Ruteador de Llamadas en PHP (4 Horas)

Moisés Silva <moy@ivsol.net>

Tipo de Propuesta: Taller 4 horas

Track: Desarrollo de Software / Aplicaciones

Resumen:

Asterisk ha comenzado a jugar un papel muy importante en las comunicaciones globales; mas que un PBX, es un servidor de comunicaciones. Una de las facilidades de las que provee Asterisk que lo hace tan flexible es AGI (Asterisk Gateway Interface) y AMI (Asterisk Manager Interface). AGI permite que, desde cualquier lenguaje de programación que pueda escribir al STDOUT y leer del STDIN, puedas tomar decisiones de ruteo y ejecución de aplicaciones sobre las llamadas. AMI por su parte te da un excelente control sobre todo el sistema; permitiendonos conectarnos por medio de un socket TCP desde PHP (o cualquier otro lenguaje con soporte para sockets) y recibir notificaciones sobre eventos que ocurren en Asterisk (Nuevas llamadas, tono de colgado etc.), asi como emitir acciones (cuelga esta llamada, provee tono de marcado etc.).

PHP es un lenguaje orientado a web. Sin embargo en este taller podremos ver que es posible desarrollar aplicaciones de propósito general, como lo es posible con otros lenguajes de scripting.

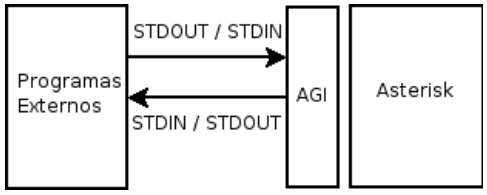
Durante el taller se desarrollará un ruteador de llamadas que permitirá decidir, en base a los patrones de marcado, hacia donde enviar la llamada. Se explicará la forma de comunicación entre Asterisk y los scripts AGI.

Adicionalmente presentaremos un parche conocido como "MAGI", que agrega una aplicación que permite la ejecución de comandos AGI por medio de la AMI, de tal forma que nos llevará al diseño de un daemon ruteador de llamadas orientado a eventos.

1. Asterisk Gateway Interface (AGI)

AGI nació de la necesidad de tener un control mas flexible sobre el ruteo de las llamadas. Cuando deseamos que el ruteo de nuestras llamadas dependa de cosas mas complejas como horario, estatus de mensajería instantánea, registro de otras extensiones etc, la funcionalidad nativa aplicable en extensions.conf simplemente deja de ser suficiente. Asterisk permite que un programa hecho en cualquier lenguaje de programación tome el control de la llamada. Existen aplicaciones que permiten hacerlo como EAGI, FastAGI, DeadAGI, PHP, MAGI. Sin embargo, a excepción de las últimas dos, todas las demás funcionan esencialmente bajo el mismo esquema. A continuación se describe el funcionamiento de las aplicaciones antes mencionadas:

AGI: Aplicación que crea un nuevo proceso cada vez que se le manda llamar. Asterisk ejecuta un fork() conectandose al proceso que corre nuestro programa mediante el STDIN y STDOUT. Cualquier cosa escrita desde nuestro programa al STDOUT será leida por el STDIN de Asterisk y será tratado como un comando AGI, la respuesta será escrita por Asterisk al STDOUT y pued ser leída desde nuestro programa del STDIN.



EAGI: Exactamente el mismo funcionamiento que presenta AGI, con la excepción de un descriptor de archivo número 3 (STDIN 0, STDOUT 1, STDERR 2) que es utilizado para audio fuera de banda (out of band).

DeadAGI: Funciona igual que AGI, solo que es utilizado para la ejecución de scripts cuando el canal ya se encuentra colgado (en la extensión h).

FastAGI: Igual comportamiento que las anteriores, con la excepción de que la comunicación no se lleva a cabo haciendo un fork y redireccionando el STDIN y STDOUT. Con FastAGI es posible reducir la carga en el servidor delegandole la ejecución de los scripts y conexiones a bases de datos a otra computadora. FastAgi abre un socket de comunicación por donde fluiran los comandos AGI y sus respuestas.

PHP: Esta aplicación debe ser utilizada unicamente si se cuenta con la extensión asociada de PHP. Es decir, un módulo de php que permite ejecutar funciones de Asterisk nativamente desde PHP.

MAGI (Manager AGI): Esta aplicación solo se encuentra disponible en caso de que Asterisk se encuentre propiamente parchado. No es una aplicación común. Permite la ejecución de comándos AGI desde el socket de la interfaz del Manager.

A continuación veremos como hacer un "hello world" con AGI. Debido a que la comunicación entre Asterisk y el script ocurrira usando STDIN y STDOUT, es altamente recomendable que se deshabilite la salida de errores en php.ini y activar el log de errores en un archivo. Cualquier salida no controlada provocará que Asterisk intente interpretar esa salida como comando y nos puede ocasionar conflictos dificiles de encontrar. En php.ini ponemos:

```
display_errors=Off
display_startup_errors=Off
log_errors=On
error_log=/path/to/log
```

Ahora todo lo que necesitamos es abrir el archivo extensions.conf y crear un nuevo contexto como el siguiente:

```
[hello-world]
exten => _X.,1,Answer()
exten => _X.,2,AGI(hello-world.php)
exten => _X.,3,Hangup()
```

Ahora creamos el archivo /var/lib/asterisk/agi-bin/hello-world.php y ponemos lo siguiente:

```
#!/usr/bin/php
<?php
do
{
   $read_string = fread(STDIN, 1024);
} while ( FALSE === strpos($read_string, 'agi_accountcode:') );
print "EXEC Playback hello-world";
?>
```

Finalmente abrimos una consola de Asterisk y marcamos del algún teléfono que tengamos configurado con el contexto creado. Debemos ver algo como esto:

-- Executing Answer("SIP/33-9c1c", "") in new stack
-- Executing Playback("SIP/33-9c1c", "hello-world") in new stack
-- Playing 'hello-world' (language 'en')
-- Executing Hangup("SIP/33-9c1c", "") in new stack

