multicast

MULTICAST(4) OpenBSD Programmer's Manual MULTICAST(4)

NAME

multicast - Multicast Routing

SYNOPSIS

options MROUTING

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip_mroute.h>
#include <netinet6/ip6_mroute.h>

int
getsockopt(*int s*, *IPPROTO_IP*, *MRT_INIT*, *void *optval*, *socklen_t *optlen*);

int
setsockopt(*int s*, *IPPROTO_IP*, *MRT_INIT*, *const void *optval*, *socklen_t optlen*);

int
getsockopt(*int s*, *IPPROTO_IPV6*, *MRT6_INIT*, *void *optval*, *socklen_t *optlen*);

int
setsockopt(*int s*, *IPPROTO_IPV6*, *MRT6_INIT*, *const void *optval*, *socklen_t optlen*);

DESCRIPTION

Multicast routing is used to efficiently propagate data packets to a set of multicast listeners in multipoint networks. If unicast is used to replicate the data to all listeners, then some of the network links may carry multiple copies of the same data packets. With multicast routing, the overhead is reduced to one copy (at most) per network link.

All multicast-capable routers must run a common multicast routing protocol. The Distance Vector Multicast Routing Protocol (DVMRP) was the first developed multicast routing protocol. Later, other protocols such as Multicast Extensions to OSPF (MOSPF), Core Based Trees (CBT), Protocol Independent Multicast - Sparse Mode (PIM-SM), and Protocol Independent Multicast - Dense Mode (PIM-DM) were developed as well.

To start multicast routing, the user must enable multicast forwarding via the **sysctl(8)** variables *net.inet.ip.mforwarding* and/or *net.inet.ip6.mforwarding*. The user must also run a multicast routing capable user-level process, such as **mrouted(8)**. From a developer's point of view, the programming guide described in the *Programming Guide* section should be used to control the multicast forwarding in the kernel.

Programming Guide

This section provides information about the basic multicast routing API. The so-called ``advanced multicast API'' is described in the *Advanced Multicast API Programming Guide* section.

First, a multicast routing socket must be open. That socket would be used to control the multicast forwarding in the kernel. Note that most operations below require certain privilege (i.e., root privilege):

```
/* IPv4 */
int mrouter_s4;
mrouter_s4 = socket(AF_INET, SOCK_RAW, IPPROTO_IGMP);

int mrouter_s6;
mrouter_s6 = socket(AF_INET6, SOCK_RAW, IPPROTO_ICMPV6);
```

Note that if the router needs to open an IGMP or ICMPv6 socket (IPv4 or IPv6, respectively) for sending or receiving of IGMP or MLD multicast group membership messages, then the same *mrouter_s4* or *mrouter_s6* sockets should be used for sending and receiving respectively IGMP or MLD messages. In the case of BSD-derived kernels, it may be possible to open separate sockets for IGMP or MLD messages only. However, some other kernels (e.g., Linux) require that the multicast routing socket must be used for sending and receiving of IGMP or MLD messages. Therefore, for portability reasons, the multicast routing socket should be reused for IGMP and MLD messages as well.

After the multicast routing socket is open, it can be used to enable or disable multicast forwarding in the kernel:

```
/* IPv4 */
int v = 1;        /* 1 to enable, or 0 to disable */
setsockopt(mrouter_s4, IPPROTO_IP, MRT_INIT, (void *)&v, sizeof(v));

/* IPv6 */
int v = 1;        /* 1 to enable, or 0 to disable */
setsockopt(mrouter_s6, IPPROTO_IPV6, MRT6_INIT, (void *)&v, sizeof(v));
...
/* If necessary, filter all ICMPv6 messages */
struct icmp6_filter filter;
ICMP6_FILTER_SETBLOCKALL(&filter);
setsockopt(mrouter_s6, IPPROTO_ICMPV6, ICMP6_FILTER, (void *)&filter,
           sizeof(filter));
```

After multicast forwarding is enabled, the multicast routing socket can be used to enable PIM processing in the kernel if either PIM-SM or PIM-DM are being used (see **pim(4)**).

