

LINUX SOCKET PART 15

Advanced TCP/IP - The TCP/IP Protocols & RAW Socket

snapdeal.com
Bachatey raho!
SAVINGS DAY
FLAT
₹4500 OFF
on Electronics
SHOP >

Menu

[Network Story 1](#)
[Network Story 2](#)
[Network Story 3](#)
[Network Story 4](#)
[Network Story 5](#)
[Network Story 6](#)
[Socket Example 1](#)
[Socket Example 2](#)
[Socket Example 3](#)
[Socket Example 4](#)
[Socket Example 5](#)
[Socket Example 6](#)
[Socket Example 7](#)
[Advanced TCP/IP 1](#)
[Advanced TCP/IP 2](#)
[Advanced TCP/IP 3](#)
[Advanced TCP/IP 4](#)
[Advanced TCP/IP 5](#)

My Training Period: hours

This is a continuation from Part III series, [TCP & UDP Client-server program examples](#). Working program examples if any compiled using [gcc](#), tested using the public IPs, run on **Linux / Fedora Core 3**, with several times of update, as root or SUID 0. The Fedora machine used for the testing having the "No Stack Execute" disabled and the SELinux set to default configuration. This Module will concentrate on the TCP/IP stack and will try to dig deeper till the packet level.

Abilities that should be acquired for this session:

- Able to understand the 7 layers OSI stack.
- Able to understand the 4 layers TCP/IP stack/suite/layer.
- Able to understand protocols in TCP/IP stack.
- Able to find and appreciate the RFCs and Standards.
- Able to understand and use the RAW socket (vs cooked socket).
- Able to understand and use for good purposes of the useful network tools that can be developed using RAW socket.

(Host-to-Host) Transport Layer

The Transport layer has two major jobs:

1. It must subdivide user-sized data buffers into network layer sized datagrams, and
2. It must enforce any desired transmission control such as reliable delivery.

The Transport layer is responsible for end-to-end data integrity. The two most important protocols in this layer are **Transmission Control Protocol (TCP)** and **User Datagram Protocol (UDP)**. TCP provides reliable data delivery service with end-to-end error detection and correction and also enables hosts to maintain multiple, simultaneous connections. UDP provides low-overhead, connectionless datagram delivery service. Both protocols deliver data between the Application layer and the Internet layer. Applications programmers can choose whichever service is more appropriate for their specific applications. Protocols defined at this layer accept data from application protocols running at the Application layer, encapsulate it in the protocol header, and deliver the data segment thus formed to the lower IP layer for routing. Unlike the IP protocol, the transport layer is aware of the identity of the ultimate user representative process. As such, the Transport layer, in the TCP/IP suite, embodies what data communications are all about: The delivering of information from an application on one computer to an application on another computer.

User Datagram Protocol (UDP – RFC768)

Gives application programs direct access to a datagram delivery service, like the delivery service that IP provides. This allows applications to exchange messages over the network with a minimum of protocol overhead. UDP is an unreliable (it doesn't care about the quality if deliveries it make), connectionless (doesn't establish a connection on behalf of user applications) datagram protocol. Within your computer, UDP will deliver data correctly. UDP is used as a data transport service when the amount of data being transmitted is small, the overhead of creating connections and ensuring reliable delivery may be greater than the work of retransmitting the entire data set. Broadcast-oriented services use UDP, as do those in which repeated, out of sequence, or missed requests have no harmful side effects. Since no state is maintained for UDP transmission, it is ideal for repeated, short operations such as the **Remote Procedure Call (RPC)** protocol. UDP packets can arrive in any order. If there is a network bottleneck that drops packets, UDP packets may not arrive at all. It's up to the application built on UDP to determine that a packet was lost, and to resend it if necessary.

NFS and **NIS** are built on top of UDP because of its speed and statelessness. While the performance advantages of a fast protocol are obvious, the stateless nature of UDP is equally important. Without state information in either the client or server, crash recovery is greatly simplified.



UDP is also the transport protocol for several well-known application-layer protocols, including **Network File System (NFS)**, **Simple Network Management Protocol (SNMP)**, **Domain Name System (DNS)**, and **Trivial File Transfer Protocol (TFTP)**. The following figure shows the UDP datagram format.

