Chapter 6

LMS IMPLEMENTATION

In the previous chapter we used simulation to evaluate the performance of LMS in large, simulated topologies. The simulations showed that LMS performs very well in terms of limiting implosion and exposure, and maintains low recovery latency. In this chapter we complement our simulation by describing the implementation and evaluation of LMS in the kernel of a real system, namely NetBSD Unix.

As we described in the previous chapter, large scale deployment and evaluation of LMS is a task of enormous proportions, one which we are not equipped to perform. Such a task requires access to a large, multi-node network, preferably under our complete control, which is very hard to achieve. Our resources only allowed us to deploy LMS over just four nodes. However, despite our limited setup, we believe that the implementation we present in this chapter contributes some important insights. Even though our implementation testbed is minimal, our work constitutes a significant portion of essential preliminary work towards a future wide deployment of LMS. In addition, our work is invaluable to anyone wishing to understand how to implement LMS in other platforms, and thus can help in promoting both a better understanding of LMS, and its dissemination. It is our hope that we can migrate our implementation to a larger size testbed, perhaps utilizing one of the few experimental networks in existence today.

In this chapter we present a software implementation of the LMS forwarding services, which is the major new component in LMS, and thus, the most interesting and important to understand. We

assume the following environment: a multicast group has been created and all its members have joined; the replier state has been established; and a transport protocol is in place that detects losses, and uses LMS to send retransmission requests and retransmissions in the form of directed multicasts (dmcasts). We will refer to this protocol as *LTP*; we will not implement this transport protocol again, since we have already done so in the previous chapter, in our simulations. We will simply focus on implementing the LMS forwarding services at the routers and the LMS support services at the endnodes that allow applications to use the LMS forwarding services.

Our main objective in evaluating the forwarding services of LMS using implementation is to study the feasibility of integrating LMS into the existing environment. Other objectives include capturing the effort that must be expended in integrating LMS into the existing, highly optimized and possibly delicately balanced system, namely the networking code, as well as any performance penalties. Specifically, the objectives of our implementation as presented in this chapter, are the following:

- Determine if LMS can be implemented. This objective is very important. It essentially seeks to determine whether LMS can be incorporated into the existing router architecture, or if there is a fundamental incompatibility between LMS and the existing environment.

- Determine how much effort it would take to implement LMS. This objective is to quantify the complexity of implementing LMS in terms of programming effort.

- Determine how much change it would require to the existing architecture, if any. This objective is to determine how much impact (if any) LMS has on the existing architecture. We are especially interested in determining what kind of support LMS needs, and whether implementing LMS will require major changes in the existing architecture.

- Quantify the overhead of LMS compared to normal packet forwarding. this objective is to evaluate the performance of LMS in terms of state and processing cycles.

From an implementation point of view, we can divide LMS into the following components:

- **LMS-FWD**: Forwarding component: this LMS component resides at the routers and implements the special forwarding of LMS packets. This component is needed only at routers and resides entirely in the kernel, specifically at the IP layer.

- **LMS-API**: Application API: this component is required at the endnodes to enable applications to use LMS (i.e., send and receive LMS packets). This component implements any changes needed to the existing application and protocol interface. Fortunately, LMS did not require any changes to the socket interface, only small changes at the UDP level.

- **LMS-ENCAP**: LMS encapsulation: this is the component required to encapsulate multicast packets in unicast packets in order to send directed multicasts (dmcasts). Its functionality is very similar to the IPIP encapsulation protocol used in tunnels, but generalized to work without the existence of tunnels; in other words, sending LMS encapsulated packets does not require the existence of a tunnel, and packets can be sent to any unicast address, not just the tunnel peer.

- **LMS-HIER**: Replier hierarchy component: this is the component that allows endpoints and routers to communicate in order to maintain the replier hierarchy. It includes some modifications to the join procedure, and inter-router message exchange protocol or mechanism to maintain the replier hierarchy. This component is shared among routers and endpoints and resides in the kernel.

Figure 6.1 shows the LMS components and their relation with each-other along with the relevant applications and protocols. On the left we show the LMS components at an endnode. The boxes labeled “application” and “LTP”, are the multicast application that requires reliable multicast, and the transport protocol, i.e., the module that uses LMS to provides the error control. Note that we do not consider LTP to be part of LMS; it is simply a user of LMS services. For the remaining of our discussion we assume that LTP is implemented in user space, either as part of the application or as a separate library. However, there is no reason why LTP cannot be implemented in the kernel, as another protocol in the protocol stack next to TCP and UDP.

LTP communicates with LMS via the LMS-API module by exchanging control information, such as the turning point location, and data information, such as requests and retransmissions. The

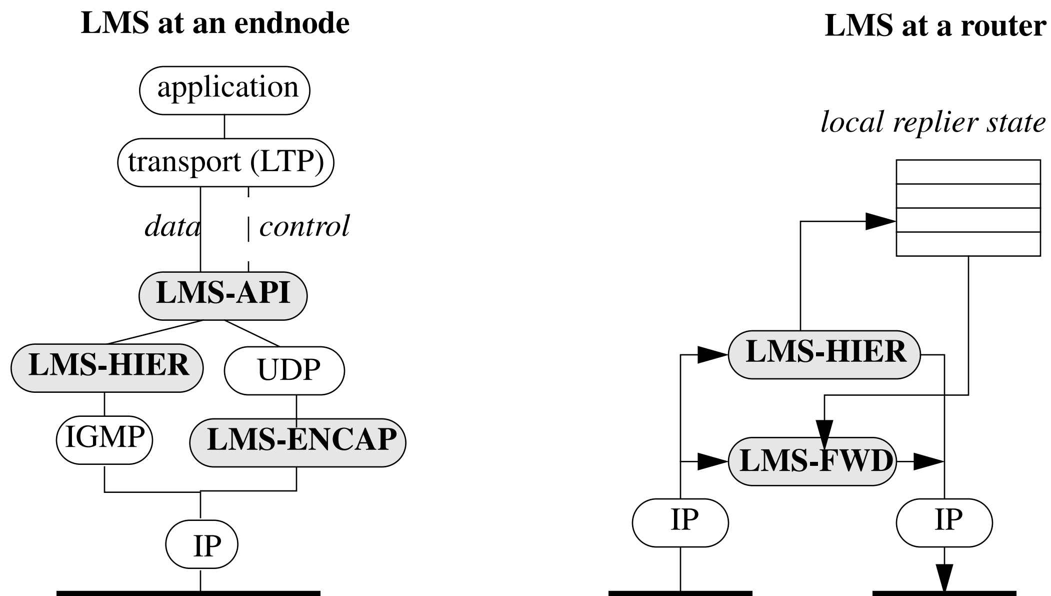

Figure 6.1: LMS components

task of the LMS-API module is as follows: on the sending side, it formats control data as an IP option and passes it to UDP/IP; on the receiving side, it extracts control data from the IP option attached to the LMS packet, and passes it to LTP. When LTP sends a dmcast, UDP passes the data and the control information to LMS-ENCAP, which prepares the encapsulated packet and passes it to IP for transmission.

LTP also uses the LMS-API module to communicate with the LMS-HIER module to update its replier status. The LMS-HIER module in turn uses IGMP to send these updates to the router.

