INTERNATIONAL
HELLENIC
UNIVERSITY

Performance Analysis of TCP Fast Open

Dimitrios Grendas

SID: 3307190007

SCHOOL OF SCIENCE & TECHNOLOGY

A thesis submitted for the degree of

Master of Science (MSc) in Cybersecurity

April 2021

THESSALONIKI – GREECE

INTERNATIONAL
HELLENIC
UNIVERSITY

Performance Analysis of TCP Fast Open

Dimitrios Grendas

SID: 3307190007

Supervisor:	Assistant Prof. Anastasios Politis
Supervising Committee Members:	Prof. Cheilas
	Prof. Antoniou

SCHOOL OF SCIENCE & TECHNOLOGY

A thesis submitted for the degree of

Master of Science (MSc) in Cybersecurity

April 2021

THESSALONIKI – GREECE

Abstract

After a short analysis of the TCP mechanism and a thorough analysis of the design and implementation of TCP Fast Open (TFO), I will conduct my own testing using a Lab with the appropriate equipment in order to test the speed under several case scenarios and produce the final conclusions. Finally, I will examine the privacy and security concerns and talk about the future of the web.

Dimitrios Grendas

29/04/2021

Contents

ABSTRACT ..... **III**

CONTENTS ..... **IV**

1 INTRODUCTION **5**

1.1 STRUCTURE OF THE DISSERTATION 6

2 TRANSMISSION CONTROL PROTOCOL **7**

2.1 INTRODUCTION 7

2.2 TCP: CONNECTION ORIENTED TRANSPORT 8

2.3 TCP ON THE WEB TODAY 17

3 TCP FAST OPEN **19**

3.1 INTRODUCTION 19

3.2 TFO MECHANISM 20

3.3 TFO PERFORMANCE AND IMPLEMENTATION 25

3.4 SECURITY AND PRIVACY CONCERNS 34

4 EXPERIMENTAL **37**

4.1 TESTING ENVIRONMENT AND PREPARATION 37

4.2 TESTING WITH DIRECT CONNECTION 43

4.3 TESTING WITH MIDDLEBOX 45

4.4 TESTING WITH MIDDLEBOX AND FIREWALL 48

4.5 RESULTS SUMMARY 52

5 CONCLUSIONS **55**

BIBLIOGRAPHY **57**

APPENDIX **59**

1 Introduction

The Transmission Control Protocol (TCP) is one of the main protocols used on the internet today. The majority of the internet applications running today like webservers, file transferring, remote administration and more are utilizing the TCP protocol in order to provide reliable and error-checked communication.

Over the years since the adoption of the Internet Protocol suite the internet services have evolved vastly while the networking protocols do not follow the same rate of evolution. While the available bandwidth has increased, since the majority of the web flows are small TCP flows the TCP handshake mechanism creates significant delay to the overall latency.

In an effort to improve the communication latency over TCP under certain circumstances researchers and companies have been experimenting with a new TCP option. TCP Fast Open (TFO) is a mechanism in the TCP which allows a client, who previously had completed a TCP handshake on a server, to establish a connection with one less RTT when comparing to the traditional TCP handshake. This improves latency times in certain circumstances like small flows on the web.

Despite the advantages of the new TFO option there are numerous disadvantages which are equally important. Although it is not a new mechanism and relies to the TCP protocol, the new option introduces several compatibility issues along with critical security concerns.

1.1 Structure of the dissertation

On the first chapter a small introduction is written about the TCP network protocol and the need that led to the development of the TCP Fast Open mechanism.

On the second chapter a brief analysis will be conducted on the TCP protocol mechanism focusing on the three-way handshake. In addition, it will be briefly presented how the TCP is working today on the web trying to show the need for additional mechanisms to solve problems that have occurred.

On the third chapter a deep presentation will be written for the TCP Fast Open mechanism along with privacy and security concerns presented during the initial design of the extension.

On the fourth chapter it will describe the testing methods and procedures along with the results showing the performance gains.

On the fifth and final chapter conclusions about the mechanism will be presented along with a brief look into the future of the web technologies.

2 Transmission Control Protocol

2.1 Introduction

Back in 1970s, in a period when packet-switched networks began to emerge, researchers around the world developed multiple networks with the precursor of today's internet ARPAnet being one of them. While the principle of multiple networks connected with each other is the fundamental concept of the worldwide network today, the problem was that each of these networks had its own set of protocols. Soon this was recognized by two researchers Vinton Cerf and Robert Kahn who invented the TCP/IP that was intended to be cross-network compatible. Today the Transmission control protocol is one of the major protocols used for reliable communication between applications.[3]

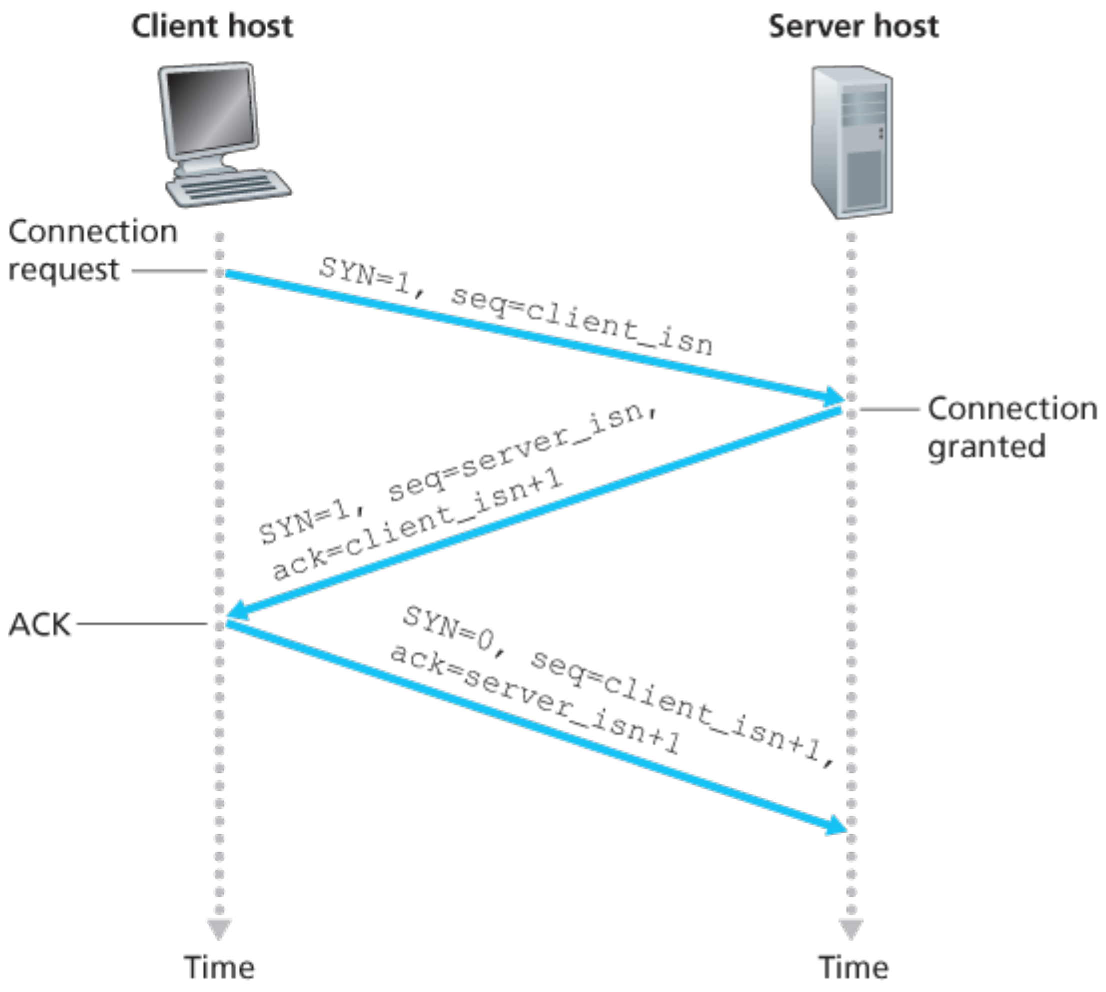
Unlike UDP, the TCP protocol is providing connection-oriented and reliable end to end communication. This protocol is widely used by the majority of applications today with the need of reliable data transfer without having to implement any mechanism on the application layer. Following the principles of reliable data transfer, it provides mechanisms including error detection, retransmissions, cumulative acknowledgements, timers and header fields for sequence and acknowledgment numbers.[2][3]

The connection of TCP is a logical one compared to a circuit-switched network that implements an end-to-end TDM or FDM. This means that the protocol runs only in the systems of the sender-receiver while keeping the connection state on both sides. The intermediate network devices that transfer the data between the hosts are unaware of the TCP connection state and are only able to see the information as datagrams.[3]

Another important capability of a TCP connection is the full-duplex service. While two applications have an open connection the application layer data can flow both ways simultaneously. In addition, it is only capable of point-to-point connections meaning that one process can have an open connection only with one other process making multicasting not possible.[3]

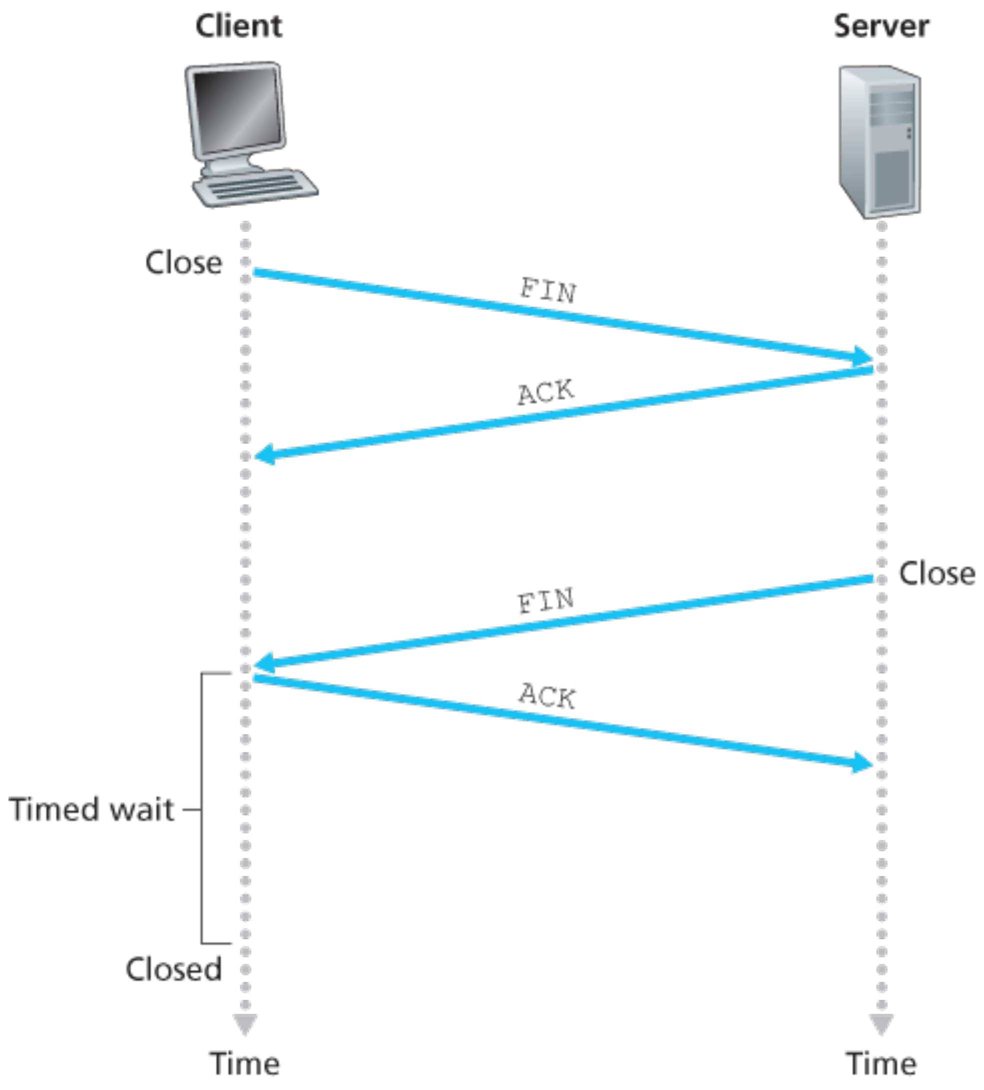
2.2 TCP: Connection Oriented Transport

The basic principle of TCP, as mentioned before, is that it is a connection-oriented communication end to end between only two host processes. In order for the connection to be initiated both sides have to exchange some information and agree to start the data exchange. During this procedure, the client proceeds to establish a TCP connection he first sends a special TCP segment then the server responds with another special segment and finally the client responds back with a segment that is starting the connection. Also, for the protocol to work correctly, the proper termination of the connection is essential.[2][3]



TCP three-way handshake: connection establishment [3]

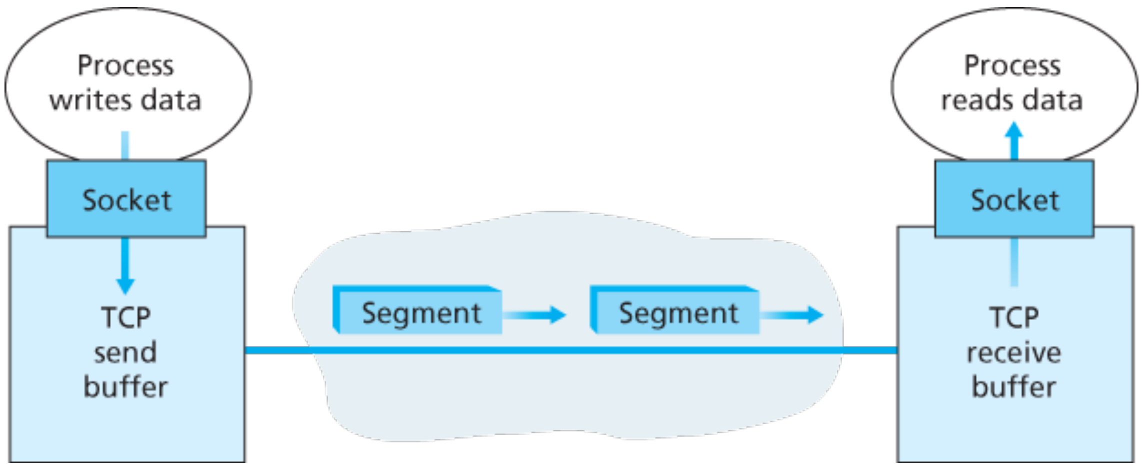
As we introduced above, the main mechanism of the TCP protocol is the establishment of the initial connection called three-way handshake. Firstly, starting from the client, a special TCP segment with the SYN bit set to 1 is sent to the server. Also, the sequence number is set to a random number generated by the sender in order to avoid some security attacks during this stage of connection. Once the IP datagram is received by the server, a segment is sent back to the client as a response that the connection is granted. This time the header acknowledgement field is set to the sequence number received from the client plus 1 (sequence number + 1), the sequence number field is set to a random number generated by the server and the SYN bit continues to be 1. Basically, in a nutshell, the server responds that the connection request received and granted with a sequence number it selects. This is also called SYNACK. Finally, the client receives the response from the server and sends back the last segment with the SYN bit set to 0 and the acknowledgement number set to the server's sequence number plus 1 (sequence number + 1). Now that the connection is established, the SYN flag continues to be 0 and the two parties can start exchanging data. [2][3]



TCP connection termination [3]

Now that the connection is established both parties can terminate it. The process is initiated by sending a TCP segment with the FIN bit set to 1 and then wait for the receiver to send back an acknowledgement segment. This exchange of segments must be repeated for each side of the connection which means that the receiver now, after sending the ACK segment in response to the FIN, must now send a FIN segment while waiting again for the ACK segment from the sender that initiated the termination previously. [2][3]

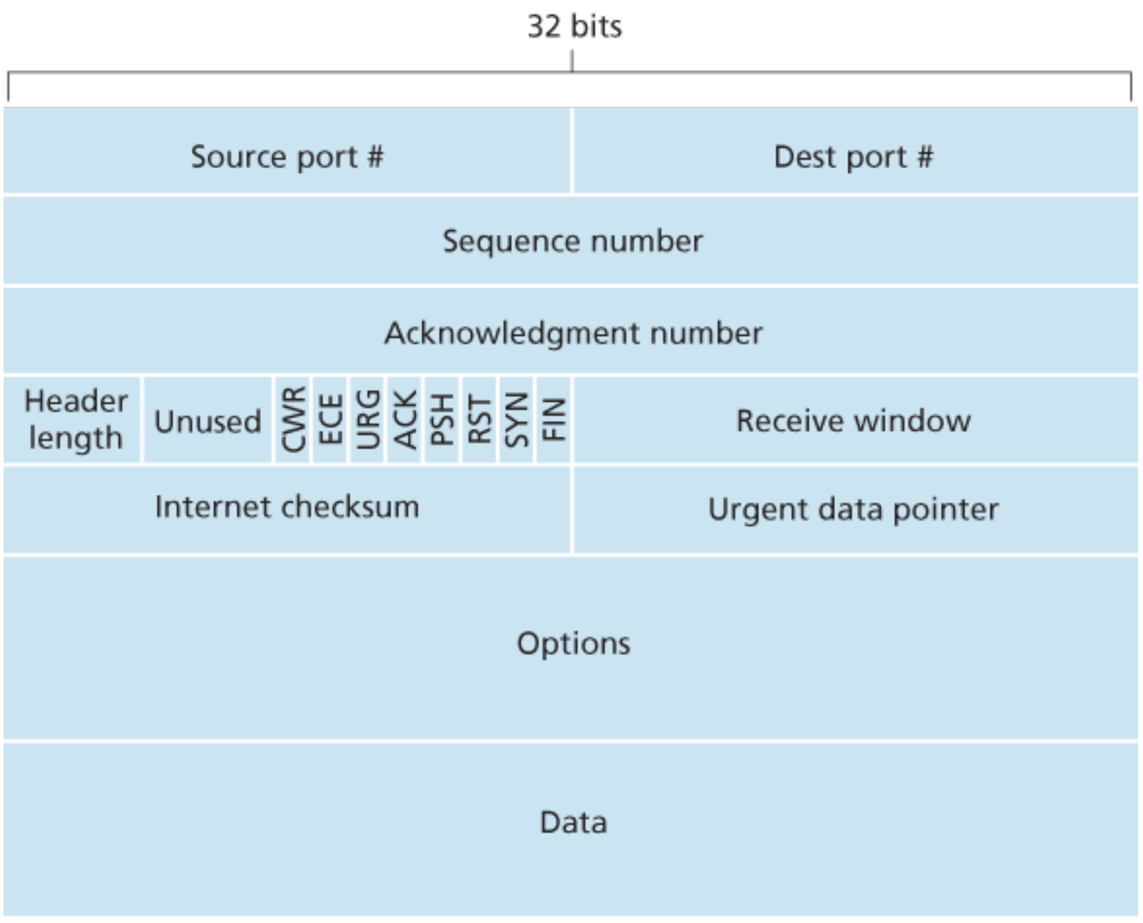
Once the connection has been initiated the data exchange can start flowing between the two end points. When a process sends a set of bytes as data through the appropriate socket, the TCP directs them to the connection's send buffer which acts as a que of data to be sent over in chunks. The frequency of data pulled from the buffer is not described in the RFC 793 stating that TCP should "send that data in segments at its own convenience". The size of that data sent with each packet is called Maximum segment size or MSS and it is typically 1460 bytes of data plus 40 bytes for the TCP header consisting of 1500 bytes which is the maximum MTU size of the Ethernet and PPP link-layer protocols for a frame. It is important to clarify that the MSS value indicates the maximum size of the data from the application layer and not the maximum size of the TCP segment with the headers included. [2][3]



