UNIVERSITE CATHOLIQUE DE LOUVAIN

ECOLE POLYTECHNIQUE DE LOUVAIN

Multipath TCP and OpenVPN

Supervisor: Olivier Bonaventure
Readers: Gregory Detal
Marc Lobelle

Thesis submitted for the Master's degree
in computer science and engineering
options: SINF2M1
by Alois Paulus

Louvain-la-Neuve
Academic year 2014-2015

CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Introduction

Having a fast and reliable internet connection is something that individuals and compagnies would want.

Nowadays, there are multiple solutions to this, for example compagnies can get a fiber optic connection but it is still very expensive.

Individual or small entreprise have Multihoming. It uses multiple internet connections at the same time and balances traffic amongst them using a router configured especially for that. Unfortunatly it does not always improve the speed of the connection and can be slow to react to failures.

In this paper, I describe and analyse another solution which is less expensive than optic fiber and will make better use of brandwidth and be more tolerant to failures than Multihoming. This solution implies using MultiPath TCP, OpenVPN and OpenWRT.

Multipath TCP (MPTCP) has been implemented by the IP Networking Lab at Université catholique de Louvain(UCL) in Louvain-la-Neuve for the linux kernel. It is an extension of the TCP protocol, and it's goal is to support the use of multiple transport paths between a pair of hosts. The benefits of MPTCP include a performance improvement in term of throughput and a better handling of failures.

OpenVPN is an open source application implementing a virtual private network (VPN) to create secure connections between hosts over a network. OpenWrt is a linux distribution for embedded devices typically home routers. It is built from the ground up to be a full-featured, easily modifiable and light weight operating system.

In this setup, MPTCP will be use to balance the traffic between multiple internet connections and recover from a possible failure of one of them. OpenVPN will be use to create the link between the MPTCP compatible server and the home router running a modified version of OpenWRT supporting MPTCP. For this to work, the user/entreprise should have two or more internet connections and a server with a high brandwith running a MPTCP kernel.

In this paper TODO

I hope this will give the possibility to more people to get a fast and reliable internet connection.

1.2 Multipath TCP (MPTCP)

Multipath TCP has been implemented by the IP Networking Lab at Université catholique de Louvain(UCL) in Louvain-la-Neuve for the linux kernel. MPTCP is an attempt to solve problems of the TCP protocol : the inability to move TCP connection from one IP to another and the inability to use more than one path for a single TCP connection. Many attempts have been made but some involve changes in the applications and others lose advantages of current TCP protocol.

Multipath TCP also need to be backward compatible with normal TCP to ensure that even if the other end is not compatible they can still communicate. Another thing to achieve is the fairness between MPTCP and TCP: if MPTCP and normal TCP share a bottleneck link, MPTCP should not be able to use more bandwith than a normal TCP connection.

1.2.1 Architecture

Multipath TCP is an extention of the TCP protocol,. It does not change the interface of the socket API for applications which allows all applications to benefit from it without need for modification. It removes the strong link between IP and TCP so that devices can use multiple IP for a single MPTCP connection.

MPTC uses TCP options to annonce MPTCP support and communicate MPTCP informations between host.

The setup of a MPTCP connection is as follow:

1. First the initiator of the connection annonces that he is MPTCP compatible (MP_CAPABLE)
2. Then if the receiver is MPTCP compatible, it establish a new MPTCP connection
3. After that, it adds new subflow on every know path to the existing MPTCP connection (MP_JOIN)
4. Finally it transmit data on the MPTCP connection

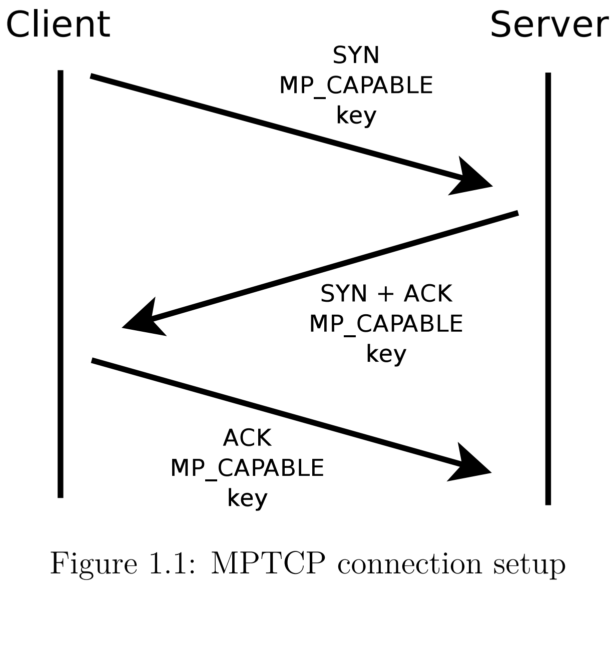

Figure 1.1: MPTCP connection setup

To establish a MPTCP connection it uses a three-way handshake as show in figure 1.1. A client send a packet having the MP_CAPABLE option to tell the server he is compatible. The server send back a ACK containing a MP_CAPABLE option and a random key if it is compatible. After that the client send a ACK to the server containing the key, this will setup the first subflow.

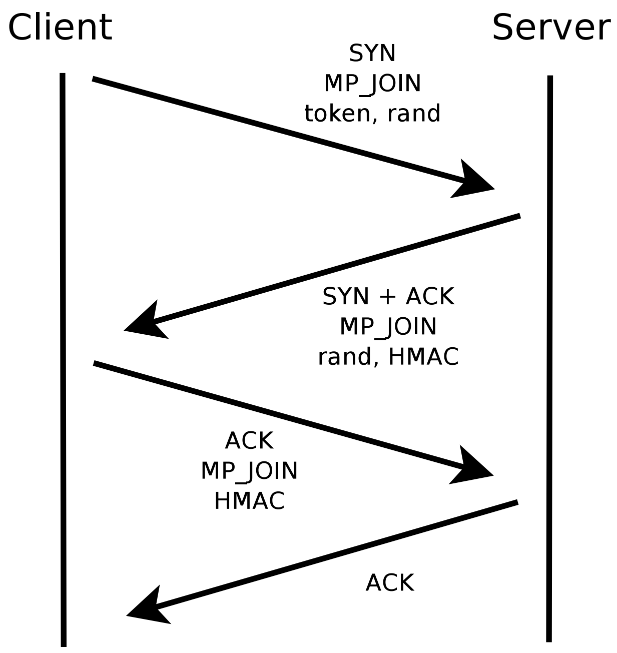

Figure 1.2: MPTCP Add subflow

If the client has more than one physical link connected with the server, MPTCP will try to add another subflow to the existing connection. For this it will use a three-way handshake and the MP_JOIN option with a unique token.(figure 1.2)

1. First a SYN with the MP_JOIN Option, a unique token and random key will be sent
2. Then the client and the server will compute an hash-based message authentification code computed over the random key exchanged.

The token is there to be able to uniquely identify a subflow, because in normal TCP a connection is identified by <srcip,srcport,dstip,dstport> which in this case is not sufficient to uniquely identify a subflow in case of a host behind a NAT.

When multiple subflows have been established, data can be transmited over any of them. This can be used to increase the throughput of the connection or to retransmit data if one subflow fail. Each subflow is equivalent to a single TCP connection.

If multiple subflows are use for a single TCP connection MPTCP must decide on which one to send the data and how much data. For this purpose, MPTCP use a scheduler, there are two type of scheduler in MPTCP :

1. The default scheduler used by MPTCP will first send data on subflows with the

lowest RTT until their congestion-window is full then it will start transmitting on the subflows with the next higher RTT.

2. The roundrobin scheduler will transmit X consecutive segment on each path in a round robin fashion (X can be changed and default is 1).

Using multiple subflows also require to order the data within each subflow and then order the data between these different subflows, because some can be faster than others. When receiving data it first reorders them at the subflow level using the subflow sequence number, a 32bits number. Then it reorders the data between the different sublows at the connection level using the data sequence number.

1.2.2 Congestion

Congestion control in MPTCP is more complex than with regular TCP because it uses multiples subflows for one connection. On these subflows the level of congestion can be very different, if one uses standard TCP congestion algorithm the performance for MPTCP it usually become unfair to regular TCP and in some situations decreases the performances.

To solve that problem a few congestion control algorithms have been developped for MPTCP like LIA, OLIA, wVegas. These algorithms are described below.

Linked Increase Algorithm (LIA)

The LIA algorithm has three main goals :

1. Having a total throughput bigger than the one TCP can get using the best path
2. Not being more aggressive than TCP, be fair to TCP.
3. Balancing the congestion over paths

As it accomplishes the two first goal but fails to accomplish the third one, a new algorithm has been developped, OLIA.

In LIA, the slow start, fast retransmit, and fast recovery algorithms, as well as the multiplicative decrease of the congestion avoidance state are the same as in standard TCP.

What changes is the increase phase of the congestion avoidance state, the window is increased by the minimum between the increase that would get normal TCP and the computed increase for the multipath subflow. The computed increase is calculated using a parameter α which defines the aggresiveness of the MPTCP flow. The value of this parameter α is chosen such that the aggregate throughput of the multipath flow is equal to the rate a TCP flow would get if it ran on the best path. α need to be calculated for each MPTCP flow. The formula which is used has been derived by equalizing the rate of the multipath flow with the rate of a TCP running on the best path.

This guarantees the goal number two: not being more aggressive than TCP. Goal one is accomplished by computing an increase for the multipath subflow equal to the throughput a TCP flow would get if it ran on the best path.

Opportunistic Linked-Increases Algorithm (OLIA)

OLIA is a window-based congestion-control algorithm that couples the increase of congestion windows and uses unmodi

fed TCP behavior in the case of a loss. Like LIA, the algorithm only modifies to the increase part of the congestion avoidance phase.

It uses a set of best paths devided in two, a set with max windows and the rest of the paths. For the paths that have a small windows, OLIA increase the window faster. For the path that have the max windows, the increase is slower.

wVegas

wVegas is a delay-based congestion control based on TCP vegas. It is more sensitive to change because it does not have to wait for packet losses to react and shift between subflows to adapt to network congestion. For each subflow it calculate the difference between the expected sending rate and actual sending rate

During slow-start it double the congestion window every other RTT. If the difference is bigger than a threshold, it switches to congestion-avoidance. During congestion-avoidance it increase the window by one packet every RTT if the difference is small otherwise it reduces it by one packet.

This means that overall the window grow slower than other algorithm and this avoids losses. However when the BDP is large, it switches too fast to the congestion avoiding phase and gives bad throughput because the windows is too small.

1.2.3 Usages

MPTCP can be use in different fields. It is usefull for devices like smartphones that have multiple network interfaces with different throughput and latency. Today these devices only use one interface at a time and if it loses the connection to that interface, all the currently open TCP connection will be terminated and will need to be reopenned.

This is a situation that MPTCP can solve by having one subflow for the 3G interface and one subflow over the wifi interface: when one subflow fails, the data are sent over the other subflow.

MPTCP can also be very usefull in data center where different topologies use multiple paths accross the server layer and the traffic need to be balanced between the paths to avoid congestion.

1.3 VPN

A Virtual private network provide a point-to-point connection between two or more nodes on top of an existing network, it create a tunnel between the connected nodes.(figure 1.3)

In VPN privacy is a concern: The data which are transmited using these virtual network need to be encrypted so that if someone intercept it on the global network, it cannot be read. The data are encrypted before entering the tunnel and will be decrypted after leaving it, the VPN will usually add extra header information to the packets for the remote node to be able to decrypt.

Another technique used in VPN is the authentification. Client and server must provide information about who they are so that they can prove their idendity. This is mostly done by using a Message Authentication Code (MAC).

