



Sebastian Duda, Fotolia

A Perl script implements a singing, musical Internet

CATCHY LOGS

Instead of just monitoring incoming requests in your web server's log file, a sound server makes them audible and lets you listen to the tune of users surfing the site. **BY MICHAEL SCHILLI**

Whenever I upload a new version of our blog-like newsletter [1], send an email announcement, or update the RSS feed, I tend to check the web server access log to watch the first information-hungry visitors read the latest news and click on the high-res images.

Of course, deciphering the server log entries scrolling by is fairly tedious. It would be much better to monitor the requests in the background and start working on something else in the meantime. One way to do this would be to transform web hits into sound. Many moons ago, I read in *Netscape Time*, by Jim Clark, that the early Netscapers used to output incoming hits via PC speaker after creating a new release [2]. A Netscape browser download for Windows croaked like a frog, the sound of

breaking glass played for Macs, and Unix downloads were announced with a cannon shot. This meant that the Internet pioneers could share the sound of success in their cubicles after the long coding stretch that preceded the launch.

Implementing something like this in Perl is fairly easy. In my case, however, things are not quite as simple because

the web server is somewhere in a hosting provider's server farm. Although the hoster allows *ssh*-based shell access, it can't transmit sound.

Firewall Tunnel

The *boom-sender* script on the hosting provider's shared server monitors the web server's *access.log* file and sends messages for specific URLs through an SSH tunnel to the *boom-receiver* sound server script running on my home PC.

Figure 1 shows the setup: The sound server is a script implemented by the CPAN *POE* module that listens for sound

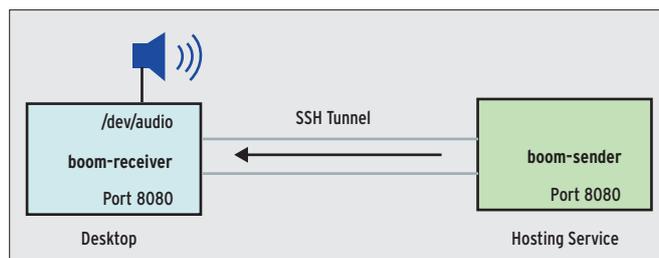


Figure 1: The script that monitors the access logfile on the web server hosted by my provider uses an SSH tunnel to communicate with the sound generator on the desktop at home.

commands on port 8080 of the local machine. Another *POE* script runs provider-side, reacting to changes in the access log and sending messages home as a TCP client. Because my home machine resides

behind a firewall, the boom-sender log checker can't talk to it directly. Instead, a tunnel setup that uses the command

```
home$ ssh -R 8080:localhost:8080
host.xyz-hosting.com
```

on my home PC connects the two dialog partners. The log script on the hosted server just has to send its messages to its local port 8080, and – hey presto – they are whisked away through the tunnel to port 8080 on the home PC as if the firewall never existed.

Sound on Demand

The local sound machine receives names of sound files in this way and proceeds to play them on Linux with the *play* utility from the Sox package treasure trove.

By default, the Play program is included with Ubuntu and can handle both WAV files and MP3s, assuming you configure Ubuntu to support this.

Figure 2 shows the interaction of a test client with the sound server running. The *telnet* command is launched to connect to localhost's port 8080 and receives a greeting from the server and a list of the sound files it has. When the client sends the name of one of these WAV files to the server, the server plays the file. For security reasons, only file names are allowed, rather than paths.

The default location in which boom-receiver looks for these sound files is the current working directory (*.*), specified as *\$SOUND_DIR* in line 9 of Listing 1.

Because it offers a plethora of server and client components that just need to be put together in creative ways, POE is a good choice of server and client

technology. The *poe.perl.org* website and the POE chapter in *Advanced Perl Programming* [3] both offer useful introductions to POE, which requires a non-traditional, event-based programming model that takes some time getting used to.

The server in Listing 1 defines callbacks for the states *ClientConnected* (client has opened a connection), *ClientInput* (client has sent a line of text), and *sound_ended*, the state that handles the clean

up work (described below) after playing a sound.

