Sockets

By Paul Krzyzanowski

Last update: April 20, 2012

Introduction

We have seen that data can sent and received between machines via IP. We also saw that TCP and UDP were developed as connection-oriented and connectionless transport-layer protocols, respectively, over IP. Transport layer protocols allow applications to communicate with other applications. How then do we associate these TCP or UDP data streams with our applications?

When we examined transport-layer communication, we saw that a server must define a transport address for a service and associate that address with the service. The client must be able to use this address in addition to the machine address to access the service. The most popular implementation of this concept is **sockets**. They were developed by the University of California, Berkeley for providing inter-process communication for the 4.2 BSD variant of the UNIX operating system. Since then, the interface has become pervasive among practically ever operating system that supports networking, including the various versions of Linux, OS X, Windows, and a wide variety of embedded systems.

Sockets were designed as a generalized IPC model with the following set of goals:

- Communication between processes should not depend on whether they are on the same machine
- Efficiency: this should be an efficient layer on top of network communication facilities
- Compatibility: processes that just read from a standard input file and write to a standard output file should continue to work in distributed environments
- Must support different protocols and naming conventions (different "communication domains" or "address families")

The **socket** is an abstract object from which messages are sent and received, much like a file is an abstract object on which file data is read and written. It is created in a communications domain roughly similar to that of a file in a file system. Sockets exist only as long as they are referenced. A socket allows an application to request a particular style of communication (virtual circuit, datagram, message-based, in-order delivery, ...). Unrelated processes should be able to locate communication endpoints, so sockets need to be named. The name of a socket is something that is meaningful within the specific communications domain. For example, within the IP domain, a "name" is an IP address and port number.

The achieve the goal of compatibility, a socket presents itself as a file descriptor. Once the right set of system calls have been executed to identify it and establish a connection, standard file system *read* and *write* system calls can be used on the socket, just like they can on a file.

Programming with sockets

There are several steps involved in creating a socket, locating the remote endpoint, and communicating over the socket. This section cannot cover the entire topic fully. Several of the references listed provide more complete information. On-line manual pages will provide you with the latest information on acceptable parameters and functions. The interface described here is the system call interface provided by the Berkeley family of operating systems (including OS X) and is generally extremely similar amongst all Unix systems (and many other operating systems).

1. Create a socket

A socket is created with the socket system call:

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

All the parameters as well as the return value are integers.

- *domain*, or *address family*: communication domain in which the socket should be created. Some of address families are AF_INET (IP family), AF_UNIX (local channel, similar to pipes), AF_NS (Xerox Network Systems protocols).
- *type*: type of service. This is selected according to the properties required by the application: SOCK_STREAM (virtual circuit service), SOCK_DGRAM (datagram service), SOCK_RAW (direct IP service). Check with your address family to see whether a particular service is available.
- *protocol*: indicates a specific protocol to use in supporting the sockets operation. This is useful in cases where some families may have more than one protocol to support a given type of service.

The return value from the *socket* system call is a file descriptor (a small integer). The analogy of creating a socket is that of requesting a telephone line from the phone company.

Creating a socket is conceptually similar to performing an *open* operation on a file with the important distinction that *open* creates a new reference to a possibly existing object whereas a *socket* creates a new instance of an object.

2. Name a socket

When we talk about *naming* a socket, we are talking about assigning a transport address to the socket. This operation is called **binding an address**. The analogy is that of assigning a phone number to the line that you requested from the phone company in step 1 or that of assigning an address to a mailbox.

You can explicitly assign an address or allow the system to assign one. The address is defined in a socket address structure. Applications find addresses of well-known services by looking up their names in a database (e.g., the file /etc/services). The system call for binding is:

```
int error = bind(int s, const struct sockaddr *addr, socklen_t addrlen)
```

where *s* is the socket descriptor obtained in step 1, *addr* is the address structure (`struct sockaddr *`) and *addrlen* is an integer containing the address length. One may wonder why don't we name the socket when we create it. The reason is that in some domains it may be useful to have a socket without a name. Not forcing a name on a socket will make the operation more efficient in those cases and remove confusion. Also, some communication domains may require additional information before binding (such as selecting a grade of service).

