Network Sniffer with GTK2 GUI

Traffic Control

The Glade GUI Builder brings the power of drag and drop to GUI building. Scripts parse the XML-formatted description at runtime. As an example, we will be looking at a network sniffer with a neat interface.

BY MICHAEL SCHILLI

www.sxc.hu

If you need to know who is currently logged on to your local network and prefer a GTK2 interface to view this information, this month's script, *capture.pl*, is just what the doctor ordered. It uses a CPAN module called *Net::Pcap* (see also [1]) to sniff traffic off the wire or on wireless networks, decodes the captured packets, determines if the sender is on the local network, and displays the IP addresses it has identified in a text view widget (see Figure 1).

The script stores its latest findings at the top of the list and dynamically updates the list. The *File* menu has a *Reset* entry that allows users to delete previously discovered addresses from the list, and there is a *Quit* entry to quit the program.

Figure 1: The GTK2 program *capture.pl* displays all active computers on the LAN in a list.

XML-based GUI

The script does not use fixed statements to define the GUI, but parses an XML description at runtime. Programmers can use the Glade 2 tool from [2] to create the file. After parsing the definition, *capture.pl* goes on to set up the GUI and handle incoming events.

Most GUI builders use a different approach. Developers can use drag and drop to place widgets and define events, but the builder will then convert the results to code, leaving the fine tuning to the developer. Unfortunately, GUI builders are typically incapable of reading the code after manual editing.

Glade can handle both approaches; it either generates C code or an XML definition, which can be parsed by a program that uses the Libglade library. *Gtk2::GladeXML* from CPAN gives developers a Perl wrapper.

Figure 2 shows Glade in action. The main window is shown top left. Here, a user is creating a new project. The toolbar at the bottom left contains a collection of widgets; the finished application is shown in the center. The window top right handles the widget attributes, such as the name, size, editing capabilities, and the signals it handles. The window bottom right is the widget tree window showing a hierarchical view of the widgets available to the application.

To create a new GUI description, click the main window icon in the toolbar (the icon with the blue stripes top left). This opens an empty application window, as shown in Figure 3. A new Vbox container creates the menu bar at the top and the text field at its bottom. To add more widgets, click the field in the toolbar, and then click the application window.

We are still missing a menu, a scrolled window, and the text view widget. Figure 4 shows the application window just before completion. There are a lot of unused menu entries for a simple application such as *capture.pl*, but clicking on *Edit Menus* in the Properties window pops up a dialog where we can easily get rid of all but the most important ones (see Figure 5).

Any other modifications you may need are just as simple. For example, the length and width of the main window in the *capture.pl* GUI have been set to 300 and 120 respectively via Properties.

The next step is to click on the *Save* button in the main Glade window and type the project name, *capture*, to store two files: *capture.glade* and *capture.pglade*. We do not need the second file,

but the first file contains the XML definition of the GUI.

The *capture.pl* script parses this description in line 27 when we call the *Gtk2::GladeXML* constructor. The XML file contains individual widget definitions, their position relative to the GUI, and the attributes. For example, the XML description of the text view widget defines the following properties:

```
<property name=
"editable">False</property>
<propertyname="cursor_visible">
</property>
```

In our example, the developer has used Glade's point and click abilities to create a non-editable widget with an invisible text cursor. The following two lines of code would have the same effect:

```
$text->set_editable(0);
$text->set_cursor_visible(0);
```

Signals

The *signal_autoconnect_all* method in line 56 defines the dynamic part of the statically defined GUI. It links the widgets in the XML description with associated signals, such as *on_quit1_activate* (*File* | *Quit* menu entry selected) and *on_reset1_activate* (*File* | *Reset* entry selected) with corresponding Perl functions.

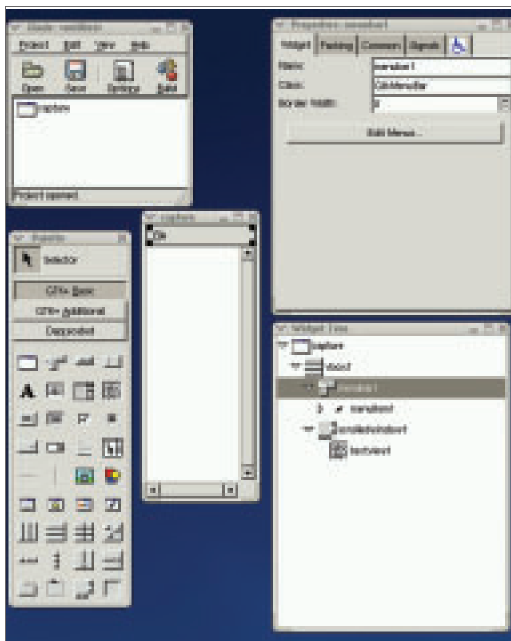

Figure 2: Developers can use Glade for convenient GUI building. The results are stored in an XML definition file.