The sound server handles multiple client connections quasi-simultaneously. The POE component logic takes care of the low-level implementation details and ensures smooth request and error handling behind the scenes. Just like any other POE script, the program code first defines the behavior for any possible events and then calls

Listing 1: boom-receiver

```
01 #!/usr/local/bin/perl -w
02 use strict;
03 use POE;
04 use POE::Component::Server::TCP;
05 use POE::Wheel::Run;
06 use File::Basename;
07 use Log::Log4perl qw(:easy);
08
09 my $SOUND_DIR = ".";
10 my @SOUND_FILES = map { basename $_
11     <$SOUND_DIR/*.wav>;
12
13 Log::Log4perl->easy_init($DEBUG);
14
15 POE::Component::Server::TCP->new(
16     Port => 8080,
17
18     ClientConnected => sub {
19         $_[HEAP]{client}->put("Soundfiles: [".
20             join(" ", @SOUND_FILES) . "]" );
21
22         $_[HEAP]{client}->put(
23             "Ready when you are.");
24     },
25
26     ClientInput => sub {
27         my $client_input = $_[ARGO];
28
29         if( $client_input !~ /\^[\\w.-]+$/ ) {
30             $_[HEAP]{client}->put(
31                 "Illegal input.");
32             return;
33         }
34
35         if( $client_input eq "q" ) {
36             POE::Kernel->yield("shutdown");
37             return;
38         }
39
40         my $msg = sound_play(
41             $_[HEAP],
42             basename($client_input));
43
44         $_[HEAP]{client}->put( $msg );
45     },
46
47     InlineStates => {
48         sound_ended => sub {
49             my ($heap, $wid) = @_[HEAP, ARGO];
50             DEBUG "Deleting wheel $wid";
51             delete $heap->{players}->{$wid};
52         },
53     },
54 );
55
56 POE::Kernel->run();
57 exit;
58
59 #####
60 sub sound_play {
61     #####
62     my($heap, $file) = @_;
63
64     if(! -f "$SOUND_DIR/$file") {
65         return "$file doesn't exist";
66     }
67
68     POE::Kernel->sig(CHLD => "reaped");
69
70     my $wheel =
71         POE::Wheel::Run->new(
72             Program => "/usr/bin/play",
73             ProgramArgs => ["$SOUND_DIR/$file"],
74             StderrEvent => 'ignore',
75             CloseEvent => 'sound_ended',
76         );
77
78     DEBUG "Creating wheel ", $wheel->ID;
79     $heap->{players}->{ $wheel->ID } = $wheel;
80
81     return "Played $file";
82 }
```

```

mschilli@mybox:/mnt/data/big1/mschilli.do.not.delete
$ telnet localhost 8080
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.

Soundfiles: [article-page.wav, forum-page.wav,
             forum-post.wav, image.wav]
Ready when you are.

article-page.wav
Played article-page.wav

forum-post.wav
Played forum-post.wav

blah.wav
blah.wav doesn't exist

foo.wav; rm *
Illegal input.

q

Connection closed by foreign host.
$

```

Figure 2: The test client, telnet, connects with the receiving server, which plays a sound file on demand.

`POE::Kernel->run()` to launch the POE kernel. The kernel runs until the program ends, until a fatal error occurs, or until the user terminates the script.

Don't Dawdle

The `sound_play()` function in line 60 of Listing 1 plays a sound file passed to it by name. It creates `POE::Wheel`, a cog-wheel in the POE system's works that allows the POE kernel to talk to the world outside.

To allow the system to process multiple tasks quasi-simultaneously, Perl code in POE should only run uninterrupted as long as it proceeds at full speed. Any interactions with files, sockets, or other processes obviously cause delays because disk or network access is far slower than processing CPU instructions or accessing RAM, and it would be extremely inefficient to let the CPU sit idle while waiting for these tasks to complete. Instead, they are handed off to wheels, which accomplish them one slice at a time and report results back asynchronously to the POE kernel.

Applying the `play` command to start a new Unix process, passing a short sound file to it, and waiting for it to play takes more than a second. If the script was blocked for this time, it would delay the client response and not be available for new requests.

Instead, a wheel is spun off with the process task, the callback returns immediately, and the POE kernel reassumes control, leaving everything else to run in the background.

The wheel – `POE::Wheel::Run` – expects as parameters an external program to launch, its arguments, and a `StderrEvent` callback, triggered if the process writes anything to its `STDERR` channel. Of course, this is not relevant for the Play program, which does not normally output error messages and simply terminates after playing a sound file. Boom-receiver simply defines a non-existing state for this event, which POE later ignores.

When the wheel notices that the `play` process has terminated, it triggers `CloseEvent` in line 75, assigned to a subroutine in line 48. Then it removes the remaining reference to the wheel from the system, which unleashes the

POE kernel's garbage collector to clean up its remains.

Ideally, the wheel would just launch a process such as `xmms`, which would run permanently, and then occasionally pass sound files to it. However, the POE component for this on CPAN is badly out of date and won't compile with the current version of XMMS. Pity!

Beady-eyed!

