Refreshing netstat output with Perl

NETWORK VIEW

rotwild, photocase.com

The netstat utility reveals how your Linux box interacts with the local network. With a few Perl modules, you can develop a tool that displays the data dynamically, exactly the way *top* does. **BY MICHAEL SCHILLI**

When you want to know which ports are currently being used, you call *netstat*. This practical Linux utility can be run in several modes, which the user controls through command-line options. For example, the *-s* option produces network traffic statistics (Figure 1), and *-put* displays the ports of all applications that are currently communicating over TCP (Figure 2). Both outputs are useful, but what is really interesting is the chronological progression of events rather than a snapshot at a given moment.

Top for the Network

The *top* utility serves as a model for this kind of dynamic output, displaying and continually updating the CPU load, memory usage, and other basic data of currently running processes. Thanks to CPAN, creating a dynamic terminal application like this from the static output from *top* is not really difficult.

The *Curses::UI* module delivers the necessary framework here, providing the dynamic output and making it possible to react to keys pressed by the user. The module's event loop can be merged easily into the kernel of the Perl Object Environment (POE), making it possible to execute many different tasks within a process or a thread.

Don't Freeze!

As with all GUI applications that call other programs, you will encounter a problem. While the external program runs, the calling application no longer reacts to user input and mouse clicks, giving the user the impression that the application is frozen.

The *netstat* command generally finishes quite quickly, but with the *-put* option, it resolves the hostname of the IP address associated with a socket with a reverse DNS lookup. With a slow DNS server or many sockets, this can lead to considerable delays; sometimes it takes several seconds for *netstat* to complete. Delays can be prevented with the *-n* option, but the user must be content with just the IP address. But users want it all: the luxury of name resolution and a fast-reacting GUI at the same time.

Racer Components

The GUI application based on *Curses::UI* works together with the POE kernel, processing key presses and mouse clicks as well as stdout events from other POE sessions.

The *netstat* command should also play well with the POE kernel, which should start the process but not wait for the result. Rather, it immediately should return control to the kernel so that the latter can again refresh the output and

```
mschilli@mybox:/home/mschilli
# netstat -s | head -25
Ip:
    261054 total packets received
    0 forwarded
    0 incoming packets discarded
    261054 incoming packets delivered
    252259 requests sent out
Icmp:
    6787 ICMP messages received
    0 input ICMP message failed.
    ICMP input histogram:
        destination unreachable: 1703
        echo requests: 2079
        echo replies: 3005
    3595 ICMP messages sent
    0 ICMP messages failed
    ICMP output histogram:
        destination unreachable: 1516
        echo replies: 2079
Tcp:
    7962 active connections openings
    3948 passive connection openings
    0 failed connection attempts
    1042 connection resets received
    1 connections established
    227112 segments received
#
```

Figure 1: The netstat -s command delivers statistical data about network traffic managed by Linux.

react to GUI events. If the output from *netstat* finally arrives, the kernel again receives this as an event and calls a function that filters and stores the data into varaibles for subsequent processing.

The *POE::Wheel::Run* module is a component of the POE distribution from CPAN. The module takes care of an external process and changes the state of an automaton in case something happens on the standard output of the process. Other events occur if the process ends successfully or if an error occurs.

Figure 2: The output of the netstat -put command shows a list of all active TCP ports.

Wheels in the Kernel

The wheel, in turn, finds its place in a *POE::Session*, a state automaton, which latches itself into the POE kernel. The kernel ensures that now and then the automaton gets a time slice but must guarantee that all of the registered sessions eventually get a turn. In contrast to a preemptive Linux kernel that sometimes pulls the rug out from under a process, the POE framework relies on the amicable behavior of all sessions.

Each session must immediately return control to the kernel, as soon as it can no longer run at full speed. With this cooperative multitasking, it is important that all tasks that are waiting for something (hard disk, a network event, or output from an external process) play along. An inconsiderate component can paralyze the entire system. Each session has a private data area – the heap – which is implemented as hash and stores key-value pairs with session data.