The modules at a LMS router are shown on the right side of Figure 6.1. The LMS-HIER component receives updates either from endnodes or other routers and updates the local replier hierarchy state at the router. It then decides if the updates must be propagated to other routers, in which case it sends appropriate messages. Finally, the LMS-FWD module implements the heart of LMS, i.e., the forwarding services. This module receives LMS packets, consults the local replier state and forwards the packets accordingly after possibly adding the turning point information.

In this chapter, we present the implementation of three of the aforementioned LMS components, namely LMS-FWD and LMS-API, and LMS-ENCAP. We did not implement the fourth component, LMS-HIER, because it is not yet clear how much of it is needed and what its precise functionality should be. We discuss the issues around the implementation of LMS-HIER and sketch an implementation of this module at the end of this chapter.

In the remaining of this chapter we present our software implementation of LMS. We have modified the NetBSD Unix kernel to support handling of requests and directed multicasts. The kernel modifications include the addition of two new IP multicast options. The modified kernel components are: UDP output processing, IP and UDP input processing, and kernel multicast forwarding. The implementation of LMS presented no major problems and demonstrated that LMS can indeed be implemented. The implementation took about 250 lines of new C-code, which is less than the existing forwarding code; moreover, it took less that 2 weeks to write, most of that time spent on understanding the existing code. Our implementation presented no major disruptions to the existing networking code; however, it did point out some minor changes in handling IPIP encapsulated packets that may be needed to accommodate LMS. LMS processing did not affect the existing multicast forwarding, and the processing overhead of the added code is minimal.

This chapter is organized as follows: we first provide some background of the existing architecture of NetBSD, which will be useful in understanding our implementation. We then describe in detail the modifications we made to NetBSD to implement LMS.Then we present our experimental setup and the experiments we used to evaluate LMS. We discuss the issues in the module we did not implement, namely LMS_HIER and sketch a future implementation of this module. Finally, we conclude with a summary of the chapter.

6.1. Background

LMS requires operating system support in two areas: (a) in exchanging control information between the LTP and UDP/IP, and (b) special LMS forwarding support from hosts acting as a multicast routers. In this section we introduce the components and mechanisms available in NetBSD to support these operations. These are, (a) the system calls provided by NetBSD that allow control information exchange between LTP and the protocol stack (called *ancillary data* in NetBSD terminology); and (b) the NetBSD multicast forwarding architecture, upon which LMS builds. In the next

section we will describe the modifications we made to these components to support LMS. As we will see, these modifications are minimal and we were able to reuse a substantial part of the existing code.

6.1.1. Control Information Exchange: Ancillary Data

As we have seen before, LTP must convey some control information to the protocol stack in order to send a request or a directed multicast (dmcast); in turn, the protocol stack must relay some information back to LTP when a request or a retransmission is received. For example, the control information provided by LTP when sending a request includes the original sender's address, in order to identify the correct multicast tree; for a dmcast, LTP must provide control information which includes the address of the turning point router and the link id. Conversely, when receiving a request, the turning point information carried in the request must be passed to LTP as control information. It is important to note that since each message may carry control information, control exchange between the kernel and LTP must take place at a message level granularity.

The NetBSD socket interface provides two ways of exchanging information between user space (where we assume LTP resides) and the protocol stack. The first is via the system calls setsockopt and getsockopt; the second is via the system calls sendmsg and recvmsg. We describe both next, in order to determine which pair is best suited for LMS.

The set/getsockopt pair is used to set/get socket options that typically remain in effect for the lifetime of the socket. Examples include multicast group membership for the socket, the size of the socket buffer, the time-to-live (TTL) value for outgoing multicast packets, etc. Thus, the set/getsockopt pair is more appropriate for manipulating values that are long-lived rather than per-message operations, which is what LMS requires. If we wanted to use these calls LTP must follow every read or precede every write with the appropriate sockopt system call to exchange control information. However, not only is this method very cumbersome, it is also expensive since it requires twice the number of system calls. We thus conclude that the set/getsockopt pair is inappropriate for LTP/LMS.

Fortunately, as we will see, the send/recvmsg system call pair is perfectly suited for the requirements of LTP/LMS. These system calls essentially behave like normal read and write calls; however, in addition to data, these system calls accept additional control parameters called *ancillary*

data. These parameters are carried along with the normal data when a system call is made, as depicted in Figure. 6.2.

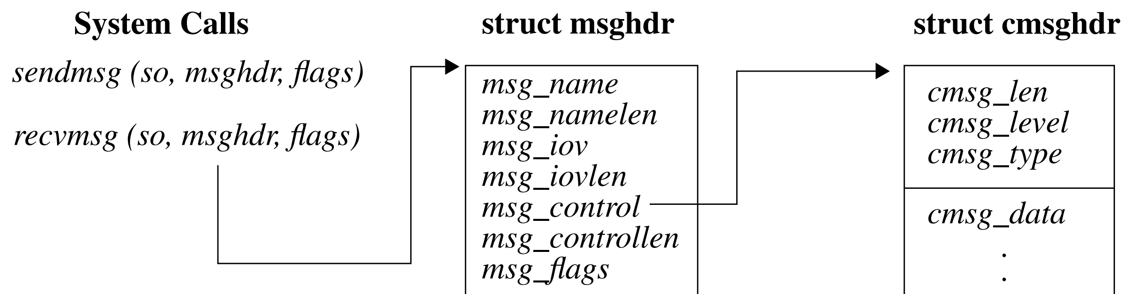

Figure 6.2: Ancillary data in NetBSD.

Control information is passed as follows: the send/recvmsg system calls take as an argument a rich data structure called msghdr. Structure msghdr contains the destination address of the message, the data for the message and a set of flags; in addition, this structure contains a pointer to another data structure of type cmsghdr which carries control data. The control data is preceded by a header containing the length of the data (cmsg_len), the level (or protocol) the data is destined for or received from (cmsg_level), and a protocol-specific type (cmsg_type).

The exchange of control data is depicted in Figure 6.3, and proceeds as follows: before every send/recvmsg system call, LTP prepares a cmsghdr structure; for a send call, data to be sent is pointed to by cmsg_data. For a receive call, an empty buffer is passed whose length is specified in msg_controllen. On sending, the data will be delivered to the protocol specified by the field cmsg_level; in the case of LTP/LMS this level is UDP. UDP passes the packet with the options unchanged to the IP layer, where the options are inserted into the packet, which is then passed to the network interface for transmission.

On the receiving side, the reverse takes place. The packet is delivered to the IP layer by the network interface, where the options are examined. If they are IP related options, they are processed by the ip_dooptions function. If the packet is not forwarded or it does not contain an error, processing of the packet continues at the IP layer, which finally delivers it to the UDP layer. If there are still options remaining in the packet and LTP has specified that it wants to receive such options

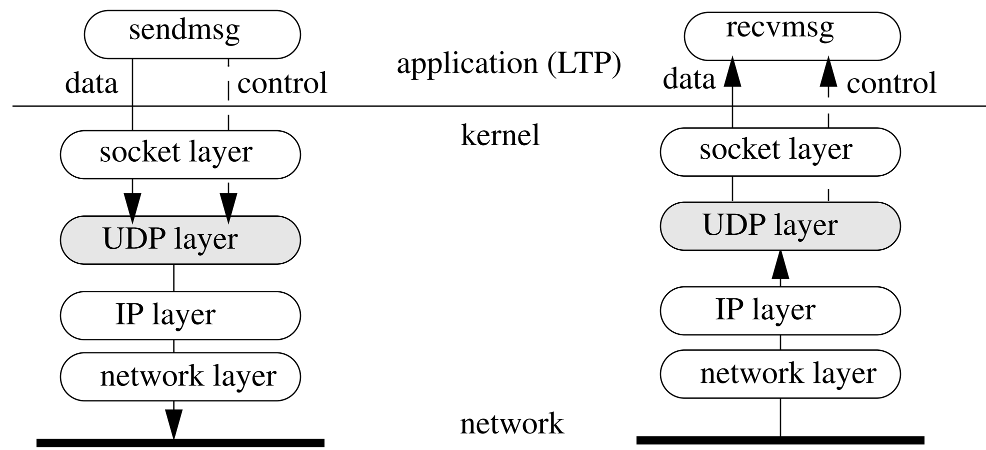

Figure 6.3: Exchange of control data

(appropriately done via the setsockopt system call; we will revisit this later), the options are appended to the socket buffer and delivered to LTP in the control portion of the recvmsg system call, in a cmsghdr structure.