As users, we might want names to be user-friendly, such as *Bob's print server*. We don't get that here. Sockets is a low-level interface that is designed to operate comfortably with the layers of abstraction provided by the networking stack. If a network allows user-friendly textual names then *bind* would let us use them. For TCP and UDP, however, *bind* refers to assigning an IP address and port number.

3a. Accept connections (server-side operation)

For connection-based communication, the server has to first state its willingness to accept connections. This is done with the *listen* system call:

```
int error = listen(int s, int backlog)
```

The *backlog* is an integer specifying the upper bound on the number of pending connections that should be queued for acceptance. After a *listen*, the socket *s* is set to manage the queue of connection requests; it will not be used for data exchange.

Connections can now be accepted with the *accept* system call, which extracts the first connection request on the queue of pending connections. It creates a new socket with the same properties as the listening socket and allocates a new file descriptor for it. By default, socket operations are synchronous, or blocking, and accept will block until a connection is present on the queue. The syntax of accept is:

```
struct sockaddr *clientaddr;
socklen_t clientaddrlen = sizeof(struct sockaddr);
int snew = accept(int s, clientaddr, &clientaddrlen);
```

The `clientaddr` structure allows a server to obtain the client address. *accept* returns a file descriptor that is associated with a new socket. The address length field initially contains the size of the address structure and, on return, contains the actual size of the address. Communication takes place on this new socket. The original socket is used only for managing a queue of connection requests (you can, and often will, still listen for other requests on the original socket).

None of this is needed for connectionless sockets. For those, *recvmsg* and *recvfrom* system calls were created that allow one to specify the address and port for incoming messages.

3b. Connect (client-side operation)

For connection-based communication (TCP/IP), the client initiates a connection with the connect system call:

```
int error = connect(int s, const struct sockaddr *serveraddr, socklen_t serveraddrlen)
```

where *s* is the socket (type int) and *serveraddr* is a pointer to a structure containing the address of the server (`struct sockaddr *`). Since the structure may vary with different transports, connect also requires a parameter containing the size of this structure (`serveraddrlen`).

This call can also be used for connectionless service. In this case, no connection is established but the operating system will send datagrams and maintain an association between the socket and the remote address so that you don't have to specify the address each time you send or receive a message.

4. Exchange data

Data can now be exchanged with the regular file system *read* and *write* system calls using the socket descriptors. This is the most significant part about the desire for compatibility with file descriptors. After a connection has been established, the code can be completely unaware of networking and simply treat the socket as a file input/output stream, no different than a user's terminal or a disk-based file.

Additional system calls were added to support datagram service and additional networking features. The *send*/*recv* calls are similar to *read*/*write* but support an extra *flags* parameter that lets one peek at incoming data and to send out-of-band data. The *sendto*/*recvfrom* system calls are similar to *send*/*recv* but allow callers to specify or receive addresses of the peer with whom they are communicating (most useful for connectionless sockets). Finally, *sendmsg*/*recvmsg* support a full IPC interface and allow access rights to be sent and received. Could this have been designed cleaner and simpler? Most likely. The point to remember is that the *read*/*write* or *send*/*recv* calls must be used for connection-oriented communication and *sendto*/*recvfrom* or *sendmsg*/*recvmsg* must be used for connectionless communication. Also note that when you send data, it's possible that the other side may have to perform multiple reads to get results from a single write (because of fragmentation of packets) or vice versa (a client may perform two writes and the server may read the data via a single read).

