Stephen B. Jenkins

Making Servers Dynamically Configurable

In the course of my job at the Aerodynamics Laboratory, I typically have to write two or three TCP/IP servers a year. They can range from simple, single-client status applications to complex forking daemons servicing multiple clients that are critical to the execution of our experiments. Regardless of size, all of these servers have two things in common: They are expected to run unattended for long periods, and they need to have their configurations changed from time to time. By configuration, I mean things such as verbose mode (on or off), error reporting (off, to a log file, or via e-mail), and current working directory. I've found that the best way to provide for these types of changes is to make the servers dynamically configurable, rather than the traditional method of requiring the application to be stopped and restarted with new command-line or configuration-file values. Because most of my code is running in a web-based environment already (see "Reference" for a previous article I've written about this environment), it makes sense for me to use the Web for this task as well. And while I'm going to all this trouble, I may as well add some status-reporting abilities.

A Simple Example

Because it's easier to explain this type of concept by example, I've included complete programs for a simple server and a command-line client to test it, as well as some snippets of code for a CGI configuration editor. I'll explain the server and test client first, then move on to the CGI program. Please note that in order to keep this example as simple as possible, I've decided to ignore several important issues that I'll discuss after the basics have been covered.

The Server

The program shown in Listing 1 (available electronically at http://www.tpj.com/source/) is a simple server that returns a date and time string when a connecting client sends it a *time* message. After the usual preamble to *use* the pragmas and modules, I create an anonymous hash and populate it with the default configuration settings. I then set up the server socket and block, waiting for clients to connect. When that occurs, the *nummess* counter is incremented, and the incoming message is compared to the three types of requests that the server understands: *time*, *status*, and *config*. If the client is requesting the time, a string is created and sent off, the socket is closed, and the message and its time are recorded in the configuration hash. The other two types of requests are more complex as well as more interesting.

Stephen is the senior programmer/analyst at the Aerodynamics Laboratory of the Institute for Aerospace Research, National Research Council of Canada. For more information, go to http://www.erudil.com/.

If the client is requesting the current status of the server, *Data::Dumper* is used to serialize the master configuration hash into a single string; this will allow clients to reconstruct the hash just by using *eval* on the *Dumper* output. The configuration string is sent to the client, and as before, the socket is closed and the message and its time are recorded.

If the client is requesting a configuration change, each key of the configuration hash is used to search the incoming message for *key=value* pairs. If the two keys match, the configuration hash is updated with the new value. The configuration message and its time are stored in their own location in the master hash. Once again, the socket is closed and the message and its time are recorded.

Lastly, if the message doesn't contain an understandable request, the server responds to the client with a polite and informative reply.

The Test Client

The test client shown in Listing 2 (available electronically at http://www.tpj.com/source/) is much simpler than the server because the user is expected to do most of the work.

After the appropriate *use* statements, the client's TCP/IP information is initialized, and a *while* loop is used to block waiting for input from STDIN. When a command is entered, a connection is made to the server and the user's input is sent. The reply is read and displayed, the socket is closed, and the *while* loop waits for more input.

Putting Them Together

The easiest way to see how everything works is to run the server and client programs in separate terminal or command windows. Try entering *time* and *status* requests in the client to ensure that it and the server are running properly. You should see the request echoed in the server window and the response in the client window. Next, enter a configuration change request such as *config verbose=0 defpath=/foo/bar/*. You can use a *status* request or the server's console output to verify your changes. That's it. You now have a dynamically configurable server, although it's rather awkward to use with the test client.

The biggest complication for this technique in the real world is security

The Web-Based Client

Since the reason for adding dynamic configuration to the server was to make life simpler, the next step is to replace the command-line client with an easy-to-use web-based client. Because of space limitations and all the boilerplate code needed for CGI programs, I'll only show snippets of the most important parts of the web client. The complete program, as well as the server and test client programs, is available at http://www.tpj.com/source/.

