BY MARK JOHNSTON

DTrace

FreeBSD offers a plethora of tools and tricks for answering questions about activity in the network stack.

Utilities like systat(1) and netstat(1) are great at giving us a few starting points: they can, for instance, show packet and bit rates on a per interface or per protocol basis. More sophisticated tools will make use of the Berkeley Packet Filter (BPF) to track individual packets as they enter and leave the system; iftop (available in ports as sysutils/iftop) uses this technique to display bit rates per 4-tuple. The venerable tcpdump(1) also uses the BPF interface to capture and log packets in real time, making it possible to answer questions based on post-capture analysis.

With FreeBSD 10, the kernel contains a new set of DTrace probes which give users a great deal of visibility into the inner workings of the network stack. Specifically, users can now write scripts based on packet send and receive events in the IP, TCP and UDP layers of the FreeBSD kernel, and additionally peer into the internal TCP state of a given connection in real time. This is a powerful addition to any programmer's or sysadmin's toolbox, since it provides a framework for answering arbitrary questions about the behavior of FreeBSD's IP stack; rather than being limited by the output of existing utilities, DTrace allows people to develop their own tools to explore network activity, whether the motivation is to monitor performance indicators, pinpoint the source of problems, or to simply learn more about network protocols.

This article will give an overview of each of the new probes, explaining their use and providing examples. It assumes a basic knowledge of and familiarity with DTrace, but all of the examples used this article are either dtrace(1) commands—which can be run directly from the shell—or executable scripts. They have all been developed and tested on FreeBSD 10, and readers with an available testing system are encouraged to try and run them to get a feel for DTrace's capabilities. They are available for download at http://people.freebsd.org/~markj/dtrace/network-providers/examples/.

One caveat is that the FreeBSD implementation of these probes is relatively new, so it is naturally possible that you may run into bugs or hard-to-explain behavior when experimenting with them. DTrace guarantees that scripts cannot crash the system or otherwise corrupt its state, so there is no danger in running the examples or any DTrace scripts on FreeBSD. However, if you run into problems making use of the new probes or when using DTrace, please report them on the freebsd-dtrace@FreeBSD.org mailing list. Problems and questions related to DTrace which aren't specific to FreeBSD should be reported on the dtracediscuss@lists.dtrace.org mailing list; many of the original and current developers of DTrace are subscribed and will readily respond to posts on this list.

DTrace on FreeBSD

Users of FreeBSD's DTrace implementation will possibly be familiar with the fbt provider, which is used to trace function calls in the FreeBSD kernel as they occur. This provider is exceptionally handy for users already familiar with FreeBSD kernel internals, but suffers from a few downsides:

- Its use requires a moderately good understanding of kernel code, which is an unreasonably high barrier to entry for most users wishing to write their own scripts.
- The fbt probes are by definition tightly coupled to kernel code; if the code underlying a script changes, the script may fail to run or may produce incorrect results. So scripts written for one version of FreeBSD may not work on another, and almost certainly won't work on other operating systems.
- Individual fbt probes often do not correspond nicely to logical system events. Suppose you wish to write a DTrace script which prints the destination address of each IP packets as FreeBSD hands them over to the network card driver. It turns out that this is a discouragingly difficult task: it involves instrumenting at least four different functions in various parts of the IPv4 and IPv6 code, each of which is called with different arguments.

The new network probes allow users to write scripts and trace network-related events using an interface that doesn't suffer from the problems above. They provide a stable and simple window into the kernel's activity: to trace the destination address of IP packets, simply run:

Example 1:

```
# dtrace -n 'ip:::send {printf("%s", args[2]->ip_daddr);}'
```

Implementing this exact functionality using the fbt provider would involve writing at least fifty lines of fairly impenetrable D code, at least by the author's estimation after a half-hearted and mostly failed attempt at the exercise. By comparison, this example is quite straightforward and transparent, aside from the somewhat enigmatic "`args[2]`." In English, it essentially reads as "every time we send an IP packet, print its destination address." When this command is run on the author's laptop for several seconds while simultaneously pinging an internet address, we get the following output:

```
# dtrace -n 'ip:::send {printf("%s", args[2]->ip_daddr);}'
dtrace: description 'ip:::send ' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0  36564                           :send 8.8.178.110
  0  36564                           :send 8.8.178.110
  0  36564                           :send 8.8.178.110
  0  36564                           :send 8.8.178.110
```