Autonomous Automatons

The POE world encapsulates autonomous state automatons in so-called components. The application simply loads

Listing 1: PoCoRunner.pm

```
01 package PoCoRunner;
02 use strict;
03 use warnings;
04 use POE::Wheel::Run;
05 use POE;
06
07 ###################################
08 our $PKG = __PACKAGE__;
09
10 ###################################
11 sub new {
12 ###################################
13   my ( $class, %options ) =
14     @_;
15
16   my $self = {%options};
17
18   POE::Session->create(
19     package_states => [
20       $PKG => [
21         qw(_start stdout
22           finish run)
23       ]
24     ],
25     heap =>
26       { self => $self },
27   );
28
29   bless $self, $class;
30 }
31
32 ###################################
33 sub _start {
34 ###################################
35   my ( $kernel, $session ) =
36     @_[ KERNEL, SESSION ];
37   $kernel->post( $session,
38     "run" );
39 }
40
41 ###################################
42 sub run {
43 ###################################
44   my ( $kernel, $heap,
45     $session )
46     = @_[ KERNEL, HEAP,
47     SESSION ];
48
49   my $wheel =
50     POE::Wheel::Run->new(
51     Program => $heap->{self}
52       ->{command},
53     ProgramArgs => [
54       $heap->{self}->{args}
55     ],
56     StdoutEvent => "stdout",
57     ErrorEvent  => "finish",
58     CloseEvent  => "finish",
59     );
60
61   $heap->{"wheel"}  = $wheel;
62   $heap->{"stdout"} = "";
63 }
64
65 ###################################
66 sub stdout {
67 ###################################
68   my ( $input, $heap ) =
69     @_[ ARG0, HEAP ];
70
71   $heap->{stdout} .=
72     "$input\n";
73 }
74
75 ###################################
76 sub finish {
77 ###################################
78   my ( $kernel, $heap ) =
79     @_[ KERNEL, HEAP ];
80
81   ${ $heap->{self}->{data} }
82     = $heap->{stdout};
83
84   $kernel->delay( "run",
85     $heap->{self}
86       ->{interval} );
87 }
88
89 1;
```

these classes and creates new objects as needed, and the POE kernel includes their state machines and runs them autonomously behind the scenes.

The *PoCoRunner.pm* listing (PoCo is the usual abbreviation for POE Component) shows a component that takes the name of a program (*netstat*, for example) with options and creates a "wheel," which then shoots off an external process with the given program (Listing 1). Afterward, the wheel immediately returns control again to the kernel without waiting for the result.

With each line that appears on the standard output of the process, POE receives a stdout event and calls the *PoCoRunner.pm* method *stdout()*. There, the session-specific heap variable *data* (a scalar) collects the process output as text. If *netstat* has terminated, the automaton changes to the method *finish()*, regardless of whether the process has terminated successfully or there was an error. It then copies the collected stdout data into the scalar instance variable *data* of the PoCoRunner object. Because the constructor *new- > ()* has received a reference to it from the calling program, the main script *nettop* ends up with the output data in either *$stats_data* or *$conns_data*.

Subsequently, *finish()*, starting from line 76, calls the kernel method *delay()* and causes it to call the PocoRunner method *run()* with a pre-defined delay, which then resets the heap-variable *data* and again calls the wheel *POE::Wheel::Run* with the specified external program *netstat*.

Thus, if someone links this component into a POE program and calls the constructor with an external command plus command-line parameters, an interval duration, and a reference to a scalar, the component not only starts the external command again and again, but also ensures that the newest and complete output is stored in the scalar.

The PoCoRunner state automaton is defined by calling the *POE::Session*'s *create()* method in line 18.

The states are:

- *_start*: Start state
- *_run*: Fires off the process
- *_stdout*: Process sends a batch of data to stdout
- *_finish*: Process terminates

POE maps these states to the functions of the same name within the module *PoCoRunner.pm*, based on the parameter *package_states* in line 19.

Parameters POE-Style

POE passes session parameters in its own way; for example, if you have an event handler taking arguments like

```
  my($kernel, $heap) =
@_[KERNEL, HEAP];
```

the session passes the handler a set of arguments in the Perl-typical array @_. The handler grabs only two of these through the macros KERNEL and HEAP. These constant functions are imported by POE into the namespace and return integer values, so the construct above represents a so-called array slice, which returns a subset of the parameters in the array as a list.

The *nettop* listing (Listing 2) pulls in two instances of PoCoRunner in lines 12 and 18: one for *netstat -s* and an additional one for *netstat -put*. The output ends up in the scalars *$stats_data* and *$conns_data*.

