Télécom ParisTech
Promotion 2017
Sylvain DASSIER

RAPPORT DE STAGE

Étude de l'apport du protocol MPTCP dans l'optimisation du trafic

Département :	***Département d'Informatique***
Option :	***INFRES***
Encadrants :	***M. Luigi IANNONE, M. Antoine FRESSANCOURT***
Dates :	***18/07/2016 - 17/01/2017***
Adresse :	***Télécom ParisTech, 23 Avenue d'Italie, 75013 Paris***

Declaration d'intégrité relative au plagiat

Je soussigné DASSIER Sylvain *certifie sur l'honneur :*

1. Que les résultats décrits dans ce rapport sont l'aboutissement de mon travail.
2. Que je suis l'auteur de ce rapport.
3. Que je n'ai pas utilisé des sources ou résultats tiers sans clairement les citer et les référencer selon les règles bibliographiques préconisées.

Je déclare que ce travail ne peut être suspecté de plagiat.

17 janvier 2017 Signature : Sylvain Dassier

Abstract

In urban developed areas characterised by sustainable economic growth and high quality of life, we have seen high levels of investment in the domain of vehicle to vehicle and vehicle to infrastructure technologies. CarFi is a project that deals with connected vehicles and allows them to take advantage of Wifi hotspots in addition to 2G, 3G or 4G to connect to the Internet while on the move. During this internship we have tried to look at the solutions provided by the MPTCP protocol in order to maintain perpetual connection to the Internet. A part of our work is dedicated to putting in place a debugging system for the better comprehension of the MPTCP protocol. We have particularly looked at the enhanced socket API for Multipath TCP developed by L'Université Catholique de Louvain. This API allows us to manipulate sub flows from the Application Layer. In fact, we have put in place a testbed with the application Netcat to verify the functioning of the API according to our needs. Finally we have tried to evaluate the performance of our developed netcat-mptcp.

Résumé

Dans les villes urbaines et dévéloppées, caractérisées par une croissance économique stable et une qualité de vie élevée, nous avons constaté un grand nombre d'investissements dans le domaine des technologies véhicule à véhicule et véhicule à infrastructure. CarFi est un projet qui inclut les véhicules connectées et qui leur permet de profiter des avantages des hotspots Wifi en complément des réseaux 2G, 3G ou 4G pour se connecter à Internet en étant en mouvement. Pendant ce stage nous avons essayé de voir les solutions proposées par le protocol MPTCP pour maintenir une connexion internet perpetuelle. Une partie de notre travail est dédiée à la mis en place d'un système de déboggage pour mieux comprendre le protocol MPTCP. Nous avons en particulier regardé une API basée sur MPTCP et dévéloppée par l'Université Catholique de Louvain. L'API nous a permis de manipuler les sous flux de la couche application. Nous avons également mis en place un banc d'essai avec l'application Netcat pour vérifier le bon fonctionnement de l'API selon nos besoins. Par ailleurs, nous avons évalué la performance de notre solution proposée (netcat-mptcp).

Table des matières

1 Introduction

1.1 Context

Today, connected vehicles make use of 2G, 3G or 4G networks in order to connect to the Internet while in motion. Whether it be for GPS, simple browsing, emergency calls, e-calls or music, every consumer has his/her own needs.

Apart from the usual connection glitches, such connectivity is rather expensive with limited bandwidth. Even though workarounds have been implemented, most of them are either inefficient or are not completely transparent. These limitations stand in the way of developement of connected vehicles.

In fact, there has been a lot of research work in present times on vehicle to vehicle and vehicle to infrastructure technologies [1, V2x]. Car manufacturers and infrastructure suppliers are investing in such projects to keep up with the growing demand. However we are still waiting for a concrete outcome which is immediately deployable. Our project **CarFi** may play the role of an intermediary as something functional but which can be further developed with these vehicle to vehicle and vehicle to infrastructure technologies [2, 3, V2x].

MultiPath TCP (MPTCP) is an effort towards enabling the simultaneous use of several IP-addresses/interfaces by a modification of TCP. It presents a regular TCP interface to applications, while in fact spreading data across several subflows. Benefits of this include better resource utilisation, better throughput and smoother reaction to failures. The **CarFi** project aims to exploit these advantages of MPTCP. A potential add-on would be the usage of the WiFi network when available. Most urban areas are covered via Mobile Network Operator or ISP WiFi hotspots. One may envisage a scenario where the default connection is established over Wifi and when it is no longer available, the communication carries on over 3G.

Our Objective : To develop a functional prototype for the **CarFi** project. This prototype is based on the **Enhanced Socket API for Multipath TCP** [4, API]. It ensures continuous connectivity to the network, keeping certain channels like the 3G as pivots/fallbacks and others like the Wifi to join and leave. It may also be used for optimum usage of network resources while using all the channels available.

1.2 Document Outline

This document is divided into two main parts comprising different sections. The first part involves section 2 where we describe how to set up a ***debugging environment for MPTCP***. This will help us to follow the different system calls during the establishment of a flow or a sub-flow. The next sections form the other part, dealing with the new socket API that enables us to control the MPTCP stack from user space. Section 3 gives a description of the socket API along with some details on its implementation. Section 4 elaborates a use case of this API, in our case a **Netcat** with **MPTCP**. Section 5 summarises our results 5.1, elucidates certain statistics 5.2 and mentions some of the underperformances and anomalies 7 encountered.

PART I

2 Setting up a debugging environment for MPTCP :

In order to understand the different stages of running of the MPTCP linux kernel, we have put in place a debugging environment. In fact, after doing a little bit of research, we have come across various methods for debugging a protocol. Solutions like **Eclipse + GDB**, **NS3 + DCE for in-kernel protocol implementation** [5, NS3+DCE], **net-next-nuse** [6, net-next-nuse] or **net-next-sim** [7, net-next-sim] exist. In any case, all debugging systems involving a protocol implementation are very complex.

We have chosen and succesfully implemented [8, LibOS] (an MPTCP version of the library operating system of the linux kernel) with [9, DCE] (Direct Code Execution). A library version of a protocol is faster and easier to compile and put in place, hence the choice. Everything was done on a XUbuntu 14.04 64bit virtual machine with DCE 1.8. During the setting up of the debugging system we received the valuable help from **M. Matthieu Coudron** of **L'Université Pierre-et-Marie-Curie**. The discussions can be found here : [10, ns-3-dce]. The following illustrates how to set up the debug environment :

1. **Install the dependencies :**
   Since we are doing everything on a virgin operating system, we need to install certain dependencies to make things work. These dependencies are essential for the different components of the debugging system viz. **NS3**, **DCE** etc. to run properly and collaborate with one another.

```
sudo apt-get install vim git mercurial gcc g++ python python-dev
    qt4-dev-tools libqt4-dev bzr cmake libc6-dev libc6-dev-i386
    g++-multilib gdb valgrind gsl-bin libgsl0-dev libgsl0ldbl
    flex bison libfl-dev tcpdump sqlite sqlite3 libsqlite3-dev
    libxml2 libxml2-dev libgtk2.0-0 libgtk2.0-dev vtun lxc
    uncrustify doxygen graphviz imagemagick texlive
    texlive-extra-utils texlive-latex-extra texlive-font-utils
    dvipng python-sphinx dia python-pygraphviz python-kiwi
    python-pygoocanvas libgoocanvas-dev ipython
    libboost-signals-dev libboost-filesystem-dev openmpi-bin
    openmpi-common openmpi-doc libopenmpi-dev libncurses5-dev
    libncursesw5-dev unrar unrar-free p7zip-full autoconf
    libpcap-dev cvs libssl-dev wireshark
```

2. **Build DCE using bake :**
   The easiest way to build **DCE** is to use **bake** [11, bake]. It automates the building process by handling dependencies, downloading required sources, correctly building required modules, providing off-line installation and build capabilities and providing the possibility to configure according to one's needs.

```
hg clone http://code.nsnam.org/bake bake
export BAKE_HOME=`pwd`/bake
export PATH=$PATH:$BAKE_HOME
export PYTHONPATH=$PYTHONPATH:$BAKE_HOME
mkdir dce
cd dce
bake.py configure -e dce-ns3-1.8
bake.py download
bake.py build
```

3. **Download and build the *mptcp_trunk_libos* branch of *net-next-nuse* :**
   Now we need the **mptcp_trunk_libos** of **net-next-nuse**. This trunk contains the source code of **MPTCP** that we can modify. We may include the modules that are needed by using *make menuconfig*. Once we build the shared library, we obtain the file *liblinux.so* which **DCE** loads by default. This is however not the correct library for **DCE** as it searches for the function *sim_init* unavailable in *liblinux.so* (which rather has the initiation function as *lib_init*). The correct shared library however is *libsim-linux.so* found at *$HOME/net-next-nuse/arch/lib/tools* which contains the *sim_init* function required by **DCE**. Hence we try to mislead **DCE** by renaming the current *liblinux.so* to *liblinux0.so* and by creating a symbolic link for *libsim-linux.so* under the name *liblinux.so* in the default **DCE** search folder *$HOME/net-next-nuse*.

```
git clone -b mptcp_trunk_libos
    https://github.com/libos-nuse/net-next-nuse.git
cd net-next-nuse
make menuconfig ARCH=lib
```

```
make library ARCH=lib
```

Now we rename *liblinux.so* to *liblinux0.so* and create the symbolic link for *libsim-linux.so* as follows :

```
ln -s $HOME/net-next-nuse/arch/lib/tools/libsim-linux.so
    $HOME/net-next-nuse/liblinux.so}
```

4. **Download *DCE 1.8* :**
   We now download the source code of **DCE** in the **$HOME** directory. We have found the compatible version to be **dce-1.8**.