For each network interface (e.g., physical or a virtual tunnel) that would be used for multicast forwarding, a corresponding multicast interface must be added to the kernel:

```
/* IPv4 */
struct vifctl vc;
```

```
    memset(&vc, 0, sizeof(vc));
    /* Assign all vifctl fields as appropriate */
    vc.vifc_vifi = vif_index;
    vc.vifc_flags = vif_flags;
    vc.vifc_threshold = min_ttl_threshold;
    vc.vifc_rate_limit = max_rate_limit;
    memcpy(&vc.vifc_lcl_addr, &vif_local_address, sizeof(vc.vifc_lcl_addr));
    if (vc.vifc_flags & VIFF_TUNNEL)
        memcpy(&vc.vifc_rmt_addr, &vif_remote_address,
               sizeof(vc.vifc_rmt_addr));
    setsockopt(mrouter_s4, IPPROTO_IP, MRT_ADD_VIF, (void *)&vc,
               sizeof(vc));
```

The *vif_index* must be unique per vif. The *vif_flags* contains the VIFF_* flags as defined in <*netinet/ip_mroute.h*>. The *min_ttl_threshold* contains the minimum TTL a multicast data packet must have to be forwarded on that vif. Typically, it would be 1. The *max_rate_limit* contains the maximum rate (in bits/s) of the multicast data packets forwarded on that vif. A value of 0 means no limit. The *vif_local_address* contains the local IP address of the corresponding local interface. The *vif_remote_address* contains the remote IP address for DVMRP multicast tunnels.

```
      /* IPv6 */
      struct mif6ctl mc;
      memset(&mc, 0, sizeof(mc));
      /* Assign all mif6ctl fields as appropriate */
      mc.mif6c_mifi = mif_index;
      mc.mif6c_flags = mif_flags;
      mc.mif6c_pifi = pif_index;
      setsockopt(mrouter_s6, IPPROTO_IPV6, MRT6_ADD_MIF, (void *)&mc,
                 sizeof(mc));
```

The *mif_index* must be unique per vif. The *mif_flags* contains the MIFF_* flags as defined in <*netinet6/ip6_mroute.h*>. The *pif_index* is the physical interface index of the corresponding local interface.

A multicast interface is deleted by:

```
      /* IPv4 */
      vifi_t vifi = vif_index;
      setsockopt(mrouter_s4, IPPROTO_IP, MRT_DEL_VIF, (void *)&vifi,
                 sizeof(vifi));

      /* IPv6 */
      mifi_t mifi = mif_index;
      setsockopt(mrouter_s6, IPPROTO_IPV6, MRT6_DEL_MIF, (void *)&mifi,
                 sizeof(mifi));
```

After multicast forwarding is enabled, and the multicast virtual interfaces have been added, the kernel may deliver upcall messages (also called signals later in this text) on the multicast routing socket that was open earlier with MRT_INIT or MRT6_INIT. The IPv4 upcalls have a *struct igmpmsg* header (see <*netinet/ip_mroute.h*>) with the *im_mbz* field set to zero. Note that this header follows the structure of *struct ip* with the protocol field *ip_p* set to zero. The IPv6 upcalls have a *struct mrt6msg* header (see <*netinet6/ip6_mroute.h*>) with the *im6_mbz* field set to zero. Note that this header follows the structure of *struct ip6_hdr* with the next header field *ip6_nxt* set to zero.

The upcall header contains the *im_msgtype* and *im6_msgtype* fields, with the type of the upcall IGMPMSG_* and MRT6MSG_* for IPv4 and IPv6, respectively. The values of the rest of the upcall header fields and the body of the upcall message depend on the particular upcall type.

If the upcall message type is IGMPMSG_NOCACHE or MRT6MSG_NOCACHE, this is an indication that a multicast packet has reached the multicast router, but the router has no forwarding state for that packet. Typically, the upcall would be a signal for the multicast routing user-level process to install the appropriate Multicast Forwarding Cache (MFC) entry in the kernel.

