Simplemux v2 Readme file

Content

About Simplemux

There are some situations in which multiplexing a number of small packets into a bigger one is desirable. For example, a number of small packets can be sent together between a pair of machines if they share a common network path. Thus, the traffic profile can be shifted from small to larger packets, reducing the network overhead and the number of packets per second to be managed by intermediate routers.

Simplemux is a generic multiplexing protocol, described in draft-saldana-tsvwg-simplemux. It is able to encapsulate a number of packets belonging to different protocols into a single packet. It includes the "Protocol" field on each multiplexing header, thus allowing the inclusion of a number of packets belonging to different protocols in a packet of another protocol (Fig. 1). The size of the multiplexing headers is kept very low (it may be a single byte when multiplexing small packets) in order to reduce the overhead.

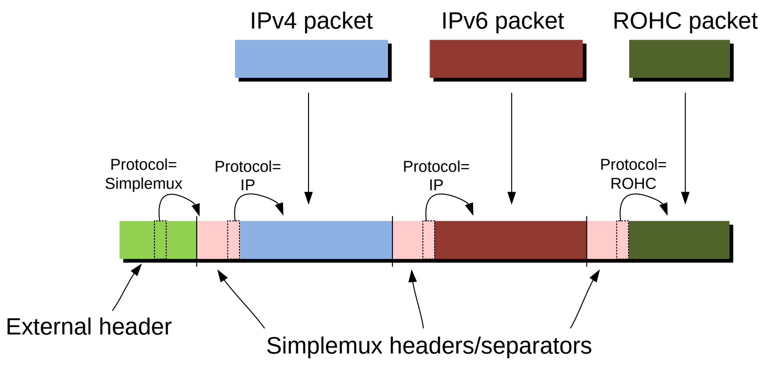


Fig. 1. Example of a Simplemux packet including packets of different protocols

Simplemux is designed to optimize together several flows sharing a common network path or segment (Fig. 2). Optimization in the end host is (in principle) not useful. The multiplexing is performed between a pair of machines called ingress-optimizer and egress-optimizer.

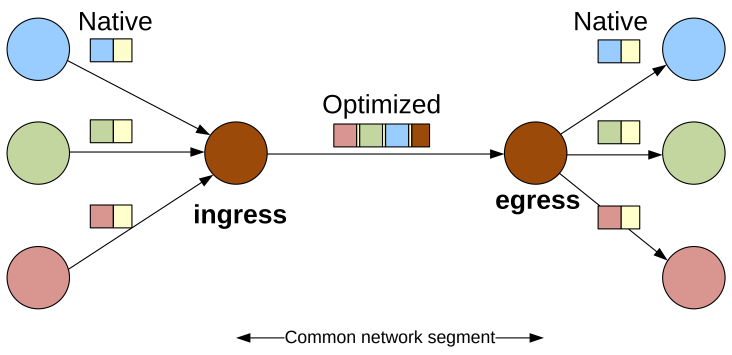


Fig. 2. Scheme of the optimization

The present code is an implementation of **Simplemux**, using the next options for the *tunneling protocol:*

- IP
- UDP/IP
- TCP/IP

and the next options for the *multiplexed protocol:*

- IP packets
- ROHC (RFC 5795) compressed packets
- Ethernet frames

As illustrated in Fig. 3, it can perform traffic optimization, combining multiplexing with header compression. This may result on significant bandwidth savings and pps reductions for small-packet flows (e.g. VoIP, online games).

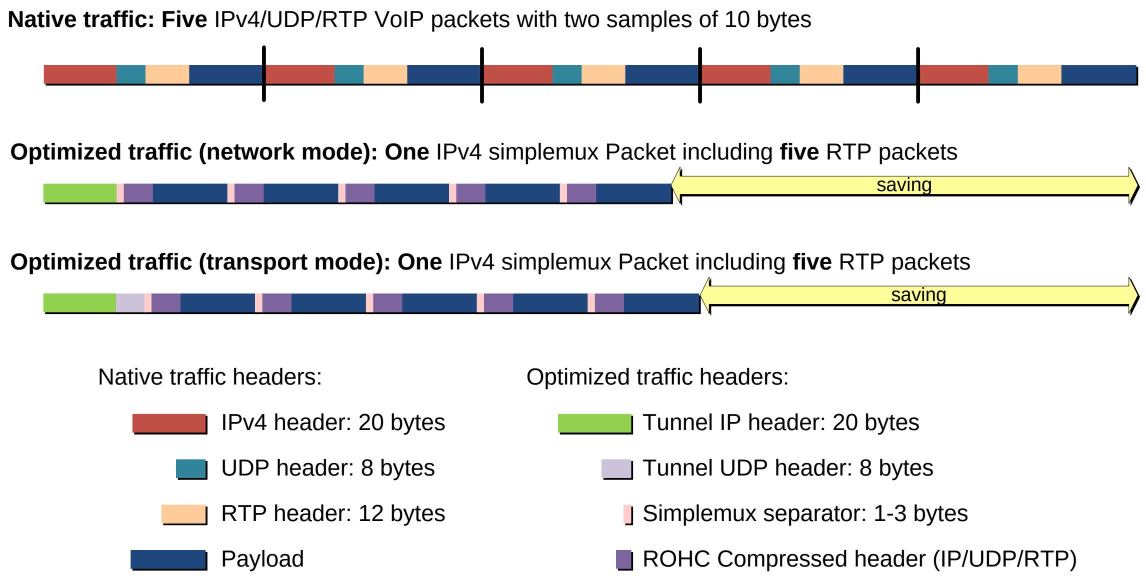

Native traffic: Five IPv4/UDP/RTP VoIP packets with two samples of 10 bytes
Optimized traffic (network mode): One IPv4 simplemux Packet including **five** RTP packets
saving
Optimized traffic (transport mode): One IPv4 simplemux Packet including **five** RTP packets
saving
Native traffic headers:
IPv4 header: 20 bytes
UDP header: 8 bytes
RTP header: 12 bytes
Payload
Optimized traffic headers:
Tunnel IP header: 20 bytes
Tunnel UDP header: 8 bytes
Simplemux separator: 1-3 bytes
ROHC Compressed header (IP/UDP/RTP)

Fig. 3. Optimization of five VoIP RTP packets

Modes

Simplemux can run in four *modes*:

- **Network mode**: the multiplexed packet is sent in an IP datagram using Protocol Number 253 or 254 (according to IANA, these numbers can be used for experimentation and testing[1]).

- **UDP mode**: the multiplexed packet is sent in an UDP/IP datagram. In this case, the protocol number in the outer IP header is that of UDP (17) and both ends must agree on a UDP port (the implementation uses 55555 or 55557 by default).

- **TCP server mode**: the multiplexed packet is sent in a TCP/IP datagram. In this case, the protocol number in the outer IP header is that of TCP (4) and both ends must agree on a TCP port (the implementation uses 55555 or 55557 by default).

- **TCP client mode**: same as TCP server mode, but it is the TCP client.

Flavors

Simplemux has two *flavors*:

- **Normal**: it tries to compress the separators as much as possible. For that aim, some single-bit fields are used.

- **Fast**: it sacrifices some compression on behalf or speed. In this case, all the separators are 3-byte long, and all have the same structure.

Simplemux *fast* must be used in TCP mode.

Tunneling modes

Simplemux uses Linux TUN/TAP virtual interface. Two *tunneling modes* are allowed:

- TUN tunneling mode: IP packets are multiplexed between the two endpoints.

- TAP tunneling mode: Ethernet frames are multiplexed between the two endpoints.

Fig. 4 shows the protocol stack used for sending an Ethernet frame (TAP tunneling mode) carrying a UDP/IP packet, using IP, UDP and TCP modes.

[1] Protocol numbers, http://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml

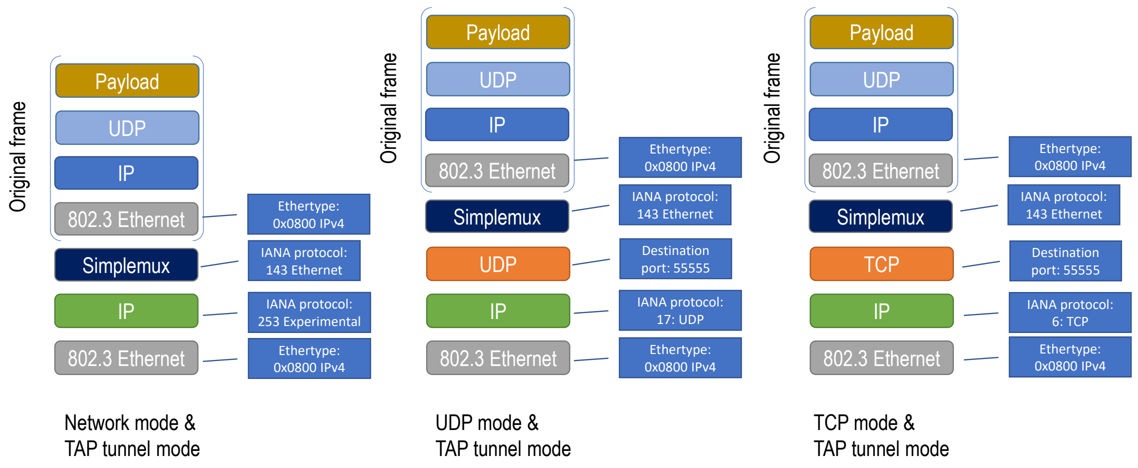


Fig. 4. Sending a UDP Ethernet frame in TAP tunneling mode: IP, UDP and TCP modes.

