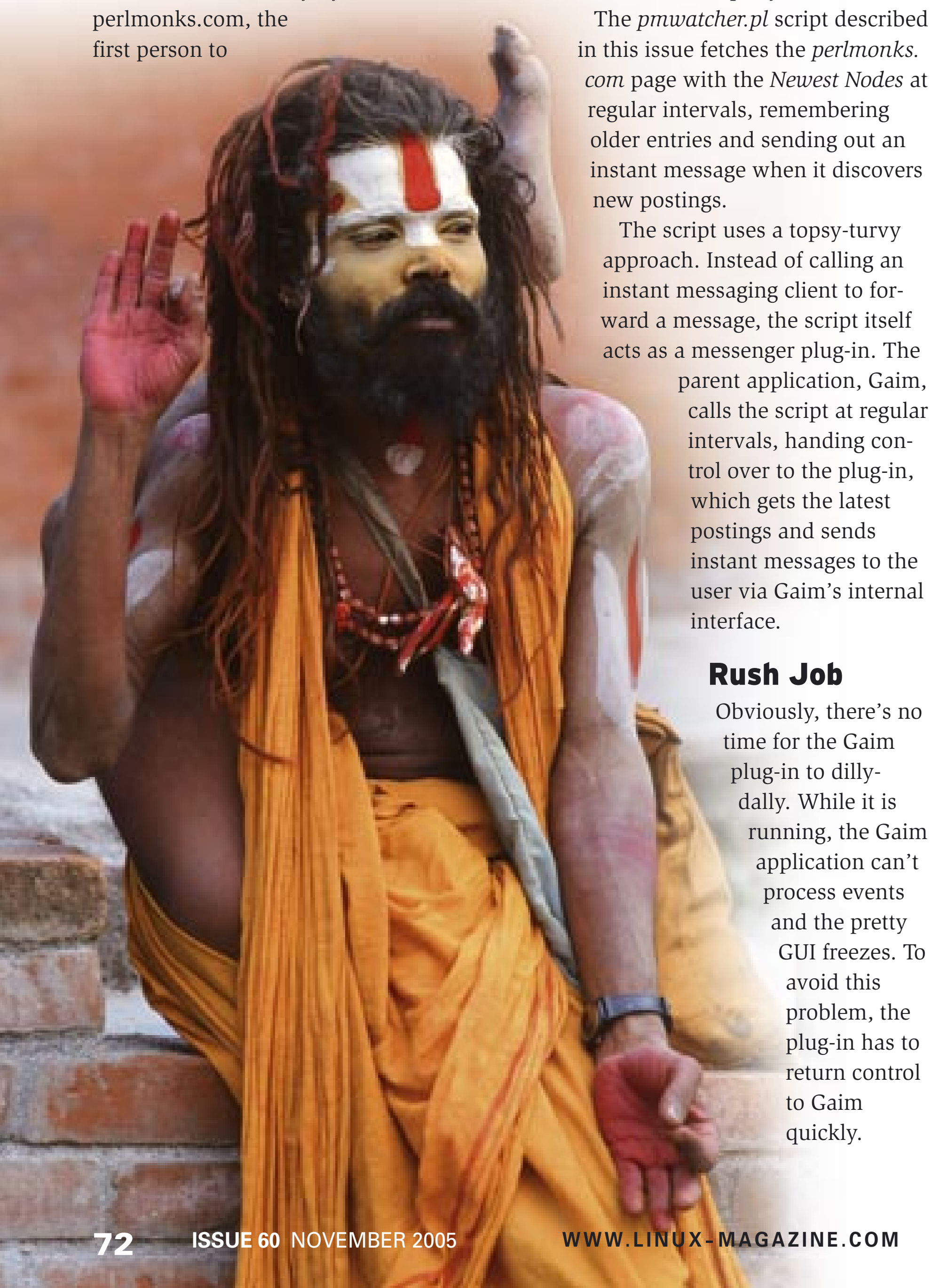
Get the news from perlmonks.com with a Gaim plug-in

SEEKING WISDOM

The Gaim project offers an instant messenger client that speaks a large number of protocols. We'll show you how to extend Gaim with Perl plugins. **BY MICHAEL SCHILLI**



First-time visitors to *perlmonks.com* are rubbing their eyes in disbelief: High-caliber Perl hackers are jumping to answer even the simplest of newbie questions. The reason for this is that the community assigns XP (experience) points for the best replies. And the more XP you have, the higher you climb in the ranking, from a novice, to a monk, and slowly to expert status.