This is DTrace's default output formatting. We can get more control by adding "`-x quiet`" to the dtrace(1) arguments and printing the newline ("\n") ourselves:

```
# dtrace -x quiet -n 'ip:::send {printf("%s\n", args[2]->ip_daddr);}'
8.8.178.110
8.8.178.110
8.8.178.110
8.8.178.110
```

Within a D script, the same effect can be achieved by adding the following line to the beginning of the file:

```
#pragma D option quiet
```

The New Network Probes

The network probes discussed in this article originate from Solaris and are also present in illumos and OS X. They belong to the new ip, tcp and udp DTrace providers; the following command in Example 2 will list them on your system:

Example 2:

```
# dtrace -l -P ip -P tcp -P udp
ID         PROVIDER      MODULE                   FUNCTION NAME
36492      ip            kernel                   receive
36493      ip            kernel                   send
36494      tcp           kernel                   accept-established
36495      tcp           kernel                   accept-refused
36496      tcp           kernel                   connect-established
36497      tcp           kernel                   connect-refused
36498      tcp           kernel                   connect-request
36499      tcp           kernel                   receive
36500      tcp           kernel                   send
36501      tcp           kernel                   state-change
36502      udp           kernel                   receive
36503      udp           kernel                   send
```

From this we can see that FreeBSD now has probes for IP, TCP and UDP packet send and receive events. The IP send and receive probes (i.e. `ip:::send` and `ip:::receive`) fire whenever FreeBSD sends or receives an IPv4 or IPv6 packet, respectively. Similarly, the TCP and UDP send and receive probes fire when the kernel sends or receives a TCP or UDP packet. As we can see, the tcp provider contains additional probes corresponding to TCP protocol events; for now, we will look at the send and receive probes and construct several examples.

Each of the network probes takes several arguments which together describe the packet which caused the probe to fire. The arguments themselves are each collections of related information. For instance, the third argument to `ip:::send` ("`args[2]`" in our earlier example) contains the high-level IP fields common to both IPv4 and IPv6: the IP version (`ip_ver`), the length of the payload (`ip_plength`), and the source and destination addresses (`ip_saddr` and `ip_daddr`). The fourth argument contains information describing the network interface used to transmit the packet, and the fifth and sixth arguments respectively yield the detailed IPv4 and IPv6 fields of the packet. That is, if the packet uses IPv4, "`args[4]`" will expose the fields of its IPv4 header, whereas an IPv6 packet will have its header fields exposed through "`args[5]`". A comprehensive list and descriptions of the `ip:::send` probe arguments are available in [1], so they will not be duplicated here; corresponding pages are available for the tcp and udp providers at [2] and [3] respectively. Using the send and receive probes, we can print output to the terminal for each packet in real time. On a heavily loaded system, this will of course be impractical; in this situation, users will want to make use of DTrace's data aggregation facilities or add predicates to allow selective tracing of packets. The following script prints basic information about each TCP packet as it enters and leaves the system. Note that forwarded TCP packets will not cause the tcp probes to fire since they are not examined by the TCP code.

Example 3 makes use of three probes. The `dtrace:::BEGIN` probe is used to print column

Example 3:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10Hz

dtrace:::BEGIN
{
    printf(“ %30s %-6s %30s %-6s %-6s %s\n\n”, “SADDR”, “SPORT”,
        “DADDR”, “DPORT”, “BYTES”, “FLAGS”);
}

tcp:::receive,
tcp:::send
{
    printf(“ %30s %-6u %30s %-6u %-6u (%s%s%s%s%s%s\b)\n”,
        args[2]->ip_saddr, args[4]->tcp_sport,
        args[2]->ip_daddr, args[4]->tcp_dport,
        args[2]->ip_plength - args[4]->tcp_offset,
        (args[4]->tcp_flags & TH_FIN) ? “FIN|” : “”,
        (args[4]->tcp_flags & TH_SYN) ? “SYN|” : “”,
        (args[4]->tcp_flags & TH_RST) ? “RST|” : “”,
        (args[4]->tcp_flags & TH_PUSH) ? “PSH|” : “”,
        (args[4]->tcp_flags & TH_ACK) ? “ACK|” : “”,
        (args[4]->tcp_flags & TH_URG) ? “URG|” : “”);
}
```

headers for the output of the tcp probe actions. As the headers suggest, the tcp probes print the 4-tuple associated with each TCP packet, along with the TCP payload size and the TCP flags. Some sample output shows NAT’ed SSH, HTTP and IMAPS traffic:

```
                          SADDR SPORT                            DADDR DPORT  BYTES  FLAGS

fe80:3::fa1a:67ff:fe03:f659     22      fe80:3::250:b6ff:fe0e:a825   42705  36     (PSH|ACK)
fe80:3::fa1a:67ff:fe03:f659     22      fe80:3::250:b6ff:fe0e:a825   42705  628    (PSH|ACK)
 fe80:3::250:b6ff:fe0e:a825     42705  fe80:3::fa1a:67ff:fe03:f659   22     0      (ACK)
fe80:3::fa1a:67ff:fe03:f659     22      fe80:3::250:b6ff:fe0e:a825   42705  100    (PSH|ACK)
 fe80:3::250:b6ff:fe0e:a825     42705  fe80:3::fa1a:67ff:fe03:f659   22     0      (ACK)
               192.168.0.27     41116               173.194.76.108   993    37     (PSH|ACK)
             173.194.76.108     993                   192.168.0.27   41116  20     (ACK)
             173.194.76.108     993                   192.168.0.27   41116  63     (PSH|ACK)
               192.168.0.27     41116               173.194.76.108   993    0      (ACK)
            173.252.102.241     443                   192.168.0.27   50220  429    (PSH|ACK)
               192.168.0.27     50220              173.252.102.241   443    911    (PSH|ACK)
            173.252.102.241     443                   192.168.0.27   50220  20     (ACK)
               192.168.0.27     16039                 31.13.69.160   443    37     (PSH|ACK)
               31.13.69.160     443                   192.168.0.27   16039  20     (ACK)
```

The tcp probe action is a single printf() call which uses the third and fifth arguments to `tcp:::send` and `tcp:::receive`; these arguments contain the IP and TCP headers of the corresponding packet and make it easy to retrieve the 4-tuple associated with the packet. The TCP payload size is a bit trickier: the IP header contains the IP payload size, and the TCP header contains the offset from the beginning of the TCP header to the TCP payload; thus their difference gives the size of the TCP payload. Note that the TCP payload sizes reported by DTrace for outbound segments may be larger than the MSS for the connection if TSO is enabled on the outbound interface. Finally, `args[4]->tcp_flags` contains the segment's TCP flags, and the DTrace TCP library (found in `/usr/lib/dtrace/tcp.d` on FreeBSD) contains symbolic names for the each of the TCP flags.

Though the script above nicely demonstrates the information available through network probes, it is not particularly useful except as a learning tool since it generally prints an unmanageable amount of output on any system with continuous TCP activity. Fortunately, DTrace makes it easy to visualize aggregations of data, and the network providers can be used to examine the distributions of variables such as connection duration, latency, bitrates, packet count, and packet size. Moreover, DTrace's flexibility makes it possible to measure these quantities over virtually any independent variable(s): by host, port, L3 protocol, or network interface. It is also possible to examine network traffic on a per-process basis, though on FreeBSD this currently requires a special trick which will be shown later in the article.

A quick demonstration of this is given in Example 4, which will print a per-interface his-

Example 4:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10Hz

ip:::send,
ip:::receive
{
     @num[args[3]->if_name] = lquantize(args[2]->ip_plength, 0, 1500, 100);
}
```

Example 5:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10Hz

tcp:::state-change
/(args[5]->tcps_state == TCPS_CLOSED && args[3]->tcps_state == TCPS_SYN_SENT) ||
 (args[5]->tcps_state == TCPS_LISTEN && args[3]->tcps_state == TCPS_SYN_RECEIVED)/
{
     dur[args[1]->cs_cid] = timestamp;
}

