Virtual Router Manual and API Description

Florian Baumgartner

IAM-01-001

November 2001

Contents

Chapter 1

Source Installation and System Configuration

1.1 Obtaining the Sources

However the sources where obtained by CVS or as a tar archive. After check out (or extraction) the sources are assumed to be located in a directory called microvar. It is strongly recommended to install the sources as a normal user and *not as the superuser*.

1.2 Compiling and Installing the Base System

Even if most of the Virtual Router system builds automatically, the Softlink device has to be compiled and installed separately due to its dependencies of the Linux kernel. (see Section 1.3).
To compile the sources enter the microvar directory and type `make`. The Makefile processes all subdirectories and builds the appropriate binaries. The Makefile in the microvar directory contains also the destination the Virtual Router system will finally be installed to. This location should be adapted before installing the binaries. Adjust the `PREFIX` to yur needs. After that a `make install` will install the binaries and set up all required files.

1.3 Setting up the Softlink Devices

The Virtual Router allows to be integrated into real networks. For this connection the Softlink kernel module is required. If you do not want to connect the virtual network to a physical one, you can skip this section. Even if the Virtual Router itself is a normal user space program, the Softlink Devices are Kernel modules and may require some adjustments to the Linux kernel sources. You will also need superuser rights to install the Softlink kernel modules.

1.3.1 Configuring the Linux Kernel Source

The Softlink module crucially depends on a correct set up of the Linux kernel source tree and the according header files. This Section describes the appropriate setup of the Linux kernel sources to allow a compilation of the Softlink Device modules. The Linux kernel source is assumed to be installed under /usr/src/linux. To compile the Softlink module it has to made sure, that the compiler uses the corresponding Linux kernel header files, usually located under /usr/include. This is while the linux and the asm directory located there have to be softlinked (man ln) with the appropriate directories in /usr/src/linux/include[1].

1.3.2 Compiling the Softlink Device

After the header files were adjusted correctly, the Softlink Device can be compiled by entering the microvar/softlink directory and typing `make`.

1.3.3 Loading the Softlink Module

Now the Softlink module can be loaded into the kernel. You have to be root and use the `modprobe` or the `insmod` commands. It should be mentioned here, that your only have to load the kernel module once, even if you want multiple Softlink devices. Per default the module is configured to provide eight devices. These setting might be changed by adjusting the line `#define NOF 8` in the file softlink.h. additionally the correct device entries in the /dev directory have to be created.

1.3.4 Creating the Device files

The Softlink devices maps a network device to character device /dev/sol0,...,/dev/sol$N$ with the major number 63 and the minor numbers $0, ..., N$ with N is the number of configured Softlink devices. (see also man mknod for information about the creation of device entries). The script microvar/softlink/makedevices will create the device files for you. Edit the script file to adjust the permissions of the device file.
It is recommended to change the owner and the group of the /dev/sol devices from root access only to a standard user or a user group so the devices can be accessed by normal users without root permissions. It is important for the system to work properly, that both read and write access is enables on the devices.

1.3.5 Setting up Interfaces and Routers

Once the softlink module is loaded into the kernel, the Linux network devices can be configured. They have the name sol0 to sol7 in the default setup. The Linux interfaces can simply be added by the Linux `ifconfig` command. (see man ifconfig). For example the command (you have to be root!)

```
ifconfig sol0 10.1.1.1 netmask 255.255.255.0 broadcast 10.1.1.255
```

will set up and configure the network interface `sol0`. Finally your machines interfaces may look like this listing produces by the Linux `ifconfig` command.

[1]This requires /usr/src/linux to contain a fully configured kernel tree (e.g. at least one time used to compile a kernel for this system)

```
eth0      Link encap:Ethernet  HWaddr 00:30:48:21:35:A4
          inet addr:130.92.70.30  Bcast:130.92.70.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1667863 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1475811 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          Interrupt:31 Base address:0x2000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:3924  Metric:1
          RX packets:536509 errors:0 dropped:0 overruns:0 frame:0
          TX packets:536509 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0

sol0      Link encap:Ethernet  HWaddr 00:53:4F:46:54:4C
          inet addr:10.1.1.1  Bcast:10.1.1.255  Mask:255.255.255.0
          UP BROADCAST RUNNING NOARP MULTICAST  MTU:1500  Metric:1
          RX packets:92 errors:0 dropped:0 overruns:0 frame:0
          TX packets:170 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
```

`eth0` is the normal Linux network interface

`lo` is the standard loop back device

`sol0` is the softlink device. Packets send to this network device may be read from the device file /dev/sol0

To route packets to the virtual topology two routing table entries have to be created. First of all the entry regarding the `10.1.1.0` subnet is automatically created by adding the `sol0` device with the `ifconfig` command. The entry dealing with the `172.1.1.0` subnet have to be created because of the address translation system later described in Section 3.3. The following lines show the output of the `route` program on the Linux host.

```
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
172.1.1.0       *               255.255.255.0   U     0      0        0 sol0
10.1.1.0        *               255.255.255.0   U     0      0        0 sol0
localnet        *               255.255.255.0   U     0      0        0 eth0
default         enos.cnds.unibe 0.0.0.0         UG    0      0        0 eth0
```

1.4 The Virtual Router Configuration File

The Virtual Router can read a configuration file `vr.conf` during startup. These configuration file allows to modify the basic behavior of the program. The configuration file is not thought to allow the configuration of the whole system, but only of a few basic aspects, as the access mechanisms, which will be used for the later configuration.

`binary_shell:` A yes or a no defines, whether the internal shell switches to raw binary mode and expects binary data instead of human readable commands. In binary mode, the output will of course also be binary (see Chapter 5). This is useful if other configuration tools shall be used instead of the internal shell. Ths binary mode is not available if the shell is used in console mode.

`hostname_of_shell:` This allows to tell the shell, to which host address it should bind the socket. If this entry is commented out, the Virtual Router will perform the usual `gethostname()` call to get the address of the host. This parameter is useful to bind to a specific address as to `localhost` if no network system is defined.

`port_of_shell:` This is the port the shell listens to for a TCP connection. If this parameter is left empty or set to 0 the shell works in console mode.

`lob_extension:` This is the extension the Virtual Router expects for LOBs. As default this is .vr.

`lob_path:` the path, the Virtual Router looks for the LOBs

`shell_password:` if defined the Virtual Router will wait for an password before allowing shell access.

`shell_password_tries:` defines how often the user is allowed to type a wrong password until the shell kicks him out.

`restrict_source:` if an IP address or a host name is set here, the Virtual Router will only accept TCP connections from this address. This is very useful to restrict shell access to a specific machine. (to be implemented)

`if_bandwidth:` the default bandwidth for Virtual Router interfaces in Megabit per second. The value defined here is used for the first initialisation of the interface. When not defined 1.0 Mbps is assumed

`if_bucketsize:` the default bucketsize for Virtual Router interfaces in bytes used during first initialisation. Is set to 2048 byted, if not defined.

`tbf_bandwidth:` the default bandwidth for new instantiated TBFs in the Virtual Routers queueing system in Megabit per second (see also 5.7.7). If not defined, the Virtual Router initialises it's tbfs with 2.0 Mbps.

`tbf_bucketsize:` the default bucketsize for new instantiated TBFs in the Virtual Routers queueing system in bytes. (see also 5.7.7). If not defined, the Virtual Router initialises it's tbfs with 2048 bytes.

`dpt_queuelen:` the default queuelen the Virtual Router uses for the droptail queues in the queueing system (see also 5.7.7) If not defined, the Virtual Router initialises its droptail queues with a capacity of 16 packets.

`softlink_basename:` the basic string with is used to create the filename of the softlink devices. The complete filename is this string plus the number of the device. The default for this string is /dev/sol causing the devicefile name /dev/sol0, /dev/sol1, ...

`fifo_basename:` this is the string used for accessing the FIFO files. For more information about FIFO files see section 3.1. As defaul the fifo files are searched under PREFIX/fifos. (see Section 1.2).

`iptrans_enable:` this switch enables or disables iptranslation for the complete router. Per default ip translation is off (see Section 3.3).

Chapter 2

Testing the Installation

2.1 One Router only

This simple example checks whether the Virtual Router itself is operational and if the Loadable Objects can be found. This is done by simply starting the router, creating an interface and sending echo requests to this interface.

```
MicroVar Shell
> ifconfig add if0 10.1.1.1
name:                       if0
ip-address:            10.1.1.1
netmask:          255.255.255.0
broadcast:           10.1.1.255
bandwidth(bps):         1000000   bucketsize(bytes):           2048
drops:                        0   errors:                         0
rx:                           0   tx:                             0
connection                 none
> load ping 10.1.1.1
object 'ping' loaded, id is 0, mode is STAY
> 64 bytes from 10.1.1.1: icmp_seq=0 icmp_id=42 ttl=64 time= 0.0 ms
64 bytes from 10.1.1.1: icmp_seq=1 icmp_id=42 ttl=64 time= 0.0 ms
64 bytes from 10.1.1.1: icmp_seq=2 icmp_id=42 ttl=64 time= 0.0 ms
3 packets transmitted, 3 packets received
round-trip min/avg/max =  0.0/ 0.0/ 0.0 ms
```

2.2 Two Virtual Routers

In the second configuration two routers are set up and connected with fifo links. The routers get the ip addresses 10.1.1.1 and 10.1.1.2. Finally a set of pings is send from one router to the other.

