Gtk-GUIs and Web requests play along in the Perl Object Environment

Winning Team Player

The Perl Object Environment (POE) provides a platform for scripts to perform cooperative multitasking without any help from the operating system's scheduler. This month's application lets a Gtk-driven graphical interface interact with time-consuming web requests without hiccups. BY MICHAEL SCHILLI

GUI-based applications are usually event driven. The program will typically have a main loop that it uses to wait for events such as mouse clicks and keyboard input. It is important for the program to process these events without any delay and quickly return to the main event loop. This prevents the user from noticing the temporary unavailability of the interface.

The stock price ticker program we will be looking at in this month's article, periodically connects to the Yahoo financial pages to request an update for selected share prices (see Figure 1). Depending on the network connection, a request including DNS resolution of the server name may take a few seconds to complete. I would like the application's display to keep on working during this period.

Developers can use multiprocessing or multithreading to achieve this aim. However, both techniques make the program far more complex. Critical sections need to be protected against parallel access to ensure data integrity, thus avoiding errors that might be difficult to debug. If you have ever needed to analyze a core dump with 200 active threads, you will know what I mean.

There is an alternative that avoids both these drawbacks – cooperative multitasking with POE, the Perl Object Environment [2] centered around its main developer, Rocco Caputo. The environment is implemented as a state machine that runs exactly one process with a single thread, but has a userspace "kernel" that allows multiple tasks to be performed quasi simultaneously.

Keeping Track

Developers who want to query share prices in Perl will typically opt for the CPAN module Yahoo::FinanceQuote:

```
use Finance::YahooQuote;
my @quote = ➥
getonequote($symbol);
```

Unfortunately, the module works synchronously, and this makes it difficult to achieve the smooth scrolling effect we were looking for. The *getonequote* function sends a HTTP request to the Yahoo server, waits for a response and then returns the results.

We want to keep the display running while the program is waiting – and Murphy's law states that someone will drag another window over the ticker precisely at this point. In this case the application has to redraw the concealed area (this is known as refreshing).

Unfortunately, because it is busy waiting for HTTP results to trickle in, it doesn't receive the redraw event, and this leaves a nasty gray hole on the desktop – not a pretty sight.

Asynchronous Approach

It would be more elegant to transmit a Web request, and get back to refreshing the graphical display immediately, without waiting for the results. When a response from the Yahoo server finally arrives, it should cause some kind of alert. That would mean quickly updating the ticker window and jumping back into the main GUI loop.

This is exactly what the POE framework does. It provides a kernel where individual applications register sessions, and state machines move between states and exchange messages. I/O activity is asynchronous. Instead of opening a file or a socket and waiting for its data, you simply say "Hey Kernel, I want to read some information. Can you wake me up when it becomes available?"

Data in the Fast Lane

Although read and write operations are not actually asynchronous (under the hood, POE simply uses the non-blocking syswrite or sysread functions), any available information is drawn in or pushed out at top speed.

The cooperative aspect of POE is the fact that the sessions rely on competing sessions not to dilly-dally. If a task does not totally preoccupy the CPU, the session has to hand control back to the kernel. A single uncooperative part in a program would impact the whole system.

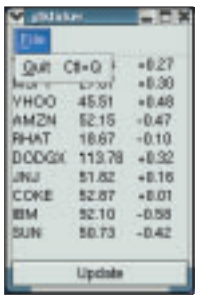

Figure 1: The GTK-based share price ticker periodically contacts the Yahoo financial page.

Multitasking with a single thread facilitates program development – you don't need to worry about locks, there are no surprises with race conditions, and even if an error occurs, it is typically easy to locate. POE will cooperate with the main event loops of several graphical environments. POE recognizes both Perl/Tk and gtkperl automatically and integrates them seamlessly.

This allows the kernel to assign GUI events time slots just like the explicitly defined sessions. This is the answer to the refresh problem.

Alerting Kernel

The ticker in Listing 2 uses the POE::Session state machine as shown in Figure 2. The initialization phase, *_start*, generates the GTK interface and sets the alias name to *ticker* to allow us to more easily identify the session later. Control is returned to the kernel following this initial state. The state machine enters the *wake_up* state every 60 seconds (via an alert), or when someone clicks the *Update* button in the GUI. This launches another *POE::Component::Client::HTTP* type state machine, and then immediately hands control back to the kernel.