Fig. 5 presents the protocol stack used for sending a UDP/IP uncompressed packet (TUN mode), using IP, UDP and TCP modes.

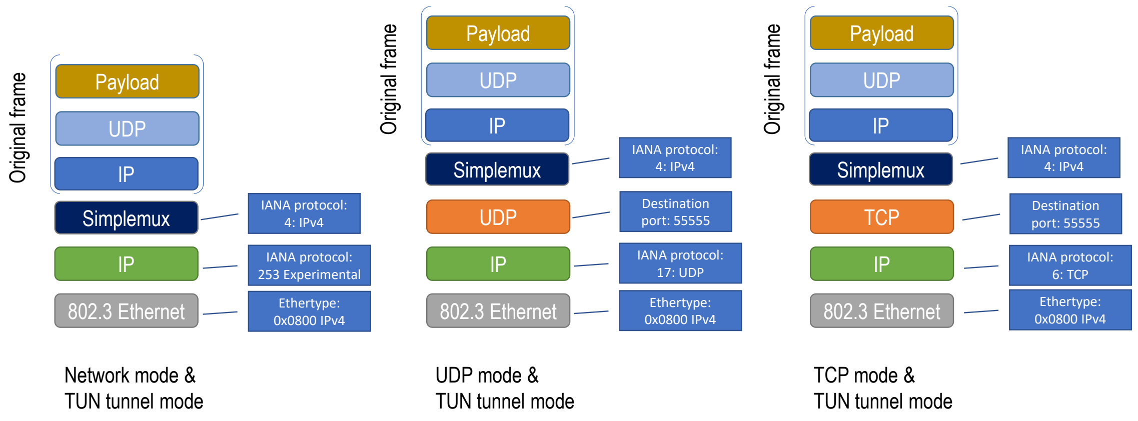


Fig. 5. Tunneling a UDP packet in TUN tunneling mode: IP, UDP and TCP modes.

Fig. 6 presents the protocol stack used for sending a ROHC compressed packet (TUN mode), using IP, UDP and TCP modes.

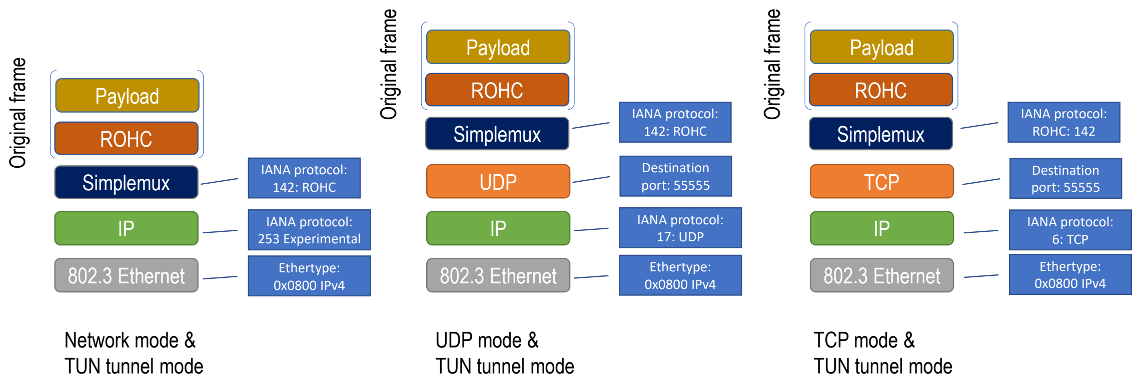


Fig. 6. Tunneling a ROHC compressed packet in TUN tunneling mode: IP, UDP and TCP modes.

Simplemux can be used as an option at the multiplexing layer of TCM[2], a combination of protocols for **T**unneling, **C**ompressing and **M**ultiplexing, allowing the optimization of small-packet flows. TCM may use of a number of different standard algorithms for header compression, multiplexing and tunneling, combined in a similar way to RFC 4170, as proposed in draft-saldana-tsvwg-tcmtf.

The implementation of Simplemux is written in C for Linux. It compresses headers using an implementation of ROHC by Didier Barvaux (https://rohc-lib.org/).

Header compression

This **Simplemux** implementation includes these ROHC modes:

- ROHC unidirectional.

- ROHC bidirectional optimistic

- ROHC bidirectional reliable is not yet implemented.

ROHC cannot be enabled in one of the peers and disabled in the other peer.

ROHC is able to compress this traffic flows:

- IP/UDP/RTP: If the UDP packets have the destination ports 1234, 36780, 33238, 5020, 5002, the compressor assumes that they are RTP.
- IP/UDP
- IP/TCP
- IP/ESP
- IP/UDP-Lite

Tunneling

In this implementation, tunneling is performed this way:

- In **network mode**, the external IP header is the tunneling header. It uses Protocol number 253 or 254 (*fast* flavor).

- In **UDP mode**, a UDP/IP header is added before the first **Mux separator.** By default, the destination UDP port is 55555 or 55557 (*fast* flavor).

- In **TCP mode**, a TCP/IP header is added before the first Mux separator. By default, the destination TCP port is 55555 or 55557 (*fast* flavor).

ROHC feedback information (when using ROHC Bidirectional modes) is sent in UDP packets using port 55556 by default.

[2] Jose Saldana *et al, "Emerging Real-time Services: Optimizing Traffic by Smart Cooperation in the Network*," IEEE Communications Magazine, Vol. 51, n. 11, pp 127-136, Nov. 2013. doi 10.1109/MCOM.2013.6658664

Multiplexing

Normal flavor

The **Simplemux separator** (see Fig. 7) has two different formats: one for the *First header* (the separator before the first packet included in the multiplexed bundle), and another one for *Non-first headers* (the rest of the separators).

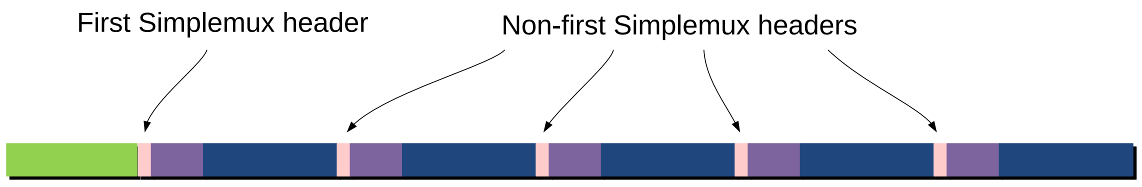


Fig. 7. First and Non-first Simplemux headers (also known as separators)

Format of the First Simplemux header/separator

In order to allow the multiplexing of packets of any length, the number of bytes expressing the length is variable, and the field Length Extension (`LXT`, one bit) is set to `0` if the current byte is the last one including length information.

These are the fields of the header:

- **Single Protocol Bit (`SPB`, one bit)** only appears in the first Simplemux header. It is set to `1` if all the multiplexed packets belong to the same protocol (in this case, the *Protocol* field will only appear in the first Simplemux header). It is set to `0` when each packet MAY belong to a different protocol.

- **Length Extension (`LXT`, one bit)** is `0` if the current byte is the last byte where the length of the first packet is included, and `1` in other case.

- **Length (`LEN`, 6, 13, 20, etc. bits).** This is the length of the multiplexed packet (in bytes), not including the length field. If the length of the multiplexed packet is less than 64 bytes (less than or equal to 63 bytes), the first `LXT` is set to `0` and the 6 bits of the length field are the length of the multiplexed packet. If the length of the multiplexed packet is equal or greater than 64 bytes, additional bytes are added. The first bit of each of the added bytes is the `LXT`. If `LXT` is set to `1`, it means that there is an additional byte for expressing the length. This allows to multiplex packets of any length (see Fig. 8).

- **Protocol (8 bits)** is the *Protocol* field of the multiplexed packet, according to IANA "Assigned Internet Protocol Numbers".