An MFC entry is added by:

```
/* IPv4 */
struct mfcctl mc;
memset(&mc, 0, sizeof(mc));
memcpy(&mc.mfcc_origin, &source_addr, sizeof(mc.mfcc_origin));
memcpy(&mc.mfcc_mcastgrp, &group_addr, sizeof(mc.mfcc_mcastgrp));
mc.mfcc_parent = iif_index;
for (i = 0; i < maxvifs; i++)
    mc.mfcc_ttls[i] = oifs_ttl[i];
setsockopt(mrouter_s4, IPPROTO_IP, MRT_ADD_MFC,
           (void *)&mc, sizeof(mc));

/* IPv6 */
struct mf6cctl mc;
memset(&mc, 0, sizeof(mc));
memcpy(&mc.mf6cc_origin, &source_addr, sizeof(mc.mf6cc_origin));
memcpy(&mc.mf6cc_mcastgrp, &group_addr, sizeof(mf6cc_mcastgrp));
mc.mf6cc_parent = iif_index;
for (i = 0; i < maxvifs; i++)
    if (oifs_ttl[i] > 0)
        IF_SET(i, &mc.mf6cc_ifset);
setsockopt(mrouter_s4, IPPROTO_IPV6, MRT6_ADD_MFC,
           (void *)&mc, sizeof(mc));
```

The *source_addr* and *group_addr* fields are the source and group address of the multicast packet (as set in the upcall message). The *iif_index* is the virtual interface index of the multicast interface the multicast packets for this specific source and group address should be received on. The *oifs_ttl[]* array contains the minimum TTL (per interface) a multicast packet should have to be forwarded on an outgoing interface. If the TTL value is zero, the corresponding interface is not included in the set of outgoing interfaces. Note that for IPv6 only the set of outgoing interfaces can be specified.

An MFC entry is deleted by:

```
/* IPv4 */
struct mfcctl mc;
memset(&mc, 0, sizeof(mc));
memcpy(&mc.mfcc_origin, &source_addr, sizeof(mc.mfcc_origin));
memcpy(&mc.mfcc_mcastgrp, &group_addr, sizeof(mc.mfcc_mcastgrp));
setsockopt(mrouter_s4, IPPROTO_IP, MRT_DEL_MFC,
           (void *)&mc, sizeof(mc));

/* IPv6 */
struct mf6cctl mc;
memset(&mc, 0, sizeof(mc));
memcpy(&mc.mf6cc_origin, &source_addr, sizeof(mc.mf6cc_origin));
memcpy(&mc.mf6cc_mcastgrp, &group_addr, sizeof(mf6cc_mcastgrp));
setsockopt(mrouter_s4, IPPROTO_IPV6, MRT6_DEL_MFC,
           (void *)&mc, sizeof(mc));
```

The following method can be used to get various statistics per installed MFC entry in the kernel (e.g., the number of forwarded packets per source and group address):

```
/* IPv4 */
struct sioc_sg_req sgreq;
memset(&sgreq, 0, sizeof(sgreq));
memcpy(&sgreq.src, &source_addr, sizeof(sgreq.src));
memcpy(&sgreq.grp, &group_addr, sizeof(sgreq.grp));
ioctl(mrouter_s4, SIOCGETSGCNT, &sgreq);

/* IPv6 */
struct sioc_sg_req6 sgreq;
memset(&sgreq, 0, sizeof(sgreq));
memcpy(&sgreq.src, &source_addr, sizeof(sgreq.src));
memcpy(&sgreq.grp, &group_addr, sizeof(sgreq.grp));
ioctl(mrouter_s6, SIOCGETSGCNT_IN6, &sgreq);
```

The following method can be used to get various statistics per multicast virtual interface in the kernel (e.g., the number of forwarded packets per interface):

```
/* IPv4 */
struct sioc_vif_req vreq;
memset(&vreq, 0, sizeof(vreq));
vreq.vifi = vif_index;
ioctl(mrouter_s4, SIOCGETVIFCNT, &vreq);

/* IPv6 */
struct sioc_mif_req6 mreq;
memset(&mreq, 0, sizeof(mreq));
mreq.mifi = vif_index;
ioctl(mrouter_s6, SIOCGETMIFCNT_IN6, &mreq);
```

Advanced Multicast API Programming Guide

Adding new features to the kernel makes it difficult to preserve backward compatibility (binary and API), and at the same time to allow user-level processes to take advantage of the new features (if the kernel supports them).

One of the mechanisms that allows preserving the backward compatibility is a sort of negotiation between the user-level process and the kernel:

1. The user-level process tries to enable in the kernel the set of new features (and the corresponding API) it would like to use.

