Protocol Independent Multicast-Sparse Mode (PIM-SM): Implementation Document

Ahmed Helmy

Computer Science Department/ISI
University of Southern California
Los Angeles, CA 90089
ahelmy@usc.edu

PIM-SM-implementation.ps

January 19, 1997

Abstract

This document describes the details of the PIM-SM [1, 2, 3] version 2 implementation for UNIX platforms; namely SunOS and SGI-IRIX. A generic kernel model is adopted, which is protocol independent, however some supporting functions are added to the kernel for encapsulation of data packets at user level and decapsulation of PIM Registers.

Further, the basic model for the user level, PIM daemon (pimd), implementation is described.

Implementation details and code are included in supplementary appendices.

1 Introduction

In order to support multicast routing protocols in a UNIX environment, both the kernel and the daemon parts have to interact and cooperate in performing the processing and forwarding functions.

The kernel basically handles forwarding the data packets according to a multicast forwarding cache (MFC) stored in the kernel. While the protocol specific daemon is responsible for processing the control messages from other routers and from the kernel, and maintaining the multicast forwarding cache from user level through traps and system calls. The daemon takes care of all timers for the multicast routing table (MRT) entries according to the specifications of the multicast protocol; PIM-SMv2 [3].

The details of the implementation are presented as follows. First, an overview of the system (kernel and daemon) is given, with emphasis on the basic functions of each. Then, a structural model is presented for the daemon, outlining the basic building blocks for the multicast routing table and the virtual interface table at user space. An illustrative functional description is given, thereafter, for the daemon-kernel interface, and the kernel. Finally, supplementary appendices provide more detailed information about the implementation specifics[1].

[1]The models discussed herein are merely illustrative, and by no means are they exhaustive nor authoritative.

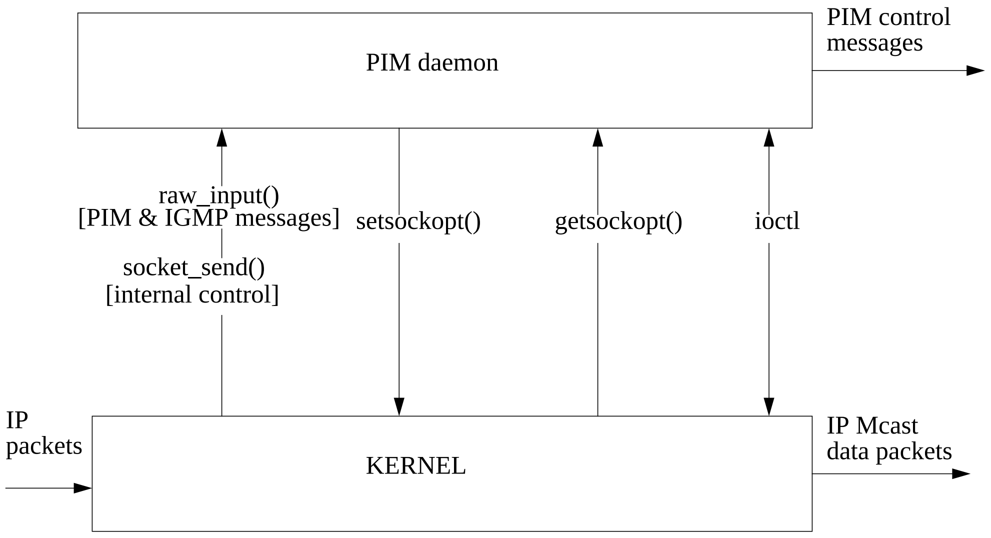

Figure 1: System overview

2 System Overview

The PIM daemon processes all the PIM control messages and sets up an appropriate kernel environment to deliver multicast data packets. The kernel has to support multicast packets forwarding (see figure 1).

2.1 The kernel level

When the kernel receives an IP packet, it passes it through the IP handling routine [ip-intr()]. ip-intr dispatches the packet to the appropriate handling machinery, based on the destination address and the IP protocol number. Here, we are only concerned with the following cases:

- If the packet is a multicast packet then it passes through the multicast forwarding machinery in the kernel [ip-mforward()]. Subsequently, if there is a matching entry (source and group addresses), with the right incoming interface (iif), we get a 'cache hit', and the packet is forwarded to the corresponding outgoing interfaces (oifs) through the fast forwarding path. Otherwise, if the packet does not match the source, group, and iif, we get a 'cache miss', and an internal control message is passed up accordingly to the daemon for processing.

- If the IP packet is a PIM packet (i.e. has protocol number of IPPROTO-PIM), it passes through the PIM machinery in the kernel [pim-input()], and in turn is passed up to the socket queue using the raw-input() call.

- If the IP packet is an IGMP packet (i.e. has protocol number of IPPROTO-IGMP), it passes through the IGMP machinery in the kernel [igmp-input()], and in turn is passed up to the socket queue using the raw-input() call.

2.2 The user level (daemon)

All PIM, IGMP and internal control (e.g. cache miss and wrong incoming interface) messages are passed to PIM daemon; the daemon has the complete information to creat multicast routing table (MRT). It also updates the multicast forwarding cache (MFC) inside the kernel by using the 'setsockopt()' system call, to facilitate multicast packets forwarding (see figure 2).