Listing 1: Configuration file

```
01 # ~/.gtkticker
02 TWX
03 MSFT
04 YHOO AMZN RHAT
05 DODGX
06 JNJ COKE IBM SUN
```

PoCoCli::HTTP is a so-called component of the POE framework: a state machine that defines its own session (named *useragent* in Listing 2, line 73), accepts Web requests in its *request* state, and then enters the POE framework until it receives a full HTTP response. At this point, *useragent* asks the kernel to tell the calling session, *ticker*, to enter a state called *yhoo_response* that was passed to *useragent* previously.

The kernel tells the *ticker* session to do exactly that, and the session accepts the HTTP response, which is waiting for the session, refreshes the share price widgets in the display, before handing control back to the kernel without any delay. In line 69, the *POE::Component::Client::HTTP* component launches with *spawn()* and specifies that *gtkticker/0.01* should appear in the server-side *User Agent* string, and that requests should time out after 60 seconds.

Yahoo Manual

Lines 10 to 12 in Listing 2 define the URL for Yahoos share price service. The CGI interface of this service expects two parameters:

- A format parameter (*f=*) with the names of requested fields: *s* (symbol), *l1* (share price) and *c1* (percentage change since the last stock exchange working day) and
- a symbol parameter that contains a comma-separated list of the stock exchange abbreviations for the listed companies we are interested in, for example *YHOO,MSFT,TWX*.

The Yahoo server responds as follows:

```
"YHOO",45.38,+0.35
"MSFT",27.56,+0.19
"TWX",18.21,+0.75
```

Gtkticker accepts the response line by line, uses the commas to split the strings, and bundles the information off to the GUI display.

Home Directory Configuration

Lines 13 and 14 specify *.gtkicker* in the user's home directory as the symbol file to be displayed by the ticker. Lines 30 through 37 parse the file line by line, discarding any lines that start with # as comments (line 33). The implicit for loop at the end of line 35

```
... for /(\S+)/g;
```

executes the expression to its left for all the words in a line, and leaves the stock exchange symbol in the *$_* variable. This allows multiple symbols separated by space characters to exist in a single line. The push function stacks the stock symbols in the *@SYMBOLS* array. Listing 1 shows a sample file.

Despite using the POE framework, gtkticker employs regular synchronous I/O functions to read the configuration file, as the file is short and the POE kernel is not running at this point.

Let the Dance Begin

Line 39 defines the ticker state machine. The *inline_states* parameter uses a hash

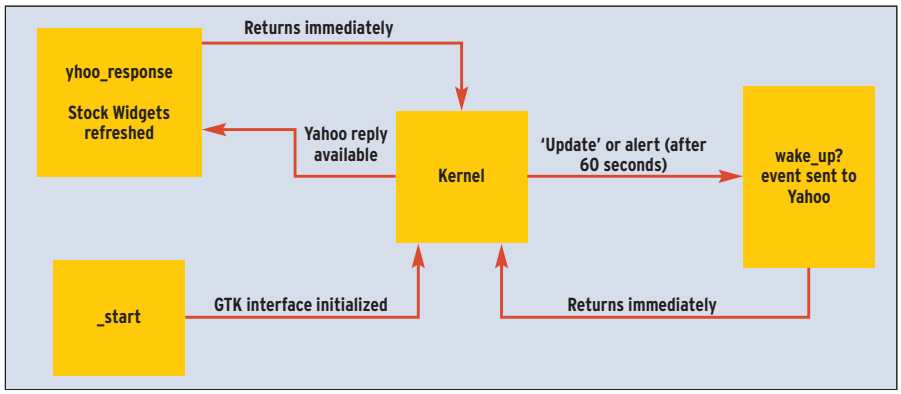

Figure 2: The gtkticker state machine. After the initialization state, *_start*, control is handed back to the kernel. The machine enters the *wake_up* state every 60 seconds, wakes up another state machine to perform an HTTP request and hands control back to the kernel.

reference to map functions to the states. The kernel will jump to these functions if and when the machine enters the corresponding state. Line 53 then pushes the *wake_up* state for the *ticker* session to the kernel

```
$poe_kernel->post("ticker", ➥
"wake_up");
```

via the variable *$poe_kernel* exported by *POE*. Line 55 launches the main kernel loop

```
$poe_kernel->run();
```

in which the program remains until it is shut down. That's it!

