            Socket Interface Extensions for Multicast Source Filters

Status of this Memo

Copyright Notice

Abstract

   The Internet Group Management Protocol (IGMPv3) for IPv4 and the
   Multicast Listener Discovery (MLDv2) for IPv6 add the capability for
   applications to express source filters on multicast group
   memberships, which allows receiver applications to determine the set
   of senders (sources) from which to accept multicast traffic. This
   capability also simplifies support of one-to-many type multicast
   applications.

   This document specifies new socket options and functions to manage
   source filters for IP Multicast group memberships. It also defines
   the socket structures to provide input and output arguments to these
   new application program interfaces (APIs). These extensions are
   designed to provide access to the source filtering features, while
   introducing a minimum of change into the system and providing
   complete compatibility for existing multicast applications.

Table of Contents

1. Introduction

   The de facto standard application program interface (API) for TCP/IP
   applications is the "sockets" interface. Although this API was
   developed for Unix in the early 1980s it has also been implemented on
   a wide variety of non-Unix systems. TCP/IP applications written
   using the sockets API have in the past enjoyed a high degree of
   portability and we would like the same portability with applications
   that employ multicast source filters. Changes are required to the
   sockets API to support such filtering and this memo describes these
   changes.

   This document specifies new socket options and functions to manage
   source filters for IP Multicast group memberships. It also defines
   the socket structures to provide input and output arguments to these
   new APIs. These extensions are designed to provide access to the
   source filtering features required by applications, while introducing
   a minimum of change into the system and providing complete
   compatibility for existing multicast applications.

   Furthermore, RFC 3493 [1] defines socket interface extensions for
   IPv6, including protocol-independent functions for most operations.

However, while it defines join and leave functions for IPv6, it does
not provide protocol-independent versions of these operations. Such
functions will be described in this document.

The reader should note that this document is for informational
purposes only, and that the official standard specification of the
sockets API is [2].

2. Design Considerations

There are a number of important considerations in designing changes
to this well-worn API:

    o The API changes should provide both source and binary
       compatibility for programs written to the original API. That
       is, existing program binaries should continue to operate when
       run on a system supporting the new API. In addition, existing
       applications that are re-compiled and run on a system
       supporting the new API should continue to operate. Simply put,
       the API changes for multicast receivers that specify source
       filters should not break existing programs.

    o The changes to the API should be as small as possible in order
       to simplify the task of converting existing multicast receiver
       applications to use source filters.

    o Applications should be able to detect when the new source
       filter APIs are unavailable (e.g., calls fail with the ENOTSUPP
       error) and react gracefully (e.g., revert to old non-source-
       filter API or display a meaningful error message to the user).

    o Lack of type-safety in an API is a bad thing which should be
       avoided when possible.

Several implementations exist that use ioctl() for a portion of the
functionality described herein, and for historical purposes, the
ioctl API is documented in Appendix A. The preferred API, however,
includes new functions. The reasons for adding new functions are:

    o New functions provide type-safety, unlike ioctl, getsockopt,
       and setsockopt.

    o A new function can be written as a wrapper over an ioctl,
       getsockopt, or setsockopt call, if necessary. Hence, it
       provides more freedom as to how the functionality is
       implemented in an operating system. For example, a new
       function might be implemented as an inline function in an

      include file, or a function exported from a user-mode library
      which internally uses some mechanism to exchange information
      with the kernel, or be implemented directly in the kernel.

   o At least one operation defined herein needs to be able to both
      pass information to the TCP/IP stack, as well as retrieve
      information from it. In some implementations this is
      problematic without either changing getsockopt or using ioctl.
      Using new functions avoids the need to change such
      implementations.

2.1. What Needs to be Added

   The current IP Multicast APIs allow a receiver application to specify
   the group address (destination) and (optionally) the local interface.
   These existing APIs need not change (and cannot, to retain binary
   compatibility). Hence, what is needed are new source filter APIs
   that provide the same functionality and also allow receiver multicast
   applications to:

   o Specify zero or more unicast (source) address(es) in a source
      filter.

   o Determine whether the source filter describes an inclusive or
      exclusive list of sources.