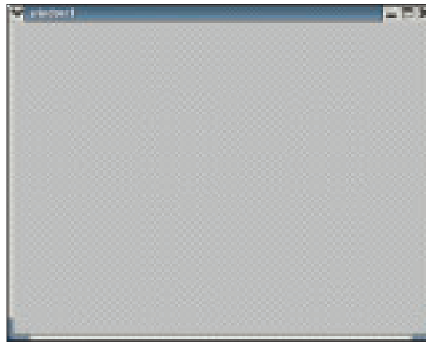

Figure 3: The empty main window in Glade, which we will be adding widgets to later.

The names were automatically assigned by Glade (see Figure 5). If you prefer, you can change them by editing Glade's property fields. In line 68 *capture.pl* enters the *main()* loop; if everything works out, the GUI should appear on screen, allowing users to click to their hearts' desire.

Smooth Scrolling

Network sniffing requires some CPU power, and that means that the process won't be able to handle the GUI while actively capturing. Sluggish or downright frozen GUIs are clearly unacceptable, though. To promptly respond to every occurring user event while using the Perl *Net::Pcap* module to sniff the wire, *capture.pl fork()*s a child process in line 35. The parent keeps the GUI in shape while the child busily examines network packets.

Prior to the *fork()*, we called *pipe()* to create a pipe between the child and parent processes. When the child discovers a new IP address, it uses the *WRITEHANDLE* to send a string through the pipe to its parent, the GUI manager, which uses the *READHANDLE* to pick up the message at the other end of the pipe.

To allow the GUI to ignore the pipe until something turns up, and to handle user input, line 76 has a watch

```
Glib::IO->add_watch
( fileno(READHANDLE), 'in',
\&watch_callback);
```

which calls the callback function defined in line 84 (*watch_callback*) whenever data arrive at the *READHANDLE*. GTK2 is based on the Glib library and is thus capable of accessing the low level services the library provides. *add_watch()* expects a file descriptor, rather than a file handle, so we need a call to the Perl *fileno()* function to convert the *READHANDLE*.

In the Sniffer's Office

Net::Pcap from CPAN is an interface to the libpcap library. The library grabs packets off the network and analyzes and filters these packets at high speeds according to previously defined criteria. Programs such as Ethereal are based on libpcap.

The *snooper()* function in line 107 first attempts to read from the first active network interface on the current host (typically *eth0*) using *Net::Pcap::lookupdev*. The call to *Net::Pcap::lookupnet* then identifies the corresponding network address and mask.

Net::Pcap::open_live() in line 125 opens a live capture and grabs up to 1024 bytes per packet for analysis purposes. As the third parameter is a *1*, the function enables promiscuous mode for the card, that is, it tells the card to grab any packets it sees, and not just the packets destined for its own address. The fourth parameter, *-1*, says that we do not need a timeout. (If we did, this setting would be a value in milliseconds.)

The *Net::Pcap::loop* function in 128 jumps into a loop that executes the appropriate callback function, *snooper_callback()*, whenever it finds a packet. The second parameter, *-1*, tells the program to keep on sniffing indefinitely, rather than stopping after reaching a certain number of packets.

The last parameter in the call to *Net::Pcap::loop* is a reference to an array

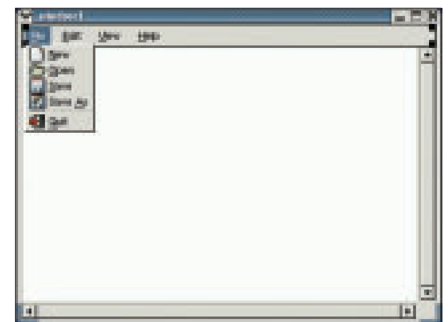

Figure 4: The GUI is almost done. We just need to get rid of the superfluous menu entries.