tcp:::state-change
/(args[3]->tcps_state == TCPS_CLOSING ||
  args[3]->tcps_state == TCPS_FIN_WAIT_2 ||
  args[3]->tcps_state == TCPS_LAST_ACK) &&
 dur[args[1]->cs_cid] != 0/
{
    @["Connection duration (ms)"] = quantize((timestamp - dur[args[1]->cs_cid]) / 1000000);
     dur[args[1]->cs_cid] = 0;
}
```

togram of IP payload sizes using a linear distribution. This script does not print anything to the terminal while it is running; rather, it continuously collects data and prints a summary when it exits, which can happen when the user enters Ctrl-C or the script calls the built-in exit() function. In this case, the script runs until the user ends it:

In this case, a sample output on a system with a single active interface (wlan0 in this case) is

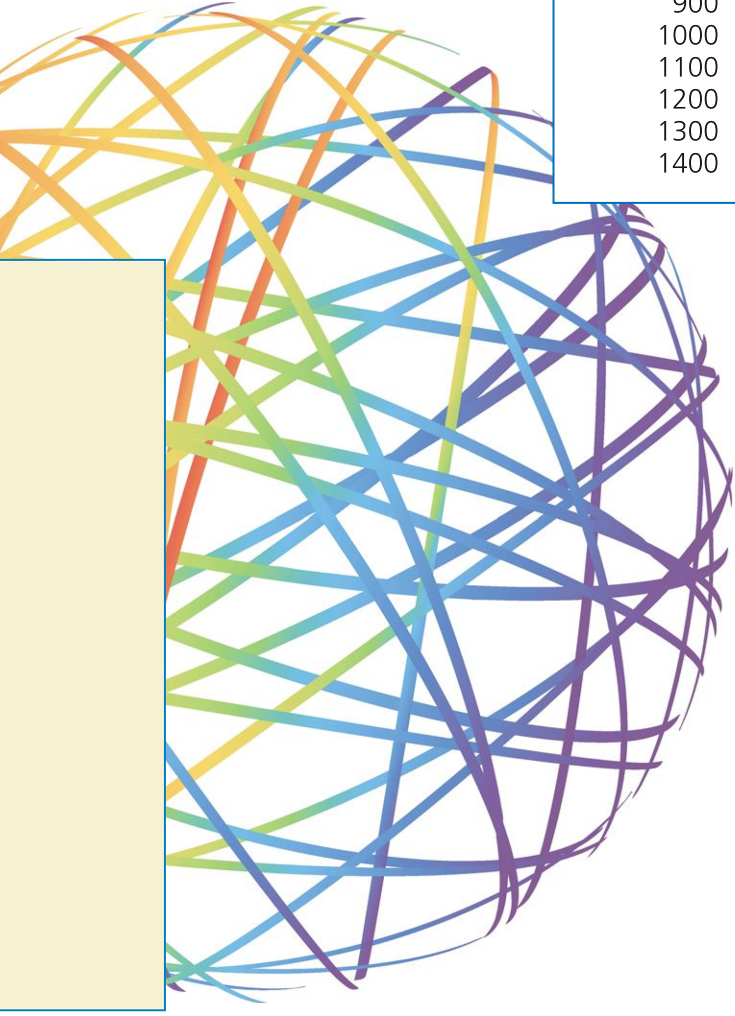
A system with multiple interfaces will print a histogram for each interface.

For another example, we can use the `tcp:::state-change` probe in Example 5 to show the distribution of TCP connection durations. This probe gives us the to and from states when the transitions happen, so we may measure durations by recording a timestamp when either

- a connection transitions from CLOSED to SYN-SENT, or
- a connection transitions from LISTEN to SYN-RECEIVED.

We then consider a connection to have ended once it enters the FIN-WAIT2, CLOSING or LAST-ACK states, once this happens, we may record the time difference between the two events. To store the initial connection timestamps, we use an array indexed by `args[1]->cs_cid`, an opaque integer which uniquely identifies a connection. That is, we can assume that multiple simultaneous connections will not share a connection ID. The full script then looks like this:

```
wlan0
          value------------------- Distribution ------------------- count
            < 0                                                     0
              0   |@@@@@@@@@@@@@@@@@@@@@@@@@                        479
            100   |@@@                                              61
            200   |@                                                16
            300   |@                                                24
            400   |@                                                27
            500   |@                                                14
            600   |                                                 5
            700   |                                                 10
            800   |                                                 5
            900   |                                                 9
           1000   |                                                 5
           1100   |                                                 5
           1200   |@                                                11
           1300   |@@@@@@@@                                         152
           1400   |                                                 0
```



In the `tcp:::state-change` probe `args[5]->tcps_state` gives the from-state, and `args[3]->tcps_state` yields the to-state. Note that the second probe checks whether a timestamp for the connection exists by verifying that `dur[args[1]->cs_cid]` is non-zero: this is to ensure that we do not record data for connections that already exist when the script is started.

The examples presented thus far have hopefully convinced you of the potential of the network providers as building blocks for network tools, whether they are tailored to investigate some specific problem, or to track data that is difficult to obtain using existing monitoring tools. However, most of the examples we have seen so far could in principle be re-implemented with custom programs that use the BPF to intercept and inspect packets. On the other hand, the TCP probes give us a window into the internal state of the associated TCP connection; this is not accessible via BPF. This allows us to, for instance, monitor the TCP state transitions of connections as they occur:

Example 6

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10Hz

dtrace:::BEGIN
{
    printf(" %30s %-6s %30s %-6s %-9s\n", "LADDR", "LPORT",
        "RADDR", "RPORT", "DELTA(us)");
}

int last[uint64_t];

tcp:::state-change
{
    this->delta = last[args[1]->cs_cid] != 0 ?
        (timestamp - last[args[1]->cs_cid]) / 1000 : 0;
    last[args[1]->cs_cid] = timestamp;

    printf(" %30s %-6u %30s %-6u %-9u %s -> %s\n",
        args[3]->tcps_laddr, args[3]->tcps_lport,
        args[3]->tcps_raddr, args[3]->tcps_rport,
        this->delta,
        tcp_state_string[args[5]->tcps_state],
        tcp_state_string[args[3]->tcps_state]);
}

tcp:::state-change
/args[3]->tcps_state == TCPS_CLOSING ||
 args[3]->tcps_state == TCPS_FIN_WAIT_2 ||
 args[3]->tcps_state == TCPS_LAST_ACK/
{
    last[args[1]->cs_cid] = 0;
}
```

The script in Example 6 records a per-connection timestamp of the last state change in the "last" array, indexed by the connection ID stored in **`args[1]->cs_cid`**. Each time a state transition occurs, the 4-tuple associated with the connection is printed, along with the amount of time elapsed since the last state transition and the transition itself, e.g. "**`state-established -> state-close-wait`**".

The **`tcp_state_string`** array is defined in **`/usr/lib/dtrace/tcp.d`** and provides string representations of each of the TCP states. Symbolic names for the states are also available, e.g. **`TCPS_TIME_WAIT, TCPS_LAST_ACK`**.

Note that we clear entries in the "last" array by setting them to 0 when TCP connections end, i.e. when they enter the FIN-WAIT-2, CLOSING or LAST-ACK states. At this point the connection state inside the kernel is about to be torn down, so the array entry will not be valid if the CID is reused for a future connection.

Advanced Tricks

Earlier we mentioned that it was possible to track network activity per process. In the future it will hopefully become possible to access the PID of the associated process through the network probe arguments, but this is currently not possible because of the way that the relevant data is organized within the kernel. However, using the fbt provider it is possible to associate PIDs or process names with the connection IDs available through **`args[1]->cs_cid`** in the TCP and UDP probes. The D code to accomplish this is somewhat opaque; however, it is included in the example below and can be reused in other scripts.

The following script in Example 7 prints a summary of per-process TCP activity every two seconds, reporting the total number of bytes transmitted and received over TCP, as well breakdown by process and 4-tuple:

This script is built up of several pieces. The `dtrace:::BEGIN` probe is used to initialize a pair of global variables used to count the number of TCP payload bytes that were transmitted and received in the current interval; they are reset by the `profile:::tick-2sec` probe, which prints summary data to the terminal every two seconds. The four FBT probes are used to populate and clear the `procs` array, which maps connection IDs to process names (e.g. "firefox" or "sshd"). In particular, the `fbt::tcp_usr_attach` probes fire when a new TCP socket is created, the

Example 7:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
     in = 0;
     out = 0;
}

fbt::tcp_usr_attach:entry
{
     self->so = args[0];
}

fbt::tcp_usr_attach:return
/args[1] == 0 && self->so != NULL/
{
     procs[(uintptr_t)self->so->so_pcb] = execname;
     self->so = NULL;
}

fbt::sosend_generic:entry, fbt::soreceive:entry
/args[0]->so_proto->pr_protocol == IPPROTO_TCP/
{
     procs[(uintptr_t)args[0]->so_pcb] = execname;
}

fbt::in_pcbdetach:entry
{
     procs[(uintptr_t)args[0]] = 0;
}

tcp:::send
/procs[args[1]->cs_cid] != ""/
{
     this->bytes = args[2]->ip_plength - args[4]->tcp_offset;

