Using PERL to poll a Web Server
Mark Dickinson 2004

Table of Contents

Document Purpose

This is mainly for my internal use. I don't use PERL much anymore and was starting to find it a major annoyance to have to keep refering back to the manuals when I wanted to play with sockets. So I wrote this so I can just refer to it occasionally for that as needed.

It is not intended to be a PERL socket tutorial, just to jog my memory occasionally. If you are interested in using PERL as a server application you should probably refer to other downloads from my website which include such things as a fully functional client/server application in PERL.

This tutorial only covers the minimum steps needed for a transient client (one write, one read, stop) to poll a web server; it is not a PERL sockets tutorial.

Disclaimer

This document was created for my own use. In the possibility it may be usefull to others it is freely available.

However all information contained in this document *is based on trial and error* implementations I was using on a local web server, and has not been checked against any standards documents.

You use the information contained here at your own risk.

Redistribution

Change History

2004 September – wrote the HTML minibook

2009 January – converted to a PDF document and made some formatting changes. Deleted a few irrelevant sections and added some more clarification to various sections.

1. Why would you want to, What benefit can be gained ?.

1.1 Why would you want to poll a web server ?

There are many reasons to poll a web server if the web server is a *customer facing* web server. Primarily it can alert you to possible problems before the customers do. It can also be usefull for tracking response time data over time, or detecting slowdowns in web server responsiveness.

It is also usefull for testing environments, if you have a lot of dynamic pages you can use pre-built PERL scripts to perform testing or regression-testing against the site when you make changes to it.

1.2 What benefit can be gained ?

With a regulary executing script to poll key areas of your web server you can achieve the following benefits

- Alerts when the site is down, you may find out before your customers do
- Alerts when the site has slow response times, you can start taking action before your customers complain
- Response time recording, you may want to keep historical data on **real** response times from your site
- Detection of server busy conditions, where the server may start responding with retry or server busy errors you will be alerted for these
- Site testing. If you have created a set of scripts that test the main pages on your site you will quickly know when they start failing

1.3 Seems like a good idea, what do I need to do

Follow though this mini-book and review the chapters you need. For general *I'm alive tests* a simple static page retrieval may be all you need. If you want to test how long server side scripts or servlets take to run you will need to read the sections on submitting GET data and performing POST requests also.

You should not jump in headfirst and start monitoring web servers. Please refer to the guidelines provided on how, and more importantly from where, you should be monitoring those web servers you consider to be critical.

2. GET page request for a static page, and timing it

Retrieving a static page is the most common test of a web servers availablility. By retrieving a static fixed page that doesn't change (normally a never changing test page set aside for this purpose) you can guarantee that the page exists and the server will return it if the server is able to. It also guarantees a known data return to be used for response time recording, as its an unchanging page the only deviation in response time will be caused by web server load or load caused by other jobs on the machine hosting the web server.

A static page is preferable for timing web page responses. Pages that are returned as a result of scripts or other dynamic processing may to too affected by other workload issues on the web server machine.

2.1 GET request syntax for a static page

To retrieve a static page from a web server all that is required is to connect to port 80 on the web server and use a GET request to identify the static page needed.
The format of the static page GET request is below

```
GET /pagename HTTP/1.0
newline
newline
```

The two newlines **(inserted blank lines, don't type the word newline please)** after the GET request are required. They indicate to the web server that we have finished sending http header data to it. Also note the leading slash (/) character in the *pagename* to be retrieved. This is required as the web server expects to serve up the page from a known location, so we will specify the web server documentroot.