Listing 1: capture.pl

```
001 #!/usr/bin/perl
002 ################################
003 # capture -- Gtk2 GUI
004 #        observing the network
005 # Mike Schilli, 2004
006 # (m@perlmeister.com)
007 ################################
008 use warnings;
009 use strict;
010
011 use Gtk2 -init;
012 use Gtk2::GladeXML;
013 use Glib;
014 use Net::Pcap;
015 use NetPacket::IP;
016 use NetPacket::Ethernet;
017 use Socket;
018
019 our @IPS = ();
020 our %IPS = ();
021
022 die "You need to be root.\n"
023   if $> != 0;
024
025 # Load GUI XML description
026 my $g =
027   Gtk2::GladeXML->new(
028     'capture.glade');
029
030 # Child/Parent comm pipe
031 pipe READHANDLE, WRITEHANDLE
032   or die "Cannot open pipe";
033
034 # Fork off a child
035 our $pid = fork();
036 die "failed to fork"
037   unless defined $pid;
038
039 if ($pid == 0) {
040        # Child, never returns
041     snooper(\*WRITEHANDLE);
042 }
043
044   # Parent, init text window
045 my $buf =
046   Gtk2::TextBuffer->new();
047
048 $buf->set_text(
049     "No activity yet.\n");
050
051 my $text = $g->get_widget(
052     'textview1');
053
054 $text->set_buffer($buf);
055
056 $g->signal_autoconnect_all(
057     on_quit1_activate =>
058       sub {
059
060         # Stop snooper
061         kill('KILL', $pid);
062         wait();
063         Gtk2->main_quit;
064       },
065
066     on_reset1_activate =>
067       sub {
068
069         # Reset display
070         @IPS = ();
071         %IPS = ();
072         $buf->set_text("");
073       },
074 );
075
076 Glib::IO->add_watch(
077     fileno(READHANDLE), 'in',
078     \&watch_callback);
079
080 # Enter main loop
081 Gtk2->main();
082
083 ################################
084 sub watch_callback {
085 ################################
086   chomp(my $ip =
087         <READHANDLE>);
088
089   # Register IP if unknown
090   unshift @IPS, $ip unless
091       exists $IPS{$ip};
092
093   $IPS{$ip}++;
094
095   my $text = "";
096
097   $text .= "$_\n" for @IPS;
098
099   $buf->set_text($text);
100
101   # Return true to
102   # keep watch
103   1;
104 }
105
106 ################################
107 sub snooper {
108 ################################
109   my($fd) = @_;
110
111   my($err, $addr, $netmask);
112
113   my $dev =
114     Net::Pcap::lookupdev(
115        \$err);
116
117   if(Net::Pcap::lookupnet(
118          $dev, \$addr,
119          \$netmask, \$err)) {
120     die "lookupnet on " .
121          "$dev failed";
122   }
123
124   my $object =
125     Net::Pcap::open_live(
126       $dev, 1024, 1, -1,
127       \$err );
128   Net::Pcap::loop(
129     $object, -1,
130     \&snooper_callback,
131     [$fd, $addr, $netmask]
132   );
133 }
134
135 ################################
136 sub snooper_callback {
137 ################################
138   my($user_data, $header,
139       $packet) = @_;
140
141   my($fd, $addr,
142      $netmask) = @$user_data;
143
144   my $edata =
145    NetPacket::Ethernet::strip
146     ($packet);
147
148   my $ip =
149     NetPacket::IP->decode(
150       $edata);
151
152   if((inet_aton(
153          $ip->{src_ip}) &
154       pack('N', $netmask)) eq
155          pack('N', $addr)){
156     syswrite($fd,
157          "$ip->{src_ip}\n");
158   }
159 }
```

Article continued on p72.

of useful data, which is passed to *snooper_callback()* as the first parameter each time it executes. The array contains three values: *[$fd, $addr, $netmask]*: a file descriptor *$fd*, which will be sent through the pipe to the parent process, and the previously identified network address and mask.

Packet Analysis

Net::Pcap::loop at line 128 jumps into a neverending loop, which calls *snooper_callback()* for every packet captured, passing header and content information. Within *snooper_callback()*, *NetPacket::Ethernet::strip* extracts the Ethernet information from the packet; *NetPacket::IP->decode()* tackles the IP layer and returns a reference to a hash, which stores the source IP address in *src_ip*.

inet_aton() from the *Socket* module converts this “AA.BB.CC.DD” formatted string to a binary format in network byte order. The previously identified values for the network address (*$addr*) and mask (*$netmask*) are stored in the processor’s native binary format (Big or Little Endian). The call to *pack* in line 154 converts them to the machine-independent network format.

capture.pl then goes on to check if the IP address *$ip* belongs to the *$network_addr* network, verifying if *($ip & $mask)* matches the network address in *$network_addr*. This condition is fulfilled for packets originating from the local subnet. *syswrite()* in line 156 sends the IP address string to the parent process without buffering.