Router A

```
MicroVar Shell
> ifconfig add if0 10.1.1.1
name:                       if0
ip-address:            10.1.1.1
netmask:          255.255.255.0
broadcast:           10.1.1.255
```

```
bandwidth(bps):        1000000   bucketsize(bytes):           2048
drops:                       0   errors:                         0
rx:                          0   tx:                             0
connection                none
> ifconfig if0 connect fifo 0 r
name:                      if0
ip-address:           10.1.1.1
netmask:         255.255.255.0
broadcast:          10.1.1.255
bandwidth(bps):        1000000   bucketsize(bytes):           2048
drops:                       0   errors:                         0
rx:                          0   tx:                             0
connection               local   id:                             2
end of cable                 r
> ifconfig if0 qs create droptail
Component created with id 1
> ifconfig if0 qs chain 0 1
components connected
> ifconfig if0 qs chain 1 0
components connected
> route add 10.1.1.0/24 if0
route 10.1.1.0/255.255.255.0 to if0 added
>
```

Router B

```
MicroVar Shell
> ifconfig add if0 10.1.1.2
name:                      if0
ip-address:           10.1.2.1
netmask:         255.255.255.0
broadcast:          10.1.1.255
bandwidth(bps):        1000000   bucketsize(bytes):           2048
drops:                       0   errors:                         0
rx:                          0   tx:                             0
connection                none
> ifconfig if0 connect fifo 0 l
name:                      if0
ip-address:           10.1.1.1
netmask:         255.255.255.0
broadcast:          10.1.1.255
bandwidth(bps):        1000000   bucketsize(bytes):           2048
drops:                       0   errors:                         0
rx:                          0   tx:                             0
connection               local   id:                             2
end of cable                 l
> ifconfig if0 qs create droptail
Component created with id 1
> ifconfig if0 qs chain 0 1
components connected
> ifconfig if0 qs chain 1 0
components connected
> route add 10.1.1.0/24 if0
route 10.1.1.0/255.255.255.0 to if0 added
> load ping 10.1.1.1
object ’ping’ loaded, id is 0, mode is STAY
64 bytes from 10.1.1.1: icmp_seq=0 icmp_id=42 ttl=63 time= 0.1 ms
```

```
64 bytes from 10.1.1.1: icmp_seq=1 icmp_id=42 ttl=63 time= 0.1 ms
64 bytes from 10.1.1.1: icmp_seq=2 icmp_id=42 ttl=63 time= 0.1 ms
3 packets transmitted, 3 packets received
round-trip min/avg/max =  0.1/ 0.1/ 0.1 ms
>
```

Chapter 3

General Aspects

3.1 Connections

A Virtual Router has to receive and to send packets. Usually, this is done over the attached communication channels.

connections to a softlink device: This type of connection is used to enable the communication between a Virtual Router and a real network. The interface is connected to a softlink device, provided by a Linux kernel module as described in Section 1.3. This allows the Virtual Router to exchange packets with a real network

connections between VRs via FIFO pipes: This connection type is used to connect two VRs running on the same computer. It allows to forward packets from one VR to another and is simply based on Unix pipes between two processes. Since Unix pipes are simplex, two pipes are used for each connection.

connections between VRs via UDP tunnels: Tunnelling based on UDP is used to exchange packets between VRs on different computers. IP packets to be forwarded to another VR are encapsulated within an UDP packet and sent to a specific port of the remote host. The opposite Virtual Router has to be configured to listen to this UDP port and read packets from there. Like any other connection it must be duplex, since the VR's interfaces on both sides of the connection have to receive and transmit UDP packets.

Received data is processed by an IP network address translation unit (NAT). This allows to force the routing of packets through an emulated topology by modifying the destination/source address pair within a VR. Therefore it is possible to set up large networks on a single computer. The address translation mechanism is powerful but complex. A more detailed description of the address translation feature will be given in Section 3.3.

3.2 Topology, Layout and VR Distribution

The idea of a Virtual Router is to emulate a single router, not an end system. Real end systems are meant to be used as traffic sources and sinks.

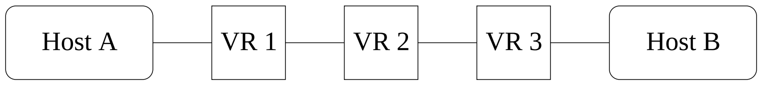


Figure 3.1: *Two end systems connected over three Virtual Routers*

The diagram above shows a typical simple set-up of two hosts (A,B) connected via three VRs (1,2,3), allowing to send traffic (UDP, TCP, ...) from host A to host B and vice versa. To establish a connection to a computer's network layer (see Section 3.1) a softlink device has to be used. This requires a VR running on the same computer the softlink device is installed. Nevertheless, several set-ups are possible.

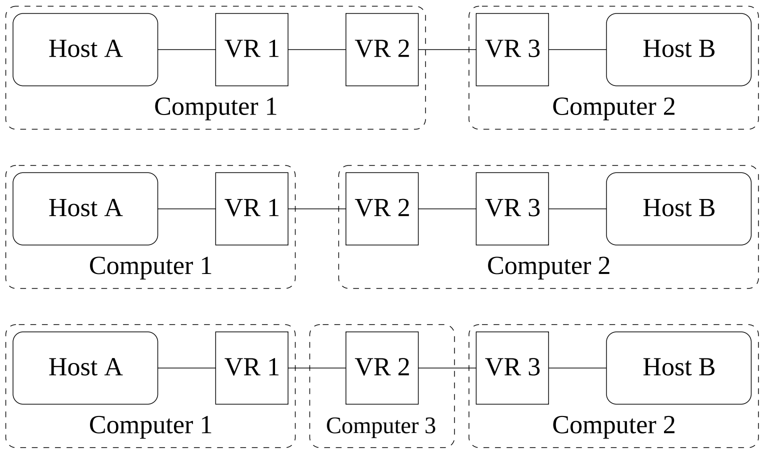


Figure 3.2: *Distributing VRs to end systems*

Diagram 3.2 shows how Virtual Routers may be distributed. On each computer, which has an interface to be connected to a VR via softlink device, a VR has to be started. VR 2 might be placed either on computer 1 or 2 or on an additional computer 3 between. A concrete set-up would depend on the available processing power of the computers and the bandwidth of the network the computers are connected with.

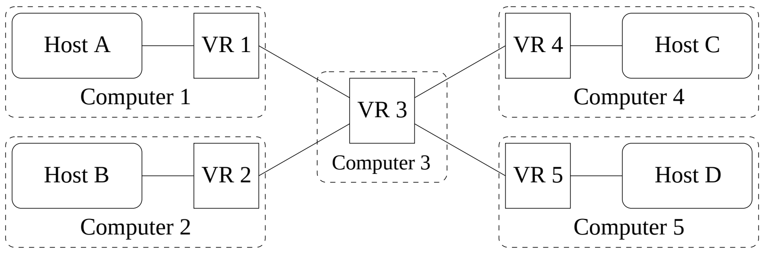


Figure 3.3: *Increasing the number of end systems*

The use of separate end systems for each traffic source or sink would cause a significant demand for computers in a topology of a reasonable size as can be seen in Figure 3.3. Fortunately, IP addresses are usually bound to the network interfaces (or logical interfaces) of that computer. As mentioned in Section 3.1, the softlink device is an emulation of a normal network interface and like other interfaces it owns an IP address. Since it is possible to create several softlink devices on a computer (up to 256), a computer may appear at different points in a topology as source and sink at the same time (see Figure 3.4).

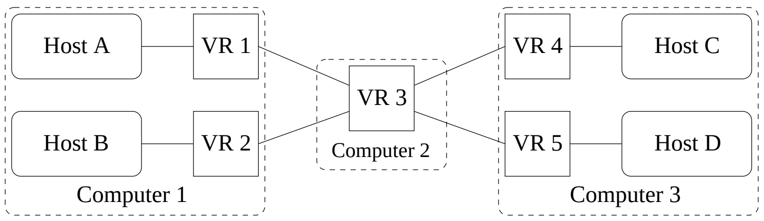


Figure 3.4: *using multiple softlink devices*

Unfortunately it is not simple to use a computer as source and sink simultaneously and forward traffic from the computer through an emulated topology back to itself. At least the Linux network layer will detect, that the destination address is one of the local (softlink) interfaces and will ignore the according entry in the routing table. Without changes to the network layer, there is no workaround for that behaviour. Fortunately, there is a way to trick the Linux router to forward the packet to the emulated topology using the address translation mechanism.

3.3 Address Translation

To use a single computer as multiple sources, sinks and also as location for one or a couple of Virtual Routers the address translation mechanism can be used [BB00b]. In Figure 3.5 this simple set-up is shown.
Packets on host A shall now be sent through the Virtual Router to host B. The same computer acts as two different hosts by setting up two softlink devices. The Virtual Router connecting the hosts runs on the same computer.
If a packet now is directly sent to the address IP_A of host A, the network layer of the computer will detect that IP_A is an interface of the same computer and process the packet internally, instead of sending it through the softlink devices and the Virtual Router.
To cope with this problem, an address translation within the softlink connection is performed. A packet, which is received by the Virtual Router over a softlink connection with the source, destination address pair (IP_{src}, IP_{dest}) is mapped to a packet $(IP_{src}, IP_{dest}^{mapped})$.
A packet, which is forwarded by the Virtual Router over a softlink connection to an end system (IP_{src}, IP_{dest}) is mapped to a packet $(IP_{src}^{mapped}, IP_{dest})$.
This mechanism allows to forwarding packets transparently through one or multiple VRs even, when the destination interface is placed on the same host as the source interface.
The following sequence shows the translations and the change to packet headers during the forwarding of the packet.
We use the network shown on the diagram above to demonstrate the mapping. A packet shall be sent from A_1 oder VR 1 to A_2. The packet is now not directly sent to IP_{dest} but to IP_{dest}^{mapped}.
So the computer sends a packet with the addresses

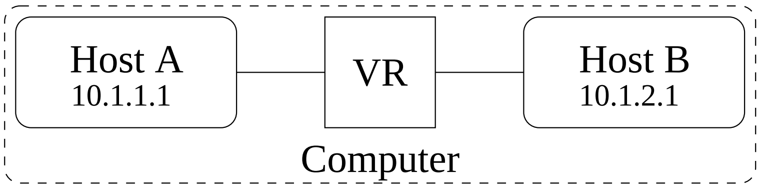


Figure 3.5: *A Virtual Router connected to two host interfaces of the same computer*

$$(IP_{src}, IP_{dest}^{mapped})$$

over interface A_1. Receiving this packet on a softlink connection, the VR translates the destination address to the correct destination address.

$$(IP_{src}, IP_{dest})$$

Assuming the routing tables within the VR are set up correctly, the VR will forward the packet to the VR-interface connected to A_2. Here the source address IP_{src} will be translated to IP_{src}^{mapped}.

$$(IP_{src}^{mapped}, IP_{dest})$$

The computer receiving this packet on host interface B can directly reply to the packet by using the IP_{src}^{mapped} as destination address. The same translation will take place in the other direction.
As the set-up of this address mapping scheme can be quite complicated, the VR maps in his default set-up any address fitting `172.0.0.0/8` to `10.0.0.0/8`, with only the first byte being replaced. So a packet sent to `172.1.19.22` and routed to a VR is there mapped to a `10.1.19.22`. The same is done in the opposite direction. This settings work fine, if the softlink interface of the hosts (A,B) have addresses of the `10.0.0.0` network.
To illustrate that mechanism a short example shall be given. As shown in Figure 3.5 Host A has the address `10.1.1.1` and host B the address `10.1.2.1`. Both hosts are realised by softlink interfaces of the same computer and are connected by a Virtual Router running on this computer. A packet shall be sent from host A to host B over the Virtual Router. If the packet would be sent directly to `10.1.2.1` it would not pass the VR but be processed within the computer.
Therefore, the packet is sent to address `172.1.2.1` instead of `10.1.2.1`. On the computer a route was set up sending packets for `172.1.2.*` to the Virtual Router. When the Virtual Router receives the packet, it translates `172.1.2.1` (the destination address) back to `10.1.2.1` and forwards the packet – according to its routing rules – to host B. Leaving the Virtual Router, the packet's source address is modified. The original source address `10.1.1.1` of host A is replaced by `172.1.1.1`. Being received by Host B, the packet is now addressed to `10.1.2.1` and originates from `172.1.1.1`. The translation of the source address allows a direct answer back to this address, automatically forcing a routing through the Virtual Router.
Even when this mechanism interacts smoothly with external network devices, it is complicated and mainly thought to set up small networks for development purposes. In such a scenario the capability to work on only a single computer is very important.

