Multipath TCP – documentation

Release 2022

Olivier Bonaventure

Nov 07, 2022

CONTENTS:

CHAPTER

ONE

MULTIPATH TCP ON RECENT LINUX KERNELS

Starting with version 5.6, the official Linux kernel includes support for Multipath TCP. The set of features supported by this implementation has increased over time as shown by its ChangeLog.

To avoid any interference with regular TCP, this implementation only creates a Multipath TCP connection if the application has created its `socket` using the `IPPROTO_MPTCP` protocol. Applications will probably be modified in the coming months and years to add specific support for Multipath TCP, but in the mean time, the Multipath TCP developers have created a work around to force legacy applications to use Multipath TCP with the `mptcpize` command which is bundled with the mptcpd daemon. We use this solution in this section.

To illustrate Multipath TCP, we use a very simple setup with a Linux client using Ubuntu 22 and a Linux server using Debian. The client uses Linux kernel version 5.15 while the server uses version 5.17. The server has a single network interface with an IPv4 and an IPv6 address. The client has both a Wi-Fi and an Ethernet interface. These two interfaces are connected to the same router that allocates IP addresses in the same subnet on both interfaces. The client has both an IPv4 and an IPv6 address.

1.1 Enabling Multipath TCP

Multipath TCP is a feature that needs to be compiled inside the kernel. If you compile your own kernel, you can manually select Multipath TCP.

Most users prefer to rely on already compiled Linux kernels that are included in their distribution. The following distributions support Multipath TCP:

- CentOS starting with
- Debian starting with
- Ubuntu starting with 22.04

You need to to install a recent kernel to benefit from Multipath TCP. On some distributions, this installation will be part of the regular upgrade. On other distributions, you will need to add it manually.

Once the kernel has been installed and your computer has rebooted, you first need to verify that Multipath TCP is enabled.

```
sudo sysctl -a | grep mptcp.enabled
net.mptcp.enabled = 1
```

Here, the kernel supports Multipath TCP. If, for any reason, you want to disable Multipath TCP, you need to set this `sysctl` variable to `0`.

To illustrate the basic operation of `mptcpize`, let us first use the netcat command over the loopback interface. This is obviously not the target use case for Multipath TCP, but a nice way to perform tests.

Netcat allows to easily launch clients and servers. We start the server using: `mptcpize run nc -l -p 12345`. This is a TCP server that listens on port `12345`. The client connects to this server using the `mptcpize run nc 127.0.0.1 12345` command. The connection is established and all text lines sent by the client are printed by the server on standard output.

```
# mptcpize run nc -l -p 12345
Simple test
```

There are several ways to check that Multipath TCP is used for this connection. First, the `ss` command provides information about the status of the different sockets.

```
ss -iaM
State    Recv-Q    Send-Q       Local Address:Port        Peer Address:Port    Process
ESTAB    0         0                127.0.0.1:12345          127.0.0.1:52854
         subflows_max:2 remote_key token:5bba80d9 write_seq:2266a099179e2476 snd_
↪una:2266a099179e2476 rcv_nxt:de9999038d0a29a2
ESTAB    0         0                127.0.0.1:52854          127.0.0.1:12345
         subflows_max:2 remote_key token:c1f12b87 write_seq:de9999038d0a29a2 snd_
↪una:de9999038d0a29a2 rcv_nxt:2266a099179e2476
```

ss provides several useful information to debug a Multipath TCP connection. The first column indicates that the connection is in the Established state, which means that it can currently transfer data. It also indicates the length of the Send and Receive queues at the TCP level and the four-tuple that identifies the connection. The next line provides Multipath TCP information with the maximum number of subflows which can be attached to the connection, the token assigned by the remote host and the write_seq, snd_una and rcv_next parameters of the sate machine. The next two lines provide information about the other direction of the connection.

It is also possible to capture packets on the loopback interface to verify that Multipath TCP is used. The output below provides the first collected packets:

```
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode listening on␣
↪lo, link-type EN10MB (Ethernet), snapshot length 262144 bytes
18:43:42.676396 IP 127.0.0.1.52854 > 127.0.0.1.12345: Flags [S], seq 904893125, win␣
↪65495, options [mss 65495,sackOK,TS val 4026038040 ecr 0,nop,wscale 7,mptcp capable␣
↪v1], length 0
18:43:42.676426 IP 127.0.0.1.12345 > 127.0.0.1.52854: Flags [S.], seq 1804351310, ack␣
↪904893126, win 65483, options [mss 65495,sackOK,TS val 4026038040 ecr 4026038040,nop,
↪wscale 7,mptcp capable v1 {0x45edb502d861e7b1}], length 0
18:43:42.676472 IP 127.0.0.1.52854 > 127.0.0.1.12345: Flags [.], ack 1, win 512, options␣
↪[nop,nop,TS val 4026038040 ecr 4026038040,mptcp capable v1 {0xdbb760db1d55e07b,
↪0x45edb502d861e7b1}], length 0
18:44:59.519697 IP 127.0.0.1.52854 > 127.0.0.1.12345: Flags [P.], seq 1:13, ack 1, win␣
↪512, options [nop,nop,TS val 4026114884 ecr 4026038040,mptcp capable v1
↪{0xdbb760db1d55e07b,0x45edb502d861e7b1},nop,nop], length 12
18:44:59.519755 IP 127.0.0.1.12345 > 127.0.0.1.52854: Flags [.], ack 13, win 512,␣
↪options [nop,nop,TS val 4026114884 ecr 4026114884,mptcp dss ack 16040019788386937262],␣
↪length 0
```

The first packet is the `SYN` that includes the `MP_CAPABLE` option. The server replies with the `SYN+ACK` with the `MP_CAPABLE` containing the server key. The client returns the third `ACK` with the `MP_CAPABLE` and the two keys. As the server did not send any data, the `MP_CAPABLE` option is sent again in the packet containing the `Simple test` string. This packet also contains the `DSS` option. The server replies with an acknowledgment that carries the `DSS` option.

We can now use the netcat application to explore the operation of Multipath TCP over the Internet. Let us start with a very simple example.

```
mptcpize run nc serverv4 12345
hello
```

The netcat process listens on port 12345 on the server. This results in the following Multipath TCP connection :

```
09:05:23.695876 IP host-78-129-5-171.dynamic.voo.be.41510 > serverv4.12345: Flags [S],␣
↪seq 3525674027, win 64240, options [mss 1460,sackOK,TS val 2619832768 ecr 0,nop,wscale␣
↪7,mptcp capable v1], length 0
09:05:23.696076 IP serverv4.12345 > host-78-129-5-171.dynamic.voo.be.41510: Flags [S.],␣
↪seq 1745741580, ack 3525674028, win 65160, options [mss 1460,sackOK,TS val 3069340264␣
↪ecr 2619832768,nop,wscale 7,mptcp capable v1 {0x82aa42ef4245f0d0}], length 0
09:05:23.707909 IP host-78-129-5-171.dynamic.voo.be.41510 > serverv4.12345: Flags [.],␣
↪ack 1, win 502, options [nop,nop,TS val 2619832783 ecr 3069340264,mptcp capable v1
↪{0x9dc8e3972e3d9f25,0x82aa42ef4245f0d0}], length 0
09:05:30.776312 IP host-78-129-5-171.dynamic.voo.be.41510 > serverv4.12345: Flags [P.],␣
↪seq 1:7, ack 1, win 502, options [nop,nop,TS val 2619839851 ecr 3069340264,mptcp␣
↪capable v1 {0x9dc8e3972e3d9f25,0x82aa42ef4245f0d0},nop,nop], length 6
09:05:30.776484 IP serverv4.12345 > host-78-129-5-171.dynamic.voo.be.41510: Flags [.],␣
↪ack 7, win 510, options [nop,nop,TS val 3069347345 ecr 2619839851,mptcp dss ack␣
↪1561335003985645838], length 0
```

This is a Multipath TCP connection since it includes the Multipath TCP options, but the client does not create an additional subflow and the server does not announce its other addresses. This is the expected behavior since these operations are controlled by the path manager. On Linux, the Multipath TCP path manager can be configured using the ip-mptcp command. This command can be used to configure different parameters that are associated to an IP address. The path manager associates a numeric identifier to each IP address or endpoint. The `ip mptcp endpoint show` command lists the identifiers of the active IP addresses on the host. Here is an example of the output of this command on our client:

```
ip mptcp endpoint show
fe80::3934:7572:b1ff:b555 id 1 dev wlp3s0
192.168.0.43 id 2 dev wlp3s0
fe80::5642:39bd:3390:43d3 id 3 dev enp2s0
192.168.0.37 id 4 dev enp2s0
2a02:2788:10c4:123:3d66:f590:d891:8fb3 id 5 dev wlp3s0
2a02:2788:10c4:123:6636:10c6:692b:18cc id 6 dev enp2s0
2a02:2788:10c4:123:2a09:5ec7:9b99:4a97 id 7 dev enp2s0
```

The two `fe80` addresses are the IPv6 link local addresses configured on the Ethernet (`enp2s0`) and Wi-Fi (`wlp3s0`) interfaces of our client host. There are three flags which can be associated with each endpoint identifier:

- `subflow`. When this flag is set, the path manager will try to create a subflow over this interface when a Multipath TCP is created or the interface becomes active while there was an ongoing Multipath TCP connection. This flag is mainly useful for clients.
- `signal`. When this flag is set, the path manager will announce the address of the endpoint over any Multipath TCP connection created using other addresses. This flag can be used on clients or servers. It is mainly useful on servers that have multiple addresses.
- `backup`. This flag can be combined with the two other flags. When combined with the `subflow` flag, it indicates that a backup subflow will be created. When combined with the `signal` flag, it indicates that the address will be advertised as a backup address.

On our client host, we can configure the Wi-Fi interface as a backup interface that creates subflows as follows :

```
sudo ip mptcp endpoint del id 2
sudo ip mptcp endpoint add 192.168.0.43 dev wlp3s0 subflow backup
sudo ip mptcp endpoint show
fe80::3934:7572:b1ff:b555 id 1 dev wlp3s0
fe80::5642:39bd:3390:43d3 id 3 dev enp2s0
192.168.0.37 id 4 dev enp2s0
2a02:2788:10c4:123:3d66:f590:d891:8fb3 id 5 dev wlp3s0
2a02:2788:10c4:123:6636:10c6:692b:18cc id 6 dev enp2s0
2a02:2788:10c4:123:2a09:5ec7:9b99:4a97 id 7 dev enp2s0
192.168.0.43 id 8 subflow backup dev wlp3s0
```

We had to first remove the configuration for this endpoint because a default one was already active. Then we added the new parameters and verified them.

The path manager also has some limits which can be configured using the `ip mptcp limits` command. Two limits can be set.

- `ip mptcp limits set subflow n` where `n` is an integer. This restricts the Multipath TCP connection to use up to `n` different subflows. Servers should protect themselves by setting this limit to a few subflows. Most use cases would work well with 2 or 4 subflows.
- `ip mptcp limits set add_addr_accepted n` where `n` is an integer. This parameter limits the number of addresses that are learned over each Multipath TCP connection. This parameter could be used to protect the Multipath TCP implementation against attacks where two many addresses are advertised. Most use cases would work with 4 accepted addresses.

These parameters control the path manager, but before creating Multipath TCP subflows over different paths, we need to configure the IP routing table of our client host. Our client host has two network interfaces: Wi-Fi and Ethernet. By default, Linux prefers the Ethernet interface to Wi-Fi. The two interfaces are configured as follows :

```
ip -4 addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen␣
↪1000
    inet 127.0.0.1/8 scope host lo
    valid_lft forever preferred_lft forever
2: enp2s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group␣
↪default qlen 1000
    inet 192.168.0.37/24 brd 192.168.0.255 scope global dynamic noprefixroute enp2s0
    valid_lft 75697sec preferred_lft 75697sec
3: wlp3s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group␣
↪default qlen 1000
    inet 192.168.0.43/24 brd 192.168.0.255 scope global dynamic noprefixroute wlp3s0
    valid_lft 75696sec preferred_lft 75696sec
```

By default, Linux creates the two following default routes.

```
route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.0.1     0.0.0.0         UG    100    0        0 enp2s0
0.0.0.0         192.168.0.1     0.0.0.0         UG    600    0        0 wlp3s0
```

We need to configure the routing tables to be able to use the two interfaces simultaneously. For this, we need to ensure that packets with source address `192.168.0.37` are sent over the `enp2s0` interface while packets with source address `192.168.0.43` are sent over the `wlp3s0` interface. This can be achieved using two different routing tables.

```
# create the two routing tables
ip rule add from 192.168.0.37 table 1
ip rule add from 192.168.0.43 table 2

# Configure routing table 1 for enp2s0
ip route add 192.168.0.0/24 dev enp2s0 scope link table 1
ip route add default via 192.168.0.1 dev enp2s0 table 1

# Configure routing table 2 for wlp3s0
ip route add 192.168.0.0/24 dev wlp3s0 scope link table 2
ip route add default via 192.168.0.1 dev wlp3s0 table 2

# Configure a default route to regular internet
ip route add default scope global nexthop via 192.168.0.1 dev enp2s0
```

We can check the routing tables using the ip command.

```
ip rule show
0:   from all lookup local
32764:       from 192.168.0.43 lookup 2
32765:       from 192.168.0.37 lookup 1
32766:       from all lookup main
32767:       from all lookup default

ip route
default via 192.168.0.1 dev enp2s0
default via 192.168.0.1 dev enp2s0 proto dhcp metric 100
default via 192.168.0.1 dev wlp3s0 proto dhcp metric 600
169.254.0.0/16 dev wlp3s0 scope link metric 1000
192.168.0.0/24 dev enp2s0 proto kernel scope link src 192.168.0.37 metric 100
192.168.0.0/24 dev wlp3s0 proto kernel scope link src 192.168.0.43 metric 600

ip route show table 1
default via 192.168.0.1 dev enp2s0
192.168.0.0/24 dev enp2s0 scope link

ip route show table 2
default via 192.168.0.1 dev wlp3s0
192.168.0.0/24 dev wlp3s0 scope link
```

We can verify that the two routing tables are correct using `nc` by forcing it to use a specific source address.

```
echo -e "GET / HTTP/1.0\r\n" | nc -4 -s 192.168.0.37 test.multipath-tcp.org 80
HTTP/1.0 200 OK
Content-Type: text/html
ETag: "4215149735"
Last-Modified: Tue, 05 Jul 2022 16:11:47 GMT
Content-Length: 389
Connection: close
Date: Wed, 06 Jul 2022 11:34:24 GMT
Server: lighttpd/1.4.59

<!DOCTYPE html>
```

(continues on next page)

(continued from previous page)

```
<html>
<head>
<title>Welcome to test.multipath-tcp.org!</title>
<style>
body {
width: 35em;
margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif;
}
</style>
</head>
<body>
<h1>Welcome to test.multipath-tcp.org !</h1>
<p>This web server runs Multipath TCP v1</p>

<p><em>Thank you for using Multipath TCP.</em></p>
</body>
</html>
```

You should get the same result when using the second interface, IP address `192.168.0.43` in our example.

```
echo -e "GET / HTTP/1.0\r\n" | nc -4 -s 192.168.0.43 test.multipath-tcp.org 80
```

The next step is to verify that Multipath TCP is working correctly and that two subflows are created. For this, we'll use the `-i` parameter of `nc` to add a delay between the two lines of the HTTP GET. We will leverage this delay to check that MPTCP is correctly working using `ss` or `tcpdump`

```
echo -e "GET / HTTP/1.0\r\n" | mptcpize run nc -4 -i 5 test.multipath-tcp.org 80
```

We can observe the creation of the connection and the subflow using both `ss` and `tcpdump`. `ss` shows that there are two subflows towards `test.multipath-tcp.org`.

```
ss -4 -iatM  dst test.multipath-tcp.org
Netid  State  Recv-Q  Send-Q         Local Address:Port      Peer Address:Port  Process
tcp    ESTAB  0       0        192.168.0.43%wlp3s0:34801     5.196.67.207:http
      cubic wscale:7,7 rto:220 rtt:17.439/8.719 mss:1448 pmtu:1500 rcvmss:536␣
↪advmss:1448 cwnd:10 bytes_acked:1 segs_out:2 segs_in:2 send 6.64Mbps lastsnd:1776␣
↪lastrcv:1776 lastack:1764 pacing_rate 13.3Mbps delivered:1 rcv_space:14480 rcv_
↪ssthresh:64088 minrtt:17.439
tcp    ESTAB  0       0               192.168.0.37:47672     5.196.67.207:http
      cubic wscale:7,7 rto:216 rtt:14/5.405 mss:1448 pmtu:1500 rcvmss:536 advmss:1448␣
↪cwnd:10 bytes_sent:16 bytes_acked:17 segs_out:3 segs_in:3 data_segs_out:1 send 8.
↪27Mbps lastsnd:1808 lastrcv:1808 lastack:1792 pacing_rate 16.5Mbps delivery_rate␣
↪790kbps delivered:2 busy:16ms rcv_space:14480 rcv_ssthresh:64088 minrtt:13.905
mptcp  ESTAB  0       0               192.168.0.37:47672     5.196.67.207:http
      subflows:1 subflows_max:8 remote_key token:e1e3cdeb write_seq:1045ecfa3f05f4ea snd_
↪una:1045ecfa3f05f4ea rcv_nxt:d0568f430363c9aa
```

The line starting with `mptcp` indicates that the Multipath TCP connection above has one additional subflow.

The `tcpdump` output reveals precisely which packets have been sent over each network interface.

```
sudo tcpdump -n -i any host test.multipath-tcp.org and port 80tcpdump: data link type␣
↪LINUX_SLL2
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on any, link-type LINUX_SLL2 (Linux cooked v2), snapshot length 262144 bytes
13:43:26.620667 enp2s0 Out IP 192.168.0.37.47672 > 5.196.67.207.80: Flags [S], seq␣
↪3585891423, win 64240, options [mss 1460,sackOK,TS val 3993892549 ecr 0,nop,wscale 7,
↪mptcp capable v1], length 0
13:43:26.634537 enp2s0 In  IP 5.196.67.207.80 > 192.168.0.37.47672: Flags [S.], seq␣
↪1788691420, ack 3585891424, win 65160, options [mss 1460,sackOK,TS val 1030255900 ecr␣
↪3993892549,nop,wscale 7,mptcp capable v1 {0x54f04ad5bd2d9f42}], length 0
13:43:26.634609 enp2s0 Out IP 192.168.0.37.47672 > 5.196.67.207.80: Flags [.], ack 1,␣
↪win 502, options [nop,nop,TS val 3993892563 ecr 1030255900,mptcp capable v1
↪{0xff2ec3a2f6151881,0x54f04ad5bd2d9f42}], length 0
13:43:26.634718 enp2s0 Out IP 192.168.0.37.47672 > 5.196.67.207.80: Flags [P.], seq 1:17,
↪ ack 1, win 502, options [nop,nop,TS val 3993892563 ecr 1030255900,mptcp capable v1
↪{0xff2ec3a2f6151881,0x54f04ad5bd2d9f42},nop,nop], length 16: HTTP: GET / HTTP/1.0
13:43:26.649351 enp2s0 In  IP 5.196.67.207.80 > 192.168.0.37.47672: Flags [.], ack 17,␣
↪win 509, options [nop,nop,TS val 1030255916 ecr 3993892563,mptcp dss ack␣
↪1172603837543216362], length 0
13:43:26.649351 enp2s0 In  IP 5.196.67.207.80 > 192.168.0.37.47672: Flags [.], ack 17,␣
↪win 509, options [nop,nop,TS val 1030255916 ecr 3993892563,mptcp dss ack␣
↪1172603837543216362], length 0
13:43:26.649498 wlp3s0 Out IP 192.168.0.43.34801 > 5.196.67.207.80: Flags [S], seq␣
↪2321572505, win 64240, options [mss 1460,sackOK,TS val 1218002018 ecr 0,nop,wscale 7,
↪mptcp join id 8 token 0xeef7df2f nonce 0xc0d346f6], length 0
13:43:26.666895 wlp3s0 In  IP 5.196.67.207.80 > 192.168.0.43.34801: Flags [S.], seq␣
↪1973196884, ack 2321572506, win 65160, options [mss 1460,sackOK,TS val 1030255931 ecr␣
↪1218002018,nop,wscale 7,mptcp join id 0 hmac 0xc7489cf7056428b4 nonce 0xa54f9af],␣
↪length 0
13:43:26.666966 wlp3s0 Out IP 192.168.0.43.34801 > 5.196.67.207.80: Flags [.], ack 1,␣
↪win 502, options [nop,nop,TS val 1218002035 ecr 1030255931,mptcp join hmac␣
↪0xb4e6a41bf5861313df7f5f454966998ad7e698a4], length 0
13:43:26.677776 wlp3s0 In  IP 5.196.67.207.80 > 192.168.0.43.34801: Flags [.], ack 1,␣
↪win 510, options [nop,nop,TS val 1030255944 ecr 1218002035,mptcp dss ack␣
↪1172603837543216362], length 0
13:43:31.635023 enp2s0 Out IP 192.168.0.37.47672 > 5.196.67.207.80: Flags [P.], seq␣
↪17:18, ack 1, win 502, options [nop,nop,TS val 3993897563 ecr 1030255916,mptcp dss ack␣
↪56871338 seq 1172603837543216362 subseq 17 len 1,nop,nop], length 1: HTTP
13:43:31.646703 enp2s0 In  IP 5.196.67.207.80 > 192.168.0.37.47672: Flags [.], ack 18,␣
↪win 509, options [nop,nop,TS val 1030260913 ecr 3993897563,mptcp dss ack␣
↪1172603837543216363], length 0
13:43:31.647276 enp2s0 In  IP 5.196.67.207.80 > 192.168.0.37.47672: Flags [P.], seq␣
↪1:602, ack 18, win 509, options [nop,nop,TS val 1030260914 ecr 3993897563,mptcp dss␣
↪ack 1172603837543216363 seq 15012343925868579242 subseq 1 len 601,nop,nop], length␣
↪601: HTTP: HTTP/1.0 200 OK
13:43:31.647300 enp2s0 Out IP 192.168.0.37.47672 > 5.196.67.207.80: Flags [.], ack 602,␣
↪win 501, options [nop,nop,TS val 3993897576 ecr 1030260914,mptcp dss ack␣
↪15012343925868579843], length 0
13:43:31.647276 enp2s0 In  IP 5.196.67.207.80 > 192.168.0.37.47672: Flags [.], ack 18,␣
↪win 509, options [nop,nop,TS val 1030260914 ecr 3993897563,mptcp dss fin ack␣
↪1172603837543216363 seq 15012343925868579843 subseq 0 len 1,nop,nop], length 0
13:43:31.647321 enp2s0 Out IP 192.168.0.37.47672 > 5.196.67.207.80: Flags [.], ack 602,␣
↪win 501, options [nop,nop,TS val 3993897576 ecr 1030260914,mptcp dss ack␣
↪15012343925868579844], length 0
```

(continues on next page)