2. The kernel returns the (sub)set of features it knows about and is willing to be enabled.

3. The user-level process uses only that set of features the kernel has agreed on.

To support backward compatibility, if the user-level process does not ask for any new features, the kernel defaults to the basic multicast API (see the *Programming Guide* section). Currently, the advanced multicast API exists only for IPv4; in the future there will be IPv6 support as well.

Below is a summary of the expandable API solution. Note that all new options and structures are defined in <*netinet/ip_mroute.h*> and <*netinet6/ip6_mroute.h*>, unless stated otherwise.

The user-level process uses new **getsockopt**()/**setsockopt**() options to perform the API features negotiation with the kernel. This negotiation must be performed right after the multicast routing socket is open. The set of desired/allowed features is stored in a bitset (currently, in *uint32_t* i.e., maximum of 32 new features). The new **getsockopt**()/**setsockopt**() options are MRT_API_SUPPORT and MRT_API_CONFIG. An example:

```
   uint32_t v;
   getsockopt(sock, IPPROTO_IP, MRT_API_SUPPORT, (void *)&v, sizeof(v));
```

This would set *v* to the pre-defined bits that the kernel API supports. The eight least significant bits in *uint32_t* are the same as the eight possible flags MRT_MFC_FLAGS_* that can be used in *mfcc_flags* as part of the new definition of *struct mfcctl* (see below about those flags), which leaves 24 flags for other new features. The value returned by **getsockopt**(*MRT_API_SUPPORT*) is read-only; in other words, **setsockopt**(*MRT_API_SUPPORT*) would fail.

To modify the API, and to set some specific feature in the kernel, then:

```
   uint32_t v = MRT_MFC_FLAGS_DISABLE_WRONGVIF;
   if (setsockopt(sock, IPPROTO_IP, MRT_API_CONFIG, (void *)&v, sizeof(v))
       != 0) {
       return (ERROR);
   }
   if (v & MRT_MFC_FLAGS_DISABLE_WRONGVIF)
       return (OK);       /* Success */
   else
       return (ERROR);
```

In other words, when **setsockopt**(*MRT_API_CONFIG*) is called, the argument to it specifies the desired set of features to be enabled in the API and the kernel. The return value in *v* is the actual (sub)set of features that were enabled in the kernel. To obtain later the same set of features that were enabled, use:

```
      getsockopt(sock, IPPROTO_IP, MRT_API_CONFIG, (void *)&v, sizeof(v));
```

The set of enabled features is global. In other words, **setsockopt**(*MRT_API_CONFIG*) should be called right after **setsockopt**(*MRT_INIT*).

Currently, the following set of new features is defined:

```
#define MRT_MFC_FLAGS_DISABLE_WRONGVIF (1 << 0)/*disable WRONGVIF signals*/
#define MRT_MFC_FLAGS_BORDER_VIF   (1 << 1)  /* border vif              */
#define MRT_MFC_RP                 (1 << 8)  /* enable RP address       */
#define MRT_MFC_BW_UPCALL          (1 << 9)  /* enable bw upcalls       */
```

The advanced multicast API uses a newly defined *struct mfcctl2* instead of the traditional *struct mfcctl*. The original *struct mfcctl* is kept as is. The new *struct mfcctl2* is:

```
/*
 * The new argument structure for MRT_ADD_MFC and MRT_DEL_MFC overlays
 * and extends the old struct mfcctl.
 */
struct mfcctl2 {
        /* the mfcctl fields */
        struct in_addr  mfcc_origin;       /* ip origin of mcasts       */
        struct in_addr  mfcc_mcastgrp;     /* multicast group associated*/
```

```
        vifi_t          mfcc_parent;       /* incoming vif              */
        u_char          mfcc_ttls[MAXVIFS];/* forwarding ttls on vifs   */

        /* extension fields */
        uint8_t         mfcc_flags[MAXVIFS];/* the MRT_MFC_FLAGS_* flags*/
        struct in_addr  mfcc_rp;            /* the RP address           */
};
```

The new fields are *mfcc_flags[MAXVIFS]* and *mfcc_rp*. Note that for compatibility reasons they are added at the end.