We have briefly described how the send/recvmsg pair can be conveniently used to receive or deliver data and control information with every read or write request. Note that our use of this interface is consistent with what the others envision: the method we described is very similar to the method being considered for setting and retrieving information carried in IPv6 header options [41].

6.1.2. NetBSD Multicast Forwarding Architecture

Having described the mechanism by which LTP/LMS can exchange control data with the protocol stack, we turn our attention to multicast forwarding. Our aim is to describe what happens to a multicast packet once it has been received at the IP layer, which is where the multicast forwarding decisions are made. We will briefly cover route lookup and the multicast forwarding loop, where each interface is queried to see if there are members downstream that should receive the packet. We will not describe details like joining and leaving a group here; we will also not talk about routing, since LMS has no impact on routing whatsoever. We will simply restrict our discussion to the IP layer, since this is the only layer where changes are needed to accommodate our implementation of LMS.

In order to perform multicast forwarding, the NetBSD kernel must (a) be configured as a multicast router (i.e., compiled with the MROUTING option), and (b) must have two or more network interfaces. The kernel is only responsible for forwarding; all routing related operations like route discovery and update, are performed by a user-level program, the multicast routing daemon, or *mrouted*. mrouted runs the multicast routing protocol and exchanges routing information with neighboring mrouteds to update multicast routes. mrouted communicates with the kernel to update forwarding entries in the *multicast forwarding cache* (mfc), which is located in the kernel for performance reasons. The structure of an entry in the multicast forwarding cache is shown in Figure

```
(file: ip_mroute.h)
/*
 * The kernel's multicast forwarding cache entry structure.
*/
struct mfc {
        LIST_ENTRY(mfc) mfc_hash;
        struct   in_addr mfc_origin;            /* ip origin of mcasts */
        struct   in_addr mfc_mcastgrp;          /* multicast group associated */
        vifi_t   mfc_parent;                    /* incoming vif */
        u_int8_t mfc_ttls[MAXVIFS];             /* forwarding ttls on vifs */
        u_long   mfc_pkt_cnt;                   /* pkt count for src-grp */
        u_long   mfc_byte_cnt;                  /* byte count for src-grp */
        u_long   mfc_wrong_if;                  /* wrong if for src-grp */
        int      mfc_expire;                    /* time to clean entry up */
        struct   timeval mfc_last_assert;       /* last time I sent an assert */
        struct   rtdetq *mfc_stall;             /* pkts waiting for route */
    };
```

Figure 6.4: A multicast forwarding cache entry

6.4. There is one such entry in the cache for every <S, G> pair.

The multicast forwarding architecture is shown in Figure 6.5. A multicast packet arriving on a network interface is delivered to the IP layer through the function ipintr (like any other IP packet). If the kernel is configured as a multicast router kernel, multicast packets are directed through the function ip_mforward, which performs multicast forwarding. ip_mforward attempts to find the correct mfc entry using the macro MFCFIND; if no entry exists, mrouted is called to install one.

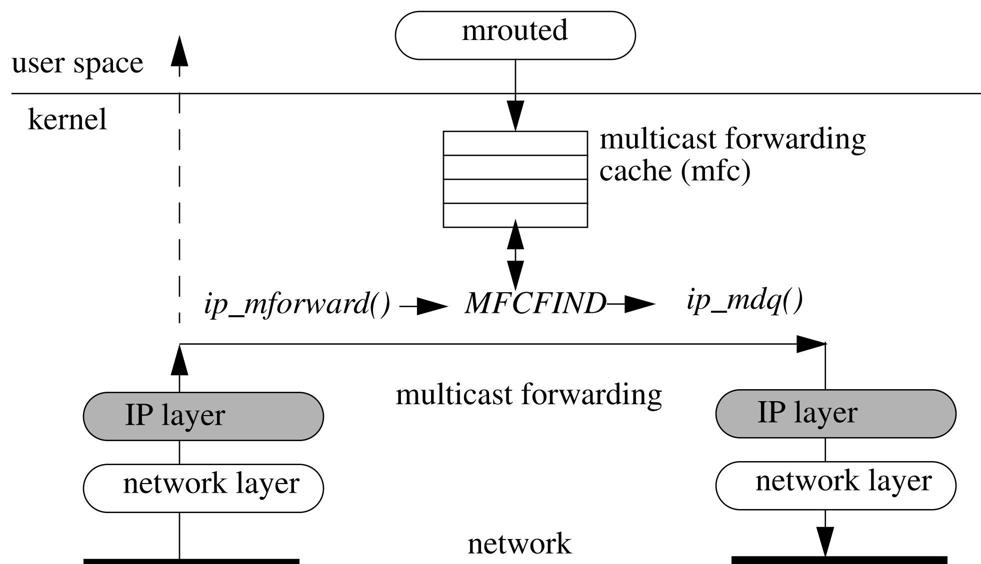

Figure 6.5: Multicast forwarding architecture in NetBSD

After an entry is found, ip_mforward calls ip_mdq, which checks if the packet arrived from the correct parent interface and loops through all outgoing interfaces sending the packet if (a) the packet's TTL exceeds the interface's threshold, and (b) there are group members downstream that interface. This function handles both real interfaces and tunnels. When ip_mforward returns, packet processing continues locally, in case there are multicast receivers on this host.

6.2. LMS in NetBSD

Having described the components in NetBSD relevant to LMS, we now proceed to describe how LMS is implemented. In our description we will list the files that were touched, describe the changes in existing functions and list new functions that we have written. We begin our discussion by describing the new IP options required to identify LMS messages. We then proceed to describe how LTP sends and receives messages containing LMS IP options and the information carried in them. We finish by describing how LMS messages are handled at multicast routers.

The files we changed in order to implement LMS are shown in Figure 6.6. Here we also list the

Files changed: *ip.h, ip_input.c, ip_mroute.h, ip_mroute.c, ip_var.h, udp_usrreq.c*

New IP Options: IPOPT_MREQ, IPOPT_DMCAST

New function name	Brief description
ipudp_encap	encapsulate a multicast packet into a unicast packet. Used by repliers when sending dmcasts
ip_domoptions	similar to ip_dooptions, this function processes multicast requests
ip_mfwdrequest	handles requests at the turning point router fills TP information
ip_dmcast	receives, decapsulates and dmcasts replies at the TP router

Figure 6.6: Summary of LMS implementation

new IP options and functions we added, the latter accompanied by a brief description of their operation. We will give more details of their use as we encounter them in our discussion below.