(continued from previous page)

```
13:43:31.647330 wlp3s0 Out IP 192.168.0.43.34801 > 5.196.67.207.80: Flags [.], ack 1,␣
↪win 502, options [nop,nop,TS val 1218002046 ecr 1030255944,mptcp dss ack␣
↪15012343925868579844], length 0
13:43:31.648565 wlp3s0 In  IP 5.196.67.207.80 > 192.168.0.43.34801: Flags [.], ack 1,␣
↪win 510, options [nop,nop,TS val 1030255944 ecr 1218002035,mptcp dss fin ack␣
↪1172603837543216363 seq 15012343925868579843 subseq 0 len 1,nop,nop], length 0
13:43:36.635392 enp2s0 Out IP 192.168.0.37.47672 > 5.196.67.207.80: Flags [.], ack 602,␣
↪win 501, options [nop,nop,TS val 3993897576 ecr 1030260914,mptcp dss fin ack␣
↪15012343925868579844 seq 1172603837543216363 subseq 0 len 1,nop,nop], length 0
13:43:36.635416 wlp3s0 Out IP 192.168.0.43.34801 > 5.196.67.207.80: Flags [.], ack 1,␣
↪win 502, options [nop,nop,TS val 1218007017 ecr 1030255944,mptcp dss fin ack␣
↪15012343925868579844 seq 1172603837543216363 subseq 0 len 1,nop,nop], length 0
13:43:36.636468 enp2s0 Out IP 192.168.0.37.47672 > 5.196.67.207.80: Flags [.], ack 602,␣
↪win 501, options [nop,nop,TS val 3993897576 ecr 1030260914,mptcp dss fin ack␣
↪15012343925868579844 seq 1172603837543216363 subseq 0 len 1,nop,nop], length 0
13:43:36.636482 wlp3s0 Out IP 192.168.0.43.34801 > 5.196.67.207.80: Flags [.], ack 1,␣
↪win 502, options [nop,nop,TS val 1218007017 ecr 1030255944,mptcp dss fin ack␣
↪15012343925868579844 seq 1172603837543216363 subseq 0 len 1,nop,nop], length 0
13:43:36.640425 enp2s0 Out IP 192.168.0.37.47672 > 5.196.67.207.80: Flags [.], ack 602,␣
↪win 501, options [nop,nop,TS val 3993897576 ecr 1030260914,mptcp dss fin ack␣
↪15012343925868579844 seq 1172603837543216363 subseq 0 len 1,nop,nop], length 0
13:43:36.640431 wlp3s0 Out IP 192.168.0.43.34801 > 5.196.67.207.80: Flags [.], ack 1,␣
↪win 502, options [nop,nop,TS val 1218007017 ecr 1030255944,mptcp dss fin ack␣
↪15012343925868579844 seq 1172603837543216363 subseq 0 len 1,nop,nop], length 0
13:43:36.645605 enp2s0 In  IP 5.196.67.207.80 > 192.168.0.37.47672: Flags [.], ack 18,␣
↪win 509, options [nop,nop,TS val 1030265912 ecr 3993897576,mptcp dss ack␣
↪1172603837543216364], length 0
13:43:36.645659 enp2s0 Out IP 192.168.0.37.47672 > 5.196.67.207.80: Flags [F.], seq 18,␣
↪ack 602, win 501, options [nop,nop,TS val 3993902574 ecr 1030265912,mptcp dss ack␣
↪15012343925868579844], length 0
13:43:36.645674 wlp3s0 Out IP 192.168.0.43.34801 > 5.196.67.207.80: Flags [F.], seq 1,␣
↪ack 1, win 502, options [nop,nop,TS val 1218012014 ecr 1030255944,mptcp dss ack␣
↪15012343925868579844], length 0
13:43:36.646315 wlp3s0 In  IP 5.196.67.207.80 > 192.168.0.43.34801: Flags [.], ack 1,␣
↪win 510, options [nop,nop,TS val 1030260930 ecr 1218002046,mptcp dss ack␣
↪1172603837543216364], length 0
13:43:36.647699 enp2s0 In  IP 5.196.67.207.80 > 192.168.0.37.47672: Flags [F.], seq 602,␣
↪ack 18, win 509, options [nop,nop,TS val 1030265912 ecr 3993897576,mptcp dss ack␣
↪1172603837543216364], length 0
13:43:36.647718 enp2s0 Out IP 192.168.0.37.47672 > 5.196.67.207.80: Flags [.], ack 603,␣
↪win 501, options [nop,nop,TS val 3993902576 ecr 1030265912,mptcp dss ack␣
↪15012343925868579844], length 0
13:43:36.648629 wlp3s0 In  IP 5.196.67.207.80 > 192.168.0.43.34801: Flags [F.], seq 1,␣
↪ack 1, win 510, options [nop,nop,TS val 1030265912 ecr 1218002046,mptcp dss ack␣
↪1172603837543216364], length 0
13:43:36.648649 wlp3s0 Out IP 192.168.0.43.34801 > 5.196.67.207.80: Flags [.], ack 2,␣
↪win 502, options [nop,nop,TS val 1218012017 ecr 1030265912,mptcp dss ack␣
↪15012343925868579844], length 0
13:43:36.657040 enp2s0 In  IP 5.196.67.207.80 > 192.168.0.37.47672: Flags [.], ack 19,␣
↪win 509, options [nop,nop,TS val 1030265923 ecr 3993902574,mptcp dss ack␣
↪1172603837543216364], length 0
13:43:36.662211 wlp3s0 In  IP 5.196.67.207.80 > 192.168.0.43.34801: Flags [.], ack 2,␣
↪win 510, options [nop,nop,TS val 1030265928 ecr 1218012014,mptcp dss ack␣
↪1172603837543216364], length 0
```

(continues on next page)

(continued from previous page)

```

```

If your host is dual stack, you also need to do the same configuration for IPv6 as well. Our test host uses the following IPv6 addresses.

```
ip -6 addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 state UNKNOWN qlen 1000
    inet6 ::1/128 scope host
    valid_lft forever preferred_lft forever
2: enp2s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 state UP qlen 1000
    inet6 2a02:2788:10c4:123:f468:1851:9a9f:7d44/64 scope global temporary dynamic
    valid_lft 592298sec preferred_lft 73422sec
    inet6 2a02:2788:10c4:123:6636:10c6:692b:18cc/64 scope global dynamic mngtmpaddr␣
↪noprefixroute
    valid_lft 1209600sec preferred_lft 604800sec
    inet6 fe80::5642:39bd:3390:43d3/64 scope link noprefixroute
    valid_lft forever preferred_lft forever
3: wlp3s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 state UP qlen 1000
    inet6 2a02:2788:10c4:123:3d66:f590:d891:8fb3/64 scope global dynamic noprefixroute
    valid_lft 1209600sec preferred_lft 604800sec
    inet6 fe80::3934:7572:b1ff:b555/64 scope link noprefixroute
    valid_lft forever preferred_lft forever
```

We thus had to configure the following IPv6 routing tables. This is similar to the commands used to configure the IPv4 routing tables.

```
ip -6 rule add from 2a02:2788:10c4:123:6636:10c6:692b:18cc table 1
ip -6 rule add from 2a02:2788:10c4:123:3d66:f590:d891:8fb3 table 2
ip route add 2a02:2788:10c4:123::/64 dev enp2s0 scope link table 1
ip route add 2a02:2788:10c4:123::/64 dev wlp3s0 scope link table 2
ip route add default via fe80::10:18ff:fe07:fc33 dev enp2s0 table 1
ip route add default via fe80::10:18ff:fe07:fc33 dev wlp3s0 table 2
ip route add default scope global nexthop via fe80::10:18ff:fe07:fc33 dev enp2s0
```

Remember that if you want to create subflows using IPv6 addresses, you also need to configure the stack with `ip mptcp endpoint add` as you did for the IPv4 addresses.

Note: The current versions of the Linux kernel only use one address family at a time. If a connection was created using IPv4, then only IPv4 addresses will be used to create new subflows. Future versions of the kernel will allow to mix IPv4 and IPv6 subflows.

1.2 Analyzing the output of ss

CHAPTER

TWO

ANALYZING THE OUTPUT OF NSTAT

The Linux TCP/IP stack maintains a lot of counters that track various events inside the kernel. These counters are very useful for system administrators who need to manage Linux hosts and debug some specific networking problems.

Linux supports a few hundred counters associated to the protocols in the network and transport layers. Other operating systems have defined their own counters to track similar networking events. Fortunately, the IETF has standard some counters that are common to different operating systems and TCP/IP implementations. These counters are exported as variables which can be queried using a management protocol such as SNMP. This enables a management server to collect statistics for a series of hosts to process and analyze them. Several versions of SNMP exist, but we will not discuss them in details in this document. Instead, we focus on the Linux TCP/IP implementation and explain the different counters that the nstat application exposes to the user.

Linux kernel version 5.18 collects 363 different counters that are divided in 7 categories :

- 67 counters track the IPv4 implementation
- 80 counters track the ICMPv4 implementation
- 32 counters track the IPv6 implementation
- 46 counters track ICMPv6
- 135 counters track TCP
- 35 counters track UDP
- 46 counters track Multipath TCP

Some of these counters are part of the Management Information Bases (MIB) defined within the IETF, e.g. **RFC 1213** for IPv4 and ICMPv4, **RFC 4293** for IPv6 and ICMPv6, **RFC 4022** for TCP, **RFC 4113** for UDP. As of this writing, there is no official IETF MIB for Multipath TCP.

2.1 Using nstat

In this document, we describe the counters that are exposed by nstat for the different protocols of the TCP/IP stack. Before discussing these counters, it is useful to understand how nstat works.

nstat is a command line tool that supports a small number of arguments

```
nstat --help
Usage: nstat [OPTION] [ PATTERN [ PATTERN ] ]
       -h, --help            this message
       -a, --ignore  ignore history
       -d, --scan=SECS       sample every statistics every SECS
       -j, --json            format output in JSON
```

(continues on next page)

(continued from previous page)

```
       -n, --nooutput        do history only
       -p, --pretty  pretty print
       -r, --reset           reset history
       -s, --noupdate        don't update history
       -t, --interval=SECS   report average over the last SECS
       -V, --version output version information
       -z, --zeros           show entries with zero activity
```

By default, nstat displays the counters whose value has changed since the latest invocation of the tool. This is usually a small subset of the counters that depends on the networking activity of the host.

nstat can collect historical information and provides average counters.

nstat can also list the current value of the different counters.

```
#nstat -az
       #kernel
       IpInReceives                    1073367            0.0
       IpInHdrErrors                   0                  0.0
       IpInAddrErrors                  0                  0.0
       IpForwDatagrams                 0                  0.0
       IpInUnknownProtos               0                  0.0
       IpInDiscards                    0                  0.0
       IpInDelivers                    1072518            0.0
       IpOutRequests                   484889             0.0
       IpOutDiscards                   0                  0.0
       IpOutNoRoutes                   0                  0.0
       IpReasmTimeout                  0                  0.0
       IpReasmReqds                    0                  0.0
       IpReasmOKs                      0                  0.0
       IpReasmFails                    0                  0.0
       IpFragOKs                       0                  0.0
       IpFragFails                     0                  0.0
       IpFragCreates                   0                  0.0
       IcmpInMsgs                      561                0.0
       IcmpInErrors                    125                0.0
       IcmpInCsumErrors                0                  0.0
       IcmpInDestUnreachs              6                  0.0
       IcmpInTimeExcds                 125                0.0
       IcmpInParmProbs                 0                  0.0
       IcmpInSrcQuenchs                0                  0.0
       IcmpInRedirects                 0                  0.0
       IcmpInEchos                     298                0.0
       IcmpInEchoReps                  0                  0.0
       IcmpInTimestamps                33                 0.0
       IcmpInTimestampReps             0                  0.0
       IcmpInAddrMasks                 99                 0.0
       IcmpInAddrMaskReps              0                  0.0
       IcmpOutMsgs                     331                0.0
       IcmpOutErrors                   0                  0.0
       IcmpOutDestUnreachs             0                  0.0
       IcmpOutTimeExcds                0                  0.0
       IcmpOutParmProbs                0                  0.0
```

(continues on next page)

(continued from previous page)

```
IcmpOutSrcQuenchs               0                  0.0
IcmpOutRedirects                0                  0.0
IcmpOutEchos                    0                  0.0
IcmpOutEchoReps                 298                0.0
IcmpOutTimestamps               0                  0.0
IcmpOutTimestampReps            33                 0.0
IcmpOutAddrMasks                0                  0.0
IcmpOutAddrMaskReps             0                  0.0
IcmpMsgInType3                  6                  0.0
IcmpMsgInType8                  298                0.0
IcmpMsgInType11                 125                0.0
IcmpMsgInType13                 33                 0.0
IcmpMsgInType17                 99                 0.0
IcmpMsgOutType0                 298                0.0
IcmpMsgOutType14                33                 0.0
TcpActiveOpens                  3330               0.0
TcpPassiveOpens                 252                0.0
TcpAttemptFails                 0                  0.0
TcpEstabResets                  78                 0.0
TcpInSegs                       3202615            0.0
TcpOutSegs                      6431616            0.0
TcpRetransSegs                  7584               0.0
TcpInErrs                       0                  0.0
TcpOutRsts                      102                0.0
TcpInCsumErrors                 0                  0.0
UdpInDatagrams                  18972              0.0
UdpNoPorts                      0                  0.0
UdpInErrors                     0                  0.0
UdpOutDatagrams                 19257              0.0
UdpRcvbufErrors                 0                  0.0
UdpSndbufErrors                 0                  0.0
UdpInCsumErrors                 0                  0.0
UdpIgnoredMulti                 19989              0.0
UdpMemErrors                    0                  0.0
UdpLiteInDatagrams              0                  0.0
UdpLiteNoPorts                  0                  0.0
UdpLiteInErrors                 0                  0.0
UdpLiteOutDatagrams             0                  0.0
UdpLiteRcvbufErrors             0                  0.0
UdpLiteSndbufErrors             0                  0.0
UdpLiteInCsumErrors             0                  0.0
UdpLiteIgnoredMulti             0                  0.0
UdpLiteMemErrors                0                  0.0
Ip6InReceives                   2198489            0.0
Ip6InHdrErrors                  0                  0.0
Ip6InTooBigErrors               0                  0.0
Ip6InNoRoutes                   200                0.0
Ip6InAddrErrors                 0                  0.0
Ip6InUnknownProtos              0                  0.0
Ip6InTruncatedPkts              0                  0.0
Ip6InDiscards                   0                  0.0
Ip6InDelivers                   2177604            0.0
```

(continues on next page)

(continued from previous page)

```
       Ip6OutForwDatagrams             0                  0.0
       Ip6OutRequests                  1567967            0.0
       Ip6OutDiscards                  0                  0.0
       Ip6OutNoRoutes                  6                  0.0
       Ip6ReasmTimeout                 0                  0.0
       Ip6ReasmReqds                   0                  0.0
       Ip6ReasmOKs                     0                  0.0
       Ip6ReasmFails                   0                  0.0
       Ip6FragOKs                      0                  0.0
       Ip6FragFails                    0                  0.0
       Ip6FragCreates                  0                  0.0
       Ip6InMcastPkts                  20785              0.0
       Ip6OutMcastPkts                 13                 0.0
       Ip6InOctets                     2578707266         0.0
       Ip6OutOctets                    3533261025         0.0
       Ip6InMcastOctets                1442288            0.0
       Ip6OutMcastOctets               1252               0.0
       Ip6InBcastOctets                0                  0.0
       Ip6OutBcastOctets               0                  0.0
       Ip6InNoECTPkts                  2060704            0.0
       Ip6InECT1Pkts                   0                  0.0
       Ip6InECT0Pkts                   137799             0.0
       Ip6InCEPkts                     0                  0.0
       Icmp6InMsgs                     7525               0.0
       Icmp6InErrors                   0                  0.0
       Icmp6OutMsgs                    7511               0.0
       Icmp6OutErrors                  0                  0.0
       Icmp6InCsumErrors               0                  0.0
       Icmp6InDestUnreachs             10                 0.0
       Icmp6InPktTooBigs               0                  0.0
       Icmp6InTimeExcds                0                  0.0
       Icmp6InParmProblems             0                  0.0
       Icmp6InEchos                    2                  0.0
       Icmp6InEchoReplies              6                  0.0
       Icmp6InGroupMembQueries         0                  0.0
       Icmp6InGroupMembResponses       0                  0.0
       Icmp6InGroupMembReductions      0                  0.0
       Icmp6InRouterSolicits           0                  0.0
       Icmp6InRouterAdvertisements     0                  0.0
       Icmp6InNeighborSolicits         4316               0.0
       Icmp6InNeighborAdvertisements   3189               0.0
       Icmp6InRedirects                0                  0.0
       Icmp6InMLDv2Reports             2                  0.0
       Icmp6OutDestUnreachs            0                  0.0
       Icmp6OutPktTooBigs              0                  0.0
       Icmp6OutTimeExcds               0                  0.0
       Icmp6OutParmProblems            0                  0.0
       Icmp6OutEchos                   6                  0.0
       Icmp6OutEchoReplies             2                  0.0
       Icmp6OutGroupMembQueries        0                  0.0
       Icmp6OutGroupMembResponses      0                  0.0
       Icmp6OutGroupMembReductions     0                  0.0
```

(continues on next page)

(continued from previous page)

```
        Icmp6OutRouterSolicits          0                  0.0
        Icmp6OutRouterAdvertisements    0                  0.0
        Icmp6OutNeighborSolicits        3179               0.0
        Icmp6OutNeighborAdvertisements  4316               0.0
        Icmp6OutRedirects               0                  0.0
        Icmp6OutMLDv2Reports            8                  0.0
        Icmp6InType1                    10                 0.0
        Icmp6InType128                  2                  0.0
        Icmp6InType129                  6                  0.0
        Icmp6InType135                  4316               0.0
        Icmp6InType136                  3189               0.0
        Icmp6InType143                  2                  0.0
        Icmp6OutType128                 6                  0.0
        Icmp6OutType129                 2                  0.0
        Icmp6OutType135                 3179               0.0
        Icmp6OutType136                 4316               0.0
        Icmp6OutType143                 8                  0.0
        Udp6InDatagrams                 460                0.0
        Udp6NoPorts                     0                  0.0
        Udp6InErrors                    0                  0.0
        Udp6OutDatagrams                95                 0.0
        Udp6RcvbufErrors                0                  0.0
        Udp6SndbufErrors                0                  0.0
        Udp6InCsumErrors                0                  0.0
        Udp6IgnoredMulti                0                  0.0
        Udp6MemErrors                   0                  0.0
        UdpLite6InDatagrams             0                  0.0
        UdpLite6NoPorts                 0                  0.0
        UdpLite6InErrors                0                  0.0
        UdpLite6OutDatagrams            0                  0.0
        UdpLite6RcvbufErrors            0                  0.0
        UdpLite6SndbufErrors            0                  0.0
        UdpLite6InCsumErrors            0                  0.0
        UdpLite6MemErrors               0                  0.0
        TcpExtSyncookiesSent            0                  0.0
        TcpExtSyncookiesRecv            0                  0.0
        TcpExtSyncookiesFailed          0                  0.0
        TcpExtEmbryonicRsts             0                  0.0
        TcpExtPruneCalled               3791               0.0
        TcpExtRcvPruned                 0                  0.0
        TcpExtOfoPruned                 0                  0.0
        TcpExtOutOfWindowIcmps          0                  0.0
        TcpExtLockDroppedIcmps          0                  0.0
        TcpExtArpFilter                 0                  0.0
        TcpExtTW                        2283               0.0
        TcpExtTWRecycled                0                  0.0
        TcpExtTWKilled                  0                  0.0
        TcpExtPAWSActive                0                  0.0
        TcpExtPAWSEstab                 11                 0.0
        TcpExtDelayedACKs               31995              0.0
        TcpExtDelayedACKLocked          47                 0.0
        TcpExtDelayedACKLost            282                0.0
```

(continues on next page)

(continued from previous page)

```
        TcpExtListenOverflows           0                  0.0
        TcpExtListenDrops               0                  0.0
        TcpExtTCPHPHits                 699069             0.0
        TcpExtTCPPureAcks               997468             0.0
        TcpExtTCPHPAcks                 1235546            0.0
        TcpExtTCPRenoRecovery           0                  0.0
        TcpExtTCPSackRecovery           2526               0.0
        TcpExtTCPSACKReneging           0                  0.0
        TcpExtTCPSACKReorder            36858              0.0
        TcpExtTCPRenoReorder            0                  0.0
        TcpExtTCPTSReorder              85                 0.0
        TcpExtTCPFullUndo               1                  0.0
        TcpExtTCPPartialUndo            67                 0.0
        TcpExtTCPDSACKUndo              11                 0.0
        TcpExtTCPLossUndo               0                  0.0
        TcpExtTCPLostRetransmit         184                0.0
        TcpExtTCPRenoFailures           0                  0.0
        TcpExtTCPSackFailures           0                  0.0
        TcpExtTCPLossFailures           0                  0.0
        TcpExtTCPFastRetrans            7084               0.0
        TcpExtTCPSlowStartRetrans       0                  0.0
        TcpExtTCPTimeouts               168                0.0
        TcpExtTCPLossProbes             345                0.0
        TcpExtTCPLossProbeRecovery      82                 0.0
        TcpExtTCPRenoRecoveryFail       0                  0.0
        TcpExtTCPSackRecoveryFail       0                  0.0
        TcpExtTCPRcvCollapsed           0                  0.0
        TcpExtTCPBacklogCoalesce        10938              0.0
        TcpExtTCPDSACKOldSent           300                0.0
        TcpExtTCPDSACKOfoSent           49                 0.0
        TcpExtTCPDSACKRecv              317                0.0
        TcpExtTCPDSACKOfoRecv           2                  0.0
        TcpExtTCPAbortOnData            25                 0.0
        TcpExtTCPAbortOnClose           54                 0.0
        TcpExtTCPAbortOnMemory          0                  0.0
        TcpExtTCPAbortOnTimeout         4                  0.0
        TcpExtTCPAbortOnLinger          0                  0.0
        TcpExtTCPAbortFailed            0                  0.0
        TcpExtTCPMemoryPressures        0                  0.0
        TcpExtTCPMemoryPressuresChrono 0                   0.0
        TcpExtTCPSACKDiscard            0                  0.0
        TcpExtTCPDSACKIgnoredOld        2                  0.0
        TcpExtTCPDSACKIgnoredNoUndo     272                0.0
        TcpExtTCPSpuriousRTOs           0                  0.0
        TcpExtTCPMD5NotFound            0                  0.0
        TcpExtTCPMD5Unexpected          0                  0.0
        TcpExtTCPMD5Failure             0                  0.0
        TcpExtTCPSackShifted            34290              0.0
        TcpExtTCPSackMerged             11301              0.0
        TcpExtTCPSackShiftFallback      40480              0.0
        TcpExtTCPBacklogDrop            0                  0.0
        TcpExtPFMemallocDrop            0                  0.0
```

(continues on next page)

