Simplemux Readme file

About simplemux

Simplemux is a traffic optimization tool written in C for Linux. It compresses headers using ROHC (RFC 3095), and multiplexes these header-compressed packets between a pair of machines (called optimizers). The multiplexed bundle is sent in an IP/UDP packet, as shown in Fig. 1. This may result on significant bandwidth savings and pps reductions for small-packet flows (e.g. VoIP, online games).

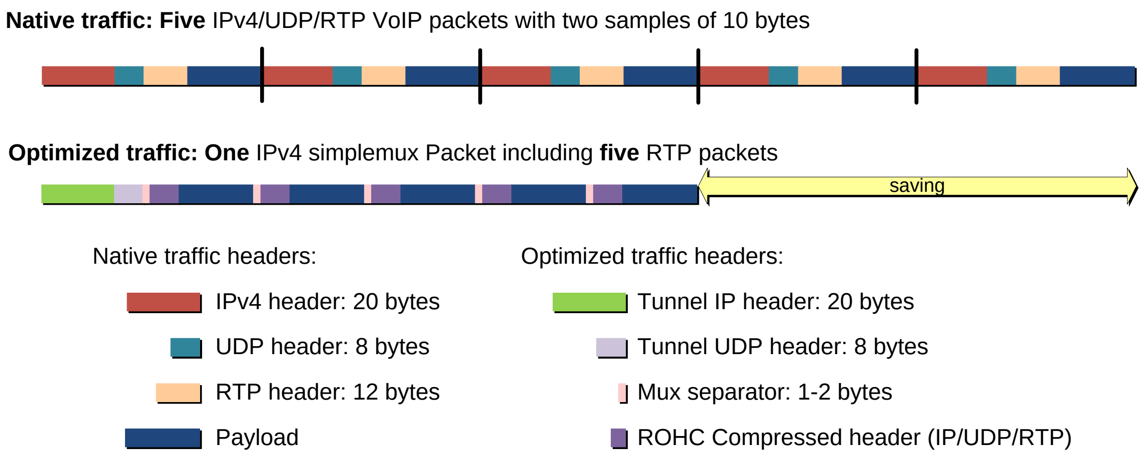


Fig. 1. Optimization of five RTP packets of VoIP

Simplemux is designed to optimize together a number of flows sharing a common network path or segment (Fig. 2). Optimization in the end host is (in principle) not useful, since a number of real-time flows departing from the same host are unusual.

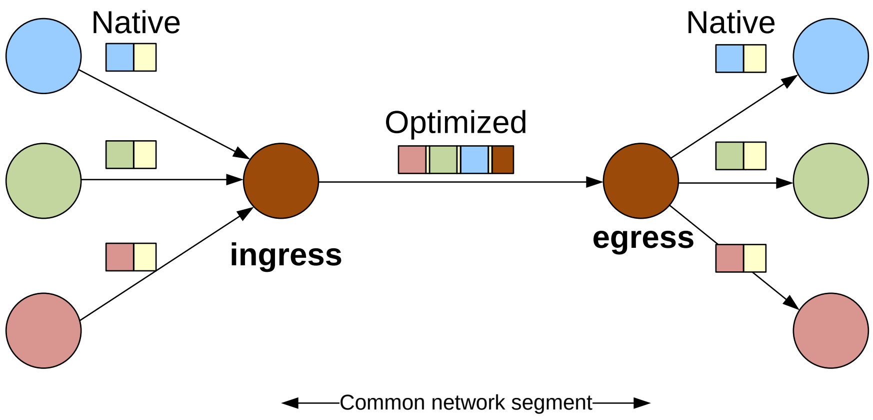


Fig. 2. Scheme of the optimization

Simplemux can be seen as a naïve implementation of TCM[1], a protocol combining **T**unneling, **C**ompressing and **M**ultiplexing for 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.

Simplemux uses Linux TUN virtual interface (TAP may also work, but the original idea is to multiplex at network level).

Header compression

Simplemux uses an implementation of ROHC by Didier Barvaux (https://rohc-lib.org/).

