SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2010; **00**:1–7 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

Communicating between the kernel and user-space in Linux using Netlink sockets

Pablo Neira Ayuso*,*[1], Rafael M. Gasca[1] and Laurent Lefevre[2]

[1] *QUIVIR Research Group, Departament of Computer Languages and Systems, University of Seville, Spain.*
[2] *RESO/LIP team, INRIA, University of Lyon, France.*

SUMMARY

When developing Linux kernel features, it is a good practise to expose the necessary details to user-space to enable extensibility. This allows the development of new features and sophisticated configurations from user-space. Commonly, software developers have to face the task of looking for a good way to communicate between kernel and user-space in Linux. This tutorial introduces you to Netlink sockets, a flexible and extensible messaging system that provides communication between kernel and user-space. In this tutorial, we provide fundamental guidelines for practitioners who wish to develop Netlink-based interfaces.

KEY WORDS: kernel interfaces, netlink, linux

1. INTRODUCTION

Portable open-source operating systems like Linux [1] provide a good environment to develop applications for the real-world since they can be used in very different platforms: from very small embedded devices, like smartphones and PDAs, to standalone computers and large scale clusters. Moreover, the availability of the source code also allows its study and modification, this renders Linux useful for both the industry and the academia.

The core of Linux, like many modern operating systems, follows a monolithic † design for performance reasons. The main bricks that compose the operating system are implemented

*Correspondence to: Pablo Neira Ayuso, ETS Ingenieria Informatica, Department of Computer Languages and Systems. Avda. Reina Mercedes, s/n, 41012 Seville, Spain.
*E-mail: pneira@us.es
Contract/grant sponsor: Spanish Ministerio de Educación e Innovación; contract/grant number: TIN2009-13714

Received 6 December 2009
Revised 31 March 2010

in kernel-space whereas most kernel subsystem configurations, such as networking and device driver configurations, are set up by means of administrative tools that reside in user-space. In general, the kernel-space subsystems have to provide interfaces to user-space processes to obtain resources, information and services; and to configure, tune and monitor kernel subsystems.

Kernel Interfaces are key parts of operating systems. The more flexible the interface to communicate kernel and user-space is, the more likely tasks can be efficiently implemented in user-space. As a result, this can reduce the common bloat of adding every new feature into kernel-space. This idea is not new as it was introduced in the micro-kernel design by means of flexible messaging-based interfaces.

In this article, we focus on Netlink, which is one of the interfaces that Linux provides to user-space. Netlink is a socket family that supplies a messaging facility based on the BSD socket interface to send and retrieve kernel-space information from user-space. Netlink is more flexible than other Linux kernel interfaces that have been used in Unix-like operating systems to communicate kernel and user-space. Netlink is portable [‡], highly extensible and it supports event-based notifications.

Currently Netlink is used by networking applications like advanced routing [2], IPsec key management tools [3], firewall state synchronization [4], user-space packet enqueueing [5], border gateway routing protocols [6], wireless mesh routing protocols [7] [8] among many others. Nevertheless, there is some initial use of it in other non-networking kernel subsystems like the ACPI subsystem [9].

In this tutorial, we assume that you are familiar with basics on the Linux kernel [10] [11] [12], BSD sockets [13] and C programming [14]. This work is organized as follows: in Section 2, we provide an outlook on the existing interfaces available in Linux for communication between kernel and user-space. Then, Section 3 details Netlink sockets including features and an extensive protocol description. We have also covered GeNetlink in Section 4, a generic Netlink multiplexer which is widely used these days. We continue with a short introduction to Netlink sockets programming and we provide one reference to online source code examples in Section 5. This work concludes with the list of existing related works and documentation in Section 6; and the conclusions in Section 7.

2. LINUX KERNEL INTERFACES

Linux provides several interfaces to user-space applications that are used for different purposes and that have different properties by design. We have classified the desired properties that a kernel interface should provide, they are:

[†]The core of Linux is monolithic, but most non-essential features can be optionally built as dynamically loadable modules.
[‡]From architectural point of view as opposed to accross different Operating Systems. We have to remark that Netlink is only implemented in Linux by now.

1. Architectural portability: some architectures allow the use of different word size in user and kernel-space. For example, the x86_64 architecture in compatibility mode allows applications to use 32-bit word size in user-space and native 64-bit word size in kernel-space. This is an issue if the kernel interface allows to pass data between kernel and user-space whose size depends on the word size, like pointers and long integer variables. Although in essence all of the kernel interfaces can manage data in a portable format, the necessary mechanisms to ensure architectural portability are not usually covered in the interface design. To resolve this issue, several Linux kernel interfaces require a compatibility layer to convert data to the appropriate word size.
2. Event-based signaling mechanisms: they allow to deliver events so that user-space processes do not have to poll for data to make sure that have up-to-date information on some kernel-space aspect.
3. Extensibility: if the information that is passed between user and kernel-space is represented in a fixed format, like a list of values separated by commas in plain text or a data structure, this format cannot be changed, otherwise backward compatibility will be broken. Thus, if new features require to modify the information format between user and kernel-space, you will have to add a new redundant operation to ensure backward compatibility.
4. Large data transfers: the kernel interface should allow to transfer large amounts of information. This is particularly useful to restore large configurations. Efficient data transfer is also a desired property to reduce the time required to load configurations.