Ahora expliquemos que ha sucedido. Asterisk recibe la petición de llamada y el número marcado. Ejecuta la aplicación Answer() (la ejecutamos directamente desde extensions.conf), posteriormente ejecuta la aplicación AGI() quien lanza un nuevo proceso totalmente independiente y conectado a Asterisk solo por STDIN, STDOUT y STDERR. A continuación Asterisk ejecutará como comando AGI cualquier salida al STDOUT desde nuestro script, y nos enviara la respuesta a las ejecuciones a través del STDIN del script. Analizemos ahora el script. El while hace una tarea muy simple, lee todo del STDIN hasta encontrarse con que la lectura devuelve una cadena de texto que contiene unicamente “agi_accountcode”. Una vez hecho esto, hay una linea única de código que manda al STDOUT (print, echo escriben al buffer de salida) una cadena que dice "EXEC Playback hello-world.gsm". Asi que Asterisk lee esta salida y ejecuta el comando EXEC con los argumentos Playback y hello-world.gsm. El comando EXEC de AGI permite la ejecución de aplicaciones de Asterisk. Finalmente Asterisk termina el script y cuelga la llamada.

Finalmente solo queda la pregunta, por que fué necesario el while() ?? Bueno, para que un programa pueda tomar decisiones de ruteo hace falta cierta información que Asterisk provee al iniciar el script AGI. Esta información es enviada siempre por Asterisk en la forma "parametro: valor\n" terminando la lista de parámetros por un "\n". En este primer script AGI hemos ignorado completamente esos datos ya que no eran necesarios para nuestro ejemplo.

Cualquier problema con AGI (como por ejemplo que el script se queda trabado) puede ser debuggeado con el comando de Asterisk "agi debug", lo que habilita que Asterisk reporte las lecturas y escrituras desde y hacia nuestro script. El debuggeo se deshabilita usando "agi no debug".

Ejercicio: Modificar el script anterior para que lea y guarde en una variable, de forma organizada. Las variables enviadas inicialmente por Asterisk.

Ahora, imaginemos que tenemos que crear un script de ruteo que decida, en base al número marcado, si comunicará al usuario con una extensión IAX, SIP o con la PSTN. La primer pregunta es como distinguir entre estas 3 operaciones. La respuesta se encuentra en los patrones de marcado o marcaciones. Una marcación es toda la información que un usuario puede proveer desde su teléfono, por ello es la marcación la que decide principalmente que debe hacerse. Asi que debemos definir 3 diferentes marcaciones. Una para extensiones SIP, otra para IAX y finalmente otra para la PSTN. Las marcaciones serán:

3X (IAX2)
2X (SIP)
XXXXXXXX (Zap PSTN)

Sin embargo, antes de continuar, queda como tarea elaborar una clase o libreria de funciones para manejar la comunicación con Asterisk de forma simple. El resultado final debe ser algo como el archivo AgiConnector.php adjunto en este tutorial. Las reglas a seguir son simples:

- Al iniciar debemos, obligatoriamente, leer todas las variables que nos envia Asterisk.
- Los comandos se ejecutan escribiendo al STDOUT, terminando el comando con un salto de línea.
- Para ejecutar una aplicación podemos usar el comando EXEC <Nombre Aplicacion> <Argumentos Opcionales>
- Al terminar de ejecutar cualquier comando, es recomendable leer la respuesta de Asterisk. Para esto leemos del STDIN justo después de escribir nuestro comando al STDOUT. Asterisk responde en la forma:

<response code> <result=number> <(optional data)>

Para mas información sobre comandos y posibles referencias de documentación, revisar apendice de este documento.

Una vez finalizada la clase o librerias que nos ayudarán a las tareas comunes de comunicación con Asterisk, procedemos a crear el script de ruteo para los patrones previamente definidos. Veamos cuales son los pasos para lograr nuestro objetivo:

1. Leer el número marcado.
2. Identificar si la llamada corresponde a SIP, IAX o ZAP
3. Marcar la extensión deseada.

Como resolvemos el primer problema? como obtenemos el número marcado por el usuario desde AGI?. Una forma sería leer la variable ${EXTEN} de Asterisk. Sin embargo, Asterisk nos provee de este número al iniciar el script AGI. En nuestra clase AmiConnector hemos puesto todas esas variables enviadas al inicio por Asterisk en un ArrayObject (un objeto wrapper de un array). De tal forma que unicamente necesitamos el siguiente código para darnos cuenta que el paso uno, es pan comido:

```
#!/usr/bin/php
<?php
require_once 'AgiConnector.php';
$connector = new AgiConnector();
$connector->Write('Starting Router');
$connector->Write('you have dialed: ' . $connector->GetAgiVariable('extension'));
```

```
$connector->Playback('hello-world');
$connector->SayDigits($connector->GetAgiVariable('extension'));
?>
```

El segundo paso es también muy simple, solo debemos identificar si la llamada corresponde a IAX, SIP o Zap. Un Simple switch basta. Asi que nos adelantaremos al tercer paso que involucra marcar la extensión deseada. Veamos el script final:

```
#!/usr/bin/php
<?php
require_once 'AgiConnector.php';
$connector = new AgiConnector();
$connector->Write('Starting Router');
$connector->Write('you have dialed: ' . $connector->GetAgiVariable('extension'));
$dialed_number = $connector->GetAgiVariable('extension');
if ( FALSE !== ereg('^3[0-9]$', $dialed_number) )
{
   $channel = 'IAX2';
}
elseif ( FALSE !== ereg('^2[0-9]$', $dialed_number) )
{
   $channel = 'SIP';
}
elseif ( FALSE !== ereg('^[0-9]{8}$') )
{
   $channel = 'Zap/1';
}
$connector->Dial($channel . "/" . $dialed_number);
?>
```

