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

English

Résumé

Français

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 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.

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 sub-flows. Benefits of this include better resource utilisation, better throughput and smoother reaction to failures. The project CarFi, 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.

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 emphasises on the utility 5.3 of our work.

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. This has been done with [1, LibOS] (an MPTCP version of the library operating system of the linux kernel) and [2, DCE] (Direct Code Execution). Everything was done on a XUbuntu 14.04 64bit virtual machine with DCE 1.8. The following illustrates how :

1. **Install the dependencies :**

   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 :**

   (a) hg clone http://code.nsnam.org/bake bake

   (b) export BAKE_HOME=‘pwd‘/bake

   (c) export PATH=$PATH :$BAKE_HOME

   (d) export PYTHONPATH=$PYTHONPATH :$BAKE_HOME

   (e) mkdir dce

   (f) cd dce

   (g) bake.py configure -e dce-ns3-1.8

   (h) bake.py download

   (i) bake.py build

3. **Build the *mptcp_trunk_libos* branch of *net-next-nuse***

   (a) git clone -b mptcp_trunk_libos https://github.com/libos-nuse/net-next-nuse.git

   (b) cd net-next-nuse

(c) make menuconfig ARCH=lib

(d) make library ARCH=lib

(e) Since DCE by default, calls the library *liblinux.so* (not exactly the correct one), and that the correct library is *libsim-linux.so* found at *$HOME/net-next-nuse/arch/lib/tools* we rename the existing *liblinux.so* to *liblinux0.so* and create a symbolic link for the correct library as follows :

ln -s $HOME/net-next-nuse/arch/lib/tools/libsim-linux.so $HOME/net-next-nuse/liblinux.so. This will "mislead" DCE into loading the correct library.

4. **Build *iproute2* version *2.6.38***

(a) Download the compressed source code from *https://kernel.googlesource.com/pub/scm/linux/kernel/git/shemminger/iproute2/+archive/fcae78992cab7bd267785b392b438306c621e583.tar.gz* , extract it and rename the folder to *iproute2-2.6.38*.

(b) cd iproute2-2.6.38

(c) patch -p1 -i ../ns-3-dce/utils/iproute-2.6.38-fix-01.patch

(d) $(KERNEL_INCLUDE) should point to the liblinux.so directory (for me it is $HOME/net-next-nuse)

Hence I modified the following part in the Makefile :

Config :

sh configure /home/lawrence/net-next-nuse

 # sh configure $(KERNEL_MODULE)

(e) LDFLAGS=-pie make CCOPTS='-fpic -D_GNU_SOURCE -O0 -U_FORTIFY_SOURCE'

5. **Set the *DCE_PATH***

export DCE_PATH=$HOME/net-next-nuse :$HOME/iproute2-2.6.38/ip

6. **Build *ns-3-dce***

(a) hg clone http://code.nsnam.org/ns-3-dce -r dce-1.8

(b) cd ns-3-dce

(c) ./waf configure –with-ns3=$HOME/dce/build –enable-kernel-stack=$HOME/net-next-nuse/arch –prefix=$HOME/dce/build

(d) ./waf build

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

(a) cd ns-3-dce

(b) Without *GDB* : *./waf –run dce-iperf-mptcp*

(c) 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 our project CarFi, 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. 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 [3] :

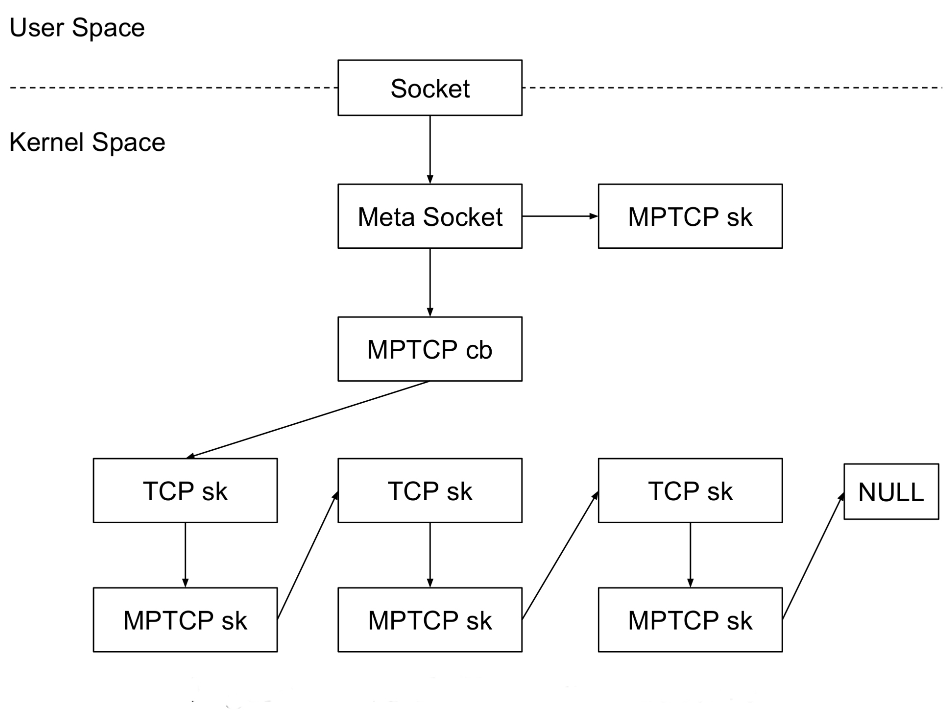

FIGURE 1 – MPTCP socket structure

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 [3] :