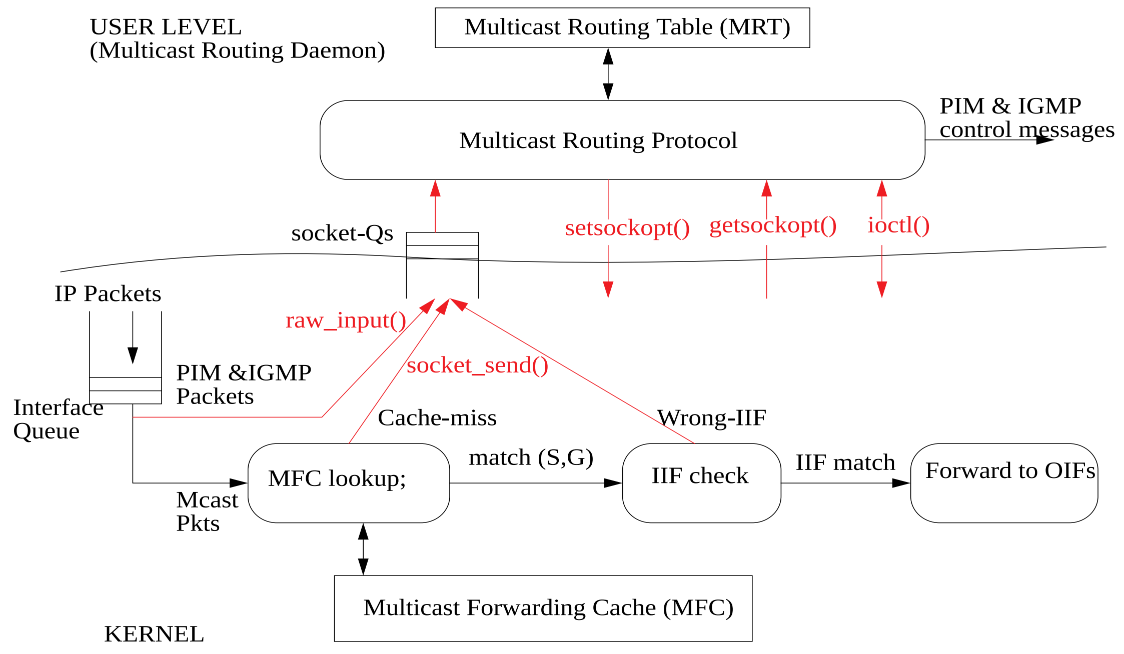

Figure 2: IP-multicast implementation model

The PIM daemon listens to PIM and IGMP sockets and receives both PIM and IGMP messages. The messages are processed by the corresponding machinery in the daemon [accept-pim() and accept-igmp(), respectively], and dispatched to the right component of the protocol.

Modifications and updates are made to the multicast routing table (MRT) according to the processing, and appropriate changes are reflected in the kernel entries using the setsockopt() system call .

Other messages may be triggered off of the state kept in the daemon, to be processed by upstream/downstream routers.

3 User-level Implementation (The Multicast Routing Daemon)

The basic functional flow model for the PIM daemon is described in this section (see figure 3).

The user level implementation is broken up into modules based on functionality, and includes modules to handle the multicast routing table (MRT) [in mrt.c], the virtual interface (vif) table [in vif.c], PIM & IGMP messages [in pim.c & igmp.c], protocol specific routing functions [in route.c], timers and housekeeping [in timer.c], kernel level interface [in kern.c],etc.

Following is an explanation of these modules, in addition to the data structures.

3.1 Data Structures

There are two basic data structures, the multicast routing entry (mrtentry) and virtual interface entry (vifentry). These structures are created and modified in the daemon by receiving the PIM control messages. The definitions of the data structures are given in 'pim.h' file.

3.1.1 Multicast Routing Table

The multicast routing entry is shown below :

```
struct mrtentry{
    struct srcentry *source;     /* source                         */
```