Since for larger topologies usually at least two computers are used, the address translation is not necessary.

Chapter 4

The Internal Shell

In addition to the described binary format, the VR also provides a parser to process human readable commands and to produce readable ASCII format. In this chapter the available commands and their output will be presented. It should be mentioned here, that these interfaces do not provide an alternative access to the base forwarding layer. These interface functions only parse a command string and translate it to appropriate binary data. The binary data is sent to the base forwarding layer over an API channel. Vice versa output received on the API channel is translated to a human readable format. It is strongly recommended to use the binary format for automatic configuration because these human readable commands may change frequently.

4.1 Configuring the Interfaces

This command covers all configurations of and VRs interfaces. Each interface has an unique name, which is used to identify the interface to be configured. The length of the name is limited to 9 chars. In the following section it will be described, how interfaces are set up and configured.

Command Syntax:

```
ifconfig help:

 ifconfig
 ifconfig add <ifname> <ip-address> [netmask] [broadcast]
 ifconfig <ifname>
 ifconfig <ifname> delete
 ifconfig <ifname> reset
 ifconfig <ifname> if <newid>
 ifconfig <ifname> bw <bandwidth> [buckettime]
 ifconfig <ifname> address <ip> [netmask] [broadcast]
 ifconfig <ifname> connect ...
 ifconfig <ifname> disconnect
 ifconfig <ifname> ttx ...
 ifconfig <ifname> trx ...
 ifconfig <ifname> qs
 ifconfig <ifname> qs conns
 ifconfig <ifname> qs list
 ifconfig <ifname> qs create [droptail|tbf|scheduler|classifier]
 ifconfig <ifname> connect sol <#no>
 ifconfig <ifname> connect fifo <#no> <{l|r}>
```

```
  ifconfig <ifname> connect tunnel <hostname> <dest_port> <src_port>
  ifconfig <ifname> connect ipip <hostname> [mtu]
  ifconfig <ifname> qs chain <component id1> <component id2>
  ifconfig <ifname> qs <component id> reset
  ifconfig <ifname> qs <component id> status
  ifconfig <ifname> qs <tbf-id> bw <bandwidth> [bucket size]
  ifconfig <ifname> qs <droptail-id> ql <queuelen>
  ifconfig <ifname> qs <scheduler-id> wt <component id> <weight>
  ifconfig <ifname> qs <dsmarker-id> mark ... as ...
  ifconfig <ifname> qs <dsmarker-id> erase ... as ...
  ifconfig <ifname> qs <trio-id> ql len0 len1 len2
  ifconfig <ifname> qs <trio-id> mode {hard|linear}
  ifconfig <ifname> qs <scheduler-id> mode {wfq|prr|rr|pwfq}
```

Interface configuration, which is supported by the API can be done by this command. The command without any arguments results in a listing of all configured interfaces with their parameters.

4.1.1 Creating a new Interface

```
> ifconfig add if0 10.1.1.1
ip-address:           10.1.1.1
netmask:         255.255.255.0
broadcast:          10.1.1.255
bandwidth(bps):        1000000   bucket size(bytes):        125000
drops:                       0   errors:                         0
rx:                          0   tx:                             0
rx-t-ip               172.0.0.0   rx-t-nm                  255.0.0.0
rx-t-val               10.0.0.0   rx-t-pat                 255.0.0.0
tx-t-ip                10.0.0.0   tx-t-nm                  255.0.0.0
tx-t-pat              255.0.0.0
connection                 none
```

This interface was added without any parameters except the interface name `if0` and the IP address `10.1.1.1`. The rest of the parameters are the default values.

4.1.2 Connecting and Diconnecting Interfaces

Also the connection of interfaces is handled by the ifconfig command. There are three ways for connecting interfaces:

- Connections between VRs running on the same host (IPC). These connections can be established by submitting the link number and the ”end” of the cable to the ifconfig command.

  ```
  ifconfig <ifname> connect fifo #no {l|r}
  ```

  These connections are mapped to the appropriate FIFO queues on the system. The ”left” end of the ”cable” is associated with the end id `0`, whereas the ”right” end has the id `1`.

- Connections between VRS on different hosts.

  ```
  ifconfig <ifname> connect tunnel remotehost tx_port rx_port
  ```

To establish a connection between two VRs on different hosts UDP tunnels are used. These connections require the specification of the remote host and two port numbers. The port is the port, where the packets are sent to, the second number specifies the port, where incoming packets from these hosts are accepted.

- A connection to a Softlink device can be established by the command.

  ```
  ifconfig <ifname> connect sol #no
  ```

  The number specifies the softlink device on the system. An interface can be disconnected with the

  ```
  ifconfig <ifname> disconnect
  ```

  command.

- Setting up IP over IP tunnels

  ```
  ifconfig <ifname> connect ipip 10.42.10.43 1200
  ```

  The tunnel connection mentioned above is used to connect the interfaces of two virtual routers. This tunnel is comparable to an Ethernet cable connecting two real routers. In contrast there is a possibility to encapsulate IP packets within other IP packets and tunnel them through a network to another router. This mechanism is also called IP over IP tunnels. Setting up an ipip connection, packets routed to that interface are encapsulated within an IP packet and forwarded to the specified destination address. These destination address can also be a remote (not neighbouring) router. A router receiving these packets will decapsulate the packet and treat its payload as a normal IP packet.

4.1.3 Configuring an Interface's Queueing System

Each interface has an own queueing system attached. This queueing system may consist of several components connected to each other. A more detailed description about the single components is provided in Chapter 5.7. The minimum queueing system consists out of a single droptail queue. One from a couple of components may be chosen and instantiated using the command

```
ifconfig <if> qs create <component type>
```

where `<if>` is the name of the interface the queueing system is attached to and the name of the component type to be instantiated is given by `<component type>`. See the above mentioned Chapter 5.7 about the queueing system for a list of available component types. The command returns an id for the newly created component. All configuration commands will use this id as reference for the component.
The next step is to connect this component to other components. Each component gets packets from at least one other component and has at least one component it may forward packets to. Other component-types (e.g. schedulers) allow to receive packets from several other components, other ones allow to forward packets to multiple other components.
To tell a component with the id 3 to send packets to the component 17 the command

```
ifconfig <if> qs chain 3 17
```

is used. If the component 3 allows to forward packets to additional components, the same command may be used again. So an additional

```
ifconfig <if> qs chain 3 19
```

allows component 3 to forward packets to both components 17 and 19. Which packet is forwarded to which component is not affected by this configuration. This decision depends only on the internal mechanisms of number 3 and may have to be configured separately.
The other case, that a component received packets from multiple other components may be configured analogous.
As the queueing systems set up might have to be adapted these links between the components might be also removed by the command:

```
ifconfig <if> qs unchain <id1> <id2>
```

Several commands are available to maintain the instantiated components and the links between them.

`ifconfig <if> qs avail` shows a list with all available component types.

`ifconfig <if> qs list` will show a short list of all instantiated components with their component id and their type.

`ifconfig <if> qs conns` lists all connections between components. For each component the type and the id is shown.

`ifconfig <if> qs <id> status` shows the configuration and the status of the component with id `<id>`. Dependent on the component type the information printed here might be more or less detailed.

`ifconfig <if> qs <id> reset` resets all statistical counters for the element. These are mainly counters for in and outgoing packets.

As mentioned above the links between the single quite generic modules of the queueing system define only the basic layout. For final behaviour also the configuration of the single components is important. The following list gives a short description of the configurable parameters for each component.

token bucket filter (tbf): The following command allows to modify the bandwidth and (optional) the bucket size of a token bucket filter.

```
ifconfig <if> qs <id> bw <bandwidth> [bucket size]
```

The `<id>` value has to be the id of a tbf component. The command expects the bandwidth in Mbps (e.g. 2.0 for 2 Megabit per second) and the bucket size in Megabit.

droptail queue (droptail): At the moment the only parameter available, which can be set at the droptail queue is the queuelen. This value determines how many packets might be stored in the queue.

```
ifconfig <if> qs <dpt-id> ql <queuelen>
```

generic scheduler (scheduler): The generic scheduler allows more parameter to be set. Since the scheduler may operate in different modes, these mode may be changed. For a detailed description of modes see Chapter 5.7.

```
ifconfig <if> qs <sched-id> mode {wfq|rr|prr|pwfq}
```

Dependent on the mode an additional parameter is necessary. This parameter is called weight and might be interpreted as bandwidth share, as a priority or is simply ignored by the scheduler.

```
ifconfig <if> qs <sched-id> wt <id> <wt>
```

The weight value can be set for each component preceding the scheduler (a scheduler has usually several incoming links), so the second `<id>` value specifies which weight shall be set.

classifier: The classifier has usually multiple follow up components. The classifier allows to set rules, specifying which packet is forwarded to which component. The syntax to set up such a rule is:

```
ifconfig <if> qs <cls-id> <dip/nm> <sip/nm>
                 [<p>] [<t>] to <id>
```

The `<dip/nm>` and `<sip/nm>` specify a range of destination and source IP addresses. If the netmask is omitted the netmask `255.255.255.255.` is assumed. The specification of the protocol id and the ToS byte value are optional. The `<id>` following the key word `to` specifies the component packets matching this filter will be forwarded. There may be multiple such rules pointing to the same component.

