Sebastian Duda, Fotolia

Un script Perl que implementa una Internet musical

LOGS MUSICALES

En lugar de monitorizar las peticiones que llegan a su página Web en el archivo de log, un servidor de sonido las hace audibles y permite escuchar la melodía de los usuarios al navegar por la página.

POR MICHAEL SCHILLI

Cada vez que subo una nueva versión de la newsletter de nuestra especie de blog [1], envío un anuncio por email o actualizo la fuente RSS, suelo verificar el log de accesos del servidor Web para vigilar a los primeros visitantes ávidos de la última información o de las imágenes en alta resolución.

Por supuesto, descifrar las entradas del log del servidor haciendo scroll es bastante tedioso. Sería mucho mejor poder monitorizar las peticiones en segundo plano y comenzar a trabajar en otra cosa mientras tanto. Una manera de hacerlo sería transformando los hits de la Web en sonido. Hace mucho tiempo, leí en *Netscape Time*, de Jim Clark, que los primeros Netscapers solían sacar los hits a través del altavoz del PC cuando lanzaban una nueva versión [2]. Una descarga de navegador Netscape para Windows croaba como una rana, el sonido de un cristal roto correspondía a los Macs y las descargas para Unix se anunciaban con el disparo de un cañón. Esto significaba que los pioneros de Internet podían compartir el sonido del éxito en sus cubículos tras el largo proceso de programación que precedía al lanzamiento.

Implementar algo como esto en Perl es bastante sencillo. En mi caso, sin embargo, las cosas no son tan simples debido a que el servidor Web está en alguna parte de una granja de servidores del proveedor. A pesar de que el host permite acceso shell basado en *ssh*, no puede transmitir sonido.

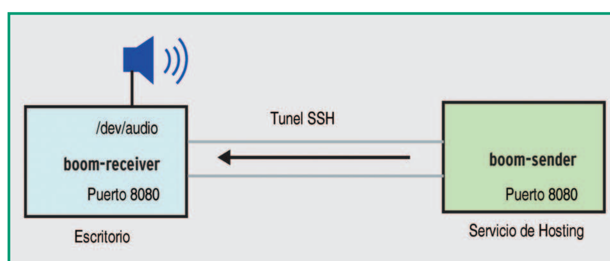

Figura 1: El script que monitoriza el archivo de log de accesos en el servidor Web alojado en el proveedor usa un túnel SSH para comunicarse con un generador de sonidos en la máquina de casa.

Túnel en el Cortafuegos

El script *boom-sender* en el servidor compartido del proveedor monitoriza el archivo *access.log* del servidor Web y envía mensajes a URLs específicas a través de un túnel hasta el script servidor de sonido *boom-receiver* que se ejecuta en mi PC de casa.

La Figura 1 muestra la configuración: el servidor de sonido

es un script implementado por el módulo *POE* de CPAN que escucha los comandos de sonido en el puerto 8080 de la máquina local. Otro script POE se ejecuta en el lado proveedor, reaccionando a los cambios en el log de acceso y enviando mensajes a casa como cliente TCP. Debido a que el equipo que tengo en casa se ubica tras un cortafuegos, el analizador de logs de boom-sender no puede comunicarse directamente. En su lugar, una configuración de túnel mediante el comando

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

en el PC de mi casa, conecta a las dos partes de la comunicación. El script de log en el servidor de host sólo tiene que enviar sus mensajes al puerto local 8080, y al instante se transfieren a través del túnel del puerto 8080 del PC de casa como si el cortafuegos nunca hubiese existido.

Sonido Bajo Demanda

La maquina de sonidos local recibe nombres de archivos de sonido de esta manera y procede a reproducirlos en Linux con la herramienta *play*, un tesoro oculto en el paquete Sox.

Por defecto, el programa Play se incluye con Ubuntu y puede reproducir tanto archivos WAV como MP3, suponiendo que configuremos Ubuntu para ello.

La Figura 2 muestra la interacción de un cliente de prueba con el servidor de sonido en ejecución. El comando *telnet* se inicia para conectarse al puerto 8080 de localhost y recibe una bienvenida del servidor y una lista de los archivos de sonido de que dispone. Cuando el cliente envía el nombre de uno de ellos, el servidor lo reproduce. Por motivos de seguridad, sólo se permiten nombres de archivo, en lugar de rutas.