Admittedly, the implementation shown here wastes resources on the local machine, but it can indeed convert quasi-parallel requests into sounds. To allow this to happen, the script keeps a reference to the wheel object that generates the sound because POE cleans the object up immediately if nobody takes care of it. The wheel's task of playing the sound does not end at `sound_play()`, because the POE kernel processes it slice by slice after the function terminates. To avoid an untimely demise, while at the same time avoiding keeping wheels for longer than necessary, line 79 saves a reference to the wheel object in the POE session heap with the key `players` and the wheel's ID.

Because the wheel defines a `CloseEvent` with a callback `sound_ended`, POE calls the function defined in line 48 when the sound process terminates; in turn, the function deletes the wheel reference to let POE move in for the kill.

Another issue is that `POE::Wheel::Run` does not automatically clean up terminated child processes, instead leaving

them lying around as zombies on the Unix system. Therefore, line 68 defines a `SIGCHLD` handler that tells the parent process to issue a `wait()` for the terminated child process and prevent it from turning into a zombie.

As soon as a client connects to port 8080 on the `POE::Component::Server::TCP` server component, its state machine changes state to `ClientConnected`. In the callback, `$_[HEAP]{client}` contains a client object whose `put()` method is used by the server to send messages to the client. The server uses `ClientConnected` to inform the connecting client of the available sound files before announcing `Ready when you are`.

Whenever the client sends a line to the server, the server jumps to the subroutine mapped to the `ClientInput` state. The received message is available in `$_[ARGO]`, one of the `@_` argument array fields typical of POE.

To prevent the client from attacking the server with nasty shell commands, instead of sending a sound file as expected, line 29 checks the file name to see whether it contains anything apart from the normal characters and immediately issues an error message and rejects the request in this case.

The client sends the `q` character to indicate that it wants to quit the session; the server then switches to the `shutdown` state, terminating the current client connection but leaving the server running. If the client really does send the name of an existing sound file, the `sound_play` function plays the file and returns a status string, which the server sends via `put()` to the client to confirm successful execution.

End of the Tunnel

At the other end of the tunnel, the POE script (`boom-sender`) in Listing 2 monitors the web server's access logfile. It runs on the hosted machine and uses the POE framework's TCP client component to keep in touch with the server.

Among other things, the `Client::TCP` POE component defines the `ServerInput` and `ConnectError` events; the script jumps to the callbacks for these events if the server sends text back or a connection fails.

Boom-sender uses `InlineStates` to define the `send` state, which uses `put()` to

send a message to the server that was passed in.

Thanks to the *FollowTail* wheel from the POE toolbox, the logfile monitoring session defined in line 29 notices when the web server appends a line to the logfile defined in line 35. Again, it is important to have a reference to the wheel to prevent POE cleaning it up after the *_start* callback terminates.

The reference is kept in the POE session heap under the *tail* key while the session is active – that is, until boom-sender terminates.

Production systems will tend to rotate their logfiles daily; *FollowTail* is prepared for this and jumps to the *got_log_rollover* callback mapped to *ResetEvent* in this case. All this does is write a debug message to let the user know what is going on. Whenever the wheel finds a newly appended line in the log, it changes state to *got_log_line* and executes the matching callback. It uses the CPAN *ApacheLog::Parser* module to analyze the new lines, which have the following format:

```
67.195.37.108 - - [01/Sep/2008:17:25:20]
-0700] "GET /1/p3.html HTTP/
1.0" 200 8678
 "-" "Mozilla/5.0 (X11; U; Linux i686
(x86_64); en-US; rv:1.8.1.4)
Gecko/20080721 BonEcho/2.0.0.4"
```

The *parse_line_to_hash()* function exported by this module returns a hash containing the file requested by the http request under the *file* key.

In line 12, the TCP client component defines an alias (*boom*) for its session. The *FollowTail* wheel, running in another session defined in line 29, can use the following lines to tell the TCP server which sound file it needs to play:

```
POE::Kernel->post("boom", "send",
file2sound($file));
```

Because two different sessions are communicating here, I can't use *yield()* to send the event; instead, I must use *post()* with the alias of the receiving session. Then the name of the WAV file is sent by the POE kernel to the *send* call-

back in the *boom* session as argument *ARG0*. The callback then uses *put()* to send the name to the TCP client in line 21, which in turn passes it on to the sound server – not directly, but to port 8080 on the local machine, and thus through the tunnel to port 8080 on the sound server.

Avoiding Cacophony

If every entry in the access log were to trigger a sound, a web page with 20 images, which the browser retrieves in short succession, would trigger an annoying cluster of superimposed noises. For this reason, boom-sender filters the access log output and only transmits to the sound server in case of index pages, high-res images, and discussion forum activity.

The *file2sound()* function defined in line 59 expects the file path requested by the browser (for example, */index.html*) and returns the name of the sound file to play.