5. Close the connection

The *shutdown* system call may be used to stop all further read and write operations on a socket:

```
int shutdown(int socket, int how)
```

Alternatively, the file system *close* system call can also be used to terminate all communications on a socket. *shutdown* offers more options with the *how* parameter, which can be set to:

- 0 (SHUT_RD): you can send but not receive data on this socket
- 1 (SHUT_WR): you can receive but not send more data on this socket
- 2 (SHUT_RDWR): you can neither send nor receive more data on this socket

Synchronous or asynchronous

Network communication, and file system access in general, system calls may operate in two modes: *synchronous* or *asynchronous*. In the synchronous mode, socket routines return only when the operation is complete. For example, *accept* returns only when a connection arrives. In the asynchronous mode, socket routines return immediately: system calls become non-blocking calls (e.g., *read* does not block). You can change the mode with the *fcntl* system call. For example,

```
fcntl(s, F_SETFF, FNDELAY);
```

sets the socket *s* to operate in asynchronous mode.

Sockets internals

Figure 1. Logical upward flow of data from a device to a socket.

Sockets are how an operating system exposes its networking subsystem to applications. Figure 1 shows a logical flow of data through this subsystem (the BSD implementation is used as a guide here). The logical layers include the following:

Network Interface Layer (Network Device Driver)

The network interface layer (link layer) is responsible for interfacing with network devices. It implements the network device driver that interfaces with the network device; for example, an ethernet driver interfacing with an ethernet transceiver on the computer. This layer responsible for performing packet encapsulation (wrapping packets within an ethernet packet, for instance) or decapsulation (stripping off an ethernet header). This layer corresponds to the link layer of the OSI reference model. A key difference between network devices and other devices is that they do not appear in the file system (e.g., under `/dev`). The device itself cannot be accessed via *read/write* operations. The I/O interface to network devices is packet-based, not the arbitrary byte stream of character devices or the fixed-size blocks of block devices.

Network Layer

The network layer is responsible for the delivery of data between network devices and higher levels of the networking stack. It needs to be aware of packet routing and must be able to select the appropriate outbound interface. It corresponds to the network layer of the OSI reference model.

Transport Layer

The transport layer maintains an association between a socket and transport layer addressing. For instance, it needs to identify the socket that corresponds to a particular <address, port> tuple for incoming data and generate TCP and UDP headers with appropriate addresses and port numbers for outbound packets.

Data flows are asynchronous. Incoming packets are received by the network device and passed onto per-protocol queues. The operating system schedules a kernel thread to process operations in these network queues. Processing a queue item may place it into another protocol's queue until the transport-layer interface is known. At that time, the data is sent to a receive queue for the associated socket. Figure 2 illustrates the flow of a packet between applications and network interfaces.

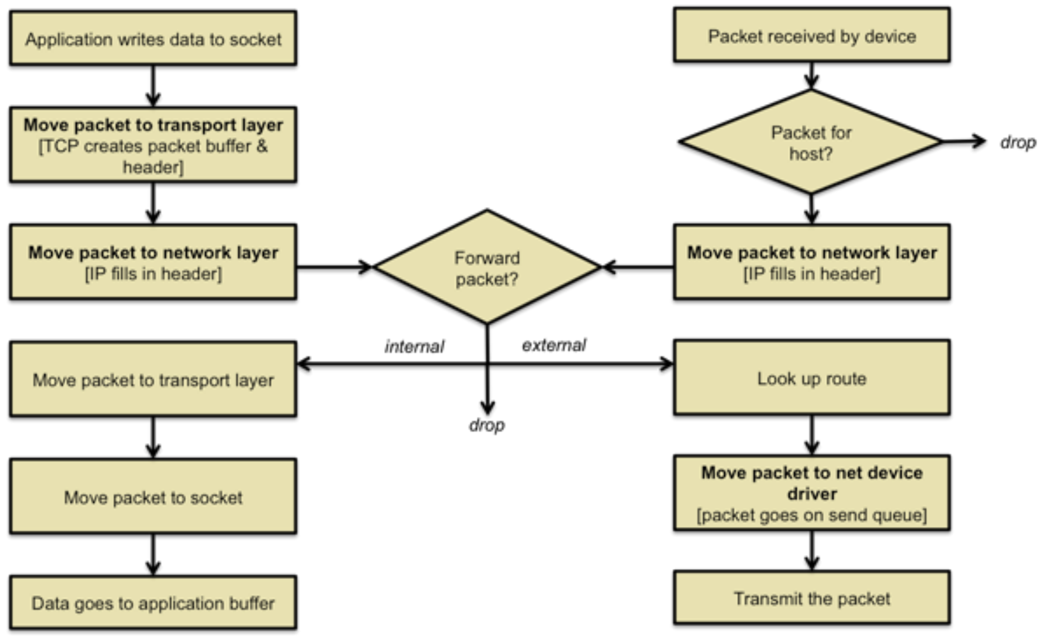
Application writes data to socket
Move packet to transport layer [TCP creates packet buffer & header]
Move packet to network layer [IP fills in header]
Move packet to transport layer
Move packet to socket
Data goes to application buffer
Packet received by device
Packet for host?
drop
Move packet to network layer [IP fills in header]
Forward packet?
internal
external
drop
Look up route
Move packet to net device driver [packet goes on send queue]
Transmit the packet