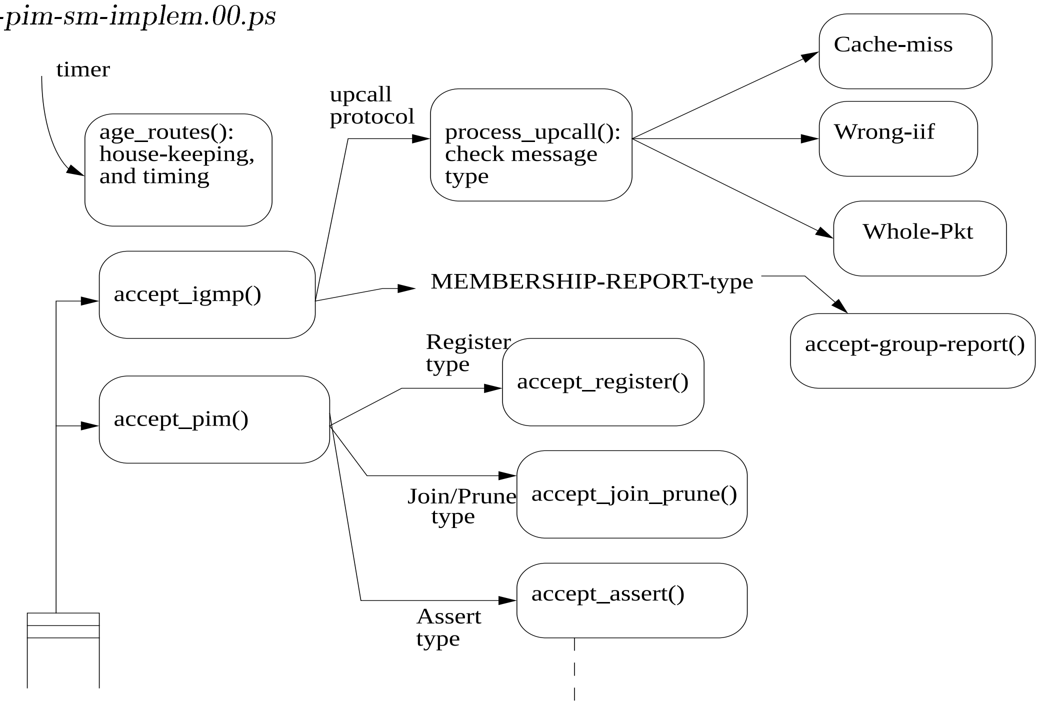

Figure 3: Basic functional flow of the PIM daemon

```
    struct grpentry *group;      /* group                        */
    vifbitmap-t      outgoing;   /* outgoing vifs to downstream  */
    vifbitmap-t      vifmaskoff; /* deleted vifs                 */
    struct mrtentry *srcnext;    /* next entry of same source    */
    struct mrtentry *srcprev;    /* prev entry of same source    */
    struct mrtentry *grpnext;    /* next entry of same group     */
    struct nbrentry *upstream;   /* upstream router, needed as in
                                  * RPbit entry, the upstream
                                  * router is different than
                                  * the source upstream router.
                                  */
    u-long          pktrate-prev;/* packet count of prev check     */
    u-long          idlecnt-prev;/* pkt cnt to check idle states   */
    u-int           data-rate-timer;/* keep track of the data rate */
    u-int           reg-rate-timer;/* keep track of Register rate at
                                    * RP
                                    */
    u-int           reg-cnt;       /* keep track of the Register
                                    * count at RP
                                    */
    u-char          *timers;       /* vif timer list               */
    u-short         timer;         /* entry timer                  */
    u-short         join-prune-timer; /* periodic join/prune timer  */
    u-short         flags;         /* flags                        */
    u-int           assert-rpf-timer;/* Assert timer               */
    u-short         registerBitTimer;/* Register-Suppression timer  */
};
```

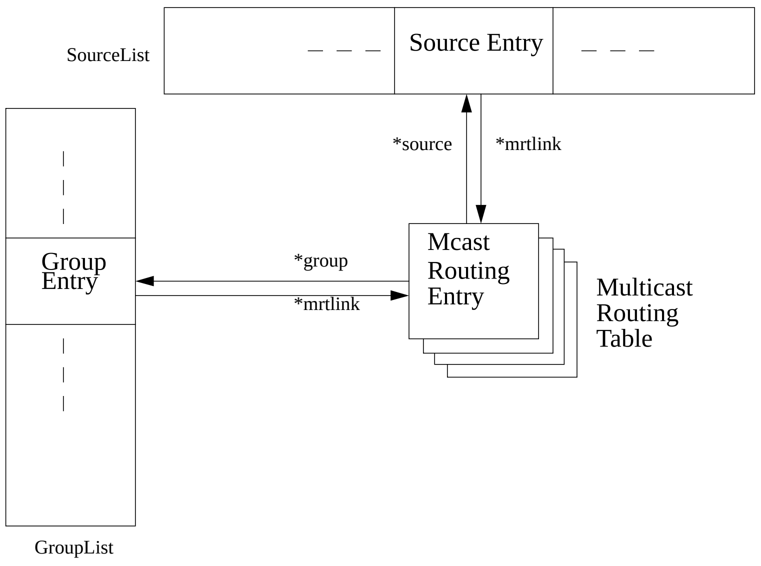

Figure 4: The multicast routing table overall structure at user level

```
struct srcentry {
    u-long           source;     /* subnet source of multicasts         */
    vifi-t           incoming;   /* incoming vif                        */
    struct nbrentry *upstream;   /* upstream router                     */
    u-short          timer;      /* timer for recompute incoming        */
    u-long           metric;     /* Unicast Routing Metric for source   */
    u-long           preference; /* The metric preference value         */
    struct mrtentry *mrtlink;    /* link to routing entries             */
    struct srcentry *next;       /* link to next entry                  */
};

struct grpentry {
    u-long           group;      /* subnet group of multicasts          */
    vifbitmap-t      leaves;     /* outgoing vif to host                */
    u-char           *timers;    /* vif timer list                      */
    struct mrtentry  *mrtlink;   /* link to routing entries             */
    struct grpentry  *next;      /* link to next entry                  */
    struct mrtentry  *rp-entry;  /* Pointer to the (*,G) entry          */
    struct rplist    *active-rp; /* Pointer to the active RP            */
};
```

The multicast routing table is the collection of all routing entries, which are organized in a linked list in the daemon.