Rules defined at the classifier can be removed by:

```
ifconfig <if> qs <cls-id> del <dip/nm> <sip/nm> [<p>] [<t>]
```

DiffServ Marker

The Differentiated Services Marker component can be configured with a set of rules, specifying which flow shall be marked with which DSCP up to which bandwidth.

```
ifconfig <if> qs <dsms-id> mark [source a.b.c.d/n]
[dest a.b.c.d/n] [proto p] [tos t] as <service>
```

The command allows to specify the flow by source and destination addresses, the protocol and the ToS Byte value. The service specifies which DSCP shall be set. The service can simply be EF for the Expedited Forwarding service of a term like

```
af[1|2|3|4]:bw1[:bs1]:bw2:[bs2]
```

for Assured Forwarding. The initial af1, af2, af3, or af4 specifies the AF class. This parameter has to be followed either by two or by four other parameters, setting either the maximum allowed bandwidths for low and medium drop precedence or the bandwidth values and the bucket sizes of the according Token Bucket filters.

TRIO queue

The TRIO queue is a special queue design for the AF service by dropping packets with specific DSCPs with different probability. This behaviour can be influenced by setting different queue lengths for the dropping probabilities. The command

```
ifconfig <if> qs <trio-id> ql 10 20 30
```

will configure a TRIO queue to a maximum queuelength of 30 packets. Packets with high drop precedence are dropped if the queuelength exceeds 10 packets, whereas packets with medium drop precedence are dropped only if the queue exceeds 20 packets. There is an additional parameter to influence the algorithm used to calculate the dropping probability. This parameter can be set by

```
ifconfig <if> qs <trio-id> mode {hard|linear}
```

The different algorithms are described in section 5.7.7.

4.2 Setting up Routes

This command allows to add, list and delete static routes in the base layer's routing table. Even when the base layer's routing rules support source, protocol and ToS based routing and the binary API also allows to set up those routes, the ASCII based command front end currently only supports the set up of ”standard” routing rules.

Command Syntax:

```
route help:
route
route add <ip/nmb> <interface>
route add <ip> <nm> <interface>
```

4.3 Querying system information

The system commands offer a possibility of access general access to system information, as the central scheduler or the module loader.

Command Syntax:

```
sys help:

sys events
sys filters
sys lobs
```

sys events: lists the registered scheduler events. A typical output for a shell and two interfaces looks somewhat like:

```
shell susp 0.0 type   R thrown - read 0 write 0 hits 6
  if0 susp inf type  RW thrown - read 3 write 3 hits 1
  if1 susp inf type  RW thrown - read 5 write 4 hits 1
```

sys filters: not yet implemented

sys lobs: this command allows to list all currently loaded objects.

```
name hellot, id 0, mode THREAD , size 10 kb, up 110.10 min
name pyan, id 3, mode THREAD , size 90 kb, up 3.10 min
```

4.4 Loading new Objects to the VR

This command allows the loading of additional modules, so called Loadable Objects into the VR kernel. The command syntax is simple and allows to pass additional command line arguments to the LOB.

```
load help:

 load <lob> [arg1] [arg2] [arg3] [...]
```

The shell replies with some information about the loaded object or an error code.
A typical output might look like:

```
object 'hellot' loaded, id is 0, mode is THREAD
```

4.5 Connectiing external APIs

To connect external APIs the attach command allows to use two file system fifo file (see also man mkfifo) to be used to exchange binary control and result blocks. The command expects the filename of the fifo files.

```
attach <fifo_out> <fifo_in>
```

Chapter 5

Virtual Router API

This document describes the interface between the Virtual Router's (VR) core mechanisms and an application running on top of the VR. The application may use either the loadable object mechanism to access the Virtual Router or is run as an external process using a communication channel to interact with the Virtual Router.

5.1 API channels and VRCB handles

A Virtual Router may have several APIs each used by another application like a shell, a packet monitor or a graphical front end. Each API establishes an API channel, a duplex connection between the Virtual Router and the program.
Since only this channel is used for the communication, the program may also access a Virtual Router on a remote computer.
The communication is based on Virtual Router Control Blocks (VRCBs) and Virtual Router Result Blocks (VRRBs). The Virtual Router receives a control block, parses it, executes the command and returns an appropriate result block. There are several control and result blocks for different commands and the according results. All data types are in network byte order. A ulong specifies a four byte, a ushort a two byte wide integer.
To allow a simple parsing of the control and result blocks a hierarchy of control block exists. Each block starts with a generic header (VRCB/VRRB) and is followed by command dependent data. A specifier at the beginning of the control block defines the format of the following data. Additionally, the basic VRCB contains a handle and information about the control block lengths.

VRCB	**Control Block**
ushort	Handle of the control block, it will be referenced in the returned result block
ushort	total length of control block
ushort	VRCB command specifier

The handle is unique and referenced by the returned result block. This allows a simple mapping between commands and the according results. Currently, the following VRCB command specifiers are defined:

VRCB command specifiers	
1	add an interface to the Virtual Router
2	delete an interface
3	get list of interfaces. This will return a list of numbers used to reference an interface
4	query an interface by its name.
10	changing interface parameters (bandwidth, ...), modify the queueing system
20	list, add and delete routing table entries
21	query the event handler and return the event status of scheduled events or devices like interfaces
30	setup, list and remove filter setup
31	add a new protocol stack to the Virtual Router
40	list, add and remove loadable objects
50	send an IP packet to the Virtual Router

The Virtual Router returns a specific Virtual Router Control Block according to the received control block. Each result block starts with a basic VRRB.

VRRB	**Result Block**
ushort	handle referencing the according control block
ushort	type of the returned result block
ushort	total length of the result block

It references the according VRCB by a handle. The result block type is important to process asynchronous events. Usually a control block is directly responded by the according result block (type 0). Some configurations can cause the Virtual Router to send additional information asynchronously over the API channel. Therefore this field is set to signal the type of result block received. Currently the following values are defined:

Result Block Types	
0	synchronous RB sent as an answer to a CB
1	filtered packet
2	packet for protocol stack

If a filtered packet is sent over the API channel, the handle of the result block refers to the handle of the control block sent to set up the filter. The API guarantees that there are no asynchronous packets sent between a control block and its synchronous answer. However there might be some asynchronous packets still in the line. Therefore incoming datagrams should be checked for their handle and result block type.

This control block - result block mechanism is used for all types of configuration. The control blocks are structured. Additionally to the VRCB command specifier there might be additional command specifiers controlling certain parts of a component (e.g. control of the queueing system of an interface).

5.2 Adding an Interface

To add an interface a special VRCB has to be sent over the API. This VRCB contains fundamental information about the interface to be added.

IF_ADDINTERFACE_CB	**Control Block**
VRCB	control block with handle, length and base command specifier
byte[10]	name of the interface. The string has to be followed by a 0-byte
ulong	ip address in network byte order as returned by `inet_addr()`
ulong	netmask in network byte order
ulong	broadcast address

The name field has to be terminated by a null byte. A different name should be used for each interface. The Virtual Router will return block containing information about the interface.

IF_INFORMATION_RB	**Result Block**
VRRB	result block with the reference handle and the length
byte	the internal number of the interface within the VR. This parameter is set to 0xffff if the request fails.
byte[10]	the name of the interface
ulong	ip address
ulong	netmask
ulong	broadcast address
ulong	number of received packets (rx)
ulong	number of transmitted packets (tx)
ulong	errors during transmission/reception
ulong	packets dropped by the interfaces queueing system
ulong	bandwidth in bytes per second
ulong	bucket size of the interface's rate limiter in bytes
ulong	bucket level of the limiter in bytes
ulong	ip translation for outgoing packets: address
ulong	ip translation for outgoing packets: netmask
ulong	ip translation for outgoing packets: new address
ulong	ip translation for outgoing packets: pattern
ulong	ip translation for incoming packets: address
ulong	ip translation for incoming packets: netmask
ulong	ip translation for incoming packets: new address
ulong	ip translation for incoming packets: pattern
byte	connection type; 0: not connected, 1: softlink, 2: udp-tunnel, 3: local link (FIFO), 4: ipip tunnel (logical interface)
ulong[3]	connection parameters

On error the interface number field is set to 0xffff. See table on page 5.6.2 for a description of connection parameters.

5.3 Deleting Interfaces

IF_DELETEINTERFACE_CB	**Control Block**
VRCB	the control block header
ushort	the number of the interface to be deleted

As a response to this control block the Virtual Router returns:

IF_DELETEINTERFACE_RB	**Result Block**
VRRB	result block header
ushort	number of the deleted interface, 0xffff on error

5.4 Querying Interface Numbers

To query a list of existing interface numbers, a VRCB with the according VRCB command specifier is sent to the VR. The VR will return a result block with a list of valid interface numbers.

IF_IDQUERY_CB	**Result Block**
VRRB	result block header
byte[]	returns a list of interface numbers

The length field in the result block header can be used to calculate the number of returned interface numbers.

5.5 Query Interface by Name

The special command here is to query the interface id by the name of the interface. The according control block is:

IF_IDNAMEQUERY_CB	**Control Block**
VRCB	control block header
byte[10]	name of the interface the number has to be queried

The Virtual Router will return an interface information result block as described on page 31. If there is no matching interface within the Virtual Router the number field of the returned result block will be set to 0xffff.

5.6 Configuration of a specific Interface

There are several interface specific parameters or commands. Each control block addressing a specific interface starts with an extended VRCB called IF_VRCB providing the number of the interface and an interface specific command code.

IF_VRCB	**Control Block**
VRCB	the control block header
ushort	interface number
ushort	interface specific command

The interface specific command determines also the format of the rest of the control block. The following table lists the currently defined command codes.

interface specific command codes and their parameters		
1	-	query interface information (see section 5.6.1)
2	-	reset all counters of an interface (rx,tx,dropped packets, errors)
101	byte[10]	set the interface name. The string should be ended by a 0 byte
102	3 x ulong	change of ip-address, netmask, broadcast (see section 5.6.2)
105	2 x ulong	configure the interface bandwidth (bytes per s) and bucket size (bytes) (see section 5.6.2)
107	4 x ulong	set up of translation table for received packets (ip address, netmask, new value, pattern)
108	4 x ulong	set up of translation table for packets to be sent (ip address, netmask, new value, pattern)
109	byte + ulong + 2x ushort	connect or disconnect the interface (see section 5.6.2)
120	...	configuration of the interface's queueing (see section 5.7)

5.6.1 Querying Interface Configuration

This interface command takes no arguments. An IF_VRCB with the according command value 1 is passed to the Virtual Router. The Virtual Router returns an IF_INFORMATION_RB as described on page 31. If an error occurs the field of the result block containing the interface number is set to 0xffff.