6.2.1. IP Options for LMS

LMS introduces two new types of messages that must be identified at the IP layer. These messages are (a) messages sent by LTP and destined for the replier, which we will call requests; and (b) messages sent by LTP containing directed multicasts, which are unicast messages encapsulating multicast messages and we will call dmcasts. We identify these types of messages by attaching to them two new IP options, named IPOPT_MREQ and IPOPT_DMCAST. Thus the first order of business is to define these two options, which is done as follows in the file *ip.h*:

```
/*
 * New LMS Options
 */
#define IPOPT_MREQ          138 /* LMS request */
#define IPOPT_DMCAST        139 /* LMS directed multicast */
```

The numbers we selected for these options are not important at the moment; we simply picked the next available option number from the options defined in ip.h. In a real implementation we would perhaps opt to introduce only one option (which could be called IPOPT_LMS) and then introduce sub-options to demultiplex LMS-related messages. However, for clarity, we will continue to use two IP options for the remaining of our discussion.

The next step is to define the structure of the IP options used by LMS. We can use the same structure for both requests and dmcasts, which we define in the file *ip_var.h*:

```
struct ipmopt_lms {
     u_int8_t   ipt_code               /* option type */
     u_int8_t   ipt_len;               /* option length */
     u_int16_t  ipt_tpif;              /* turning point iface */
     struct in_addr  ipt_tpaddr,       /* turning point addr */
                     ipt_src,          /* original source */
                     ipt_group;        /* multicast group */
}
```

The first field is the option type, which we described earlier. The second field is the option length, which includes the header. The third field is the identifier for turning point interface; in requests, it is filled by the turning point router; in dmcasts it contains the interface the packet should be multicast. The next three fields hold the internet addresses of the turning point router, the original source of the data requested or retransmitted, and the multicast group respectively. The last field, *ipt_group*, may appear redundant, since the address of the multicast group can be obtained from the destination address of the multicast packet. However, it is useful for repliers that may subscribe to multiple groups as an easy means to identify for which group the request is for.

6.3. LMS Implementation at Endpoints

So far we have defined two new IP options that distinguish requests and dmcasts, and we have also defined the structure of these options. We now proceed to describe how these options are used to send and receive requests and dmcasts at the endpoints. We start with requests.

6.3.1. Sending/Receiving Requests

LTP needs to send a request after is detects a gap in the sequence number of received packets. After a gap is detected, LTP creates a retransmission request with a list of the missing packets. The precise structure of the request is dependent on the particular transport protocol, but we expect that at a minimum it will contain the fields shown in Figure 6.7:

```
struct LmsNak {
      int   nak_lo;
      int   nak_hi;
      int   nak_seqn;
}
```

Figure 6.7: The data portion of a LMS request

The fields nak_lo and nak_hi specify the left and right edge of the gap; the field nak_seqn is the sequence number of the request (recall that the request may be sent multiple times in case of time-outs).

The above information is sent in the data portion of the request and is delivered as data to the replier; it is not examined by routers. Before the request is ready for transmission, LTP must prepare the control portion of the request which will be examined by each LMS router. Recall that the control portion will be delivered to the protocol via the sendmsg system call and will be carried with the request as an IP option. The control portion is prepared at the sending side as follows: LTP allocates a cmsghdr structure (see Figure 6.2), and fills its fields with the information shown in Figure 6.8. The cmsg_data field contains the information that will be placed in the IP option. In our implementation we chose to structure the control information as an IP option at LTP, in order to simplify the job of the transport protocol: thus we format the control information as the structure ipmopt_lms (see previous page); all UDP has to do once it receives the control information is strip the top portion and pass the cmsg_data portion to IP as an option. Following the creation of the IP option, LTP then allocates a msghdr structure, attaches the data portion of the request to the msg_iov field and the control portion to the msg_control field and finally calls sendmsg to send the request.

struct cmsghdr

cmsg_len = sizeof (struct cmsghdr) + sizeof (struct ipmopt_lms) *cmsg_level = IPPROTO_UDP* *cmsg_type = IPOPT_MREQ*
cmsg_data = struct ipmopt_lms *ipt_code =IPOPT_MREQ* *ipt_len = sizeof (struct ipmopt_lms)* *ipt_tpif = -1* *ipt_tpaddr = 0* *ipt_src = addr of sender* *ipt_group = addr of multicast group*

Figure 6.8: The control part of a LMS request

In the kernel the call reaches UDP, where the control portion is extracted and passed as an option to ip_output along with the actual request, as shown in Figure. 6.9. ip_ouput then forms

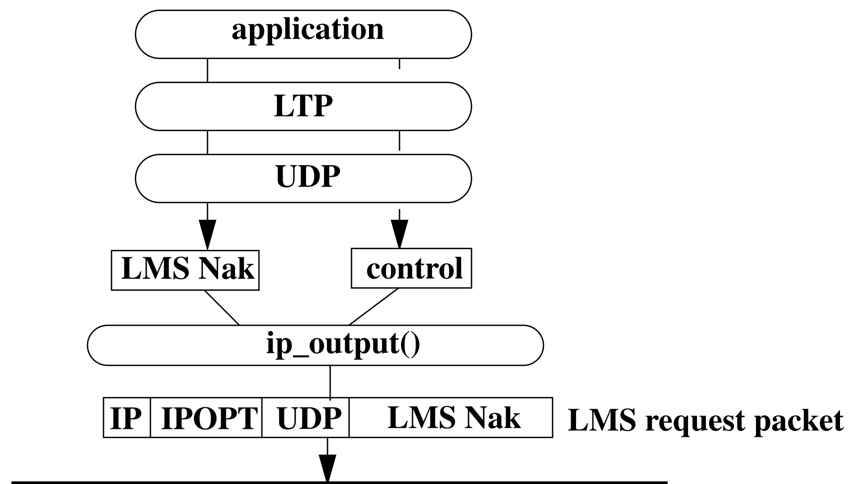

Figure 6.9: Combining control and data in a LMS request packet

the actual request packet, which is then multicast through the normal path.

The previous discussion described the sending of a request. We now proceed to describe what happens when a request is received. As one might expect, the operations are symmetrical. One

important difference, however, is that for an application like LTP to receive control data, it must explicitly tell the socket layer via a setsockopt system call. This is done as follows:

```
/*
* enable reception of IP options
*/
on = 1;
if (setsockopt (sock, IPPROTO_IP, IP_RECVOPTS, (char *)&on,
      sizeof (on)) < 0) {
      perror ("setsockopt IP_RECVOPTS");
      exit (1);
}
```

With receiving of options enabled, reception of a request proceeds as follows: a request reaches the IP layer via the ipintr function. The IP packet looks exactly like the request in Figure 6.9, except that the turning point information has been filled when the request passed through the turning point router. IP separates the data and the IP option from the packet, and passes both to UDP. Since LTP has specified that it wants to receive options, UDP calls the function sbappendaddr which appends the data and the control information contained in the option to the receive socket buffer. LTP then retrieves both with the recvmsg system call.

Only one file was changed to accommodate the above implementation, and that is udp_usrreq.c. Changes required to the existing code to accommodate option processing for LMS as described above, were minimal: NetBSD version 1.40, which is the version we used, did not allow sending of data together with IP options. We simply removed the code that rejected options, and added simple code to strip the cmsghdr from the control information. Since the control data was correctly formatted as an option by LTP, the control data was passed directly to the lower layer.