La ubicación por defecto en la que boom-receiver busca estos archivos de sonido es el directorio actual (.), especificado como *$SOUND_DIR* en la línea 9 del Listado 1.

Debido a que tenemos toda una plétora de componentes del servidor y cliente que tienen que combinarse de manera creativa, POE es una buena elección de tecnología de servidor y cliente. La página Web *poe. perl. org* y el capítulo sobre POE en *Advanced*

Listado 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: [".join(", ",
   @SOUND_FILES) . "]" );
20
21     $_[HEAP]{client}->put
   ("Listo.");
22   },
23
24   ClientInput => sub {
25     my $client_input =
   $_[ARG0];
26
27     if( $client_input !~
   /^[\w.-]+$/ ) {
28         $_[HEAP]{client}->put
   ("Entrada Ilegal.");
29         return;
30     }
31
32     if( $client_input eq "q" )
   {
33         POE::Kernel->yield
   ("shutdown");
34         return;
35     }
36
37     my $msg =
   sound_play($_[HEAP],
   basename($client_input));
38
39     $_[HEAP]{client}->put(
   $msg );
40   },
41
42   InlineStates => {
43     sound_ended => sub {
44       my ($heap, $wid) =
   @_[HEAP, ARG0];
45       DEBUG "Deleting wheel
   $wid";
46       delete
   $heap->{players}->{$wid};
47     },
48   },
49 );
50
51 POE::Kernel->run();
52 exit;
53
54 ###############################
55 sub sound_play {
56 ###############################
57   my($heap, $file) = @_;
58
59   if(! -f "$SOUND_DIR/$file")
   {
60     return "$file doesn't
   exist";
61   }
62
63   POE::Kernel->sig(CHLD =>
   "reaped");
64
65   my $wheel =
66     POE::Wheel::Run->new(
67       Program      =>
   "/usr/bin/play",
68       ProgramArgs =>
   ["$SOUND_DIR/$file"],
69       StderrEvent => 'ignore',
70       CloseEvent   =>
   'sound_ended',
71   );
72
73   DEBUG "Creating wheel ",
   $wheel->ID;
74   $heap->{players}->{
   $wheel->ID } = $wheel;
75
76   return "Played $file";
77 }
```

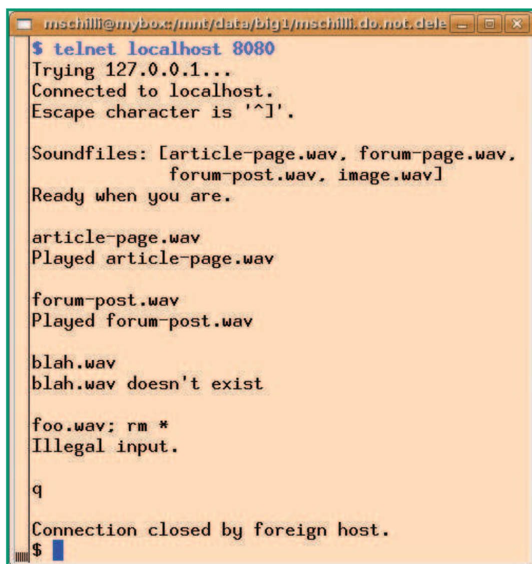

Figura 2: El cliente de prueba, telnet, conecta con el servidor en recepción, que reproduce un archivo de sonido bajo demanda.

Perl Programming [3] ofrecen una útil introducción a POE, que requiere un modelo de programación no tradicional, basada en eventos, al que lleva un tiempo acostumbrarse. El servidor del Listado 1 define los callbacks de los estados *ClientConnected* (el cliente ha abierto una conexión), *ClientInput* (el cliente ha enviado una línea de texto) y *sound_ended*, el estado que controla la limpieza (descrita más abajo) tras reproducir un sonido.

El servidor de sonido administra múltiples conexiones de cliente casi simultáneamente. La lógica del componente POE cuida de los detalles de implementación a bajo nivel y asegura un control de flujo suave de las peticiones y errores tras el escenario. Al igual que cualquier otro script POE, el código del programa define primero el comportamiento de cualquier posible evento y hace entonces la llamada *POE::Kernel->run()* para iniciar el kernel de POE. El kernel se ejecuta hasta que termine el programa, ocurra un error fatal o hasta que el usuario finalice el script.

No se Entretenga

La función *sound_play()* de la línea 55 del Listado 1 reproduce un archivo de sonido que se le pasa por el nombre. Crea *POE::Wheel*, una wheel (rueda) dentada en los trabajos del sistema POE que permite al kernel POE comunicarse con el mundo exterior.

Para permitir al sistema procesar múltiples tareas casi simultáneamente, el código Perl en el POE sólo debería ejecutarse mientras avance a toda potencia. Cualquier interacción con los archivos, sockets u otros procesos obviamente causará retrasos, ya que el acceso a disco o a la red es mucho más lento que procesar las instrucciones en la CPU o acceder a la RAM, y sería extremadamente ineficiente dejar que la CPU quede ociosa mientras espera a que se completen esas tareas. En lugar de esto, se administran en las "ruedas", que las ejecutan en intervalos una a una y reportan los resultados de manera asíncrona de vuelta al kernel POE.