Figure 2. Packet data flow in the operating system

* * *

System call interface

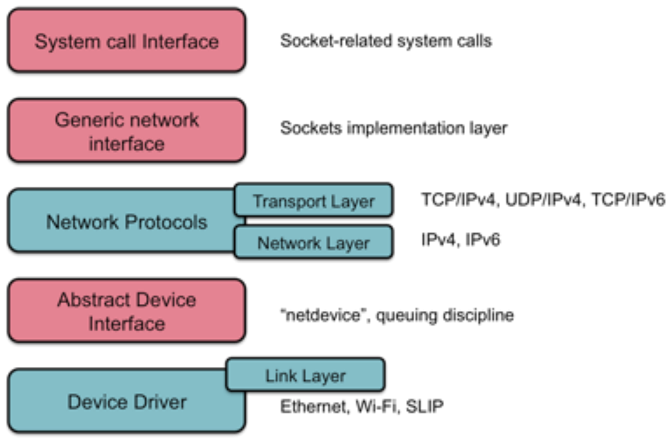
System call Interface
Socket-related system calls
Generic network interface
Sockets implementation layer
Network Protocols
Transport Layer
TCP/IPv4, UDP/IPv4, TCP/IPv6
Network Layer
IPv4, IPv6
Abstract Device Interface
"netdevice", queuing discipline
Device Driver
Link Layer
Ethernet, Wi-Fi, SLIP

Figure 3. Network stack

Within the operating system (Linux is the example here, but others are similar), the implementation of the networking subsystem comprises five layers (figure 3).

System calls provide the interface between application programs and the operating system. To an application, a socket looks like a file descriptor; that is, a small integer. That integer is an index into a per-process table in the kernel that keeps track of a process' open files. There are two ways to access the networking interface via the system call interface.

One method is via the several **socket-specific system calls** (*socket*, *bind*, *shutdown*, etc.). These calls are actually implemented as a single system call that take a parameter identifying the requested command (*sys_socketcall* defined in `socket.c`). The code in *sys_socketcall* directs the request to the appropriate function in the kernel.

The other method is via a **file descriptor operation**. That is, by a system calls that accept file descriptors as parameters (e.g., *read*, *write*, *close*, etc.). Since sockets were designed to be compatible with file descriptors, they reside along with file descriptors in the file descriptor table. Sockets are *not* implemented as a file system, however, and hence are not implemented under the Virtual File System (VFS). The distinction between a file and socket descriptor takes place just above the VFS layer. However, there is a direct parallel to the VFS structure in that a socket structure's `f_ops` field points to a set of functions that can be made on the socket. A socket acts as a queuing point for data that is being transmitted and received and has both send and receive queues associated with it. The queues contain high watermarks to avoid resource exhaustion. Only so much data can be queued before operations will block.