6.3.2. Sending/Receiving DMCASTS

We now move to the description of how dmcasts are sent and received at the endpoints. Recall that a replier who wishes to send a reply targeted to a multicast subtree does so by sending a multicast packet encapsulated in a unicast packet at the turning point router.

The process of sending a dmcast is depicted in Figure 6.10. In our current implementation, LTP sends a dmcast in a manner similar to sending a request: LTP creates a message containing the reply

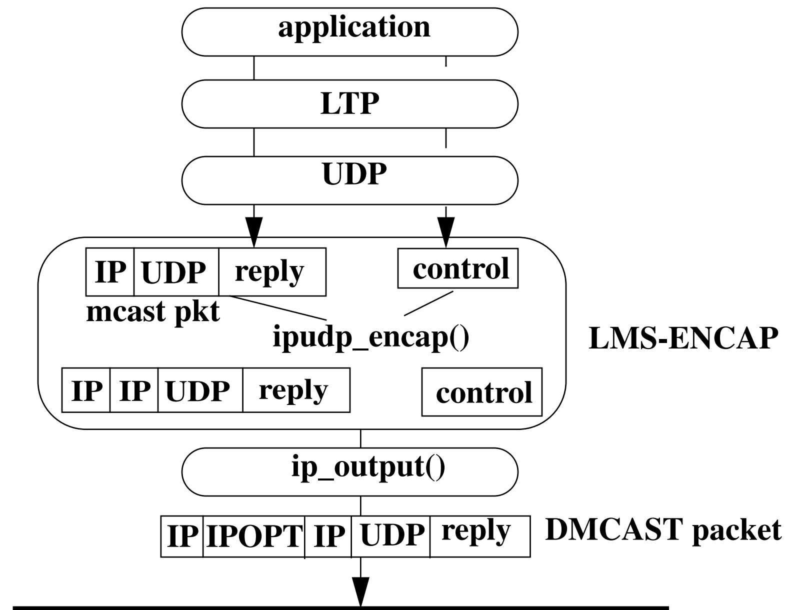

Figure 6.10: Sending a directed multicast at an endpoint

and a control message of type cmsghdr containing an ipmopt_lms structure with type IPOPT_DMCAST; unlike a request, where the turning point information is left empty, for a dmcast LTP fills the turning point fields with the control information received in the earlier request. LTP then calls sendmsg with both the reply and the control information, and multicasts the packet to the group.

When the packet reaches UDP it is intercepted, and passed to a new LMS function, ipudp_encap. This function receives from UDP the multicast packet containing the reply and the UDP/IP headers, and then prepares the IPOPT_DMCAST option by copying in the relevant control information provided by LTP. Then, it creates the encapsulating IP header, with the destination set to the router at the turning point (also provided by LTP). The encapsulating header's protocol field is set to IPPROTO_IPIP. Finally, the new packet and the option are passed to ip_output which attaches the IP option and unicasts it to the turning point router.

At the receiving side, no special operations take place since a dmcast becomes a regular multicast after the turning point router.

The services provided by LMS greatly simplify LTP's implementation over non-assisted protocols. When loss is detected LTP simply multicasts a message with the appropriate IP option, sets a timer, and waits for a retransmission. There is no topology state to guesstimate, no back-off timers for duplicate suppression, no need to scope retransmissions, no parent/children state, and no recovery groups to join. In that respect, LTP is more reminiscent of a unicast rather than a multicast protocol.

6.4. Handling LMS Messages at Routers

In the previous sections, we described how LMS requests and dmcasts are sent and received at the endpoints. In this section we describe how requests and dmcasts are handled at the routers. We start by describing the new state added at each LMS router, and continue to describe the new forwarding operations introduced by LMS.

6.4.1. New Router State

The new state LMS requires at the routers consists of two items: (a) the replier interface identifier, and (b) the replier cost. We added these items as new fields in the multicast forwarding cache entry, which was described earlier. We repeat the MFC entry here, in Figure 6.11, with the LMS fields included (shown in bold). Note that only four bytes are added for the LMS fields in the multicast forwarding cache entry. No other state is required for LMS.

6.4.2. Handling LMS Packets at a Router

The handling of LMS packets at the routers is depicted in Figure 6.12. When a multicast packet arrives at a router and is delivered to the IP layer, the function responsible for handling it is ipintr. We have modified the multicast handling portion of the input function ipintr to check for options in multicast packets. When a multicast packet is detected as carrying an option, it is passed to a new LMS function called ip_domoptions. Similar to the existing IP function ip_dooptions, ip_domoptions processes multicast options one at a time. Since currently the only two multicast options defined are ours, the packet is simply demultiplexed to one of two functions depending on the option it is carrying: ip_mfwdrequest, which handles multicast packets carrying the multicast option IPOPT_MREQ, or ip_dmcast, which handles multicast packets carrying the multicast option IPOPT_DMCAST. Shaded functions are new functions added by LMS.

```
(file: ip_mroute.h)
/*
 * The kernel's multicast forwarding cache entry structure, including the LMS state.
*/
struct mfc {
        LIST_ENTRY(mfc) mfc_hash;
        struct   in_addr mfc_origin;          /* ip origin of mcasts */
        struct   in_addr mfc_mcastgrp;        /* multicast group associated */
        vifi_t     mfc_parent;                /* incoming vif */
        vifi_t   mfc_replier;                 /* the replier interface */
        u_int16_t mfc_replier_cost;           /* the replier cost */
        u_int8_t mfc_ttls[MAXVIFS];           /* forwarding ttls on vifs */
        u_long   mfc_pkt_cnt;                 /* pkt count for src-grp */
        u_long   mfc_byte_cnt;                /* byte count for src-grp */
        u_long   mfc_wrong_if;                /* wrong if for src-grp */
        int      mfc_expire;                  /* time to clean entry up */
        struct   timeval mfc_last_assert;     /* last time I sent an assert */
        struct   rtdetq *mfc_stall;           /* pkts waiting for route */
  };
```