(continued from previous page)

```
    TcpExtTCPMinTTLDrop             0                  0.0
    TcpExtTCPDeferAcceptDrop        0                  0.0
    TcpExtIPReversePathFilter       0                  0.0
    TcpExtTCPTimeWaitOverflow       0                  0.0
    TcpExtTCPReqQFullDoCookies      0                  0.0
    TcpExtTCPReqQFullDrop           0                  0.0
    TcpExtTCPRetransFail            0                  0.0
    TcpExtTCPRcvCoalesce            100585             0.0
    TcpExtTCPOFOQueue               15954              0.0
    TcpExtTCPOFODrop                0                  0.0
    TcpExtTCPOFOMerge               38                 0.0
    TcpExtTCPChallengeACK           0                  0.0
    TcpExtTCPSYNChallenge           0                  0.0
    TcpExtTCPFastOpenActive         0                  0.0
    TcpExtTCPFastOpenActiveFail     0                  0.0
    TcpExtTCPFastOpenPassive        0                  0.0
    TcpExtTCPFastOpenPassiveFail    0                  0.0
    TcpExtTCPFastOpenListenOverflow 0                  0.0
    TcpExtTCPFastOpenCookieReqd     0                  0.0
    TcpExtTCPFastOpenBlackhole      0                  0.0
    TcpExtTCPSpuriousRtxHostQueues  0                  0.0
    TcpExtBusyPollRxPackets         0                  0.0
    TcpExtTCPAutoCorking            73847              0.0
    TcpExtTCPFromZeroWindowAdv      40                 0.0
    TcpExtTCPToZeroWindowAdv        40                 0.0
    TcpExtTCPWantZeroWindowAdv      2870               0.0
    TcpExtTCPSynRetrans             91                 0.0
    TcpExtTCPOrigDataSent           5948573            0.0
    TcpExtTCPHystartTrainDetect     34                 0.0
    TcpExtTCPHystartTrainCwnd       1880               0.0
    TcpExtTCPHystartDelayDetect     3                  0.0
    TcpExtTCPHystartDelayCwnd       261                0.0
    TcpExtTCPACKSkippedSynRecv      0                  0.0
    TcpExtTCPACKSkippedPAWS         9                  0.0
    TcpExtTCPACKSkippedSeq          11                 0.0
    TcpExtTCPACKSkippedFinWait2     0                  0.0
    TcpExtTCPACKSkippedTimeWait     0                  0.0
    TcpExtTCPACKSkippedChallenge    0                  0.0
    TcpExtTCPWinProbe               0                  0.0
    TcpExtTCPKeepAlive              67                 0.0
    TcpExtTCPMTUPFail               0                  0.0
    TcpExtTCPMTUPSuccess            0                  0.0
    TcpExtTCPDelivered              5951000            0.0
    TcpExtTCPDeliveredCE            0                  0.0
    TcpExtTCPAckCompressed          3021               0.0
    TcpExtTCPZeroWindowDrop         0                  0.0
    TcpExtTCPRcvQDrop               0                  0.0
    TcpExtTCPWqueueTooBig           0                  0.0
    TcpExtTCPFastOpenPassiveAltKey  0                  0.0
    TcpExtTcpTimeoutRehash          72                 0.0
    TcpExtTcpDuplicateDataRehash    0                  0.0
    TcpExtTCPDSACKRecvSegs          371                0.0
```

(continues on next page)

(continued from previous page)

```
       TcpExtTCPDSACKIgnoredDubious    0                  0.0
       TcpExtTCPMigrateReqSuccess      0                  0.0
       TcpExtTCPMigrateReqFailure      0                  0.0
       IpExtInNoRoutes                 0                  0.0
       IpExtInTruncatedPkts            0                  0.0
       IpExtInMcastPkts                62                 0.0
       IpExtOutMcastPkts               24                 0.0
       IpExtInBcastPkts                19989              0.0
       IpExtOutBcastPkts               0                  0.0
       IpExtInOctets                   533061309          0.0
       IpExtOutOctets                  5153892360         0.0
       IpExtInMcastOctets              7448               0.0
       IpExtOutMcastOctets             3592               0.0
       IpExtInBcastOctets              2082276            0.0
       IpExtOutBcastOctets             0                  0.0
       IpExtInCsumErrors               0                  0.0
       IpExtInNoECTPkts                1073527            0.0
       IpExtInECT1Pkts                 0                  0.0
       IpExtInECT0Pkts                 0                  0.0
       IpExtInCEPkts                   0                  0.0
       IpExtReasmOverlaps              0                  0.0
       MPTcpExtMPCapableSYNRX          0                  0.0
       MPTcpExtMPCapableSYNTX          2203               0.0
       MPTcpExtMPCapableSYNACKRX       2172               0.0
       MPTcpExtMPCapableACKRX          0                  0.0
       MPTcpExtMPCapableFallbackACK    0                  0.0
       MPTcpExtMPCapableFallbackSYNACK 22                 0.0
       MPTcpExtMPFallbackTokenInit     0                  0.0
       MPTcpExtMPTCPRetrans            0                  0.0
       MPTcpExtMPJoinNoTokenFound      0                  0.0
       MPTcpExtMPJoinSynRx             0                  0.0
       MPTcpExtMPJoinSynAckRx          0                  0.0
       MPTcpExtMPJoinSynAckHMacFailure 0                  0.0
       MPTcpExtMPJoinAckRx             0                  0.0
       MPTcpExtMPJoinAckHMacFailure    0                  0.0
       MPTcpExtDSSNotMatching          0                  0.0
       MPTcpExtInfiniteMapRx           0                  0.0
       MPTcpExtDSSNoMatchTCP           0                  0.0
       MPTcpExtDataCsumErr             0                  0.0
       MPTcpExtOFOQueueTail            0                  0.0
       MPTcpExtOFOQueue                0                  0.0
       MPTcpExtOFOMerge                0                  0.0
       MPTcpExtNoDSSInWindow           0                  0.0
       MPTcpExtDuplicateData           0                  0.0
       MPTcpExtAddAddr                 0                  0.0
       MPTcpExtEchoAdd                 0                  0.0
       MPTcpExtPortAdd                 0                  0.0
       MPTcpExtAddAddrDrop             0                  0.0
       MPTcpExtMPJoinPortSynRx         0                  0.0
       MPTcpExtMPJoinPortSynAckRx      0                  0.0
       MPTcpExtMPJoinPortAckRx         0                  0.0
       MPTcpExtMismatchPortSynRx       0                  0.0
```

(continues on next page)

(continued from previous page)

```
        MPTcpExtMismatchPortAckRx       0                  0.0
        MPTcpExtRmAddr                  0                  0.0
        MPTcpExtRmAddrDrop              0                  0.0
        MPTcpExtRmSubflow               0                  0.0
        MPTcpExtMPPrioTx                0                  0.0
        MPTcpExtMPPrioRx                0                  0.0
        MPTcpExtMPFailTx                0                  0.0
        MPTcpExtMPFailRx                0                  0.0
        MPTcpExtMPFastcloseTx           0                  0.0
        MPTcpExtMPFastcloseRx           0                  0.0
        MPTcpExtMPRstTx                 17                 0.0
        MPTcpExtMPRstRx                 0                  0.0
        MPTcpExtRcvPruned               0                  0.0
        MPTcpExtSubflowStale            0                  0.0
        MPTcpExtSubflowRecover          0                  0.0
```

Among all these variables, the ones named `\*Ext\*` are Linux specific variables that are not defined in IETF MIBs. The others are usually defined in an IETF RFC. The counters maintained by the Linux kernel are defined in include/uapi/linux/snmp.h and net/mptcp/mib.h for the Multipath TCP counters. Each of the counters exposed by nstat correspond to one specific identifier in the Linux kernel. For example, the beginning of the IP part of the counters is defined as follows:

```
enum
{
    IPSTATS_MIB_NUM = 0,
    /* frequently written fields in fast path, kept in same cache line */
    IPSTATS_MIB_INPKTS,                     /* InReceives */
    IPSTATS_MIB_INOCTETS,                   /* InOctets */
    IPSTATS_MIB_INDELIVERS,                 /* InDelivers */
    IPSTATS_MIB_OUTFORWDATAGRAMS,           /* OutForwDatagrams */
    IPSTATS_MIB_OUTPKTS,                    /* OutRequests */
    IPSTATS_MIB_OUTOCTETS,                  /* OutOctets */
    /* other fields */
```

Before looking at the precise meaning of each of the counters managed by nstat, it is interesting to recall the definition of the Case diagrams. This graphical representation of SNMP variables can be really useful to understand the meaning of the Linux networking counters.

2.2 The Case diagrams

The Case diagrams were introduced by Jeffrey Case and Craig Partridge in 1989 in the paper Case diagrams: a first step to diagrammed management information bases. This article describes a simple but powerful graphical representation of the interactions among the different SNMP variables that a networking stack maintains.

A *Case diagram* represents the flow of packets through a stack and the different variables that are updated as the packet progress through the stack. The incoming packets are represented as progressing from the bottom layer of the stack to the upper layer, while the outgoing packets are represented in the other direction. The progression of these packets is represented using a large arrow. An horizontal line that crosses this arrow indicates the point in the stack where the associated SNMP counter is updated. A small that leaves the main packet processing flow indicates a specific treatment for a packet and a counter that is updated. In some cases, an arrow enters the main workflow and updates the associated counter.

The original paper used the IP counters of the MIB-2 to illustrate the *Case diagrams*. This figure is reproduced below in ASCII format to simplify the updates to the document.

```
                   Transport Layer
-----------------------------------------------------
                         /\
                         ||
                         ||
 IpInDelivers +++++++++++++++
                         ||
                         ||
                         |+-----------> IpInUnknownProtos
                         ||
                         ||
 IpInDiscards <----------+|
                         ||
                         |<----------------  IpReasmOKs
                         ||                     /\
                         ||   IpReasmFails  <---+|
                         ||                     ||
                         |+--------------> IpReasmReqds
                         ||
                         ||
 IpForwDatagrams <-------+|
                         ||
                         |+--------------> IpInAddrErrors
                         ||
                         ||
 IpInHdrErrors <---------+|
                         ||
                         ||
                         ||
                      +++++++++++++++++++ IpInReceives
                         ||
-----------------------------------------------------
                   Interface Layer
```

The *Case diagram* above shows how the packets are processed by the IP stack. First, the Interface layer extracts the payload of the received frame and passes it to the IP layer. At this point, the IpInReceives counter is incremented. The processing of the IPv4 packet starts. First, the stack checks for errors inside the IPv4 header. If an error is detected in the IPv4 header, the packet is dropped and IpInHdrErrors is incremented. Then, the destination address is checked. If the address is incorrect, the packet processing stops and IpInAddrErrors is incremented.

If IP forwarding is enabled and the packet is not destined to this host, then the packet is forwarded using the FIB. The IpForwDatagrams counter is incremented.

The next step is to check whether the received packet is a fragment of a larger packet that needs to be reassembled. If the received packet is a fragment, then the IpReasmReqds counter is incremented and the packet passed through the reassembly process. This reassembly can take time since more fragments can be required to recover a complete packet. If the packet reassembly succeeds, then IpReasmOKs is incremented and the processing of the full packet continues. If the reassembly fails, e.g. because a fragment is missing before the timeout expires, then IpReamsFails gets incremented.

A this point, the packets have almost finished to be processed by the IP stack. Most packets will be delivered to the transport layer and increment the IpInDelivers counter except if the IP queue becomes full. In this case, the IpInDiscards counter is incremented. The incoming packet could also be discarded if its *Protocol* field does not match one of the transport layers supported by the stack (i.e. UDP, TCP, DCCP, …). In this case, the `IpInUnknownProtos` counter is incremented.

CHAPTER

THREE

THE MULTIPATH TCP COUNTERS

Linux version 5.18 maintains 46 counters for Multipath TCP. These counters correspond to different parts of the protocol and can be organized in four groups. The first group gathers the counters that are incremented when TCP packets containing the `MP_CAPABLE` option are processed. The second group gathers the counters that are incremented when processing packets with the `MP_JOIN` option. The third group gathers the counters that are modified when packets with the `ADD_ADDR`, `RM_ADDR` or `MP_PRIO` option are processed. The fourth group gathers the remaining counters of the Multipath TCP stack.

Two versions of Multipath TCP have been specified within the IETF. Version 0 was initially defined in **RFC 6824**. The off-tree but well maintained set of patches distributed by https://www.multipath-tcp.org implemented this version of Multipath TCP. Based on the experience gathered with this implementation and also Apple's implementation, Multipath TCP evolved and the IETF published version 1 in **RFC 8684**. The Multipath TCP counters correspond to this version of Multipath TCP.

3.1 The MPCapable counters

This group gathers the following counters: `MPTcpExtMPCapableSYNRX`, `MPTcpExtMPCapableSYNTX`, `MPTcpExtMPCapableSYNACKRX`, `MPTcpExtMPCapableACKRX`, `MPTcpExtMPCapableFallbackACK`, `MPTcpExtMPCapableFallbackSYNACK` and `MPTcpExtMPFallbackTokenInit`. They relate to the establishment of the initial Multipath TCP subflow which is described in the *The Multipath TCP handshake* section.

The `MPTcpExtMPCapableSYNTX` counter is similar to the `TcpActiveOpens` counter maintained by TCP. It counts the number of Multipath TCP connections that this host has tried to establish. Its value will usually be much smaller than `TcpActiveOpens`. When a Multipath connection is initiated using the `connect` system call, both `MPTcpExtMPCapableSYNTX` and `TcpActiveOpens` are incremented. Although the name of the counter is `MPTcpExtMPCapableSYNTX`, it is only incremented once per Multipath TCP connection if the `SYN` packet needs to be retransmitted.

The `MPTcpExtMPCapableSYNACKRX` counter is incremented every time a Multipath TCP connection is confirmed by the reception of a `SYN+ACK` with the `MP_CAPABLE` option to a `SYN` packet that it sent earlier. The value of this counter should be lower than `MPTcpExtMPCapableSYNTX` since only a subset of the connections initiated by a host will typically reach a Multipath TCP compliant server. If a client receives a `SYN+ACK` without the `MP_CAPABLE` option in response to a `SYN` sent with the `MP_CAPABLE` option, then the `MPTcpExtMPCapableFallbackSYNACK` counter is incremented. This counter tracks the Multipath TCP connections that were forced to fall back to regular TCP during the three-way handshake of the initial subflow.

On the other hand, the `MPTcpExtMPCapableSYNRX` counter tracks the number of Multipath TCP connections that were accepted by the host. Its value will usually be much smaller than `TcpPassiveOpens` which tracks all accepted TCP connections. When a Multipath connection is accepted, both `MPTcpExtMPCapableSYNRX` and `TcpPassiveOpens` are incremented. As for `MPTcpExtMPCapableSYNTX`, the `MPTcpExtMPCapableSYNRX` counter is only incremented once per connection and not each time a packet is received. Upon reception of a `SYN` with the `MP_CAPABLE` option, a Multipath TCP server returns a `SYN+ACK` with the `MP_CAPABLE` option. The `MPTcpExtMPCapableACKRX` counter is incremented upon reception of the third ACK containing the MP_CAPABLE option. If this option is not present in this ACK, then the MPTcpExtMPCapableFallbackACK gets incremented. If this counter increases, it probably indicates some interference with a middlebox that injects acknowledgments during the three-way handshake.

Listing 3.1: The MPCapable counters (active opens)

```
                        ||
                        ||
MPTcpExtMPCapableSYNTX+++++++++++++++
                        ||
                        ||
                        |+-----------> MPTcpExtMPCapableSYNACKRX
                        ||
                        ||
                        |+-----------> MPTcpExtMPCapableFallbackSYNACK
                        ||
                        ||
```

Listing 3.2: The MPCapable counters (passive opens)

```
                        ||
                        ||
MPTcpExtMPCapableSYNRX+++++++++++++++
                        ||
                        ||
                        |+-----------> MPTcpExtMPCapableACKRX
                        ||
                        ||
                        |+-----------> MPTcpExtMPCapableFallbackACK
                        ||
                        ||
```

3.2 The Join counters

There are thirteen counters in this group. They are incremented when a host processes SYN packets corresponding to additional subflows.

The first counter, MPTcpExtMPJoinSynRx is incremented every time a SYN packet with the MP_JOIN option is received. Upon reception of a such packet, the host first verifies that it knows the token of the Multipath TCP connection. If so, the processing continues and the host returns a SYN+ACK packet with the MP_JOIN option, its random number and a HMAC. Otherwise, the MPTcpExtMPJoinNoTokenFound counter is incremented. The host then waits for the third ACK which contains the MP_JOIN option and the HMAC computed by the remote host. It then checks the validity of the received HMAC. If the HMAC is invalid, then the MPTcpExtMPJoinAckHMacFailure counter is incremented.

The MPTcpExtMPJoinSynRx counter will increase on Multipath TCP hosts that accept subflows, typically servers. The value of the MPTcpExtMPJoinACKRX counter should be close to the previous one. If the two other counters, MPTcpExtMPJoinNoTokenFound or MPTcpExtMPJoinAckHMacFailure increase, then the system administrator should probably investigate as these are indication of possible attacks.

Listing 3.3: The Join counters when accepting subflows

```
                        ||
                        ||
```

(continues on next page)

(continued from previous page)

```
MPTcpExtMPJoinSynRX +++++++++++++++
                        ||
                        ||
                        |+-----------> MPTcpExtMPJoinNoTokenFound
                        ||
                        ||
MPTcpExtMPJoinACKRX +++++++++++++++
                        ||
                        |+-----------> MPTcpExtMPJoinAckHMacFailure
                        ||
                        ||
```

Unfortunately, there is no counter that tracks the creation of new subflows by a host. The TCP stack counts these new subflows as active opens, but there is no specific Multipath TCP counter. However, the `MPTcpExtMPJoinSynAckRX` counter tracks the reception of `SYN+ACK` packets containing the `MP_JOIN` option. This is thus an indirect way to track the creation of new subflows. Upon reception of such a packet, in response to a previously sent `SYN` packet with the `MP_JOIN` option, a host checks the validity of the received HMAC. If the HMAC is invalid, the `MPTcpExtMPJoinSynAckHMacFailure` is incremented. This counter should rarely increase. If it increases, then the problem should be investigated by collecting packet traces.

Listing 3.4: The Join counters when initiating subflows

```
                        ||
                        ||
MPTcpExtMPJoinSynAckRX +++++++++++++++
                        ||
                        ||
                        |+-----------> MPTcpExtMPJoinSynAckHMacFailure
                        ||
                        ||
```

A Multipath TCP host will usually accept additional subflows on the address and ports where the initial subflow was accepted. The following counters track the arrival of packets destined to different port numbers:

- `MPTcpExtMPJoinPortSynRx`
- `MPTcpExtMPJoinPortSynAckRx` `MPTcpExtMPJoinPortAckRx`

The last two counters, `MPTcpExtMismatchPortSynRx` and `MPTcpExtMismatchPortAckRx` are a bit different. They are incremented when a `SYN` or `ACK` sent to a different port number are received.

The `MP_JOIN` option contains a `B` that indicates whether the new subflow should be considered as a backup subflow or a regular one. This information is used by the path manager, but no counter tracks the value of the backup bit in the `MP_JOIN` option. Once a subflow has been established, its backup status can be changed using the `MP_PRIO` option. The `MPTcpExtMPPrioTx` counter is incremented every time such an option is sent. The `MPTcpExtMPPrioRx` counter is incremented by each received `MP_PRIO` option.

3.3 The address advertisement counters

There are six counters in this group. The advertisement of addresses by Multipath TCP is described in ref:*Address management <mmtpbook:mptcp-addr-management>*.

When a host receives a packet with a valid `ADD_ADDR` option with its `Echo` bit set to zero, the `MPTcpExtAddAddr` counter is incremented. If this option includes an optional port number, the `MPTcpExtPortAdd` counter is also incremented. In addition to these two counters, the `MPTcpExtAddAddrDrop` tracks the address advertisements that were received by the host, but not processed by the path manager, e.g. because no user space path manager was active.

Multipath TCP does not track the advertisements of addresses by sending the `ADD_ADDR` option. However, it tracks the reception of packets containing the `ADD_ADDR` option with the `Echo` bit set to one with the `MPTcpExtEchoAdd` counter. These packets are echoed by the remote host.

Similarly, the `MPTcpExtRmAddr` counter tracks the number of received `RM_ADDR` options. These options typically indicate a change in the addresses owned by a remote peer. Mobile hosts are likely to send these options when they move from one type of network to another. The `MPTcpExtRmAddrDrop` is incremented when the path manager cannot process an incoming `RM_ADDR` option.

When a host receives a `RM_ADDR` option from a remote peer, its path manager should remove the subflows associated with this address. The `MPTcpExtRmSubflow` counter tracks the number of subflows that have been destroyed by a path manager.

3.4 The connection termination counters

There are seven counters in this group. They track the abnormal termination of a Multipath TCP connection. A normal Multipath TCP connection should end with the exchange of `DATA_FIN` in both directions. However, are scenarios are possible. First, one of the hosts may wish to quickly terminate the Multipath TCP connection without having to maintain state. Multipath TCP uses the `FAST_CLOSE` option in this case. The `MPTcpExtMPFastcloseTx` and `MPTcpExtMPFastcloseRx` counters track the transmission and the reception of such options.

Multipath TCP was designed to prevent as much as possible interference from middleboxes, but there are some types of interferences that force Multipath TCP to fallback to regular TCP. In this case, the host that first noticed the interference (e.g. problem during the handshake, DSS checksum problem, …) sends a packet with the `MP_FAIL` option. This forces the Multipath TCP connection to fall back to a regular TCP connection. The `MPTcpExtMPFailTx` and `MPTcpExtMPFailRx` counters track the transmission and the reception of the `MP_FAIL` option. During some types of fall backs, a host may also send an infinite DSS mapping. The `MPTcpExtInfiniteMapRx` counter tracks the reception of such infinite DSS mappings.

An increase of these counters would indicate some type of middlebox interference which should be investigated since it could prevent a complete utilization of Multipath TCP.

Like TCP, Multipath TCP uses TCP `RST` to terminate subflows. Multipath TCP also defines the `MP_TCPRST` option which can contain an option reason code and flags indicating some information about the reason for the transmission of the `RST`. The `MPTcpExtMPRstTx` and `MPTcpExtMPRstRx` counters track the transmission and the reception of such `RST` packets.

3.5 The other counters

The remaining eleven counters are mainly related to processing of data.

If the DSS checksum is enabled, the `MPTcpExtDataCsumErr` is incremented every time a check of the DSS checksum fails. This should be a rare event that likely indicates the presence of middleboxes. It should be correlated with the `MPTcpExtMPFailTx` and `MPTcpExtMPFailRx` counters discussed in the previous section.

Three counters track the DSS option of the incoming packets : `MPTcpExtDSSNotMatching`, `MPTcpExtDSSNoMatchTCP` and `MPTcpExtNoDSSInWindow`. The first counter is incremented when a mapping is received for data that has already been mapped and the new mapping is not the same as the existing one. The second counter is incremented when the TCP sequence numbers found in the mapping do not match with the current TCP sequence numbers. The third counter is incremented upon reception of a packet that indicates a DSS option that is outside the current window. These three counters should rarely increase.

The last counter that tracks data at the Multipath TCP connection level is `MPTcpExtDuplicateData`. It counts the number of received packets whose data has been ignored because it had already been received earlier. Such duplicated data can occur with Multipath TCP when data sent over a subflow is retransmitted over another subflow. It would be interesting to follow the evolution of this counter on a server that interacts with mobile devices.

Multipath TCP tracks losses on the subflows that compose a Multipath TCP connection. If one subflow accumulates losses, it may be marked as stale and the packet scheduler will stop using it to transmit data until the losses have been recovered. The `MPTcpExtSubflowStale` counter is incremented every time a subflow is marked as being stale. The `MPTcpExtSubflowRecover` counter tracks the transitions from stale to active.