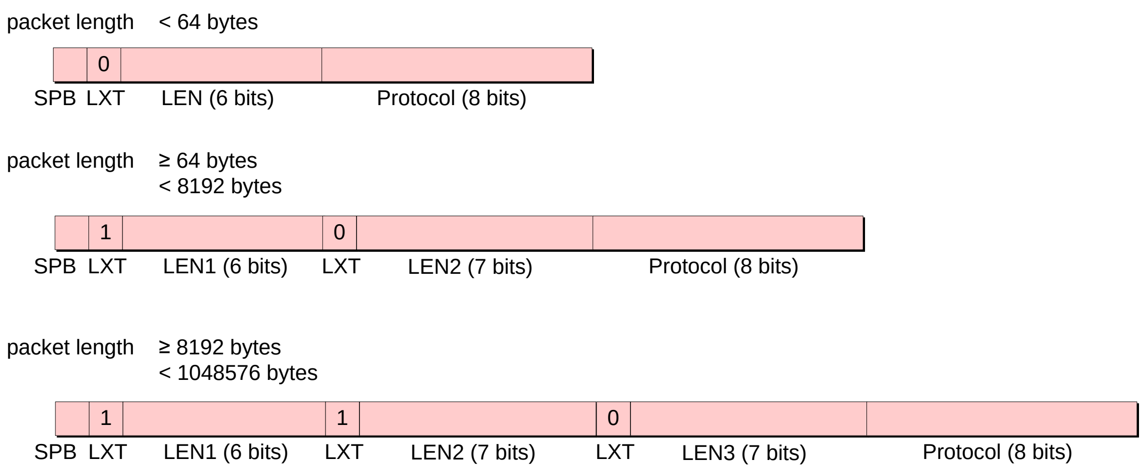

packet length < 64 bytes
0
SPB LXT LEN (6 bits) Protocol (8 bits)
packet length ≥ 64 bytes < 8192 bytes
1
0
SPB LXT LEN1 (6 bits) LXT LEN2 (7 bits) Protocol (8 bits)
packet length ≥ 8192 bytes < 1048576 bytes
1
1
0
SPB LXT LEN1 (6 bits) LXT LEN2 (7 bits) LXT LEN3 (7 bits) Protocol (8 bits)

Fig. 8. Fields of the First Simplemux header/separator

For example, in the case of a packet of 65 bytes, the Simplemux separator will be 3 bytes long. In this case, the length of the packet will be the number expressed by the concatenation of the bits of Length 1 - Length 2 (total 13 bits). Length 1 includes the 6 most significant bits and Length 2 the 7 less significant bits.

More bytes can be added to the length if required, using the same scheme: 1 `LXT` byte plus 7 bits for expressing the length.

Format of the Non-first Simplemux header/separator

The Non-first Simplemux headers also employ a format allowing the multiplexing of packets of any length, so the number of bytes expressing the length is variable, and the field Length Extension (`LXT`, one bit) is set to `0` if the current byte is the last one including length information.

If the `SPB` (Single Protocol Bit) of the First Simplemux header is set to `1`, it means that all the multiplexed packets belong to the same protocol. In this case, the format is the one presented in Fig. 9:

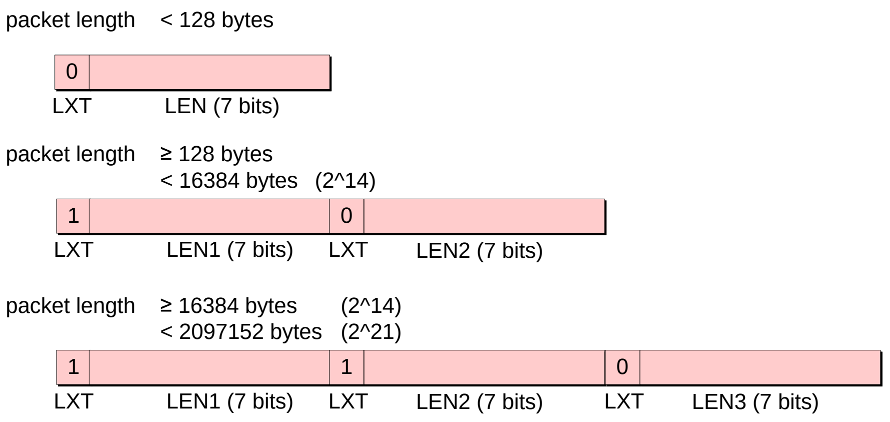


Fig. 9. Fields of the Non-first Simplemux header/separator, when the `SPB` bit of the First header is `1`

If the `SPB` of the First Simplemux header is set to `0`, then each packet may belong to a different protocol, so the *Protocol* field is also included, as shown in Fig. 10:

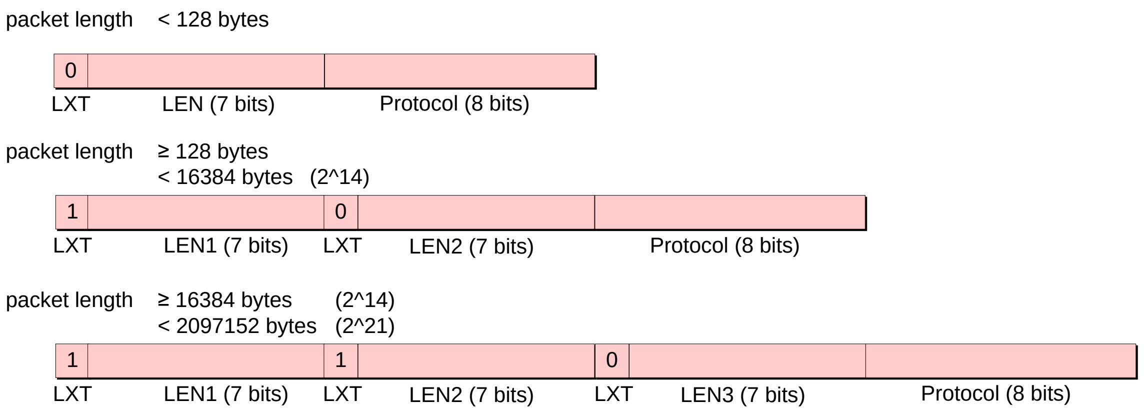


Fig. 10. Fields of the Non-first Simplemux header/separator, when the `SPB` bit of the First header is `0`

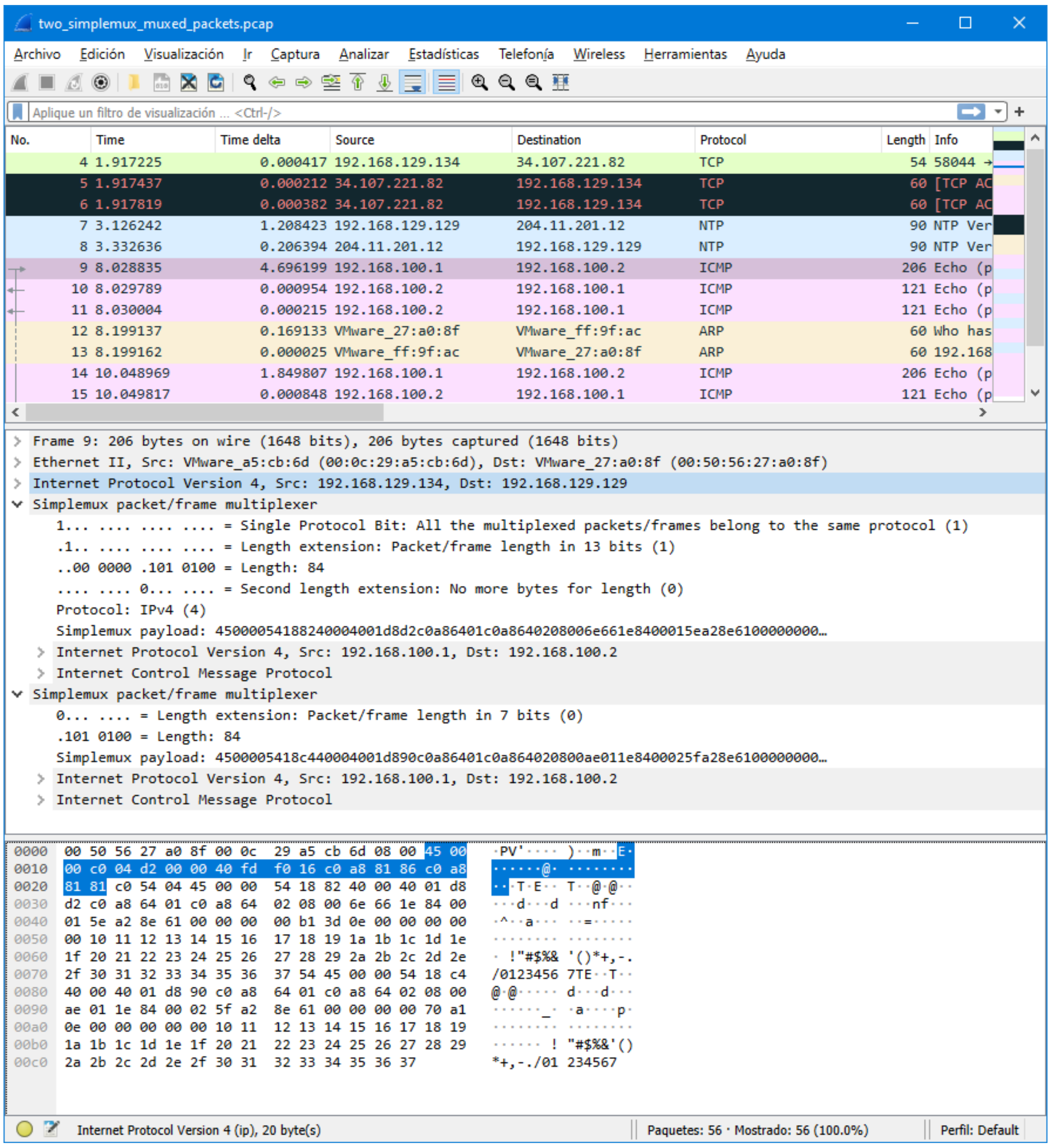
These are the fields:

- **Length Extension (`LXT`, one bit)** is `0` if the current byte is the last byte where the length of the first packet is included, and `1` in other case.