Figure 6.11: LMS state at the router

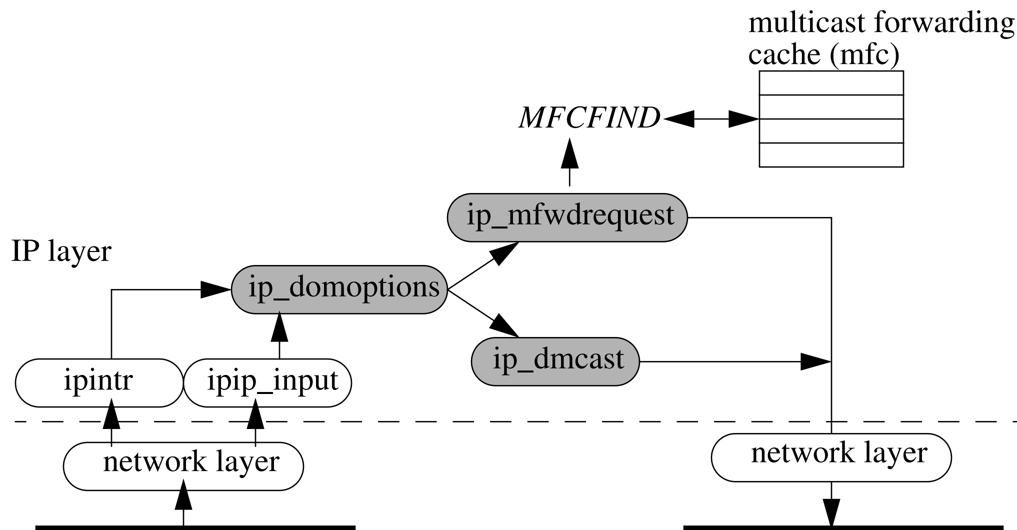

Figure 6.12: Handling of LMS packets at a router

6.4.3. Handling Requests at a Router

Requests are handled by ip_mfwdrequest. This function first copies the source and group addresses present in the option IPOPT_MREQ and uses them to perform a route lookup using the macro MFCFIND. The lookup returns the MFC entry for the original source. Then, the mfc_replier field in the entry is compared to the request's incoming interface. If they match, the packet came from the replier link, in which case it is sent to the mfc_parent interface; otherwise it is sent to the mfc_replier interface. In the latter case, before forwarding the packet, ip_mfwdrequest fills the turning point information in the IPOPT_MREQ option.

6.4.4. Handling DMCASTs at a Router

Having described how requests are handled at the router, we now proceed to describe the handling of dmcasts. Recall that a dmcast arrives at the router as a multicast packet encapsulated in a unicast packet. The encapsulation protocol is IPIP.

We have modified the ipip_input function to detect packets with multicast options and pass them to ip_domoptions. Thus, when a packet containing the IPOPT_DMCAST option arrives, it is passed to the function ip_dmcast, as shown in the previous figure. This function retrieves the interface index from the option, strips the unicast IP header and the option from the packet, and, if the interface index is valid, forwards the multicast packet out the specified interface.

6.4.5. A Conflict with NetBSD's Handling of Encapsulated Packets

Our current implementation of the dmcast service, pointed out an area in the NetBSD networking code that must be modified to accommodate dmcasts. Specifically, it relates to the way IP encapsulated packets are handled. NetBSD currently assumes that if an IPIP encapsulated packet is received, then the host must have a tunnel. Therefore, ipip_input, which is the function responsible for receiving IPIP packets and performing decapsulation, rejects any IPIP packets unless at least one tunnel exists at the host, and the originating host was the other end of the tunnel. Thus, clearly, a dmcast packet arriving at a host would be promptly rejected unless it arrives from the other side of a multicast tunnel. We avoid this problem in our implementation by adding the multicast option to the encapsulating IP header, detecting its presence in ipip_input, and immediately switching over to LMS processing. Ideally, we would like to see NetBSD being able to handle IPIP packets that

may arrive at a router outside a tunnel; when this capability is present, the multicast option that is now attached to the encapsulating header should be moved inside, to the encapsulated header. Thus, the encapsulated packet will pass through IP twice, once to remove the encapsulation header, and once more to process the multicast packet and its option.

6.5. Evaluation

Our evaluation is comprised of two parts: in the first part we state the LMS overhead in terms of control bytes that are exchanged between the endpoints and the routers. In the second part, we run experiments to compare the actual processing overhead of the LMS forwarding services with normal packet forwarding at the routers.

6.5.1. Request Overhead at Endpoints

The overhead in terms of bytes for sending and receiving requests is summarized in Figure.

Send overhead	Receive overhead
28 control bytes 16 byte IP option	28 bytes at recvmsg 16 bytes into sockbuf 16 byte IP option

Figure 6.13: Request overhead at the endpoints

6.13. The table shows overhead *in addition* to the normal sendmsg/recvmsg overhead. At the sending side, the overhead consists of 28 control bytes, out of which 16 bytes are for the IP option and 12 bytes are for cmsghdr. This is followed by a call to ip_pcbopts to prepare the IP option which is 16 bytes (option type and length, src addr, router addr, group addr, and router link). At the receiving side, the 16 bytes of the IP option are copied into a cmsghdr structure and delivered to LTP via the recvmsg system call in a cmsghdr structure.

6.5.2. Dmcast Overhead at Endpoints

The overhead for sending a directed multicast is similar to sending a request, except that now we have an additional IP header for the encapsulation. There is no additional receiving overhead, however, since dmcasts are received as regular multicast packets. The overhead is summarized in Figure 6.14.

Send overhead	Receive overhead
28 bytes at sendmsg encapsulation IP hdr 16 byte IP option	None

Figure 6.14: Dmcast overhead at the endpoints

We observe that the LMS overhead in terms of bytes is very small. At the host, LMS requires 28 more bytes of control data to be delivered to the kernel. If we assume an application the uses 1K size packets, this is only a 3% overhead. The number of bytes that actually reach the network, however, is even smaller, only 16 bytes. We believe this is negligible overhead, even for requests. For example, a TCP ACK packet is 40 bytes; an LMS packet including the LMS option and UDP/IP headers is only 44 bytes, without counting the actual NAK carried in the packet. The size of the NAK is 12 bytes; note, however, that the TCP SACK option[57] takes a minimum of 10 bytes (8*n+2 bytes, where n is the number of gaps specified in the option).

6.5.3. Processing Overhead

We have evaluated the processing overhead of our implementation of LMS using the testbed shown in Figure 6.15. Even though our topology is very simple, it still allows the measurement of processing cycles required for all LMS forwarding operations. We measured the forwarding overhead at the LMS router; we did not measure the additional processing at the hosts because our changes at the hosts were minor. Moreover, processing of a LMS packet at the endnode happens only twice, during send and receive, while processing at the routers can potentially happen many times, depending on how many routers a LMS packet visits. All three machines in our testbed were Pentium II class machines, connected together with a 155 Mbps ATM network. The multicast

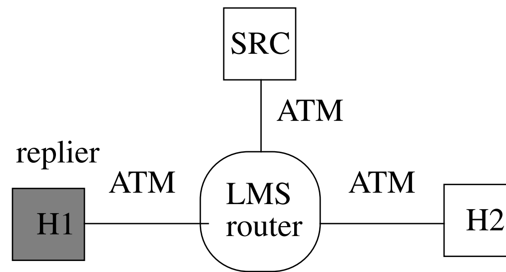

Figure 6.15: Experimental Testbed

sender was on host marked SRC; hosts H1 and H2 are the receivers. Host H1 (shaded) was designated as the replier.

The purpose of our experiments was twofold: (a) to verify that the forwarding operations in LMS worked correctly, and (b) to estimate their overhead and compare it with normal multicast forwarding. For the latter, we configured the processor at the LMS router to record the number of cycles spent on forwarding packets.

After setting up a multicast group that included all 3 hosts, we first verified that all receivers have joined the group successfully, and were receiving packets sent by SRC. Then, we conducted experiments to verify the correct operation of LMS. Specifically we ran experiments to test the following scenarios:

Scenario 1: Host H2 multicasts a request message. We verified that the request at the endpoint was sent correctly and contained the appropriate IP option, the LMS router received the message, filled out the turning point information, and forwarded it towards the replier interface. We verified that H1 received the message correctly and printed out its contents. We checked the turning point information and verified it to be correct.

Scenario 2: Host H1 multicasts a request message. We again verified the correct operation at the host, as before. We verified that the LMS router picked up the message and correctly forwarded it towards SRC. SRC received the message and printed out its contents, verifying that the packet was forwarded correctly.