5.6.2 Setting of Interface Parameters

As can be seen in the previous table several parameters of the interface can be modified. Each of these configurations is done by an IF_VRCB with the appropriate command specifiers and some additional command dependent data.
Any of this requests will be answered by a returned IF_INFORMATION_RB with the interface number field set to 0xffff if an error occurred. The following list will briefly describe the commands and the required parameters.

interface bandwidth/bucket size (105): The first parameter specifies the bandwidth in bytes per second, the second one the bucket size in bytes. If the second parameter is set to 0xffffffff the bandwidth only is modified.

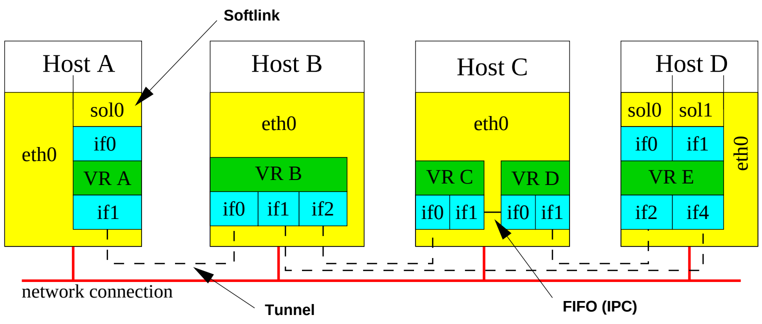
ip, netmask and broadcast addresses (102): This API call requires three parameters. If a parameter is set to 0xffffffff, the according value of the interface is not changed. Each address has to be provided in network by order as returned by the `inet_addr()` function.

(dis)connect a Virtual Router interface (109): Since a Virtual Router interface may be connected in different manners this control block takes a set of parameters. The meaning of this parameters depend on the type of the connection.

IF_CONNECTION_CB	**Control Block**
IF_VRCB	interface control block header
byte	connection type
ulong[3]	connection parameters

The connection parameters and their meaning are defined in the following table.

Connection Types and Parameters				
	type	[0]	[1]	[2]
disconnected	0	-	-	-
Softlink	1	-	#no	-
IPC	2	-	#no	1/2
UDP	3	destination address	tx port	rx port
IPIP tunnel	4	destination addressR	-	MTU



The Virtual Router will return an result block with interface informations as described on page 31. According to the type of error either the interface number field is set to 0xffff or the connection parameters shows a (dis)connected device.

5.7 Modifying the Queueing System

Because of its flexibility the queueing system is one of the most complex parts of the interface. It consists of a set of components like queues, filters, shapers and schedulers, that can be combined and configured during runtime. Currently the following components are offered:

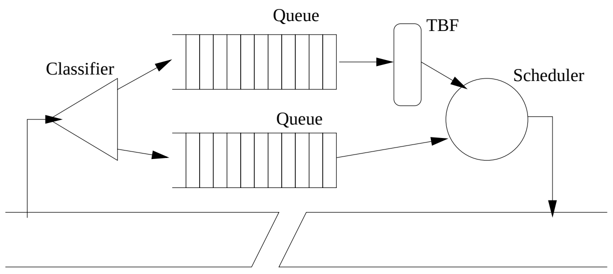


Figure 5.1: *Several small queueing components set up more complicated queueing systems*

- a generic classifier
- a token bucket filter
- a drop tail (FIFO) queue
- a random early detection queue (RED) [FJ93]
- a Weighted Round Robin (WRR) scheduler
- a simple Round Robin scheduler
- a Priority Round Robin (PRR) scheduler
- a TRIO queue [HBWW99]
- a Differentiated Services marker
- a Priority Weighted Round Robin (PWRR) scheduler

Each Virtual Router interface has an own queueing system attached. It stores and processes packets put to the interface by the Virtual Router. Every time the interface is able to transmit, a packet is removed from the queue and sent over the interface connection.
The queueing system implemented within the Virtual Router consists of several small basic components as listed above. These components can be linked together to set up complicated systems favouring certain kinds of packets or limiting the bandwidth of others.
As can be seen in Figure 5.1, each queueing component has a number of input links and a number of output links. The number of input and output links depend on the type of the component. While a FIFO queue will have one input and one output link, a classifier will have one input and multiple output links. For the set up of a queueing system the following steps are required:

creation of components: Since a queueing system will contain multiple queues or token bucket filters, a component first has to be instantiated. Each instance gets an unique id, that has to be used as reference to this instance during later operations.

linkage of instances: The instances (referenced by their id numbers) can be linked together. The number of links allowed for a component depends on its type.

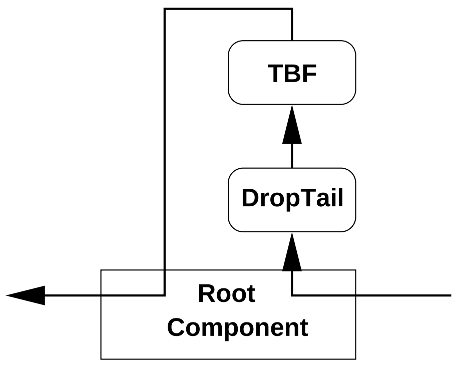


Figure 5.2: *The root component is the basis for all other components and connects the queueing system with the interface*

component configuration: Each instance of a queue, a scheduler or any other component can be configured separately. Of course it is possible to define some reasonable global default values like FIFO queue lengths. But since the behaviour of the queueing system will depend on a proper set up of each single component, bandwidth values and bucket sizes for the token bucket filters can of course be set separately as well as the weights or priorities of PRR/WRR/PWRR schedulers.

The basis to link the queueing components is the *root component* as shown in Figure 5.2. A packet forwarded to the interface is sent to the root component. The root component will forward the packet to the next component.
The root component is also accessed if the interface can send data and has to extract packets from the queueing system. Therefore, each packet leaving the Virtual Router has to pass the root component twice. First when sent to the interface and a second time, when extracted from the queueing system to be transmitted.

Each different component type has an type id as listed on the following table. This id is used to signal a components type within the control and result blocks.

Component types and ids	
component type id	
0	the root component itself
1	droptail queue, a simple FIFO queue
2	Token Bucket Filter to limit bandwidths with variable rate and bucket size
3	generic multi field classifier
4	generic scheduler with RR, WRR and PWRR support
5	trio queue for differentiated services
6	a packet marker as required for differentiated services

The components listed in the table may be created, connected, disconnected and removed. All commands for the queueing system or queueing components start with

If_Q_VRCB extending the IF_VRCB control block used for interface configuration. To access the queueing system the appropriate command specifiers have to be set in the VRCB, the IF_VRCB and in the IF_Q_VRCB. The following table lists the command specifiers for the IF_Q_VRCB.

IF_Q_VRCB	**Control Block**
IF_VRCB	the interface specific control block, containing also the fundamental VRCB
ushort	queueing system specific command

The table lists the command set which can be sent to an interface's queueing system. This set covers mainly tasks concerning all components. To configure a single component the command specifier #10 provides direct access to a component. The configuration of specific components depends of course crucially on the component's type requiring special control and result blocks as will be described in the next section.

basic queueing system commands		
1	-	list all components (section 5.7.1)
2	-	list all connections (section 5.7.2)
3	ushort	create component (section 5.7.3)
4	ushort	remove component
5	2 x ushort	connect two components (section 5.7.5)
6	2 x ushort	disconnect two components (section 5.7.6)
10	...	component specific configuration

5.7.1 Query List of Components

To request a list of components the IF_Q_VRCB does only contain the command specifier, and no additional data is needed. The returned result block looks like this:

	Result Block
VRRB	the result block header
ushort[]	list of (c-id/c-type) pairs

Each list element has two fields. The first one specifies the component's id, the second one the type of the component. As mentioned previously, each component in the queueing system has an unique id used to address the component. To calculate the number of components contained in the result block, the length field within the VRCB can be used.

5.7.2 Query list of Connections

A control block with the queueing command specifier #2 will return a result block with a listing of all connections between the components.

	Result Block
VRRB	the result block header
ushort[]	list of [c-id/c-type, c-id/c-type] tuples

Each list elements contains information for both components a connection. Therefore four values are included For each end point a c-id/c-type pair is provided. The first pair specifies the start point of the connection, the last one the end point. As usual the length field in the VRRB can be used to calculate the number of connections.

5.7.3 Create a new Component

A new component is created by sending the component type to the API.

	Control Block
IF_Q_VRCB	the queueing system header block
ushort	type of component to be created

The API answers with an ip/type tuple.

	Result Block
VRRB	the common result block header
ushort	the component id of the created component
ushort	the type of the created component

On error the component id is set to `0xffff`. In future releases the component type may be used for further error codes.

5.7.4 Remove a Component

For the removal of a component the component has to be disconnected first. If the component is not connected the following control block will remove the component.

	Control Block
IF_Q_VRCB	the queueing system header block
ushort	the id of the component to be deleted

The API answers with an ip/type tuple.

	Result Block
VRRB	the common result block header
ushort	the component id of the removed component
ushort	the type of the removed component

The component id is obsolete after the removal of a component and might be re-used if other new components are created. On error the returned component id is set to 0xffff, the component type field indicates the error more precisely as listed on the following table.

Error codes during component removal		
component id	component type	
0xffff	1	cannot delete root component
0xffff	2	no such component
0xffff	3	component busy

5.7.5 Connect two Components

A connection between two components allows to pass packets from one component to the next. A component may have multiple connections to other components, as for example a classifiers puts packets to different queues. Whether a component allows multiple connections or not is determined by the component itself. The TBF has one input and one output slot. Therefore a TBF can have two connections:

$\rightarrow$ **tbf** The tbf is the end point of a connection and receives packets from another componennt.

tbf $\leftarrow$ The tbf is the start point of a connection and sends packets to the nect component.

Any additional connection for a Token Bucket Filter would result in an error.

	Control Block
IF_Q_VRCB	the queueing system header
ushort	component id 1
ushort	component id 2

The Virtual Router returns a result block with a pair of both component ids.

	Result Block
VRRB	the result block header
ushort	component id 1
ushort	component id 2

Both component ids are also used to indicate errors as shown on the following table.

Error codes during Component Connection		
id 1	id 2	description
0xffff	1	first id invalid, no such component
0xffff	2	second id invalid, no such component
0xffff	3	connection to next failed, component busy
0xffff	4	connection to previous failed, component busy
0xffff	5	invalid length

5.7.6 Disconnect two Components

The disconnection requires a similar data format as used for the connection of queueing components. Also the error codes returned within the result block if an error occurs are similar to those used during component connection.

