Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Pascal Sprenger

Design and Implementation of an ECN Proxy for Performance Improvements in the Internet

Semester Thesis
October 2015 to January 2016

Tutor: Laurent Vanbever
Supervisors: Brian Trammell and Mirja Kühlewind

Abstract

Explicit Congestion Notification is a TCP/IP extension that allows congestion signaling without packet loss and therefore can greatly increase the performance on the Internet. ECN is widely implemented in end systems but it is barely used. If the host that opens the connection doesn't want to use ECN, it's not used.

The task of this semester thesis is to design, implement and test a proxy to boost the usage of ECN. If only one host wishes to use ECN, this proxy simulates for both hosts the corresponding counterpart. Installed on a home router the proxy would enable the usage of ECN with a server if the server is ECN capable but the client not. A system administrator can install such a proxy and enable ECN for every TCP connection in the internet originating from this router.

Contents

Chapter 1

Introduction

1.1 Motivation

Explicit Congestion Notification (ECN) [2] is a TCP/IP extension that allows congestion signaling without packet loss and therefore can greatly increase the performance on the Internet. Even though ECN was standardized in 2001, and it is widely implemented in end systems, it is barely used [1]. More and more webserver support ECN but often the client that opens the connection does not request ECN support. If only one endpoint doesn't want to use ECN, then it is not used.

On the other hand, we have the routers. If there isn't enough traffic that uses ECN, it's not worthwhile to change their operating systems to allow marking packets instead of dropping as it would be possible with the ECN protocol. The current state is a vicious circle. If nobody uses ECN, the routers aren't motivated to mark packets because it makes no difference and if router don't mark packets there is no advantage for the clients to enable ECN.

This semester thesis presents a possible solution to break this vicious circle. The solution is to install a proxy in the home network that negotiates instead of the clients ECN usage with the servers in the internet. Without changing anything on the machines in the internal network, be it a laptop, a smartphone or a desktop computer, the traffic with a server using ECN on the long path through the internet would be ECN enabled. Enabling ECN on the long path from the home router to the server where the congestion most likely will occur has the potential to increase the throughput and decrease the delay of the TCP connections for every end-node in this internal network. Even if no router would mark the packets instead of dropping it would at least increase the percentage of ECN enables traffic of all TCP traffic.

1.2 Task

The goal of this semester project is to implement and test an ECN proxy that could be installed in home routers to speed-up ECN deployment in case one of the connection endpoints is not ECN-capable. This leads to the following tasks:

1. Design and implementation of an ECN proxy
2. Set-up of a testbed to test the ECN proxy implementation
3. Evaluation of ECN performance

Chapter 2

Background

Following we have a short summary of TCP, ECN and Netfilter to make the reader familiar with both of the protocols fundamental to my work and an important part of the Linux kernel.

2.1 TCP - Transmission Control Protocol

TCP is a protocol for a connection oriented communication and part of the transport layer of the internet protocol suite. Unlike UDP (User Datagram Protocol) the communication with TCP focuses on being reliable and complete. The receiver acknowledges the received packets to the sender. If the sender doesn't receive an acknowledgment after a certain time he assumes the packet was lost.

As the sender and receiver are both interested in a fast transmission of the data, they try to communicate as fast as possible. They start slow and get gradually faster and faster until they reach a set limit or a router on the way between the sender and receiver can't handle all the packets sent at a time. If a router has too much packets waiting in his buffer, it has to decline new packets and has to drop them. Even if the router would observe its queue and notice that the buffer will probably overflow and therefore packets will be dropped if the packets continue to arrive at the same rate it has no possibility to communicate with the sender or receiver other than to drop a packet before the queue reaches its limit. Therefore, a packet is lost, this loss has to be detected and the packet has to be sent again. This results in a reduced throughput and an increased delay.

2.2 ECN - Explicit Congestion Notification

ECN is an addition to IP (Internet Protocol) and TCP. It tries to solve the problem of the communication between sender/receiver and router. If the router detects that its queue most likely will overflow (or just be too big for an acceptable latency) he can set a flag in the IP header and tell the receiver to notify the sender, he should send slower. The router communicates without a packet loss with the sender. There is no packet loss to be detected and therefore the goodput is increased and the overall delay is reduced.

ECN uses 2 bits in the IP-header and 2 flags in the TCP-header. Table 2.2 explains the different values of the two bits in the IP-header. The two flags in the TCP-header are the ECE (ECN Echo) and the CWR (Congestion Window Reduced). These two flags are used for the negotiation whether ECN is used for the current connection and for communicating when congestion was experienced. With the task to design a proxy that negotiates and handles an ECN connection if one part doesn't want to use it in mind, we take a closer look at how the use of ECN is negotiated and how sender, receiver and router communicate with each other.

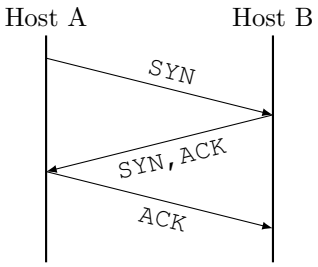


Figure 2.1: A doesn't want to use ECN

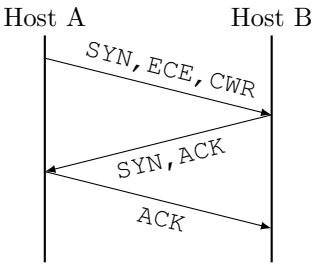


Figure 2.2: A wants to use ECN but B doesn't

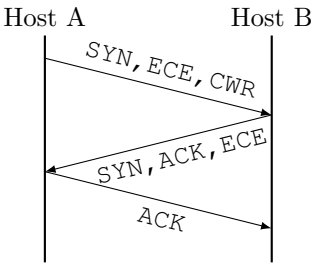


Figure 2.3: A and B want to use ECN, negotiation successful

ECN Field			
0	0	Not-ECT	ECN is not used for this packet. The router has to drop it.
0	1	ECT(1)	ECN is used for this connection and this packet can be marked with CE
1	0	ECT(0)	ECN is used for this connection and this packet can be marked with CE
1	1	CE	Congestion Experienced - This packet would have been dropped

Table 2.1: Explanation of the different ECN Field values

Negotiation whether ECN is used

ECN is an addition to IP and TCP and can't be used by default. It's not guaranteed that both communicating partners can handle the flags and react accordingly. Therefore, the use of ECN has to be negotiated. This is done simultaneously with the normal connection establishment of a TCP connection. A connection can only use ECN if both partners want to use it. This gives rise to three possibilities:

- If the initiating host doesn't want to use ECN, it won't be used for this connection. The three-way-handshake at the beginning of each TCP connection remains unchanged, pictured in figure 2.1.
- If the initiating host wants to use ECN it sends the normal initial SYN packet but with additionally setting the ECE and CWR mark. Every router part of the route and the receiving host can tell that the sender of this packet knows ECN and wants to use it. The receiver has now two choices, either use ECN or don't use it. If he doesn't want to use it, he simply answers with a normal SYN-ACK packet. This signals the initiating host that its communication partner can't or doesn't want to use ECN. The handshake in this case can be seen in figure 2.2. In this case ECN isn't used.
- Lastly if both hosts want to use ECN, the opening host sends the "ECN-setup SYN packet" (flags ACK, ECE & CWR set) and the receiver responds with the "ECN-setup SYN-ACK packet" (flags SYN, ACK & ECE set and CWR **not** set). The final ACK is the same as in the normal handshake and concludes the connection establishment and ECN negotiation. The handshake in case of a successful negotiation can be seen in figure 2.3

Communication between Sender, Receiver and Router

If successfully negotiated to use ECN, both the sender and receiver can set the ECT (ECN-capable Transport) codepoint in the IP-header. This codepoint signals to every router on the path that this connection uses ECN and that the router can instead of dropping the packet just mark it with the CE code point. A router doesn't have to mark a packet instead of dropping it, but it has the possibility to do so.
Using ECN makes the hosts responsible to signal a received CE to the sender. This is done with the flags ECE & CWR which are part of the TCP-header and previously used to negotiate ECN. If a host receives a CE he has to set for every packet the ECE flag until he gets a response with the CWR flag set. This ensures that the sender is notified about the experienced congestion. In figure 2.4 such a chain is depicted. If a host receives a TCP-packet with the ECE flag set, he

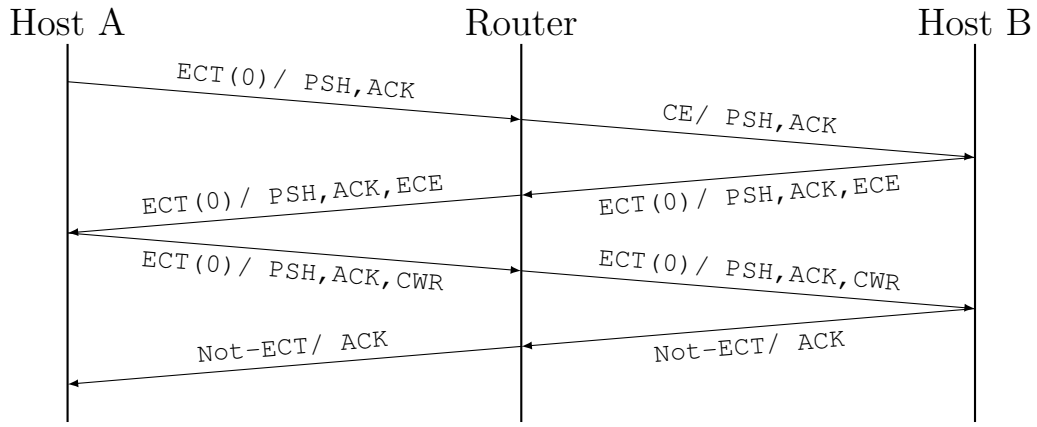


Figure 2.4: Router marks Packet from Host A due to Congestion. First the router receives a packet and decides to drop it, but after checking for the CET mark he changes the code point to CE and forwards the packet to the destination. The receiver inspects the packet and notices the CE, hence he sends the next packet with ECE set and every subsequent packet until he receives a packet with CWR set from the sender of the CE causing packet.)

knows that at least one of his sent packets would have been dropped. According to the ECN protocol he now has to react as if the packet was dropped and notify the other host that he received and reacted to the ECE via setting the CWR flag.

2.3 Netfilter

Netfilter is a part of the Linux kernel that provides a framework for various network related tasks. The probably most known program based on Netfilter is Iptables. Iptables allows to set filter and rules for incoming and outgoing IP-packets. This can for example be used to create a firewall. The structure of Netfilter can be shown by the implementation of Iptables. The rules of Iptables are organized in chains. There are 5 different chains for Iptables, as shown in figure 2.5. Each chain contains rules and filters for network packets. If a packet arrives at an interface Netfilter calls the PREROUTING chain of Iptables for this packet. Depending on the packet and the defined rules the packet enters the FORWARD chain (if the packet is not destined for the local system) or INPUT chain (the packet is for the local machine). It is possible to delete, alter or create packets and queue them at every chain, for example if the system wants to send a new packet it queues the packet in the OUTPUT chain.
These Iptables chains are implemented with Netfilter hooks. Each chain has his own hook. The input argument of a hook is the packet and the return argument is what happens with this packet and the possibly changed packet. Hooks have numbers, the hooknumber. The hook with the lowest number gets called first and has therefore the possibility to analyze the packet before all the other hooks. Every hook has complete authority over what happens with the packet with which it's called. They can decide to alter the packet, to drop it or even what the next hook should be that gets called for the packet. Returning to the Iptables example, if we look at the first chain, PREROUTING, we see that after calling the hook of this chain there are two possible next chains that can be called. This means that in one case at least one hook gets overstepped and not called. Netfilter instead continues with a higher hooknumber, similar to a jump in a normal program code.
A new hook can be registered. For this it's required to have an object of the `static struct nf_hook_ops` with a few options set. The two most important are the address of the function to call if the hook gets called and the hooknumber that decides when the hook is called.

2.3.1 Conntrack

An important feature built upon Netfilter is the possibility to track connections. The Conntrack modules delivers this functionality for TCP packets. Only with connection tracking is it possible

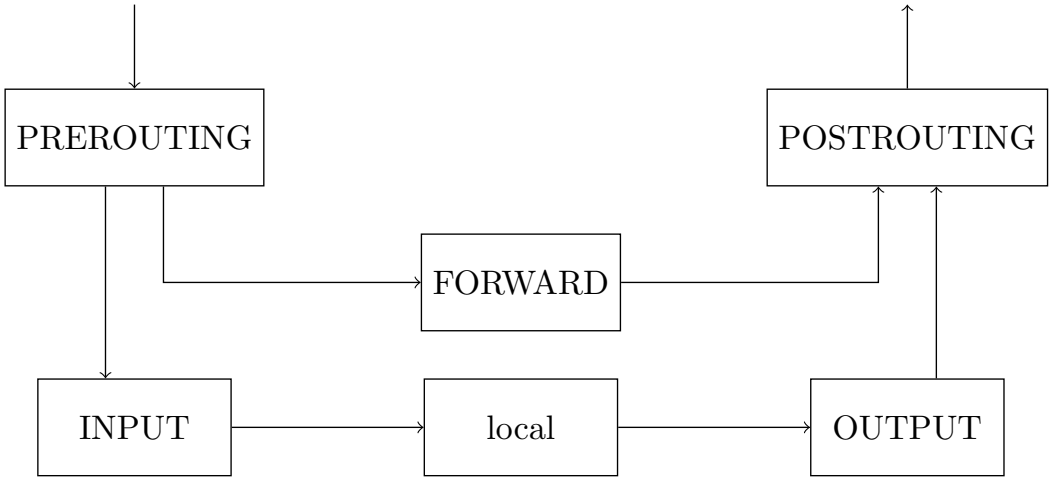


Figure 2.5: Iptables Chain Structure. Packet arrives at PREROUTING, destined either for forwarding or for the local machine. Depending on this the packet is handled by either the INPUT chain or the FORWARD chain. From the INPUT chain may come a possible new or changed packet for sending to the OUTPUT chain. The FORWARD and OUTPUT chain give their packets to the POSTROUTING chain which prepares the packet for sending

to determine if a packet belongs to an already existing connection, if it's related or if it's entirely new. The tracking is made possible by the fact that for a single TCP connection the destination and source addresses and ports stay the same. Without this module it would be impossible to make rules and filters depending on the state of a TCP connection.

Chapter 3

Design and Implementation of the Proxy

ECN can be split in two phases when looking at the whole lifetime of a TCP connection. There is the first phase where both hosts negotiate if ECN is used and the second phase when ECN is actually used. These two phases are separated in time. The proxy has therefore two main tasks for every single TCP connection:

1. In the first phase the proxy has to observe which hosts want or don't want to use ECN. It may have to change TCP headers in order to allow the server to respond with an ECN-setup-SYN-ACK.

2. The second phase starts right after to final ACK packet of the TCP three-way-handshake at the beginning of each connection. If both hosts either want or doesn't want to use ECN, the proxy can't do anything to improve the connection with ECN. But if only one of them wants to use ECN he has to simulate the corresponding counterpart for each host. Neither one of them should notice that the other host uses or doesn't use ECN. It simulates a non-ECN-host for the incapable host and for the one using ECN it simulates a partner that reacts to the marks and flags of the traversing packets.

3.1 Design

The proxy was designed as a Finite State Machine (FSM). The proxy has to change headers not only based on the current packet but has to consider previous packets. Based on this requirement it makes sense to implement the proxy as an FSM.

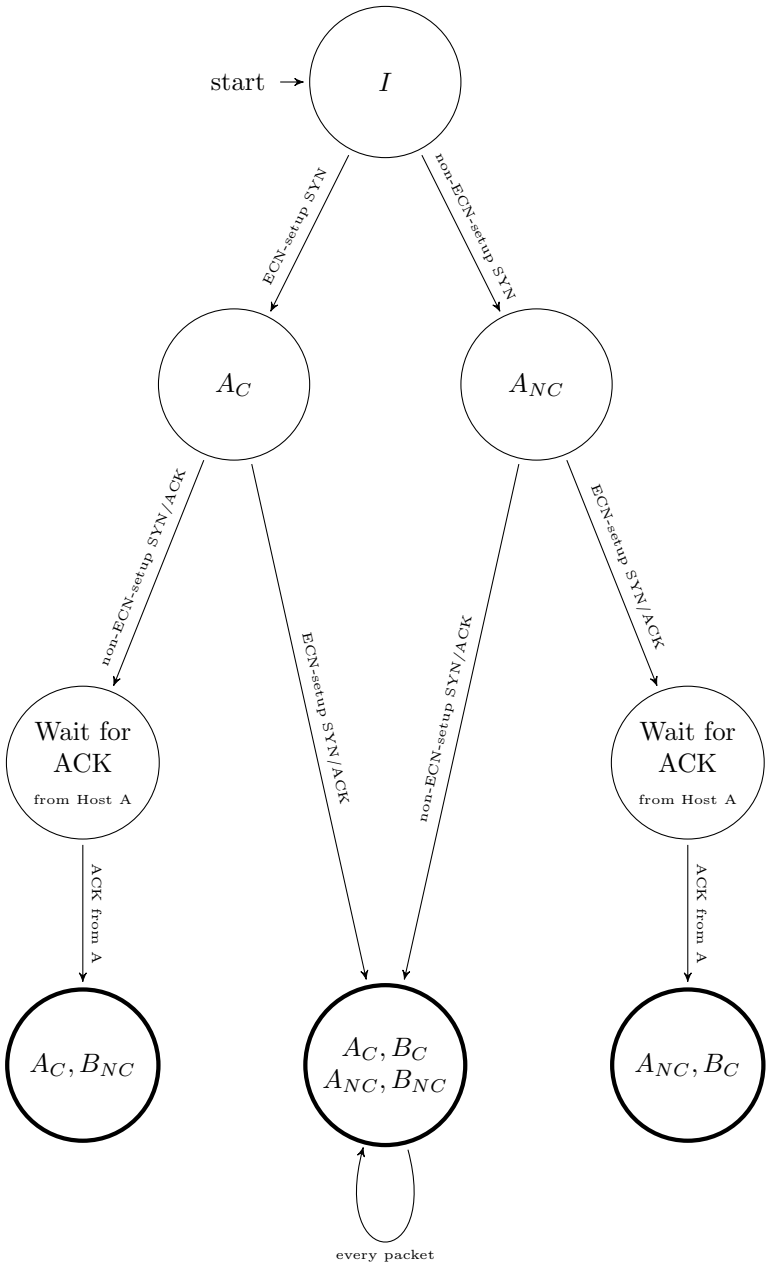
3.1.1 First Phase - Negotiation

The FSM of the first phase has a straight forward structure, similar to a binary tree, as can be seen in figure 3.1. Both hosts can either want to use ECN or not, depending on the flags set in the TCP-header. This results in four different end-states. The two cases where both either want to use it or not result in the same behavior for the proxy in the next phase and can therefore be merged.

Behavior of each State

It is assumed that Host A always initiates the connection and Host B is always the connection partner for this conversation. This halves the number of states and doesn't restrict the capability of the proxy. If host B would be the initiating host we can simply set Host B as Host A and vice versa upon receiving the first packet. In this phase only TCP-headers are changed, no IP-header is touched yet.
There are three different ending states. Depending in which state the connection is in after the handshake we continue to the second state machine or not. If only one of the hosts wants to use ECN we remember which host and continue to the state machine of the second phase.