- **Length (`LEN`, 7, 14, 21, etc. bits).** This is the length of the multiplexed packet (in bytes), not including the *Length* field. If the length of the multiplexed packet is less than 128 bytes (less than or equal to 127 bytes), the first `LXT` is set to `0` and the 7 bits of the length field are the length of the multiplexed packet. If the length of the multiplexed packet is equal or greater than 128 bytes, additional bytes are added. The first bit of each of the added bytes is the `LXT`. If `LXT` is set to `1`, it means that there is an additional byte for expressing the length. This allows to multiplex packets of any length.

- **Protocol (8 bits)** is the *Protocol* field of the multiplexed packet, according to IANA "Assigned Internet Protocol Numbers". It is only included when the `SPB` of the First Multiplexing header is `0`.

This is a Wireshark screenshot showing two multiplexed IP packets (tun *mode*), over IP protocol (network *mode*):


two_simplemux_muxed_packets.pcap
Archivo Edición Visualización Ir Captura Analizar Estadísticas Telefonía Wireless Herramientas Ayuda
Aplique un filtro de visualización … <Ctrl-/>

```
No.   Time       Time delta Source           Destination      Protocol  Length Info
   4 1.917225   0.000417 192.168.129.134 34.107.221.82     TCP        54 58044 →
   5 1.917437   0.000212 34.107.221.82   192.168.129.134   TCP        60 [TCP AC
   6 1.917819   0.000382 34.107.221.82   192.168.129.134   TCP        60 [TCP AC
   7 3.126242   1.208423 192.168.129.129 204.11.201.12     NTP        90 NTP Ver
   8 3.332636   0.206394 204.11.201.12   192.168.129.129   NTP        90 NTP Ver
   9 8.028835   4.696199 192.168.100.1   192.168.100.2     ICMP      206 Echo (p
  10 8.029789   0.000954 192.168.100.2   192.168.100.1     ICMP      121 Echo (p
  11 8.030004   0.000215 192.168.100.2   192.168.100.1     ICMP      121 Echo (p
  12 8.199137   0.169133 VMware_27:a0:8f VMware_ff:9f:ac   ARP        60 Who has
  13 8.199162   0.000025 VMware_ff:9f:ac VMware_27:a0:8f   ARP        60 192.168
  14 10.048969  1.849807 192.168.100.1   192.168.100.2     ICMP      206 Echo (p
  15 10.049817  0.000848 192.168.100.2   192.168.100.1     ICMP      121 Echo (p
```

```
> Frame 9: 206 bytes on wire (1648 bits), 206 bytes captured (1648 bits)
> Ethernet II, Src: VMware_a5:cb:6d (00:0c:29:a5:cb:6d), Dst: VMware_27:a0:8f (00:50:56:27:a0:8f)
> Internet Protocol Version 4, Src: 192.168.129.134, Dst: 192.168.129.129
v Simplemux packet/frame multiplexer
    1... .... .... .... = Single Protocol Bit: All the multiplexed packets/frames belong to the same protocol (1)
    .1.. .... .... .... = Length extension: Packet/frame length in 13 bits (1)
    ..00 0000 .101 0100 = Length: 84
    .... .... 0... .... = Second length extension: No more bytes for length (0)
    Protocol: IPv4 (4)
    Simplemux payload: 4500005418824000401d8d2c0a86401c0a8640208006e661e8400015ea28e6100000000…
  > Internet Protocol Version 4, Src: 192.168.100.1, Dst: 192.168.100.2
  > Internet Control Message Protocol
v Simplemux packet/frame multiplexer
    0... .... = Length extension: Packet/frame length in 7 bits (0)
    .101 0100 = Length: 84
    Simplemux payload: 4500005418c440004001d890c0a86401c0a864020800ae011e8400025fa28e6100000000…
  > Internet Protocol Version 4, Src: 192.168.100.1, Dst: 192.168.100.2
  > Internet Control Message Protocol
```

```
0000  00 50 56 27 a0 8f 00 0c  29 a5 cb 6d 08 00 45 00   ·PV'···· )··m··E·
0010  00 c0 04 d2 00 00 40 fd  f0 16 c0 a8 81 86 c0 a8   ······@· ········
0020  81 81 c0 54 04 45 00 00  54 18 82 40 00 40 01 d8   ···T·E·· T··@·@··
0030  d2 c0 a8 64 01 c0 a8 64  02 08 00 6e 66 1e 84 00   ···d···d ···nf···
0040  01 5e a2 8e 61 00 00 00  00 b1 3d 0e 00 00 00 00   ·^··a··· ··=·····
0050  00 10 11 12 13 14 15 16  17 18 19 1a 1b 1c 1d 1e   ········ ········
0060  1f 20 21 22 23 24 25 26  27 28 29 2a 2b 2c 2d 2e   · !"#$%& '()*+,-.
0070  2f 30 31 32 33 34 35 36  37 54 45 00 00 54 18 c4   /0123456 7TE··T··
0080  40 00 40 01 d8 90 c0 a8  64 01 c0 a8 64 02 08 00   @·@····· d···d···
0090  ae 01 1e 84 00 02 5f a2  8e 61 00 00 00 00 70 a1   ······_· ·a····p·
00a0  0e 00 00 00 00 00 10 11  12 13 14 15 16 17 18 19   ········ ········
00b0  1a 1b 1c 1d 1e 1f 20 21  22 23 24 25 26 27 28 29   ······ ! "#$%&'()
00c0  2a 2b 2c 2d 2e 2f 30 31  32 33 34 35 36 37         *+,-./01 234567
```

Internet Protocol Version 4 (ip), 20 byte(s) | Paquetes: 56 · Mostrado: 56 (100.0%) | Perfil: Default

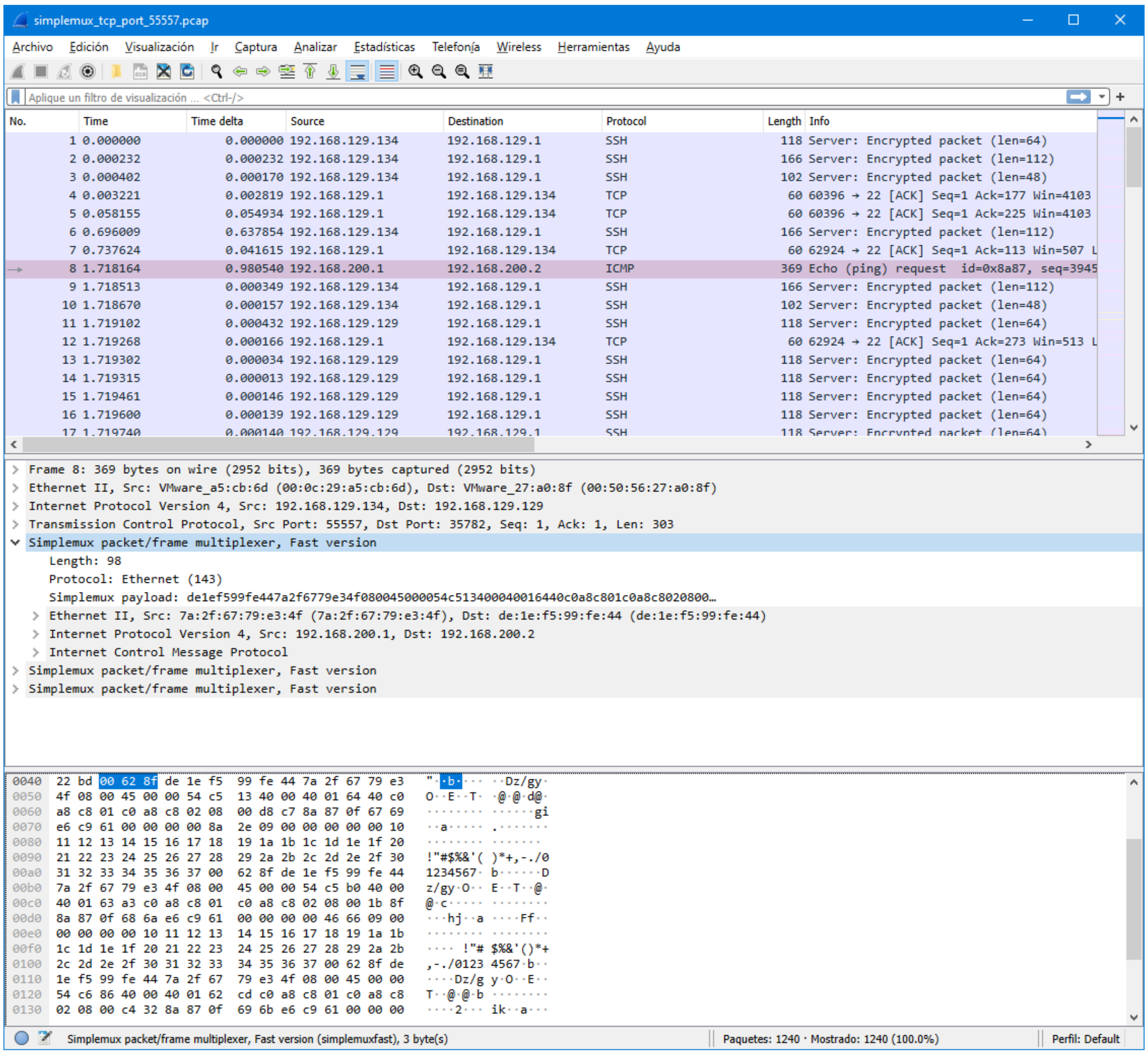
Fast flavor

All the separators have the same structure:

- **Length (`LEN`, 16 bits).** This is the length of the multiplexed packet (in bytes), not including the *Length* field.

