Send an arbitrary IPv4 datagram using a raw socket in C

Content

Tested on

Debian (Lenny)
Ubuntu (Lucid)

Objective

To send an arbitrary IPv4 datagram using a raw socket in C

Background

Most programs that communicate using the Internet Protocol do so through a transport-layer protocol such as TCP or UDP and have no need to deal directly with Internet Protocol datagrams, but there are some circumstances where it is necessary to interact with the network stack at a lower level. These include:

- implementation of transport-layer protocols that are not built in to the network stack, and
- production of malformed or otherwise non-standard datagrams for testing purposes.

Scenario

Suppose that you wish to send an ICMP echo request to a given IPv4 address. (This is what the `ping` command does to determine whether there is a reachable host at that address.)

There is no POSIX API call that provides this functionality *per se*. You therefore intend to assemble an ICMP message with the required content then send it as the payload of an IP datagram using a raw socket.

Method

Overview

The method described here has five steps:

1. Select the required protocol number.
2. Create the raw socket.
3. Optionally, set the `IP_HDRINCL` socket option.
4. Construct the datagram.
5. Send the datagram.

The following header files will be needed:

```
#include <errno.h>
#include <unistd.h>
#include <netdb.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

Rate this page

Note that POSIX-compatible operating systems are not obliged to support raw sockets at all, and the API that has been fully standardised is quite restrictive. For this reason it is often necessary for programs that use raw sockets to stray into the realm of implementation-defined behaviour. They are also likely to require elevated privileges in order to run.

Select the required protocol number

All IPv4 traffic is labelled with a protocol number to distinguish between the various transport-layer protocols (such as TCP and UDP) that IPv4 can carry. You will need this number:

- when opening the raw socket (unless you choose `IPPROTO_RAW` for the protocol number on a system that interprets this as a wildcard), and/or
- when constructing the IP datagram header (if you choose to do this yourself instead of allowing it to be added automatically).

There are several sources from which protocol numbers can be obtained:

- Some protocol numbers are defined as constants by the API. POSIX defines `IPPROTO_TCP`, `IPPROTO_UDP` and `IPPROTO_ICMP`, and glibc defines many more.
- Protocol numbers can be looked up at run time by calling the function `getprotobyname`.
- IANA maintains a list of assigned protocol numbers.

Unlike a TCP or UDP port number there is little risk of an assigned IP protocol number ever needing to change, especially for a widely-used protocol such as ICMP. For this reason there is no real need to look up the protocol number at runtime, and it is quite reasonable for the required value to be hard-coded.

For this particular example there is a symbolic constant, `IPPROTO_ICMP`, that all POSIX-compatible operating systems are supposed to provide. The simplest solution would be to use that. If you instead want to call `getprotobyname` then this can be done as follows:

```
const char* protocol_name="icmp";
struct protoent* protocol=getprotobyname(protocol_name);
if (!protocol) {
    die("Protocol %s not found",protocol_name);
}
int protocol_number=protocol->p_proto;
```

Note that `getprotobyname` is not thread-safe. In a multi-threaded program it would be advisable to look up any required protocol numbers at the outset if this is practicable.

Create the raw socket

The socket that will be used to send the IP datagram should be created using the `socket` function. This takes three arguments:

1. the domain (`AF_INET` in this case, meaning IPv4),
2. the socket type (`SOCK_RAW` in this case, meaning that the socket should provide direct access to the network layer without any transport-layer protocol), and
3. the protocol (normally corresponding to the `protocol` field in the Internet Protocol header).

An alternative to specifying the protocol number as the third argument is to use the value `IPPROTO_RAW`. POSIX does not generally allow this, but some implementations use it as a wildcard or a dummy value. (In the case of Linux it allows any protocol to be sent (with headers) but nothing can be received.)

In this instance the socket will be used for sending ICMP messages, therefore the third argument should be set to `IPPROTO_ICMP`:

```
int fd=socket(AF_INET,SOCK_RAW,IPPROTO_ICMP);
if (fd==-1) {
    die("%s",strerror(errno));
}
```

Optionally, set the IP_HDRINCL socket option

POSIX does not specify the format in which a datagram should be written to a raw socket, however the following behaviour is typical:

- By default the header is generated automatically, therefore only the payload should be written.
- If the `IP_HDRINCL` socket option is set then the header should be constructed by the caller and both header and payload written to the socket.

The protocol level for `IP_HDRINCL` is `IPPROTO_IP`. The parameter is a boolean value that is usually represented by an `int`. It should be set to zero to disable header inclusion or non-zero to enable it:

```
int hdrincl=1;
if (setsockopt(fd,IPPROTO_IP,IP_HDRINCL,&hdrincl,sizeof(hdrincl))==-1) {
    die("%s",strerror(errno));
}
```

Support for `IP_HDRINCL` is quite common, but the details vary as to:

- the byte order that should be used for each of the header fields (which is not necessarily the same for all fields), and
- which fields (if any) are filled in automatically.

Some operating systems set `IP_HDRINCL` implicitly when `IPPROTO_RAW` is selected (on the grounds that it would make little sense not to supply a header in that case) but others require an explicit call to `setsockopt`. If you want to enable header inclusion then it is prudent to set it regardless, in order to accommodate either behaviour.

Send the datagram

Raw datagrams can in principle be sent using any function that is capable of writing to a file descriptor, however it is often necessary to use either `sendto` or `sendmsg` so that a destination address can be specified. There are two possible reasons for this:

- If the header will be constructed automatically then the network stack needs to know what the destination address field should be set to.
- You may want to route the datagram towards an address that differs from the one specified in the IP header.

Of `sendto` and `sendmsg` the latter is the more flexible option, but at the cost of a signficiantly more complex interface. Details for each function are given below.

Regardless of which function you choose, each function call will result in a separate datagram being sent. For this reason you must either compose each datagram payload as a single, contiguous block of memory, or make use of the scatter/gather capability provided by `sendmsg`.

In this particular example the payload to be sent is an ICMP echo request, which can be constructed as follows:

```
const size_t req_size=8;
struct icmphdr req;
req.type=8;
req.code=0;
req.checksum=0;
req.un.echo.id=htons(rand());
req.un.echo.sequence=htons(1);
req.checksum=ip_checksum(&req,req_size);
```

This makes use of the `icmphdr` structure provided by glibc and the `ip_checksum` function described in the microHOWTO ‘Calculate an Internet Protocol checksum in C’. Note that `sizeof(req)` cannot be used to obtain the size of the payload because `struct icmphdr` is not specific to echo requests, so the constant `req_size` has been defined for this purpose.

Send the datagram (using sendto)

To call `sendto` you must supply the content of the datagram and the remote address to which it should be sent:

```
if (sendto(fd,&req,req_size,0,
    res->ai_addr,res->ai_addrlen)==-1) {
    die("%s",strerror(errno));
}
```

The fourth argument is for specifying flags which modify the behaviour of `sendto`, none of which are needed in this example.

The value returned by `sendto` is the number of bytes sent, or `-1` if there was an error. Raw datagrams are sent atomically, so unlike when writing to a TCP socket there is no need to wrap the function call in a loop to handle partially-sent data.

Send the datagram (using sendmsg)

To call `sendmsg`, in addition to the datagram content and remote address you must also construct an `iovec` array and a `msghdr` structure:

```
struct iovec iov[1];
iov[0].iov_base=&req;
iov[0].iov_len=req_size;

struct msghdr message;
message.msg_name=res->ai_addr;
message.msg_namelen=res->ai_addrlen;
message.msg_iov=iov;
message.msg_iovlen=1;
message.msg_control=0;
message.msg_controllen=0;

if (sendmsg(fd,&message,0)==-1) {
    die("%s",strerror(errno));
}
```

The purpose of the `iovec` array is to provide a scatter/gather capability so that the datagram payload need not be stored in a contiguous region of memory. In this example the entire payload is stored in a single buffer, therefore only one array element is needed.

The msghdr structure exists to bring the number of arguments to recvmsg and sendmsg down to a managable number. On entry to sendmsg it specifies where the destination address, the datagram payload and any ancillary data are stored. In this example no ancillary data has been provided.

If you wish to pass any flags into sendmsg then this cannot be done using msg_flags, which is ignored on entry. Instead you must pass them using the third argument to sendmsg (which is zero in this example).

Variations

Sending to the IPv4 broadcast address

By default, attempts to send a datagram to the broadcast address are rejected with an error (typically EACCES, however it is not obvious from the POSIX specification which error should occur). This is a safety measure intended to reduce the risk of making unintended broadcasts. It can be overridden by setting the SO_BROADCAST socket option:

```
int broadcast=1;
if (setsockopt(fd,SOL_SOCKET,SO_BROADCAST,
    &broadcast,sizeof(broadcast))==-1) {
    die("%s",strerror(errno));
}
```

Alternatives

Sending at the link layer

See: Send an arbitrary Ethernet frame using libpcap
Send an arbitrary Ethernet frame using an AF_PACKET socket in C

Raw sockets of the type described above operate at the network layer. An alternative would be to inject packets at the link layer, for example in the form of Ethernet frames. This can be done using libpcap or (on Linux-based systems) using an AF_PACKET socket.

This approach makes it possible to implement any network-layer protocol, whether or not it is explicitly supported by the network stack, but also brings a number of disadvantages which result from operating at a lower level of abstraction:

- The sender must construct the network layer header, and depending on the method of injection, perhaps also the link layer header.
- The sender must take responsibility for routing and link-layer address resolution (although it may be possible to delegate these tasks back to the operating system rather than implementing them from scratch).
- The above cannot normally be done without knowledge of the link layer protocol, which will typically need to be coded into the sending program on a case-by-case basis.

For these reasons, use of a raw socket is recommended unless you specifically need the extra functionality provided by working at the link layer.

See also

- Send a UDP datagram in C
- Establish a TCP connection in C

Further reading

- raw(7) (Linux manpage)
- The Open Group, sendto, Base Specifications Issue 6
- The Open Group, sendmsg, Base Specifications Issue 6
- ithilgore, SOCK_RAW Demystified, May 2008

Tags: c | posix | socket