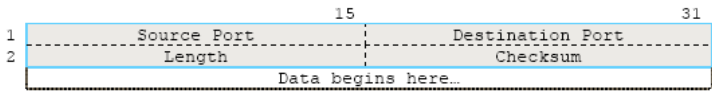


Figure 11: The UDP Datagram Format.

A brief description:

| Field | Description |
|----------------------------|---|
| Source Port (16 bits) | This field is optional and specifies the port number of the application that is originating the user data. |
| Destination Port (16 bits) | This is the port number pertaining to the destination application. |
| Length (16 bits) | This field describes the total length of the UDP datagram, including both data and header information. |
| UDP checksum (16 bits) | Integrity checking is optional under UDP. If turned on, this field is used by both ends of the communication channel for data integrity checks. |

Table 4: UDP fields description.

Well, let revised what we have already covered till now. The following figure is the TCP/IP stack mentioned before. When data is sent from a host to another host, depend on the application (protocols), it has to go through the layers. Every layer will encapsulate the appropriate header.

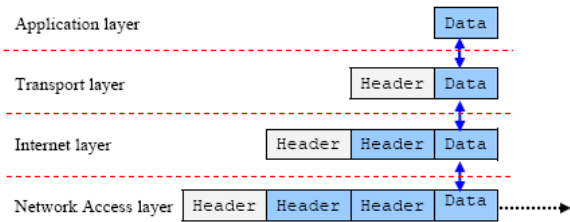


Figure 12: TCP/IP header encapsulation, illustrated vertically.

To make it clearer, the following figure is a packet that horizontally rearranged of the previous figure. The Data... may also contain other upper protocol header, the Transport layer.

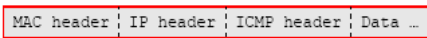


Figure 13: TCP/IP header encapsulation, illustrated horizontally.

As an example, by assuming there is no other information inserted between the Transport and the Internetwork layers, the following figure shows the packet when the data has gone through the Transport and Internetwork layers.

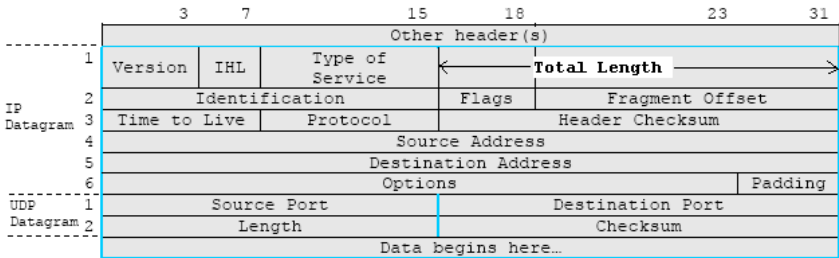


Figure 14: The UDP and IP headers in a packet.

From the above figure, what IP considers to be data field is in fact just another piece of formatted information including both UDP header and user protocol data. To IP it should not matter what the data field is hiding. The details of the header information for each protocol should clearly convey to the reader purpose of the protocol. Keep in mind that at machine level, all the fields in the packet actually just a combination of the 0s and 1s digits. Let continue with another important protocol in Transport layer, the TCP.

Transmission Control Protocol (TCP)

Transmission Control Protocol (TCP) is a required TCP/IP standard defined in RFC 793, "Transmission Control Protocol (TCP)", that provides a reliable, connection-oriented packet delivery service. The Transmission Control Protocol:

1. Guarantees delivery of IP datagrams.
2. Performs segmentation and reassembly of large blocks of data sent by programs.
3. Ensures proper sequencing and ordered delivery of segmented data.
4. Performs checks on the integrity of transmitted data by using checksum calculations.
5. Sends positive messages depending on whether data was received successfully. By using selective acknowledgments, negative acknowledgments for data not received are also sent.
6. Offers a preferred method of transport for programs that must use reliable session-based data transmission, such as client/server database and e-mail programs.