The *POE::Session* object construction shown previously had a side effect. It ran the *start()* routine defined in line 58, which maps to the *_start* state. The latter sets the alias name for the session to *ticker* and then jumps to *my_gtk_init()*. This function, which starts in line 98, constructs the GTK GUI.

GUIs with GTK

Gtk is a CPAN module by Marc Lehmann, who was so kind as to check the draft for this article. The module has actually been replaced by *Gtk2*, but the new version has some issues with POE. Never mind, the venerable GTK module does a fantastic job.

An object of the *Gtk::Window* class represents the main window of the application. It has a typical menu bar at the top, giving access to a *File* pull-down menus, which in turn has a single entry for *Quit*. This entry uses a callback routine, *Gtk->exit(0)*, to terminate the application. There is a *Gtk::AccelGroup* object that allows the user to press [Ctrl]+[Q] to quit the program. The object adds

Listing 2: Gtkticker

```
001 #!/usr/bin/perl
002 ##################################
003 # gtkticker
004 # Mike Schilli, 2004
005 # (m@perlmeister.com)
006 ##################################
007 use warnings;
008 use strict;
009
010 my $YHOO_URL =
011  "http://quote.yahoo.com/d?".
012  "f=sl1c1&s=";
013 my $RCFILE   =
014  "$ENV{HOME}/.gtkticker";
015 my @LABELS       = ();
016 my $UPD_INTERVAL = 60;
017 my @SYMBOLS;
018
019 use Gtk;
020 use POE qw(
021     Component::Client::HTTP);
022 use HTTP::Request;
023 use Log::Log4perl qw(:easy);
024 use Data::Dumper;
025
026 Log::Log4perl->easy_init(
027                       $DEBUG);
028
029     # Read config file
030 open FILE, "<$RCFILE" or
031    die "Cannot open $RCFILE";
032 while(<FILE>) {
033     next if /^\s*#/;
034     push @SYMBOLS, $_
035                 for /(\S+)/g;
036 }
037 close FILE;
038
039 POE::Session->create(
040  inline_states => {
041    _start   => \&start,
042    _stop    => sub {
043              INFO "Shutdown" },
044    yhoo_response =>
045                \&resp_handler,
046    wake_up  =>
047              \&wake_up_handler,
048    }
049 );
050
051 my $STATUS;
052
053 $poe_kernel->post(
054         "ticker", "wake_up");
055 $poe_kernel->run();
056
057 ##################################
058 sub start {
059 ##################################
060
061   DEBUG "Starting up";
062
063   $poe_kernel->alias_set(
064                      'ticker');
065   my_gtk_init();
066
067   $STATUS->set("Startup");
068
069  POE::Component::Client::HTTP
070   ->spawn(
071     Agent   =>
072           'gtkticker/0.01',
073     Alias   => 'useragent',
074     Timeout => 60,
075   );
076 }
077
078 ##################################
079 sub upd_quotes {
080 ##################################
081
082   my $request =
083     HTTP::Request->new(
084       GET => $YHOO_URL .
085          join ",", @SYMBOLS);
086
087   $STATUS->set(
088           "Fetching quotes");
089
090   $poe_kernel->post(
091     'useragent',
092     'request',
093     'yhoo_response',
094     $request);
095 }
096
097 ##################################
098 sub my_gtk_init {
099 ##################################
100
101   my $w = Gtk::Window->new();
102   $w->set_default_size(
103                      150,200);
104
105       # Create Menu
106   my $accel =
107       Gtk::AccelGroup->new();
108   $accel->attach($w);
109   my $factory =
110     Gtk::ItemFactory->new(
111       'Gtk::MenuBar',
112       "<main>", $accel);
113
114   $factory->create_items(
115     { path =>  '/_File',
116       type =>  '<Branch>',
117     },
118     { path        =>
119               '/_File/_Quit',
120       accelerator =>
121                 '<control>Q',
122       callback    =>
123       [sub { Gtk->exit(0) }],
```

so-called accelerators to the mouse controls.

Factory-Made Menus

The menu is created by using the *Gtk::ItemFactory* class which is first used to create a *Gtk::MenuBar* menu bar. The menu entries and their subordinate pull-downs are created using the *create_items()* method.

The *path* parameter specifies the position of the menu item – for example, */_File/_Quit* defines the *Quit* entry below *File* in the menu bar. The underscores _ tell *Gtk* to underline the following character, allowing the user to press a keyboard shortcut (such as [Alt] + [F]) to navigate the menu. The *callback* parameter specifies the function that *Gtk* will perform whenever the user selects the entry with the mouse, or presses the keyboard shortcut defined by the *accelerator* parameter.