- **Protocol (8 bits)** is the *Protocol* field of the multiplexed packet, according to IANA "Assigned Internet Protocol Numbers".

This is a Wireshark screenshot showing three multiplexed ethernet frames (tap *tunnel mode*), over TCP protocol (tcp *mode*):


simplemux_tcp_port_55557.pcap
Archivo Edición Visualización Ir Captura Analizar Estadísticas Telefonía Wireless Herramientas Ayuda
Aplique un filtro de visualización … <Ctrl-/>

```
No.   Time      Time delta Source          Destination     Protocol Length Info
    1 0.000000    0.000000 192.168.129.134 192.168.129.1   SSH      118 Server: Encrypted packet (len=64)
    2 0.000232    0.000232 192.168.129.134 192.168.129.1   SSH      166 Server: Encrypted packet (len=112)
    3 0.000402    0.000170 192.168.129.134 192.168.129.1   SSH      102 Server: Encrypted packet (len=48)
    4 0.003221    0.002819 192.168.129.1   192.168.129.134 TCP       60 60396 → 22 [ACK] Seq=1 Ack=177 Win=4103
    5 0.058155    0.054934 192.168.129.1   192.168.129.134 TCP       60 60396 → 22 [ACK] Seq=1 Ack=225 Win=4103
    6 0.696009    0.637854 192.168.129.134 192.168.129.1   SSH      166 Server: Encrypted packet (len=112)
    7 0.737624    0.041615 192.168.129.1   192.168.129.134 TCP       60 62924 → 22 [ACK] Seq=1 Ack=113 Win=507 L
    8 1.718164    0.980540 192.168.200.1   192.168.200.2   ICMP     369 Echo (ping) request  id=0x8a87, seq=3945
    9 1.718513    0.000349 192.168.129.134 192.168.129.1   SSH      166 Server: Encrypted packet (len=112)
   10 1.718670    0.000157 192.168.129.134 192.168.129.1   SSH      102 Server: Encrypted packet (len=48)
   11 1.719102    0.000432 192.168.129.129 192.168.129.1   SSH      118 Server: Encrypted packet (len=64)
   12 1.719268    0.000166 192.168.129.1   192.168.129.134 TCP       60 62924 → 22 [ACK] Seq=1 Ack=273 Win=513 L
   13 1.719302    0.000034 192.168.129.129 192.168.129.1   SSH      118 Server: Encrypted packet (len=64)
   14 1.719315    0.000013 192.168.129.129 192.168.129.1   SSH      118 Server: Encrypted packet (len=64)
   15 1.719461    0.000146 192.168.129.129 192.168.129.1   SSH      118 Server: Encrypted packet (len=64)
   16 1.719600    0.000139 192.168.129.129 192.168.129.1   SSH      118 Server: Encrypted packet (len=64)
   17 1.719740    0.000140 192.168.129.129 192.168.129.1   SSH      118 Server: Encrypted packet (len=64)
```

```
> Frame 8: 369 bytes on wire (2952 bits), 369 bytes captured (2952 bits)
> Ethernet II, Src: VMware_a5:cb:6d (00:0c:29:a5:cb:6d), Dst: VMware_27:a0:8f (00:50:56:27:a0:8f)
> Internet Protocol Version 4, Src: 192.168.129.134, Dst: 192.168.129.129
> Transmission Control Protocol, Src Port: 55557, Dst Port: 35782, Seq: 1, Ack: 1, Len: 303
v Simplemux packet/frame multiplexer, Fast version
    Length: 98
    Protocol: Ethernet (143)
    Simplemux payload: de1ef599fe447a2f6779e34f080045000054c513400040016440c0a8c801c0a8c8020800…
  > Ethernet II, Src: 7a:2f:67:79:e3:4f (7a:2f:67:79:e3:4f), Dst: de:1e:f5:99:fe:44 (de:1e:f5:99:fe:44)
  > Internet Protocol Version 4, Src: 192.168.200.1, Dst: 192.168.200.2
  > Internet Control Message Protocol
> Simplemux packet/frame multiplexer, Fast version
> Simplemux packet/frame multiplexer, Fast version
```

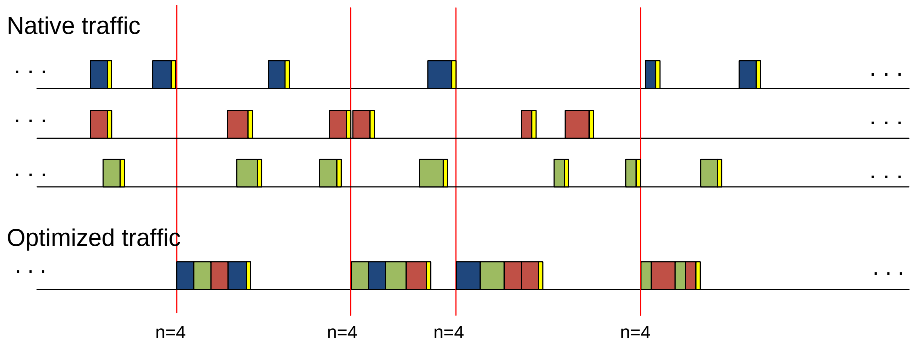
```
0040  22 bd 00 62 8f de 1e f5  99 fe 44 7a 2f 67 79 e3   "··b···· ··Dz/gy·
0050  4f 08 00 45 00 00 54 c5  13 40 00 40 01 64 40 c0   O··E··T· ·@·@·d@·
0060  a8 c8 01 c0 a8 c8 02 08  00 d8 c7 8a 87 0f 67 69   ········ ······gi
0070  e6 c9 61 00 00 00 00 8a  2e 09 00 00 00 00 00 10   ··a····· .·······
0080  11 12 13 14 15 16 17 18  19 1a 1b 1c 1d 1e 1f 20   ········ ········
0090  21 22 23 24 25 26 27 28  29 2a 2b 2c 2d 2e 2f 30   !"#$%&'( )*+,-./0
00a0  31 32 33 34 35 36 37 00  62 8f de 1e f5 99 fe 44   1234567· b······D
00b0  7a 2f 67 79 e3 4f 08 00  45 00 00 54 c5 b0 40 00   z/gy·O·· E··T··@·
00c0  40 01 63 a3 c0 a8 c8 01  c0 a8 c8 02 08 00 1b 8f   @·c····· ········
00d0  8a 87 0f 68 6a e6 c9 61  00 00 00 00 46 66 09 00   ···hj··a ····Ff··
00e0  00 00 00 00 10 11 12 13  14 15 16 17 18 19 1a 1b   ········ ········
00f0  1c 1d 1e 1f 20 21 22 23  24 25 26 27 28 29 2a 2b   ···· !"# $%&'()*+
0100  2c 2d 2e 2f 30 31 32 33  34 35 36 37 00 62 8f de   ,-./0123 4567·b··
0110  1e f5 99 fe 44 7a 2f 67  79 e3 4f 08 00 45 00 00   ····Dz/g y·O··E··
0120  54 c6 86 40 00 40 01 62  cd c0 a8 c8 01 c0 a8 c8   T··@·@·b ········
0130  02 08 00 c4 32 8a 87 0f  69 6b e6 c9 61 00 00 00   ····2··· ik··a···
```

Simplemux packet/frame multiplexer, Fast version (simplemuxfast), 3 byte(s) | Paquetes: 1240 · Mostrado: 1240 (100.0%) | Perfil: Default

Multiplexing policies

Four different conditions can be used and combined for triggering the sending of a multiplexed packet (in the figures, the triggering moment is expressed by red lines):

- **number of packets**: a number of packets have arrived at the multiplexer.