It is fully reliable, connection-oriented, end-to-end packet delivery, acknowledged, byte stream protocol that provide consistency for data delivery across the network in a proper sequence. TCP supports data fragmentation and reassembly. It also support multiplexing/demultiplexing using source and destination port numbers in much the same way they are used by UDP. Together with the Internet Protocol (IP), TCP represents the heart of the Internet protocols. TCP provides reliability with a mechanism called **Positive Acknowledgement with Retransmission (PAR)**. Simply said, a system using PAR resends the data, unless it hears from the remote system that the data received is okay. The unit of data exchanged between co-operating TCP modules is called a **segment**. The following is a TCP segment format.

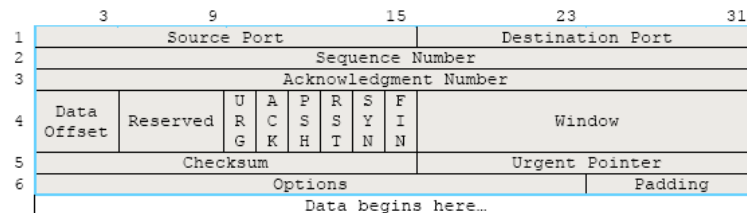


Figure 15: The segment format of the TCP Protocol.

A brief field description:

| Field | Description |
|----------------------------------|---|
| Source port (16 bits) | Specifies the port on the sending TCP module. |
| Destination port (16 bits) | Specifies the port on the receiving TCP module. |
| Sequence number (32 bits) | Specifies the sequence position of the first data octet in the segment. When the segment opens a connection, the sequence number is the Initial Sequence Number (ISN) and the first octet in the data field is at sequence ISN+1. |
| Acknowledgement number (32 bits) | Specifies the next sequence number that is expected by the sender of the segment. TCP indicates that this field is active by setting the ACK bit, which is always set after a connection is established. |
| Data offset 4 bits) | Specifies the number of 32-bit word in the TCP header. |
| Reserved (6 bits) | Must be zero. Reserved for future use. |
| Control bits (6 bits) | <p>The six control bits are as follow:</p> <ol style="list-style-type: none"> 1. URG - When set, the Urgent Pointer field is significant. 2. ACK - When set, the acknowledgement Number field is significant. 3. PSH - Initiates a push function. 4. RST - Forces a reset of the connection. 5. SYN - Synchronizes sequencing counters for the connection. This bit is set when a segment request opening of a connection. 6. FIN - No more data. Closes the connection. |
| Window (16 bits) | Specifies the number of octets, starting with the octet specified in the acknowledgement number field, which the sender of the segment can currently accept. |
| Checksum (16 bits) | An error control checksum that covers the header and data fields. It does not cover any padding required to have the segment consists of an even number of octets. The checksum also covers a 96-pseudoheader as shown below; it includes source and destination addresses, the protocol, and the segment length. The information is forwarded with the segment to IP to protect TCP from miss-routed segments. The value of the segment length a field includes the TCP header and data, but doesn't include the length of the pseudoheader. |
| Urgent Pointer (16 bits) | Identifies the sequence number of the octet following urgent data. The urgent pointer is a positive offset from the sequence number of the segment. |
| Options (variable) | Options are available for a variety of functions. |
| Padding (variable) | 0-value octets are appended to the header to ensure that the header ends on a 32-bit word boundary. |

Table 5: TCP segment fields description.

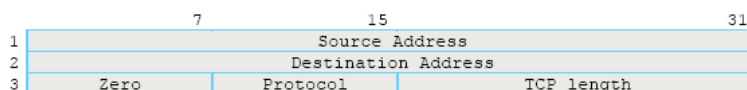


Figure 16: The format of the TCP pseudoheader.

TCP Three-Way Handshake

Each segment contains a checksum that the recipient uses to verify that the data is undamaged. If the data segment is received undamaged, the receiver sends a positive acknowledgement back to the sender. If the data segment is damaged, the receiver discards it. After an appropriate time-out period, the sending TCP module retransmits any segment for which no positive acknowledgement has been received. TCP is connection-oriented. It establishes a logical end-to-end connection between the two communication hosts. Control information, called a **handshake**, is exchanged between the two endpoints to establish a dialogue before data is transmitted. TCP indicates the control function of a segment by setting the appropriate bit in the flags field of the segment header. The type of handshake used by TCP is called a **three-way handshake** because three segments are exchanged. The following figure illustrates the three-way handshake mechanism.