   ```
   hg clone http://code.nsnam.org/ns-3-dce -r dce-1.8
   ```

   DCE 1.8 is found in the folder *$HOME/ns-3-dce*

5. **Build *iproute2* version *2.6.38* :**
   For **DCE** to function correctly, we need the correct version of one of it's dependency *iproute*. For us it is the version *iproute2-2.6.38*. Hence we need to dowload the correct source code and apply the patch avaiable at *$HOME/ns-3-dce/utils/iproute-2.6.38-fix-01.patch*.

   Download the compressed source code from : *https://kernel.googlesource.com/pub/scm/linux/kernel/git/shemminger/iproute2/+archive/fcae78992cab7bd267785b392b438306c621e583.tar.gz*
   Now extract it and rename the folder to *iproute2-2.6.38*.
   Next we apply the patch as follows :

   ```
   cd iproute2-2.6.38
   patch -p1 -i ../ns-3-dce/utils/iproute-2.6.38-fix-01.patch
   ```

   Now the $(KERNEL_INCLUDE) variable in the *Config* section of the *Makefile* of *iproute2-2.6.38* should point to the liblinux.so directory (for us it is $HOME/net-next-nuse). Hence we modified the following part in the *Makefile* :

   ```
   Config:
      sh configure /home/lawrence/net-next-nuse
      # sh configure $(KERNEL_INCLUDE)
   ```

   Finally we make *iproute2-2.6.38* as follows :

   ```
   cd iproute2-2.6.38
   LDFLAGS=-pie make CCOPTS='-fpic -D_GNU_SOURCE -O0
       -U_FORTIFY_SOURCE'
   ```

6. **Set the *$DCE_PATH* variable :**

   We need to adjust the *$DCE_PATH* so as to indicate to **DCE** where to look for the shared libraries and executables.

   ```
   export
      DCE\_PATH=\$HOME/net-next-nuse:\$HOME/iproute2-2.6.38/ip
   ```

7. **Build *DCE 1.8* :**

   ```
   cd ns-3-dce
   ./waf configure --with-ns3=$HOME/dce/build
      --enable-kernel-stack=$HOME/net-next-nuse/arch
      --prefix=$HOME/dce/build
   ./waf build
   ```

8. **Run *dce-iperf-mptcp* with or without *GDB***

   ```
   cd ns-3-dce
   ```

   Without **GDB** :

   ```
   ./waf --run dce-iperf-mptcp
   ```

   With **GDB** :

   ```
   ./waf --run dce-iperf-mptcp --command-template=‘‘gdb --args
      %s’’
   ```

   Once we enter the *GDB* prompt we must put a breakpoint at one of the functions in the *mptcp* folder to stop there. Kindly refer to the files found at *$HOME/net-next-nuse/net/mptcp* to choose the function to define as a breakpoint.
   An example :
   Suppose we would like to stop the execution at the function *mptcp_set_default_path_manager()* found at *$HOME/net-next-nuse/net/mptcp/mptcp_pm.c*, then we give the following command at the *GDB* prompt :
   b mptcp_set_default_path_manager
   GDB will ask the following :
   Function "mptcp_set_default_path_manager" not defined.
   Make breakpoint available on future shared library load ? (y or [n])
   Type in ***y*** and press enter. We may run the program by typing ***r*** and then pressing enter. *GDB* will pause at the necessary breakpoint.

PART II

3 An Enhanced socket API for Multipath TCP :

In the CarFi project, we would like to control the MPTCP kernel stack from the application layer i.e. manage(open/close) sub flows according to the kind of application that uses it and its requirements. For example, for a streaming application it is preferable to communicate over the Wifi channel. In the current Linux Kernel implementation of MPTCP, the path managers may not be fit for all kinds of applications. For optimum usage, advanced applications may want to know the number of sub flows available or the state of the active sub flows. When the application possesses such information it may want to create a new sub flow, terminate an existing one, change a sub flow's priority etc.

3.1 Implementation

The enhanced socket API has been implemented over the existing ***getsockopt()*** and ***setsockopt()*** system calls. The following figure illustrates the MPTCP socket structure [4] :

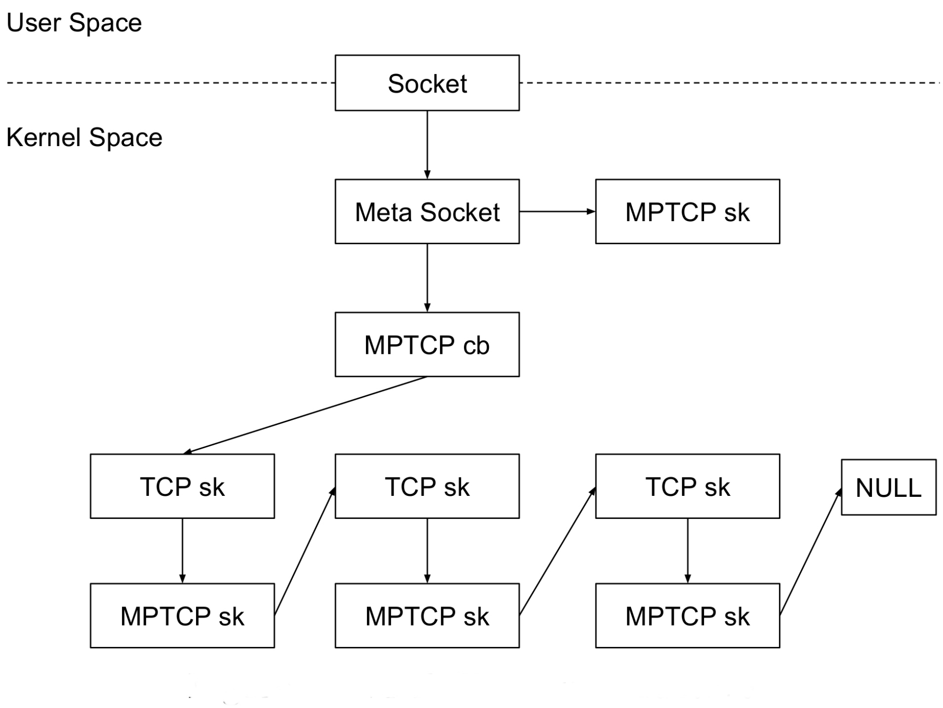


FIGURE 1 – MPTCP socket structure

From the application's point of view, no other socket other than the **Meta Socket** is visible. Underneath the **Meta Socket** lie several subsockets, each representing a sub flow. The structure **mptcp_cb** points towards the head of the subflow list. The structure **mptcp_sk** hence points indirectly towards the next subflow. Till now there is no way for the application to know what hides beyond the **Meta Socket**. This is where the socket options come into play. The enhanced socket API lists the following socket options for the user [4] :

Name	Input	Output	Description
MPTCP_GET_SUB_IDS	-	subflow list	Get the current list of subflows viewed by the kernel
MPTCP_GET_SUB_TUPLE	id	sub tuple	Get the ip and ports used by the subflow identified by id
MPTCP_OPEN_SUB_TUPLE	tuple	-	Request a new subflow with pair of ip and ports
MPTCP_CLOSE_SUB_ID	id	-	Close the subflow identified by id
MPTCP_SUB_GETSOCKOPT	id, sock opt	sock ret	Redirects the getsockopt given in input to the subflow identified by id and return the value returned by the operation
MPTCP_SUB_SETSOCKOPT	id, sock opt	-	Redirects the setsockopt given in input to the subflow identified by id

TABLE 1 – Implemented MPTCP socket options

The following example shows how we may use the socket option
MPTCP_OPEN_SUB_TUPLE and **getsockopt()**to open a sub flow [4] :
First we introduce the **mptcp_sub_tuple** structure which represents the subflow :

```
struct mptcp_sub_tuple {
    _u8 id;        // this is an output signifying the ''id'' of the subflow
    _u8 prio;      // this field determines if the sub flow is backup or not
    _u8 addrs[0]; // pair array of size two depicting (source, destination)
}
```

Now we use this structure to open a sub flow as follows :

```
unsigned int optlen;
struct mptcp_sub_tuple *sub_tuple;
struct sockaddr_in *addr;
optlen = 42;

int error;

optlen = sizeof(struct mptcp_sub_tuple) + 2 * sizeof(struct sockaddr_in);
sub_tuple = malloc(optlen);

sub_tuple->id = 0;
sub_tuple->prio = 0;
```

```
addr = (struct sockaddr_in*) &sub_tuple->addrs[0];

// source address
addr->sin_family = AF_INET; // address family IPv4
addr->sin_port = htons(12345); // source port
inet_pton(AF_INET, "10.0.0.1", &addr->sin_addr); // source IP

addr++;

// destination address
addr->sin_family = AF_INET; // address family IPv4
addr->sin_port = htons(1234); // destination port
inet_pton(AF_INET, "10.1.0.1", &addr->sin_addr); // destination IP

error = getsockopt(sockfd, IPPROTO_TCP, MPTCP_OPEN_SUB_TUPLE,
       sub_tuple, &optlen); // establishment of sub flow using getsockopt()
```

The result is a flow from the pair **(10.0.0.1 : 12345)** to the pair **(10.1.0.1 : 1234)**.

4 Netcat with MPTCP (netcat-mptcp) :

In order to have a concrete testbed for the enhanced socket API, we have thought of a usecase involving **Netcat**. **Netcat** or **nc** is a featured networking utility which reads and writes data across network connections, using the TCP/IP protocol [12]. It will serve as our application that will establish multiple subflows.

