Kernel Korner

Why and How to Use Netlink Socket

Kevin Kaichuan He

Abstract

Use this bidirectional, versatile method to pass data between kernel and user space.

Due to the complexity of developing and maintaining the kernel, only the most essential and performance-critical code are placed in the kernel. Other things, such as GUI, management and control code, typically are programmed as user-space applications. This practice of splitting the implementation of certain features between kernel and user space is quite common in Linux. Now the question is how can kernel code and user-space code communicate with each other?

The answer is the various IPC methods that exist between kernel and user space, such as system call, ioctl, proc filesystem or netlink socket. This article discusses netlink socket and reveals its advantages as a network feature-friendly IPC.

Introduction

Netlink socket is a special IPC used for transferring information between kernel and user-space processes. It provides a full-duplex communication link between the two by way of standard socket APIs for user-space processes and a special kernel API for kernel modules. Netlink socket uses the address family AF_NETLINK, as compared to AF_INET used by TCP/IP socket. Each netlink socket feature defines its own protocol type in the kernel header file include/linux/netlink.h.

The following is a subset of features and their protocol types currently supported by the netlink socket:

- NETLINK_ROUTE: communication channel between user-space routing dæmons, such as BGP, OSPF, RIP and kernel packet forwarding module. User-space routing dæmons update the kernel routing table through this netlink protocol type.
- NETLINK_FIREWALL: receives packets sent by the IPv4 firewall code.
- NETLINK_NFLOG: communication channel for the user-space iptable management tool and kernel-space Netfilter module.
- NETLINK_ARPD: for managing the arp table from user space.

Why do the above features use netlink instead of system calls, ioctls or proc filesystems for communication between user and kernel worlds? It is a nontrivial task to add system calls, ioctls or proc files for new features; we risk polluting the kernel and damaging the stability of the system. Netlink socket is simple, though: only a constant, the protocol type, needs to be added to netlink.h. Then, the kernel module and application can talk using socket-style APIs immediately.

Netlink is asynchronous because, as with any other socket API, it provides a socket queue to smooth the burst

of messages. The system call for sending a netlink message queues the message to the receiver's netlink queue and then invokes the receiver's reception handler. The receiver, within the reception handler's context, can decide whether to process the message immediately or leave the message in the queue and process it later in a different context. Unlike netlink, system calls require synchronous processing. Therefore, if we use a system call to pass a message from user space to the kernel, the kernel scheduling granularity may be affected if the time to process that message is long.

The code implementing a system call in the kernel is linked statically to the kernel in compilation time; thus, it is not appropriate to include system call code in a loadable module, which is the case for most device drivers. With netlink socket, no compilation time dependency exists between the netlink core of Linux kernel and the netlink application living in loadable kernel modules.

Netlink socket supports multicast, which is another benefit over system calls, ioctls and proc. One process can multicast a message to a netlink group address, and any number of other processes can listen to that group address. This provides a near-perfect mechanism for event distribution from kernel to user space.

System call and ioctl are simplex IPCs in the sense that a session for these IPCs can be initiated only by user-space applications. But, what if a kernel module has an urgent message for a user-space application? There is no way of doing that directly using these IPCs. Normally, applications periodically need to poll the kernel to get the state changes, although intensive polling is expensive. Netlink solves this problem gracefully by allowing the kernel to initiate sessions too. We call it the duplex characteristic of the netlink socket.

Finally, netlink socket provides a BSD socket-style API that is well understood by the software development community. Therefore, training costs are less as compared to using the rather cryptic system call APIs and ioctls.

Relating to the BSD Routing Socket

In BSD TCP/IP stack implementation, there is a special socket called the routing socket. It has an address family of AF_ROUTE, a protocol family of PF_ROUTE and a socket type of SOCK_RAW. The routing socket in BSD is used by processes to add or delete routes in the kernel routing table.

In Linux, the equivalent function of the routing socket is provided by the netlink socket protocol type NETLINK_ROUTE. Netlink socket provides a functionality superset of BSD's routing socket.

Netlink Socket APIs

The standard socket APIs—socket(), sendmsg(), recvmsg() and close()—can be used by user-space applications to access netlink socket. Consult the man pages for detailed definitions of these APIs. Here, we discuss how to choose parameters for these APIs only in the context of netlink socket. The APIs should be familiar to anyone who has written an ordinary network application using TCP/IP sockets.

To create a socket with socket(), enter:

```
int socket(int domain, int type, int protocol)
```

The socket domain (address family) is AF_NETLINK, and the type of socket is either SOCK_RAW or SOCK_DGRAM, because netlink is a datagram-oriented service.