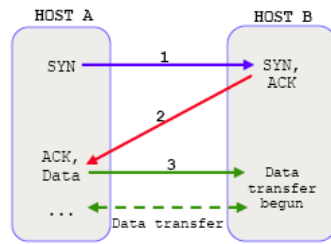


Figure 17: A Three-Way Handshake of the TCP initialization.

The client who needs to initialize a connection sends out a SYN segment (Synchronize) to the server along with the initial sequence number. No data is sent during this process, and the SYN segment contains only TCP header and IP header. When the server receives the SYN segment, it acknowledges the request with its own SYN segment, called SYN-ACK segment. When the client receives the SYN-ACK, it sends an ACK for the server's SYN. At this stage the connection is "established."

Unlike TCP connection initialization, which is a three-way process, connection termination takes place with the exchange of four-way packets. The following figure illustrates the TCP termination process.

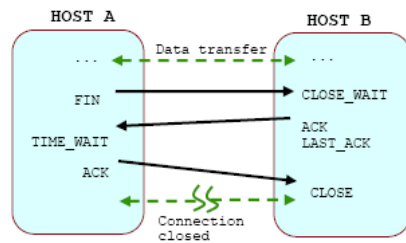


Figure 18: A Four-Way of the TCP termination.



1. The client who needs to terminate the connection sends a FIN segment to the server that is a TCP Packet with the FIN flag set, indicating that it has finished sending the data.
2. The server, upon receiving the FIN segment, does not terminate the connection but enters into a "passive close" (CLOSE_WAIT) state and sends an ACK for the FIN back to the client with the sequence number incremented by one. Now the server enters into LAST_ACK state.
3. When the client gets the last ACK from the server, it enters into a TIME_WAIT state, and sends an ACK back to the server with the sequence number incremented by one.
4. When the server gets the ACK from the client, it closes the connection.

Reliability and Acknowledgement

TCP employs the positive acknowledgement with retransmission technique for the purpose of achieving reliability in service.

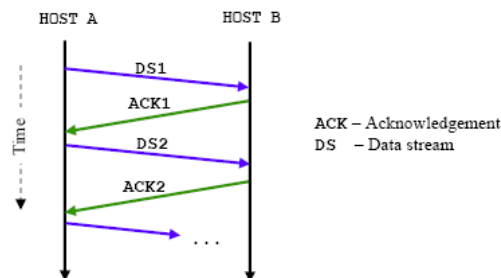


Figure 19: The positive acknowledgement with retransmission technique.

Figure 19 illustrates a simple ladder diagram depicting the events taking place between two hosts. The arrows represent transmitted data and/or acknowledgements, and time is represented by the vertical distance down the ladder. When TCP send a data segment, it requires an acknowledgement from the receiving end. The acknowledgement is used to update the connection state table. An acknowledgement can be positive or negative. A positive acknowledgement implies that the receiving host recovered the data and that it passed the integrity check. A negative acknowledgement implies that the failed data segment needs to be retransmitted. It can be caused by failures such as data corruption or loss.

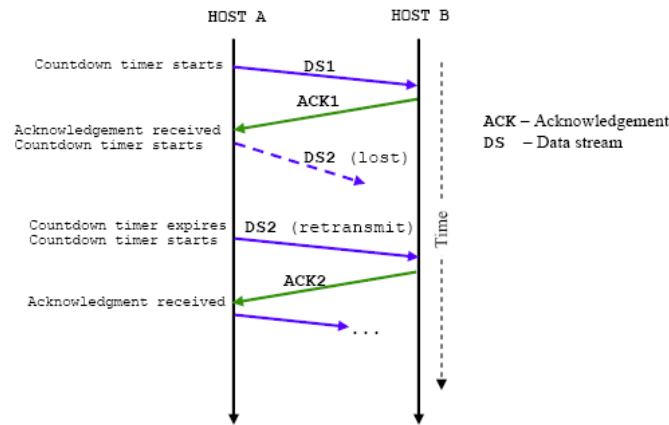


Figure 20: TCP implementation of the time-out mechanism to keep track of loss segments.

