SEE TEXT ONLY

INTERNETWORKING in FreeBSD

By George V. Neville-Neil

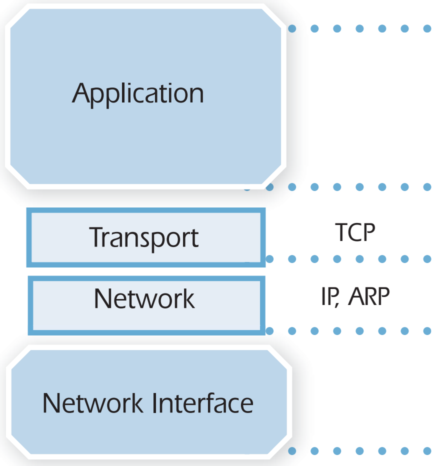
The FreeBSD network stack is the kernel software that, when taken together, allows programs on one computer to talk to programs on another. It is a highly reusable software artifact that has been employed not only within the operating system, but has been ported many times to other environments, including embedded operating systems completely unrelated to FreeBSD. For most developers, working both inside and outside the operating system's kernel, the network stack is a black box, something to be used but rarely understood. This article provides a short overview of the network stack software and how it works.

TCP/IP in a Nutshell

The Internet is built upon a small number of networking protocols which, when taken together, implement all that is necessary for one computer to talk to another. All these network protocols are implemented within the operating system's kernel. The reason network protocols are implemented within the kernel is that they are a resource that is shared among all of the users of a system, whether those users are humans or programs. Figure 1 shows some of the network protocols that implement the current Internet. We see that the protocols are layered and that they form a stack, hence the term Network Stack. Network protocols are thought of as layers because a system as complex as the Internet is built up of several, cooperating protocols, each of which has specific responsibilities and is dependent upon the layers below it to provide services necessary for the sum of the parts to implement the whole. Our figure shows three protocols, Address Resolution (ARP), Internet (IP) and Transmission Control (TCP).

For two computers to successfully talk to each other over an

Fig. 1 TCP/IP Model



Internetwork, three things must take place. The first is that information from one computer must be broken down into small enough pieces, which we call packets, to be transmitted between the source and destination of the communication. When you click on a link on the web and request a page, that page of data doesn't get transmitted as one large image of a cat; that image has to be broken down into pieces, often about 512 bytes in size; each piece must be sent across the network, and the destination computer must put all the pieces back together. The Transmission Control Protocol (TCP) is the protocol and layer that is responsible for the process of breaking down an image into packets, and giving each packet a sequence number, so that the destination computer can reassemble the packets it receives into the original image.

Once data is broken down into packets, they must be transported between the source and destination computers. The Internet Protocol (IP) is responsible for getting each individual packet between the source and destination. Every IP packet has both a source and destination address that indicates the ultimate endpoints of communication, but the Internet is a store-and-forward, packet-switching network, which means that most endpoints are not directly connected to the same local network. For an IP packet to reach its destination, it must be passed over a series of hops between one or more routers, before it reaches its final destination. The IP layer in the operating system kernel is responsible for finding the next hop along the path between the source and destination endpoints.

The Internet Protocols are abstract enough that various types of networking hardware can pass these packets without knowing anything about what is contained within the packets. Whether a packet goes over a wired or wireless link is unimportant to IP, but at some point, every computer or router has to contact the next hop along the path to the ultimate destination. For Ethernet-based networks, the Address Resolution Protocol (ARP) is responsible for finding the hardware address for the next hop along the path in the network. Every piece of Ethernet hardware has a 6-byte source and destination address, and it is ARP's job to translate a local IP address, usually of the next hop router, into a hardware address that the router is listening to.

Most programmers interact with the TCP/IP stack via the `sockets(2)` set of system calls. One of the key innovations of sockets is that it provides the programmer with an API that looks very much like a simple file access where all of the standard APIs, such as `read(2)` and `write(2)` work in the same way for network communication as they do for local file access. The Socket code can be thought of as a layer on top of the TCP/IP protocols that crossed the User/Kernel boundary to give programs access to network communication.