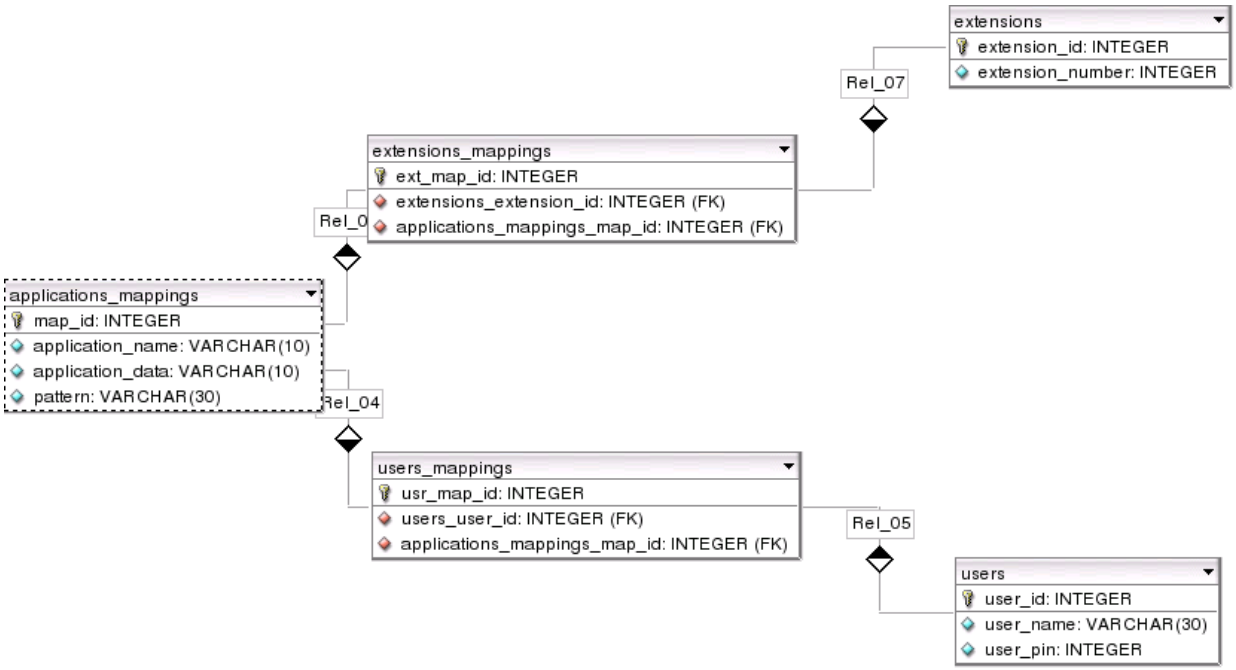
Este script es muy ilustrativo, sin embargo, carece de varias cosas esenciales como:

1. Autenticación del usuario. No siempre cualquier persona o extensión tiene acceso a marcar todos los patrones. Sería útil tener una tabla en una base de datos con patrones, y otra tabla con extensiones. Finalmente unir ambas tablas mediante una tabla que indique que patrones tiene derecho a marcar cierta extensión.

2. Una identificación del patrón mas robusta. No podemos simplemente agregar mas elseifs para cada patrón que se puede necesitar. Mas aún, los patrones deberían ser configurables desde una interfaz web, asi que es obvio que es necesario guardar los patrones en una base de datos.

3. Aparte de iniciar una llamada, existen otras cosas que puede desear un usuario telefónico, como por ejemplo, acceder a su voicemail. Asi que la aplicación Dial(), tampoco debe ser llamada para todos los casos.

Es tiempo de resolver las 3 necesidades planteadas anteriormente. Para base de datos usaremos sqlite. Esto nos permitira ahorrarnos la instalación y configuración de una base de datos mas robusta y conocida como postgresql o mysql. La estructura de nuestra base de datos será la siguiente.



Notese que DbDesigner (el programa que use para el modelo de la base de datos) no me dejo nombrar las llaves foraneas como yo quería, de modo que el nombre en el modelo difiere un poco del nombre en las sentencias. En sqlite conseguimos lo anterior y algunos registros de prueba con las siguientes sentencias para SQLite:

```
BEGIN TRANSACTION;
```

```
CREATE TABLE applications_mappings (map_id INTEGER PRIMARY KEY, application_name VARCHAR(10), application_data
VARCHAR(10), pattern VARCHAR(30));
INSERT INTO "applications_mappings" VALUES(1, 'dial', 'IAX2', '3X');
INSERT INTO "applications_mappings" VALUES(2, 'dial', 'SIP', '2X');
INSERT INTO "applications_mappings" VALUES(3, 'dial', 'Zap', 'XXXXXXXX');
INSERT INTO "applications_mappings" VALUES(4, 'voicemail', '', '66');
INSERT INTO "applications_mappings" VALUES(5, 'conference', '', '7X');
CREATE TABLE extensions (extension_id INTEGER PRIMARY KEY, extension_number INTEGER);
INSERT INTO "extensions" VALUES(1, 33);
CREATE TABLE extensions_mappings (ext_map_id INTEGER PRIMARY KEY, extension_id INTEGER, map_id INTEGER);
INSERT INTO "extensions_mappings" VALUES(1, 1, 2);
CREATE TABLE users (user_id INTEGER PRIMARY KEY, user_name VARCHAR(30), user_pin INTEGER);
INSERT INTO "users" VALUES(1, 'moises silva', 123456);
CREATE TABLE users_mappings (usr_map_id INTEGER PRIMARY KEY, user_id INTEGER, map_id INTEGER);
INSERT INTO "users_mappings" VALUES(1, 1, 1);
INSERT INTO "users_mappings" VALUES(2, 1, 2);
INSERT INTO "users_mappings" VALUES(3, 1, 4);
INSERT INTO "users_mappings" VALUES(4, 1, 5);
COMMIT;
```

Estas 5 tablas son todo lo que necesitamos para ilustrar la idea. Cual será ahora el flujo de la llamada?

1. La llamada es recibida y se obtiene el número destino (agi_extension) y el callerid (agi_callerid)
2. Se revisa la tabla de extensiones en busca del callerid.
3. Se revisa que la extension en cuestión tenga autorizado el patron de marcado solicitado. Se hace a través del map_id en la tabla extensions_mapping.
4. Si tiene autorización se procede a ejecutar la aplicación relacionada con el patrón. De lo contrario le solicitamos introducir un PIN de usuario.
5. Si el PIN corresponde al de un usuario con autorización, se procede, de lo contrario se le solicita el PIN hasta 3 veces. Excedidos los 3 intentos, colgamos la llamada.