Multiplexing

The **Mux separator** follows the idea of the PPPMux separator of RFC 3153, section 1.1 (see Fig. 3)

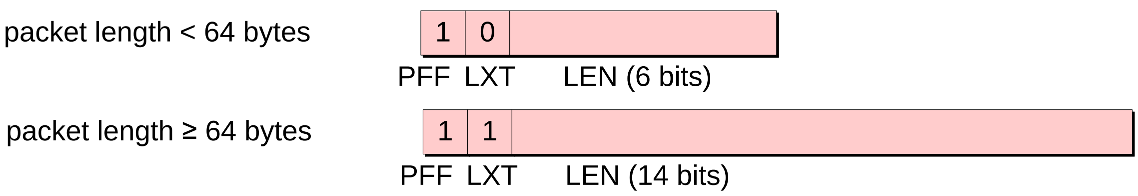


Fig. 3. Fields of the Mux separator

These are the three fields of the Mux separator:

1. **Protocol Field Flag (PFF):** It is the most significant bit of the first byte of each separator. It is always set to 1 (the reason for this is to keep similarity with PPPMux. Future work may modify this).

2. **Length Extension (LXT):** It is the second most significant bit of the first byte. This one bit field indicates whether the separator is one byte or two bytes long. If the LXT bit is 0, then the length field is one byte long (a PFF bit, a length extension bit, and 6 bits of muxed packet length).If the LXT bit is set to 1, then the muxed packet length field is two bytes long (a PFF bit, a length extension bit, and 14 bits of packet length).

3. **Muxed Packet Length (LEN):** This is the length of the muxed packet (header + payload) in bytes, not including the length field. If the length of the muxed packet is less than 64 bytes (less than or equal to 63 bytes), LXT is set to zero and the last six bits of the length field is the muxed packet length. If the length of the muxed packet is greater than 63 bytes, LXT is set to one and the last 14 bits of the length field is the length of the muxed packet. The maximum length of a muxed packet is 16,383 bytes.

Tunneling

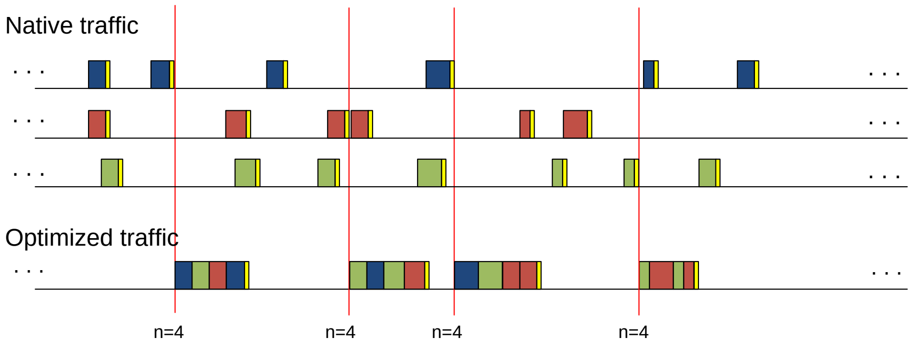
In **Simplemux,** tunneling is implemented in a very simple way: an IP/UDP header is added before the first **Mux separator.** By default, the destination UDP port is 55555.

[1] 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 policies

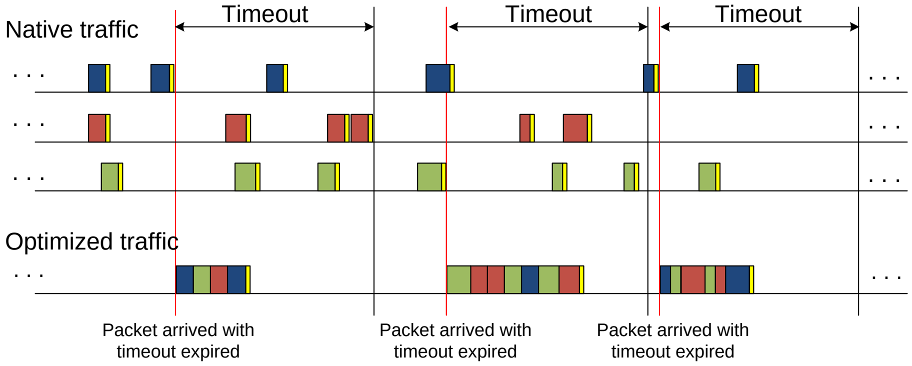
Four different conditions can be used and combined for triggering the sending of a multiplexed packet (in the figures, the triggering moment is expressed by red lines):