Figure 3.1: State Diagram of the Finite State Machine determining who wants to use ECN or not during the connection establishment



- **I Initial state**
  Every connection starts in this state. The first SYN determines the next state
  State Transitions and changes in Headers
  - **if** ECN-setup-SYN **then** next_state $\rightarrow A_C$
    no change in TCP-header
  - **if** non-ECN-setup-SYN **then** next_state $\rightarrow A_{NC}$
    TCP-header changed to ECN-setup-SYN
- **A_C A ECN capable**
  Host A wants to use ECN, waiting for SYN/ACK from Host B
  State Transitions and changes in Headers
  - **if** ECN-setup-SYN/ACK **then** next_state $\rightarrow A_C, B_C$ A_{NC}, B_{NC}
    no change in TCP-header
  - **if** non-ECN-setup-SYN/ACK **then** next_state $\rightarrow$ Wait for ACK, A_C, B_{NC}-Version
    no change in TCP-header
- **A_{NC} A not ECN capable**
  Host A doesn't want to use ECN, waiting for SYN/ACK from Host B
  State Transitions and changes in Headers
  - **if** ECN-setup-SYN/ACK **then** next_state $\rightarrow$ Wait for ACK, A_{NC}, B_C-Version
    change TCP-header to non-ECN-setup-SYN/ACK
  - **if** non-ECN-setup-SYN/ACK **then** next_state $\rightarrow A_C, B_C$ A_{NC}, B_{NC}
    no change in TCP-header
- **Wait for ACK, A_C, B_{NC}-Version**
  Wait for the ACK from Host A. Last part of TCP handshake.
  State Transitions and changes in Headers
  - **if** ACK from host A **then** next_state $\rightarrow A_C, B_{NC}$
    no change in TCP-header
- **Wait for ACK, A_{NC}, B_C-Version**
  Wait for the ACK from Host A. Last part of TCP handshake.
  State Transitions and changes in Headers
  - **if** ACK from host A **then** next_state $\rightarrow A_{NC}, B_C$
    no change in TCP-header
- **A_C, B_{NC} A ECN capable, B not ECN capable**
  One host wants to use ECN the other not. This is the default state for the FSM of the second phase.
- **A_C, B_C A_{NC}, B_{NC} Both hosts the same**
  The proxy doesn't do anything for the rest of this connection. It just forwards the packets.
- **A_{NC}, B_C A not ECN capable, B ECN capable**
  One host wants to use ECN the other not. This is the default state for the FSM of the second phase.

3.1.2 Second Phase - Operation

In this phase the proxy simulates for each host the corresponding counterpart, ECN-capable or not-ECN-capable. The only information we need from the previous state machine of the first phase is which host wants to use ECN. After the connection is established there is no longer a classification in initiating and accepting host. What matters is only who uses ECN and who doesn't.

The proxy sets in this phase the ECT-codepoint in the IP-header for the traffic from the non-ECN-host (the host that doesn't use ECN) to the ECN-host (the host that uses ECN). For traffic

in the other direction it has to set the Not-ECN-codepoint. There are four different states in this second phase after the handshake, depending on the signaled congestion:

- No congestion in any direction
- Congestion in direction from ECN-host to non-ECN-host
- Congestion in direction from non-ECN-host to ECN-host
- Congestion in both directions

We know that there is congestion on the path from ECN-host to the non-ECN-host if we receive a packet with the CE mark set. The non-ECN-host shouldn't notice the ECN usage. Therefore, the proxy has to reset the CE in the packet and remember to set the ECE flag in the TCP-header for packets to the ECN-host. The proxy sets the ECE flag in every packet for the ECN-host until he receives a packet from it with the CWR flag set. The proxy has the possibility to communicate with the congestion causing host without dropping packets.
It's considerably more difficult to notify the non-ECN-host that his packet would have been dropped. We consider three solutions for this problem:

Not setting the ECT(1)-codepoint for traffic from non-ECN-host to ECN-host

After successfully negotiating ECN setting the ECT-codepoint is voluntarily. Each host can decide if he wants to set this codepoint. A simple solution is therefore to just not setting the codepoint for traffic from the non-ECN-host to the ECN-host. If the codepoint isn't set no router can set the CE codepoint. The packet would be dropped instead. The ECN-host never receives a CE and therefore never marks a packet with the ECE flag forcing us to notify the non-ECN-host. The router communicates the normal way with the congestion causing host via dropping the packet. The major drawback of this solution is that ECN isn't used on the entire path in one direction. Neither is the traffic ECN enabled nor any of the good properties of ECN have an impact in this direction.

Proxy drops the Packet instead of Router

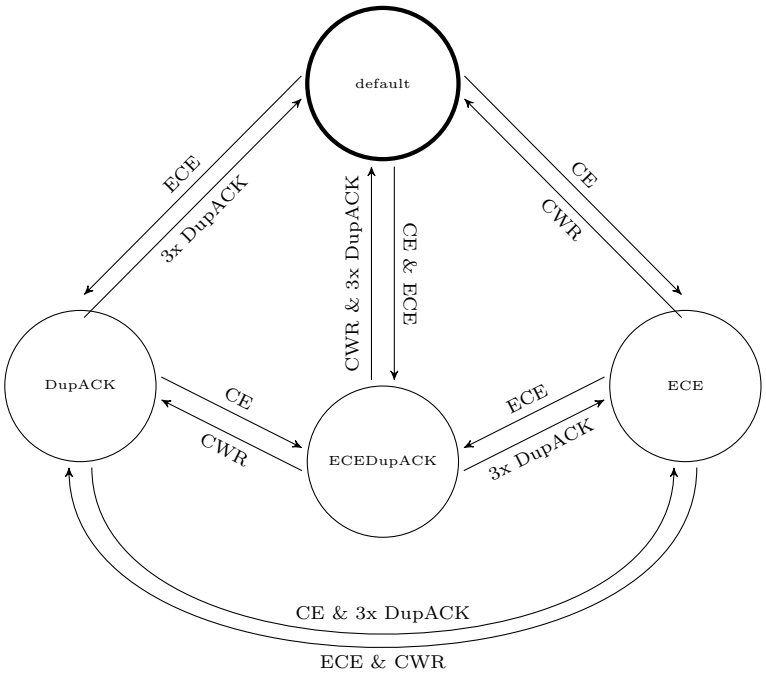
If the proxy marks packets from the non-ECN-host with ECT there will most likely be packets arriving from the ECN-host with the ECE mark set. The proxy now has to notify the non-ECN-host that a packet would have been dropped. The simplest communication is to drop the next packet from the non-ECN-host. This is similar to the first solution but with the difference that the packets have the ECT codepoint set on the path to the ECN-host. The routers on this path have the illusion that ECN is used in both ways. But still no good properties of ECN have an impact in the direction from non-ECN- to ECN-host.

Simulate dropping the Packet

The third alternative to notify the non-ECN-host is to simulate as if the packet was dropped. The sending host observes the ACK-numbers of the arriving packets. If there is three times a duplicated acknowledgment number (4 times in a row the same number) the sender assumes that the packet was dropped. The proxy can now remember the ACK-number of the packet with the ECE flag set instead of dropping the next packet. The router then changes the ACK-number of the following three packets to the same value. The ECN-host receives every packet, but the non-ECN-host is under the impression that a packet was lost. The non-ECN-host resends the packet that was never dropped. The ECN-host receives a duplicated packet and discards it, because it's already been correctly transmitted. The proxy sets the CWR flag after sending three duplicated ACK. With this solution the traffic is ECN enabled for both directions between proxy and ECN-host. The throughput stays the same as in the second solution. But one good property of ECN can be used, even if one host isn't using ECN: The decreased delay. Because the drop is only simulated the packet arrives on time at the destination and not after the drop was detected.

We decided to implement the third solution for the proxy. With this solution we are closest to the optimum from all three solutions.

Figure 3.2: State Diagram of the Finite State Machine after the connection is established



The flags indicating congestion for one direction can arrive at any time and even at the same time. This is the reason why every state is reachable from every state in the state diagram in figure 3.2. The observed flags can only be set in packets arriving from the ECN-host. There are three different flags which can cause a state transition: ECE, CWR and CE. For this FSM to work we need two variables, one to save the ACK-number and the second one to count the number of times we already sent the duplicated ACK. The "!" in front of flag means that this flag isn't set.

- **default**
  Default state. The proxy starts in this state when entering the second phase. No congestion has to be communicated in this state
  State Transitions and changes in Headers
  - **if** (Direction from non-ECN-host) **then** next_state → current_state
    Set the CET(1) mark in the IP-header
  - **if** (Direction from ECN-host) && !(CE) && !(ECE) **then** next_state → current_state
    Set the Not-ECN mark in the IP-header
  - **if** (Direction from ECN-host) && (CE) && !(ECE) **then** next_state → ECE
    Set the Not-ECN mark in the IP-header
  - **if** (Direction from ECN-host) && (CE) && (ECE) **then** next_state → ECEDupACK
    Set the Not-ECN mark in the IP-header
    Save the ACK-number of the packet
    Set the counter #ACK for duplicated ACK to zero
  - **if** (Direction from ECN-host) && !(CE) && (ECE) **then** next_state → DupACK
    Set the Not-ECN mark in the IP-header
    Save the ACK-number of the packet
    Set the counter #ACK for duplicated ACK to zero
- **DupACK**
  This state simulates a packet drop with changing the ACK-number of the current packet to a previous ACK-number. There has been congestion in the direction from non-ECN-host to ECN-host
  State Transitions and changes in Headers

- **if** (Direction from non-ECN-host) **then** next_state → current_state
  Set the CET(1) mark in the IP-header
- **if** (Direction from ECN-host) && !(CE) && !(#ACK==3) **then** next_state → current_state
  Set the Not-ECN mark in the IP-header
  Set the ACK-number to the saved ACK-number
  Add one to the #ACK counter
- **if** (Direction from ECN-host) && (CE) && !(#ACK==3) **then** next_state → ECEDupACK
  Set the Not-ECN mark in the IP-header
  Set the ACK-number to the saved ACK-number
  Add one to the #ACK counter
- **if** (Direction from ECN-host) && (CE) && (#ACK==3) **then** next_state → ECE
  Set the Not-ECN mark in the IP-header
  Set the ACK-number to the saved ACK-number
  Add one to the #ACK counter
- **if** (Direction from ECN-host) && !(CE) && (#ACK==3) **then** next_state → default
  Set the Not-ECN mark in the IP-header
  Set the ACK-number to the saved ACK-number
  Add one to the #ACK counter

• **ECEDupACK**
This state does both, simulating a packet drop and notify the ECN-host with ECE that he caused a CE codepoint. Congestion was experienced in both directions
State Transitions and changes in Headers

- **if** (Direction from non-ECN-host) **then** next_state → current_state
  Set the CET(1) mark in the IP-header
  Set the ECE flag in the TCP-header
- **if** (Direction from ECN-host) && !(CWR) && !(#ACK==3) **then** next_state → current_state
  Set the Not-ECN mark in the IP-header
  Set the ACK-number to the saved ACK-number
  Add one to the #ACK counter
- **if** (Direction from ECN-host) && (CWR) && !(#ACK==3) **then** next_state → DupACK
  Set the Not-ECN mark in the IP-header
  Set the ACK-number to the saved ACK-number
  Add one to the #ACK counter
- **if** (Direction from ECN-host) && (CWR) && (#ACK==3) **then** next_state → default
  Set the Not-ECN mark in the IP-header
  Set the ACK-number to the saved ACK-number
  Add one to the #ACK counter
- **if** (Direction from ECN-host) && !(CWR) && (#ACK==3) **then** next_state → ECE
  Set the Not-ECN mark in the IP-header
  Set the ACK-number to the saved ACK-number
  Add one to the #ACK counter

• **ECE**
Traffic from the ECN-host was marked with the CE on its way to the proxy. The proxy has to set the ECE flag until it receives a packet with the CWR flag set
State Transitions and changes in Headers

- **if** (Direction from non-ECN-host) **then** next_state → current_state
  Set the CET(1) mark in the IP-header
  Set the ECE flag in the TCP-header

- **if** (Direction from ECN-host) && !(CWR) && !(ECE) **then** next_state → current_state
  Set the Not-ECN mark in the IP-header
- **if** (Direction from ECN-host) && (CWR) && !(ECE) **then** next_state → default
  Set the Not-ECN mark in the IP-header
- **if** (Direction from ECN-host) && (CWR) && (ECE) **then** next_state → DupACK
  Set the Not-ECN mark in the IP-header
  Save the ACK-number of the packet
  Set the counter #ACK for duplicated ACK to zero
- **if** (Direction from ECN-host) && !(CWR) && (ECE) **then** next_state → ECEDupACK
  Set the Not-ECN mark in the IP-header
  Save the ACK-number of the packet
  Set the counter #ACK for duplicated ACK to zero

3.2 Implementation as a Netfilter Hook

The proxy was implemented as a Netfilter hook. The hook receives the packet in form of a sk_buff object (Socket Buffer). For our proxy to work we must be able to identify the current packet and be able to tell if it belongs to an already established connection and in what state this connection currently is. The sk_buff object is created for every new packet. There is no possibility to store the state directly in the sk_buff. Therefore, we had to find a way to store the state of a connection and we have to be able to tell if a packet belongs to an already established connection.
The Conntrack module already implemented in the kernel has a similar functionality, just the number of states is smaller and we can't overwrite these states. We decided to introduce three new variables in a header of the Conntrack module. Because of the changed header we had to rebuild the kernel with the changed header. Another possibility would have been to implement an own variant of the Conntrack module and keep track of the connections on our own, but this would have been impossible to realize in the scope of this semester thesis.
The changed header introduces three additional variables in the Conntrack module: An 8-bit variable for the current state of the proxy, a 32-bit variable for the ACK-number saved for the simulated drop and another 8-bit variable for the counter how many times the ACK-number has been repeated already.
The hook itself is a kernel module. The source code is on page 48. Like every kernel module it has an init_module() and a cleanup_module() function. These functions only register and unregister the Netfilter hook. There are a lot of small functions, often reused and therefore extracted from the main hook function. The hook function itself consists of steps which every packet goes through:

- The hook first checks if the sk_buff object is NULL and if the iphdr object, which is part of the sk_buff, is NULL. If one of them is NULL, we stop the hook and return an ACCEPT.
- Then we check if the packet is a TCP packet, we only continue if that is the case
- We then continue to retrieve the saved variables from the Conntrack module. We retrieve the current state of the proxy, the saved ACK-number, the counter of the number of duplicated ACK sent and the host that initiated the connection.
- In the next step we check if this packet is from a new connection. If this is the case, we set the state to INIT and save from which interface the first packet came.
- The next step reduces the states used for the proxy by half. By setting the direction the first packet came from to be direction A we always can assume that Host A is the initiating host - the client - and don't have to check for this later.
- The next part is a switch statement that implements the pseudo code of the two state machines of the first phase (see figure 3.1) and the second phase (see figure 3.2).
- The last thing to do now is to save the changed values back and return ACCEPT

3.2.1 Difficulties occurred during Implementation

First at all it has to be said that implementing a kernel module is a lot of work. Unlike "normal" programming where you can look up the answer to a problem in the internet and will most probably find an answer because someone before had the same problem, you don't find an answer for your questions concerning kernel code. When writing a kernel module, the source code of the kernel is your best friend. Especially if you have to find the right place where to save your state if you have to access it for following packets from the same connection.
Most of the time used for programming the proxy was used to find the functions, where the needed variables are stored and how to make sure that the called functions always return without an error. Finding out what caused an error in a kernel module can take some time. If there is an error the machine freezes and has to be restarted. Once we had all the code pieces together, like how to change a header or how to get the name of the interface, the actual translation of the pseudo code of the proxy to code written in C was rather easy and fast.

Problem with simulating a packet drop

During simple functionality tests the simulation of a packet drop didn't cause any problems. But later when the proxy was tested with higher rates of throughput the connection was disrupted after some time. We didn't find a solution for this behavior. In order to have a working proxy we decided to change the code to have the first method proposed in chapter 3.1.2 implemented. This method disables the ECT marking for traffic from the non-ECN-host to the ECN-host. This implementation worked with larger throughput and we were able to run the tests with this implementation.

3.2.2 Reasons for an Implementation as a Netfilter Hook

One of the first ideas was to implement the proxy with Iptables. This isn't a good idea mainly for two reasons:

1. The first reason is that Iptables can't change a TCP-header. We would have to use another kernel module or program that would do that for us. This is impractical, especially because Iptables isn't designed for communicating with other programs and it would be tricky to implement and debug a proxy this way. And if you would write your own netfilter module to change the TCP-header the effort to coordinate Iptables and this module would be greater than implementing the whole proxy in the netfilter hook.

2. The second reason is that keeping track of the states in Iptables itself would have been very confusing. Iptables isn't designed to implement a state machine. For connection tracking it uses Conntrack. But Conntrack itself lacks in number of states. Also with the way Iptables handles its rules it would have been horrifying to implement the proxy.

Chapter 4

Testing the Implementation of the Proxy

The main goal of the testing isn't to show how good ECN is but rather how good the proxy works compared to real ECN and not using ECN. The testing should show that the proxy is close to real ECN and at least not worse than not using the proxy.

4.1 Test Setup

To test the performance of our proxy we have to create a test environment. We need a client and a server that communicate with each other. The path between them consists of a router that is able to mark packets instead of dropping them and a machine running the proxy. The router needs therefore and Active Queue Management (AQM) that predicts congestion before it occurs. This simple setup can be seen in figure 4.1 and was used for testing the proxy without external congestion from other sources other than the client and server. In addition to this basic setup we want to guarantee congestion on the router. Therefore, we have other machines sending packets over the router. This expanded test setup is illustrated in figure 4.2. Additionally, to the server and client two hosts generate traffic on the same interfaces the proxy and server already use.
The basic setup is a subset of the expanded setup with additional hosts. When the two hosts of the expanded setup don't send any traffic it's the same as the basic setup. Therefore, we can use the expanded setup and disable the two congestion causing hosts to get the basic setup.

Figure 4.1: Basic Test Setup, Client connected to the Server via the Proxy and a Router

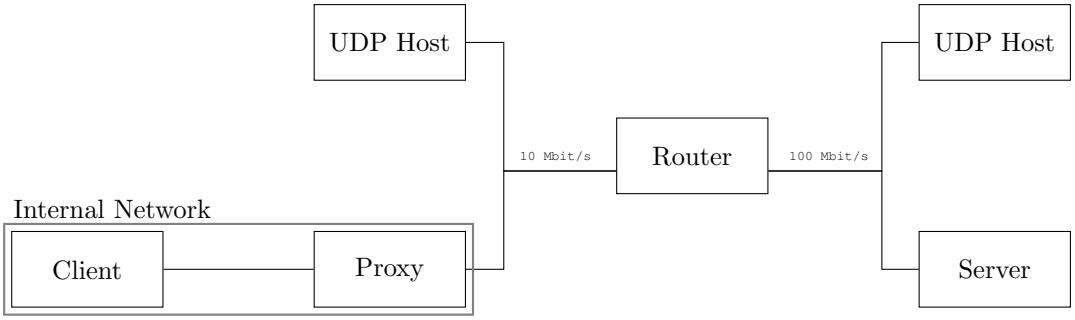


Figure 4.2: Expanded Test Setup, Client connected to the Server via the Proxy and a Router. Additionally, there are two Hosts sending IP-packets over the router to cause the buffers to fill and eventually cause congestion

4.1.1 Virtual Machines in VirtualBox

All six machines were simulated with VirtualBox. Every machine except the machine which runs the proxy have Ubuntu Server 14.04 LTS with the kernel version 3.19 installed on them. Out of convenience the proxy-machine runs on a desktop version of Ubuntu 14.04 LTS (The code was written inside this virtual machine and early testing was done with Wireshark). The IP-addresses are static and forwarding was enabled. The connections between the machines were made with internal networks of VirtualBox, the three networks were 192.168.0.0/24, 192.168.1.0/24, 192.168.2.0/24. The machines had the following mentionable specifications:

- **Client**
  The client is connected to one interface of the proxy with the internal network 192.168.0.0/24. The client has the IP-address 192.168.0.10. The standard gateway is 192.168.0.1 (the proxy).

- **Server**
  The server is in the internal network 192.168.2.0/24 with one interface of the router and a UDP host. The IP-address is 192.168.2.10 and the standard gateway is 192.168.2.1 (the router).

- **Proxy**
  The proxy has two interfaces. One interface is in the network 192.168.0.0/24 with the IP-address 192.168.0.1 and the other interface is in 192.168.1.0/24 with the address 192.168.1.1. The implemented netfilter-hook is inserted on this machine and observes packets from one of these interfaces to the other one. The standard gateway is 192.168.1.10 (the router).