Multipath TCP uses an out-of-order queue to reorder the data received over the different subflows. The `MPTcpExtOFOQueueTail` and `MPTcpExtOFOQueue` counters track the insertion of data at the tail and in the out-of-order queue. The `MPTcpExtOFOMerge` is incremented when data present in the out-or-order queue can be merged.

Finally, the `MPTcpExtRcvPruned` tracks the number of packets that were dropped because the memory available for Multipath TCP was full. If this counter increases, you should probably check the memory configuration of your host.

CHAPTER

FOUR

THE TRANSMISSION CONTROL PROTOCOL (TCP)

TCP is a connection-oriented transport protocol. This means that a TCP connection must be established before communicating hosts can exchange data. A connection is a logical relation between the two communication hosts. Each hosts maintains some state about the connection and uses it to manage the connection.

TCP uses the three-way handshake as shown in Fig. 4.1. To initiate a connection, the client sends a TCP segment with the `SYN` flag set. Such a segment is usually called a `SYN` segment. It contains a random sequence number (*x* in Fig. 4.1). If the server accepts the connection, it replies with a `SYN+ACK` segment whose `SYN` and `ACK` flags are set. The acknowledgment number of this segment is set to `x+1` to confirm the reception of the `SYN` segment sent by the client. The server selects a random sequence number (*y* in Fig. 4.1). Finally, the client replies with an *ACK* segment that acknowledges the reception of the `SYN+ACK` segment. TCP was designed to be extensible. The TCP header contains a TCP Header Length (THL) field that indicates the total length of the TCP header in four-bytes words. For the normal header, this field is set to 5, which corresponds to the 20 bytes long TCP header. Larger values of the THL field indicate that the segment contains one or more TCP options. TCP options are encoded as a Type-Length-Value field. The first byte specifies the Type, the second byte indicates the length of the entire TCP option in bytes. The utilization of TCP options is usually negotiated during the three-way-exchange. The client adds a TCP option in the `SYN` segment. If the server does not recognize the option, it simply ignores it. If the server wants to utilize the extension for the connection, it simply adds the corresponding option in the `SYN+ACK` segment. This is illustrated in Fig. 4.2 with the Selective Acknowledgments extension [1] as an example. A TCP connection is identified by using four fields that are included inside each TCP packet:

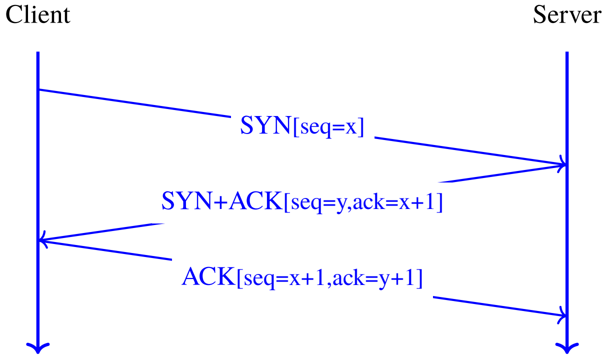


Fig. 4.1: Establishing a TCP connection using the three-way handshake

- the client IP address
- the server IP address
- the client-selected port
- the server port

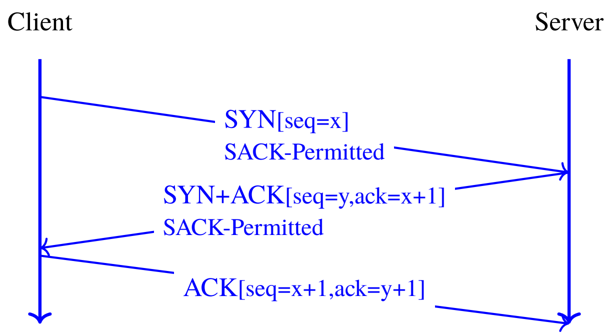


Fig. 4.2: Negotiating the utilization of Selective Acknowledgments during the three-way handshake

All TCP packets that belong to a connection contain these four fields in the IP and TCP header. When a host receives a packet, i

- `snd.una`, the oldest unacknowledged sequence number
- `snd.nxt`, the next sequence number of be sent
- `rcv.win`, the latest window advertised by the remote host

A TCP sender also stores the data that has been sent but has not yet been acknowledged. It also measures the round-trip-time and its variability to set the retransmission timer and maintains several variables that are related to the congestion control scheme.

A TCP receiver also maintains state variables. These include `rcv.next`, the next expected sequence number. Data received in sequence can be delivered to the application while out-of-sequence data must be queued.

Finally, TCP implementations store the state of the connection according to the TCP state machine [2].

TCP implementations include lots of optimizations that are outside the scope of this brief introduction. Let us know briefly describe how TCP sends data reliably. Consider a TCP connection established between a client and a server. Fig. 4.3 shows a simple data transfer between these two hosts. The sequence number of the first segment starts at `1234`, the current value of `snd.nxt`. For TCP, each transmitted byte consumes one sequence number. Thus, after having sent the first segment, the client's `snd.nxt` is set to `1238`. The server receives the data in sequence and immediately acknowledges it. A TCP receiver always sets the acknowledgment number of the segments that it sends with the next expected sequence number, i.e. `rcv.nxt`. In practice, TCP implementations use the Nagle algorithm [3] and thus usually try to send full segments. They use the Maximum Segment Size (MSS) option during the handshake and PathMTU discovery the determine the largest segment which can be safely sent over a connection. Furthermore, TCP implementations usually delay acknowledgments and only acknowledge every second segment when these are received in sequence. This is illustrated in Fig. 4.4. TCP uses a single segment type and each segment contains both a sequence number and an acknowledgment number. The sequence number is mainly useful when a segment contains data. A receiver only processes the acknowledgment number if the `ACK` flag is set. In practice, TCP uses cumulative

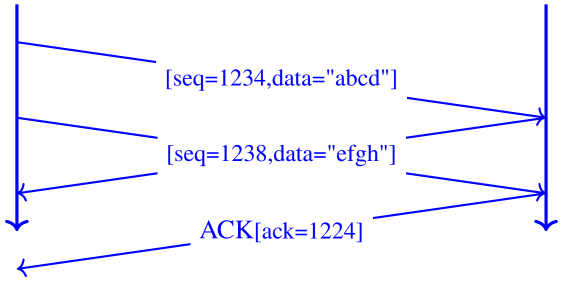


Fig. 4.3: TCP Reliable data transfer

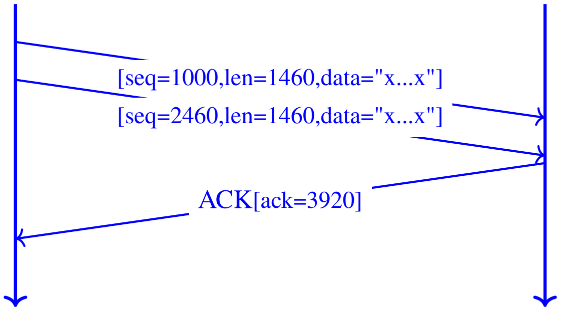

[seq=1000,len=1460,data="x...x"]
[seq=2460,len=1460,data="x...x"]
ACK[ack=3920]

Fig. 4.4: TCP Reliable data transfer with delayed acknowledgments.

acknowledgments and all the segments sent on a TCP connection have their `ACK` flag set. The only exception is the `SYN` segment sent by the client to initiate a connection. TCP uses different techniques to retransmit corrupted or lost data. The TCP header contains a 16 bits checksum that is computed over the entire TCP segment and a part of the IP header. The value of this checksum is computed by the sender and checked by the receiver to detect transmission errors. TCP copes with these errors by retransmitting data. The simplest technique is to rely on a retransmission timer. TCP continuously measure the round-trip-time, i.e. the delay between the transmission of a segment and the reception of the corresponding acknowledgment. It then sets a per-connection retransmission timer based on its estimations of the mean rtt and its variance [4]. This is illustrated in Fig. 4.6 where the arrow terminated with red cross corresponds to a lost segment. Upon expiration of the retransmission timer, the client retransmits the unacknowledged segment. For performance reasons, TCP implementations try to avoid relying on the retransmission timer to retransmit the lost segments. Modern TCP implementations use selective acknowledgments which can be negotiated during the handshake. This is illustrated in Fig. 4.7. A selective acknowledgment reports blocks of sequence number that have been received correctly by the receiver. Upon reception of the `SACK` option, the sender knows that sequence numbers `1234-1237` have not been received while sequence numbers `1238-1250` have been correctly received. When the client and the sender have exchanged all the required data, they can terminate the connection. TCP supports two different methods to terminate a connection. The reliable manner is that each host closes its direction of data transfer by sending a segment with the `FIN` flag set. The sequence number of this segment marks the end of the data transfer and the recipient of the segment acknowledges it once it has delivered all the data up to the sequence number of the `FIN` segment to its application. The release of a TCP connection is illustrated in Fig. 4.8. To reduce the size of the figure, we have set the `FIN` flag in segments that contains data. The server considers the connection to be closed upon reception of the `FIN+ACK` segment. It discards the state that it maintained for this now closed TCP connection. The client also considers the connection to be closed when it sends the `FIN+ACK` segment since all data has been acknowledged. However, it does not immediately discard the state for this connection because it needs to be able to retransmit the `FIN+ACK` segment in case it did not reach the server.

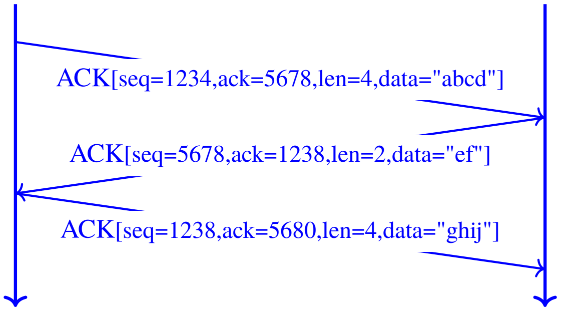

ACK[seq=1234,ack=5678,len=4,data="abcd"]
ACK[seq=5678,ack=1238,len=2,data="ef"]
ACK[seq=1238,ack=5680,len=4,data="ghij"]

Fig. 4.5: TCP piggybacking.

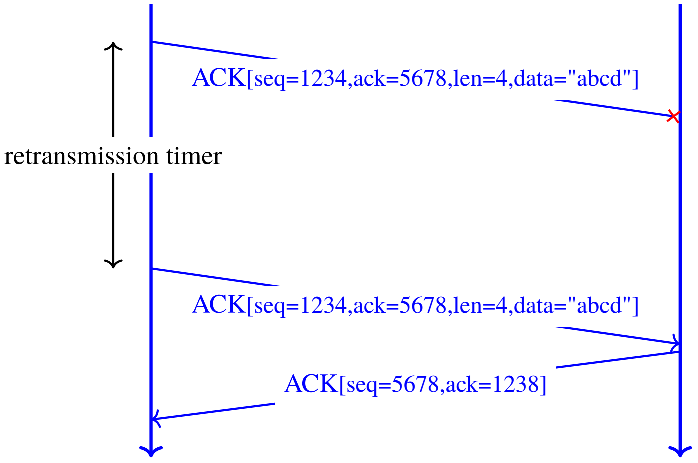

ACK[seq=1234,ack=5678,len=4,data="abcd"]
retransmission timer
ACK[seq=1234,ack=5678,len=4,data="abcd"]
ACK[seq=5678,ack=1238]

Fig. 4.6: TCP protects data by a retransmission timer

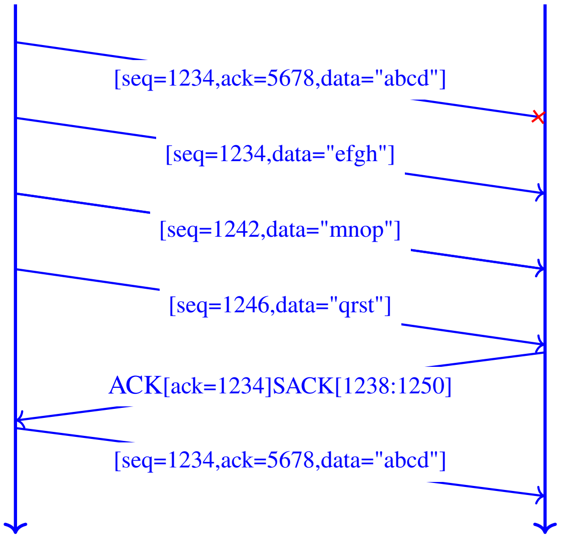

[seq=1234,ack=5678,data="abcd"]
[seq=1234,data="efgh"]
[seq=1242,data="mnop"]
[seq=1246,data="qrst"]
ACK[ack=1234]SACK[1238:1250]
[seq=1234,ack=5678,data="abcd"]

Fig. 4.7: TCP leverages selective acknowledgments to retransmit lost data

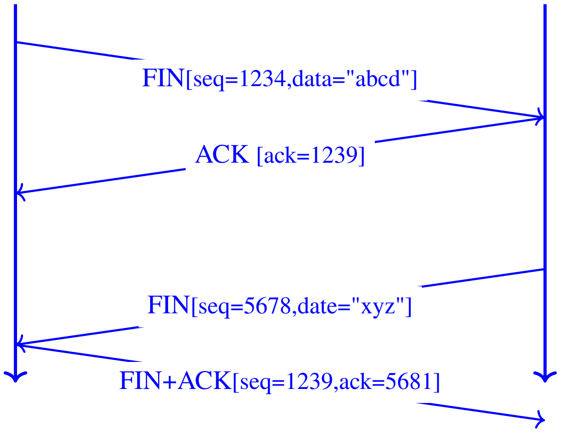


Fig. 4.8: Closing a TCP connection using the `FIN` flag

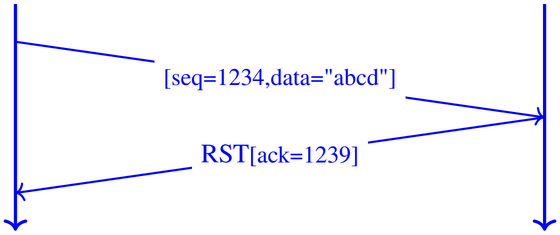


Fig. 4.9: Closing a TCP connection using a RST segment

CHAPTER

FIVE

MULTIPATH TCP

Multipath TCP [5] is an extension to the TCP protocol [2] that was described in chapter *The Transmission Control Protocol (TCP)*. We start with an overview of Multipath TCP. Then we explain how a Multipath TCP connection can be established. Then we analyze how data is exchanged over different paths and explain the multipath congestion control schemes. Finally, we explain how Multipath TCP connections can be terminated.

5.1 A brief overview of Multipath TCP

The main design objective for Multipath TCP [6] was to enable hosts to exchange the packets that belong to a single TCP connection over different network paths. Several definitions are possible for a network path. Considering a TCP connection between a client and a server, a network path can be defined as the succession of the links and routers that create a path between the client and the server. For example, in Fig. 5.1, there are many paths between the client host C and the server S, e.g. $C \rightarrow R1 \rightarrow R2 \rightarrow R4 \rightarrow S$ and $C \rightarrow R1 \rightarrow R3 \rightarrow R4 \rightarrow S$, but also $C \rightarrow R1 \rightarrow R3 \rightarrow R5 \rightarrow R4 \rightarrow S$ or even $C \rightarrow R1 \rightarrow R2 \rightarrow R4 \rightarrow R3 \rightarrow R5 \rightarrow R4 \rightarrow S$. During the first discussions on Multipath TCP within the IETF, there was a debate on the types of paths that Multipath TCP could use in IP networks. Although networks provide a wide range of paths between a source and a destination, it is not necessarily simple to use all these paths in a pure IP network. Looking a Fig. 5.1 and assuming that all links have the same IGP weight, packets sent by C will follow one of the two shortest paths, i.e. $C \rightarrow R1 \rightarrow R2 \rightarrow R4 \rightarrow S$ or $C \rightarrow R1 \rightarrow R3 \rightarrow R4 \rightarrow S$. Since routers usually use hash-based load-balancing [7] to distribute packets over equal cost paths, all the packets from a given connection will follow either the first or the second shortest path. In most networks, the path followed by a TCP connection will only change if there are link or router failures on this particular path.

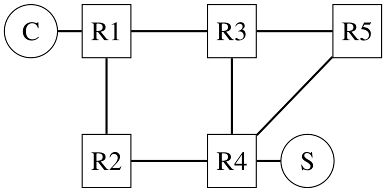


Fig. 5.1: A simple network providing multiple paths between C and S

When Multipath TCP was designed, the IETF did not want to design techniques to enable the transport layer to specify the paths that packets should follow. They opted for a very conservative definition of the paths that Multipath TCP can use [8]. Multipath TCP assumes that the endpoints of a TCP connection are identified by their IP addresses. If two hosts want to exchange packets over different paths, then at least one of them must have two or more IP addresses. This covers two very important use cases:

- mobile devices like the smartphones that have a cellular and a Wi-Fi network interface each identified by its own IP address
- dual-stack hosts that have both an IPv4 and an IPv6 address

In this document, we will often use smartphones to illustrate Multipath TCP client hosts. This corresponds to a widely deployed use case that simplifies many of the examples, but is not the only possible deployment.

Note: Using non-equal cost paths with Multipath TCP

When Multipath TCP was designed, there was no standardized solution that enabled a host to control the path followed by its packets inside a network. This is slowly changing. First, the IETF has adopted the Segment Routing architecture [9]. This architecture is a modern version of source routing which can be used in MPLS and IPv6 networks. In particular, using the IPv6 Segment Routing Header [10], a host can decide the path that its packets will follow inside the network. This opens new possibilities for Multipath TCP. Some of these possibilities are explored by the Path Aware Networking Research Group of the Internet Research Task Force.

A second important design question for the Multipath TCP designers was how use two or more paths for a single connection ? As an example, let us consider a smartphone that interacts with a server. This smartphone has two different IP addresses: one over its Wi-Fi interface and one over its cellular interface. A naive way to use these two networks would be to operate as shown Fig. 5.2. The smartphone would initiate a TCP connection over its Wi-Fi interface as shown in blue in Fig. 5.2. This handshake creates a connection and thus some shared state between the smartphone and the server. Given this state, could the smartphone simply sent the next date over the cellular interface (shown in red in Fig. 5.2) ? Unfortunately, this utilization of the two paths between the smartphone and the server poses different problems. First, the server must be able to accept the packet sent by the smartphone over the cellular interface and associate it with the connection created over the Wi-Fi interface. However, the packets sent over the cellular interface use a different source address than those sent over the Wi-Fi interface. When the server receives such a packet, how can it be associated with an existing connection ? If the server blindingly accept this packet from another address than the one used during the handshake, then there are obvious security risks. By sending a single packet, an attacker could inject data inside an existing connection. Furthermore, he could cause a denial of service attack by sending a spoofed packet in an existing connection that requests the server to send a large volume of data to the spoofed address. Furthermore, a middlebox such as a firewall on the cellular path between the smartphone and the server could block the packet because it does not belong to a TCP connection created on the cellular path.

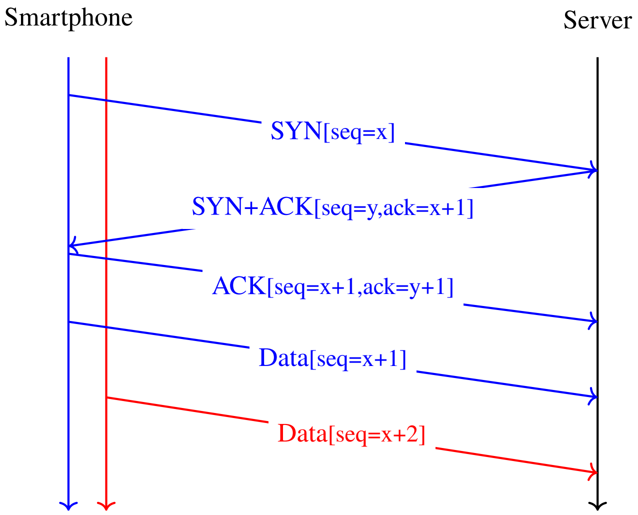


Fig. 5.2: A naive approach to create a Multipath TCP connection

To cope with this problem, the Multipath TCP designers opted for an architecture where a Multipath TCP connection combines several TCP connections that are called subflows over the different paths. In the above example, the smartphone would first create a connection over the Wi-Fi interface. It would later initiate a TCP connection over its cellular interface and use Multipath TCP to link it to the connection created over the Wi-Fi interface.

A Multipath TCP connection starts with a three-way handshake like a regular TCP connection. As with all TCP extensions, the client uses an option in the SYN to indicate its willingness to use the multipath extensions. The server confirms that it agrees to use this extension by sending the same option in the SYN+ACK. This is illustrated in Fig. 5.3 where the client sends a SYN with the MPC option to negotiate a Multipath TCP connection with a server. If the server replies with the same option, the handshake succeeds and creates the first subflow belonging to this Multipath TCP connection. The client and the server can send data over this connection as over any TCP connection. To use a second path, the client (or the server), must initiate another TCP handshake over the new path. The SYN sent over this second path uses the MPJ option to indicate that this is an additional subflow that must be linked to an existing Multipath TCP connection. This is illustrated in Fig. 5.3. These two three-way handshakes create two TCP connections called subflows in the Multipath TCP terminology. It is useful to analyze how these two connections are identified on the server. A host identifies a TCP connection using four identifiers that are present in all the packets of this connection:

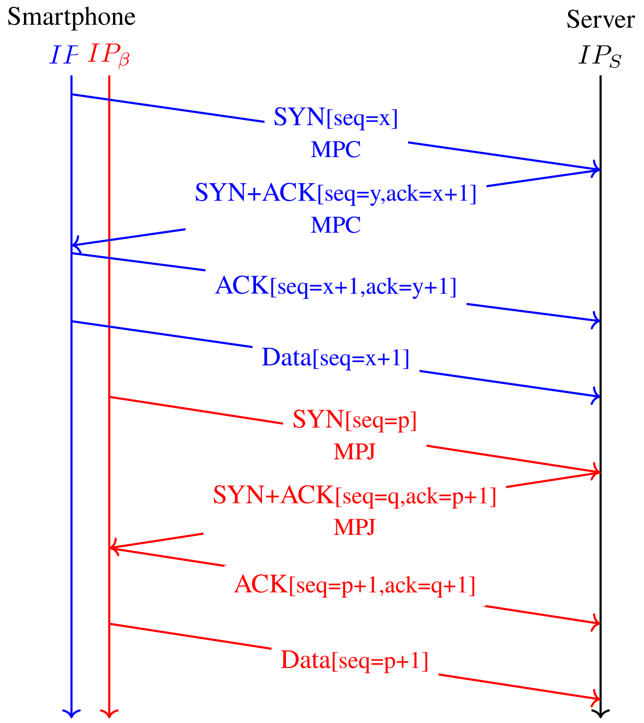


Fig. 5.3: A Multipath TCP connection with two subflows

- the local IP address
- the remote IP address
- the local port
- the remote port

Assume that the client uses IP address IP_α on its Wi-Fi interface and IP_β on its cellular interface and that p is the port used by the server. If the client used port p_1 to create the initial subflow, then the identifier of this subflow on the server is $< IP_S, IP_\alpha, p, p_1 >$. Similarly, the second subflow is identified by the $< IP_S, IP_\beta, p, p_2 >$ tuple on the server. Note that these two connection identifiers differ by at least one IP address as specified in [8].