Scenario 3: Host H1 sends a dmcast to the LMS router. We verified that the router received the correct message, decapsulated and multicast the message towards H2. Host H2 received the multicast packet correctly and printed out its contents.

After verifying that all LMS forwarding functions worked correctly, we set up our experiments to measure processing overhead. The measurements were taken using the processor cycle counter register in the Pentium processor. We measured the processing at the entire IP layer, from the moment a packet was received at IP until the packet was passed to the network interface. The machines we used were all 300MHz Pentium II class machines. The number of cycles was counted from when a packet entered the IP layer (at the beginning of function ipintr), until the packet exited the IP layer (right before ip_output calls the first network layer function).

Baseline Experiments

We ran two baseline experiments: in the first experiment, we sent about 6 million packets from SRC while only H1 was a member of the multicast group; in the second experiment we sent the same number of packets from SRC, but with both H1 and H2 being members. We measured the number of cycles spent at the router to forward packets in both experiments. These numbers provided us with a baseline estimate of how many cycles it takes to forward a regular multicast packets. The results are shown in Table 6.1.

LMS Experiments

Following our baseline experiments, we measured the processing overhead of the forwarding operations at the LMS router. Specifically, we ran two experiments: one to measure the processing overhead to forward a request, and another to measure the overhead for a dmcast. In the first experiment, host H2 sent about 6 million requests to the replier, which the router received and forwarded to H1. In the second experiment, host H1 sent about 6 million dmcasts which were multicast on the

interface leading to H2. The results of these experiments along with the baseline experiments, are summarized in Table 6.1.

Table 6.1: Forwarding cost: Normal v.s. LMS processing (300MHz Pentium II)

Normal IP mcast forwarding to 1 receiver	Normal IP mcast forwarding to 2 receivers	Forwarding a LMS request	Forwarding a LMS dmcast
3702 cycles	6686 cycles	3979 cycles	3734 cycles
12.3 μs	22.3 μs	13.3 μs	12.4 μs

The Table shows the average number of cycles spent at the IP layer to process each packet in the four experiments we described. The Table also shows the average number of microseconds taken to process each packet, which we obtained simply by dividing the number of cycles with the processor speed. A more detailed breakdown of the results is shown in in the plot in Figure 6.16.

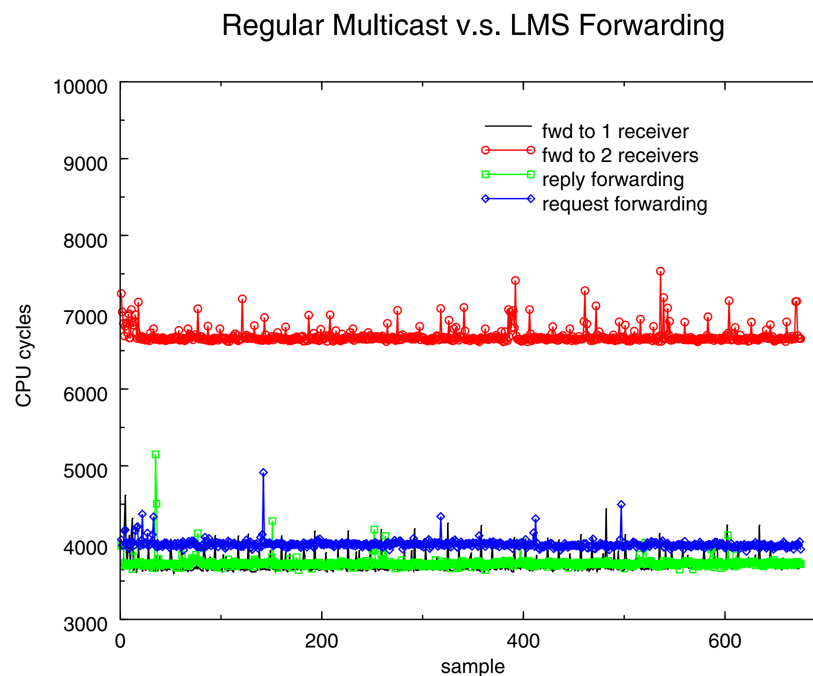

Figure 6.16: Cost of forwarding normal and LMS packets at a router

As we can see from both the table and the plot, the cost of forwarding LMS packets is approximately the same as the cost of forwarding a single multicast packet. It appears that the forwarding

cost of regular multicast packets increases almost linearly as the router has more member interfaces. The cost of LMS packets however, by design remains constant, regardless of how many member interfaces the router has.

The important result of this section is that the cost of forwarding LMS packets is at *most* on par with the cost of forwarding a regular multicast packet; moreover, it remains constant regardless of the router fan-out. This proves that LMS processing at the routers is *not* a bottleneck.

6.6. Summary

In this chapter we presented the implementation of LMS into the networking portion of NetBSD Unix. With our implementation we were able to provide answers to the questions we posed at the beginning of the chapter. Specifically:

Can LMS be implemented in software?

The answer to this question is “yes.” We have described how LMS can be implemented and we have identified the necessary modifications and supplied the new code that must be added to the existing networking code.

How much effort did it take?

The effort to introduce LMS to the existing architecture turned out to be minimal. The operations required by LMS are simple and easy to understand, and we were able to reuse many of the components of the existing architecture.

How much change did it require to the existing architecture?

Some. The modifications we had to make are, (a) allow LTP to specify options that should be inserted in multicast packets, and allow IP encapsulated packets to carry options. The first modification is very simple, and we expect that it will be in place soon to support IPv6 options [41]. The second, however, is a bit more involved. The current NetBSD code does not allow the reception of IPIP packets unless a tunnel is already in place; we got around this problem by switching over to LMS processing as soon as an IPIP packet with options was detected and performing IP decapsulation in the LMS portion. However, we believe that this should change in the future, if we are to

allow the implementation of services like dmcast. In terms of programming effort, this is a minor change.

How much overhead for LMS packet forwarding?

As we have seen from our experiments, the cost of LMS packet forwarding is on par with normal multicast forwarding when the router has only one downstream member interface. As more interfaces join the group, the cost of normal multicast forwarding increases almost linearly with the number of interfaces, but the cost of LMS forwarding remains the same. Thus, LMS processing at worst, is still the same as normal multicast forwarding. This is an important result because it negates any concerns about LMS adding too much processing at the routers.

In summary, in this chapter we have shown that LMS is not only implementable, but it requires very little effort to implement, only minor changes to existing systems and introduces no more processing overhead than normal multicast. Therefore, our work has provided very strong evidence that there are no serious obstacles in implementing LMS.

6.7. Future Work

In this section we discuss two areas of future work in LMS. The first is the implementation of the LMS_HIER module, and the second is an exploration of whether LMS can be incorporated in the router fast path.

As we mentioned at the beginning of this chapter, we did not include the LMS-HIER module in our implementation. The reason is that it is not yet clear what the functionality of this module should be. For example, one may argue that LMS may require no such module, and routers can simply pick repliers at random with no feedback from receivers. Such an option is attractive in backbone routers, which may typically handle hundreds of thousands of groups at a given time. Our simulation experiments have indicated that the performance of LMS in the presence of random loss is good, even if repliers are picked at random.

Another example where the LMS-HIER component may not be needed is a randomized approach is proposed by Costello and McCanne, called Search Party [30]. This work, which builds on our work by aiming to provide LMS with better robustness and better load distribution at the

expense of increased duplicates and latency. To do so, Search Party allows routers to randomly distribute multiple copies of requests between the downstream and the upstream links. In such an approach, the router does not maintain a specific replier link, and thus does not require the LMS-HIER component.

