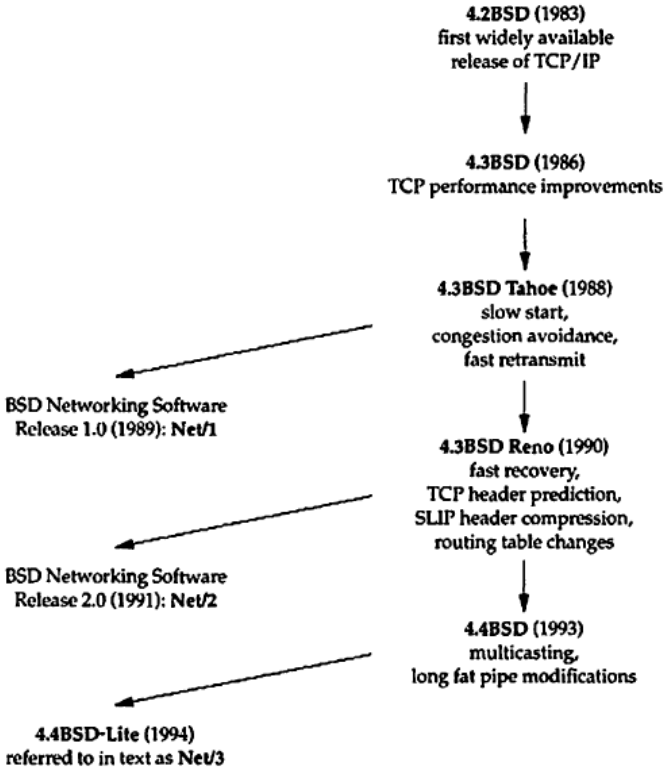


Figure 2.1: Various BSD releases with important TCP/IP features

The influence of the Berkley design on this project is especially strong in the lower layers of the `inet_os`, such as the Ethernet and IP layers, the ARP module and parts of the TCP layer as well. The reason for that is since the Berkley design is very low level and provides the pure foundations on which a networking OS should be built on, such as fulfilling all the requirements of the RFC.

The RFC (Request for Comments) is a name given to a set of documents published by the IETF (Internet Engineering Task Force) and the Internet Society, that define the principal technical development and standards-setting bodies for the Internet. Any system which intend to meet these standard is required to follow these document. As the `inet_os` is a real OS, it was very important for me to make sure these standards are fulfilled.

2.2 FreeBSD

The more advanced parts of the project, such as the TCP layer implementation and the socket layer implementation, is more influenced by the modern branch version of the 4.4BSD Lite, which is called FreeBSD, and in particular the most latest version of it, version 10.2 [4]. This version is very popular and is maintained on-line as a GIT project [5]. As we'll see in Figure 2.2: The Development of BSD below, the famous MAC OSX is also based on this implementation.

FreeBSD's distinguished roots derive from the BSD software releases from the Computer Systems Research Group at the University of California, Berkeley. Over twenty years of work have been put into enhancing FreeBSD, adding industry-leading scalability, network performance, management tools, file systems, and security features. As a result, FreeBSD may be found across the Internet, in the operating system of core router products, running root name servers, hosting major web sites, and as the foundation for widely used desktop operating systems. This is only possible because of the diverse and world-wide membership of the volunteer FreeBSD Project.

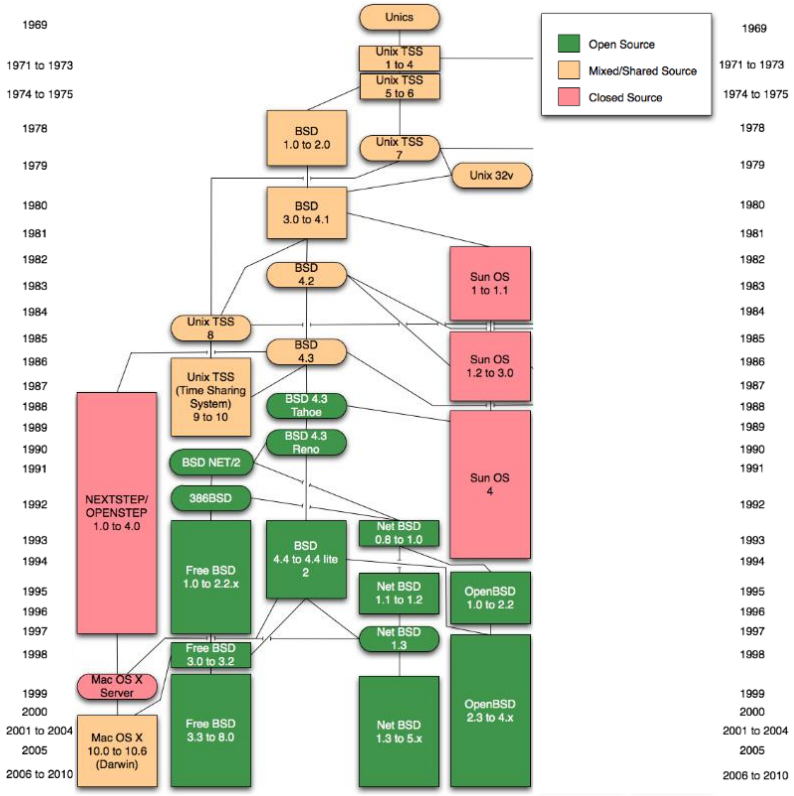


Figure 2.2: The Development of BSD

In this figure we can see an historical summary of the UNIX based system, focusing on the BSD development throughout the years. On the bottom left corner we can see that the famous MAC OS system is actually based on 4.4BSD Lite 2 version.

2.3 Object Oriented Programing

There is no doubt that the original C design works excellent, however, nowadays this design and implementation can be much cleaner, abstract and more efficient. I achieved these goals using Object Oriented Programming (OOP) techniques, which were implemented using C++, and the advanced changes that C++11 introduced in particular [6].

The OOP design introduced many option which the C design lacked. The most fundamental concept that OOP opened up is Abstraction. Using abstraction, I designed abstract classes that represent each of the layers, and an abstract class that represented a general protocol. This feature allows the `inet_os` to use different implementations for the layers, and change implementation in run time.

Though these are very powerful features, the actual intention of the abstraction, is to use the Inheritance principle. The project is designed in such way that students can receive a partial implemented OS, and add their own implementation to complete an assignment. For example, students can receive an OS equipped with layer 1 (NIC), 2 (Ethernet), 4 (TCP) and 5 (sockets) and only fill in the missing 3rd layer (IP).

The other fundamental principle that OOP allowed me to use, is Encapsulation. Since this project is intended to be deployed in the "Advanced Communication Network Lab", I could not reveal the source could to the students. By using abstract classes, and providing only the binaries to the students, they could simply implement the missing functions and run the program without actually knowing how the other parts are implemented. In addition, the students are free to be creative and try different methods rather than stick with my design.

3 Theoretical Background

Though I mentioned that I assume that the reader has knowledge regarding networking, I will supply the necessary background to both complete missing holes that may exist and as a reminder of what each layer does.

3.1 Layer 0: The Wire

Though this may not actually be considered an official layer, since everything in this project is implemented in software, so does the wire, meaning that we have a degree of control over the passing packets. The wire is very simple and may be considered a simple buffer. It supports features such as a random dropping of packets, as well as a random delay of packets based on pre-set supported distribution.

3.2 Layer 1: Network Interface Card

This layer is a simple layer that simply passes the data upwards. It is important to state that this layer is usually responsible for the 4 byte CRC calculation at the end of an Ethernet frame, and the 8 byte preamble at the end:

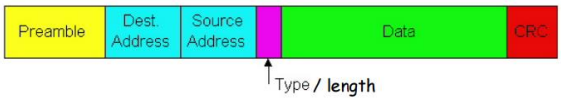


Figure 3.1: Traditional Ethernet Frame Structure [7]

This is usually implemented in hardware (for efficiency), as part of a real network card. This makes it impossible to interfere with this operation, even using pcap. Therefore, I do not take the 4 CRC bytes and the 8 preamble bytes, into account at all and assume that such supported network card is already installed in the host computer.

3.3 Layer 2: Ethernet

This is also a simple layer, which basically parse the Ethernet Header and decides whether the packet is an IP packet, an ARP packet (other link-layer protocols are not supported) or an unsupported packet. In addition, it tests if the packet is a broadcast/multicast, however since both are not supported, these test result in dropping the packet, and are there only for future support to these features. Before sending out the packet, the layer also builds and attaches the Ethernet Header to the beginning of the packet, which space must have been pre-allocated by a higher layer.

3.4 The Address Resolution Protocol

This module resolves the MAC address of the IP address that it is queried for. The algorithm is simple: maintain an ARP cache table (hopefully a very efficient one) which take into account the validation of its entries while satisfying the RFC specification of ARP flood avoidance [8]. If not found, send an ARP request. If the packet is an ARP request for this device, answer accordingly. I elaborate on all steps in the following explanation.

The ARP table is traditionally implemented as a linked list struct, however nowadays, there are much more efficient ways such as a hash table (provided by STL). Before explaining the pros and cons of both suggestions, I talk about the validation of an ARP entry.

There are two aspects to consider: first, we cannot keep an entry in the table forever (both since space is limited and for security reasons), hence we invalidate an entry after a predetermined time period. The second, is to avoid "ARP Flooding" as demanded by RFC 1433 [9]. An Internet device must not send more than 1 ARP packet each second, and after 5 unanswered requests, a timeout of 20 seconds must be waited (the parameters are subjected to change however these are standard time periods).

If an entry is found to be invalid, we must clean up the entry. The linked listed is efficient in that, as cleanup of old entries is done while performing a lookup query. The hash table can support a very quick lookup, however if the entry is invalid it must call some sort of cleanup function that may result in looking through the entire table. As this implementation is for educational purposes, I think that implementing a hash table based on STL (using inheritance from `std::map` [10] for example), provides a more advanced way of solving the problem, appropriate for an advanced lab. However, as the entire base class, including the linked list struct itself, is abstract, any implementation can be chosen. I hope to see creativity from the students in such sections of the project.

Another aspect that I took into account is that when the IP layer tries to send a packet, and the appropriate ARP entry for its destination does not exists (meaning that the ARP module must call the ARP request function), the ARP module holds on to the unresolved packet and keeps it, as it is, in the cache table. After resolving the address, the ARP module must call the Ethernet output interface with the held packet for a retransmission. The ARP module holds up to 1 packet, for the sake of simplicity, however can easily be expanded to a small stack with dynamic or static size.

The transfer of the held packet is done by transferring the pointer, rather than copying the entire data, which is a major inefficiency. Therefore, it is an excellent example of how great smart pointers [11] are: when the ARP module holds on to the pointer to the packet (which is an STL `std::vector`), it returns `nullptr`, which is similar to `NULL` – but smarter [12], to the Ethernet layer. This indicates that the entry was not found and the packet is now held by the ARP module. Since I am using a `shared_ptr` smart pointer type [13], the ownership over that pointer is transferred to the ARP module object. Therefore, after the calling thread continues with the Ethernet layer code, it will eventually terminate. At the termination point, the caller may not know what actually happened with the packet, and if it is clear to free it.

This is a perfect place for a memory leak! A student can "forget" that ARP holds on to that packet and delete the memory, or may have to copy the entire data, inside the ARP module, to avoid that, which is a major drawback for network device which should be as quick as possible.

Smart pointers avoid such memory leaks, since the ownership of the pointer was transferred to the ARP module prior to the thread termination. This means that it is in charge of actually deleting the allocated space, so if the caller thread will try delete the smart pointer, it will simply update the pointer object that the thread does not use the pointer anymore. Now, when the ARP module release the ownership over the pointer, it knows that no one else is using it, and can call the actual delete operator to free the allocated space, without the risk of a memory leak.

3.5 Layer 3: IP

The IP layer is responsible for routing the packet to the desired network using IP addresses and process IP options. Since we do not support routing nor IP options, the layer is atrophied, as it acts similarly to the Ethernet layer. In general: it parses the IP header, tests for various validation and transfer the data to the upper layer, and on the other way it fills up the IP header and sends the packet.

The key aspect of this class, in my implementation, is validation. I must check for a valid CRC and pass the packet to the appropriate protocol (using the `inetsw` proto stack held by the OS that I'll show in 4.1). In addition, I must make sure to switch the representation of the data from network byte order to the host order using winsock's functions, for example, such as `htons()`.

In my implementation, as I provide an abstract class, I made sure to add support for expanding the layer to include routing, multicasting and IP options. A basic implementation for these features exists, however not ready for use. This implementation is left to ease the expanding of this layer and is protected by a set of `MACRO` definitions, as can be seen in the appendix [Appendix].

3.6 Layer 4: TCP

This is the largest layer of all, and requires a good knowledge regarding TCP in order to understand the implementation. Therefore, this section is only a reminder to the key features of this layer and does not replace actual understanding of the protocol.

In addition, note that the separation of TCP Tahoe and TCP Reno is not a major aspect of this layer, as I initially thought it would, and should not be considered as two equal assignments to be handed to the students. I suggest that the TCP Reno can be offered as a bonus and let the students implement only TCP Tahoe, though the difference is not very significant in terms of implementation.

3.6.1 The 3-Way Handshake

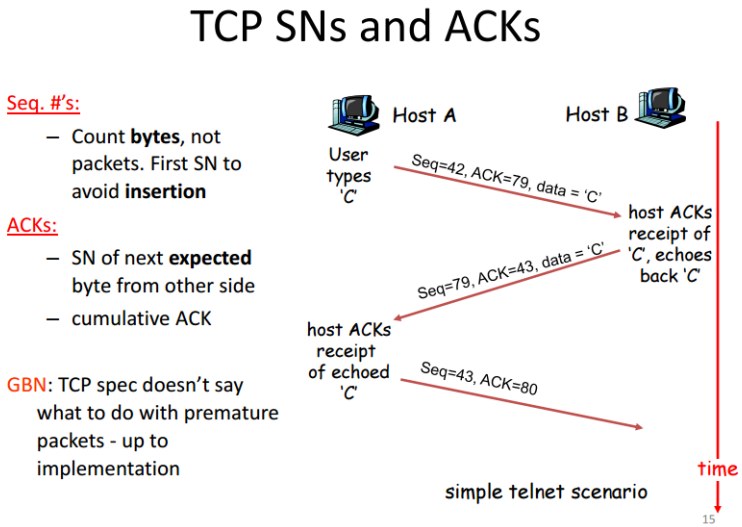


Figure 3.2: The 3-Way Handshake [14]

In Figure 3.2 [14], we can see that when a client request to open a TCP connection with a server it sends a SYN packet to a port which the server listens to. The server replies with an ACK and a SYN and the client finally ACKs that and the connection is established.

3.6.2 TCP State Machine

After a connection is established, each side of the connection maintain 2 buffers (send and receive) and a window of dynamic size. The implementation chosen for these buffers is a circular buffer, which provides very good performance in terms of memory management, and especially reuse of reallocated memory space.

After a packet is sent, the sender expects an ACK for that packet in order to make sure that the packet has arrived. This is a good opportunity to remind that the TCP is a Reliable Transmission Protocol, and thus is responsible to transfer all packets in-order (or at least reassemble out of order packets) to the correct destination. TCP assures that using ACKs and timeouts.

The following figure summarize how and when the window size change and when does a timeout occurs. The figure shows TCP Reno, however includes TCP Tahoe as well:

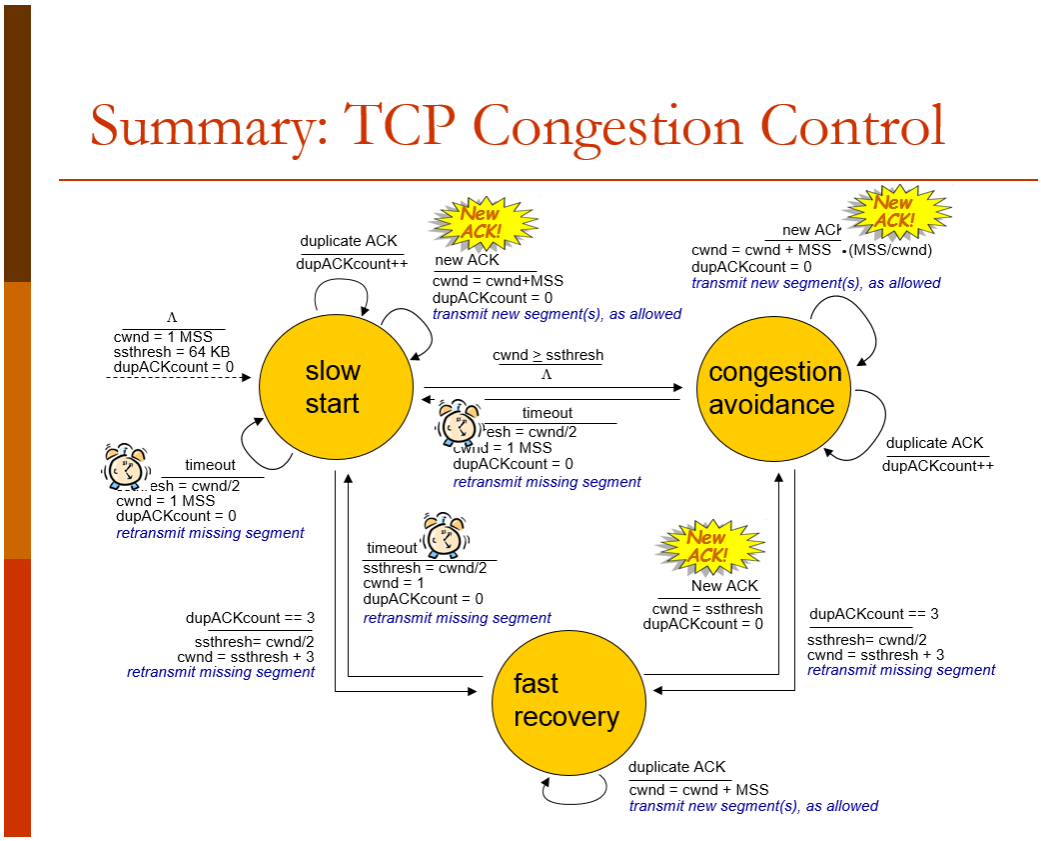


Figure 3.3: TCP State Machine [15]

In Figure 3.3: we can see the 3 states: Slow Start, Congestion Avoidance and Fast Recovery. The Slow Start sets the `ssthreshold` (a parameter in the `tcpcb` class) to be `64KB` (which is the max TCP packet size), and the window to `1 MSS`. I will explain about `MSS` in the following pages.

When ACKs are received with not problems, the `cwnd` grows exponentially: for each ACK we increase `cwnd` by `1`. When `cwnd` passes the `ssthresh`, we switch to the congestion avoidance and increase `cwnd` linearly by `MSS2 / cwnd`.

Problems are marked using timeouts: in TCP Tahoe we have only one kind of timeout, and in TCP Reno we have two kinds of timeout. The common timeout that both TCP implementations hold, takes the state machine back to the Slow Start state, resetting `cwnd` to `1` and `ssthresh` to `cwnd / 2`. In TCP Reno we have another timeout which allows to distinguish between major and minor errors. Major errors, are dealt the same way TCP Tahoe deals with them. Minor errors are maintained using a `dupack` counter which may lead to the Fast Recovery state. In this state, instead of resetting `cwnd` to `1` and starting a new exponential growth, we cut `ssthresh` by half but continue with the linear growth. This improves the performance.

3.6.3 TCP Timers [16]

Timers are maintained using 2 threads, for fast and slow timers. The function `tcp_fasttimo`, is called by the OS every `200 ms` (or other user specified time period). It handles only the delayed ACK timer. A delayed ACK timer is set when TCP receives data that must be acknowledged, but need not be acknowledged immediately. Instead, TCP waits up to `200 ms` before sending the ACK. If, during this `200 ms` time period, TCP has data to send on this connection, the pending acknowledgment is sent along with the data (called piggybacking).

The function `tcp_slowtimo`, is called by the OS every `500 ms` (or other user specified timer). It handles the other six TCP timers: `connection-establishment`, `retransmission`, `persist`, `keepalive`, `FIN_WAIT_2`, and `2MSL`. I will now explain them all:

1. A `connection-establishment` timer starts when a SYN is sent to establish a new connection. If a response is not received within 75 seconds, the connection establishment is aborted.

2. A `retransmission` timer is set when TCP sends data. If the data is not acknowledged by the other end when this timer expires, TCP retransmits the data. The value of this timer (i.e., the amount of time TCP waits for an acknowledgment) is calculated dynamically, based on the round-trip time measured by TCP for this connection, and based on the number of times this data segment has been retransmitted. The `retransmission` timer is bounded by TCP to be between 1 and 64 seconds.

3. A `persist` timer is set when the other end of a connection advertises a window of 0, stopping TCP from sending data. Since window advertisements from the other end are not sent reliably (that is, ACKs are not acknowledged, and only data is acknowledged), there's a chance that a future window update, allowing TCP to send some data, can be lost. Therefore, if TCP has data to send and the other end advertises a window of 0, the `persist` timer is set and when it expires, 1 byte of data is sent to see if the window has opened. Like the retransmission timer, the `persist` timer value is calculated dynamically, based on the round-trip time. The value of this is bounded by TCP to be between 5 and 60 seconds.

4. A `keepalive` timer can be set by the process using the `SO_KEEPALIVE` socket option. If the connection is idle for 2 hours, the `keepalive` timer expires and a special segment is sent to the other end, forcing it to respond. If the expected response is received, TCP knows that the other host is still up, and TCP won't probe it again until the connection is idle for another 2 hours. Other responses to the `keepalive` probe tell TCP that the other host has crashed and rebooted. If no response is received to a fixed number of `keepalive` probes, TCP assumes that the other end has crashed, although it can't distinguish between the other end being down (i.e., it crashed and has not yet rebooted) and a temporary lack of connectivity to the other end (i.e., an intermediate router or phone line is down).