The overall structure of the multicast routing table in the daemon is shown in figure 4.

One of the frequently used fields in the mrtentry is the 'flags' field, where the values assigned to that field can be one/combination of the following:

```
#define MRTF-SPT                0x0001          /* shortest path tree bit */
#define MRTF-WC                 0x0002          /* wildcard bit           */
#define MRTF-RP                 0x0004          /* RP bit                 */
```

```
#define MRTF-CRT                0x0008          /* newly created        */
#define MRTF-IIF-REGISTER       0x0020          /* iif = reg-vif         */
#define MRTF-REGISTER           0x0080          /* oif includes reg-vif   */
#define MRTF-KERNEL-CACHE       0x0200          /* kernel cache mirror    */
#define MRTF-NULL-OIF           0x0400          /* null oif cache         */
#define MRTF-REG-SUPP           0x0800          /* register suppress      */
#define MRTF-ASSERTED           0x1000          /* RPF is an assert winner*/
#define MRTF-SG                 0x2000          /* pure (S,G) entry       */
```

3.1.2 Virtual Interface List

The virtual interface data structure is shown below :

```
struct vifentry {
    u-short          flags;            /* VIFF- flags                 */
    u-char           threshold;        /* min ttl required            */
    u-long           local;            /* local interface address     */
    u-long           remote;           /* remote address              */
    u-long           subnet;           /* subnet number               */
    u-long           subnetmask;       /* subnet mask                 */
    u-long           broadcast;        /* subnet broadcast addr       */
    char             name[IFNAMSIZ];   /* interface name              */
    u-char           timer;            /* timer for sending queries   */
    u-char           gq-timer;         /* Group Query timer, used by DR*/
    struct nbrentry *neighbors;        /* list of neighboring routers  */
    u-int           rate-limit;        /* max rate                    */
};
```

The virtual interface table is the collection of all virtual interface entries. They are organized as an array (viflist[MAXVIFS]; MAXVIFS currently set to 32).

In addition to defining ‘mrtenry’, ‘vifentry’ and other data structures, the ‘pim.h’ file also defines all default timer values.

3.2 Initialization and Set up

Most of the initialization calls and socket setup are handled through main() [in main.c]. Basically, after the alarm and other signals are initialized, PIM and IGMP socket handlers are setup, then the function awaits on a ‘select’ call. The timer interval is set to TIMER-INTERVAL [currently 5 seconds], after which the timer function is invoked, if no packets were detected by ‘select’. The timer function basically calls the other timing and housekeeping functions; age-vifs() and age-routes(). Also, another alarm is scheduled for the following interval.

3.3 PIM and IGMP message handling

3.3.0.1 PIM All PIM control messages (in PIM-SMv2) have an IP protocol number of IPPROTO-PIM (assigned to 103), and are *not* part of IGMP like PIMv1 messages.

Incoming PIM messages are received on the pim socket, and are dispatched by accept-pim() [in pim.c], according to their type.

PIM types are:

```
PIM-HELLO                    0
PIM-REGISTER                 1
PIM-REGISTER-STOP            2
PIM-JOIN-PRUNE               3
PIM-BOOTSTRAP                4
PIM-ASSERT                   5
PIM-GRAFT                    6
PIM-GRAFT-ACK                7
PIM-CANDIDATE-RP-ADVERTISEMENT 8
```

Outgoing PIM messages are sent using send-pim() and send-pim-unicast() [in pim.c].

3.3.0.2 IGMP IGMP messages are dispatched using similar machinery to that of PIM, only IGMP messages are received on the igmp socket, dispatched by accept-igmp(), and are sent using send-igmp() [in igmp.c].

3.4 MRT maintenance

The functions handling the MRT creation, access, query and update are found in 'mrt.c' file.

Major functions include route lookups; as in find-route(), find-source(), and find-group().

The hash function and group to RP mapping are also performed by functions in 'mrt.c'.

3.5 Protocol Specific actions (Routing)

File 'route.c' contains the protocol specific functions, and processes the incoming/outgoing PIM messages.

Functions processing incoming PIM messages include accept-join-prune(), accept-assert(), accept-register(), accept-register-stop()..etc, and other supporting functions.

Functions triggering outgoing PIM messages include event-join-prune(), send-register(), trigger-assert(), send-C-RP-Adv()..etc, and supporting functions.

In addition, route.c also handles the internal control messages through 'process-kernelCall()', which dispatches the internal control messages according to their type.

Currently, there are three types of internal control messages:

```
IGMPMSG-NOCACHE        1 /* indicating a cache miss             */
IGMPMSG-WRONGVIF       2 /* indicating wrong incoming interface */
IGMPMSG-WHOLEPKT       3 /* indicating whole data packet; used
                          * for registering
                          */
```

These messages are dispatched to process-cacheMiss(), process-wrongiif(), and process-wholepkt(), respectively.

3.6 Timing

The clock tick for the timing system is set to TIMER-INTERVAL (currently 5 seconds). That means, that all periodic actions are carried out over a 5 second granularity.

On every clock tick, the alarm interrupt calls the timer() function in main.c, which, in turn, calls age-vifs() [in vif.c] and age-routes() [in timer.c]. In this subsection, the functions in 'timer.c' are explained.