Generic network interface: sockets layer

All network communication takes place via a **socket**. Each socket that a process creates refers to a unique **socket structure** that keeps all the state of a socket, including the protocol and the operations that can be performed upon it. Similar to VFS for file systems, this layer provides common functions to support a variety of lower-level protocols (such as TCP, UDP, IP, raw ethernet, and other networks). In Linux systems, this structure is defined in `include/net/sock.h` in the Linux kernel source. Protocol-specific information (e.g., data specific to TCP/IP and UDP/IP rather than generic operations to send or receive data) are kept in a separate structure called **struct sock** that is associated with the socket.

A socket structure acts as a queueing point for data being sent from and received by the process that interacts with the socket. Each socket has send and receive queues associated with it which contain messages that the process wrote to the socket and messages that the process can read from the socket. Each of these messages point to a socket buffer that contains the data.

The socket buffer: `sk_buff`

The core component for managing the flow of a packet between the application and the device is a structure called `sk_buff`, or the **socket buffer**, which is defined in `include/linux/skbuff.h` in the Linux kernel. The socket buffer is a fixed place to store all the data related to a message (including various protocol headers) so that the kernel will not have to waste time copying parameters and packet data at each layer of the network stack.

This is a kernel data structure that contains the data packet, state, and control data encompassing multiple layers of the protocol stack. It contains fields that point to specific layers in the networking stack. For example, the `transport_header` contains transport-layer (layer 4; typically TCP or UDP) information; `network_header` contains network layer (layer 3; typically IP) information; and `mac_header` contains link layer (layer 2; typically ethernet) information. Packet data is never copied between the layers of the protocol stack; that would be too inefficient. Instead, a pointer to the socket buffer is moved among the various queues of the layers of the stack. In fact, packet data is copied only twice: once to move it from the user application into the socket buffer and a second time to move it from the socket buffer out to the network device (or vice versa). Where possible, the kernel would use DMA to move data between the socket buffer's memory and the network device.

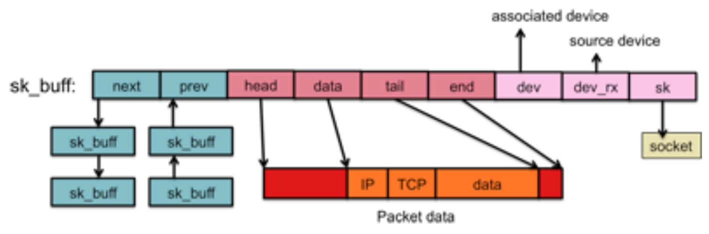

Figure 4. Socket buffer (sk_buff)

The socket buffer is created when network data arrives: either from a network device driver or from a user socket-based operation (*write*, *sendto*, *sendmsg* system calls). Each packet that is sent or received is associated with an `sk_buff` structure. The packet data is kept track of by the sk_buff and is identified by the pointer elements `data` and `tail` (start and end of data, respectively). The total allocated packet buffer is pointed to by the `head` and `end` elements. The reason for the two sets of pointers is to avoid reallocating and copying data to handle encapsulation. For instance, when we receive a TCP/IP packet from the ethernet interface, it is enveloped by an ethernet MAC header. Within it, we have the the IP packet. Within that, we have the TCP/IP packet. Within that, we have the data. Each layer of processing can adjust the `data` and `tail` elements to point to the reduced or increased packet that would be of interest to the next layer.

Sk_buffs are organized as a doubly linked list, so it is easy to move an element from one list to another list. Each sk_buff also identifies the ultimate network device in a structure `net_device`. The `rx_dev` element points to the network device that received the packet. The `dev` element identifies the network device on which the buffer operates. This is often the same as `rx_dev` but, if a routing decision has been made to a different interface, this contains that outbound interface.

Network protocols

Network protocols comprise implementations of all the specific protocols available to the system (e.g., TCP, UDP). Each protocol or family of protocols is a module. As with device drivers and file systems, the module may be a part of the bootable kernel or may be loaded dynamically. Also like other modules, each module is initialized and registered with the system at start-up. For example, the *proto_register* function for the built-in IP family of protocols calls the the *inet_init* function to registers them with the kernel. The *proto_register* function adds the protocol to the active protocol list and optionally allocates caches and buffers (e.g., TCP needs buffers to store connection state). Additional protocols can be added by calling the kernel function *inet_register_proto_sw*.