As in many other modern operating systems, system calls are the main kernel interface in Linux. System calls provide generic and standarised services to user-space processes that aim to be portable between different operating systems (this is the case of POSIX system calls). However, Linux kernel developers are very reluctant to add new system calls as they must be proved to be oriented for general purpose, not for every single specific kernel subsystem operation and configuration.

There are several Linux kernel interfaces that are based upon system calls. Basically, these interfaces support a subset of the existing system calls as methods to operate with them. These interfaces can be grouped into virtual filesystems, that provide file-like access to kernel-space features (Sect. 2.1), and BSD sockets (Sect 2.2) that allow to send data to and receive it from kernel subsystems using the BSD sockets API.

Most virtual filesystems and sockets support a system call method that has been commonly used to set up configurations from user-space, the so-called *ioctls*. Basically, each configuration operation is uniquely identified by an ioctl number. The ioctl method also take, as parameter, a pointer to the memory address that contains the configuration expressed in a certain format. Traditionally, fixed-layout data structures are used to format the information that is passed between user and kernel-space in ioctls. Thus, if the addition of new configurations requires to modify the structure layout, you will have to add a new ioctl operation. Otherwise, the layout modification results in backward compatibility issues since the binary interface changes. In general, the fixed-layout format is a problem for Linux kernel interfaces that use it to transfer configurations between kernel and user-space.

In the following subsections, we discuss the existing Linux kernel interfaces categorized by group with regards to the desired properties that we have proposed. We have also summarized these kernel interfaces and their design properties in Table. I.

2.1. Virtual filesystem interfaces

Linux provides several virtual filesystem interfaces that can be used to communicate kernel subsystems and user-space applications. They are the character and block driver interfaces, /proc files and sysfs.

The character and block driver interfaces allow to read data from and write it to a specific device like accessing a file (that is located in the /dev directory). These interfaces allow the transfer of data between kernel and user-space; and they are rather extensible, provided sufficient preparation on the receiver. However, these interfaces are usually reserved for kernel device drivers ¶. Therefore, Linux kernel developers consider that it is not a good practise to use them for new kernel subsystems that are not attached to some hardware device. The duplication of these sort of interfaces, that provide different semantics for hardware that is similar, is also a common problem.

Another choice is the /proc interface. To operate with it, the user-space application has to open the /proc file to read and write information to kernel-space like in regular files. The information format is usually human-readable for accessing or changing configurations. This format is impractical for user-space applications since they have to parse and convert data to an internal representation. Moreover, this mechanism does not allow neither any sort of event-based signaling nor extensions without breaking backward compatibility, and data transfers are limited to one memory page. They are also usually written ad-hoc for one specific task. These interfaces were originally designed to store process information ¶.

Sysfs [15] is a virtual filesystem that was introduced to provide a file-like interface to device drivers and other kernel configurations; and to avoid the abuse of the /proc interface for non-process information. It allows to export kernel objects like directories, object attributes like regular files and object relationships like symbolic links to user-space. The information is organized in plain text and it provides event-based signaling which is implemented over Netlink. Sysfs gets rid of the ad-hoc information format that /proc interfaces provide by following the "one configuration parameter per file" basis. Nevertheless, since the Linux kernel does not provide a stable ABI [16], there are aspects of the this interface that may not be stable across kernel releases. Sysfs also provides a text-based interface that is impractical for user-space application. Data transfers are also limited to one memory page.

2.2. BSD Socket interfaces

In Linux networking kernel subsystems, it has been a common practise to use datagram sockets and ioctl operations to set up configurations from user-space. In this case, user-

¶Although they have been abused for many different purposes.

Table I. Linux kernel interfaces by design properties

Type	Architectural portable[†]	Event-based signaling	Easily extensible[†]	Large data transfers
System call	No	No[‖]	No	Yes
/dev	No	No[**]	No	Yes
/proc	Yes[‡]	No	No	No
Sysfs	Yes[‡]	Yes[‡‡]	No	No
Sockets[*]	No	No	No	Yes
Netlink	Yes	Yes	Yes	Yes

space applications create a datagram socket in some specific domain, such as AF_INET for IPv4. Then, user-space and networking kernel subsystems use fixed-layout data structures that contain the networking configuration to be added, updated or deleted; and they transfer these informations by means ioctls. As we have discussed, the use of the fixed-layout data structures is not flexible enough to add new features and extensions.

Another mechanism to configure networking aspects are socket options. This is the case of the firewalling kernel subsystem in Linux [17] that provides several specific socket options. These socket options, that are attached to the AF_RAW socket family, allows user-space administrative tools to add, update and delete firewall rule-sets. Basically, the user-space application creates an AF_RAW socket that allows to send and receive fixed-layout data structures that contain the firewall configuration. Thus, fixed-layout data structures, with their limitations, are again used [§].

Netlink is a socket family [††] that provide by design an extensible and architectural portable data format to communicate user and kernel-space. This format ensures that new features can be added without breaking backward compatibility. Thus, Netlink overcomes the limitations that we have exposed in the existing Linux kernel interfaces. Moreover, it provides event-based notifications and it allows large data transfers in an efficient way, as we further detail in this work.