Layout Manager

Two different approaches are used to arrange widgets geometrically: *Gtk::VBox* and *Gtk::Table*. The *Gtk::VBox* container element aligns the widgets it contains vertically. The *pack_start()* method places the elements from the top down, while *pack_end()* stacks widgets bottom up. We can see that the following statement:

```
$vb->pack_start($menu_bar, ➥
$expand, $fill, $padding);
```

places the menu bar at the top of the VBox. In line 132 gtkticker uses *$factory->get_widget('<main>')* to retrieve the bar's object by name. The *$expand* parameter used in *pack_start()* specifies whether the area that the widget occupies should grow if the user

Listing 2: Gtkticker

```
124     });
125
126   my $vb = Gtk::VBox->new(
127                           0,0);
128   my $upd = Gtk::Button->new(
129                     'Update');
130
131   $vb->pack_start(
132     $factory->get_widget(
133           '<main>'), 0, 0, 0);
134
135       # Button at bottom
136   $vb->pack_end($upd,
137                 0, 0, 0);
138
139     # Status line on top
140     # of buttons
141   $STATUS= Gtk::Label->new();
142   $STATUS->set_alignment(
143                     0.5, 0.5);
144   $vb->pack_end($STATUS,
145                      0, 0, 0);
146   my $table =
147     Gtk::Table->new(
148       scalar @SYMBOLS, 3);
149
150   $vb->pack_start($table,
151                   1, 1, 0);
152
153   for my $row (0..
154                 @SYMBOLS-1) {
155     for my $col (0..2) {
156       my $label =
157            Gtk::Label->new();
158       $label->set_alignment(
159                     0.0, 0.5);
160       push @{$LABELS[$row]},
161            $label;
162
163      $table->attach_defaults(
164        $label, $col, $col+1,
165          $row, $row+1);
166       }
167   }
168
169   $w->add($vb);
170
171       # Destroying window
172   $w->signal_connect(
173     'destroy', sub {
174                 Gtk->exit(0)});
175
176     # Pressing update button
177   $upd->signal_connect(
178     'clicked', sub {
179       DEBUG "Sending wakeup";
180       $poe_kernel->post(
181         'ticker', 'wake_up')}
182   );
183   $w->show_all();
184 }
185
186 ###################################
187 sub resp_handler {
188 ###################################
189   my ($req, $resp) =
190     map { $_->[0] }
191                 @_[ARG0, ARG1];
192
193   if($resp->is_error()) {
194       ERROR $resp->message();
195       $STATUS->set(
196            $resp->message());
197       return 1;
198   }
199
200   DEBUG "Response: ",
201         $resp->content();
202
203   my $count = 0;
204
205   for(split /\n/,
206           $resp->content()) {
207
208     my($symbol, $price,
209        $change) =
210                 split /,/, $_;
211
212       chop $change;
213       $change = "" if
214         $change =~ /^0/;
215
216       $symbol =~ s/"//g;
217       $LABELS[$count][0]->
218                 set($symbol);
219       $LABELS[$count][1]->
220                  set($price);
221       $LABELS[$count][2]->
222                 set($change);
223       $count++;
224   }
225
226   $STATUS->set("");
227
228   1;
229 }
230
231 ###################################
232 sub wake_up_handler {
233 ###################################
234   DEBUG("waking up");
235
236       # Initiate update
237   upd_quotes();
238
239       # Re-enable timer
240   $poe_kernel->delay(
241   'wake_up', $UPD_INTERVAL);
242 }
```

drags the mouse to expand the main window. If so, fill specifies whether the widget itself should also grow – this will allow buttons to grow to enormous proportions. Finally, *$padding* specifies the minimum number of pixels that should separate the widget vertically from its neighbors. Gtkticker displays status messages in an unobtrusive *Gtk::Label* widget directly above the *Update* button. The *set_alignment()* method uses the following syntax

```
$STATUS->set_alignment➥
(0.5, 0.5);
```

to align the text horizontally and vertically. If you want to experiment, a horizontal value of *0.0* stands for left-justified, and *1.0* for right-justified text.