Basically, age-routes() browses through the routing lists/tables, advancing all timers pertinent to the routing tables. Specifically, the group list is traversed, and the leaf membership information is aged by calling timeout-leaf(). Then a for loop browses through the multicast route entries, aging the outgoing interfaces [timeout-outgo()], and various related timers (e.g. RegisterBitTimer, Assert timer..etc) for each entry, as well as checking the rate thresholds for the Registers (in the RP), and data (in the DRs). In addition, garbage collection is performed on the timed out entries in the group list, the source list and the MRT. The multicast forwarding cache (MFC) in the kernel is also timed out, and deleted/updated accordingly. Note that in this model, the MFC is passive, and all timing is done in at user level, then communicated to the kernel, see section 4.

The periodic Join/Prune messaging is performed per interface, by calling periodic-vif-join-prune(). Then the RPSetTimer is aged, and periodic C-RP-Adv messages are sent if the router is a Candidate RP.

3.7 Virtual Interface List

Functions in 'vif.c' handle setting up the viflist array in both the user level and the kernel, through 'start-vifs()'. Special care is given to the reg-vif; a dummy interface used for encapsulating/decapsulating Registers, see section 7. The reg-vif is installed by add-reg-vif(), and k-add-vif(reg-vif-num) calls.

Other, per interface, tasks are carried out by vif.c; like query-groups(), accept-group-report and query-neighbors(), in addition to the periodic age-vifs() timing function.

3.8 Configuration

pimd looks for the configuration file at "/etc/pimd.conf". Configuration parameters are parsed by 'config.c'. Current PIM specific configurable parameters include the register/data rate thresholds, after which the RP/DRs switch to the SPT, respectively. A candidate RP can be configured, with optional interface address and C-RP-Adv period, and a candidate bootstrap router can be configured with optional priority.

Following is an example 'pimd.conf':

```
#   Command formats:
#
# phyint <local-addr> [disable]  [threshold <t>]
# candidate-rp <local-addr> [time <number>]
# bootstrap-router <local-addr> [priority <number>]
# switch-register-threshold [count <number> time <number>]
# switch-data-threshold [count <number> time <number>]
#
candidate-rp time 60
bootstrap-router priority 5
switch-register-threshold count 10 time 5
```

3.9 Interfacing to Unicast Routing

For proper implementation, PIM requires access to the unicast routing tables. Given a specific destination address, PIM needs at least information about the next hop (or the reverse path forwarding 'RPF' neighbor) and the interface (iif) to reach that destination. Other information, are the metric used to reach the destination, and the unicast routing protocol type, if such information is attainable. In this document, only the RPF and iif information are discussed.

Two models have been employed to interface to unicast routing in pimd.

The first model, which requires ‘routing socket’ support, is implemented on the ‘IRIX’ platform, and other ‘routing socket’ supporting systems. This model requires no kernel modifications to access the unicast routing tables. All the functionality required is provided in ‘routesock.c’. In this document, we adopt this model.

The second model, is implemented on ‘SunOs’ and other platforms not supporting routing sockets, and requires some kernel modifications. In this model, an ‘ioctl’ code is defined (e.g. ‘SIOCGETRPF’), and a supporting function [e.g. get-rpf()] is added to the multicast code in the kernel, to query the unicast forwarding information base, through ‘rtalloc()’ call.

Other models may also be adopted, depending on the operating system capabilities.

In any case, the unicast routing information has to be updated periodically to adapt to routing changes and network failures.

4 User level - Kernel Interfaces

Communication between the user level and the kernel is established in both directions. Messages for kernel initialization and setup, and adding, deleting and updating entries from the viftable or the mfc in the kernel, are triggered by the multicast daemon. While PIM, IGMP, and internal control messages are passed from the kernel to user level.

4.1 User level to kernel messages

Most user level interfacing to the kernel is done through functions in ‘kern.c’. Traps used are ‘setsockopt’, ‘getsockopt’, and ‘ioctl’. Following is a brief description of each:

- setsockopt(): used by the daemon to modify and update the kernel environment; including the forwarding cache, the viftable.. etc.

  Options used with this call are:

  ```
  MRT-INIT        initialization
  MRT-DONE        termination
  MRT-ADD-VIF     add a virtual interface
  MRT-DEL-VIF     delete a virtual interface
  MRT-ADD-MFC     add an entry to the multicast forwarding cache
  MRT-DEL-MFC     delete an entry from the multicast forwarding cache
  MRT-PIM         set a pim flag in the kernel [to stub the pim code]
  ```

- getsockopt(): used to get information from the kernel.

  Options used with this call are:

  ```
  MRT-VERSION     get the version of the multicast kernel
  MRT-PIM        get the pim flag
  ```

- ioctl(): used by the daemon for 2 way communication with the kernel.

  Used to get interface information [in config.c and vif.c]. ‘kern.c’ uses ‘ioctl’ with option ‘SIOCGETSGCNT’ to get the cache hit packet count for an (S,G) entry in the kernel. Also, ioctl may be used to get unicast routing information from the kernel using the option ‘SIOCGETRPF’, if such model is used to get unicast routing information, see section 3.9.