Best of Class

Due to the community dynamics on perlmonks.com, the first person to answer a question correctly typically gets the most XP. Instead of pulling the web page with the latest questions time and time again, it makes sense to script the process and have the script let you know when a new query arrives.

The *pmwatcher.pl* script described in this issue fetches the *perlmonks.com* page with the *Newest Nodes* at regular intervals, remembering older entries and sending out an instant message when it discovers new postings.

The script uses a topsy-turvy approach. Instead of calling an instant messaging client to forward a message, the script itself acts as a messenger plug-in. The parent application, Gaim, calls the script at regular intervals, handing control over to the plug-in, which gets the latest postings and sends instant messages to the user via Gaim's internal interface.

Rush Job

Obviously, there's no time for the Gaim plug-in to dilly-dally. While it is running, the Gaim application can't process events and the pretty GUI freezes. To avoid this problem, the plug-in has to return control to Gaim quickly.

However, it can take a few seconds to download the content of a remote web page. Both DNS name resolution and the process of retrieving the content of the requested web page can take some time, during which the CPU should return to other tasks. The tried-and-trusted POE [3] framework provides exactly what we need. The POE kernel runs a single process (and only a single thread), but uses cooperative multitasking between concurrent tasks to ensure that each one is processed.

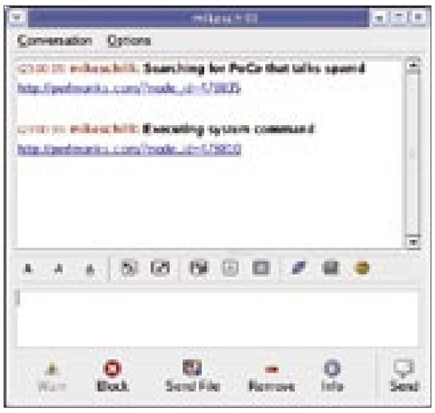
Typically, POE runs the show with its own *kernel*, the POE task manager. However, it can just as easily be integrated with other event loops, such as the ones provided by Gtk or Perl/Tk. In the Gaim scenario, however, with a plugin that gets called by the main application at regular intervals, we need to think differently: in *plugin_load()*, which Gaim calls during startup, the *pmwatcher.pl* plug-in script defines the various tasks that it will be running later. Before *plugin_load()* hands the reins back to Gaim, it calls *Gaim::timeout_add($plugin, $UPDATE,\&refresh);*, and this guarantees that Gaim will call the *refresh()* plug-in function *refresh()* exactly *$UPDATE* seconds later. This is where POE cuts in to handle the most urgent tasks before handing control back to Gaim once more, but of course, not without scheduling another event to ensure that the main application will call *refresh()* after a specified interval, and thus entering an endless loop. *pmwatcher.pl* stores the plug-in object passed to *plugin_load()* in the global *$PLUGIN* variable to allow *refresh()* to request a new timeout from Gaim.

Additionally, the calls to *Gaim::signal_connect* in lines 61 and 68 handle events from Gaim users logging on and off. When these events occur, Gaim jumps to the *buddy_signed_on_callback* and *buddy_signed_off_callback* functions defined in the plugin, which check if the username reported matches the name specified in line 25. If so, *buddy_signed_on_callback* stores the Gaim user structure in the global *$BUDDY* variable. The variable is referenced later when sending a message to the user.

buddy_signed_on_callback sets the *$ACTIVE* flag to 1 when this special user logs on; and *buddy_signed_off_callback* sets it to 0 when the user logs off. If *$ACTIVE* is set, the plugin will run a single POE timeslice; if it isn't, nothing will happen in the plugin's *refresh* method.

If *refresh* were to call the POE *run()* method, it would never return. Instead, line 162 calls *run_one_timeslice()*, which simply processes the most pressing task in the POE pipeline, but returns immediately afterwards instead of continuing or waiting for further events.

