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).

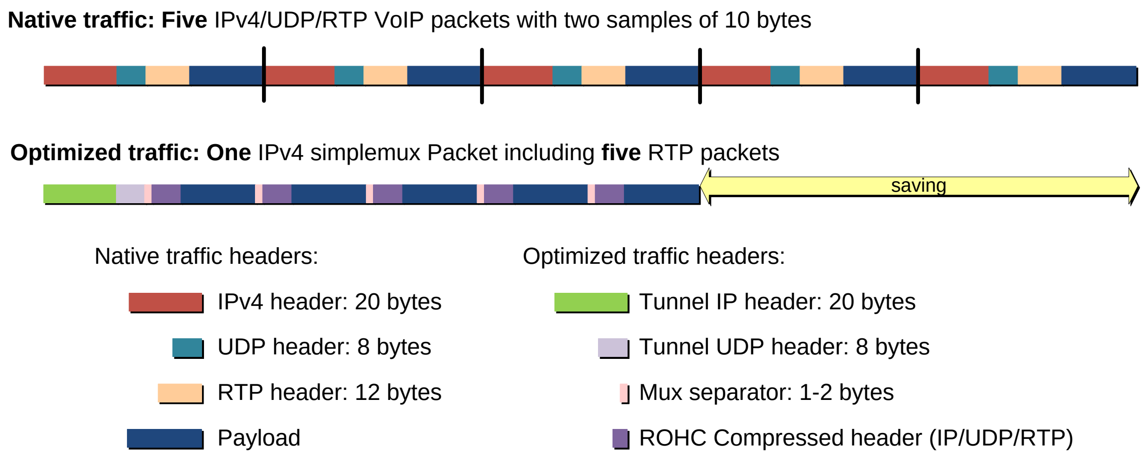

Native traffic: Five IPv4/UDP/RTP VoIP packets with two samples of 10 bytes
Optimized traffic: One IPv4 simplemux Packet including five RTP packets
saving
Native traffic headers:
IPv4 header: 20 bytes
UDP header: 8 bytes
RTP header: 12 bytes
Payload
Optimized traffic headers:
Tunnel IP header: 20 bytes
Tunnel UDP header: 8 bytes
Mux separator: 1-2 bytes
ROHC Compressed header (IP/UDP/RTP)

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.

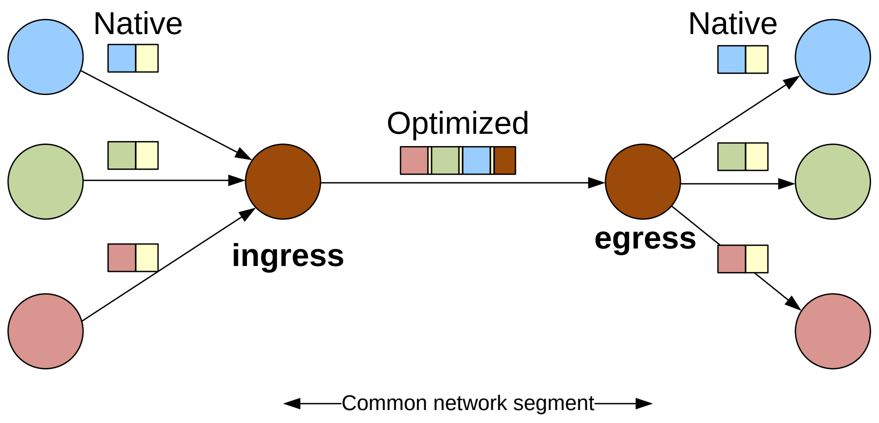


Fig. 2. Scheme of the optimization

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)