5. A `FIN_WAIT_2` timer. When a connection moves from the `FIN_WAIT_1` state to the `FIN_WAIT_2` state and the connection cannot receive any more data (implying the process called close, instead of taking advantage of TCP's half-close with shutdown), this timer is set to 10 minutes. When this timer expires it is reset to 75 seconds, and when it expires the second time the connection is cropped. The purpose of this timer is to avoid leaving a connection in the `FIN_WAIT_2` state forever, if the other end never sends a FIN.

6. A `TIME_WAlT` timer, often called the `2MSL` timer. The term `2MSL` means twice the `MSL`, the maximum segment lifetime. It is set when a connection enters the `TIME_WAIT` state, that is, when the connection is actively closed. The timer is set to 1 minute (Net/3 uses an `MSL` of 30 seconds) when the connection enters the `TIME_WAIT` state and when it expires, the TCP control block and Internet PCB are deleted, allowing that socket pair to be reused.

3.6.4 TCP Options [16]

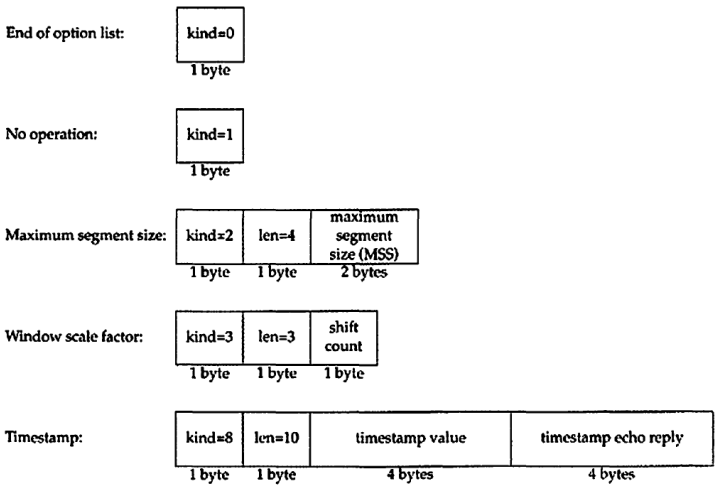


Figure 3.4: TCP Options supported by Net/3 [17]

Every option begins with a 1 byte kind that specifies the type of option. The first two options (with kinds of 0 and 1) are single-byte options. The other three are multi byte options with a `len` byte that follows the kind byte. The length is the total length, including the kind and `len` bytes.

The multi byte integers, the `MSS` and the two timestamp values, are stored in network byte order.

The final two options, window scale and timestamp, are were new in 1995, and therefore were not expected to be supported by many systems. To provide interoperability with these older systems, the following rules apply.

1. TCP can send one of these options (or both) with the initial SYN segment corresponding to an active open (that is, a SYN without an ACK). Net/3 does this for both options if the global `tcp_do_rfc1323` is nonzero (it defaults to 1). This is done in `tcp_newtcpcb`.

2. The option is enabled only if the SYN reply from the other end also includes the desired option.

3. If TCP performs a passive open and receives a SYN specifying the option, the response (the SYN plus ACK) must contain the option if TCP wants to enable the option.

3.6.4.1 Window Scale Option

The window scale option, defined in RFC1323 [18], avoids the limitation of a 16-bit window size field in the TCP header. Larger windows are required for what are called long fat pipes, networks with either a high bandwidth or a long delay (i.e., a long `RIT`).

The 1-byte shift count, in Figure 3.4: TCP Options supported by Net/3 , is between 0 (no scaling performed) and 14. This maximum value of 14 provides a maximum window of 1,073,725,440 bytes (65535 x 214). Internally Net/3 maintains window sizes as 32-bit values, not 16-bit values. The window scale option can only appear in a SYN segment; therefore the scale factor is fixed in each direction when the connection is established.

The two variables `snd_scale` and `rcv_scale` in the TCP control block specify the shift count for the send window and the receive window, respectively. Both default to 0 for no scaling. Every 16-bit advertised window received from the other end is left shifted by `snd_scale` bits to obtain the real 32-bit advertised window size. Every time TCP sends a window advertisement to the other end, the internal 32-bit window size is right shifted by `rcv_scale` bits to give the value that is placed into the TCP header.

When TCP sends a SYN, either actively or passively, it chooses the value of `rcv_scale` to request, based on the size of the socket's receive buffer.

3.6.4.2 Timestamp Option

The timestamp option is also defined in RFC 1323 [18] and lets the sender place a timestamp in every segment. The receiver sends the timestamp back in the acknowledgment, allowing the sender to calculate the RIT for each received ACK. Figure 3.5: Summary of variables used with timestamp option , summarizes the timestamp option and the variables involved.

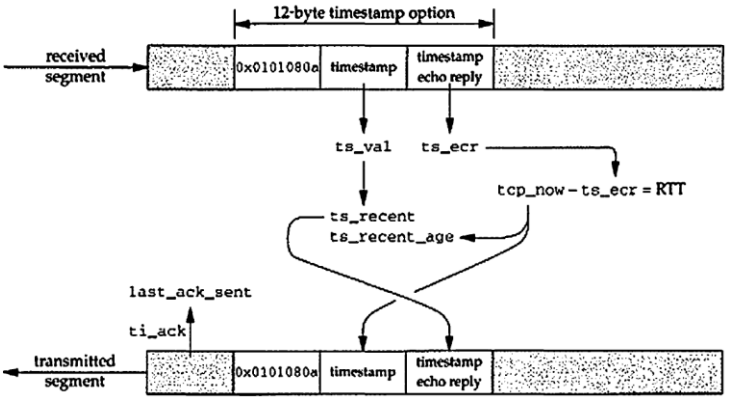


Figure 3.5: Summary of variables used with timestamp option [17]

4 Implementation

In this chapter I will explain the implementation. As my implementation include over 20,000 lines of code, it will not be possible to explain every detail, based on Stevens comment in his book: "Presenting 15,000 lines of source code, regardless of the topic, is a challenge in itself" [19] and the fact that he managed to do so in 1200 pages long book. However, an extremely detailed documentation of the entire code is attached in the appendix [Appendix]. In addition, to ease the navigation through the documentation file, Latex was used for linking, and a site has been built, including a search function.

4.1 The `inet_os`

This is the actual OS that manage the resource of the network card, as well as shared memory and runtime. The design of this class is based on abstract layers and general calls so that it can be used with any layer. It acts as a network OS and can be assigned a virtual IP address, virtual MAC address and more:

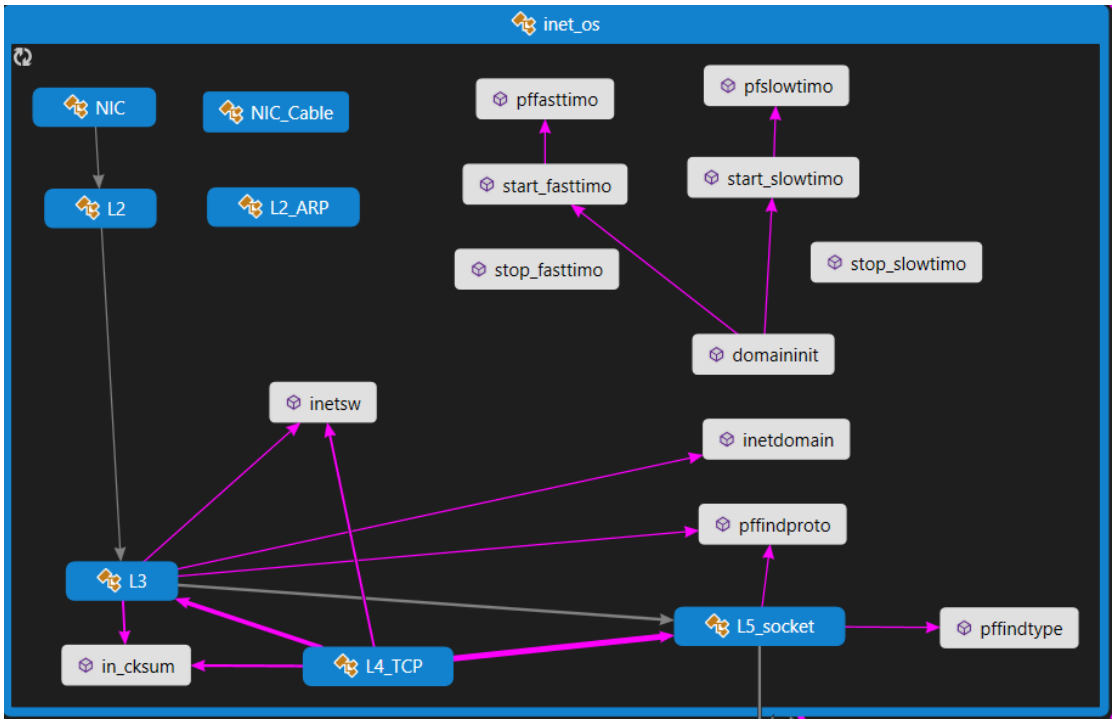


Figure 4.1: `inet_os`, in depth view

This is a detailed figure, and I will explain it in 4 parts:

1. In the upper left corner, we can see that the OS holds 4 low-level layers, which are more of an interfaces than protocols. In a real OS these would be called "devices" or "interfaces" and hence I refer to them the same way.

2. In the bottom left corner we see the protocols part of the OS. The `inetsw` is a static size array which hold pointers to `protosw` classes, which are abstract classes of general protocol. Each implemented protocol must inherit from the `protosw` class, making it possible to assign any user defined protocol implementation to the OS.

3. In the bottom right corner we see the socket layer, which does not need a single managed layer, as all the other layers need, as the sockets are maintained by the application. In addition we can see some find functions to extract a desired protocol from the array.

4. In the upper right corner, we can see timer functions. When a call to `start_XXXtimo` is done, a thread is spawned which periodically (the period is defined in the argument to the function) go over the `inetsw` array and calls the `pr_XXX` function of any protocol. This is used in the TCP timers' implementation. There is also a stop function that allows starting and stopping timers in run-time, which is used as part of the deleting procedure.

Note that this is not an abstract class as I do not expect the students to implement an operating system. Also note that the fact that this is an .hpp file doesn't have any particular reason, and simply was chosen to create a division between .cpp+.h classes, and only .h classes. The code is open to the students to see, as they should be aware of the architecture of this operating system. The `in_checksum` function should not be included as it is part of the IP layer and therefore its implementation is hidden in a .cpp file.

4.2 Layer 1: Network Interface Card

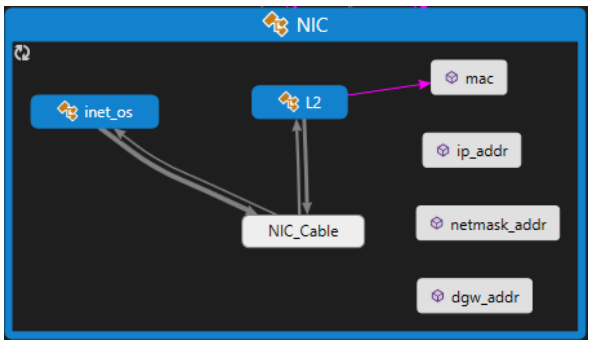


Figure 4.2: Network Interface Card

In this figure, we can see the fundamental relations inside the NIC class. It is held by the `inet_os`, and responsible of the `NIC_Cable` class, which is virtually the cable of the network. This class gets the actual bits from `NIC_Cable`, process them, and pass on to L2 class and similarly, it receives output data, process it, and writes it on the cable. In addition, as this is a virtual device, it holds the device addresses such as: mac address, IP address, Network mask and Default Gateway.

Reads and writes of this kind are usually done through "Raw Sockets" [20], however, Microsoft Windows does not fully support them [11]. Therefore, the actual writes on the cable are done by injecting packets into the real card using WinPcap library [21] which enable this operation. The actual read are more complicated, and the technique chosen to implement them was using a sniffer. The sniffer also supports the BPF (Berkley Packet Filter) standard filter [22] [23] making it very comfortable for debugging. I chose to use Tins sniffer, which showed very good results [24], however means that we must include the POSIX for windows library as well [25].

Following is a more detailed examination of the NIC:

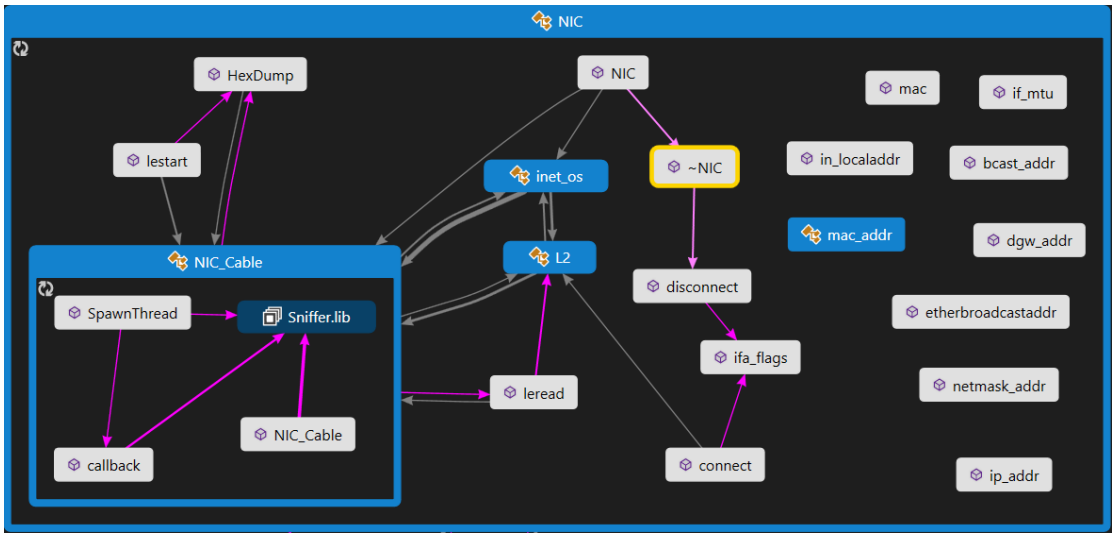


Figure 4.3: Network Interface Card, in depth view

Figure 4.3 shows a more detailed overview of the NIC class. Note the additional members on the right of the figure, and refer to the appendix for more details. When the NIC is created, it is possible to assign to it any MAC address or IP address that the user chooses. In addition, other parameters can be tweaked, but shouldn't. When a NIC is created it automatically notify the OS and adds itself to the OS.

4.2.1 The `NIC_Cable` and the `L0_buffer` Classes

An important module of the NIC class, is the `NIC_Cable` class, which was upgraded at the last moment to also include the `L0_buffer` class:

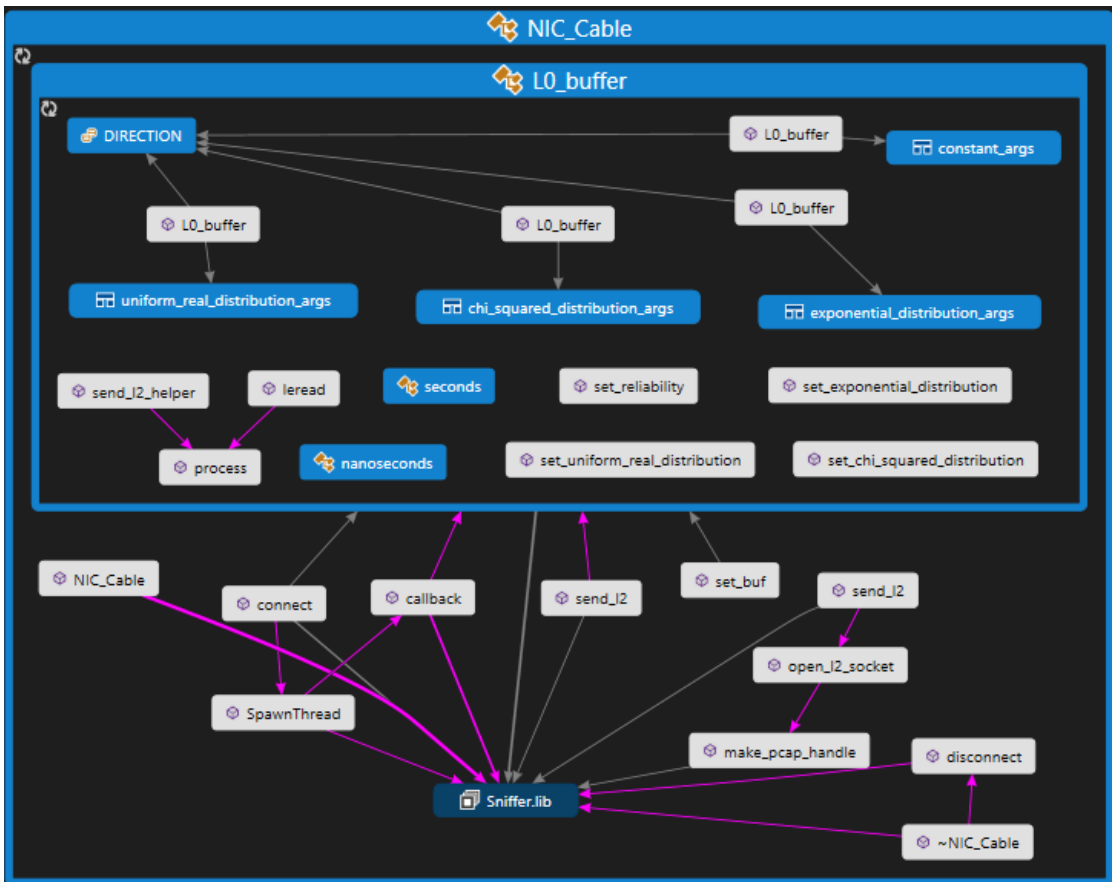


Figure 4.4: `NIC_Cable` and `L0_buffer` Classes

The `NIC_Cable` class maintains the `callback` function which is called for each received packet, meaning that each received packet rides on its own spawned thread, and there cannot be a simulation process of more than one packet.

The `Sniffer.lib` is used for the sniffer implementation. The filter expression can be found at [23]. I suggest to consider allowing the student to use the `mac_addr` class for easy dealing with mac addresses an prints, and to remove the `in_localaddr` that tests if the address is in the subnet or not.

The `L0_buffer` class was added in order to support advanced testing of the TCP layer. Since `inet_os` functions as a real OS, it sends messages on the wire itself. This makes testing packet drops very hard, as I do not have control over the real network. Therefore, the `L0_buffer` is a kind of hack to support such tests.

The user can set the buffer in real time via the `inet_os` class. The buffer can drop packets, based on the `reliability` parameter (that can also be changed in real time). Another feature that the class offer is a delay for packets, based on one of three distributions: `chi-squared`, `exponential` and `uniform`. The parameters are inserted in units of seconds, however the resolution of the random results are up to nanoseconds.

The user can also choose the direction on which the buffer operates: Incoming packets, outgoing packets or both. It is recommended to leave the default of incoming, as the duplicate channel is not necessary.

The buffer is set by the `set_buf` function, and it should be atomic, meaning that the `new` operator should be declared inside the `set_buf` function, as the class takes care of the deletion of the buffer.

4.3 Layer 2: Ethernet

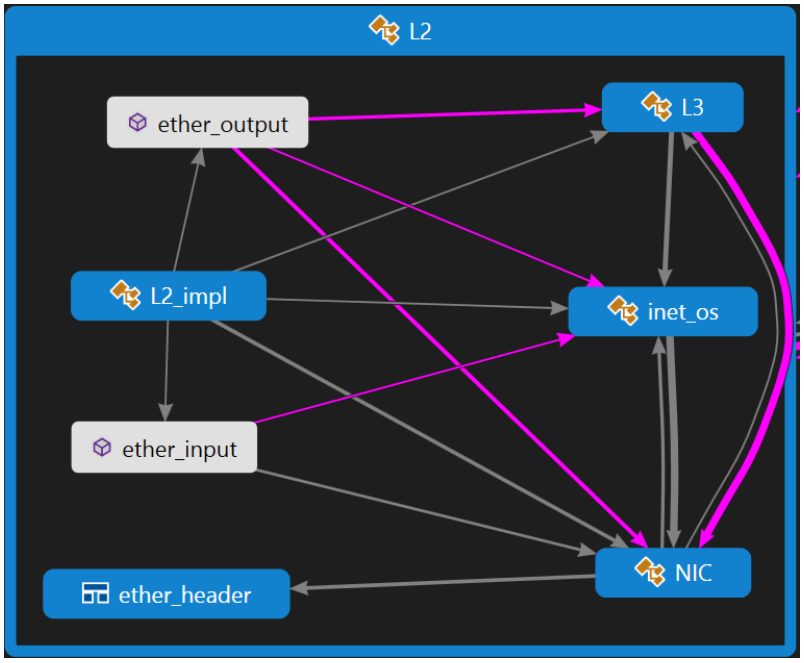


Figure 4.5: Layer 2, Ethernet

In this figure, we can see the relations inside the Layer 2 class. It has 2 main functions that respectively process input (from NIC/L3) and output data (to L3/NIC). This class is intended to be given as a possible assignment to the student, hence it is an abstract class. In order to use this class, one must implement the pure virtual functions, and hence the `L2_impl` class appears. This is an implementation class, in which I implemented the two main functions, as well as all the other helper function that I used (however not necessarily vital to the operation of the layer and hence were not considered pure virtual). Note that the `ether_header` label, refers to a declared struct that must be implemented as well. It is declared in the abstract class since other layers need to use it too.