- **Router**
  The router connects the test setup with the internet. The first interface is connected to the internet. The second interface is in the network 192.168.1.0/24 with the address 192.168.1.10 and the third in 192.168.2.0/24 with the address 192.168.2.1. On the two interfaces connected to the internal networks runs an AQM. The used AQM is Random Early Detection (RED) with the following parameters:

  – The RED AQM for both interfaces is set with the following parameters: Limit = 40000 bytes, Min = 2000 bytes, Max = 8000 bytes, Average Packet Size = 1000 bytes, ECN is used, Burst = 4, Drop Probability = 0.3

  These parameters are chosen very strictly. We want the router to mark the packets early, long before it would have to in order to prevent congestion. Further the bandwidths for the two internal interfaces are limited. This is done to create a bottleneck on the router. For the downloading case (server upload, client download) the bandwidth of the interface on the proxy side was limited to 10 Mbit/s and the other internal interface was limited to 100 Mbit/s. Packets arrive 10 times faster from the server than the router can send them to the client. This is the configuration depicted in figure 4.2. For the uploading case it's the other way around. The interface on the server side is 10 times slower at 10 Mbit/s than the other interface.

- **UDP Host**
  There are two UDP hosts. One in each internal network that is connected to the router. They have the IP-addresses 192.168.1.20 and 192.168.2.20 respectively. They both send UDP traffic to each other in order to cause congestion on the router.

4.2 Throughput Measurement

We tested only on a virtual internal network. For every test all the network components were simulated. VirtualBox isn't transparent for ECN traffic from a virtual machine to the real internet. Therefore, we couldn't test in the real internet with our virtual test setup.

4.2.1 Test Cases

To test the performance of the proxy we looked at different configurations of the test setup. For every configuration we tested the five possible scenarios:

1. Neither Client nor Server use ECN. Proxy netfilter-hook is not inserted.
2. Neither Client nor Server use ECN. Proxy netfilter-hook is inserted.
3. Server wants to use ECN, Client doesn't. Proxy netfilter-hook is inserted.
4. Both, client and server, want to use ECN. Proxy netfilter-hook is not inserted.
5. Both, client and server, want to use ECN. Proxy netfilter-hook is inserted.

For the throughput testing the program Iperf is used. Iperf is a network testing tool that is able to create UDP and TCP traffic and measure the throughput of this traffic. Each of these scenarios is tested with the same throughput tests:
The proxy is tested with larger files and smaller files with a variable number of simultaneous connections. All tests are run for downloading from and uploading to the server, likewise every test is run with cross traffic and without cross traffic. Cross traffic is the traffic generated by the two UDP hosts used to congest the router. To measure the time needed for transmission we use the `time` command from Linux.
Every scenario is tested with these tests:

- Download from Server
  - Without Cross Traffic
    - 1,2,3 and 5 connections transmit simultaneously each 200 MB
    - 1 connection transmits 100 times 2 MB
  - With Cross Traffic
    - 1,2,3 and 5 connections transmit simultaneously each 20 MB
    - 1 connection transmits 100 times 2 MB
- Upload to Server
  - Without Cross Traffic
    - 1,2,3 and 5 connections transmit simultaneously each 100 MB
    - 1 connection transmits 100 times 2 MB
  - With Cross Traffic
    - 1,2,3 and 5 connections transmit simultaneously each 20 MB
    - 1 connection transmits 200 times 2 MB

Summarized we run $|\text{Upload, Download}| * |\text{Cross Traffic, No Cross Traffic}| * |\text{1,2,3,5 connections \& small files}| * |\text{\# of configurations}| = 2 * 2 * 5 * 5 = 100$ different tests.

4.2.2 Automated Testing and Changing of Test Parameters

We used bash scripts for an automated testing. There are two different scripts for testing the throughput. The first one, listed like all the scripts used in listing D.4 in appendix D on page 61, is used to test the throughput without the cross traffic from the two UDP hosts. The second, listed in listing D.5, is used if there is cross traffic. These two scripts are executed from another script which has the task to first set up the correct test environment before running the corresponding tests. This script is listed in listing D.3 on page 59.
Only switching from uploading to downloading requires a change impossible to make inside a running virtual machine. To change from downloading to uploading the bottleneck has to be changed. The bandwidth limitations have to be reversed in order to guarantee congestion on the router. All the other changes between different tests can be done inside the running virtual machines. The only problem is the communication between the different machines. When the uploader finishes a test he has to signal the other machines to change their configuration.

We solved this problem with Netcat. This is a simple program to read and write from network connections. With Netcat we can set up a simple web server. This web server waits until it gets called once and sends the message it has to. After this the program terminates and the next command of the script can be executed. We therefore set up a command chain, the chain is illustrated in figure 4.3. If the uploader wants to change a setting, he signals it to the downloader. The downloader periodically checks for an update from the uploader. If there is a setup change signaled, he checks if it's a change meant for him or if another machine has to change a setting. If the message is for another machine the downloader himself sets up a web server and waits for the next machine in the chain to read from this webserver. Each machine that sends a message can only continue if the next machine in the chain has read this message. It is important that the uploader doesn't start with his test until every machine got their setting update and reacted to it. Therefore, we have to make sure that the last message is always to the downloader. The downloader only reads this message if he has nothing else to do and has therefore successfully signaled all previous messages not meant for him. The scripts used for this command-chain are listed in the appendix on page 62ff.

Figure 4.3: Signaling Chain for automatically changing the Configuration of the Test Setup

4.3 Results

We expect the proxy to improve the throughput in the downloading scenario if ECN would improve the throughput. In the uploading scenario we expect the proxy to be at least as good as the non-ECN scenarios. The proxy should never slow down the traffic. Instead it should increase the throughput as much as possible.
Every test for each test scenario was taken multiple times in order to get an average performance. The results of every test are listed in the tables in appendix C on page 42 and the box-plots graphically representing these results are listed in appendix B on page 27.

4.3.1 Downloading from an ECN capable Server

Representative for all the results obtained when downloading from the server are the two figures 4.4 and 4.5. The first figure, figure 4.4, is the boxplot of the time it took to simultaneously download three times 200 MB from the server to the client without any cross traffic. The second figure, figure 4.5, shows the results of the same scenario but only downloading 20 MB per connection and with cross traffic.

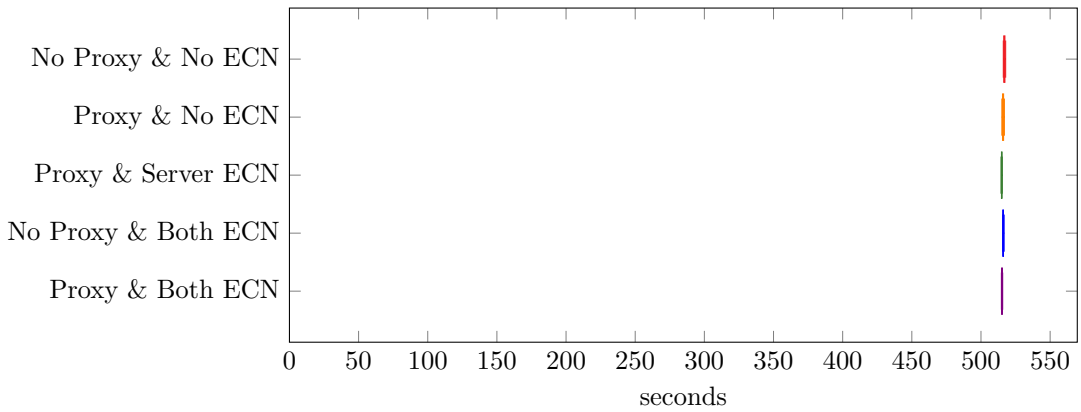


Figure 4.4: Throughput analysis with 3 simultaneous connections, a transfer of 200 MB per connection and no Cross-Traffic, Server Upload and Host Download

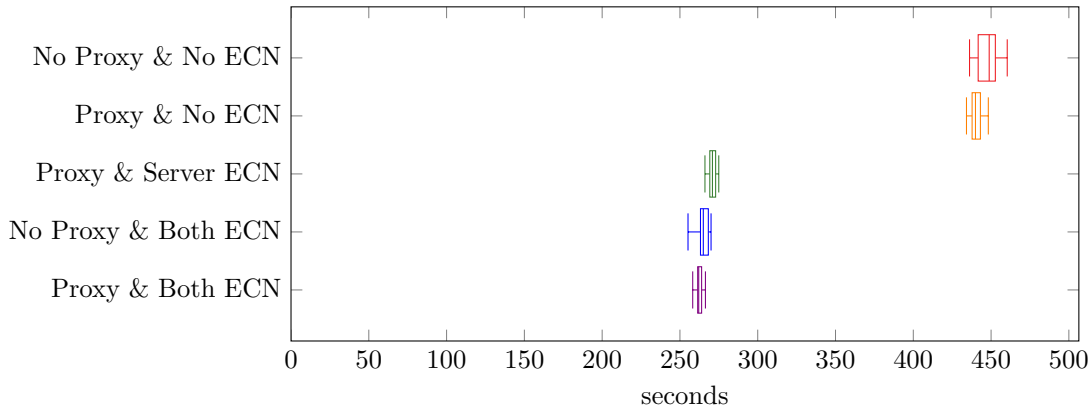


Figure 4.5: Throughput analysis with 3 simultaneous connections, a transfer of 20 MB per connection and Cross-Traffic, Server Upload and Host Download

The differences in the case without any cross traffic are very small. It seems that the pure ECN was a little bit faster but with the small sample size we took that could just be noise. All 5 configurations are very close to each other. This behavior can be explained if you look at Wireshark (A program that captures packets sent over a certain interface). Although we tried to make a bottleneck on the router the number of dropped packets (and therefore retransmissions) or CE codepoints (if ECN is used), is very small. Without cross traffic and therefore only one source and destination the router doesn't have to drop or mark packets very often and therefore using ECN is nearly equal to not using ECN.
Having a look at the case when there is cross traffic we see that the performance of the real ECN is much better than that of not using ECN. We see as well that the performance of the proxy, where the ECN is only simulated, is close to the real one. Analyzing the traffic with Wireshark we see a significant increase in the number of dropped or CE marked packets. This explains the difference in the time required to transmit the 200 MB.

We can say that the proxy works as it should for the case when a client wants to download something from a server and the proxy is installed in the clients' internal network. If there is no congestion the proxy doesn't slow down the throughput and if there is, the proxy is nearly as good as the real ECN. The proxy simulates ECN as expected in the downloading case.

4.3.2 Uploading to an ECN capable Server

We had some problems with the proxy in the case when the client uploads to the server. As mentioned in chapter 3.2.1 the proxy stopped working for the current connection after a few thousand packets were sent. We didn't find the bug and changed the program code such that the packets from the non-ECN-host to the ECN-host no longer are marked with the ECT code point indicating that this packet can be marked instead of dropped. The proxy works with this simplification but with the drawback mentioned in chapter 3.1.2.
Like in the downloading case all results obtained yield to the same conclusion. Representative for all results we selected the two diagrams illustrating the throughput tests with 3 simultaneous connections. Figure 4.6 display the results obtained when each connection uploaded 100 MB from the client to the server over a router not congested with traffic from the two UDP hosts. In figure 4.7 the results of uploading 20 MB each with a congested path introduced by the UDP hosts are shown.
Compared to the results obtained in the downloading case without cross traffic, the results for uploading to the server without congestion are much more spread (figure 4.6). But the results itself stays the same. Without congestion ECN usage and not using ECN is not clearly separable. The proxy too is in the same range and takes the same time to complete the task. We can say that the proxy doesn't slow down the throughput for this scenario.
Having a look at figure 4.7 we see that ECN enabled traffic is for the scenario with cross traffic distinctively faster than without using ECN. The proxy was changed to not mark the traffic with the ECT mark and therefore not using ECN for the uploading direction. The performance

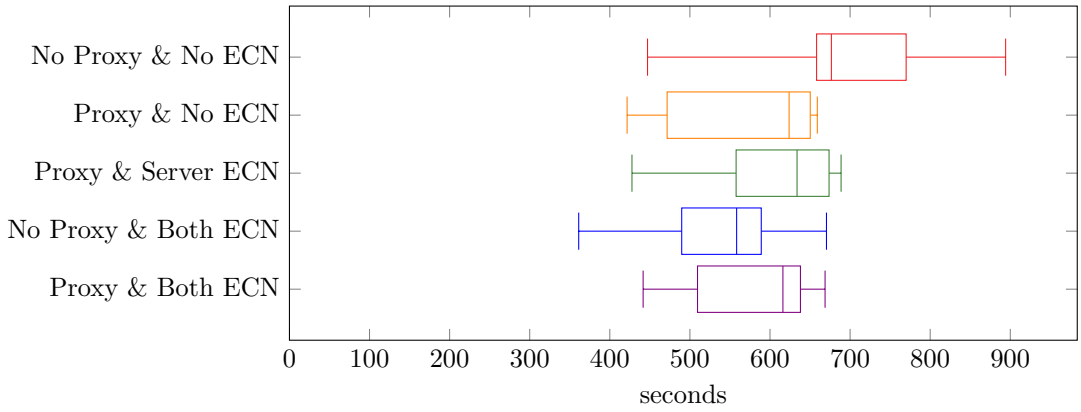


Figure 4.6: Throughput analysis with 3 simultaneous connections, a transfer of 100 MB per connection and no Cross-Traffic, Client Upload and Server Download. **One Outlier was removed from `noEcn_noProxy_noCross`**

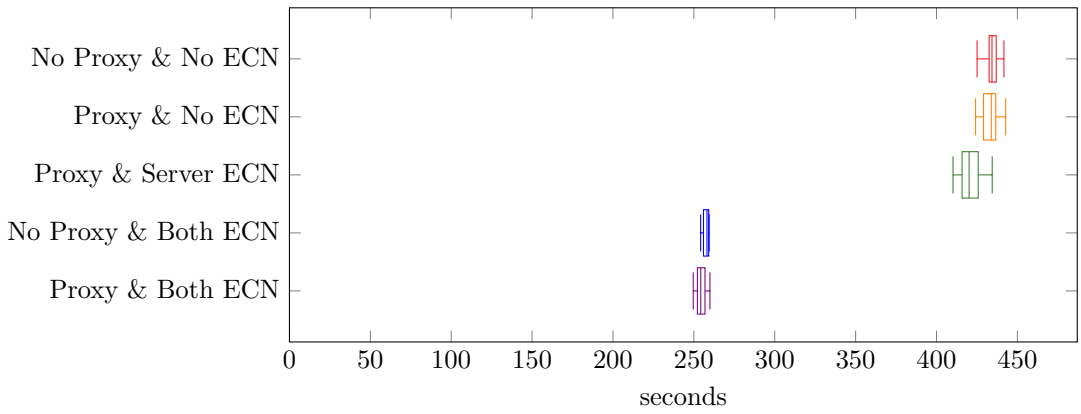


Figure 4.7: Throughput analysis with 3 simultaneous connections, a transfer of 20 MB per connection and Cross-Traffic, Client Upload and Server Download. **One Outlier was removed from `noEcn_proxy_cross`**

of the proxy simulating ECN in the downloading direction is a little bit better than not using ECN but clearly slower than the real ECN. It seems the simulated ECN for downloading from the server has some influence on the throughput but the performance is clearly similar to the scenarios where ECN isn't used.

In the uploading scenario the proxy behaves as it should. Because of the forced change in the program code due to the bug the proxy can't profit from ECN for the packets from client to server. As expected the proxy doesn't slow down the throughput in the uploading scenario.

Chapter 5

Conclusion

5.1 Improvements for a Real World Deployment

Before the proxy can be installed on home routers all over the world it needs some improvements:

- In the current implementation a change in a header file of the Linux kernel is needed and therefore a rebuilding of the kernel. This is much too much effort for most of the possible users of this proxy. If the proxy would handle the connection tracking on its own without changing the Conntrack module it would be standalone and wouldn't need a rebuilding of the kernel. This change takes a lot of work but would enormously improve the usability of the proxy.
- Another change would be to make the interfaces generic and to enable the support of more than two. Right now the interfaces are hardcoded in the code and are limited to two. For a router with more than two interfaces the current implementation doesn't work.
- The support of IPv6 wasn't considered for the current implementation and has to be tested and if necessary included.
- The configurability from user space might be a deciding criterion for some users and would improve the usability of the proxy
- For a user it would be interesting to have some statistics on the percentage of ECN usage and simulation, how many devices already use ECN and for how many devices the proxy simulates ECN.

5.2 Conclusion and Outlook

We showed that the idea of a proxy that negotiates ECN with an end-node if the other end-node doesn't want to use it works. For the scenario where a client downloads data from an ECN capable server the proxy comes close to the performance to end-to-end ECN. This scenario is the most common use for a normal user at home. If he watches YouTube videos, uses a video streaming provider like Netflix or just browses through the internet, the user would profit from using ECN.

In the other scenario where the user uploads to an ECN enabled server, for example synchronizing pictures with Dropbox, the proxy doesn't do any damage. The throughput is at least as good as without the proxy. With the error found and fixed that caused the problem in the uploading scenario the proxy would most probably even improve the throughput. With a proxy that works as intended for the uploading case it would also be interesting to see what happens if both server and client are behind such a proxy. With the current implementation they would negotiate ECN but not mark the packets for either direction. But it would be interesting to see the result when they both simulate a packet drop when they receive an ECE.

If someone would make the effort and implement the proxy as a standalone kernel module, the proxy would have the possibility to impact the ECN usage measurably, at least on a regional

scale. For example, if Swisscom, Sunrise or another service provider would decide to use the proxy for every of their routers, the usage of ECN would greatly increase for traffic originating in Switzerland. With the ECN proxy we have an instrument to enable ECN on traffic independently of manufacturers and OS developers. Every a little bit tech-savvy user that has access to a router has with this proxy the tool to improve the TCP traffic for all the end-nodes that use that router.