In contrast to *Gtk::VBox*, the *Gtk::Table* container element provides Perl programmers with a tool to conveniently arrange widgets in a table. The *attach_defaults()* method expects five parameters: the widget, which is to be aligned and two column and row coordinates between which the widget will be located. For example, the following statement:

```
$table->attach_defaults➥
($label, 0, 1, 1, 2);
```

specifies that the *Gtk::Label* object that *$label* points to will be added to the first row ("between 0 and 1") and the second column ("between 1 and 2") of a table called *$table*.

And Action!

You can assign actions to *Gtk::Button* type widgets. *Gtk* performs the action assigned when a user presses the button. The method called in line 177, *signal_connect()*, specifies that *Gtk* should send a *wake_up* event to the POE kernel when the user clicks the *Update* button.

The main window also has an action assigned – users can click the *X* in the top right-hand corner of the window to terminate the application. As in the following code:

```
$w->signal_connect('destroy',
  sub {Gtk->exit(0)});
```

the subroutine tidies up the *Gtk* session and quits the program. After completing the widget definitions, the *show* method (line 183) displays them in the main window (line 183) of the screen.

The Kernel Strikes Back

In the *yhoo_response* state, the POE kernel jumps to the function listed below line 187, *resp_handler*. By definition, *POE::Component::Client::HTTP* will store a request and a response packet in *ARG0* and *ARG1* when this happens. POE uses this slightly strange approach to passing parameters after introducing new functions representing numerical constants, such as *KERNEL*, *HEAP*, *ARG0*, *ARG1*. POE's authors expect programmers to use them to index the array of function parameters, *@_*. For example, *$_[KERNEL]* will always return the kernel object, helping to keep the index that *KERNEL* points to transparent.

The request and response packets just mentioned are references to arrays, whose first elements contain *HTTP::Request* or *HTTP::Response* objects. The *map* command in line 190 extracts them to *$req* and *$resp*.

If a HTTP error occurs, line 195 generates an appropriate message in the status widget and the function returns. Otherwise, the two dimensional global array of label widgets is refreshed. The widgets display the stock exchange symbol, the current price, and the change as a percentage (the special case zero percent is simply ignored).

Periodically Delayed Alert

A *wake_up* event in the POE kernel calls the *wake_up_handler()* routine defined in line 232 and the following lines. It calls the *upd_quotes()* function, which is implemented in line 79 and following. The function defines a *HTTP::Request* object and uses an event to send it to the *POE::Component::Client::HTTP* component. The target state for the *ticker* is set to *yhoo_response*.

After completing these preparatory steps, *wake_up_handler()* uses the kernel's *delay()* method to set an alert. This will cause a *wake_up* event to occur in the *ticker* session when the number of seconds defined as the *$UPD_INTERVAL* (60 seconds in this case) has elapsed. From this point onward, the ticker will automatically update its share prices every 60 seconds, without the user having to press the *Update* button.

Installation

It is best to use a CPAN shell to install the required POE modules, POE and *POE::Component::Client::HTTP*. If the *POE::Component::Client::DNS* module is also installed, DNS requests will then be performed asynchronously; *gethostbyname()* may otherwise cause a slight delay.

Installing *Gtk* from CPAN means resolving a few dependencies and caused a few issues on my own system. However,

```
touch ./Gtk/build/➥
perl-gtk-ref.pod
perl Makefile.PL ➥
--without-guessing
```

in the distribution directory, followed by *make install* worked around these obstacles.

The script logs debugging messages to STDOUT, if this gets in your way, simply set the verbosity via *Log::Log4perl* (also from CPAN) and in line 27 from $DEBUG to $ERROR. It is fascinating to see how smooth the display is. Even if you are fooling around with the menu while the application is performing an automatic update across a slow network, the GUI stays intact. Expensive-looking, but cheaply implemented! ■

INFO

[1] Listings for this article: *http://www.linux-magazine.com/Magazine/Downloads/42/Perl/*

[2] POE: *http://poe.perl.org*

[3] Jeffrey Goff, "A Beginner's Introduction to POE", 2001: *http://www.perl.com/pub/a/2001/01/poe.html*

[4] Matt Sergeant, "Programming POE", talk at TPC 2002: *http://axkit.org/docs/presentations/tpc2002*

[5] Gtkperl: *http://gtkperl.org*

[6] Gtkperl tutorial: *http://personal.riverusers.com/~swilhelm/gtkperl-tutorial/*

[7] Eric Harlow, "Developing Linux Applications with GTK+ and GDK": New Riders, 1999, ISBN 0735700214