2.2 Example PERL script to retrieve a page and time it

This script is pretty simple. It connects to the web server, requests a page, checks the http return code and the response time taken. Any errors it writes a message.

```
#!/usr/bin/perl
#
# check_webserver_response.pl : PERL
#
# This requests a page known to exist from the web server
# and checks the http response code for errors.
#
# If either an http error code is returned, or a socket
# connection failure occurs, then we will write out an
# error message.
#
# If there is no problem, the script returns nothing.
#
# When invoked the user may allow to default, or override
# the hostname, port number and page to be retrieved.
# If the user provides parameters then all three must
# be provided, or the defaults will be used.
```

```
#
# Syntax: check_webserver_response "hostname" "portnum" "/pagename"
#
use IO::Socket;

# 0. - See if user is providing their own aguments
$hostname = $ARGV[0];
$hostport = $ARGV[1];
$hostpage = $ARGV[2];
if ( $hostpage eq "" ) {   # need all three, so if this is missing default all
   $hostname = "localhost";
   $hostport = "80";
   $hostpage = "/alivecheck.html";
}
$alertsecs = 5;    # more than a five second response indicates a slowdown

# 1. - create a socket for the web server connection
my $sock = new IO::Socket::INET (
                                  PeerAddr => $hostname,
                                  PeerPort => $hostport,
                                  Proto => 'tcp',
                                );

# 2. - if the socket was created request the test page
#      and read the page response. we don't really care
#      about the contents of the page so discard
#      everything but thr HTTP response line.
if ( defined $sock ) {

# 3. - request the test page from the web server
   print $sock "GET $hostpage HTTP/1.0\n\n\n";
   $starttime = time();      # record start time

# 4. - read the web server responses until the web server
#      has finished sending the page and closes the connection.
   # note: if, chomp and split will assume $_, so read to that
   while ( defined( $_ = <$sock> ) ) {
      next if not /^HTTP(.*)/;   # discard all but the http response line
      chomp;

# 5. - check for the OK response codes, error msg if not ok
      ($ver,$respcode,$other) = split(/ /);
      if ( ($respcode ne "200") && ($respcode ne "304") ) {
         print "Got response code error $respcode, host $hostname, port $hostport, page
$hostpage \n";
      }
      # else OK, write nothing
   }
   $endtime = time();    # record end time
# 6. - remember to close out our socket
   close($sock);
# 7. - check the response time taken
   $secstaken = $endtime - $starttime;
   if ( $secstaken gt $alertsecs ) {
      print "Slow response, took $secstaken seconds to get page $hostpage from host
$hostname, port $hostport \n";
   }
}

# 2.b - ELSE no socket, the connect failed
else {
   print "Could not connect to port $hostport on $hostname \n";
}
```

```
# Done
exit 0;
```

Suggested enhancements:
Some things I do in various sites are e-mail if any problems, and generate alerts to third party alert monitoring tools (or to my alert toolkit). Enhancements will be site dependant so I havn't clutted up the example above with them.

3. GET script request for a dynamic page

Retrieving a dynamically build page may be done using either the GET or POST request methods.

This chapter covers the **GET** request method, the next chapter the POST request method.

To see if the web server is running and serving pages you would generally retrieve a static page. You would only poll/health-check a dynamically built page it you were testing for an error condition that could be uniquely identified by that page, for example you may request a *known to exist* name from an address book and check that the correct page was returned as a test of whether the database was online or not.

As a general rule dynamically built pages should not be used for server response time reporting; responses to these are dependant upon external factors such as a possibly slow application, and the probability the databases that are used are remote to the web server machine itself, you will not be timing the web server so why bother.

Actually there are reasons why you would bother, these are covered in the guidelines at the end on where and why monitoring scripts should be run.

3.1 GET request syntax for a dynamic page

To retrieve a dynamic page from a web server; connect to port 80 on the web server and use a GET request to identify the page needed, the only difference between a dynamic GET request and the static GET request covered in the previous chapter is that you need to provide the data input fields that would normally be filled in by a user in the URL youe script is using.

The format of the static page GET request is below, note that the data fields are seperated from the ULR part with the **?** symbol, each data field is seperated with the **&** character and any spaces need to be replaced by %20. There are other special characters that also need to be replaced by %nn notation but I am not covering those here; this is not an HTML protocol tutorial.

```
GET /cgidir/pagename?field1=value1&field2=value%202 HTTP/1.0
newline
newline
```

The two newlines **(inserted blank lines, don't type the word newline please)** after the GET request are required. They indicate to the web server that we have finished sending data to it. Also note the leading slash (/) character in the *pagename* to be retrieved. This is required as the web server expects to serve up the page from a known location, so we will specify the web server documentroot.

Also in the example above we are running a script in the cgi directory. If your web server is using php do generate dynamic pages rather than cgi scripts thats fine, just use a valid URL with the correct data that is expected to be passed to the page.

3.2 Important considerations

It is insufficient when testing the response to a dynamic page to simply check the http response

header. With dynamic pages even if the application on the web server was unable to process you request due to bad user input it is presumably going to send back a perfectly legal html page with an application error message in it. **What does this mean to you ?**, if you are doing dynamic page retrievals from the web server for health-check tests you will have to parse the output of the entire page data stream returned checked for known OK or failure indicators.