[†]The interface provides generic facilities by design (as opposed to ad-hoc) to fulfill these properties.
[‖]Although there is primitive event notification in the form of poll() and select().
[**]Although you may implement the appropriate queuing infrastructure to support event reporting upon ioctls. This is the case of the Linux kernel input subsystem and their /dev/input/eventX files.
[‡]They are text-based interfaces.
[‡‡]It uses Netlink.
[*]Including Socket options. Excluding Netlink.
[§]In the specific case of the firewalling kernel subsystem, there is a primitive revision mechanism that allows to change the structure layout. This mechanism was introduced years after the initial interface design.
[††]BSD Unix introduced the use of specific socket families, such as AF_ROUTE, to configure network routing. This socket family is similar in nature to Netlink since it provides more flexible data format and it allows event

3. NETLINK SOCKETS

Netlink was added by Alan Cox during Linux kernel 1.3 development as a character driver interface [‡] to provide multiple kernel and user-space bidirectional communications links. Then, Alexey Kuznetsov extended it during Linux kernel 2.1 development to provide a flexible and extensible messaging interface to the new advanced routing infrastructure. Since then, Netlink sockets have become one of the main interfaces that kernel subsystems provide to user-space applications in Linux.

Netlink follows the same design principle of the Linux development (quoting Linus Torvalds: "Linux is evolution, not intelligent design"), this means that there is no complete specification (see Section. 6 for a detailed list of existing documentation) nor design documents of Netlink rather than the source code.

3.1. What is Netlink?

Netlink is a datagram-oriented messaging system that allows passing messages from kernel to user-space and vice-versa. It can be also used as an InterProcess Communication (IPC) system. In this work we focus on communicating kernel and user-space, thus, we do not cover the IPC aspect of Netlink that allows communicating two user-space processes or even two different kernel subsystems.

Netlink is implemented on top of the generic BSD socket infrastructure, thus, it supports usual primitives like *socket*(), *bind*(), *sendmsg*() and *recvmsg*() as well as common socket polling mechanisms.

There was an initial effort in 2001 started by Jamal Hadi Salim [25] at the ForCES IETF group [26] to standarize it as a protocol between a *Forwarding Engine Component*, the part of the router that enables the forwarding, and a *Control Plane Component*, the counterpart responsible for managing and configuring the forwarding engine. This effort was discontinued and, instead, a domain specific protocol was designed.

3.2. Netlink socket bus

Netlink allows up to 32 busses [§] in kernel-space. Generally speaking, each bus is attached to one kernel subsystem although several kernel subsystems may share the same. This is the case of the Netfilter [18] bus *nfnetlink* which is used by all the firewalling subsystems available in Linux, and the networking bus *rtnetlink* which is used by the networking device management, routing, neighbouring and queueing discipline subsystems.

notifications to report changes in the routing to user-space applications. This socket family is emulated over Netlink in Linux. See FreeBSD's route(4) manpages for more information.

[‡]Via /dev interface which has been traditionally reserved for device drivers, as exposed in Sect. 2.

[§]Currently, this is artificially limited to 32 busses for efficiency, although this limit could be removed to reach up to 256 busses in the future.

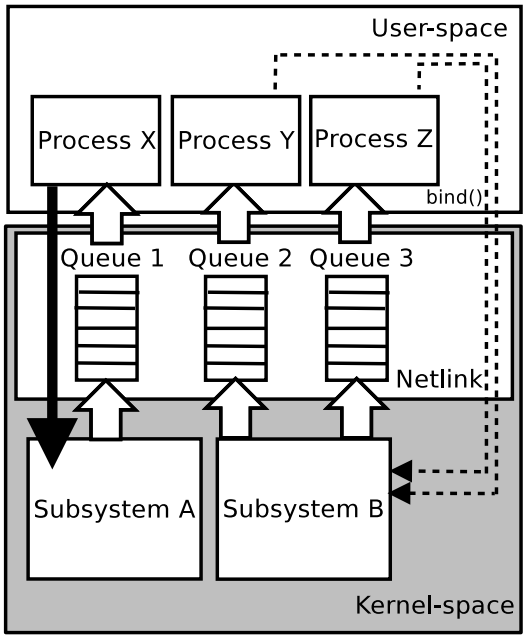

Figure 1. Example scenario of unicast and multicast Netlink sockets

3.3. Netlink communication types

The Netlink family allows two sort of communications, unicast and multicast:

1. Unicast is useful to establish a 1:1 communication channel between a kernel subsystem and one user-space process. Typically, unicast channels are used to send commands to kernel-space, to receive the result of commands, and to request some information to a given kernel subsystem.
2. Multicast is useful to establish a 1:N communication channels. Typically the sender is the kernel and there are N possible listeners in user-space. This is useful for event-based notification. In general, event notifications are grouped in different kinds, thus, each Netlink bus may offer several multicast groups that user-space listeners can subscribe to, depending on what kind of event notifications they are interested in. You can create up to 2^{32} multicast groups (since Linux kernel 2.6.14, before it was limited to 32 groups).

In Figure 1, we illustrate an example scenario in which there are three user-space processes $Process_X$, $Process_Y$ and $Process_Z$. The $Process_X$ has requested some information to the kernel-space subsystem A via unicast, whereas $Process_Y$ and $Process_Z$ are listening to event notifications from the kernel subsystem B via multicast. Note that $Process_X$'s request that goes from user to kernel-space is not enqueued, it is directly handled by the corresponding Netlink kernel subsystem, thus, the behaviour is synchronous. On the other hand, messages going from kernel to user-space are enqueued so the communication is asynchronous.