4.4 The Address Resolution Protocol

The next layer, is not actually an official layer, but more of a module:

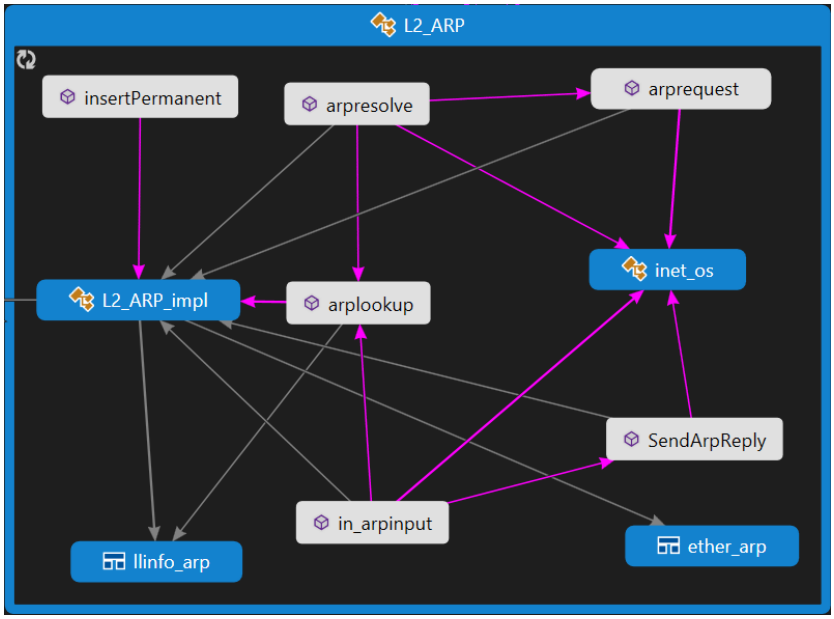


Figure 4.6: The Address Resolution Protocol

In this figure, we can see the relations inside the ARP module class. It has 6 main functions that handle the address resolving. This is also an abstract class which is implemented in the `L2_ARP_impl` class. The names of the function are pretty straightforward. It's important to note that the `llinfo_arp` and the `ether_arp` structs are declared similarly to the `ether_header` struct, that I previously described, and must be implemented in order to use the class. The `llinfo_arp` struct holds the relevant entries for the linked list of the ARP cache table.

In the following figure I open up the `ArpCache` class, as well show a more detailed explanation of the entire module:

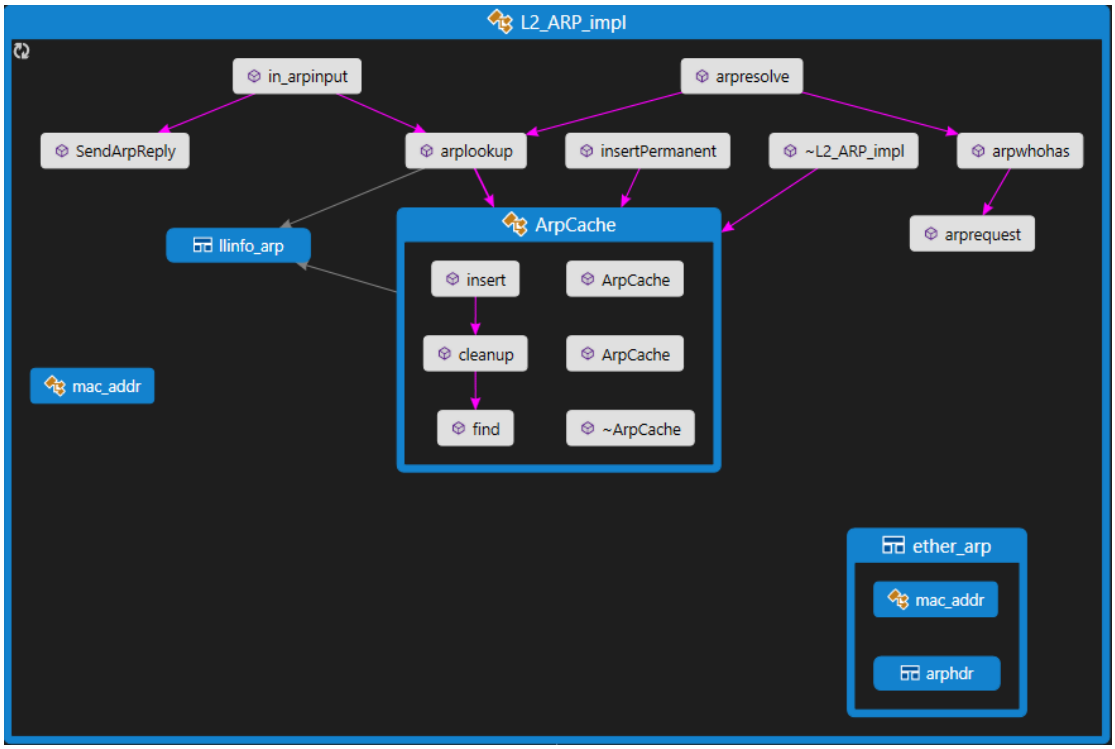


Figure 4.7: Address Resolution Protocol, in depth

I chose to implement the `ArpCache` using STL map. This required an implementation of the `operator=` and several more that allowed support of STL map operations. Each entry receives a `timesamp` of the current time and a counter to count how many times the request was sent.

This is to support the avoidance of ARP flooding. The `cleanup` function is called when an invalid entry is found. This is a very inefficient function which scans the table and removes any invalid entries. The benefits of this data structure, however, is that it is very fast. This is sufficient, for the purposes of this lab, as I do not expect the cleanup function to be called often (if at all).

4.5 Protocols

Before talking about the protocols implemented, I show an abstract definition of what is a protocol:

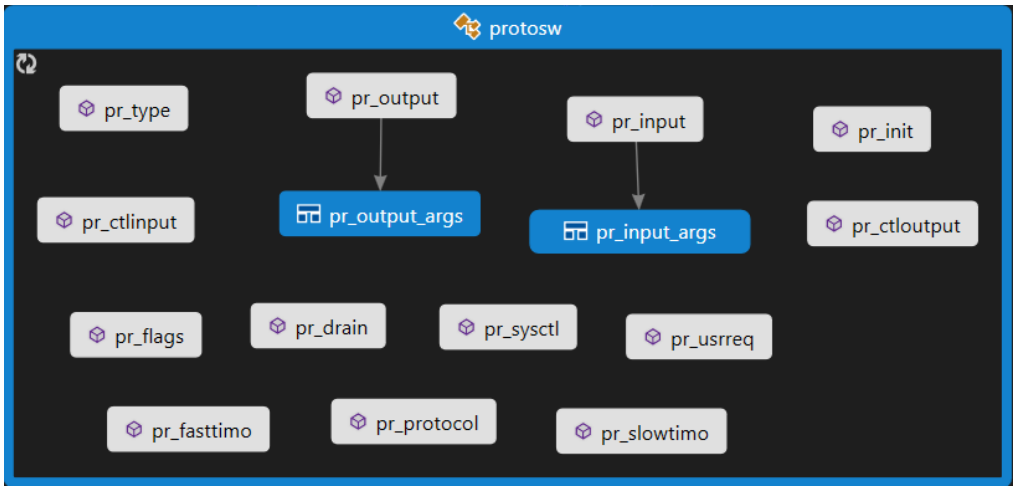


Figure 4.8: Protocols

This abstract class consist of pure virtual functions that grasp anything a general protocol can do. Some of these function are not actually implemented, however this class is important in order to support more protocols that may require these functions. I show a brief of the important functions to implement:

1. All protocols must implement `pr_output` and `pr_input` for processing data, as well as `pr_init` in order to initiate the global class members. Note that the `pr_output_args` is an empty, implemented struct that support inheritance. Hence, it allows an implementation layer to use `dynamic_cast` in order to cast the struct to a derived struct that the implementation layer defined. This hack allows the use of the same function, however with different arguments for each layer. `pr_input` function also supports that, however it is not necessary to do so.

2. `pr_usrreq` is used in Layer 4 protocols such as TCP or UDP. It receives many arguments, however not all of them are used in each call. `pr_fasttimo` and `pr_slowtimo` are used in the TCP and are called by the OS threads [4.1].

I now continue with the implemented protocols.

4.6 Layer 3: IP

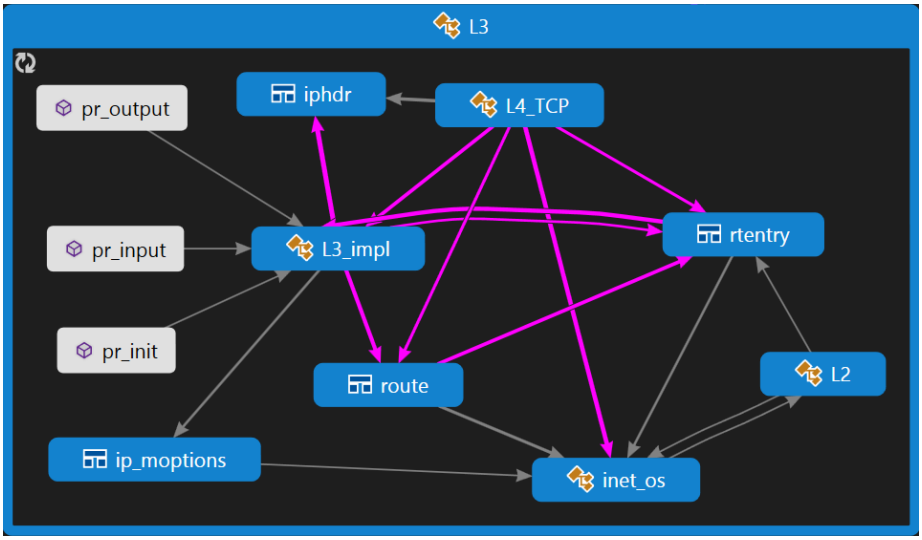


Figure 4.9: Layer 3, IP

In this layer I "silently implemented" the not important pure virtual functions from `protosw`, which means that I moved them to the private part of the class, and trivially implemented them to do nothing or return 0. The 3 functions that we can see in figure are left to be pure virtual and are implemented by the implementation class `L3_impl`. The class also consist of 4 declared structs that must be defined as well. These structs are left to allow future support to routing, we can also see that the upper layer is not called directly, as the layer must take the desired protocol from the OS protocol array [4.54.5 above].

4.7 Layer 4: TCP

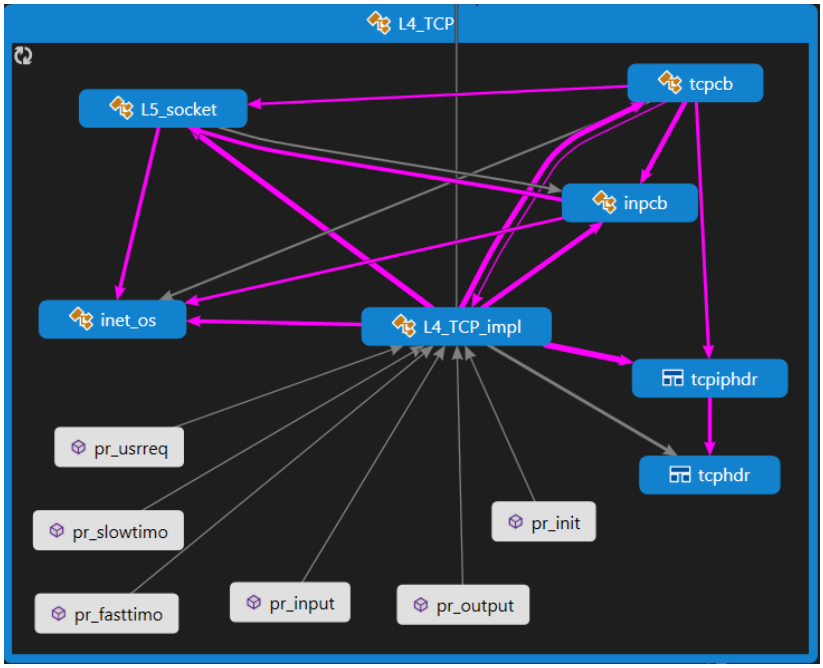


Figure 4.10: Layer 4, TCP

The TCP layer is the biggest layer and extremely complicated. For example, the `pr_input` function is about 2000 lines of code. The structure consist of similar patterns as before: we have the `tcpiphdr` and `tcphdr` structs that are declared, however not defined (they are later defined in the implementation class). We also include the `pr_usrreq`, `pr_slowtimo` and `pr_fasttimo` functions, in addition to the `pr_init`, `pr_input` and `pr_output` functions.

The `pr_usrreq` is very complicated function which purpose is to handle requests from the socket layer regarding a certain connection. The `pr_slowtimo` and `pr_fasttimo` are called by the OS every predefined time (can change based on the parameters in the OS [4.1 above]) and are default set to every `500 ms` for the slow timer, and `200 ms` for the fast timer. Both timers will be discussed later.

In this layer we are also introduced to the `inpcb` and `tcpcb` classes, which stand for internet protocol control block and tcp control block. Both classes are huge and consists of numerous members variables that supply details about the connection. The `inpcb` class is the abstract base class (a matching implementation class is attached as well) and the `tcpcb` class inherit from the implementation class.

There is no need to define a new abstract class `tcpcb` and add a `tcpcb_impl` class, like I did before, since I already have an abstract base class: `inpcb`. Having said that, users should be aware that `tcpcb` must inherit from the implementation class of `inpcb`, and not directly from it, as we will use elements of the implemented `inpcb`, which require the implementation.

The separation between the `inpcb` and `tcpcb` classes was done, since the UDP implementation use only the base `inpcb` class, therefore adding a UDP implementation can use the `inpcb` class, which is already implemented.

4.8 Layer 5: Sockets

This layer was not part of the project, however its implementation allows actually using the TCP layer [4.7 above]. The TCP layer works great without the socket layer, however since TCP is a not a stateless protocol, we must maintain ongoing data regarding the connection.

The TCP layer [4.7 above] introduced the `tcpcb` class for maintain a connection, the socket layer introduce a socket to maintain IO commands for the user. The user can simply call `send` or `recv` function, and the socket layer will pass the data to the TCP layer, instead of the user. I now provide a short introduction to what can be seen in the figure:

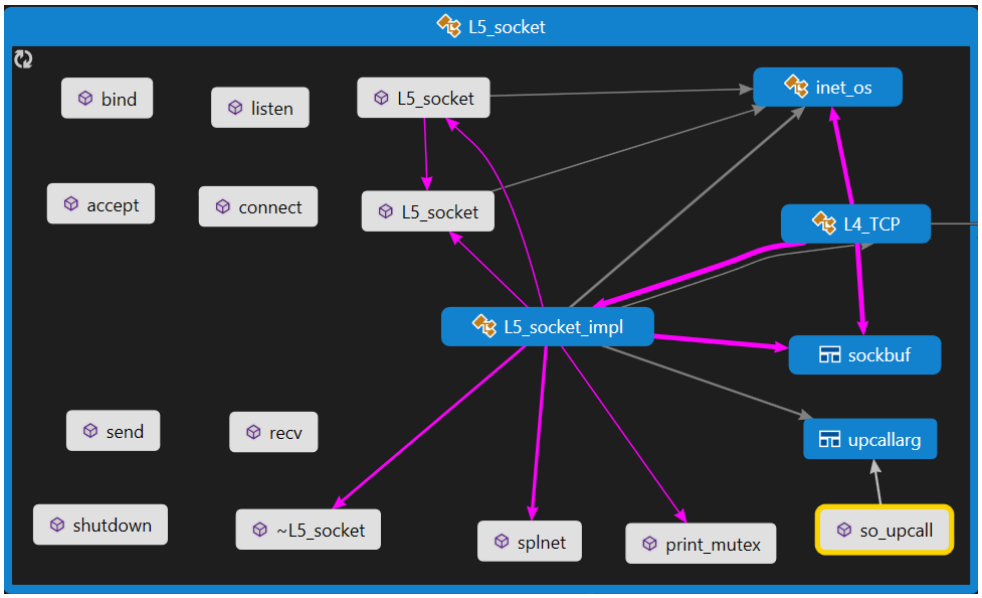


Figure 4.11: Layer 5, Sockets

1. On the upper left corner, we can see the 5 famous system calls for creating a socket and opening a connection. Note that two `L5_socket` system calls (similar to the winsock's "`socket`" call) are supplied, for different purposes. These function are responsible to open up a TCP connection, and maintain the famous 3-Way Handshake.

2. On the bottom left corner, we can see two `send\recv` system calls, as well as two system calls for closing a connection. The first two are blocking, and are responsible to pass the data from the user space, to the OS space (from which the TCP layer operates). This is to support transmission of large files. These two function also receive a "`chunk`" argument, declaring the size of chunks to pass/receive between the user space and the OS space.

   This argument is advised to be played with, as if the default are left, the function may block forever: the scenario that enables this is if we fill the receive buffer with small

chunks, and at the end a chunk is left, which size is less than the `chunk` argument in the function, resulting in waiting forever for more data to arrive.

3. On the bottom middle part, we can see `splnet` and `print_mutex`, both are mutexes (standard STL `std::mutex` [26]). The second one was already in use, however I now explain its purpose. This is the only static OS mutex, and it should be used for debug purposes only. Even in a simple simulation, consisting of just two OS that communicate with one another, we will have at least 6 different threads working: 2x2 for the timers on both OSs, 1 for the main thread, and 1 for the sniffer thread. Therefore, even a simple call for `std::cout` may be cutoff by another thread making it impossible to use printing for debugging, which is a major drawback for the students. Hence, I introduce the `print_mutex`, which locks the `std::cout` prior to the print, and unlocks after.

   The other mutex, `splnet`, is called like that since BSD used an interrupt with the same name. The meaning of this is a regular network interrupt, however I do not use priorities in my implementation, hence it represent a general interrupt. Since I transfer the data from user space to OS space, the implementation of this layer required smart use of an OS locking mechanism.

   The reason for that is that the TCP layer constantly works, processing input and out packets regardless of the user. When a user wants to send something I must make sure to be synchronized with other processes, and especially with the TCP layer (its input processing thread for example, is constantly working). For this I use the OS `splnet` mutex, `std::locks` [27] and `std::lock_guard` [28] (which are standards as of C++11 [6]).

4. The bottom right corner consist of an `upcall` function with its corresponding argument struct. This function is not declared however exists in order to support future calls for upper layers. If upper layers are not supported, this function can be trivially implemented to do nothing.

5. The last part is the `sockbuf` struct, which is declared however not implemented in the abstract class in order to support different implementations. I chose to implement this struct using boost library [29] implementation of a space optimized circular buffer [30]. This buffer is extremely efficient and support the standard STL container operations [31], making it very fast, comfortable and portable. However, this is an independent implementation and may suffer from software bugs. The `sockbuf` struct also maintain its own read\write mutex with a corresponding `std::condition_variable` [32] to support waiting (for example, waiting for a buffer to be free or to not be empty).

4.9 Additional Details

The FreeBSD implementation used the complicated struct called `mbuf` [33]. The struct allows an efficient allocation and use of the memory, however ugly use of `MACROS` and pointers which can easily result in memory leaks. But, we must not forget that in that time they did not have C++11 [6]. To implement an `mbuf`, I am using a smart pointer of type `shared_ptr` [13], and a `std::vector<byte>` container [34]. First, STL containers offer efficiency as well as convenience: memory allocation, support to the `<algorithm>` standard library and more. The smart pointer is used to keep the pointer of the `std::vector`. The smart pointer offer advanced memory control to C++.

The following documentation [13] explain that:

> *"`std::shared_ptr` is a smart pointer that retains shared ownership of an object through a pointer. Several `shared_ptr`objects may own the same object. The object is destroyed and its memory deallocated when either of the following happens:*
>
> - *The last remaining `shared_ptr` owning the object is destroyed.*
> - *The last remaining `shared_ptr` owning the object is assigned another pointer via operator= or reset()."*

This allowed me to use the smart pointers to pass the pointer of the packet vector, and not worry about memory leaks, as once the thread owning the last object keeping the pointer of a packet, terminates, it is automatically destructed. I used an iterator to pass the offset of the current packet so that each layer knows from where to start.

4.10 Debugging

As I implemented a real network device, I used Wireshark [35] for debugging. I addition, as mention, I used prints on the console. Each layer provides a printing function to show the content of the layer. Printing are defined using `MACROS` as can be seen in the appendix.

5 Testing and Use Cases

The testing has been performed using Visual Studio 2013 Ultimate, and using Wireshark. A number of different scenarios will be tested:

1. Resolving an IP address using ARP.

2. Opening a TCP Connection using the TCP 3-way handshake.

3. Sending a small packet using TCP.

4. Sending a large packet using TCP.

5. Closing a connection.

6. Shutting down a connection.

7. Combined Test: Sending a large packet using TCP via an unreliable and delayed channel, using the `L0_buffer` class.

8. Application Use Case: Implementing a simulation of a simple chat server, processing 3 on-line clients, transferring messages between them. In this simulation I will also resolve the mac address using ARP, and hold onto the packet as the ARP module does.

9. Cwnd fall test.

10. Advertise window gets full.

Each test will result in a detailed Wireshark trace file which can be examined. I will test speed, as well as correctness and ease of use. Results will show if the implementation work, and will point out possible bugs.