3.3 Example - Parsing a GET dynamic page

This example is virtually identical to the sample PERL script in the previous chapter. The only difference of note is that as we are getting a dynamic page page we must check the contents of the page itself rather than check for the http response code. We know the page will have a Phone Number:xxxxxxx so check for that and validate the test phone number we query.

```
#!/usr/bin/perl
# Request phone number for Mark Dickinson, expect it to be returned
# somewhere on the page as "Phone Number:1234567".
# So we search for Phone Number:, then check the phone number returned.
# Write a messaeg if an error, else write nothing.

use IO::Socket;

$allOK = NO";
my $sock = new IO::Socket::INET (
                                  PeerAddr => 'localhost',
                                  PeerPort => '80',
                                  Proto => 'tcp',
                                );
if ( defined $sock ) {
   print $sock "GET /cgi-bin/find_phone.pl?name=Mark%20Dickinson HTTP/1.0\n\n\n";
   while ( defined( $_ = <$sock> ) ) {
      next if not /Phone Number:(.*)/;   # discard all but the Phone Number (if
returned)
      chomp;
      ($xx,$yy) = split/:/;
      if ( $yy eq "1234567" ) {
         $allOK = "YES"
      }
      else {
         print "Incorrect phone number returned from query !. \n";
         $allOK = "ZZ";  # stop no phone num returned msg
      }
   }
   close($sock);
   if ( $allOK = "NO" ) {
      print "No phone number returned from query request !. \n";
   }
else {
   print "Could not connect to port 80 on localhost \n";
}
exit 0;
```

4. POST script request for a dynamic page

Retrieving a dynamically build page may be done using either the GET or POST request methods. This chapter covers the **POST** request method, the previous chapter covered the GET request method.

You would only poll/health-check a dynamically built page it you were testing for an error condition that could be uniquely identified by that page, for example you may request a *known to exist* name from an address book and check that the correct page was returned as a test of whether the database was online or not.

4.1 POST request syntax for a dynamic page

A major difference between GET and POST methods is that the POST method requires a lot more information to be provided by your script. It is not enough to just send a POST request line to the web server, you must also at a minimum send the Content-Type and Content-Length fields, the Content-Length field indicates to the web server how much data you are sending as part of the POST request.

So to retrieve a dynamic POST page from a web server; connect to port 80 on the web server as per a GET request, send the POST request to identify the page needed and pass the additional content lines required to emulate the form fields and values being tested to the script handling the POST request.

The format of the POST request is below, note that the each data field is seperated from the next with the **&** character and any spaces need to be replaced by +, this is different from the GET where spaces were replaced by %20. **There also must be a blank line between the end of the header information and the data to be passed** as well as the normal two blank lines at the end of the request header.

```
POST /cgidir/pagename HTTP/1.0
Content-Type: applicatiom/x-www-form-urlencoded
Content-Length: 34
newline
field1=value1&field2=value+2
newline
newline
```

The two newlines **(inserted blank lines, don't type the word newline please)** after the POST request are required. They indicate to the web server that we have finished sending data to it. Also note the leading slash (/) character in the *pagename* to be retrieved. This is required as the web server expects to serve up the page from a known location, so we will specify the web server documentroot.

Also there must be the blank line between the request header information and the actual data field values.

And to make it a little more difficult the content length must be the last field and ***must*** be set correctly or it just won't work. As you can see from the example in section 4.3 that I got working this length seems to be the text length, plus the 5 new line characters (which includes the one on the Content-Length line), plus one for luck (**I'm guessing**, I wrote this document in 2004 before I was

internet connected so everything here was from trial and error on my own server; I don't intend to research the subject anymore so I will leave that to you).

4.2 Important considerations

It is insufficient when testing the response to a dynamic page to simply check the http response header. With dynamic pages even if the application on the web server was unable to process you request due to bad user input it is going to send back a perfectly legal html page with an application error message in it. **What does this mean to you ?**, if you are doing dynamic page retrievals from the web server for health-check tests you will have to parse the output of the entire page data stream returned checked for known OK or failure indicators.

4.3 Example - Parsing a POST dynamic page

This example is virtually identical to the sample PERL script in the previous chapter discussing the dynamic GET request. The only difference of note is that we are using the POST request type so need to write more data to the web server to provide the information required by the web server.