Aplicar el comando *play* para iniciar un nuevo proceso Unix, pasarle un pequeño archivo de sonido y esperar a que se reproduzca puede llevar más de un segundo. Si el script se ha bloqueado esta vez, retrasará la respuesta del cliente y no estará disponible para nuevas peticiones.

En su lugar, se lanza una wheel con la tarea del proceso, la función callback regresa inmediatamente y el kernel POE vuelve a asumir el control, dejando todo lo demás que se ejecute en segundo plano.

La wheel (*POE::Wheel::Run*) aguarda como parámetros un programa externo para iniciar, sus argumentos y un callback *StderrEvent*, desencadenado si el proceso escribe algo en su canal STDERR. Por supuesto, esto no es relevante para el programa Play, que normalmente no genera mensajes de error y simplemente finaliza tras reproducir un archivo de sonido. Boom-receiver simplemente define un estado no existente para este evento, que POE ignora más tarde.

Cuando la wheel advierte que el proceso *play* ha terminado, dispara *CloseEvent* en la línea 70, asignada a una subrutina en la línea 43. Entonces elimina del sistema la referencia pendiente a la wheel, que desencadena que el recolector de basuras del kernel POE limpie los restos.

Idealmente, la wheel lanzaría un proceso como *xmms*, que se ejecutaría permanentemente, y entonces le pasaría de manera ocasional los archivos de sonido. Sin embargo, el componente POE para esto en CPAN es muy antiguo y no compila con las versiones actuales de XMMS. ¡Una pena!

Ojos Vigilantes

Hay que reconocer que la implementación mostrada aquí desperdicia recursos en la máquina local, pero de hecho puede convertir peticiones casi paralelas en sonido. Para conseguir que suceda, el script mantiene una referencia al objeto wheel que genera el sonido, ya que POE limpia el objeto inmediatamente si nadie se ocupa de él. La tarea de la wheel para reproducir el sonido no termina con *sound_play()*, ya que el kernel POE la procesa trozo a trozo tras la finalización de la función. Para evitar un deceso fuera de tiempo, evitando al mismo tiempo mantener las wheels más tiempo del necesario, la línea 74 guarda una referencia al objeto wheel en el heap de la sesión POE con los *players* clave y las ID's de la wheel.

Debido a que la wheel define un *CloseEvent* con un callback *sound_ended*, POE llama a la función definida en la línea 43 cuando finaliza el proceso de sonido. A cambio, la función borra la referencia a la wheel para dejar a POE que los mate.

Otro problema es que *POE::Wheel::Run* no limpia automáticamente los procesos hijo finalizados, sino que los deja deambular como zombies en el sistema Unix. Por tanto, la línea 63 define un manejador *SIGCHLD* que indica al proceso padre que ejecute un *wait()* para el proceso hijo finalizado y evite que se convierta en un zombie.

Tan pronto como se conecta un cliente al puerto 8080 del componente de servidor *POE::Component::Server::TCP*, su máquina de estados cambia el estado a *ClientConnected*. En el callback, *$_[HEAP]{client}* contiene un objeto cliente cuyo método *put()* es usado

por el servidor para enviar mensajes al cliente. El servidor usa *ClientConnected* para informar al cliente que se conecta acerca de los archivos de sonido disponibles antes de anunciar el mensaje *Ready when you are*.

Cada vez que el cliente envía una línea al servidor, el servidor salta a la subrutina mapeada en el estado *ClientInput*. El mensaje recibido está disponible en *$_[ARG0]*, uno de los campos de argumento de array *@_* típicos de POE.

