Time-Zone Processing with Asterisk, Part II

Matthew Gast

Abstract

Part II of our series on time-zone processing with Asterisk.

Last month, I wrote about a system for handling telephone calls with Asterisk that automatically handled the call depending on the time of day at a remote location. Use of the system, however, depended on the user performing the critical task of setting up the remote location’s time with a telephone call. Rather than rely on the user to initiate the telephone call manually, it would be easier if the call occurred automatically.

If the setup call occurs automatically, the user won't forget to do it. The initial SIP registration may occur at a very strange hour in the home location, but the SIP registration occurs because a user has plugged something in. Therefore, we know the user is awake and can take a short call. Asterisk provides a management interface that reports when SIP registrations occur and can be used to take action based on it. With a bit of additional processing, a script talking to the manager can initiate calls only when the SIP registration is “new”.

Introduction to the Asterisk Manager

The Asterisk Manager reports events processed by Asterisk and accepts commands over the interface. The form of the interface is a text-based protocol that separates event reports and commands into clusters of lines with a key: value format. For example, the registration of extension 300 using the SIP protocol looks like this:

```
Event: PeerStatus
PeerStatus: Registered
Peer: SIP/300
```

Before gaining access to the Asterisk manager, clients must authenticate against the list of administrative users stored in /etc/asterisk/manager.conf. Once logged in, a client can issue commands or query the value of variables with a set of lines that starts with a first line of Action:, followed by a command. Responses to commands typically start with Response: Success.

Because the protocol is text-based, it can be scripted in a language like Expect. A component is also available for the Perl Object Environment (POE), a framework that builds event-driven programs in Perl. The freely available component provides the base-level response parsing that would need to be written in Expect, so it is a much more extensible foundation for programs controlling the Asterisk manager.

The Program Core and Inline States
The main code for the program is simple. POE sets up a system where state handlers are called in response to program states. A state can be defined by the programmer or by an external event. The typical flow through the program is to notice a SIP registration, check to see whether it has an active time-zone registration and if not, to initiate a configuration call.

To execute code in response to an event, the POE framework uses a hash called CallBacks. Every entry in CallBacks defines a state based on the event received from the manager. When an event matches a callback, the handler defined for the state is triggered. To set up a trigger with the CallBacks clause, identify every line in the event and set up a hash so that the left-hand side of each line of the event is the key value for a line of the hash. As an example, consider the callback definition for the SIP registration event earlier:

```perl
Event: PeerStatus register => { 'Event' => 'PeerStatus', PeerStatus: Registered 'PeerStatus' => 'Registered', }
Peer: SIP/300
```

To link the callback to a handler, the inline_states hash has a list of states and references to the corresponding code to call. Although it is possible to inline event-handler code, for readability I have separated the code out into external procedures. Code called in response to a CallBack cannot be passed arguments:

```perl
inline_states => { register => \&register_state_handler,
}
```

Based on the flow of the program, three events are of interest. First, SIP registration events are used to start the entire process. SIP registrations typically occur hourly, so it is important to initiate the call only when a registration is the “first” registration. To prevent duplicate telephone calls from being initiated, the program will request data from both the Asterisk internal database AstDB as well as the SIP peer information. The second and third events will handle responses to commands and database queries, respectively. A fourth event will handle initiating the telephone call after receiving data back from the queries. The code I am currently using also has a state defined for unregister events, though it is a stub for an event that I am not currently using.