5.1 Basic Use Case

I will provide short examples, with the source code, of the tests. This also provide a usage example of the functions. Every example starts with creating a few OS machines, giving the IP and MAC addresses as follows:

```
1.          /* Declaring the server */
2.          inet_os inet_server = inet_os();
3.          /* Declaring the server's NIC */
4.          NIC nic_server(
5.                  inet_server,     // Binding this NIC to our server
6.                  "10.0.0.10",     // Giving it an IP address
7.                  "aa:aa:aa:aa:aa:aa",    // Givinig it a MAC address
8.                  nullptr,         // Using my real machine default gateway address.
9.                  nullptr,         // Using my real machine broadcast address.
10.                 true,            // Setting the NIC to be in promisc mode
11.                 "(arp and ether src bb:bb:bb:bb:bb:bb) or (tcp port 8888 and not ether src
12.                             aa:aa:aa:aa:aa:aa)"); // Declaring a filter to make a cleaner testing.
13.
14.         /* Declaring the server's datalink using my L2_impl */
15.         L2_impl datalink_server(inet_server);
16.
17.         /* Declaring the server's arp using my L2_ARP_impl */
18.         L2_ARP_impl arp_server(
19.                 inet_server,    // Binding this NIC to our server
20.                 10,             // arp_maxtries parameter
21.                 10000);         // arpt_down parameter
22.
23.         /* Declaring protocols is a bit different: */
24.         inet_server.inetsw(
25.                 new L3_impl(inet_server, 0, 0, 0),      // A default IP layer is defined, using my L3_impl,
26.                                                                 // as in a real BSD system
27.                 protosw::SWPROTO_IP);  // I place the layer in the appropriate place, though any place should
28.                                                 // do.
29.         inet_server.inetsw(
30.                 new L4_TCP_impl(inet_server), // Defining the TCP Layer using my L4_TCP_impl
31.                 protosw::SWPROTO_TCP);        // Placing it in the appropriate place.
32.         inet_server.inetsw(
33.                 new L3_impl(                  // The actual IP layer we will use.
34.                 inet_server,                  // Binding this NIC to our server
35.                 SOCK_RAW,                     // The protocol type
36.                 IPPROTO_RAW,                  // The protocol
37.                 protosw::PR_ATOMIC | protosw::PR_ADDR), // Protocol flags
38.                 protosw::SWPROTO_IP_RAW);     // Placing it in the appropriate place.
39.
40.         inet_server.domaininit();       // This calls each pr_init() for each defined protocol.
41.
42.         arp_server.insertPermanent(nic_server.ip_addr().s_addr, nic_server.mac()); // Inserting my address
43.
44.         /* Client is declared similarly: */
45.         inet_os inet_client = inet_os();
46.         NIC nic_client(
47.                 inet_client,
48.                 "10.0.0.15",
49.                 "bb:bb:bb:bb:bb:bb",
50.                 nullptr,
51.                 nullptr,
52.                 true,
53.                 "(arp and ether src aa:aa:aa:aa:aa:aa) or (tcp port 8888 and not ether src
54.                             bb:bb:bb:bb:bb:bb)");
55.
56.         L2_impl datalink_client(inet_client);
57.         L2_ARP_impl arp_client(inet_client, 10, 10000);
58.         inet_client.inetsw(new L3_impl(inet_client, 0, 0, 0), protosw::SWPROTO_IP);
59.         inet_client.inetsw(new L4_TCP_impl(inet_client), protosw::SWPROTO_TCP);
60.         inet_client.inetsw(new L3_impl(inet_client, SOCK_RAW, IPPROTO_RAW,
61.                             protosw::PR_ATOMIC | protosw::PR_ADDR), protosw::SWPROTO_IP_RAW);
62.         inet_client.domaininit();
63.         arp_client.insertPermanent(nic_client.ip_addr().s_addr, nic_client.mac()); // My
64.
65.         /* Spawning both sniffers, 0U means continue forever */
66.         inet_server.connect(0U);
67.         inet_client.connect(0U);
68.
69.         // The socket address to be passed to bind
70.         sockaddr_in service;
71.
```

```
72.         //----------------------
73.         // Create a SOCKET for listening for
74.         // incoming connection requests
75.         netlab::L5_socket_impl *ListenSocket(new netlab::L5_socket_impl(AF_INET, SOCK_STREAM, IPPROTO_TCP,
76.                                                 inet_server));
77.
78.            //----------------------
79.         // The sockaddr_in structure specifies the address family,
80.         // IP address, and port for the socket that is being bound.
81.         service.sin_family = AF_INET;
82.         service.sin_addr.s_addr = inet_server.nic()->ip_addr().s_addr;
83.         service.sin_port = htons(8888);
84.
85.         //----------------------
86.         // Bind the socket.
87.         ListenSocket->bind((SOCKADDR *)&service, sizeof(service));
88.
89.         //----------------------
90.         // Listen for incoming connection requests
91.         // on the created socket
92.         //
93.         ListenSocket->listen(5);
94.
95.         //----------------------
96.         // Create a SOCKET for connecting to server
97.         netlab::L5_socket_impl *ConnectSocket(new netlab::L5_socket_impl(AF_INET, SOCK_STREAM, IPPROTO_TCP,
98.            inet_client));
99.
100.        //----------------------
101.        // The sockaddr_in structure specifies the address family,
102.        // IP address, and port of the server to be connected to.
103.        sockaddr_in clientService;
104.        clientService.sin_family = AF_INET;
105.        clientService.sin_addr.s_addr = inet_server.nic()->ip_addr().s_addr;
106.        clientService.sin_port = htons(8888);
107.
108.        //----------------------
109.        // Connect to server.
110.        ConnectSocket->connect((SOCKADDR *)& clientService, sizeof(clientService));
111.
112.        //----------------------
113.        // Create a SOCKET for accepting incoming requests.
114.        netlab::L5_socket_impl *AcceptSocket = nullptr;
115.
116.        //----------------------
117.        // Accept the connection.
118.        AcceptSocket = ListenSocket->accept(nullptr, nullptr);
119.
120.        int str_size(1024);
121.        std::string send_str(str_size, 'T');
122.        std::string ret("");
123.
124.        ConnectSocket->send(send_str);
125.
126.        AcceptSocket->recv(ret, str_size);
```

All of the test are located in the test file, as functions with a name corresponding to the test number.

5.2 Resolving an IP address Using ARP

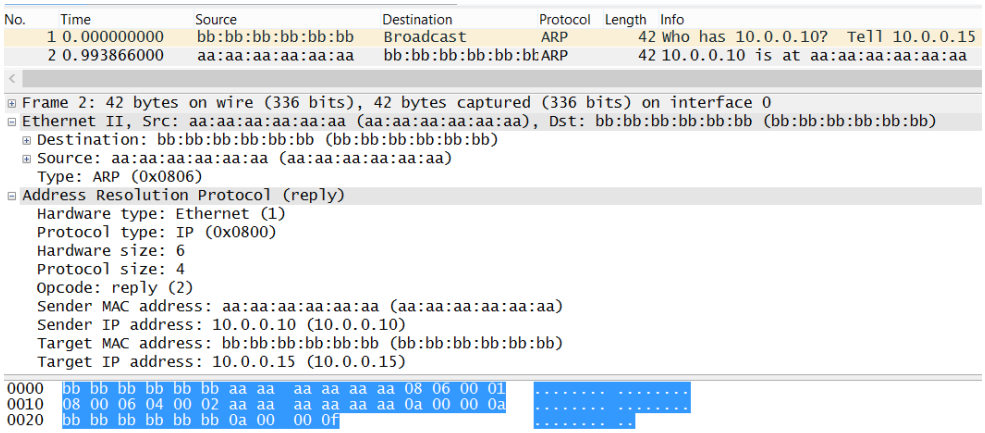


Figure 5.1: Test#1, resolving an IP address using ARP

We can see that the address is resolved successfully and quickly (1 second). This is not as quickly as we would expect from a real system, however for the purposes of this project it should be sufficient.

5.3 Opening a TCP Connection Using the TCP 3-way Handshake

Now that we've seen that the ARP works, I add the following lines that manually resolve the address on system initialization (however can call these function anywhere else as they are meant to be used during run-time):

```
1.          arp_client.insertPermanent(nic_server.ip_addr().s_addr, nic_server.mac()); // server
2.          arp_server.insertPermanent(nic_client.ip_addr().s_addr, nic_client.mac()); // client
```

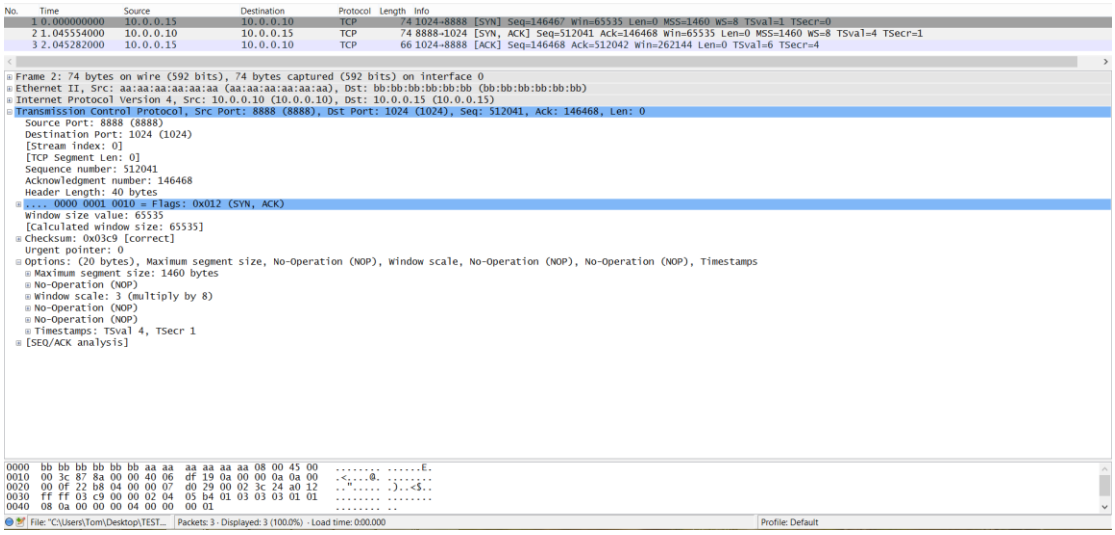


Figure 5.2: Test#2, Opening a TCP Connection using the TCP 3-way handshake

```
|Time      | 10.0.0.15                                |
|          |                    | 10.0.0.10           |
|0.000000000|          SYN       |                     |Seq = 146467
|          |(1024)    ------------------>   (8888)    |
|1.045554000|          SYN, ACK  |                     |Seq = 512041 Ack = 146468
|          |(1024)    <------------------   (8888)    |
|2.045282000|          ACK       |                     |Seq = 146468 Ack = 512042
|          |(1024)    ------------------>   (8888)    |
```

Figure 5.3: Test#2, Wireshark TCP Flow Graph

In the TCP Flow graph we can see that the 3-way handshake operates as expected and the numbers are correct. Wireshark marks wrong, and show warning for various errors. Note that in Figure 5.2: Test#2, Opening a TCP Connection using the TCP 3-way handshake, Wireshark does not detect errors in the packets, meaning that they are valid TCP packets. Also note, that the SYN packet includes the MSS and Window Scale options, and that the server successfully agrees to these options.

5.4 Sending a Small Packet Using TCP

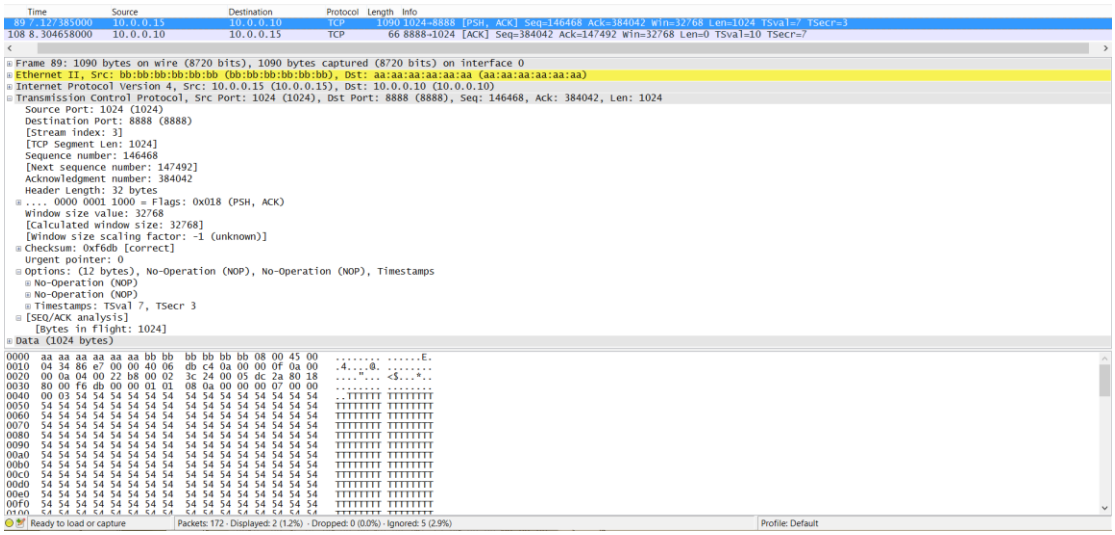


Figure 5.4: Test#3, sending a small packet using TCP

```
|Time      | 10.0.0.15                                 |
|          |                      | 10.0.0.10          |
|0.000000000|          PSH, ACK - Len: 1024            |Seq = 146468 Ack = 384042
|          |(1024)    ------------------->   (8888)    |
|1.177273000|          ACK        |                    |Seq = 384042 Ack = 147492
|          |(1024)    <-------------------   (8888)    |
```

Figure 5.5: Test#3, Wireshark TCP Flow Graph

Here I send a 1MB long packet, consists of 1024 chars of 'T', after a connection was made. Note that the second ACK is exactly: `147492 = 146468 + 1024` as we expect from a TCP flow.

Also, note that the PSH flag is on. This is since we're sending everything we've got from the send buffer. This is intended for receiving systems that only pass received data to an application when the PSH flag is received or when a buffer fills. However, Net/3 never holds data in a socket receive buffer waiting for a received PSH flag, thus setting it on should just provide more support.

5.5 Sending a Large Packet Using TCP

We show an example of sending 256MB of data (the maximum buffer size of a socket). Since the trace is very long, I will avoid attaching screen shots, and present graphs instead:

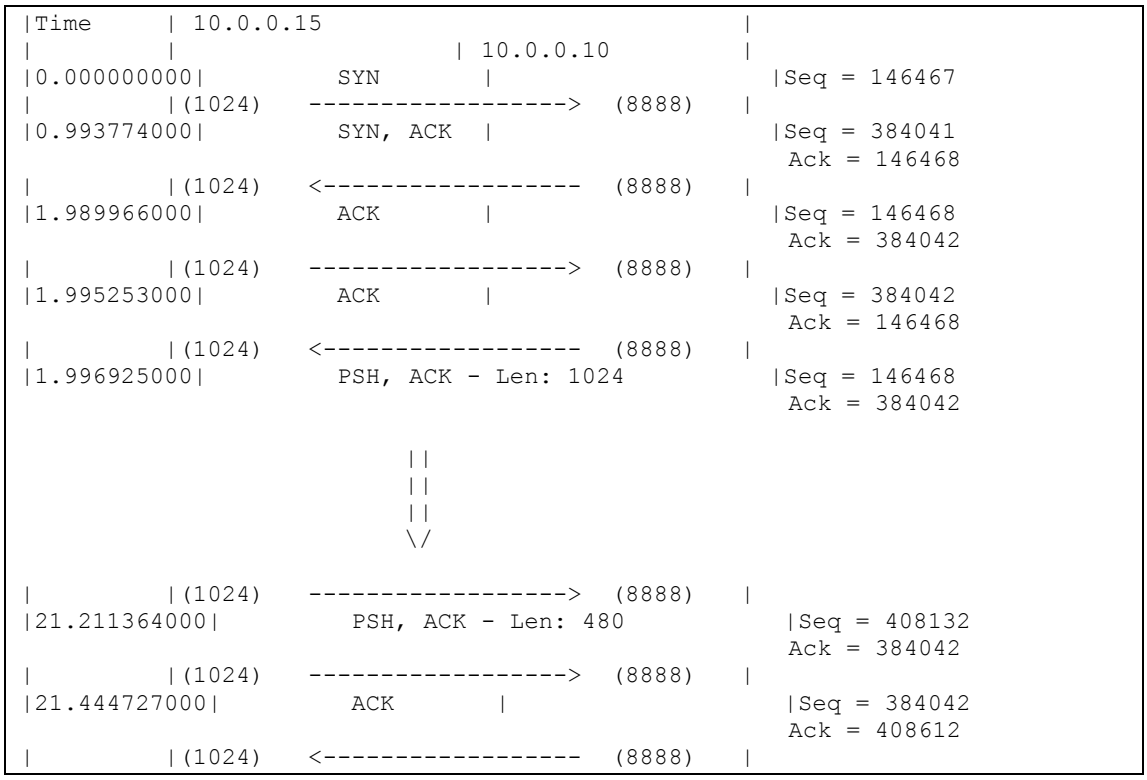


Figure 5.6: Test#4, Wireshark TCP Flow Graph

I validate, again, that the entire packet was sent by looking at the first sequence number and comparing it to the last sequence number minus the length of the sent packet: `146468=408612-262144`. The entire trace is attached in the TEST directory.

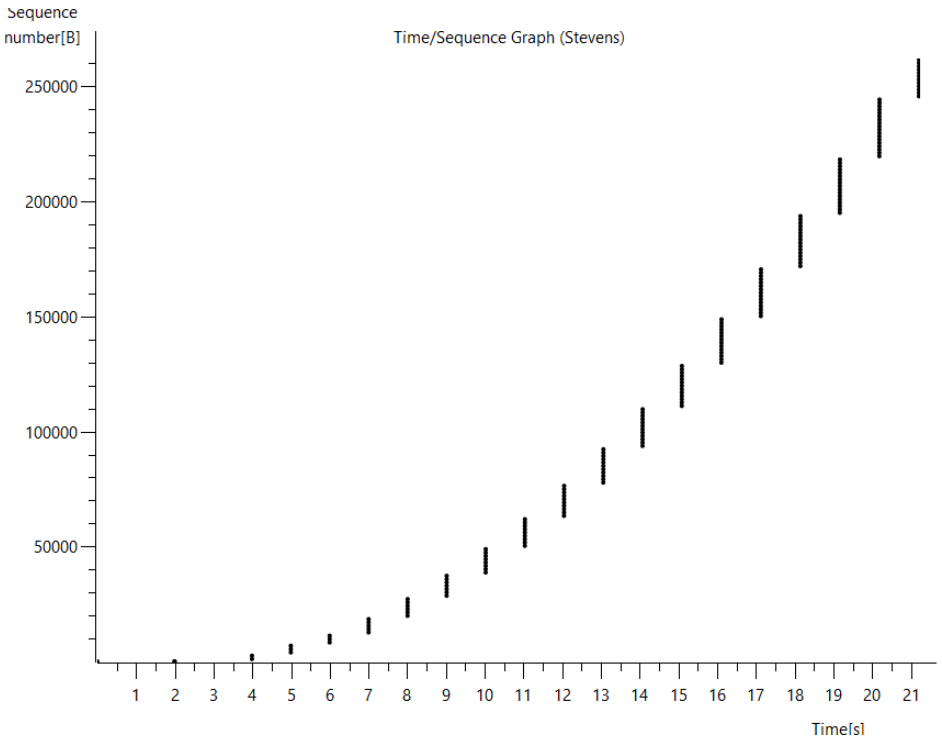


Figure 5.7: Time/Sequence Graph

This is literally a perfect time/sequence graph! We can see the slow start growing as expected. Since we tested a simple send, there is no drops.

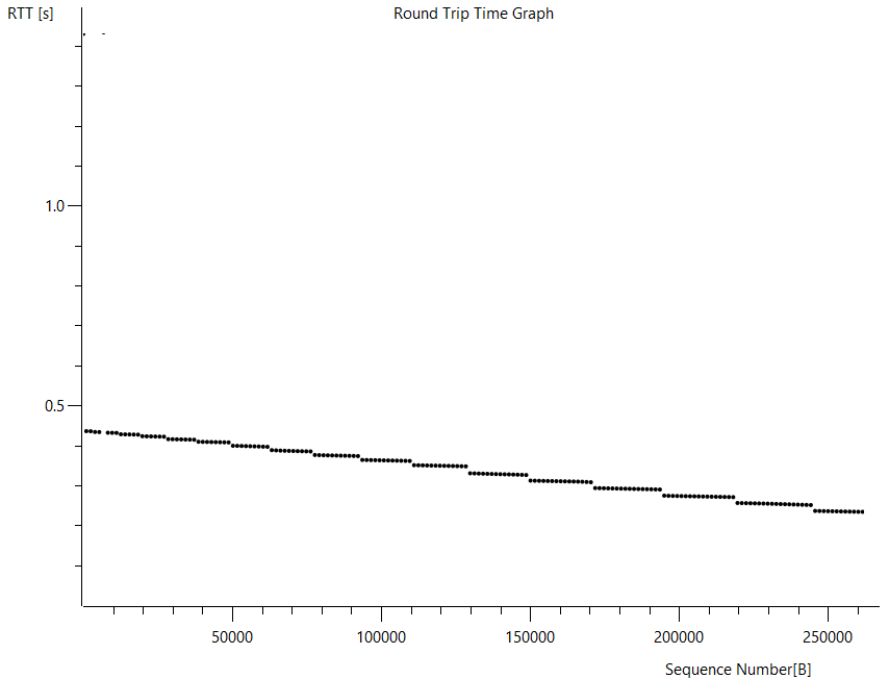


Figure 5.8: Test#4, Round Trip Time Graph

The RTT graph show that the RTT estimators that are implemented, show better results as time goes on, which shows that they are a benefit to the TCP layer.