TCP send and receive buffers [3]

With the data buffer available pulled parts of data are fetched by TCP and paired with a header forming the TCP segments. Following the encapsulation method in networking these segments are included in the network-layer IP datagrams and sent over to the other end of the TCP connection. Upon receiving this segment, the TCP on the recipient uses the TCP receive buffer from which the data is available for the application to pull from. [2][3]

Having taken a brief look at the TCP connection, let's examine the TCP segment structure. The TCP segment includes the header fields and a data field. The figure bellow shows the exact structure which is 1500 bytes in size, 40 bytes maximum size for the headers and 1460 bytes maximum size for the application data. [2][3]

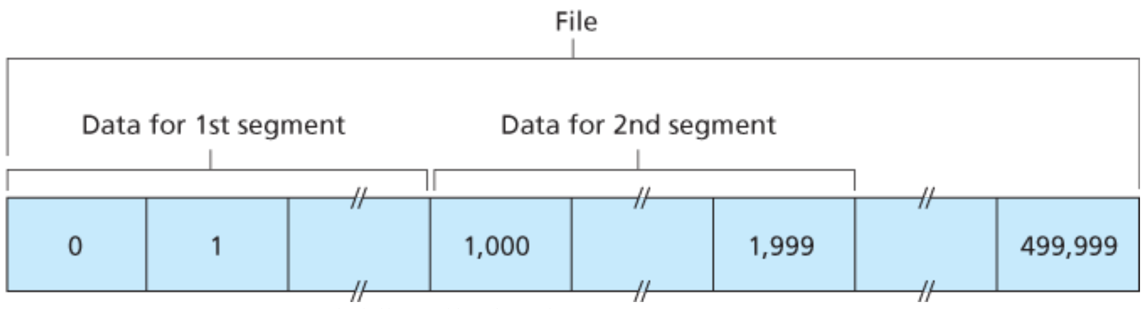


TCP segment structure [3]

The header fields consist of all the information is needed by the TCP protocol to have a successful communication along with other functions and options. While the maximum size can be 40 bytes long the usual size is around 20 bytes depending on the functions used and the state of the connection. As shown from the figure above, the header contains the source and destination port numbers which is used for multiplexing and demultiplexing data through the layers. Very critical also is the internet checksum field that is responsible for the integrity of the data. The 32 bit sequence number field and 32 bit acknowledgment number field are essential for the reliability of the data transfer service as we discussed extensively previously. Flow control is also needed for a reliable TCP connection which is using the 16 bit receive window field to indicate the number of bytes the client is accepting while the options field is used to implement other functions of the protocol which is typically empty thus the length of the header is usually 20 bytes long. Another field we see in the header structure is the 16 bit urgent data pointer field which, being the last field, is responsible for informing the location of the last byte of data that the flag with URG bit enabled is informing that exists, but usually it is not used. Last but not least, essential field for the protocol to function properly is the flag field which is 6 bit long. Depending on the state of the connection this field changes and is used appropriately by the protocol. The ACK bit indicates that the value of the acknowledgement field is correct and the RST, SYN, FIN bits are used for connection initialization and termination. The CWR and ECE bits are used by the congestion mechanism and the PSH bit informs the receiver to pass data to the upper layer straight away. Lastly, the URG bit is used to indicate that there is “urgent” data that needed to be handled appropriately by the receiver. In real world scenarios it is typical that some header fields staying unused thus empty. [2][3]

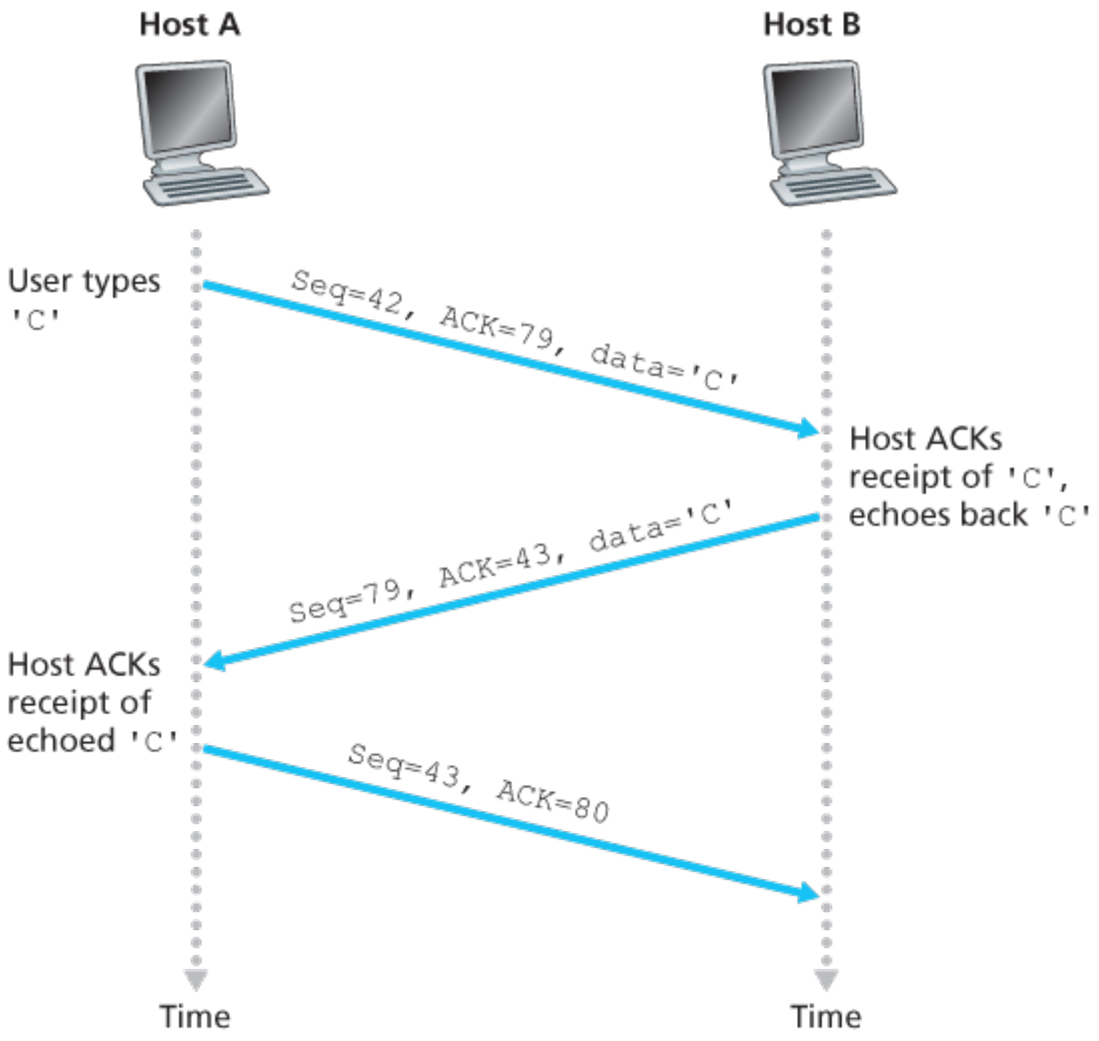
The data field contains the actual application data that is needed to be transferred and, as mentioned above, the size of the field cannot exceed the MSS limits. Because of that, larger files such as images while loading a web page are sent in pieces of the maximum size allowed by the MSS. While it is necessary to respect the size limits in order for the protocol to work correctly, data field size can be lower than the maximum allowed. This highly depends on the application, for example segments sent by a telnet application may be only 21 bytes in length (20 bytes for the headers field and 1 byte for the data field). [2][3]

One of the main characteristics of the TCP protocol is the ability to perform reliable data transferring. It is very important that the receiver responds on each data segment with the appropriate acknowledgment which ensures the sender that the data is received and awaits for the next stream of bytes. [2][3]



Dividing file data into TCP segments [3]

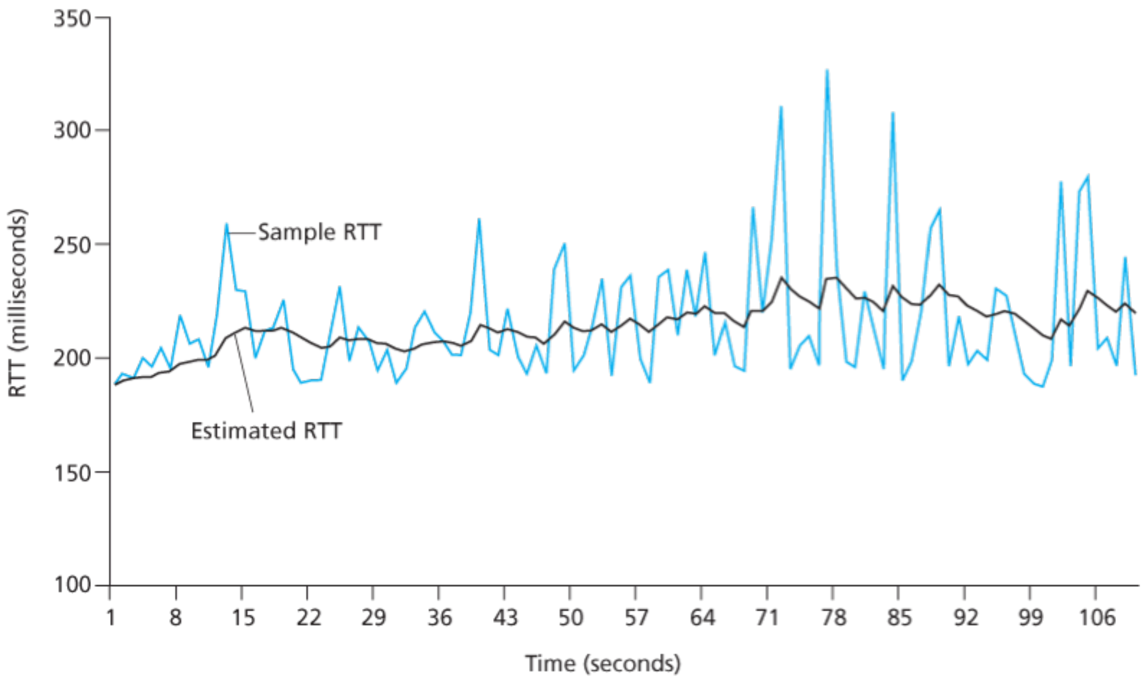
It is crucial to discuss about the mechanism which is responsible to send larger data sizes over a TCP connection. As we mentioned before, when the data is larger than 1460 bytes the protocol divides the data into parts and sends them one by one. The protocol "sees" the data as unstructured stream of bytes so it starts by numbering them from the start to the end, for example in a 10.000 bytes data stream the first byte is number 0 and so on. When the first segment is sent, the data is inserted to the data field in respect to the MSS limits and the sequence number on the header is the number from the first byte of the stream. For example, in the case of 1000 bytes MSS limit, the first sequence number is 0, the second 1000, the third 2000 and so on. [2][3]



TCP Sequence and acknowledgement numbers for a Telnet application [3]

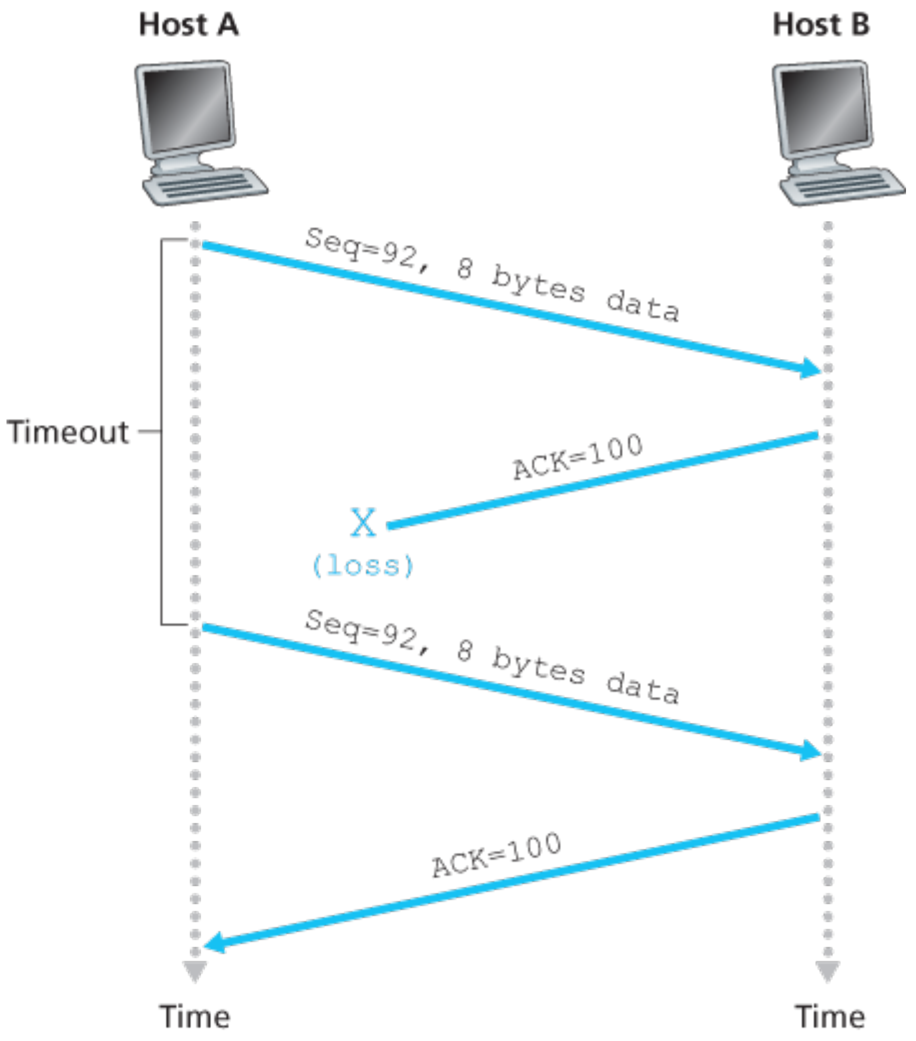
As described earlier, the receiver of the data must respond to the sender with an appropriate ACK number. The acknowledgement field is filled with the number of the next byte of data stream that the receiver is expecting. For example, if the client sends a data stream with sequence number of 40 then the server must respond with acknowledgement number of 40 + 1 = 41. Since the TCP is full-duplex and data can flow in both directions, in our example, if the receiver sends also data the same mechanism apply with the sequence and acknowledgement numbers. Also, it is important to mention that the sender can send multiple data stream segments without waiting each time for the receiver's response. In this case TCP protocol implements other mechanisms that are responsible for lost segments, reliable data transfer, congestion management, etc. [2][3]

In order for the TCP protocol to recover from lost segments it uses a timeout/retransmit mechanism. First, it needs to calculate an estimated round-trip time (RTT) by taking a sample at certain times. This RTT indicates the time between a sent segment and the receipt of the appropriate acknowledgement. [2][3]



RTT samples and RTT estimates [3]

When the estimated RTT is calculated the TCP mechanism starts a timer which, when exceeds the limits of the calculated value, is used to indicate that the sender has not received the ACK from the receiver. As a result, the sender will retransmit the segment that caused the timeout event and will restart the timer. There are also additional mechanisms that help adapt in certain scenarios like doubling the timeout interval. [2][3]



Retransmission due to a lost acknowledgement [3]

The mechanism of timeout and retransmission can often cause significant delays. This is due to the timeout period that can be a relatively long. Actually, the sender can usually detect packet loss before the timeout period by receiving duplicate ACKs. When this scenario occurs, the sender performs a fast retransmit which basically means that the missing segment is sent out before the timeout period expires. The TCP error-recovery mechanism can retransmit the lost segments using two methods, the go-back-n and selective repeat. In a nutshell, the go-back-n mechanism retransmits all the lost packets from the lost one to the latest transmitted but not the other hand the selective repeat retransmits only the missing segment. [2][3]

Of course, the TCP protocol is a bit chapter in the networking world and while it has many more details and implements more mechanisms for flow control, congestion management, etc. that, for the purpose of this thesis, are not going to be further analyzed. [2][3]

2.3 TCP on the web today

Since the birth of the first web page in 1990 the total size that the clients have to download in order to load the complete page has increased drastically. In addition to the total size, the inclusion of multiple images, videos, CSS/JS files and more increased the TCP flows on the internet over the years.

Web pages larger and more complicated resulted to multiple TCP flows for a single page to load which over the years created a problem with the increasing latency of the traditional three-way handshake mechanism produced in such scenarios. Reports from httparchive.org shows that from a median desktop web page size has increased 335.8% over the last ten years which verifies the recent years' trend of websites evolution. [8]

Of course, web pages are not the only technology that uses the TCP protocol. In fact, almost every network capable application that requires reliable data transfer over the network implements the TCP mechanism. With the rise of the smartphone industry and the evolution of software and networking technologies, nowadays every single network client uses multiple TCP enabled applications. These applications, depending on the use case scenario, usually create multiple short TCP flows which also introduces latency. For example, a complicated progressive web application, which is relative new method for delivering applications on smartphones using web technologies, can suffer from latency when loading elements using multiple TCP connections.

These are only a few examples where the traditional TCP connection mechanism can affect negatively the user experience and, in some cases, cause problems in latency sensitive applications. TCP protocol, since it's invention, has not evolved mechanisms at the same rate as the web evolution. This created the need for refinements and introduction of new technologies and mechanism that either work together or supplement the protocol in order to keep it up to date in terms of performance and compatibility.