Es importante aclarar que para cuando una llamada inicia en una interfaz Zap, no se puede utilizar agi_extension, debido a que es común que se utlice la extensión 's', lo que obviamente no es un número. Por otro lado, el callerid no es una fuente 100% fiable, pero suficiente para un ambiente controlado como una oficina pequeña.

Para lograr nuestro objetivo necesitamos algunas rutinas básicas de acceso a la base de datos para que nuestro script router no tenga que hacerlo por el mismo. En un archivo adjunto llamado RouteController.php tenemos algunas rutinas necesarias. Sin embargo, ponganse creativos antes de revisarlo. Analicen los pasos y vean si pueden crear las rutinas necesarias. Mostraré el script de ruteo final:

```
require_once 'AgiConnector.php';
```

```
require_once 'RouteHoncho.php';
/* create a new connection with Asterisk */
$connector     = new AgiConnector();
$connector->Write('Starting Router');
$connector->Write('user has dialed: ' . $connector->GetAgiVariable('extension'));


try
{
   /* create a new instance of the expert in routing */
   $route_honcho  = new RouteHoncho();


   /* get the number dialed by the user */
   $dialed_number = $connector->GetAgiVariable('extension');


   /* get whos calling */
   $callerid      = $connector->GetAgiVariable('callerid');


   /* get where wants to call */
   $mapping       = $route_honcho->GetNumberMapping($dialed_number);
   if ( NULL === $mapping )
   {
      $connector->Playback('iss_invalid_pbx_number_en');
      exit;
   }
   /* is the extension authorized to make this call? */
   $try_times = 0;
   $pin       = NULL;
   do
   {
      if ( $route_honcho->CallIsAuthorized($callerid, $mapping['map_id'], $pin) )
      {
         /* this can be much, much better passing control to "Application Drivers"
          * instead of having hard coded Applications
          * */
         switch ( $mapping['name'] )
         {
            case RouteHoncho::APPLICATION_DIAL:
               $connector->Dial($mapping['data'] . '/' . $dialed_number);
               break;


            case RouteHoncho::APPLICATION_VOICEMAIL:
               $connector->EnterVoiceMail($dialed_number);
               break;

            case RouteHoncho::APPLICATION_CONFERENCE:
               $connector->StartConference($dialed_number);
```

```
                break;


            default:
                $connector->Playback('iss_hangup_en');
                break;
        }
        exit;
    }
    /* is not the first time, then play sound to indicate incorrect auth PIN */
    if ( $try_times > 0 )
    {
        $connector->Playback('iss_wrong_auth_en');
    }
    /* retrieve DTMF ddigits from the user asking for PIN */
    $pin = $connector->GetDigitsFromUser('iss_ask_auth_en');
    $connector->Write('user has pressed: ' . $pin);
    $try_times++;
  }
  while ( $try_times <= 3 );
  $connector->Playback('iss_hangup_en');
}
catch ( Exception $error )
{
  /* ooops something has gone wrong, try to play technical failure sound */
  $connector->Write($error->getMessage());
  $connector->Playback('iss_technical_failure_en');
}
?>
```

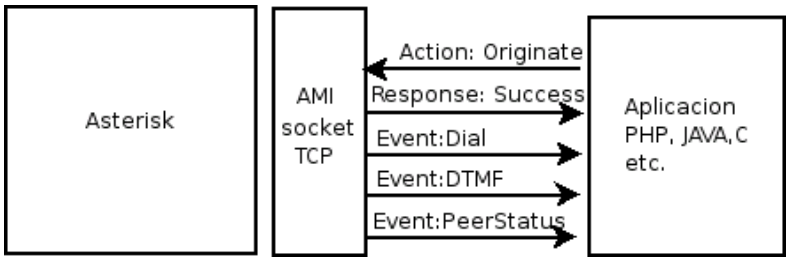
Durante este pequeño how-to para desarrollar un ruteador de llamadas en PHP hemos aprendido la forma de comunicación mas simple entre Asterisk y programas escritos en otros lenguajes. En el siguiente how-to describiré otra pontente interfaz de comunicación que expone Asterisk: AMI (Asterisk Manager Interface). Que permite que nuestros programas se conecten por medio de un socket TCP para controlar Asterisk y recibir eventos sobre la actividad del PBX y asi poder reaccionar ante ellos. AMI es la forma mas practica para conseguir una aplicación "Click To Dial".

2. Asterisk Manager Interface (AMI)

AMI, a diferencia de AGI. Es una interfaz hacia Asterisk visto como un "TODO", es decir, de forma global. AGI se concreta a ejecutar comandos únicamente en un canal específico (el canal que haya ejecutado la aplicación AGI) de forma que si 10 nuevas llamadas necesitan ser atendidas en un segundo, durante ese segundo se crearan 10 procesos nuevos e independientes entre si, para la ejecución del script AGI. AMI por su parte, es una interfaz de acceso directo a Asterisk, que no se limita a ser ejecutada unicamente durante un llamado explicito por parte de Asterisk. AMI esta pendiente (un thread escuchando en un socket TCP en localhost:5038 por default) de cualquier agente externo que desee conectarse (un script de php por ejemplo). Una vez conectado el programa externo, existe un pequeño protocolo para el envio información entre Asterisk y los sistemas conectados al socket.

Que clase de información puede intercambiarse?

Existen 3 tipos de "paquetes" que pueden ser enviados. acciones (Action), respuestas (Response) y eventos (Event). El siguiente gráfico muestra el flujo de información clásico entre una aplicación y Asterisk.



En el gráfico se aprecian los siguientes sucesos.

1. La aplicación envia un paquete tipo "Action" con el nombre "Originate". Esta es una acción que le indica a Asterisk que debe iniciar una llamada. Desde luego la sola linea Action:Originate no es suficiente. Debe indicar hacia donde se originará la llamada y a donde se conectará, así como otros datos miscelaneos. Mas adelante veremos como se envian estos datos extra.

2. Después de ejecutar la acción, Asterisk envia un paquete de tipo "Response" con el resultado "Success". Indicando que la acción se llevo a cabo correctamente.