Summing up, the TCP/IP protocol suite running in a kernel on top of a wired Ethernet must: break a data stream into packets (TCP), address those packets so that they can move between a source and destination (IP), and finally figure out which piece of hardware is the next hop along the path between the source and destination (ARP). The sockets API ties the network stack back into user space so that programs can have access to the networking code. All of these layers are implemented in the FreeBSD kernel,

The Network Stack

Our goal here is not a full read through of the source code, which is well beyond the scope of this article, but, instead, to give you, the reader, an idea of how these pieces fit together and how you might even start learning how the code works.

The FreeBSD Network Stack is implemented in a set of C files contained in the operating system's `sys/` directory. The TCP and IP protocols are contained within `sys/netinet/`, and the ARP protocol, as well as much of the support for various link layers, such as Ethernet, are kept in `sys/net/`. The sockets API is mostly contained in the `sys/kern/` directory, along with much of the generic kernel infrastructure.

As the network stack is implemented as a set of layers, we must be concerned not only with the responsibilities of each layer, but also the form and function of how data is passed between them. Looking down from the top of the stack, a program hands a set of bytes of arbitrary length down into the network stack via the sockets API.

Each endpoint of communication for a program on a system is represented by a socket. The socket structure contains a great deal of metadata about the data it handles, as well as two queues for data—one for inbound communication, and one for outbound, which we call the receive and send socket buffers respectively.

All memory in the network stack is kept in a single, unified, data type called an `mbuf`, short for memory buffer. A set of `mbufs` can be chained together via a forward pointer, and this is how large areas of memory are broken down into packets that can be properly handled by the lower layers of the network stack. The `mbuf` system is a kernel private memory pool which user programs are never exposed to.

A `write(2)` call on a socket takes an area of memory and, via the system mechanism, makes that data available in the kernel, placing it into a socket buffer. Each socket buffer maintains a list of buffers that contain the data that's to be transmitted or that is in the process of being received. Once the data is contained in the socket buffers, the TCP machinery is invoked in the kernel and the `mbuf`s contained in the socket buffer are updated and modified to be TCP segments, which are then further broken down by the IP protocol machinery and finally transmitted by the kernel via an Ethernet device driver.

Receiving data in the network stack is more complicated than reading from a file on a local system because network data can arrive at any time. A web server does not know to call `read(2)` before a web browser contacts it, and so the FreeBSD kernel must be constantly waiting for data and ready to create new communication endpoints as they arrive. When a client contacts a server, a TCP packet with a special flag (`SYN`) is received by the kernel, and the kernel then attempts to set up all of the state required for communication with the new endpoint. Only after the kernel has satisfied itself that it can communicate with the new endpoint does it alert a user space program that there is a new incoming connection, which the user space program can now `accept(2)` and then begin to `read(2)` data.

Look but Don't Touch (DTrace and the Network Stack)

How can we look at the network stack without trying to read the entire source code all at once? Using the built-in DTrace tracing system on FreeBSD, we can actually see each layer in action while running some simple test programs. As DTrace is completely safe to use on a running system, we can start to explore the network stack without having to modify and recompile the kernel code. For those not familiar with DTrace, you might want to start with the tutorial at https://wiki.freebsd.org/DTrace/Tutorial. Since this article presents a set of worked examples, it's easy enough to run these commands on a system of your own without knowing all the ins and outs of DTrace.

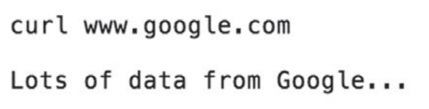
For simplicity we'll start with looking at an outgoing network connection. The `curl` command can be used to retrieve a single web page, and so we'll use www.google.com as our example.

As `curl` does not encrypt its data, unlike `ssh`, it is an excellent test tool with which we can look at the network stack.

Figure 2 shows how we can manually retrieve the base page from Google's website. The web page presented by Google is deceptively simple when seen in a web browser, but they embed a lot of code in their base page and so the output from the `GET` command runs to a few pages in a standard terminal. We're not interested in the output; we're interested in what happens when we initiate the communication.