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 [3] :
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;
```

```c
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()
```

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 [4]. 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). 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.

What happens is, the client usually connects to the server via the default interface. Then according to the option passed in the **Netcat** command, it either opens subflows

on a single interface or on 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 :

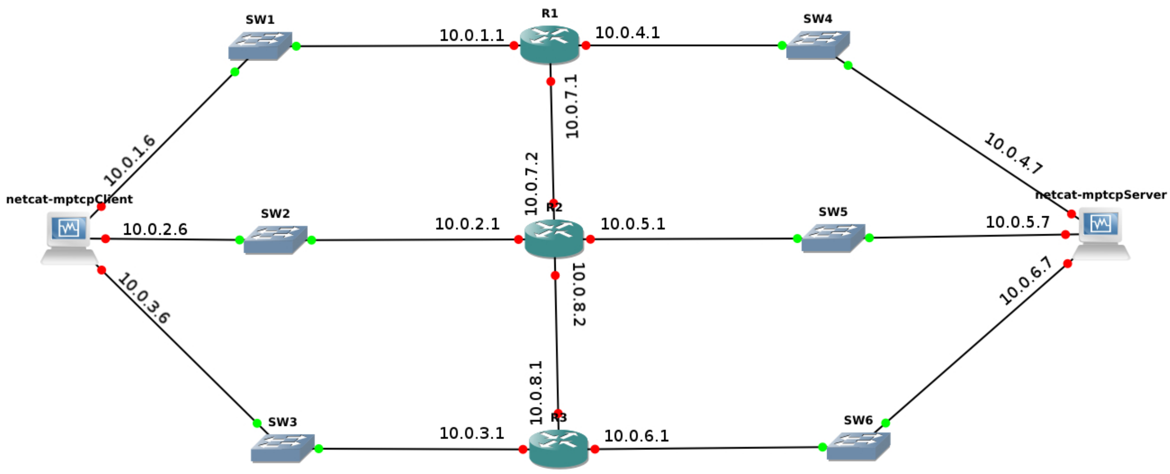

FIGURE 2 – Testbed topology for netcat-mptcp

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, 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 [5].
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**, 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 sunflow 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.
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 avaiable at the **src** folder of the github repository : [6, 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
2. In ***netcat-mptcp/src/Makefile.in*** line **153 - 164** and **183 - 186** : Appendix Makefile.in
3. In ***netcat-mptcp/src/netcat.h*** line **203 - 205** : Appendix netcat.h
4. In ***netcat-mptcp/src/netcat.c*** line **58 - 60**, **192**, **194**, **222**, **227**, **233 - 235**, **239 - 241**, **357 - 359** : Appendix netcat.c
5. In ***netcat-mptcp/src/core.h*** line **1 - 37** : Appendix core.h
6. In ***netcat-mptcp/src/core.c*** line **394 - 405** and **535 - 688** : Appendix core.c

5 Results, Statistics and Utility

5.1 Results

5.2 Statistics

5.3 Utility

6 Conclusion

Conclusion

7 Further developments

In the above experiments

8 Acknowledgements

Acknowledgement

9 Bibliography

Références

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

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

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

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

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

[6] 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>
```

```c
#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 :

```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");
}
```

.
.
.

```c
/* ... */


void addAllSubflows(int sock) {

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

        // d'abord trouver les interfaces disponible, puis tablir 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) {
```

```c
            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;
        }


```

```c
}


/* ... */

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;
}


/* ... */
```

11 Glossary

RA : *Département d'Informatique*

IETF : *Internet Engineering Task Force*

L2 : *Layer 2/Link Layer of the OSI model*

L3 : *Layer 2/IP Layer of the OSI model*

DHCP : *Dynamic Host Configuration Protocol*

DNS : *Domain name system*

MLD : *Multicast Listener Discovery*

IP : *Internet Protocol*

RFC : *Request for Comment*
ARP : *Address Resolution Protocol*
VLAN : *Virtual local area network*
AP : *Access Point*
RS : *Router Solicitation*
NS : *Neighbour Solicitation*
NA : *Neighbour Advertisement*
mDNS : *multicast Domain Name System*
LLMNR : *Link-Local Multicast Name Resolution*
SLAAC : *Stateless Address Autoconfiguration*