3. Asterisk envia 3 paquetes de tipo "Event" reportando que que ha iniciado una llamada, se presionó un digito DTMF y que el estatus de un peer ha cambiado. Cada uno de estos paquetes en la realidad trae mas información. En el diagrama solo mostramos la cabecera.

La explicación anterior fué un tanto escueta, asi que veamos algo práctico. Para ello necesitamos tener correctamente configurado Asterisk y un cliente "telnet". Primero Asterisk, abrimos el archivo /etc/asterisk/manager.conf y necesitamos algo como esto:

```
[general]
enabled=yes
port=5038
bindadd=127.0.0.1

[fsl]
secret=junio2006
deny=0.0.0.0/0.0.0.0
permit=127.0.0.1/255.0.0.0
permit=<tu-ip-local>/<tu-mascara-de-red>
read=system,call,log,verbose,command,agent,user
write=system,call,log,verbose,command,agent,user
```

En el contexto general indicamos que deseamos habilitar AMI y la información del socket. Todos los contextos adicionales son para manejar la autenticación. En este caso hemos creado un usuario llamado "fsl" con password "junio2006", a este usuario no se le permiten conexiones desde fuera de la máquina (de cualquier modo en este caso no podría por que unicamente configuramos el socket para localhost) y tiene permisos para "system, call, log, verbose, command, agent, user". Estos permisos indican una clasificación de las acciones que puede ejecutar y los eventos que podrá leer.

Ahora, para conectarnos, sin tener que escribir primero un cliente en PHP, podemos usar telnet. En gentoo ejecutamos el comando "emerge netkit-telnetd" para descargar un cliente (incluye también servidor) de forma rápida. Una vez instalado ejecutamos el siguiente comando:

```
# telnet localhost 5038
```

Lo que debe de mostrarnos lo siguiente:

```
Trying 127.0.0.1...
Connected to localhost
Escape character is '^]'.
Asterisk Call Manager/1.0
```

Si en lugar de ello, obtenemos un mensaje como "connection refused" o simplemente nada, algo hay mal configurado (lease firewall o Asterisk mismo). Para asegurarnos que Asterisk está escuchando correctamente en el puerto 5038 podemos ejecutar el siguiente comando:

 # netstat -lpn | grep -i asterisk

Esto nos mostrará todos los sockets en donde Asterisk se encuentra escuchando. Una salida típica es:

```
tcp        0      0 127.0.0.1:5038          0.0.0.0:*               LISTEN      14469/asterisk
udp        0      0 0.0.0.0:5060            0.0.0.0:*                           14469/asterisk
udp        0      0 0.0.0.0:4569            0.0.0.0:*                           14469/asterisk
unix  2      [ ACC ]     STREAM     LISTENING     21419  14469/asterisk      /var/run/asterisk/asterisk.ctl
```

La primer línea muestra claramente que existe un socket TCP perteneciente a Asterisk que se encuentra escuchando en 127.0.0.1:5038, las siguientes dos conexiones UDP corresponden a chan_sip y chan_iax2 respectivamente, son sockets que esperan paquetes para los protocolos SIP e IAX. El socket de UNIX al final se encarga de manejar la línea de comandos.

Iniciaremos con una acción de prueba llamada "Ping" que debe devolvernos como respuesta "Pong". Escribiré a continuación todo el proceso, repitiendo incluso el paso anterior de conexión inicial. Es recomendable abrir una consola de Asterisk para observar los mensajes de Asterisk al aceptar o rechazar la conexión.

```
# telnet localhost 5038
Trying 127.0.0.1...
Connected to localhost
Escape character is '^]'.
Asterisk Call Manager/1.0
Action: Login
Username: fsl
Secret: junio2006

Response: Success
Message: Authentication accepted

Action: Ping

Response: Pong

Action: Logoff
```

Response: Goodbye
Message: Thanks for all the fish.

Connection closed by foreign host.

Ante esto, Asterisk reporta en la consola lo siguiente:

== Parsing '/etc/asterisk/manager.conf': Found
== Manager 'fsl' logged on from 127.0.0.1
== Manager 'fsl' logged off from 127.0.0.1

En color café se muestra lo que hemos escrito, y en azul lo que Asterisk responde. Iniciamos la conexión mediante el cliente telnet, que abre un socket en localhost al puerto 5038. Asterisk al detectar esta nueva conexión, responde con un mensaje de bienvenida diciendo: Asterisk Callmanager/1.0. Antes de poder ejecutar cualquier acción útil, tenemos que enviar la acción Login para iniciar sesión y poder enviar otro tipo de acciones y recibir eventos (limitados por el archivo manager.conf). Antes de revisar los paquetes que enviamos, veamos cual debe ser la estructura de los paquetes.

<Tipo Paquete>: Nombre de paquete <CRLF>
<Header1>: <valor header 1><CRLF>
<Header2>: <valor header 2><CRLF>
<HeaderN>: <valor header N><CRLF>
<CRLF>

Como vemos, se empieza con el tipo de paquete (Action, Response, Event). Nuestras aplicaciones se limitan a enviar paquetes de tipo Action y recibir paquetes de tipo Response y Event de parte de Asterisk. El espacio entre el el header y su valor es INDISPENSABLE (<Header1>:<espacio necesario AQUI><valor header 1><CRLF>), lo mismo naturalmente para el tipo de paquete. Ante la acción Ping, Asterisk nos responde con Pong. Finalmente enviamos la acción Logoff, y Asterisk nos responde amistosamente para luego cerrar la conexión.

Ahora procederemos a hacer algo útil. Originaremos una llamada hacia nuestra extensión y la conectaremos a un contexto que nos diga "hello world".

Action: Originate
Channel: SIP/32
Context: hello-world
Exten:s

Priority: 1

Response: Success
Message: Originate successfully queued

Esto debe haber originado una llamada hacia el canal SIP/32, que al ser contestado (el usuario levanta la bocina) es conectado al contexto hello-world a la extensión "s" prioridad "1".

Como podemos aplicar el conocimiento adquirido para hacer una aplicación Web "Click To Dial"?