In figure 20, illustrates what happens when a packet is lost on the network and fails to reach its ultimate destination. When a host sends data, it starts a countdown timer. If the timer expires without receiving an acknowledgement, this host assumes that the data segment was lost. Consequently, this host retransmits a duplicate of the failing segment. TCP keep a copy of all transmitted data with outstanding positive acknowledgement. Only after receiving the positive acknowledgement is this copy discarded to make room for other data in its buffer.

Data Stream Maintenance

The interface between TCP and a local process is a port, which is a mechanism that enables the process to call TCP and in turn enables TCP to deliver data streams to the appropriate process. Ports are identified by port numbers. To fully specify a connection, the host IP address is appended to the port number. This combination of IP address and port number is called a **socket**. A given socket number is unique on the internetwork. A connection between two hosts is fully described by the sockets assigned to each end of the connection.

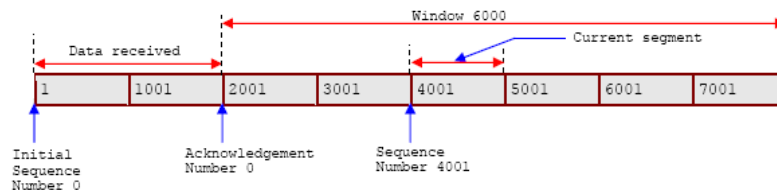


Figure 21: A TCP data stream that starts with an Initial Sequence Number (ISN) of 0.

In figure 21, the receiving system has received and acknowledged 2000 bytes. So the current Acknowledgement Number is 2000. The receiver also has enough buffer space for another 6400 bytes, so it has advertised a Window of 6000. The sender is currently sending a segment of 1000 bytes starting with Sequence Number 4001. The sender has received no acknowledgement for the bytes from 2001 on, but continues sending data as long as it is within the window. If the sender fills the window and receives no acknowledgement of the data previously sent, it will, after an appropriate time-out, resend the data starting from the first unacknowledged byte. Retransmission would start from byte 2001 if no further acknowledgements are received. This procedure ensures that data is reliably received at the far end of the network.

From the perspective of Applications, communication with the network involves sending and receiving continuous streams of data. It seems that the Application is not responsible for fragmenting the data to fit lower-layer protocols. The whole process can be illustrated in the following figure.

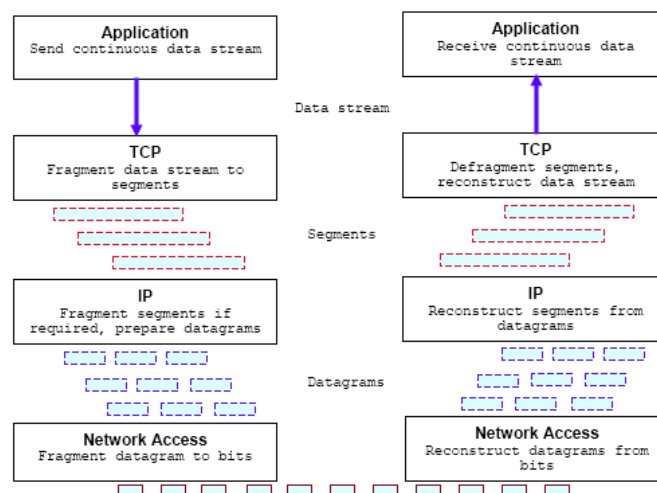


Figure 22: How data is processed as they travel down the protocol stack, through the network, and up the protocol stack of the receiver.

Brief description:

1. TCP receives a stream of data from the upper-layer process.
2. TCP may fragment the data stream into segments that meet the maximum datagram size of IP.
3. IP may fragment segments as it prepares datagrams that are sized to conform to restrictions of the network types: Ethernet, Token Ring etc.
4. Network protocols transmit the datagram in the form of bits.