The *mfcc_flags[MAXVIFS]* field is used to set various flags per interface per (S,G) entry. Currently, the defined flags are:

```
#define MRT_MFC_FLAGS_DISABLE_WRONGVIF (1 << 0)/*disable WRONGVIF signals*/
#define MRT_MFC_FLAGS_BORDER_VIF       (1 << 1) /* border vif          */
```

The MRT_MFC_FLAGS_DISABLE_WRONGVIF flag is used to explicitly disable the IGMPMSG_WRONGVIF kernel signal at the (S,G) granularity if a multicast data packet arrives on the wrong interface. Usually this signal is used to complete the shortest-path switch for PIM-SM multicast routing, or to trigger a PIM assert message. However, it should not be delivered for interfaces that are not set in the outgoing interface, and that are not expecting to become an incoming interface. Hence, if the MRT_MFC_FLAGS_DISABLE_WRONGVIF flag is set for some of the interfaces, then a data packet that arrives on that interface for that MFC entry will NOT trigger a WRONGVIF signal. If that flag is not set, then a signal is triggered (the default action).

The MRT_MFC_FLAGS_BORDER_VIF flag is used to specify whether the Border-bit in PIM Register messages should be set (when the Register encapsulation is performed inside the kernel). If it is set for the special PIM Register kernel virtual interface (see **pim(4)**), the Border-bit in the Register messages sent to the RP will be set.

The remaining six bits are reserved for future usage.

The *mfcc_rp* field is used to specify the RP address (for PIM-SM multicast routing) for a multicast group G if we want to perform kernel-level PIM Register encapsulation. The *mfcc_rp* field is used only if the MRT_MFC_RP advanced API flag/capability has been successfully set by **setsockopt**(*MRT_API_CONFIG*).

If the MRT_MFC_RP flag was successfully set by **setsockopt**(*MRT_API_CONFIG*), then the kernel will attempt to perform the PIM Register encapsulation itself instead of sending the multicast data packets to user level (inside IGMPMSG_WHOLEPKT upcalls) for user-level encapsulation. The RP address would be taken from the *mfcc_rp* field inside the new *struct mfcctl2*. However, even if the MRT_MFC_RP flag was successfully set, if the *mfcc_rp* field was set to INADDR_ANY, then the kernel will still deliver an IGMPMSG_WHOLEPKT upcall with the multicast data packet to the user-level process.

In addition, if the multicast data packet is too large to fit within a single IP packet after the PIM Register encapsulation (e.g., if its size was on the order of 65500 bytes), the data packet will be fragmented, and then each of the fragments will be encapsulated separately. Note that typically a multicast data packet can be that large only if it was originated locally from the same hosts that performs the encapsulation; otherwise the transmission of the multicast data packet over Ethernet for example would have fragmented it into much smaller pieces.

Typically, a multicast routing user-level process would need to know the forwarding bandwidth for some data flow. For example, the multicast routing process may want to time out idle MFC entries, or for PIM-SM it can initiate (S,G) shortest-path switch if the bandwidth rate is above a threshold for example.

The original solution for measuring the bandwidth of a dataflow was that a user-level process would periodically query the kernel about the number of forwarded packets/bytes per (S,G), and then based on those numbers it would estimate whether a source has been idle, or whether the source's transmission bandwidth is above a threshold. That solution is far from being scalable, hence the need for a new mechanism for bandwidth monitoring.

Below is a description of the bandwidth monitoring mechanism.

- If the bandwidth of a data flow satisfies some pre-defined filter, the kernel delivers an upcall on the multicast routing socket to the multicast routing process that has installed that filter.

- The bandwidth-upcall filters are installed per (S,G). There can be more than one filter per (S,G).

- Instead of supporting all possible comparison operations (i.e., < <= == != > >=), there is support only for the <= and >= operations, because this makes the kernel-level implementation simpler, and because practically we need only those two. Furthermore, the missing operations can be simulated by secondary user-level filtering of those <= and >= filters. For example, to simulate !=, then we need to install filter ``bw <= 0xffffffff'', and after an upcall is received, we need to check whether ``measured_bw != expected_bw''.

- The bandwidth-upcall mechanism is enabled by **setsockopt**(*MRT_API_CONFIG*) for the MRT_MFC_BW_UPCALL flag.

- The bandwidth-upcall filters are added/deleted by the new **setsockopt**(*MRT_ADD_BW_UPCALL*) and **setsockopt**(*MRT_DEL_BW_UPCALL*) respectively (with the appropriate *struct bw_upcall* argument of course).