As each timeslice can process only a small part of a request, the whole HTTP request can take about 20 *refresh()*



[illegible] mikeschilli: Searching for PoCo that talks spamd
http://perlmonks.com/?node_id=[illegible]

[illegible] mikeschilli: Executing system command
http://perlmonks.com/?node_id=[illegible]

Figure 1: New Perlmonks postings are reported via instant messaging, giving you an opportunity to gain XP by being first to offer help.

cycles, but this is not really an issue. The only important thing is that the CPU keeps on running at full speed during the callback and does not wait for external events such as HTTP response data rolling in. POE uses POE::Component::Client::HTTP to take care of the details involved in getting a page off the Web. If POE::Component::Client::DNS is installed, even the hostname will be resolved asyncronously instead of by using *gethostbyname()*.

The initial POE state defined in line 83, *_start* does nothing but initiate the next state, *http_start*. The *post* method sends a request to the POE component *POE::Component::Client::HTTP*, which is already set up and running in the POE kernel; the *spawn* method call in line 76 takes care of this. The HTTP client component object (labeled *ua*) will change state to *http_ready* whenever HTTP response data trickles in. Before *http_start* hands control back to the POE kernel, it requests a POE timeout 10 minutes in the future; this again triggers an *http_start* state to get the Website again.

The *http_ready* state handler is passed a reference to an array in *$_[ARG1]*; the first element in the array is an object of type HTTP::Response with the results of the web request (for more details on POE's unusual approach to passing parameters, check out [3].)

Needle in a Haystack

To extract the links and text passages from the questions section of the down-

ADVERTISEMENT

loaded Perlmonks page, the *qparse* function in line 237 implements an HTML parser. Not every link on the page will belong to a new query, because there are other sections associated with the page, such as Discussion or Meditations, that we want our *pmwatcher.pl* script to ignore.

HTML::TreeBuilder creates a tree of HTML elements from an HTML document. *qparse()* navigates the tree built by HTML::TreeBuilder, first going to a known < *A* > element with *toc-Questions* in its *name* attribute.

From the HTML::Element type node that this finds, the *parent()* method takes us one level up the tree, where the *while* loop in line 269 looks for a