The function *conns_parse* in line 241 works through the *netstat* output (Figure 2), extracts the important columns (local IP, network IP, status, program), creates an array of arrays from the table format, and returns a reference to it.

On the other hand, *stats_parse* in line 173 analyzes the output from *netstat -s*, as in Figure 1, and stores the output in a hash of hashes. The sub-headers (e.g., "Ip:") become entries in the top-level hash and the labels of the individual

Listing 2: nettop

```
001 #!/usr/bin/perl -w
002 use strict;
003 use Curses::UI::POE;
004 use List::Util qw(max);
005
006 my ( $STATS, $CONNS );
007 my $netstat      = "netstat";
008 my $REFRESH_RATE = 1;
009
010 use PoCoRunner;
011
012 PoCoRunner->new(
013   command => $netstat,
014   args    => "-s",
015   data    => \my $stats_data,
016   interval => 1,
017 );
018 PoCoRunner->new(
019   command => $netstat,
020   args    => "-put",
021   data    => \my $conns_data,
022   interval => 1,
023 );
024
025 my $CUI =
026   Curses::UI::POE->new(
027   -color_support => 1,
028   inline_states  => {
029     _start => sub {
030       $poe_kernel->delay(
031         'wake_up',
032         $REFRESH_RATE
033       );
034     },
035     wake_up =>
036       \&wake_up_handler,
037     chld => sub {
038       waitpid $_[ARG1], 0;
039     },
040   }
041   );
042
043 my $WIN =
044   $CUI->add(
045   qw( win_id Window ));
046
047 my $TOP = $WIN->add(
048   qw( top Label
049     -y 0 -width -1
050     -paddingspaces 1
051     -fg white -bg blue
052     ), -text => top_text()
053 );
054
055 my $LBOX = $WIN->add(
056   qw( lb Listbox
057     -padtop 1 -padbottom 1
058     -border 1 ),
059 );
060
061 my $BOTTOM = $WIN->add(
062   qw( bottom Label
063     -y -1 -width -1
064     -paddingspaces 1
065     -fg white -bg blue
066     ),
067   -text => "TCP Watcher v1.0"
068 );
069
070 $CUI->set_binding(
071   sub { exit 0; }, "q" );
072 $poe_kernel->sig( "CHLD",
073   "chld" );
074 $CUI->mainloop;
075
076 ###################################
077 sub wake_up_handler {
078 ###################################
079   # Re-enable timer
080   $poe_kernel->delay(
081     'wake_up',
082     $REFRESH_RATE
083   );
084   data_refresh();
085   $TOP->text( top_text() );
086   $TOP->draw();
087
088   my $state_fmt = col_fmt(
089     [
090       map $_->{state},
091       @$CONNS
092     ],
093     8
094   );
095   my $prog_fmt = col_fmt(
096     [
097       map $_->{prog}, @$CONNS
098     ],
099     20
100   );
101   my $rem_fmt = col_fmt(
102     [
103       map $_->{remote},
104       @$CONNS
105     ],
106     32
107   );
108   my $loc_fmt = col_fmt(
109     [
110       map $_->{local},
111       @$CONNS
112     ],
113     20
114   );
115
116   my @lines = map {
117     $state_fmt->(
118       $_->{state} )
119       . " "
120       . $prog_fmt->(
121       $_->{prog} )
122       . " "
123       . $rem_fmt->(
124       $_->{remote} )
125       . " "
126       . $loc_fmt->(
127       $_->{local} )
128       . " " . "";
129   } sort conn_sort @$CONNS;
130
131   $LBOX->{ -values } =
132     [@lines];
133   $LBOX->{-labels} =
134     { map { $_ => $_ }
135       @lines };
```

values (e.g., "incoming packets delivered") end up as keys in the subordinate hash. The values stored under it correspond to the column of numbers as read from the *netstat* output. Altogether, *stats_parse()* uses three different regular expressions in order to grab the intermediate headings, as well as two different line output formats from *netstat*.