One of the main motivation for VPN is that it is cheaper to use multiple virtual networks implemented on a global network than use multiple smaller physical networks. It is usually used by compagnies to create a private network over the internet.

There are several types of VPN technologies : Point-to-Point Tunneling Protocol,Secure Socket Layer/Transport Layer Security (SSL/TLS) or Internet Protocol Security. They all use the same basic principle to transfert data accross by tunneling it.

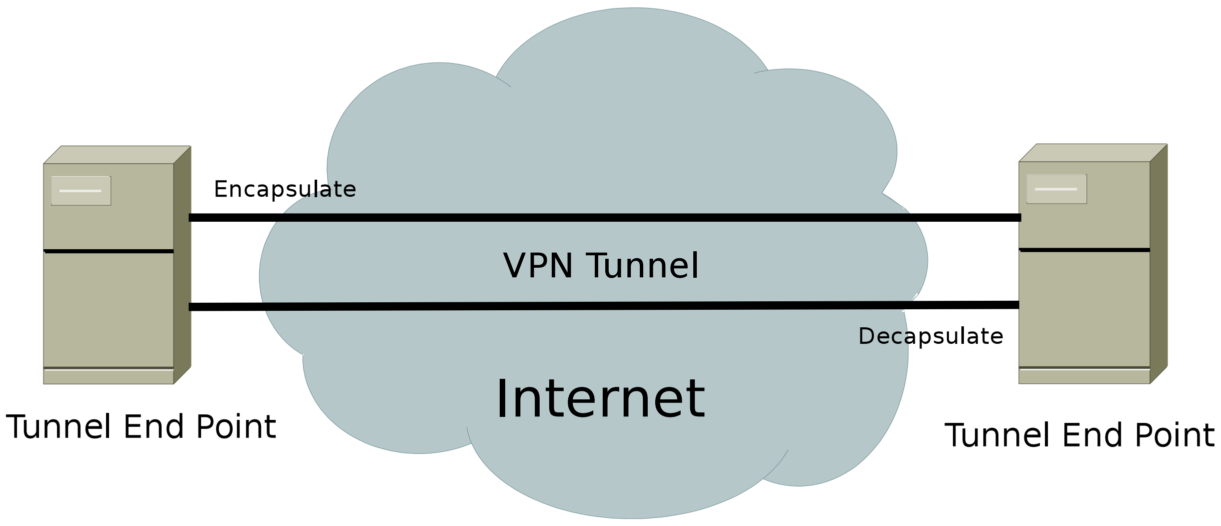

Figure 1.3: VPN Tunnel

The principle is that the packet is encrypted and encapsulated within another packet which will be delivered to the other node. The node will then decapsulate the packet and decrypt it.

1.3.1 Why using a VPN

For this article, we will use a end-to-end, one-to-one connection to connect the router and the server with each other and encapsulate all the clients traffic into the OpenVPN tunnel.

So even in the case clients do not support MPTCP, all the packets sent by the client will be encapsulated on the router within OpenVPN packet and because the router is MPTPC capable, the OpenVPN packets will use MPTCP to transit between the router and the server.

1.3.2 OpenVPN

OpenVPN is an open source application implementing a virtual private network to create secure connections over a network, it is developped by James Yonan.

It uses SSL/TLS in its custom security protocol and uses the OpenSSL library to do the encryption of data and the authentication. There are several authentication methods like static key, certificate or username/password. Using Encryption is more secure but needs more ressource because packets need to be decrypted.

OpenVPN uses two channels, a data channel that carries the users' IP datagrams and a control channel that handles key negotiation and configuration.

OpenVPN can use UDP or TCP to transport the packets and provide several security feature. UDP is usually advised because it is generaly faster.

OpenVPN is available on a lot of operating system : Linux, Windows, Mac OS X and multiples router firmware such as DD-WRT, OpenWRT ... This gives the possibility to make your router act as a OpenVPN client and users connected to that router can access a VPN without having to install OpenVPN on their computers.

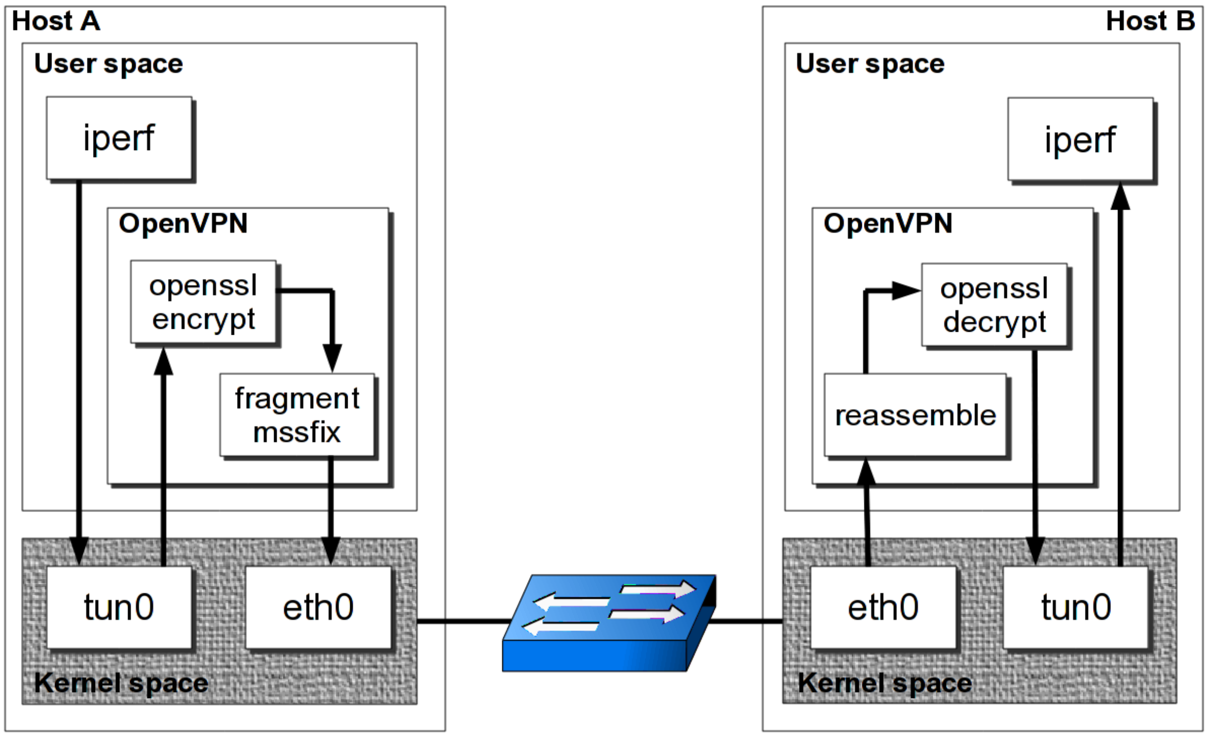

Figure 1.4: VPN Packets flow

OpenVPN runs in user space which provides good security and portability but packets need to travel between kernel space and user space which is called context switching. Context-switching takes CPU cycle, which can impact a lot the performances if the hardware is not powerfull enough.

On figure 1.4 you can see how the packets flow from an application to the network and from the network to the application.

To create the tunnel OpenVPN uses the tunnel driver that is provided as a module for linux. This driver enable the creation of virtual interfaces called tun or tap devices. The difference between tun and tap are the layer at which they are, tap is a layer 2 and tun is a layer 3 device.

In practice this means that tap is able to send ethernet frame and add more overhead and tun is only able to send packet but add less overhead. For the tests presented in this paper the interface tun is enough and is a better choice because it add less overhead.

CHAPTER 2

SETUP

2.1 Setup

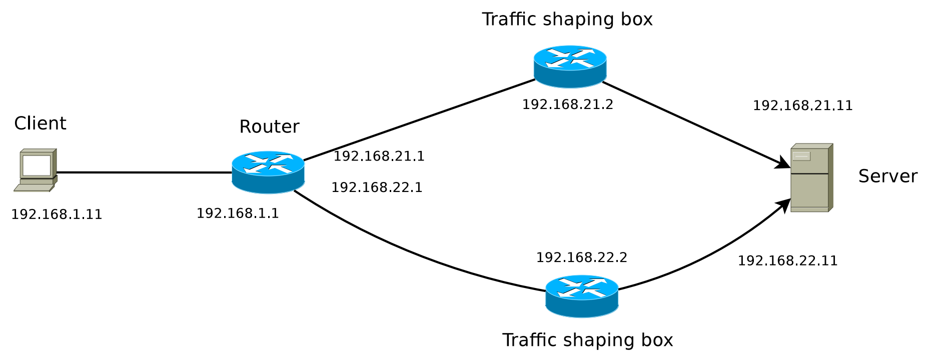

Figure 2.1: Setup

As show in figure 2.1 the setup consists of five machines :

- One home computer, a client.
- One server for the OpenVPN server
- One home router connected to the OpenVPN server, by two ethernet cable to enable MPTCP to use two subflows.

- Two routers to do some traffic shaping.

OpenVPN client

It is a TP-LINK N900 router, its CPU is a PPC P1014@800MHz and it has 128Mb of RAM [19]. It will be running OpenWRT with a custom kernel 3.10.49 with MPTCP v0.89.0-rc compiled by Sébastien Barré and Gregory Detal.

Traffic shaping boxes

The traffic shaping boxes will mainly limit the brandwith and add delay between the router and the server to emulate real internet connections speed. They are TP-Link TL-WR741ND router, CPU Atheros AR7240 @ 350 MHz and 32Mb of RAM. It will be running OpenWRT with tc, a shaping software, installed.

OpenVPN server

The OpenVPN server is a computer with a Intel Pentium 4 CPU and 2Go of RAM. It will be using debian running kernel 3.14.33 from the MPTCP Github repository **??** with MPTCP v0.89.5 compiled with all the TCP congestion algorithms.

The client computer will connect to the router and access the server through it.

For the purpose of these tests we decided not to use encryption or authentification on the OpenVPN tunnel, mainly because it need more CPU power and the TP-LINK N900 is not powerfull. Using encryption and authentification would have decrease performances and given that the main goal of this paper is not security, encryption is not needed.

2.1.1 OpenVPN configuration

Listing 2.1 presents the important parameters in the OpenVPN configuration file. The whole configuration can be found in the annexes.

```
auth none
cipher none
sndbuf 0
```

```
rcvbuf 0
```

Listing 2.1: Openvpn configuration

As *auth* and *cipher* are about encryption and anthentification, they are disable because, as explained earlier, they add more overhead to packet and are not needed for these tests because privacy is not our concern.

sndbuf and *rcvbuf* are two very important settings that are used by OpenVPN to set the *SO_SNDBUF* and *SO_RCVBUF* options on individual socket with setsockopt. The *SO_SNDBUF* and *SO_RCVBUF* options are the send and receive buffer size. When they are set to 0, OpenVPN does not set value with setsockopt and the value from the OS are used.

After reading quite a lot about these two parameters the most common advise for Linux is to use a value of 0 for *sndbuf* and *rcvbuf* because it let the OS taking care of adjusting the TCP window size.

For the purpose of the tests in this paper, it is important to keep control on the size of the TCP window to be able to evaluate their impact. Consequently, I will set values for *sndbuf* and *rcvbuf* manually.

The advantage of using these settings instead of set globally the value with the help of sysctl is that it only affect sockets created by OpenVPN and not all the sockets and it is easier to change.

2.1.2 Delay boxes

The two delay boxes are placed between the OpenWRT router and the server, one for each link. Their goal is to limit the bandwidth and introduce delay on links.