In this first code block, I check to see if the form is being submitted. If not, the program calls *getconfig()*.

```
#
# check for submit button
#
if( param('Submit') ) {
    &handlesubmit($remotehost,$port);   # this never
                                        # returns
}

#
# otherwise, get the current config
#
my $config = &getconfig($remotehost,$port);
```

In this routine, a connection is made to the time server and the current configuration structure is requested with a *status* command:

```
sub getconfig {
  my $remotehost  = shift;
  my $port        = shift;
#
# connect to the server and get the serialized
# configuration structure
#
  my $socket = IO::Socket::INET->new(
    PeerAddr    => $remotehost,
    PeerPort    => $port,
    Proto       => "tcp",
    Type        => SOCK_STREAM )
  or &htmlexit("<h2>Couldn't connect to
                 server.pl : $@\n</h2>");

  print $socket "status\n";
  my $status = <$socket>;
  chomp $status;
  close($socket);

#
# turn the Dumper output back into a data structure
#
  if( $status =~ /^(\$VAR1[^;]*;)$/ ) {
                    # keep -T happy
    $status = $1;
  } else {
    &htmlexit("<h2>ERROR - untaint of status
                    message failed\n</h2>");
  }
  my $VAR1;
  eval( $status );
  if( $@ ) {
    &htmlexit("<h2>ERROR - eval of status message
                           failed:$@\n</h2>");
  }

  return $VAR1;
}
```

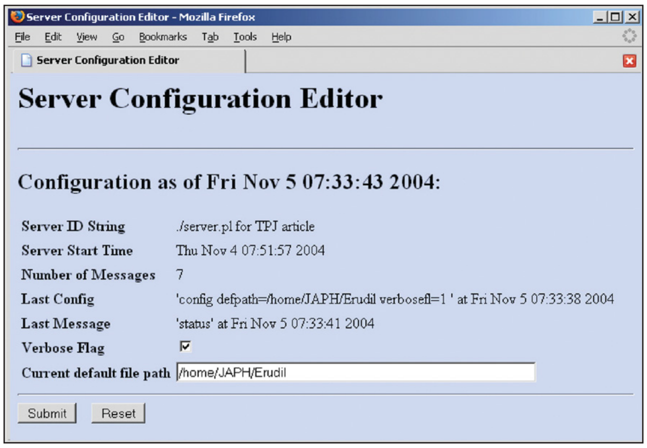
After the reply is read and the socket closed, the string is untainted and then *eval*ed to recreate the configuration hash. Control then returns to the main block where the hash data is munged into an HTML table, with the read-only parameters as text strings and the modifiable parameters as HTML elements:

```
my $verbosehtml = checkbox(
    -name    => 'verbosefl',
    -label   => '',
    -checked => $config->{'verbosefl'}
);

my $defpathhtml = textfield(
    -name      => 'defpath',
    -size      => 60,
    -maxlength  => 132,
    -value     => $config->{'defpath'}
);

my $html = "";

$html .= start_form( -method => 'POST');

$html .= qq{
  <h2>Configuration as of $localtime: </h2>
  <table cellpadding=2>
    <tr><th align="left">Server ID String</th>
      <td>$config->{'serverstr'}</td>
    </tr>
    <tr><th align="left">Server Start Time</th>
      <td>$startstr</td>
    </tr>
    <tr><th align="left">Number of Messages</th>
      <td>$config->{'nummess'}</td>
      </tr>
      <tr><th align="left">Last Config</th>
        <td>$confstr</td>
      </tr>
      <tr><th align="left">Last Message</th>
        <td>$messstr</td>
      </tr>
      <tr><th align="left">Verbose Flag</th>
        <td>$verbosehtml</td>
      </tr>
      <tr><th align="left">Current default file
                                    path</th>
        <td>$defpathhtml</td>
      </tr>
    </table>
  };
```

Submit and Reset buttons are created and added to the form, and the whole thing is sent to the user's browser to appear as it does in Figure 1.