From an application point of view, a developer needs to know about the following:

```
/*
 * Structure for installing or delivering an upcall if the
 * measured bandwidth is above or below a threshold.
 *
 * User programs (e.g. daemons) may have a need to know when the
 * bandwidth used by some data flow is above or below some threshold.
 * This interface allows the userland to specify the threshold (in
 * bytes and/or packets) and the measurement interval. Flows are
 * all packet with the same source and destination IP address.
 * At the moment the code is only used for multicast destinations
 * but there is nothing that prevents its use for unicast.
 *
 * The measurement interval cannot be shorter than some Tmin (3s).
 * The threshold is set in packets and/or bytes per_interval.
 *
 * Measurement works as follows:
 *
 * For >= measurements:
 * The first packet marks the start of a measurement interval.
```

```
 * During an interval we count packets and bytes, and when we
 * pass the threshold we deliver an upcall and we are done.
 * The first packet after the end of the interval resets the
 * count and restarts the measurement.
 *
 * For <= measurement:
 * We start a timer to fire at the end of the interval, and
 * then for each incoming packet we count packets and bytes.
 * When the timer fires, we compare the value with the threshold,
 * schedule an upcall if we are below, and restart the measurement
 * (reschedule timer and zero counters).
 */

struct bw_data {
        struct timeval  b_time;
        uint64_t        b_packets;
        uint64_t        b_bytes;
};

struct bw_upcall {
        struct in_addr  bu_src;         /* source address            */
        struct in_addr  bu_dst;         /* destination address       */
        uint32_t        bu_flags;       /* misc flags (see below)    */
#define BW_UPCALL_UNIT_PACKETS (1 << 0) /* threshold (in packets)    */
#define BW_UPCALL_UNIT_BYTES   (1 << 1) /* threshold (in bytes)      */
#define BW_UPCALL_GEQ          (1 << 2) /* upcall if bw >= threshold */
#define BW_UPCALL_LEQ          (1 << 3) /* upcall if bw <= threshold */
#define BW_UPCALL_DELETE_ALL   (1 << 4) /* delete all upcalls for s,d*/
        struct bw_data  bu_threshold;   /* the bw threshold          */
        struct bw_data  bu_measured;    /* the measured bw           */
};

/* max. number of upcalls to deliver together */
#define BW_UPCALLS_MAX                          128
/* min. threshold time interval for bandwidth measurement */
#define BW_UPCALL_THRESHOLD_INTERVAL_MIN_SEC    3
#define BW_UPCALL_THRESHOLD_INTERVAL_MIN_USEC   0
```

The *bw_upcall* structure is used as an argument to **setsockopt**(*MRT_ADD_BW_UPCALL*) and **setsockopt**(*MRT_DEL_BW_UPCALL*). Each **setsockopt**(*MRT_ADD_BW_UPCALL*) installs a filter in the kernel for the source and destination address in the *bw_upcall* argument, and that filter will trigger an upcall according to the following pseudo-algorithm:

```
 if (bw_upcall_oper IS ">=") {
    if (((bw_upcall_unit & PACKETS == PACKETS) &&
         (measured_packets >= threshold_packets)) ||
        ((bw_upcall_unit & BYTES == BYTES) &&
         (measured_bytes >= threshold_bytes)))
       SEND_UPCALL("measured bandwidth is >= threshold");
  }
  if (bw_upcall_oper IS "<=" && measured_interval >= threshold_interval) {
    if (((bw_upcall_unit & PACKETS == PACKETS) &&
         (measured_packets <= threshold_packets)) ||
        ((bw_upcall_unit & BYTES == BYTES) &&
         (measured_bytes <= threshold_bytes)))
       SEND_UPCALL("measured bandwidth is <= threshold");
  }
```

In the same *bw_upcall*, the unit can be specified in both BYTES and PACKETS. However, the GEQ and LEQ flags are mutually exclusive.

Basically, an upcall is delivered if the measured bandwidth is >= or <= the threshold bandwidth (within the specified measurement interval). For practical reasons, the smallest value for the measurement interval is 3 seconds. If smaller values are allowed, then the bandwidth estimation may be less accurate, or the potentially very high frequency of the generated upcalls may introduce too much overhead. For the >= operation, the answer may be known before the end of *threshold_interval*, therefore the upcall may be delivered earlier. For the <= operation however, we must wait until the threshold interval has expired to know the answer.