Antes de iniciar con la aplicación click to dial. Haremos un pequeño script para enviar una acción Ping y recibir la respuesta correspondiente. Usaremos fsockopen() para crear el socket nosotros mismos, de forma que no necesitaremos de ejecutar telnet con proc_open() ni nada similar. Por lo tanto, será más eficiente :)

```
<?php
$errno         = -1;
$errstr        = '';
$manager_socket = fsockopen('localhost', 5038, $errno, $errstr);
if ( is_resource($manager_socket) )
{
   $welcome       = fread($manager_socket, 1024);
   print $welcome . "\n";
   fwrite($manager_socket, "Action: Login\r\n");
   fwrite($manager_socket, "Username: fsl\r\n");
   fwrite($manager_socket, "Secret: junio2006\r\n");
   fwrite($manager_socket, "\r\n");
   $response      = fread($manager_socket, 1024);
   print $response . "\n";
   fwrite($manager_socket, "Action: Ping\r\n\r\n");
   $response = fread($manager_socket, 1024);
   print $response;
}
else
{
   print "error: {$errstr}\n";
}
?>
```

Asi tenemos nuestro primer script de comunicación con AMI. La salida de este script debe ser:

Asterisk Call Manager/1.0

Response: Success
Message: Authentication accepted

Response: Pong

Ahora procederemos a terminar nuestro primer objetivo. Lograr una originación de llamada y conectarla con un contexto que le diga "hello world". Para ello modificaremos nuestro script anterior para enviar la acción Originate con sus respectivos campos.

```
<?php
$errno         = -1;
$errstr        = '';
$manager_socket = fsockopen('localhost', 5038, $errno, $errstr);
if ( is_resource($manager_socket) )
{
   $welcome       = fread($manager_socket, 1024);
   print $welcome . "\n";
   fwrite($manager_socket, "Action: Login\r\n");
   fwrite($manager_socket, "Username: fsl\r\n");
   fwrite($manager_socket, "Secret: junio2006\r\n");
   fwrite($manager_socket, "\r\n");
   $response      = fread($manager_socket, 1024);
   print $response . "\n";
   if ( stripos($response, "Success") )
   {
      fwrite($manager_socket, "Action: Originate\r\n");
      fwrite($manager_socket, "Channel: SIP/33\r\n");
      fwrite($manager_socket, "Context: hello-world\r\n");
      fwrite($manager_socket, "Exten: s\r\n");
      fwrite($manager_socket, "Priority: 1\r\n");
      fwrite($manager_socket, "\r\n");
      $response = fread($manager_socket, 1024);
      print $response;
   }
   else
   {
      print "Failed to login into Asterisk Manager\n";
   }
}
```

```
else
{
   print "error: {$errstr}\n";
}
?>
```

Listo, tenemos un script que nos genera llamadas automaticamente. Este sencillo ejemplo debe ser capaz de mostrar todo el potencial existente de AMI. Puedes hacer paginas web para tus clientes en los que incluya un campo tipo "text" y un botón diciendo "Presione Aqui Para Comunicarse Con Nuestro Departamento de Ventas!". De esta forma el usuario de la página web teclearia su número telefónico y Asterisk llamaría al departamento de ventas para comunicarlo directamente con el cliente!. Veamos como logramos esto. La idea es:

1. Se ingresa a la página web donde se muestra un campo de texto para introducir el número a marcar y se presenta un botón para que al hacer click se origine la llamada.
2. Al hacer click recibimos el número via POST o GET
3. Abrimos un socket al manager y le enviamos la acción de originate con el número solicitado.
4. En caso de error mostramos servicio no disponible. de lo contrario mostramos mensaje de éxito.

Expresada en código es algo como esto:

```
<?php
ignore_user_abort();
ob_implicit_flush();
define('SALES_SIP_CHANNEL', 'SIP/33');
define('ZAP_CONTEXT', 'localline');
define('ZAP_PRIORITY', 1);
define('MANAGER_USER', 'fsl');
define('MANAGER_SECRET', 'junio2006');
print <<<TOP
<html>
<head>
<title>Click To Dial!</title>
</head>
<body>
TOP;
if ( isset($_POST['number']) && is_numeric($_POST['number']) )
{
   $number = $_POST['number'];
   $manager_socket = fsockopen('localhost', 5038);
   if ( is_resource($manager_socket) )
   {
      $welcome = fread($manager_socket, 1024);
      fwrite($manager_socket, "Action: Login\r\n");
      fwrite($manager_socket, "Username: " . MANAGER_USER . "\r\n");
```

```
        fwrite($manager_socket, "Secret: " . MANAGER_SECRET . "\r\n");
        fwrite($manager_socket, "\r\n");
        $response       = fread($manager_socket, 1024);
        if ( FALSE !== stripos($response, "Success") )
        {
           print "Originating call...<br />";
           fwrite($manager_socket, "Action: Originate\r\n");
           fwrite($manager_socket, "Channel: " . SALES_SIP_CHANNEL . "\r\n");
           fwrite($manager_socket, "Context: " . ZAP_CONTEXT . "\r\n");
           fwrite($manager_socket, "Exten: " . $number . "\r\n");
           fwrite($manager_socket, "Priority: " . ZAP_PRIORITY . "\r\n");
           fwrite($manager_socket, "\r\n");
           $response = fread($manager_socket, 1024);
           if ( FALSE !== stripos($response, "Success") )
           {
              print 'The call has been made. Please wait a few seconds.';
           }
           else
           {
           else
           {
              print 'Service Currently Unavailable';
           }
        }
        /* Failed Login, print error message */
        else
        {
           print 'Service Currently Unavailable';
        }
     }
     /* unable to open socket, print error string */
     else
     {
        print 'Service Currently Unavailable';
     }
}
else
{
     print <<<FORM
     <form name="clicktodial" method="post" action="click_to_dial.php">
        <input type="text" name="number" /><br />
        <input type="submit" value="Click To Dial">
     </form>
     </body>
     </html>
FORM;
}
?>
```

Con esto finalizamos nuestra aplicación click to dial. Sin duda hace falta darle un mejor diseño. No es muy elegante tampoco hacer llamadas directamente a fsockopen, fwrite etc. Lo ideal es hacer nuestra clase de conexión con AMI. Un AmiConnector. De hecho, incluyo una clase mas robusta como archivo adjunto a este documento.