Remember that when getting a dynamic page page we must check the contents of the page itself rather than check for the http response code. We know the page will have a Phone Number:xxxxxxx so check for that and validate the test phone number we query.

```
#!/usr/bin/perl
# Assuming find_phone.pl is a perl script to provide the phone
# number of the person requested we emulate a form field called
# "name" containing the value "Mark Dickinson".
# We expect somewhere in the html data returned the text
#  'Phone Number:1234567'
# the known 1234567 we can test for.
# Write a message if an error, else write nothing.

use IO::Socket;

$allOK = NO";
my $sock = new IO::Socket::INET (
                                  PeerAddr => 'localhost',
                                  PeerPort => '80',
                                  Proto => 'tcp',
                                );
if ( defined $sock ) {
   print $sock "POST /cgi-bin/find_phone.pl HTTP/1.0\n";
   print $sock "Content-Type: application/x-www-form-urlencoded\n";
   print $sock "Content-Length: 25\n\n";
   print $sock "name=Mark+Dickinson\n";
   print $sock "\n\n";
   while ( defined( $_ = <$sock> ) ) {
      next if not /Phone Number:(.*)/;   # discard all but the Phone Number (if
returned)
      chomp;
      ($xx,$yy) = split/:/;
      if ( $yy eq "1234567" ) {
         $allOK = "YES"
      }
      else {
         print "Incorrect phone number returned from query !. \n";
         $allOK = "ZZ";  # stop no phone num returned msg
      }
```

```
   }
   close($sock);
   if ( $allOK = "NO" ) {
      print "No phone number returned from query request !. \n";
   }
else {
   print "Could not connect to port 80 on localhost \n";
}
exit 0;
```

5. Recomendations and Guidelines on what to test.

5.1 Determine what you really want to achieve

The first thing to do is decide what you are responsible for. If you are not responsible for something then don't try to run scripts to measure those things, you will only needlessly tie up the server.

- **Are you responsible for network response time management ?**, if not then only run any health check scripts on the local server.
- **Are you responsible for database tuning and database response times ?**, if not then only use scripts that retrieve static pages.

Also you need to decide what you are trying to measure, this determines where to run the script to do the measurement.

- **Check the web server is up**, use a static page test running on the local server. Use a static page with mininal output to parse as a response
- **The web server is responding in a consistent timeframe**, use a static page test running on the local server. Use a static page with minimal output to parse as a response response
- **The web server and database servers are responding in a consistent timeframe**, use a dynamic page GET request test script running on the local server. Use a page with mininal output to parse as a response, the returned page should have clearly defined keywords you can parse for to detect sucess or failure of the request. There are additional considerations for database tests to take into account as discussed below
- *Customer experience* measurements are the only tests that need to be run other than on the local web server machine. These tests need to be run remotely, as close to the real customer interface to the web server as possible; on a site external facing firewall, or even on a server external to the site itself.

5.2 Local Server Checks

Local server checks are the easiest, they run on the same machine as the web server itself. They can ensure that the web server is running, and that static pages can be delivered in a consistent timeframe. By running on the local machine you eliminate any network overhead so get a true indication of the web server responses.

The benefit of doing local server checks is that when someone says the web server is delivering pages too slowly, you can prove it is responing normally and its a database (if an external database) or network slowdown problem.

Doing checks against a static page will let you determine if the web server is available, and give you timings on web server responsiveness to detect web server slowdowns.

Doing checks against a dynamicly built page that does database queries will, as long as you only repeat the same query each time, give you timings on combined web server responsiveness and

database responsiveness; so in conjunction with a static test will let you know if it is a slowdown on the webserver or the database (or no slowdown at all so a network problem between the user and your server, or an issue on the users PC, or a user issue; but definitely not your problem).

5.3 Remote checks against the web server

Why would you want to do this ?. Really you would only wish to do this if you were responsible for network performance issues or customer experience monitoring.

5.3.1 Network problem indications

For network response issues **only use the static page method** of testing. This is the only method that can indicate a slowdown between the machine you are running the test scripts on and the machine running the web server.

You must us ethe static page method as simply put you don't care about any possible database slowdowns (which is why you are testing against a static page) as you are only testing network response times to the server not whats going on behind the server. ***However*** if there is a slowdown on the web server itself that will skew your response times, so obviously this needs to be run in conjunction with local tests to determine if it is a web server slowdown or a network one.

5.3.2 Customer experience monitoring, just don't

For customer experience monitoring, *I personally consider this a waste of time*.

Yes it is used, but what does it show. That there may be a problem somewhere in the world (not necessarily on your site), it doesn't give a clue as to where so a lot of people spend a lot of time chasing something that may clear itself up before anybody can locate it.

A customer experience test that indicates a slowdown or failure to complete could be caused by slow or broken databases, slow or broken (re-routed) networks, slow web server, slow firewalls; **or** if you are testing fully from a completely external site connection as you should be it could even be that the internet provider connected to may be slow or the PC you are using has just failed out some memory or CPU. So where do you look first, on most sites nobody knows.