   The new API design must enable this functionality for both IPv4 and
   IPv6.

2.2. Data Types

   The data types of the structure elements given in this memo are
   intended to be examples, not absolute requirements. Whenever
   possible, data types from POSIX 1003.1g [2] are used: uintN_t means
   an unsigned integer of exactly N bits (e.g., uint32_t).

2.3. Headers

   When function prototypes and structures are shown, we show the
   headers that must be #included to cause that item to be defined.

2.4. Structures

   When structures are described, the members shown are the ones that
   must appear in an implementation. Additional, nonstandard members
   may also be defined by an implementation. As an additional

precaution, nonstandard members could be verified by Feature Test
Macros as described in [2]. (Such Feature Test Macros are not
defined by this RFC.)

The ordering shown for the members of a structure is the recommended
ordering, given alignment considerations of multibyte members, but an
implementation may order the members differently.

3. Overview of APIs

There are a number of different APIs described in this document that
are appropriate for a number of different application types and IP
versions. Before providing detailed descriptions, this section
provides a "taxonomy" with a brief description of each.

There are two categories of source-filter APIs, both of which are
designed to allow multicast receiver applications to designate the
unicast address(es) of sender(s) along with the multicast group
(destination address) to receive.

   o Basic (Delta-based): Some applications desire the simplicity of
      a delta-based API in which each function call specifies a
      single source address which should be added to or removed from
      the existing filter for a given multicast group address on
      which to listen. Such applications typically fall into either
      of two categories:

      + Any-Source Multicast: By default, all sources are accepted.
         Individual sources may be turned off and back on as needed
         over time. This is also known as "exclude" mode, since the
         source filter contains a list of excluded sources.

      + Source-Specific Multicast: Only sources in a given list are
         allowed. The list may change over time. This is also known
         as "include" mode, since the source filter contains a list
         of included sources.

         This API would be used, for example, by "single-source"
         applications such as audio/video broadcasting. It would
         also be used for logical multi-source sessions where each
         source independently allocates its own Source-Specific
         Multicast group address.

   o Advanced (Full-state): This API allows an application to define
      a complete source-filter comprised of zero or more source
      addresses, and replace the previous filter with a new one.

           Applications which require the ability to switch between filter
           modes without leaving a group must use a full-state API (i.e.,
           to change the semantics of the source filter from inclusive to
           exclusive, or vice versa).

           Applications which use a large source list for a given group
           address should also use the full-state API, since filter
           changes can be done atomically in a single operation.

   The above types of APIs exist in IPv4-specific variants as well as
   with protocol-independent variants. One might ask why the protocol-
   independent APIs cannot accommodate IPv4 applications as well as
   IPv6. Since any IPv4 application requires modification to use
   multicast source filters anyway, it might seem like a good
   opportunity to create IPv6-compatible source code.

   The primary reasons for extending an IPv4-specific API are:

       o To minimize changes needed in existing IPv4 multicast
          application source code to add source filter support.

       o To avoid overloading APIs to accommodate the differences
          between IPv4 interface addresses (e.g., in the ip_mreq
          structure) and interface indices.

4. IPv4 Multicast Source Filter APIs

   Version 3 of the Internet Group Management Protocol (IGMPv3) [3] and
   version 2 of the Multicast Listener Discovery (MLDv2) protocol [4]
   provide the ability to communicate source filter information to the
   router and hence avoid pulling down data from unwanted sources onto
   the local link. However, source filters may be implemented by the
   operating system regardless of whether the routers support IGMPv3 or
   MLDv2, so when the source-filter API is available, applications can
   always benefit from using it.

4.1. Basic (Delta-based) API for IPv4

   The reception of multicast packets is controlled by the setsockopt()
   options summarized below. An error of EOPNOTSUPP is returned if
   these options are used with getsockopt().