- **size**: two different options apply:

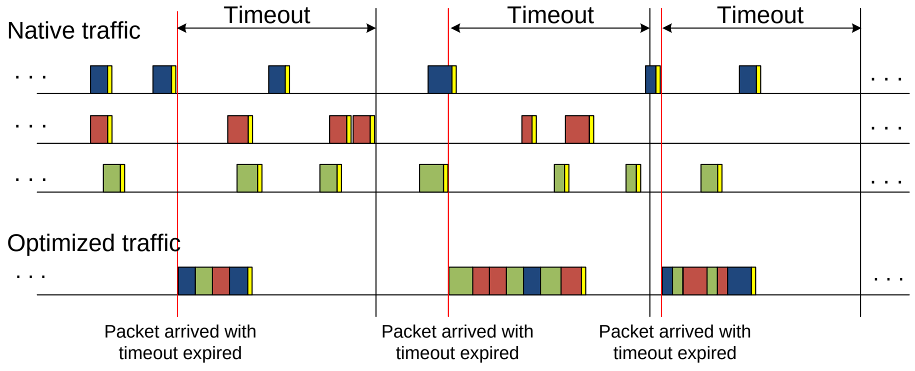
  - the size of the multiplexed packet has exceeded the size threshold specified by the user, but not the MTU. In this case, a packet is sent, and a new period is started with the buffer empty.
  - the size of the multiplexed packet has exceeded the MTU (and the size threshold consequently). In this case, a packet is sent without the last one. A new period is started, and the last arrived packet is stored for the next period.

  If you want to specify an MTU different from the one of the local interface, you can use the `-m` option. You may use other tools for getting the MTU of a network path. For example with the command[3]:

```
# tracepath 192.168.137.3 | grep Resume | cut -c 19-22
```

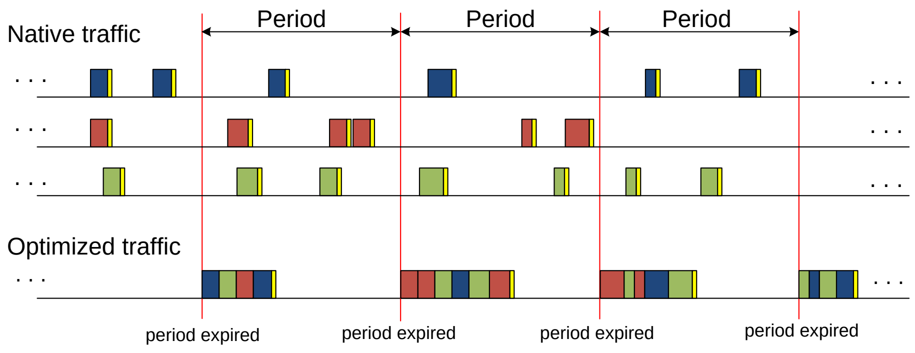
  you will obtain the MTU of the path to 192.168.137.3.

- **timeout**: a packet arrives, and a timeout since the last sending has expired.


Timeout
Timeout
Timeout
Native traffic
Optimized traffic
Packet arrived with timeout expired
Packet arrived with timeout expired
Packet arrived with timeout expired

- **period**: an active waiting is performed, and a multiplexed packet including all the packets arrived during a period is sent.

[3] http://packetlife.net/blog/2008/aug/18/path-mtu-discovery/



More than one condition can be set at the same time. Please note that if (timeout > period), then the timeout has no effect. Note that **only period policy guarantees an upper bound for the multiplexing delay.**

Simplemux is symmetric, i.e. both machines may act as ingress and egress simultaneously. However, different policies can be established at each of the optimizers, e.g. in one side you can send a multiplexed packet every two native ones, and in the other side you can set a timeout.

What will you find in this package

You will find two different source files:

`simplemux.c`	The complete version of **Simplemux**
`simplemux_multiplexing_delay.pl`	A perl script for reading the traces
`simplemux_throughput_pps.pl`	A perl script for reading the traces
`driveGnuPlotStreams.pl`	A perl script for creating real-time graphs
`simplemux.lua`	A dissector for **Simplemux**, *normal* flavor
`simplemuxfast.lua`	A dissector for **Simplemux**, *fast* flavor

Required tools

ROHC

First, you have to install rohc 1.7.0 from https://rohc-lib.org/. Download and uncompress the ROHC package.

```
cd rohc-1.7.0
./configure --prefix=/usr
make all
make check
make install
```

Compiling simplemux

```
gcc -o simplemux -g -Wall $(pkg-config rohc --cflags) simplemux.c $(pkg-config rohc --libs)
```

Usage of simplemux

```
./simplemux -i <ifacename> -e <ifacename> -c <peerIP> -M <'network' or 'udp' or 'tcpclient' or 'tcpserver'> [-T 'tun' or 'tap'] [-p <port>] [-d <debug_level>] [-r <ROHC_option>] [-n <num_mux_tun>] [-m <MTU>] [-b <num_bytes_threshold>] [-t <timeout (microsec)>] [-P <period (microsec)>] [-l <log file name>] [-L]
```

```
./simplemux -h

-i <ifacename>: Name of tun/tap interface to be used for capturing native packets (mandatory)
-e <ifacename>: Name of local interface which IP will be used for reception of muxed packets, i.e., the tunnel local end (mandatory)
-c <peerIP>: specify peer destination IP address, i.e. the tunnel remote end (mandatory)
-M <mode>: 'network' or 'udp' or 'tcpclient' or 'tcpserver' mode (mandatory)
-T <tunnel mode>: 'tun' (default) or 'tap' mode
-f: Fast mode (compression rate is lower, but it is faster). Compulsory for TCP mode
-p <port>: port to listen on, and to connect to (default 55555)
-d <debug_level>: Debug level. 0:no debug; 1:minimum debug; 2:medium debug; 3:maximum debug (incl. ROHC)
-r <ROHC_option>: 0:no ROHC; 1:Unidirectional; 2: Bidirectional Optimistic; 3: Bidirectional Reliable (not available yet)
-n <num_mux_tun>: number of packets received, to be sent to the network at the same time, default 1, max 100
-m <MTU>: Maximum Transmission Unit of the network path (by default the one of the local interface is taken)
-b <num_bytes_threshold>: size threshold (bytes) to trigger the departure of packets (default MTU-28 in transport mode and MTU-20 in network mode)
```

```
-t <timeout (microsec)>: timeout (in usec) to trigger the departure of packets
-P <period (microsec)>: period (in usec) to trigger the departure of packets. If ( timeout < period )
then the timeout has no effect
-l <log file name>: log file name. Use 'stdout' if you want the log data in standard output
-L: use default log file name (day and hour Y-m-d_H.M.S)
-h: prints this help text
```

If you have to use the same local interface more than once

It may happen that you have to create more than one tunnel using the same local interface. In that case, you may obtain a message `Is already in use`.

For example, if you run these two commands in the same machine, you may obtain this error message:

```
./simplemux -i tun0 -e wlan0 -M network -T tun -c 10.1.10.4
./simplemux -i tun1 -e wlan0 -M network -T tun -c 10.1.10.6
```

To avoid the problem, you can use an alias (see e.g. https://www.cyberciti.biz/faq/linux-creating-or-adding-new-network-alias-to-a-network-card-nic/). So you can do `ifconfig wlan0:0 x.y.z.t. up`, and then you can use `wlan0` in one case, and `wlan0:0` in the other.

Format of the Simplemux traces

Using the options `–l log file name` or `-L`, you can obtain a text file with traces. This is the format of these traces:

timestamp	event	type	size	sequence number	from/to	IP	port	number of packets	triggering event(s)
%"PRIu64"	text	text	%i	%lu	text	%s	%d	%i	text
microseconds	rec	native	packet size in bytes	sequence number	-	-	-	-	-
		muxed			from	ingress IP address	port (only in transport mode)		
		ROHC_feedback							
	sent	muxed			to	egress IP address	port (only in transport mode)	number	numpacket_limit size_limit timeout period MTU
		demuxed			-	-	-	-	-
	forward	native			from	ingress IP address	port	-	-
	error	bad_separator			-	-	-	-	-
		demux_bad_length			-	-	-	-	-
		decomp_failed			-	-	-	-	-
		comp_failed			-	-	-	-	-
	drop	too_long			to	egress IP address	port	number	-

	drop	no_ROHC_mode			-	-	-	-	-

- timestamp: it is in microseconds. It is obtained with the function GetTimeStamp().

- event and type:

  - rec: a packet has been received:

    - native: a native packet has arrived to the ingress optimizer.

    - muxed: a multiplexed packet has arrived to the egress optimizer.

    - ROHC_feedback: a ROHC feedback-only packet has been received from the decompressor. It only contains ROHC feedback information, so there is nothing to decompress

  - sent: a packet has been sent

    - muxed: the ingress optimizer has sent a multiplexed packet.

    - demuxed: the egress optimizer has demuxed a native packet and sent it to its destination.

  - forward: when a packet arrives to the egress with a port different to the one where the optimization is being deployed, it is just forwarded to the network.

  - error:

    - bad_separator: the Simplemux header before the packet is not well constructed.

    - demux_bad_length: the length of the packet expressed in the Simplemux header is excessive (the multiplexed packet would finish after the end of the global packet).

    - decomp_failed: ROHC decompression failed.

    - comp_failed: ROHC compression failed.