4.1 Setup and structure

Usually with **Netcat**, there is only one flow between a client and a server. Our objective is to have multiple interfaces on the client side which will connect to the same or multiple interfaces on the server side. For simplicity we have envisaged a scenario where the client has three interfaces (viz. default, wifi and cellular) and the server has three. However for demonstration purposes we need to connect to only one of them. Our goal after the modifications is to use either the Cellular or Wifi or both interfaces, the interface type and address being defined in a configuration file accessible by the client. The way the client connects to the server is changed. In fact, with our modifications in the source code, we are able to pass three additional arguments in the netcat command :

1. **-a** : Find all the remaining interfaces on the client side and establish sub flows to the server.
2. **-W** : Read the configuration file, extract the IP corresponding to the wifi interface and establish a sub flow using this interface to the server.
3. **-C** : Read the configuration file, extract the IP corresponding to the cellular interface and establish a sub flow using this interface to the server.

For our purpose, we need the kernel not to do anything *suo moto*. Hence the **path_manager** used is the **"default" path_manager**. What happens after our modifications is that the client connects to the server via the default interface as usual. Then according to the option passed in the **Netcat** command, it either opens a subflow on the default interface or in addition on a subset of all available interfaces. This is done using the function *getsockopt()* in the source code just after the TCP *connect()* system call.

The following diagram depicts the setup along with the addresses :

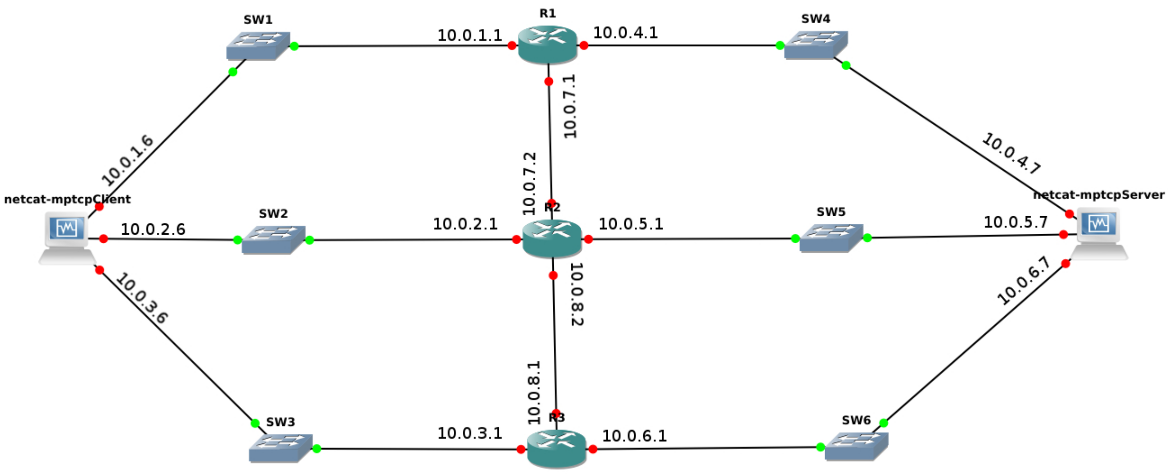


FIGURE 2 – Testbed topology for netcat-mptcp

Here is an example of the **Netcat** command :

On the server side, it listens on it's default interface **10.0.4.7** on port **64000** with the help of the following command :

nc -l -p 64000

On the client side, we use our own **Netcat** executable to establish a flow / multiple flows as follows :

-a : **./netcat-mptcp/src/netcat -a 10.0.4.7 64000**

-W : **./netcat-mptcp/src/netcat -W 10.0.4.7 64000**

-C : **./netcat-mptcp/src/netcat -C 10.0.4.7 64000**

4.2 Configure Addresses and Routing

For the testbed, we have set up the above topology on **GNS3** using the Cisco Router **c3745** and the virtual machine enabled with the enhanced Multipath TCP API.

4.3 Client and Server

The following command example assigns the address **10.0.1.6** to the interface **eth0** :
Client side :

```
ip addr add 10.0.1.6/24 dev eth0
```

Appendix 10.1 and 10.2 illustrate the scripts that are run to assign addresses on the client side and the server side.
With multiple addresses defined on several interfaces, we would also like to tell the kernel to use specific interfaces and gateways and not the default ones acccording to the source addresses. This has been achieved by configuring one routing table per outgoing interface, each routing table being identified by a number. The route selection process then happens in two phases : First the kernel does a lookup in the policy table (that we need to configure with ***ip rules***). The policies in our case, will be that for so and so source prefix, go to so and so routing table (the routing table indicated by a number). The corresponding routing table is examined to select the gateway based on the destination addresses [13].
Appendix 10.3 and 10.5 illustrate the scripts that are run to manually configure the routing policies on the client side and the server side.
Appendix 10.4 and 10.6 illustrate the outputs for the different commands for showing the routing policies.

4.4 Routers

In figure 2 the three routes need to be configured to properly deliver packets to the correct destination. We have connected to the routers via telnet to configure them.
Appendix 10.7, 10.9 and 10.11 illustrate the commands that must be given to the routers **R1, R2** and **R3** respectively.
Appendix 10.8, 10.10 and 10.12 illustrate the outputs for the **sh ip route** command.

4.5 Code simplification, addition and function calls at the correct place

The above code involving opening a sub flow may appear complex. During our participation at the **IETF'97 Hackathon** at **École Polytechnique de Louvain** [14, hackathon], one of my fellow participants had simplified the usage of the **getsockopt()** function by deploying simpler function calls. Our aim was to find in the source code of **Netcat** where the **connect()** system call was being made. Once the the exact place found, we were to simply use the subflow opening code in the simplified form and establish the desired subflows. We have added three different scenarios for the establishment of subflows as described in the section Setup and structure. We have three

different options while the **Netcat** connection : **"-a"** for opening subflows on all remaining interfaces, **"-C"** for opening a subflow on the Cellular interface and **"-W"** for opening a subflow on the Wifi interface.
Besides the classes ***makeaddr.c, subinfo.c, submanip.c and suboption.c*** and the header files ***makeaddr.h, subinfo.h, submanip.h and suboption.h*** which are required simplify the opening of subflows and which are available at the **src** folder of the github repository : [15, https ://github.com/lawrenceFR/netcat-mptcp], the following addition of code was also necessary for the proper functioning of ***netcat-mptcp*** :

1. In ***netcat-mptcp/src/Makefile.am*** line **28 - 39** : Appendix Makefile.am
   Required to include the added **.c** files during **make**.
2. In ***netcat-mptcp/src/Makefile.in*** line **153 - 164** and **183 - 186** : Appendix Makefile.in
   Required to include the added **.c** files during **make**.
3. In ***netcat-mptcp/src/netcat.h*** line **203 - 205** : Appendix netcat.h
   Here we declare the three options as **extern** variables.
4. In ***netcat-mptcp/src/netcat.c*** line **58 - 60**, **192**, **194**, **222**, **227**, **233 - 235**, **239 - 241**, **357 - 359** : Appendix netcat.c
   Here we add the three cases corresponding to the three different options (**switch** variable).
5. In ***netcat-mptcp/src/core.h*** line **1 - 37** : Appendix core.h
   Here we declare the variables and methods that we have defined and used in **core.c** in order to bring about the modifications.
6. In ***netcat-mptcp/src/core.c*** line **394 - 405** and **535 - 688** : Appendix core.c
   Here we define the functions that bring in the actual modifications in the functioning of **Netcat**.

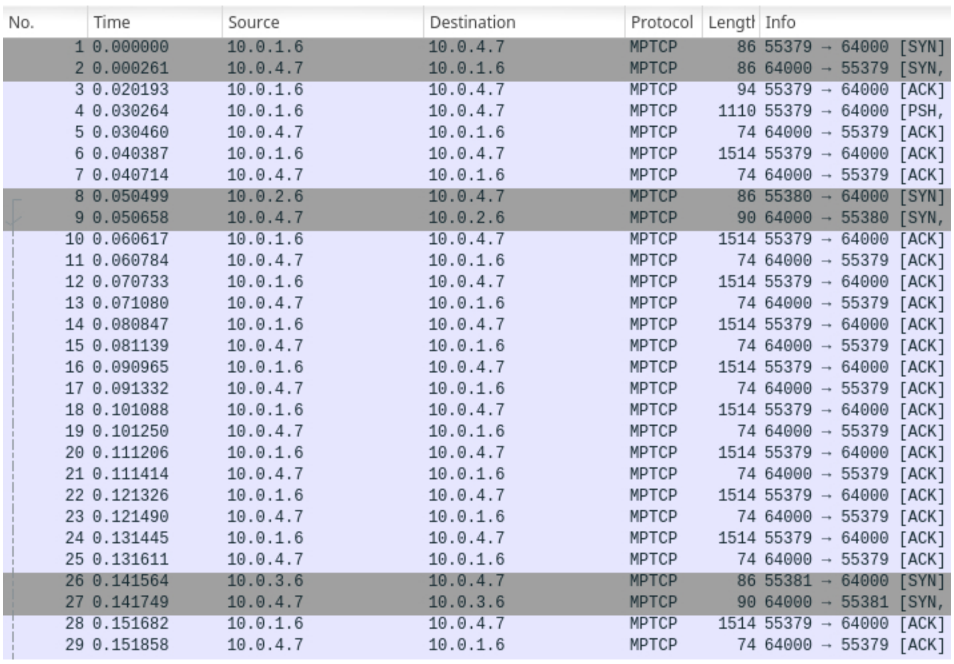
Finally, a file **config.conf** is placed in **netcat-mptcp/src** so as to indicate the addresses of the available interfaces. The format of the config file is shown in Appendix config.conf.