Listing 1: pmwatcher.pl

```
001 #!/usr/bin/perl -w
002 ##############################
003 # pmwatcher - Gaim plugin to
004 #         watch perlmonks.com
005 ##############################
006 use strict;
007 use Gaim;
008 use HTML::TreeBuilder;
009 use URI::URL;
010 use CGI qw(a);
011 use Cache::FileCache;
012 use POE
013  qw(Component::Client::HTTP);
014 use HTTP::Request::Common;
015
016 our $FETCH_INTERVAL = 600;
017 our $FETCH_URL      =
018     "http://perlmonks.com/"
019   . "?node=Newest%20Nodes";
020
021 our $ACTIVE = 0;
022
023 # Call plugins every second
024 our $UPDATE = 1;
025 our $USER   = "mikeschilli";
026 our $BUDDY  = undef;
027 our $PLUGIN = undef;
028
029 our %PLUGIN_INFO = (
030  perl_api_version => 2,
031  name     => "pmwatcher",
032  summary =>
033    "Perlmonks Watch Plugin",
034  version     => "1.0",
035  description =>
036    "Reports latest postings "
037    . "on perlmonks.com, "
038    . "Mike Schilli, 2005"
039    . "(m\@perlmeister.com)",
040  load => "plugin_load",
041 );
042
043 our $cache =
044   new Cache::FileCache(
045  {
046   namespace => "pmwatcher",
047  }
048   );
049
050 ##############################
051 sub plugin_init {
052 ##############################
053  return %PLUGIN_INFO;
054 }
055
056 ##############################
057 sub plugin_load {
058 ##############################
059  my ($plugin) = @_;
060
061  Gaim::signal_connect(
062   Gaim::BuddyList::handle(),
063   "buddy-signed-on",
064   $plugin,
065   \&buddy_signed_on_callback,
066  );
067
068  Gaim::signal_connect(
069   Gaim::BuddyList::handle(),
070   "buddy-signed-off",
071   $plugin,
072  \&buddy_signed_off_callback,
073  );
074
075  POE::Component::Client::HTTP
076    ->spawn(
077   Alias    => "ua",
078   Timeout => 60,
079  );
080
081  POE::Session->create(
082   inline_states => {
083    _start => sub {
084     $poe_kernel->yield(
085      'http_start');
086    },
087
088    http_start => sub {
089     Gaim::debug_info(
090      "pmwatcher",
091      "Fetching $FETCH_URL\n"
092     );
093     $poe_kernel->post(
094      "ua",
095      "request",
096      "http_ready",
097      GET $FETCH_URL);
098     $poe_kernel->delay(
099      'http_start',
100      $FETCH_INTERVAL
101     );
102    },
103
104    http_ready => sub {
105     Gaim::debug_info(
106     "pmwatcher",
107     "http_ready $FETCH_URL\n"
108     );
109     my $resp = $_[ARG1]->[0];
110     if($resp->is_success()) {
111      pm_update(
112        $resp->content());
113     } else {
114      Gaim::debug_info(
115       "pmwatcher",
116       "Can't fetch " .
117       "$FETCH_URL: " .
118       $resp->message()
119      );
120     }
121    },
122   }
123  );
124
125  Gaim::timeout_add($plugin,
126   $UPDATE, \&refresh);
127  $PLUGIN = $plugin;
128 }
129
130 ##############################
131 sub buddy_signed_on_callback{
132 ##############################
133  my ($buddy, $data) = @_;
134
135  return if
136   $buddy->get_alias ne $USER;
137  $ACTIVE = 1;
138  $BUDDY  = $buddy;
139 }
140
141 ##############################
142 sub
143 buddy_signed_off_callback {
144 ##############################
145  my ($buddy, $data) = @_;
146
147  return if
148   $buddy->get_alias ne $USER;
149  $ACTIVE = 0;
150  $BUDDY  = undef;
151 }
152
153 ##############################
```

<table> element by calling *right()* to move to the right at this level of the tree. The table has rows of links with the questions in the first column of the table and links to the posters in the second column.

This is why the first *for* loop in 274 goes to each *<tr>* element (the table rows), and why the inner loop searches for *<a>* links when it gets there. Starting at the current element, the *look_down()* method of a tree element searches downward for nodes with specific properties and hands matching elements back as a list. The *_tag => $tagname* condition searches for tags with the required names. *attrname => $attrvalue* checks if the tags that this

Listing 1: pmwatcher.pl

```
154 sub refresh {
155 ###################################
156  Gaim::debug_info(
157  "pmwatcher",
158  "Refresh (ACTIVE=$ACTIVE)\n"
159  );
160  if ($ACTIVE) {
161   $poe_kernel
162     ->run_one_timeslice();
163  }
164
165  Gaim::timeout_add($PLUGIN,
166   $UPDATE, \&refresh);
167 }
168
169 ###################################
170 sub pm_update {
171 ###################################
172  my ($html_text) = @_;
173
174  if (my @nws =
175    latest_news($html_text)) {
176
177   my $c =
178   Gaim::Conversation::IM::new
179   ($BUDDY->get_account(),
180    $BUDDY->get_name()
181   );
182
183   $c->send("$_\n") for @nws;
184  }
185 }
186
187 ###################################
188 sub latest_news {
189 ###################################
190  my ($html_string) = @_;
191
192  my $start_url =
193    URI::URL->new($FETCH_URL);
194
195  my $max_node;
196
197  my $saved =
198    $cache->get("max-node");
199
200  $saved = 0
201    unless defined $saved;
202
203  my @aimtext = ();
204
205  for my $entry (
206   @{ qparse($html_string) }){
207
208   my ($text, $url) = @$entry;
209
210   my ($node) =
211     $url =~ /(\d+)$/;
212   if ($node > $saved) {
213    Gaim::debug_info(
214     "pmwatcher",
215     "New node $text ($url)");
216
217    $url =
218    a({ href => $url }, $url);
219
220    push @aimtext,
221      "<b>$text</b>\n$url";
222   }
223
224   $max_node = $node
225     if !defined $max_node
226     or $max_node < $node;
227  }
228
229  $cache->set("max-node",
230   $max_node)
231    if $saved < $max_node;
232
233  return @aimtext;
234 }
235
236 ###################################
237 sub qparse {
238 ###################################
239  my ($html_string) = @_;
240
241  my $start_url =
242    URI::URL->new($FETCH_URL);
243
244  my @questions = ();
245
246  my $parser =
247    HTML::TreeBuilder->new();
248
249  my $tree =
250    $parser->parse(
251                $html_string);
252
253  my ($questions) =
254    $tree->look_down(
255     "_tag", "a",
256     "name", "toc-Questions");
257
258  if (!$questions) {
259   Gaim::debug_info(
260    "pmwatcher",
261    "No Questions section"
262   );
263   return undef;
264  }
265
266  my $node =
267    $questions->parent();
268
269  while($node->tag()
270         ne "table") {
271    $node = $node->right();
272  }
273
274  for my $tr (
275   $node->look_down(
276    "_tag", "tr"
277   )) {
278
279   for my $a (
280    $tr->look_down(
281     "_tag", "a"
282    )) {
283    my $href =
284      $a->attr('href');
285    my $text = $a->as_text();
286    my $url  =
287      URI::URL->new($href,
288     $start_url);
289
290    push @questions,
291      [ $text, $url->abs() ];
292
293   # Process only the question
294   # not the author's node
295    last;
296   }
297  }
298
299  $tree->delete();
300  return \@questions;
301 }
```