  - drop:

    - no_ROHC_mode: a ROHC packet has been received, but the decompressor is not in ROHC mode.

- size: it expresses (in bytes) the size of the packet. If it is a muxed one, it is the global size of the packet (including IP header). If it is a native or demuxed one, it is the size of the original (native) packet.

- sequence number: it is a sequence number generated internally by the program. Two different sequences are generated: one for received packets and other one for sent packets.

- IP: it is the IP address of the peer Simplemux optimizer.

- port: it is the destination port of the packet.

- number of packets: it is the number of packets included in a multiplexed bundle.

- triggering event(s): it is the cause (more than one may appear) of the triggering of the multiplexed bundle:

  - numpacket_limit: the limit of the number of packets has been reached.

  - size_limit: the maximum size has been reached.

  - timeout: a packet has arrived once the timeout had expired.

  - period: the period has expired.

  - MTU: the MTU has been reached.

Trace examples

In the ingress optimizer you may obtain:

```
1417693720928101    rec    native    63   1505
1417693720931540    rec    native    65   1506
1417693720931643    rec    native    52   1507
1417693720936101    rec    native    48   1508
1417693720936210    rec    native    53   1509
1417693720936286    rec    native    67   1510
1417693720937162    rec    native    57   1511
1417693720938081    sent   muxed     237  1511  to     192.168.137.4    55555  7   period
```

This means that 7 native packets (length 63, 65, … 57, and sequence numbers 1505 to 1511) have been received, and finally the period has expired, so they have been sent together to the egress Simplemux optimizer at 192.168.137.4, port 55555.

In the egress optimizer you may obtain:

```
1417693720922848    rec    muxed     237  210    from   192.168.0.5    55555
1417693720922983    sent   demuxed   63   210
1417693720923108    sent   demuxed   65   210
1417693720923186    sent   demuxed   52   210
1417693720923254    sent   demuxed   48   210
1417693720923330    sent   demuxed   53   210
1417693720923425    sent   demuxed   67   210
1417693720923545    sent   demuxed   57   210
```

This means that a multiplexed packet (sequence number 210) has been received from the ingress optimizer 192.168.0.5 with port 55555, and it has been demuxed, resulting into 7 different packets of lengths 63, 65, … 57.

Scripts for calculating compression statistics

This package includes the next Perl scripts:

Calculate throughput and packets per second

`simplemux_throughput_pps.pl`

It is able to calculate the throughput and the packet-per-second rate, from a Simplemux output trace. The result is in three columns:

```
tick_end_time(us)   throughput(bps)     packets_per_second
1000000             488144              763
2000000             490504 7            59
3000000             475576              749
4000000             483672              760
5000000             481784              758
6000000             487112              762
7000000             486824              760
8000000             488792              765
9000000             483528              761
10000000            486360              760
```

Usage:

```
$perl simplemux_throughput_pps.pl <trace file> <tick(us)> <event> <type> <peer IP> <port>
```

Examples:

```
# $ perl simplemux_throughput_pps.pl tracefile.txt 1000000 rec native all all
# $ perl simplemux_throughput_pps.pl log_simplemux 1000000 rec muxed all all
# $ perl simplemux_throughput_pps.pl log_simplemux 1000000 rec muxed 192.168.0.5 55555
# $ perl simplemux_throughput_pps.pl log_simplemux 1000000 sent demuxed
```

Calculate the multiplexing delay of each packet

Multiplexing delay is the time each packet is stopped in the multiplexer, i.e. the interval between its arrival as native packet and its departure inside a multiplexed packet.

```
simplemux_multiplexing_delay.pl
```

It is able to calculate the multiplexing delay of each packet, from a Simplemux output trace. The result is an output file in two columns:

```
packet_id    multiplexing_delay(us)
1      5279
2      1693
3      1202
4      507
5      10036
6      8471
7      6974
8      5588
9      1143
10     10435
11     8935
12     7522
13     5981
14     4520
15     3011
...
```

And other results are shown in stdout:

```
total native packets:     6661
Average multiplexing delay:     5222.47680528449 us
stdev of the multiplexing delay:3425.575192789 us
```

Usage:
```
$ perl simplemux_multiplexing_delay.pl <trace file> <output file>
```

Scripts for drawing the instantaneous throughput and pps

You can make Simplemux generate real-time graphs of the throughput and the amount of packets per second. For that aim, you have to use pipes in order to combine two perl scripts.

You will need to install the `gnuplot-x11` Linux package.

- Send the log of simplemux to `stdout`, using `–l stdout` option.

- Use the script `simplemux_throughput_pps_live.pl` to generate a summary every e.g. 10 ms of packets coming from (or going to) 192.168.0.5 using port 55555.

The output is something like this:

```
0:492800
1:532000
2:700
3:700
0:492800
1:532000
2:700
3:700
```

Where each row represents a value of

0: native throughput

1: multiplexed throughput

2: native pps

3: multiplexed pps

- Use the script `driveGnuPlotStreams.pl`, by Andreas Bernauer[4] to represent different graphs.

Some examples:

The next command presents two windows, each one with two graphs of 300 samples width, titled "native", "muxed", "ppsnat" and "ppsmux"

```
$ ./simplemux -i tun0 -e eth0 -c 192.168.0.5 -M udp -T tun -d 0 -r 2 -l stdout | perl
simplemux_throughput_pps_live.pl 10000 192.168.0.5 55555 |
perl ./driveGnuPlotStreams.pl 4 2 300 300 0 1000000 0 200 500x300+0+0 500x300+0+0 'native' 'muxed'
'ppsnat' 'ppsmux' 0 0 1 1
```

The next command presents one window with two graphs of 300 samples width, titled "native" and "muxed"

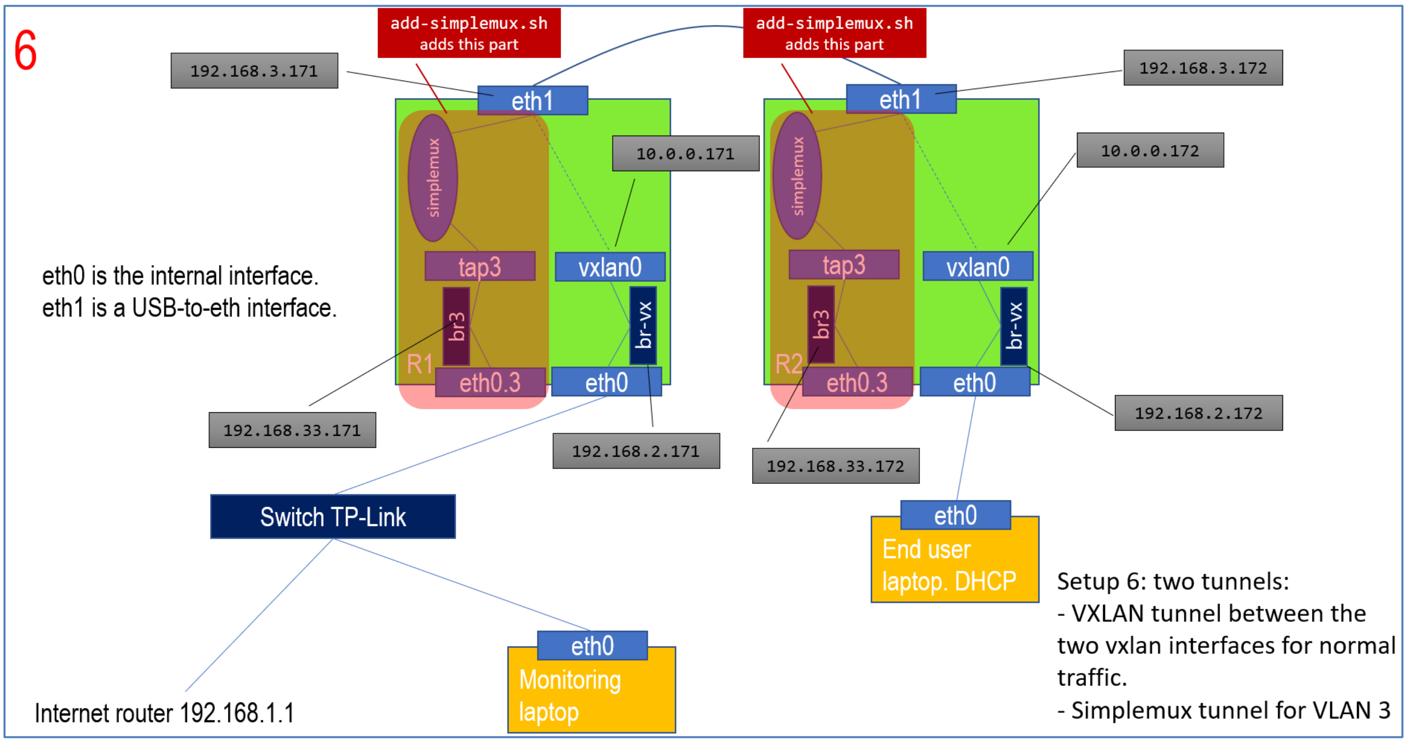
```
./simplemux -i tun0 -e eth0 -c 192.168.0.5 -M udp -T tun -d 0 -r 2 -l stdout | perl
simplemux_throughput_pps_live.pl 10000 192.168.0.5  | perl ./driveGnuPlotStreams.pl 2 1 300 0 1000000
500x300+0+0 'native' 'muxed' 0 0
```

[4] http://www.lysium.de/blog/index.php?/archives/234-Plotting-data-with-gnuplot-in-real-time.html

Scenario 1: two Raspberries

Using tc qdisc and tc filter, you can build this scheme between two machines:

Note: the “right” part of each machine includes a VXLAN tunnel, but this is not important here.