The reason our experiments and the Search Party approach indicate that the LMS-HIER component may not be required, is that for error control there is a certain flexibility as to where data is recovered from; in other words, in error control, who sends the retransmission is less important; what is of primary importance is that a retransmission is sent. Who sends a retransmission, however, becomes important when we consider issues like security and latency. In addition, as we described in Chapter 4, we envision that LMS will be used for other applications besides error control where tight control of the replier selection may be crucial; thus, if LMS is to be used in such applications, the LMS-HIER component must be implemented. Below we sketch a possible implementation of an LMS-HIER component that maintains tight control over the replier selection. The implementation details of the LMS-HIER module are left as part of future work.

6.7.1. The LMS-HIER Component

Recall that the main purpose of LMS-HIER is to create, disseminate and maintain the replier information among all endpoints and routers participating in a multicast group. This information is maintained on a per-multicast tree basis, i.e., for each <*src, group*> pair in the case of source-based trees, or each <*group*> in the case of shared trees. To do so, LMS-HIER needs to communicate information between (a) the endpoints and the local router, and (b) the network routers that participate in the multicast tree. We call the former the *host-router component*, and the latter the *router-router component*.

The Host-Router Component

To implement the host-router component, we propose two small modifications to the Internet Group Management Protocol (IGMP). IGMP is the protocol used between hosts and routers on the same physical network to tell all systems which hosts belong to which multicast groups. This information is required by routers to know which multicast packets to forward on which network. IGMP is defined in [43].

Briefly, IGMP works as follows: to join a group, a host multicasts an IGMP REPORT message with destination address set to the address of the group it wants to join. Routers by default receive all multicast messages, and are thus notified of the wishes of this host and take appropriate actions to join the group. To maintain group membership, the local router periodically generates an IGMP QUERY message, which it multicasts to the *all-hosts* group address (224.0.0.1), which all multicast-capable hosts are required to join; hosts respond with an IGMP REPORT message for every multicast group they currently subscribe to. A simple randomized back-off algorithm with duplicate suppression ensures that the router is not flooded with reports. IGMP version 2[43], introduces a group-specific query message, and an IGMP LEAVE message. IGMP version 3[58], which is still at a preliminary stage of development, extends IGMP v.2 with GROUP-SOURCE REPORTs and LEAVEs, to allow receivers to receive packets only from a specified set of sources rather than everyone in the multicast group.

To support the LMS-HIER component we propose to modify IGMP v.3 source-group reports to carry an additional field, namely the replier cost. This field is set by receivers wishing to act as repliers. In addition to this new field, one more change is needed to IGMP: in response to a router's IGMP QUERY message, elected repliers instead of participating in the randomized back-off contest, respond immediately with an IGMP REPORT message; other receivers' responses are suppressed as before. If a potentially large number of distinct groups with distinct repliers exist on a particular network, the randomized back-off contest may be re-introduced, but receivers who are not repliers scale the randomized back-off interval to allow repliers to contest first. Repliers are also allowed to send unsolicited IGMP reports (i.e., ones that were not triggered by IGMP queries) when their replier cost has changed. The router accepts these reports and updates its replier cost accordingly.

We believe that the above modifications are simple and will not disrupt the existing operation of IGMP.

The Router-Router Component

Recall that the router-router component of LMS-HIER requires that routers send information to upstream routers to update the replier cost. One approach, shown in Figure 6.17, is to use existing messages from multicast routing protocols such as DVMRP as vehicles to carry the LMS replier

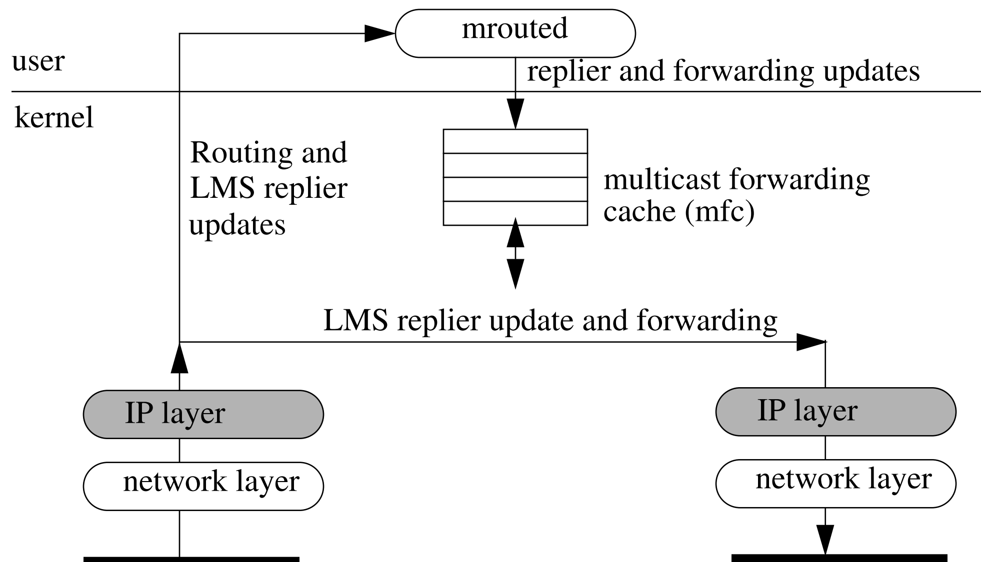

Figure 6.17: Combined routing/LMS updates at a router

information. Such messages are exchanged frequently between routing daemons like mrouted. By piggybacking replier information to routing updates, replier updates can be exchanged between mrouteds, which in turn can update the replier information by making the appropriate changes to the kernel multicast forwarding cache. One problem with this approach is that replier updates are scheduled only when routing updates take place, which may not be frequently enough to guard against replier failure.

Another approach, which responds faster to replier failure, is to create a new replier update protocol for the exclusive use of LMS. The protocol would be similar to protocols used for routing updates, in that it would exchange messages between neighboring mrouteds. This protocol should allow both periodic and on-demand replier updates. Periodic updates would prevent soft state from expiring, and on-demand updates would allow fast convergence of the replier hierarchy.

6.7.2. LMS in the Fast Path

In addition to the implementation of LMS-HIER component, another important question we are not able to answer in this chapter is whether LMS can be incorporated into routers which employ a *fast forwarding path*. A fast path is a highly optimized version of the portion of the forwarding code

used during the common case forwarding of a packet; for speed reasons, routers (like high-end backbone routers) implement this portion in hardware. However, evaluating whether LMS can be deployed in a router's fast path is a difficult without access to information about the hardware architecture of such routers. While in this chapter we can only deal with the issue of integrating LMS into an existing software architecture, our work is still useful for the following reasons:

- Not all routers in the Internet have a hardware-supported fast path, and thus our software implementation may be adopted directly by such routers.

- Our software implementation shows that the added code is similar in complexity, structure and overhead to the existing forwarding code. This leads us to speculate that if the existing forwarding operations can be implemented in hardware, then it is very likely that LMS can also be implemented in hardware.

- Many routers come with line cards which contains processors running software. Such line-cards can very easily accommodate LMS.

As future work, we would like to evaluate the possibility of a fast path implementation of LMS and in router line cards.