finds have an attribute with the specified name.

The link text and *href* attribute value are extracted from the first link to be found, the latter is converted to an absolute URL, and finally, both values are added to the *@questions* array. The link in the second row (containing poster name and details) is squashed by a pair of *for* loops (lines 274 and 279) with a *last* instruction following the end of the inner loop. Another important thing is to *delete()* the tree after parsing to avoid wasting valuable memory.

Alternative parsers include XML::LibXML or XML::XSH, both of which use powerful XPath syntax. However, both take offense at poorly written HTML documents and ball out where a web browser would be more forgiving.

The script still needs to make a note of the questions it has already seen to disambiguate them from new queries. To do so, it leverages the fact that Perlmonks.com assigns a unique numeric node ID, which is hidden in the URL for the question. A regular expression extracts the ID from the URL and compares it with the last number saved in a persistent Cache::FileCache object. A question is new if it has a higher ID than the last known node. Then the last known node ID is cached for the following queries.

The *latest_news()* function returns an array of formatted IM messages for the user. If the array is empty, there is no news. If there is news, lines 179 and 180 reference the user's globally stored Gaim structure in *$BUDDY* to create a *Gaim::Conversation::IM* class object. Calling this object's *send()* method then makes the HTML-formatted message show up in the user's IM window, just as if an IM buddy had sent it.

Installation

If you do not have Gaim, it is a good idea to download the latest version 1.4.0 from *gaim.sourceforge.net*. The Perl interface, *Gaim.pm*, is not available from CPAN, but it is included with the Gaim distribution. Enter the following to install the *Gaim.pm* Perl module:

```
cd plugins/perl/common
perl Makefile.PL
make install
```

You need to manually create the plug-in directory in your home directory by entering *mkdir ~/.gaim/plugins*. Any Perl scripts located there and ending in *.pl* are picked up by Gaim's Perl loader as Gaim plug-ins when the program launches. Just copy *pmwatcher.pl* to the *plugins* directory, make it executable, and then launch Gaim. You can then select *Tools->Preferences->Plugins* (Figure 3) to enable the plug-in permanently by selecting the appropriate check box. Gaim takes the data shown in the dialog from the *%PLUGIN_INFO* hash, which is returned by *plugin_init()* (defined at line 51 in *pmwatcher.pl*).