Para evitar que el cliente ataque el servidor con comandos de shell maliciosos, en lugar de enviar un archivo de sonido como se esperaba, la línea 25 verifica el nombre de archivo para ver si contiene alguna otra cosa aparte de los caracteres normales, en cuyo caso genera inmediatamente un mensaje de error y rechaza la petición.

El cliente envía el carácter *q* para indicar que quiere finalizar la sesión. El servidor conmuta entonces al estado *shutdown*, finalizando la conexión del cliente actual pero dejando al servidor en ejecución. Si el cliente realmente envía el nombre de un archivo de sonido existente, la función *sound_play* reproduce el archivo y devuelve una cadena de estado, que el servidor envía a través de *put()* al cliente para confirmar el éxito en la ejecución.

El Final del Túnel

Al otro lado del túnel, el script POE (*boom-sender*) del Listado 2 monitoriza el archivo de log de accesos al servidor Web. Se ejecuta en la máquina de la empresa de hosting y usa los componentes del cliente TCP del marco de trabajo POE para mantenerse en contacto con el servidor.

Entre otras cosas, el componente *Client::TCP* de POE define los eventos *ServerInput* y *ConnectError*. El script salta a los callbacks de estos eventos si el servidor envía texto de vuelta o falla la conexión.

Boom-sender usa *InlineStates* para definir el estado *send*, que usa *put()* para enviar un mensaje al servidor al que se lo ha pasado.

Gracias a la wheel *FollowTail* del toolbox POE, la sesión del archivo de log de monitorización definida en la línea 29 detecta cuándo el servidor Web añade una línea al archivo de log definido en la línea 34. De nuevo, es importante tener una referencia a la wheel para evitar que POE lo limpie después de que el callback *_start* finalice.

La referencia se guarda en un heap de sesión POE bajo la clave *tail* mientras que esté activa la sesión, esto es, hasta que finalice boom-sender.

Listado 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($DEBU
   G);
10
11 POE::Component::Client::TCP->
   new(
12   Alias         => ‘boom’,
13   RemoteAddress =>
   ‘localhost’,
14   RemotePort    => 8080,
15   ServerInput   => sub {
16       DEBUG “Server says:
   $_[ARG0]”;
17   },
18   InlineStates => {
19     send => sub {
20       DEBUG “Enviando
   [$_[ARG0]] a servidor”;
21       $_[HEAP]->{server}->put
   ($_[ARG0]);
22     },
23   },
24   ConnectError => sub {
25       LOGDIE “Imposible
   conctar a servidor”;
26   }
27 );
28
29 POE::Session->create(
30   inline_states => {
31     _start => sub {
32       $_[HEAP]->{tail} =
33 POE::Wheel::FollowTail->new
34          (Filename =>
“/var/log/apache2/access.log”,
35           InputEvent =>
   “got_log_line”,
36           ResetEvent =>
   “got_log_rollover”,
37       );
38     },
39     got_log_line => sub {
40       my %fields =
   parse_line_to_hash $_[ARG0];
41       my $file = $fields{ file
   };
42       if(my $sound =
   file2sound($file)) {
43         POE::Kernel->post
   (“boom”, “send”, $sound);
44
45       }
46     },
47     got_log_rollover => sub {
48       DEBUG “Log rolled
   over.”;
49     },
50   }
51 );
52
53 POE::Kernel->run();
54 exit;
55
56 ##############################
57 sub file2sound {
58 ##############################
59     $_ = $_[0];
60
61     DEBUG “Got $_”;
62
63     s#/$#/index.html#;
64
65     m#/index.html$# and
66         return
   “article-page.wav”;
67
68     m#/posting.php# and
69         return
   “forum-post.wav”;
70
71     m#/viewforum.php# and
72         return
   “forum-page.wav”;
73
74     m#/images/.*html# and
75         return “image.wav”;
76
77     return “”;
78 }
```