For this I used tc, a traffic control tool in the linux kernel, used to configure and control the network schedulers. It enables to choose between a sets of queing systems and mechanisms to control how the packets are received and transmitted. This is usually called QoS (Quality of service).

It gives the possibility to :

- Limit the total bandwidth of a link using TBF or HTB

- Give priority to latency sensitive traffic
- Distribute unreserved bandwidth equitably
- Reserve bandwidth for a particular application
- Limit the bandwidth of a particular user, service or client
- Introduce delay, loss, duplication, reordering (netem)

From the many possibilities offered by tc, I use in this paper the first and the last one, using TBF and netem. Here is an example :

```
# tc qdisc add dev eth0 root handle 1:0 tbf rate 8mbit burst 50k
latency 5ms
# tc qdisc add dev eth0 parent 1:1 handle 10: netem delay 50ms

```

The first line of this script will add a *qdisc* which is a scheduler to the interface *eth0.* The handle is the name given for this *qdisc* on the interface *eth0.* The handle is defined by a number of form major minor, it is followed by a class type, tbf which is a queuing discipline to attach to the *qdisc.* It will limit the rate to 8mbit and set the buffer to 50k.

The second line add a class to the interface *eth0* on the parent handle (from line 1). Again we give a handle and then we give the class type, netem. It will add a delay of 50ms to the interface *eth0.*

In this example we use netem to add delay to the link and then use TBF to add a bandwidth limit. This simple process is adequate to our needs.

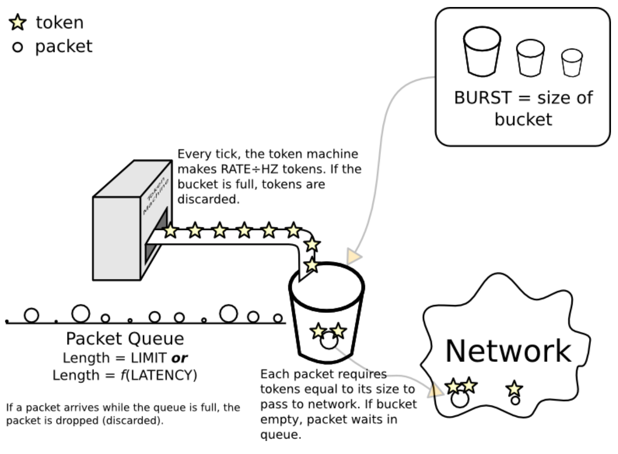
☆ token
○ packet
BURST = size of bucket
Token Machine
Every tick, the token machine makes RATE÷HZ tokens. If the bucket is full, tokens are discarded.
Packet Queue
Length = LIMIT *or*
Length = *f*(LATENCY)
If a packet arrives while the queue is full, the packet is dropped (discarded).
Each packet requires tokens equal to its size to pass to network. If bucket empty, packet waits in queue.
Network

Figure 2.2: Token Bucket Filter

Source: [14]

Figure 2.2 presents how TBF works in practice. The burst parameter is equivalent to the size of the bucket, it holds a limited amount of tokens and if too many are added the bucket overflows and the tokens are lost.

The latency or limit is the number of packets that can sit into the line waiting to get a token. This occurs when the bucket is empty or if the bucket does not have enough tokens for all the packets.

When the kernel send a packet it first needs to obtain a tokens. If it does not obtains a token it is put in a line to wait for a token and if this line is full it is dropped.

2.2 Tests Procedures

To mesure the differences in term of performance when using OpenVPN we will measure the throughput, the delay and the CPU usage using different configurations such as follow :

1. UDP protocol as OpenVPN protocol
2. TCP protocol as OpenVPN protocol with MPTCP using 2 links (2 subflow)

The tests will be ran using different link speed to mimic real internet connection speed. For that I looked for the different FAI in Belgium and listed their available speeds in April 2015.

Here are the one selected for the tests :

- 8mbps (edpnet)
- 32mpbs (voo telenet)
- 100mbps (numericable)

Delay will also be introduces on the links to see how the performances may vary.

- 1ms
- 25ms
- 200ms

2.2.1 Bulk data transfer

A common type of network traffic on the internet especially with the cloud is bulk data transfers. It is used when pulling files from a FTP or accessing data on your shared drive (google drive or dropbox).

To simulate this type of traffic, I use the TCP_MAERTS a test from Netperf, it measure bulk data transfer performance. This will measure how fast the server can send data to the client.

TCP_MAERTS

TCP_MAERTS test open a single TCP connection to the remote machine and makes as many rapid write calls as possible. These write calls have a specific size which by default is the sender buffer send size.

The socket buffers on either end will be sized according to the systems default and all TCP options will be at their default settings.

Options :

- -s size Sets the local socket send and receive buffers (size in bytes)
- -S size Sets the remote socket send and receive buffers (size in bytes)
- -m size Sets the local send message size to size
- -M size Sets the remote receive message size to size
- -D Sets the TCP_NODELAY socket option on both the local and remote systems

2.2.2 TCP Request/Response

Another common type of network traffic on the internet between client and server is the request/response model. This model is based on individual transactions between the client and the server. The client sends a request to the server, the server receives it, process it and sends a response back to the client.

To mimic this type of traffic, the TCP_CRR test from Netperf will be used. Apache Benchmark (AB) will also be used because the TCP_CRR is sequential and it is interesting to analyse the impact on the results when multiple connections are used at the same time. The server will have a Apache server with the default settings and will serve the default apache welcome page.

TCP_CRR

TCP_CRR measures the performances of establishing a connection, exchanging a single request/response transaction, and tearing-down that connection. This is similar to the process of an HTTP 1.0 or HTTP 1.1 connection when HTTP Keepalives are not used. It is a synchronous, one transaction at a time, request/response test. By default the message size is 1 byte but it can be changed using options.

Options :

- -r req,resp sets the size of the request or response message, or both
- -s size sets the size of the local socket send and receive buffers to size bytes
- -S size sets the size of the remote socket send and receive buffers to size bytes
- -D sets the TCP_NODELAY socket option on both the local and remote system

Apache Benchmark (AB)

Apache Benchmark (AB) is a tool for benchmarking HTTP server. It shows how many request per second the server can deliver to the client by sending a arbitrary number of concurrent request. It has been created to benchmark the Apache web server.

2.3 Git Repository

To make all the tests reproductable, I have try to automize as much as possible, using mainly python and bash script. Everything is avaible on a git repository on Github `https://github.com/alokhan/memoire`.

In this repository you find all the configuration files used for the router and server, all the script used to generate the graphics, ,all the *.csv* files containing the test results used for this paper and also some documentation.

CHAPTER 3

TESTS RESULTS

This section prensent the test results.

Firstly I present the performance difference between TCP and UDP used as the OpenVPN protocol by testing the throughput and the number of request/response per second. Secondly I test the behavior of MPTCP using link with different RTT and bandwith. Finally I try different congestion control and analyse wether it help increasing performances.

3.1 TCP vs UDP

In this section I compare the througput of a TCP connection using TCP and UDP as the OpenVPN tunnel protocol to determine where MPTCP can be interesting and where not. Many combinaison of bandwidth and delay will be used they will be specified for each test. The size of the openvpn buffers will also be specify as it plays a role in the overall throughput. TCP windows autotuning mechanism of Linux will always be enable.

3.1.1 Bulk data transfer

This first test is a bulk data transfer.

Test setup :

- First link bandwith of 8mbit/s, 32mbit/s, 100mbit/s and RTT: 2ms (default route)
- Second link bandwith of 8mbit/s, 32mbit/s, 100mbit/s and RTT: 2ms
- Using default mptcp scheduler
- OpenVPN buffer default size 65536 bytes on client and server
- Congestion control : cubic

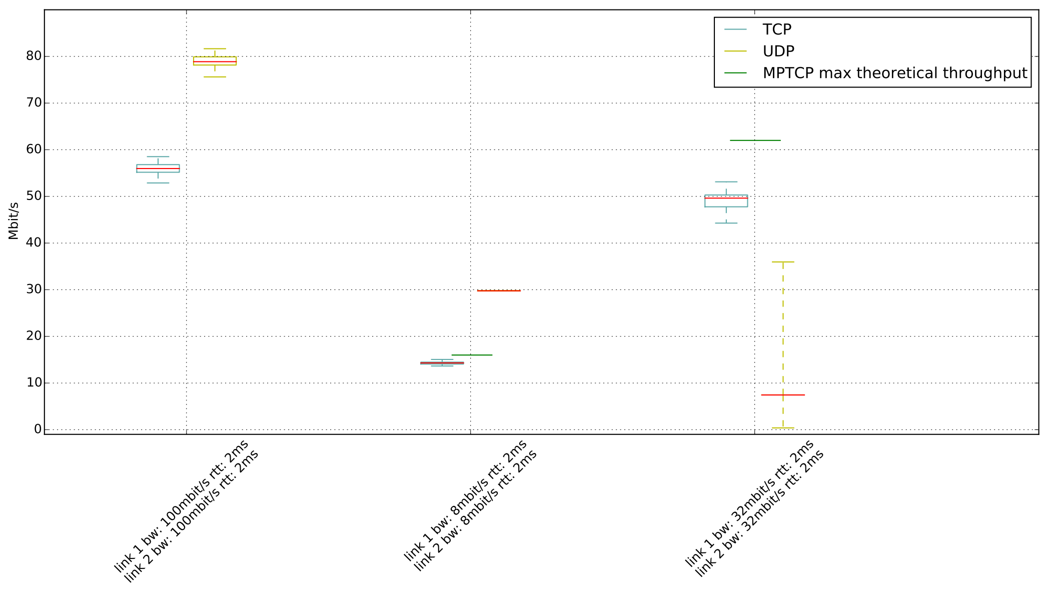

Figure 3.1: TCP vs UDP RTT: 2ms Buffer size: 64k

When using TCP as the OpenVPN protocol, MPTCP will creates two subflows, one on each link. Actually the full-mesh path manager create four subflow on this setup but the cross IP subflows has been blocked with the help of IPTables rules. On the other hand UDP will only use one of the available links.

For this test my hypothesis is that using MPTCP would always give a better throughput because it can use double of the bandwith than UDP can.

However results in figure 3.1 shows that TCP is faster than UDP for small maximum bandwidth like 8 or 32mbit/s but for greater maximum bandwidth like 100mbit/s, TCP is slower than UDP.

Besides, for the 32mbit/s bandwidth, the capacity of the two links are not fully used otherwise the throughput would be of about 60-64mbit/s. Further, even when using UDP and a maximum of 100mbit/s bandwidth, the whole bandwidth available is not used.

These results are mainly due to two things :

1. The router CPU is used at about 100% while the tests are running for the 32mbit/s (tcp) and 100mbit/s (tcp and udp) maximum link bandwidth
2. TCP over TCP can be bad because it can trigger multiple retransmissions [17]

The first reason is linked to the fact that the router has to encapsulate and decapsulate the packet from the OpenVPN tunnel as explained in section 1.3.2 causing the router to switch between kernel space and userspace. This is very costly in CPU cycle and the router CPU is a bit too weak.

The second reason relates to having two layers of TCP connection. TCP is a protocol that assumes an unreliable carrier. So if you have multiple layers of TCP connections each layer will guarantee that every data arrive to the other end. If the encapsulating or Internet layer drops a packet, both TCP streams will attempt to correct the error and retransmit duplicate data. This queues up data transmissions exponentially and slow down the data transmission.

Bofre concluding, it is interesting to analyse how the traffic was split accross the subflows. For this, I used *tcpdump* to capture packets and *mptcptrace* to analyse these captures. The traffic was captured on the OpenVPN server, on the sending side, to avoid having too much stress put on the router.