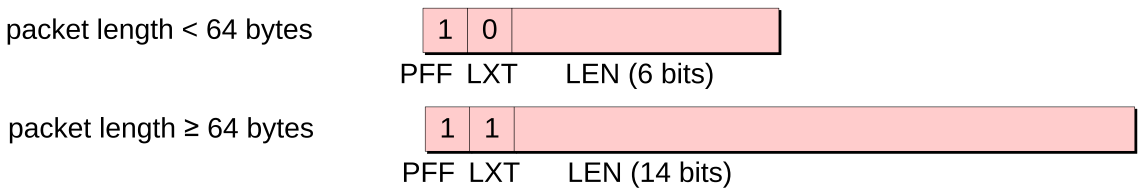


Fig. 3. Fields of the Mux separator

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

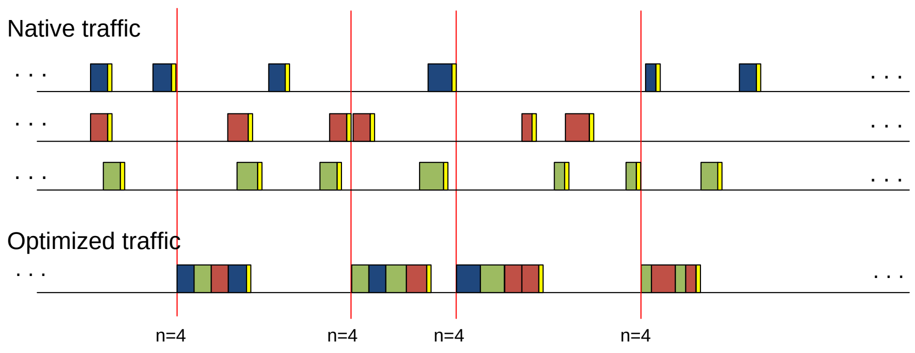
In **simplemux,** tunneling is implemented in a very simple way: an IP/UDP header is added before the first **Mux separator.** By default, the destination UDP port is 55555.

[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

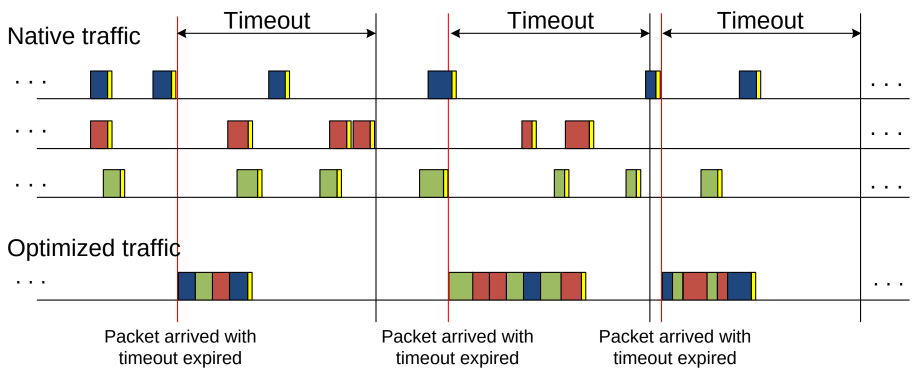
Four different conditions can be used and combined for triggering the sending of a multiplexed packet (in the figures, the triggering moment is expressed by red lines):

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

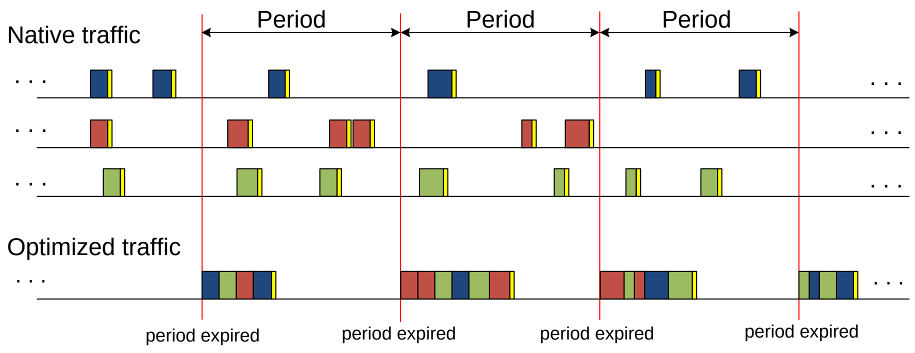


- **size**: the size of the multiplexed packet has reached a threshold.

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


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.



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.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.c
$(pkg-config rohc --libs )
```

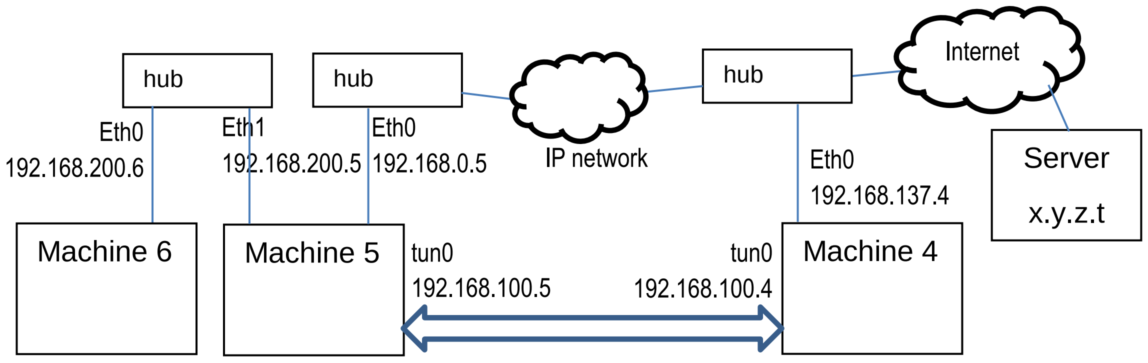
Usage of simplemux

```
./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)>]

./simplemux -h

-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 1400
-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
-h: prints this help text
```

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/