     out += this->bytes;
     @bytes[procs[args[1]->cs_cid], args[2]->ip_saddr, args[4]->tcp_sport,
         args[2]->ip_daddr, args[4]->tcp_dport] = sum(this->bytes);
}

tcp:::receive
/procs[args[1]->cs_cid] != ""/
{
     this->bytes = args[2]->ip_plength - args[4]->tcp_offset;

     in += this->bytes;
     @bytes[procs[args[1]->cs_cid], args[2]->ip_daddr, args[4]->tcp_dport,
         args[2]->ip_saddr, args[4]->tcp_sport] = sum(this->bytes);
}

profile:::tick-2sec
{
     out /= 1024;
     in /= 1024;
```

(Example 7 continues next page)

(Example 7 continued)

```
        printf("%Y, TCP in: %6dKB, TCP out: %6dKB, TCP total: %6dKB\n", walltimestamp,
             in, out, in + out);

        printf("%-12s %-15s %5s %-15s %5s %9s\n", "PROC", "LADDR", "LPORT",
             "RADDR", "RPORT", "SIZE");

        printa("%-12s %-15s %5d %-15s %5d %@9d\n", @bytes);
        printf("\n");

        trunc(@bytes);
        in = 0;
        out = 0;
}
```

fbt::sosend_generic and **fbt::soreceive** probes fire when a process transmits or receives data over TCP, and the **fbt::in_pcbdetach** probe fires when a TCP connection is closed.

The remainder of the script performs the actual per-process accounting. Each time the TCP stack sends or receives a packet corresponding to an entry in the **procs** array, the **in** and **out** global variables are incremented appropriately, and the bytes array is updated. This array is indexed by process name and the 4-tuple of the connection, and its contents are printed and cleared in the **profile:::tick-2sec** probe. This makes it possible to drill down into the TCP usage of specific processes, a task which is quite difficult without DTrace.

Of course, the script in Example 7 can be modified to perform different types of per-process or per-user accounting. For instance, one could keep running totals rather than clearing statistics every two seconds. When a process exits (signaled by the **proc:::exit** probe) its total TCP usage could be saved for later analysis. Additionally, with a few tweaks, Example 7 can be modified to track UDP usage rather than TCP, as shown in Example 8.

As a final example, we demonstrate how DTrace may be used to dig into some of the more advanced aspects of the TCP protocol.

Example 8:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
     in = 0;
     out = 0;
}

fbt::udp_attach:entry
{
     self->so = args[0];
}

fbt::udp_attach:return
/args[1] == 0 && self->so != NULL/
{
     procs[(uintptr_t)self->so->so_pcb] = execname;
     self->so = NULL;
}

fbt::sosend_dgram:entry, fbt::soreceive:entry
/args[0]->so_proto->pr_protocol == IPPROTO_UDP/
```

(Example 8 continues next page)

(Example 8 continued)

```
{
    procs[(uintptr_t)args[0]->so_pcb] = execname;
}

fbt::in_pcbdetach:entry
{
    procs[(uintptr_t)args[0]] = 0;
}

udp:::send
/procs[args[1]->cs_cid] != ""/
{
    /* Subtract UDP header size. */
    this->bytes = args[4]->udp_length - 8;

    out += this->bytes;
    @bytes[procs[args[1]->cs_cid], args[2]->ip_saddr, args[4]->udp_sport,
        args[2]->ip_daddr, args[4]->udp_dport] = sum(this->bytes);
}

udp:::receive
/procs[args[1]->cs_cid] != ""/
{
    /* Subtract UDP header size. */
    this->bytes = args[4]->udp_length - 8;

    in += this->bytes;
    @bytes[procs[args[1]->cs_cid], args[2]->ip_daddr, args[4]->udp_dport,
        args[2]->ip_saddr, args[4]->udp_sport] = sum(this->bytes);
}

profile:::tick-2sec
{
    out /= 1024;
    in /= 1024;

    printf("%Y, UDP in: %6dKB, UDP out: %6dKB, UDP total: %6dKB\n",
            walltimestamp, in, out, in + out);

    printf("%-12s %-15s %5s %-15s %5s %9s\n",
        "PROC", "LADDR", "LPORT", "RADDR", "RPORT", "SIZE");

    printa("%-12s %-15s %5d %-15s %5d %@9d\n", @bytes);
    printf("\n");

    trunc(@bytes);
    in = 0;
    out = 0;
}
```

Our main tool here is the fourth argument (`args[3]`) passed to each of the TCP probes. This argument contains information which describes the internal state of the connection, and is useful in exploring phenomena that are not easily observed by examining the headers in individual packets.