3 TCP Fast Open

3.1 Introduction

Throughout the course of the years, as we mentioned earlier, the TCP protocol has been revised multiple times and also introduced some extensions in order to solve problems or improve performance. In July 1994, Bob Braden published the experimental RFC 1644 “T/TCP – TCP Extensions for Transactions – Functional Specification” which attempted to fill the gap between TCP and UDP and solve the problem of the latency introduced by the three-way handshake along with the delay produced when closing a connection. In order to improve performance in these two areas, he introduced a way to bypass the three-way handshake and also reduce the delay in the TIME-WAIT state. [10]

As this was the earliest attempt to solve performance problems this approach suffered from several major security problems as described later in 1996 in a memo written by Charles Hannum. These security problems resulted in moving the RFC 1644 and RFC 1379 to Historic Status by a new RFC 6247 published in 2011. [10][11][12][13]

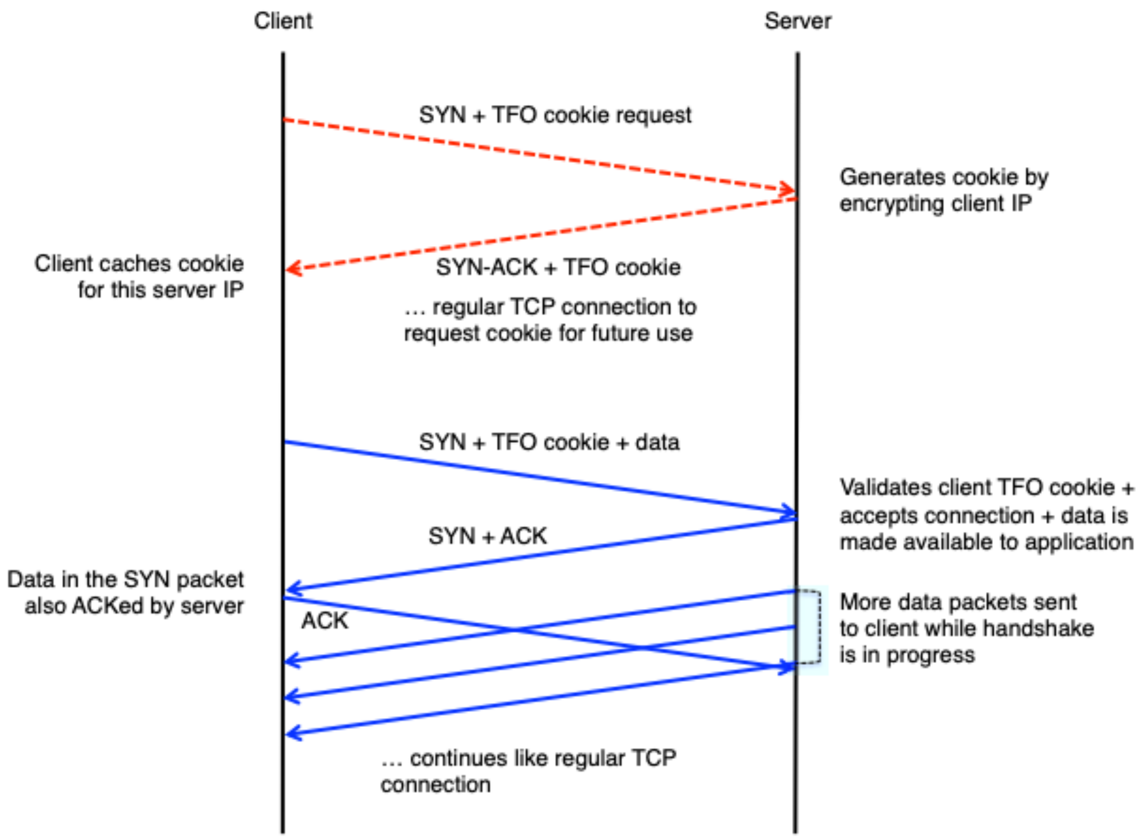
The same year, by the end of 2011 a new proposal presented by Google, UC San Diego and ICSI that described a new mechanism as an extension of the TCP protocol in an attempt to solve the problem of the increased latency of today’s multiple short TCP transfers which led to a publication of an experimental RFC 7413 later in December 2014. Applying technics in the three-way handshake mechanism along with security practices this proposal helped mitigating the problem of cold requests on the web by reducing the latency produced by the connection initialization.

Although the TFO proposal was an effective way to solve the performance impact introduced with the evolution of the web, it also widened the attack surface of the protocol which introduced numerous security issues as we will describe later on.

3.2 TFO Mechanism

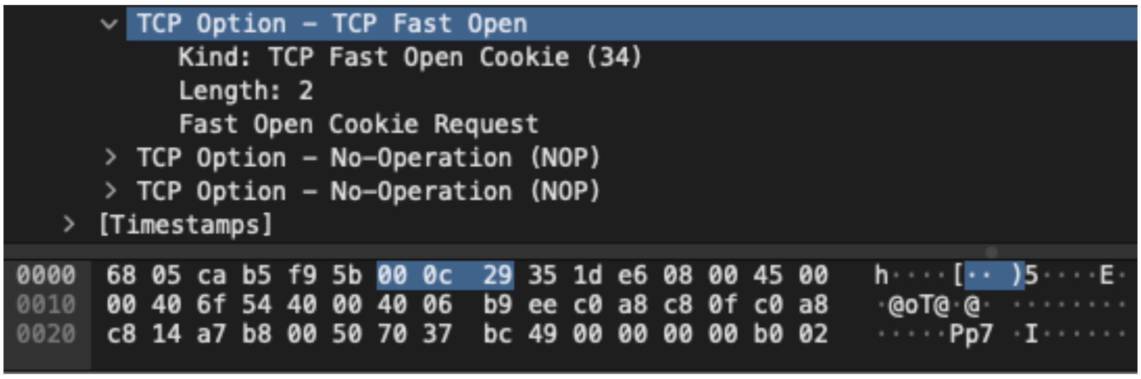
With the introduction of the TCP Fast Open mechanism the main purpose was to reduce the latency of the initial TCP connections. Allowing data to be exchanged during the initial three-way handshake there is a reduction of one RTT when initiating connections while this is benefit can show up after the second connection between two machines.

The “traditional” three-way handshake, as we described in a previous chapter, is the mechanism to initiate a TCP connection from one machine to another and until the procedure completes every step only then data can flow. The TFO introduces a new mechanism in a form of a TCP option which allows hosts to exchange data during the initial handshake. This results in a reduced latency in multiple flows due to the fact that each time a connection is initiated, one step from the three-way handshake is eliminated. [14]


Client
Server
SYN + TFO cookie request
Generates cookie by encrypting client IP
Client caches cookie for this server IP
SYN-ACK + TFO cookie
... regular TCP connection to request cookie for future use
SYN + TFO cookie + data
Validates client TFO cookie + accepts connection + data is made available to application
SYN + ACK
Data in the SYN packet also ACKed by server
ACK
More data packets sent to client while handshake is in progress
... continues like regular TCP connection

TFO connection overview [14]

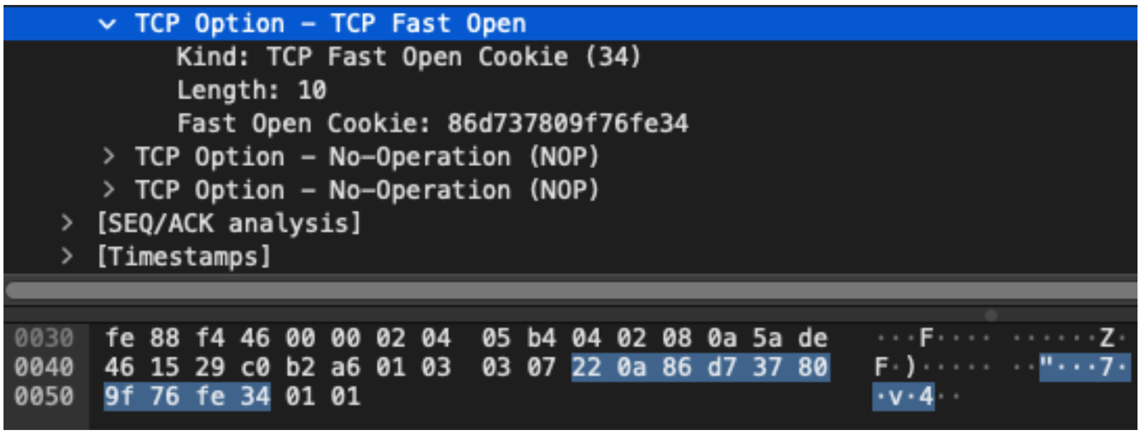
Assuming that the host and the client have TFO support, as it will be discussed later on, the client starts the procedure to establish connection with a server by sending a traditional SYN packet but this time with a Fast Open Cookie Request TCP option. Receiving this packet, the server generates a cookie by encrypting the source IP and saves it to a local cache space for future use. Then it proceeds with sending back the SYN-ACK packet but adding also the generated cookie in the TCP option header field. The client also caches this cookie for future connections to the same destination and continues with the normal behavior of the TCP protocol by starting the data exchange. [14]



```
      ∨ TCP Option - TCP Fast Open
          Kind: TCP Fast Open Cookie (34)
          Length: 2
          Fast Open Cookie Request
      > TCP Option - No-Operation (NOP)
      > TCP Option - No-Operation (NOP)
    > [Timestamps]

0000  68 05 ca b5 f9 5b 00 0c  29 35 1d e6 08 00 45 00   h····[·· )5····E·
0010  00 40 6f 54 40 00 40 06  b9 ee c0 a8 c8 0f c0 a8   ·@oT@·@· ········
0020  c8 14 a7 b8 00 50 70 37  bc 49 00 00 00 00 b0 02   ·····Pp7 ·I······
```

Fast Open Cookie Request option



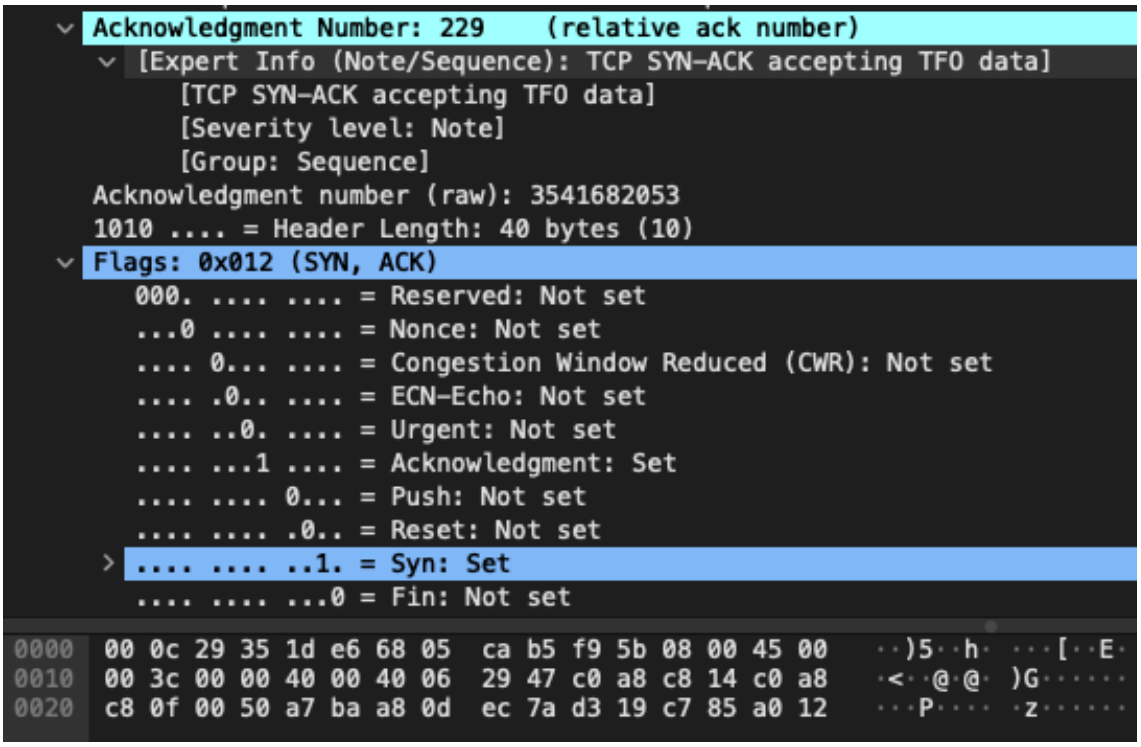
```
      ∨ TCP Option - TCP Fast Open
          Kind: TCP Fast Open Cookie (34)
          Length: 10
          Fast Open Cookie: 86d737809f76fe34
      > TCP Option - No-Operation (NOP)
      > TCP Option - No-Operation (NOP)
    > [SEQ/ACK analysis]
    > [Timestamps]

0030  fe 88 f4 46 00 00 02 04  05 b4 04 02 08 0a 5a de   ···F···· ······Z·
0040  46 15 29 c0 b2 a6 01 03  03 07 22 0a 86 d7 37 80   F·)····· ··"···7·
0050  9f 76 fe 34 01 01                                  ·v·4··
```

Fast Open Cookie sent with SYN-ACK to the sender

As we notice, at this stage we have no performance gains from the use of this option. The performance increase will start appearing with the second connection to the same destination.

When the client wants to initiate a TCP connection to the same destination, assuming this is the same server as we described above, it will use the cookie received and saved to the cache. The client will now send a SYN packet but instead of a Fast Open Cookie request will use the cookie on the same TCP option field. Also, on the same packet, the data field will be populated with the first data stream without the need for the connection to be initialized before starting the data flow. Upon receiving this SYN packet, the server will validate the cookie with the one in the cache by decrypting it and comparing the IP address. If the cookie is valid then the server will send back a SYN-ACK that not only acknowledges the SYN but also the data received. [14]



```
  ∨ Acknowledgment Number: 229    (relative ack number)
    ∨ [Expert Info (Note/Sequence): TCP SYN-ACK accepting TFO data]
        [TCP SYN-ACK accepting TFO data]
        [Severity level: Note]
        [Group: Sequence]
    Acknowledgment number (raw): 3541682053
    1010 .... = Header Length: 40 bytes (10)
  ∨ Flags: 0x012 (SYN, ACK)
      000. .... .... = Reserved: Not set
      ...0 .... .... = Nonce: Not set
      .... 0... .... = Congestion Window Reduced (CWR): Not set
      .... .0.. .... = ECN-Echo: Not set
      .... ..0. .... = Urgent: Not set
      .... ...1 .... = Acknowledgment: Set
      .... .... 0... = Push: Not set
      .... .... .0.. = Reset: Not set
    > .... .... ..1. = Syn: Set
      .... .... ...0 = Fin: Not set
0000  00 0c 29 35 1d e6 68 05  ca b5 f9 5b 08 00 45 00   ··)5··h· ···[··E·
0010  00 3c 00 00 40 00 40 06  29 47 c0 a8 c8 14 c0 a8   ·<··@·@· )G······
0020  c8 0f 00 50 a7 ba a8 0d  ec 7a d3 19 c7 85 a0 12   ···P···· ·z······
```

SYN-ACK response from the server accepting connection and data

After this exchange of packets, the server may send additional segments to the sender before receiving the first ACK. The client also has to send an ACK to inform the server's SYN packet before continuing with the normal TCP data flow. [14]

It is crucial to mention that in the case of either the client or the server do not support the TFO option the connection will fall back to the traditional three-way handshake. For example, if a client sends a SYN packet with TFO cookie request and the server does not implement this extension, the option header field will be discarded and respond with normal SYN-ACK packet. The client will adapt to this response by continuing the normal handshake. Another example is when the client has already a TFO cookie in cache and proceeds with an additional connection to the same server, but in this scenario the server stopped supporting the TFO extension. The server will respond the same way as the previous example by ignoring the TFO option but also the data carried out. [14]

```
                                +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
                                |      Kind     |    Length     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                            Cookie                             ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Kind              1 byte: value = 34
Length            1 byte: range 6 to 18 (bytes); limited by
                          remaining space in the options field.
                          The number MUST be even.
Cookie            0, or 4 to 16 bytes (Length - 2)
```

Fast Open Option [15]

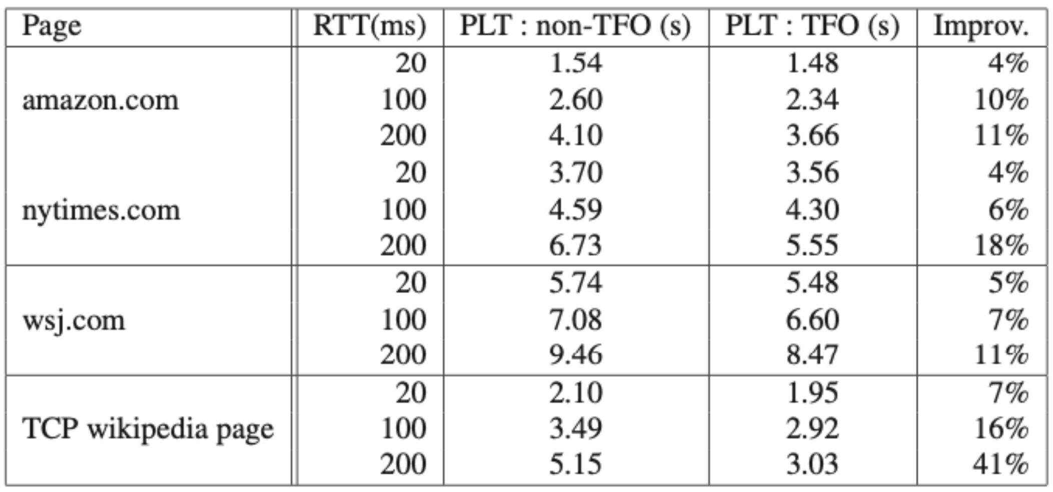
The cookie used by the TFO extension is the core element in order for the mechanism to work. It was designed not only to validate the client and be able to reduce the connection establishment latency but also with security in mind. As mentioned before, it is generated by the server only while it can be minimum 4 bytes and up to 16 bytes. The encryption mechanism implemented can vary depending on the configuration of the server, for example AES 128 can be used which is very fast on today’s processors. Using the client’s IPv4 or IPv6 along with a secret key, an encrypted value is produced by the server. Without the secret key the cookie cannot be decrypted or validated thus improving the security aspect of the mechanism. Also there are no strict restrictions on how many cookies will be generated for each client and it is possible for a server to encode additional information in the cookie and thus accepting more than one cookie for the same IP address. Additionally, cookie expiration is very important from the security aspect and the server can invalidate cookies according to the configuration. For example it can happen periodically by encoding the timestamp or by changing the secret key. It is also possible for the key changing to be performed automatically every day or any period configured. Also it is possible for a server to have multiple keys active at the same time for example larger implementations may find this useful and more secure. [15]

From the client side, the TFO cookie is cached and used for initiating connections by sending a SYN packet with Data. The client may also cache the advertised MSS by the server in order to send the appropriate size of data when sending the SYN packet and this is important because in the TFO mechanism the client starts sending data before the receiver advertises this value. Without the MSS cached the client will use the default MSS of 536 bytes for IPv4 (RFC 1122) and 1220 bytes for IPv6 (RFC 2460). Along with the cookies and the MSS value the client should also cache any negative responses from each server such as not acknowledging data in SYN or ICMP errors in order to possibly disable the TFO option. Lastly it is recommended for the client to cache also the server's port numbers as the TFO is enabled on a per-service-port and the cookies are not discriminated by port number. [15]

3.3 TFO Performance and Implementation

TCP Fast Open upon introduction to the public, despite being an interesting mechanism it is crucial to measure the performance in real world scenarios. Among the positives that it needs minimal configuration and basically promises improvements on the TCP protocol latency it has to be examined thoroughly not only for the performance gains but also for the application on the existing users and servers. In this chapter, not only we will take a look on the performance results from several studies conducted in the past, but also examine the implementation from the end-user side while taking into a deeper and more practical view on how to enable it on the most popular browsers.

