Applying Adhearsion to Asterisk

Jay Phillips

Abstract

Tackle your sticky VoIP projects with Ruby and Adhearsion.

Following the trend of other corporate servers shifting from closed-source platforms like Windows NT and UNIX, VoIP too now booms with an exodus to the open-source Asterisk PBX and Linux. In its own picture, Asterisk looks readier than ever to meet these needs. The great folks at Digium, Asterisk's maintaining company, stress product quality and lead VoIP innovation.

Integrating technologies with VoIP also makes the picture even prettier. With standard computers crunching the calls, connecting the dots plainly makes sense. Traditional ways to integrate Asterisk really exist as hacks though, not as comprehensive solutions.

Enter Adhearsion

This new open-source framework takes on integration issues ranging from enterprise must-haves to living-room hacking projects. Written in Ruby, Adhearsion comes out of the box with more than what many companies spend thousands on. Though its name derives from being “adhesion you can hear”, it can work even without Asterisk when trying to splice two or more technologies together outside of the VoIP picture frame. Because Ruby and Adhearsion both aim to improve productivity, VoIP systems now more than ever make fun projects for your free time or effective tools in bringing your employees, customers and the world closer.

When a call comes in on an Asterisk server managed by Adhearsion, Asterisk acts as a metaphorical kernel for the call by managing low-level hardware, converting codecs, canceling echo and so forth. For the logic it should perform during the call, such as playing messages, connecting through to someone or taking input, it establishes a TCP connection to a running Adhearsion dæmon and receives commands one at a time, executing each and sending back a response.

Let's get a working Adhearsion box running to demonstrate.

Up and Running

Adhearsion, like most other Ruby projects, uses the fantastic package manager RubyGems. If you read *Linux Journal*'s July '06 issue on Ruby or have picked at the language, you've likely bumped into this great software. If not, package managers for any popular Linux distro barely differ. The Adhearsion installation process constitutes installing Ruby and RubyGems, then pulling Adhearsion down with a `gem install` command.

The installation example below uses Ubuntu/Debian's apt-get package manager. You, therefore, may need to make minor modifications to the package names for your distro. If you don't have a RubyGems system package, drop by its Web site (see Resources) and pick up the tarball:

```
apt-get install ruby1.8 ruby1.8-dev irb1.8
tar zxf rubygems*.tgz
cd rubygems*
ruby setup.rb
gem install adhearsion --include-dependencies
```

With the last command, RubyGems created the ahn (pronounced Anne) command.

You can use ahn to create new Adhearsion projects, execute an Adhearsion process normally or as a dæmon, install open-source framework extensions remotely, read help documentation and so forth. Type **`ahn help`** for more information.

This guide assumes you have an existing Asterisk server configured. If you would like more information on how to do this, see the Resources section. It takes adding only a simple instruction to the extensions.conf configuration script to have Asterisk utilize Adhearsion. If you had your phone users routed to your “internal” context, you would modify the “internal” context's definition as such:

```
[internal]
exten => _X.,1,AGI(agi://192.168.1.2)
```

Next, create a new Adhearsion project with the command:

```
ahn create ~/helloworld
```

This generates an entire directory structure with which you can mingle safely. To start up Adhearsion, supply the path to your Adhearsion directory to ahn:

```
ahn start ~/helloworld
```

Read on to learn ways you can modify this new project to suit your interests.

Make Call Instructions Fun

Traditionally, engineers route Asterisk calls in one of two ways: modifying the extensions.conf file or writing an Asterisk Gateway Interface (think CGI) script. Most generally use the former but, in situations where integration matters, one may use them together. Typically, developers write AGI scripts on the filesystem of their Asterisk machine, which communicate plain text over STDIN and STDOUT.

But, trouble arises when this scales both in complexity of code and complexity of use. Because Asterisk executes this script as a subprocess, it actually builds up whatever overhead is needed for the AGI program (for example, the Perl interpreter or a database socket), executes the code and then tears it all down again. Although tolerable in smaller situations, processes carry greater tolls on memory and running time as concurrent use increases. Additionally, without a framework, scripts in this way also contain suboptimal amounts of repetition.