4.2 Kernel to user level messages

The kernel uses two calls to send PIM, IGMP and internal control messages to user level:

- raw-input(): used by the kernel to send messages/packets to the raw sockets at user level.

  Used by both the pim machinery [pim-input()], and igmp machinery [igmp-input()], in the kernel, to pass the messages to the raw socket queue, and in turn to pim and igmp sockets, to which the pim daemon listens.

- socket-send(): used by the multicast forwarding machinery to send internal control messages to the daemon.

  Used by the multicast code in the kernel [in ip-mroute.c], to send internal, multicast specific, control messages:

  1. ip-mforward(), triggers an 'IGMPMSG-NOCACHE' control message, when getting a cache miss.
  2. ip-mdq(), triggers an 'IGMPMSG-WRONGVIF' control message, when failing the RPF check (i.e. getting a wrong iif).
  3. register-send(), relays 'IGMPMSG-WHOLEPKT' messages containing the data packet, when called by ip-mdq() to forward packets to the 'reg-vif'.

5 IP Multicast Kernel Support

The kernel support for IP multicast is mostly provided through 'ip-mroute.{c,h}', providing the structure for the multicast forwarding cache (MFC), the virtual interface table (viftable), and supporting functions.

5.1 The Multicast Forwarding Cache

The Multicast Forwarding Cache (MFC) entry is defined in 'ip-mroute.h', and consists basically of the source address, group address, an incoming interface (iif), and an outgoing interface list (oiflist). Following is the complete definition:

```
struct mfc {
    struct in-addr  mfc-origin;                 /* ip origin of mcasts      */
    struct in-addr  mfc-mcastgrp;               /* multicast group associated*/
    vifi-t          mfc-parent;                 /* incoming vif             */
    u-char          mfc-ttls[MAXVIFS];          /* forwarding ttls on vifs  */
    u-int           mfc-pkt-cnt;                /* pkt count for src-grp    */
    u-int           mfc-byte-cnt;               /* byte count for src-grp   */
    u-int           mfc-wrong-if;               /* wrong if for src-grp     */
    int             mfc-expire;                 /* time to clean entry up   */
};
```

The multicast forwarding cache table (mfctable), is a hash table of mfc entries defined as:

```
struct mbuf    *mfctable[MFCTBLSIZ];
```

where MFCTBLSIZ is 256.

In case of hash collisions, a collision chain is constructed.

5.2 The Virtual Interface Table

The viftable is an array of ‘vif’ structures, defined as follows:

```
struct vif {
    u-char              v-flags;        /* VIFF- flags defined above         */
    u-char              v-threshold;    /* min ttl required to forward on vif*/
    u-int               v-rate-limit;   /* max rate                          */
    struct tbf         *v-tbf;          /* token bucket structure at intf.   */
    struct in-addr      v-lcl-addr;     /* local interface address           */
    struct in-addr      v-rmt-addr;     /* remote address (tunnels only)     */
    struct ifnet       *v-ifp;          /* pointer to interface              */
    u-int               v-pkt-in;       /* # pkts in on interface            */
    u-int               v-pkt-out;      /* # pkts out on interface           */
    u-int               v-bytes-in;     /* # bytes in on interface           */
    u-int               v-bytes-out;    /* # bytes out on interface          */
    struct route        v-route;        /* Cached route if this is a tunnel  */
#ifdef RSVP-ISI
    u-int               v-rsvp-on;      /* # RSVP listening on this vif      */
    struct socket      *v-rsvpd;        /* # RSVPD daemon                    */
#endif /* RSVP-ISI */
};
```

One of the frequently used fields is the ‘v-flags’ field, that may take one of the following values:

```
VIFF-TUNNEL      0x1           /* vif represents a tunnel end-point */
VIFF-SRCRT       0x2           /* tunnel uses IP src routing        */
VIFF-REGISTER    0x4           /* vif used for register encap/decap */
```

5.3 Kernel supporting functions

The major standard IP multicast supporting functions are:

- ip-mrouter-init()

  Initialize the ‘ip-mrouter’ socket, and the MFC.

  Called by setsockopt() with option MRT-INIT.

- ip-mrouter-done()

  Disable multicast routing.

  Called by setsockopt() with option MRT-DONE.

- add-vif()

  Add a new virtual interface to the viftable.

  Called by setsockopt() with option MRT-ADD-VIF.

- del-vif()

  Delete a virtual interface from the viftable.

  Called by setsockopt() with option MRT-DEL-VIF.

- add-mfc()

  Add/update an mfc entry to the mfctable.

  Called by setsockopt() with the option MRT-ADD-MFC.

- del-mfc()

  Delete an mfc entry from the mfctable.

  Called by setsockopt() with the option MRT-DEL-MFC.

- ip-mforward()

  Receive an IP multicast packet from interface 'ifp'. If it matches with a multicast forwarding cache, then pass it to the next packet forwarding routine [ip-mdq()]. Otherwise, if the packet does not match on an entry, then create an 'idle' cache entry, enqueue the packet to it, and send the header in an internal control message to the daemon [using socket-send()], indicating a cache miss.