Taking a deeper look on the original paper written by members of Google and ICSI we can see the performance benefits from the conducted tests both on the client side and on the server side. The case scenario was a whole page download simulating as close as possible a real-world user using his computer at home. Using a single computer running Linux and Chrome browser with TFO enabled, the scientists benchmarked the most popular websites from the Alexa top 500 websites back in 2011 using a tool called "Google web page replay tool" for both TFO-enabled Chrome and for standard Chrome browser. Firstly, using the "record mode" of this tool, the researchers recorded all the network traffic while loading the webpages and particularly filtering the DNS and TCP traffic which was then saved in a local database. Afterwards, using the "replay mode" of the tool which runs a DNS server on the local computer, it redirects the Chrome's HTTP requests to a local proxy server. In this mode a dummynet was used to emulate the network setup for this experiment which was 4 Mbps download, 256 Kbps upload, 128 KB buffer and using all the times the loopback interface with MTU of 1500 bytes. Using the replay method of the Google web page replay tool, the webpages of Amazon.com, New York Times, the Wall Street Journal and the Wikipedia was replayed 20 times with and without TFO in the Chrome browser with three different RTTs of 20 ms, 100 ms and 200 ms accumulating 120 samples in total. In order to have reliable results, the browser cache was empty in every single replay avoiding caching issues and therefore inaccurate measurements. Measuring the page load time from the time chrome browser begins processing the link until the on-load event begins, as shown at the table below, TFO resulted in interesting results.[14]

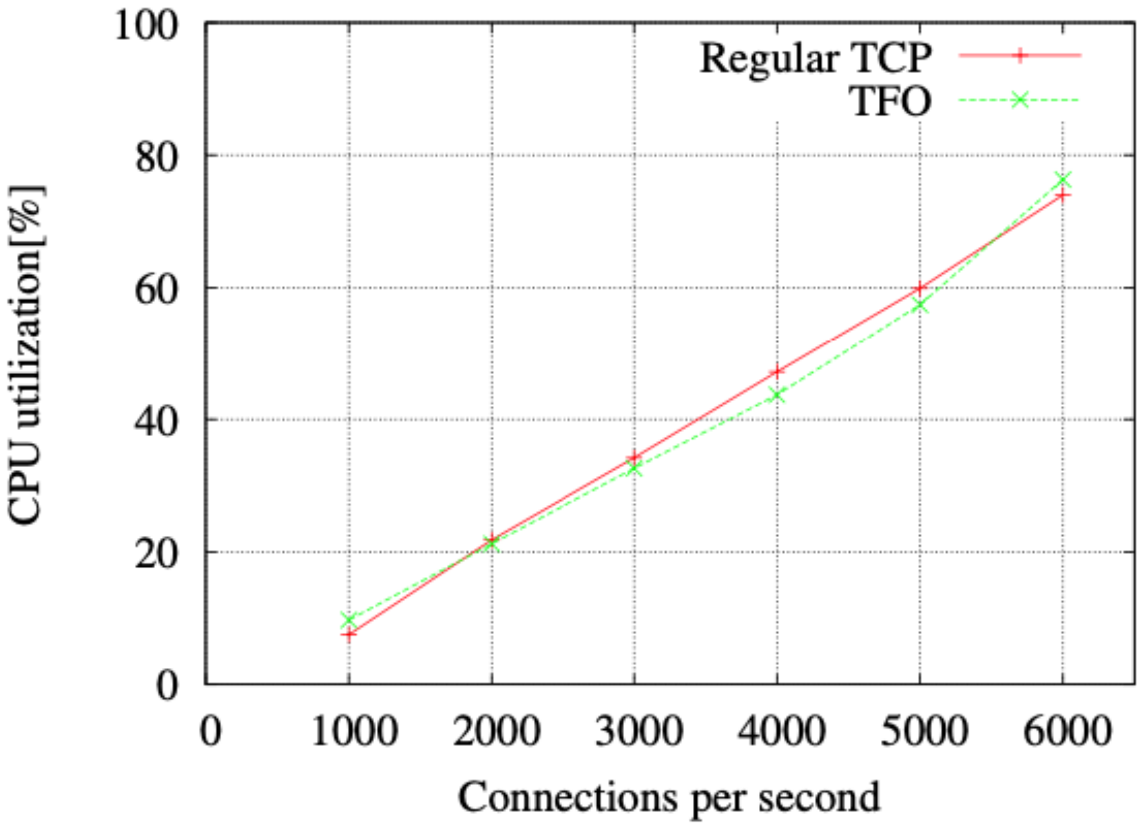


Page	RTT(ms)	PLT : non-TFO (s)	PLT : TFO (s)	Improv.
amazon.com	20	1.54	1.48	4%
	100	2.60	2.34	10%
	200	4.10	3.66	11%
nytimes.com	20	3.70	3.56	4%
	100	4.59	4.30	6%
	200	6.73	5.55	18%
wsj.com	20	5.74	5.48	5%
	100	7.08	6.60	7%
	200	9.46	8.47	11%
TCP wikipedia page	20	2.10	1.95	7%
	100	3.49	2.92	16%
	200	5.15	3.03	41%

Average page load time (PLT) in seconds for various pages [14]

As seen from the results, the TFO mechanism significantly improves the page load time when the RTT is high for all the different TFO-enabled websites which is expected since the benefits are higher when the RTT is high and having one less RTT. Having less benefit is where the RTTs are low but despite the fact that this is expected the benefits are still present even with pages that have heavy content. It is important to mention that, especially for the 200 RTT values, it is in fact simulating a mobile device with cellular connection since this is close to real world scenario.[14] But today, with the 5G cellular networks, maybe this is not so representative especially if the next generation mobile network evolves to the expected 1 ms latency.[14]

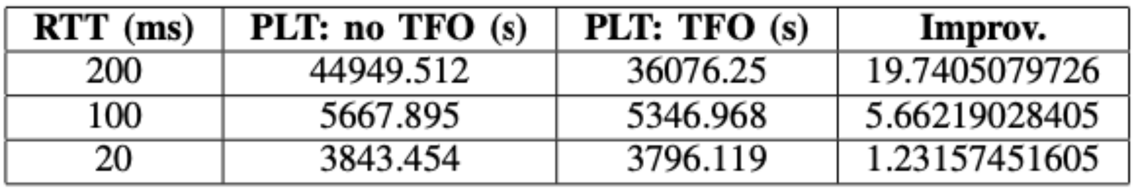
Although client performance is critical to the user experience, another fact as important is the server performance. When talking about multiple users trying to access a TFO-enabled server to load a specific webpage, in order to have the existing servers working properly without the need for upgrading, it is critical to measure the impact of this mechanism. In extension of the previous tests, the researchers initiated CPU utilization tests on several server load levels in order to have a conclusion on how much impact the TFO mechanism has on a traditional Apache server. Using a custom written program that generates HTTP 1.0 requests and fetching a 5 KB webpage, they were able to take accurate measurements when thousands of requests occur on the same machine. This server was connected through Gigabit ethernet switch with around 100 μs RTT while it was configured with only one CPU in order to avoid multitasking and have reliable results. Testing for each number of connections per second was conducted with 5 minutes length for both TFO and non-TFO modes. As shown in the figure below the results are impressive since even with the overhead of the cookie generation/validation process the average CPU utilization was almost the same as without the TFO mechanism enabled. Particularly the CPU load, when using TFO, between 2000 and 5000 requests per second was marginally lower since there are fewer packets that the server needs to process. In addition to that, the AES encryption used for the cookie validation is a putting less that 0.3% overhead on the CPU utilization even when the requests are 6000 per second.[14]



CPU utilization vs. web server load level [14]

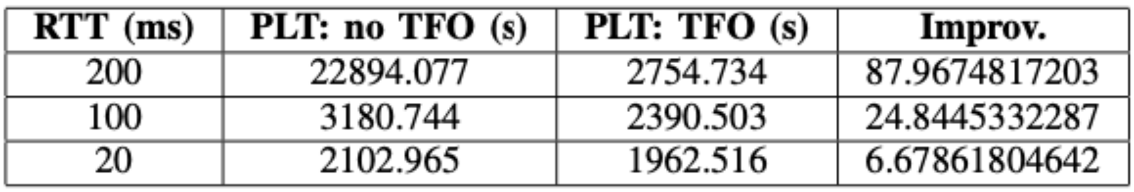
Although the performance analysis from the original paper back in 2011 was thorough and detailed trying to emulate a real world case, later on, other papers have emerged conducting their own testing using similar methods. Erich Meissner from the Georgia Institute of Technology wrote a paper analyzing this mechanism using the same RTT numbers for each test as the previous mentioned testing. On this case, he used three websites from www.washingtonpost.com, www.admission.gatech.edu and www.web.mit.edu with and without the TFO mechanism. It is particularly mentioned that RTT numbers higher than 100 ms are emulating devices connecting through Wi-Fi and, as mentioned from the previous paper, the benefits are greater on these cases. Also, when the processing time of the client is higher than the network times it is natural that the TFO benefits are lower. The results were as expected on how the mechanism works on improving the page load times but while testing these particular websites some numbers exceeded expectations and were quite impressive. While the Washington post website presented the weakest improvements, the other two pages yielded in greater improvements by 88% for the afmission.gatech.edu and 92% for the web.mit.edu due to the type of the content in these websites.[16] As it is known up until now from the investigation of the mechanism, the type of the website and the content is critical for the performance gains.

TABLE I
HTTP://WWW.WASHINGTONPOST.COM



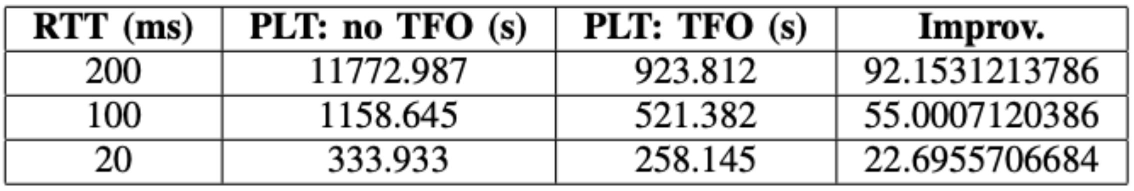
RTT (ms)	PLT: no TFO (s)	PLT: TFO (s)	Improv.
200	44949.512	36076.25	19.7405079726
100	5667.895	5346.968	5.66219028405
20	3843.454	3796.119	1.23157451605

TABLE II
HTTP://WWW.ADMISSION.GATECH.EDU



RTT (ms)	PLT: no TFO (s)	PLT: TFO (s)	Improv.
200	22894.077	2754.734	87.9674817203
100	3180.744	2390.503	24.8445332287
20	2102.965	1962.516	6.67861804642

TABLE III
HTTP://WWW.WEB.MIT.EDU



RTT (ms)	PLT: no TFO (s)	PLT: TFO (s)	Improv.
200	11772.987	923.812	92.1531213786
100	1158.645	521.382	55.0007120386
20	333.933	258.145	22.6955706684

Results of Page Load Times from websites as tested [16]

Another analysis for the performance of the TFO mechanism is described in the paper written by members of University Carlos III of Madrid in Spain along with member of Simula Research Laboratory in Norway. In this setup the analysis was different by crowdsourcing and analyzing results from 46 users in 18 different countries and 22 different ISPs when they try to establish TFO connections in multiple ports on a server. This analysis is different by not examining the performance gains but the success rate of the TFO while passing through middleboxes. The results are quite interesting showing that only 41.3% of the packets with TFO option were able to arrive to the server while 39.13% arrive with the option removed. In addition, 67.86% of the packets arrived in the server with the data on the SYN packet removed by the middlebox.[17] This result is very important when considering that in real world scenarios, depending on the client's location and the targeted website, the packets pass through different network paths involving various middleboxes. Along with this paper there are multiple tests on the internet concluding that the problem with middleboxes is important, for example an article from the LWN.net estimates that around 5% of these intermediate devices on the web removes the TFO option.[20] This information, along with multiple security concerns that are going to be presented later on, is critical for the future of the mechanism.

Having taken a brief look on the performance and practical testing of the TFO it is important for the Operating systems and the browser clients to implement and support this mechanism. In particular, three main operating systems are going to be discussed along with three browsers on which extend the TFO option is supported and how practically is enabled.

Since the introduction of the TFO mechanism, over the years multiple vendors of software updated their software in order to support the new option most of them as an experimental feature. Particularly the TFO was implemented in the following versions of browsers and operating systems:

- Mozilla Firefox: from version 58 [25]
- Google Chrome and Chromium
- Linux kernel version 3.7 [18]
- Microsoft Edge since Windows 10 Preview build 14352 [22]
- Apple iOS 9 [21]
- MacOS X 10.11 [21]

- FreeBSD version 10.3 for servers and version 12.0 for clients [23][24]

Although the TFO option over the studies and tests has presented numerous benefits, the support for the browsers had never left the experimental state. Despite the fact that over the years software companies have supported it and included the option to enable it, today is not found in the options and therefore it is impossible to show how to enable it on the current versions of Chrome, Microsoft Edge and Firefox which dropped support due to concerns mainly about security. In addition, even when supported it was not enabled by default thus forcing the user to take a deep dive in the experimental features of each browser. Due to these reasons, older versions of these browsers were selected in order to show the process of enabling the option.

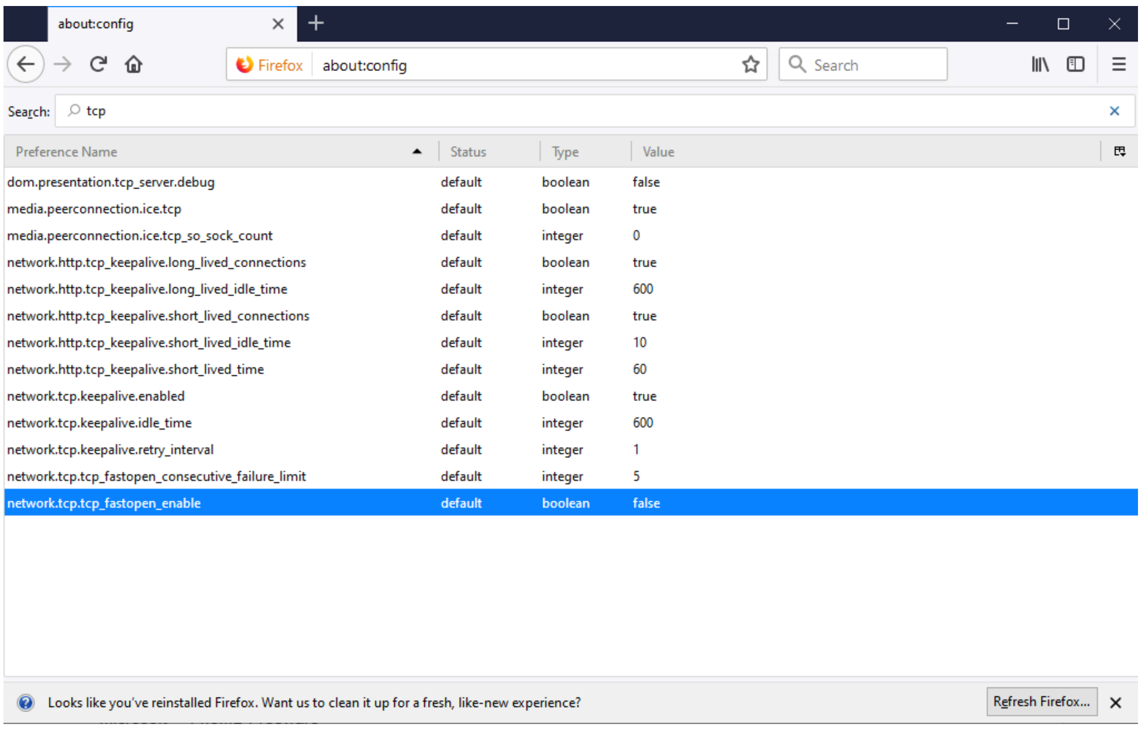
In order to enable the TFO on the system level in the Linux operating system, a system file needs to be modified in order to implement the option depending on the use case scenario. It can be enabled solely as a client or as a server but also supporting both modes. As described later on the detailed procedure, it is not a GUI option but requires a terminal. Lastly, on the operating systems Windows 10 and MacOS X the option is enabled depending on the browser version and support.

On the web browser applications, since the terminal is not an applicable option, the TFO is enabled by using the build-in GUI on the application window but still some developer knowledge is required since it is an experimental feature not intended for the average user.

On the Mozilla Firefox version 58, the option can be turned on by writing the following URL on the address bar:

- about:config

Afterwards, a list with the developer options appears and the boolean value “network.tcp.tcp_fastopen_enable” can be changed to “true” as shown in the image bellow.


about:config
Firefox about:config
Search
Search: tcp
Looks like you've reinstalled Firefox. Want us to clean it up for a fresh, like-new experience?
Refresh Firefox...

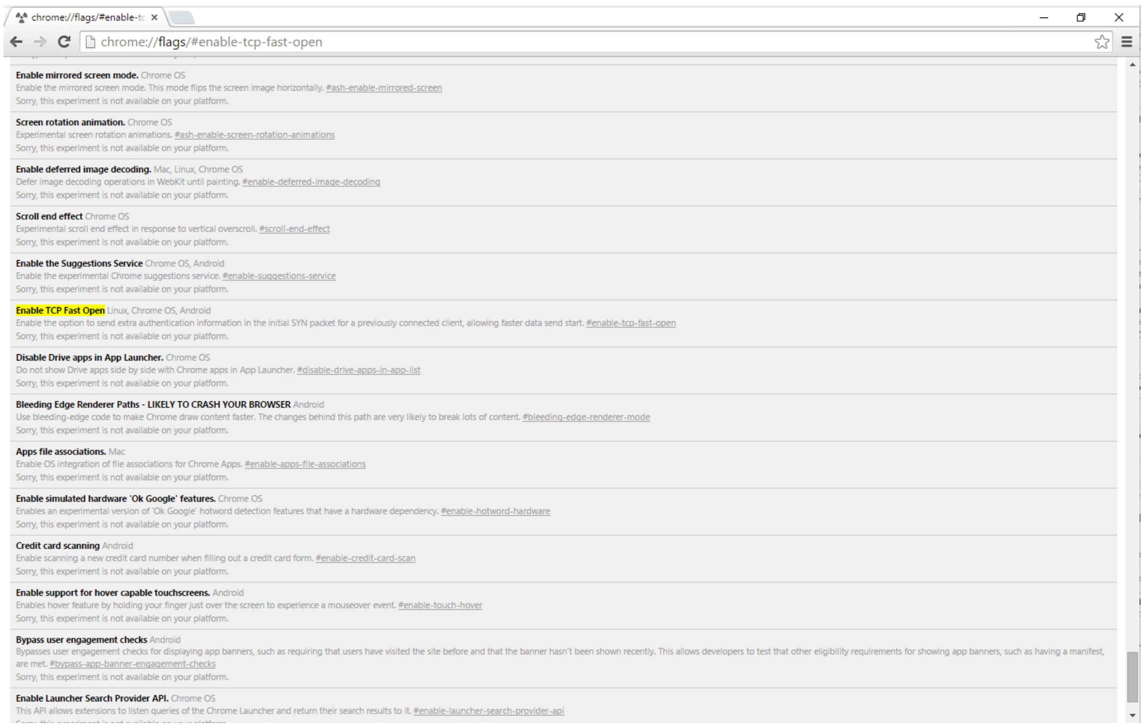
Preference Name	Status	Type	Value
dom.presentation.tcp_server.debug	default	boolean	false
media.peerconnection.ice.tcp	default	boolean	true
media.peerconnection.ice.tcp_so_sock_count	default	integer	0
network.http.tcp_keepalive.long_lived_connections	default	boolean	true
network.http.tcp_keepalive.long_lived_idle_time	default	integer	600
network.http.tcp_keepalive.short_lived_connections	default	boolean	true
network.http.tcp_keepalive.short_lived_idle_time	default	integer	10
network.http.tcp_keepalive.short_lived_time	default	integer	60
network.tcp.keepalive.enabled	default	boolean	true
network.tcp.keepalive.idle_time	default	integer	600
network.tcp.keepalive.retry_interval	default	integer	1
network.tcp.tcp_fastopen_consecutive_failure_limit	default	integer	5
network.tcp.tcp_fastopen_enable	default	boolean	false

Mozilla Firefox version 58 TCP Fast Open option

Moving on to the Google Chrome browser, similar procedure has to be followed by writing on the address bar the URL:

- chrome://flags

Afterwards, a list with the developer options appears and the Enable TCP Fast Open option can be enabled as shown in the image bellow. In this case it is shown that the feature is not available due to the current situation on the mechanism as described earlier.



chrome://flags/#enable-tc

chrome://flags/#enable-tcp-fast-open

Enable mirrored screen mode. Chrome OS
Enable the mirrored screen mode. This mode flips the screen image horizontally. #ash-enable-mirrored-screen
Sorry, this experiment is not available on your platform.