3.4. Netlink message format

Netlink messages are aligned to 32 bits and, generally speaking, they contain data that is expressed in host-byte order [†]. A Netlink message always starts by a fixed header of 16 bytes defined by *struct nlmsghdr* in *<include/linux/netlink.h>*. We have represented this header in Figure 2. This header contains the following fields:

- Message length (32 bits): size of the message in bytes, including this header.
- Message type (16 bits): the type of this message. There are two sorts, data and control messages. Data messages depend on the set of actions that the given kernel-space subsystem allows. Control messages are common to all Netlink subsystems, there are currently four types of control messages, although there are 16 slots reserved (see NLM_MIN_TYPE constant in linux/netlink.h). The existing control types are:
  - NLMSG_NOOP: no operation, this can be used to implement a Netlink ping utility to know if a given Netlink bus is available.
  - NLMSG_ERROR: this message contains an error.
  - NLMSG_DONE: this is the trailing message that is part of a multi-part message. A multi-part message is composed of a set of messages all with the NLM_F_MULTI [‡] flag set.
  - NLMSG_OVERRUN: this control message type is currently unused.
- Message flags (16 bits): several message flags like:
  - NLM_F_REQUEST: if this flag is set, this Netlink message contains a request. Messages that go from from user to kernel-space must set this flag, otherwise the kernel subsystem must report an *invalid argument* (EINVAL) error to the user-space sender.
  - NLM_F_CREATE: the user-space application wants to issue a command or add a new configuration to the kernel-space subsystem.
  - NLM_F_EXCL: this is commonly used together with NLM_F_CREATE to trigger an error if the configuration that user-space wants to add already exists in kernel-space.
  - NLM_F_REPLACE: the user-space application wants to replace an existing configuration in the kernel-space subsystem.
  - NLM_F_APPEND: append a new configuration to the existing one. This is used for ordered data, like routing information, where the default is otherwise to prepend.
  - NLM_F_DUMP: the user-space application wants a full resynchronization with the kernel subsystem. The result is a batch of Netlink messages, also known as multi-part messages, that contain the kernel subsystem information.

[†] There are some exceptions like nfnetlink which encodes the value of the attributes in network-byte order.
[‡] Some buggy kernel subsystems used to forget to set the NLM_F_MULTI flag in multi-part messages.

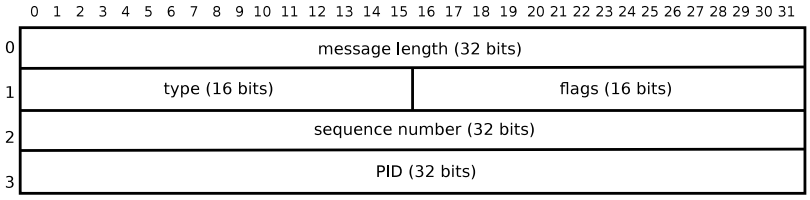

Figure 2. Layout of a Netlink message header

  - NLM_F_MULTI: this is a multi-part message. A Netlink subsystem replies with a multi-part message if it has previously received a request from user-space with the NLM_F_DUMP flag set.
  - NLM_F_ACK: the user-space application requested a confirmation message from kernel-space to make sure that a given request was successfully performed. If this flag is not set, the kernel-space reports the error synchronously via *sendmsg()* as errno value.
  - NLM_F_ECHO: if this flag is set, the user-space application wants to get a report back via unicast of the request that it has send. However, if the user-space application is also subscribed to event-based notifications, it does not receive any notification via multicast as it already receives it via unicast.

- Sequence number (32 bits): message sequence number. This is useful together with NLM_F_ACK if an user-space application wants to make sure that a request has been correctly issued. Netlink uses the same sequence number in the messages that are sent as reply to a given request [§]. For event-based notifications from kernel-space, this is always zero.
- Port-ID (32 bits): this field contains a numerical identifier that is assigned by Netlink. Netlink assigns different port-ID values to identify several socket channels opened by the same user-space process. The default value for the first socket is the Process IDentifier (PID). Under some circunstances, this value is set to zero, they are:

  - This message comes from kernel-space.
  - This message comes from user-space, and we want Netlink automatically set the value according to the corresponding port ID assigned to this socket channel.

Some existing Linux kernel subsystems also add an extra header of fixed size and layout after the Netlink header that enables multiplexing over the same Netlink bus, thus, several kernel subsystems can share only one Netlink socket bus. This is the case of GeNetlink, that we cover in this work in Section 4.

[§]The sequence number is used as a tracking cookie since the kernel does not change the sequence number value at all.