The protocol (protocol type) selects for which netlink feature the socket is used. The following are some

predefined netlink protocol types: NETLINK_ROUTE, NETLINK_FIREWALL, NETLINK_ARPD, NETLINK_ROUTE6 and NETLINK_IP6_FW. You also can add your own netlink protocol type easily.

Up to 32 multicast groups can be defined for each netlink protocol type. Each multicast group is represented by a bit mask, 1<<i, where 0<=i<=31. This is extremely useful when a group of processes and the kernel process coordinate to implement the same feature—sending multicast netlink messages can reduce the number of system calls used and alleviate applications from the burden of maintaining the multicast group membership.

bind()

As for a TCP/IP socket, the netlink bind() API associates a local (source) socket address with the opened socket. The netlink address structure is as follows:

```
struct sockaddr_nl
{
  sa_family_t    nl_family;  /* AF_NETLINK   */
  unsigned short nl_pad;     /* zero         */
  __u32          nl_pid;     /* process pid */
  __u32          nl_groups;  /* mcast groups mask */
} nladdr;
```

When used with bind(), the nl_pid field of the sockaddr_nl can be filled with the calling process' own pid. The nl_pid serves here as the local address of this netlink socket. The application is responsible for picking a unique 32-bit integer to fill in nl_pid:

```
NL_PID Formula 1:  nl_pid = getpid();
```

Formula 1 uses the process ID of the application as nl_pid, which is a natural choice if, for the given netlink protocol type, only one netlink socket is needed for the process.

In scenarios where different threads of the same process want to have different netlink sockets opened under the same netlink protocol, Formula 2 can be used to generate the nl_pid:

```
NL_PID Formula 2: pthread_self() << 16 | getpid();
```

In this way, different pthreads of the same process each can have their own netlink socket for the same netlink protocol type. In fact, even within a single pthread it's possible to create multiple netlink sockets for the same protocol type. Developers need to be more creative, however, in generating a unique nl_pid, and we don't consider this to be a normal-use case.

If the application wants to receive netlink messages of the protocol type that are destined for certain multicast groups, the bitmasks of all the interested multicast groups should be ORed together to form the nl_groups field of sockaddr_nl. Otherwise, nl_groups should be zeroed out so the application receives only the unicast netlink message of the protocol type destined for the application. After filling in the nladdr, do the bind as follows:

```
bind(fd, (struct sockaddr*)&nladdr, sizeof(nladdr));
```

Sending a Netlink Message

In order to send a netlink message to the kernel or other user-space processes, another struct sockaddr_nl nladdr needs to be supplied as the destination address, the same as sending a UDP packet with sendmsg(). If the message is destined for the kernel, both nl_pid and nl_groups should be supplied with 0.

If the message is a unicast message destined for another process, the nl_pid is the other process' pid and nl_groups is 0, assuming nlpid Formula 1 is used in the system.

If the message is a multicast message destined for one or multiple multicast groups, the bitmasks of all the destination multicast groups should be ORed together to form the nl_groups field. We then can supply the netlink address to the struct msghdr msg for the sendmsg() API, as follows:

```
struct msghdr msg;
msg.msg_name = (void *)&(nladdr);
msg.msg_namelen = sizeof(nladdr);
```

The netlink socket requires its own message header as well. This is for providing a common ground for netlink messages of all protocol types.

Because the Linux kernel netlink core assumes the existence of the following header in each netlink message, an application must supply this header in each netlink message it sends:

```
struct nlmsghdr
{
  __u32 nlmsg_len;   /* Length of message */
  __u16 nlmsg_type;  /* Message type*/
  __u16 nlmsg_flags; /* Additional flags */
  __u32 nlmsg_seq;   /* Sequence number */
  __u32 nlmsg_pid;   /* Sending process PID */
};
```

nlmsg_len has to be completed with the total length of the netlink message, including the header, and is required by netlink core. nlmsg_type can be used by applications and is an opaque value to netlink core. nlmsg_flags is used to give additional control to a message; it is read and updated by netlink core. nlmsg_seq and nlmsg_pid are used by applications to track the message, and they are opaque to netlink core as well.