The sequence graphic generated by mptcptrace showed that the traffic was well splitted accross the subflows, which is the expected result because the scheduler used during these tests was the default and both links had the same RTT and same bandwidth.

Figure 3.2 showshow the traffic was split. When the green line approaches the +1 limit, it is the initial subflow which is mainly use and when the blue line approaches the -1 limit it is the second subflow which is mainly use. Here, the two lines are near 0, meaning that they are both used equaly.

To conclude for this test, even if UDP is faster than TCP in some cases, it should bet taken into account that with UDP if the link fails, the connection is lost. Depending on

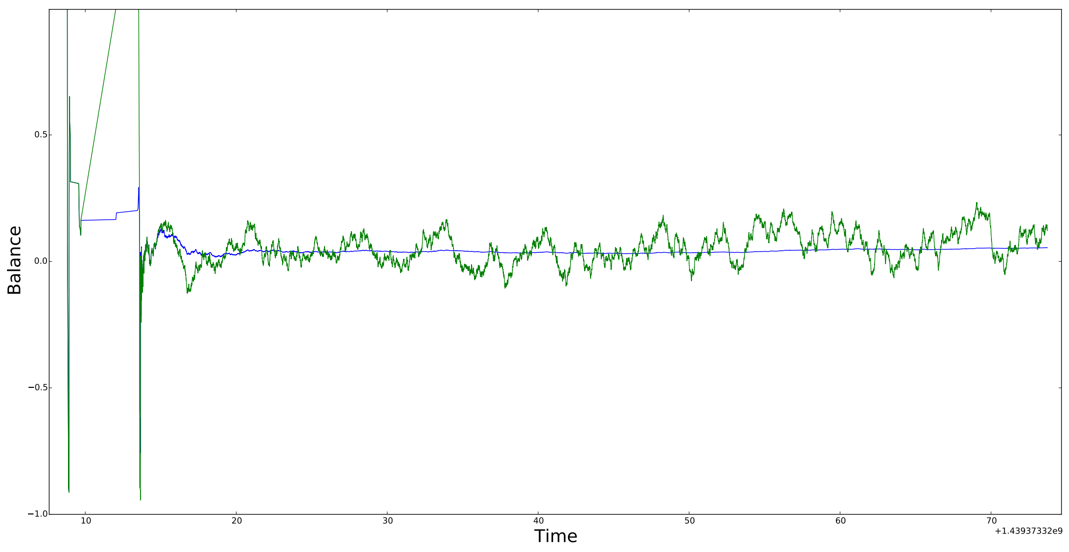

Figure 3.2: MPTCP Subflows Balance

what you are looking for, it can be usefull to have a more robust connection even if it is slower and in that case use TCP.

3.1.2 Delay introduced on links

This section presents the same test than in section 3.1.1 adding the insertion of delay on links and I then analyse which effect it has on the throughput. The delay are 25ms and 200ms on both links which give a 50RTT and a 400RTT. The buffers sizes are the default OpenVPN buffer size, 64k.

On figure 3.3 the effect of delay on the througput using TCP and UDP as OpenVPN protocol is presented

- For TCP the OpenVPN buffer size (64k) is a real disaster and the overall speed is very low compared to the 2ms RTT (see figure 3.1).
- For UDP he OpenVPN buffer size is not as bad as for TCP but the overall speed still decrease a lot.

For TCP this is mainly caused by wrong buffer size. Indeed the computation of the maximum throughput possible with a buffer of 64k and a RTT of 400ms is obtain

1.3Mbit/s. The same computation for a RTT of 50ms is 10Mbit/s. This is show on figure 3.3.

Figure 3.4 is the windows graph created by mptcp for the link bandwith of 32mbit/s with a RTT of 400ms. You can see that the window is totally full.

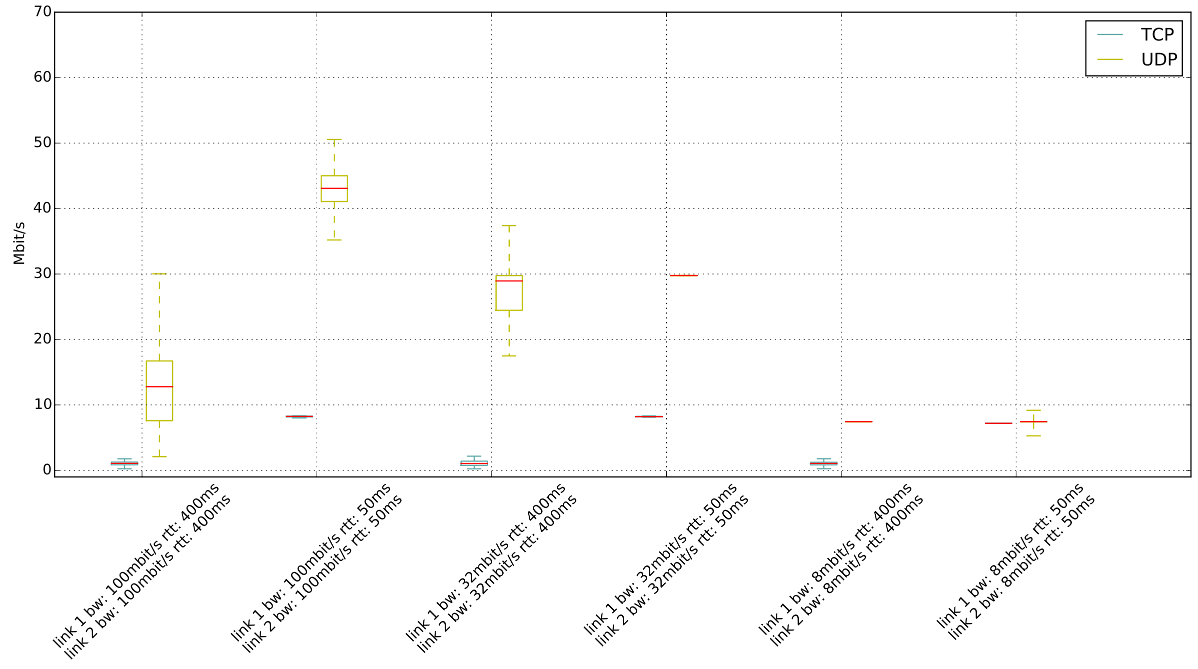

Figure 3.3: TCP vs UDP with high RTT buffer: 64k

This time the router CPU is around 20%-30% usage so there is no bottleneck. Section 4.1 presents and attempt to resolve this situation by changing buffer size on the openvpn client and openvpn server.

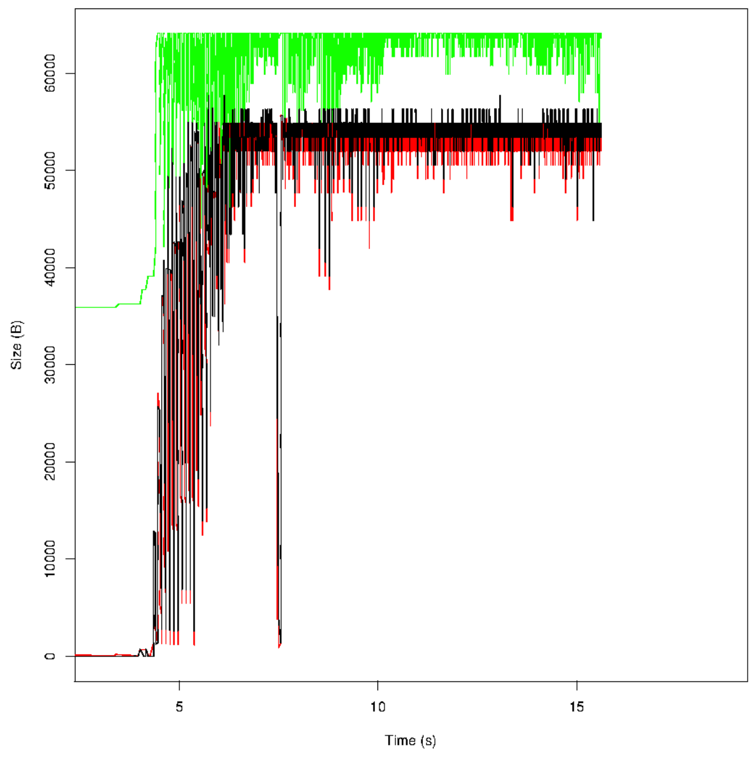

Figure 3.4: TCP Window being full

3.1.3 Request/Response

In this section I test the number of request / response per second when using TCP or UDP as the OpenVPN protocol. In this purpose I use test from Netperf and Apache Benchmark tool. My goal is to determine wether a browsing experience for a user would be possible or not depending on the network.

TCP_CRR

Test setup :

- First link bandwith of 8mbit/s, 32mbit/s, 100mbit/s and RTT: 2ms, 400ms (default route)
- Second link bandwith of 8mbit/s, 32mbit/s, 100mbit/s and RTT: 2ms, 400ms
- Using default mptcp scheduler
- OpenVPN buffer default size 65536 bytes on client and server
- Congestion control : cubic

- Request message size : 1 byte
- Response message size : 1 byte

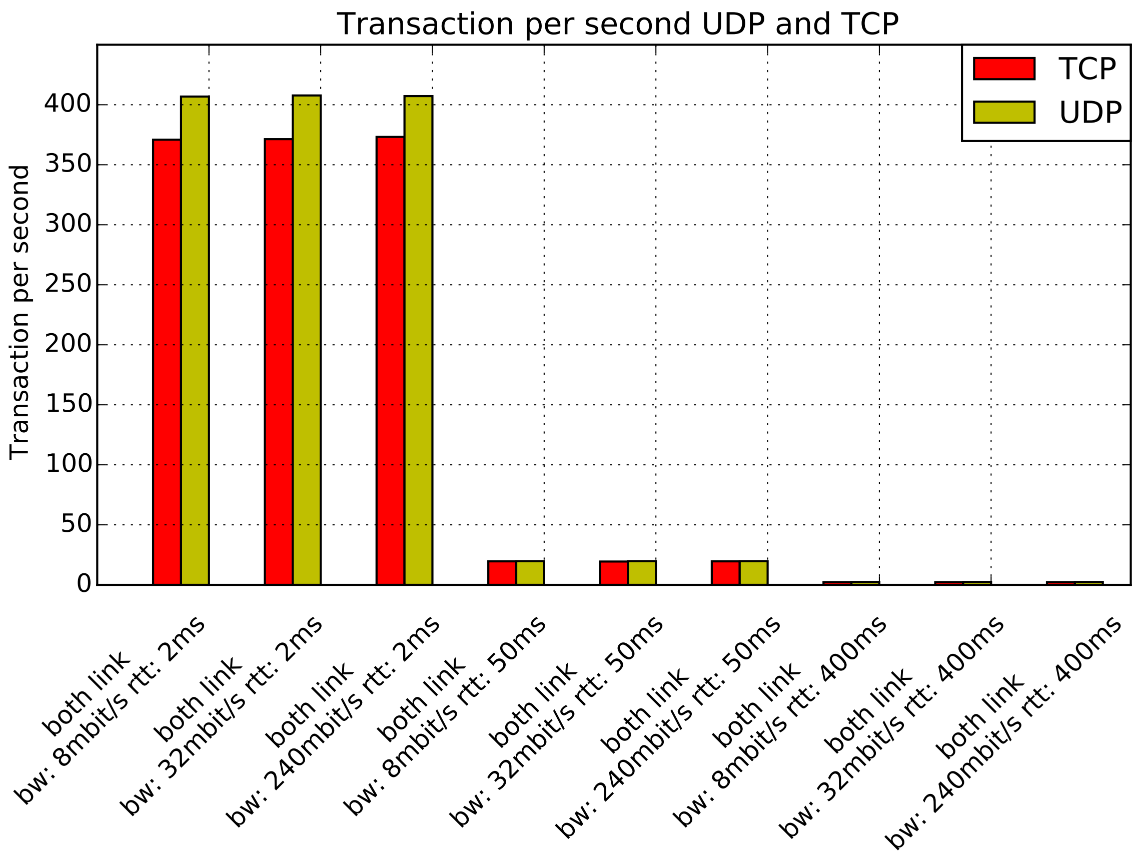

Figure 3.5: TCP vs UDP transaction per second request and response size : 1 byte

In this test the CPU and links bandwith do not seems to be the bottleneck. What seems to be the problem is the overhead due to creating and destroying TCP connection for each transaction.