   The following structures are used by both the Any-Source Multicast
   and the Source-Specific Multicast API:

   #include <netinet/in.h>

   struct ip_mreq {
      struct in_addr imr_multiaddr; /* IP address of group */
      struct in_addr imr_interface; /* IP address of interface */
   };

   struct ip_mreq_source {
      struct in_addr imr_multiaddr; /* IP address of group */
      struct in_addr imr_sourceaddr; /* IP address of source */
      struct in_addr imr_interface; /* IP address of interface */
   };

4.1.1. IPv4 Any-Source Multicast API

   The following socket options are defined in <netinet/in.h> for
   applications in the Any-Source Multicast category:

   Socket option   Argument type
   IP_ADD_MEMBERSHIP   struct ip_mreq
   IP_BLOCK_SOURCE   struct ip_mreq_source
   IP_UNBLOCK_SOURCE   struct ip_mreq_source
   IP_DROP_MEMBERSHIP   struct ip_mreq

   IP_ADD_MEMBERSHIP and IP_DROP_MEMBERSHIP are already implemented on
   most operating systems, and are used to join and leave an any-source
   group.

   IP_BLOCK_SOURCE can be used to block data from a given source to a
   given group (e.g., if the user "mutes" that source), and
   IP_UNBLOCK_SOURCE can be used to undo this (e.g., if the user then
   "unmutes" the source).

4.1.2. IPv4 Source-Specific Multicast API

   The following socket options are available for applications in the
   Source-Specific category:

   Socket option   Argument type
   IP_ADD_SOURCE_MEMBERSHIP   struct ip_mreq_source
   IP_DROP_SOURCE_MEMBERSHIP struct ip_mreq_source
   IP_DROP_MEMBERSHIP   struct ip_mreq

   IP_ADD_SOURCE_MEMBERSHIP and IP_DROP_SOURCE_MEMBERSHIP are used to
   join and leave a source-specific group.

IP_DROP_MEMBERSHIP is supported, as a convenience, to drop all
sources which have been joined for a particular group and interface.
The operations are the same as if the socket had been closed.

4.1.3. Error Codes

When the option would be legal on the group, but an address is
invalid (e.g., when trying to block a source that is already blocked
by the socket, or when trying to drop an unjoined group) the error
generated is EADDRNOTAVAIL.

When the option itself is not legal on the group (i.e., when trying a
Source-Specific option on a group after doing IP_ADD_MEMBERSHIP, or
when trying an Any-Source option without doing IP_ADD_MEMBERSHIP) the
error generated is EINVAL.

When any of these options are used with getsockopt(), the error
generated is EOPNOTSUPP.

Finally, if the implementation imposes a limit on the maximum number
of sources in a source filter, ENOBUFS is generated when an operation
would exceed the maximum.

4.2. Advanced (Full-state) API for IPv4

Several implementations exist that use ioctl() for this API, and for
historical purposes, the ioctl() API is documented in Appendix A.
The preferred API uses the new functions described below.

4.2.1. Set Source Filter

    #include <netinet/in.h>

    int setipv4sourcefilter(int s, struct in_addr interface,
                            struct in_addr group, uint32_t fmode,
                            uint32_t numsrc, struct in_addr *slist);

On success the value 0 is returned, and on failure, the value -1 is
returned and errno is set accordingly.

The s argument identifies the socket.

The interface argument holds the local IP address of the interface.

The group argument holds the IP multicast address of the group.

The fmode argument identifies the filter mode. The value of this
field must be either MCAST_INCLUDE or MCAST_EXCLUDE, which are
likewise defined in <netinet/in.h>.

The numsrc argument holds the number of source addresses in the slist
array.

The slist argument points to an array of IP addresses of sources to
include or exclude depending on the filter mode.

If the implementation imposes a limit on the maximum number of
sources in a source filter, ENOBUFS is generated when the operation
would exceed the maximum.

4.2.2. Get Source Filter

```
   #include <netinet/in.h>

   int getipv4sourcefilter(int s, struct in_addr interface,
                           struct in_addr group, uint32_t *fmode,
                           uint32_t *numsrc, struct in_addr *slist);
```

On success the value 0 is returned, and on failure, the value -1 is
returned and errno is set accordingly.

The s argument identifies the socket.

The interface argument holds the local IP address of the interface.

The group argument holds the IP multicast address of the group.

The fmode argument points to an integer that will contain the filter
mode on a successful return. The value of this field will be either
MCAST_INCLUDE or MCAST_EXCLUDE, which are likewise defined in
<netinet/in.h>.