It may be possible to design a set of explicit requests to check each step of the path the customer would take through your application to narrow down a problem, but the application will have to have been designed with those checkpoints installed within it. Unless such a clear path of component checking has been designed into your application (ie: dummy response check at firewall, and every intermediate hop to the web server, static page check, dynamic page check on database1 *only*, on database2 *only*, and even some way of testing the ISP equiptment etc... then it is pointless to do customer experience testing. And if you have a slowdown in a transaction that traverses multiple databases you **must** have checks to test each individual database as a unique element, don't expect your support staff to figure out what database out of the many involved is slowing down; either make your checks tell them exactly where the problem is or don't waste their time running customer experience checks.

If you are serious about customer experience monitoring you will have already spent millions on active network and database performance tools and not need a polling script.

5.3.3 Customer experience monitoring, just don't, example

An example I can give is a large ***international*** financial institution I used to work at. Their customer experience monitoring was done from the head office *in a different country* to the one in which the web server itself was running, and it was done by the business groups (not the IT groups) and used by the business for availability reporting.

It also produced real time alerts to the business areas when there were issues, that the IT staff didn't get to see until after literally years of lobying.

And when there were slowdown issues reported by that tool, ok sometimes they were happening; but 99% of the time all the *local* server, web server and database checks were just fine, no slowdown at all. Try to convince a business group that everything is fine and the problem could be a network slowdown in their offices, in their firewall, in their ISP, in the international carrier network, or even that they should stop the running virus scan on the monitoring pc... no chance, from thousands of miles away they saw a slowdown reported in their tool and it must therefore be our problem, even though we had local response time checks to prove it wasn't I'm afraid IT produced numbers don't count with a business group.

So just don't allow anyone to implement end-to-end customer experience monitoring unless the solution doing the monitoring can report response times from every component in the network chain... end-to-end monitoring is useless unless you can get statistics from the hundres, possibly thousands, of pieces in the middle as well.

In this example the tool used only did end-to-end tests, it had no visibility of any equiptment between one country and another. So while it could accurately report on a user experience slowdown it couldn't identify where and was effectively useless... even more usless than you might think as there were no international users of the application, only local country users; but they installed it in a different country for business SLA reporting anyway.

In Summary: **Do not use third party or even home grown tools to monitor your web server applications** (oops, I've just made this entire tutorial useless for business use). **They imply that the developers have forgotten or overlooked the need for response time instrumentation embedded in the application where it belongs.** Needing to use an external to the application end-to-end monitoring solution implies the web served application being monitored has just not been properly designed. End-to-end monitoring should never be used to replace properly intrumented code.

6. Important Considerations Left out of the documentation

The astute amongst you will have already noted one very large omission in the sample scripts provided; which is not a problem to me as they are only samples.

This mini-book is not a PERL tutorial, **the samples provide the bare minimum to get the job done**. Before implementing the sample scripts into a production environment please consider the following issues. You may like to use your PERL knowledge to customise the sample scripts extensively for your environment.

6.1 Whats missing ?

6.1.1 Timers !.

All the examples presented in this mini-book assume that the web server will eventually provide a response or an error page, that may not always be the case.

In one site I have worked on the http server had so many requests queued up it could take up to 20 minutes to server a page back. Browsers timed out before then, the sample scripts won't. You should implement timers, if an acceptable response time is 30 seconds then throw in a timer to abort after 120 seconds; obviously you know its a problem then so don't wait.
Why was the queue so high you ask, because users were used to almost immediate response times and when a slowdown started to occur all 500+ of them started to make active use of the browser refresh button, repeatedly. The web server never had a chance to recover.

6.1.2 Socket error handlers

The supplied samples assume a response will be recieved, while a socket death should not cause a major concern the scripts don't really handle it. The chapter 2 example is a prime candidate. An early socket close here before reading the HTTP response code would assume all was well. You may want to throw some checks in so that if no HTTP response code is found it is an error, the code in that sample as it stands will report OK if the web server was shutdown and closed the socket (which may be OK ?, a problem and it's being fixed).

6.1.3 SSL connections

SSL connections are not covered here. As noted earlier this is not a PERL tutorial, **nor** is it a secure sockets tutorial. You are welcome to investigate PERL SSL options, or pherhaps the STUNNEL tool for running scripts against SSL encrypted sites in your own time.

6.1.4 Other Stuff

There will be more. The samples are intended as functional samples only, not as recomendations of how you should implement them.