Figure 3.5 shows that UDP seems to be a bit better. TCP being a connection-oriented protocol, it takes a bit more time to set up a connection than UDP. On top of that, the amount of data to be transmitted inside one transaction is very small, which means that a single connection is destroyed almost immediatly and it creates another one right away, this make the connection setup time very impactful.

In figure 3.5 is also visible the huge effect of the delay on the number of transaction per second, goes up to 400 transaction per second with a RTT of 2ms and it goes down to 2 transaction per second with a RTT of 400ms.

A high RTT, means that the creation and the teardown of a connection takes more time and because the test are synchronous, A transaction has to wait for the previous one transaction to finish before starting.

Increasing the data transmited into one transaction would be a more realistic and fair test between TCP and UDP. I used a 100 bytes request message size and a 170 bytes response message size to have identical size than the test done with Apache Benchmark (see section 3.1.3).

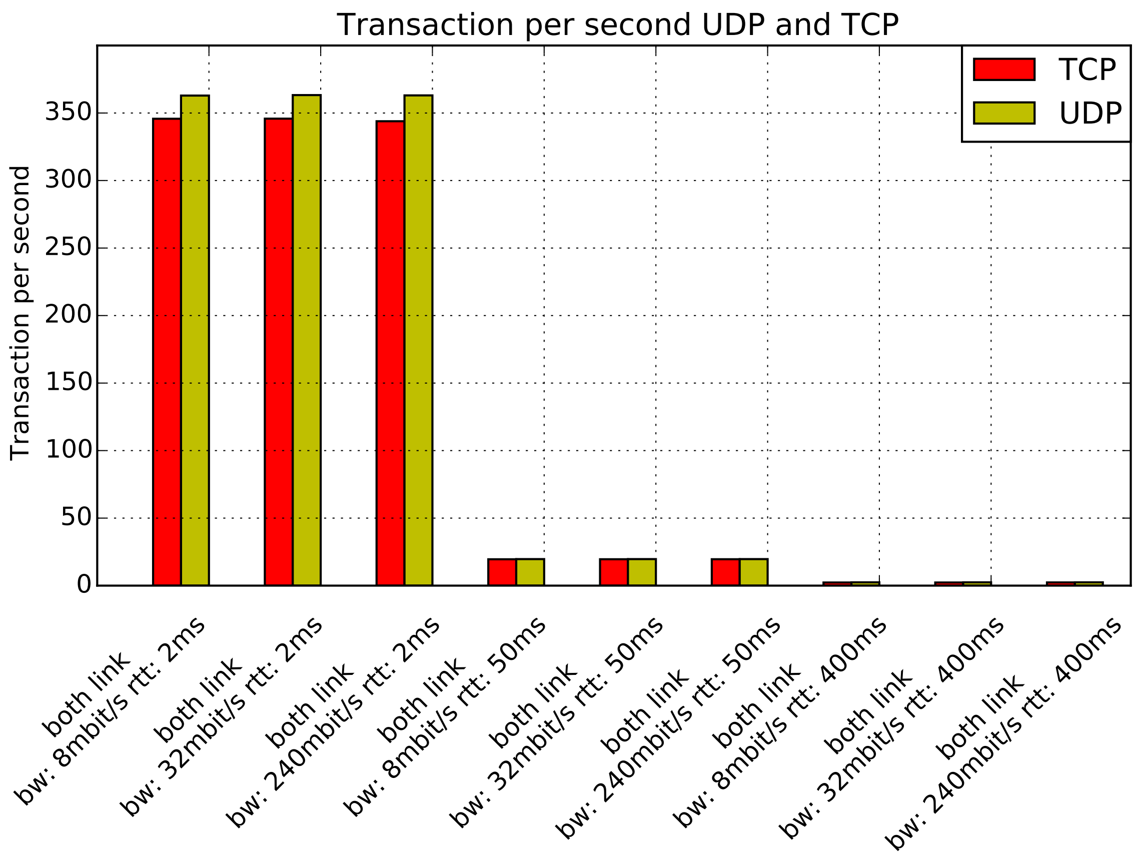

Figure 3.6: TCP vs UDP transaction per second request size : 100 / response size : 170 byte

Figure 3.6 shows that the difference between TCP and UDP is smaller because the data transmitted inside one transaction is bigger and the connection setup time has less impact. However the effect of the delay is still very strong and confirm my previous conclusion.

In conclusion, for this kind of traffic, MPTCP will not improve performance. The advantage of using multiple paths is not very usefull (only if one path fails) because it has no concurrent transaction and they are executed synchronously. In this case, UDP seems to be a better choice.

Hewever these tests might not reflect reality because normaly one would use one TCP connection for multiple transactions and establish multiple connections, unfortunatly Netperf does not provide such a test. To do that I use Apache Benchmark see section 3.1.3 below.

Apache Benchmark (AB)

The choice to run AB is to provide a more realistic way to test the process of requestresponse to a real web server. It also gives the possibility to use simultanious connections with the http *keep-alive* option.

When *keep-alive* option is set to true, it uses a single TCP connection to send and receive multiple HTTP request/response, as opposed to opening a new connection for every single request/response transaction like in TCP_CCR from netperf.

The settings used to run AB are :

- 6 concurrent connections, usually browser use between 6(Firefox,Chrome) and 8(Opera)
- maximum time of 60 seconds
- option *keep-alive* enable

AB always stops after 50000 transaction even if the total time has not been totaly spent. The requests made by AB have a size of 100 bytes and the responses by the Apache server have a size of 170 bytes.

Figure 3.7 shows that the number of transaction with UDP and TCP using apache benchmark. The number are quite bigger than the one from the previous tests TCP_CRR. The maximum is 1800 transaction per second when using UDP and a RTT of 2ms. The minimum is 6 transaction per second when using TCP and having a RTT of 400ms.

You can see that the links maximum bandwith don't have a big impact on the number of transaction per second, it is mainly the RTT. In this test UDP still dominate even with the concurrent connections being used.

It is the same as TCP_CRR test, the CPU and link bandwith are not the bottle neck,

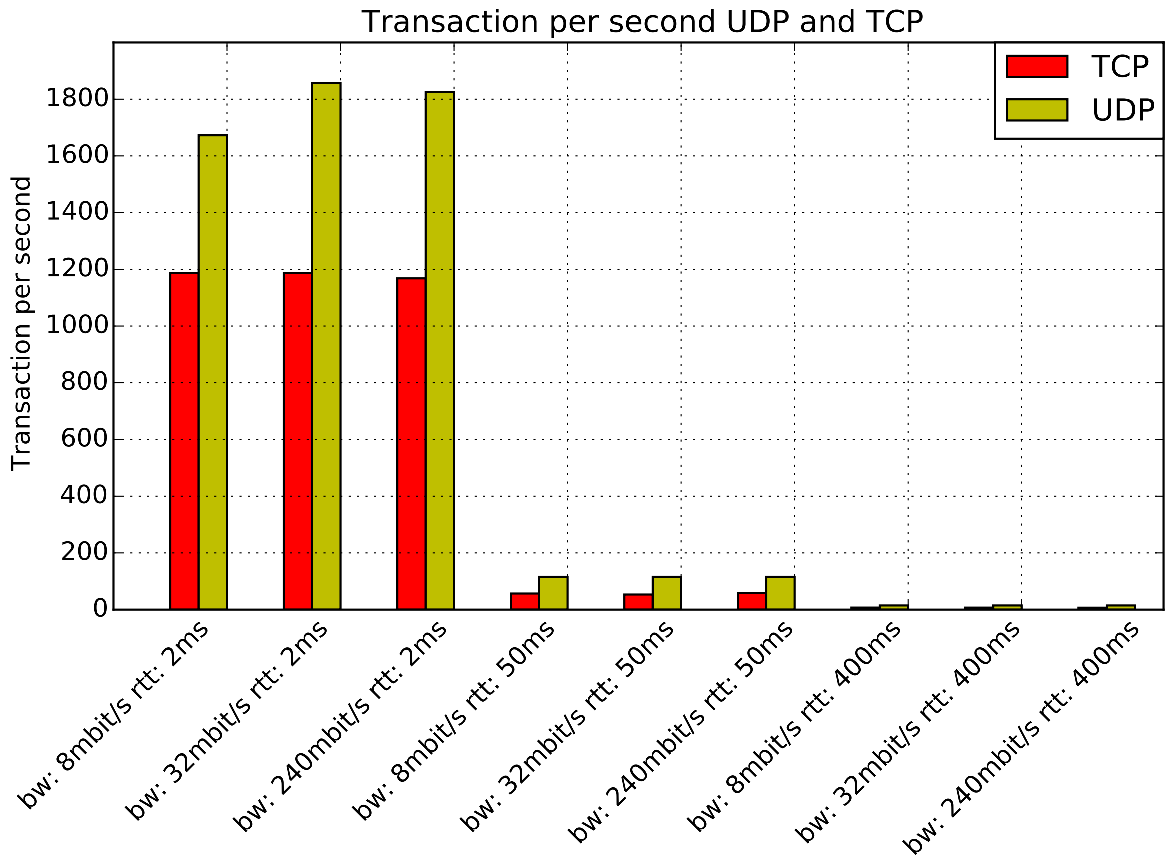

Figure 3.7: Request/Response Apache benchmark

the CPU usage of the router is around 25-30% during the tests. The problem is the time it takes to establish a connection when using TCP.

In conclusion TCP gives reasonnable performances compare to UDP and I think could be use without impacting too much on the user experience.

Lets look at what happen if the main link fail during a certain amount of time and see if this time MPTCP has the advantage.

The test length is 30 second and the main link will be set to 100% packet lost after 5 second and then set back to 0% after 15 seconds. The links RTT are 2ms and max bandwidth is 8mbit/s and 32mbit/s.

On figure 3.8, we see that this time the difference between the transaction per second for UDP and TCP is smaller because TCP continue to transmit data while UDP is stopped.

If you insert more down time on the link this behavior will be even more visible and

UDP will be slower than TCP. This is because MPTCP switch the traffic to another subflow when it see that the one it uses start to have timeout, UDP however cannot do that and will not be able to transmit data during this time.