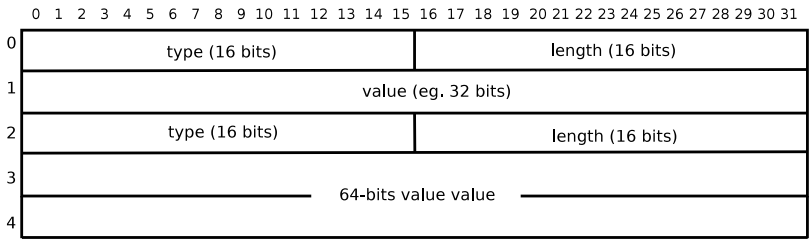

Figure 3. An example of a hypothetical Netlink payload in TLV format

The payload of Netlink messages is composed of a set of attributes that are expressed in Type-Length-Value (TLV) format. TLV structure are used in many protocols due to their extensible nature, e.g. IPv4 options field and IPv6 extension headers. Each Netlink attribute header is defined by *struct nlattr* and it is composed of the following fields:

- Type (16 bits): the attribute type according to the set of available types in the kernel subsystem. The two most significant bits of this field are used to encode if this is a nested attribute (bit 0), which allows you to embed a set of attribute in the payload of one attribute, and if the attribute payload is represented in network byte order (bit 1). Thus, the maximum number of attributes per Netlink subsystem is limited to 16384.
- Length (16 bits): size in bytes of the attribute. This includes this header header plus the payload size of this attribute without alignment to 32 bits.
- Value: this field is variable in size but it is always aligned to 32 bits.

In Figure. 3 we have represented a hypothetical payload in TLV format composed on two attributes, one of 32 bits and another of 64 bits.

The TLV format allows the addition of new attributes without breaking backward compatibility of existing applications. As opposed to ioctl-based mechanisms, you do not have to add a new operation to an existing subsystem. Instead, you only have to add a new attribute and update your user-space application in case that you want to use this new feature. Old user-space applications or kernel-space subsystems will simply ignore new attributes in a message as they do not know how to interpret them.

However, the use of the TLV format is up to the programmer, you may use Netlink to send fixed structures to kernel-space encapsulated in messages although you will not gain the flexibility that the Netlink message format provides. Moreover, if the code is intended for Linux kernel submission, it is likely that such a code will be rejected by other Linux kernel developers. Building and parsing TLV-based messages is more expensive than using a fixed-layout structure but it provides more flexibility.

Another interesting feature of Netlink is attribute nesting that allows you to embed a set of attributes in the payload of one single attribute. For example, if you have a traffic-flow object that you have to expose to user-space via Netlink, the object's layer 3 address can be in IPv4 (32-bits) or IPv6 (128-bits) format, assuming that the system supports both IPv4 and IPv6 at the same time.

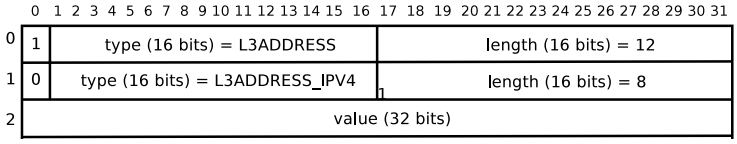

Figure 4. Example of a nested attribute

In Figure. 4, we have represented an example in which we have defined a generic attribute L3ADDRESS that may contain a set of attributes like L3ADDRESS_IPV4 and L3ADDRESS_IPV6. The attributes L3ADDRESS_IPV4 and L3ADDRESS_IPV6 can be used to encapsulate the specific address information. Thus, in the particular case of an IPv4 traffic-flow, L3ADDRESS encapsulates L3ADDRESS_IPV4 attributes. Moreover, if we want to add support for a new layer 3 protocol, eg. Internetwork Packet Exchange (IPX) ¶, we only have to add the new attribute. Basically, there is not limit in the number of nestings, although over-nesting increases the cost of the message building and parsing.

3.5. Netlink error messages

To report errors to userspace, Netlink provides a message type that encapsulates the so-called Netlink error header that we have represented in Figure. 5. This header is defined by *struct nlmsgerr* in *<include/linux/netlink.h>* and it contains two fields:

- Error type (32 bits): this field contains a standarized error value that identifies the sort of error. The error value is defined by *errno.h*. These errors are perror() interpretable.
- Netlink message which contains the request that has triggered the error.

With regards to message integrity, the kernel subsystems that support Netlink usually report *invalid argument* (EINVAL) via *recvmsg()* if user-space sends a malformed message, eg. a Netlink message that does not include attributes that are mandatory for some specific request.

3.6. Netlink reliability mechanisms

In Netlink-based communications, there are two possible situations that may result in message loss:

1. Memory exhaustion: there is no memory available to allocate the message.
2. Buffer overrun: there is no space in the receiver queue that is used to store messages. This may occur in communications from kernel to user-space.

¶IPX is a layer 3 protocol developed by Novel which is very similar to IPv4 although it is in decline since mid-90s.