A server usually manages a large number of simultaneous connections. Furthermore, a client may establish several connections with the same server. To associate a new subflow with an existing Multipath TCP connection, a server must be able to link an incoming SYN with the corresponding Multipath TCP connection. For this, the client must include an identifier of the associated Multipath TCP connection in its MPJ option. This identifier must unambiguously identify the corresponding Multipath TCP connection on the server.

A first possible identifier is the four tuple that identifies the initial subflow, i.e. $< IP_S, IP_\alpha, p, p_1 >$. If the server received this identifier in the MPJ option, it could link the new subflow to the previous one. Unfortunately, this solution does not work in today's Internet. The main concern comes from the middleboxes such as NATs and transparent proxies. To illustrate the problem, consider a simple NAT, such as the one used on most home Wi-Fi access points. Fig. 5.4 illustrates a TCP handshake in such an environment. The smartphone uses a private IP address, IP_P and the NAT uses a public address IP_N. If we assume that the NAT only changes the client's IP address, then the connection is identified by the $< IP_P, IP_S, p, p_1 >$ tuple on the smartphone and $< IP_S, IP_N, p, p_1 >$ on the server. Note some NATs also change the client port. If the smartphone places its local connection identifier inside an MPJ option, the server might not be able to recognize the corresponding connection in the SYN packets that it received.

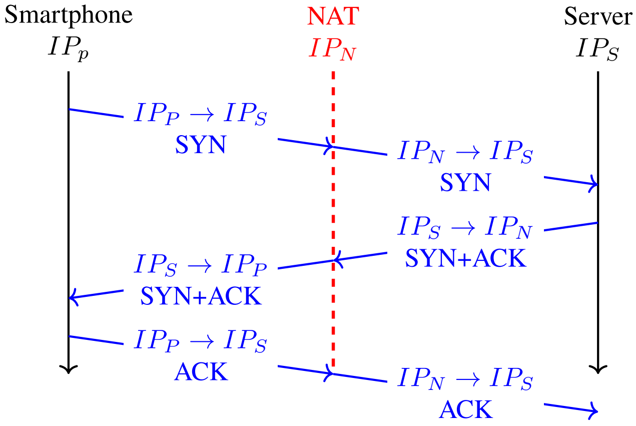


Fig. 5.4: Network Address Translation interferes with TCP

To cope with this problem, Multipath TCP uses a local identifier, called *token* in the Multipath TCP specification, to identify each Multipath TCP connection. The client assigns its token when it initiates a new Multipath TCP connection. A server assigns its token when it accepts a new Multipath TCP connection. These two tokens are chosen independently by the client and the server. For security reasons, these tokens should be random. The MPJ option contains the token assigned by the remote host. This is illustrated in Fig. 5.5. The server assigns token *456* to the Multipath TCP connection created as the first subflow. It informs the smartphone by sending this token in its MPC option in the SYN+ACK. When the client creates the second subflow, it includes its token in the MPJ option of its SYN.

Note: Multipath TCP in datacenters

The Multipath TCP architecture [8] assumes that at least one of the communicating hosts use different IP addresses to identify the different paths used by a Multipath TCP connection. In practice, this architectural requirement is not always enforced by Multipath TCP implementations. A Multipath TCP implementation can combine different subflows into one Multipath TCP connection provided that each subflow is identified by a different four-tuple. Two subflows between two communicating hosts can differ in their client-selected ports. This solution has been chosen when Multipath TCP was proposed to mitigate congestion in datacenter networks [11].

Several designs exist for datacenter networks, but the fat-tree architecture shown in Fig. 5.6 is a very popular one.

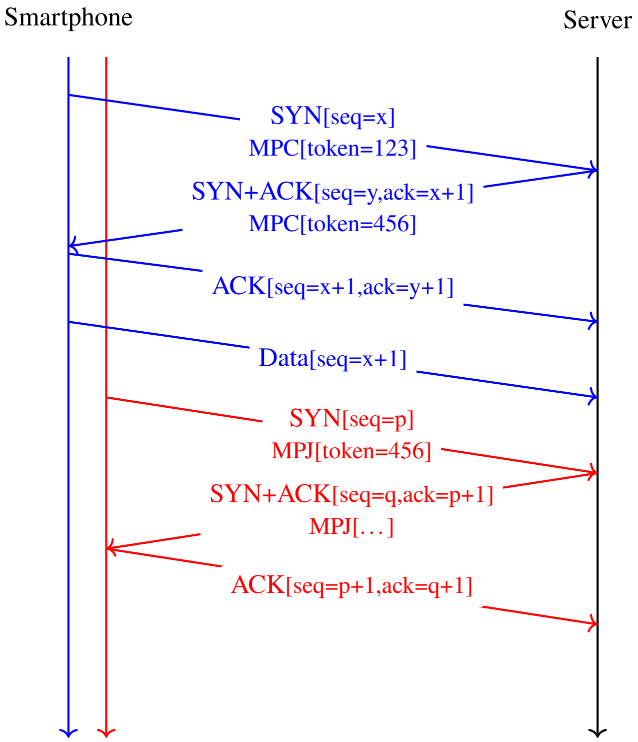


Fig. 5.5: The tokens exchanged during the handshake allow to associate subsequent subflows to existing Multipath TCP connections

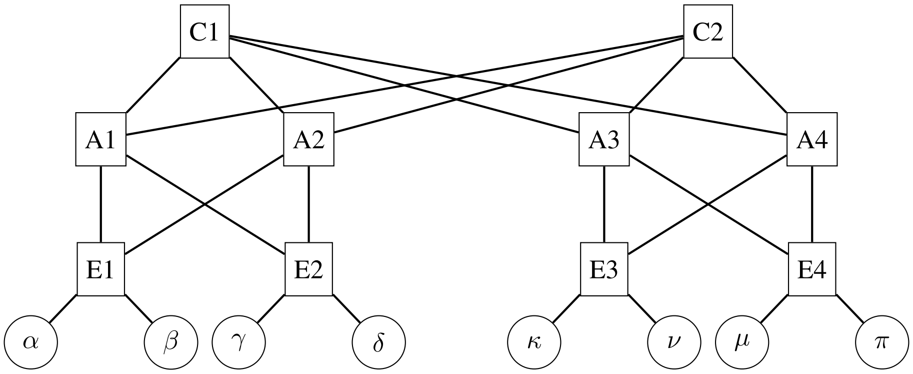


Fig. 5.6: A simple datacenter network

This network topology exposes a large number of equal cost paths between the servers that are shown using circles in Fig. 5.6. For example, consider the paths between the α and π hosts. The paths start at $E1$. This router can reach $E4$ and π via $A1$ or $A2$. Each of these two aggregation routers can reach π via one of the two core routers. These two routers can then balance the flows via both $A3$ and $A4$. There are $2^4 = 16$ different paths between α and π in this very small network. If each of these routers balance the incoming packets using a hash function [7] that takes as input their source and destination addresses and ports, then the subflows of a Multipath TCP connection that use different client problems will be spread evenly across the network topology. Raiciu et al. provide simulations and measurements showing the benefits of using Multipath TCP in datacenters [11].

Once a Multipath TCP connection and the additional subflows have been established, we can use them to exchange data. An important point to remember is that a Multipath TCP connection provides a bidirectional bytestream service like a regular TCP connection. This service does not change even if Multipath TCP uses different subflows to carry the data between the sender and the receiver. As an example, consider a sender that sends `ABCD` one byte at a time over a Multipath TCP connection composed of two subflows. A naive approach to send these bytes would be to simply placed them in different TCP segments. This is illustrated in Fig. 5.7 where we assume that the two TCP subflows have already been established. In this example, the Smartphone slowly sends data in sequence. The server receives the data in sequence over the two subflows and the server could simply deliver the data as soon as it arrives over each subflow. This is illustrated with the `DATA.ind(...)` primitives that represent the delivery of the data to the server application. However, consider now that the first packet sent on the red subflow is lost and is retransmitted together with the fourth byte as shown in Fig. 5.8. In Fig. 5.8, it is clear that the server cannot simply deliver the data as soon as it receives it to its application. If the server behaves this way, it will deliver `ACBD` to its application instead of the `ABCD` bytestream send by the smartphone. To cope with the reordering of the data sent over the different subflows, Multipath TCP includes bytestream-level data sequence numbers that enable it to preserve the ordering of the data sent over the bytestream. This is illustrated in Fig. 5.9 with the bytestream-level sequence number shown as `bseq`. We will detail later how this sequence number is exactly transported by Multipath TCP. Thanks to the bytestream sequence number, the server can reorder the data received over the different subflows and preserve the ordering in the bytestream.

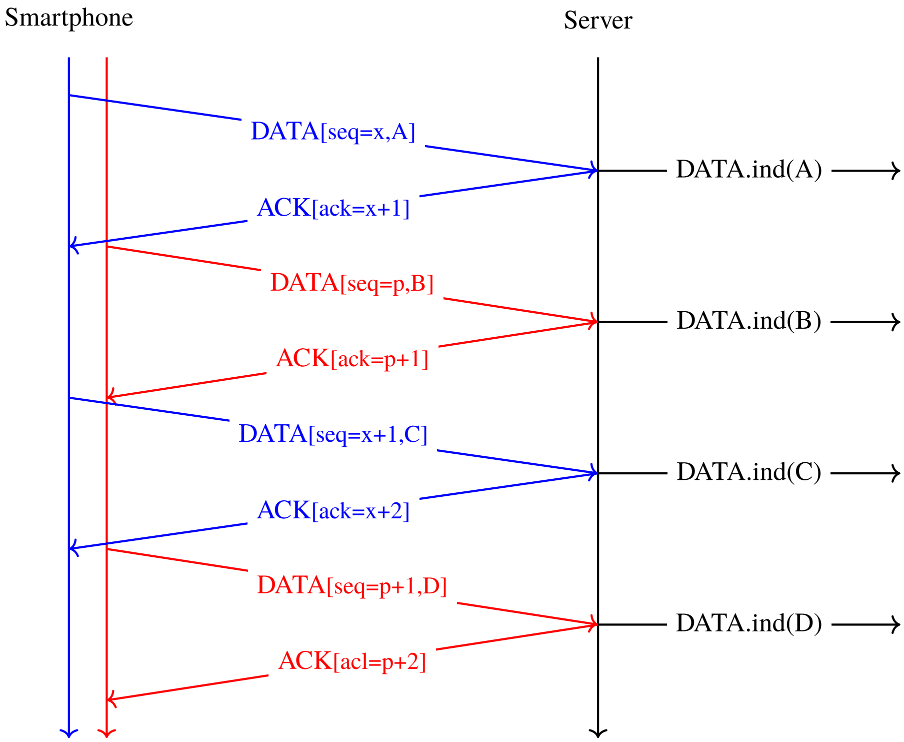


Fig. 5.7: A naive approach to send data over a Multipath TCP connection

5.2 Creating a Multipath TCP connection

Before delving into the details of how a Multipath TCP connection is created, let use first analyze the main requirements of this establishment and how they can be met without considering all the protocol details. During the three-way handshake, TCP hosts agree to establishment a connection, select the initial sequence number in each direction and negotiate the utilization of TCP extensions. In addition to these objectives, the handshake used by Multipath TCP also allows the communicating hosts to:

- agree to use the Multipath TCP extension

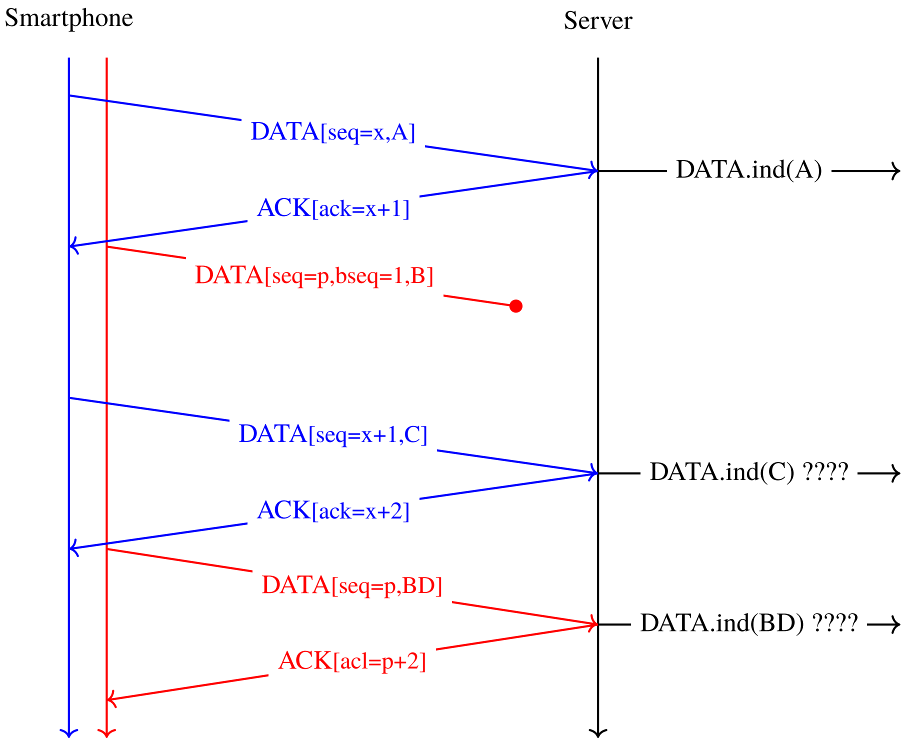


Fig. 5.8: A naive approach to send data over a Multipath TCP connection

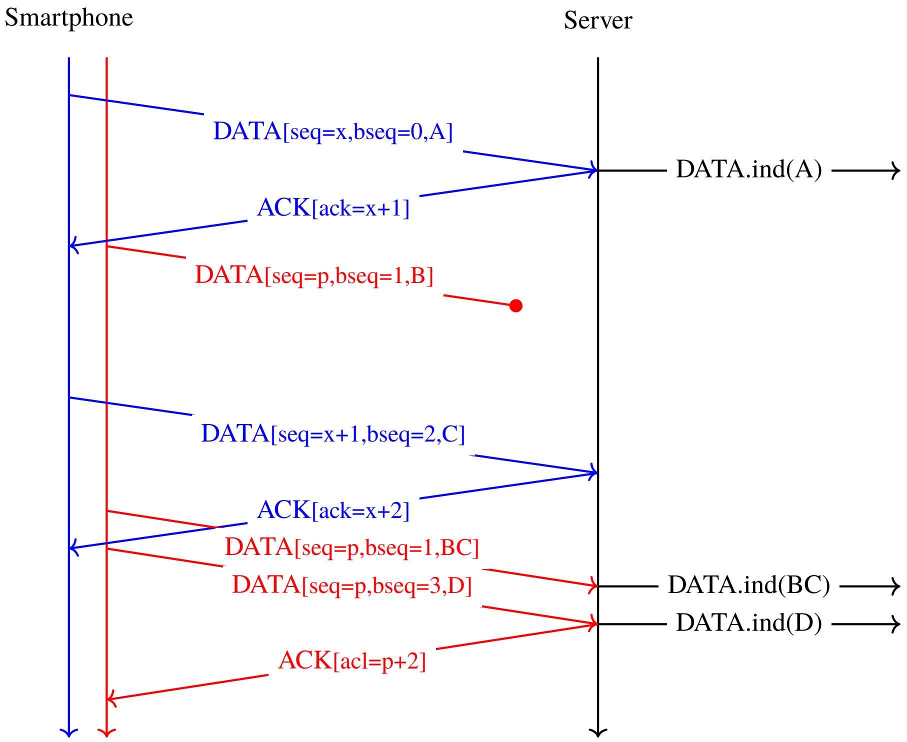


Fig. 5.9: A naive approach to send data over a Multipath TCP connection

- exchange the tokens used to identify the connection
- agree on initial bytestream sequence numbers

To meet the first objective, the client simply needs to send a Multipath TCP option (MPO) in its SYN. If the server supports Multipath TCP, it will respond with a SYNC+AC that carries this option.

To meet the second objective, the simplest solution is reserve some space, e.g. 64 bits, in the MPO option to encode the token chosen by the host that sends the SYN or SYN+ACK. With this approach, each host can autonomously select the token that it uses to identify each Multipath TCP connection. To meet the third objective, the simplest solution is also to place the initial sequence number in the MPO option. Fig. 5.10 illustrates a handshake using the MPO option. The Multipath TCP working group was worried about the risk of attacks with this approach. When the smartphone creates an additional subflow, it includes the token allocated by the server inside the MP_JOIN option. This token serves two different purposes. First, it identifies the relevant Multipath TCP connection on the server. Second, it also "authenticates" that the SYN also originates from this client. Authenticating the client is a key concern from a security viewpoint. The main risk is that an on-path attacker who has observed the token in the MP_JOIN option can reuse it to create additional subflows from any other source. To cope with this problem, Multipath TCP relies on a shared secret that the client and the server exchange during the initial handshake. The client proposes one halve of the secret and the server the other halve. This is illustrated in Fig. 5.11. The client proposes its part of the shared secret in the SYN ($Client_{secret}$). The server replies with its part of the secret in the SYN+ACK. Using these two components of the shared secret, the client and the server must be able to authenticate the additional subflows without revealing the shared secret to an attacker who is able to capture packets on the path of the additional subflow. Multipath TCP requires each host to perform a HMAC [12] of a random number to confirm their knowledge of the shared secret. This is illustrated in the second part of Fig. 5.11. To create the additional subflow, the client send a SYN with the MP_JOIN option containing the $Server_{token}$ and a random nonce, $Client_{random}$. The server confirms the establishment of the subflow by sending a SYN+ACK containing the HMAC computed using the $Client_{random}$ and the $Client_{secret}$

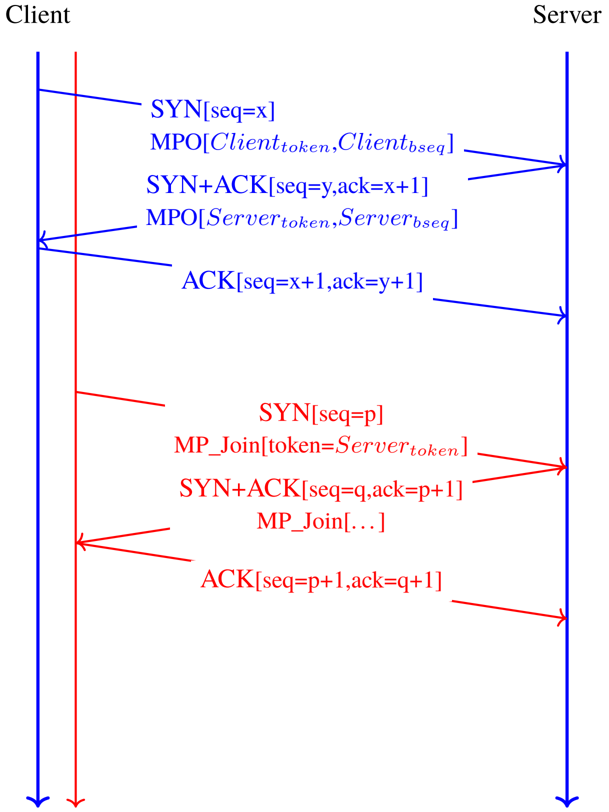
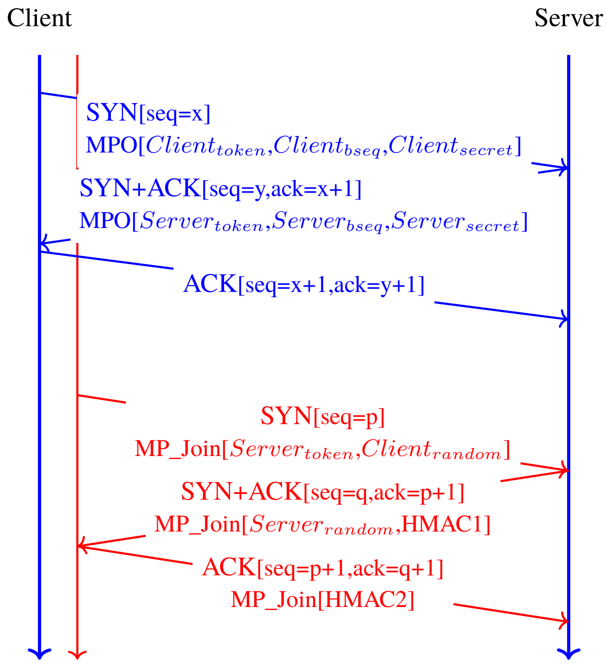


Fig. 5.10: Opening a Multipath TCP connection with a MPO option


Client
Server
SYN[seq=x]
MPO[$Client_{token}$,$Client_{bseq}$,$Client_{secret}$]
SYN+ACK[seq=y,ack=x+1]
MPO[$Server_{token}$,$Server_{bseq}$,$Server_{secret}$]
ACK[seq=x+1,ack=y+1]
SYN[seq=p]
MP_Join[$Server_{token}$,$Client_{random}$]
SYN+ACK[seq=q,ack=p+1]
MP_Join[$Server_{random}$,HMAC1]
ACK[seq=p+1,ack=q+1]
MP_Join[HMAC2]

HMAC1=HMAC(key=$Server_{secret}$||$Client_{Secret}$,
msg=$Server_{random}||Client_{random}$)

HMAC2=HMAC(key=$Client_{secret}$||$Server_{Secret}$,
msg=$Client_{random}||Server_{random}$)

Fig. 5.11: Creating a Multipath TCP connection with a MPO option

and $Server_{secret}$ input. Thanks to this HMAC computation, the server can reveal that it knows $Client_{secret}$ and $Server_{secret}$ without explicitly sending them. The server also places a random number, $Server_{random}$ in the MP_JOIN option of the SYN+ACK. The client computes a HMAC and returns it in the third ACK. With these two HMACs, the client and the server can authenticate the establishment of the additional subflow without revealing the shared secret.

Note: The security of Multipath TCP depends on the security of the initial handshake

The ability of correctly authenticate the addition of new subflows to a Multipath TCP connection depends on the secrecy of the $Client_{secret}$ and $Server_{secret}$ exchanged in the SYN and SYN+ACK of the initial handshake. An on-path attacker which is able to capture this initial handshake has all the information required to attach a new subflow to this Multipath TCP connection at any time. Multipath does not include the strong cryptographic techniques (besides HMAC) that would have been required to completely secure the establishment the protocol and the establishment of additional subflows in particular. This threat was considered acceptable for Multipath TCP [13] because an attacker who can capture the packets of a single path TCP connection can also inject data inside this connection. To be fully secure Multipath TCP would need to rely on cryptographic techniques that are similar to those used in Transport Layer Security [14].

The solution described above meets the requirements of the Internet Engineering Task Force. From a security viewpoint, the $Client_{secret}$, $Server_{secret}$ and the random nonces should be as large as possible to prevent attacks where their values are simply guessed. Unfortunately, since Multipath TCP uses TCP options to exchange all this information, we need to ensure that it fits inside the extended header of a TCP SYN. The TCP specification [2] reserves up to 40 bytes to place the TCP options in a SYN. Today's TCP stacks already consume 4 bytes for the MSS option [2], 3 for the Window Scale option [15], 2 for SACK Permitted [1] and 10 for the timestamp option [15]. This leaves only 20 bytes to encode a Multipath TCP option that must contain an initial sequence number, a token and a secret. Multipath TCP solves this problem by deriving these three values from a single field encoded in a TCP option. Let us now analyze the Multipath TCP handshake in more details.