5.7.7 Component Configuration

The commands dealing with the queueing system presented so far covered tasks more general like the creation of queueing components and their connection.
In this section the configuration of specific queueing components will be described. As explained before each component has an unique identifier. To address a specific component the previously introduced IF_Q_VRCB control block is extended to an IF_QS_VRCB.

IF_QS_VRCB	**Control Block**
IF_Q_VRCB	queueing system control block header
ushort	component id of the addressed queueing component
ushort	component specific command

As mentioned before the control blocks follow some kind of hierarchy. Therefore, the IF_Q_VRCB header contains other control block headers. To illustrate that hierarchy, the following table lists a complete IF_QS_VRCB.

IF_QS_VRCB	**Control Block**
ushort	control block handle
ushort	length of the control block
ushort	VRCB command specifier = 10
ushort	interface number
ushort	interface specific command (IF_VRCB) = 120
ushort	queueing system command = 10
ushort	component id of the addressed queueing component
ushort	component specific command

Theoretically a component specific command code might have different meanings for different components. Since there are enough command specifiers available and a multiple usage of the same value would make things more complicated, different component command codes were defined. As can be seen in the following table, component specific codes of a value less than 100 are dedicated to functionalities common to all components like the information query or the resetting of statistical counters. The meaning of command codes over 100 may differ for different components.

Command codes for specific queueing components		
code	component	
1	all	reset the statistical counters of the component
2	all	information request
210	2	configure Token Bucket Filter
310	1	configure Droptail (FIFO) queue
410	4	change the scheduler mode
411	4	modify a scheduler weight/priority
511	3	adding a classifier rule
512	3	removing a classifier rule
611	6	add a rule to the Differentiated Services marker
612	6	delete a rule within the Differentiated Services marker
710	5	configure the Differentiated Services TRIO queue

Of course each of the component dependent commands requires a specific configuration datagram. The control blocks have only the IF_QS_VRCB part in common.

Reset the Counters of a Component

A resetting of a component affects all statistical counters of this component. The function is common to all components of the queueing system. An IF_QS_VRCB with the appropriate command code is sent to the API. As result an empty VRRB is returned.

Query a Component's Status

For a request of component parameters and statistic counters an IF_QS_VRCB control block with the appropriate command code only is necessary. Obviously, the amount and type of data returned depends crucially on the component type. However, each result block returned starts with:

QS_STAT_VRRB	**Result Block**
VRRB	the general result block header
ushort	component id
ushort	component type

The rest of the result block depends on the component. The following list described the result blocks returned by each component.

Droptail Queue: contains information about the only parameter of a droptail queue, the queue length and statistical information, how many packets passed the queue, how many packets were dropped a.s.o.

	Result Block
QS_STAT_VRRB	common statistical header
ushort	length of queue in packets
ulong	dropped packets
ulong	deque requests
ulong	deque hits
ulong	deque fails

Root Component: Since all packets sent to the queueing system pass the root component, the statistical data in this datagram gives a very global impression about the queueing system's activities.

	Result Block
QS_STAT_VRRB	common statistical header
ulong	enqued packets
ulong	dequed packets
ulong	packets stored in the queueing system
ulong	packets dropped by the queueing system

Token Bucket filter: The token bucket filter is defined by a token rate and a bucket allowing to buffer a certain amount of tokens.

	Result Block
QS_STAT_VRRB	common statistical header
ulong	bucket rate in bytes per second
ulong	bucket size in bytes
ulong	enque hits (internal)
ulong	enque fails (internal)
ulong	enque empty (internal)
ulong	deque hits (internal)
ulong	deque fails (internal)
ulong	deque empty (internal)
ulong	bucket overflows

Generic Scheduler: A query for statistical data reveals data for each connected component. Each component sending data to the scheduler is described by a GEN_SCHED_REC.

GEN_SCHED_REC	**Data Record**
ushort	type of preceding component
ushort	id of the preceding component
ulong	successful deque events
ushort	priority/weight (depends on scheduler mode)

These records and also additional information about the current scheduler mode are contained in the result block returned following to an information request.

	Result Block
QS_STAT_VRRB	common statistical header
ushort	type of next component
ushort	id of next component
ushort	scheduler mode (PPR, WRR, PWRR)
GEN_SCHED_REC[]	the scheduler records

Classifier: The status information of the classifier mainly contains information about the rules used to forward packets to other components. The VRRB contains a record for each outgoing component. These records are appended to the result block header.

	Result Block
QS_STAT_VRRB	common statistical header
ulong	deque events
ulong	enque events
QS_CLASS_REC[]	the classifier's records

QS_CLASS_REC	**Data Record**
ushort	component type
ushort	component id
ulong	source ip address
ulong	source netmask
ulong	destination address
ulong	destination netmask
ushort	ToS byte, use 0xffff to ignore this field
ushort	protocol, use 0xffff to ignore this field

Differentiated Services Marker: The Differentiated Services marker returns the following structure with statistical information. The structure contains overall information about the marking and the remarking the component has performed and information about each marker rule.

	Result Block
QS_STAT_VRRB	common statistical header
ulong	enqueued packets
ulong	new marks set by the marker
ulong	packet kept old mark
DSM_REC[]	the records with the marker rules

This structure is followed by a number of DSM_RECs. These records are related to the marker rules set up by preceding control blocks.

DSM_REC	**Data Record**
ulong	source ip address
ulong	source netmask
ulong	destination address
ulong	destination netmask
ushort	ToS byte, use 0xffff to ignore this field
ushort	protocol, use 0xffff to ignore this field
ushort	the service type: EF, AF, ..
ulong	service dependent parameters

The precise number of DSM_RECs depends of course on the number of set up rules. The number of records contained within the control block can be calculated by the control block's size.

TRIO queue: The datagram contains mainly data about the queue lengths set for each dropping precedence. Additionally to some statistical information the mode the TRIO queue is currently working in is returned.

	Result Block
ushort	the mode of the TRIO queue: linear, boolean, red
ushort[3]	three queue lengths
ushort	packets currently in the queue
ulong[3]	packets dropped with the different drop probabilities
ulong	deque requests
ulong	deque hits
ulong	deque fails

Configuration of Token Bucket Filters

To reduce the throughput of a component to a certain maximum a *token bucket filter* can be applied. Therefore a combination of a FIFO queue and a token bucket filter can be used to shape traffic. Since the token bucket filter only acts as limiter, a queue is needed to drop the packets. Therefore a strict policer will contain a queue with a minimal queue length and a token bucket filter, to limit the rate the queue can be emptied with.

	Control Block
IF_QS_VRCB	control block header for the queueing system
ulong	bandwidth in bits per second
ulong	bucket size in bits

This datagram changes the settings of a token bucket filter component. The two only parameter are the bandwidth and the bucket size. The bandwidth is measured in bits per second, the bucket size in bits. If a parameter is set to 0xffffffff, the API will not change the according token bucket filter parameter. The Virtual Router returns a datagram with statistical information about the token bucket filter as described on page 42.

Configuration of the Droptail Queue

The droptail/FIFO queue is a simple queue for buffering packets. If the queue is full, new packets are discarded. The only parameter is the number of packets the queue can hold.

	Control Block
IF_QS_VRCB	control block header for the queueing system
ushort	new queue length

This allows to modify the maximum number of allowed packets in the droptail or FIFO queue. The control block starts with the usual IF_QS_VRCB and contains an additional field for the queue length only. The Virtual Router returns the statistical result block for the droptail component.

Modify Scheduler Weights/Prios

To merge packets coming from different components a scheduler is used. There are different scheduler algorithms available to be able to achieve different behaviours. All different scheduler modes are integrated into a *generic scheduler*.

	Control Block
IF_QS_VRCB	control block header for the queueing system
ushort	component id of the incoming component
ushort	new weight/priority

Weights and priorities of the scheduler determine – dependent on the scheduler – mode the amount of bandwidth the connected component can achieve. In weighted round robin (WRR) mode the share s of a component x is calculated by:

$$s_x^{wrr} = \frac{w_x}{\sum_{i=0}^{N} w_i}$$

In the PWRR mode the component with the heightest weight $x = 0$ is processed if a packet is available. The share s for the other components $x > 0$ can be calculated by:

$$s_x^{pwrr} = \frac{w_x}{\sum_{i=1}^{N} w_i}$$

In the Round Robin mode the weight of the components is ignored, in the Priority Round Robin mode incoming packets are processed in order of their component's weight. The Virtual Router returns a result block containing the changed record (GEN-_SCHED_REC) (see page 43). If an error occurs (e.g. there is no such outgoing component) a plain VRRB will be returned.

Change Scheduler Mode

The mode of the scheduler is changed by this datagram.

	Control Block
IF_QS_VRCB	control block header for the queueing system
ushort	new mode

The VR returns statistical data for the scheduler (see page 43) but without the records for the connected components. The appropriate mode values are 0 for Priority Round Robin, 1 for Weighted Round Robin, 2 for Round Robin and 3 for Priority Weighted Round Robin. A change of the mode parameter does not affect the connected components. The weights used during weighted fair queueing are interpreted as priorities during RR, PRR and PWRR. So the mode may be switched without reconfiguring the weights/priorities.

Adding a Classifier rule

The classifier is used to distinguish between packets and forward them according to a set of rules to other components. The classifier might work in MF of BA mode, checking either various fields of the IP header or only the DSCP value, depending on the rules set up. The rules consist of a set of the following specifiers:

- source address (specified by IP address and netmask)
- destination address (specified by IP address and netmask)
- Differentiated Services Code Point
- maximum packet size
- protocol type

Of course for a pure BA classifier only rules regarding the DSCP would be applied, while the other fields are interesting for a multi field classification only. Even without Differentiated Services a queueing system therefore might assure a certain share of bandwidth to TCP flows in order to protect them against aggressive protocols.

	Control Block
IF_QS_VRCB	control block header for the queueing system
ushort	id of the outgoing component
ulong	source address
ulong	netmask for source address
ulong	destination address
ulong	netmask for destination address
ushort	protocol
ushort	ToS

This datagram adds a filter rule for an outgoing component. To avoid a filtering by protocol or ToS value set according values might be set to 0xffff. The Virtual Router returns an information block for the classifier with the added record appended has explained on page 43.