A netlink message thus consists of nlmsghdr and the message payload. Once a message has been entered, it enters a buffer pointed to by the nlh pointer. We also can send the message to the struct msghdr msg:

```
struct iovec iov;

iov.iov_base = (void *)nlh;
iov.iov_len = nlh->nlmsg_len;

msg.msg_iov = &iov;
msg.msg_iovlen = 1;
```

After the above steps, a call to sendmsg() kicks out the netlink message:

```
sendmsg(fd, &msg, 0);
```

Receiving Netlink Messages

A receiving application needs to allocate a buffer large enough to hold netlink message headers and message payloads. It then fills the struct msghdr msg as shown below and uses the standard recvmsg() to receive the netlink message, assuming the buffer is pointed to by nlh:

```
struct sockaddr_nl nladdr;
struct msghdr msg;
struct iovec iov;

iov.iov_base = (void *)nlh;
iov.iov_len = MAX_NL_MSG_LEN;
msg.msg_name = (void *)&(nladdr);
msg.msg_namelen = sizeof(nladdr);

msg.msg_iov = &iov;
msg.msg_iovlen = 1;
recvmsg(fd, &msg, 0);
```

After the message has been received correctly, the nlh should point to the header of the just-received netlink message. nladdr should hold the destination address of the received message, which consists of the pid and the multicast groups to which the message is sent. And, the macro NLMSG_DATA(nlh), defined in netlink.h, returns a pointer to the payload of the netlink message. A call to close(fd) closes the netlink socket identified by file descriptor fd.

Kernel-Space Netlink APIs

The kernel-space netlink API is supported by the netlink core in the kernel, net/core/af_netlink.c. From the kernel side, the API is different from the user-space API. The API can be used by kernel modules to access the netlink socket and to communicate with user-space applications. Unless you leverage the existing netlink socket protocol types, you need to add your own protocol type by adding a constant to netlink.h. For example, we can add a netlink protocol type for testing purposes by inserting this line into netlink.h:

```
#define NETLINK_TEST  17
```

Afterward, you can reference the added protocol type anywhere in the Linux kernel.

In user space, we call socket() to create a netlink socket, but in kernel space, we call the following API:

```
struct sock *
netlink_kernel_create(int unit,
           void (*input)(struct sock *sk, int len));
```

The parameter unit is, in fact, the netlink protocol type, such as NETLINK_TEST. The function pointer, input, is a callback function invoked when a message arrives at this netlink socket.

After the kernel has created a netlink socket for protocol NETLINK_TEST, whenever user space sends a netlink message of the NETLINK_TEST protocol type to the kernel, the callback function, input(), which is registered by netlink_kernel_create(), is invoked. The following is an example implementation of the callback function input:

```
void input (struct sock *sk, int len)
{
 struct sk_buff *skb;
 struct nlmsghdr *nlh = NULL;
```

```
 u8 *payload = NULL;

 while ((skb = skb_dequeue(&sk->receive_queue))
       != NULL) {
 /* process netlink message pointed by skb->data */
 nlh = (struct nlmsghdr *)skb->data;
 payload = NLMSG_DATA(nlh);
 /* process netlink message with header pointed by
  * nlh and payload pointed by payload
  */
 }
}
```

This input() function is called in the context of the sendmsg() system call invoked by the sending process. It is okay to process the netlink message inside input() if it's fast. When the processing of netlink message takes a long time, however, we want to keep it out of input() to avoid blocking other system calls from entering the kernel. Instead, we can use a dedicated kernel thread to perform the following steps indefinitely. Use **`skb = skb_recv_datagram(nl_sk)`** where **`nl_sk`** is the netlink socket returned by netlink_kernel_create(). Then, process the netlink message pointed to by skb->data.

This kernel thread sleeps when there is no netlink message in nl_sk. Thus, inside the callback function input(), we need to wake up only the sleeping kernel thread, like this:

```
void input (struct sock *sk, int len)
{
  wake_up_interruptible(sk->sleep);
}
```

This is a more scalable communication model between user space and kernel. It also improves the granularity of context switches.

Sending Netlink Messages from the Kernel

Just as in user space, the source netlink address and destination netlink address need to be set when sending a netlink message. Assuming the socket buffer holding the netlink message to be sent is struct sk_buff *skb, the local address can be set with:

```
NETLINK_CB(skb).groups = local_groups;
NETLINK_CB(skb).pid = 0;   /* from kernel */
```

The destination address can be set like this:

```
NETLINK_CB(skb).dst_groups = dst_groups;
NETLINK_CB(skb).dst_pid = dst_pid;
```

Such information is not stored in skb->data. Rather, it is stored in the netlink control block of the socket buffer, skb.