Adhearsion utilizes Asterisk's FastAGI feature for managing calls from another IP-addressable machine. In this way, precious RAM and CPU cycles receive virtually no impact on the Asterisk box and, just as important, Adhearsion has already established the overhead for each call before it comes in.

With your new example Adhearsion project created earlier, open your extensions.rb file. Below, I've included an example file from which you can get a feel for how Adhearsion works to handle calls. Take a minute now to read it over:

```
# File extensions.rb
internal {
  case extension
    when 101...200
      employee = User.find_by_extension extension
      if employee.busy? then voicemail extension
      else
        dial employee, :for => 10.rings
        voicemail unless last_call_successful?
      end
    when 888
      play weather_report("Dallas Texas")
    when 999 then +joker_voicemail
  end
}

joker_voicemail {
  play %w"a-connect-charge-of 22
            cents-per-minute will-apply"
  sleep 2.seconds
  play 'just-kidding-not-upset'
  check_voicemail
}
```

If you feel adventurous, try using this now with any phones on your Asterisk box.

Note that this is all valid Ruby. Ruby fundamentally permits the modification of how aspects of the language function. As such, Ruby code can become modified for a particular need or particular domain of needs -- hence the name commonly used in the Ruby scene: domain-specific language or DSL. The full benefit of the Ruby language remains, however, giving it the best of both worlds.

Because our example Asterisk extensions.conf from above invokes Adhearsion within a context named internal, Adhearsion maps this directly over by convention. Adhearsion creates many call-related variables as simple local variables before it processes a context. The case statement for the auto-generated extension variable used shows Ruby's great support for ranges of numbers, but a Perl-like regular expression literal could very well replace this.

The next line **`employee = User.find_by_extension extension`** may prove surprisingly subtle. You easily can infer there exists some collection of users with extensions and some method of finding them by their extension. If ActiveRecord is new to you, you probably didn't realize immediately that this actually abstracts database access. Before diving into a more complete explanation of how this works, let's finish grokking this file.

Implementations for determining whether the user should be disturbed can vary, but we need to know only a yes or no answer. Users are allowed to remain focused by having callers sent silently to their voice mail.

Ruby Methods *9519s1.qrk*

Notice in the example how Ruby methods generally differ from other programming languages. Calling a method does not require the parentheses when no ambiguity exists. This may seem strange at first, but if you can convince yourself to accept it, you may just find reading and writing Ruby code somewhat enjoyable.

Also, Ruby allows question and exclamation marks to conclude a method name to better express meaning. This follows Ruby's guiding Principle of Least Surprise to identify methods typically used to represent some condition on which the user may act or, with the exclamation mark, to signify optionally that the method is destructive to the variable.

Users dialing 888 have a weather report converted from an Internet source to a series of played sound files that come with the standard asterisk-sounds package. If they dial 999, things get fancy -- the plus sign before joker_voicemail does just what you might think: it executes the joker_voicemail block.

This joker_voicemail section uses `%w""`, a fantastic Ruby literal for an array of words. Automatically, Ruby will break apart this string by whitespace, creating an array containing a series of sounds intuitively named for the speech they produce. But you ask, “What about this 22 here?”

When Adhearsion encounters numbers, instead of executing the Playback() application, it uses SayNumbers(). This breaks the number into the words twenty and two -- two words for which sounds do exist. The end result contains no commas, no quotes between indices and no specification of the application used to play the file. Why should it?

You can find more detailed, working dial plans and documentation freely on the Adhearsion Web site. See the Resources section for more information.

Powerfully Use Your Database Software

If you've checked out Ruby on Rails, you likely know of its rock-star object relational mapper ActiveRecord. Instead of keeping ActiveRecord deeply intertwined with the Rails framework, the brilliant folks over at 37signals allow anyone to utilize this fantastic library through a gem. When you install Adhearsion, the **`--include-dependencies`** part pulls this in for you.

From a database modeling perspective, typical VoIP applications have users and groups of users. Therefore, in our database -- whether you use MySQL, PostgreSQL, SQLite or nearly any other popular relational database management system -- it makes sense to have a table called users and a table called groups. Because each record represents an individual user or group in the database, it also makes sense to have a User object and a Group object. Beginning to see how an object relational mapper applies here?