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

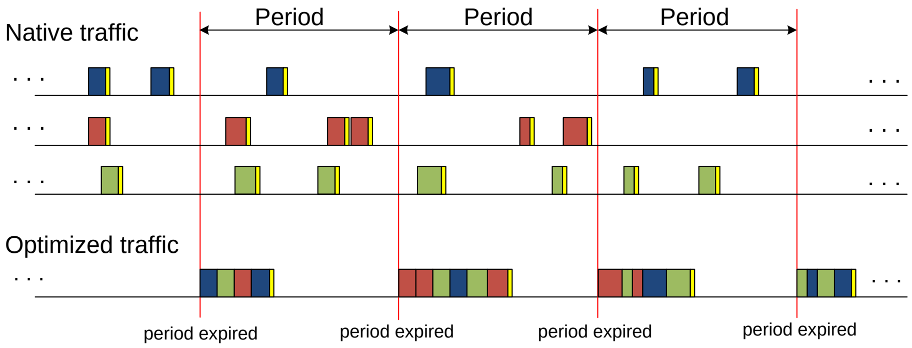


- **size**: the size of the multiplexed packet has reached a threshold.

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



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



More than one condition can be set at the same time. Please note that if (timeout < period), then the timeout has no effect.

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.

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

What will you find in this package

You will find two different source files:

Simplemux_vX.Y.c	The complete version of **simplemux**
simplemux-no-compress.c	A *light* version which does not compress headers

Required tools

ROHC (not required for simplemux-no-compress.c)

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
```

openvpn

openvpn: to create and destroy tun/tap devices. In Debian:

```
#apt-get install openvpn
```

Compiling simplemux

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

Usage of simplemux

```
./simplemux -i <ifacename> [-c <peerIP>] [-p <port>] [-u|-a] [-d
<debug_level>] [-r] [-n <num_mux_tap>] [-b <num_bytes_threshold>] [-t
<timeout (microsec)>] [-P <period (microsec)>] [-l <log file name>] [-
L]

-i <ifacename>: Name of tun/tap interface to use (mandatory)
-e <ifacename>: Name of local interface to use (mandatory)
-c <peerIP>: specify peer destination address (-d <peerIP>)
(mandatory)
-p <port>: port to listen on, and to connect to (default 55555)
-u|-a: use TUN (-u, default) or TAP (-a)
-d: outputs debug information while running. 0:no debug; 1:minimum
debug; 2:medium debug; 3:maximum debug (incl. ROHC)
-r: compresses and decompresses headers using ROHC
-n: number of packets received, to be sent to the network at the same
time, default 1, max 100
-b: size threshold (bytes) to trigger the departure of packets,
default 1472 (1500 - 28)
-t: timeout (in usec) to trigger the departure of packets
-P: period (in usec) to trigger the departure of packets. If ( timeout
< period ) then the timeout has no effect
-l: log file name
-L: use default log file name (day and hour Y-m-d_H.M.S)-h: prints
this help text
```

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:

<table>
<tr><th>timestamp</th><th>event</th><th>type</th><th>size</th><th>sequence number</th><th>from/to</th><th>IP</th><th>port</th><th>number of packets</th><th>triggering event(s)</th></tr>
<tr><td>%"PRIu64"</td><td>text</td><td>text</td><td>%i</td><td>%lu</td><td>text</td><td>%s</td><td>%d</td><td>%i</td><td>text</td></tr>
<tr><td rowspan="10">microseconds</td><td rowspan="3">rec</td><td>native</td><td rowspan="10">packet size in bytes</td><td rowspan="10">sequence number</td><td>-</td><td>-</td><td>-</td><td rowspan="3">-</td><td rowspan="3">-</td></tr>
<tr><td>muxed</td><td rowspan="2">from</td><td rowspan="2">ingress IP address</td><td rowspan="2">port</td></tr>
<tr><td>ROHC_feedback</td></tr>
<tr><td rowspan="2">sent</td><td>muxed</td><td>to</td><td>egress IP address</td><td>port</td><td>number</td><td>numpacket_limit
size_limit
timeout
period
MTU</td></tr>
<tr><td>demuxed</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td></tr>
<tr><td>forward</td><td>native</td><td>from</td><td>ingress IP address</td><td>port</td><td>-</td><td>-</td></tr>
<tr><td rowspan="4">error</td><td>bad_separator</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td></tr>
<tr><td>demux_bad_length</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td></tr>
<tr><td>decomp_failed</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td></tr>
<tr><td>comp_failed</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td></tr>
</table>

- `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 packet has been demultiplexed, but one of these options applies:

      a) the ROHC packet was a non-final segment, so at least another ROHC segment is required to be able to decompress the full ROHC packet

      b) the ROHC packet was a feedback-only packet, it contained only feedback information, so there was 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.

- `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 some 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             759
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

`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
```