Out of all of the connections, the ones with the status *ESTABLISHED* are often the most interesting, so the sort routine *conn_sort()* in line 206 sorts them to the top. As usual in Perl, the sort function called in line 129 is passed the comparison function as a parameter. With each comparison in the sorting process, sort then calls *conn_sort()* and fills the spe-

Listing 2: nettop

```
136
137   $LBOX->draw(1);
138 }
139
140 ################################
141 sub top_text {
142 ################################
143   my $ip  = $STATS->{Ip};
144   my $tcp = $STATS->{Tcp};
145
146   return sprintf
147       "Packets rcvd:%s " .
148       "sent:%s TCPopen "
149     . "active:%s passive:%s",
150     $ip->{
151     'total packets received'
152     },
153     $ip->{
154     'requests sent out'},
155     $tcp->{
156 'active connections openings'
157     },
158     $tcp->{
159 'passive connection openings'
160     };
161 }
162
163 ################################
164 sub data_refresh {
165 ################################
166   $STATS =
167     stats_parse($stats_data);
168   $CONNS =
169     conns_parse($conns_data);
170 }
171
172 ################################
173 sub stats_parse {
174 ################################
175   my ($output) = @_;
176
177   my $section;
178   my $data = {};
179   my $key  = qr/\w[\w\s]+/;
180
181   for ( split /\n/, $output )
182   {
183     if (/($key):$/) {
184       $section = $1;
185       next;
186     }
187     elsif (/($key): (\d+)/) {
188       $data->{$section}
189         ->{$1} = $2;
190     }
191     elsif (/(\d+)\s+($key)/)
192     {
193       $data->{$section}
194         ->{$2} = $1;
195     }
196     else {
197       die
198      "Can't parse line '$_'";
199     }
200   }
201
202   return $data;
203 }
204
205 ################################
206 sub conn_sort {
207 ################################
208   return -1
209     if $a->{state} eq
210     "ESTABLISHED";
211   return 1
212     if $b->{state} eq
213     "ESTABLISHED";
214   return 0;
215 }
216
217 ################################
218 sub col_fmt {
219 ################################
220   my ( $cols, $max_space ) =
221     @_;
222
223   my $max_len =
224     max map { length $_ }
225     @$cols;
226   $max_len = $max_space
227     if $max_len > $max_space;
228
229   return sub {
230     return sprintf(
231       "%${max_len}s",
232       substr(
233         shift, 0,
234         $max_len
235       )
236     );
237   };
238 }
239
240 ################################
241 sub conns_parse {
242 ################################
243   my ($output) = @_;
244
245   my $data = [];
246
247   for ( split /\n/, $output )
248   {
249     my (
250       $proto,  $rec,
251       $snd,    $local,
252       $remote, $state,
253       $prog
254       )
255       = split ' ', $_;
256
257     next if $proto ne "tcp";
258     push @$data,
259       {
260       local  => $local,
261       remote => $remote,
262       state  => $state,
263       prog   => $prog
264       };
265   }
266
267   return $data;
268 }
```

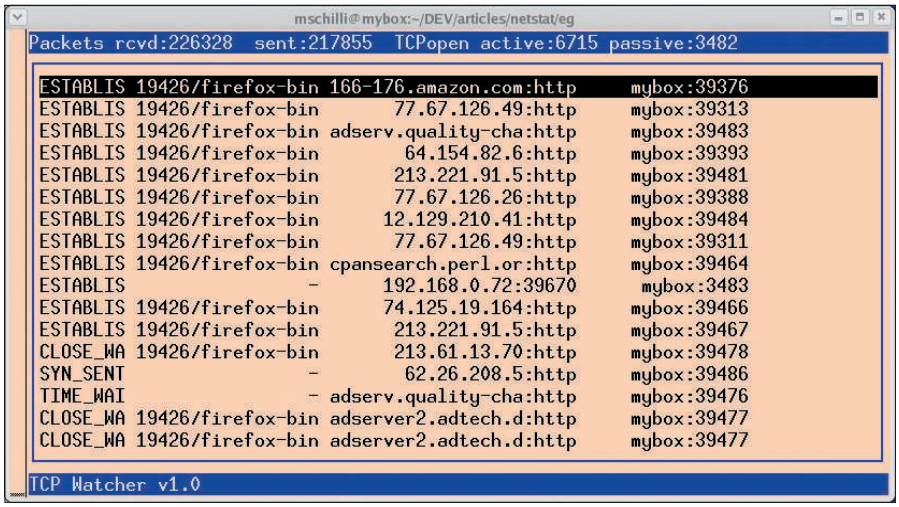

Figure 3: The monitor written in Perl continually displays all currently running TCP connections.

cial variables *$a* and *$b* with two values to be sorted. If *conn_sort* returns *-1*, then *$a* is "smaller" than *$b* and thus migrates up. On the other hand, if *+1* is returned, *$b* moves up. If neither of these candidates is in the status *ESTABLISHED*, *conn_sort* returns the value *0*. Thus, both candiates end up in the display somewhere below the *ESTABLISHED* section.

