Stock exchange alerts via instant message

Bulls and Bears

A Pidgin plugin written in Perl alerts the user via instant message when specific shares become volatile. *By Michael Schilli*

The Pidgin instant messaging client [1] not only runs on a variety of operating systems, it supports a large number of IM protocols. Whether you prefer Yahoo or MSN, Google Talk or IRC, the free Pidgin plugin groups them under its umbrella, facilitating communications with online buddies who use different services. Pidgin's plugin architecture helps the omnipresent product keep on top of newly released or changing protocols and continually extends its scope.

You don't even need to write plugins in Pidgin's native language, C, and provide them as shared libraries. Scripts written in high-level languages like Perl can be linked in at startup, and if they cooperate with Pidgin's GLib-based event loop, they can even execute complex operations like retrieving websites while the Pidgin core and its graphical interface keep running uninterrupted.

Although it might sound crazy to let a C program play hopscotch with a Perl script in the same event loop, if you recall that Perl is just an abstraction layer on top of a C layer, and thus just as good at triggering and receiving events, the whole thing looks far more logical.

Ups and Downs

The Pidgin plugin that I will describe here is unobtrusive for the most part, but it regularly contacts a share ticker service in the background. Because of this, it can let you know whether one of the shares stored in the configuration file starts to climb like crazy or plummet. If either of these happens, the plugin opens a communication window to the logged in user, as shown in Figure 1, and displays the complete list of monitored shares, including the day's changes as a percentage.

"Pidgen supports a large number of IM protocols."

In the configuration file shown in Figure 2, the user can specify which ticker icons to monitor, as well as the scale of change that warrants an alert. For lines that contain the share shortcut but no percentage, the plugin automatically assumes a tolerance of two percent.

The actual plugin code contains just 82 lines (Listing 1) and is fairly simple [2].

Extending Pidgin in Perl

To begin, Pidgin loads the code in Listing 1, assuming you've installed the Perl script with a `.pl` suffix in its plugin directory and assuming the user has activated the plugin in the corresponding dialog box (see the "Installation" section).

The plugin must define the callbacks required by the Pidgin API, `plugin_init()` and `plugin_load()`. Pidgin accesses `plugin_init()` to retrieve information about the plugin and be able to display it in the *Tools | Plugins* menu.

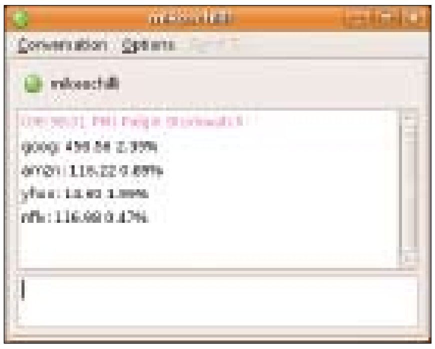
When stock market players click to enable the plugin later, the `plugin_load` function (lines 46-57) starts running. It defines the plugin's tasks by submitting them to the event loop, which will process them later. If the news from the market is so bad that the budding broker disables the plugin, it will handle any necessary cleanup in the `plugin_unload()` callback before Pidgin releases it from memory.

Outsourcing Quote Watch

The WatchQuotes module described later helps track stock quotes. It parses the plugin's

MIKE SCHILLI

Mike Schilli works as a software engineer with Yahoo! in Sunnyvale, California. He can be contacted at *mschilli@perlmeister.com*. Mike's homepage can be found at *http://perlmeister.com*.



```
Conversation  Options
mikeschilli
goog: 498.58 2.39%
amzn: 118.22 0.89%
yhoo: 13.93 1.99%
nflx: 116.90 0.47%
```

Figure 1: The stock exchange monitor alerting the user because the price of Google shares has risen by more than two percent.

configuration file when line 54 of the plugin calls its `init()` method. Line 55 then launches the permanent monitoring process by calling the `watch()` method. The plugin script passes a reference to the `quotes_update` callback defined in lines 60-82 to the method; the callback later wraps any message passed to it into an IM conversation and forwards the whole thing to the pleasantly, or unpleasantly, surprised user.