Screen rotation animation. Chrome OS
Experimental screen rotation animations. #ash-enable-screen-rotation-animations
Sorry, this experiment is not available on your platform.

Enable deferred image decoding. Mac, Linux, Chrome OS
Defer image decoding operations in WebKit until painting. #enable-deferred-image-decoding
Sorry, this experiment is not available on your platform.

Scroll end effect Chrome OS
Experimental scroll end effect in response to vertical overscroll. #scroll-end-effect
Sorry, this experiment is not available on your platform.

Enable the Suggestions Service Chrome OS, Android
Enable the experimental Chrome suggestions service. #enable-suggestions-service
Sorry, this experiment is not available on your platform.

Enable TCP Fast Open Linux, Chrome OS, Android
Enable the option to send extra authentication information in the initial SYN packet for a previously connected client, allowing faster data send start. #enable-tcp-fast-open
Sorry, this experiment is not available on your platform.

Disable Drive apps in App Launcher. Chrome OS
Do not show Drive apps side by side with Chrome apps in App Launcher. #disable-drive-apps-in-app-list
Sorry, this experiment is not available on your platform.

Bleeding Edge Renderer Paths - LIKELY TO CRASH YOUR BROWSER Android
Use bleeding-edge code to make Chrome draw content faster. The changes behind this path are very likely to break lots of content. #bleeding-edge-renderer-mode
Sorry, this experiment is not available on your platform.

Apps file associations. Mac
Enable OS integration of file associations for Chrome Apps. #enable-apps-file-associations
Sorry, this experiment is not available on your platform.

Enable simulated hardware 'Ok Google' features. Chrome OS
Enables an experimental version of 'Ok Google' hotword detection features that have a hardware dependency. #enable-hotword-hardware
Sorry, this experiment is not available on your platform.

Credit card scanning Android
Enable scanning a new credit card number when filling out a credit card form. #enable-credit-card-scan
Sorry, this experiment is not available on your platform.

Enable support for hover capable touchscreens. Android
Enables hover feature by holding your finger just over the screen to experience a mouseover event. #enable-touch-hover
Sorry, this experiment is not available on your platform.

Bypass user engagement checks Android
Bypasses user engagement checks for displaying app banners, such as requiring that users have visited the site before and that the banner hasn't been shown recently. This allows developers to test that other eligibility requirements for showing app banners, such as having a manifest, are met. #bypass-app-banner-engagement-checks
Sorry, this experiment is not available on your platform.

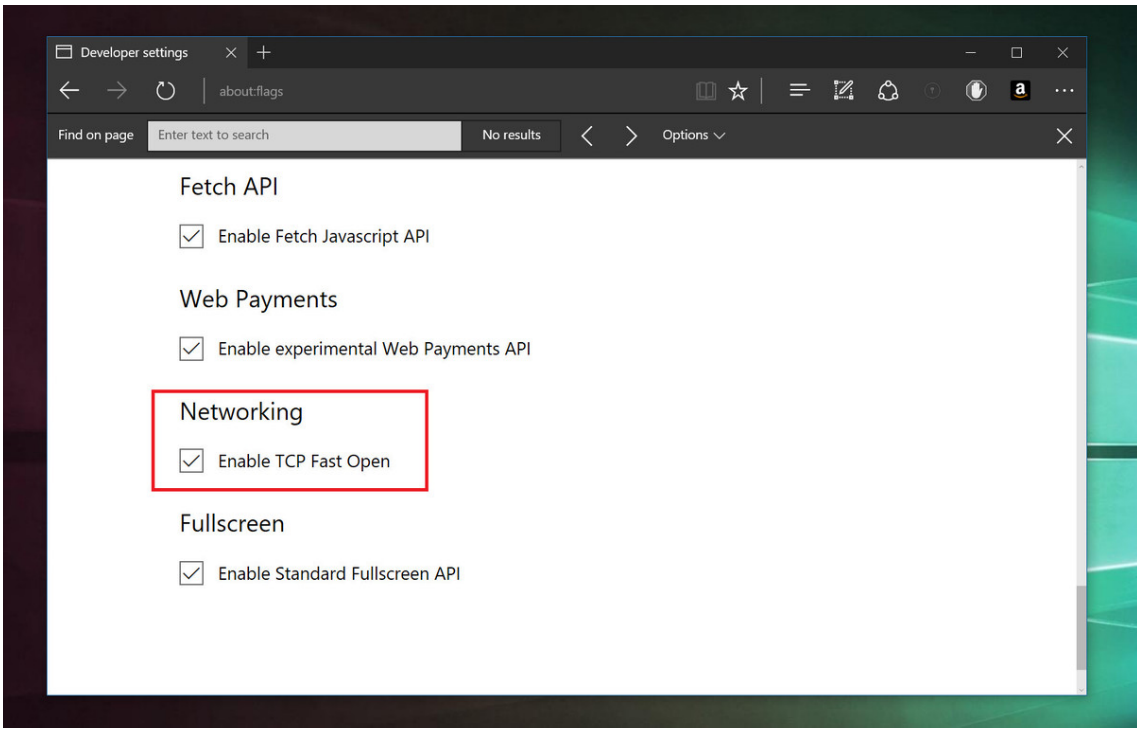
Enable Launcher Search Provider API. Chrome OS
This API allows extensions to listen queries of the Chrome Launcher and return their search results to it. #enable-launcher-search-provider-api

Google Chrome enable TCP Fast Open option

And last but not least, on the Microsoft Edge browser similarly a procedure has to be followed by writing on the address bar the URL:

- about:flags

Again, the option "Enable TCP Fast Open" has to be enabled from the list and under the section of "Networking" as it is shown in the image bellow. It is important to know that this version of Microsoft Edge is not based on the Chromium engine which is the only one available currently.



Fetch API

☑ Enable Fetch Javascript API

Web Payments

☑ Enable experimental Web Payments API

Networking

☑ Enable TCP Fast Open

Fullscreen

☑ Enable Standard Fullscreen API

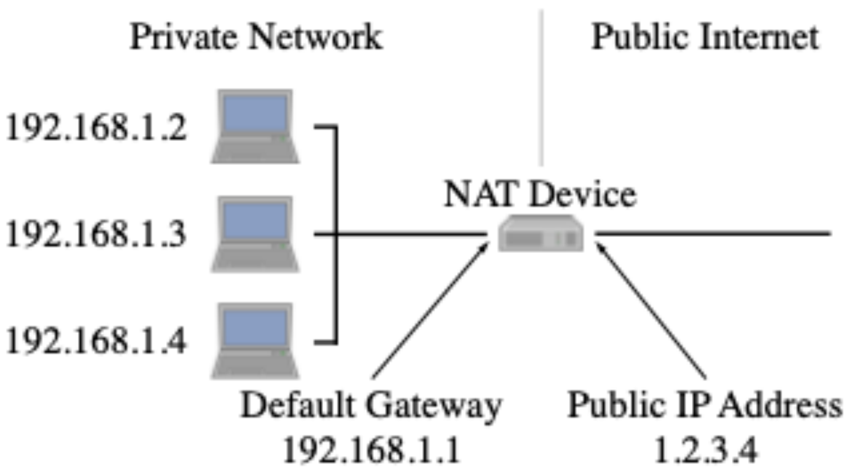
Microsoft Edge Enable TCP Fast Open

3.4 Security and Privacy Concerns

When talking about a new technology or an improvement on a current protocol, performance and support are not the only metrics to evaluate and come to conclusions. Security evaluations and privacy concerns are also important and since this mechanism has the target to improve the performance of the most popular protocol used on the web today it is only fitted that a discussion has to be done. Security and privacy of this mechanism is, among others, one of the main reasons that the TFO has not been widely supported over the years and these concerns are not only described on the original paper but also in other studies from other parties over the years.

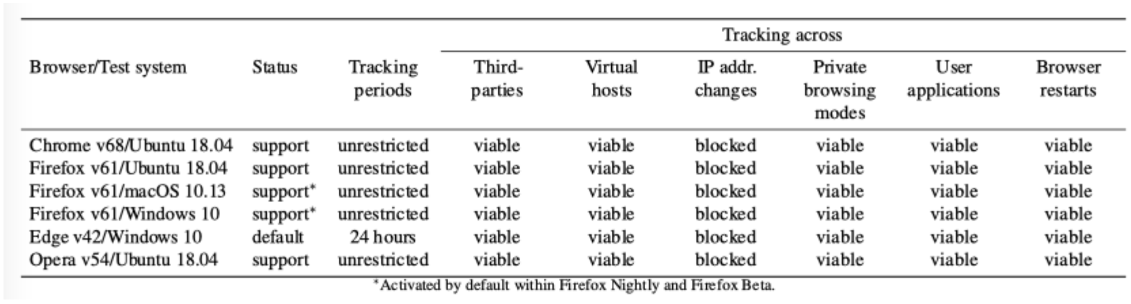
In 2019 Erik Sy did a presentation talking about the privacy concerns about the TCP Fast Open mechanism as a part of a joint written paper. In this presentation, he described the weaknesses of the TFO which shows very interesting results. One of the first findings is that the TFO cookies generated by the mechanism are a kernel-based tracking method. Even though, traditionally from the client side, when browsing the web many privacy issues are handled by the browser, the operating system has a separate tracking mechanism using this cookie. This also means that even in private browsing or using other technics to avoid tracking on the browser level are not applicable. Also, if a man in the middle attack occurs, the network observer is able to identify and track the web usage by the user depending solely on the cookie present on the SYN packets. [26][27]

Network address translation also known as NAT is also a big concern. When multiple clients exist behind a NAT network and share a single public IP address the cookie is the same for all the clients and the only way to mitigate this by a carrier level techniques used today like changing the public IP of the client network. [14] But always possible to protect against tracking, there is a way to distinguish a client among others behind a NAT network when the tracker provides each device with a unique TFO cookie against other clients thus eliminating the advantage of the anonymity behind such a network. [27]



NAT device used for a private network [27]

The operating systems, that as we mentioned before are responsible for the cookie storing and managing, have the possibility of preventing privacy issues by clearing the cookie cache on each restart. But this depends on the configuration and it is not standard, as shown below by the tracking period available before clearing the cache. [27]



Browser/Test system	Status	Tracking periods	Tracking across					
			Third-parties	Virtual hosts	IP addr. changes	Private browsing modes	User applications	Browser restarts
Chrome v68/Ubuntu 18.04	support	unrestricted	viable	viable	blocked	viable	viable	viable
Firefox v61/Ubuntu 18.04	support	unrestricted	viable	viable	blocked	viable	viable	viable
Firefox v61/macOS 10.13	support*	unrestricted	viable	viable	blocked	viable	viable	viable
Firefox v61/Windows 10	support*	unrestricted	viable	viable	blocked	viable	viable	viable
Edge v42/Windows 10	default	24 hours	viable	viable	blocked	viable	viable	viable
Opera v54/Ubuntu 18.04	support	unrestricted	viable	viable	blocked	viable	viable	viable

*Activated by default within Firefox Nightly and Firefox Beta.

Tracking table [27]

There are many other scenarios for tracking the user using the TFO cookie. The attacker can target a specific website and link all the activities performed by a user using his identifier, this cookie. To that extend he can target multiple websites and link the same user so he can create a browsing history. This is particularly effective when a single server hosts many domains virtually. [27]

As mentioned previously, the cookie is maintained at the system level, so even a private browsing window will have no effect on identifying what the user is visiting when investigating the local machine.

Privacy issues by the cookie transaction are not the only concern that has been created by the research world regarding the TFO mechanism. The TCP Fast Open option is also vulnerable to security attacks and flaws which are also mentioned in the first paper published. The method of SYN flooding also applies in the TFO mechanism in which the attacker can flood the server with SYN packets containing valid cookies forcing an exhaust in resources even though, as mentions previously, the cookie authentication process puts very low overhead on the CPU. In the case that the attacker does not use valid cookie the server can be defended using existing techniques due to the fall back on the traditional three-way handshake. As a mitigation technique, it is suggested to use a cache which counts the TFO connections accepted but not yet been migrated to full TCP state which is only feasible after the first ACK. But this method, which is configured by the administrator, will force the server to disable TFO for all incoming connections. [14]

Another attack that could target the TFO mechanism is the amplified reflection attack. While normally the traditional TCP transaction of data could happen only after one SYN-ACK packet the TFO lets the server send data packets following the SYN-ACK to the source IP address of the SYN packet. The TFO mechanism mitigates the attack by limiting the amount of the connections the server accepts so it is protected when this limit is reached. Also the attacker has to steal a valid cookie which indicated a compromised system and therefore the attacker will probably have little interest in the TCP connection. But he can still leverage the attack in order to disrupt the targeted network but only after he obtains multiple cookies from many server destinations. [14]

As discussed, although the security issues are mainly as present as the traditional TCP mechanism, the TFO introduces many new privacy concerns which are the core reason for the mechanism not to be widely implemented. This led, over the years the industry to focus on new technologies in order to achieve the high privacy level and performance gains which are going to be presented in the end of this thesis.

4 Experimental

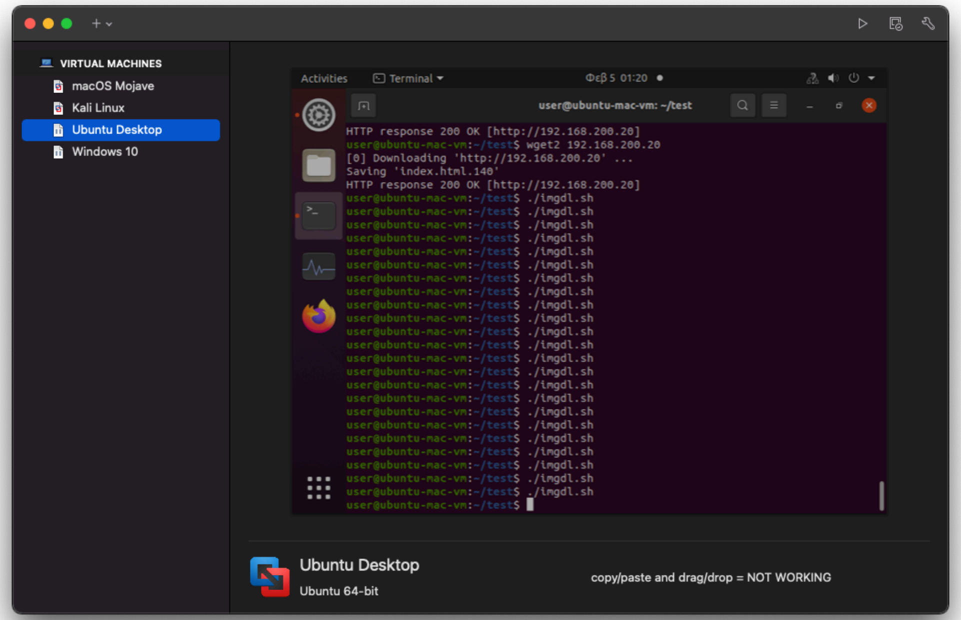
4.1 Testing Environment and Preparation

The scope of this testbench is to see in practice the performance gains of the TFO option when loading a web page. Tests will be conducted with and without the option by implementing different scenarios and comparing the results.

The testing will be conducted on a local environment using personal computers as a client and server emulation. Using two different physical machines is important in order to simulate the scenarios as close as possible to the real world taking into consideration also the medium.

For the purpose of the experiment a lab environment was built, and certain hardware and software choices were made taking into important consideration facts such as the scalability, ease of troubleshooting, easy recovery and restore in case of failed configuration testing, network reliability, elimination of hardware limitations with virtualization and software reliability. As the main Operating Systems throughout the testing, Linux Ubuntu 20.04 LTS Desktop was selected and installed in both the client and the server.

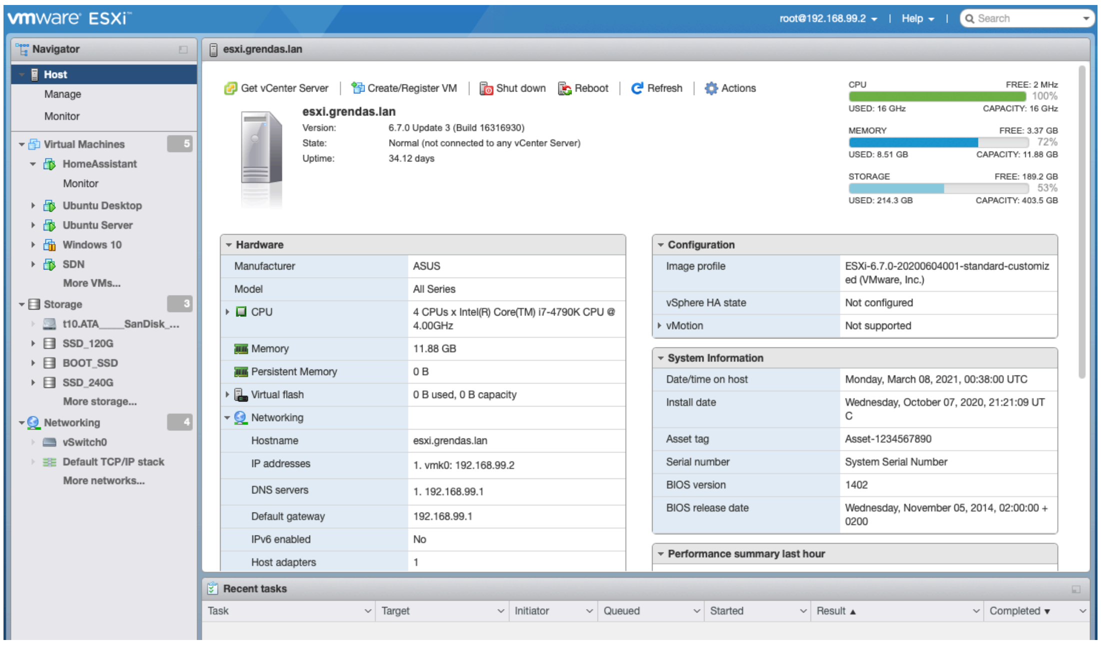
As a client machine Linux Ubuntu was installed in a type 2 virtualization software VMware Fusion running on a MacOS operating system. Since the host machine is equipped with two network interfaces, passthrough option was selected giving full hardware control of the ethernet interface to the virtual machine and thus eliminating any latency or issues by sharing this resource with the host system. This ethernet interface will be used by the Linux OS in order to perform all the testing scenarios.



```
HTTP response 200 OK [http://192.168.200.20]
user@ubuntu-mac-vm:~/test$ wget2 192.168.200.20
[0] Downloading 'http://192.168.200.20' ...
Saving 'index.html.140'
HTTP response 200 OK [http://192.168.200.20]
user@ubuntu-mac-vm:~/test$ ./imgdl.sh
user@ubuntu-mac-vm:~/test$ ./imgdl.sh
user@ubuntu-mac-vm:~/test$ ./imgdl.sh
user@ubuntu-mac-vm:~/test$ ./imgdl.sh
user@ubuntu-mac-vm:~/test$ ./imgdl.sh
user@ubuntu-mac-vm:~/test$ ./imgdl.sh
user@ubuntu-mac-vm:~/test$ ./imgdl.sh
user@ubuntu-mac-vm:~/test$ ./imgdl.sh
user@ubuntu-mac-vm:~/test$ ./imgdl.sh
user@ubuntu-mac-vm:~/test$ ./imgdl.sh
user@ubuntu-mac-vm:~/test$ ./imgdl.sh
user@ubuntu-mac-vm:~/test$ ./imgdl.sh
user@ubuntu-mac-vm:~/test$ ./imgdl.sh
user@ubuntu-mac-vm:~/test$ ./imgdl.sh
user@ubuntu-mac-vm:~/test$ ./imgdl.sh
user@ubuntu-mac-vm:~/test$ ./imgdl.sh
user@ubuntu-mac-vm:~/test$ ./imgdl.sh
user@ubuntu-mac-vm:~/test$ ./imgdl.sh
user@ubuntu-mac-vm:~/test$ ./imgdl.sh
user@ubuntu-mac-vm:~/test$ ./imgdl.sh
user@ubuntu-mac-vm:~/test$ ./imgdl.sh
user@ubuntu-mac-vm:~/test$ ./imgdl.sh
user@ubuntu-mac-vm:~/test$ ./imgdl.sh
user@ubuntu-mac-vm:~/test$ 
```