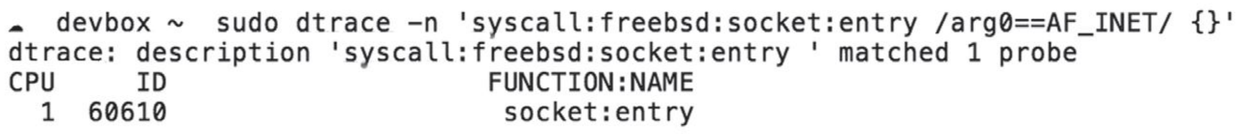
Starting from the socket layer, we can see how `curl` begins communicating with Google by looking for calls to the `socket(2)` system call. All network communication requires a socket to be created. The socket call as seen by DTrace in Figure 3 doesn't show much of interest, and that's because all the `socket(2)` call does is set up the program's local endpoint for communication. To catch `curl` communicating with Google, we need to look instead at the `connect(2)` call. From a program's standpoint, `connect(2)` is the actual beginning of communication and is also the routine that will start the network stack's machinery which we can then observe. The `connect(2)` system call eventually leads, in the kernel, to TCP's connect routine being called, which can be seen using DTrace's `tcp:::connect-request` tracepoint. The connect-request tracepoint has quite a lot of information

Fig. 2



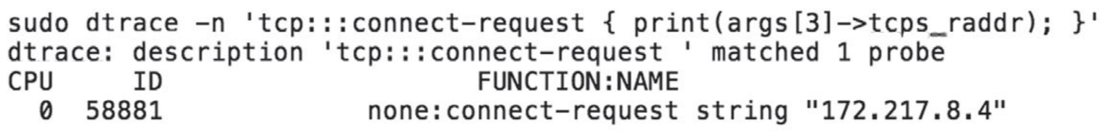
```
curl www.google.com

Lots of data from Google...
```

Fig. 3



```
  devbox ~  sudo dtrace -n 'syscall:freebsd:socket:entry /arg0==AF_INET/ {}'
dtrace: description 'syscall:freebsd:socket:entry ' matched 1 probe
CPU     ID                    FUNCTION:NAME
  1  60610                     socket:entry
```

Fig. 4



```
sudo dtrace -n 'tcp:::connect-request { print(args[3]->tcps_raddr); }'
dtrace: description 'tcp:::connect-request ' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0  58881            none:connect-request string "172.217.8.4"
```

about the connection being attempted but for now we simply wish to see where the connection is going.

Figure 4 shows the output of a DTrace one-liner that catches `curl` in the act of contacting Google. We specify the connect-request tracepoint so that we can see our source machine trying to contact the destination, and we print the `tcps_raddr` (remote address) to see from what IP address Google is currently willing to communicate with us. If you run the `curl` command three times, you will see three lines of output, one for each connection.

Connecting the source system to the destination system requires the execution of the TCP state machine. Using the `/usr/share/dtrace/tcptrack`, we can see all of these state changes when contacting Google and retrieving its home page. Figure 5 shows the entire set of state transitions that the TCP layer moves through in order to set up a connection, retrieve data, and then close the connection and clean up after itself. Each socket starts out in a closed state (`state-closed`) and waits there until communication is initiated. When our source connects to the destination, it sends a special packet marked with a `SYN` flag, which moves the state machine into the `state-syn-sent`. Our socket will remain in this state until the destination replies and continues to set up the connection, indicated by state-established. The TCP state machine remains in the established state until one or the other side of the connection wishes to close it. When the connection is closed, the state machine moves through various states, all shown on the last three lines of Figure 5. A fuller discussion of the TCP state machine can be found in *The Design and Implementation of the FreeBSD Operating System*, but for our purposes, once the state transitions to `state-time-wait`, we are satisfied that this connection is closed.