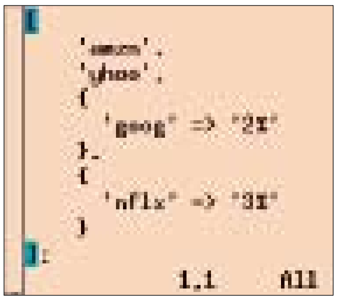
For this to happen, the `find()` method of the `Purple::Accounts` object in line 69 finds the user, who is (hopefully) logged in; line 73 then initiates a conversation to the user in the form of a `Purple::Conversation` object. If something goes wrong, say, because the user defined in `$USER` is not logged in to the specified service, line 77 returns, and the message never gets to see the light of a turbulent day at the stock exchange.

If everything works correctly, the `write()` method called in line 79 sends the message using the plugin name as the sender. The final parameter uses the `time()` function to set the current time, which Pidgin displays for each message.

Cool Kids' Framework

The `$USER` and `$PROTOCOL` variables in lines 21 and 22 define the username to alert in case of trouble, and the IM service in which to do this. Here, the service is `prpl-yahoo`, which is Yahoo's messenger protocol. The plugin script `pidgin-stockwatch.pl` pulls in the WatchQuotes module in line 7. When implementing this module, it's important to make sure that it processes events asynchronously with Pidgin's event loop. This ensures that the GLib kernel processes mouse clicks on Pidgins's UI by the user during long-winded operations, such as retrieving data from a website. If the plugin just issued web requests and waited for the data to trickle in, the GUI would turn into a pillar of salt until the result came back.

Several frameworks can be employed to accomplish this. I have used POE [3] in several Perl columns before, so this time, I'll give a newcomer a chance. The one that is the buzz with the cool kids of the Perl scene right now is the AnyEvent Framework [4]. It doesn't bind directly to a specific event loop but cooperates with half a dozen implementations.

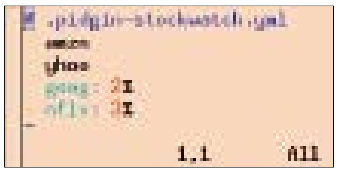


```
[
  'amzn',
  'yhoo',
  {
    'goog' => '2%'
  },
  {
    'nflx' => '3%'
  }
]
                    1,1      All
```

Figure 3: Perl creates a data structure from the YAML file.

Jack of All Trades

The advantage with using AnyEvent is that you can implement a module like `WatchQuotes.pm` to be totally generic and run it with all kinds of event loops and on a variety of platforms. The only precondition for this is that the main program maintains a reference to the AnyEvent object. If the last reference to it vanishes, AnyEvent will clean up after itself and no more events will be processed. To keep it running, I will store a reference to the `WatchQuotes` object in the global `$WATCH_QUOTES` variable, which then references AnyEvent's objects and keeps them alive.



```
.pidgin-stockwatch.yml
amzn
yhoo
goog: 2%
nflx: 3%
                    1,1      All
```

Figure 2: Users can add stock symbols to the watch list in the YAML file and define a threshold as a percentage if needed.

This approach is necessary because Pidgin only calls the defined plugin callbacks briefly at startup, and their local variables disappear immediately after the program flow returns to the regular Pidgin biotope.

Bind Loosely

Loose event loop binding with AnyEvent offers another benefit: The `WatchQuotes.pm` module can be tested independently, of Pidgin, for example, by running the script in Listing 2. The script instantiates the `WatchQuotes` class and calls its `init()` method, which tells `WatchQuotes` to parse the `~/.pidgin-stockwatch.yml` configuration file defined by the user and convert it internally into a Perl data structure (Figure 3).

As a functional example of an event loop, AnyEvent defaults to a loop implemented in Perl. Listing 2 uses `condvar()` to define a conditional variable to which the AnyEvent kernel can send messages. The subsequent method call to `recv()` in line 14 then waits for a message, which never comes; in the mean time, it lets the AnyEvent kernel handle events from modules like WatchQuotes.

The test script used here retrieves share prices at regular intervals and calls the callback defined in lines 17-20 with a list of formatted stock quotes, if the price of one of the monitored shares defined in the configuration file exceeds the defined thresholds. The call to `print()` in line 19 then sends the message to stdout. The script continues to run and keeps refreshing stock data from the Yahoo ticker service until the user stops it by pressing Ctrl + C. This gives programmers a convenient approach to identifying any errors before the script is injected into the

RTFM – IF YOU HAVE ONE

Documentation on writing Pidgin plugins is hard to find and incomplete. Although Pidgin maintainer Sean Egan wrote a book, *Building and Extending Gaim* [5], the name of the project (now Pidgin instead of Gaim) and more or less every single function call and data structure has changed since then. The book, although well written, is only useful for studying the underlying Pidgin architecture.