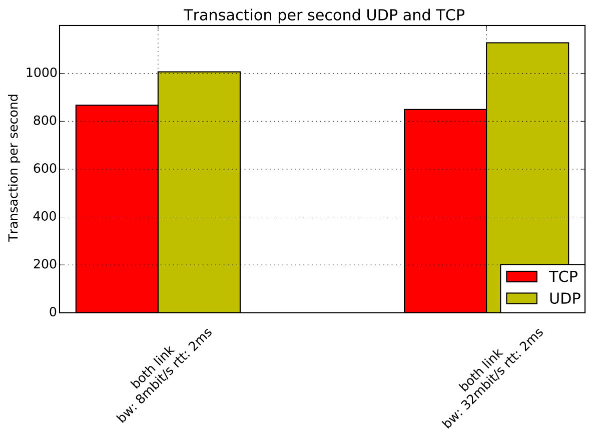

Figure 3.8: Request/Response Apache benchmark one link down

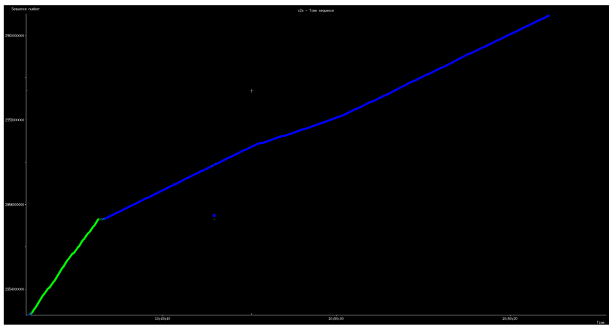

Figure 3.9: Request/Response Apache benchmark one link down subflow details

Look at figure 3.9, the sequence number graph of the MPTCP connection, You can see the switch from the main subflow (the green line) to the alternative subflow (the blue line) when the link goes down.

3.2 Links with different bandwidth and delay

For these test I will go back to builk data transfer test and I will look at the impact on the throughput when using links with different bandwith and delay, for these tests I will only use TCP has OpenVPN protocol. UDP would not be very interesting because it only use one link.

This is interesting because we will see how MPTCP decide to split traffic on the two subflows depending on their RTT and their bandwidth.

3.2.1 First test setup

- First link bandwith of 32mbit/s and RTT: 2ms (default route)
- Second link bandwith of 8mbit/s and a RTT: 2ms
- Using default mptcp scheduler
- OpenVPN buffer set to 64000 byte (TCP windows size) on client and server
- Congestion control : cubic

In theory we would like to get around 40mbit/s of throughput. In practice we get a 35mbit/s througput, the router is not the bottleneck the cpu is utilized around 60%.

More test will be using this setup in section 4.2

3.2.2 Second test setup

- First link bandwith of 32mbit/s and RTT: 2ms (default route)
- Second link bandwith of 8mbit/s and a RTT: 400ms
- Using default mptcp scheduler
- OpenVPN buffer set to 400000 byte (TCP windows size) on client and server
- Congestion control : cubic

In theory we could expect a throughput around 40mbit/s which is the sum of the two link Bandwith.

In practice MPTCP will only use the link with the lowest RTT and it will not use the second link, this gives a throughput around 30mbit/s.

To see how the traffic was split accross the two subflows I used mptcptrace to generate the sequence number graph and script to generate figure 3.10. In this graph you can see that the traffic is mainly on first subflow (first link), the green line. The second subflow (second link) is sometime use to send some packet and if we look at the sequence graph we see that because of the high RTT on the second subflow, packets send from this subflow, will be reinjected on the first subflow.

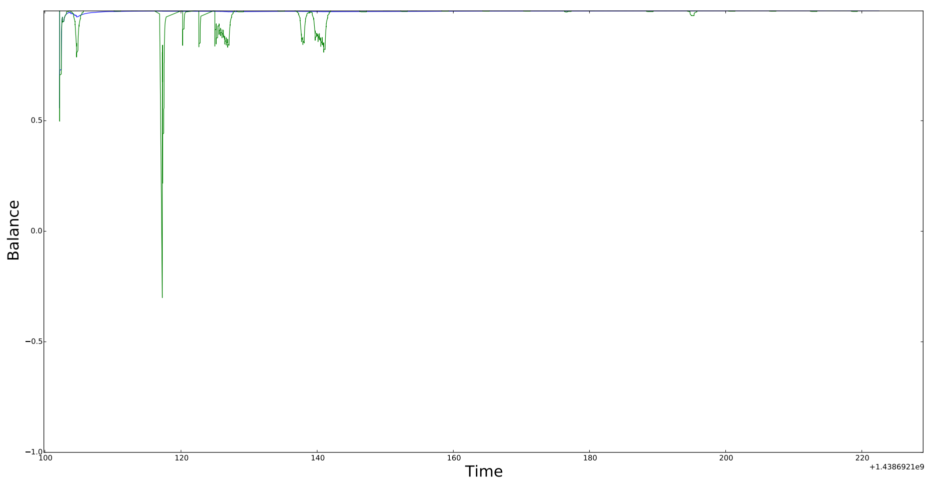

Figure 3.10: Subflow usage graph second test setup

3.2.3 Third test setup

- First link bandwith of 32mbit/s and RTT: 400ms (default route)
- Second link bandwith of 8mbit/s and RTT: 2ms
- Using default mptcp scheduler
- OpenVPN buffer set to 1600000 byte (TCP windows size) on client and server
- Congestion control : cubic

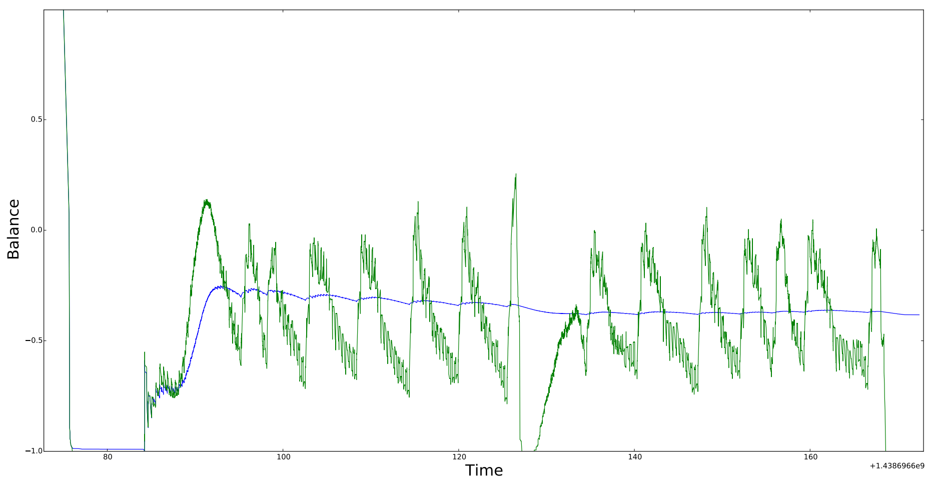

Figure 3.11: Subflow usage graph third test setup

Here mptcp use both links but the throughput is only around 8mbit/s. On figure 3.11 you can see that it use the second subflow (second link) in blue and it also use the first subflow (first link) quite often.

If we look at the sequence graph we see that it retransmit some packet from the first subflow to the second subflow because of the high RTT of the first subflow.

In conclusion for some of these setups MPTCP is not optimal and there is room for improvement. In section 4.2 and 4.3 I will use differents congestion control algorithms and the roundrobin mptcp scheduler and see if it can help.

CHAPTER 4

IMPROVING PERFORMANCE

4.1 Improving throughput by changing buffer size

In test we saw that using the same bandwith but introducing high delay on links gives decreave the throughput by a significant amount. This is likely because of wrong buffer size, TCP window size, in this section we will try to adjust buffer size and see if it improve. I will only use TCP as the openvpn protocol because UDP is not my main focus here.

To adjust the TCP maximum send and receive window size, I will set value of *sndbuf* and *rcvbuf* in the openvpn configuration. To compute the size I will be using the formula :

$$(2 * delay * bandwidth) * 2$$

.

First I will try on one specific setup from test 4.1, the 8mbit/s bandwidth limit and 400 RTT.

Now lets use a buffer size of 800000 bytes (390 kbytes) instead of the default one used by openvpn (64kbytes)

In order to do that we must modify the openvpn configuration file of client and server see 4.1:

```
  proto tcp
  sndbuf 800000
  rcvbuf 800000
```

Listing 4.1: set buffer size openvpn configuration

And modify the maximum buffer size that applications can request by changing /proc/sys/net/core/rmem_max and /proc/sys/net/core/wmem_max on the onpenvpn client and openvpn server [15]. You might also have to change the /proc/sys/net/ipv4/tcp_wmem and /proc/sys/net/ipv4/tcp_rmem if they are not big enough.(see 4.2)

```
sysctl -w net.core.wmem_max=800000
sysctl -w net.core.rmem_max=800000
sysctl -w net.ipv4.tcp_rmem='4096 87380 6291456'
sysctl -w net.ipv4.tcp_wmem='4096 16384 4194304'
```

Listing 4.2: set max buffer size

This gave me a 10mbit/s throughput and an average RTT of 450 (see figure 4.1), this is way better than the 2mbit/s throughput seen on figure 3.3.

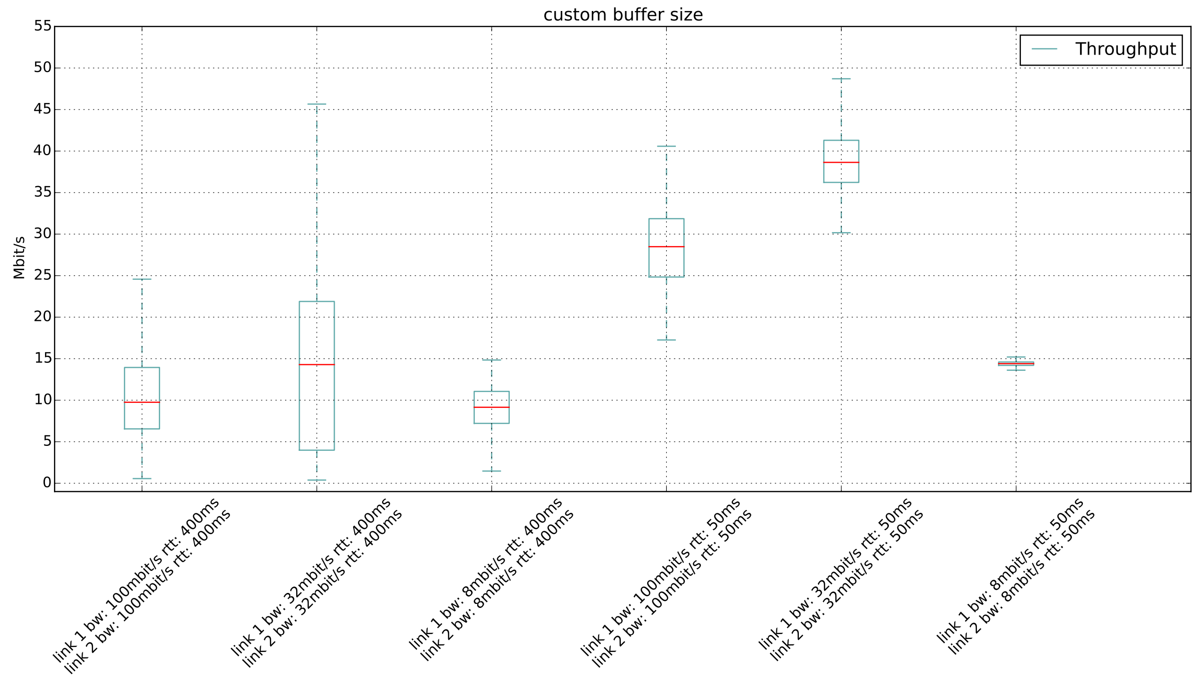

Figure 4.1: TCP custom buffer size

On figure 4.1 you see the same technique applied for 32mbit/s and 100mbit/s. It is interesting to see that it improve the throughput in all situation but it does not equals the result obtained with a 2ms RTT. I have tried many different buffer sizes but I was unable to match it.