With this script, you create the “red” part in the left machine:

```
# raspberry 171

# add the interface eth0.3, part of VLAN 3
ip link add link eth0 name eth0.3 type vlan id 3

# add the interface tap3
ip tuntap add dev tap3 mode tap user root

# set the interface up
ip link set tap3 up

# add a bridge where we will put eth0.3 and tap3
ip link add br3 type bridge

# add tap3 interface to the bridge
ip link set tap3 master br3

# add an IP address to tap3 (probably not needed)
ifconfig tap3 10.0.3.171 netmask 255.255.255.0

# add an IP address to br3 (useful for testing that the setup is correct)
ifconfig br3 192.168.33.171

# add eth0.3 to the bridge
ip link set eth0.3 master br3

# set the bridge up
ip link set br3 up

# create a tunnel to the other side

# using UDP:
#./simplemux/simplemux -i tap3 -e eth1 -M udp -T tap -c 192.168.3.172 -d 2

# using TCP server:
```

```
./simplemux/simplemux -i tap3 -e eth1 -M tcpserver -T tap -c 192.168.3.172 -d 3 -n 1 -f
```

With this script you can add the red part in the right machine:

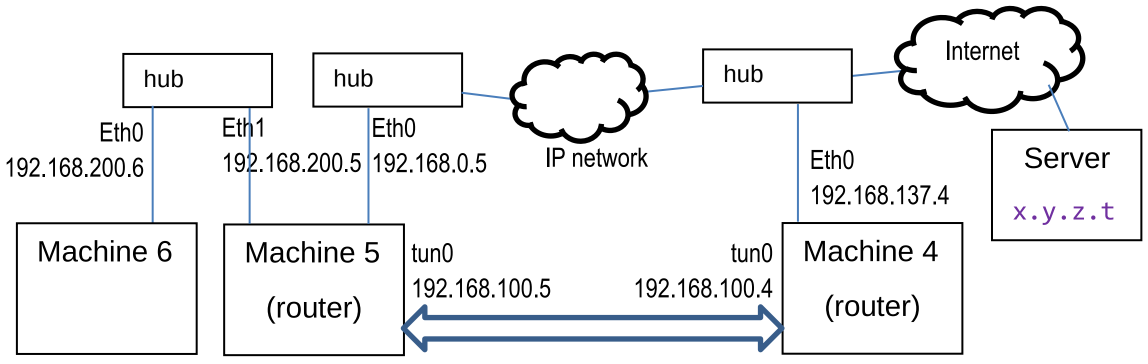
```
# raspberry 172
# add the interface eth0.3, part of VLAN 3
ip link add link eth0 name eth0.3 type vlan id 3

# add the interface tap3
ip tuntap add dev tap3 mode tap user root

# set the interface up
ip link set tap3 up

# add a bridge where we will put eth0.3 and tap3
ip link add br3 type bridge

# add tap3 interface to the bridge
ip link set tap3 master br3

# add an IP address to tap3 (probably not needed)
ifconfig tap3 10.0.3.172 netmask 255.255.255.0

# add an IP address to br3 (useful for testing that the setup is correct)
ifconfig br3 192.168.33.172

# add eth0.3 to the bridge
ip link set eth0.3 master br3

# set the bridge up
ip link set br3 up

# create a tunnel to the other side
# using UDP:
#./simplemux/simplemux -i tap3 -e eth1 -M udp -T tap -c 192.168.3.171 -d 2

# using TCP, client:
./simplemux/simplemux -i tap3 -e eth1 -M tcpclient -T tap -c 192.168.3.171 -d 2 -n 1 -f
```

You can now ping from Raspberry 1 to 192.168.33.172, and you will see if the tunnel works.

Scenario 2: three machines

This is the setup:

Machine 6 is the source. Machine 5 and Machine 4 are the two optimizers. Server `x.y.z.t` is the destination.



Create a tun interface in machine 4

`# ip tuntap add dev tun0 mode tun user root`[5]

(`#openvpn --mktun --dev tun0 --user root` will work in OpenWrt and also in other Linux distributions[6]).

`# ip link set tun0 up` (or `ifconfig tun0 up` for e.g. OpenWRT)

If you want to add an IP address to the `tun0` interface, use:

```
# ip addr add 192.168.100.4/24 dev tun0
```

If you do not need an IP address, you can omit the previous command.

Create a tun interface in machine 5

`# ip tuntap add dev tun0 mode tun user root`
(`#openvpn --mktun --dev tun0 --user root` will also work)

```
# ip link set tun0 up
# ip addr add 192.168.100.5/24 dev tun0
```

Establish the simplemux tunnel between machine 4 and machine 5

In machine4:

```
# ./simplemux -i tun0 -e eth0 -M udp -T tun -c 192.168.0.5
```

In machine5:

```
# ./simplemux -i tun0 -e eth0 –M udp -T tun -c 192.168.137.4
```

Now you can ping from machine 5 or machine 6, to machine 4:

```
$ ping 192.168.100.4
```

The ping arrives to the `tun0` interface of machine 5, goes to machine 4 through the tunnel and is returned to machine 6 through the tunnel.

[5] For removing the interface use `ip tuntap del dev tun0 mode tun`

[6] Openvpn is used to create and destroy tun/tap devices. In Debian you can install it this way: `# apt-get install openvpn`
In OpenWRT you will not be able to run `ip tuntap`, so you should install `openvpn` with: `# opkg install openvpn-nossl` (do `opkg update` before).

How to steer traffic from Machine 6 to server x.y.z.t through the tunnel

Steer the desired traffic to the tunnel using *ip rule* and *iptables*

The idea of **simplemux** is that it does not run at endpoints, but on some “optimizing” machines in the network. Therefore, you have to define policies to steer the flows of interest, in order to make them go through the TUN interface of the ingress (machine 5). This can be done with `ip rule` and `iptables`.

Following with the example:

In Machine 5, add a rule that makes the kernel route packets marked with `2` through `table 3`:

```
# ip rule add fwmark 2 table 3
```

In Machine 5, add a new route for `table 3`:

```
# ip route add default dev tun0 table 3 [7]
# ip route flush cache
```

If you show the routes of table 3

```
# ip route show table 3
```

Then you should obtain this:

```
        default via 192.168.100.5 dev tun0
```

And now you can use `iptables` in order to mark certain packets as `2` if they have a certain destination IP, or a port number.

Adding iptables entries:

All packets with destination IP address x.y.z.t

```
iptables -t mangle -A PREROUTING -p udp -d x.y.z.t -j MARK --set-mark 2
```

All packets with destination UDP port 8999

```
iptables -t mangle -A PREROUTING -p udp --dport 8999 -j MARK --set-mark 2
```

All packets with destination TCP port 44172

```
iptables -t mangle -A PREROUTING -p tcp --dport 44172 -j MARK --set-mark 2
```

Remove the table rule

```
iptables -t mangle -D PREROUTING -p tcp --dport 44172 -j MARK --set-mark 2
```

Show the table

```
iptables -t mangle –L
```

Examples implementing different policies

Set a period of 50 ms

```
./simplemux -i tun0 -e eth0 –M udp -T tun -c 192.168.0.5 –P 50000
```

Send a multiplexed packet every 2 packets, use ROHC Bidirectional Optimistic

```
./simplemux -i tun0 -e eth0 –M udp -T tun -c 192.168.0.5–n 2 –r 2
```

Send a multiplexed packet if the size of the multiplexed bundle is 400 bytes

```
./simplemux -i tun0 -e eth0 –M udp -T tun -c 192.168.0.5 –b 400
```

[7] If you have set an IP address in the `tun0` interface, this command should also work:

```
# ip route add default via 192.168.100.5 table 3
```

Send a timeout of 50ms, and a period of 100 ms (to set an upper bound on the added delay), use ROHC Unidirectional

```
./simplemux -i tun0 -e eth0 –M udp -T tun -c 192.168.0.5 –t 50000 –P 100000 –r 1
```

Credits

The author of **Simplemux** is Jose Saldana. **Simplemux** has been written for research purposes, so if you find it useful, I would appreciate that you send a message sharing your experiences, and your improvement suggestions.

Thanks to Didier Barvaux for his ROHC implementation.

Thanks to Davide Brini for his simpletun program.

If you have some improvement suggestions, do not hesitate to contact me.

The extension from Simplemux to Simplemux (October 2021) has been done by Jose Saldana, working at CIRCE Foundation, as a part of the H2020 FARCROSS project. This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 864274.