Worse, the online documentation [6] [7] is not as up to date as one might like. The automatically generated Doxygen documentation lacks, as is often the case, a touch of TLC; it is also incomplete and thus fairly useless. The most effective method of finding a parameter for a function call is to investigate recent real-life plugins [8] (Table 2).

inimical Pidgin environment, where debugging is difficult – especially if Pidgin fails to load the plugin correctly for some reason (see the "RTFM – If You Have One" box).

The `WatchQuotes.pm` module in Listing 3 starts in typical Perl style by defining a `new()` constructor that creates an object and locates the user's home directory. It accepts additional parameters such as the name of the configuration file in `conf_file` but falls back to default settings if they're not provided and stores them in the object hash.

YAML for Humans and Machines

The `init()` method defined in lines 33-52 calls the YAML module's `LoadFile()` function to parse the configuration. This format has the advantage of being just as easily readable for humans as for machines (Figure 2).

The application supports both simple array entries, such as `- amzn`, and hash data structures, such as `- goog: 2%`, which YAML stores as references to a hash providing the mapping (`"goog" => "2%"`). Line 41 checks to see if Perl's `ref` function returns the `HASH` string, which indicates a hash reference, to distinguish it from simple entries.

If an empty string is returned, the value is a simple scalar and line 49 inserts the default threshold value of two percent. The module stores monitored ticker symbols in the `conf` entry in the object hash and assigns the configured trigger percentage values to them.

The `watch` method in lines 55-67 creates a periodic timer. The `after => 10` parameter tells the timer to call the callback defined in `cb` exactly 10 seconds after starting. This delay is intentional and necessary if the plugin starts before the user is reachable on the instant messaging network. Without this pause for thought, early messages would disappear into a black hole instead of going to the stockholder. The `interval` parameter, on the other hand, carries a value of `300` to tell the timer to call the callback every five minutes after the first run to pick up the latest share prices from the Yahoo server and figure out if the current trades exceed the configured trigger values.

"The most effective method of finding a parameter for a function call is to investigate recent real-life plugins."

Asynchronous Web Fetch

The callback function reference passed to the `watch()` method points to the

LISTING 1: pidgin-stockwatch.pl

```perl
#!/usr/local/bin/perl -w
use strict;

use Pidgin;
use local::lib;
use Glib;
use WatchQuotes;

our %PLUGIN_INFO = (
 perl_api_version => 2,
 name => "Pidgin Stockwatch",
 summary =>
   "Stock Alert via IM",
 version => "1.0",
 author  => "Mike Schilli " .
   "<m\@perlmeister.com>",
 load   => "plugin_load",
 unload => "plugin_unload",
);

our $USER = "yahoo-username";
our $PROTOCOL = "prpl-yahoo";

our $WATCH_QUOTES =
  WatchQuotes->new();

#############################
sub plugin_init {
#############################
 return %PLUGIN_INFO;
}

#############################
sub plugin_unload {
#############################
 my ($plugin) = @_;

 Purple::Debug::info(
  "stockwatch",
  "Plugin unloaded.\n");

 1;
}

#############################
sub plugin_load {
#############################
 my ($plugin) = @_;

 Purple::Debug::info(
  "stockwatch",
  "Plugin loaded.\n");

 $WATCH_QUOTES->init();
 $WATCH_QUOTES->watch(
  \&quotes_update);
}

#############################
sub quotes_update {
#############################
 my ($msg) = @_;

 Purple::Debug::info(
  "stockwatch",
  "Updating Quotes.\n");

 my $account =
   Purple::Accounts::find(
  $USER, $PROTOCOL);

 my $conv =
   Purple::Conversation->new(
  1, $account, $USER);

 # user not online?
 return unless defined $conv;

 $conv->get_im_data->write(
  $PLUGIN_INFO{name}, $msg,
  0, time);
}
```

main script's `quotes_update()` function. The timer then passes it on to the `fetch()` method (lines 70-91), which retrieves the ticker data from Yahoo's website.