On input, the numsrc argument holds the number of source addresses
that will fit in the slist array. On output, the numsrc argument
will hold the total number of sources in the filter.

The slist argument points to buffer into which an array of IP
addresses of included or excluded (depending on the filter mode)
sources will be written. If numsrc was 0 on input, a NULL pointer
may be supplied.

   If the application does not know the size of the source list
   beforehand, it can make a reasonable guess (e.g., 0), and if upon
   completion, numsrc holds a larger value, the operation can be
   repeated with a large enough buffer.

   That is, on return, numsrc is always updated to be the total number
   of sources in the filter, while slist will hold as many source
   addresses as fit, up to the minimum of the array size passed in as
   the original numsrc value and the total number of sources in the
   filter.

5. Protocol-Independent Multicast Source Filter APIs

   Protocol-independent functions are provided for join and leave
   operations so that an application may pass a sockaddr_storage
   structure obtained from calls such as getaddrinfo() [1] as the group
   to join. For example, an application can resolve a DNS name (e.g.,
   NTP.MCAST.NET) to a multicast address which may be either IPv4 or
   IPv6, and may easily join and leave the group.

5.1. Basic (Delta-based) API

   The reception of multicast packets is controlled by the setsockopt()
   options summarized below. An error of EOPNOTSUPP is returned if
   these options are used with getsockopt().

   The following structures are used by both the Any-Source Multicast
   and the Source-Specific Multicast API: #include <netinet/in.h>

   struct group_req {
       uint32_t   gr_interface; /* interface index */
       struct sockaddr_storage gr_group;   /* group address */
   };

   struct group_source_req {
       uint32_t   gsr_interface; /* interface index */
       struct sockaddr_storage gsr_group;   /* group address */
       struct sockaddr_storage gsr_source;   /* source address */
   };

   The sockaddr_storage structure is defined in RFC 3493 [1] to be large
   enough to hold either IPv4 or IPv6 address information.

   The rules for generating errors are the same as those given in
   Section 5.1.3.

5.1.1. Any-Source Multicast API

      Socket option   Argument type
      MCAST_JOIN_GROUP   struct group_req
      MCAST_BLOCK_SOURCE   struct group_source_req
      MCAST_UNBLOCK_SOURCE   struct group_source_req
      MCAST_LEAVE_GROUP   struct group_req

      MCAST_JOIN_GROUP and MCAST_LEAVE_GROUP are used to join and leave an
      any-source group.

      MCAST_BLOCK_SOURCE can be used to block data from a given source to a
      given group (e.g., if the user "mutes" that source), and
      MCAST_UNBLOCK_SOURCE can be used to undo this (e.g., if the user then
      "unmutes" the source).

5.1.2. Source-Specific Multicast API

      Socket option   Argument type
      MCAST_JOIN_SOURCE_GROUP   struct group_source_req
      MCAST_LEAVE_SOURCE_GROUP   struct group_source_req
      MCAST_LEAVE_GROUP   struct group_req

      MCAST_JOIN_SOURCE_GROUP and MCAST_LEAVE_SOURCE_GROUP are used to join
      and leave a source-specific group.

      MCAST_LEAVE_GROUP is supported, as a convenience, to drop all sources
      which have been joined for a particular group and interface. The
      operations are the same as if the socket had been closed.

5.2. Advanced (Full-state) API

      Implementations may exist that use ioctl() for this API, and for
      historical purposes, the ioctl() API is documented in Appendix A.
      The preferred API uses the new functions described below.

5.2.1. Set Source Filter

         #include <netinet/in.h>

         int setsourcefilter(int s, uint32_t interface,
                             struct sockaddr *group, socklen_t grouplen,
                             uint32_t fmode, uint_t numsrc,
                             struct sockaddr_storage *slist);

      On success the value 0 is returned, and on failure, the value -1 is
      returned and errno is set accordingly.

The s argument identifies the socket.

The interface argument holds the interface index of the interface.

The group argument points to either a sockaddr_in structure (for
IPv4) or a sockaddr_in6 structure (for IPv6) that holds the IP
multicast address of the group.