The message crosses the pipe, which we defined in line 31, and causes an event (thanks to the watch defined in line 76) which calls *watch_callback()* in the parent process. The global array, which contains all known IP addresses, unsurprisingly called *@IPS*, is updated. Newly identified IPs are not yet stored in the *%IPS* hash, and *unshift()* sends them to the start of an array, unsurprisingly named *@IPS*, which determines the display order. Line 81 puts together a text string containing all the IP addresses known to the script, separated by newlines. And line 99 uses the string to update the text view widget.

Installation

As it uses the GTK2 GUI, the script needs a whole bunch of modules. Here are some of the most important ones: *ExtUtils::Depends*, *ExtUtils::PkgConfig*, *Glib*, *Gtk2*, *Gtk2::GladeXML*, *Net::Pcap*, and *NetPacket*. It is easiest to use a CPAN shell for the install, but some manual modifications are needed at times. If libglade is not installed on your machine, surf to [3] and download the library. The Glade program is available from [2].

While installing *Net::Pcap* make sure that you run the test phase (*make test*) as root, even if the installation itself does not need root privileges. If you still see an error message, try calling *make install* in the build directory.

Before launching *capture.pl,* users need to ensure that the XML GUI definition really is stored in *capture.glade*. If you want to keep everything under one roof, you might prefer to modify line 27 as follows:

```
my $xml = join "\n", <DATA>;
my $g = Gtk2::GladeXML->⮐
new_from_buffer($xml);
```

Then copy the XML content from *capture.glade* to a *DATA* section at the end of the *capture.pl* script:

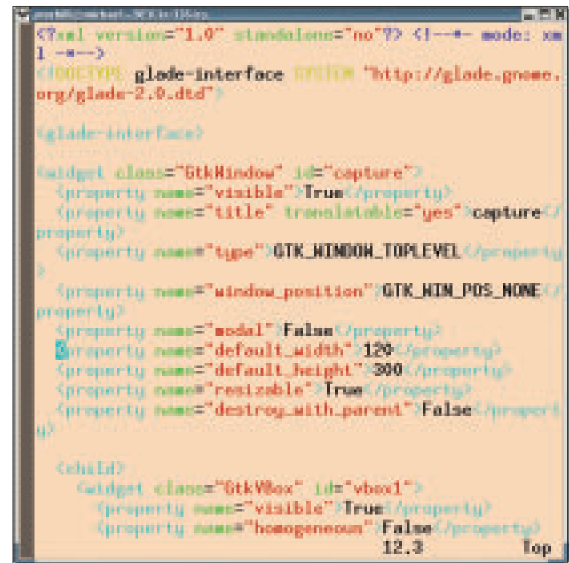

Figure 6: The XML definition of the GUI in the *capture.glade* file created by Glade.

```
# ... End of capture.pl
__DATA__
<?xml version="1.0" ...
<!DOCTYPE glade-interface ...
```

Because the script will switch your network card to promiscuous mode, you need to run *capture.pl* with root privileges.

Glade gives you the power to create far more complex GUIs. The platform-independent XML representation is elegant and removes the need for bulky, static widget definitions in the code, allowing developers to focus on the more important dynamic aspects of the software . ■

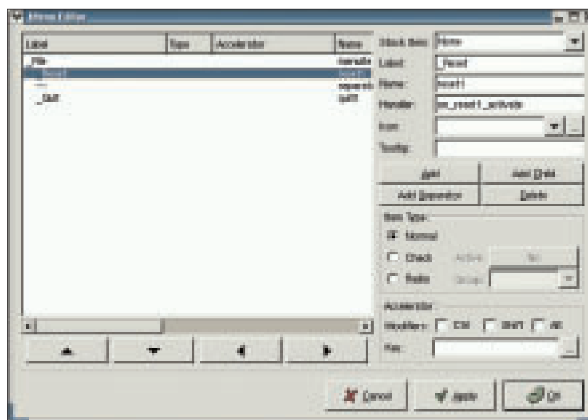

Figure 5: The menu editor makes editing menu widgets simple.

INFO

[1] Robert Casey: “Monitoring Network Traffic with Net::Pcap”, The Perl Journal 7/2004, page 6 ff.

[2] Glade homepage: *http://glade.gnome.org*

[3] Sources for libglade: *http://ftp.gnome.org/pub/GNOME/sources/libglade*

THE AUTHOR

Michael Schilli works as a Software Developer at Yahoo!, Sunnyvale, California. He wrote “Perl Power” for Addison-Wesley and can be contacted at mschilli@perlmeister.com. His homepage is at http://perlmeister.com.