5 Results, Statistics and Utility

This section analyses packet captures (captured with the help of **Wireshark** with a topology simulation on **GNS3**) to verify if the manipulation of **Netcat** was successful. It depicts the establishment of the different sub flows along with the details of the **TCP options** to prove that the **MP_CAPABLE** option is actually passed. It also gives an idea of the distribution of packets sent from the three interfaces available on the client's side, keeping in mind that the **path_manager** used is **"default"** (i.e. the kernel will not establish supplementary sub flows on its own).

5.1 Results

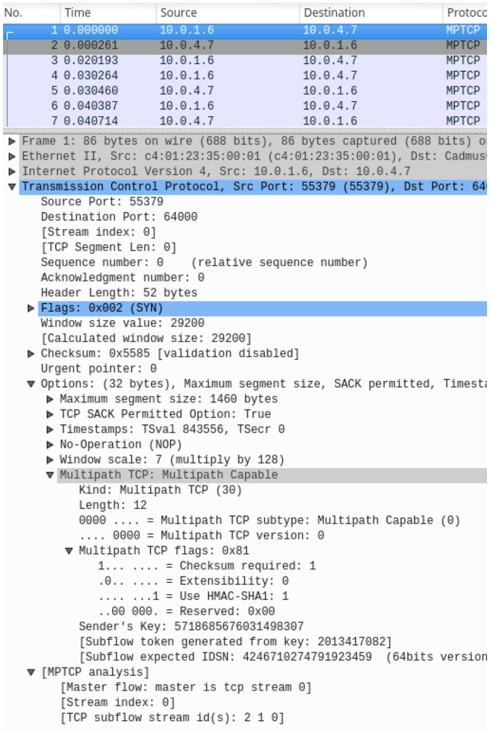
Figure 3 is a screenshot of a packet capture showing that the kernel successfully establishes the second and third sub flows when we use the option **-a** as explained previously. First, the see the **3 way handshake** from the client address **10.0.1.6** to the server address **10.0.4.7**. Next due our the manipulation in the **Netcat** source code we have the two other **3 way hanshakes** from the client addresses **10.0.2.6** and **10.0.3.6**. This is in accordance with the function of the option **-a** in the **Netcat** command defined by us (i.e. establish supplementary sub flows on all the remaining available interfaces of the client).



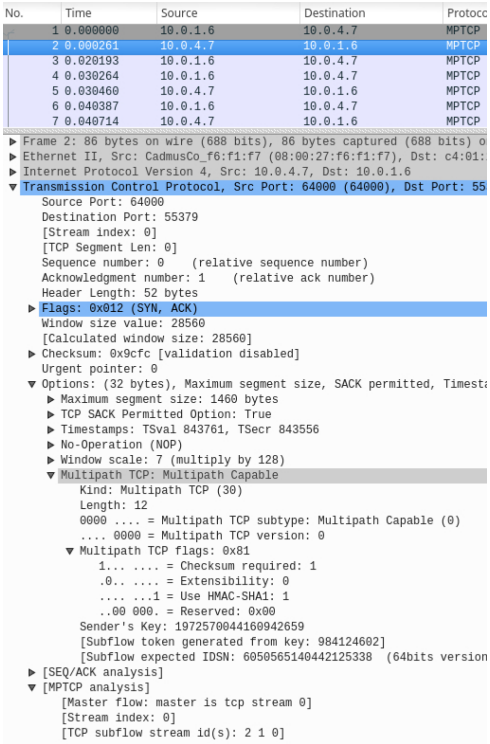
No.	Time	Source	Destination	Protocol	Lengtl	Info
1	0.000000	10.0.1.6	10.0.4.7	MPTCP	86	55379 → 64000 [SYN]
2	0.000261	10.0.4.7	10.0.1.6	MPTCP	86	64000 → 55379 [SYN,
3	0.020193	10.0.1.6	10.0.4.7	MPTCP	94	55379 → 64000 [ACK]
4	0.030264	10.0.1.6	10.0.4.7	MPTCP	1110	55379 → 64000 [PSH,
5	0.030460	10.0.4.7	10.0.1.6	MPTCP	74	64000 → 55379 [ACK]
6	0.040387	10.0.1.6	10.0.4.7	MPTCP	1514	55379 → 64000 [ACK]
7	0.040714	10.0.4.7	10.0.1.6	MPTCP	74	64000 → 55379 [ACK]
8	0.050499	10.0.2.6	10.0.4.7	MPTCP	86	55380 → 64000 [SYN]
9	0.050658	10.0.4.7	10.0.2.6	MPTCP	90	64000 → 55380 [SYN,
10	0.060617	10.0.1.6	10.0.4.7	MPTCP	1514	55379 → 64000 [ACK]
11	0.060784	10.0.4.7	10.0.1.6	MPTCP	74	64000 → 55379 [ACK]
12	0.070733	10.0.1.6	10.0.4.7	MPTCP	1514	55379 → 64000 [ACK]
13	0.071080	10.0.4.7	10.0.1.6	MPTCP	74	64000 → 55379 [ACK]
14	0.080847	10.0.1.6	10.0.4.7	MPTCP	1514	55379 → 64000 [ACK]
15	0.081139	10.0.4.7	10.0.1.6	MPTCP	74	64000 → 55379 [ACK]
16	0.090965	10.0.1.6	10.0.4.7	MPTCP	1514	55379 → 64000 [ACK]
17	0.091332	10.0.4.7	10.0.1.6	MPTCP	74	64000 → 55379 [ACK]
18	0.101088	10.0.1.6	10.0.4.7	MPTCP	1514	55379 → 64000 [ACK]
19	0.101250	10.0.4.7	10.0.1.6	MPTCP	74	64000 → 55379 [ACK]
20	0.111206	10.0.1.6	10.0.4.7	MPTCP	1514	55379 → 64000 [ACK]
21	0.111414	10.0.4.7	10.0.1.6	MPTCP	74	64000 → 55379 [ACK]
22	0.121326	10.0.1.6	10.0.4.7	MPTCP	1514	55379 → 64000 [ACK]
23	0.121490	10.0.4.7	10.0.1.6	MPTCP	74	64000 → 55379 [ACK]
24	0.131445	10.0.1.6	10.0.4.7	MPTCP	1514	55379 → 64000 [ACK]
25	0.131611	10.0.4.7	10.0.1.6	MPTCP	74	64000 → 55379 [ACK]
26	0.141564	10.0.3.6	10.0.4.7	MPTCP	86	55381 → 64000 [SYN]
27	0.141749	10.0.4.7	10.0.3.6	MPTCP	90	64000 → 55381 [SYN,
28	0.151682	10.0.1.6	10.0.4.7	MPTCP	1514	55379 → 64000 [ACK]
29	0.151858	10.0.4.7	10.0.1.6	MPTCP	74	64000 → 55379 [ACK]

FIGURE 3 – Subflow establishment

The following figures give the details of the **TCP options** for the first sub flow established. We see the **Multipath Capable** option that is passed :

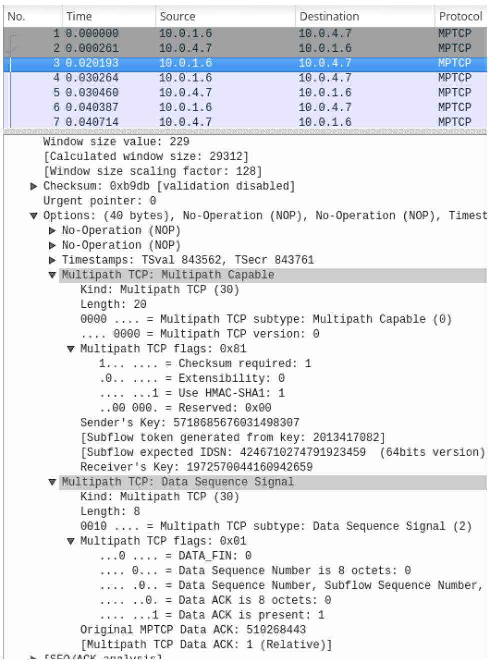


(a) First subflow SYN



(b) First subflow SYN + ACK

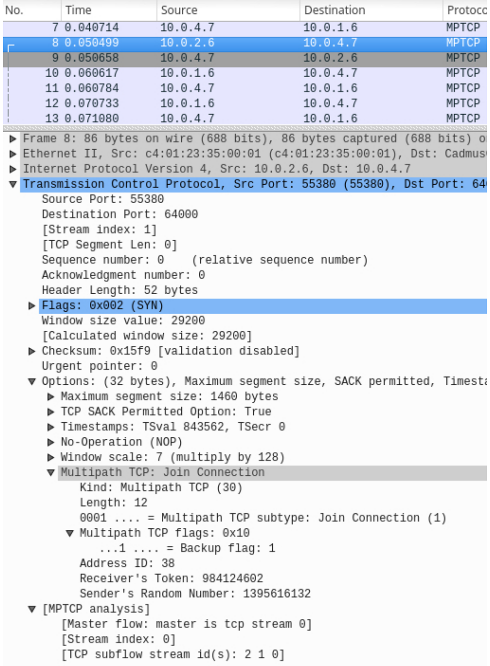
FIGURE 4 – First subflow details



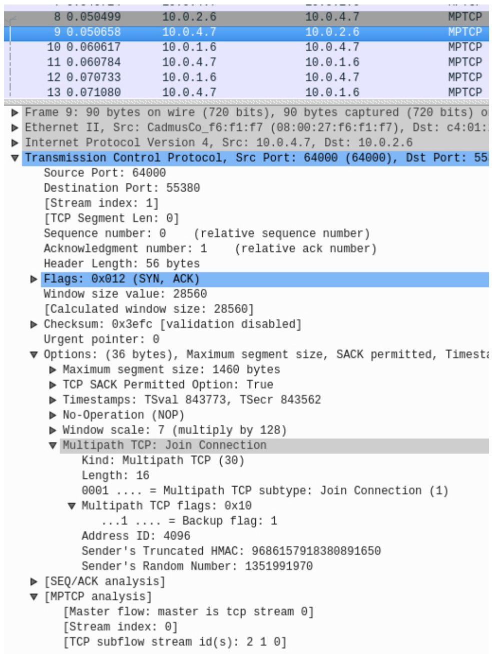
The **3 way handshake** illustrates the key exchange procedure. We see that the **Multipath Capable** option contains the sender and receiver tokens. This ensures the security aspect of **MPTCP**. The **SYN** contains the client's key. The **SYN + ACK** contains the server's key. Finally the client replies with an **ACK** containing both the keys.