Removing a Classifier rule

	Control Block
IF_QS_VRCB	control block header for the queueing system
ushort	id of the outgoing component
ulong	source address
ulong	netmask for source address
ulong	destination address
ulong	netmask for destination address
ushort	protocol
ushort	ToS

This datagram allows to delete a filter rule. The first rule matching the datagram will be removed from the classifier's table. If multiple matching rules exist, this datagram has to be used repeatedly. The Virtual Router returns a result block with classifier statistics and the deleted record GEN_CLASS_REC appended (see page 43).

Adding a Differentiated Services Marker Rule

The implementation of Differentiated Services requires two additional components: a *Differentiated Service marker* and a queue capable to handle multiple drop precedences as required for Assured Forwarding. The Differentiated Service marker implemented within a Virtual Router allows to specify a set of rules. These rules include a pattern for the packets to be marked and a traffic profile. The pattern includes similar parameters like the multi field classifier.

- source address range (specified by address and netmask)
- destination address range (specified by address and netmask)
- Differentiated Services Code Point
- protocol type

The profile determines the type of service and additionally required parameters. For Expedited Forwarding this is simply a bucket rate and a bucket size. Packets matching a pattern are marked with the EF DSCP up to the specified bandwidth. The bucket size allows to add a certain acceptance for bursts.

Within an AF class packets have to be marked for different drop precedences. Therefore a two rate single three colour marker as suggested by Heinanen has been implemented as shown in Figure 5.3. Of course Assured Forwarding requires a specification about which Assured Forwarding class is to be used.

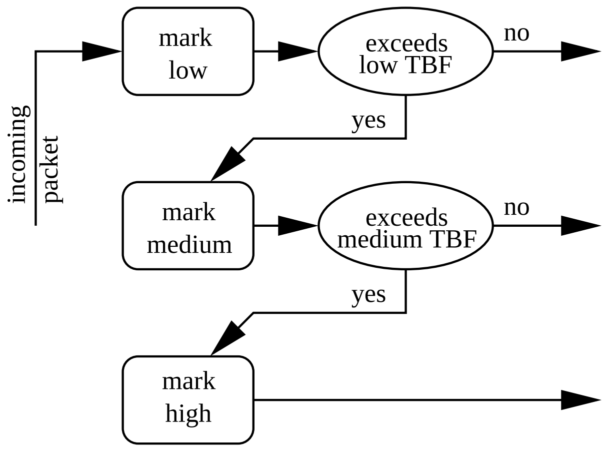


Figure 5.3: *VR implementation of a two rate three colour marker. A Packet is marked with a higher drop precedence if the packet length exceeds the number of tokens in the according bucket.*

	Control Block
IF_QS_VRCB	control block header for the queueing system
ulong	source address
ulong	netmask for source address
ulong	destination address
ulong	netmask for destination address
ushort	protocol
ushort	Differentiated Service Code Point (DSCP)
ushort	new DSCP
ulong[4]	parameters

The datagram allows to add a rule to the Differentiated Service marker. The addresses and netmasks allow to specify a set of flows. If the values for the protocol and the DSCP are to be ignored, those fields in the datagram might be set to 0xffff. The API returns a statistical block with the added rule (DSM_REC) appended as explained on page 43. The parameter block (ulong[4]) depends on the service type. If the service type is EF the parameter block might remain empty. Using the Assured Forwarding service the block has the following meaning:

Service Parameters for Assured Forwarding	
#0	max bandwidth in bytes for low drop precedence
#1	bucket size in bytes for low drop precedence
#2	max bandwidth in bytes for medium drop precedence
#3	bucket size in bytes for medium drop precedence

Removing a Differentiated Services Marker Rule

	Control Block
IF_QS_VRCB	control block header for the queueing system
ulong	source address
ulong	netmask for source address
ulong	destination address
ulong	netmask for destination address
ushort	protocol
ushort	ToS
ushort	new ToS value

The datagram allows to remove a previously set rule from the Differentiated Service marker's internal table. The API returns a block containing statistics about the marker and the record with the removed rule(see page 43).

Modifying the TRIO queue

The second component implemented especially for DiffServ, is the TRIO queue, required for Assured Forwarding. The TRIO queue has to differentiate between packets according to their DSCP value and drops Assured Forwarding packets with a high drop precedence earlier than those with a low one. The common algorithm for such a queue is based on RED. Since the benefits of RED to multi protocol flows are questionable as shown by ns simulations, the VR TRIO queue implementation supports different algorithms.

	Control Block
IF_QS_VRCB	control block header for the queueing system
ushort	mode
ushort	th_{low}
ushort	th_{medium}
ushort	th_{high}

boolean (0) This is the droptail or FIFO mode. For each drop precedence a threshold is defined. If a packet arrives and the actual queue length l exceeds the according threshold the packet is discarded. Obviously

$$th_{low} > th_{medium} > th_{high}$$

has to be true for Assured Forwarding to work properly. The probability p for the different drop precedences are:

$$p_{low} = \begin{cases} 0 & \text{for } l \leq th_{low} \\ 1 & \text{else} \end{cases}$$

$$p_{medium} = \begin{cases} 0 & \text{for } l \leq th_{medium} \\ 1 & \text{else} \end{cases}$$

$$p_{high} = \begin{cases} 0 & \text{for } l \leq th_{hig} \\ 1 & \text{else} \end{cases}$$

linear A pair of queue lengths (th_{min}, th_{max}) is defined for each drop precedence. Between these two queue lengths the dropping probability is increased linearly. Instead of using the actual queue length l an averaged queue length avg is calculated:

$$avg = (1 - w_q)avg + l \cdot w_q$$

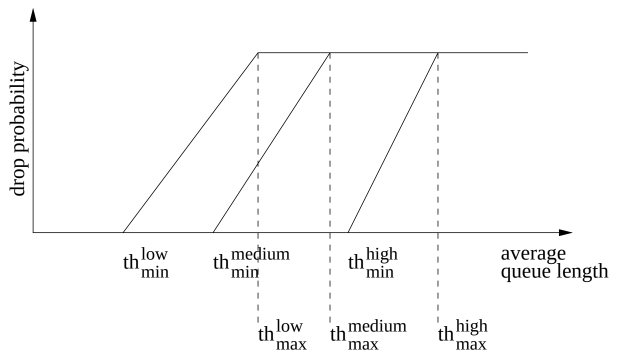
The value of w_q defines how fast the queue react to bursts.

$$p_{low} = \begin{cases} 0 & \text{for } avg < th_{min}^{low} \\ \frac{avg - th_{min}^{low}}{th_{max}^{low} - th_{min}^{low}} & \text{for } th_{min}^{low} \leq avg < th_{max}^{low} \\ 1 & \text{for } avg \geq th_{max}^{low} \end{cases}$$

$$p_{medium} = \begin{cases} 0 & \text{for } avg < th_{min}^{medium} \\ \frac{avgl - th_{min}^{medium}}{th_{max}^{medium} - th_{min}^{medium}} & \text{for } th_{min}^{medium} \leq avg < th_{max}^{medium} \\ 1 & \text{for } avg \geq th_{max}^{medium} \end{cases}$$

$$p_{high} = \begin{cases} 0 & \text{for } avg < th_{min}^{high} \\ \frac{avg - th_{min}^{high}}{th_{max}^{high} - th_{min}^{high}} & \text{for } th_{min}^{high} \leq avg < th_{max}^{high} \\ 1 & \text{for } avg \geq th_{max}^{high} \end{cases}$$

The following figure illustrates the dropping probabilities for the different drop precedences:



As the graph shows, this mode is some kind of simplified RED. It has been shown, that the choice of good RED parameters is complex and the benefits of the RED algorithm in a scenario of mixed protocols is questionable. Due to the lower number of parameters this algorithm can be adapted more easily.

RED This mode provides a TRIO queue based on the original RED algorithm. The dropping probability is calculated using an exponentially weighted moving average algorithm (EWMA). This TRIO queue has to be configured with a set of ten parameters. The probability p to drop a packet of a certain precedence d between the two thresholds th^d_{min} and th^d_{max} is:

$$p_d = \frac{max^d_p \frac{avg - th^d_{min}}{th^d_{max} - th^d_{min}}}{(1 - count^d) \cdot max^d_p \frac{avg - th^d_{min}}{th^d_{max} - th^d_{min}}}$$

Obviously the impact of the different parameters is not as obvious as in the simplified RED version.

5.8 Routing

This section specifies the data structures used to set up and query the routes used to forward IP packets to specific interfaces. Like the interface configuration the configuration is done by VRCBs and VRRBs as described on page 29. Similar to the interface configuration a generic control block is defined to access the routing system.

ROUTE_VRCB	**Control Block**
byte	routing specific command

The routing system needs three different commands only:

Routing Command Codes	
1	add a route to the routing table
2	delete a route from the routing table
3	list all routes of the table

A route is represented by a ROUTE_REC datagram, which is used by all control blocks dealing with routing table entries.

ROUTE_REC	**Data Record**
ulong	source address, 0x0 to disable source based routing
ulong	source netmask
ulong	destination address, 0x0 to ignore disable destination based routing (any practical use ?)
ulong	destination netmask
ushort	protocol, 0xffff to ignore this field
ushort	DSCP value, 0xffff to ignore
ushort	number of interface, the packet shall be routed over

5.8.1 Adding routes

The control block to add a route is:

ROUTE_RECORD_VRCB	**Control Block**
ROUTE_VRCB	routing control block header
ROUTE_REC	routing record

The fields to be ignored by the routing mechanism have to be set to the default values. For a normal ”unix-like” routing, only the destination based routing entries have to be set. The interface id can be queried as described in 5.5. The Virtual Router will return a result block like:

ROUTE_RECORD_VRRB	**Result Block**
VRRB	result block header
ROUTE_REC	routing record

On error, the interface number is set to 0xffff. The DSCP field may then be used for more detailed error codes.

5.8.2 Deleting Routes

The deleting of routes works in the same way as the adding of routes. Only the subcommand specifier is different.

5.8.3 Querying Routes

To query the routes a ROUTE_VRCB with the appropriate subcommand specifier has to be sent over the API. The Virtual Router returns a list of ROUTE_RECs. The length field on the VRRB header has to be used to calculate for the number of returned routes.

	Result Block
VRRB	result block header
ROUTE_REC[]	records of the routing table

5.9 Filter Setup

Filters are applied to the central forwarding mechanism. Each filter has a couple of fields specifying the wanted packets, a reference to the object where matching packets shall be sent to and a priority. The priority defines in which order the filters are applied to the transported packets. This is important as a filter might remove a matching packet from the network. As a consequence the packet will not reach the following filters.
After all filters have been applied, the packet is processed by the internal routing system. The following commands are defined to setup, list, modify and remove filters.