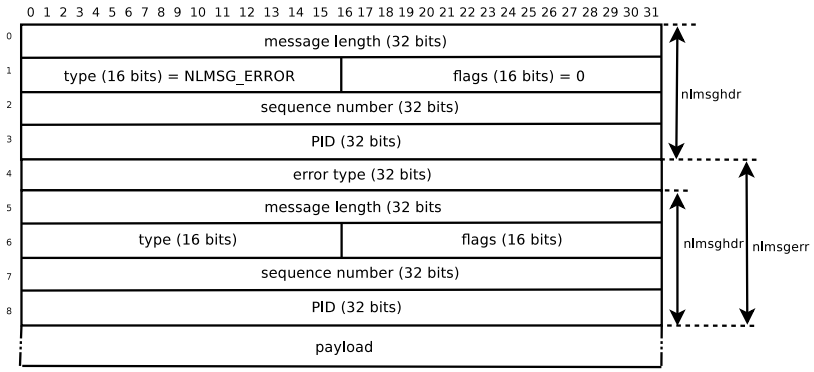

Figure 5. Layout of a Netlink error message

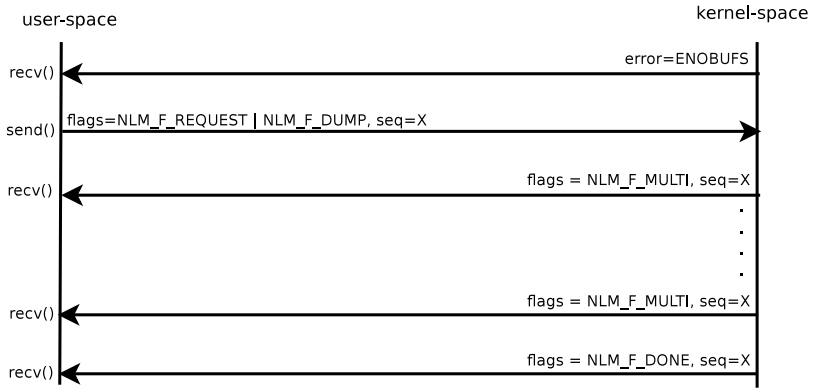

Figure 6. Sequence diagram of a Netlink dump operation

The buffer overrun situation is likely to occur if:

1. A user-space listener is too slow to handle all the Netlink messages that the kernel subsystem sends at a given rate.
2. The queue that is used to store messages that go from kernel to user-space is too small.

If Netlink fails to deliver a message that goes from kernel to user-space, the *recvmsg()* function returns the *No buffer space available* (ENOBUFS) error. Thus, the user-space process knows that it is losing messages so, if the kernel subsystem supports dump operations (that are requested by means of a Netlink message with the NLM_F_DUMP flag set), it can resynchronize itself to obtain up-to-date information. In dump operations, Netlink uses flow control to prevent the overrun of the receiver queue by delivering one packet per *recvmsg()* invocation. This packet consumes one memory page and it contains several multi-part Netlink messages. The sequence diagram in Figure. 6 illustrates the use of the dump operation during a resynchronization.

On the other hand, buffer overruns cannot occur in communications from user to kernel-space since *sendmsg()* synchronously passes the Netlink message to the kernel subsystem.

If blocking sockets are used, Netlink is completely reliable in communications from user to kernel-space since memory allocations would wait, so no memory exhaustion is possible.

Netlink also provides acknowledgment mechanisms so if the user-space sender sets the NLM_F_ACK flag in a request, Netlink reports to user-space the result of the operation that has been requested in a Netlink error message.

4. GENETLINK: GENERIC NETLINK MULTIPLEXATION

GeNetlink, or Generic Netlink, is a multiplexer that is built on top of a Netlink bus and was introduced during the 2.6.15 development cycle. Over the years, Netlink has become very popular, this has brought about a real concern that the number of Netlink busses may be exhausted in the near future. In response to this the Generic Netlink was created [19].

GeNetlink allows to register up to 65520 families that share a single Netlink bus. Each family is intended to be equivalent to a virtual bus. The families that are registered in GeNetlink are identified by a unique string name and ID number. The string name remains the primary key to identify the family, thus, the ID number may change accross different systems.

When GeNetlink is loaded, it initially registers a control family, the so-called *nlctrl*, that provides a lookup facility to obtain the ID number from the string name that identifies families, and event reports to inform about the registration and unregistration of new GeNetlink services and its features. The control family is the only GeNetlink service that has a fixed ID number (GENL_ID_CTRL which is equal to 16) while the vast majority of other families use a system-dependant ID number that is assigned in run-time.

GeNetlink families can also register multicast groups. In this case, the multicast groups are identified by a unique string name that remains the primary key. During the registration of the multicast group, a unique ID number for that group is set. This ID number can change accross different systems. Thus, user-space listeners have to initially obtain the ID number of the multicast group from the string name that they want to join. For that task, they have to use *nlctrl* to look up the ID.

4.1. GeNetlink message format

GeNetlink messages start by the Netlink header, whose message type is the ID number of the GeNetlink service, then it follows the GeNetlink header and finally one optional header that is specific of the given GeNetlink service. Thus, GeNetlink messages may contain up to three headers before the TLV-based payload. Since the ID number of the service is required to build the GeNetlink message, GeNetlink provides a lookup facility to resolve the ID number from the service string name. We have represented the GeNetlink header in Figure. 7.

The GeNetlink header contains three fields:

- The *command* field (8 bits), that is a specific message type of the GeNetlink service.
- The *version* field (8 bits): that contains a revision value to allow changing the format without breaking backward compatibility.

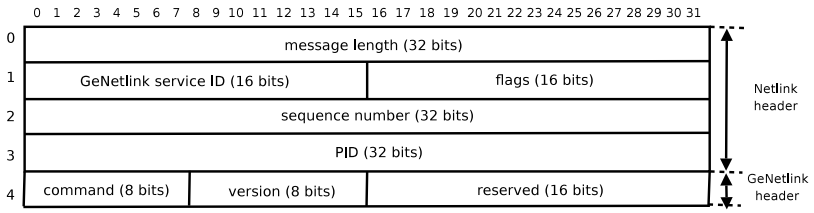

Figure 7. Layout of a GeNetlink header message

- One reserved field (16 bits) which is introduced for padding reasons which is currently unused.