EXAMPLES

```
struct bw_upcall bw_upcall;
/* Assign all bw_upcall fields as appropriate */
memset(&bw_upcall, 0, sizeof(bw_upcall));
memcpy(&bw_upcall.bu_src, &source, sizeof(bw_upcall.bu_src));
memcpy(&bw_upcall.bu_dst, &group, sizeof(bw_upcall.bu_dst));
bw_upcall.bu_threshold.b_data = threshold_interval;
bw_upcall.bu_threshold.b_packets = threshold_packets;
bw_upcall.bu_threshold.b_bytes = threshold_bytes;
if (is_threshold_in_packets)
    bw_upcall.bu_flags |= BW_UPCALL_UNIT_PACKETS;
if (is_threshold_in_bytes)
    bw_upcall.bu_flags |= BW_UPCALL_UNIT_BYTES;
do {
    if (is_geq_upcall) {
        bw_upcall.bu_flags |= BW_UPCALL_GEQ;
        break;
    }
    if (is_leq_upcall) {
        bw_upcall.bu_flags |= BW_UPCALL_LEQ;
        break;
    }
    return (ERROR);
} while (0);
setsockopt(mrouter_s4, IPPROTO_IP, MRT_ADD_BW_UPCALL,
          (void *)&bw_upcall, sizeof(bw_upcall));
```

To delete a single filter, use MRT_DEL_BW_UPCALL, and the fields of bw_upcall must be set to exactly same as when MRT_ADD_BW_UPCALL was called.

To delete all bandwidth filters for a given (S,G), then only the *bu_src* and *bu_dst* fields in *struct bw_upcall* need to be set, and then just set only the BW_UPCALL_DELETE_ALL flag inside field *bw_upcall.bu_flags*.

The bandwidth upcalls are received by aggregating them in the new upcall message:

```
#define IGMPMSG_BW_UPCALL  4  /* BW monitoring upcall */
```

This message is an array of *struct bw_upcall* elements (up to BW_UPCALLS_MAX = 128). The upcalls are delivered when there are 128 pending upcalls, or when 1 second has expired since the previous upcall (whichever comes first). In an *struct upcall* element, the *bu_measured* field is filled in to indicate the particular measured values. However, because of the way the particular intervals are measured, the user should be careful how *bu_measured.b_time* is used. For example, if the filter is installed to trigger an upcall if the number of packets is >= 1, then *bu_measured* may have a value of zero in the upcalls after the first one, because the measured interval for >= filters is ``clocked'' by the forwarded packets. Hence, this upcall mechanism should not be used for measuring the exact value of the bandwidth of the forwarded data. To measure the exact bandwidth, the user would need to get the forwarded packets statistics with the **ioctl**(*SIOCGETSGCNT*) mechanism (see the *Programming Guide* section) .

Note that the upcalls for a filter are delivered until the specific filter is deleted, but no more frequently than once per *bu_threshold.b_time*. For example, if the filter is specified to deliver a signal if bw >= 1 packet, the first packet will trigger a signal, but the next upcall will be triggered no earlier than *bu_threshold.b_time* after the previous upcall.

SEE ALSO

getsockopt(2), **recvfrom(2)**, **recvmsg(2)**, **setsockopt(2)**, **socket(2)**, **icmp6(4)**, **inet(4)**, **inet6(4)**, **intro(4)**, **ip(4)**, **ip6(4)**, **pim(4)**, **mrouted(8)**, **sysctl(8)**

AUTHORS

The original multicast code was written by David Waitzman (BBN Labs), and later modified by the following individuals: Steve Deering (Stanford), Mark J. Steiglitz (Stanford), Van Jacobson (LBL), Ajit Thyagarajan (PARC), Bill Fenner (PARC).

The IPv6 multicast support was implemented by the KAME project (*http://www.kame.net*), and was based on the IPv4 multicast code. The advanced multicast API and the multicast bandwidth monitoring were implemented by Pavlin Radoslavov (ICSI) in collaboration with Chris Brown (NextHop).

This manual page was written by Pavlin Radoslavov (ICSI).

OpenBSD 5.1 May 31, 2007 OpenBSD 5.1