Hasta ahora hemos visto como ejecutar acciones y leer las respuestas. Sin embargo, en todos los ejemplos anteriores hemos asumido que justo despues de enviar la acción (fwrite()) lo único que puede venir es la respuesta a esa acción. Dos preguntas.

1. Que pasa cuando hay multiples clientes conectados con AMI y cada quien envia sus acciones de forma independiente? como sabemos que la respuesta nos corresponde?

2. Si al momento de enviar la acción, un nuevo cliente SIP se registra y Asterisk envia un evento, incorrectamente asumiremos que el evento es la respuesta a nuestra acción. Como lo solucionamos?

La respuesta a la segunda es simple. Después de leer con fread() debemos asegurarnos que lo que se lee es al menos un paquete completo (terminado por \r\n\r\n) de tipo Response, e ignorar cualquier paquete de tipo Event. La segunda pregunta es simple también, pero su implementación requiere de algun esfuerzo extra. Se necesita de un proxy. Por ahora no entraremos en la necesidad del proxy. Eso será motivo de otro documento. Por ahora para estar cerca del final de nuestro how-to, necesitamos aprender a manejar los eventos. Veamos como recibimos un evento de nuevo canal es lanzado por Asterisk.

```
<?php
/* this function reads data from the provided socket
 * and return complete packets
 * @param resource $Socket
 * @return array
 * */
function read_packets($Socket)
{
   static $buffer  = '';
   $packets        = array();
   print "reading \n";
   $buffer .= fread($Socket, 1024);
   /* at least 1 complete packet ? */
   if ( FALSE !== strpos($buffer, "\r\n\r\n") )
   {
      /* separe each packet */
      $packets = explode("\r\n\r\n", $buffer);
      /* the last packet ALWAYS is empty or not complete, save it in buffer */
      $buffer  = array_pop($packets);
   }
```

```
   return $packets;
}
/* connect and login */
$socket = fsockopen('localhost', 5038);
fread($socket, 1024); /* discard welcome message */
fwrite($socket, "Action: Login\r\n");
fwrite($socket, "Username: vaio\r\n");
fwrite($socket, "Secret: managerpass\r\n");
fwrite($socket, "\r\n");


/* infinite loop waiting for Newchannel events */
while ( TRUE  )
{
   $packets = read_packets($socket);
   foreach ( $packets as $packet )
   {
      /* is this a Newchannel packet? */
      if ( FALSE !== stripos($packet, 'Newchannel') )
      {
         $headers = explode("\r\n", $packet);
         foreach ( $headers as $header )
         {
            if ( FALSE !== stripos($header, "Channel") )
            {
               list(,$channel) = explode(':', $header);
               $channel        = trim($channel);
            }
         }
         print "New Channel: {$channel}\n";
      }
   }
}
?>
```

La salida para una llamada pudiera ser algo como esto:

reading
reading
New Channel: SIP/33-0911
reading
reading
reading
New Channel: SIP/34-195e
New Channel: SIP/34-195e
reading
reading
reading

El tiempo para esa salida es indetermidado, ya que el script permanecera esperando siempre recibir eventso de tipo Newchannel. Que pasa si en lugar de simplemente imprimir "New Channel: <canal>" hacemos algo util con la con esa información?. Como objetivo muy simple se me ocurre que podemos crear un contador de llamadas que funcione como "DataSource" para el sistema de graficación "cacti", de forma que podamos ver cuales son los minutos de mayor tráfico en el sistema.

Que pasa cuando se junta la funcionalidad del Manager con la funcionalidad de AGI? A una persona llamada David Pollak se le ocurrio hace unos años, y creo un parche para Asterisk que permite ejecutar comandos de AGI usando AMI. En la siguiente sección veremos como con este parche podemos crear un daemon ruteador de llamadas hecho en PHP. Para ello usaremos la clase adjunta AmiConnector.php y reutilizaremos la base de datos y algunas rutinas del RouteHoncho creadas en la sección de AGI.

Empezaremos por parchar Asterisk con asterisk-magi.patch incluido con este documento. Es importante entender que al aplicar el parche, dejara de funcionar la aplicación AGI por si sola. En adelante asumire que el parche se encuentra correctamente aplicado. Para comprobar que el parche se encuentra correctamente aplicado usamos el siguiente comando:

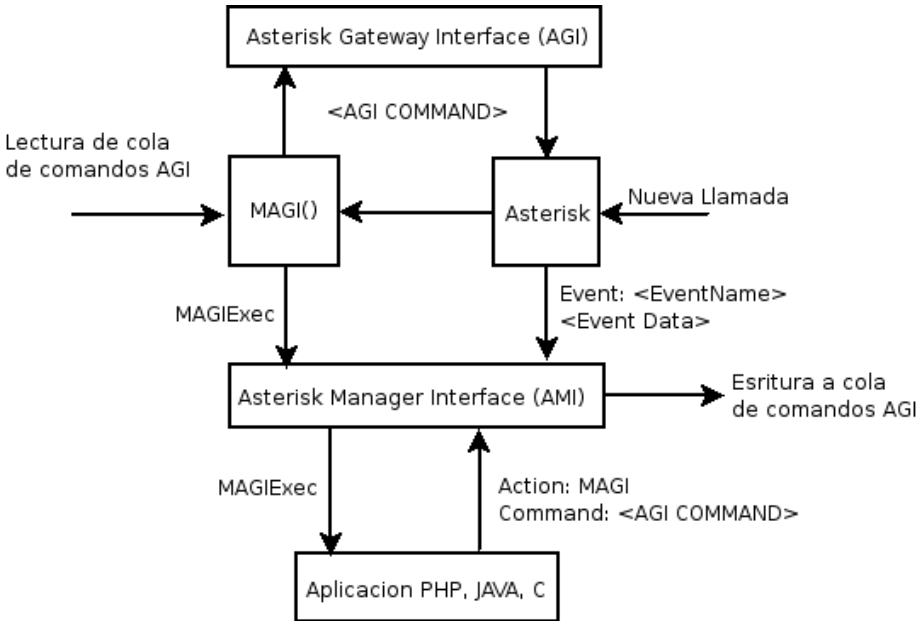
```
# asterisk -rx 'show applications' | grep -i magi
```