The client: VMware Fusion with Linux Ubuntu installed

As a server a more complicated approach was taken in order to give more choices later on. A physical computer was built up from the ground up deploying type 1 virtualization operating system, VMware ESXi. The hardware specifications of this machine are not important due to the very low resource requirements of the project. On the other hand there is one important hardware installed, the network card interface. After researching an enterprise Intel NIC (Network Interface Card) was installed in order to be passthrough to the Linux server virtual machine. Exactly like the client side, a Linux Ubuntu 20.04 LTS was installed as a virtual machine with the option to passthrough the Intel NIC.



esxi.grendas.lan

Version:	6.7.0 Update 3 (Build 16316930)
State:	Normal (not connected to any vCenter Server)
Uptime:	34.12 days

CPU FREE: 2 MHz 100% USED: 16 GHz CAPACITY: 16 GHz

MEMORY FREE: 3.37 GB 72% USED: 8.51 GB CAPACITY: 11.88 GB

STORAGE FREE: 189.2 GB 53% USED: 214.3 GB CAPACITY: 403.5 GB

Hardware

Manufacturer	ASUS
Model	All Series
CPU	4 CPUs x Intel(R) Core(TM) i7-4790K CPU @ 4.00GHz
Memory	11.88 GB
Persistent Memory	0 B
Virtual flash	0 B used, 0 B capacity
Networking	
Hostname	esxi.grendas.lan
IP addresses	1. vmk0: 192.168.99.2
DNS servers	1. 192.168.99.1
Default gateway	192.168.99.1
IPv6 enabled	No
Host adapters	1

Configuration

Image profile	ESXi-6.7.0-20200604001-standard-customized (VMware, Inc.)
vSphere HA state	Not configured
vMotion	Not supported

System Information

Date/time on host	Monday, March 08, 2021, 00:38:00 UTC
Install date	Wednesday, October 07, 2020, 21:21:09 UTC
Asset tag	Asset-1234567890
Serial number	System Serial Number
BIOS version	1402
BIOS release date	Wednesday, November 05, 2014, 02:00:00 +0200

Performance summary last hour

VMware ESXi Server

Additional hardware used are the network cable which is UTP Cat 5e RJ-45 and the physical middleboxes Edgerouter ER-X and Mikrotik hAP Lite which are used in some testing scenarios.

Physical Middleboxes: Edgerouter ER-X and Mikrotik hAP lite

A setup configured using these hardware and methods was selected also due to the fact that in real world, a web server usually is installed in a virtualized environment using Linux software. As with the rest of the setup, an effort was made for the test to be as accurate and close to the reality but operating in a local environment. Remote servers and other options were considered and discarded due to the need to have a controlled environment and also for simplicity reasons as this is not a deep scientific analysis.

In order to perform the testing some configuration and installation of software is needed on both the client and the server. All the necessary configurations were made using the terminal through a GUI (Graphical User Interface). As mentioned in a previous chapter, since the TFO is implemented natively on the Linux kernel, in order for the option to be used by an application, no further package installation is required. However, the OS has the option to configure the TFO on the system by altering a system file as follows:

- "echo 0 > /proc/sys/net/ipv4/tcp_fastopen" = disable TFO
- "echo 1 > /proc/sys/net/ipv4/tcp_fastopen" = enable TFO for outgoing connections (client)
- "echo 2 > /proc/sys/net/ipv4/tcp_fastopen" = enable TFO for listening sockets (server)
- "echo 3 > /proc/sys/net/ipv4/tcp_fastopen" = enable TFO for both outgoing and incoming connections (client/server)

In order to check the TFO status on the system the following command is used:

- "cat /proc/sys/net/ipv4/tcp_fastopen" : depending on the number printed

Depending on the scenario this option is altered, for example if the testing is with the TFO disabled the appropriate configuration needs to be performed followed by a system restart. The restarts were performed because sometimes the configuration was not applied successfully after numerous alterations. In addition to that configuration, the client was configured to use the wget2 package for all the testing scenarios. After testing numerous tools along with browsers this tool was selected eventually due to the configuration abilities it has and also excellent TFO support. Loading the web pages with this tool made it easier to test fast and many times by using it with a simple script and also eliminated any browser-specific limitation or latency giving also full control on the requests.

At the server side of the testing environment the same configuration has to be performed with the TFO option by altering the system file and enabling the appropriate option for making sure the TFO option is enabled for listening sockets. The main configuration of the server is the web server application which is going to serve the web pages to the client. As a server application NGINX was selected due to the configuration options offered, it is light and fast and also widely used on the web today. As for Nginx TFO support, most Nginx packages do not include this by default. However, it can be specified if you build Nginx from source by adding the appropriate compiler flag to NGINX's configure script. The procedure followed as follows:

- Prior compiling NGINX from source some libraries needed to be installed
- NGINX source code should be pulled from the official source
- Adding the -DTCP_FASTOPEN=23 compiler flag
- Building the NGINX and testing the operation

In order for the testing scenarios described later to be conducted, some images are also downloaded and saved to the folder that NGINX uses to store the content for the clients. At this point there is no other configuration needed for the server to operate as a web-server but later we will need to make some network interface configuration in order to apply to some testing scenarios.

For capturing and measuring the times a page finishes downloading we will use the popular Wireshark application which will be installed on the client side capturing the ethernet interface. All the captures of the testing scenarios are saved separately as .pcap files and the measurements are recorded in a Microsoft Excel document in order to be presented later.

The scenarios selected to be implemented were three and for each of them two different types of webpage simulations were performed with both TFO disabled and enabled. Firstly, tests were performed in a client-server model directly connected to each other with an ethernet cable. Then, a middlebox was added in order to perform the same tests and see if there an impact on either the performance or the functionality of the TFO mechanism. Lastly, a firewall was implemented to perform again the same tests in order to see if there is an impact on a stateful inspection firewall both on the client-server and also to the functionality of the middlebox itself. The first type of webpage was a simple HTML text with one TCP connection for each test. The second type, was multiple images of different sizes creating a small script in order to perform multiple TCP connections for each test, providing a closer scenario to today's modern webpages. The amount of tests repetition for each case was ten times and the same for all the scenarios, in order to have more consistent results based on the same number of measurements.

In order to present the result numbers, multiple tables were created presenting in detail the measurements for each connection in seconds. Then additional graphs were created in order to better present the RTT times visually and help understand better the benefits of this mechanism. Lastly, an average number was calculated for each scenario and test type in order to have conclusive results about the performance gains. The calculation for the percentage increase was calculated with the following simple mathematical form:

- Increased Time = TFO Disabled – TFO Enabled
    - Increase Percentage = Increased Time / TFO Disabled x 100

For each case, the performance gains will be presented using all the methods described and a conclusion will be made for each scenario. Also, if any, malfunctions of the mechanism and testbench will be presented.

4.2 Testing with direct connection

The first series of tests were designed to be as simple as possible with a direct connection to the server and without any middleboxes. For this reason, some configuration is needed for the client and the server in order to communicate. Static IP addresses of the same subnet were used in on both machines using the addresses:

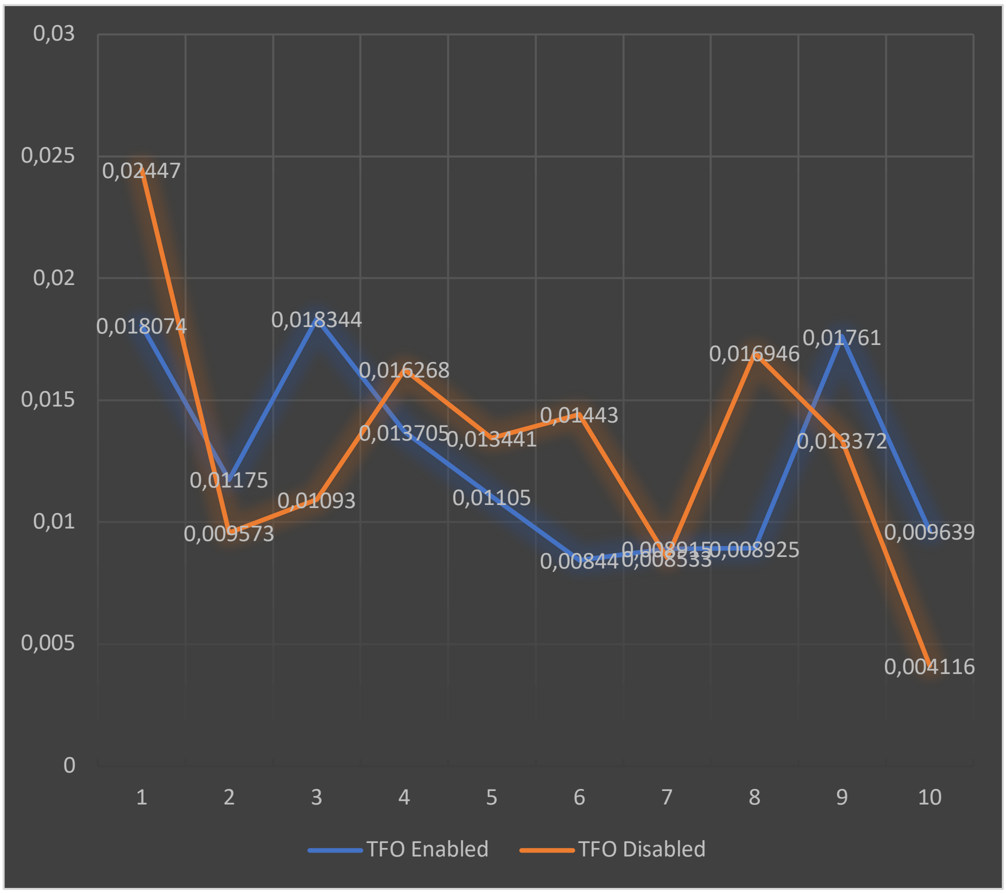
- 192.168.200.20/24 for the server
- 192.168.200.12/24 for the client

In the following table, the results are shown in detail for each testing:

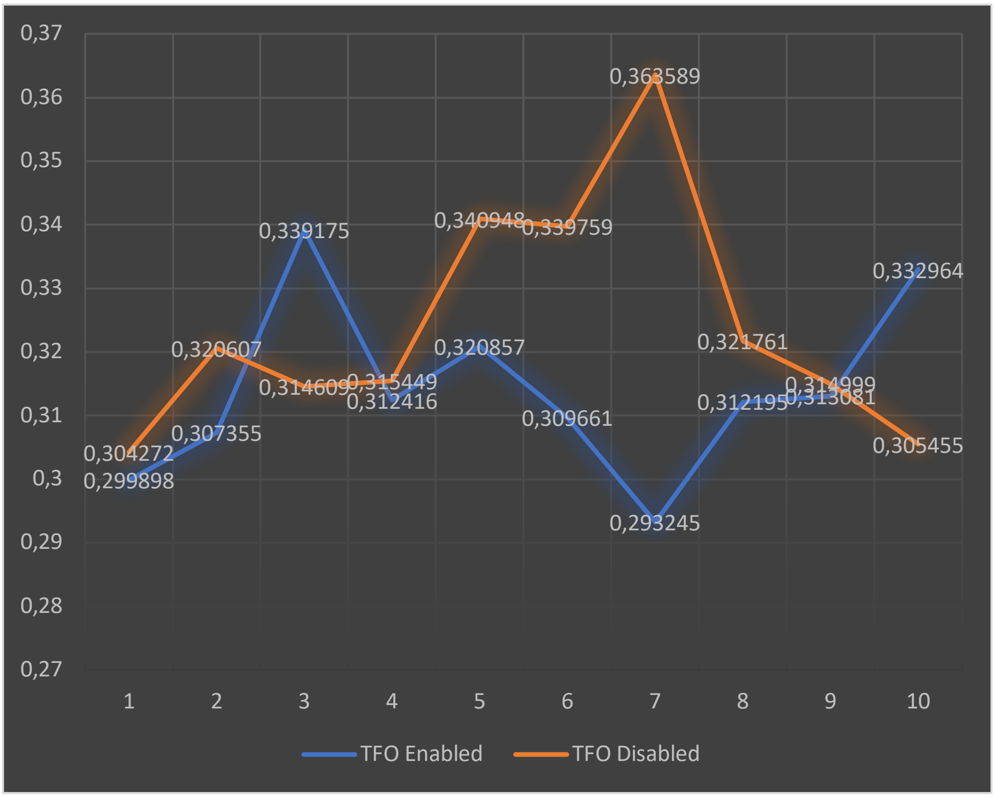
Test No.	TFO disabled HTML with text	TFO enabled HTML with text	TFO disabled HTML with images	TFO enabled HTML with images
1	0,02447	0,018074	0,304272	0,299898
2	0,009573	0,01175	0,320607	0,307355
3	0,01093	0,018344	0,314609	0,339175
4	0,016268	0,013705	0,315449	0,312416
5	0,013441	0,01105	0,340948	0,320857
6	0,01443	0,00844	0,339759	0,309661
7	0,008533	0,008915	0,363589	0,293245
8	0,016946	0,008925	0,321761	0,312195
9	0,013372	0,01761	0,314999	0,313081
10	0,004116	0,009639	0,305455	0,332964

Test results: Total RTT in seconds

In order to see the results on a chart and understand easier the performance gains some charts were created as shown in the following images:



Performance chart for HTML website containing text



Performance chart for HTML page containing multiple images

The following table shows the average RTT numbers and the performance gains in percentage calculated by the mathematical formula mentioned previously.

Type	TFO Disabled	TFO Enabled	RTT Time Decrease
Text	0,0132079	0,0126452	4,45 %
Images	0,3241448	0,3140847	3,2 %

Performance gains table

4.3 Testing with middlebox

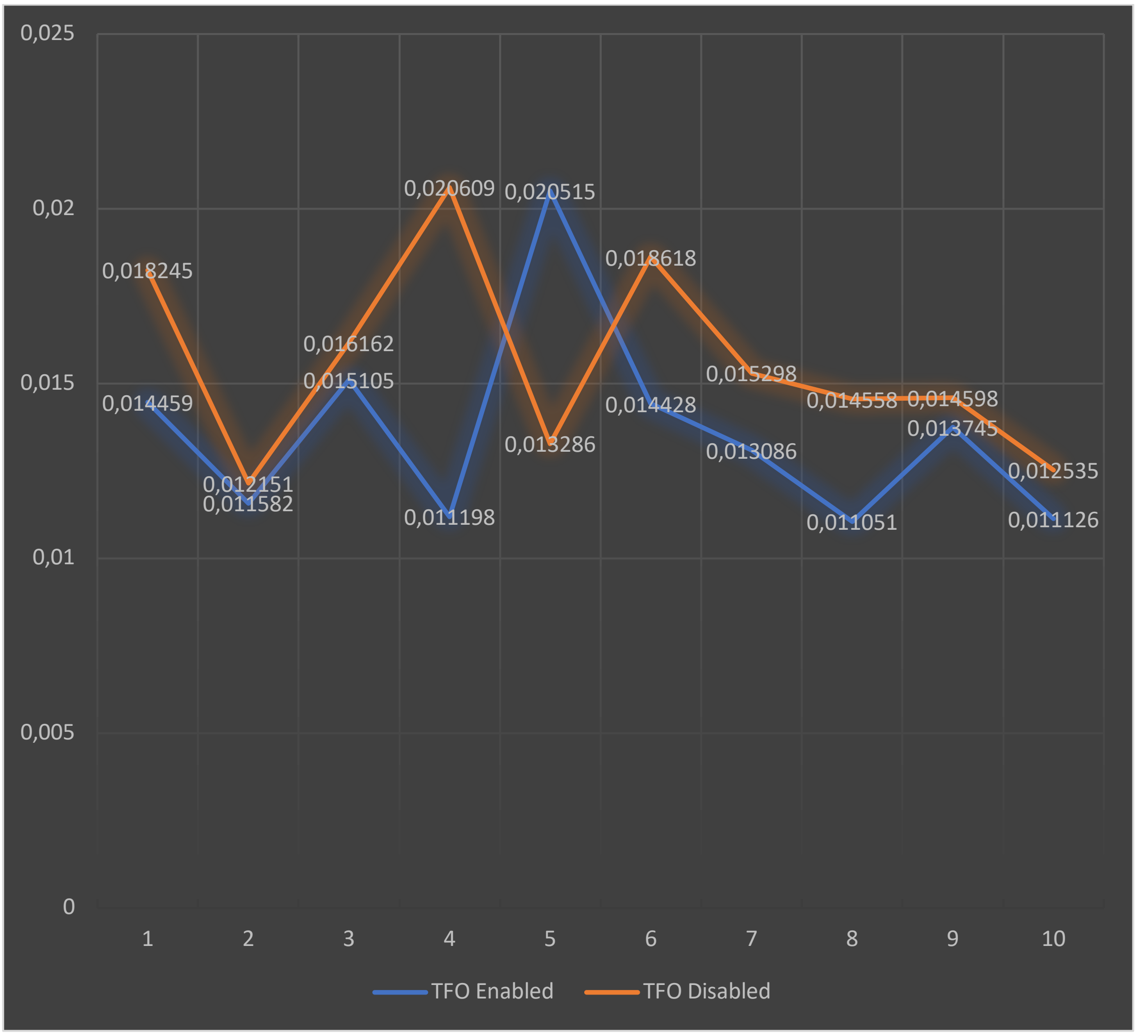
The next series of testing conducted using a middlebox in between the client and the server in order to test not only the latency introduced by the processing of the packets but also any effects on the TCP Fast Open option.