Although this can take a couple of seconds if the going is tough on the Internet, the `http_get()` function called in line 84 comes from the treasure trove of the CPAN AnyEvent::HTTP module and processes the request asynchronously. The function expects the URL for the share price service and a callback, to which it jumps once the data has trickled in completely.

Note that Perl immediately carries on with the program flow after calling `http_get()`, without the requested HTTP data being available at this point in time.

Free Ticker Data

As the documentation for the CPAN Finance::YahooQuote [9] module reveals, Yahoo's stock ticker service supports a whole bunch of parameters, from which the `WatchQuotes.pm` picks only those listed in Table 1: the ticker symbol (`s`), the previous day's price (`p`), the current price (`l1`), and the percent change (`p2`).

Bundled together as a string, WatchQuotes throws `sl1p2` plus `s=goog+yahoo` at the Yahoo service for my sample user, who is interested in how the Google and Yahoo shares are faring.

LISTING 2: watch-quotes

```
#!/usr/bin/perl -w
use strict;
use lib local::lib;
use AnyEvent;
use WatchQuotes;

my $watcher =
  WatchQuotes->new();
$watcher->init();
$watcher->watch(\&callback);

my $quit_program =
  AnyEvent->condvar;
$quit_program->recv;

#############################
sub callback {
#############################
 print "$_[0]\n";
}
```

The data are returned in CSV format, which is two lines of text like the following:

```
"GOOG",467.49,475.83,"+1.78%"
"YHOO",14.89,14.94,"+0.34%"
```

The simple regular expression-based parser in `parse_csv()` (lines 94-119) creates a data structure from them. The `data` hash entry of the `WatchQuotes` object then contains a pointer to an array that contains the previous day's price, the latest price (typically with a delay of 20 minutes), and the change as a percentage.

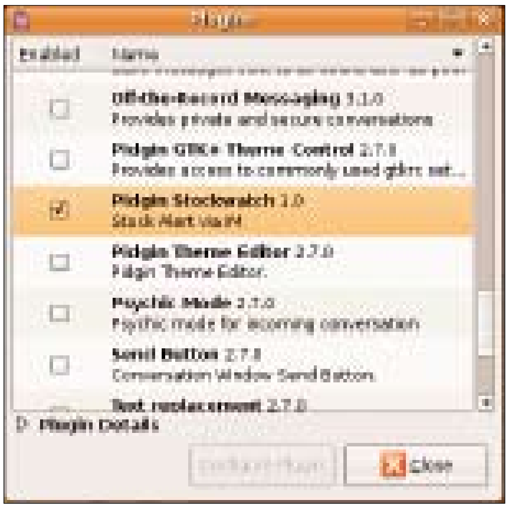
To Alert or Not To Alert

It remains to be seen whether the price fluctuations warrant alerting the user; this is handled by the `check` method in lines 122-157. Again, the callback that will potentially use IM to contact the user is passed to the method.

The code in line 129 copies the latest set of data from the `{data}` object entry into the `{refdata}` section of the archive to give the code a reference for comparison purposes later. Instead of just saving the reference itself, the `{ %{ $self->{data} } }` line copies the data held by the `$self->{data}` reference, creates a hash from the results, and returns a reference to it.

If archived data are available before this call, the `for` loop in lines 143-145 would modify the entries for the previous day's price to match the current price. I don't want the code to send a new message every five minutes for a share price that has risen once, but I do want to know if the price continues to change and again exceeds the threshold.

The `noteworthy()` (lines 180-200) method checks to see whether the share price has exceeded the previous day's price (or simply the previous price if an alert has already occurred today). In this case, line 153 calls the `message()` method in line 160 to format the data of all monitored shares and creates a text string. The same line then calls the passed-in Pidgin plugin callback function, which uses IM to alert the user.



Off-the-Record Messaging 3.1.0
Provides private and secure conversations
Pidgin GTK+ Theme Control 2.7.0
Provides access to commonly used gtkrc set...
Pidgin Stockwatch 1.0
Stock Alert via IM
Pidgin Theme Editor 2.7.0
Pidgin Theme Editor.
Psychic Mode 2.7.0
Psychic mode for incoming conversation
Send Button 2.7.0
Conversation Window Send Button.
Text replacement 2.7.0

Figure 4: A mouse click enables the newly installed plugin, which Pidgin then lists in the Plugins menu.