5.2.1 The Multipath TCP handshake

A Multipath TCP connection starts with a three-way handshake like a regular TCP connection. To indicate that it wishes to use Multipath TCP, the client adds the MP_CAPABLE option to the SYN segment. In the SYN segment, this option only contains some flags and occupies 4 bytes. The server replies with a SYN+ACK segment than contains an MP_CAPABLE option including a server generated 64 bits random key that will be used to authenticate connections over different paths. The client concludes the handshake by sending an MP_CAPABLE option in the ACK segment containing the random keys chosen by the client and the server.

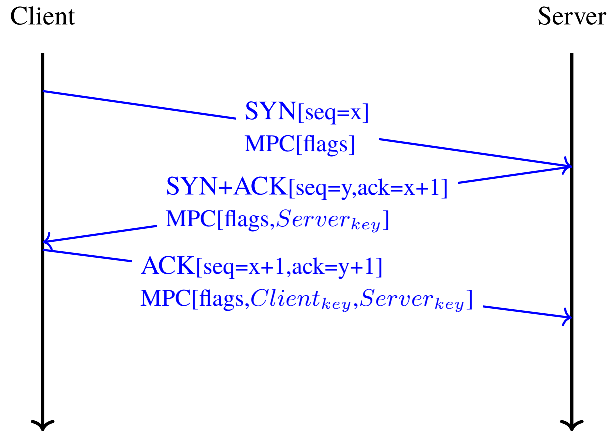


Fig. 5.12: Negotiating the utilization of Multipath TCP during the three-way handshake

Note: Multipath TCP version 0

The first version of Multipath TCP used a slightly different handshake [6]. The MP_CAPABLE option sent by the client contains the 64 bits key chosen by the client. The SYN+ACK segment contains an MP_CAPABLE option with 64 bits key chosen by the server. The client echoes the client and server keys in the third ACK of the handshake.

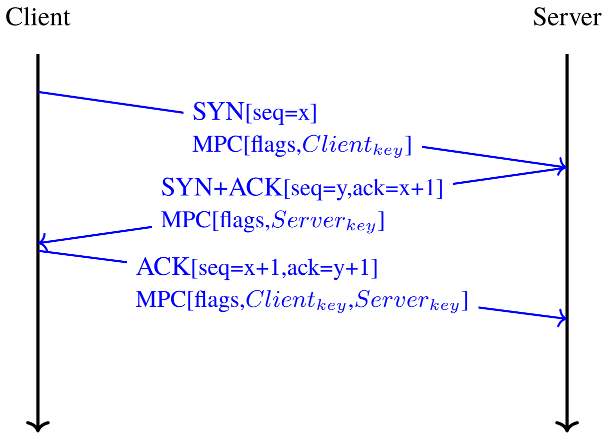

Client
Server
SYN[seq=x]
MPC[flags,$Client_{key}$]
SYN+ACK[seq=y,ack=x+1]
MPC[flags,$Server_{key}$]
ACK[seq=x+1,ack=y+1]
MPC[flags,$Client_{key}$,$Server_{key}$]

Fig. 5.13: Negotiating the utilization of Multipath TCP version 0

The 64 bits random keys chosen by the client and the server play three different roles in Multipath TCP. Their first role is to identify the Multipath TCP connection to which an additional connection must be attached. Since a Multipath TCP connection can combine several TCP connections, Multipath TCP cannot use the IP addresses and port numbers to identify a TCP connection. Multipath TCP uses a specific identifier that is called a token. For technical reasons, this token is derived from the 64 bits key as the most significant 32 bits of the SHA-256 [16] hash of the key. The second role of the 64 bits keys is to authenticate the establishment of additional connections as we will see shortly. Finally, the keys are also used to compute random initial sequence numbers.

The main benefit of Multipath TCP is that a Multipath TCP connection can combine different TCP connections that potentially use different paths. Starting from now on, we will consider a client with two network interfaces and a server with one network interface. This could for example correspond to a client application running on a smartphone that interacts with a server. We explore more complex scenarios later.

We can know explain how a Multipath TCP connection can combine different TCP connections. According to the Multipath TCP specification, these connections are called subflows [5]. We also adopt this terminology in this document. Fig. 5.14 shows a Multipath TCP that combines two subflows. To establish the Multipath TCP connection, the client initiates the initial subflow by using the MP_CAPABLE option during the three-way handshake. At the end of the initial handshake, the client and the server have exchanged their keys. Based on their keys, they have both computed the token that the remote host uses to identify the Multipath TCP connection.

To attach a second subflow to this Multipath TCP connection, the client needs to create it. For this, it starts a three-way handshake with the server by sending a SYN segment containing the MP_JOIN option. This option indicates that the client uses Multipath TCP and wishes to attach this new connection to an existing Multipath TCP connection. The MP_JOIN option contains two important fields:

- the token that the server uses to identify the Multipath TCP connection
- a random nonce

The client has derived the token from the key announced by the server in the MP_CAPABLE option of the SYN+ACK segment on the initial subflow. Thanks to this token, the server knows to which Multipath TCP connection the new subflow needs to be attached.

The server uses the random nonce sent by the client and its own random nonce to prove its knowledge of the keys exchanged during the initial handshake. The server computes $HMAC(Key = (Server_{key}||Client_{key}), Msg = (nonce_{Server}||nonce_{Client}))$, where || denotes the concatenation operation. It then returns the high order 64 bits of this HMAC in the MP_JOIN option of the SYN+ACK segment together with its 32 bits nonce. The client computes $HMAC(Key = (Client_{key}||Server_{key}), Msg = (nonce_{Client}||nonce_{Server}))$ and sends the 160 bits HMAC in the ACK segment.

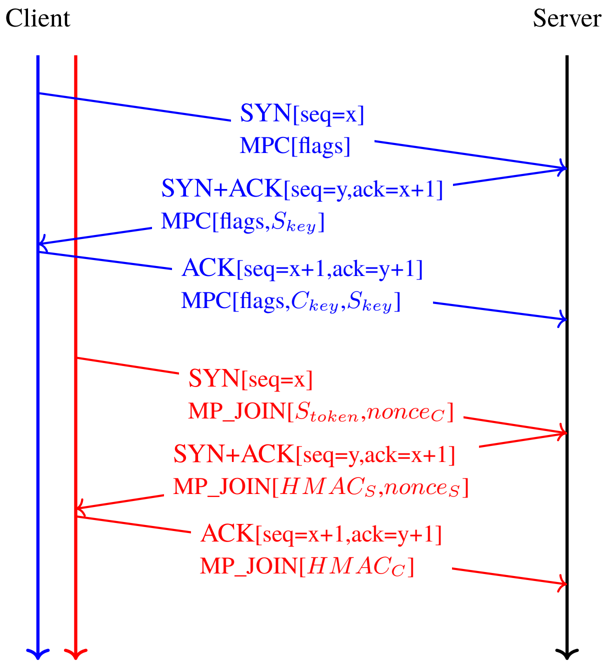


Fig. 5.14: A client creates a second subflow by creating a TCP connection with the MP_JOIN option

Note: Generating Multipath TCP keys

From a security viewpoint, the keys that Multipath TCP hosts exchange in the MP_CAPABLE option should be completely random to prevent them from being guessed by attackers. However, since the token is derived from the key, it cannot be completely random. A host will typically generate a random key and hash it into a token to verify that it does not correspond to an existing connection. On clients, with a few tens of connections, this is not a concern, but on servers, the delay to generate random keys increases with the number of established Multipath TCP connections [17]. This does not prevent servers from supporting large numbers of Multipath TCP connections [18].

A Multipath TCP connection combines a number of subflows which can change during the connection lifetime. It starts with an initial subflow, but this subflow may terminate before the connection. A Multipath TCP connection is a pair of states that are maintained on the client and the server.

The above figure shows how a client adds a subflow to an existing Multipath TCP connection. This is the most common way of adding subflows to a connection. According to the specification, a server could also add subflows to a Multipath TCP connection. For this, the server needs to be able to determine the client addresses. This is the role of the address subflow management parts of Multipath TCP.

5.3 Address and subflow management

Each Internet host has one address per network interface. A smartphone with active Wi-Fi and cellular interfaces has two network addresses. With the advent of IPv6, a large fraction of the hosts are dual-stack and have both an IPv4 and an IPv6 address for each network interface. Multipath TCP specifies options that allow a host to advertise all its addresses to the other host. Given the limited size of the TCP header, these options cannot be exchanged during the handshake. They are typically attached to packets that carry acknowledgments.

Each host maintains a list of its active addresses and associates a numeric identifier to each address. To advertise an address, the host simple adds the Multipath TCP `ADD_ADDR` option to one outgoing packet. This option contains four main fields:

- the IPv4 or IPv6 address of the host
- the numeric identifier of the address
- an optional port number
- a truncated HMAC to authenticate the address advertisement

The IP address is the main information contained in the `ADD_ADDR` option. The identifier allows the host to advertise the list of all its addresses one option at a time. The port number allows to indicate that the hosts listens to another port number than the one used for the subflow where the option is sent. This can be useful if a client wishes to accept subflows initiated by the server or if a server uses another port to listen for additional subflows. The HMAC is the 64 bits truncation of $HMAC(Key = (Server_{key}||Client_{key}), Msg = (Addressidentifier||IPaddress||port))$ when the server advertises an address and $HMAC(Key = (Client_{key}||Server_{key}), Msg = (Addressidentifier||IPaddress||port))$ for an address advertised by the client. The HMAC allows to prevent attacks where an attacker sends spoofed packets containing an `ADD_ADDR` option.

In addition to these four fields, the `ADD_ADDR` option contains an `Echo` bit. The `ADD_ADDR` option is usually sent inside a TCP acknowledgment. A host can easily send an acknowledgment even if it did not recently receive data. Unfortunately, TCP acknowledgments are, by design, unreliable. As TCP uses cumulative acknowledgments, the loss of an acknowledgment is compensated by the next acknowledgment. This is true for the acknowledgment number, but not for the options that were contained in the loss packet. The first version of Multipath TCP did not try to deal with the loss of `ADD_ADDR` options. The current version relies on the `Echo`. A host advertises an address by sending its `ADD_ADDR` option with the `Echo` bit set to `0`. To confirm the reception of this address, the peer simply replies with an acknowledgment containing the same option but with its `Echo` bit set to one. A host that sent an `ADD_ADDR` option needs to retransmit it if it does not receive it back. This is illustrated in Fig. 5.15. Thanks to the `ADD_ADDR` option, a host can advertise all its addresses at the beginning of a Multipath TCP connection. Since the option can be sent at any time, a mobile host that learns a new address, e.g. a smartphone attached to a new Wi-Fi network, can advertise it immediately. This makes Multipath TCP agile on mobile hosts. A host may also stop being able to use an IP address. This occurs when a mobile hosts goes away from a wireless network. In this case, the host should inform its peer about the loss of the corresponding address. This is the role of the `REMOVE_ADDR` option that contains the numeric identifier of the removed address. In contrast with the `ADD_ADDR` option, the `REMOVE_ADDR` option is not authenticated using a truncated HMAC. The protocol specification suggests that when a host receives a `REMOVE_ADDR` option, it should first check whether it is currently used by an active subflow. If no, the address can be removed. If yes, it should send a TCP Keepalive on this subflow to verify whether the address still works. If it does not receive a response to its keepalive, the address can be removed and the associated subflow is reset. Otherwise, the `REMOVE_ADDR` option is ignored.

Multipath TCP hosts use the `ADD_ADDR` and `REMOVE_ADDR` options to maintain the list of addresses used by their peer. However, this is not the only source of information that Multipath TCP uses. A Multipath TCP hosts also learns the source addresses of the established subflows. The first addresses are those used for the initial subflow. The client remembers the server's address as address `0` on this Multipath TCP connection. The server does the same with the client address. When the client creates a new subflow, it places the numeric identifier of the source address of this subflow in the `MP_JOIN` option. This enables the server to learn additional addresses and their associated numeric identifiers. This is illustrated in Fig. 5.16. The server first learns that the client is reachable via the address used for

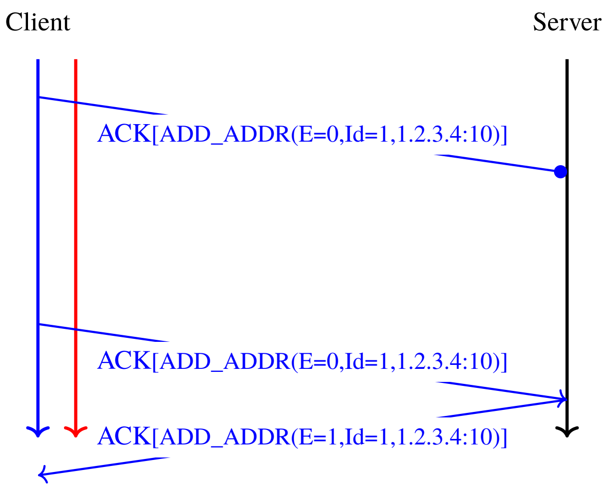

Client
Server
ACK[ADD_ADDR(E=0,Id=1,1.2.3.4:10)]
ACK[ADD_ADDR(E=0,Id=1,1.2.3.4:10)]
ACK[ADD_ADDR(E=1,Id=1,1.2.3.4:10)]

Fig. 5.15: Thanks to the Echo bit, a Multipath TCP host can retransmit lost ADD_ADDR options.

the initial subflow (IP_A). The identifier of this address is 0. Then, the server learns that the client is also reachable through IP address IP_B. Thanks to the identifier contained in the `MP_JOIN` option, the server also learns the identifier (2) of this address. Then, the server learns the third address (IP_{C}) using the `ADD_ADDR` option.

Note: Is the `ADD_ADDR` option required on all Multipath TCP hosts ?

The previous section has explained how Multipath TCP hosts learn the addresses of their peers by using the `ADD_ADDR` and `REMOVE_ADDR` options. These options are important for a server that has multiple addresses (e.g. an IPv4 and an IPv6 address) and wants to advertise them to its clients. On the other hand, servers rarely create subflows and thus they do not really need to learn the client addresses. In fact, Apple's implementation of Multipath TCP on the iPhones does not use the `ADD_ADDR` option. iPhones simply create subflows over the cellular and Wi-Fi interfaces as when needed and the server relies on the `MP_JOIN` option to validate these subflows. It is interesting to note that the `REMOVE_ADDR` option remains useful even if the `ADD_ADDR` option is not used. Consider a smartphone that has created an initial subflow over its Wi-Fi interface and a second subflow over the cellular one. If the smartphone looses its Wi-Fi interface, it can send a `REMOVE_ADDR` option over the subflow that uses the cellular interface to inform the server that it cannot be reached anymore through its Wi-Fi interface.

5.4 Data transfer

Thanks to the `MP_CAPABLE` and `MP_JOIN` option, Multipath TCP hosts can associate one of more subflows to a Multipath TCP connection. Each host can send and receive data on any of the established subflows. As these subflows follow different paths, packets experience different delays. To preserve the in-order bytestream, the receiver must be able to reorder the data received over the different subflows.

A simple approach to perform this reordering would be to rely on the TCP sequence number that is included in the TCP header. This approach is illustrated in Fig. 5.17. The client creates two subflows and uses the same initial sequence numbers on the different subflows. The server also selects the same initial sequence numbers. The client then sends three bytes: `A` over the initial subflow, `B` over the second subflow and `C` over the initial one. Each byte has its own sequence number and the receiver can reorder them. However, note that sequence number `x+2` is not sent over the initial subflow. Furthermore, sequence numbers `x+1` and `x+3` are not sent over the second subflow. Unfortunately, this simple approach suffers from several problems. First, it assumes that the client and the server use the same initial sequence numbers. On the client side, this might be feasible, but on the server side, this would prohibit the utilization of techniques such as SYN cookies that are important to protect from denial of service attacks. Another concern is

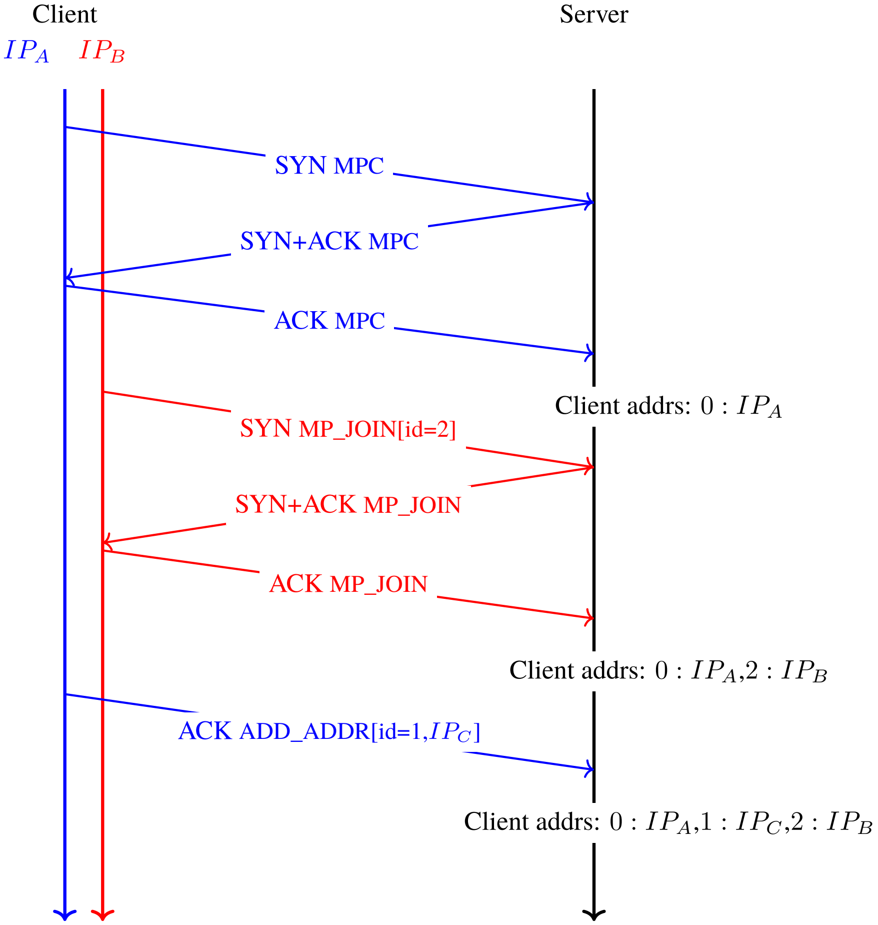


Fig. 5.16: A Multipath TCP hosts remembers the addresses used by its peer

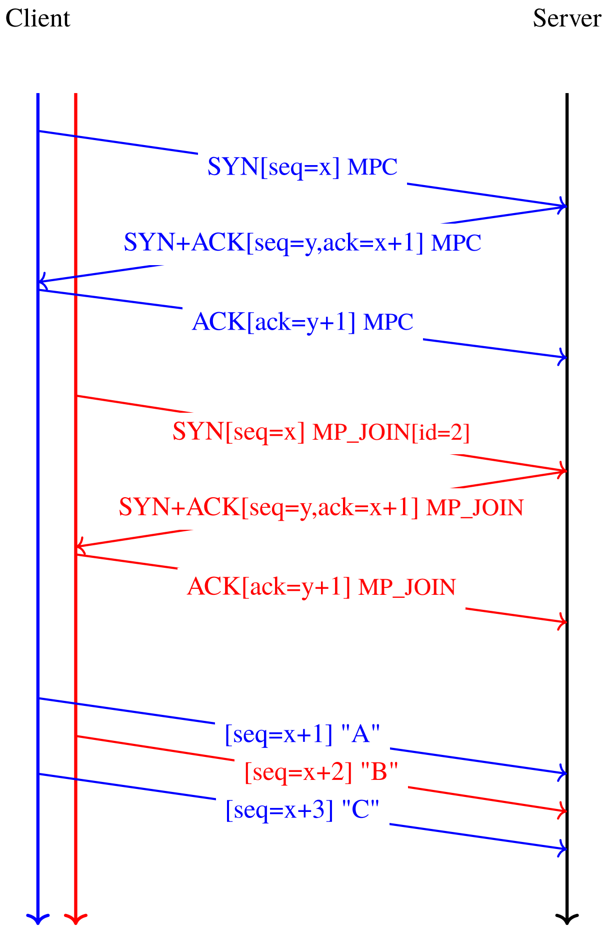


Fig. 5.17: A naive approach to exchange data over different subflows

that there will be gaps in the sequence numbers that are used over each path. These gaps might cause problems with middleboxes such as firewalls. The same problem applies for the acknowledgments. Although TCP supports selective acknowledgments **RFC 2018**, these were not designed to support a large number of gaps.

Multipath TCP solves these problems by using a second level of sequence numbers that are encoded inside TCP options. Conceptually, Multipath TCP associates a data sequence number to the first byte of the payload of each TCP packet. Each Multipath TCP packet carries two different sequence numbers. The first is the sequence number that is included in the TCP header and is called the subflow sequence number. This sequence number plays the same role as in a regular TCP connection. It enables the receiver to reorder the received packets on a given subflow and detect losses. The data sequence number corresponds to the bytestream. It indicates the position of the first byte of the payload of the TCP packet in the bytestream. This data sequence number is used by the receiver to reorder the data received over different subflows and detect losses at this level. Multipath TCP also uses acknowledgments to confirm the reception of data. At the subflow level these are regular TCP acknowledgments (or selective acknowledgments if this extension is active). At the Multipath TCP connection level, the receiver always returns a data acknowledgment that contains the next expected in-sequence data sequence number. This is illustrated in Fig. 5.18.

The client sends the first byte of the bytestream over the initial subflow. This byte is sent in a TCP packet whose sequence number is `x+1`. It carries a Multipath TCP option that contains the data sequence number, i.e. `0` since this is the first byte of the bytestream. The server returns an acknowledgment that indicates that the `x+2` is the next expected sequence number over the initial subflow. This TCP ACK also contains a Multipath TCP option that indicates that `1` is the next expected data sequence number. The sends the second byte over the second subflow. For this, it sends a packet whose sequence number is set to `w+1`, i.e. the first sequence number over this subflow. This packet contains a Multipath TCP option that indicates that this is the second byte (data sequence set to `1`) of the bytestream. The server confirms the reception of this packet with an acknowledgment. Fig. 5.19 shows a slightly different example where the first data packet sent by the client is lost. When the server receives the second byte of the bytestream on the second subflow, it acknowledges it at the subflow level (`ack=w+2`) but not at the connection level since the previous byte of the bytestream is missing. The server stores the received byte in the reordering buffer associated with the connection. When the server receives the second packet sent over the initial subflow, it stores it in the buffer associated with the initial subflow. Since it has neither received the byte that has sequence number `x+1` on the initial subflow, it cannot update its acknowledgment number. It could send a selective acknowledgment if these were enabled on the connection. The retransmission of the first data packet sent over the initial subflow fills the buffer associated to this subflow. The server can thus update the subflow level acknowledgment number (`ack=x+2`). The data received in order can now be passed to the connection-level buffer. The data at this level is also in-sequence and the server returns a data acknowledgment indicating that the next data sequence number it expects is `3`. The three bytes `ABC` are delivered in sequence to the server application. The example of Fig. 5.19 showed how Multipath TCP copes with packet losses. These are frequent events on a TCP connection. A Multipath TCP only needs to cope with the loss of an entire subflow. Consider the same example as above, but the initial subflow was established over a Wi-Fi interface that stops shortly after the reception of the acknowledgment for the second data packet. The client detects the problem and sends a `REMOVE_ADDR` over the second subflow. It also retransmits the first packet that had not been acknowledged, but this time over the second subflow. Conceptually, a Multipath TCP implementation can be viewed as composed of a set of queues. On the sender side, the bytestream is pushed in a queue that keeps the data until it has been acknowledged at the connection level. A packet scheduler extracts blocks of data from this queue and places them with the associated date sequence numbers in the per-subflow queues that represent the sending buffers associated to each subflow. TCP uses these per-subflow queues to send the data and perform the retransmission when required. On the receiver side, there is one queue associated with each subflow. This queue corresponds to the TCP receive buffer. TCP uses this queue to reorder the received data based on their TCP sequence numbers, but does not deal with the data sequence numbers that are contained in TCP options. Once data is in-order in a subflow receive buffer, it goes in the connection-level reorder queue that uses the data sequence numbers contained in TCP options to recover the bytestream. Multipath TCP creates the data sequence acknowledgments from the data contained in this buffer. Once data is in-sequence inside this buffer, it is passed to the application through a `recv` system call.