Los sistemas de producción tenderán a rotar sus archivos de log diariamente. *FollowTail* está preparado para esto y salta al callback *got_log_rollover* mapeado a *ResetEvent* en este caso. Todo lo que hace es escribir un mensaje de depuración para que el usuario conozca qué está pasando. Cada vez que la wheel encuentra una nueva línea añadida en el log, cambia el estado a *got_log_line* y ejecuta el callback coincidente. Usa el módulo *ApacheLog::Parser* de CPAN para analizar las nuevas líneas, que tiene el siguiente formato:

```
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"
```

La función *parse_line_to_hash()* exportada por este módulo devuelve un hash que contiene el archivo requerido por la petición http bajo la clave *file*.

En la línea 12, el componente del cliente TCP define un alias (*boom*) para su sesión. La wheel *FollowTail*, en ejecución en otra sesión definida en la línea 29, puede usar las siguientes líneas para indicarle al servidor TCP qué archivo de sonido necesita reproducir:

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

Debido a que aquí se están comunicando dos sesiones diferentes, no se puede usar *yield()* para enviar el evento. En su lugar, debe utilizarse *post()* con el alias de la sesión a recibir. A continuación, el kernel POE envía al nombre del archivo WAV al callback *send* de la sesión *boom* como el argumento *ARG0*. El callback usa entonces *put()* para enviar el nombre al cliente TCP de la línea 21, que a cambio lo pasa al servidor de sonido, no de manera directa, sino por el puerto 8080 de la máquina local, y de esta manera a través del túnel hasta el puerto 8080 del servidor de sonido.

Evitar Cacofonías

Si cada entrada del log de accesos desencadenase un sonido, una página Web con 20 imágenes, que el navegador recupera en una corta sucesión, generaría una molesta aglomeración de ruidos superpuestos. Por esta razón, boom-sender filtra la salida del log de accesos y sólo transmite al servidor de sonidos en el caso de páginas de inicio, imágenes en alta resolución y actividad en los foros de discusión.

La función *file2sound()* definida en la línea 57 aguarda la ruta al archivo requerido por el navegador (por ejemplo, */index.html*) y devuelve el nombre del archivo de sonido a reproducir.

Para permitir que ocurra esto, hace algunas suposiciones, por ejemplo que la ruta que acaba con / debería generar un archivo *index.html*, el cual puede que tengamos que modificar en la instalación.

Instalación

El script *boom-sender* se instala en la máquina en hosting. Los módulos que se necesitan para esto están disponibles desde CPAN. *AccessLog::Parser* tiene dependencias para *Getopt::Helpful*, *Date::Piece*, *File::Fu* y *Class::Accessor::Classy*.

Si su proveedor rechaza instalarlos, podemos añadir un directorio de módulos en el área de escritura para el usuario en la máquina en hosting y añadir la directiva

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

al script Perl para apuntarlo a la nueva ubicación.

De forma alternativa, podemos consultar nuestra propia instalación Perl en el área de escritura para el usuario de la máquina en Hosting.

De igual manera, podremos considerar el toolkit PAR, que nos permite empaquetar archivos de módulo e incluso ejecutables sin tener que preocuparnos de la instalación, de una forma similar a los archivos JAR de Java.

Para reflejar nuestra configuración local, necesitamos modificar la configuración de los mapeos a las URL de los archivos de sonido que se han configurado por la función *file2sound()* en boom-sender.

Debemos asegurarnos de que los archivos de sonido que referenciamos están disponibles en el servidor de sonido. Dentro del servidor de sonido, los archivos de sonido se instalan en *$SOUND_DIR*. El directorio */usr/share/sounds/purple* tiene una útil sección de sonidos breves.

En este directorio, el cliente de mensajería instantánea Pidgim (anteriormente conocido como Gaim) guarda la información de los sonidos que genera el programa cuando nuestros contactos entran o salen, o que usa para notificar al usuario mensajes entrantes o salientes.

Tras iniciar el servidor de sonido boom-receiver y dejarlo en ejecución en primer plano, ejecutar una pequeña prueba con el cliente *telnet* en otro terminal y configurar el túnel SSH referido al primero, ya podemos iniciar nuestro script de monitorización del archivo del log en la máquina en el proveedor, sentarnos y disfrutar del concierto.

Mejoras

Además de las rutas URL requeridas, el servidor de sonido podría reproducir también sonidos cada vez que falla una petición. El servidor de Web guarda el código devuelto para cada petición en *access.log*, y el analizador de logs proporciona acceso a éste con la entrada de hash *$fields{code}*. ■

RECURSOS

[1] USA newsletter (en Alemán): *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", 2ª edición. O'Reilly. 2005.

[4] Listados de este artículo: *http://www.linux-magazine.es/Magazine/Downloads/51*