There is also some unexpected results like the 100mbit/s links with a RTT of 50 only giving a throughput around 30 mbit/s. To but sure I have looked at the sender window but it did not fill the receiver window, this was not the problem. Analysing the capture with MPTCPTrace I saw on the sequence graph some ̈flaẗincrease suggesting that packet losses were happening. Using TCPTrace I looked at the encapsulated connection and indeed there was some packet losses which could be the reason for the window not to increase.

Looking at the packet capture I saw that the more higher RTT the more retransmissions were happening which implies a decrease in the throughput.

Another very interesting thing is that we only see the retransmissions happening in the encapsulated TCP connection. Because we run TCP over TCP, we have a upper layer which is the OpenVPN packet supporting MPTCP and a lower layer which is the traffic from the computer client with normal TCP encapsulated in the upper layer.

The retransmission could happen in both of the layer but here we observe that it only happen in the lower layer. This is good in the sense that you don't retransmit multiple times on the different layers meaning that you don't have the tcp meltdown effect [17] but this also tell us that the upper layer, the MPTCP connection is limited by the lower layer.

This is illustrated in figure 4.2 and figure 4.3. Figure 4.2 is the sequence graph generated with MPTCP on the upper layer TCP connection, on this figure you can see one that the line is pretty flat between time 105 and 120. Figure 4.3 is the sequence graph generated by tcptrace on the lower layer TCP connection at time 105, on this figure you can see a lot of SACK and some retransmissions, meaning that you have packet losses at this time.

4.2 Improving throughput by changing congestion control algorithm

In this test I will try switching between different congestion control algorithms. I will use cubic, lia, olia and wvegas advised by Olivier Bonaventure and Benjamin Hesmans.

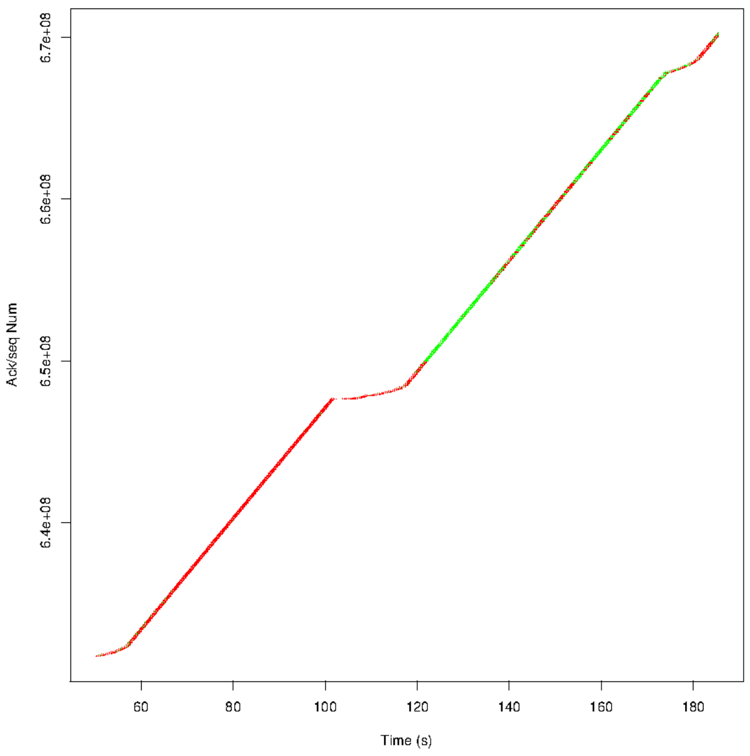

Figure 4.2: Sequence graph of the upper layer

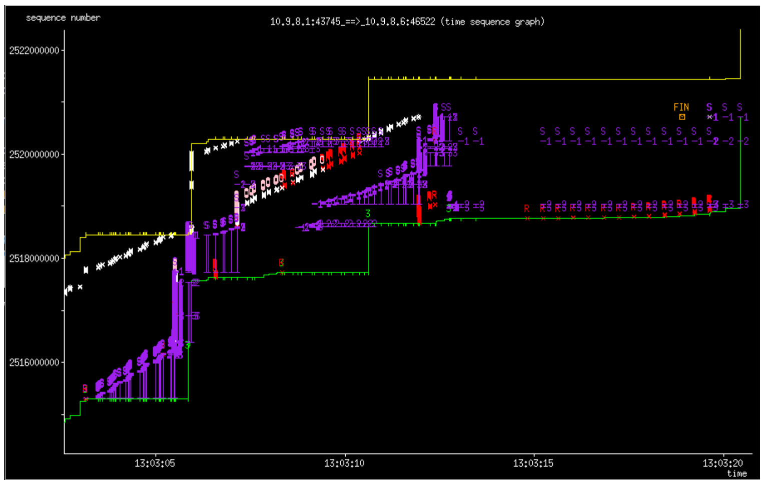

Figure 4.3: Sequence graph of the lower layer at time 105

First I will take the setup from the first test fo section 3.1.1.

On figure 4.4 ou can see that they all give similar results expect wvegas which use both subflow but keep the maximum throughput equivalent to the maximum throughput of only one link.

Second I will use setups from test 3.2.1 and 3.2.3 to see if the algorithms can make a better use of the two subflows when they have different bandwith and rtt.

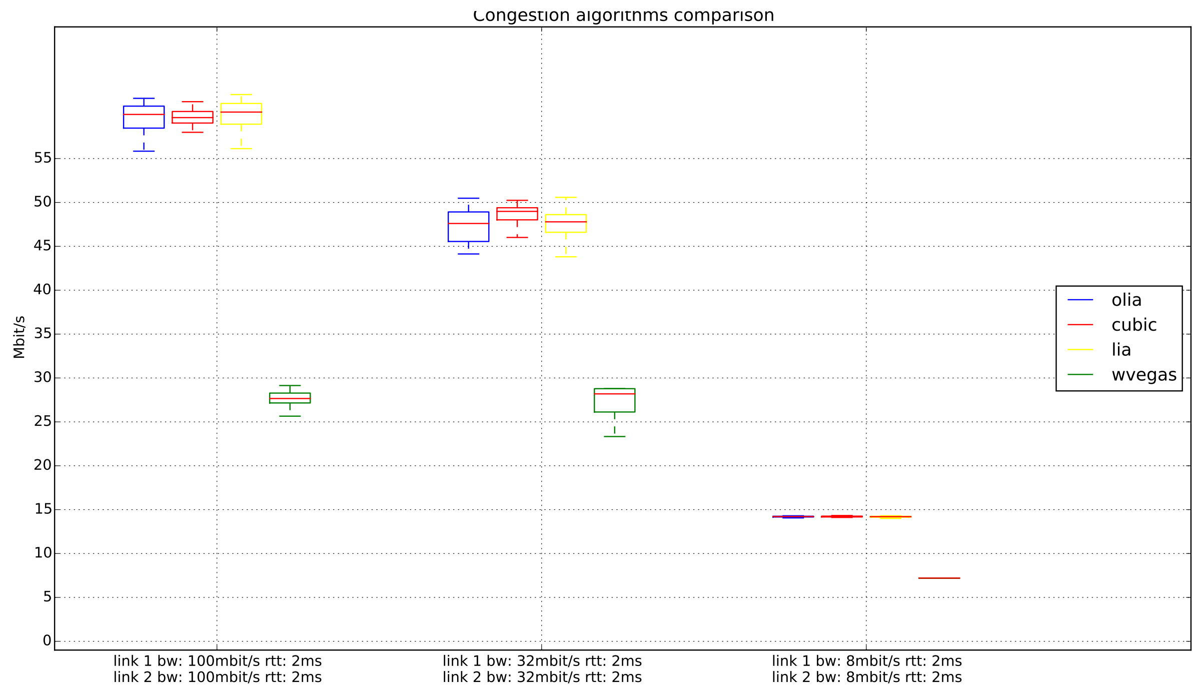

Figure 4.4: Congestion algorithms comparison

Figure 4.6 show that for setup 3.2.1 the throughput is the almost the same accross all the algorithms except it is a bit lower for wvegas and olia. This is likely because they are more fair to normal TCP compare to the others.

For setup 3.2.3, the wvegas algorithm does not provide a very good throughput.

From paper [4], the autors says that wvegas algorithm is not very good with High Delay Bandwith product link. The cause is the slow start phase of wvegas being very short and switching to congestion avoidance very early. I tried using a single link with a High delay and High bandwidth and it gave me a very low throughput about 100kbit/s on a link with a bandwidth of 32mbit/s and a RTT 400. Another thing wvegas seems to have a hard time with is link with different RTT, it will usually pick only the link having the lowest RTT and not use the link with the Highest RTT even if it is not a huge RTT.

To see what happened with wvegas I analyzed the capture with mptcptrace and tcptrace. mptcptrace showed that both subflow were used but the one with the lowest RTT was use more often. Another interesting thing is that the TCP windows on the sender side was very small which would explain why we get this very low throughput (see figure 4.5)

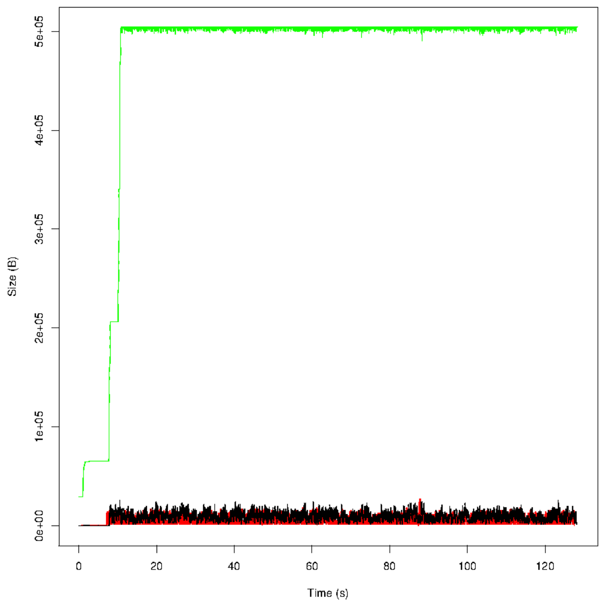

Figure 4.5: Sender windows size with wVegas

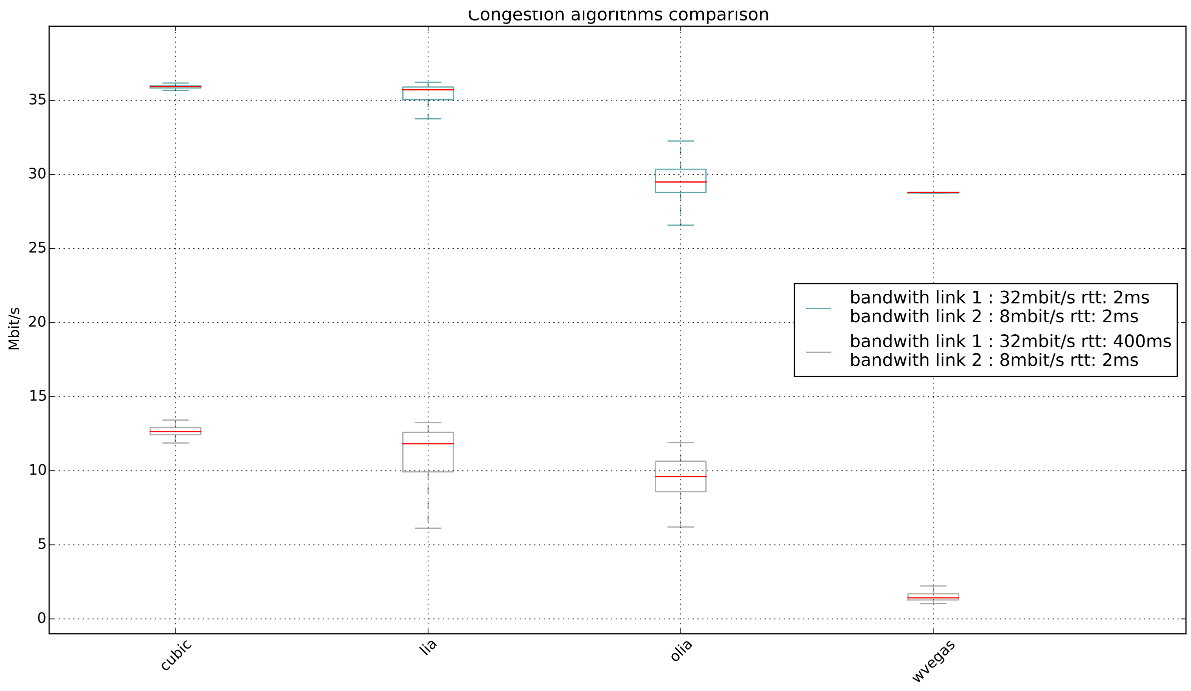

Figure 4.6: Congestion algorithms comparison

4.3 Improving throughput by changing mptcp scheduler

In test 3.2.3, we saw that because MPTCP scheduler choose the subflow with the lowest RTT and from time to time use the subflow with the high RTT. Lets try to force MPTCP to use the High RTT subflow.