After the user has modified the configuration parameters and submitted the form, the CGI client is executed again—this time calling *handlesubmit()*:

```
  sub handlesubmit {
    my $remotehost      = shift;
    my $port            = shift;

  #
  # build the config string from the HTML elements
  #
    my $configstr = "config ";

    foreach my $option ( qw( defpath ) ) {
      # would normally have several options
      $configstr .= "$option=$_ "
                 if( $_ = param($option) );
    }

    foreach my $flag ( qw( verbosefl ) ) {
      # would normally have several flags
      $configstr .= param($flag)
                 ? "$flag=1 " : "$flag=0 ";
    }

  #
  # set up the connection to the server
  # and send the config string
  #
    my $socket = IO::Socket::INET->new(
      PeerAddr     => $remotehost,
      PeerPort     => $port,
      Proto        => "tcp",
      Type         => SOCK_STREAM )
    or &htmlexit("<h2>Couldn't connect to
                  server : $@\n</h2>");
    print $socket "$configstr\n";
    close($socket);

    &htmlexit("<h3>Submitted:</h3>
               <h2>$configstr</h2>");
  }
```


Server Configuration Editor - Mozilla Firefox

Server Configuration Editor

Configuration as of Fri Nov 5 07:33:43 2004:

Server ID String	./server.pl for TPJ article
Server Start Time	Thu Nov 4 07:51:57 2004
Number of Messages	7
Last Config	'config defpath=/home/JAPH/Erudil verbosefl=1 ' at Fri Nov 5 07:33:38 2004
Last Message	'status' at Fri Nov 5 07:33:41 2004
Verbose Flag	✓
Current default file path	/home/JAPH/Erudil

Figure 1: CGI client web page.

It's so easy that you can even allow knowledgeable end users to make the changes in your absence

This routine converts the form element information into a string of *key=value* pairs, with the *config* command prepended. The CGI client once again connects to the time server, sending it the configuration change request. Finally, the configuration string is displayed in the user's browser as a confirmation message.

The Real World

In the real world, things are quite a bit more complex, of course. All of the socket I/O must have appropriate timeout handling, and the server's responses to configuration messages should reflect their validity. The biggest complication for this technique in the real world, however, is security. The messages need much more thorough taint checking and the server should only allow clients from trusted IP addresses to connect. (In my case, I have a small range of addresses belonging to a private network protected by a local firewall for the wind tunnel control room, all hidden behind a corporate firewall.) You cannot rely solely on the security built into Apache to protect your system, since anyone can bypass it completely by creating a simple program like the test client.

To improve both security and robustness, I usually restrict the values of the most sensitive configuration parameters by setting up hashes whose keys are the acceptable values. That way I can easily check the incoming values like this:

```
  %pathgoodvalue =  ( '/tmp'     => 1,
                      '/usr/tmp  => 1 );
  #
  # put code here to read the config request string
  # and isolate the value
  #

  if exists $pathgoodvalue{$requestvalue} {
```

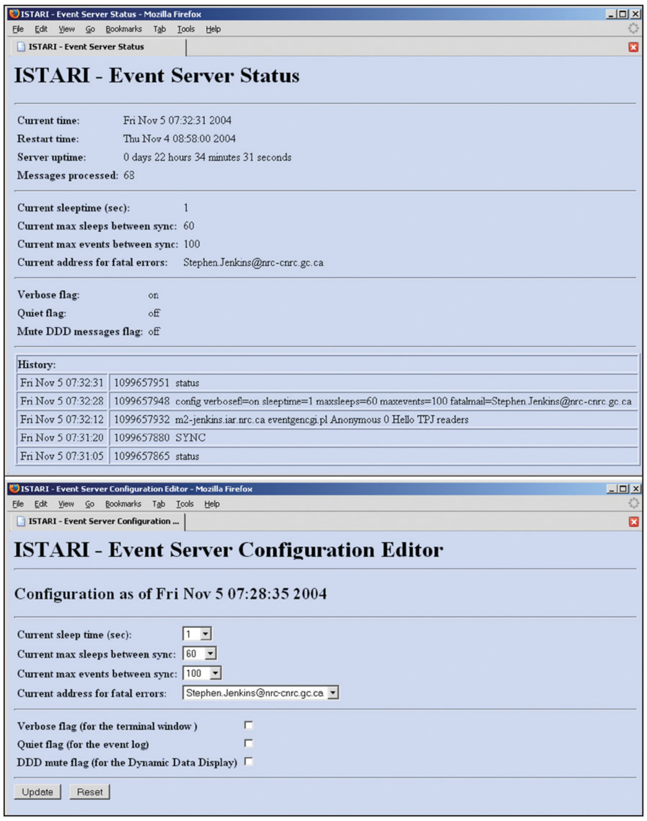
If the hashes containing the acceptable values are placed in a module, they can be included in the CGI program where their keys can easily be used in the creation of HTML select elements.