4.2. GeNetlink control family

The GeNetlink control family *nlctrl* provides the following commands to user-space applications:

- CTRL_CMD_GETFAMILY: this command allows to look up for the family ID number from the family name. The response message not only includes the family ID number but also the operations and the multicast groups that the GeNetlink family supports, in the form of TLV attributes.
- CTRL_CMD_GETOPS: this allows to obtain the set of operations that are available for a given family. This message requires the family ID number.
- CTRL_CMD_GETMCAST_GRP [¶]: this allows to obtain the multicast groups that belong to a given GeNetlink family.

Since the family and multicast IDs are assigned in run-time, we initially have to look up for the IDs to send requests and to subscribe to GeNetlink multicast groups from user-space. For that task, the user-space application sends a CTRL_CMD_GETFAMILY request to the GeNetlink control family. Then, once it receives the look-up response that contains the family ID number, the supported operations and the existing multicast groups; the user-space application can build the message for the specific GeNetlink family subsystem to request some operation. The message payload must also contain the mandatory attributes for that operation.

5. PROGRAMMING NETLINK SOCKETS

Adding Netlink support from scratch for some Linux kernel subsystem requires some coding in user and kernel-space. There are a lot of common tasks in parsing, validating, constructing

[¶]As of Linux kernel 2.6.32, this command is not yet implemented.However, it is planned to do it in the future.

of both the Netlink header and TLVs that are repetitive and easy to get wrong. Instead of replicating code, Linux provides a lot of helper functions. Moreover, many existing subsystems in the Linux kernel already support Netlink sockets, thus, the kernel-space coding could be skipped.

From the user-space side, Netlink sockets are implemented on top of the generic BSD sockets interface. Thus, programming Netlink sockets in user-space is similar to programming common TCP/IP sockets. However, we have to take into consideration the aspects that make Netlink sockets different from other socket families, more relevantly:

1. Netlink sockets do not hide protocol details to user-space as other protocols do. In fact, Netlink passes the whole message, including the Netlink header and attributes in TLV format as well as multi-part messages, to user-space. This makes the data handling different than common TCP/IP sockets since the user-space program have to appropriately parse and build Netlink messages according to its format. However, there are no standard facilities to perform these tasks so you would need to implement your own functions or use some existing library to assist your development.
2. Errors that comes from Netlink and kernel subsystems are not returned by *recvmsg()* as an integer. Instead, errors are encapsulated in the Netlink error message. There is one exception to this rule that is the *No buffer space available* (ENOBUFS) error, which is not encapsulated since its purpose is to report that we cannot enqueue a new Netlink message. Standard generic socket errors, like *Resource temporarily unavailable* (EAGAIN), which are commonly used together with polling primitives, like *poll()* and *select()*, are still returned as an integer by *recvmsg()*.

These aspects render the use of Netlink at the BSD socket API layer a daunting task for most developers. In order to simplify the work with Netlink sockets in user-space, we propose the use of user-space libraries such as *libnl* [21] and *libmnl* [20]. These libraries are written in C and that are targeted to Netlink developers.

Detailing Netlink implementation aspects more in-depth is not in the scope of this tutorial. However, we provide reference to commented source code examples online for both kernel and user-space implementations online [29].

6. RELATED WORKS

Several works have tried to document the Netlink sockets in some aspects. The first effort is a Netlink overview created in 1998 [22], this approach provides an introduction and cover some aspects without much detail, it still contains some information that is not very precise.

The most complete tutorial so far was done by Neil Horman [23] that focus on Netlink as the Linux networking *control plane*. There are also available a couple of articles from Linux Magazine [24] covering Netlink sockets from the user-space side to issue commands and obtain information from existing kernel-space networking subsystems that support Netlink [24].

There was an initial effort in 2001 started by Jamal Hadi Salim [25] at the ForCES IETF group [26] to standarize it as a protocol between a *Forwarding Engine Component*, the part

of the router that enables the forwarding, and a *Control Plane Component*, the counterpart responsible for managing and configuring the forwarding engine. This is a good reference, but it is not intended to be a tutorial. There are also some manual pages available, although its content remains limited and quite crytic [27] [28].

The existing tutorials do not cover the kernel-space aspects, they are missing important aspects of the Netlink protocol and they contain imprecisions. For that reason, Netlink main reference has been the Linux kernel source code so far. We expect that this tutorial covers the missing gap in this field.

7. CONCLUSIONS

Netlink is a Linux socket family that in the tradition of Linux kernel development environment is still evolving. Documenting an evolving infrastructure is and reviewing the source should still act as the most up-to-date reference. However, looking at the code as a starting point is hard. For that reason, we expect that this tutorial provides a complete overview for developers and practitioners as a beginning of yet another Netlink/GeNetlink interface for their brand new kernel-space feature or, alternatively, allow them to implement some new feature in user-space from some existing Linux kernel subsystem with Netlink support.