It is interesting to see why the printout should only be used for debug purposes in the following graph (in which I enabled the printouts):

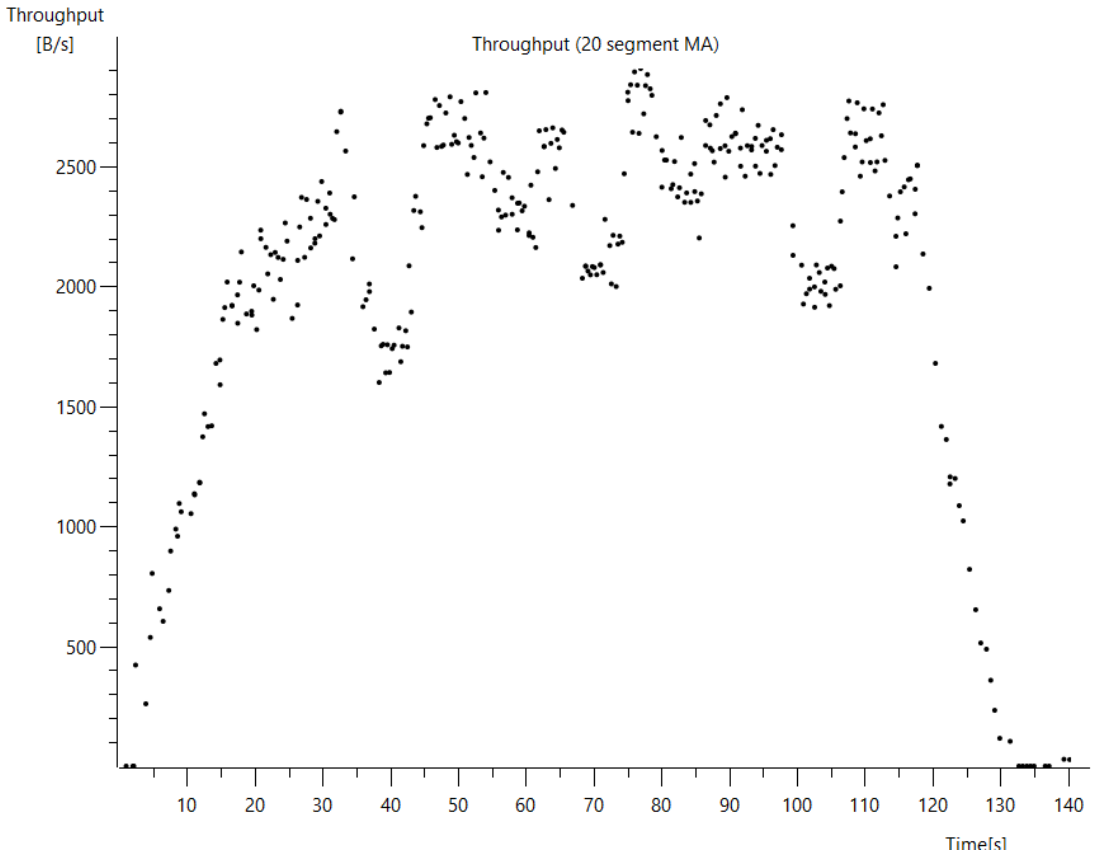


Figure 5.9: Test#4 with prints, Throughput

Before explaining the differences, I remind that each print call requires locking the static print mutex, and this blocks the entire system. Since prints are conducted in each layer, even if they are very quick, they damage the normal network flow and damage the performance.

In Figure 5.9: Test#4 with prints, Throughput we can see the decrease in performances: the total send time is increased by a factor of 7 which is huge in terms of networking. In addition we can see the instability of the graph, compared with the results disabling the printouts. The effect can be also seen via the RTT graph:

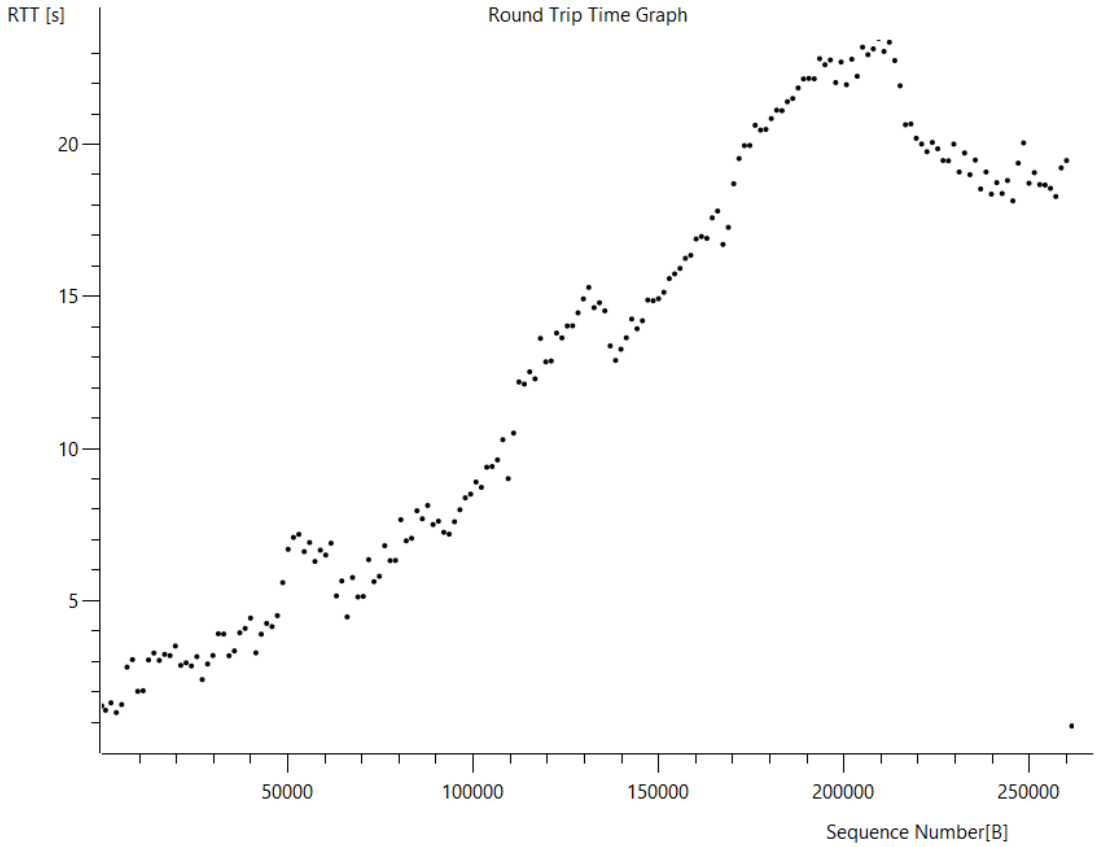


Figure 5.10: Test#4 with prints, Round Trip Time Graph

In this graph we can see that the RTT estimators calculate drastically different results, over 40 times higher than without printouts! This happens since the print mutex hack is not natural and severely damage the proper functionality of both systems.

5.6 Closing a TCP Connection

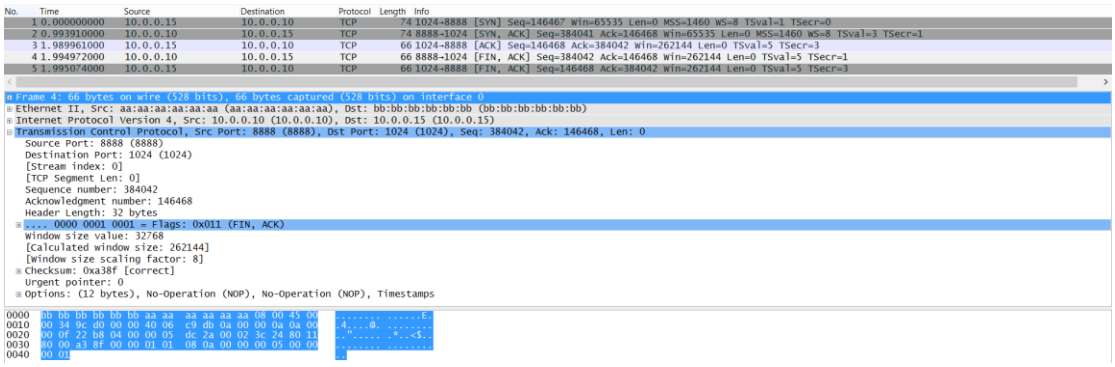


Figure 5.11: Test#5, Closing a TCP Connection

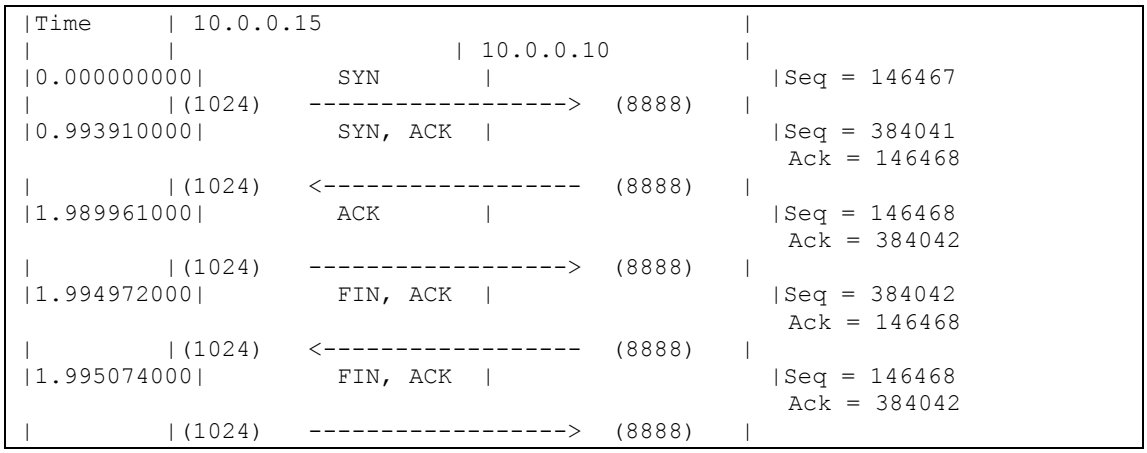


Figure 5.12 Test#5, Wireshark TCP Flow Graph

The close of a TCP connection, means using the delete operator on a created socket. The results are successful, and show a correct closing of a connection.

It is important to understand that since the timer threads are constantly working, they try to access a `tcpcb` object and see if it has anything to send. Therefore, if we want to delete a socket (which will result in deleting a `tcpcb` object) we must first stop the timers. Stopping the timer is the system way of synchronizing the timer threads, and the user thread.

The following code is an example of how to properly close a socket using the delete operator:

```
1.          inet_client.stop_fasttimo();
2.          inet_client.stop_slowtimo();
3.
4.          inet_server.stop_fasttimo();
5.          inet_server.stop_slowtimo();
6.          std::this_thread::sleep_for(std::chrono::seconds(1));
7.
8.          delete ConnectSocket;
9.          delete AcceptSocket;
```

Notice the `std::this_thread::sleep_for` [36] that is called. This is since the stop function simply sets a Boolean member from "true" to "false" which result in exiting the while loop that runs in each timer. The meaning of this is that once we call the stop function, it can take up to `500 ms` (if the default slow timer was not changed) to actually stop. A safety step was taken by waiting 1 second.

After calling delete, the user may restart the timers to continue with the normal operation.

#Important: If the stop timer function won't get called, the program may crash as the timer thread will try to access a bad memory (a `tcpcb` object that does not exists).

5.7 Shutting Down a TCP Connection

In order to shut down a connection, and test of it was indeed shut down, I introduce the following test:

```
1.          ConnectSocket->shutdown(SD_SEND);
2.          try
3.          {
4.                  ConnectSocket->send(string(1024, 'T'));
5.          }
6.          catch (runtime_error &e)
7.          {
8.                  cout << e.what() << endl;
9.          }
10.         AcceptSocket->shutdown(SD_RECEIVE);
11.         string re("");
12.         try
13.         {
14.                 AcceptSocket->recv(re, 1024);
15.         }
16.         catch (runtime_error &e)
17.         {
18.                 cout << e.what() << endl;
19.         }
20.
21.         AcceptSocket->shutdown(SD_SEND);
22.         try
23.         {
24.                 AcceptSocket->send(string(1024, 'T'));
25.         }
26.         catch (runtime_error &e)
27.         {
28.                 cout << e.what() << endl;
29.         }
30.         ConnectSocket->shutdown(SD_RECEIVE);
31.         try
32.         {
33.                 ConnectSocket->recv(re, 1024);
34.         }
35.         catch (runtime_error &e)
36.         {
37.                 cout << e.what() << endl;
38.         }
```

In this test I use the shutdown function to shut down each socket, one part at a time. The try/catch blocks are there simply to test that an exception is thrown after each call to the shutdown function.

The following results are printed into the console after these tests:

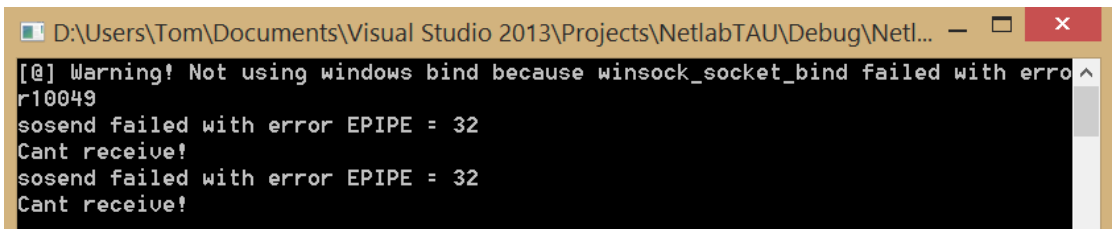


Figure 5.13 Test#6, Shutting Down a TCP Connection

First of all, the first warning does not relate to this test. When using the `bind` function on a socket, as we did on the listen socket, `L5_socket_impl` tries to use the winsock original `bind` socket, in case we initiated the OS with the default parameters, meaning that we assign an OS object our real system parameters, such IP address and MAC address. Since I used a costume address (`10.0.0.10` and `aa:aa:aa:aa:aa:aa`) the call for winsock `bind` function failed and an exception was thrown. The `L5_socket_impl` catches the exception and printout a warning massage, to remind the user that he is using a costume address.

Continuing with the examination of the printout from my test function, we can see 4 lines printed, corresponding to the 4 try\catch clocks in the test. This means that an exception is correctly thrown and that the shutdown operation was successful (at least in terms of blocking user `send/recv` requests after calling these functions).

In order to examine the actual shutdown, I explain the Wireshark traces:

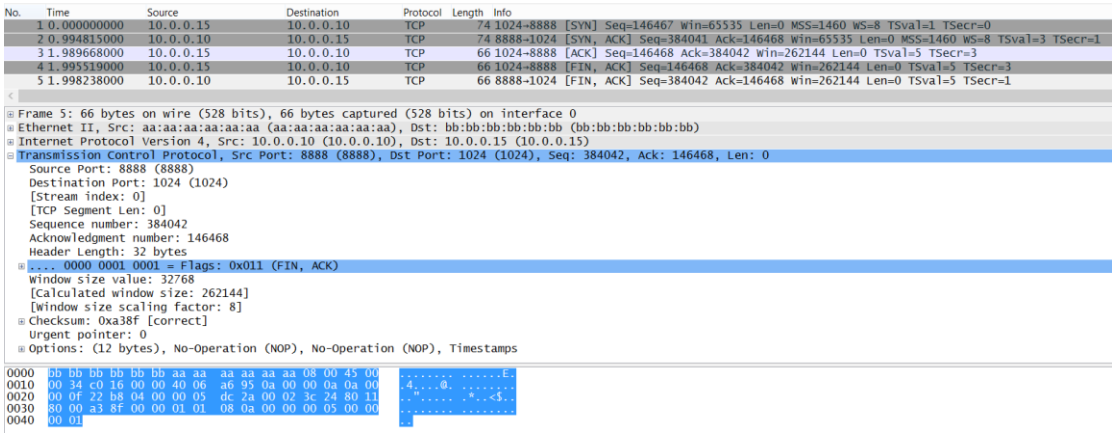


Figure 5.14: Test#6, Wireshark Screenshot

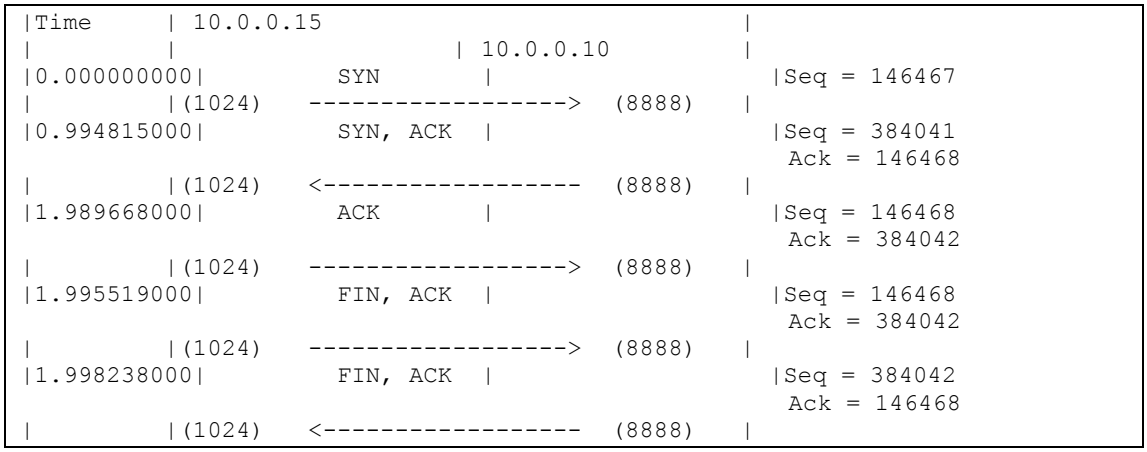


Figure 5.15: Test#6, Wireshark TCP Flow Graph

The trace proves a correct close of the connection, and we can see that with the two FIN packets at the end, similar to the closing of a connection.

Notice that the same technique [5.6] must be used, if we are to shutdown both sides for send and receive (meaning that we must stop the timers prior to the shutdown call). However, user may still receive some retransmissions as can be seen in the following figure:

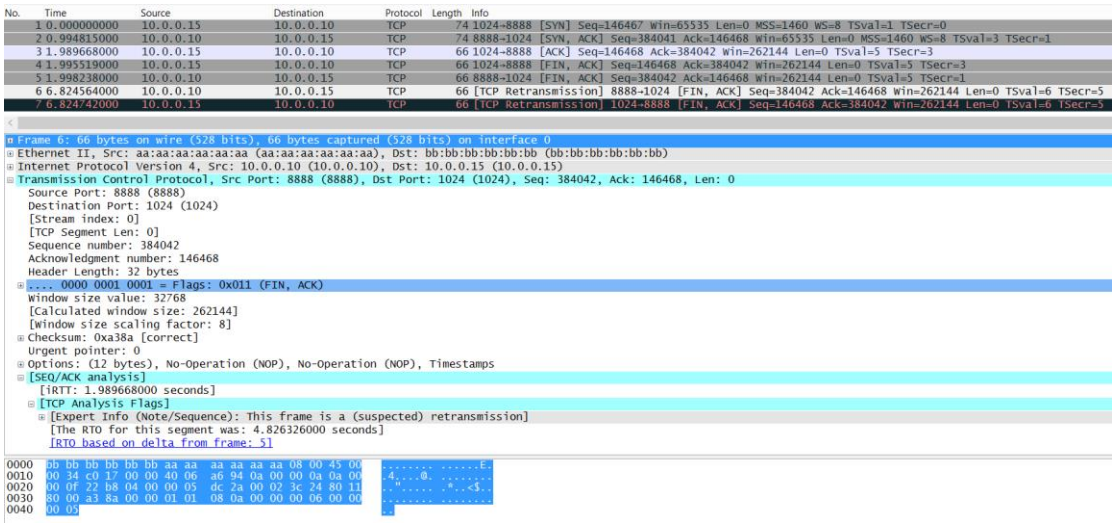


Figure 5.16: Test#6, Wrong Shutdown

The two unexpected FIN retransmissions are caused by several scenarios. Assume that we have two socket: A and B which a connection has been established between them:

1. If A was shutdown for sending, it will not be able to acknowledge a FIN packet from B.

2. If A was shutdown for receive, it will not be able to receive the FIN packet from B.

In the second scenario, if we didn't stop the fast timer thread of B, we will see the unexpected FIN retransmissions from [Figure 5.16: Test#6, Wrong Shutdown]. Unlike the close socket function, not calling the stop timer function won't lead to crash, but user may experience more retransmissions of the FIN packets that will be ignored.

5.8 Combined Test: Unreliable and Delayed Channel

In this test I used the new module added to the NIC, the `L0_buffer`. This class allows testing the TCP mechanism, and is necessary as without it is very hard to create packet drops.