The core of the program is only 35 lines, most of which defines the program event states and shows what code will be used in response to those states. Note that the state of call is defined by the program and not by a callback, so the call state can be entered only by the program itself and not in response to an event from the manager. (A full listing of the program is available on the Linux Journal FTP site; see Resources.)

```perl
POE::Component::Client::Asterisk::Manager->new{
   Alias => 'monitor',
   RemoteHost => 'localhost',
   RemotePort => 5038,
   Username => 'autotzcaller',
   Password => 'secretpassword',
   CallBacks => {
      input => ':all',
      response => {
         'Response' => 'Success',
      },
   },
}```
Two of the state handlers are only stubs. The input state handler prints out whatever it gets if a debug flag is set, and it is there for development purposes. It catches any unrecognized events that come from the manager, and it can be useful when testing that callbacks are catching the important events. The unregister state handler currently doesn’t do anything, but it is there as a hook to expand if I choose in the future to take any action based on that.

With the core of the program in place, let’s look at each of the states in the order they will be called through a typical program execution flow.

### Registration Event Handling

The register state handler is called whenever a SIP registration event is received from a new extension. Its main purpose is to get the data required for setting up the configuration telephone call when a new extension pops up. Whether a call is made depends on the state of the extension as far as time-zone processing, so this routine requests information to determine whether the extension is registered, its IP address and other components. To get the extension, we have to take the channel name, which is prefaced with the technology and a slash (for example SIP/) and strip the leading part away.

One wrinkle of the event handler is that POE handlers run to completion. There is no way to interrupt a handler when it is running. The sub-procedure getTZChannelVars will request information on the time-zone offset and IP address, but that information will not become available until the registration handler completes and the responses return via the manager. At the end of the procedure, the registration handler uses the delay_set POE method to queue up the call state for a delay in the future so that the requests will have returned their information by that point. The delay is set by a global variable in the program. I have found that one second is more than adequate for a single-user PBX with only one outstanding extension requiring setup, but the delay is set to three seconds for safety.

Communication between state handlers is a bit different from that in a procedure-driven program. POE state handlers pass references to the POE kernel, which is used in scheduling, as well as the POE heap,
which is needed to issue commands to the Asterisk Manager. POE defines constants so the heap and kernel are easily accessible to event handlers as $[HEAP]$ and $[KERNEL]$. Any other information available is located at $[ARG0]$, which is a constant defined in such a way that it is the first argument.

Any lines in the event that defines the state will be passed as the hash $[ARG0]$ and are accessible by asking for the hash key that appears on the left-hand side of the desired line. In the registration response, it is possible to get at the peer extension by referring to $[ARG0]->{Peer}$, which returns SIP/300:

Event: PeerStatus
PeerStatus: Registered
Peer: SIP/300

On SIP registration, the program needs to identify the extension, request information about it and then set up further processing of the extension data after a delay. When an event is called through the delay_set method, it is possible to pass an extension to the state handler, such as the extension number used here:

```
sub register_state_handler {
    my $kernel = $[KERNEL];
    # Split peer extension off from technology
    my $peer = $[ARG0]->{Peer};
    debug_print("\tExtension is $peer; ");
    my @exten_parts = split('/',$peer);
    my $ext = @exten_parts[1];
    debug_print("extension number is $ext\n");

    getTZChannelVars($[HEAP], $ext);

    debug_print("Queuing call event for ");
    debug_print("$REG_CALL_DELAY seconds\n");
    $kernel->delay_set("call", $REG_CALL_DELAY, $ext);
}
```

As part of the extension registration process, we collect variables about the state of the channel in the getTZChannelVars procedure. The POE heap, which is passed as the first argument, can be used to issue commands to the manager. For example, the put argument to the server can be used to issue commands. To get the SIP peer data, which includes the current IP address of the peer, the command looks like this:

```
$heap->{server}->put({'Action' => 'SIPShowPeer',
    'Peer' => $ext });
```

To get a database variable, the action in the put command is a DBGet. The time-zone data is stored as keys in the tz family, so it is necessary to specify both the family and assemble the correct key name, which is of the form 300-TIMESKEW or similar:

```
$heap->{server}->put({'Action' => 'DBGet',
    'Family' => 'tz',
    'Key' => $ext . '-' . 'TIMESKEW'});
```

Four database requests and the SIP peer data are requested by getTZChannelVars. Because this function is called by an event handler, it also is not interruptible. Therefore, it sends four database query events to the manager, but it does not process responses directly. (Complete code for the five requests within the full procedure is available on the Linux Journal FTP site.)
Command and Database Responses

In the gap between issuing requests and the time the call state is scheduled, responses flow in from the SIP data request and the database queries. From the SIP data request, we need to pick out the peer IP address, which appears on a line in the manager response reading `Address-IP: 192.168.1.5`. Conveniently, the POE module parses out the lines in the response, so all we need to do is look for the Address-IP line by getting the value of the Address-IP hash element in one of the arguments passed to the handler. The POE heap is accessible across events, so adding the value of the SIP peer IP address to the heap makes it accessible to other event handlers:

```perl
sub response_state_handler {
    my $peer_ip = $_[ARG0]->{'Address-IP'};
    if (defined($peer_ip)) {
        debug_print("SIP context found; Peer IP address";
        debug_print("is $peer_ip\n");
        $_[HEAP]->{'SIP-Peer-IP'}=$peer_ip;
    }
} # response_state_handler
```

After the SIP data response comes back, the four database queries should return responses. Responses to the queries look like this:

- **DBGetResponse: Success**
- **Family**: tz
- **Key**: 300-TIMESKEW
- **Val**: -8

The callback handler is triggered whenever there is a `DBGetResponse: Success` event from the manager, with an argument of a hash that has each of the lines in the packet. Our interest is in the key and value lines, which can be retrieved from the arguments passed to the state handler. As with the previous handler, responses are stored in the POE task heap to make it available to other handlers:

```perl
sub db_response_state_handler {
    my $family = $_[ARG0]->{'Family'};
    my $key = $_[ARG0]->{'Key'};
    my $value = $_[ARG0]->{'Val'};
    if (defined($family)) {
        debug_print("Key $key in DB family $family");
        debug_print("has value = $value\n");
        # Store in heap
        $_[HEAP]->{$key} = $value;
    }
} # db_response_state_handler
```

Making the Call

Every registration event triggers a “call” event to happen after a delay. The delay is used to collect information used to determine whether to initiate a call. The setup telephone call should be triggered only if the time-zone setup has expired or the SIP device has changed its IP address and the record is no longer valid.

Because the call state handler is placed in the queue for execution by the registration handler, it does have one argument, the extension number of the call in question. The extension can be retrieved as
$_[ARG0]. All the data we have added to the heap by processing the database responses and SIP data request is also readily available:

```perl
sub call_state_handler {
    # Get extension out of arguments to function
    my $exten = $_[ARG0];
    my $hp = $_[HEAP];
    # Variables we use to determine if the call is required
    my $skew = $hp->{$exten.'-TIMESKEW'};
    my $skew_addr = $hp->{$exten.'-TIMESKEW_ADDR'};
    my $skew_start = $hp->{$exten.'-TIMESKEW_START'};
    my $skew_end = $hp->{$exten.'-TIMESKEW_END'};
    my $sip_peer_ip = $hp->{'SIP-Peer-IP'};
    my $now = time();

    To determine whether the call is required, the handler compares the current time with the expiration of the time-zone offset record and the IP address of the SIP device against the IP address stored in the time-zone offset record. If the IP addresses match and the offset has not expired, no call is required. Otherwise, a call is needed and made with the makeTZSetupCall function:

    if ($now > $skew_end) {
        debug_print("Make call - offset has expired.\n");
        makeTZSetupCall($_[HEAP], $exten);
    } elsif (!($skew_addr eq $sip_peer_ip)) {
        debug_print("Make call - SIP IP addr changed\n");
        makeTZSetupCall($_[HEAP], $exten);
    } else {
        debug_print("No call -- record OK & same IP\n");
    }

    As a final step, the handler needs to remove the variables placed on the heap. The heap is used only to pass variables between state handlers, and the variables are not needed once that function is complete. Each of the variables can be undefined with the undef function:

    # Need to clean up heap
    undef $_[HEAP]->{$exten.'-TIMESKEW'};
    undef $_[HEAP]->{$exten.'-TIMESKEW_ADDR'};
    undef $_[HEAP]->{$exten.'-TIMESKEW_START'};
    undef $_[HEAP]->{$exten.'-TIMESKEW_END'};
    undef $_[HEAP]->{'SIP-Peer-IP'};
} # call_state_handler
```

Making the setup call uses the Asterisk Manager's Originate command, but it is protected by one final check. I've defined a set of extensions as the remote channel list. Only extensions on the remote channel list will have time-zone setup calls made to them. Initially, the list consists of my softphone and an analog telephone adapter, but I may need to add more in the future. Before originating the call, I ensure that the number is on a remote channel list, which is defined in the global array REMOTE_CHANNEL_LIST. The Originate command can take several arguments as well. The extension, priority and context must refer to where the setup menu is defined. In my case, these values are extension *89 (for *-T-Z), priority 1 and the context from-internal. I also can supply the caller-ID text of “Time Zone Setup” to the phone I am calling:

```perl
sub makeTZSetupCall {
```
my $heap = $_[0];
my $exten = $_[1];
my $callOK = 0;

# Check that extension to call is a remote channel
foreach $number (@REMOTE_CHANNEL_LIST) {
    if ($number == $exten) {
        $callOK = 1;
    }
}

if ($callOK) {
    $heap->{server}->put({
        'Action' => 'Originate',
        'Channel' => 'SIP/'.$exten,
        'Context' => $TZ_CONTEXT,
        'Exten' => $TZ_EXTEN,
        'Priority' => $TZ_PRIORITY,
        'Callerid' => $CALLERID,
    });
}

} # makeTZSetupCall

If the Originate command is triggered, the newly registered telephone rings, and I go through the voice menu described in last month's article.

**Resources**


Information on the Asterisk Manager API: [http://www.voip-info.org/wiki-Asterisk+manager+API](http://www.voip-info.org/wiki-Asterisk+manager+API)

Perl POE Framework: [http://poe.perl.org](http://poe.perl.org)

Perl Asterisk Manager Component: [http://search.cpan.org/~xantus/POE-Component-Client-Asterisk-Manager-0.06/Manager.pm](http://search.cpan.org/~xantus/POE-Component-Client-Asterisk-Manager-0.06/Manager.pm)