In the following figures the second and third subflows contain the **TCP option Join Connection** proving the usage of **MP_JOIN**.

FIGURE 5 – First subflow ACK

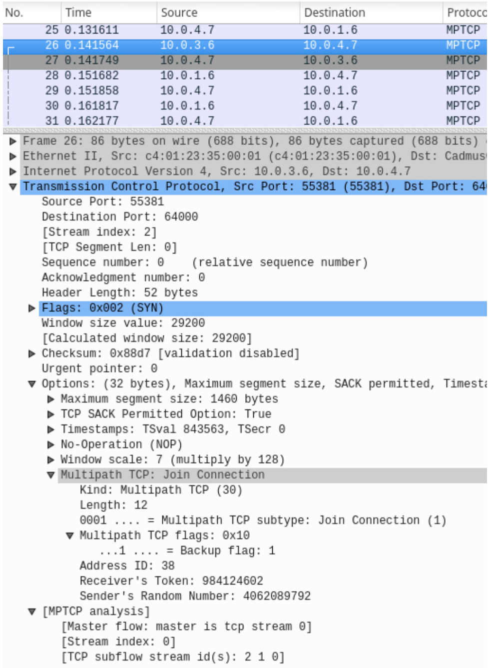


(a) Second subflow SYN

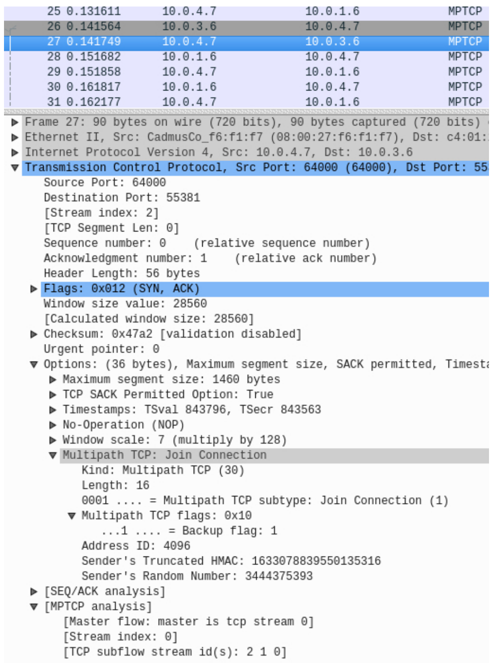


(b) Second subflow SYN + ACK

Figure 6 – Second subflow details



(a) Third subflow SYN



(b) Third subflow SYN + ACK

Figure 7 – Third subflow details

5.2 Statistics

This section mentions a few statistics regarding the performance of **MPTCP** once the supplementary subflows are established.

5.2.1 IPv4/All Addresses packet distribution

Our first look is at the **IPv4/All Addresses** packet distribution which indicates the percentage contribution of each interface in the exchange of data. We have tried to send a fixed amount of random data (10MB) at bulks of 1MB for 10 counts. Command on the client side :

dd if=/dev/zero bs=1M count=10 | ./netcat-mptcp/src/netcat -a 10.0.4.7 64000

IP	Count	Rate(ms)	Percent	Burst rate	Burst start
10.0.4.7	14076	0.1870	100.00%	0.2200	64.072
10.0.3.6	3331	0.0442	23.66%	0.2000	0.890
10.0.2.6	5021	0.0667	35,67%	0.2000	0.516
10.0.1.6	5724	0.0760	40.66%	0.2000	0.223

TABLE 2 – Packet distribution per IPv4 address

The table may seem to show that the distribution of packets is not equal. The interface **10.0.1.6** seems to send more packets than the others. This is probably due the fact that the packet emission from the second and third interfaces start a little later only after the establishment of the corresponding sub flows. Hence the kernel gets to be aware of the other sub flows later given that the **path_manager** used is the **"default"** one. This is not the case for **"fullmesh" path_manager** as the kernel knows from beforehand that it needs to use all the interfaces available for sending packets. Hence for the **"fullmesh" path_manager** we have an almost equal distribution of packet emission for all the three interfaces.

5.2.2 Round Trip Time

This section shows the round trip time for the three interfaces.
We see that they are almost the same for the three interfaces (a small exception for the third interface around sequence number 2000000). This shows if the three interfaces are homogeneous in terms of updating their receiving windows and hence the reception and emission of packets. However, when we compare this to the round trip time in the **"fullmesh" path_manager** there is hardly any anomaly in the sense that the round trip time very rarely crosses the **1800** mark.

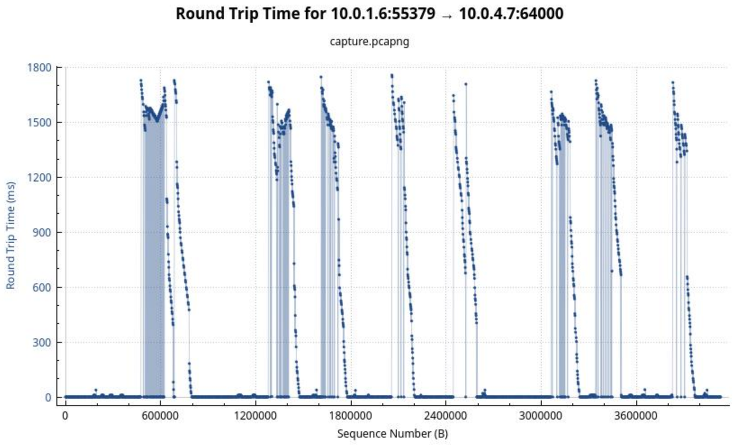


FIGURE 8 – RTT for 10.0.1.6

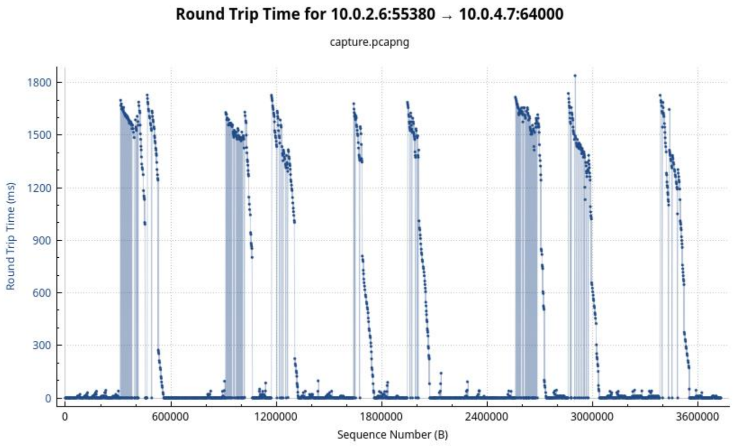


FIGURE 9 – RTT for 10.0.2.6

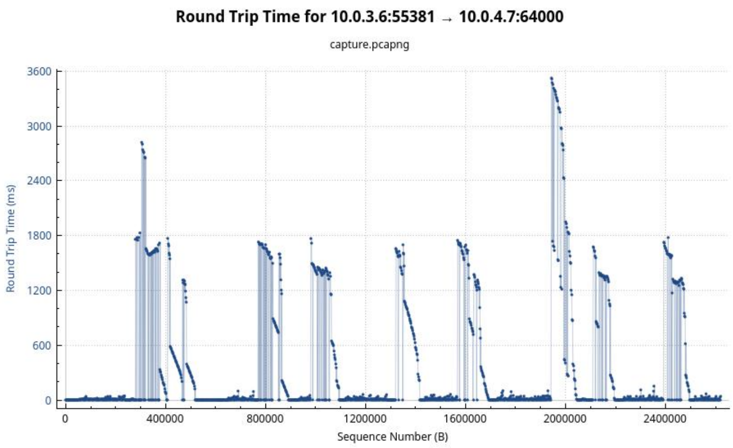


FIGURE 10 – RTT for 10.0.3.6

5.2.3 Cumulative Bytes

This section illustrates the comparison of the number of bytes sent in due course of time for the three interfaces of the client. On the Y-axis we have the cumulative number of bytes. This figure shows that the interfaces **10.0.1.6** and **10.0.2.6** run almost parallel to one another, which means that their throughput is similar. The offset is attributed to the fact that the second interface starts to send packets later. For the third interface **10.0.3.6** the throughput seems to be less compared to the other two.