Bellow it is shown in detail each test with the corresponding results:

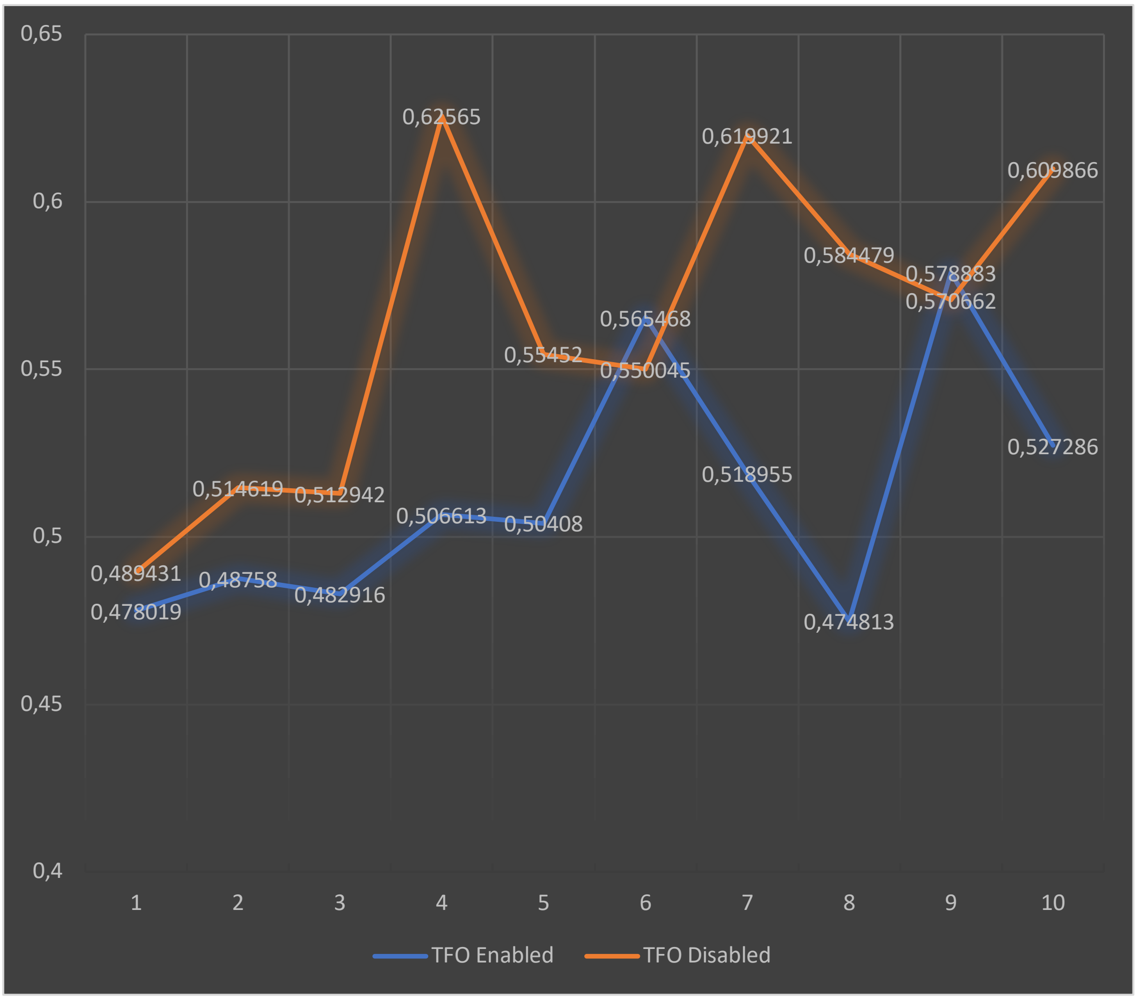
Test No.	TFO disabled HTML with text	TFO enabled HTML with text	TFO disabled HTML with images	TFO enabled HTML with images
1	0,018245	0,014459	0,489431	0,478019
2	0,012151	0,011582	0,514619	0,48758
3	0,016162	0,015105	0,512942	0,482916
4	0,020609	0,011198	0,62565	0,506613
5	0,013286	0,020515	0,55452	0,50408
6	0,018618	0,014428	0,550045	0,565468
7	0,015298	0,013086	0,619921	0,518955
8	0,014558	0,011051	0,584479	0,474813
9	0,014598	0,013745	0,570662	0,578883
10	0,012535	0,011126	0,609866	0,527286

Test results: Total RTT in seconds

Similarly, the performance charts were created in order to have a visual presentation of the performance throughout the testing as shown in the following images.



Performance chart when loading HTML webpage with text



Performance chart when loading HTML webpage with multiple images

The following table is the summary created by averaging the RTT times and calculating the performance gains in percentage.

Type	TFO Disabled	TFO Enabled	RTT Time Decrease
Text	0,015606	0,0136295	14,5 %
Images	0,5632135	0,5124613	9,9 %

Performance gains table

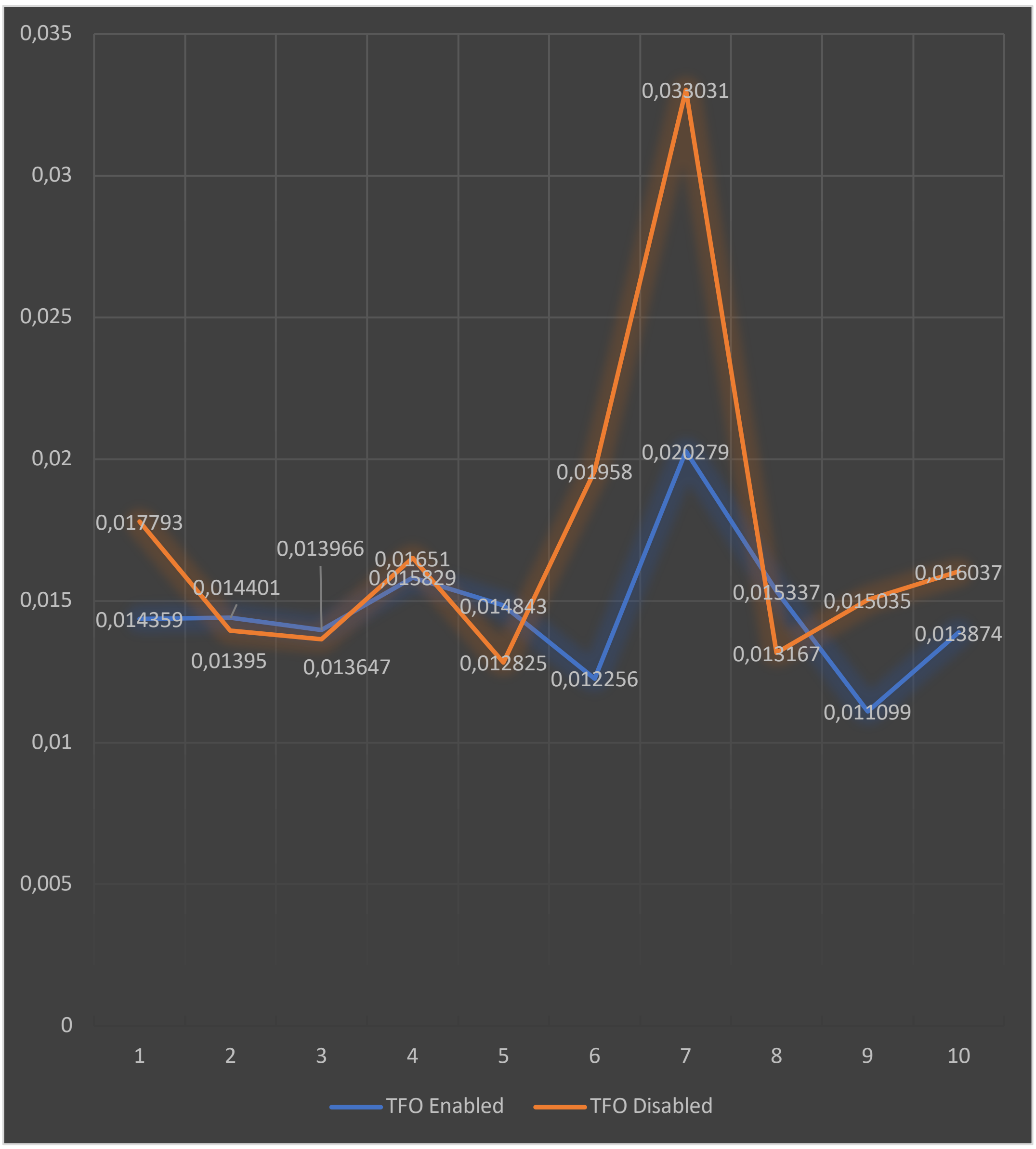
4.4 Testing with middlebox and firewall

In the last case scenario, along with the middlebox, a firewall was enabled in order to see the performance effects when an additional layer is involved during the network traffic transaction between the client and server. The router Edgerouter X was used and a simple firewall rule was created in order to have a basic filtering of the packets passing through the middlebox.

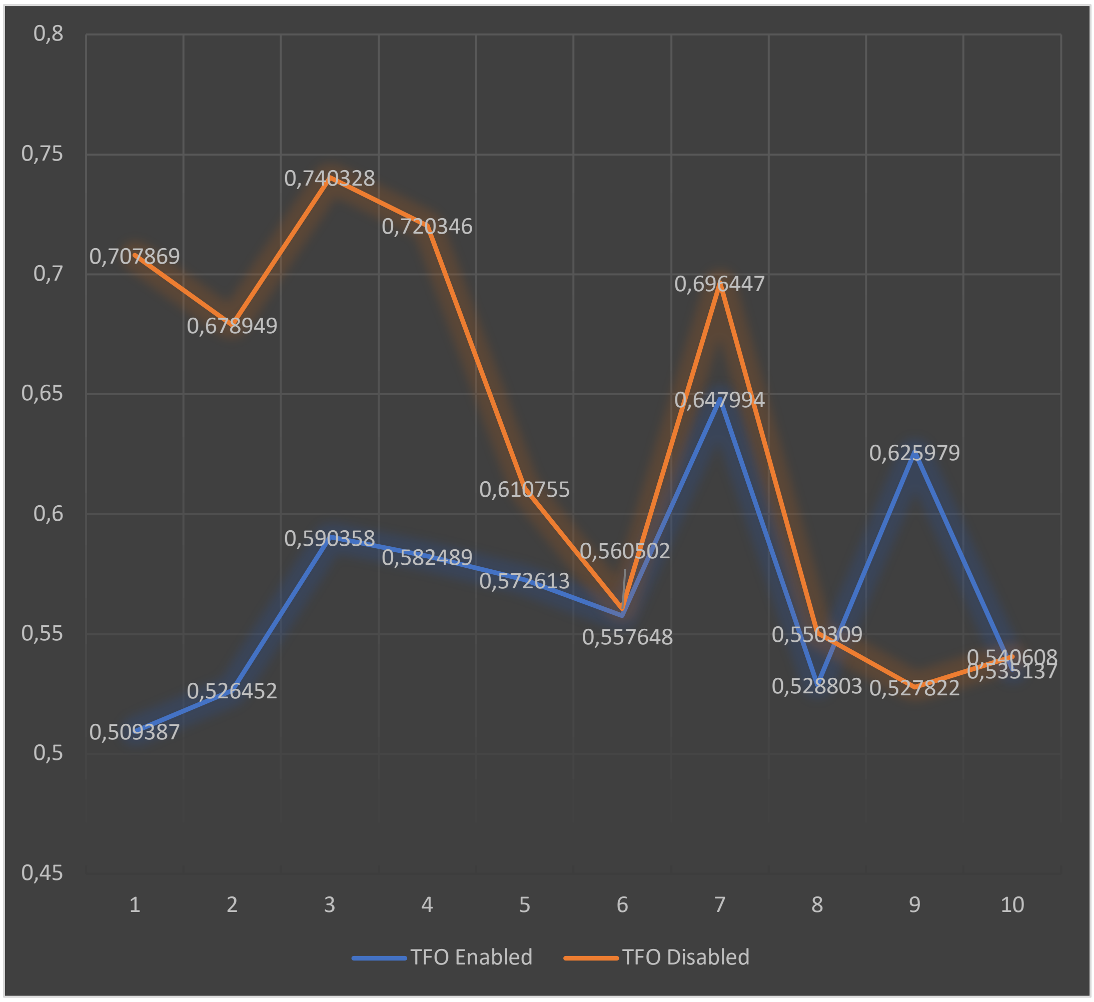
Bellow the analytical table is presented with each test number:

Test No.	TFO disabled HTML with text	TFO enabled HTML with text	TFO disabled HTML with images	TFO enabled HTML with images
1	0,017793	0,014359	0,707869	0,509387
2	0,01395	0,014401	0,678949	0,526452
3	0,013647	0,013966	0,740328	0,590358
4	0,01651	0,015829	0,720346	0,582489
5	0,012825	0,014843	0,610755	0,572613
6	0,01958	0,012256	0,560502	0,557648
7	0,033031	0,020279	0,696447	0,647994
8	0,013167	0,015337	0,550309	0,528803
9	0,015035	0,011099	0,527822	0,625979
10	0,016037	0,013874	0,540608	0,535137

Test results: Total RTT in seconds



Performance chart for HTML website containing text



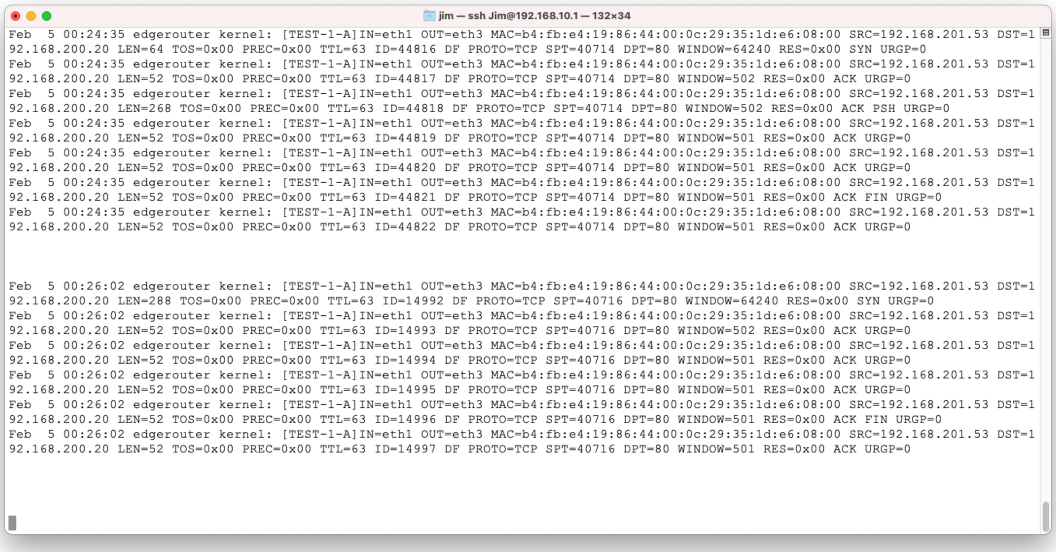
Performance chart for HTML website containing multiple images

Again the results are summarized and presented bellow after the percentage calculation in performance gains.

Type	TFO Disabled	TFO Enabled	RTT Time Decrease
Text	0,0171575	0,0146243	17,32 %
Images	0,6333935	0,567686	11,57 %

Performance gains table

Regarding the firewall rules, a terminal was open during the testing in order to check if the firewall rules were working properly on the Edgerouter X router presented on the images bellow.