To send a unicast message, use:

```
int
```

```
netlink_unicast(struct sock *ssk, struct sk_buff
                *skb, u32 pid, int nonblock);
```

where **ssk** is the netlink socket returned by netlink_kernel_create(), **skb->data** points to the netlink message to be sent and **pid** is the receiving application's pid, assuming NLPID Formula 1 is used. **nonblock** indicates whether the API should block when the receiving buffer is unavailable or immediately return a failure.

You also can send a multicast message. The following API delivers a netlink message to both the process specified by pid and the multicast groups specified by group:

```
void
netlink_broadcast(struct sock *ssk, struct sk_buff
          *skb, u32 pid, u32 group, int allocation);
```

group is the ORed bitmasks of all the receiving multicast groups. **allocation** is the kernel memory allocation type. Typically, GFP_ATOMIC is used if from interrupt context; GFP_KERNEL if otherwise. This is due to the fact that the API may need to allocate one or many socket buffers to clone the multicast message.

Closing a Netlink Socket from the Kernel

Given the struct sock *nl_sk returned by netlink_kernel_create(), we can call the following kernel API to close the netlink socket in the kernel:

```
sock_release(nl_sk->socket);
```

So far, we have shown only the bare minimum code framework to illustrate the concept of netlink programming. We now will use our NETLINK_TEST netlink protocol type and assume it already has been added to the kernel header file. The kernel module code listed here contains only the netlink-relevant part, so it should be inserted into a complete kernel module skeleton, which you can find from many other reference sources.

Unicast Communication between Kernel and Application

In this example, a user-space process sends a netlink message to the kernel module, and the kernel module echoes the message back to the sending process. Here is the user-space code:

```
#include <sys/socket.h>
#include <linux/netlink.h>

#define MAX_PAYLOAD 1024  /* maximum payload size*/
struct sockaddr_nl src_addr, dest_addr;
struct nlmsghdr *nlh = NULL;
struct iovec iov;
int sock_fd;

void main() {
 sock_fd = socket(PF_NETLINK, SOCK_RAW,NETLINK_TEST);

 memset(&src_addr, 0, sizeof(src_addr));
 src__addr.nl_family = AF_NETLINK;
```

```
 src_addr.nl_pid = getpid();  /* self pid */
 src_addr.nl_groups = 0;  /* not in mcast groups */
 bind(sock_fd, (struct sockaddr*)&src_addr,
      sizeof(src_addr));

 memset(&dest_addr, 0, sizeof(dest_addr));
 dest_addr.nl_family = AF_NETLINK;
 dest_addr.nl_pid = 0;   /* For Linux Kernel */
 dest_addr.nl_groups = 0; /* unicast */

 nlh=(struct nlmsghdr *)malloc(
                         NLMSG_SPACE(MAX_PAYLOAD));
 /* Fill the netlink message header */
 nlh->nlmsg_len = NLMSG_SPACE(MAX_PAYLOAD);
 nlh->nlmsg_pid = getpid();  /* self pid */
 nlh->nlmsg_flags = 0;
 /* Fill in the netlink message payload */
 strcpy(NLMSG_DATA(nlh), "Hello you!");

 iov.iov_base = (void *)nlh;
 iov.iov_len = nlh->nlmsg_len;
 msg.msg_name = (void *)&dest_addr;
 msg.msg_namelen = sizeof(dest_addr);
 msg.msg_iov = &iov;
 msg.msg_iovlen = 1;

 sendmsg(fd, &msg, 0);

 /* Read message from kernel */
 memset(nlh, 0, NLMSG_SPACE(MAX_PAYLOAD));
 recvmsg(fd, &msg, 0);
 printf(" Received message payload: %s\n",
        NLMSG_DATA(nlh));

 /* Close Netlink Socket */
 close(sock_fd);
}
```

And, here is the kernel code:

```
struct sock *nl_sk = NULL;

void nl_data_ready (struct sock *sk, int len)
{
  wake_up_interruptible(sk->sleep);
}

void netlink_test() {
 struct sk_buff *skb = NULL;
 struct nlmsghdr *nlh = NULL;
 int err;
 u32 pid;

 nl_sk = netlink_kernel_create(NETLINK_TEST,
                               nl_data_ready);
 /* wait for message coming down from user-space */
 skb = skb_recv_datagram(nl_sk, 0, 0, &err);

 nlh = (struct nlmsghdr *)skb->data;
 printk("%s: received netlink message payload:%s\n",
        __FUNCTION__, NLMSG_DATA(nlh));

 pid = nlh->nlmsg_pid; /*pid of sending process */
 NETLINK_CB(skb).groups = 0; /* not in mcast group */
 NETLINK_CB(skb).pid = 0;      /* from kernel */
 NETLINK_CB(skb).dst_pid = pid;
```

```
 NETLINK_CB(skb).dst_groups = 0;  /* unicast */
 netlink_unicast(nl_sk, skb, pid, MSG_DONTWAIT);
 sock_release(nl_sk->socket);
}
```