Appendix B

Results of the Throughput Testing

Downloading from the Server

Multiple Connections, one File per connection

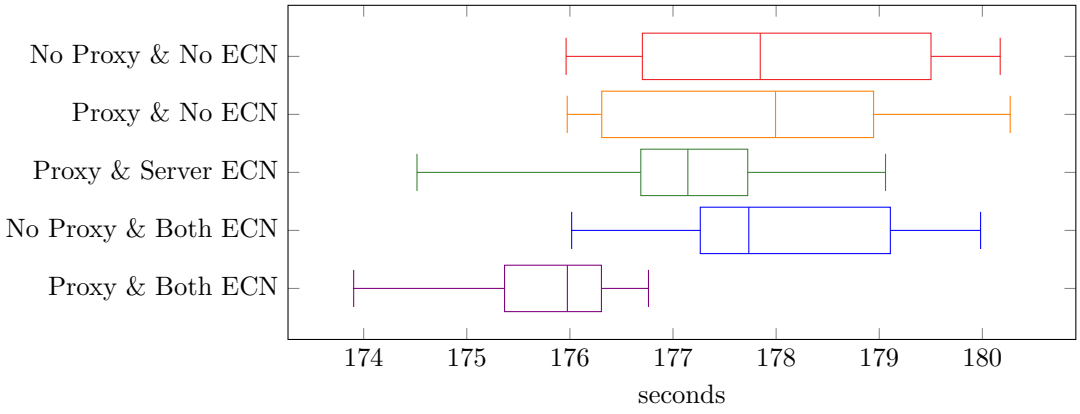


Figure B.1: Throughput analysis with 1 simultaneous connection, a transfer of 200 MB per connection and no Cross-Traffic, Server Upload and Host Download

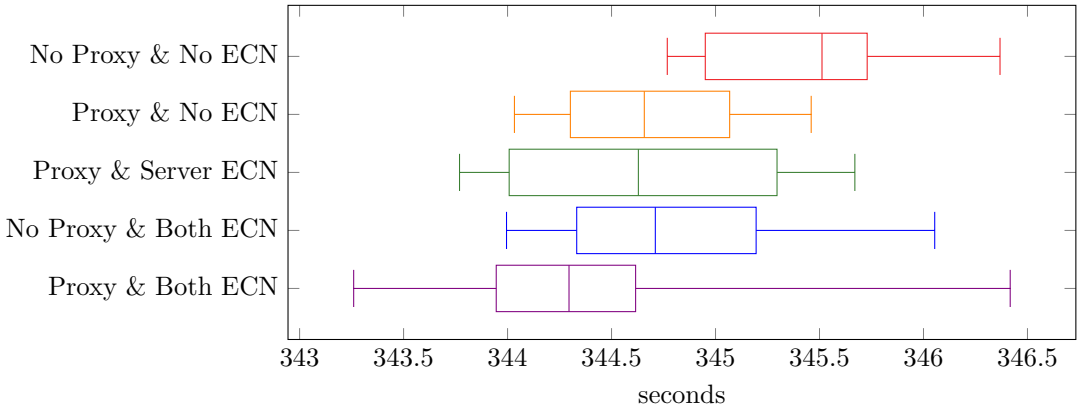


Figure B.2: Throughput analysis with 2 simultaneous connections, a transfer of 200 MB per connection and no Cross-Traffic, Server Upload and Host Download

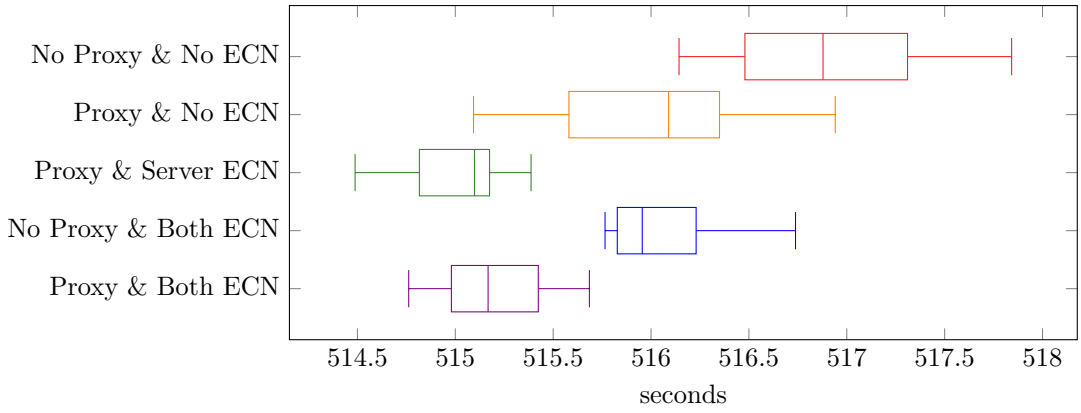


Figure B.3: Throughput analysis with 3 simultaneous connections, a transfer of 200 MB per connection and no Cross-Traffic, Server Upload and Host Download

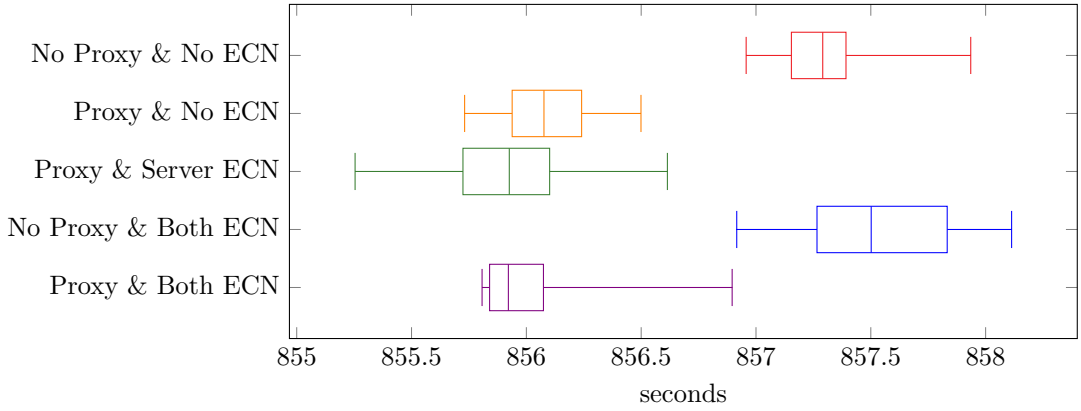


Figure B.4: Throughput analysis with 5 simultaneous connections, a transfer of 200 MB per connection and no Cross-Traffic, Server Upload and Host Download

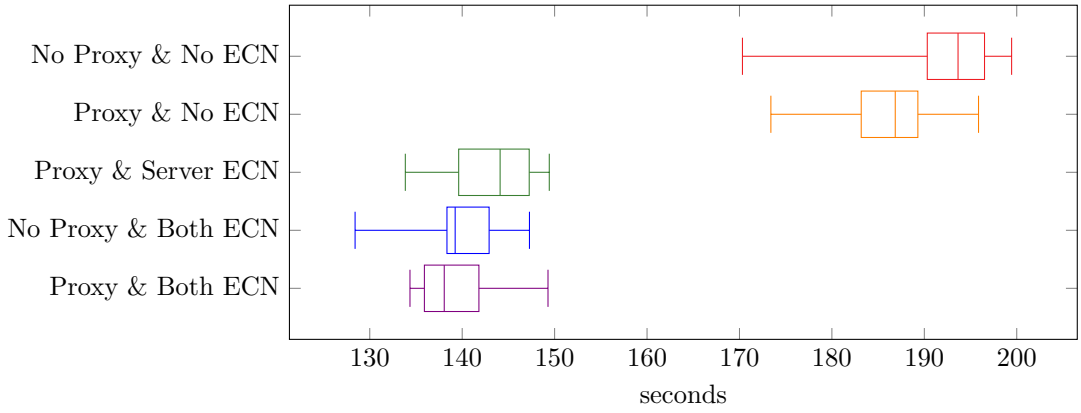


Figure B.5: Throughput analysis with 1 simultaneous connection, a transfer of 20 MB per connection and Cross-Traffic, Server Upload and Host Download

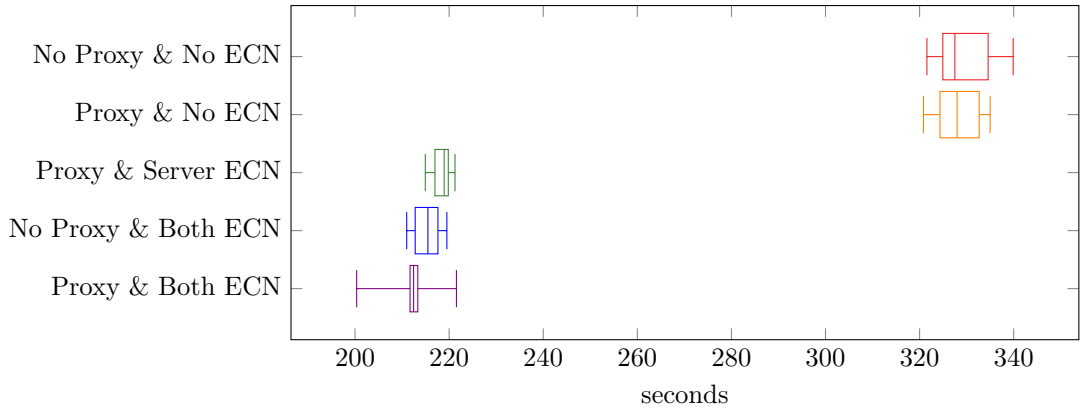


Figure B.6: Throughput analysis with 2 simultaneous connections, a transfer of 20 MB per connection and Cross-Traffic, Server Upload and Host Download

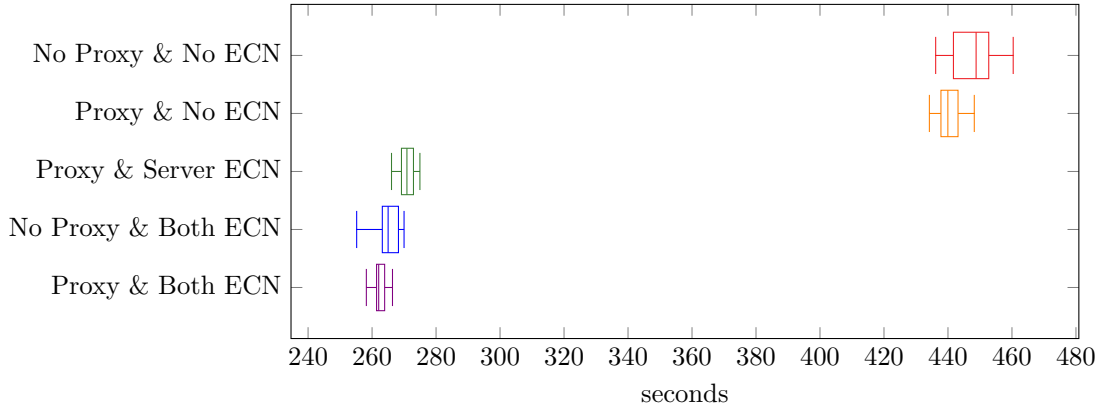


Figure B.7: Throughput analysis with 3 simultaneous connections, a transfer of 20 MB per connection and Cross-Traffic, Server Upload and Host Download

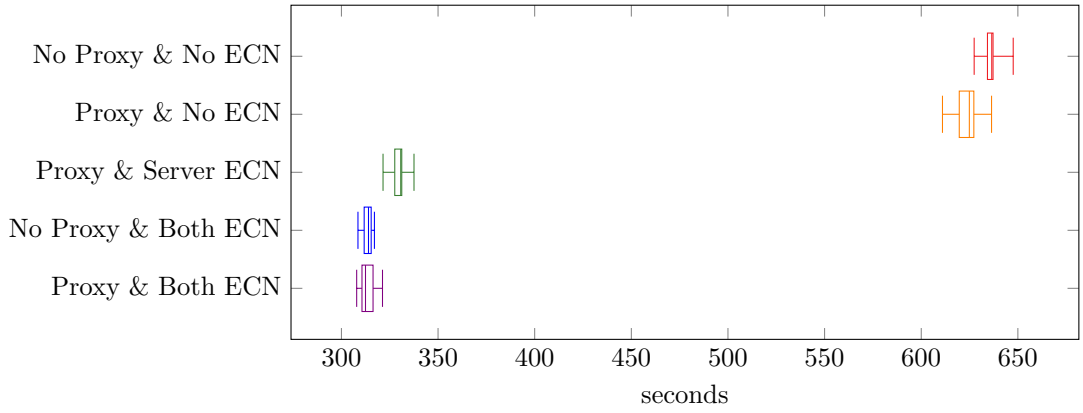


Figure B.8: Throughput analysis with 5 simultaneous connections, a transfer of 20 MB per connection and Cross-Traffic, Server Upload and Host Download

Multiple Connections, one File per connection, with respect to zero

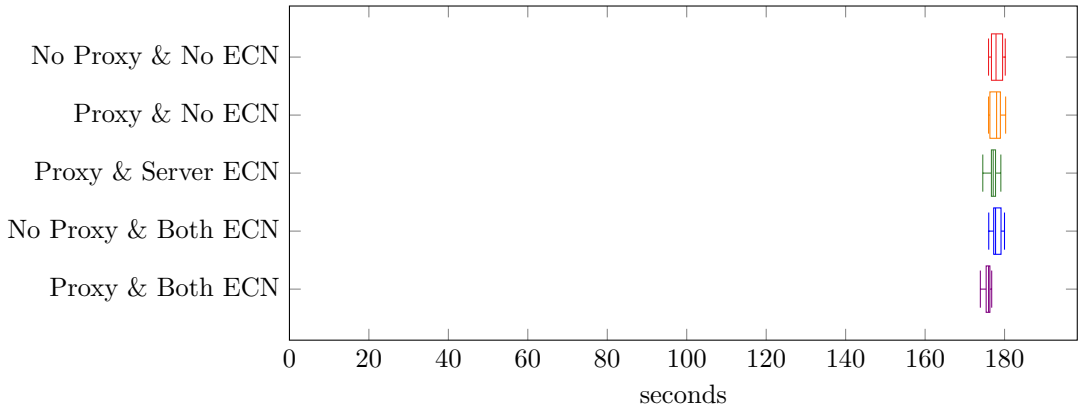


Figure B.9: Throughput analysis with 1 simultaneous connection, a transfer of 200 MB per connection and no Cross-Traffic, Server Upload and Host Download

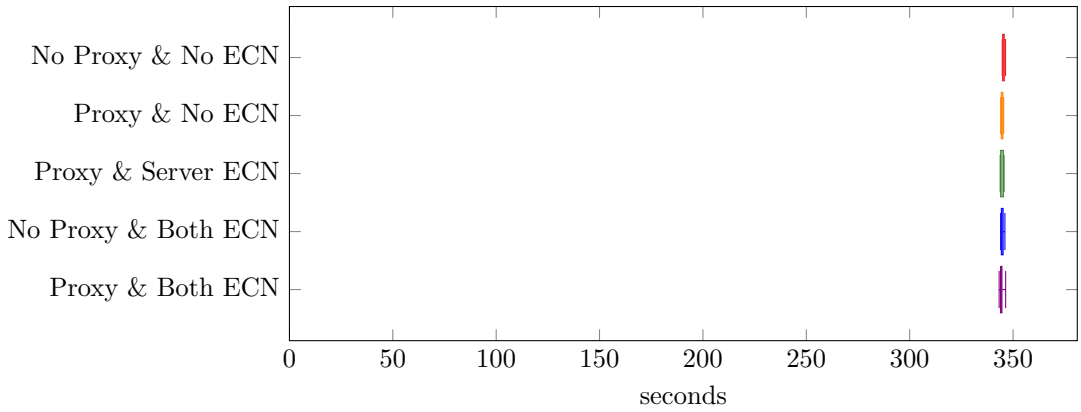


Figure B.10: Throughput analysis with 2 simultaneous connections, a transfer of 200 MB per connection and no Cross-Traffic, Server Upload and Host Download

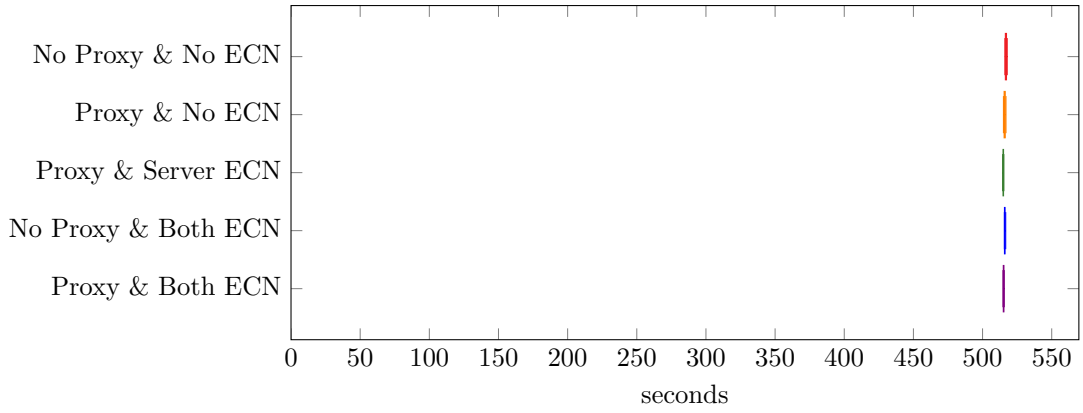


Figure B.11: Throughput analysis with 3 simultaneous connections, a transfer of 200 MB per connection and no Cross-Traffic, Server Upload and Host Download

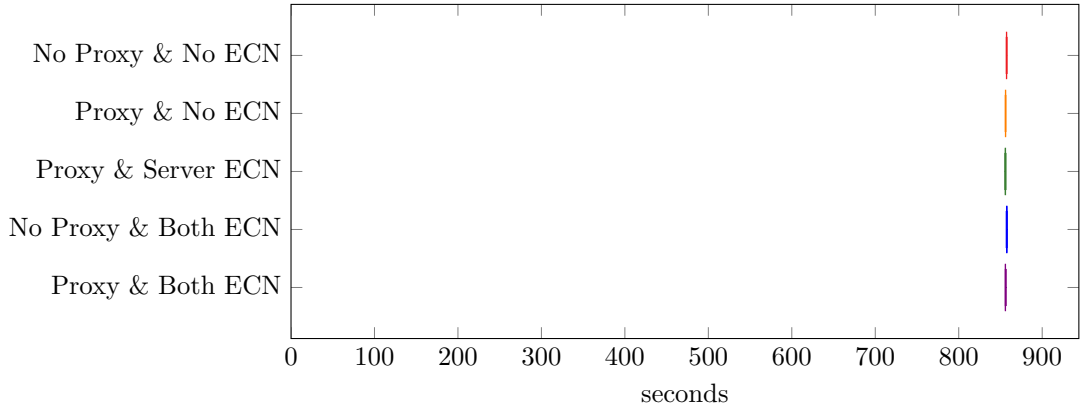


Figure B.12: Throughput analysis with 5 simultaneous connections, a transfer of 200 MB per connection and no Cross-Traffic, Server Upload and Host Download

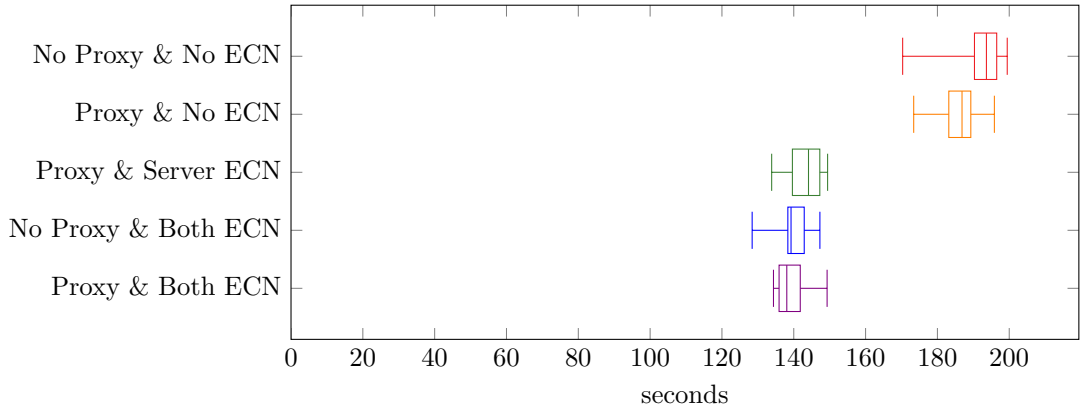


Figure B.13: Throughput analysis with 1 simultaneous connection, a transfer of 20 MB per connection and Cross-Traffic, Server Upload and Host Download

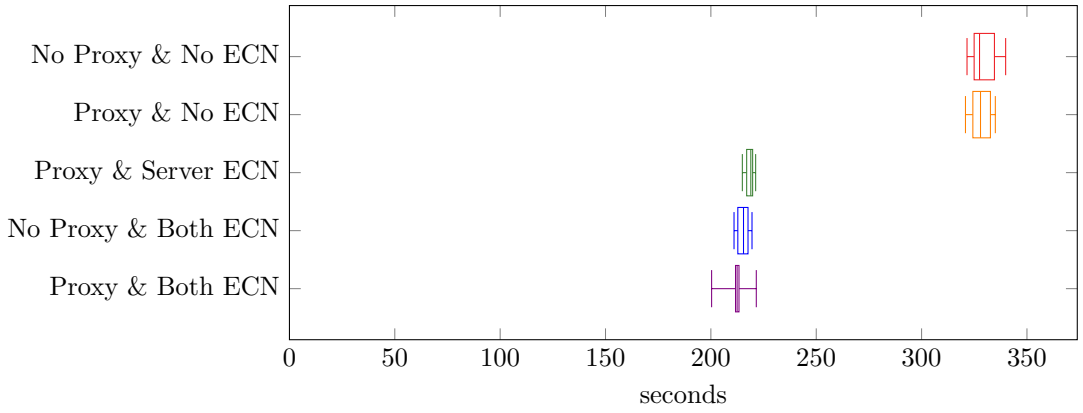


Figure B.14: Throughput analysis with 2 simultaneous connections, a transfer of 20 MB per connection and Cross-Traffic, Server Upload and Host Download