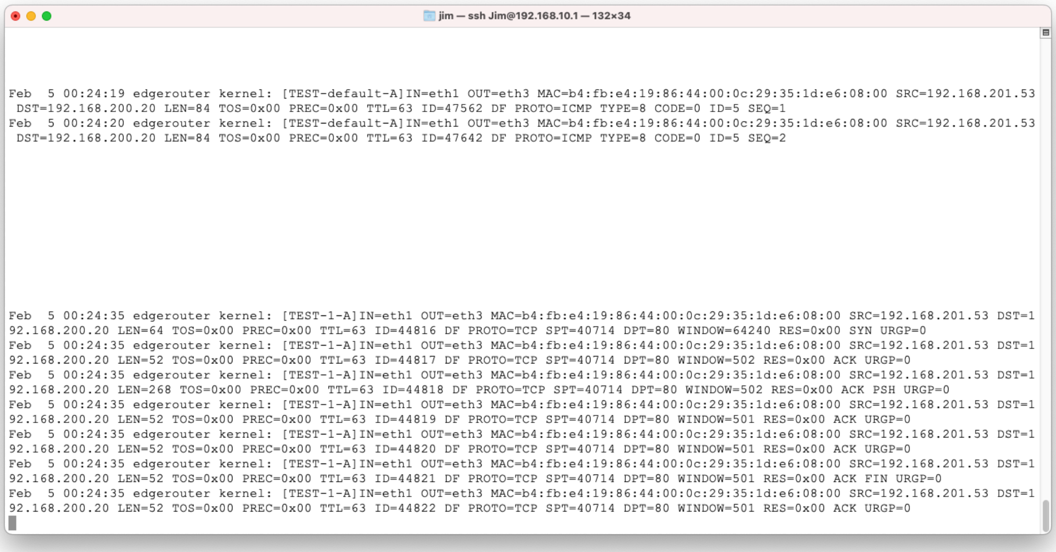

jim — ssh Jim@192.168.10.1 — 132×34

```
Feb  5 00:24:35 edgerouter kernel: [TEST-1-A]IN=eth1 OUT=eth3 MAC=b4:fb:e4:19:86:44:00:0c:29:35:1d:e6:08:00 SRC=192.168.201.53 DST=1
92.168.200.20 LEN=64 TOS=0x00 PREC=0x00 TTL=63 ID=44816 DF PROTO=TCP SPT=40714 DPT=80 WINDOW=64240 RES=0x00 SYN URGP=0
Feb  5 00:24:35 edgerouter kernel: [TEST-1-A]IN=eth1 OUT=eth3 MAC=b4:fb:e4:19:86:44:00:0c:29:35:1d:e6:08:00 SRC=192.168.201.53 DST=1
92.168.200.20 LEN=52 TOS=0x00 PREC=0x00 TTL=63 ID=44817 DF PROTO=TCP SPT=40714 DPT=80 WINDOW=502 RES=0x00 ACK URGP=0
Feb  5 00:24:35 edgerouter kernel: [TEST-1-A]IN=eth1 OUT=eth3 MAC=b4:fb:e4:19:86:44:00:0c:29:35:1d:e6:08:00 SRC=192.168.201.53 DST=1
92.168.200.20 LEN=268 TOS=0x00 PREC=0x00 TTL=63 ID=44818 DF PROTO=TCP SPT=40714 DPT=80 WINDOW=502 RES=0x00 ACK PSH URGP=0
Feb  5 00:24:35 edgerouter kernel: [TEST-1-A]IN=eth1 OUT=eth3 MAC=b4:fb:e4:19:86:44:00:0c:29:35:1d:e6:08:00 SRC=192.168.201.53 DST=1
92.168.200.20 LEN=52 TOS=0x00 PREC=0x00 TTL=63 ID=44819 DF PROTO=TCP SPT=40714 DPT=80 WINDOW=501 RES=0x00 ACK URGP=0
Feb  5 00:24:35 edgerouter kernel: [TEST-1-A]IN=eth1 OUT=eth3 MAC=b4:fb:e4:19:86:44:00:0c:29:35:1d:e6:08:00 SRC=192.168.201.53 DST=1
92.168.200.20 LEN=52 TOS=0x00 PREC=0x00 TTL=63 ID=44820 DF PROTO=TCP SPT=40714 DPT=80 WINDOW=501 RES=0x00 ACK URGP=0
Feb  5 00:24:35 edgerouter kernel: [TEST-1-A]IN=eth1 OUT=eth3 MAC=b4:fb:e4:19:86:44:00:0c:29:35:1d:e6:08:00 SRC=192.168.201.53 DST=1
92.168.200.20 LEN=52 TOS=0x00 PREC=0x00 TTL=63 ID=44821 DF PROTO=TCP SPT=40714 DPT=80 WINDOW=501 RES=0x00 ACK FIN URGP=0
Feb  5 00:24:35 edgerouter kernel: [TEST-1-A]IN=eth1 OUT=eth3 MAC=b4:fb:e4:19:86:44:00:0c:29:35:1d:e6:08:00 SRC=192.168.201.53 DST=1
92.168.200.20 LEN=52 TOS=0x00 PREC=0x00 TTL=63 ID=44822 DF PROTO=TCP SPT=40714 DPT=80 WINDOW=501 RES=0x00 ACK URGP=0



Feb  5 00:26:02 edgerouter kernel: [TEST-1-A]IN=eth1 OUT=eth3 MAC=b4:fb:e4:19:86:44:00:0c:29:35:1d:e6:08:00 SRC=192.168.201.53 DST=1
92.168.200.20 LEN=288 TOS=0x00 PREC=0x00 TTL=63 ID=14992 DF PROTO=TCP SPT=40716 DPT=80 WINDOW=64240 RES=0x00 SYN URGP=0
Feb  5 00:26:02 edgerouter kernel: [TEST-1-A]IN=eth1 OUT=eth3 MAC=b4:fb:e4:19:86:44:00:0c:29:35:1d:e6:08:00 SRC=192.168.201.53 DST=1
92.168.200.20 LEN=52 TOS=0x00 PREC=0x00 TTL=63 ID=14993 DF PROTO=TCP SPT=40716 DPT=80 WINDOW=502 RES=0x00 ACK URGP=0
Feb  5 00:26:02 edgerouter kernel: [TEST-1-A]IN=eth1 OUT=eth3 MAC=b4:fb:e4:19:86:44:00:0c:29:35:1d:e6:08:00 SRC=192.168.201.53 DST=1
92.168.200.20 LEN=52 TOS=0x00 PREC=0x00 TTL=63 ID=14994 DF PROTO=TCP SPT=40716 DPT=80 WINDOW=501 RES=0x00 ACK URGP=0
Feb  5 00:26:02 edgerouter kernel: [TEST-1-A]IN=eth1 OUT=eth3 MAC=b4:fb:e4:19:86:44:00:0c:29:35:1d:e6:08:00 SRC=192.168.201.53 DST=1
92.168.200.20 LEN=52 TOS=0x00 PREC=0x00 TTL=63 ID=14995 DF PROTO=TCP SPT=40716 DPT=80 WINDOW=501 RES=0x00 ACK URGP=0
Feb  5 00:26:02 edgerouter kernel: [TEST-1-A]IN=eth1 OUT=eth3 MAC=b4:fb:e4:19:86:44:00:0c:29:35:1d:e6:08:00 SRC=192.168.201.53 DST=1
92.168.200.20 LEN=52 TOS=0x00 PREC=0x00 TTL=63 ID=14996 DF PROTO=TCP SPT=40716 DPT=80 WINDOW=501 RES=0x00 ACK FIN URGP=0
Feb  5 00:26:02 edgerouter kernel: [TEST-1-A]IN=eth1 OUT=eth3 MAC=b4:fb:e4:19:86:44:00:0c:29:35:1d:e6:08:00 SRC=192.168.201.53 DST=1
92.168.200.20 LEN=52 TOS=0x00 PREC=0x00 TTL=63 ID=14997 DF PROTO=TCP SPT=40716 DPT=80 WINDOW=501 RES=0x00 ACK URGP=0
```

Firewall filtering on the Edgerouter X router


jim — ssh Jim@192.168.10.1 — 132×34

```
Feb  5 00:24:19 edgerouter kernel: [TEST-default-A]IN=eth1 OUT=eth3 MAC=b4:fb:e4:19:86:44:00:0c:29:35:1d:e6:08:00 SRC=192.168.201.53
 DST=192.168.200.20 LEN=84 TOS=0x00 PREC=0x00 TTL=63 ID=47562 DF PROTO=ICMP TYPE=8 CODE=0 ID=5 SEQ=1
Feb  5 00:24:20 edgerouter kernel: [TEST-default-A]IN=eth1 OUT=eth3 MAC=b4:fb:e4:19:86:44:00:0c:29:35:1d:e6:08:00 SRC=192.168.201.53
 DST=192.168.200.20 LEN=84 TOS=0x00 PREC=0x00 TTL=63 ID=47642 DF PROTO=ICMP TYPE=8 CODE=0 ID=5 SEQ=2









Feb  5 00:24:35 edgerouter kernel: [TEST-1-A]IN=eth1 OUT=eth3 MAC=b4:fb:e4:19:86:44:00:0c:29:35:1d:e6:08:00 SRC=192.168.201.53 DST=1
92.168.200.20 LEN=64 TOS=0x00 PREC=0x00 TTL=63 ID=44816 DF PROTO=TCP SPT=40714 DPT=80 WINDOW=64240 RES=0x00 SYN URGP=0
Feb  5 00:24:35 edgerouter kernel: [TEST-1-A]IN=eth1 OUT=eth3 MAC=b4:fb:e4:19:86:44:00:0c:29:35:1d:e6:08:00 SRC=192.168.201.53 DST=1
92.168.200.20 LEN=52 TOS=0x00 PREC=0x00 TTL=63 ID=44817 DF PROTO=TCP SPT=40714 DPT=80 WINDOW=502 RES=0x00 ACK URGP=0
Feb  5 00:24:35 edgerouter kernel: [TEST-1-A]IN=eth1 OUT=eth3 MAC=b4:fb:e4:19:86:44:00:0c:29:35:1d:e6:08:00 SRC=192.168.201.53 DST=1
92.168.200.20 LEN=268 TOS=0x00 PREC=0x00 TTL=63 ID=44818 DF PROTO=TCP SPT=40714 DPT=80 WINDOW=502 RES=0x00 ACK PSH URGP=0
Feb  5 00:24:35 edgerouter kernel: [TEST-1-A]IN=eth1 OUT=eth3 MAC=b4:fb:e4:19:86:44:00:0c:29:35:1d:e6:08:00 SRC=192.168.201.53 DST=1
92.168.200.20 LEN=52 TOS=0x00 PREC=0x00 TTL=63 ID=44819 DF PROTO=TCP SPT=40714 DPT=80 WINDOW=501 RES=0x00 ACK URGP=0
Feb  5 00:24:35 edgerouter kernel: [TEST-1-A]IN=eth1 OUT=eth3 MAC=b4:fb:e4:19:86:44:00:0c:29:35:1d:e6:08:00 SRC=192.168.201.53 DST=1
92.168.200.20 LEN=52 TOS=0x00 PREC=0x00 TTL=63 ID=44820 DF PROTO=TCP SPT=40714 DPT=80 WINDOW=501 RES=0x00 ACK URGP=0
Feb  5 00:24:35 edgerouter kernel: [TEST-1-A]IN=eth1 OUT=eth3 MAC=b4:fb:e4:19:86:44:00:0c:29:35:1d:e6:08:00 SRC=192.168.201.53 DST=1
92.168.200.20 LEN=52 TOS=0x00 PREC=0x00 TTL=63 ID=44821 DF PROTO=TCP SPT=40714 DPT=80 WINDOW=501 RES=0x00 ACK FIN URGP=0
Feb  5 00:24:35 edgerouter kernel: [TEST-1-A]IN=eth1 OUT=eth3 MAC=b4:fb:e4:19:86:44:00:0c:29:35:1d:e6:08:00 SRC=192.168.201.53 DST=1
92.168.200.20 LEN=52 TOS=0x00 PREC=0x00 TTL=63 ID=44822 DF PROTO=TCP SPT=40714 DPT=80 WINDOW=501 RES=0x00 ACK URGP=0
```

Firewall filtering on the Edgerouter X router

4.5 Results summary

The purpose of these tests were not to reproduce the initial measurements but to see in practice that even on the simplest scenarios there are significant performance gains. Also, although it is not taken into consideration any packet loss, the testing was conducted with no packet loss in any scenario. Also, it was not accounted for any random spikes latency due to the physical layer properties of the connection.

Seeing the final results, it is clear that even on this small number of local tests, the performance gains are visible. As it was analyzed previously the performance increases accordingly when the scenario or the RTT numbers changed while placing middleboxes and firewalls.

Starting with the first case scenario, a simple client and server were connected through a single gigabit cable without any middlebox in the middle. The tests showed a decrease in RTT of 4,45 % in a webpage containing text and 3.2% in a webpage containing images which is normal due to the simplicity of the setup.

The second scenario involved placing a simple switch in between the client and the server in order to examine the performance gains during this setup. Again, a clear performance gain is visible with webpages containing text and images showing improvements of 14,5% and 9,9% respectively which is again normal taking into consideration the latency introduced by the middlebox.

Lastly, the testing with a router implementing also a firewall introduced performance gains based on the previous pattern. The performance gains were 17,32% while loading a webpage with text only and 11,57% with the webpage containing images.

The results were are not scientific to the level previously researchers have conducted but they can show that the TFO mechanism benefits are noticeable even on the simplest scenarios.

Finally, using the particular physical middleboxes from enterprise well-known brands, the possible implications were studied in order to identify if there were any removal of the TFO option during the network data exchange. Despite the fact that many sources indicate that these implications exist, during the testing no problem was identified and therefore the TFO option worked as intended. This is also normal due to the small percentage of these incidents many researchers reported but still this can vary due to the big deviation in numbers reported in papers of the academic community.

5 Conclusions

Having taken a deep dive in how the TCP three-way handshake along with the TCP Fast Open option, the benefits are clearly visible even on the simplest testing scenarios. The examination of some middleboxes, despite the fact that did not show any complications as seen from the studies is a real problem which cannot be ignored when implementing a mechanism on such a large scale as the web. The biggest problems evolve when privacy and security is involved while using the mechanism which cannot be ignored.

Privacy concerns are way more important nowadays with the evolving of the internet and, this mechanism, while it works just as an extension of the most popular protocol it does not implement advance methods in ensuring that the user's privacy and security is ensured to the desired levels.

Among others, these reasons led to the mechanism be deprecated and basically abandoned for more advance protocols and mechanism which target not only mainly to the performance gains but also are security and privacy oriented by design.

Moving forward and with the evolution of the web a new era is coming to web technologies. The next generation of HTTP/3 will help move forward and solve many problems occurred during the years by the rapid evolving of the web. While the previous versions of HTTP the HTTP/1.1 and HTTP/2 which use the TCP as their transport, HTTP/3 uses QUIC for the transport layer. QUIC is a general-purpose transport layer protocol designed initially by Jim Roskind at Google. As of today, the QUIC is an internet draft and it is widely used by Google Chrome, Microsoft Edge, Mozilla Firefox and Apple Safari browsers even if it not supported by default. This technology offers not only performance gains but also security benefits by design which solves numerous problems presented the recent years in the web. With the rapid evolvement of the internet technologies we are not long before the full transition occurs and the web takes the next step into the evolution.

Bibliography

1. https://home.cern/science/computing/birth-web/short-history-web
2. Book Greek computer network
3. Book Computer networking a top down approach 7th
4. https://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt (/proc/sys/net/ipv4/* Variables:)
5. https://docs.nginx.com/nginx/admin-guide/
6. https://github.com/rockdaboot/wget2/blob/master/docs/wget2.md
7. https://gitlab.com/gnuwget/wget2/-/wikis/home
8. https://httparchive.org/reports/page-weight?start=earliest&end=latest&view=list
9. https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html#Trends
10. RFC 1644
11. https://web.archive.org/web/20010305122504/http://www.mid-way.org/doc/ttcp-sec.txt
12. RFC 6247
13. RFC 1379
14. TCP Fast open proposal 2011
15. RFC 7413
16. Analysis of TCP Fast Open (Erich Meissner)
17. TCP Fast Open: Initial Measurements (Anna Maria Mandalari, Marcelo Bagnulo, Andra Lutu)
18. https://kernelnewbies.org/Linux_3.7
19. https://squeeze.isobar.com/2019/04/11/the-sad-story-of-tcp-fast-open/
20. https://lwn.net/Articles/458610/#tfo

21. Your App and Next Generation Networks, Apple Inc. 2015
22. https://developer.microsoft.com/en-us/microsoft-edge/platform/changelog/desktop/14352/
23. https://svnweb.freebsd.org/base?view=revision&revision=330001
24. https://svnweb.freebsd.org/base?view=revision&revision=335610
25. https://bugzilla.mozilla.org/show_bug.cgi?id=1188435
26. https://www.youtube.com/watch?v=jpkYdI9dkgs
27. Enhanced Performance and Privacy for Core Internet Protocols, Erik Sy Phd.

Appendix

Git installation

- sudo apt-get install g++
- sudo apt-get install git

PCRE – Supports regular expressions. Required by the NGINX Core and Rewrite modules.

- wget ftp://ftp.pcre.org/pub/pcre/pcre-8.44.tar.gz
- tar -zxf pcre-8.44.tar.gz
- cd pcre-8.44
- ./configure
- make
- sudo make install

Openssl build Supports the HTTPS protocol. Required by the NGINX SSL module and others.

- wget http://www.openssl.org/source/openssl-1.1.1g.tar.gz
- tar -zxf openssl-1.1.1g.tar.gz
- cd openssl-1.1.1g
- ./Configure LIST | grep -i linux
- ./Configure linux-x86_64 --prefix=/usr

- make
- sudo make install

- zlib – Supports header compression. Required by the NGINX Gzip module.
- wget http://zlib.net/zlib-1.2.11.tar.gz
- tar -zxf zlib-1.2.11.tar.gz
- cd zlib-1.2.11
- ./configure
- make
- sudo make install

- sudo apt-get install libgd-dev

Nginx installation

- wget https://nginx.org/download/nginx-1.18.0.tar.gz
- tar zxf nginx-1.18.0.tar.gz
- cd nginx-1.18.0
- git clone https://github.com/openresty/headers-more-nginx-module
- ./configure \

```
> --conf-path=/etc/nginx/nginx.conf \
> --sbin-path=/usr/sbin \
> --error-log-path=/var/log/nginx/error.log \
> --with-threads \
> --with-stream \
> --with-stream_ssl_module \
> --with-http_image_filter_module \
> --with-pcre \
> --with-http_mp4_module \
> --with-http_secure_link_module \
> --with-http_v2_module \
> --with-http_flv_module \
> --add-module=headers-more-nginx-module \
> --with-http_gzip_static_module \
> --with-http_stub_status_module \
> --with-http_ssl_module \
> --http-proxy-temp-path=/dev/shm/proxy_temp \
> --http-client-body-temp-path=/dev/shm/client_body_temp \
> --http-fastcgi-temp-path=/dev/shm/fastcgi_temp \
> --http-uwsgi-temp-path=/dev/shm/uwsgi_temp \
> --http-scgi-temp-path=/dev/shm/scgi_temp \
> --build="v1.11.12 with TFO - UnixTeacher" \
> --with-cc-opt='-O2 -fstack-protector-strong -DTCP_FASTOPEN=23'
```

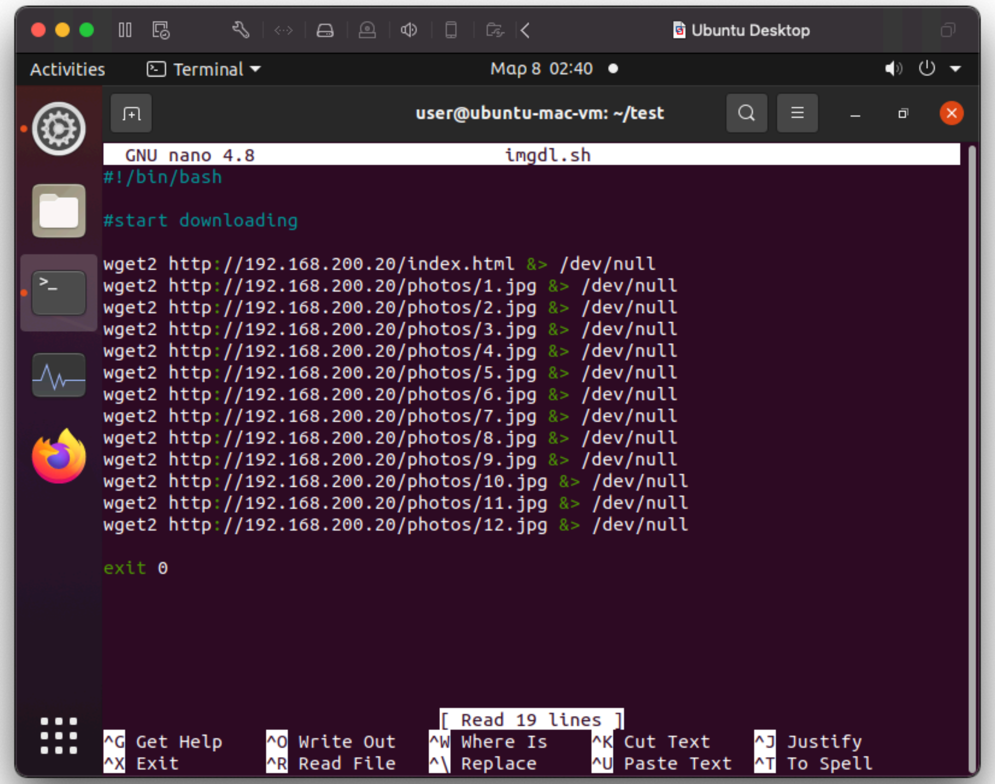
- make -j4
- sudo make install

NGINX control

- sudo nginx -t
- sudo nginx
- sudo nginx -V
- sudo nginx -s reload

wget2 commands

- wget2 http://192.168.200.20/index.html



```
  GNU nano 4.8                           imgdl.sh
#!/bin/bash

#start downloading

wget2 http://192.168.200.20/index.html &> /dev/null
wget2 http://192.168.200.20/photos/1.jpg &> /dev/null
wget2 http://192.168.200.20/photos/2.jpg &> /dev/null
wget2 http://192.168.200.20/photos/3.jpg &> /dev/null
wget2 http://192.168.200.20/photos/4.jpg &> /dev/null
wget2 http://192.168.200.20/photos/5.jpg &> /dev/null
wget2 http://192.168.200.20/photos/6.jpg &> /dev/null
wget2 http://192.168.200.20/photos/7.jpg &> /dev/null
wget2 http://192.168.200.20/photos/8.jpg &> /dev/null
wget2 http://192.168.200.20/photos/9.jpg &> /dev/null
wget2 http://192.168.200.20/photos/10.jpg &> /dev/null
wget2 http://192.168.200.20/photos/11.jpg &> /dev/null
wget2 http://192.168.200.20/photos/12.jpg &> /dev/null

exit 0

                              [ Read 19 lines ]
^G Get Help    ^O Write Out   ^W Where Is    ^K Cut Text    ^J Justify
^X Exit        ^R Read File   ^\ Replace     ^U Paste Text  ^T To Spell
```