The grouplen argument gives the length of the sockaddr_in or
sockaddr_in6 structure.

The fmode argument identifies the filter mode. The value of this
field must be either MCAST_INCLUDE or MCAST_EXCLUDE, which are
likewise defined in <netinet/in.h>.

The numsrc argument holds the number of source addresses in the slist
array.

The slist argument points to an array of IP addresses of sources to
include or exclude depending on the filter mode.

If the implementation imposes a limit on the maximum number of
sources in a source filter, ENOBUFS is generated when the operation
would exceed the maximum.

5.2.2. Get Source Filter

```
#include <netinet/in.h>

int getsourcefilter(int s, uint32_t interface,
                    struct sockaddr *group, socklen_t grouplen,
                    uint32_t fmode, uint_t *numsrc,
                    struct sockaddr_storage *slist);
```

On success the value 0 is returned, and on failure, the value -1 is
returned and errno is set accordingly.

The s argument identifies the socket.

The interface argument holds the local IP address of the interface.

The group argument points to either a sockaddr_in structure (for
IPv4) or a sockaddr_in6 structure (for IPv6) that holds the IP
multicast address of the group.

The fmode argument points to an integer that will contain the filter
mode on a successful return. The value of this field will be either
MCAST_INCLUDE or MCAST_EXCLUDE, which are likewise defined in
<netinet/in.h>.

On input, the numsrc argument holds the number of source addresses
that will fit in the slist array. On output, the numsrc argument
will hold the total number of sources in the filter.

The slist argument points to buffer into which an array of IP
addresses of included or excluded (depending on the filter mode)
sources will be written. If numsrc was 0 on input, a NULL pointer
may be supplied.

If the application does not know the size of the source list
beforehand, it can make a reasonable guess (e.g., 0), and if upon
completion, numsrc holds a larger value, the operation can be
repeated with a large enough buffer.

That is, on return, numsrc is always updated to be the total number
of sources in the filter, while slist will hold as many source
addresses as fit, up to the minimum of the array size passed in as
the original numsrc value and the total number of sources in the
filter.

6. Security Considerations

Although source filtering can help to combat denial-of-service
attacks, source filtering alone is not a complete solution, since it
does not provide protection against spoofing the source address to be
an allowed source. Multicast routing protocols which use reverse-
path forwarding based on the source address, however, do provide some
natural protection against spoofing the source address, since if a
router receives a packet on an interface other than the one toward
the "real" source, it will drop the packet. However, this still does
not provide any guarantee of protection.

7. Acknowledgments

This document was updated based on feedback from the IETF's IDMR and
MAGMA Working Groups, and the Austin Group. Wilbert de Graaf also
provided many helpful comments.

8. Appendix A: Use of ioctl() for full-state operations

   The API defined here is historic, but is documented here for
   informational purposes since it is implemented by multiple platforms.
   The new functions defined earlier in this document should now be used
   instead.

   Retrieving the source filter for a given group cannot be done with
   getsockopt() on some existing platforms, since the group and
   interface must be passed down in order to retrieve the correct
   filter, and getsockopt only supports an output buffer. This can,
   however, be done with an ioctl(), and hence for symmetry, both gets
   and sets are done with an ioctl.

8.1. IPv4 Options

   The following are defined in <sys/sockio.h>:

      o ioctl() SIOCGIPMSFILTER: to retrieve the list of source
         addresses that comprise the source filter along with the
         current filter mode.

      o ioctl() SIOCSIPMSFILTER: to set or modify the source filter
         content (e.g., unicast source address list) or mode (exclude or
         include).

```
Ioctl option                   Argument type
SIOCGIPMSFILTER                struct ip_msfilter
SIOCSIPMSFILTER                struct ip_msfilter

struct ip_msfilter {
    struct in_addr imsf_multiaddr; /* IP multicast address of group */
    struct in_addr imsf_interface; /* local IP address of interface */
    uint32_t       imsf_fmode;     /* filter mode */
    uint32_t       imsf_numsrc;    /* number of sources in src_list */
    struct in_addr imsf_slist[1];  /* start of source list */
};

#define IP_MSFILTER_SIZE(numsrc) \
    (sizeof(struct ip_msfilter) - sizeof(struct in_addr) \
    + (numsrc) * sizeof(struct in_addr))
```

   The imsf_fmode mode is a 32-bit integer that identifies the filter
   mode. The value of this field must be either MCAST_INCLUDE or
   MCAST_EXCLUDE, which are likewise defined in <netinet/in.h>.