5. Network protocols at the receiving host reconstruct datagrams from the bits they receive.
6. IP receives datagrams from the network. Where necessary datagram fragments are reassembled to reconstruct the original segment.
7. TCP presents data in segments to upper-layer protocols in the form of data streams.

Application Layer

The **Application** layer includes all processes that use the transport layer protocols to deliver data. There are many applications protocols. A good example of concerns handled by these processes is the reconciliation of differences in the data syntax between the platforms on which the applications are running. It should be clear that unless this difference in data representation is handled properly, any exchange of data involving these processes is likely to yield erroneous interpretations of numerical data. To resolve this issue, and other similar issues, TCP/IP defines the **eXternal Data Representation (XDR)** protocol. Reflecting on the nature of this problem, you can easily see that the problem has nothing to do with the underlying network topology, wiring, or electrical interference.

Application examples that use TCP:

1. TELNET: The Network Terminal Protocol provides remote login over the network.
2. FTP: The File Transfer Protocol is used for interactive file transfer between hosts.
3. SMTP: The Simple Mail Transfer Protocol acts as Mail Transfer Agent (MTA) that delivers electronic mail.

Application examples that use UDP:

1. SNMP: The Simple Network Management Protocol is used to collect management information from network devices.
2. DNS : Domain Name Service, maps IP addresses to the names assigned to network devices.
3. RIP: Routing Information Protocol, routing is the central to the way TCP/IP networks. RIP is used by the network devices to exchange routing information.
4. NFS : Network File System, this protocol allows files to be shared by various hosts on the network as if they were local drives.

RAW vs Cooked Socket

Intro

In this section and that follows, we will learn the basics of using raw sockets. Here, we will try to construct our own packet and insert any IP protocol based datagram into the network traffic. This is useful, for example, to build raw socket scanners like **nmap**, to **spoof** or to perform operations that need to send out raw sockets. Basically, you can send any packet at any time, whereas using the interface functions for your systems IP-stack (connect(), write(), bind(), etc.) as discussed in the previous Modules but you don't have direct control over the packets. This theoretically enables you to simulate the behavior of your OS's IP stack, and also to send stateless traffic (datagrams that don't belong to any valid connection).

The usage of the raw socket is to send a single packet at one time, with all the protocol headers filled in by the program (instead of the kernel). As discussed in the previous Modules, when you create a socket and bind it to a process/port, you don't care about IP or TCP header fields as long as you are able to communicate with the server. The kernel or the underlying operating system builds the packet including the checksum for your data. Thus, network programming was so easy with the traditional **cooked** sockets. Contrarily, raw sockets let you fabricate the header fields including information like source IP address etc. The following is a socket() prototype.

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

If you check the man page for socket(), the socket types defined for the **type** parameter includes:

| Type | Description |
|----------------|--|
| SOCK_STREAM | Provides sequenced, reliable, two-way, connection-based byte streams. An out-of-band data transmission mechanism may be supported. |
| SOCK_DGRAM | Supports datagrams (connectionless, unreliable messages of a fixed maximum length). |
| SOCK_SEQPACKET | Provides a sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length; a consumer is required to read an entire packet with each read system call. |
| SOCK_RAW | Provides raw network protocol access. |
| SOCK_RDM | Provides a reliable datagram layer that does not guarantee ordering. |
| SOCK_PACKET | Obsolete and should not be used in new programs. Use packet (check man page for packet) instead. |

Table 6: Socket types of socket().

There are two methods of receiving packets from the datalink layer under Linux. The original method, which is more widely available but less flexible and obsolete, is to create a socket of type SOCK_PACKET. The newer method, which introduces more filtering and performance features, is to create a socket of family PF_PACKET. To do either, we must have sufficient privileges (similar to creating a raw socket), and the third argument to socket must be a nonzero value specifying the **Ethernet** (may use other frame type such as Token Ring etc.) frame type. When using PF_PACKET sockets, the second argument to socket can be SOCK_DGRAM, for "cooked" packets with the link-layer header removed, or SOCK_RAW, for the complete link-layer packet. SOCK_PACKET sockets only return the complete link layer packet. For example, to receive all frames from the datalink, we may write:

```
/* newer systems */
fd = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
```

Or

```
/* older systems */
fd = socket(AF_INET, SOCK_PACKET, htons(ETH_P_ALL));
```

This would return frames for all protocols that the datalink receives. If we want only IPv4 frames, the call would be:

```
/* newer systems */
fd = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_IP));
```

Or

```
/* older systems */
fd = socket(AF_INET, SOCK_PACKET, htons(ETH_P_IP));
```

Other constants for the final argument are `ETH_P_ARP` and `ETH_P_IPV6`, for example.