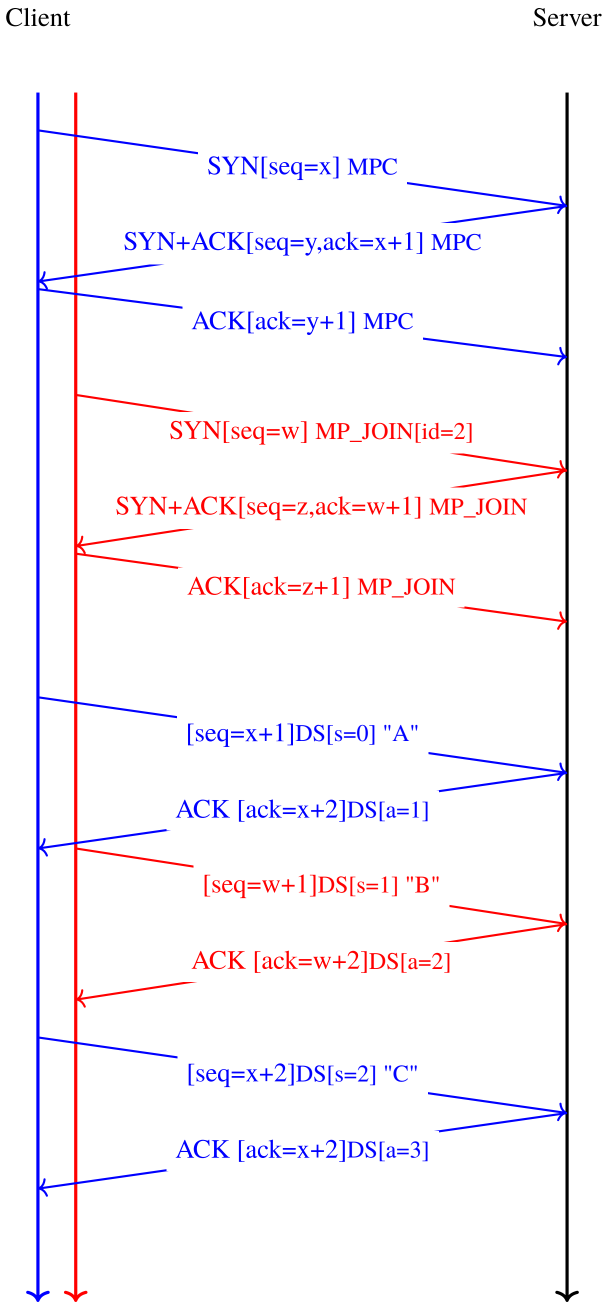


Fig. 5.18: Multipath TCP relies on data sequence numbers and acknowledgments

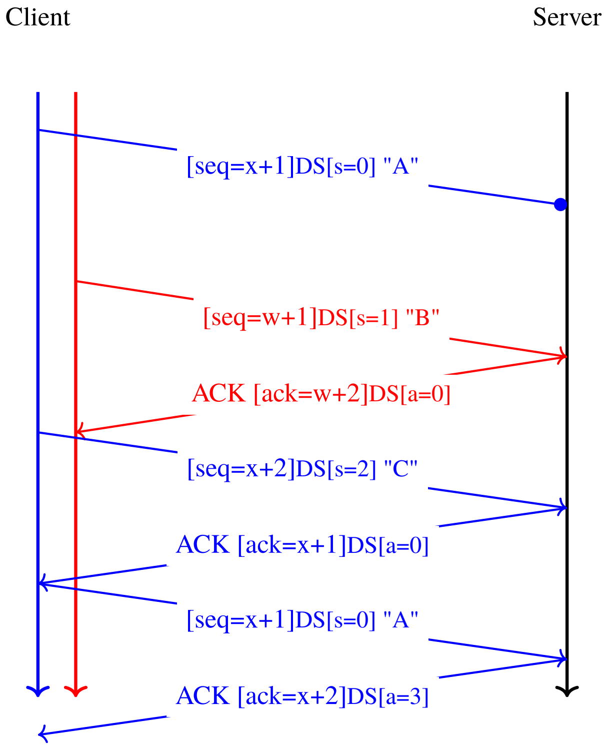


Fig. 5.19: Multipath TCP copes with packet losses

5.5 Congestion control

5.5.1 LIA

[19] and [20]

5.5.2 OLIA

[21]

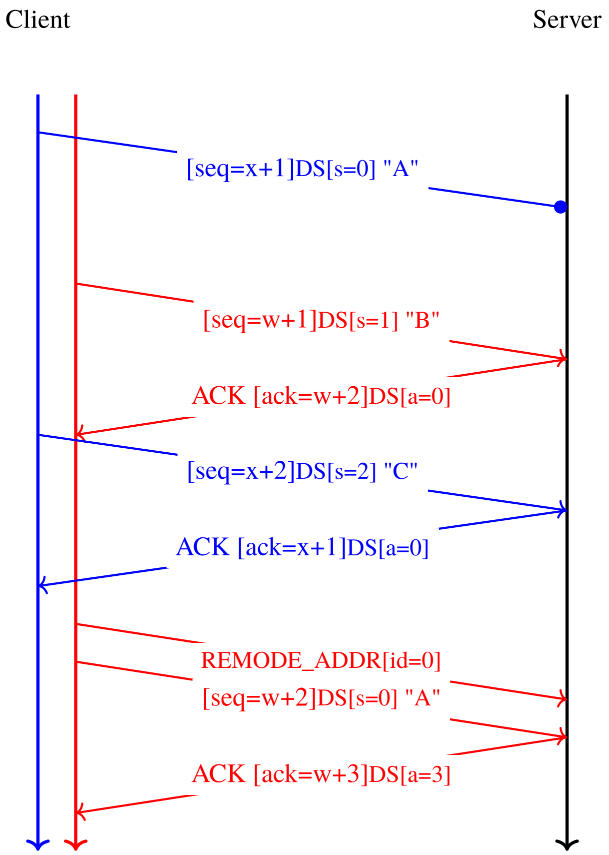


Fig. 5.20: Multipath TCP copes with subflow failures

figure from slide

Fig. 5.21: Architecture of a Multipath TCP implementation

5.5.3 BALIA

[22]

5.5.4 MPCC

[23]

5.6 Connection release

A TCP connection starts with a three-way handshake and ends with either the exchange of FIN packets to gracefully terminate the connection or when one of the hosts sends a RST packet. The main benefit of the graceful termination is that both hosts receive the confirmation that all the data that they have sent over the connection has been correctly received. Multipath also supports a graceful termination of the connection. As in regular TCP, this graceful termination is implemented by using a flag that indicates the end of the bytestream. This flag is included in the Data Sequence Number option.

Fig. 5.22 illustrates a graceful Multipath TCP connection release. We assume that the connection has two active subflows. The client sends XYZ over the initial subflow. Since this is the last byte sent over the bytestream, it adds the DATA_FIN flag to the data sequence option. This flag consumes one data sequence number as the FIN flag in the TCP header. The server returns an acknowledgment that confirms the reception of the three bytes at the subflow level (ack=x+3). At the connection level, four sequence numbers are acknowledged (a=y+4) since the DATA_FIN flag consumes one sequence number. The server decides to close its bytestream by sending its last byte, M, over the second subflow with the DATA_FIN flag set. At this point, the Multipath TCP has been gracefully closed. No data will be exchanged over the different subflows. The client and/or the server can terminate the subflows by using packets with either the FIN or the RST flag in the TCP header. The main drawback of exchanging DATA_FINs to terminate a Multipath TCP is that this takes time. Busy servers might not be willing to spend a long time waiting for the exchange of all these packets if the application already guarantees the correct delivery of the data. A regular TCP server would send a RST packet to quickly terminate such a connection. However, such RST packets can lead to denial of service attacks [24]. A regular TCP receiver mitigates these attacks by checking the sequence number of the RST packet [25]. The Multipath TCP designers did not consider this approach to be safe since an attacker who is able to observe the packets on one path could send a RST packet that would terminate all the subflows used by the connection.

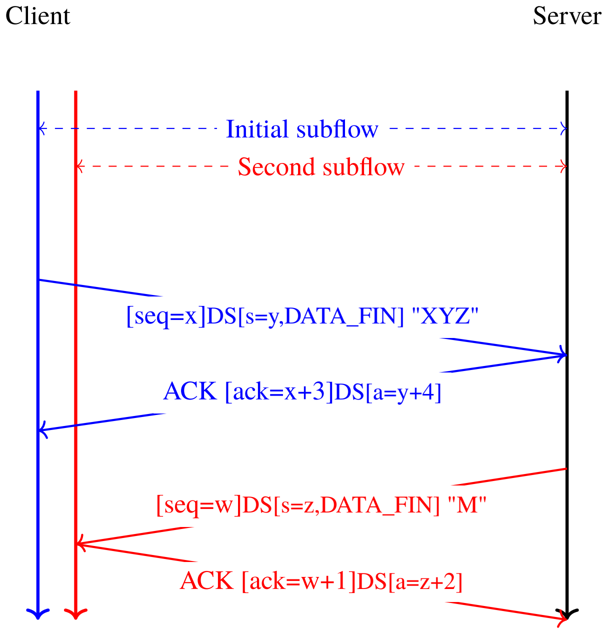


Fig. 5.22: Graceful termination of a Multipath TCP connection

To still allow a host to quickly terminate a Multipath TCP connection, Multipath TCP must be able to verify the validity of a packet that terminates a connection. For this, Multipath TCP defines the FAST_CLOSE option that includes a 64 bits security key. These keys are exchanged during the initial handshake and included in the state associated to a Multipath TCP connection. To quickly close a connection, a host simply needs to send the key of the remote host in a FAST_CLOSE option sent over one of the active subflows. The Multipath TCP specification defines two different methods to use the FAST_CLOSE option.

The first solution is to send the FAST_CLOSE option inside an ACK. Upon reception of such a packet, a host sends a RST over all active subflows. This is illustrated in Fig. 5.23.

5.7 Coping with middlebox interference

The previous sections have explained how Multipath TCP operates at a high level. They assume a simple network that is mainly composed of hosts, switches and routers. TCP and Multipath TCP are used by the hosts. They rely on IP packets that contain the TCP segments. These packets are forwarded by IP routers and possibly switches at layer-2 before reaching their final destination. In a network that uses layered protocols, the switches only inspect the layer-2 headers, the routers only read and change the layer-2 and layer-3 headers. Neither the switches nor the routers read or modify the payload of the packets that they forward. Unfortunately, this assumption is not true on the global Internet and in enterprise networks. Besides switches and routers, these networks contains other types of equipment that process packets [26]. These devices are usually called middleboxes because they reside in the middle of the network and process packets in different ways. A detailed survey of all the different types of middleboxes is outside the scope

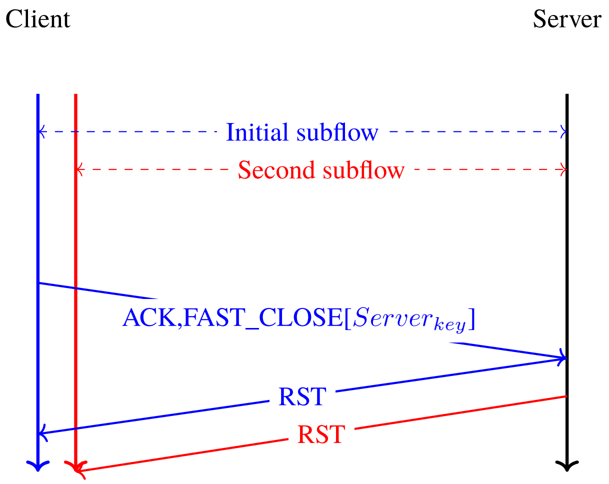


Fig. 5.23: Abrupt release of a Multipath TCP connection by sending FAST_CLOSE inside an ACK

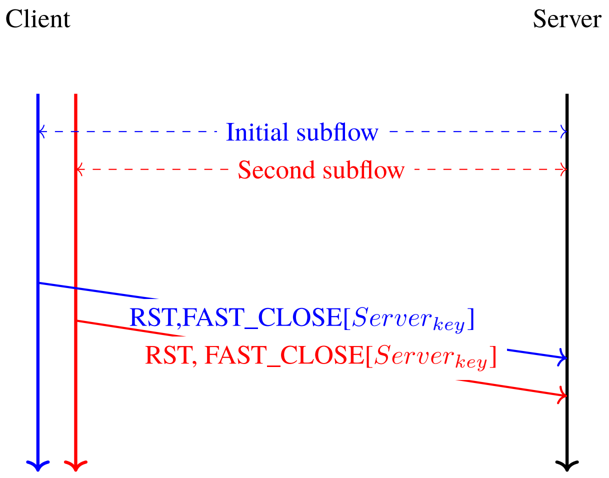

Client
Server
Initial subflow
Second subflow
RST,FAST_CLOSE[$Server_{key}$]
RST, FAST_CLOSE[$Server_{key}$]

Fig. 5.24: Abrupt release of a Multipath TCP connection by sending a RST with FAST_CLOSE on all subflows

of this document. We discuss below some of the popular middleboxes and analyze how they have influenced the design of Multipath TCP.

Our first middlebox is a firewall. A firewall is a device that receives packets, analyzes their contents and then forwards or blocks the packet. The simplest firewalls are the stateless firewalls that accept or reject each individual packet. Such a firewall can accept packet based on the source or destination addresses or port numbers. Some firewalls also check the flags or the IP header or reassemble the received packet fragments. Others analyze the TCP header and verify the utilization of the TCP options. A firewall can be configured using a white list or a black list. A white list specifies all the packet fields that are valid and all the others are invalid. On the other hand, a black list specifies the packets that must be rejected by the firewall and all the others are accepted. Many firewalls use a small white list that defines the TCP options that the firewall accepts. This list typically includes the widely deployed options such as MSS [2], timestamps [27], windows scale [28] and selective acknowledgments [1]. TCP options are encoded using the Kind, Length, Value format shown in Fig. 5.25. It is interesting to explore how such a firewall reacts when it receives a packet containing a TCP option that is not part of its whitelist. There are two possibilities. Some firewalls simply drop the packet, but this blocks a connection that could be totally legitimate. Other firewalls remove the option from the TCP header. This can be done by either removing the bytes that contain the unknown TCP option, adjust the Length field of the IP header, the TCP Header length (and possibly update the padding) and update the TCP checksum. A simpler approach is to replace the bytes of the option with byte `1`. This corresponds to the standard No-Operation TCP option [2]. The advantage of this approach is that the firewall only has to recompute the TCP checksum and does not need to adjust the packet length and move data.

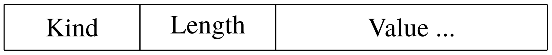


Fig. 5.25: Generic format for TCP options

The removal of TCP options by firewalls has influenced the design of Multipath TCP. Multipath TCP uses TCP options to exchange different types of information. The information carried in a `SYN` is not the same as the one exchanged in data packets. The selective acknowledgments TCP extension [1] defines two different options: a two bytes long `SACK permitted` that is used inside `SYN` and a variable length `SACK` option that carries the selective acknowledgments during the data transfer. The first versions of Multipath TCP used a similar approach with different TCP options kinds. However, the Multipath TCP designers feared that some firewalls could accept some of the Multipath TCP options and drop the others. For example, the Multipath TCP option used in the `SYN` could pass a firewall that would later drop the options used in data packets. It would have been very difficult for a Multipath TCP implementation to deal with all the corner cases that could happen since Multipath TCP [5] currently defines 9 different options. To prevent such problems, Multipath TCP uses a single TCP option kind and each Multipath TCP option contains a subtype field. This increases the length of the Multipath TCP options, but minimizes the risk of middlebox interference. Before looking at other middleboxes, it is interesting to analyze how a router forwards an IP packet that contains a TCP segment. Consider a router that receives a packet such as the one shown in Fig. 5.27. When a router forwards such a packet, it will read the IP header and may modify the fields highlighted in red:

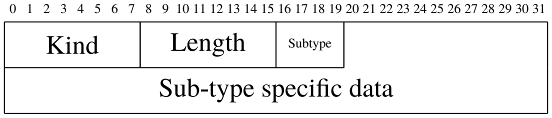


Fig. 5.26: The generic format for Multipath TCP options

- the Differentiated Services Codepoint (DSCP)
- the Explicit Congestion Notification flags (the CE bit)
- decrement the Time to Live
- update the IP header checksum

A router will never change any other field of the IP header and will not read the packet payload. Today, most TCP stacks set the Don't Fragment flag when sending TCP packets. This implies that IPv4 routers will not fragment the packet. Even if a router fragments an IPv4 packet, this is transparent for the TCP stack since the IP stack on the receiver will reassemble the packet before passing its contents to TCP.

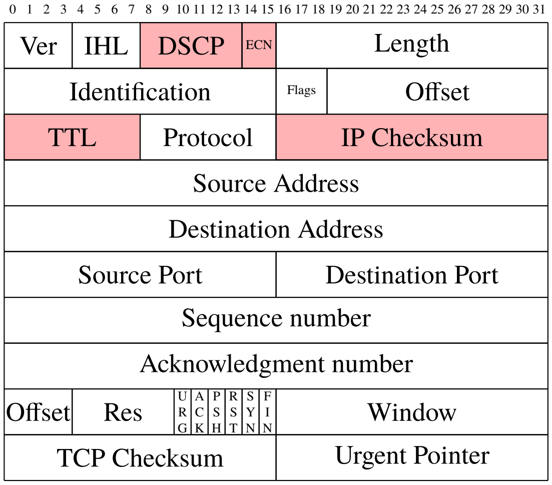


Fig. 5.27: Fields of an IPv4 packet carrying a TCP segment which can be modified by a router

Unfortunately, deployed networks also contain Network Address Translators (NAT) [29]. We consider three different types of NATs because they interfere in different ways with TCP extensions such as Multipath TCP. A NAT is usually located at the boundary between a private network and the Internet. The hosts of the private network use private IP addresses [30] and the NAT is configured with a pool of public addresses. When the NAT receives an IP packet from a host in the private network, its maps the source IP address to a public one and rewrites the packet before forwarding it to the public Internet. When the NAT receives a packet from the Internet, it checks if there is a mapping for the packet's destination address. If so, the destination address is translated and the packet is forwarded to the private host. As illustrated in Fig. 5.28, this NAT updates the source or destination address of the packet depending on the packet direction. This modification forces the NAT to recompute the IP checksum but also the TCP checksum since

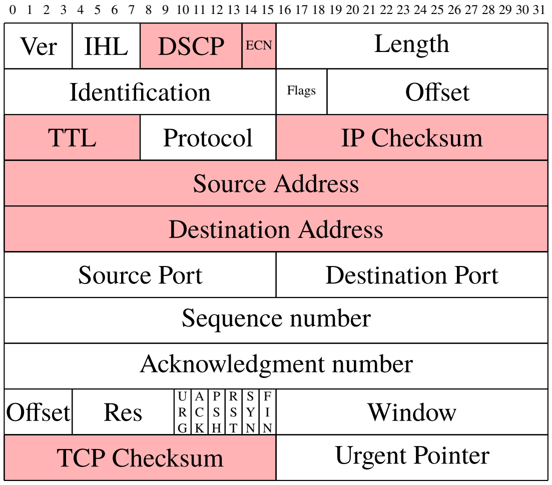


Fig. 5.28: Fields of an IPv4 packet carrying a TCP segment which can be modified by a simple NAT

it covers the TCP packet and a pseudo header that includes the IP addresses [2]. In practice, these two checksums are incrementally updated [29] and do not need to be recomputed.

Multipath TCP copes with these NATs by associating an identifier to each address that is used to create a subflow or advertise an address using the ADD_ADDR option. NAT are not aware of these identifiers and they do not modify them. The REMOVE_ADDR option only contains the identifier of the address that was removed. With this information, the receiver of the option can easily determine the affected subflows.

Most NAT deployments, in particular with IPv4, use a pool a public addresses that is much smaller than the set of private addresses that need to be mapped. These middleboxes also need to translate the source ports used by the internal hosts to map different private addresses to the same public addresses. These Network Address and Port Translators (NAPT) also modify the source or destination ports in the same way as they modify the addresses. Multipath TCP copes with simple NAPTs as with simple NATs. Unfortunately, most NATs and NAPTs also include Application-Level Gateways (ALG). ALGs were designed to enable applications such as the File Transfer Protocol (FTP) [31] to be used through NATs and NAPTs. FTP and a few other protocols use IP addresses as parameters of the application-level messages that are exchanged within the bytestream. A simple FTP session is shown in Listing 5.1. In contrast with many application-level protocols, FTP uses several TCP connection. A FTP sessions starts with a TCP connection established by the client. This connection is called the control connection [31]. This connection is used to exchange simple commands and the associated responses. The client issues commands such as USER (to indicate the client username) or PASS (to provide a password) as a single ASCII line sent over this connection. The server replies with one line that starts with a decimal number that indicates the success of the failure of the command and a textual explanation. However, this is not the only connection used between the client and the server. The client and the server also use data connections. If the client wants to retrieve a file or simply list the names of the files in a given directory, it needs to issue two successive commands. The first command (PORT) indicates the data connection that will be used to exchange the result of the subsequent command. The client listens on a local port and provides its IP address and port number as parameters of the PORT command. Upon reception of this command, the server establish a TCP connection towards the port specific by the client. If the client is behind a NAT, its private IP address and the local port must be translated by the NAT to support the establishment of a server-initiated connection.