The following lines were used in order to initiate the buffer, which is closed by default:

```
1.          inet_client.cable()->set_buf(
               new L0_buffer(inet_client, 0.5, L0_buffer::exponential_distribution_args(3)));
2.          inet_server.cable()->set_buf(
               new L0_buffer(inet_server, 1, L0_buffer::chi_squared_distribution_args(0.5)));
```

The meaning of this code is that the client uses an unreliable channel, with a `reliability` of 0.5, meaning that half of the packets are dropped. In addition, it uses an `exponential` distributed delay with the `lambda` parameter of 3. The server, uses a delayed channel only, since the `reliability` parameter is 1 (meaning that it is 100% reliable and will not drop packets). The delay is a `chi-squared` delay with a parameter `n` of 0.5.

Since the trace is very long, it can be found in the attached files in `test7()`. The following figure show the Wireshark screenshot:

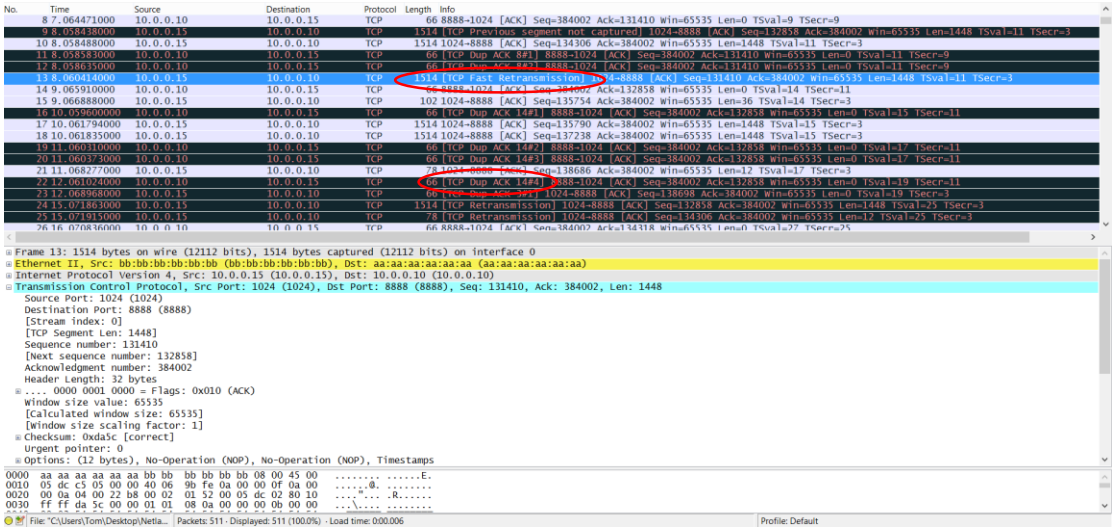


Figure 5.17: Test#7, Unreliable and Delayed Channel

The Wireshark screen shot provides us with several informative notation: First of all, we can see in the blue line (marked in a red circle) that it recognized the packet as part of the Fast Recovery state of the TCP. In addition, note that in black we can see retransmission and duplicate ACKs as well.

Note that in the other red circle, we may also see duplicate ACK#4 which is a notation that the Wireshark software inserts. Though this may seem odd at first glance, we must remember that the real duplicate ACK counter is kept within the `tcpcb` class of the connection. The reason that Wireshark sees this as a duplicate ACK#4 is since Wireshark maintain its own counter, independent of the `inet_os` counter. When the TCP reaches the 3rd duplicate ACK, it resets its `dupack` counter to 0 and starts over, unlike the Wireshark software. This means that having more than #3 duplicate ACK in Wireshark is a normal behavior of such unreliable channel.

In places that appears such #4 duplicate ACK, we expect to see the congestion window drop. Therefore, at this point the reader may wonder why the "`win`" parameter in the Wireshark trace does not drop and stay rather constant with the maximum TCP packet size. Again, we must not forget that the real `cwnd` value is kept within the `tcpcb` class of the connection. The "`win`" parameter that we see in the Wireshark trace, is only the advertised window size of the available space in the receive buffer. Since the drops are caused due to an outside cause (unreliable channel), the socket layer of the receiver manage to keep up with the sending pace, resulting in quickly clearing the buffer in time and advertising an empty buffer, which is indeed the situation.

For this reason, `MACRO` definition for the default buffer size are included in the `L5_socket` class. This is the only parameter which is most difficult to implement in runtime, which resulted in the compromise of using a `MACRO` definition in compile time. User may choose the change the default size of the buffer by setting the macro to a new size.

Continuing with the examination of the results, we concluded that the TCP congestion window effect is difficult to notice using the regular Wireshark traces. Therefore, I will turn to the sequence number and throughput graphs to see the effect:

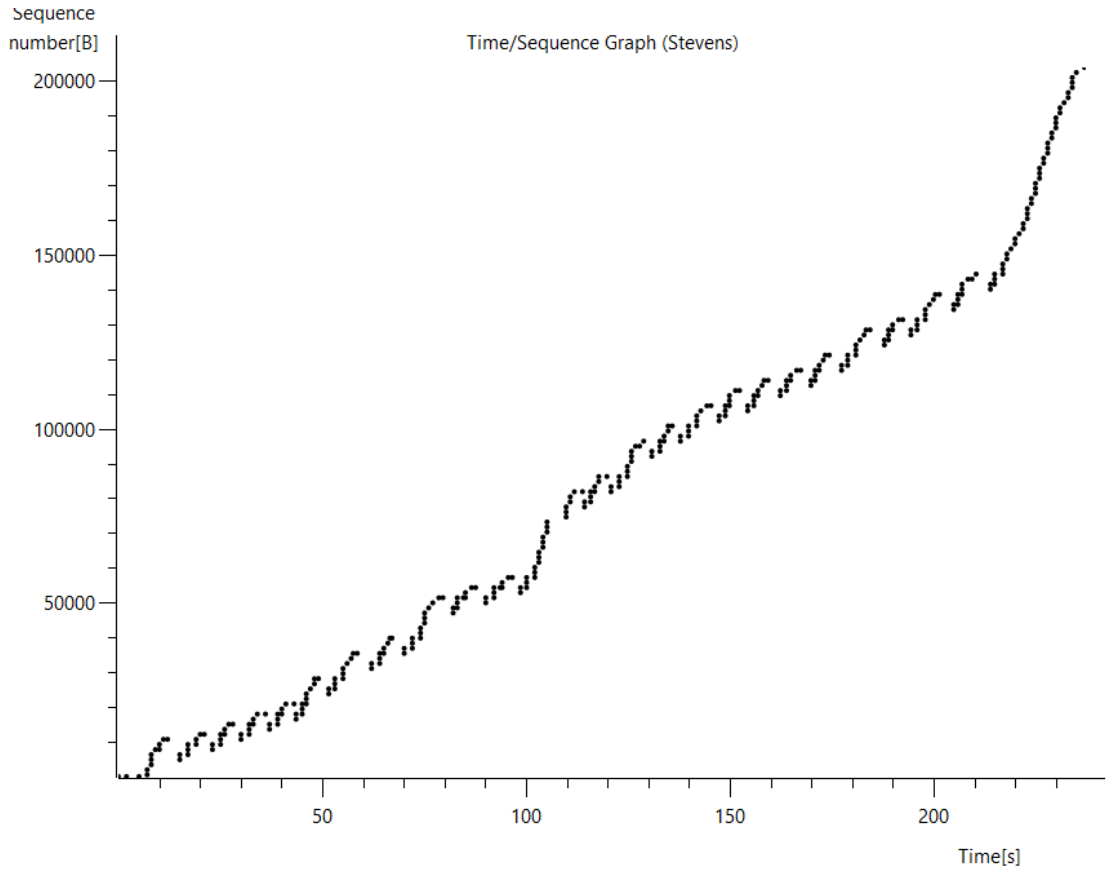


Figure 5.18: Test#7, Time/Sequence Graph

In this figure we can nicely see the template of the famous TCP's Saw Tooth pattern. By observing at the sequence numbers, we can clearly see the effect of the duplicate ACKs. In each pick of a tooth in the saw pattern, we have a sequence of duplicate ACKS. After a sequence of duplicate ACKs, we can see the retransmission effect, by the dropping of the sequence number.

In order to better understand this, I show a zoom into an interesting area and explain it in details:

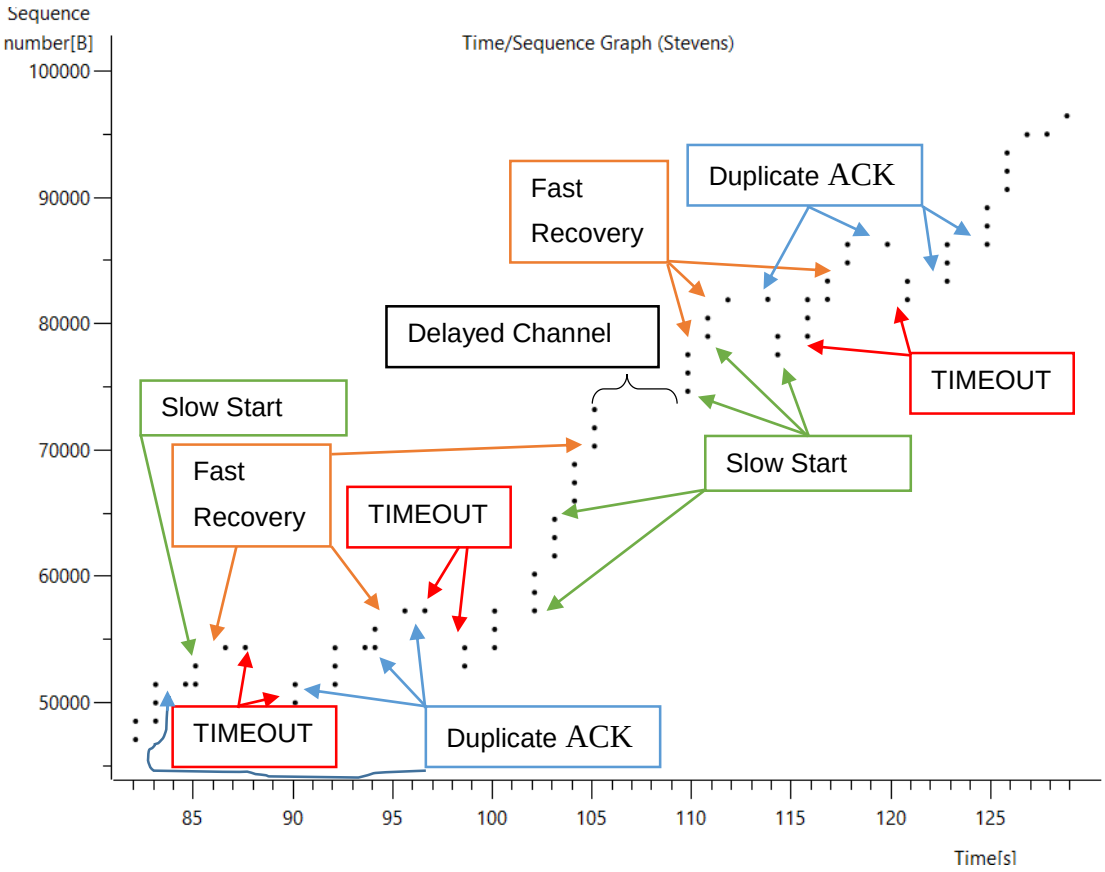


Figure 5.19: Test#7, Zoom-Into the Time/Sequence Graph

In this figure, we can see the TCP mechanism at its best: in the beginning, we can see 3 duplicate ACKs which are followed by a small Slow Start, after which comes the Fast Recovery that ends in a timeout, which drops the graph. This procedure repeats itself 4 additional times. In the middle of the graph (in black) we can see a rather long period in which nothing happens. This, as well as other such delays, is due to the delayed channel and does not relate to the TCP procedure.

Another way to observer and examine the TCP mechanism in this test is by looking at the throughput graph:

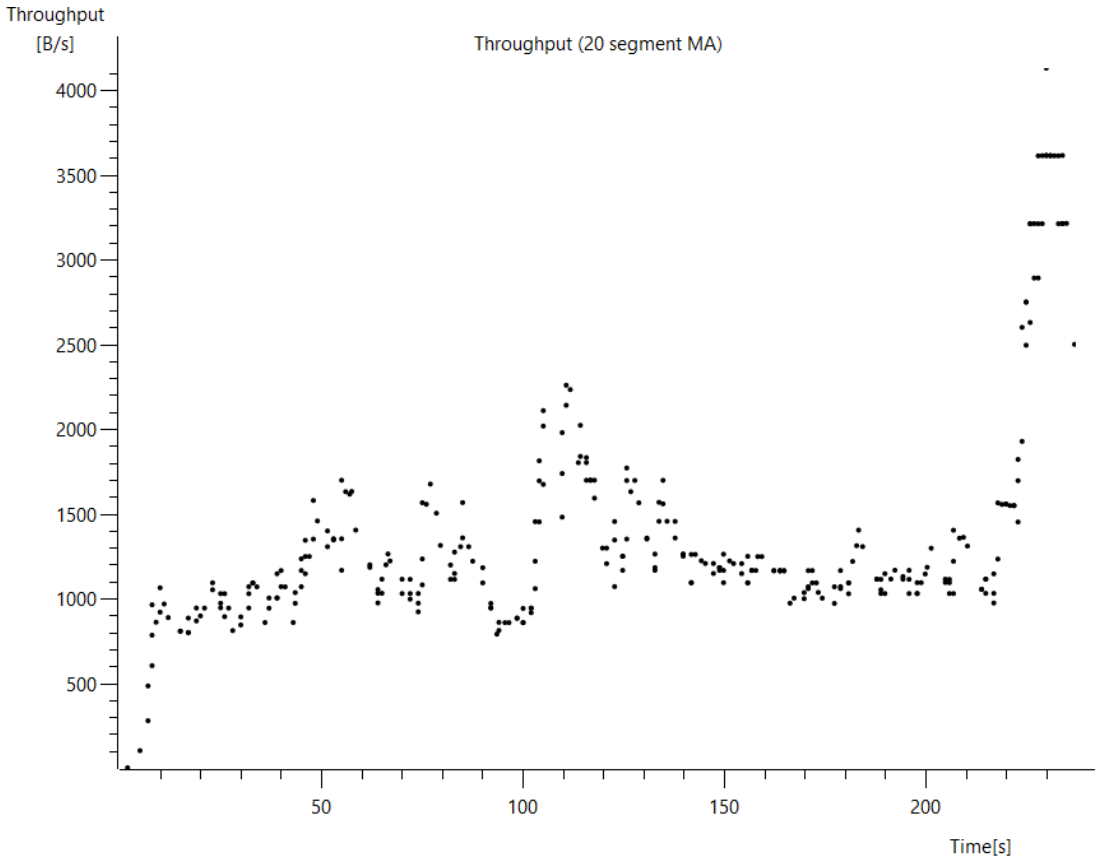


Figure 5.20: Test#7, Throughput

In this figure the famous TCP's Saw Tooth pattern is much clearer to see, as it is bigger. The throughput is the number of bytes we send, each second. Therefore, when the throughput is high, the congestion window is high. The start of the graph clearly reveals the slow start phase, which is natural. Notice that the drops followed by a linear growth belong to the duplicate ACK timeouts, and are relatively quickly (meaning that the saw teeth are thin and sharp) and the drops followed by an exponential growth belong to the regular timeouts, resulting in restarting the slow start. Note that we do not actually drop to 1 byte, but to around 1 MB each time.

As I did in the previous figure, I will now zoom into the same part and explain:

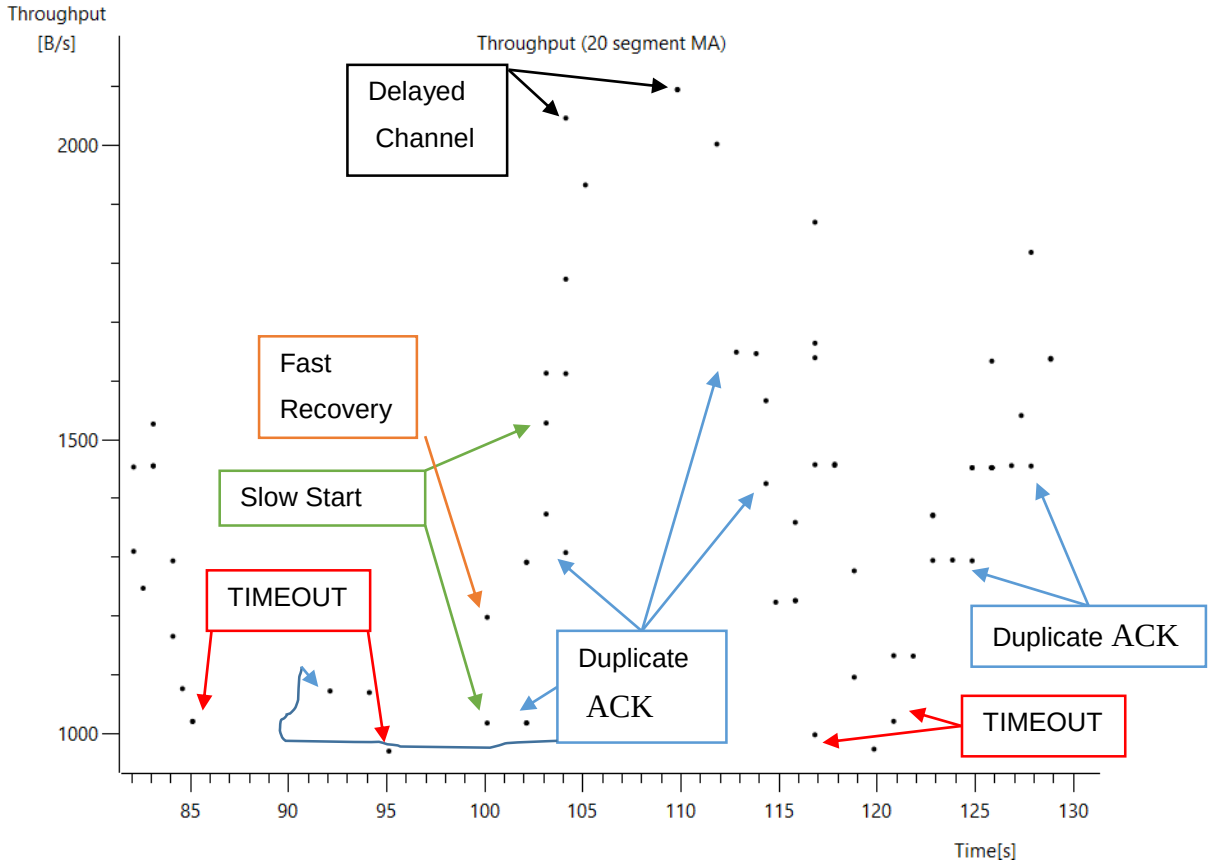


Figure 5.21: Test#7, Zoom into the Throughput

Duplicate ACKs are harder to see here, however the TCP states are clear. When the throughput grows very fast, the Slow Start step is in action. When the throughput decreases, it is usually a timeout, however, we may experience such a decrease due to a sequence of long delays in the channel, therefore this part may be more confusing to see than in the previous graph. For this reason, Fast Recovery steps are also more difficult to find. In total, this graph is better to look at, to see the general picture, however when I zoom in, other effects take place, such as the random delay of the buffer, that may influence the results.

5.9 Application Use Case

Implementing a simulation of a simple chat server, processing 3 on-line clients, transferring messages between them. In this simulation I will also resolve the MAC address using ARP, and hold onto the packet as the ARP module does.

Note that since I did not implement the select function, the server must be aware of whose turn it is to speak, and with whom does the client wish to communicate with. This may be implemented by adding some header to each message for example.

The test code for this example is long, therefore it can be found in the test file, in the `test8()` function.

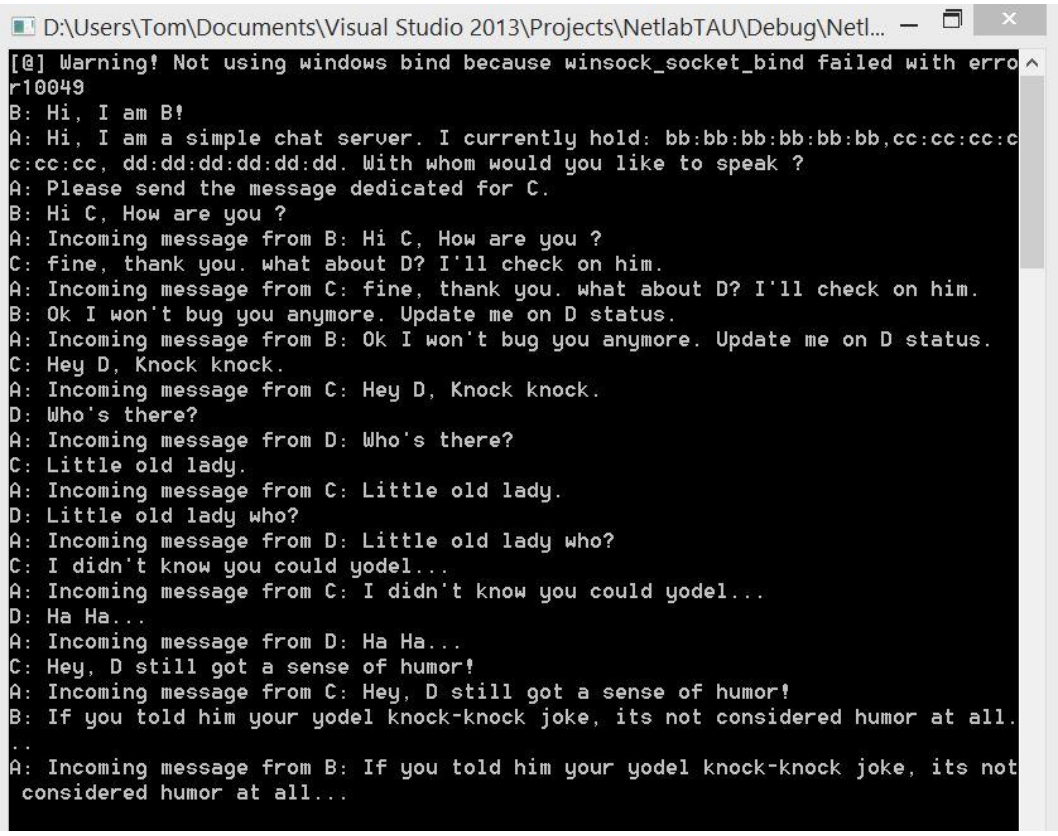


Figure 5.22: Test#8 Chat Log

In this figure, we can see 4 different OS systems: A single server (A) and 3 clients (B, C and D). All clients are sending messages to the server, which forward the message to the right client.