Each networking protocol has a structure called **proto** associated with it. The protocol structure is associated with an address family. An address family is a set of protocols. For example, the set of Internet Protocols is known as the Internet address family and specified as `AF_INET` by the programmer when creating the socket. The `proto` structure contains pointers to protocol-specific implementations of operations that can be made by the sockets layer. These include basic operations such as *create a socket*, *establish a connection with a socket*, *close a socket*, *send a message*, *receive a message*, etc.

Abstract device interface

The abstract device interface is an abstract layer that provides higher-level software with a uniform interface to network devices. It also contains a common set of functions for low-level device drivers to use to interact with the higher-level protocol stack. This layer is defined by a `net_device` structure. The actual network device driver is implemented underneath this layer.

Initialization

The abstract interface contains registration and unregistration functions for the network device (*register_netdevice*, *unregister_netdevice*) and an initialization function . The caller creates and populates a `net_device` structure and passes it for registration. During registration, the kernel calls the structure's *init* function to perform device-level initialization.

Sending

The sending capabilities of the abstract interface handle the sending of data in the `sk_buff` to the physical device. The layer's *dev_queue_xmit* function enqueues an `sk_buff` for transmission to the underlying driver. The device to which the data will be sent is defined in the `sk_buff`. The device structure contains a method called *hard_start_xmit*, which is the device driver's function for transmitting the data in the `sk_buff` to the network.

Receiving

When a network device receives a packet, it raises an interrupt. This interrupt is picket up by the kernel's interrupt handler and passed to the device driver for that network device. The driver allocates a socket buffer (`sk_buff`) as well as memory for the packet data (to which the socket buffer will point), which includes the headers and data.

If the contents are an IP packet, the `sk_buff` is passed to the network layer with a call to *netif_rx*, which causes the `sk_buff` to be placed on a queue for processing by the network layer (IP layer). It will be dequeued when the kernel thread calls *netif_rx_schedule*.

Network device drivers (data link layer)

The lowest layer of the networking stack is the set of network device drivers that interact with the physical network. Examples are drivers for ethernet, 802.11b/g/n wireless networks, and SLIP (serial line IP). Upon initialization, the driver allocates a `net_device` structure and initializes it with its device-specific functions:

- **dev->hard_start_xmit** defines how the upper layer should enqueue an `sk_buff` for transmission to the network. Typically, the packet is moved to a hardware queue and then transmitted.
- **net_rx** defines the function used to receive a packet from the hardware interface.

The device driver module calls the kernel *register_netdevice* function to make the device available to the networking stack. Unlike block and character devices, network devices do not present themselves as named devices within the file system.

The path of a packet

To put this discussion back into perspective, let's follow the path of a packet in two directions: from an application to the network and from the network to the application.

Sending a message

Write to a socket

The application writes data to a socket. For a TCP/IP socket, this will be via the *write* system call.

Socket calls the send *function*

The socket calls the *send* function that is specific for the protocol that was defined by the socket when it was created. Typically, this is AF_INET, for the Internet address family. The generic *send* function verifies the status of the socket and the protocol type. If valid, it then sends the request to the transport layer send function (e.g., TCP or UDP).

Transport layer: create an sk_buff

The transport layer send function creates a socket buffer (`struct sk_buff`) and allocates memory for the data with enough extra space for all foreseeable headers that all layers of the protocol stack may need. It then copies data from the user's memory to this kernel-level structure and populates the transport layer header (port number, options, data checksum). A pointer to this socket buffer is then passed to the network layer (typically IP).

Network layer

The network layer fills in its own header (including IP source and destination addresses, time-to-live field, header checksum). It then looks up the destination route for the packet to determine where this packet should be sent once it leaves the machine. This may not be the final destination since the packet may be sent to a router. Depending on the maximum transmission size of the destination device, the network layer may need to fragment this packet into multiple packets. Finally, the network layer uses the abstract device interface to call the link layer routines in the device driver.