Installation

Popular Linux distributions will offer a package for the Perl interface to Pidgin (e.g., Ubuntu: `pidgin` and `libpurple0`, both version 2.7.0). The `pidgin-stockwatch.pl` plugin script (Listing 1) needs to be made executable and stored in `~/.purple/plugins` below the budding broker's home directory. Note that the `.pl` suffix is required, or Pidgin won't pick it up. The `WatchQuotes.pm` module (Listing 3) must be in a path where the local Perl installation can find it. If necessary, the script code will point to the right location, as you can see from the `use local::lib` instruction in line 5 of Listing 1. What's local::lib, you might ask?

The AnyEvent and AnyEvent::HTTP modules haven't made their way into some popular distributions – and you might need to visit CPAN to retrieve them. To avoid messing up the Linux package manager's clean Perl packages with additional CPAN modules, users who appreciate a tidy system will use the local::lib CPAN module to install them below `~/perl5` in their home directories. After entering:

```
perl Makefile.PL --bootstrap
make test && make install
```

TABLE 1: Ticker Parameters

Symbol	Last closing price	Price of latest trade	Percent change
s	p	l1	p2

in the downloaded local::lib distribution from CPAN, you will probably want to append the output of the command

```
perl -I$HOME/perl5/lib/perl5 -Mlocal::lib
```

to your local `.bashrc` file. After restarting the shell (or sourcing the `.bashrc` file), local::lib sets environmental variables that tell the CPAN shell to install various modules in `~/perl5` under the user's home directory and to point Perl scripts you call to the additional search paths. But, Pidgin remains unaware of this and needs an explicit `use local::lib` instruction to put it on the right track.

Hurdles for Developers

The plugin script in Listing 1 will not run without Pidgin and throws irate error messages at anybody bold enough to try. The messages hint that it is unable to find some GLib functions in the corresponding shared libraries.

According to Pidgin developers, this is normal, although your opinion might differ. To get it working as a standalone script for testing, you can use a temporary workaround and just comment out the `use Pidgin` and `use Glib` lines. At

LISTING 3: WatchQuotes.pm (part1)

```
#############################
package WatchQuotes;
# Mike Schilli, 20100
# (m@perlmeister.com)
#############################
use strict;
use warnings;
use AnyEvent;
use AnyEvent::HTTP;
use YAML qw(LoadFile);

#############################
sub new {
#############################
 my ($class, %options) = @_;

 my ($home) = glob "~";

 my $self = {
  watcher => undef,
  data    => {},
  refdata => {},
  conf_file => ("$home/" .
    "pidgin-stockwatch.yml"),
  conf => {},
  %options,
 };

 bless $self, $class;
}

#############################
sub init {
#############################
 my ($self) = @_;

 my $yml = LoadFile(
  $self->{conf_file});

 for my $e (@$yml) {
  if (ref $e eq "HASH") {
   my ($key, $val) = %$e;
   $val =~ s/%//g;
   $self->{conf}->{$key} =
     $val;
  } else {

   # 2% by default
   $self->{conf}->{$e} = 2;
  }
 }
}

#############################
sub watch {
#############################
 my ($self, $cb) = @_;

 $self->{watcher} =
   AnyEvent->timer(
  after    => 10,
  interval => 300,
  cb       => sub {
   $self->fetch($cb);
  },
   );
}

#############################
sub fetch {
#############################
 my ($self, $cb) = @_;

 my $url =
     "http://"
   . "download.finance.yahoo.
com/d/"
   . "quotes.csvr?e=.csv"
   . "&f=sl1p2&s="
   . join('+',
      sort
      keys %{ $self->{conf} }
     );

 http_get(
  $url,
  sub {
   $self->parse_csv($_[0]);
   $self->check($cb);
  }
 );
}

#############################
sub parse_csv {
#############################
 my ($self, $csv) = @_;

 for my $line (split /\n/,
  $csv)
 {

  my (
   $symbol, $prev,
   $last,   $change
    )
    =
    map { s/[^\w\.-]//g; $_ }
    split /,/, $line;

  next
    unless defined $symbol;

  $symbol = lc $symbol;

  $self->{data}->{$symbol} =
    [ $prev, $last,
   $change ];
 }
}

#############################
sub check {
#############################
 my ($self, $cb) = @_;

 if (!scalar
  keys %{ $self->{refdata} })
 {
  $self->{refdata} =
    { %{ $self->{data} } };
 }
```