Although the *eval* of the *Data::Dumper* output in the web client may set off warning bells in the mind of some CGI programmers, this is one part of the process that is actually not as dangerous as it first appears. For this to be a security risk, someone would have had to compromise the network behind the firewall, steal the server computer's IP address, and then mimic the server program's behavior. If that has happened, there are much bigger things to worry about than a corrupted CGI form. (If you're really uncomfortable with the *eval*, you could use either *Storeable* or YAML to serialize and reconstruct the hash, since neither of these methods requires runtime code evaluation.)

For the four servers that I've written most recently, I created separate CGI programs for status reporting and configuration modification. This way, I can use standard Apache security techniques to allow everyone to view the status information, but only allow select people in our IT group to modify the configuration. See Figure 2 for an example of the web pages for one of my real-world servers.

The corresponding master configuration hash for this server is:

```
my $config = { 'serverstr' => "$0 using Perl $]",
               'starttime' => time,
               'numevents' => 0,
               'quietfl'   => $opt_q,
               'verbosefl' => $opt_v,
               'dddofffl'  => $opt_x,
               'sleeptime' => 1,
               'maxsleeps' => 60,
               'maxevents' => 100,
               'fatalmail' => $mailaddrs[0],
               'history'   => [],
             };
```



ISTARI - Event Server Status - Mozilla Firefox

ISTARI - Event Server Status

```
Current time:          Fri Nov 5 07:32:31 2004
Restart time:          Thu Nov 4 08:58:00 2004
Server uptime:         0 days 22 hours 34 minutes 31 seconds
Messages processed:    68

Current sleeptime (sec):             1
Current max sleeps between sync:     60
Current max events between sync:     100
Current address for fatal errors:    Stephen.Jenkins@nrc-cnrc.gc.ca

Verbose flag:              on
Quiet flag:                off
Mute DDD messages flag:    off
```

History:	
Fri Nov 5 07:32:31	1099657951 status
Fri Nov 5 07:32:28	1099657948 config verbosefl=on sleeptime=1 maxsleeps=60 maxevents=100 fatalmail=Stephen.Jenkins@nrc-cnrc.gc.ca
Fri Nov 5 07:32:12	1099657932 m2-jenkins.iar.nrc.ca eventgencgi.pl Anonymous 0 Hello TPJ readers
Fri Nov 5 07:31:20	1099657880 SYNC
Fri Nov 5 07:31:05	1099657865 status

ISTARI - Event Server Configuration Editor - Mozilla Firefox

ISTARI - Event Server Configuration Editor

Configuration as of Fri Nov 5 07:28:35 2004

```
Current sleep time (sec):            1
Current max sleeps between sync:     60
Current max events between sync:     100
Current address for fatal errors:    Stephen.Jenkins@nrc-cnrc.gc.ca

Verbose flag (for the terminal window )            ☐
Quiet flag (for the event log)                     ☐
DDD mute flag (for the Dynamic Data Display)       ☐
```

Figure 2: Real-world status and configuration web pages.

A few of my servers have one additional complexity. They are daemons that fork off child processes that may stay alive for days or even weeks. In order to propagate the configuration changes to them, the serialized master hash must be sent from the parent process to each of the children via a pipe. The child processes can then update their own copies of the configuration data.

Conclusion

Using this method to make your servers dynamically configurable is simple to implement, and by allowing changes to be made easily and quickly, it exhibits two of the three great virtues: laziness and impatience. In fact, it's so easy that you can even allow knowledgeable end users to make the changes in your absence, reducing telephone or pager interruptions during your time off. The only drawback I have found to this technique is the possibility of security problems; you will have to examine your own environment to decide if the benefits outweigh the potential risks.

Acknowledgment

I would like to thank Paul Fenwick of Perl Training Australia for his assistance preparing this article.

Reference

Jenkins, S.B. "A Web-Based Environment to Support Aerodynamic Testing," *IEEE Aerospace and Electronic Systems Magazine*, January 2004, pg. 3.

TPJ