To route packets, the network layer maintains a **Forwarding Information Base**, which is list of every known route and is generated from routing tables or dynamic routing algorithms. For efficiency, it also maintains a hash table that serves as a cache of frequntly-used routes.

Link layer

Now we're in the device driver, which is typically an ethernet or Wi-Fi driver. The link layer protocol adds the appropriate ethernet MAC headers to form the complete packet and places it in the transmit queue for the device. Our work is done and we have to wait for the kernel to schedule its worker thread to service queued requests for transmission.

Transmission

When the scheduler schedules the kernel worker thread to check the queue of outbound network requests, the thread sends the link header to the device and then transmits the packet. Typically DMA is used to move the data from the kernel's memory to the network card so the processor does not have to spend its cycles copying the data.

Receiving a packet

Network Device Driver: Top half

A message arrives at the network transceiver, which was typically picked up by the card because the destination ethernet address matched the one that belongs to the NIC. The NIC generates an interrupt, causing the kernel's interrupt handler to be invoked, which then calls the top half of the network device driver. Recall that the top half function of a device driver runs in interrupt context and cannot be preempted. The code tries to do as little as needed and delegate the real work to the bottom half of the driver, which will be processed by a kernel worker thread.

The top half allocates a new socket buffer and copies the incoming packet from the hardware buffer into the memory associated with the socket buffer. It then cals a function called *netif_rx* (on Linux systems), which moves the socket buffer onto a common input queue for processing. The top half then repeats this process until there are no more queued packets in the hardware.

Network Device Driver: Bottom half

The bottom half function is scheduled by the kernel and goes through the list of queued packets. For each packet, it calls a function called `net_rx_action`, which dequeues a socket buffer. Note that an ethernet header contains a protocol identifier so this link layer processing can tell what protocol is encapsulated within the data. Each protocol handler is responsible for registering itself and identifying the type of protocol that it handles. The *net_rx_action* function first calls a generic protocol handler and then the *receive* function that is registered with the protocol for that specific packet.

Network layer

IP is registered as a protocol handler for all ethernet packets whose payload is IP data (identified by `ETH_P_IP`, which happens to be 0x0800). The bottom half of the device driver passed the socket buffer to the appropriate protocol, which for this example is IP. The IP protocol will either route the packet for transmission to another destination (something routers do all the time), deliver it locally, or discard it. If routing, the packet will go to an outbound queue for transmission; we never need to look at the TCP or UDP data. In the case of this example, let's assume that it indeed is a packet for this system. The IP driver looks at the protocol field inside the IP header and calls the appropriate transport-level handler to process this incoming packet (e.g., *tcp_v4_rcv*, *udp_rcv*, *icmp_rcv*, etc.). This layer also has hooks for *netfilter* to allow for additional processing to implement a firewall (packet filter) that will make a further decision on whether to drop the packet or not.

Transport layer

Depending on the protocol value in the IP header, the packet will be passed to the appropriate transport layer protocol. Typically, this will be a call to either *tcp_v4_rcv* or *udp_rcv*. The transport layer checks for packet errors and looks for a socket that should receive this packet (based on the source and destination IP addresses and port numbers).

For TCP packets, it will then call *tcp_v4_do_rcv*, passing it the socket buffer and the socket (`struct sock`). The TCP receive function adds the socket buffer to that socket's receive queue. The process can now read the data. If the process was blocked on that socket (i.e., it was already trying to read data), then the process is moved to the *ready* state so that can be scheduled.

Socket

Finally, when the system call that reads the socket data (e.g., *read*) is ready to get the data and the process is context-switched in, the incoming data is copied from the socket buffer into the process' address space. If no data is left in the buffer, the socket buffer is freed.

Dealing with interrupts and NAPI

As networks got faster, machines could receive packets at a faster rate. Since each received packet would generate an interrupt, that meant a lot more interrupts. The median size of an ethernet packet is 413 bytes. If we assume a steady stream of network traffic, a 1 Gbps ethernet link will yield 300,000 interrupts per second. Even a 10 Mbps link will yield a whopping 3,000 interrupts per second. This can bring a processor to a state of livelock, where it spends the bulk of its time just servicing interrupts and making no progress in any other area.