- ip-mdq()

  The multicast packet forwarding routine. An incoming interface check is performed; the iif in the entry is compared to that over which the packet was received. If they match, the packet if forwarded on all vifs according to the ttl array included in the mfc [this basically constitutes the oif list]. Tunnels and Registers are handled by this function, by forwarding to 'dummy' vifs. If the iif check does not pass, an internal control message (basically the packet header) is sent to the daemon [using socket-send()], including vif information, and indicating wrong incoming interface.

- expire-upcalls()

  Clean up cache entries if upcalls are not serviced.

  Called by the Slow Timeout mechanism, every half second.

The following functions in the kernel provide support to PIM, and are part of 'ip-mroute.c':

- register-send()

  Send the whole packet in an internal control message, indicating a whole packet, for encapsulation at user level.

  Called by ip-mdq().

- pim-input()

  The PIM receiving machinery in the kernel. Check the incoming PIM control messages and passes them to the daemon using raw-input(). If the PIM message is a Register message, it is processed; the packet is decapsulated and passed to register-mforward(), and header of the Register message is passed up to the pim socket using raw-input().

  Called by ip-intr() based on IPPROTO-PIM.

- register-mforward()

  Forward a packet resulting from register decapsulation. This is performed by looping back a copy of the packet using looutput(), such that the packet is enqueued on the 'reg-vif' queue and fed back into the multicast forwarding machinery.

  Called by pim-input().

6 Appendix I

The user level code, kernel patches, and change description, are available in,

http://catarina.usc.edu/ahelmy/pimsm-implem/

or through anonymous ftp from,

catarina.usc.edu:/pub/ahelmy/pimsm-implem/

7 Appendix II: Register Models

The sender model, in PIM-SM, is based on the sender's DR registering to the active RP for the corresponding group. Such process involves encapsulating data packets in PIM-REGISTER messages. Register encapsulation requires information about the RP, and is done at the user level daemon. Added functionality to the kernel, is necessary to pull up the data packet to user level for encapsulation.

Register decapsulation (at the RP), on the other hand, is performed in the kernel, as the decapsulated packet has the original source in the IP header, and most operating systems do not allow such packet to be forwarded from user level carrying a non-local address (spoofing).

The kernel is modified to have a pim-input() machinery to receive PIM packets. If the PIM type is REGISTER, the packet is decapsulated. The decapsulated packet is then looped back and treated as a normal multicast packet.

The two models discussed above are further detailed in this section.

7.1 Register Encapsulation

Upon receiving a multicast data packet from a directly connected source, a DR [initially having no (S,G) cache] looks up the entry in the kernel cache. When the look-up machinery gets a cache miss, the following actions take place (see figure 5):

1. an idle (S,G) cache entry is created in the kernel, with oif = null,
2. the data packet is enqueued to this idle entry [a threshold of 4 packets queue length is currently enforced],
3. an expiry timer is started for the idle queue, and
4. an internal control packet is sent on the socket queue using socket-send(), containing the packet header and information about the incoming interface, and the cache miss code.

[Note that the above procedures occur for the first packet only, when the cache is cold. Further packets will be either enqueued (if the cache is idle and the queue is not full), dropped (if the cache is idle and the queue is full), or forwarded (if the cache entry is active).]

At user space, the igmp processing machinery receives this packet, the internal control protocol is identified and the message is passed to the proper function to handle the kernelCalls [process-kernelCall()].

The cache miss code is checked, then the router checks to see:

1. if the sender of the packet is a directly connected source, and
2. if the router is the DR on the receiving interface.

Figure 5: At the DR: Creating (S,G) entries for local senders and registering to the RP
Actions are numbered in the order they occur

If the check does not pass, no action pertaining to Registers is taken[2]. If the daemon does not activate the idle kernel cache, the cache eventually times out, and the enqueued packets are dropped.

If the check passes, the daemon creates an (S,G) entry with the REGISTER bit set in the 'flags' field, the iif set the interface on which the packet was received, and the reg-vif included in the oiflist, in addition to any other oifs copied from wild card entries according to the PIM spec. 'reg-vif' is an added 'dummy interface' for use in the register models. Further, the daemon installs this entry in the kernel cache; using setsockopt() with the 'ADD-MFC' option.

This triggers the add-mfc() function in the kernel, which in turn calls the forwarding machinery [ip-mdq()]. The forwarding machinery iterates on the virtual interfaces, and if the vif is the reg-vif, then the register-send() function is called. The latter function, is the added function for encapsulation support, which sends the enqueued packets as WHOLE-PKTs (in an internal control message) to the user level using socket-send().

The message flows through the igmp, and process-kernelCall machineries, then the [(S,G) REGISTER] is matched, and the packet is encapsulated and unicast to the active RP of the corresponding group.

Subsequent packets, match on (S,G) (with oif=reg-vif) in the kernel, and get sent to user space directly using register-send().

7.2 Register Decapsulation

At the RP, the unicast Registers, by virtue of being PIM messages, are received by the pim machinery in the kernel [pim-input()]. The PIM type is checked. If REGISTER, pim-input() checks the null register bit, if not set, the packet is passed to register-mforward(), which loops it back on the 'reg-vif' queue using looutput(). In any case, the header of the Register (containing the original IP header, the PIM message header and the IP header of the inner encapsulated packet) is passed to raw-input(), and in turn to pim socket, to which the PIM daemon listens (see figure 6).