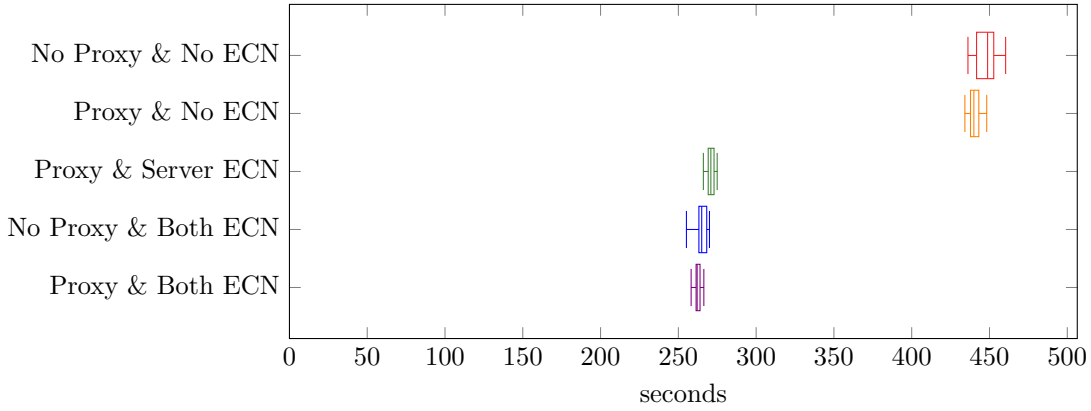


Figure B.15: Throughput analysis with 3 simultaneous connections, a transfer of 20 MB per connection and Cross-Traffic, Server Upload and Host Download

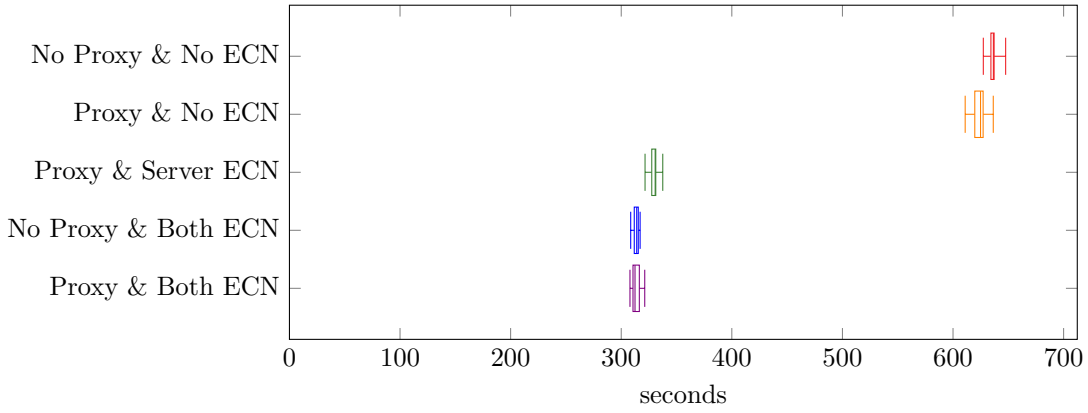


Figure B.16: Throughput analysis with 5 simultaneous connections, a transfer of 20 MB per connection and Cross-Traffic, Server Upload and Host Download

One connection, multiple files

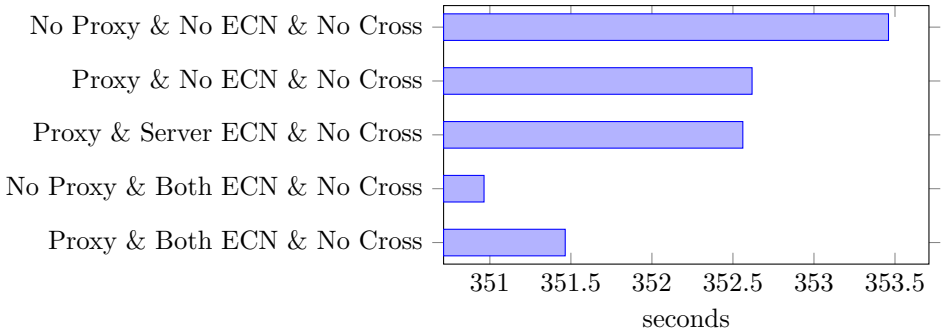


Figure B.17: Throughput analysis with successively transmitting 100 times a file of size 2 MB, no Cross-Traffic, Server Upload and Client Download

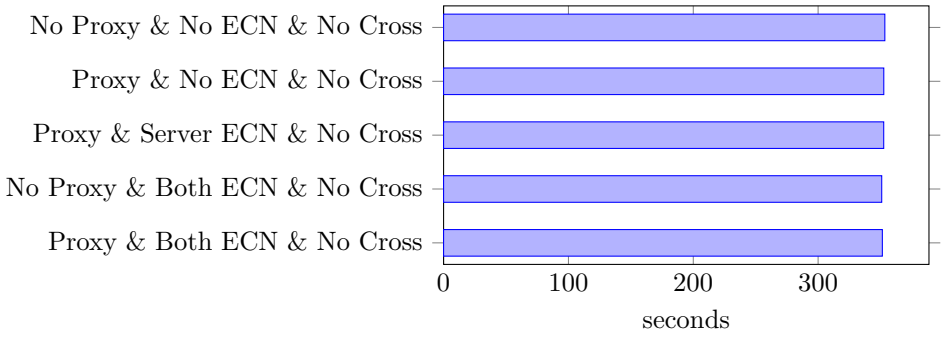


Figure B.18: Throughput analysis with successively transmitting 100 times a file of size 2 MB, no Cross-Traffic, Server Upload and Client Download, plotted with respect to zero

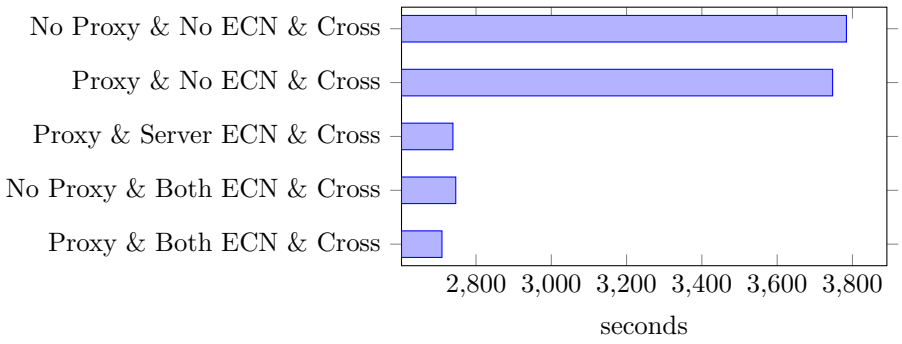


Figure B.19: Throughput analysis with successively transmitting 100 times a file of size 2 MB, with Cross-Traffic, Server Upload and Client Download

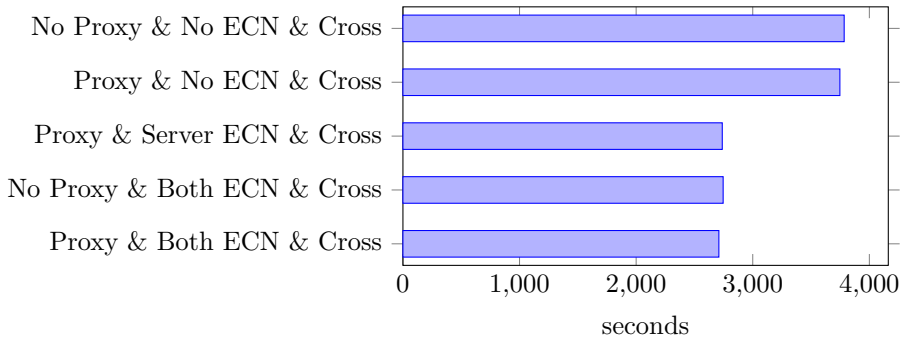


Figure B.20: Throughput analysis with successively transmitting 100 times a file of size 2 MB, with Cross-Traffic, Server Upload and Client Download, plotted with respect to zero

Uploading to the Server

Multiple Connections, one File per connection

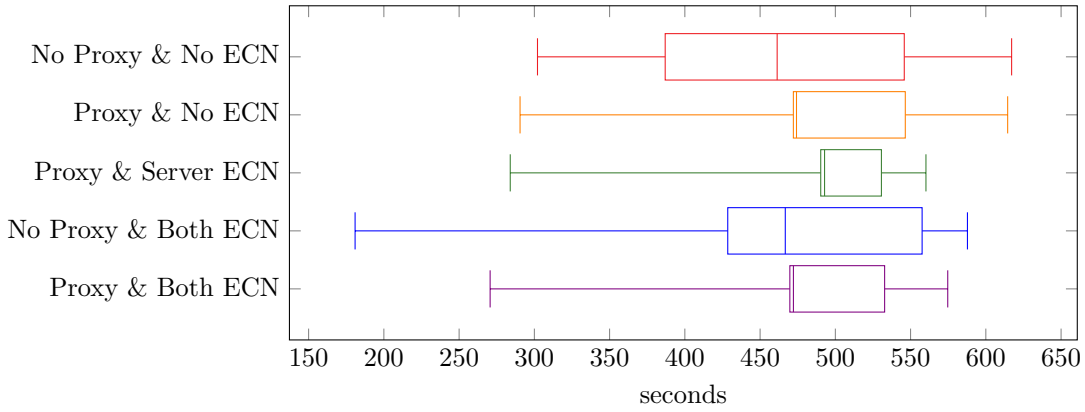


Figure B.21: Throughput analysis with 1 simultaneous connection, a transfer of 100 MB per connection and no Cross-Traffic, Client Upload and Server Download. **One Outlier was removed from `noEcn_noProxy_noCross`**

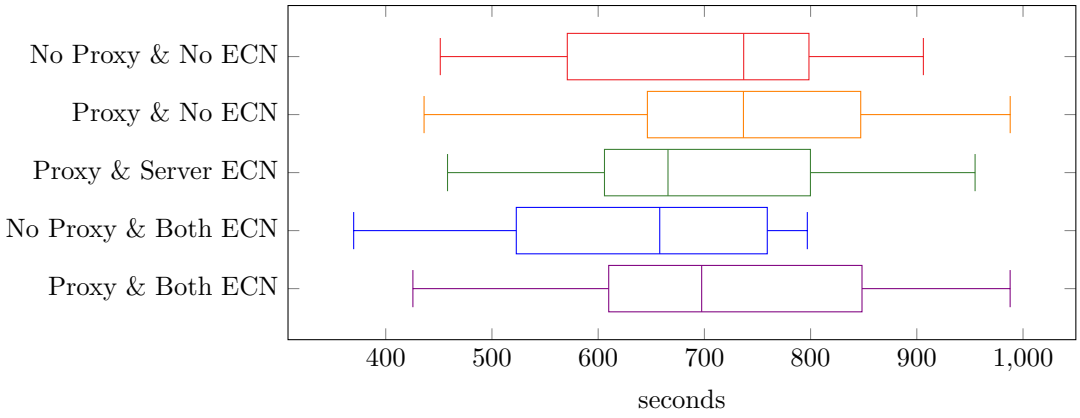


Figure B.22: Throughput analysis with 2 simultaneous connections, a transfer of 100 MB per connection and no Cross-Traffic, Client Upload and Server Download

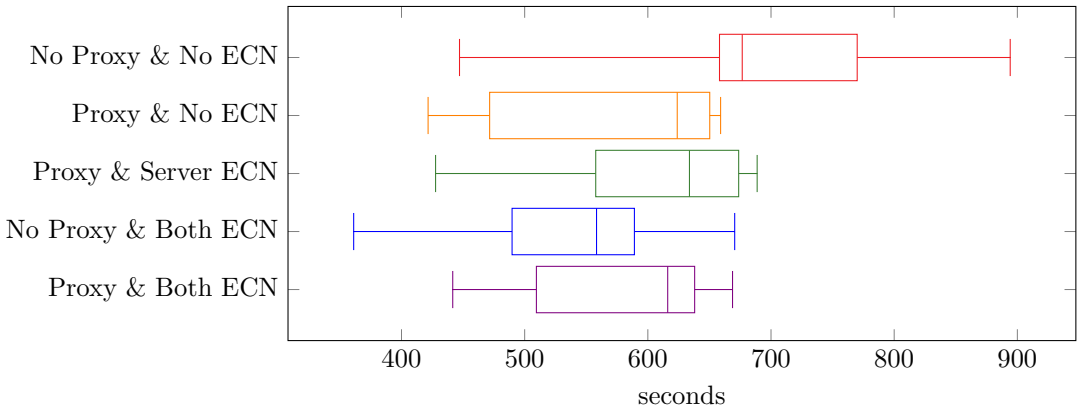


Figure B.23: Throughput analysis with 3 simultaneous connections, a transfer of 100 MB per connection and no Cross-Traffic, Client Upload and Server Download. **One Outlier was removed from `noEcn_noProxy_noCross`**

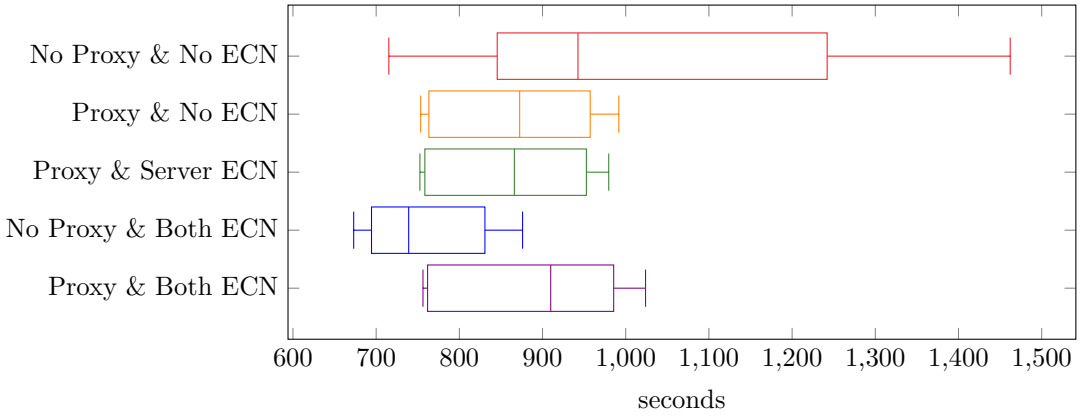


Figure B.24: Throughput analysis with 5 simultaneous connections, a transfer of 100 MB per connection and no Cross-Traffic, Client Upload and Server Download

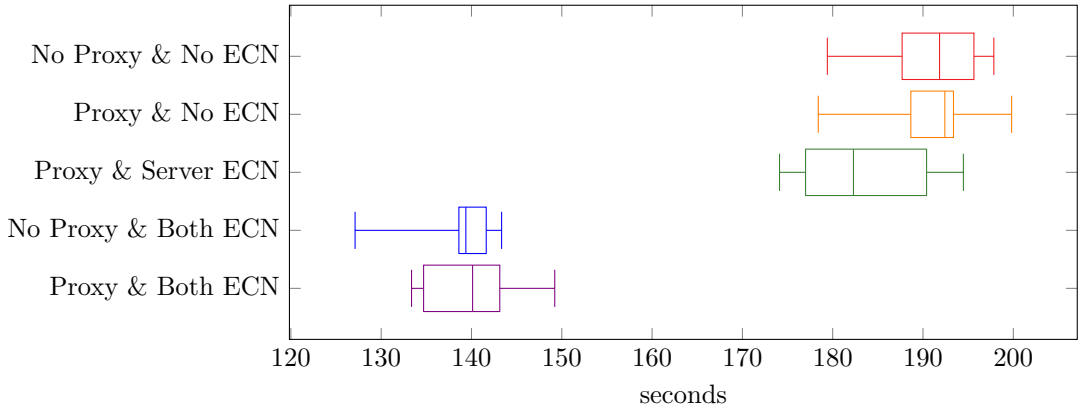


Figure B.25: Throughput analysis with 1 simultaneous connection, a transfer of 20 MB per connection and Cross-Traffic, Client Upload and Server Download

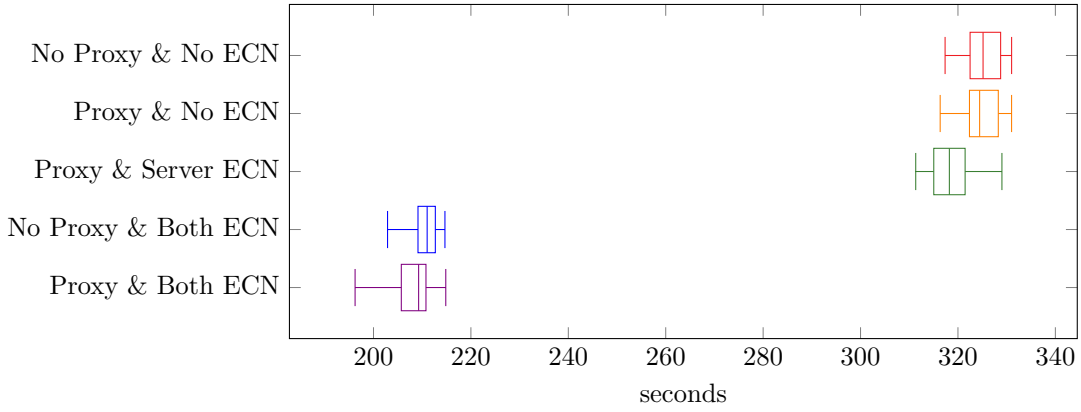


Figure B.26: Throughput analysis with 2 simultaneous connections, a transfer of 20 MB per connection and Cross-Traffic, Client Upload and Server Download

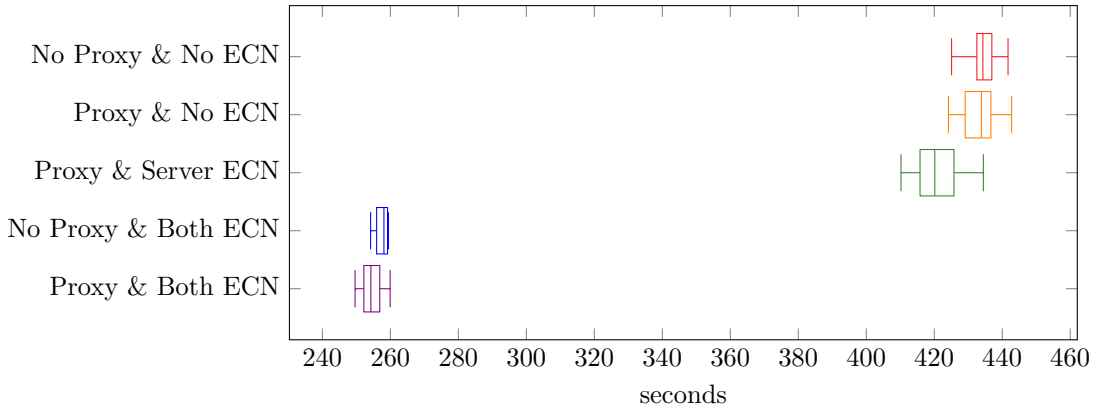


Figure B.27: Throughput analysis with 3 simultaneous connections, a transfer of 20 MB per connection and Cross-Traffic, Client Upload and Server Download. **One Outlier was removed from `noEcn_proxy_cross`**

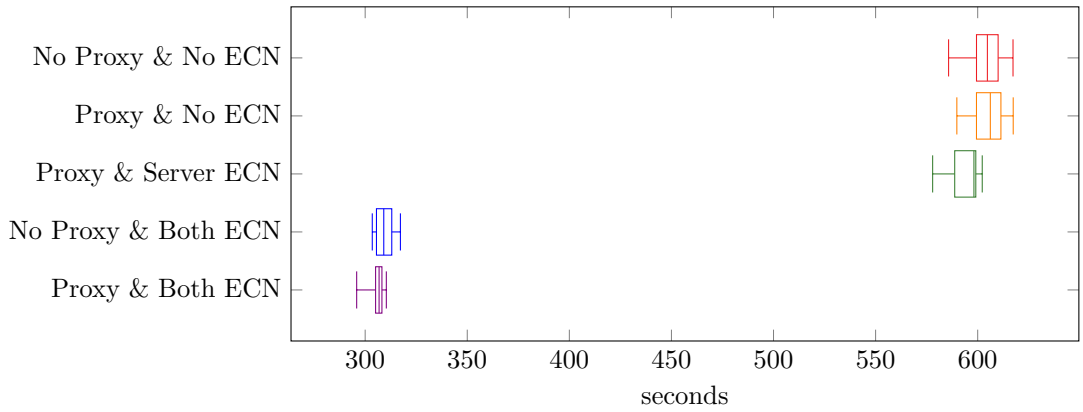


Figure B.28: Throughput analysis with 5 simultaneous connections, a transfer of 20 MB per connection and Cross-Traffic, Client Upload and Server Download