To mitigate this, a combination of interrupts and polling may be used. Linux's "New API" (not that new, since it was created in 2009) disables network device interrupts during periods of high traffic and resorts to polling. When traffic subsides, it re-enables interrupts and abandons polling. When an incoming packet arrives and causes an interrupt, interrupts are disabled and polling of the device is enabled. If a polling period results in no packet, the system decides that packets are not coming in fast enough to warrant disabling interrupts and re-enables them.

References

- The Linux Networking Architecture, Socket Buffers (http://ngn.ee.tsinghua.edu.cn/~lujx/linux_networking/index.html?page=0131777203_ch04lev1sec1.html), *The Linux® Networking Architecture: Design and Implementation of Network Protocols in the Linux Kernel*, Klaus Wehrle, Frank Pählke, Hartmut Ritter, Daniel Müller, and Marc Bechler. Prentice Hall (ISBN 0-13-177720-3), August 01, 2004 (alternate link (http://flylib.com/books/en/3.475.1.1/1/))
- Understanding Linux Network Internals (http://www.amazon.com/dp/0596002556/pkorg), Christian Benvenuti. ISBN 0596002556, © O'Reilly 2006
- Linux Networking and Network Devices APIs (http://docs.blackfin.uclinux.org/kernel/generated/networking/index.html), Linux Kernel HTML Documentation, Kernel Version: 2.6.32.8.
- Anatomy of the Linux networking stack: from sockets to device drivers (http://www.ibm.com/developerworks/linux/library/l-linux-networking-stack/), M. Tim Jones, IBM developerWorks, June 2007. [also here (http://ldn.linuxfoundation.org/article/anatomy-linux-networking-stack)]
- Common Functionality in the 2.6 Linux Network Stack (http://download.intel.com/design/intarch/PAPERS/323704.pdf), Richard Kelly & Joseph Gasparakis. Intel Embedded, April 2010.
- Linux Device Drivers (http://www.xml.com/ldd/chapter/book/ch14.html), By Alessandro Rubini & Jonathan Corbet 2nd Edition, O'Reilly, June 2001, 0-59600-008-1.
- Linux Networking Internals (http://www.scribd.com/doc/3471003/Linux-Networking-Internals), Michael Opdenacker, Oron Peled, Codefidence Ltd., 2004-2008.

- Linux Networking Subsystem; Desktop Companion to the Linux Source Code (http://serghei.net/docs/kernel/net.pdf), by Ankit Jain, Linux-2.4.18, Version 0.1, May 31, 2002.
- Linux Network Stack Walkthrough (2.4.20) (http://gicl.cs.drexel.edu/people/sevy/network/Linux_network_stack_walkthrough.html), Jonathan Sevy, Drexel University
- Chapter 14 Network Drivers (http://www.xml.com/ldd/chapter/book/ch14.html#t2) Linux Device Drivers, 2nd Edition, by Alessandro Rubini & Jonathan Corbet, 2nd Edition June 2001.
- How to achieve Gigabit speeds with Linux (http://datatag.web.cern.ch/datatag/howto/tcp.html)
- Linux Socket Buffers Introduction (http://www.ittc.ku.edu/Projects/Wireless_ATM/linux/linux-skbuff.html)
- sk_buff (http://www.linuxfoundation.org/collaborate/workgroups/networking/skbuff), The Linux Foundation, November 19, 2009.
- Linux Networking Kernel (http://www.ecsl.cs.sunysb.edu/elibrary/linux/network/LinuxKernel.pdf), Version 0.1, February 12, 2003.
- The journey of a packet through the linux 2.4 network stack (http://ftp.gnumonks.org/pub/doc/packet-journey-2.4.html), Harald Welte, October 2000.
- Kernel sockets (http://www.haifux.org/lectures/217/netLec5.pdf)