And for the `domain` parameter constants are listed below. In the previous Modules we just use the `AF_INET`.

| Name | Purpose |
|--------------------------------|---------------------------------|
| <code>PF_UNIX, PF_LOCAL</code> | Local communication. |
| <code>PF_INET</code> | IPv4 Internet protocols. |
| <code>PF_INET6</code> | IPv6 Internet protocols. |
| <code>PF_IPX</code> | IPX - Novell protocols. |
| <code>PF_NETLINK</code> | Kernel user interface device. |
| <code>PF_X25</code> | ITU-T X.25 / ISO-8208 protocol. |
| <code>PF_AX25</code> | Amateur radio AX.25 protocol. |
| <code>PF_ATMPVC</code> | Access to raw ATM PVCs. |
| <code>PF_APPLETALK</code> | Appletalk. |
| <code>PF_PACKET</code> | Low level packet interface. |

Table 7: Domain parameters of `socket()`.

The `protocol` parameter specifies a particular protocol number/name string to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family, in which a case protocol can be specified as 0 (as used in the program examples in previous Modules). However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner. The protocol number to use is specific to the communication domain. Partial list of the protocol number have been discussed in **IPv4 Datagram Format** section (or you can check `man getprotobyname()` page or RFC1700 for a complete list). To map protocol name strings to protocol numbers you may use `getprotoent()` function.

In the previous Modules we have already made familiar with `SOCK_STREAM` and `SOCK_DGRAM`. In this section, we'll be using `SOCK_RAW`, which includes the IP headers (and all subsequent protocol headers of the upper layer as needed) and data. In the previous program examples also we used `SOCK_STREAM` (TCP/connection oriented) and `SOCK_DGRAM` (UDP/connectionless) sockets as shown below:

```
socket(AF_INET, SOCK_STREAM, 0);
```

And

```
socket(AF_INET, SOCK_DGRAM, 0);
```

For raw socket we code as follows:

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

socket(PF_INET, SOCK_RAW, IPPROTO_UDP);
socket(PF_INET, SOCK_RAW, IPPROTO_TCP);
socket(PF_INET, SOCK_RAW, IPPROTO_ICMP);
```

Depending on what you want to send, you initially open a socket and give it its type. For example:

```
sockd = socket(AF_INET, SOCK_RAW, <protocol>);
```

For the `<protocol>`, you can choose from any protocol (number or string) including `IPPROTO_RAW`. The protocol number goes into the IP header verbatim. `IPPROTO_RAW` places 0 in the IP header. A socket option `IP_HDRINCL` allows you to include your own IP header along with the rest of the packet. Then, you might use it as:

```
char on = 1;
setsockopt(sockd, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on));
```

You then build the packet and use a normal `sendto()`, `recvfrom()` etc.

The Internet IPv4 layer generates an IP header when sending a packet unless the `IP_HDRINCL` socket option is enabled on the socket. When it is enabled, the packet must contain an IP header that you should include in your program. Only processes with an effective user id of 0 (root) or the `CAP_NET_RAW` capability are allowed to open raw sockets. A protocol of `IPPROTO_RAW` implies the `IP_HDRINCL` is enabled. For this case, the following is a summary for the IP header.

| IP Header fields modified on sending by <code>IP_HDRINCL</code> | |
|---|----------------------|
| IP Checksum | Always filled in. |
| Source Address | Filled in when zero. |
| Packet Id | Filled in when zero. |
| Total Length | Always filled in. |

Table 8: Some IP header default values.

If `IP_HDRINCL` is specified and the IP header has a non-zero destination address then the destination address of the socket is used to route the packet. When `MSG_DONTROUTE` is specified the destination address should refer to a local interface, otherwise a routing table lookup is done anyways but gatewayed routes are ignored. If `IP_HDRINCL` isn't set then IP header options can be set on raw sockets with `setsockopt()` as shown before. Raw sockets are usually only needed for new protocols or protocols with no user interface (like ICMP). When a packet is received, it is passed to any raw sockets which have been bound to its protocol before it is passed to other protocol handlers (e.g. kernel protocol modules).

Raw sockets use the standard `sockaddr_in` address structure defined in `ip.h`. For example:

```
...
struct sockaddr_in sin;
...
// Address family
sin.sin_family = AF_INET;
// Port numbers
sin.sin_port = srcportnum;
// IP addresses
sin.sin_addr.s_addr = inet_addr(argv[1]);
```

Raw socket options can be set with `setsockopt()` and read with `getsockopt()` by passing the `SOL_RAW` family flag. Raw sockets fragment a packet when its total length exceeds the interface **Maximum Transfer Unit (MTU)**. A raw socket can be bound to a specific local address using the `bind()` call. If it isn't bound, all packets with the specified IP protocol are received. In addition a RAW socket can be bound to a specific network device using `SO_BINDTODEVICE` (check the `socket()` man page).

An `IPPROTO_RAW` socket is send only. If you really want to receive all IP packets use a `packet()` socket with the `ETH_P_IP` protocol. Note that packet sockets don't reassemble IP fragments, unlike raw sockets.

If you want to receive all ICMP packets for a datagram socket it is often better to use `IP_RECVERR` on that particular socket.

Raw sockets may tap all IP protocols in Linux for example, even protocols like ICMP or TCP which have a protocol module in the kernel. In this case the packets are passed to both the kernel module and the raw socket(s).

Maximum Transfer Unit (MTU)

The Maximum Transfer Unit (MTU) specifies the maximum transmission unit size of an interface. Each interface used by TCP/IP may have a different MTU value specified. The MTU is usually determined through negotiation with the lower-level driver and by using that lower-level driver value. However, that value may be overridden. Each media type (used in Ethernet, FDDI, Token Ring etc) has a maximum frame size that cannot be exceeded. The link layer is responsible for discovering this MTU and reporting it to the protocols above the link layer. Network Driver Interface Specification (NDIS) drivers may be queried for the local MTU by the protocol stack. Knowledge of the MTU for an interface is used by upper-layer protocols, such as TCP, which automatically optimizes packet sizes for each medium.

From the moment the raw socket is created, you can send any IP packets over it, and receive any IP packets that the host received after that socket was created if you read() from it. Note that even though the socket is an interface to the IP header, it is transport layer specific. That means, for listening to TCP, UDP and ICMP traffic, you have to create 3 separate raw sockets, using `IPPROTO_TCP`, `IPPROTO_UDP` and `IPPROTO_ICMP` (the protocol numbers are 6 for tcp, 17 for udp and 1 for ICMP).

With this knowledge, we can, for example, create a small **sniffer program** as shown in the following code portion that dumps out the contents of all tcp packets we receive and print out the payload, the data of the session/application layer etc.

```
int fd = socket(PF_INET, SOCK_RAW, IPPROTO_TCP);
/* single packets are usually not bigger than 8192 bytes but
   depend on the media standard of the Network Access layer such as
   Ethernet, Token Ring etc
*/

...
char buffer[8192];
struct ipheader *ip = (struct ipheader *) buffer;
struct udphheader *udp = (struct udphheader *) (buffer + sizeof(struct ipheader));

...
while (read(fd, buffer, 8192) > 0)
/* packet = data + ip header + tcp header */
/* Little Endian/Big Endian must be considered here */
printf("Dump the packet: %s\n", buffer + sizeof(struct ipheader) + sizeof(struct
tcpheader));
```

Continue on next Module...TCP/IP and RAW socket, more program examples.

Software Downloads

Download software instantly





[Shop now](#)

Privacy

Further reading and digging:

1. Check the best selling C / C++, Networking, Linux and Open Source books at Amazon.com.
2. GCC, GDB and other related tools.