To allow this to happen, it makes a few assumptions – for example, that a

WANT TO KNOW WHAT'S UP NEXT?

**SUBSCRIBE TO LINUX
MAGAZINE PREVIEW,
OUR FREE MONTHLY
EMAIL NEWSLETTER!**

path that ends with a / should output an *index.html* file – that you might need to modify when installing.

Installation

The *boom-sender* script is installed on the hosted machine; the Perl modules required for this are available from CPAN. *AccessLog::Parser* has dependencies for *Getopt::Helpful*, *Date::Piece*, *File::Fu*, and *Class::Accessor::Classy*.

If your provider refuses to install these, you can add a module directory in the user-writable area on the hosted machine and add the directive

```
use lib "/home/name/perl-modules";
```

to the Perl script to point it to the new location.

Alternatively, you could set up your own Perl installation in the user-writable area of the hosted machine.

Also, you could consider the PAR toolkit, which allows you to pack module archives and even executables without installation worries in a similar style to Java JAR files.

To reflect your local setup, you will need to modify the URL sound file mappings set up by the *file2sound()* function in *boom-sender*.

Make sure the sound files you reference are available on the sound server. On the sound server, the sound files are

INFO

- [1] USA newsletter (in German): <http://usarundbrief.com>

- [2] Clark, Jim. *Netscape Time: The Making of the Billion-Dollar Startup That Took on Microsoft*. St. Martin's Griffin, 2000.

- [3] Cozens, Simon. *Advanced Perl Programming*, 2nd edition. O'Reilly, 2005.

- [4] Listings for this article: http://www.linux-magazine.com/resources/article_code

installed in *\$SOUND_DIR*. The */usr/share/sounds/purple* directory has a useful selection of short sounds.

In this directory the Pidgin IM client (formerly known as Gaim) stores the sound data that the program outputs when buddies log on or off or that it uses to notify the user of incoming or outgoing instant messages.

After starting the sound server *boom-receiver* and letting it run in the foreground, performing a short test with the *telnet* client in another terminal, and setting up the SSH tunnel referred to earlier, you can launch your logfile monitoring system script on the hosted machine, sit back, and enjoy the concert.

Extensions

In addition to the requested URL paths, the sound server could also play sounds whenever a request fails. The web server stores the return code for each request in *access.log*, and the log parser provides access to it with the *\$fields{code}* hash entry.

To make sure you get the system administrator's attention, you might like to use flatulent noises or explosions for this. ■

Listing 2: boom-sender

```
01 #!/usr/local/bin/perl -w
02 use strict;
03 use POE;
04 use POE::Wheel::FollowTail;
05 use POE::Component::Client::TCP;
06 use ApacheLog::Parser
07     qw(parse_line_to_hash);
08 use Log::Log4perl qw(:easy);
09 Log::Log4perl->easy_init($DEBUG);
10
11 POE::Component::Client::TCP->new(
12     Alias => 'boom',
13     RemoteAddress => 'localhost',
14     RemotePort => 8080,
15     ServerInput => sub {
16         DEBUG "Server says: $_[ARGO]";
17     },
18     InlineStates => {
19         send => sub {
20             DEBUG "Sending $_[ARGO] to server";
21             $_[HEAP]->{server}->put($_[ARGO]);
22         },
23     },
24     ConnectError => sub {
25         LOGDIE "Cannot connect to server";
26     }
27 );
28
29 POE::Session->create(
30     inline_states => {
31         _start => sub {
32             $_[HEAP]->{tail} =
33                 POE::Wheel::FollowTail->new(
34                     Filename =>
35                         "/var/log/apache2/access.log",
36                     InputEvent => "got_log_line",
37                     ResetEvent => "got_log_rollover",
38                 );
39         },
40         got_log_line => sub {
41             my %fields =
42                 parse_line_to_hash $_[ARGO];
43             my $file = $fields{ file };
44             if(my $sound = file2sound($file)) {
45                 POE::Kernel->post("boom", "send",
46                     $sound);
47             }
48         },
49         got_log_rollover => sub {
50             DEBUG "Log rolled over.";
51         },
52     }
53 );
54
55 POE::Kernel->run();
56 exit;
57
58 #####
59 sub file2sound {
60     #####
61     $_ = $_[0];
62
63     DEBUG "Got $_";
64
65     s#/#/#/index.html#;
66
67     m#/index.html# and
68         return "article-page.wav";
69
70     m#/posting.php# and
71         return "forum-post.wav";
72
73     m#/viewforum.php# and
74         return "forum-page.wav";
75
76     m#/images/.*/html# and
77         return "image.wav";
78
79     return "";
80 }
```