Filter specific command codes	
0	get a list of filters
2	add a filter
3	remove a filter

A filter is represented by a FILTER_REC structure:

FILTER_REC	**Data Record**
ulong	address of an appropriate ForwarderIF class
ushort	position the filter has to be applied
ulong	source ip address, NAK is 0xfffffffffd
ulong	netmask of the source address
ulong	destination address, NAK is 0xffffffff
ulong	netmask of the destination address
ushort	Type of Service, 0xffff to ignore
ushort	protocol 0xffff to ignore
ushort	IP options type
ushort	ip options value
ushort	filter mode
byte[32]	name of the filter
ushort	handle of the control block the filter was created with

The following list describes the variables in detailed:

address is a four byte wide pointer to an appropriate forwarder class. If the filter is initialised by a loadable module, the loadable module can define a function, which is directly called which an matching packet a argument. If this pointer is set to 0, the filtered packets are sent over the API channel as asynchronous VRRBs.

filter position defines in which order the filters are applied . This can be important as filters can also remove packets. The filters are processed in the order of their position. If a filter with a low position removes a packet, this packet is lost for all filters with higher positions. Packets addressed to the local host are processed at postion 1000. Any filter with an id smaller than 1000 will therefore even process packets directed to the local host.

source ip and source netmask specify the type of source addresses to be matched by the filter. The addresses have to be specified in network byte order. If no source filtering has to be applied use 0 and 0xffffffff as values.

destination ip and netmask same as the above for the destination address.

Type of Service the Type of Service byte to match the filter. 0xffff disables this filter parameter.

protocol This allows to filter all packets of a specific protocol (e.g. UDP, TCP, ICMP ...). 0xffff disables this filter function.

ip option type and value allows to react on special options in the IP header. (e.g. router alert). Set both to 0 to switch this feature off.

filter mode This parameter is a bitfield specifying the behaviour of the filter.

bit	comment
0	remove packet if filter matches
1	reassemble fragments

The appropriate control blocks to remove, list and add filters are described in the following sections.

5.9.1 Adding and Removing Filters

To add or remove a filter an appropriate VRCB has to be sent to the API. The VRCB contains the usual handle, the length field and the command to specify whether the filter has to be removed or added.

	Control Block
VRCB	the control block header with command specifier = 30
ushort	filter specific command
FILTER_REC	filter to be removed or added

The returned structure repeats the FILTER_REC contained in the according control block. On error the call back pointer is set to 0 and the name string of the FILTER_REC contains some short error string.

5.9.2 Query List of Installed Filters

This function can be used to query the actual installed filters. The structure FILTER_CB with the command code `0` results in a list of applied filters.

	Result Block
VRRB	the result block header
FILTER_REC[]	a list of filter records

As usual the length field within the VRRB header can be used to calculate the number of attached filter records.

5.9.3 Adding a Protocol Stack

Protocol stacks are accessed like filters. Either a call back function is specified within the control block or packets matching the protocol id are passed asynchronously via the API channel.

	Control Block
VRCB	the control block header with command specifier = 31
ulong	pointer to the call back function
ushort	protocol id

The Virtual Router result block contains:

	Result Block
VRRB	the result block header
ushort	protocol

On error the protocol field is set to 0;

5.10 Loadable Objects

This type of API calls allow the integration of Loadable Objects (LOBs) into the VR core. These objects are based on a class, derived from the virtual class LOB and are stored in a separate file. For some information about the commands used to load objects from the standard shell see chapter 4.4. The Virtual Router distribution also containes some examples for Loadable Objects, like two different version of a ”Hello World” program and an example for a packet filter.

The loadable object specific commands are listed in the table below:

Loadable object specific command codes	
1	load object from file
2	unload object with specific id
10	get list of information blocks

A central data structure for loadable objects is the LOB_INFO_BLOCK . This is a special structure containing all data about a specific loadable object. This structure is used for API calls querying information or is returned after the loading or unloading of an object.

LOB_INFO_BLOCK	**Data Record**
ushort	an unique id number for the loadable object
ushort	the mode of the loadable object
ulong	the size of the object code
ulong	the time in seconds since the module was loaded
byte[32]	the loadable object’s name

If an error occurs the loadable object id field contains 0xffff and the objects name field a short error string.

5.10.1 Loading an Object

To load an object the filename of the object kernel is forwarded to the API. The API then loads the object and executes its constructor. The API (and therefore also the event scheduler) is blocked as long as the constructor is executed. So no time consuming or blocking code can be executed in the constructor.

The following control block tells the API to load the according loadable object. Since it is possible to pass command line parameters to the loadable object, the control block contains a list of 0-byte separated strings, each token representing a parameter. The first token is the name of the object. If the first token contains a simple filename only, the Virtual Router automatically adds the appropriate pathname and loadable object extension. If the filename contains a ’/’ or a ’.’, the VR will assume, that an absolute pathname is given and will look for the file at the specified location.

	Control Block
VRCB	the usual control block header
ushort	lob specific command, value has to be 1 for the loading of an object
ushort	flags to be passed to the object
byte[]	null-byte separated list of command line arguments for the lob, the first token is the filename

The API will answer this call by a VRRB containing a LOB_INFO_BLOCK.

	Result Block
VRRB	result block header
LOB_INFO_BLOCK	the record with information about the object

Errors are signalled as described above. Since a loadable object might use the API channel it was loaded by during its construction, any loadable object has to be implemented carefully as any synchronisation problem within the object's constructor might harm the later communication on the API channel.

5.10.2 Querying LOB Information

To get some information about the currently loaded modules, a result block containing data about all loaded objects can be queried. This result block contains a list of LOB_INFO_BLOCKs.

	Result Block
VRRB	result block header
LOB_INFO_BLOCK[]	list of information blocks

The length field of the result block header can be used to get the number of loaded objects as usual.

5.11 Querying Scheduler Status

This command requests information about the internal event handler. This is a read only command, therefore (so far) no modification of the central event handling system is possible. However this command at least is useful for debugging purposes.
A VRCB with the command id 21 results in a VRRB containing information about all currently registered events. The structure of the result block looks like:

	Result Block
VRRB	result block header
EVENT_REC	one record per registered event

The records containing the event information have the following format:

EVENT_REC	**Data Record**
ushort	event type (READ,WRITE)
ushort	actual active flags
ushort	the time in 1/1000 seconds, the event is suspended, 0 if active
ushort	associated read file handle
ushort	associated write file handle
byte[16]	id string of the event (e.g. an interface name)
ushort	true if event execution is forced

5.12 Passing IP packets to the Router

	Control Block
VRCB	control block with id 50
ushort	flags concerning the packet handling
ushort	specifies how the packet is treated
ushort	mode dependent parameter
ip packet	the packet to be sent

The API also provides a mechanism to pass IP packets directly to the Virtual Router. For that purpose the IP packet has to be encapsulated into an VRCB. Since there are several possibilities the Virtual Router can handle the packet, a mode parameter has to be specified. Therefore a packet may either be sent using the VR's internal routing rules (see 5.8) or an interface is defined to transmit the packet.

Modes for sending IP packets	
0	The packet is directly processed by the Virtual Router's routing procedures and put to the according interface.
1	The packet is analysed as any other packet that is received by the Virtual Router. The packet is processed by the Virtual Router's protocol stacks and is also analysed and processed by the filters.
2	The Virtual Router puts the received packet directly to the specified interface. The parameter field has to contain a valid Virtual Router interface number.

Additionally, the VRCB contains information about whether an IP packet shall be sent 'as is' or if the Virtual Router shall add certain information within the IP header and recalculate the CRC. The behaviour can be configured by setting certain flags in the flags parameter. The following list gives a description of the currently used flags. Multiple flags may be set.

Flags for special IP treatment	
1	The Virtual Router will control, whether the IP header of the received packet is valid or not. If the header does not comply the standard the packet will be dropped and an error will be returned.
2	The checksum of the packet header will be set correctly
4	The Time To Live field in the packet header will be set by the Virtual Router using its default value.
8	The Virtual Router's address is used as source address of the packet.
16	The received packet will be treated as not fragmented and the header fields will be set accordingly. The Virtual Router will fragment oversized packets in any case, therefore the packet sent over the API can be up to 0xffff bytes.

After the control block with the encapsulated packet has been sent to the Virtual Router a result block is sent back containing information about how the packet was processed and whether any error occurred.

Result Block

VRRB	result block header
ushort	result code

Since the type of error depends on the mode the packet was sent with, the result code field can have multiple values.

Result Codes R for the various modes		
0	R = 0xfff	routing error, no route found
0	R < 0xfff	packet was routed to interface number R
0	R = 0xffff	not an IP packet
1	$0 \leq R \leq$ 0xfff	packet routed to interface number R
1	R = 0xfff	routing error, no route found
1	R = 0x1000	packet processed by local protocol stack
1	R = 0x1001	no matching local protocol stack
1	R = 0x2000	packet dropped due to TTL
1	R = 0xffff	not an IP packet
2	R = 0xfff	no such interface
2	R = $\leq$ 0xfff	packet sent to interface number R
2	R = 0xffff	not an IP packet

Bibliography

[BB00a] Florian Baumgartner and Torsten Braun. Quality of service and active networking on virtual router topologies. In Hiroshi Yasuda, editor, *Active Networks, Second International Working Conference, IWAN*, Lecture Notes in Computer Science, pages 211–224, Tokio, Japan, October 2000. Springer. ISBN 3-540-41179-8.

[BB00b] Florian Baumgartner and Torsten Braun. Virtual routers: A novel approach for qos performance evaluation. In Jon Crowcroft, James Roberts, and Smirnov Mikhail, editors, *Quality of Future Internet Services, First COST 263 International Workshop. QofIS*, Lecture Notes in Computer Science, pages 336–347, Berlin, Germany, September 2000. Springer. ISBN 3-540-41076-7.

[BB01] Florian Baumgartner and Torsten Braun. Distributed emulation of ip networks. Speedup Workshop, University of Berne, March 2001.

[BBC$^+$98] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weis. An architecture for differentiated services. RFC 2475, December 1998.

[FJ93] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, August 1993.

[HBWW99] Juha Heinanen, F. Baker, W. Weiss, and J. Wroclawski. Assured forwarding phb group. RFC 2597, June 1999.

[JNP99] Van Jacobson, K. Nichols, and K. Poduri. An expedited forwarding phb. RFC 2598, June 1999.

[Mis81] Misc. Internet protocol, darpa internet program protocol specification. RFC 791, September 1981.

Index