Fig. 5

```
sudo ./tcptrack
State changed from state-closed                state-syn-sent
State changed from state-syn-sent              state-established
Connection established to 216.58.194.164:80 from 172.20.10.7:22045
State changed from state-established           state-fin-wait-1
State changed from state-fin-wait-1            state-fin-wait-2
State changed from state-fin-wait-2            state-time-wait
```

Fig. 6

```
sudo dtrace -n 'tcp:::send { tracemem(args[4]->tcp_hdr, 128); }' | more

Search output for "GET" to find...

0  58883                        none:send
             0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f  0123456789abcdef
         0: 86 f2 00 50 46 df a5 bc 67 fa 89 b2 80 18 04 0b  ...PF...g.......
        10: 51 6f 00 00 01 01 08 0a 00 43 d1 bd 5f b9 ba 6e  Qo.......C.._..n
        20: 47 45 54 20 2f 20 48 54 54 50 2f 31 2e 31 0d 0a  GET / HTTP/1.1..
        30: 48 6f 73 74 3a 20 77 77 77 2e 67 6f 6f 67 6c 65  Host: www.google
        40: 2e 63 6f 6d 0d 0a 55 73 65 72 2d 41 67 65 6e 74  .com..User-Agent
        50: 3a 20 63 75 72 6c 2f 37 2e 35 36 2e 30 0d 0a 41  : curl/7.56.0..A
        60: 63 63 65 70 74 3a 20 2a 2f 2a 0d 0a 0d 0a 2f 97  ccept: */*..../.
        70: 4d 1a af 10 c3 50 53 ec c2 11 de 45 00 00 00 00  M....PS....E... . 1
```

While connection setup and teardown is a complicated affair, it actually doesn't move any of our data between systems. In order for us to see how `curl` communicates with Google, we can use DTrace's TCP send tracepoint. Figure 6 has a DTrace one-liner that will show all of the data being sent and received via TCP. If we were to try this test with `ssh(1)` or an HTTPS-enabled web server, we would not be able to find the plain text because the data would be encrypted before the TCP layer would see it, but with `curl` we get to see the plain text. In our example, we can see both the raw bytes in tabular form as well as the ASCII representation of the communication, and we can clearly see the `GET` command issued to the Google server. That's the same `GET` command that is passed down via a `write(2)` call into the network stack, which is then put into a socket buffer, handed to TCP, chopped up into packets, and finally transmitted via IP to the server.

In order to clearly see how the TCP and IP layers interact, we can compare the output of Figure 7 to Figure 6. Figure 6 was at the TCP layer and therefore only had TCP information, such as the packet's source and destination ports, sequence number, etc. Figure 7 is at the IP layer, one layer lower, and, as such, we