I now analyze this test in details, beginning with showing the ARP results and explaining what we see. Then, I explain the actual conversation and finally I show the closing of the connections and deleting the objects.

5.9.1 Part 1: ARP

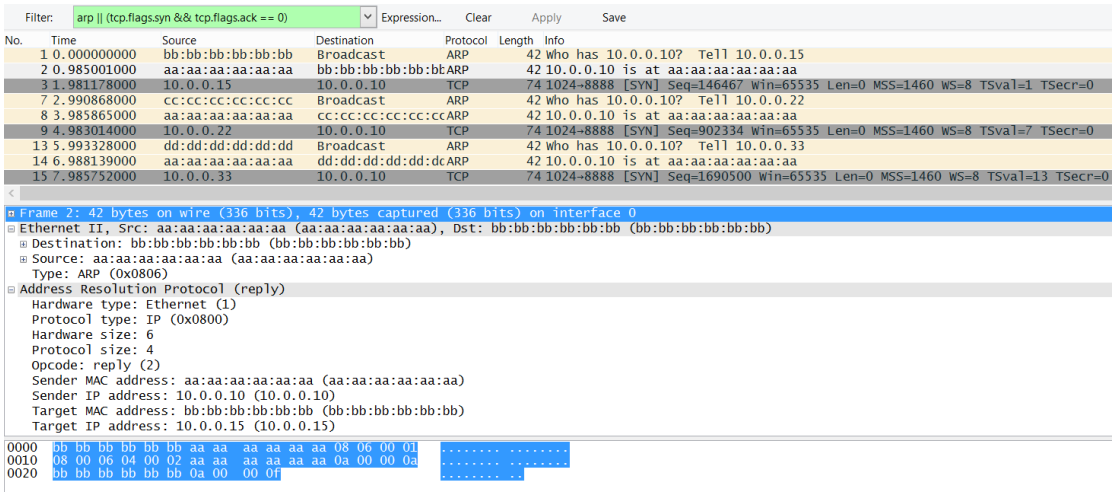


Figure 5.23: Test#8 ARP packets and SYN packets

Notice the filter that I used: I am showing all ARP messages, and TCP messages, which are pure SYN packets, in order to show the relevant part only. In total, 3 ARP packets were sent, on the connection establishment part of each connection, when a SYN packet is sent to an unresolved destination. Since all clients communicate through the server, they only need to know it MAC address.

We can see the avoidance of ARP flooding as well, from the 1 second difference between the ARP request and the ARP reply. The value is not exactly one as the function that I take for a timestamp returns a 64 bit value, and I save it to a 32 bit value, using casting.

Since the server received an ARP request, he also adds the MAC address of the requester to its ARP table, thus saving another ARP request (this is based on the assumption that if an ARP request arrived, someone wants to send us data and we are most likely are going to acknowledge it in some way, so we can save 1 second for waiting for another ARP reply).

5.9.2 Part 2: Connecting to the Server

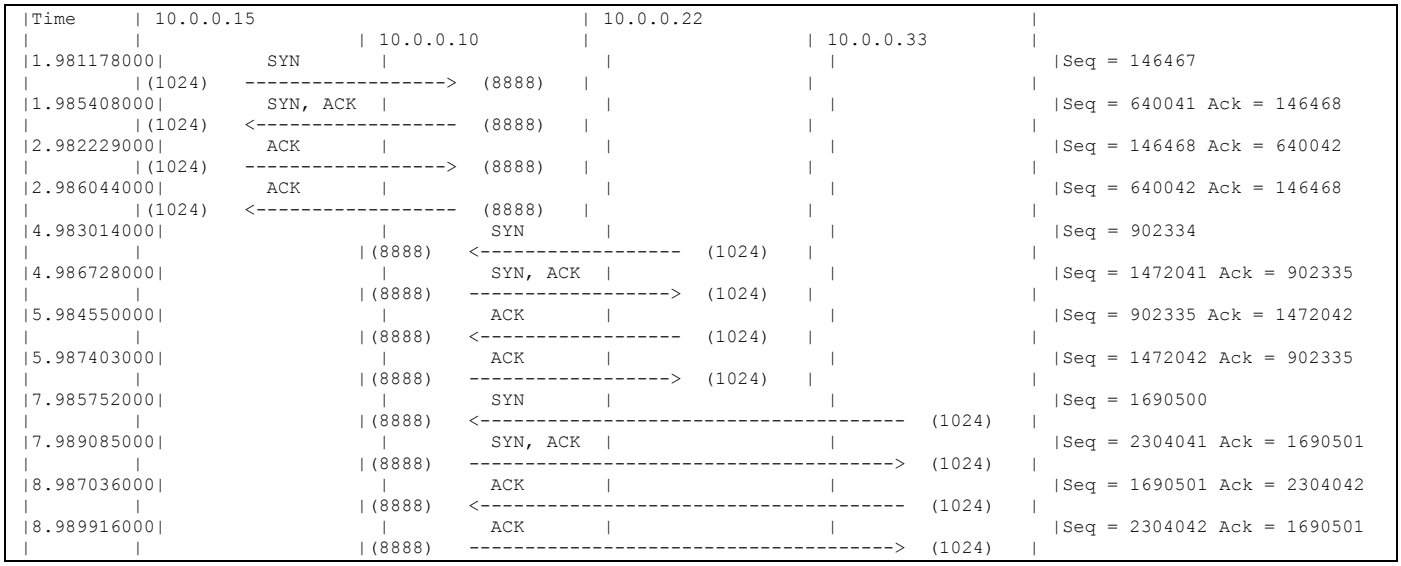


Figure 5.24: Test#8, Wireshark TCP Flow Graph for 3-way Handshake

In this figure we can see that all clients are connected to the server, which hold the IP address: `10.0.0.10` (the lowest one). Notice that we can see a duplicated ACK, which is not really a duplicated ACK but just a TCP window update messages:

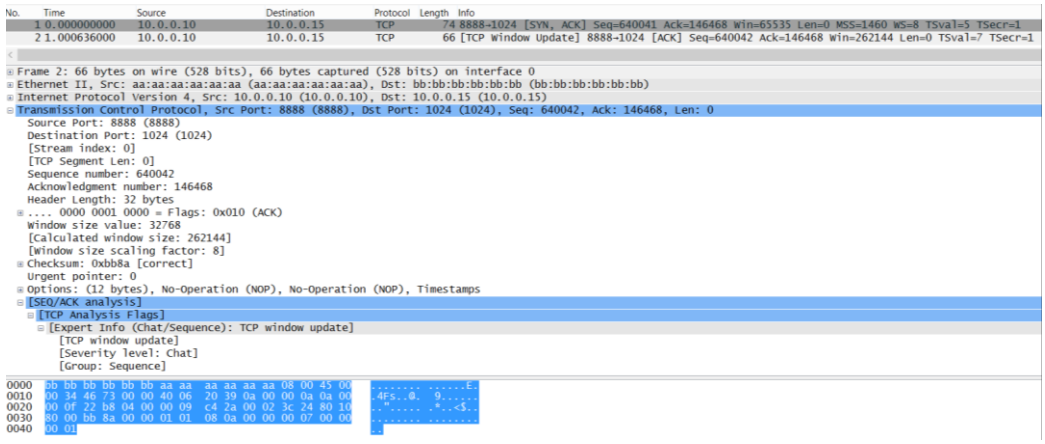


Figure 5.25: Test#8, TCP Window Update

A packet marked "TCP Window Update" simply indicates that the sender's TCP receive buffer space has increased. Observing at the window size value of the TCP header of the previous packet from the sender, and the window size value of the "TCP Window Update" packet, we can indeed see that the value has changed.

The trigger for such "TCP Window Update" packets is when an application picks up data from the receive buffer, there is now more receive buffer space available. Wireshark sees the Window Size field value has increased and marks it to let us know the Window Size field has increased, since we are going to send some data. This is normal network behavior [37].

5.9.3 Part 2: Chatting through the Server

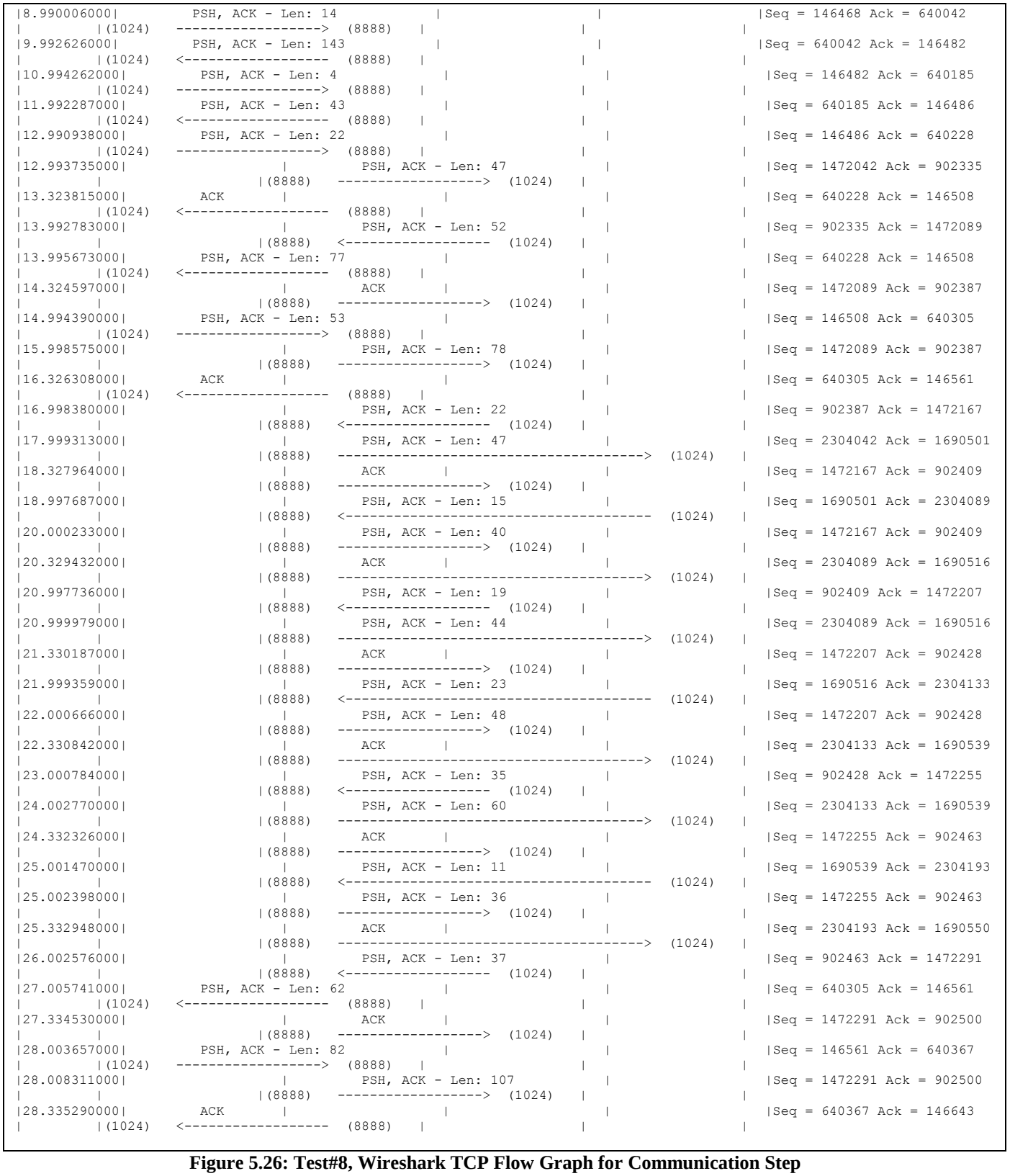


Figure 5.26: Test#8, Wireshark TCP Flow Graph for Communication Step

In this figure [Figure 5.26] we can nicely see how the server is able to send packets through each of his AcceptSockets (see the source code) that were made using the Listen socket that it opened.

In order to provide a summary, I used Wireshark statistics:

Traffic	Captured
Avg. bytes/sec	189.348
Avg. MBit/sec	0.002
Avg. packet size	99 bytes
Avg. packets/sec	1.913
Between first and last packet	19.345 sec
Bytes	3663
Packets	37

Figure 5.27: Test#8, Wireshark Statistics for the Chat

The statistics show that even when running 4 OSs together, we manage to maintain the average of 1 packet per second. This is not the performance I would have received using Microsoft TCP/IP stack or other major OS company, however surely good enough for educational uses, as this OS was intended for.

Notice that each client received the same port number, 1024. This is since I did not bind any connect socket to a specific port, resulting in the OS automatically assigning the socket a free port. The assigned port is 1024 as this test was run under administrator privileges. Under user privileges, the port number would have started at 5000 instead.

5.9.3.1 Automatic Port Assigning Under User Privileges

Using the same test, and only changing the OS owning each connect socket, as follows:

```
1.  netlab::L5_socket_impl *ConnectSocket_2(new netlab::L5_socket_impl(AF_INET, SOCK_STREAM, IPPROTO_TCP,
    inet_client)); // inet_client argument indicates the owning OS. In the original test, this was inet_client_2.
2.
3.  netlab::L5_socket_impl *ConnectSocket_3(new netlab::L5_socket_impl(AF_INET, SOCK_STREAM, IPPROTO_TCP,
    inet_client)); // And this was inet_client_3.
```

The result are a scenario in which 2 OSs (a server and a client), in which the client holds 3 different, independent sockets, and is chatting through the server using them.

To show an example of this, and the port assigning of a non-privilege user I run the test again and show the results:

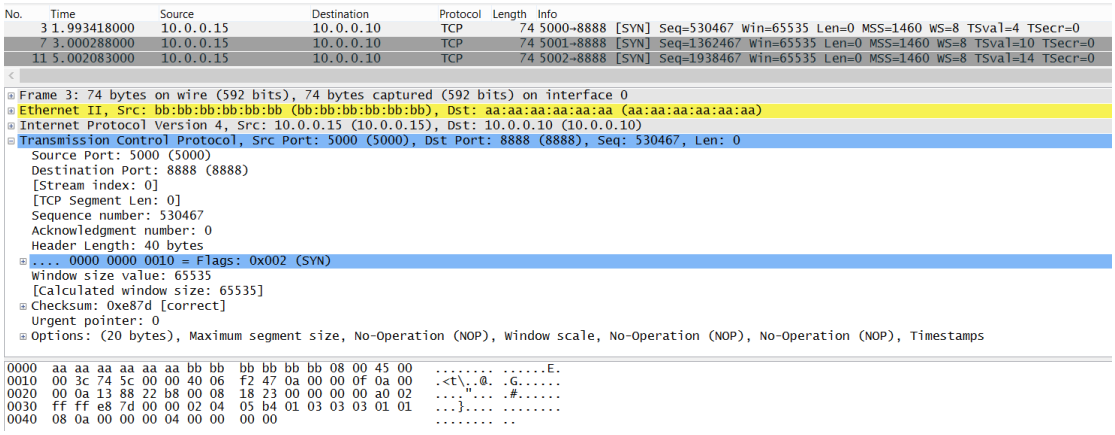


Figure 5.28: Test#8, TCP Port Assigning Under User Priveleges

In this figure, we can see that now, using a single client under user privileges, the OS automatically assigned ports starting at 5000, searching each time for the next available port. In addition, if a user under user privileges would have tried to assign a socket a port which number is less than 5000 (the reserved port number) an exception would have been thrown.

5.9.4 Part 3: Closing the connection

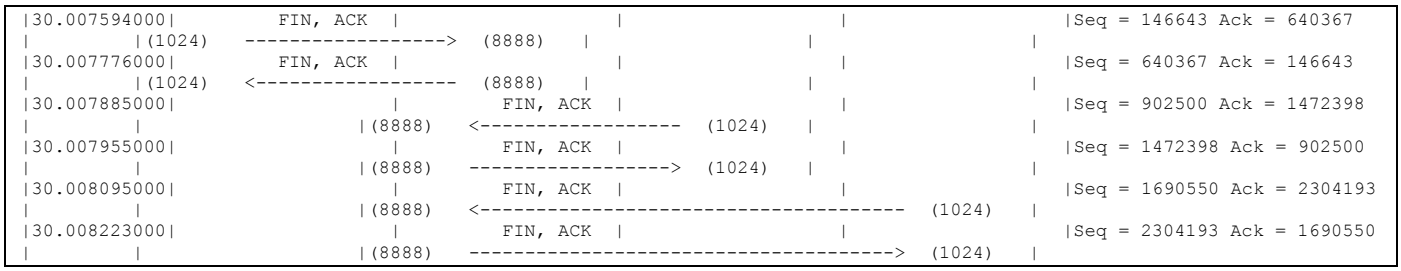


Figure 5.29: Test#8, Wireshark TCP Flow Graph for Closing the Conncetion

As we've previously seen, closing the connection is successful even when multiple clients are involved.

5.10 Cwnd Fall Test

In this test, a log module was added to the TCP layer in order to catch the exact value of the congestion window. I compare the Wireshark printout, showing the duplicate ACKs along the log file:

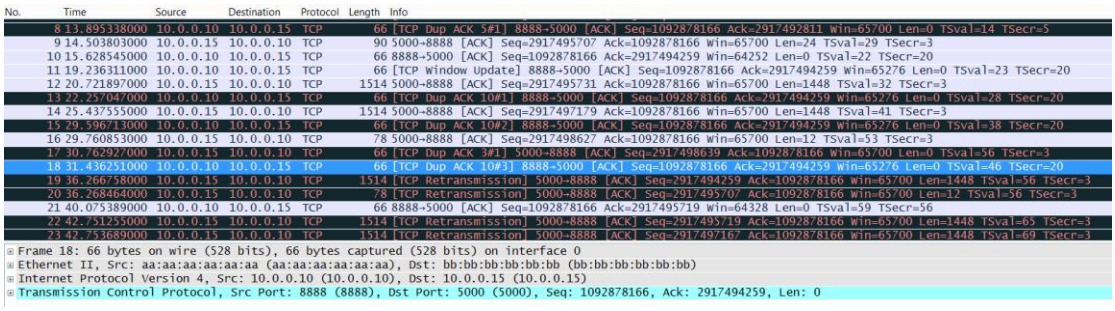


Figure 5.30: Test#9, Wireshark Trace

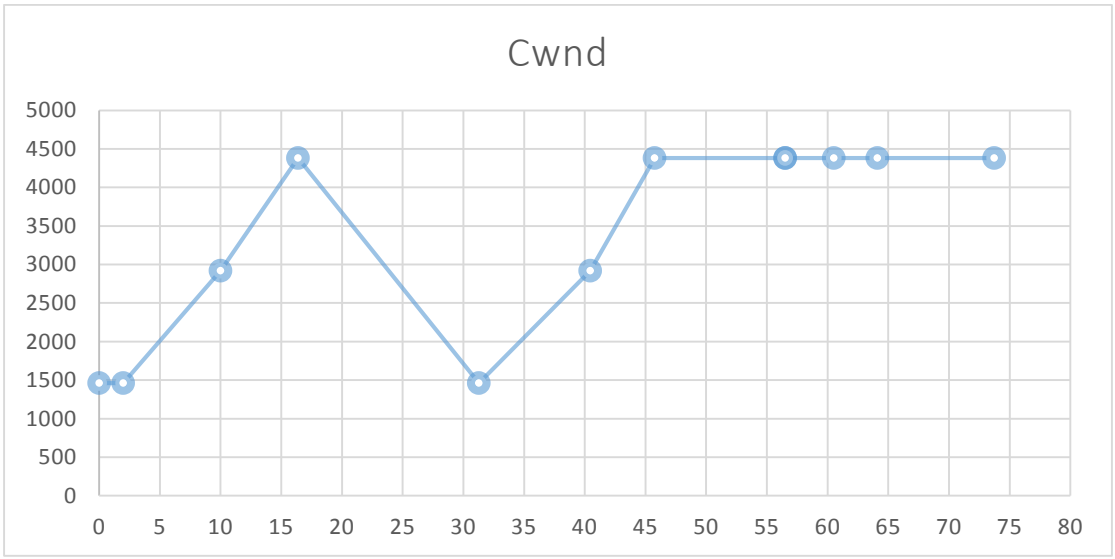
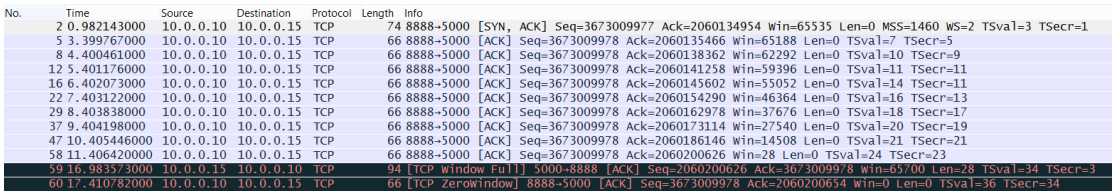


Figure 5.31: Test#9 cwnd Graph

From the printout we can indeed see that at time 31 the cwnd value dropped. The small difference in the time is due to the time it takes to click on the run button in VS and in Wireshark.