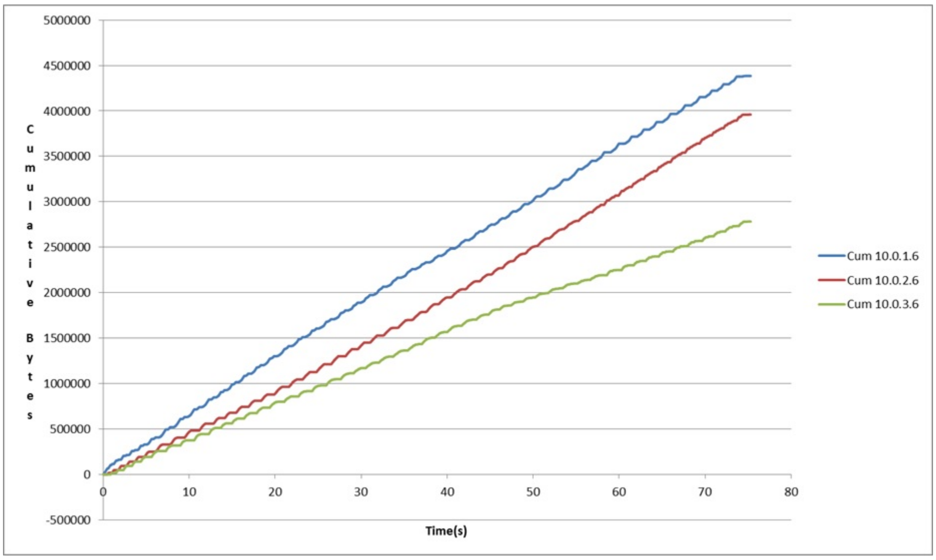


FIGURE 11 – Cumulative of bytes sent for the three interfaces

5.2.4 Data transfer and flow establishment statistics

In this section we talk about the latency, the rate of data transfer and the interval lapse among the sub flows for the two cases. The first case is where we use the **"fullmesh" path_manager** and establish a **Netcat** connection without any of the three defined command options. The second case is where we use the **"default" path_manager** and the defined **Netcat** command option **"-a"** to open sub flows on the remaining available interfaces.

To compare the two cases we caculate the mean and standard deviation for the different measures.

The mean is given by the formula :

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

The standard deviation is given by the formula :

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2}$$

In the experiment, we sent 10MB of random data at bulks of 1MB for 10 counts from the client to the server using the command :
dd if=/dev/zero bs=1M count=10 | ./netcat-mptcp/src/netcat -a 10.0.4.7 64000
The experiment was repeated 10 times.

5.2.4.1 Data Transfer : After sending 10MB of data it was seen that in the two cases, the rate of data transfer and hence the duration was drastically different. The observations were as follows :

Obs	Fullmesh			Default	
	Rate(kB/s)	Time(s)		Rate(kB/s)	Time(s)
1	515	20.3492		153	68.4308
2	516	20.3332		147	71.4054
3	549	19.105		151	69.5162
4	551	19.0169		152	68.7636
5	463	22.6526		149	70.567
6	521	20.1181		151	69.6091
7	502	20.8867		150	69.7403
8	446	23.4884		156	67.2153
9	507	20.7018		148	70.8937
10	515	20.3454		157	66.5856

TABLE 3 – Data transfer rate and time taken to send 10MB from client to server

We define :

$$\mu_{fm}^{rate} := mean\ of\ data\ transfer\ rate\ for\ fullmesh\ configuration$$

$$\mu_{df}^{rate} := mean\ of\ data\ transfer\ rate\ for\ default\ configuration$$

$$\sigma_{fm}^{rate} := standarddeviation\ of\ data\ transfer\ rate\ for\ fullmesh\ configuration$$

$$\sigma_{df}^{rate} := standarddeviation\ of\ data\ transfer\ rate\ for\ default\ configuration$$

From our observations we have : $\mu_{fm}^{rate} = 508.5\ kB/s$ and $\mu_{df}^{rate} = 151.4\ kB/s$ Hence we observe that the data transfer rate is reduced when we initiate subflows from the application level. In case of **"fullmesh"**, it is the kernel that initiates the subflows. Next if we compare the standard deviations in the two cases. While $\sigma_{fm}^{rate} = 31.2482\ kB/s$, $\sigma_{df}^{rate} = 3.07246\ kB/s$. This shows that in case of **"fullmesh"**, the fluctuation of bandwidth is higher. The same conclusions may be drawn for the total time taken to transfer 10MB ranfom data in the two cases.

5.2.4.2 Flow establishment statistics : Here we compare the time intervals among the first, second and third subflow establishments. The observations were as follows :

Obs	Fullmesh				Default		
	flow1 - flow2	flow2 - flow3	flow1 - flow3		flow1 - flow2	flow2 - flow3	flow1 - flow3
1	0.172115	0.030324	0.202439		0.030288	0.111036	0.141324
2	0.171439	0.030272	0.201711		0.030233	0.111062	0.141295
3	0.171637	0.030321	0.201958		0.131091	0.010066	0.141157
4	0.171403	0.030351	0.201754		0.131221	0.010084	0.141305
5	0.171689	0.030322	0.202011		0.130997	0.010079	0.141076
6	0.171472	0.030259	0.201731		0.131214	0.010097	0.141311
7	0.171598	0.030268	0.201866		0.030258	0.11108	0.141338
8	0.171551	0.030333	0.201884		0.130936	0.010069	0.141005
9	0.171399	0.010085	0.181484		0.030226	0.111013	0.141239
10	0.171555	0.030348	0.201903		0.131156	0.010118	0.141274

TABLE 4 – Time intervals among the first, second and third flow initiations

We define for $i, j \in \{1, 2, 3\}$:

$$\mu_{fm}^{ij} := mean\ of\ time\ interval\ for\ fullmesh\ configuration\ between\ flowi\ and\ flowj$$

$$\mu_{df}^{ij} := mean\ of\ time\ interval\ for\ default\ configuration\ between\ flowi\ and\ flowj$$

$$\sigma_{fm}^{ij} := standard\ deviation\ of\ time\ interval\ for\ fullmesh\ configuration\ between\ flowi\ and\ flowj$$

$$\sigma_{df}^{ij} := standard\ deviation\ of\ time\ interval\ for\ default\ configuration\ between\ flowi\ and\ flowj$$

From our observations we have :

1. $\mu_{fm}^{12} = 0.1715858\ s$ and $\mu_{df}^{12} = 0.090762\ s$
2. $\mu_{fm}^{23} = 0.0282883\ s$ and $\mu_{df}^{23} = 0.0504704\ s$
3. $\mu_{fm}^{13} = 0.1998741\ s$ and $\mu_{df}^{13} = 0.1412324\ s$
4. $\sigma_{fm}^{12} = 0.000210408\ s$ and $\sigma_{df}^{12} = 0.052079437\ s$
5. $\sigma_{fm}^{23} = 0.0063960738\ s$ and $\sigma_{df}^{23} = 0.0521366865\ s$
6. $\sigma_{fm}^{13} = 0.0064649789\ s$ and $\sigma_{df}^{13} = 0.0001147657\ s$

Apart from the differences that we observe in the time intervals of establishment of the sub flows, we also have an idea of the variability of these time intervals from one observation to another with.

6 Conclusion

This work is the first step towards the usage of the **Enhanced Socket API for Multipath TCP**. Our objective was to have a first functional prototype for the manipulation of subflows from the **Application Layer**. With **Netcat** as our application and the changes in its source code we have been successful in developing a versatile **netcat-mptcp**. Of course, this is a very preliminary. However the versatality and simplicity of **netcat-mptcp** will make it easy to bring in any kind of developement of new features and further manipulation.

7 Further developments

As mentioned in section Statistics, we have certain underperformances and anomalies that need to be addressed. To list a few of them :

1. **Equitable packet distribution from the three interfaces of the client** : In **netcat-mptcp** it is not as fair as in the case of ordinary **MPTCP** with the **"fullmesh" path_manager**
2. **Round Trip Time** : **Netcat-mptcp** registered some **RTT**s to be quite large as compared to the **"fullmesh" path_manager** which showed very little variation.
3. **Cumulative Bytes for the individual interfaces** : In **netcat-mptcp** we found that the third interface had a lower throughout. This is contrary to the traditional **"fullmesh" path_manager**.
4. **Performance after modifications and further tests** : We observe a declination in the performance in the **"default" path_manager** compared to the **"fullmesh" path_manager**. Our testbed conditions are not optimal to acertain de drop in performance. There are different factors that comme into play such as the allocation of resources while using **GNS3** etc. Hence we need to retry the observations in a real environment. We may also vary the amount of random data passed (1MB, 10MB, 100MB etc.) to see how the graphs evolve.

There might be other glitches which may come up in the course of further developement of **netcat-mptcp**. However, the current version is functional and attains the objective of manipulation of **MPTCP** subflows from the **Application Layer**.

8 Acknowledgements

I had the honour to work with M. Antoine Fressancourt and M. Tony Ducrocq on the CarFi project. They have been very helpful not only at work but also at other formalities not related to work. I take this opportunity to thank the organisers of the Hackathon at **L'Université Catholique de Louvain**. In particular M. Olivier Bonaventure, M. Mathieu Jadin and M. Quentin De Coninck. I must mention my fellow participant Alexis Clarembeau, for helping me out with the code during the Hackathon. Finally, I owe a lot to M. Matthieu Coudron who gave me his valuable time and support.

9 Bibliography

Références

[1] Nathan Dyer Hai Vu Nigel William, Prashan Abeysekera. Multipath tcp in vehicular to infrastructure communications.