The structure length pointed to must be at least IP_MSFILTER_SIZE(0)
bytes long, and the imsf_numsrc parameter should be set so that
IP_MSFILTER_SIZE(imsf_numsrc) indicates the buffer length.

If the implementation imposes a limit on the maximum number of
sources in a source filter, ENOBUFS is generated when a set operation
would exceed the maximum.

The result of a get operation (SIOCGIPMSFILTER) will be that the
imsf_multiaddr and imsf_interface fields will be unchanged, while
imsf_fmode, imsf_numsrc, and as many source addresses as fit will be
filled into the application's buffer.

If the application does not know the size of the source list
beforehand, it can make a reasonable guess (e.g., 0), and if upon
completion, the imsf_numsrc field holds a larger value, the operation
can be repeated with a large enough buffer.

That is, on return from SIOCGIPMSFILTER, imsf_numsrc is always
updated to be the total number of sources in the filter, while
imsf_slist will hold as many source addresses as fit, up to the
minimum of the array size passed in as the original imsf_numsrc value
and the total number of sources in the filter.

8.2. Protocol-Independent Options

   The following are defined in <sys/sockio.h>:

       o ioctl() SIOCGMSFILTER: to retrieve the list of source addresses
          that comprise the source filter along with the current filter
          mode.

       o ioctl() SIOCSMSFILTER: to set or modify the source filter
          content (e.g., unicast source address list) or mode (exclude or
          include).

   Ioctl option   Argument type
   SIOCGMSFILTER   struct group_filter
   SIOCSMSFILTER   struct group_filter

   struct group_filter {
      uint32_t   gf_interface; /* interface index */
      struct sockaddr_storage gf_group;   /* multicast address */
      uint32_t   gf_fmode;   /* filter mode */
      uint32_t   gf_numsrc;   /* number of sources */
      struct sockaddr_storage gf_slist[1]; /* source address */
   };

```
#define GROUP_FILTER_SIZE(numsrc) \
    (sizeof(struct group_filter) - sizeof(struct sockaddr_storage) \
    + (numsrc) * sizeof(struct sockaddr_storage))
```

The imf_numsrc field is used in the same way as described for
imsf_numsrc above.

9. Normative References

   [1] Gilligan, R., Thomson, S., Bound, J., McCann, J. and W.
        Stevens, "Basic Socket Interface Extensions for IPv6", RFC 3493,
        February 2003.

   [2] IEEE Std. 1003.1-2001 Standard for Information Technology --
        Portable Operating System Interface (POSIX). Open Group
        Technical Standard: Base Specifications, Issue 6, December 2001.
        ISO/IEC 9945:2002. http://www.opengroup.org/austin

10. Informative References

   [3] Cain, B., Deering, S., Kouvelas, I., Fenner, B. and A.
        Thyagarajan, "Internet Group Management Protocol, Version 3",
        RFC 3376, October 2002.

   [4] Vida, R. and L. Costa, "Multicast Listener Discovery Version 2
        (MLDv2) for IPv6", Work in Progress, December 2003.

11. Authors' Addresses

   Dave Thaler
   Microsoft Corporation
   One Microsoft Way
   Redmond, WA 98052-6399

   Phone: +1 425 703 8835
   EMail: dthaler@microsoft.com

   Bill Fenner
   75 Willow Road
   Menlo Park, CA 94025

   Phone: +1 650 867 6073
   EMail: fenner@research.att.com

   Bob Quinn
   IP Multicast Initiative (IPMI)
   Stardust.com
   1901 S. Bascom Ave. #333
   Campbell, CA 95008

   Phone: +1 408 879 8080
   EMail: rcq@ipmulticast.com

12. Full Copyright Statement

Acknowledgement