5.11 Advertise Window Gets Full

Since usually, the receiver manage to keep up with the sender pace, we did not see that the advertise window actually change. I will demonstrate that by sending many packets to the receiver, and delaying the call for recv() function.



No.	Time	Source	Destination	Protocol	Length	Info
2	0.982143000	10.0.0.10	10.0.0.15	TCP	74	8888→5000 [SYN, ACK] Seq=3673009977 Ack=2060134954 Win=65535 Len=0 MSS=1460 WS=2 TSval=3 TSecr=1
5	3.399767000	10.0.0.10	10.0.0.15	TCP	66	8888→5000 [ACK] Seq=3673009978 Ack=2060135466 Win=65188 Len=0 TSval=7 TSecr=5
8	4.400461000	10.0.0.10	10.0.0.15	TCP	66	8888→5000 [ACK] Seq=3673009978 Ack=2060138362 Win=62292 Len=0 TSval=10 TSecr=9
12	5.401176000	10.0.0.10	10.0.0.15	TCP	66	8888→5000 [ACK] Seq=3673009978 Ack=2060141258 Win=59396 Len=0 TSval=11 TSecr=11
16	6.402073000	10.0.0.10	10.0.0.15	TCP	66	8888→5000 [ACK] Seq=3673009978 Ack=2060145602 Win=55052 Len=0 TSval=14 TSecr=11
22	7.403122000	10.0.0.10	10.0.0.15	TCP	66	8888→5000 [ACK] Seq=3673009978 Ack=2060154290 Win=46364 Len=0 TSval=16 TSecr=13
29	8.403838000	10.0.0.10	10.0.0.15	TCP	66	8888→5000 [ACK] Seq=3673009978 Ack=2060162978 Win=37676 Len=0 TSval=18 TSecr=17
37	9.404198000	10.0.0.10	10.0.0.15	TCP	66	8888→5000 [ACK] Seq=3673009978 Ack=2060173114 Win=27540 Len=0 TSval=20 TSecr=19
47	10.405446000	10.0.0.10	10.0.0.15	TCP	66	8888→5000 [ACK] Seq=3673009978 Ack=2060186146 Win=14508 Len=0 TSval=21 TSecr=21
58	11.406420000	10.0.0.10	10.0.0.15	TCP	66	8888→5000 [ACK] Seq=3673009978 Ack=2060200626 Win=28 Len=0 TSval=24 TSecr=23
59	16.983573000	10.0.0.15	10.0.0.10	TCP	94	[TCP Window Full] 5000→8888 [ACK] Seq=2060200626 Ack=3673009978 Win=65700 Len=28 TSval=34 TSecr=3
60	17.410782000	10.0.0.10	10.0.0.15	TCP	66	[TCP ZeroWindow] 8888→5000 [ACK] Seq=3673009978 Ack=2060200654 Win=0 Len=0 TSval=36 TSecr=34

Figure 5.32: Test#10 Wireshark

In this figure I filtered the sender (10.0.0.15) in order to make it more clear that the advertise window drop (the sender's advertise window stays high therefore not interesting). The TCP window Full warning was left to demonstrate that when the buffer gets full, the TCP does indeed make sure to notify the other side, which replies with a TCP ZeroWindow, as expected.

6 Summary, Conclusions and Suggestions for Future Work

In this project I implemented the key features of a real networking operating system. This implementation introduces a new approach to the traditional implementation of an OS and the TCP/IP stack in particular, by using an abstract, encapsulated and modular approach.

This project is aimed for educational purposes. The modulation should help both sides of a university-level course participants: on the one hand, the abstraction provide the students with the key features that must be implemented in each part, while not exposing the source implementation. This is important not only for honor reasons, but for allowing the students to think of their own way of implementation. By this, I hope to see creative and new implementations, better implementations, and new features that will improve the current `inet_os` in general and the layers in particular.

On the other hand, the abstraction and modulation provides the academic staff the ability to explain about the most important aspects each part must have. In addition, the educator is able to provide the students with black-boxes that can easily replace a missing part. For example, in order to focus on the Ethernet layer, the educator can supply the binaries of all other layers, and let the students implement only the one Ethernet layer. Another example is that the educator can supply a test black-box which can be run independently, or even "plugged in", for an `inet_os` object, which can be used by the students on demand and enable an anchor of a working piece. Note that lab manuals should be added, though, to organize the exercises that the students must do.

The implementation is not perfect. There is much more that can be made more efficient, possible bugs that may need fixing, and most importantly – much more to add. The future of `inet_os` is a mystery. While it will most surely be deployed as an educational system in the "Advanced Communication Network Lab", Tel Aviv University, using the current implementation, there is still more to evolve.

The `inet_os` can certainly improve. First, there are still more important features that can be added, such as IP fragmentation and reassemble, TCP reassemble, multicasting, forwarding and router capabilities. But most of all, the `inet_os` provides the ability to teach the foundations of a networking OS, for educational purposes. In general, such OS may supply better performance to a network device, which does not need all the features that provides a full OS, such as FreeBSD or Windows. In addition, since the entire OS is based on abstraction and portability, this kind of OS can be enhanced to support an Abstract OS with default implemented parts, which can easily be changed in run time, based on the user's demands.

7 Bibliography

[1] University of California, Berkeley; Computer Systems Research Group (CSRG); , "4.4BSD-Lite2," Berkeley Software Distribution, 1995. [Online]. Available: https://github.com/sergev/4.4BSD-Lite2.

[2] "Advanced Laboratory in Computer Communications," Tel Aviv University, 2014. [Online]. Available: http://www.eng.tau.ac.il/~netlab/.

[3] W. R. Stevens and G. R. Wright, "Preface," in *TCP/IP Illustrated, The Implementation*, vol. 2, Addison-Wesley Professional Computing Series, 1995, pp. xix-xxi.

[4] T. F. Project, "FreeBSD 10.2-RELEASE," The FreeBSD Project, 13 August 2015. [Online]. Available: https://www.freebsd.org/releases/10.2R/announce.html.

[5] BSD, "The FreeBSD Project," The FreeBSD Project, 2015. [Online]. Available: https://github.com/freebsd/freebsd.

[6] cprogramming, "C++0x: The future of C++," cprogramming, [Online]. Available: http://www.cprogramming.com/c++11/what-is-c++0x.html.

[7] Y. Perry, "Data Link layer," *Introduction to Computer Communication*, p. 28, March 2015.

[8] D. C. Plummer, "RFC826: An Ethernet Address Resolution Protocol," November 1982. [Online]. Available: https://tools.ietf.org/html/rfc826.

[9] J. Garrett, J. Hagan and J. Wong, "RFC1433: Directed ARP," March 1993. [Online]. Available: https://tools.ietf.org/html/rfc1433.

[10] cppreference.com, "std::map," cppreference.com, 16 August 2015. [Online]. Available: http://en.cppreference.com/w/cpp/container/map.

[11] MSDN, "TCP/IP Raw Sockets," Microsoft, 2015. [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms740548(v=vs.85).aspx.

[12] cppreference.com, "nullptr, the pointer literal," cppreference.com, 20 May 2015. [Online]. Available: http://en.cppreference.com/w/cpp/language/nullptr.

[13] cppreference.com, "std::shared_ptr," cppreference.com, 25 July 2015. [Online]. Available: http://en.cppreference.com/w/cpp/memory/shared_ptr.

[14] B. Prof. Patt-Shamir, "TCP SNs and ACKs," *Introduction to Computer Communication,* p. 15, 2015.

[15] T. Mahler, M. Baruch and N. Meron, "Summary: TCP Congestion Control," *Introduction to Computer Networks,* p. 10, 2014.

[16] W. R. Stevens and G. R. Wright, "TCP Timers," in *TCP/IP Illustrated, The Implementation*, vol. 2, Addison-Wesley Professional Computing Series, 1995.

[17] W. R. Stevens and G. R. Wright, "TCP Options," in *TCP/IP Illustrated, Volume 2 The Implementation*, vol. 2, Addison-Wesley Professional Computing Series, 1995.

[18] V. Jacobson, R. Braden and D. Borman, "RFC1323: TCP Extensions for High Performance," May 1992. [Online]. Available: https://www.ietf.org/rfc/rfc1323.txt.

[19] W. R. Stevens and G. . R. Wright, "Introduction," in *TCP/IP Illustrated, The Implementation*, vol. 2, Addison-Wesley Professional Computing Series, 1995, pp. 1-31.

[20] M. Kerrisk, " raw - Linux IPv4 raw sockets," man7.org, 08 August 2015. [Online]. Available: http://man7.org/linux/man-pages/man7/raw.7.html.

[21] Riverbed Technology, "WinPcap the industry-standard windows packet capture library," Riverbed Technology, 3 March 2013. [Online]. Available: http://www.winpcap.org/.

[22] W. R. Stevens and G. R. Wright, "BPF: BSD Packet Filter," in *TCP/IP Illustrated, The Implementation*, vol. 2, Addison-Wesley Professional Computing Series, 1995.

[23] WinPcap, "Filtering Expression Syntax," WinPcap, 2007. [Online]. Available: https://www.winpcap.org/docs/docs_40_2/html/group__language.html.

[24] M. Fontanini, "Tins::Sniffer Class Reference," libtins, 25 August 2015. [Online]. Available: https://libtins.github.io/docs/latest/de/dbb/classTins_1_1Sniffer.html.

[25] R. Johnson, "POSIX Threads for Win32," sourceware, 27 May 2012. [Online]. Available: https://www.sourceware.org/pthreads-win32/.

[26] cppreference.com, "std::mutex," cppreference.com, 5 January 2015. [Online]. Available: http://en.cppreference.com/w/cpp/thread/mutex.

[27] cppreference.com, "std::lock," cppreference.com, 12 March 2015. [Online]. Available: http://en.cppreference.com/w/cpp/thread/lock.

[28] cppreference.com, "std::lock_guard," cppreference.com, 17 January 2014. [Online]. Available: http://en.cppreference.com/w/cpp/thread/lock_guard.

[29] B. Dawes, D. Abrahams and R. Rivera, "Boost C++ Libraries," Boost.org, 13 August 2015. [Online]. Available: http://www.boost.org/.

[30] J. Gaspar, "Boost.Circular Buffer," Boost, 04 August 2015. [Online]. Available: http://www.boost.org/doc/libs/1_59_0/doc/html/circular_buffer.html.

[31] V. S. 2015, "STL Containers," Microsoft, 2015. [Online]. Available: https://msdn.microsoft.com/en-us/library/1fe2x6kt.aspx.

[32] cppreference.com, "std::condition_variable," cppreference.com, 12 August 2015. [Online]. Available: http://en.cppreference.com/w/cpp/thread/condition_variable.

[33] W. R. Stevens and G. R. Wright, "Mbufs: Memory Buffers," in *TCP/IP Illustrated, The Implementation*, vol. 2, Addison-Wesley Professional Computing Series, 1995.

[34] www.cplusplus.com, "std::vector," www.cplusplus.com, 2015. [Online]. Available: http://www.cplusplus.com/reference/vector/vector/.

[35] G. Combs, "Wireshark," Wireshark Foundation, 21 August 2014. [Online]. Available: https://www.wireshark.org/about.html.

[36] cppreference.com, "std::this_thread::sleep_for," cppreference.com, 19 February 2015. [Online]. Available: http://en.cppreference.com/w/cpp/thread/sleep_for.

[37] lchappell, "expert.message == "Window update"," Wireshark, 10 November 2010. [Online]. Available: https://ask.wireshark.org/questions/901/expertmessage-window-update. [Accessed 2015].

[38] Dimitri, "Doxygen," Doxygen, 8 August 2015. [Online]. Available: http://www.stack.nl/~dimitri/doxygen/index.html.

[39] W. R. Stevens and G. R. Wright, TCP/IP Illustrated, The Implementation, vol. 2, Addison-Wesley Professional Computing Series, 1995.

8 Appendix

Following is a detailed documentation of the entire code. The document was written using Doxygen [38]. This documentation cover only the header files and is meant to provide a more low-level understanding of the code structure and how to use it.

8.1 The Source Code

In order to understand the actual implementation, detailed comments were written in the source files. Following, is an example of how to approach the source code. I will first present the entire function, and then how to read each part. We examine the `pr_slowtimo` function, written as part of the `L4_TCP_impl` class:

```
1.  void L4_TCP_impl::pr_slowtimo()
2.  {
3.          std::lock_guard<std::mutex> lock(inet._splnet);
4.
5.          /*
6.          *       tcp_rnaxidle is initialized to 10 minutes. This is the maximum amount of time
7.          *       TCP will send keepalive probes to another host, waiting for a response from that host.
8.          *       This variable is also used with the FIN_WAIT_2 timer, as we describe in Section 25.6.
9.          *       This initialization statement could be moved to tcp_ini t, since it only needs to be
10.         *       evaluated when the system is initialized (see Exercise 25.2).
11.         */
12.         tcp_maxidle = tcp_keepcnt * tcp_keepintvl;
13.
14.         /*
15.         *       Check each timer counter In all TCP control blocks:
16.         *               Each Internet PCB on the TCP list that has a corresponding TCP control block is
17.         *       checked. Each of the four timer counters for each connection is tested, and if nonzero,
18.         *       the counter is decremented. When the timer reaches 0, a PRU_SLOWTIMO request is
19.         *       issued. We'll see that this request calls the function tcp_tirners, which we describe
20.         *       later in this chapter.
21.         *               The fourth argument to tcp_usrreq is a pointer to an mbuf. But this argument is
22.         *       actually used for different purposes when the mbuf pointer is not required. Here we
23.         *       see the index i is passed, telling the request which timer has expired. The funnylooking
24.         *       cast of i to an mbuf pointer is to avoid a compile-time error.
25.         *               Notice that if there are no TCP connections active on the host (tcb. inp_next is
26.         *       null), neither tcp_iss nor tcp_now is incremented. This would occur only when the
27.         *       system is being initialized, since it would be rare to find a Unix system attached to a
28.         *       network without a few TCP servers active.
29.         *
30.         * Search through tcb's and update active timers.
31.         */
32.         class inpcb_impl *ip(dynamic_cast<inpcb_impl*>(tcb.inp_next));
33.         if (ip == nullptr)
34.                 return;
35.         class inpcb_impl *ipnxt;
36.         for (; ip != &tcb; ip = ipnxt) {
37.                 ipnxt = dynamic_cast<inpcb_impl*>(ip->inp_next);
38.                 class L4_TCP::tcpcb *tp(L4_TCP::tcpcb::intotcpcb(ip));
39.                 if (tp == nullptr || tp->t_state == L4_TCP::tcpcb::TCPS_LISTEN)
40.                         continue;
41.                 for (size_t i = 0; i < TCPT_NTIMERS; i++)
42.                         if (tp->t_timer[i] && --tp->t_timer[i] == 0) {
43.                                 (void)pr_usrreq(
44.                                   dynamic_cast<socket*>(tp->t_inpcb->inp_socket),
45.                                   PRU_SLOWTIMO,
46.                                   std::shared_ptr<std::vector<byte>>(nullptr),
47.                                   reinterpret_cast<struct sockaddr *>(i),
48.                                   sizeof(i),
49.                                   std::shared_ptr<std::vector<byte>>(nullptr));
50.
51.                                 /*
52.                                 *       Check If TCP control block has been deleted:
53.                                 *       Before examining the timers for a control block, a pointer to the next Internet PCB is
54.                                 *       saved in ipnxt. Each time the PRU_SLOWTIMO request returns, tcp_slowtirno checks
55.                                 *       whether the next PCB in the TCP list still points to the PCB that's being processed. If
56.                                 *       not, it means the control block has been deleted-perhaps the 2MSL timer expired or
57.                                 *       the retransmission timer expired and TCP is giving up on this connection-causing a
58.                                 *       jump to tpgone, skipping the remaining timers for this control block, and moving on to
59.                                 *       the next PCB.
60.                                 */
61.                                 if (ipnxt->inp_prev != ip)
62.                                         goto tpgone;
63.                         }
64.
```

```
65.                     /*
66.                     *       Count Idle time:
67.                     *       t_idle is incremented for the control block. This counts the number of 500-ms
68.                     *       clock ticks since the last segment was received on this connection. It is set to 0 by
69.                     *       tcp_input when a segment is received on the connection and used for three purposes:
70.                     *               (1)     by the keepalive algorithm to send a probe after the connection is idle for 2 hours,
71.                     *               (2)     to drop a connection in the FIN_WAIT_2 state that is idle for 10 minutes and 75
                                                                                                                    seconds, and
72.                     *               (3)     by tcp_output to return to the slow start algorithm after the connection has
73.                     *                       been idle for a while.
74.                     */
75.                     tp->t_idle++;
76.
77.                     /*
78.                     *       Increment RTT counter:
79.                     *       If this connection is timing an outstanding segment, t_rtt is nonzero and counts
80.                     *       the number of 500-ms clock ticks until that segment is acknowledged. It is initialized to
81.                     *       1 by tcp_output when a segment is transmitted whose KIT should be timed.
82.                     *       tcp_slowtimo increments this counter.
83.                     */
84.                     if (tp->t_rtt)
85.                             tp->t_rtt++;
86.             tpgone:
87.                     ;
88.             }
89.
90.             /*
91.             *       Increment initial send sequence number:
92.             *       tcp_iss was initialized to 1 by tcp_ini t. Every 500 ms it is incremented by
93.             *       64,000: 128,000 (TCP ISSINCR) divided by 2 (PR_SLOWHZ). This is a rate of about once
94.             *       every 8 microseconds, although tcp_iss is incremented only twice a second. We'll see
95.             *       that tcp_iss is also incremented by 64,000 each time a connection is established, either
96.             *       actively or passively.
97.             *               Remark: RFC 793 specifies that the initial sequence number should increment roughly every 4
                                                                                                            microseconds,
98.             *                               or 250,000 times a second. The Net/3 value increments at about one-half this rate.
99.             */
100.            TCP_ISSINCR(PR_SLOWHZ); /* increment iss */
101.
102.            /*
103.            *       Increment RFC 1323 timestamp value:
104.            *       tcp_now is initialized to 0 on bootstrap and incremented every 500 ms. It is used
105.            *       by the timestamp option defined in RFC 1323 [Jacobson, Braden, and Borman 1992),
106.            *       which we describe in Section 26.6.
107.            */
108.            tcp_now++;      /* for timestamps */
109.    }
```

In red appears all the comments, explaining in detail each portion of the code. Comments are based on TCP/IP Illustrated Volume 2 [39] a references to figures are matching the figures in the book. It is recommended to address the book for more examples.

8.2 The Site

Following is a print screen of the site:

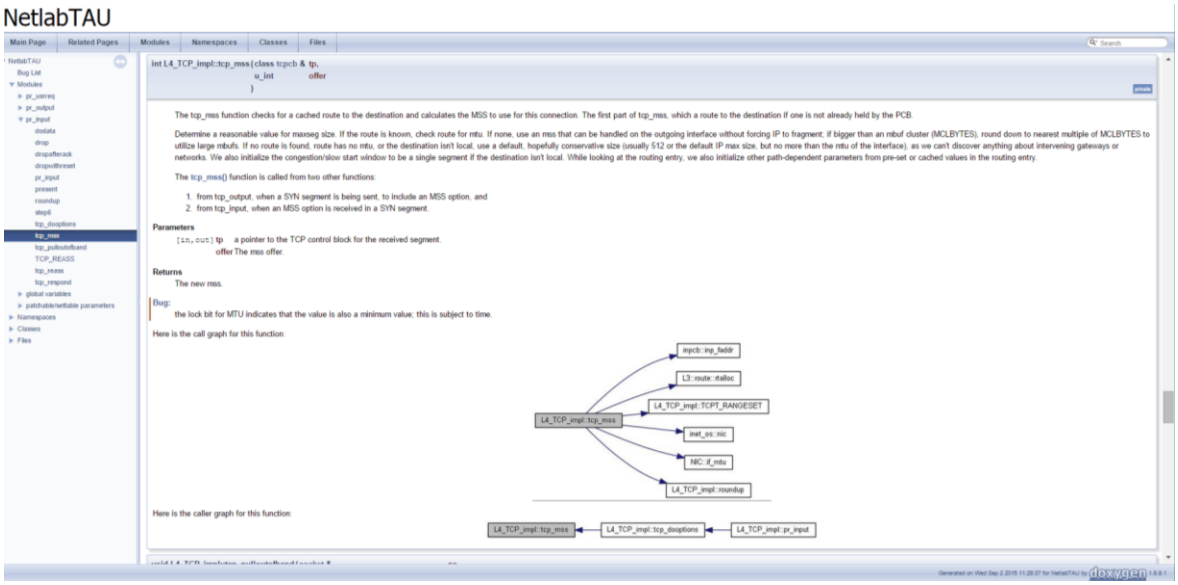


Figure 8.1: Understanding the Site

We can see that each function includes the description of what the function does, as well as the parameters that it accepts and the return value. In this particular function, we can see a "bug" section which marks a known bug that was found.

In addition, we can see the call graph of the function, as well as the caller graph. On the upper right corner there is a search function for fast navigation. On the left user can browse by the navigation panel which is sorted by classes.

8.3 The PDF Documentation

Attached is a PDF version of the same documentation. This is another alternative, which similar to the site, document the header files. The document is linked to support easy navigation.