Listing 5.1: Simple ftp session

```
#ftp -4d ftp.belnet.be
Connected to ftp-brudie.belnet.be.
220-Welcome to the Belnet public FTP server ftp.belnet.be !

All access is logged.

Currently used storage capacity : 38T / 100T on /ftp
220 193.190.198.27 FTP server ready
Name (ftp.belnet.be): anonymous
---> USER anonymous
331 Anonymous login ok, send your complete email address as your password
Password:
---> PASS XXXX
230 Anonymous access granted, restrictions apply
---> SYST
215 UNIX Type: L8
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> dir
---> PORT 192,168,0,37,133,67
200 PORT command successful
---> LIST
150 Opening ASCII mode data connection for file list
```

(continues on next page)

(continued from previous page)

```
lrw-r--r--   1 ftp      ftp            16 Feb 24  2021 arcolinux -> mirror/arcolinux
drwxr-xr-x   3 ftp      ftp           101 Jan 12  2021 belnetstyle
lrw-r--r--   1 ftp      ftp            13 Feb  1  2021 debian -> mirror/debian
226 Transfer complete
ftp> quit
---> QUIT
221 Goodbye.
```

It is interesting to analyze how an ALG modifies a packet that carries such a `PORT` command. Let us assume that the `PORT 192,168,0,37,133,67` command is sent in a single TCP packet for simplicity. Fig. 5.29 shows the contents of the packet sent by the client. Fig. 5.30 shows the packet after its translation by the NAT, assuming that the NAT maps IP address `192.168.0.37` onto address `5.6.7.8`. The packet sent by client contains 26 bytes of payload. The IP packet is thus 66 bytes long. The `PORT 192,168,0,37,133,67` indicates that the client listens on IP address `192.168.0.37` and on port $133 * 256 + 67 = 34115$. Let us assume that the NAT maps this IP address on address `5.6.7.8` and port `34115` on port $31533 = 123 * 256 + 45$. In ASCII, the `PORT` command becomes `PORT 5,6,7,8,123,45` and the NAT sends the packet shown in Fig. 5.30. The fields shown in red have been translated by the NAT. An important point to note contains 21 bytes of payload and not 66 as the packet sent by the client. This implies that the packet sent by the NAT contains the bytes having sequence numbers `12300` to `12320` while the original packet covered sequence numbers `12300` to `12325`. The NAT will thus need to adjust the sequence number of the subsequent packets sent by the client and also the acknowledgments returned by the server. As shown by the example above, an ALG can change bytes in the bytestream. It can also remove bytes from the bytestream and also add bytes in the bytestream. This happens notably when the ASCII representation of the public IP address of the NAT is longer than the private IP address of the internal host. This modification of the bytestream had a major impact on the design of Multipath TCP. It mainly affects the

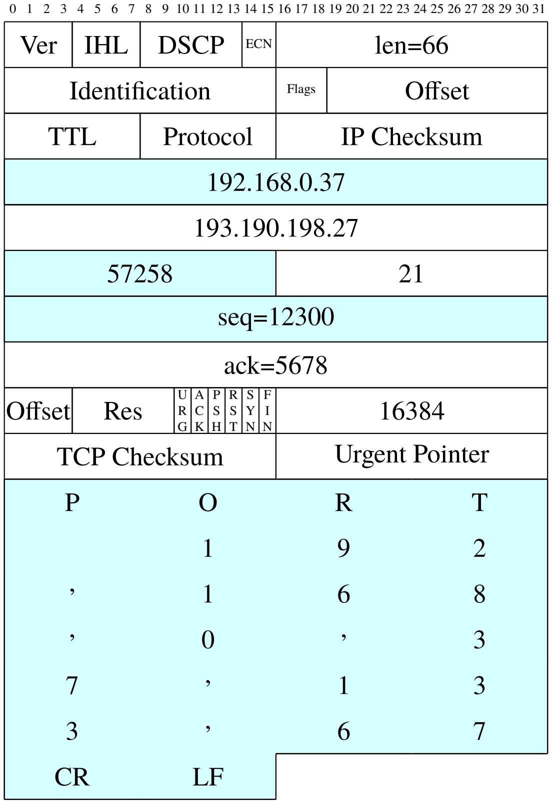


Fig. 5.29: Packet carrying a PORT command sent by a client

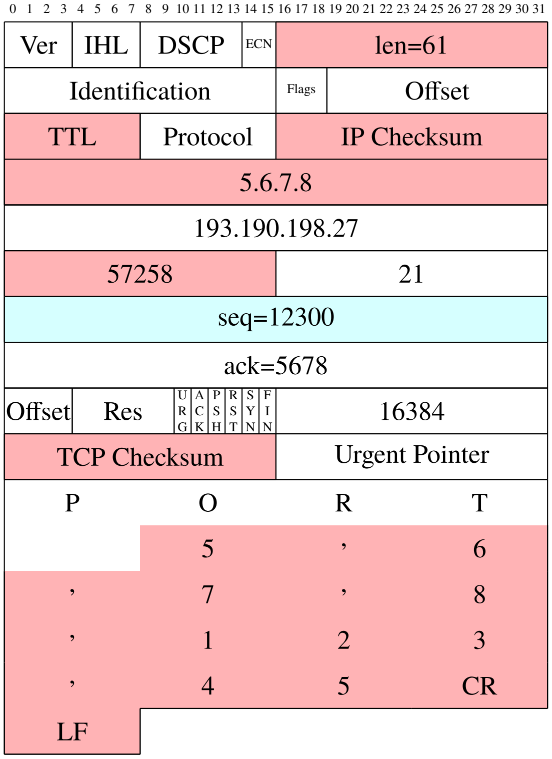


Fig. 5.30: Packet carrying a PORT command modified by the FTP ALG used by a NAT

Data Sequence Number option that carries the data sequence numbers and acknowledgments. To detect modifications from ALGs and other middleboxes, this option covers a range of sequence numbers in the bytestream and includes an optional checksum that is computed by the Multipath TCP sender and checked by the receiver. If there is a mismatch between the checksum of the option and the data, the receiver stops using Multipath TCP and falls back to regular TCP to preserve the established connection. We discuss this fallback in more details later.

Our third type of middlebox that splits or coalesces TCP packets. This is not a router that performs IPv4 fragmentation or a host that splits a large IPv6 packets in fragments. In-network fragmentation is mainly disabled in IPv4 network since modern TCP stacks set the `DF` flag of the IP header. Those middleboxes do not reside in the middle of the network. They are typically included in the network adapter used by servers and even client hosts. Measurement studies have shown that hosts can reach a higher throughput when sending and receiving large packets. For example, a recent study [32] reveals that over a 100 Gbps interface, a server was able to reach 25 Gbps with a single TCP connection using 1500 bytes packets. The same connection reached 40 Gbps by using jumbo frames, i.e. 9000 bytes packets. The jumbo frames are supported on modern Gigabit Ethernet networks but they are rarely used outside datacenters because most Internet paths still only supports 1500 bytes packets.

Modern network adapters support TCP Segmentation Offload (TSO) to improve the throughput of TCP connection are reduce the CPU load. In a nutshell, when TSO is enabled, the network adapter exposes a large maximum packet size, e.g. 16 KBytes to more, to the network stack. When the host sends such a large packet, it is automatically segmented in a sequence of small IP packets. On the receiver side, the network adapter performs the reverse operation. It coalesces small received packets into a larger one. Fig. 5.31 shows a large (2 KBytes long) TCP packet. It is interesting to analyze how the key fields of this packet will be processed by TSO to segment it in the two smaller packets. To segment the packet shown in Fig. 5.31 in two smaller packets, TSO creates two `1040` bytes long IPv4 packets. The two small packets have a different IP Identification than the large one. TSO computes an IP checksum for each small packet. It then copies the TCP header of the large packet in both small ones, but with a fed adjustments. The sequence number of the first small packet is the same as the large one. The sequence number of the sequence small packet is the one of the

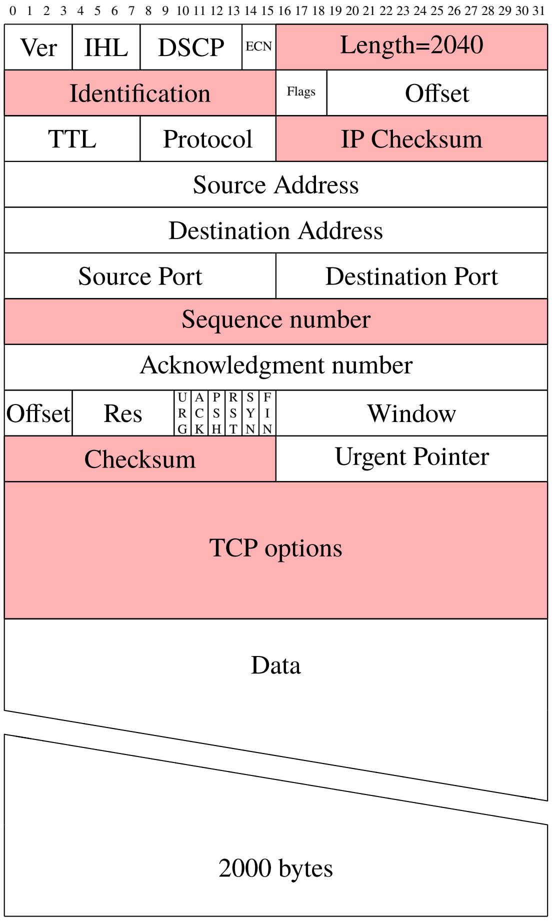


Fig. 5.31: A large IP packet containing TCP header and data

first packet increased by 1000. Concerning the TCP options, TSO could analyze the contents of the option and handle each option in a specific manner. For example, TSO could adjust the TCP timestamp option of successive packets. In practice, measurements indicate that TSO simply copies the TCP options field of the large packet in all small packets [33]. TSO places the first 1000 bytes of the payload of the large packet in the first small one and the last 1000 bytes in the second one. Finally, TSO needs to update the TCP checksum in all the small packets.

The receiver side of these network adapters implement Large Receive Offload (LRO). This basically coalesces the packets that were segmented by TSO. In this case, coalescing packets that carry different TCP options could be problematic since some of the TCP options would be lost in this process. Measurements with different TCP options show that LRO only coalesces packets that have exactly the same set of TCP implementations.

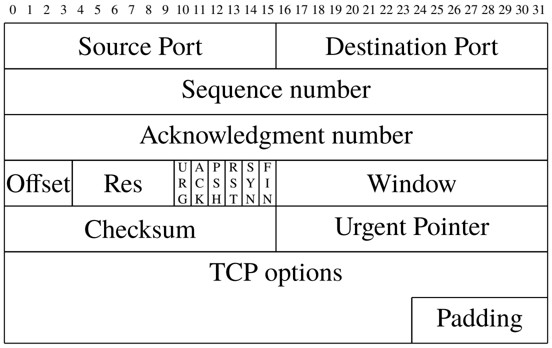


Fig. 5.32: The TCP header

5.8 The protocol details

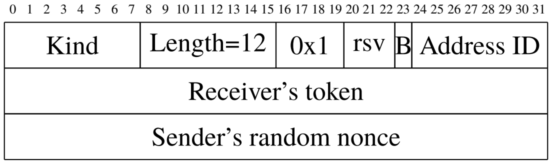


Fig. 5.33: The MP_JOIN option in a SYN packet

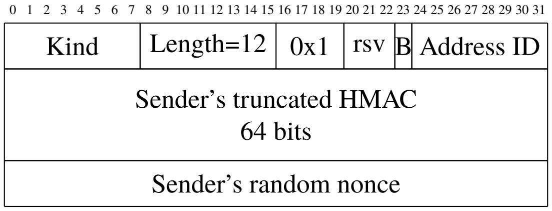


Fig. 5.34: The MP_JOIN option in a SYN+ACK packet

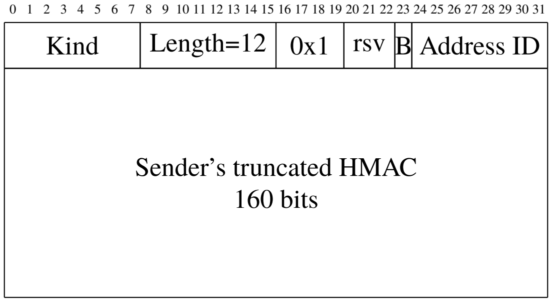


Fig. 5.35: The MP_JOIN option in the initiator's first ACK

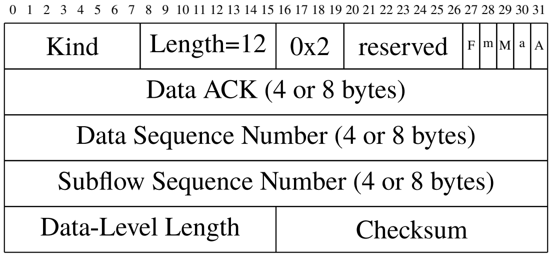


Fig. 5.36: The Data Sequence Signal option

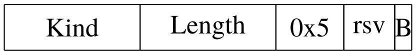


Fig. 5.37: The MP_PRIO option

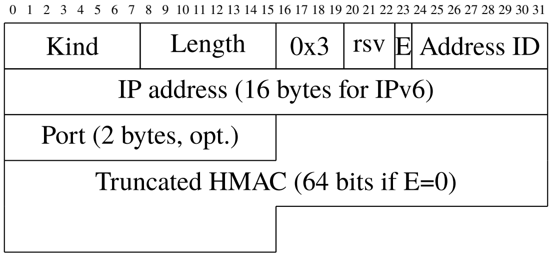


Fig. 5.38: The ADD_ADDR option

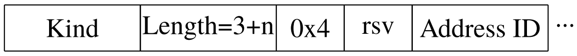


Fig. 5.39: The REMOVE_ADDR option

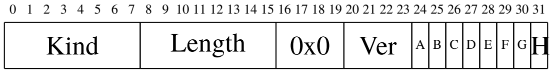


Fig. 5.40: The MP_CAPABLE option in a SYN packet

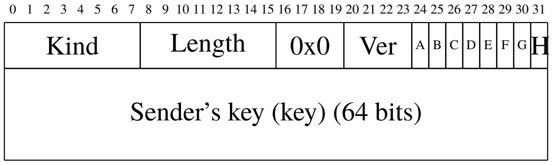


Fig. 5.41: The MP_CAPABLE option in SYN+ACK packet

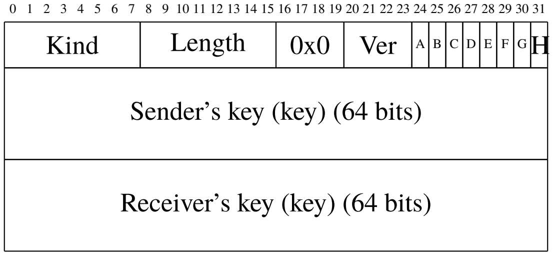


Fig. 5.42: The MP_CAPABLE option in initiator’s first ACK (without data)

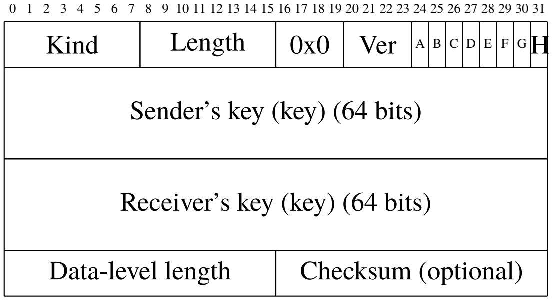


Fig. 5.43: The MP_CAPABLE option in initiator’s first ACK (with data)

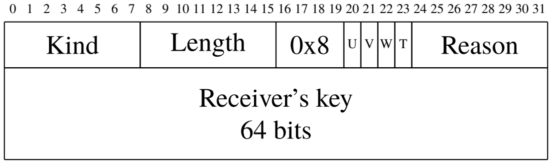


Fig. 5.44: The MP_TCPRST option

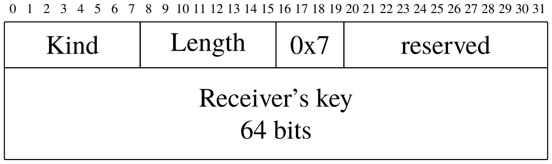


Fig. 5.45: The FAST_CLOSE option

CHAPTER

SIX

BIBLIOGRAPHY

CHAPTER SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[1] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. RFC 2018 (Proposed Standard), October 1996. URL: https://www.rfc-editor.org/rfc/rfc2018.txt, doi:10.17487/RFC2018.

[2] J. Postel. Transmission Control Protocol. RFC 793 (Internet Standard), September 1981. Updated by RFCs 1122, 3168, 6093, 6528. URL: https://www.rfc-editor.org/rfc/rfc793.txt, doi:10.17487/RFC0793.

[3] J. Nagle. Congestion Control in IP/TCP Internetworks. RFC 896 (Historic), January 1984. Obsoleted by RFC 7805. URL: https://www.rfc-editor.org/rfc/rfc896.txt, doi:10.17487/RFC0896.

[4] V. Paxson, M. Allman, J. Chu, and M. Sargent. Computing TCP's Retransmission Timer. RFC 6298 (Proposed Standard), June 2011. URL: https://www.rfc-editor.org/rfc/rfc6298.txt, doi:10.17487/RFC6298.

[5] A. Ford, C. Raiciu, M. Handley, O. Bonaventure, and C. Paasch. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 8684 (Proposed Standard), March 2020. URL: https://www.rfc-editor.org/rfc/rfc8684.txt, doi:10.17487/RFC8684.

[6] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6824 (Experimental), January 2013. Obsoleted by RFC 8684. URL: https://www.rfc-editor.org/rfc/rfc6824.txt, doi:10.17487/RFC6824.

[7] C. Hopps. Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992 (Informational), November 2000. URL: https://www.rfc-editor.org/rfc/rfc2992.txt, doi:10.17487/RFC2992.

[8] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar. Architectural Guidelines for Multipath TCP Development. RFC 6182 (Informational), March 2011. URL: https://www.rfc-editor.org/rfc/rfc6182.txt, doi:10.17487/RFC6182.

[9] C. Filsfils (Ed.), S. Previdi (Ed.), L. Ginsberg, B. Decraene, S. Litkowski, and R. Shakir. Segment Routing Architecture. RFC 8402 (Proposed Standard), July 2018. URL: https://www.rfc-editor.org/rfc/rfc8402.txt, doi:10.17487/RFC8402.

[10] C. Filsfils (Ed.), D. Dukes (Ed.), S. Previdi, J. Leddy, S. Matsushima, and D. Voyer. IPv6 Segment Routing Header (SRH). RFC 8754 (Proposed Standard), March 2020. URL: https://www.rfc-editor.org/rfc/rfc8754.txt, doi:10.17487/RFC8754.

[11] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving Datacenter Performance and Robustness with Multipath TCP. In *ACM SIGCOMM 2011*. 2011. URL: http://doi.acm.org/10.1145/2018436.2018467, doi:10.1145/2018436.2018467.

[12] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), February 1997. Updated by RFC 6151. URL: https://www.rfc-editor.org/rfc/rfc2104.txt, doi:10.17487/RFC2104.

[13] M. Bagnulo. Threat Analysis for TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6181 (Informational), March 2011. URL: https://www.rfc-editor.org/rfc/rfc6181.txt, doi:10.17487/RFC6181.

[14] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (Proposed Standard), August 2018. URL: https://www.rfc-editor.org/rfc/rfc8446.txt, doi:10.17487/RFC8446.

[15] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. RFC 1323 (Proposed Standard), May 1992. Obsoleted by RFC 7323. URL: https://www.rfc-editor.org/rfc/rfc1323.txt, doi:10.17487/RFC1323.

[16] D. Eastlake 3rd and T. Hansen. US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF). RFC 6234 (Informational), May 2011. URL: https://www.rfc-editor.org/rfc/rfc6234.txt, doi:10.17487/RFC6234.

[17] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. How hard can it be? Designing and implementing a deployable Multipath TCP. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, 29–29. Berkeley, CA, USA, 2012. USENIX Association. URL: http://inl.info.ucl.ac.be/publications/how-hard-can-it-be-designing-and-implementing-deployable-multipath-tcp.

[18] Nicolas Keukeleire, Benjamin Hesmans, and Olivier Bonaventure. Increasing broadband reach with hybrid access networks. *IEEE Communications Standards Magazine*, 4(1):43–49, 2020.

[19] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. Design, implementation and evaluation of congestion control for Multipath TCP. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, 99–112. Berkeley, CA, USA, 2011. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1972457.1972468.

[20] C. Raiciu, M. Handley, and D. Wischik. Coupled Congestion Control for Multipath Transport Protocols. RFC 6356 (Experimental), October 2011. URL: https://www.rfc-editor.org/rfc/rfc6356.txt, doi:10.17487/RFC6356.

[21] R. Khalili, N. Gast, M. Popovic, and J.-Y. Le Boudec. MPTCP is not pareto-optimal: performance issues and a possible solution. *Networking, IEEE/ACM Transactions on*, 21(5):1651–1665, Oct 2013. doi:10.1109/TNET.2013.2274462.

[22] Qiuyu Peng, Anwar Walid, Jaehyun Hwang, and Steven H Low. Multipath tcp: analysis, design, and implementation. *IEEE/ACM Transactions on networking*, 24(1):596–609, 2014.

[23] Tomer Gilad, Neta Rozen-Schiff, P Brighten Godfrey, Costin Raiciu, and Michael Schapira. Mpcc: online learning multipath transport. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, 121–135. 2020.

[24] A. Ramaiah, R. Stewart, and M. Dalal. Improving TCP's Robustness to Blind In-Window Attacks. RFC 5961 (Proposed Standard), August 2010. URL: https://www.rfc-editor.org/rfc/rfc5961.txt, doi:10.17487/RFC5961.

[25] F. de Bont, S. Doehla, M. Schmidt, and R. Sperschneider. RTP Payload Format for Elementary Streams with MPEG Surround Multi-Channel Audio. RFC 5691 (Proposed Standard), October 2009. URL: https://www.rfc-editor.org/rfc/rfc5691.txt, doi:10.17487/RFC5691.

[26] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: network processing as a cloud service. *ACM SIGCOMM Computer Communication Review*, 42(4):13–24, 2012.

[27] K. Patel, E. Chen, and B. Venkatachalapathy. Enhanced Route Refresh Capability for BGP-4. RFC 7313 (Proposed Standard), July 2014. URL: https://www.rfc-editor.org/rfc/rfc7313.txt, doi:10.17487/RFC7313.

[28] C. Partridge. Today's Programming for KRFC AM 1313 Internet Talk Radio. RFC 1313 (Informational), April 1992. URL: https://www.rfc-editor.org/rfc/rfc1313.txt, doi:10.17487/RFC1313.

[29] P. Srisuresh and K. Egevang. Traditional IP Network Address Translator (Traditional NAT). RFC 3022 (Informational), January 2001. URL: https://www.rfc-editor.org/rfc/rfc3022.txt, doi:10.17487/RFC3022.

[30] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear. Address Allocation for Private Internets. RFC 1918 (Best Current Practice), February 1996. Updated by RFC 6761. URL: https://www.rfc-editor.org/rfc/rfc1918.txt, doi:10.17487/RFC1918.

[31] J. Postel and J. Reynolds. File Transfer Protocol. RFC 959 (Internet Standard), October 1985. Updated by RFCs 2228, 2640, 2773, 3659, 5797, 7151. URL: https://www.rfc-editor.org/rfc/rfc959.txt, doi:10.17487/RFC0959.

[32] Mario Hock, Maxime Veit, Felix Neumeister, Roland Bless, and Martina Zitterbart. TCP at 100 Gbit/s–tuning, limitations, congestion control. In *2019 IEEE 44th Conference on Local Computer Networks (LCN)*, 1–9. IEEE, 2019.

[33] Michio Honda, Yoshifumi Nishida, Costin Raiciu, Adam Greenhalgh, Mark Handley, and Hideyuki Tokuda. Is it still possible to extend TCP? In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '11, 181–194. New York, NY, USA, 2011. ACM. URL: http://doi.acm.org/10.1145/2068816.2068834, doi:10.1145/2068816.2068834.

INDEX