Here we use the TCP provider to detect the arrival of out-of-order segments; specifically, the script checks for inbound data packets whose sequence numbers do not match the next anticipated sequence number (accessed through `args[3]->tcps_rnxt`). If FreeBSD sees such a segment, it will add it to the connection's reassembly queue if there is space available; otherwise it is dropped. Out-of-

Example 9:

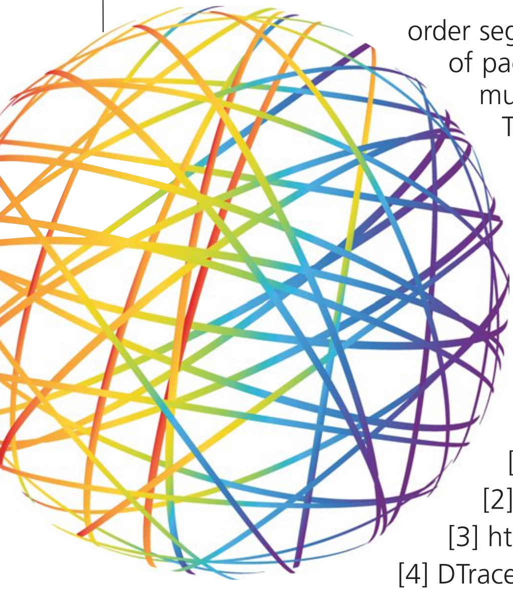
```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10Hz

tcp:::receive
/args[3]->tcps_state == TCPS_ESTABLISHED &&
 args[2]->ip_plength - args[4]->tcp_offset > 0 &&
 args[3]->tcps_rnxt != args[4]->tcp_seq/
{
  @invorderb[args[3]->tcps_raddr] = sum(args[2]->ip_plength - args[4]->tcp_offset);
  @invorderp[args[3]->tcps_raddr] = count();
}

tcp:::receive
/args[3]->tcps_state == TCPS_ESTABLISHED &&
 args[2]->ip_plength - args[4]->tcp_offset > 0/
{
  @valorderb[args[3]->tcps_raddr] = sum(args[2]->ip_plength - args[4]->tcp_offset);
  @valorderp[args[3]->tcps_raddr] = count();
}

dtrace:::END
{
  printf("%-30s %-12s %-12s %-12s %-12s\n", "RADDR", "BYTES",
      "OOO BYTES", "PACKETS", "OOO PACKETS");
  printa("%-30s %@-12d %@-12d %@-12d %@-12d\n", @valorderb, @invorderb,
      @valorderp, @invorderp);
}
```



order segments may be the result of packet drops or of having multiple routes between the TCP endpoints; in general, they hurt throughput and should be investigated if they make up a large ratio of the total number of segments in a connection. The script in Example 9 counts out-of-order segments by remote host address. It also computes the total byte and packet counts for comparison. ●

Further Reading

The purpose of this article was to present an introduction to the network providers available in FreeBSD 10 and to give readers a feel for the sorts of problems that the providers are well-suited to address. A reference for the complete set of probes is available in [1-3], and they are also described in the excellent DTrace book by Brendan Gregg and Jim Mauro[4].

[1] https://wikis.oracle.com/display/DTrace/ip+Provider
[2] https://wikis.oracle.com/display/DTrace/tcp+Provider
[3] https://wikis.oracle.com/display/DTrace/udp+Provider
[4] DTrace: Dynamic Tracing in Oracle Solaris,Mac OS X, and FreeBSD.
[5] https://people.freebsd.org/~markj/dtrace/network-providers/examples/

Mark Johnston is a software engineer living in Waterloo, Ontario. He completed a bachelor's degree in mathematics at the University of Waterloo in 2013 and has been a FreeBSD user since 2010. He is interested in all aspects of operating systems development, with a particular emphasis on debugging and performance analysis utilities. Since obtaining a commit bit, his main focus has been on improving FreeBSD's DTrace implementation. He can be reached via email at markj@FreeBSD.org.