ACKNOWLEDGEMENTS

We would like to thank to Jozsef Kadlecsik of the KFKI Institute and fellow of the Netfilter project, Patrick McHardy who is one of the Linux kernel networking maintainers and head of the Netfilter Project, and Jamal Hadi Salim for their worthy suggestions and feedback.

REFERENCES

1. Torvalds L. et al. *The Linux kernel*. Web pages at: http://www.kernel.org [6 December 2009].
2. Kuznetsov A. *iproute: advanced routing tools for Linux*. Web pages at: http://linux-foundation.org [6 December 2009].
3. Kame project. *IPsec-tools for Linux-2.6, NetBSD and FreeBSD*. Web pages at: http://ipsec-tools.sourceforge.net/ [6 December 2009].
4. Neira-Ayuso, P. *conntrack-tools: The netfilter's connection tracking userspace tools*. Web pages at: http://conntrack-tools.netfilter.org [6 December 2009].
5. Netfilter coreteam. *libnetfilter_queue*. Web pages at: http://www.netfilter.org/projects/libnetfilter_queue/index.html [6 December 2009].
6. Quagga team. *Quagga Routing Software Suite*. Web pages at: http://www.quagga.net [6 December 2009].
7. OLSRd team. *OLSRd: ad-hoc wireless mesh routing daemon*. Web pages at: http://www.olsr.org/ [6 December 2009].
8. Nordstrom H. *AODV-UU: Ad-hoc On-demand Distance Vector Routing from Upsala University*. Web pages at: http://core.it.uu.se/core/index.php/Main_Page [6 December 2009].
9. Hockin T. *ACPId: the ACPI event daemon*. Web pages at: http://acpid.sourceforge.net/ [6 December 2009].
10. Salzman, P.J. et al. *The Linux Kernel Module Programming Guide*. Web pages at: http://tldp.org/LDP/lkmpg/2.6/html/ [6 December 2009].
11. Corbet J., Rubini A., Kroah-Hartman G. *Linux Device Drivers, 3rd edition*, O'Reilly associates, 2008 Web pages at: http://lwn.net/Kernel/LDD3/ [6 December 2009].

12. Love R. *Linux Kernel Development, 2nd edition*, Novell Press, 2003.
13. Stevens R. *UNIX Network Programming, vol.1*, Prentice-Hall, 1998.
14. Kernigham B.W., Ritchie D.M. *C Programming Language, 2nd edition*. Prentice Hall, 1988.
15. Mochel P. *The sysfs filesystem*. The Linux Symposium, vol.1, 313–326, 2005. Ottawa, Canada. Web pages at: http://www.kernel.org/pub/linux/kernel/people/mochel/doc/papers/ols-2005/mochel.pdf [6 December 2009].
16. Kroah-Hartman G. *The Linux Kernel Interface: all of your questions answered and then some* . Web pages at: http://www.kernel.org/doc/Documentation/stable_api_nonsense.txt [6 December 2009].
17. Netfilter coreteam. *iptables: free software firewalling tool for Linux*. Web pages at: http://www.iptables.org [6 December 2009].
18. Netfilter coreteam. *Netfilter project: Free software firewalling tools for Linux*. Web pages at: http://www.netfilter.org [6 December 2009].
19. Salim J. *Generic Netlink HOWTO*. Web pages at: http://www.linuxfoundation.org/en/Net:Generic_Netlink_HOWTO [6 December 2009].
20. Neira-Ayuso P. *libmnl: a minimalistic user-space Netlink library*. Web pages at: http://1984.lsi.us.es/projects/libmnl/ [24 March 2010].
21. Graf T. *libnl: an user-space library to simplify work with Netlink sockets*. Web pages at: http://git.kernel.org/?p=libs/netlink/libnl.git [6 December 2009].
22. Dhandapani G., Sundaresan A. *Netlink sockets: An overview* Web pages at: http://qos.ittc.ku.edu/netlink/html/ [6 December 2009].
23. Horman N. *Understanding And Programming With Netlink Sockets*. Web pages at: http://people.redhat.com/nhorman/papers/netlink.pdf [6 December 2009].
24. Kaichuan He K. *Why and How to Use Netlink Socket*. Linux Journal, 2005. Web pages at: http://www.linuxjournal.com/article/7356 [6 December 2009].
25. Salim J., Khosravi H., Kleen A., Kuznetsov A. *RFC 3549 - Linux Netlink as an ip services protocol*. Web pages at: http://www.faqs.org/rfcs/rfc3549.html [6 December 2009].
26. Salim J., Haas R., Blake S. *Netlink2 as ForCES Protocol (Internet-Draft)*, 2004. Web pages at: http://tools.ietf.org/html/draft-jhsrha-forces-netlink2-02 [6 December 2009].
27. Linux man-pages project. *Netlink - Macros*. Web pages at: http://www.kernel.org/doc/man-pages/online/pages/man3/netlink.3.html [6 December 2009].
28. Linux man-pages project. *Netlink - Communication between kernel and userspace*. Web pages at: http://www.kernel.org/doc/man-pages/online/pages/man7/netlink.7.html [6 December 2009].
29. Neira-Ayuso P., M. Gasca R., Lefèvre L. *Netlink source code reference*. Web pages at: http://1984.lsi.us.es/projects/netlink-examples [29 March 2010].