Multiple Connections, one File per connection, with respect to zero

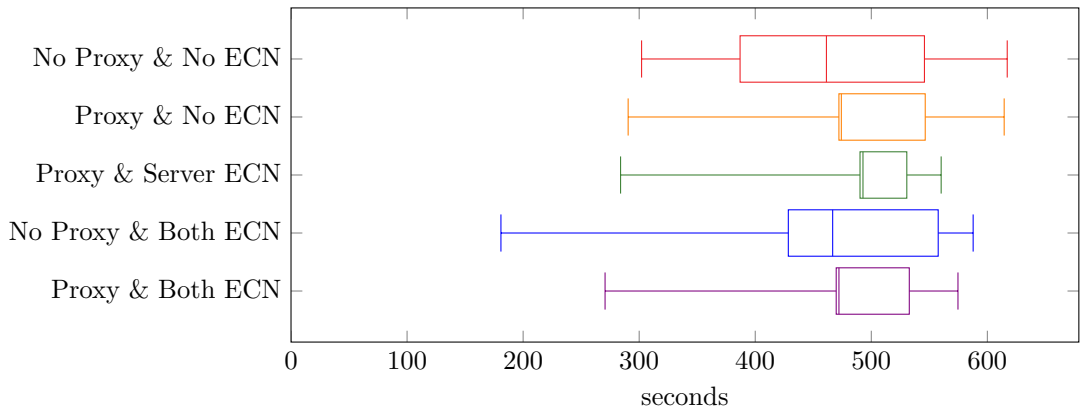


Figure B.29: Throughput analysis with 1 simultaneous connection, a transfer of 100 MB per connection and no Cross-Traffic, Client Upload and Server Download. **One Outlier was removed from `noEcn_noProxy_noCross`**

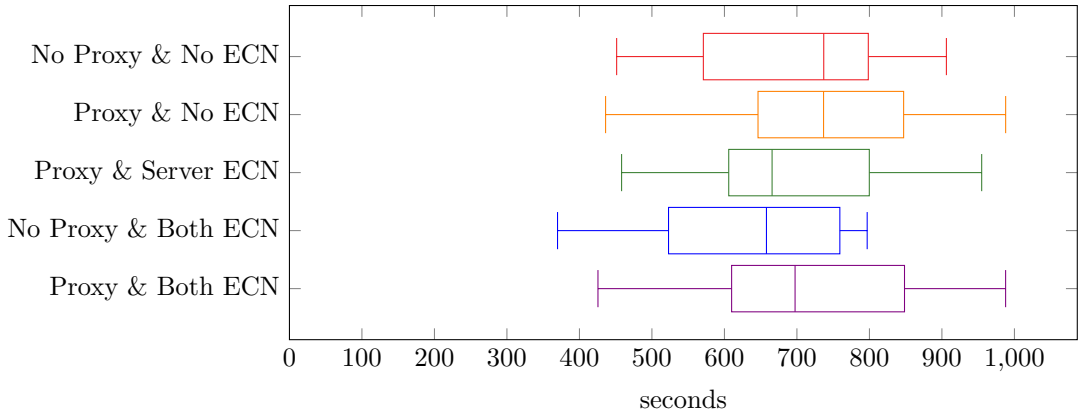


Figure B.30: Throughput analysis with 2 simultaneous connections, a transfer of 100 MB per connection and no Cross-Traffic, Client Upload and Server Download

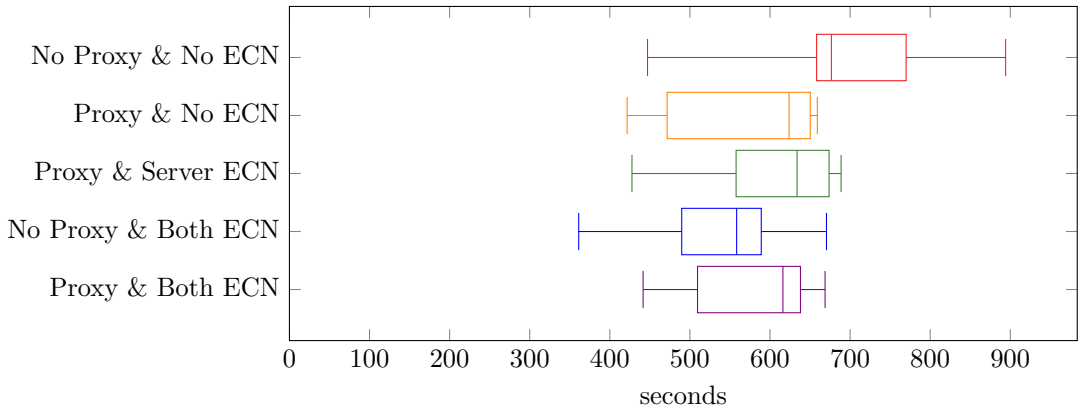


Figure B.31: Throughput analysis with 3 simultaneous connections, a transfer of 100 MB per connection and no Cross-Traffic, Client Upload and Server Download. **One Outlier was removed from `noEcn_noProxy_noCross`**

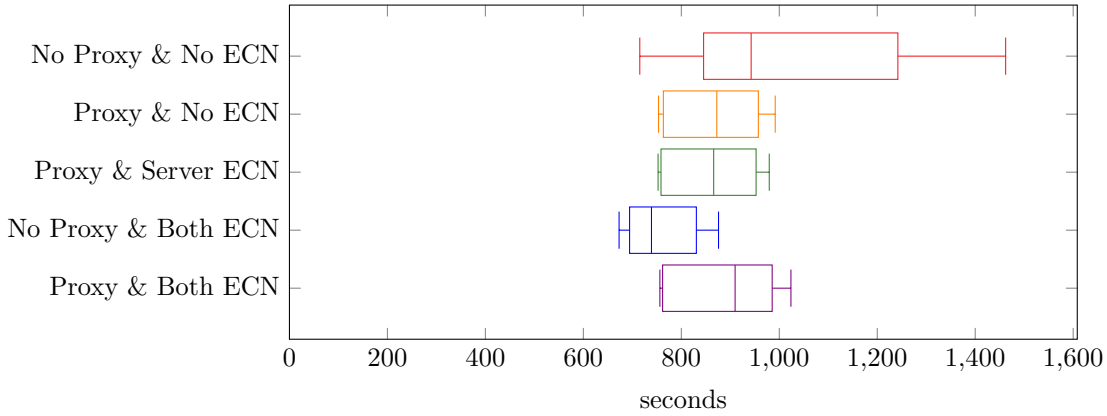


Figure B.32: Throughput analysis with 5 simultaneous connections, a transfer of 100 MB per connection and no Cross-Traffic, Client Upload and Server Download

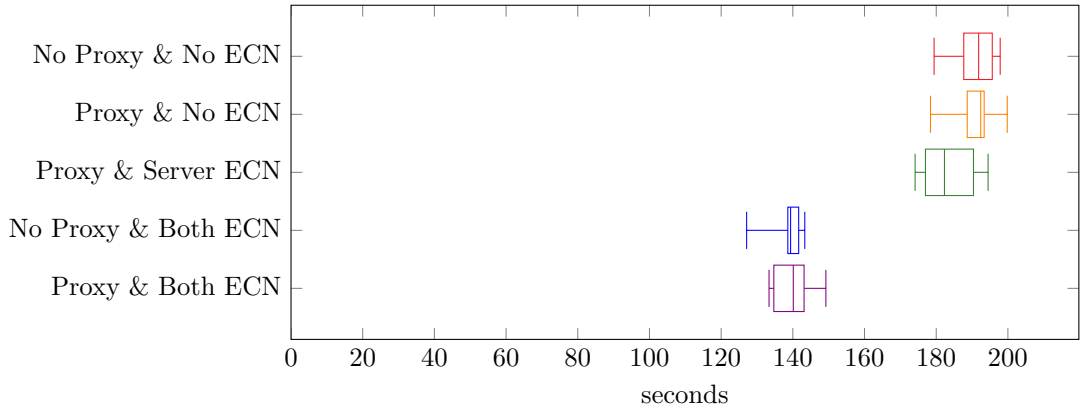


Figure B.33: Throughput analysis with 1 simultaneous connection, a transfer of 20 MB per connection and Cross-Traffic, Client Upload and Server Download

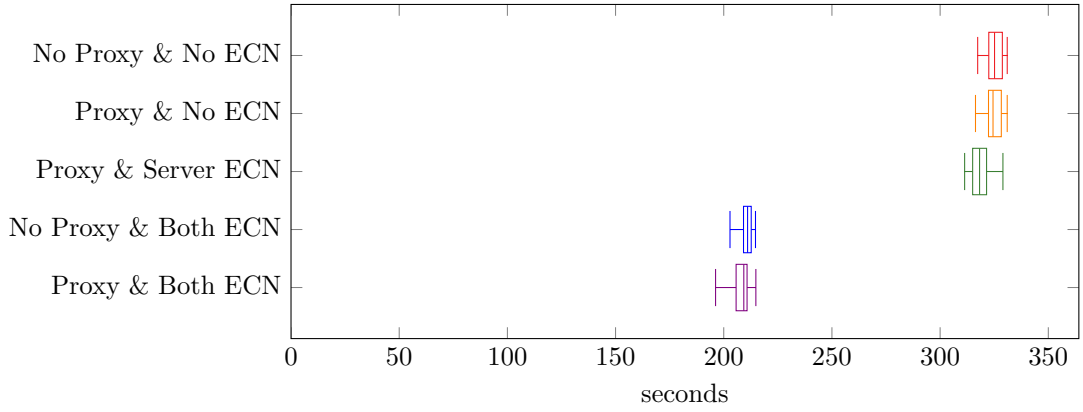


Figure B.34: Throughput analysis with 2 simultaneous connections, a transfer of 20 MB per connection and Cross-Traffic, Client Upload and Server Download

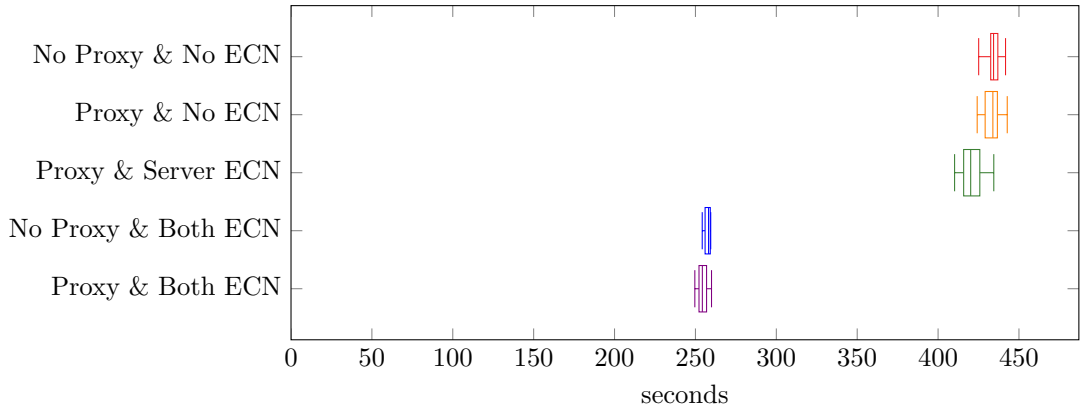


Figure B.35: Throughput analysis with 3 simultaneous connections, a transfer of 20 MB per connection and Cross-Traffic, Client Upload and Server Download. **One Outlier was removed from `noEcn_proxy_cross`**

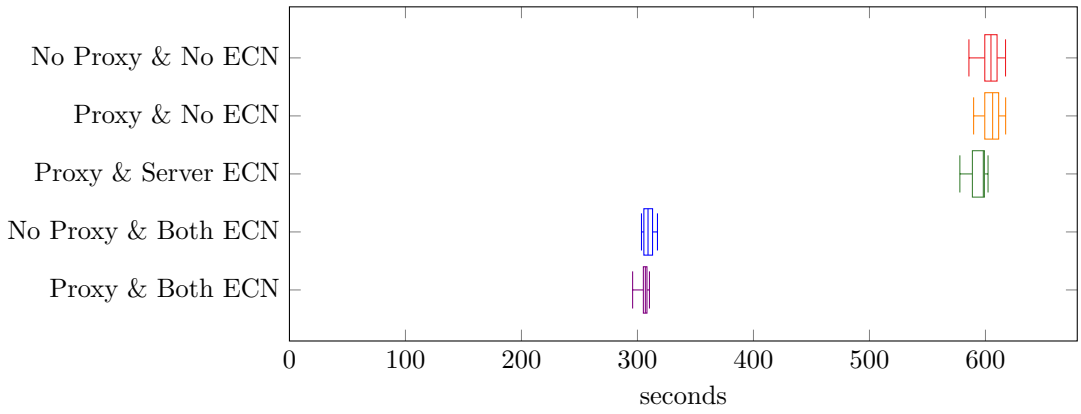


Figure B.36: Throughput analysis with 5 simultaneous connections, a transfer of 20 MB per connection and Cross-Traffic, Client Upload and Server Download

One connection, multiple files

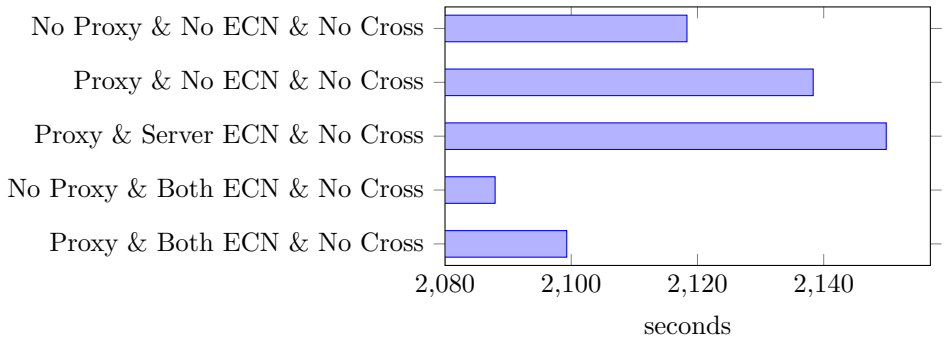


Figure B.37: Throughput analysis with successively transmitting 100 times a file of size 2 MB, no Cross-Traffic, Client Upload and Server Download

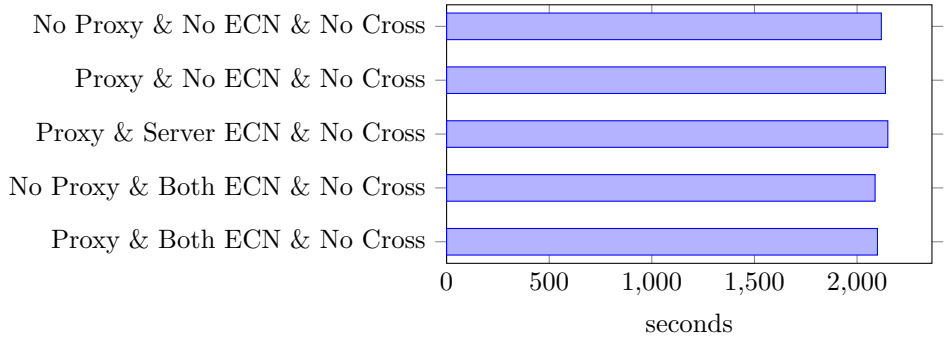


Figure B.38: Throughput analysis with successively transmitting 100 times a file of size 2 MB, no Cross-Traffic, Client Upload and Server Download, with respect to zero

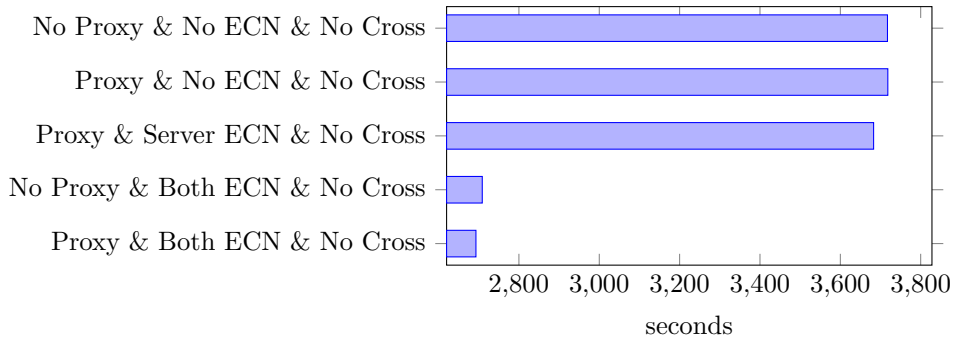


Figure B.39: Throughput analysis with successively transmitting 200 times a file of size 2 MB, with Cross-Traffic, Client Upload and Server Download

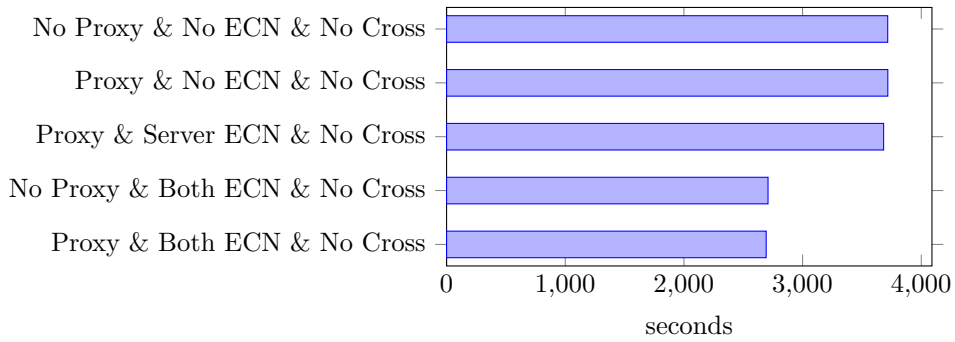


Figure B.40: Throughput analysis with successively transmitting 200 times a file of size 2 MB, with Cross-Traffic, Client Upload and Server Download, with respect to zero

Appendix C

Tables with Throughput Analysis Results

noEcn_noProxy_noCross	02:56.843	02:56.656	02:55.963	03:00.155	02:59.596
	03:00.173	02:57.250	02:59.220	02:56.072	02:58.442
noEcn_proxy_noCross	02:58.991	03:00.270	02:55.974	02:57.806	02:58.974
	02:58.859	02:56.411	02:58.183	02:56.275	02:56.089
oneEcn_proxy_noCross	02:54.517	02:55.564	02:57.385	02:57.607	02:56.771
	02:56.903	02:56.660	02:57.764	02:57.975	02:59.061
bothEcn_noProxy_noCross	02:57.209	02:57.808	02:57.431	02:59.983	02:57.662
	02:56.017	02:59.258	02:59.812	02:56.074	02:58.657
bothEcn_proxy_noCross	02:55.333	02:53.903	02:56.672	02:56.170	02:55.799
	02:55.466	02:55.260	02:56.351	02:56.152	02:56.762
noEcn_noProxy_cross	02:50.323	03:09.362	03:13.940	03:13.357	03:16.262
	03:13.194	03:05.344	03:16.597	03:19.445	03:18.123
noEcn_proxy_cross	02:53.398	03:05.850	03:02.691	03:09.611	03:01.413
	03:10.262	03:08.374	03:04.619	03:15.865	03:07.860
oneEcn_proxy_cross	02:18.833	02:21.982	02:29.423	02:18.407	02:27.675
	02:13.847	02:28.459	02:23.429	02:25.984	02:24.777
bothEcn_noProxy_cross	02:08.412	02:27.278	02:18.079	02:19.171	02:14.795
	02:19.243	02:22.668	02:22.995	02:19.237	02:23.643
bothEcn_proxy_cross	02:17.393	02:15.826	02:15.869	02:29.284	02:21.267
	02:14.351	02:21.990	02:18.724	02:26.997	02:16.036