[2]Other checks are performed according to the longest match rules in the PIM spec. Optionally, if no entry is matched, a kernel cache with oif = null may be installed, to avoid further cache misses on the same entry.

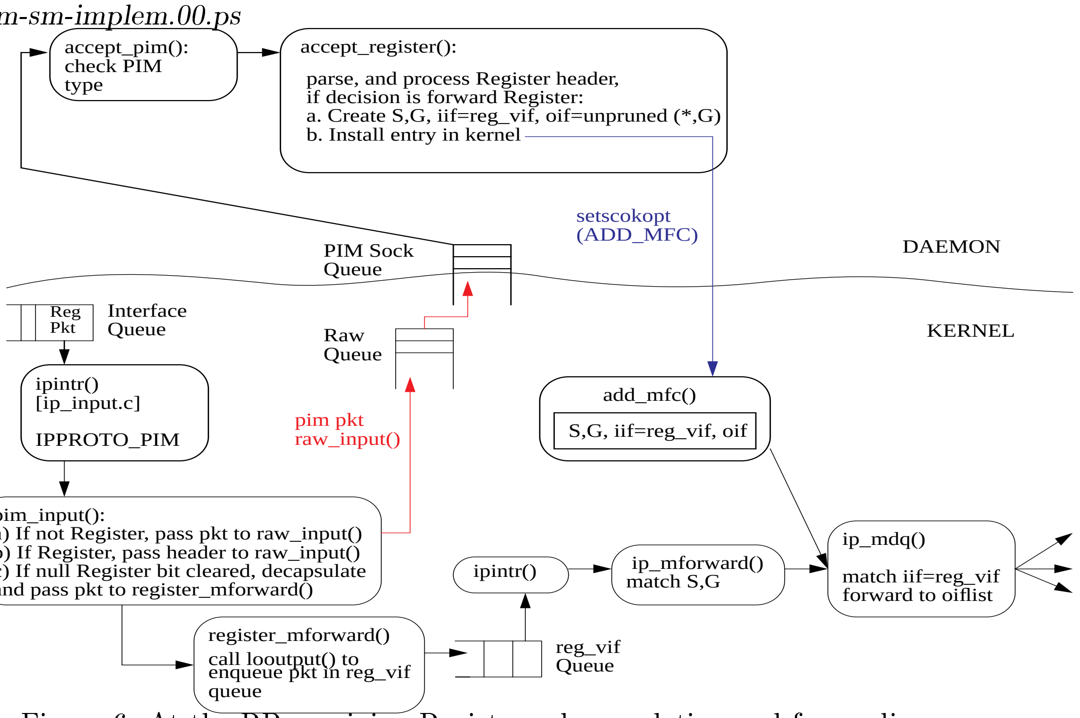

Figure 6: At the RP, receiving Registers, decapsulating and forwarding

At the PIM daemon, the message is processed by the pim machinery [accept-pim()]. REGISTER type directs the message to the accept-register() function. The Register message is parsed, and processed according to PIM-SM rules given in the spec.

If the Register is to be forwarded, the daemon performs the following:

1. creates (S,G) entry, with iif=reg-vif, and the oiflist is copied from wild card entries [the data packets are to be forwarded down the unpruned shared tree, according to PIM-SM longest match rules], and

2. installs the entry in the kernel cache, using setsockopt(ADDMFC)

At the same time, the decapsulated packet enqueued at the reg-vif queue is fed into ip-intr() [the IP interrupt service routine], and passed to ip-mforward() as a native multicast data packet. A cache lookup is performed on the decapsulated packet. If the cache hits and the iif matches (i.e. cache iif = reg-vif), the packet is forwarded according to the installed oiflist. Otherwise, a cache miss internal control message is sent to user level, and processed accordingly.

Note that, a race condition may occur, where the decapsulated packet reaches ip-mforward() before the daemon installs the kernel cache. This case is handled in process-cacheMiss(), in conformance with the PIM spec, and the packet is forwarded accordingly.

8 Acknowledgments

Special thanks to Deborah Estrin (USC/ISI), Van Jacobson (LBL), Bill Fenner, Stephen Deering (Xerox PARC), Dino Farinacci (Cisco Systems) and David Thaler (UMich), for providing comments and hints for the implementation. An earlier version of PIM version 1.0, was written by Charley Liu and Puneet Sharma at USC.

A. Helmy did much of this work as a summer intern at Silicon Graphics Inc.

PIM was supported by grants from the National Science Foundation and Sun Microsystems.

Contents

References

[1] D. Estrin, D. Farinacci, A. Helmy, D. Thaler, S. Deering, M. Handley, V. Jacobson, C. Liu, P. Sharma, and L. Wei. Protocol independent multicast (pim) : Motivation and architecture. *Experimental RFC*, December 1996.

[2] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C. Liu, L. Wei, P. Sharma, and A. Helmy. Protocol independent multicast (pim): Specification. *Internet Draft*, June 1995.

[3] D. Estrin, D. Farinacci, A. Helmy, D. Thaler, S. Deering, M. Handley, V. Jacobson, C. Liu, P. Sharma, and L. Wei. Protocol independent multicast-sparse mode (pim-sm) : Protocol specification. *Experimental RFC*, December 1996.