For that I will change the MPTCP scheduler to roundrobin and look at the effect on the overall throughput. Roundrobin scheduler does not select subflow based on the RTT, it use each subflow one after the other see section 1.2 for more information. Because of that I hope the link with a high RTT will be use and the throughput will increase.

The RoundRobin will use one as the num_segments parameter, number of consecutive segments that should be sent on one subflow and the parameter cwnd_limited will be set to true, the default value, the scheduler tries to fill the congestion window on all subflows.

The throughput with roundrobin is around 14mbit/s, on figure 4.7 you can see how the traffic was split across the two subflows.

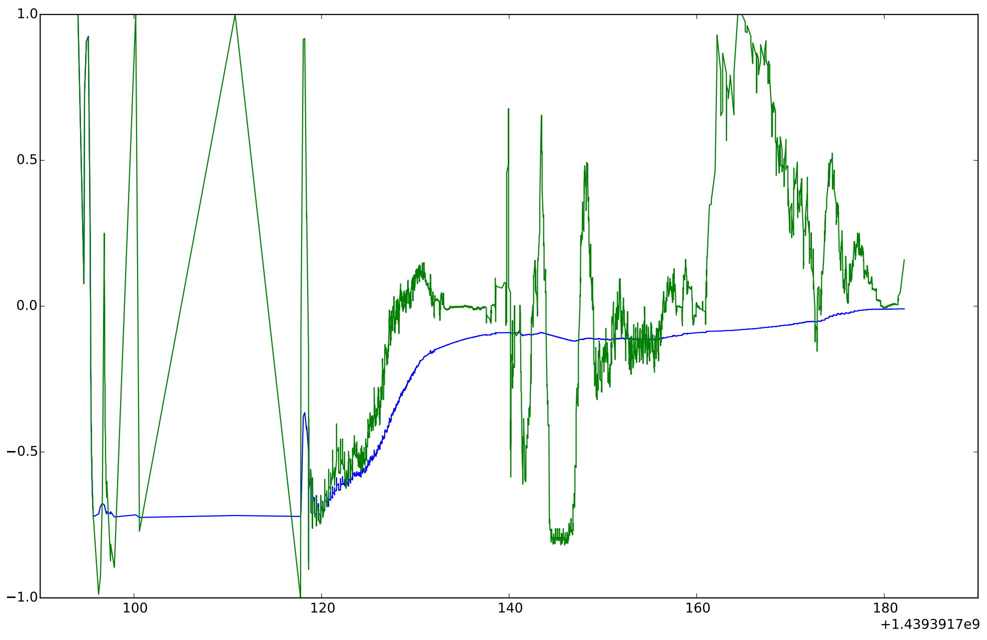

Figure 4.7: Roundrobin scheduler

CHAPTER 5

TOOLS

5.1 Measurement tools

Here are the main tools used to gatter and validate test data.

- netperf and flent (netperf-wrapper) to measure the throughput
- netperf and apache benchmark (AB) to measure request/response performance
- mpstat to check the CPU usage of the router and server
- wireshark and tcpdump to capture packets
- mptcptrace to analyse pcap

5.1.1 netperf [10]

Netperf is a tool measuring network performance, compatible with TCP and UDP protocol. It has a client and a server, the server is netserver and listen to any client request and the client is used to iniate network test with the server.

When you execute netperf, the first thing that will happen is the establishment of a control connection to the remote system. This connection will be used to pass test configuration information and results to and from the remote system. Regardless of the type of test being run, the control connection will be a TCP connection using BSD sockets.

Once the control connection is up and the configuration information has been passed, a separate connection will be opened for the measurement itself using the APIs and protocols appropriate for the test. The test will be performed, and the results will be displayed.

Netperf places no traffic on the control connection while a test is in progress

5.1.2 flent (netperf-wrapper) [18]

Netperf-wrapper, freshly called flent is a Python wrapper to run multiple simultaneous netperf/iperf/ping instances and aggregate the results. Tests are written in config files, you can run test several time using iterated tests. The results are saved into a json format so you can process them later and you can also choose between multiple format.

It can also generate plots like plot box and cdf into different formats with the help of matplotlib. And you can write a batchfile to automate test execution, I calls setup and teardown scripts, execute tests automaticly and save the result into files. This feature is very powerfull and I used it to perform multiple tests and adapt the bandwidth limit / delay on links with the help of delayboxes.

5.1.3 mpstat

I used mpstat to check the CPU usage on the router. I wanted to identify the bootleneck which was limiting the bandwitdh and indeed it was using 100% of the CPU when transmitting data throw the VPN.

5.1.4 Wireshark and tcpdump

Tcpdump have been use to sniff packets that will later be analyse with mptcptrace and tcptrace.

Wireshark have been used make sure that the MPTCP protocol is in used during the tests, looking at MPTCP options like MP_CAPABLE option, ADD_ADDR option and MP_JOIN.

5.1.5 mptcptrace [7]

I used mptcptrace to analyse pcap captured by tcpdump on the server side and get more information about the MPTCP connection. I have used many of the graph produced by mptcptrace :

- The sequence graph to get a idea on how the traffic was distributed accross the different subflows and see if packets losses were happening
- The goodput graph to see how many goodput we could get and cross verify with the one from netperf.
- The flight and flight per flow graph to see the size of the TCP window

5.1.6 tcptrace [16]

To analyse the captured packets in more detail, I also used tcptrace. It was very helpfull to look at the sequence number graph of each subflow and see the packet lost on capture from the tun interface.

CHAPTER 6

CONCLUSION

In conclusion providing a robust and fast internet connection to compagnies and individual using MPTCP coupled with OpenVPN is something possible.

The setup is minimal and not that expensive you only need two or more internet connection, a router running a MPTCP compatible kernel for the OpenVPN client and a server running a MPTCP compatible kernel for the OpenVPN server and if possible with a bandwith equal to the sum of all your internet connection.

The configuration can be a bit more complicated expsecially if you want to achieve good performances and try to maximize the speed of your internet connection.

In term of performance, you will need hardware that is powerfull enough otherwise the throughput will be limited by OpenVPN and its intensive use of the CPU to encapsulate and decapsulate packets. Here with the TP-LINK N900 router the maximum throughput I could reach before the CPU was saturated was 57mbit/s, with nowdays internet connection going up to 100mbit/s, this limit is not very high.

In this paper we saw that using OpenVPN with MPTCP on link with low bandwith delay product gives good performance however we saw that on link with high bandwith delay product the performance are weak. In some cases the use of MPTCP is not even worth it, UDP as the OpenVPN protocol would be a better choice in term of performance if we ommit the fact that it would less robust against failure.

I have investigated and tried to solve the problem concerning the high bandwith delay product link with some level of success by adjusting the TCP windows size on both openvpn end side and trying different congestion algorithms. This increased performances in every cases but depending on the configuration the increase was more or less significant.

This said you will likely want your internet connections to have a small RTT between your openvpn client and your openvpn server, normal VDSL or ADSL connection should be fine but 3G or sattelite internet connection will not be adequate for that kind of setup.

All the tests in this paper were done in a closed lab environnement, an interesting thing to do would be trying that in the real world environnement over the internet to see how it behave. Finally another interesting thing would be to try using a different technology than OpenVPN and see if it would give better performances.

BIBLIOGRAPHY

[1] Alexander Afanasyev et al. “Host-to-host congestion control for TCP”. In: *Communications Surveys & Tutorials, IEEE* 12.3 (2010), pp. 304–342.

[2] Sébastien Barré, Christoph Paasch, and Olivier Bonaventure. “Multipath TCP: from theory to practice”. In: *NETWORKING 2011*. Springer, 2011, pp. 444–457.

[3] Richard Blum. *Network Performance Open Source Toolkit: Using Netperf, tcptrace, NISTnet, and SSFNet*. John Wiley & Sons, 2003.

[4] Yu Cao, Mingwei Xu, and Xiaoming Fu. “Delay-based congestion control for multipath TCP”. In: *Network Protocols (ICNP), 2012 20th IEEE International Conference on*. IEEE. 2012, pp. 1–10.

[5] Yung-Chih Chen and Don Towsley. “On bufferbloat and delay analysis of multipath TCP in wireless networks”. In: *Networking Conference, 2014 IFIP*. IEEE. 2014, pp. 1–9.

[6] Martin Devera. *HTB LINUX queuing discipline manual–user guide*. 2004.

[7] Benjamin Hesmans and Olivier Bonaventure. “Tracing Multipath TCP Connections”. In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM ’14. Chicago, Illinois, USA: ACM, 2014, pp. 361–362. ISBN: 978-1-4503-2836-4. DOI: 10.1145/2619239.2631453. URL: http://doi.acm.org/10.1145/2619239.2631453.

[8] Osamu Honda et al. “Understanding TCP over TCP: effects of TCP tunneling on end-to-end throughput and latency”. In: *Optics East 2005*. International Society for Optics and Photonics. 2005, 60110H–60110H.

[9] *MPTCP website.* `https://github.com/multipath-tcp/mptcp`.

[10] *netperf.* `http://www.netperf.org/netperf/`.

[11] *OpenVPN.* `https://community.openvpn.net`.

[12] *OpenWRT wiki.* `http://wiki.openwrt.org`.

[13] Christoph Paasch et al. "Experimental evaluation of multipath TCP schedulers". In: *Proceedings of the 2014 ACM SIGCOMM workshop on Capacity sharing workshop.* ACM. 2014, pp. 27–32.

[14] *TBF Figure.* `http://unix.stackexchange.com/questions/100785/bucket-size-in-tbf`.

[15] *TCP Tuning.* `http://www.psc.edu/index.php/networking/641-tcp-tune`.

[16] *TCPTrace.* `http://www.tcptrace.org`.

[17] Olaf Titz. *Why TCP Over TCP Is A Bad Idea.* `http://sites.inka.de/bigred/devel/tcp-tcp.html`. 2001.

[18] Toke Høiland-Jørgensen. *The FLExible Network Tester (flent).* `https://flent.org`. Karlstad University.

[19] *TP-LINK N900.* `http://wiki.openwrt.org/toh/tp-link/tl-wdr4900`.

[20] VantagePoint. "Analysis of Satellite-Based Telecommunications and Broadband Services". In: VantagePoint. 2013.

APPENDIX A

ANNEXES

Everything can be found in a git repository at this address : `https://github.com/alokhan/memoire`