Table C.1: Time elapsed for downloading (Server uploads, Client downloads) 200MB/20MB (noCross/cross) with 1 simultaneous connection. Displayed in mm:ss

noEcn_noProxy_noCross	05:44.784	05:44.768	05:44.804	05:46.368	05:45.526
	05:45.498	05:45.391	05:45.712	05:45.735	05:46.248
noEcn_proxy_noCross	05:44.846	05:44.876	05:45.460	05:44.033	05:44.278
	05:45.138	05:45.132	05:44.374	05:44.116	05:44.469
oneEcn_proxy_noCross	05:43.784	05:44.665	05:44.436	05:43.769	05:43.865
	05:45.337	05:44.593	05:45.348	05:45.670	05:45.172
bothEcn_noProxy_noCross	05:44.216	05:46.029	05:46.054	05:44.960	05:44.367
	05:45.273	05:44.962	05:43.995	05:44.461	05:44.321
bothEcn_proxy_noCross	05:46.417	05:44.781	05:44.471	05:43.862	05:44.380
	05:44.211	05:43.776	05:44.196	05:44.664	05:43.260
noEcn_noProxy_cross	05:24.618	05:27.328	05:25.877	05:36.843	05:21.553
	05:39.888	05:24.057	05:27.657	05:27.714	05:37.243
noEcn_proxy_cross	05:34.987	05:33.600	05:27.937	05:24.277	05:24.466
	05:21.470	05:27.983	05:20.820	05:29.850	05:34.269
oneEcn_proxy_cross	03:36.939	03:40.581	03:41.247	03:34.933	03:39.952
	03:35.775	03:38.486	03:39.426	03:39.412	03:37.138
bothEcn_noProxy_cross	03:34.404	03:38.041	03:31.846	03:36.366	03:39.520
	03:35.643	03:32.248	03:38.761	03:35.334	03:30.987
bothEcn_proxy_cross	03:31.615	03:32.750	03:32.959	03:31.659	03:35.589
	03:32.149	03:41.553	03:20.350	03:33.470	03:31.809

Table C.2: Time elapsed for downloading (Server uploads, Client downloads) 200MB/20MB (noCross/cross) with 2 simultaneous connections. Displayed in mm:ss

noEcn_noProxy_noCross	08:37.068	08:37.152	08:36.143	08:36.469	08:36.689
	08:37.842	08:36.211	08:37.363	08:37.671	08:36.511
noEcn_proxy_noCross	08:35.093	08:35.522	08:36.214	08:36.092	08:36.395
	08:36.941	08:35.496	08:36.087	08:36.650	08:35.756
oneEcn_proxy_noCross	08:35.157	08:34.783	08:35.098	08:34.798	08:34.488
	08:35.387	08:34.872	08:35.206	08:35.181	08:35.098
bothEcn_noProxy_noCross	08:35.807	08:35.765	08:36.376	08:36.738	08:35.984
	08:35.889	08:36.248	08:36.177	08:35.776	08:35.926
bothEcn_proxy_noCross	08:34.950	08:35.659	08:35.127	08:34.818	08:35.461
	08:35.070	08:35.208	08:34.762	08:35.685	08:35.313
noEcn_noProxy_cross	07:28.676	07:40.386	07:20.042	07:20.760	07:35.823
	07:33.268	07:31.173	07:16.156	07:24.536	07:28.908
noEcn_proxy_cross	07:14.187	07:16.337	07:19.287	07:20.641	07:18.905
	07:28.223	07:17.435	07:24.003	07:23.572	07:21.853
oneEcn_proxy_cross	04:28.929	04:33.420	04:33.748	04:26.959	04:29.970
	04:26.078	04:31.505	04:34.957	04:31.215	04:30.576
bothEcn_noProxy_cross	04:28.142	04:26.071	04:20.147	04:28.281	04:28.597
	04:23.965	04:29.999	04:15.189	04:22.961	04:23.933
bothEcn_proxy_cross	04:23.547	04:21.810	04:26.384	04:20.495	04:18.213
	04:22.192	04:25.488	04:21.272	04:21.955	04:24.062

Table C.3: Time elapsed for downloading (Server uploads, Client downloads) 200MB/20MB (noCross/cross) with 3 simultaneous connections. Displayed in mm:ss

noEcn_noProxy_noCross	14:17.405	14:17.309	14:17.935	14:17.272	14:17.798
	14:17.352	14:16.957	14:17.131	14:17.029	14:17.223
noEcn_proxy_noCross	14:16.244	14:15.823	14:15.908	14:16.344	14:16.028
	14:16.101	14:16.230	14:15.731	14:16.499	14:16.052
oneEcn_proxy_noCross	14:15.695	14:15.810	14:15.254	14:15.850	14:16.614
	14:16.194	14:15.685	14:16.096	14:16.001	14:16.103
bothEcn_noProxy_noCross	14:17.232	14:17.473	14:17.248	14:17.318	14:17.807
	14:18.113	14:17.841	14:16.916	14:17.530	14:18.105
bothEcn_proxy_noCross	14:16.896	14:15.839	14:15.985	14:15.963	14:16.104
	14:15.828	14:15.880	14:15.842	14:16.849	14:15.807
noEcn_noProxy_cross	10:27.384	10:36.345	10:33.878	10:37.217	10:31.736
	10:36.622	10:36.931	10:35.411	10:47.586	10:47.562
noEcn_proxy_cross	10:26.509	10:28.145	10:19.630	10:11.460	10:25.357
	10:24.416	10:19.809	10:10.990	10:27.455	10:36.382
oneEcn_proxy_cross	05:30.734	05:29.289	05:27.042	05:37.557	05:21.537
	05:31.001	05:34.045	05:31.298	05:30.653	05:25.622
bothEcn_noProxy_cross	05:15.118	05:11.144	05:16.299	05:11.369	05:13.020
	05:15.498	05:08.564	05:17.076	05:15.000	05:13.000
bothEcn_proxy_cross	05:10.463	05:07.898	05:17.419	05:21.260	05:13.223
	05:19.915	05:12.900	05:12.025	05:10.620	05:10.710

Table C.4: Time elapsed for downloading (Server uploads, Client downloads) 200MB/20MB (noCross/cross) with 5 simultaneous connections. Displayed in mm:ss

noEcn_noProxy_noCross	0:05:53.460
noEcn_proxy_noCross	0:05:52.618
oneEcn_proxy_noCross	0:05:52.561
bothEcn_noProxy_noCross	0:05:50.964
bothEcn_proxy_noCross	0:05:51.465
noEcn_noProxy_cross	1:03:04.055
noEcn_proxy_cross	1:02:27.322
oneEcn_proxy_cross	0:45:38.636
bothEcn_noProxy_cross	0:45:46.146
bothEcn_proxy_cross	0:45:09.515

Table C.5: Time elapsed for downloading (Server uploads, Client downloads) 100 times a 2 MB file with 1 connection. Displayed in hh:mm:ss

noEcn_noProxy_noCross	09:31.831 05:02.082	23:39.844 07:47.708	06:04.260 07:34.967	10:17.139
noEcn_proxy_noCross	07:54.188 07:52.998	10:14.508 04:50.495	08:13.397 07:51.211	09:59.570
oneEcn_proxy_noCross	08:12.842 08:08.378	09:04.133 04:43.978	08:37.096 08:12.203	09:20.210
bothEcn_noProxy_noCross	09:47.775 09:39.683	08:55.644 07:12.513	07:46.733 03:00.878	07:04.504
bothEcn_proxy_noCross	07:50.469 07:52.137	09:16.709 04:30.609	07:49.067 08:28.793	09:34.694
noEcn_noProxy_cross	02:59.408 03:15.062	03:12.210 03:11.458	03:17.845 03:06.688	03:08.022 03:17.367
noEcn_proxy_cross	03:01.234 03:19.812	03:13.286 03:11.124	03:13.645 03:12.073	02:58.402 03:12.763
oneEcn_proxy_cross	02:59.223 02:56.646	02:57.113 03:14.473	03:05.377 03:09.835	02:54.113 03:12.078
bothEcn_noProxy_cross	02:07.109 02:17.901	02:22.027 02:19.036	02:18.837 02:19.715	02:23.334 02:21.499
bothEcn_proxy_cross	02:29.219 02:25.057	02:14.990 02:13.818	02:20.336 02:22.492	02:13.362 02:19.910

Table C.6: Time elapsed for uploading (Client uploads, Server downloads) 100MB/20MB (noCross/cross) with 1 simultaneous connection. Displayed in mm:ss. **Outlier marked red**

noEcn_noProxy_noCross	13:16.301 07:31.349	09:13.234 09:48.642	15:06.181 12:17.003	13:20.616
noEcn_proxy_noCross	12:16.762 12:20.697	15:53.688 07:16.088	11:58.404 09:34.279	16:27.920
oneEcn_proxy_noCross	11:25.404 11:05.741	15:54.862 07:38.170	11:04.999 09:06.890	15:14.289
bothEcn_noProxy_noCross	12:52.354 12:26.072	10:57.969 06:09.810	10:44.017 06:41.809	13:16.912
bothEcn_proxy_noCross	12:08.990 11:37.449	16:07.808 07:05.571	11:27.217 08:52.651	16:27.863
noEcn_noProxy_cross	05:25.826 05:19.826	05:28.671 05:17.359	05:29.034 05:31.017	05:23.379 05:24.484
noEcn_proxy_cross	05:22.843 05:27.375	05:31.007 05:16.349	05:25.400 05:20.923	05:31.011 05:23.501
oneEcn_proxy_cross	05:20.423 05:19.567	05:15.256 05:29.028	05:24.612 05:14.368	05:11.329 05:16.906
bothEcn_noProxy_cross	03:22.888 03:29.188	03:32.594 03:34.669	03:33.002 03:28.983	03:31.663 03:30.372
bothEcn_proxy_cross	03:25.205 03:31.057	03:34.841 03:30.680	03:29.696 03:16.221	03:28.839 03:25.879

Table C.7: Time elapsed for uploading (Client uploads, Server downloads) 100MB/20MB (noCross/cross) with 2 simultaneous connections. Displayed in mm:ss

noEcn_noProxy_noCross	14:54.200 10:52.873	11:14.115 11:19.162	13:20.247 07:27.100	30:01.177
noEcn_proxy_noCross	07:01.590 07:51.101	10:59.076 10:45.140	07:52.010 10:55.378	10:23.864
oneEcn_proxy_noCross	07:07.667 09:42.731	11:19.155 10:33.782	08:52.646 11:08.327	11:28.750
bothEcn_noProxy_noCross	10:12.252 11:10.516	09:18.353 09:25.929	08:04.799 06:01.171	08:14.748
bothEcn_proxy_noCross	07:21.600 08:16.027	10:32.576 10:16.129	08:43.014 11:08.768	10:43.443
noEcn_noProxy_cross	07:16.541 07:18.139	07:21.703 07:05.093	07:12.770 07:11.832	07:14.727 07:13.883
noEcn_proxy_cross	07:14.927 07:18.381	07:22.754 07:04.171	07:12.009 07:06.185	07:13.871 25:01.000
oneEcn_proxy_cross	07:14.442 06:50.207	06:56.930 06:56.012	07:03.384 07:03.735	06:55.233 07:11.997
bothEcn_noProxy_cross	04:14.198 04:19.457	04:19.259 04:15.096	04:16.264 04:19.155	04:18.943 04:17.288
bothEcn_proxy_cross	04:15.821 04:12.730	04:19.949 04:17.794	04:11.269 04:12.520	04:09.630 04:16.585

Table C.8: Time elapsed for uploading (Client uploads, Server downloads) 100MB/20MB (noCross/cross) with 3 simultaneous connections. Displayed in mm:ss. **Outlier marked red**

noEcn_noProxy_noCross	24:22.151 13:40.396	14:30.646 15:42.573	17:13.259 11:55.259	24:10.995
noEcn_proxy_noCross	12:52.593 12:33.530	16:01.602 14:32.512	12:34.134 16:31.722	15:52.986
oneEcn_proxy_noCross	12:40.697 12:36.378	15:48.525 14:26.129	12:32.588 16:19.571	15:56.863
bothEcn_noProxy_noCross	14:01.064 13:40.410	11:40.645 12:19.105	11:28.204 11:12.978	14:36.002
bothEcn_proxy_noCross	12:47.076 12:36.568	16:25.987 15:09.832	12:36.276 17:03.830	16:25.062
noEcn_noProxy_cross	09:58.285 10:17.301	10:06.876 09:59.790	09:45.744 10:11.157	10:02.614 10:09.624
noEcn_proxy_cross	10:09.823 10:17.395	10:16.023 10:09.104	10:03.273 09:59.828	09:49.788 09:58.113
oneEcn_proxy_cross	09:57.890 09:58.800	10:02.228 09:49.837	09:45.457 09:37.937	09:59.883 09:58.419
bothEcn_noProxy_cross	05:17.258 05:03.481	05:13.305 05:05.881	05:05.467 05:05.552	05:12.342 05:12.939
bothEcn_proxy_cross	05:06.689 05:05.106	05:09.070 05:10.364	05:05.111 05:07.925	04:55.842 05:06.943

Table C.9: Time elapsed for uploading (Client uploads, Server downloads) 100MB/20MB (noCross/cross) with 5 simultaneous connections. Displayed in mm:ss

noEcn_noProxy_noCross	0:35:18.333
noEcn_proxy_noCross	0:35:38.335
oneEcn_proxy_noCross	0:35:49.907
bothEcn_noProxy_noCross	0:34:47.940
bothEcn_proxy_noCross	0:34:59.284
noEcn_noProxy_cross	1:01:57.318
noEcn_proxy_cross	1:01:58.204
oneEcn_proxy_cross	1:01:22.811
bothEcn_noProxy_cross	0:45:08.781
bothEcn_proxy_cross	0:44:53.042

Table C.10: Time elapsed for downloading (Client uploads, Server downloads) 100/200 (100 times without cross traffic and 200 times with cross traffic) times a 2 MB file with 1 connection. Displayed in hh:mm:ss