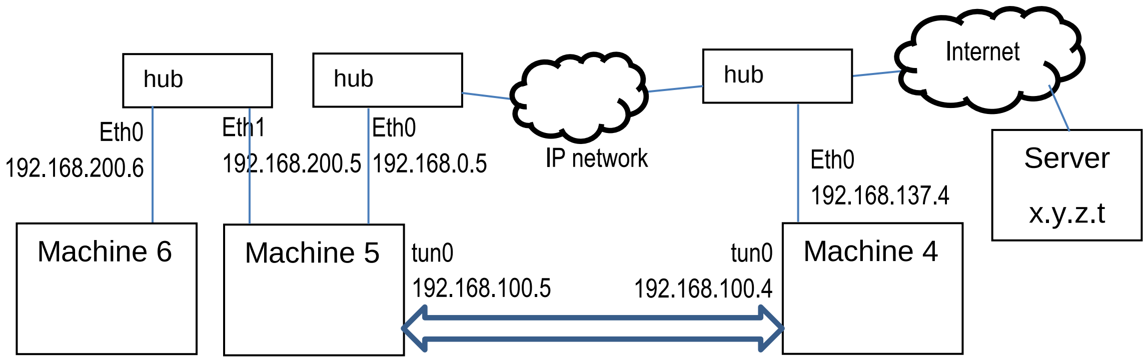
Usage:

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

Usage example with 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

```
openvpn --mktun --dev tun0 --user root
ip link set tun0 up
ip addr add 192.168.100.4/24 dev tun0
```

Create a tun interface in machine 5

```
openvpn --mktun --dev tun0 --user root

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 -c 192.168.0.5 -r
```

In machine5:

```
# ./simplemux -i tun0 -e eth0 -c 192.168.137.4 -r
```

Now you can ping from machine6 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.

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

The idea of **simplemux** is that it does not run on endpoints, but on some “optimizing” machines in the network. Soy you have to define policies in order 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 `iprules` 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 via 192.168.100.5 table 3
# ip route flush cache
```

If you show the routes of table 3, you should see this:

```
# ip route show table 3
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.

Examples:

Destination IP address x.y.z.t

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

Destination UDP port 8999

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

Destination TCP port 44172

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

Other examples implementing different policies

Set a period of 50 ms

```
./simplemux -i tun0 -e eth0 -c 192.168.0.5 -r -P 50000
```

Send a multiplexed packet every 2 packets

```
./simplemux -i tun0 -e eth0 -c 192.168.0.5 -r -n 2
```

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

```
./simplemux -i tun0 -e eth0 -c 192.168.0.5 -r -b 400
```

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

```
./simplemux -i tun0 -e eth0 -c 192.168.0.5 -r -t 50000 -P 100000
```

Credits

The author of **simplemux** is Jose Saldana (jsaldana at unizar.es). **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.

The software is released under the **GNU General Public License**, Version 3, 29 June 2007.

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.

http://diec.unizar.es/~jsaldana/