After loading the kernel module that executes the kernel code above, when we run the user-space executable, we should see the following dumped from the user-space program:

```
Received message payload: Hello you!
```

And, the following message should appear in the output of dmesg:

```
netlink_test: received netlink message payload:
Hello you!
```

Multicast Communication between Kernel and Applications

In this example, two user-space applications are listening to the same netlink multicast group. The kernel module pops up a message through netlink socket to the multicast group, and all the applications receive it. Here is the user-space code:

```
#include <sys/socket.h>
#include <linux/netlink.h>

#define MAX_PAYLOAD 1024  /* maximum payload size*/
struct sockaddr_nl src_addr, dest_addr;
struct nlmsghdr *nlh = NULL;
struct iovec iov;
int sock_fd;

void main() {
 sock_fd=socket(PF_NETLINK, SOCK_RAW, NETLINK_TEST);

 memset(&src_addr, 0, sizeof(local_addr));
 src_addr.nl_family = AF_NETLINK;
 src_addr.nl_pid = getpid();  /* self pid */
 /* interested in group 1<<0 */
 src_addr.nl_groups = 1;
 bind(sock_fd, (struct sockaddr*)&src_addr,
      sizeof(src_addr));

 memset(&dest_addr, 0, sizeof(dest_addr));

 nlh = (struct nlmsghdr *)malloc(
                           NLMSG_SPACE(MAX_PAYLOAD));
 memset(nlh, 0, NLMSG_SPACE(MAX_PAYLOAD));

 iov.iov_base = (void *)nlh;
 iov.iov_len = NLMSG_SPACE(MAX_PAYLOAD);
 msg.msg_name = (void *)&dest_addr;
 msg.msg_namelen = sizeof(dest_addr);
 msg.msg_iov = &iov;
 msg.msg_iovlen = 1;

 printf("Waiting for message from kernel\n");

 /* Read message from kernel */
 recvmsg(fd, &msg, 0);
 printf(" Received message payload: %s\n",
        NLMSG_DATA(nlh));
 close(sock_fd);
}
```

And, here is the kernel code:

```
#define MAX_PAYLOAD 1024
struct sock *nl_sk = NULL;

void netlink_test() {
 sturct sk_buff *skb = NULL;
 struct nlmsghdr *nlh;
 int err;

 nl_sk = netlink_kernel_create(NETLINK_TEST,
                               nl_data_ready);
 skb=alloc_skb(NLMSG_SPACE(MAX_PAYLOAD),GFP_KERNEL);
 nlh = (struct nlmsghdr *)skb->data;
 nlh->nlmsg_len = NLMSG_SPACE(MAX_PAYLOAD);
 nlh->nlmsg_pid = 0;  /* from kernel */
 nlh->nlmsg_flags = 0;
 strcpy(NLMSG_DATA(nlh), "Greeting from kernel!");
 /* sender is in group 1<<0 */
 NETLINK_CB(skb).groups = 1;
 NETLINK_CB(skb).pid = 0;  /* from kernel */
 NETLINK_CB(skb).dst_pid = 0;  /* multicast */
 /* to mcast group 1<<0 */
 NETLINK_CB(skb).dst_groups = 1;

 /*multicast the message to all listening processes*/
 netlink_broadcast(nl_sk, skb, 0, 1, GFP_KERNEL);
 sock_release(nl_sk->socket);
}
```

Assuming the user-space code is compiled into the executable nl_recv, we can run two instances of nl_recv:

```
./nl_recv &
Waiting for message from kernel
./nl_recv &
Waiting for message from kernel
```

Then, after we load the kernel module that executes the kernel-space code, both instances of nl_recv should receive the following message:

```
Received message payload: Greeting from kernel!
Received message payload: Greeting from kernel!
```

Conclusion

Netlink socket is a flexible interface for communication between user-space applications and kernel modules. It provides an easy-to-use socket API to both applications and the kernel. It provides advanced communication features, such as full-duplex, buffered I/O, multicast and asynchronous communication, which are absent in other kernel/user-space IPCs.