Appendix D

Sourcecode

Sourecode of the Netfilter-Hook and the changed Header-File

Listing D.1: Sourcecode of the Netfilter Hook

```
//Netfilter hook for an ECN handling proxy
#include <linux/ip.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/netdevice.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>
#include <linux/skbuff.h>
#include <linux/tcp.h>
#include <net/ip.h>
#include <net/checksum.h>
#include <net/netfilter/nf_conntrack.h>
#include <net/tcp.h>



static struct nf_hook_ops nfho;          //struct holding set of hook
    function options
static char *interface_client = "eth1"; //strings with interface
    names
static char *interface_server = "eth2";
enum ecn_states{INIT, AC, ANC, AC_WAIT, ANC_WAIT, ACBC_OR_ANCBNC,
    ACBNC, ACBNC_DUPACK, ACBNC_ECEDUPACK, ACBNC_ECE, ANCBC,
    ANCBC_DUPACK, ANCBC_ECEDUPACK, ANCBC_ECE};

void set_ect(struct iphdr* iph){
  unsigned char tosbits;
  tosbits = iph->tos;
  tosbits = tosbits | 2; //change TOS to ECT(0)
  iph->tos = tosbits;
  iph->check = 0;
  ip_send_check(iph);
}

void clear_ect(struct iphdr* iph){
  unsigned char tosbits;
  tosbits = iph->tos;
  tosbits = tosbits & ~2; //reset ECT(0)
```

```
  iph->tos = tosbits;
  iph->check = 0;
  ip_send_check(iph);
}

void clear_ce(struct iphdr* iph){
  unsigned char tosbits;
  tosbits = iph->tos;
  tosbits = tosbits & ~3; //reset CE
  iph->tos = tosbits;
  iph->check = 0;
  ip_send_check(iph);
}

void set_ece(struct tcphdr *tcph, struct iphdr *iph, struct sk_buff *
    skb){
  int tcplen;
  if(tcph->ece !=1){
    tcph->ece = 1;
    tcplen = skb->len - ip_hdrlen(skb);
    tcph->check = 0;
    tcph->check = tcp_v4_check(tcplen, iph->saddr, iph->daddr,
       csum_partial((char *)tcph, tcplen,0));
  }
}

void set_cwr(struct tcphdr *tcph, struct iphdr *iph, struct sk_buff *
    skb){
  int tcplen;
  if(tcph->cwr != 1){
    tcph->cwr = 1;
    tcplen = skb->len - ip_hdrlen(skb);
    tcph->check = 0;
    tcph->check = tcp_v4_check(tcplen, iph->saddr, iph->daddr,
       csum_partial((char *)tcph, tcplen,0));
  }
}

void clear_ece(struct tcphdr *tcph, struct iphdr *iph, struct sk_buff
     *skb){
  int tcplen;
  if(tcph->ece !=0){
    tcph->ece = 0;
    tcplen = skb->len - ip_hdrlen(skb);
    tcph->check = 0;
    tcph->check = tcp_v4_check(tcplen, iph->saddr, iph->daddr,
       csum_partial((char *)tcph, tcplen,0));
  }
}

void clear_cwr(struct tcphdr *tcph, struct iphdr *iph, struct sk_buff
     *skb){
  int tcplen;
  if(tcph->cwr != 0){
    tcph->cwr = 0;
    tcplen = skb->len - ip_hdrlen(skb);
    tcph->check = 0;
    tcph->check = tcp_v4_check(tcplen, iph->saddr, iph->daddr,
```

```
        csum_partial((char *)tcph, tcplen,0));
  }
}

void set_ack_seq(struct tcphdr *tcph, struct iphdr *iph, struct
    sk_buff *skb, u_int32_t ack_seq){
  int tcplen;
  tcph->ack_seq = ack_seq;

  tcplen = skb->len - ip_hdrlen(skb);
  tcph->check = 0;
  tcph->check = tcp_v4_check(tcplen, iph->saddr, iph->daddr,
      csum_partial((char *)tcph, tcplen,0));
}


//function called by the hook
unsigned int hook_func(unsigned int hooknum, struct sk_buff *skb,
    const struct net_device *in, const struct net_device *out, int (*
    okfn)(struct sk_buff *))
{
  struct iphdr *iph;
  struct tcphdr *tcph;
  struct nf_conn *ct;
  enum ip_conntrack_info ctinfo;
  u_int8_t new_state, new_number_ack;
  u_int32_t new_saved_ack;
  u_int8_t hostA;
  int dirA, dirB;

  if(!skb) {return NF_ACCEPT;}  //stop if skb = NULL

  iph = (struct iphdr *)skb_network_header(skb);        //IP-Header

  if (!iph) {return NF_ACCEPT;} // stop if iph = NULL

  /* IF TCP packet enter cases, else Accept*/
  if (iph->protocol == IPPROTO_TCP){
    tcph = tcp_hdr(skb); //TCP-Header

    nf_conntrack_get(skb->nfct);
    ct = nf_ct_get(skb, &ctinfo);
    if(!ct){return NF_ACCEPT;}  // stop if ct == NULL

    new_number_ack = ct->proto.tcp.number_ack;
    new_saved_ack = ct->proto.tcp.saved_ack;
    hostA = ((ct->proto.tcp.ecn_state) >>7);
    new_state = ct->proto.tcp.ecn_state & ~(1<<7); //default
       new_state = old_state

    /*If new connection set ecn_state to INIT*/
    if(ctinfo==2){
      new_state = INIT;
      if(strcmp(skb->dev->name,interface_client) == 0){ //if client
         interface is initiating host
        hostA = 0;
      }else{
        hostA = 1;
```

```
  }

}

/*From A (dirA) or from B (dirB)*/
dirA = ((strcmp(skb->dev->name,interface_client) == 0) && (hostA
    == 0)) || ((strcmp(skb->dev->name,interface_server)==0) && (
    hostA != 0));
dirB = ((strcmp(skb->dev->name,interface_server) == 0) && (hostA
    == 0)) || ((strcmp(skb->dev->name,interface_client)==0) && (
    hostA != 0));

switch(new_state){

case INIT:
  /*Is Host A input device?*/
  if(dirA){
    if(tcph->syn == 1){
      //ECN-setup ACK
      if(tcph->ece == 1 && tcph->cwr == 1){
        new_state = AC;
      }else{
        //set ECE and CWR
        set_ece(tcph, iph, skb);
        set_cwr(tcph, iph, skb);
        new_state = ANC;
      }
    }
  }
  break;

case AC:
  /*Is Host B input device?*/
  if(dirB){
    if(tcph->syn == 1 && tcph->ack == 1){
      //ECN-setup SYN/ACK
      if(tcph->ece == 1 && tcph->cwr == 0){
        //both capable
        new_state = ACBC_OR_ANCBNC;
      }else{
        //A capable, B not
        //change tcph->ece = 1 && tcph->cwr = 0;
        set_ece(tcph,iph,skb);
        clear_cwr(tcph,iph,skb);
        new_state = ACBNC;
      }
    }
  }
  break;

case ANC:
  /*Is Host B input device?*/
  if(dirB){
    if(tcph->syn == 1 && tcph->ack == 1){
      //ECN-setup SYN/ACK
      if(tcph->ece == 1 && tcph->cwr == 0){
        //A not capable, B capable
        clear_ece(tcph, iph, skb);
```

```
            clear_cwr(tcph, iph, skb);
            new_state = ANCBC;
          }else{
            //both not capable
            new_state = ACBC_OR_ANCBNC;
          }
        }
      }
      break;

    case AC_WAIT:
      if(dirA){
        if(tcph->ack == 1){
          new_state = ACBNC;
        }
      }
      break;

    case ANC_WAIT:
      //wait for the ack
      if(dirA){
        if(tcph->ack == 1){
          new_state = ANCBC;
        }
      }
      break;

    case ACBC_OR_ANCBNC:
      //don't change any headers/flags
      break;

    case ACBNC:
      if(dirA){
        if((iph->tos & 3) == 3){
          //CE codepoint
          if(tcph->ece == 1){
            //CE and ECE
            //save ACK
            new_saved_ack = tcph->ack;
            new_number_ack = 1;
            new_state = ACBNC_ECEDUPACK;
          }else{
            //CE and no ECE
            //do something
            new_state = ACBNC_ECE;
          }
        }else{
          if(tcph->ece == 1){
            //no CE but ECE
            //save ACK
            new_saved_ack = tcph->ack;
            new_number_ack = 1;
            new_state = ACBNC_DUPACK;
          }else{
            //no CE and no ECE
            //new_state = old_state
          }
        }
```

```
      //clear ECT, CE, ECE for every packet to B
      clear_ce(iph);  //clears CE and ECT(0)
      clear_ece(tcph, iph, skb);
    }
    if(dirB){
      //set ECT(0), problematic if the proxy set earlier the CE
         codepoint
      set_ect(iph);
    }
    break;

  case ACBNC_DUPACK:
    if(dirA){
      set_ack_seq(tcph,iph,skb,new_saved_ack);
      new_number_ack = new_number_ack +1;
      if((iph->tos & 3) == 3){
        if(new_number_ack > 3){
          //not finished with duplicating but CE occured
          new_state = ACBNC_ECE;
        }else{
          //finished duplicating but CE occured
          new_state = ACBNC_ECEDUPACK;
        }
      }else{
        if(new_number_ack > 3){
          //finished duplicating -> back to ACBNC
          new_state = ACBNC;
        }
      }
      //clear ECT, CE, ECE for every packet to B
      clear_ce(iph);  //clears CE and ECT(0)
      clear_ece(tcph, iph, skb);
    }
    if(dirB){
      //set ECT(0), problematic if the proxy set earlier the CE
         codepoint
      set_ect(iph);
    }
    break;

  case ACBNC_ECEDUPACK:
    if(dirA){
      set_ack_seq(tcph,iph,skb,new_saved_ack);
      new_number_ack = new_number_ack +1;
      if(tcph->cwr){
        if(new_number_ack > 3){
          //not finished with duplicating but CE occured
          new_state = ACBNC;
        }else{
          //finished duplicating but CE occured
          new_state = ACBNC_DUPACK;
        }
      }else{
        if(new_number_ack > 3){
          //finished duplicating -> back to ACBNC
          new_state = ACBNC;
        }
      }
```

```
      //clear ECT, CE, ECE for every packet to B
      clear_ce(iph);  //clears CE and ECT(0)
      clear_ece(tcph, iph, skb);
    }
    if(dirB){
      //set ECT(0), problematic if the proxy set earlier the CE
         codepoint
      set_ect(iph);
      //set the ece flag, because there was a CE
      set_ece(tcph,iph,skb);
    }
    break;

  case ACBNC_ECE:
    //printk(KERN_INFO "Ce received");
    if(dirA){
      if(tcph->cwr == 1){
        if(tcph->ece == 1){
          //new state and save ACK
          new_state = ACBNC_DUPACK;
          new_saved_ack = tcph->ack_seq;
          new_number_ack = 1;
        }else{
          new_state = ACBNC;
        }
      }else{
        if(tcph->ece == 1){
          new_state = ACBNC_ECEDUPACK;
          new_saved_ack = tcph->ack_seq;
          new_number_ack = 1;
        }
      }
      //clear ECT, CE, ECE for every packet to B
      clear_ce(iph);  //clears CE and ECT(0)
      clear_ece(tcph, iph, skb);
    }
    if(dirB){
      //set ECT(0), problematic if the proxy set earlier the CE
         codepoint
      set_ect(iph);
      //set the ece flag, because there was a CE
      set_ece(tcph,iph,skb);
    }
    break;

  case ANCBC:
    //same code as in ACBNC but dirA and dirB interchanged and
       ACBNC -> ANCBC, left out and added when testing complete
    if(dirB){
      if((iph->tos & 3) == 3){
        //CE codepoint
        if(tcph->ece == 1){
          //CE and ECE
          //save ACK
          new_saved_ack = tcph->ack;
          new_number_ack = 1;
          new_state = ANCBC_ECEDUPACK;
        }else{
```

```
            //CE and no ECE
            //do something
            new_state = ANCBC_ECE;
          }
        }else{
          int test = (int)tcph->ece;
          printk(KERN_INFO "test = %i\n",test);
          if(tcph->ece == 1){
            //no CE but ECE
            //save ACK
            new_saved_ack = tcph->ack;
            new_number_ack = 1;
            new_state = ANCBC_DUPACK;
          }else{
            //no CE and no ECE
            //new_state = old_state
          }
        }
        //clear ECT, CE, ECE for every packet to B
        clear_ce(iph);  //clears CE and ECT(0)
        clear_ece(tcph, iph, skb);
      }
      if(dirA){
        //set ECT(0), problematic if the proxy set earlier the CE
            codepoint
        set_ect(iph);
      }
      break;

    case ANCBC_DUPACK:
      if(dirB){
        set_ack_seq(tcph,iph,skb,new_saved_ack);
        new_number_ack = new_number_ack +1;
        if((iph->tos & 3) == 3){
          if(new_number_ack > 3){
            //not finished with duplicating but CE occured
            new_state = ANCBC_ECE;
          }else{
            //finished duplicating but CE occured
            new_state = ANCBC_ECEDUPACK;
          }
        }else{
          if(new_number_ack > 3){
            //finished duplicating -> back to ACBNC
            new_state = ANCBC;
          }
        }
        //clear ECT, CE, ECE for every packet to B
        clear_ce(iph);  //clears CE and ECT(0)
        clear_ece(tcph, iph, skb);
      }
      if(dirA){
        //set ECT(0), problematic if the proxy set earlier the CE
            codepoint
        set_ect(iph);
      }
      break;
```

```
case ANCBC_ECEDUPACK:
  if(dirB){
    set_ack_seq(tcph,iph,skb,new_saved_ack);
    new_number_ack = new_number_ack +1;
    if(tcph->cwr){
      if(new_number_ack > 3){
        //not finished with duplicating but CE occured
        new_state = ANCBC;
      }else{
        //finished duplicating but CE occured
        new_state = ANCBC_DUPACK;
      }
    }else{
      if(new_number_ack > 3){
        //finished duplicating -> back to ACBNC
        new_state = ANCBC;
      }
    }
    //clear ECT, CE, ECE for every packet to B
    clear_ce(iph);  //clears CE and ECT(0)
    clear_ece(tcph, iph, skb);
  }
  if(dirA){
    //set ECT(0), problematic if the proxy set earlier the CE
       codepoint
    set_ect(iph);
    //set the ece flag, because there was a CE
    set_ece(tcph,iph,skb);
  }
  break;

case ANCBC_ECE:
  if(dirB){
    if(tcph->cwr == 1){
      if(tcph->ece == 1){
        //new state and save ACK
        new_state = ANCBC_DUPACK;
        new_saved_ack = tcph->ack_seq;
        new_number_ack = 1;
      }else{
        new_state = ANCBC;
      }
    }else{
      if(tcph->ece == 1){
        new_state = ANCBC_ECEDUPACK;
        new_saved_ack = tcph->ack_seq;
        new_number_ack = 1;
      }
    }
    //clear ECT, CE, ECE for every packet to B
    clear_ce(iph);  //clears CE and ECT(0)
    clear_ece(tcph, iph, skb);
  }
  if(dirA){
    //set ECT(0), problematic if the proxy set earlier the CE
       codepoint
    set_ect(iph);
    //set the ece flag, because there was a CE
```

```
        set_ece(tcph,iph,skb);
      }
      break;

    default:
      printk(KERN_INFO "Invalid state");
    }

    //write back new values
    ct->proto.tcp.ecn_state = new_state | (hostA << 7);
    ct->proto.tcp.saved_ack = new_saved_ack;
    ct->proto.tcp.number_ack= new_number_ack;
    printk(KERN_INFO "state = %i, hostA = %i\n",(int)ct->proto.tcp.
        ecn_state, (int)hostA); //print the state and the initiating
        host for debugging
  }

  return NF_ACCEPT;
}

//Called when module loaded using 'insmod'
int init_module()
{
  nfho.hook = (nf_hookfn *) hook_func;          //function to call
     when conditions below met
  nfho.hooknum = NF_INET_FORWARD;               //called for every
     packet that gets forwarded
  nfho.pf = PF_INET;                            //IPV4 packets
  nfho.priority = NF_IP_PRI_FIRST;              //set to highest
     priority over all other hook functions
  nf_register_hook(&nfho);                      //register hook

  return 0;                                     //return 0 for success
}

//Called when module unloaded using 'rmmod'
void cleanup_module()
{
  nf_unregister_hook(&nfho);
}
```

Listing D.2: Sourcecode of the changed Header-File

```
#ifndef _NF_CONNTRACK_TCP_H
#define _NF_CONNTRACK_TCP_H

#include <uapi/linux/netfilter/nf_conntrack_tcp.h>


struct ip_ct_tcp_state {
        u_int32_t       td_end;         /* max of seq + len */
        u_int32_t       td_maxend;      /* max of ack + max(win, 1)
            */
        u_int32_t       td_maxwin;      /* max(win) */
        u_int32_t       td_maxack;      /* max of ack */
        u_int8_t        td_scale;       /* window scale factor */
        u_int8_t        flags;          /* per direction options */
};

struct ip_ct_tcp {
        struct ip_ct_tcp_state seen[2]; /* connection parameters per
            direction */
        u_int8_t        state;          /* state of the connection (
            enum tcp_conntrack) */
        /* For detecting stale connections */
        u_int8_t        last_dir;       /* Direction of the last
            packet (enum ip_conntrack_dir) */
        u_int8_t        retrans;        /* Number of retransmitted
            packets */
        u_int8_t        last_index;     /* Index of the last packet
            */
        u_int32_t       last_seq;       /* Last sequence number seen
            in dir */
        u_int32_t       last_ack;       /* Last sequence number seen
            in opposite dir */
        u_int32_t       last_end;       /* Last seq + len */
        u_int16_t       last_win;       /* Last window advertisement
            seen in dir */
        /* For SYN packets while we may be out-of-sync */
        u_int8_t        last_wscale;    /* Last window scaling factor
             seen */
        u_int8_t        last_flags;     /* Last flags set */

//These are the three inserted variables for the proxy to function
   properly
        u_int8_t ecn_state; /*state of the ecn proxy*/
        u_int32_t saved_ack; /*saved ack for duplicated acks*/
        u_int8_t number_ack; /*number of times ack has been repeated
            */
};

#endif /* _NF_CONNTRACK_TCP_H */
```

bash Scripts used for Testing

Script for the Proxy

```
while true
do
        a=$(nc 192.168.0.10 2345) # address of the client
        case $a in
        noProxy)
                echo "noProxy"
                sudo rmmod ecn_proxy
                ;;
        proxy)
                echo "proxy"
                sudo insmod ecn_proxy.ko
                ;;
        cross)
                echo "cross"
                echo "cross" > index.html
                nc -l 2345 < index.html
                ;;
        *)
                echo "sleep"
                ;;
        esac
        sleep 10
done
```

Scripts for the Uploading Host

Listing D.3: Main Script that calls the corresponding Scripts and sends the Signals to the other Hosts

```
#noCross
echo "noCross" > index.html
nc -l 2345 < index.html

cd noEcn_noProxy_noCross
#First Proxy, the Ecn
echo "noProxy" > index.html
nc -l 2345 < index.html
echo "noEcn" > index.html
nc -l 2345 < index.html
sudo sysctl -w net.ipv4.tcp_ecn=0
sleep 10
bash ../upload_testing_script.sh
cd ..

cd noEcn_proxy_noCross
echo "proxy" > index.html
nc -l 2345 < index.html
echo "noEcn" > index.html
nc -l 2345 < index.html
sudo sysctl -w net.ipv4.tcp_ecn=0
sleep 10
bash ../upload_testing_script.sh
cd ..
```

```
cd oneEcn_proxy_noCross
echo "proxy" > index.html
nc -l 2345 < index.html
echo "oneEcn" > index.html
nc -l 2345 < index.html
sudo sysctl -w net.ipv4.tcp_ecn=1
sleep 10
bash ../upload_testing_script.sh
cd ..

cd bothEcn_proxy_noCross
echo "proxy" > index.html
nc -l 2345 < index.html
echo "bothEcn" > index.html
nc -l 2345 < index.html
sudo sysctl -w net.ipv4.tcp_ecn=1
sleep 10
bash ../upload_testing_script.sh
cd ..

cd bothEcn_noProxy_noCross
echo "noProxy" > index.html
nc -l 2345 < index.html
echo "bothEcn" > index.html
nc -l 2345 < index.html
sudo sysctl -w net.ipv4.tcp_ecn=1
sleep 10
bash ../upload_testing_script.sh
cd ..

#cross
echo "cross" > index.html
nc -l 2345 < index.html

cd noEcn_noProxy_cross
echo "noProxy" > index.html
nc -l 2345 < index.html
echo "noEcn" > index.html
nc -l 2345 < index.html
sudo sysctl -w net.ipv4.tcp_ecn=0
sleep 10
bash ../upload_new_testing_script.sh
cd ..

cd noEcn_proxy_cross
echo "proxy" > index.html
nc -l 2345 < index.html
echo "noEcn" > index.html
nc -l 2345 < index.html
sudo sysctl -w net.ipv4.tcp_ecn=0
sleep 10
bash ../upload_new_testing_script.sh
cd ..

cd oneEcn_proxy_cross
echo "proxy" > index.html
nc -l 2345 < index.html
echo "oneEcn" > index.html
```

```
nc -l 2345 < index.html
sudo sysctl -w net.ipv4.tcp_ecn=1
sleep 10
bash ../upload_new_testing_script.sh
cd ..

cd bothEcn_proxy_cross
echo "proxy" > index.html
nc -l 2345 < index.html
echo "bothEcn" > index.html
nc -l 2345 < index.html
sudo sysctl -w net.ipv4.tcp_ecn=1
sleep 10
bash ../upload_new_testing_script.sh
cd ..

cd bothEcn_noProxy_cross
echo "noProxy" > index.html
nc -l 2345 < index.html
echo "bothEcn" > index.html
nc -l 2345 < index.html
sudo sysctl -w net.ipv4.tcp_ecn=1
sleep 10
bash ../upload_new_testing_script.sh
cd ..
```

Listing D.4: Script for testing the Connection without Cross-Traffic

```
# Script for automatic testing

# Create or overwrite the files for testing
echo "1 connection, big file" > 1_conn_big.txt
echo "2 connection, big file" > 2_conn_big.txt
echo "3 connection, big file" > 3_conn_big.txt
echo "5 connection, big file" > 5_conn_big.txt
echo "small files"> small.txt

for i in {1..5}
do
        for VARIABLE in 1 2 3 5
        do
                (time iperf -c 192.168.2.10 -P $VARIABLE -n 200m -i
                   10) 2>> ${VARIABLE}_conn_big.txt
        done
done

(time (for i in  {1..200}; do iperf -c 192.168.2.10 -n 2m ;done)) 2>>
    small.txt
```

Listing D.5: Script for testing the Connection with Cross-Traffic

```
# Script for automatic testing

# Create or overwrite the files for testing
echo "1 connection, big file" > 1_conn_big.txt
echo "2 connection, big file" > 2_conn_big.txt
echo "3 connection, big file" > 3_conn_big.txt
echo "5 connection, big file" > 5_conn_big.txt
echo "small files"> small.txt
```

```
for i in {1..5}
do
        for VARIABLE in 1 2 3 5
        do
                (time iperf -c 192.168.2.10 -P $VARIABLE -n 20m -i
                   10) 2>> ${VARIABLE}_conn_big.txt
        done
done

(time (for i in  {1..200}; do iperf -c 192.168.2.10 -n 2m ;done)) 2>>
    small.txt
```

Script for the Downloading Host

```
while true
do
        a=$(nc 192.168.0.10 2345)       # address of the uploading
           host
        case $a in
        noEcn)
                echo "noEcn"
                sudo sysctl -w net.ipv4.tcp_ecn=0
                ;;
        oneEcn)
                echo "oneEcn"
                sudo sysctl -w net.ipv4.tcp_ecn=1
                ;;
        bothEcn)
                echo "bothEcn"
                sudo sysctl -w net.ipv4.tcp_ecn=1
                ;;
        noProxy)
                echo "noProxy"
                echo "noProxy" > index.html
                nc -l 2345 < index.html
                ;;
        proxy)
                echo "proxy"
                echo "proxy" > index.html
                nc -l 2345 < index.html
                ;;
        cross)
                echo "cross"
                echo "cross" > index.html
                nc -l 2345 < index.html
                ;;
        *)
                echo "sleep"
                ;;
        esac
        sleep 10
done
```

Script for the UDP Hosts generating Cross-Traffic

UDP Host on Client Side

```
while true
do
        a=$(nc 192.168.1.1 2345)        # address of the ECN proxy
```

```
        sleep 1
        case $a in
        cross)
                echo "cross"
                echo "cross" > index.html
                nc -l 2345 < index.html
                sleep 1
                iperf -c 192.168.1.20 -u -b 10M  -t 100000000 -i 20
                ;;
        *)
                echo "sleep"
                ;;
        esac
        sleep 10
done
```

UDP Host on Server Side

```
while true
do
        a=$(nc 192.168.1.20 2345)       # address of the UDPA host
        case $a in
        cross)
                echo "cross"
                iperf -c 192.168.1.20 -u -b 10M -t 100000000 -i 20
                ;;
        *)
                echo "sleep"
                ;;
        esac
        sleep 10
done
```

List of Figures

List of Tables

Bibliography

[1] Mirja Kühlewind, Sebastian Neuner, and Brian Trammell. On the state of ECN and TCP options on the internet. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7799 LNCS:135–144, 2013.

[2] K Ramakrishnan, S Floyd, and D Black. The Addition of Explicit Congestion Notification (ECN) to IP, 2001.