[2] Vehicle-to-vehicle/vehicle-to-infrastructure control. http ://ieeecss.org/sites/ieeecss.org/files/documents/IoCT-Part4-13VehicleToVehicle-HR.pdf. Accessed : 06/01/2017.

[3] Smartcities : Vehicle to infrastructure and adaptive roadway lighting communication standards. http ://www.gridaptive.com/whitepapers/Smart Accessed : 06/01/2017.

[4] Olivier Bonaventure Benjamin Hesmans. An enhanced socket api for multipath tcp.

[5] Using your in-kernel protocol implementation. https ://www.nsnam.org/docs/dce/manual/html/dce-user-kernel.html. Accessed : 06/01/2017.

[6] net-next-nuse. https ://github.com/libos-nuse/net-next-nuse/wiki/Quick-Start. Accessed : 06/01/2017.

[7] net-next-sim. https ://github.com/direct-code-execution/net-next-sim. Accessed : 06/01/2017.

[8] Hajime Tazaki. libos-nuse : net-next-nuse. https ://github.com/libos-nuse/net-next-nuse.

[9] Ns-3 : Direct code execution. https ://www.nsnam.org/overview/projects/direct-code-execution.

[10] ns-3-dce.
https ://groups.google.com/d/msg/ns-3-users/M3A5ydBLktY/nNga9nvfAQAJ.
Accessed : 06/01/2017.

[11] Bake. https ://www.nsnam.org/docs/bake/tutorial/html/bake-over.html.
Accessed : 06/01/2017.

[12] Netcat. http ://netcat.sourceforge.net/. Accessed : 20/12/2016.

[13] Configure routing : Manual configuration.
http ://multipath-tcp.org/pmwiki.php/Users/ConfigureRouting/. Accessed :
20/12/2016.

[14] Hackathon à l'université catholique de louvain.
http ://blog.multipath-tcp.org/blog/html/index.html. Accessed : 06/01/2017.

[15] netcat-mptcp. https ://github.com/lawrenceFR/netcat-mptcp/. Accessed :
20/12/2016.

10 Appendix

Here we have the different additional information, notably the code and the scripts used in the proper functioning of our testbed.

10.1 Client side address assignment with the following script :

```
#!/bin/sh

# flush all ip addresses
ip addr flush dev eth0
ip addr flush dev eth1
ip addr flush dev eth2

# bring all the interfaces down
ip link set dev eth0 down
ip link set dev eth1 down
ip link set dev eth2 down

# bring all the interfaces up
ip link set dev eth0 up
ip link set dev eth1 up
ip link set dev eth2 up

# assign addresses to the interfaces
ip addr add 10.0.1.6/24 dev eth0
ip addr add 10.0.2.6/24 dev eth1
ip addr add 10.0.3.6/24 dev eth2
```

10.2 Server side address assignment with the following script :

```
#!/bin/sh

# flush all ip addresses
ip addr flush dev eth0
ip addr flush dev eth1
ip addr flush dev eth2

# bring all the interfaces down
ip link set dev eth0 down
ip link set dev eth1 down
ip link set dev eth2 down

# bring all the interfaces up
ip link set dev eth0 up
ip link set dev eth1 up
ip link set dev eth2 up

# assign addresses to the interfaces
ip addr add 10.0.4.7/24 dev eth0
ip addr add 10.0.5.7/24 dev eth1
ip addr add 10.0.6.7/24 dev eth2
```

10.3 Client side routing :

```
#!/bin/sh

# this rule creates three different routing tables that we use based on the
    source addresses
ip rule add from 10.0.1.6 table 1
ip rule add from 10.0.2.6 table 2
ip rule add from 10.0.3.6 table 3

# configure the three different routing tables
ip route add 10.0.1.0/24 dev eth0 scope link table 1
ip route add default via 10.0.1.1 dev eth0 table 1

ip route add 10.0.2.0/24 dev eth1 scope link table 2
ip route add default via 10.0.2.1 dev eth1 table 2

ip route add 10.0.3.0/24 dev eth2 scope link table 3
ip route add default via 10.0.3.1 dev eth2 table 3

# default route for the selection process of normal internet-traffic
ip route add default scope global nexthop via 10.0.1.1 dev eth0
```

10.4 Client routing output :

```
mininet@mininet-vm:~$ ip rule show
0 :    from all liiokup local
32763 : from 10.0.3.6 lookup 3
32764 : from 10.0.2.6 lookup 2
32765 : from 10.0.1.6 lookup 1
32766 : from all lookup main
32767 : from all lookup default

mininet@mininet-vm:~$ ip route
default via 10.0.1.1 dev eth0
10.0.1.0/24 dev eth0 proto kernel scope link src 10.0.1.6
10.0.2.0/24 dev eth1 proto kernel scope link src 10.0.2.6
10.0.3.0/24 dev eth2 proto kernel scope link src 10.0.3.6

mininet@mininet-vm:~$ ip route show table 1
default via 10.0.1.1 dev eth0
10.0.1.0/24 dev eth0 scope link

mininet@mininet-vm:~$ ip route show table 2
default via 10.0.2.1 dev eth1
10.0.2.0/24 dev eth0 scope link

mininet@mininet-vm:~$ ip route show table 3
default via 10.0.3.1 dev eth2
10.0.3.0/24 dev eth0 scope link
```

10.5 Server side routing :

```
#!/bin/sh

# this rule creates three different routing tables that we use based on the
    source addresses
ip rule add from 10.0.4.7 table 1
ip rule add from 10.0.5.7 table 2
ip rule add from 10.0.6.7 table 3

# configure the three different routing tables
ip route add 10.0.4.0/24 dev eth0 scope link table 1
ip route add default via 10.0.4.1 dev eth0 table 1

ip route add 10.0.5.0/24 dev eth1 scope link table 2
ip route add default via 10.0.5.1 dev eth1 table 2

ip route add 10.0.6.0/24 dev eth2 scope link table 3
ip route add default via 10.0.6.1 dev eth2 table 3

# default route for the selection process of normal internet-traffic
ip route add default scope global nexthop via 10.0.4.1 dev eth0
```

10.6 Server routing output :

```
mininet@mininet-vm:~$ ip rule show
0 :    from all liiokup local
32763 : from 10.0.6.7 lookup 3
32764 : from 10.0.5.7 lookup 2
32765 : from 10.0.4.7 lookup 1
32766 : from all lookup main
32767 : from all lookup default

mininet@mininet-vm:~$ ip route
default via 10.0.4.1 dev eth0
10.0.4.0/24 dev eth0 proto kernel scope link src 10.0.4.7
10.0.5.0/24 dev eth1 proto kernel scope link src 10.0.5.7
10.0.6.0/24 dev eth2 proto kernel scope link src 10.0.6.7

mininet@mininet-vm:~$ ip route show table 1
default via 10.0.4.1 dev eth0
10.0.4.0/24 dev eth0 scope link

mininet@mininet-vm:~$ ip route show table 2
default via 10.0.5.1 dev eth1
10.0.5.0/24 dev eth0 scope link

mininet@mininet-vm:~$ ip route show table 3
default via 10.0.6.1 dev eth2
10.0.6.0/24 dev eth0 scope link
```

10.7 Router R1 :

```
enable
conf t
interface fastEthernet0/0
ip address 10.0.1.1 255.255.255.0
no shut
exit
interface fastEthernet0/1
ip address 10.0.4.1 255.255.255.0
no shut
exit
interface fastEthernet1/0
ip address 10.0.7.1 255.255.255.0
no shut
exit

ip route 10.0.1.0 255.255.255.0 fastEthernet0/0
ip route 10.0.4.0 255.255.255.0 fastEthernet0/1
ip route 0.0.0.0 0.0.0.0 fastEthernet1/0
exit
write
sh ip route
```

10.8 Router R1 routing output :

```
    10.0.0.0/24 is subnetted, 3 subnets
C       10.0.1.0 is directly connected, FastEthernet0/0
C       10.0.7.0 is directly connected, FastEthernet1/0
C       10.0.4.0 is directly connected, FastEthernet0/1
S*   0.0.0.0/0 is directly connected, FastEthernet1/0
```

10.9 Router R2 :

```
enable
conf t
interface fastEthernet0/0
ip address 10.0.2.1 255.255.255.0
no shut
exit
interface fastEthernet0/1
ip address 10.0.5.1 255.255.255.0
no shut
exit
interface fastEthernet1/0
ip address 10.0.7.2 255.255.255.0
no shut
exit
interface fastEthernet2/0
ip address 10.0.8.2 255.255.255.0
no shut
exit
```

```
ip route 10.0.1.0 255.255.255.0 fastEthernet1/0
ip route 10.0.2.0 255.255.255.0 fastEthernet0/0
ip route 10.0.3.0 255.255.255.0 fastEthernet2/0
ip route 10.0.4.0 255.255.255.0 fastEthernet1/0
ip route 10.0.5.0 255.255.255.0 fastEthernet0/1
ip route 10.0.6.0 255.255.255.0 fastEthernet2/0
ip route 0.0.0.0 0.0.0.0 fastEthernet1/0
exit
write
sh ip route
```

10.10 Router R2 routing output :

```
     10.0.0.0/24 is subnetted, 8 subnets
C       10.0.8.0 is directly connected, FastEthernet2/0
C       10.0.2.0 is directly connected, FastEthernet0/0
S       10.0.3.0 is directly connected, FastEthernet2/0
S       10.0.1.0 is directly connected, FastEthernet1/0
S       10.0.6.0 is directly connected, FastEthernet2/0
C       10.0.7.0 is directly connected, FastEthernet1/0
S       10.0.4.0 is directly connected, FastEthernet1/0
C       10.0.5.0 is directly connected, FastEthernet0/1
S*   0.0.0.0/0 is directly connected, FastEthernet1/0
```