least this way, you'll be able to type the line

```
perl -c pidgin-stockwatch.pl
```

and check that the syntax is okay and that the script finds the `WatchQuotes.pm` module and the CPAN modules it needs. If you launch Pidgin in debug mode by entering `pidgin -d`, it will provide detailed output on what is currently going on and you can even add more messages from within the Perl plugin by using:

```
Purple::Debug::info("stockwatch",
                    "Plugin loaded.\n");
```

At least that way you can see what the plugin is doing and confirm that Pidgin finds it. If this works, and if Pidgin at least runs the `plugin_init()` routine in lines 28-43 of Listing 1, the plugin will appear as the name specified in line 11, `Pidgin Stockwatch`, in Pidgin's *Tools | Plugins* menu (Figure 4). Clicking the checkbox on the left enables the plugin and runs its `plugin_load()` routine. For test purposes, you can disable the plugin again, after which Pidgin will call the `plugin_unload()` routine in line 34.

"Popular distros offer a package for the Perl interface to Pidgin."

After doing all of this, the budding broker can then populate the `~/.pidgin-stockwatch.yml` configuration file with ticker symbols for the shares to be monitored, assign percentage values, or accept the default of two percent. After restarting, Pidgin works with the updated values.

On a quiet day at the stock exchange, you will probably forget that the plugin has been installed at all. But as soon as the stocks go on a roller coaster, you'll be alerted and can quickly contact your favorite broker and engage in panic buying or selling. ■■■

LISTING 3: WatchQuotes.pm (part2)

```

 for my $stock (
  keys %{ $self->{data} })
 {
  if (
   $self->noteworthy($stock))
  {
   $self->{refdata} =
     { %{ $self->{data} } };

   # reset &apos;prev&apos;
   for my $s (
    keys
    %{ $self->{refdata} })
   {
    $self->{refdata}->{$s}
      ->[0] =
      $self->{data}->{$s}
      ->[1];
   }

   $cb->($self->message);
   last;
  }
 }
}

#############################
sub message {
#############################
 my ($self) = @_;

 my $msg = "\n";

 for my $stock (
  keys %{ $self->{data} })
 {
  my ($prev, $last, $change)
    = @{ $self->{data}
     ->{$stock} };
  $msg .=
"$stock: $last $change%\n";
 }

 return $msg;
}

#############################
sub noteworthy {
#############################
 my ($self, $stock) = @_;

 my $price_ref =
   $self->{refdata}->{$stock}
   ->[0];

 my $price_now =
   $self->{data}->{$stock}
   ->[1];

 my $change_percent = abs(
  (
   $price_now - $price_ref
  )
 ) / $price_ref * 100;

 return ($change_percent >
    $self->{conf}->{$stock});
}

1;
```

TABLE 2: Example Plugins

Name	Description
Language Translator	Translates English to other language
Olack	Live chat for website
pidginTeX	Renders math expressions
Extended Preferences	Provides often-requested preferences

INFO

[1] Pidgin: *http://pidgin.im*

[2] Listings for this article: *http://www.linux-magazine.com/Resources/Article-Code*

[3] Perl Object Environment: *http://poe.perl.org*

[4] AnyEvent: *http://software.schmorp.de/pkg/AnyEvent.html*

[5] Egan, Sean. *Open Source Messaging Application Development: Building and Extending Gaim*. Apress, 2005. ISBN 1-59059-467-3

[6] A short guide to writing Pidgin plug-ins in Perl: *http://developer.pidgin.im/doxygen/dev/html/perl-howto.html*

[7] Sample plugin in Perl: *http://code.google.com/p/pidgin-knotifications/downloads/detail?name=knotifications.pl&can=2&q=*

[8] More third-party plugins: *http://developer.pidgin.im/wiki/ThirdPartyPlugins#DevelopmentofThird-PartyPlugins*

[9] Finance::YahooQuote: *http://search.cpan.org/~edd/Finance-YahooQuote/*