Formatter as Return Value

To create a good-looking column layout, the table-like values that appear in the GUI often must be cut. The function *col_fmt* takes two parameters: one is a reference to an array of all lines of a table column, and one is the maximum available width for this column, *$max_space*. With the function *max* from the CPAN *Module List::Util*, it calculates the longest line. If this is smaller than *$max_space*, this is the defined width of the column; otherwise, it is defined by *$max_space*.

The formatter, which is returned as a code reference, receives the lines of a column and cuts them down to the maximum width with *substr()*. If they are too short, it fills them with blanks, using *sprintf()* if necessary. Each column receives its own formatter, which makes a total of four formatters doing their jobs in *nettop*. The sum of the values for the maximum width of all columns is 80, often the width of the text window. If a column is clearly smaller than the maximum space reserved for it, the formatter gives this up to other columns. Thus, the display can move somewhat erratically with varying network traffic, but the allocation of space is optimized.

On Screen!

The module *Curses::UI::POE* does the graphic output (Figure 2). The display consists of three parts: a blue header, *$TOP*, with the statistical data from *netstat -s*; a listbox, *$LBOX*, with the current network connections (if necessary, excess entries are hidden); and a blue footer, *$BOTTOM*, that only shows the version of the program.

The parameter *paddingspaces* is set to true and therefore has *Curses::UI* fill up the right side of the blue header and footer lines. This way, the blue bars extend all the way across the screen and do not vary with the actual length of the displayed text. The method *set_binding()* in line 70 specifies that when the *q* key is pressed, the kernel terminates the program because it calls a function that executes *exit 0* when this event occurs.

Automaton Gets States

The finite automaton of the presentation layer knows two states: the starting condition *_start* and the wake up condition *wake_up*, in which the automaton updates the screen with the newest data. Instead of *package_states*, the parameter *inline_states* comes into play in *nettop* because the constructor of the POE session assigns the state names directly to anonymous subroutines and does not refer implicitely to identically named functions in the same module. Although *wake_up* still runs, it sends an event off to the kernel with *delay()*, which makes sure that the kernel will re-execute it after the number of seconds set in *$REFRESH_RATE*. Thus, an endless loop is created, which continuously updates the terminal at regular intervals.

In *wake_up*, the call to *data_refresh()* first gets the the newest data from the *netstat* process that is running via *PoCoRunner.pm* and then squeezes it into data structures pointed to by globally defined variables.

The function *top_text()* formats the dynamically updated text in the header bar and delivers it to the method *text()* of the header bar object. So that the whole thing appears on the screen, the subsequent call to the method *draw()* is necessary.

Something similar applies for the listbox, whose entries have values (*-values*) that are not of interest in this case because the user isn't going to select any list elements. On the other hand, the parameter *-labels* defines what is shown in the Curses window for each element of the list and *nettop* simply sets these labels likewise to the values already defined for the listbox entries.

Smooth Square Dance

In line 74, the *mainloop* of the graphical interface starts the POE kernel with all of the components that have been loaded so far. The smooth square dance begins and each second (adjusted by *$REFRESH_RATE*) the display receives an update of the data. This does not necessarily mean that new *netstat* is available. No race conditions arise in an enviroment in which only one thread is active, and it is guaranteed that the two scalars *$stats_data* and *$conns_data* filled by the POE components always contain the complete data from the last successful call to *netstat*.

With additional keyboard input, you can extend this script, which could split the screen and show additional details for a process currently selected in the listbox. Naturally, instead of *netstat*, the output of different utilities can be displayed in the same fashion in a *top*-like dynamically updated window. ■

INFO

[1] Listings for this article: *http://www.linuxpromagazine.com/resources/article_code*