Fig. 7

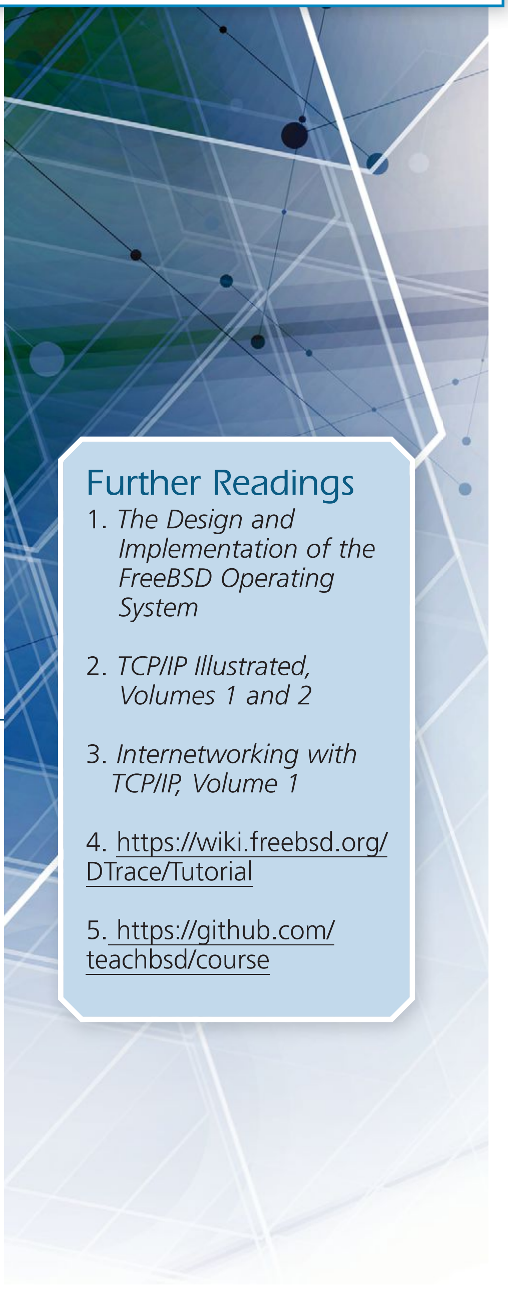
```
sudo dtrace -n 'ip:::send { tracemem(args[4]->ipv4_hdr, 128); }' | more
dtrace: description 'ip:::send ' matched 1 probe

Search output for "GET" to find...


 0  58876                        none:send
             0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f  0123456789abcdef
         0: 45 12 00 60 00 00 40 00 40 06 2e ef c0 a8 45 45  E..`..@.@.....EE
        10: c0 a8 45 01 00 16 c0 e7 7c b5 69 a2 38 44 cd 24  ..E.....|.i.8D.$
        20: 80 18 04 02 0b ea 00 00 01 01 08 0a 83 54 72 db  .............Tr.
        30: 36 50 49 f6 7e b2 8d 07 17 61 d9 5d e1 44 67 95  6PI.~....a.].Dg.
        40: 6c 8e 80 f4 f4 7e aa e0 c0 84 7a 88 b3 f8 96 81  l....~....z.....
        50: e0 b1 4e 32 59 0e c4 76 b5 05 23 ff 69 e1 58 9b  ..N2Y..v..#.i.X.
        30: a1 8c 4e 58 47 45 54 20 2f 20 48 54 54 50 2f 31  ..NXGET / HTTP/1
        40: 2e 31 0d 0a 48 6f 73 74 3a 20 77 77 77 2e 67 6f  .1..Host: www.go
        50: 6f 67 6c 65 2e 63 6f 6d 0d 0a 55 73 65 72 2d 41  ogle.com..User-A
        60: 67 65 6e 74 3a 20 63 75 72 6c 2f 37 2e 35 36 2e  gent: curl/7.56.
        70: 30 0d 0a 41 63 63 65 70 74 3a 20 2a 2f 2a 0d 0a  0..Accept: */*..
```

see that the `GET` command has moved a considerable way further down into the data stream. The information required by the IP layer, such as the destination and source network address, is responsible for this displacement of the `GET` command.

Conclusions

Networking is a complicated topic and its implementation in the operating system kernel is covered at various levels in some of the Further Readings (1, 2 3). Understanding how the network stack works requires remembering that the network stack is broken down into modules that roughly match the protocols that are being implemented. The Internet Protocols are in the IP layer, and the Transmission Control Protocol is contained in the TCP layer, and these layers interact with each other through a small number of well-defined kernel APIs. The DTrace system on FreeBSD is the perfect tool for those who wish to start exploring the network stack. Various DTrace-related tutorials can be found in the Further Readings section at right (4, 5). ●

GEORGE V. NEVILLE-NEIL works on networking and operating system code for fun and profit. He also teaches courses on various subjects related to programming. His areas of interest are code spelunking, operating systems, networking and time protocols. He is the coauthor with Marshall Kirk McKusick and Robert N. M. Watson of *The Design and Implementation of the FreeBSD Operating System*. For over 10 years he has been the columnist better known as Kode Vicious. He earned his bachelor's degree in computer science at Northeastern University in Boston, Massachusetts, and is a member of ACM, the Usenix Association, and IEEE. He is an avid bicyclist and traveler and currently lives in New York City.



Further Readings

1. *The Design and Implementation of the FreeBSD Operating System*

2. *TCP/IP Illustrated, Volumes 1 and 2*

3. *Internetworking with TCP/IP, Volume 1*

4. https://wiki.freebsd.org/DTrace/Tutorial

5. https://github.com/teachbsd/course