Ante el cual debemos ver lo siguiente:

```
 MAGI: Executes AGI commands sent to the channel via Maganer API
```

Esto significa que tenemos MAGI correctamente instalado. Como se vió anteriormente, MAGI es una mezcla entre AGI y AMI. AMI por si mismo no es suficiente para crear un ruteador de llamadas, debido a que no tiene suficientes elementos para tener el control absoluto de los canales. AGI por su parte tiene el control absoluto de los canales, pero unicamente de aquellos que lo invocan, y además un nuevo proceso debe generarse cada vez que se inicia una llamada. Es asi como entra MAGI, permitiendo que el manager ejecute comandos AGI en canales específicos. Esto permite que tengamos un script escuchando por eventos de nueva llamada, y al recibir el evento, ejecute la aplicación correcta sobre el canal que inicio la llamada, llamese Dial, Voicemail, MeetMe, Playback, o cualquiera que sea útil.

El siguiente diagrama muestra el funcionamiento de MAGI.



Para ilustrar su funcionamiento, vayamos directo al código.

```
#!/usr/bin/php
<?php
require_once 'AmiConnector.php';
try
{
   $connector = new AmiConnector();
   $connector->SetAuthenticationInfo('vaio', 'managerpass');
   $connector->ConnectToManager();
   while ( TRUE )
   {
      $packets = $connector->ReceiveManagerPackets();
      foreach ( $packets as $packet )
      {
```

```
        print_r($packet);
      }
   }
}
catch ( Exception $error )
{
   print $error->getMessage() . "\n";
}
?>
```

Este pequeño simplemente inicia una conexion con Asterisk y imprime los eventos recibidos. Ahora, en el contexto de algun extensión escribimos lo siguiente:

```
[main]
exten => _X.,1,MAGI()
```

Esto ejecutará la aplicación MAGI cuando se marque cualquier número. La aplicación MAGI lanzará un evento de MAGIExec que nosotros capturaremos desde el script anterior. Al recibirlo leeremos el nombre del canal y el número marcado para ejecutar la aplicación que corresponda. La obtención de la aplicación será usando la misma base de datos y mapeos del ejemplo del router con AGI. La diferencia aqui es que unicamente existe un proceso. Este proceso de PHP tiene el control sobre TODAS las llamadas, no solo sobre una, como cuando usamos AGI.

Tiempo de codificar!

Después de algún tiempo de codificación, el resultado es:

```
#!/usr/bin/php
<?php
require_once 'AmiConnector.php';
require_once '../agi_router/RouteHoncho.php';
Logger::SetLogLevel(100);
try
{
   $honcho    = new RouteHoncho();
   $connector = new AmiConnector();
   $connector->SetAuthenticationInfo('vaio', 'managerpass');
   $connector->ConnectToManager();
   while ( TRUE )
   {
      $packets = $connector->ReceiveManagerPackets();
      foreach ( $packets as $packet )
      {
         if ( isset($packet['Event']) && 'magiexec' == strtolower($packet['Event']) )
```

```
{
   Logger::WriteLog("Starting New call Routing");

   $channel   = $packet['Channel'];


   $magi_data = AmiConnector::ParseMagiExecResult($packet['Result']);

   /* get the dialed number by the user */
   $dialed    = $magi_data['agi_extension'];

   /* get whos calling */
   $callerid  = $magi_data['agi_callerid'];


   /* get where wants to call */
   $mapping   = $honcho->GetNumberMapping($dialed);


   Logger::WriteLog('Dialed Number: ' . $dialed);
   if ( NULL === $mapping )
   {
      Logger::WriteLog('invalid number');
      $connector->Playback($channel, 'iss_invalid_pbx_number');
      $connector->Hangup($channel);
   }
   else
   {
      Logger::WriteLog('Checking call authorization for extension: ' . $callerid);
      /* is the extension authorized to make this call? */
      if ( $honcho->CallIsAuthorized($callerid, $mapping['map_id']) )
      {
         Logger::WriteLog('Authorized mapping: ' . $mapping['name']);
         /* this can be much, much better passing control to "Application Drivers"
          * instead of having hard coded Applications
          * */
         switch ( $mapping['name'] )
         {
            case RouteHoncho::APPLICATION_DIAL:
               $connector->Dial($channel, $mapping['data'] . '/' . $dialed_number);
               break;

            case RouteHoncho::APPLICATION_VOICEMAIL:
               $connector->EnterVoiceMail($channel, $dialed_number);
               break;


            case RouteHoncho::APPLICATION_CONFERENCE:
               $connector->StartConference($channel, $dialed_number);
               break;
```

```
                    default:
                        $connector->Playback($channel, 'iss_hangup_en');
                        break;
                }
            }
            else
            {
                Logger::WriteLog('Unauthorized mapping: ' . $mapping['name']);
                $connector->Playback($channel, 'iss_invalid_pbx_number');
            }
        }
    }
    Logger::WriteLog('Sending packets');
    $connector->CommitCommands();
  }
}
catch ( Exception $error )
{
  print $error->getMessage() . "\n";
}
?>
```

He utilizado dos librerias, netsock y logger. No explicare su funcionamiento, basta saber que netsock nos sirve para crear aplicaciones con sockets (clientes y servidores) y logger es una clase para unificar mensajes de debug, warnings etc.

Este script es muy simple, no maneja "estados" de los canales. Por lo que todo se debe realizar en una sola operación, es decir, no podriamos por ejemplo solicitar digitos de marcado y esperarlos. Para ello necesitamos implementar un arreglo con los canales de las llamadas en curso y algunas variables necesarias para cada canal. De momento, creo que eso lo dejaremos para otra ocasión.

REFERENCIAS

http://www.voip-info.org (Realmente aqui encontrarán el 90% de lo que necesitan)

http://www.google.com/ (El 10% restante)