10.11 Router R3 :

```
enable
conf t
interface fastEthernet0/0
ip address 10.0.3.1 255.255.255.0
no shut
exit
interface fastEthernet0/1
ip address 10.0.6.1 255.255.255.0
no shut
exit
interface fastEthernet1/0
ip address 10.0.8.1 255.255.255.0
no shut
exit

ip route 10.0.3.0 255.255.255.0 fastEthernet0/0
ip route 10.0.6.0 255.255.255.0 fastEthernet0/1
ip route 0.0.0.0 0.0.0.0 fastEthernet1/0
exit
write
sh ip route
```

10.12 Router R3 routing output :

```
    10.0.0.0/24 is subnetted, 3 subnets
C      10.0.8.0 is directly connected, FastEthernet1/0
C      10.0.3.0 is directly connected, FastEthernet0/0
C      10.0.6.0 is directly connected, FastEthernet0/1
S*   0.0.0.0/0 is directly connected, FastEthernet1/0
```

10.13 Makefile.am :

```
netcat_SOURCES = \
   core.c \
   flagset.c \
   misc.c \
   netcat.c \
   network.c \
   telnet.c \
   udphelper.c \
   makeaddr.c \
   subinfo.c \
   submanip.c \
   suboption.c
```

10.14 Makefile.in :

```
netcat_SOURCES = \
   core.c \
   flagset.c \
   misc.c \
   netcat.c \
   network.c \
   telnet.c \
   udphelper.c \
   makeaddr.c \
   subinfo.c \
   submanip.c \
   suboption.c

.
.
.

am_netcat_OBJECTS = core.$(OBJEXT) flagset.$(OBJEXT) misc.$(OBJEXT) \
   netcat.$(OBJEXT) network.$(OBJEXT) telnet.$(OBJEXT) \
   makeaddr.$(OBJEXT) subinfo.$(OBJEXT) submanip.$(OBJEXT) \
   suboption.$(OBJEXT)
```

10.15 netcat.h :

```
extern bool opt_addAllSubflows; // option to add all the remaining subflows
extern bool opt_addWifi; // option to add the wifi subflow only
extern bool opt_addCellular; // option to add the cellular subflow only
```

10.16 netcat.c :

```
bool opt_addAllSubflows = FALSE; /* option to ad all the supplementary subflows */
bool opt_addWifi = FALSE; /* option to add the Wifi subflow */
bool opt_addCellular = FALSE; /* option to add the Cellular subflow */

.
.
.

{ ‘‘all’’,    no_argument,      NULL, ‘a’ },

...

{ ‘‘cellular’’, no_argument,    NULL, ‘C’ },

...

{ ‘‘wifi’’,   no_argument,      NULL, ‘W’ },

...

c = getopt_long(argc, argv, "acCde:g:G:hi:lL:no:p:P:rs:S:tTuvVxw:Wz",
                  long_options, &option_index);

...

case ‘a’ :
   opt_addAllSubflows = TRUE;   /* enable MPTCP all subflows */
   break;

...

case ‘C’ :
   opt_addCellular = TRUE;      /* enable MPTCP Cellular subflow */
   break;

...

case ‘W’ :
   opt_addWifi = TRUE;          /*enable MPTCP Wifi subflow */
   break;
```

10.17 core.h :

```
 #include<stdio.h>
#include<stdlib.h>
#include<string.h>

#define FILENAME "config.conf"
#define MAXBUF 1024
#define DELIM "="

```

```
struct config {
    char wifi[MAXBUF];
    char cellular[MAXBUF];
};

struct config readConfig();

void addAllSubflows(int);
void addSubflow(int, char[]);
```

10.18 core.c :

```
if(opt_addWifi) {
    printf("\nEntering Wifi\n");
    struct config configstruct = readConfig();
    addSubflow(sock, configstruct.wifi);
} else if(opt_addCellular) {
    struct config configstruct = readConfig();
    addSubflow(sock, configstruct.cellular);
} else if(opt_addAllSubflows) {
    addAllSubflows(sock);
}else {
    printf("\nNo supplementary flow initiation asked\n");
}

.
.
.

/* ... */


void addAllSubflows(int sock) {

        // structure to store the list of subflows
        struct mptcp_sub_tuple_list *list;

        // d'abord trouver les interfaces disponible, puis etablir les sous flux

        // get the subflow list
        if(mptcp_get_sub_list(sock, &list) != 0) {
            printf("\nError getting the list of subflows !");
        }
        // structure to store the subflow src dst ip port
        struct mptcp_sub_tuple_info struc;
        // get the structure mptcp_sub_tuple_info
        if(mptcp_get_sub_tuple(sock, list->subid, &struc) != 0) {
            printf("\nError getting the structure mptcp_sub_tuple_info !");
        }
        // char array storing the client interface addresses
        char client_addr[4096];
        int client_port = struc.sourceP;
        int server_port = struc.destP;
```

```
        // structures and variables for getting the characteristics of the other interfaces
        struct ifaddrs *ifaddr, *ifa;
        int family;

    if(getifaddrs(&ifaddr) != 0) {
        printf("\nError getting the interface addresses !");
    }
    for(ifa = ifaddr; ifa != NULL; ifa = ifa->ifa_next) {
        if(ifa->ifa_addr == NULL) {
            continue;
        }
        family = ifa->ifa_addr->sa_family;
        if(family == AF_INET) {
            inet_ntop(AF_INET, &((struct sockaddr_in *)ifa->ifa_addr)->sin_addr, client_addr, INET_ADDRSTRLEN);
            if((strcmp(client_addr, struc.sourceH) != 0) && (strcmp(client_addr, "127.0.0.1") != 0)) {
                if(mptcp_add_subflow(sock, AF_INET, client_addr, ++client_port, struc.destH, server_port) != 0) {
                    printf("\nError adding a subflow !");
                }
            }
        } else {
            //inet_ntop(AF_INET6, &((struct sockaddr_in6 *)ifa->ifa_addr)->sin6_addr, client_addr, INET6_ADDRSTRLEN);
            //if((strcmp(client_addr, struc.sourceH) != 0) && (strcmp(client_addr, "::1") != 0)) {
            //  if(mptcp_add_subflow(sockfd, AF_INET6, client_addr, struc.sourceP, struc.destH, struc.destP) != 0) {
            //      printf("\nError adding a subflow !");
            //  }
            //}
            printf("\nCannot treat IPv6 for the moment :/ Sorry, Yes it's kinda lame :(\n");
        }
    }
    freeifaddrs(ifaddr);



        /* display subflows */

        if(mptcp_get_sub_list(sock, &list) != 0) {
        printf("\nError getting the list of subflows !");
        }
        while(list != NULL){
            mptcp_get_sub_tuple(sock, list->subid, &struc);
            printf("(%s %d) -> (%s %d)\n", struc.sourceH, struc.sourceP, struc.destH, struc.destP);
            list = list->next;
        }
```

```

}
/* ... */

void addSubflow(int sock, char ip[]) {
    /*
    printf("\nRes = %d\n", mptcp_add_subflow(sock, AF_INET, "10.0.5.7", 64101, 10.0.4.7, 64000, 1));
    */

    printf("\nIP address read is %s\n", ip);
    // structure to store the list of subflows
        struct mptcp_sub_tuple_list *list;

        // get the subflow list
        if(mptcp_get_sub_list(sock, &list) != 0) {
            printf("\nError getting the list of subflows !");
        }
        // structure to store the subflow src dst ip port
        struct mptcp_sub_tuple_info struc;
        // get the structure mptcp_sub_tuple_info
        if(mptcp_get_sub_tuple(sock, list->subid, &struc) != 0) {
            printf("\nError getting the structure mptcp_sub_tuple_info !");
        }
        // char array storing the client interface addresses
        char client_addr[4096];
        int client_port = struc.sourceP;
        int server_port = struc.destP;
        int resultat;
    if((resultat = mptcp_add_subflow(sock, AF_INET, ip, ++client_port, struc.destH, server_port)) != 0) {
                    printf("\nError adding a subflow ! Result = %d\n", resultat);
    }


    /* display subflows */

        if(mptcp_get_sub_list(sock, &list) != 0) {
        printf("\nError getting the list of subflows !");
        }
        while(list != NULL){
            mptcp_get_sub_tuple(sock, list->subid, &struc);
            printf("(%s %d) -> (%s %d)\n", struc.sourceH, struc.sourceP, struc.destH, struc.destP);
            list = list->next;
        }


}


/* ... */

struct config readConfig() {
    struct config configstruct;
```

```
    printf("\nReading config file\n");
    FILE *file = fopen("/home/mininet/netcat-mptcp/src/config.conf", "r");

    if(file != NULL) {
        char line[MAXBUF];
        int i = 0;

        while(fgets(line, sizeof(line), file) != NULL) {
            char *cfline;
            cfline = strstr((char *)line, DELIM);
            cfline = cfline + strlen(DELIM);

            if(i == 0) {
                memcpy(configstruct.wifi, cfline, strlen(cfline));
                printf("\nWifi ip address is %s\n", configstruct.wifi);
            } else if(i == 1) {
                memcpy(configstruct.cellular, cfline, strlen(cfline));
                printf("\nCellular ip address is %s\n", configstruct.cellular);
            } else {
                printf("\nNo more lines\n");
            }

            i++;
        }
        fclose(file);
    }
    return configstruct;
}


/* ... */
```

10.19 config.conf :

```
WIFI=10.0.2.6
CELLULAR=10.0.3.6
```