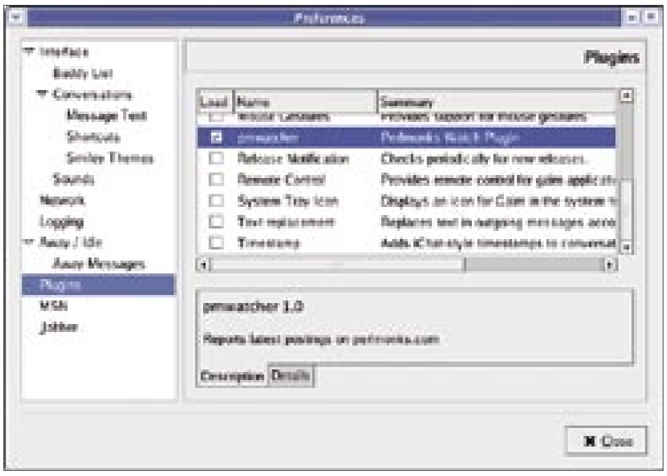
The CPAN modules *HTML::TreeBuilder*, *URI::URL*, *Cache::FileCache*, *POE*, *POE::Component::Client::HTTP* *POE::Component::Client::DNS* and *HTTP::Request::Common* need to be installed using a CPAN shell.

In the script itself, you can set *$FETCH_INTERVAL* to define the interval between web requests. The default is ten minutes, and that should be fine to keep you up to date without annoying the people who run perlmonks.com.

You can launch Gaim with the *-d* (for *debug*) option to print log messages from the *Gaim::debug_info* function in the Perl script on standard output. One thing to keep in mind: Gaim's plug-in scripts will not run at the command line, where they quit after issuing an error message. They only run within Gaim.

Don't forget to set the *$USER* variable to your preferred username. Logging on will then trigger the plug-in actions. If the user is not logged on, the plug-in is called once a second, but without triggering any web requests.

Once this user logs on via Gaim (regardless of the service), web requests start fetching data from perlmonks.com every 10 minutes, and the plug-in keeps the aspiring expert up to date with challenging questions, to help them on their path to greater wisdom. ■



Load	Name	Summary
☐	Mouse Gestures	Provides support for mouse gestures
☑	pmwatcher	Perlmonks Watch Plugin
☐	Release Notification	Checks periodically for new releases.
☐	Remote Control	Provides remote control for gaim applic…
☐	System Tray Icon	Displays an icon for Gaim in the system t…
☐	Text replacement	Replaces text in outgoing messages acco…
☐	Timestamp	Adds iChat-style timestamps to conversat…

pmwatcher 1.0

Reports latest postings on perlmonks.com

Figure 3: Enabling the new Perl plug-in.

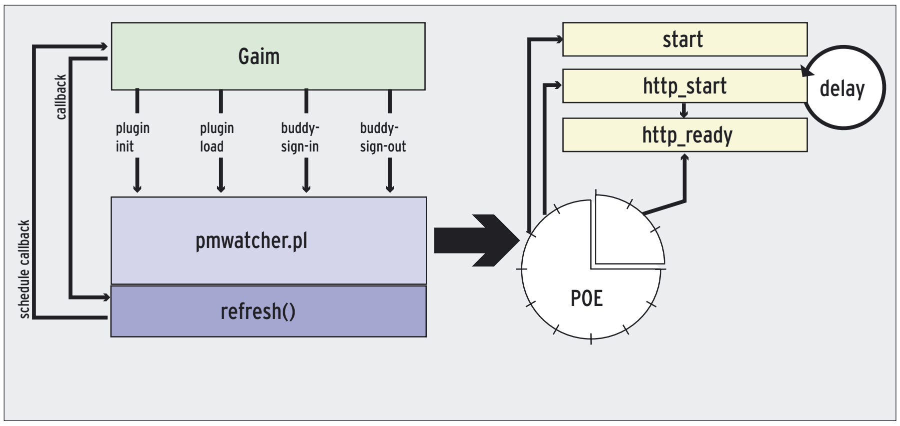


Figure 2: Gaim communicates with the Perl plug-in, which in turn controls a POE state machine.

INFO

[1] Listings for this article: *http://www.linux-magazine.com/Magazine/Downloads/60/Perl*

[2] Basic tutorial on the Gaim Perl interface: *http://gaim.sourceforge.net/api/perl-howto.html*

[3] Michael Schilli, "DJ Training," Linux Magazine 08/2004, *http://www.linux-magazine.com/issue/45/Perl_Playlist_selecting.pdf*