Here's a tangible, tasty example:

```
class User < ActiveRecord::Base
    belongs_to :group
    validates_uniqueness_of :extension
    def busy?
      # Implemented how you wish
    end
end

class Group < ActiveRecord::Base
   has_many :users
end
```

This is where the magic happens. With so little code, ActiveRecord can actually make a lot of very logical conclusions. Given the class names User and Group, it finds their appropriate table by lowercasing them and then finding the plural forms users and groups, respectively. If we had used the class name Person instead of User, ActiveRecord would have assumed its associated table to be people.

To help ourselves out, we've told ActiveRecord that a User belongs to a Group and that Groups have many users. Given this, the foreign key to which a record in users maps is assumed to be group_id by default. With this identified, one can retrieve the Group object to which a User instance belongs simply by calling jay.group. If the jay variable belonged to the Group codemecca, jay and any other potential variables could be retrieved by codemecca.users.

So let's take a look at the dial plan example above. We're calling a method on User by the name of find_by_extension. Did we have to create that method anywhere here? No. But why not? ActiveRecord actually created the method, because it peeked inside the users table, found the names of its columns and created methods like find_by_extension. Explicated into a MySQL select statement, it would read **`SELECT * FROM users WHERE EXTENSION='somenumber' LIMIT 1;`**. Nice shortcut, eh?

Use VoIP and the Web Together to Collaborate

In a corporate environment, businesses depend on collaboration, and people look for any way to improve it with good reason. Adhearsion offers a free collaboration tool that, in the spirit of adhering everything together, takes traditional collaboration a step further.

Modern IP desk phones generally have a microbrowser that can pull down XML documents over HTTP and translate them to an interactive menu on the device. Although this may seem like a great technology, there's a significant caveat: every vendor's XML format differs, none document the feature very well and the available features vary vastly. Some vendors support images, HTML forms and a special URI for initializing a call, whereas others give you about three or four XML elements from which you can choose.

If vendors can't collaborate on consistency, something must abstract their differences (and quirks) by translating. Micromenus do exactly that with a few hints of additional cleverness.

Since the Micromenus sub-framework exists as a built-in helper, you manage the menus in the config/helpers/micromenus/ directory. When an HTTP request comes in over port 1337, Adhearsion takes the first argument in the URL's path as the desired filename. For example, if you programmed a phone to use http://192.168.1.228/main as its Micromenus URL, Adhearsion would attempt loading the file config/helpers/micromenus/main.rb. Modifying and creating files here takes effect even without restarting Adhearsion.

Using this example filename, let's fill it with some Micromenu code:

```
# config/helpers/micromenus/main.rb
call "Check your voicemail" do
    check_voicemail
end

call 505, "Call Support"

item "Join a conference" do

    call 'Join Marketing' do
        join :marketing
    end
    call 'Join Support' do
        join :support
    end

    call 'Join Ninjas' do
        join :ninjas
    end
end

item "Weather forecasts" do
    call "New York, New York" do
        play weather_report('New York, New York')
     end

    call "San Jose, California" do
    play weather_report('San Jose, California')
    end

    call "University of Texas at Dallas" do
      play weather_report(75080)
    end
end

item "System Administration" do
    item 'Uptime: ' + %x'uptime'
    item "View Current SIP users" do
        PBX.sip_users.each do |user|
            item "#{user[:username]@#{user[:ip]}"
        end
    end
end
```

This simple example, albeit very terse, produces an entire, respectable company intranet. Giving the item method a block automatically turns it into a link, abstracting the URL. Placing actual Ruby code in a submenu gives the menu behavior. Using the call method on a phone actually places the call to either the extension specified (without a block) or, because Adhearsion handles the call itself too, executes the dial plan behavior within the block when it finds the user generated a call through a Micromenu. This exemplifies the benefits of having both call routing and Web-based collaboration adhered this closely together in the same framework.

Create Framework Extensions

As any IP telephony engineer can affirm, taking someone else's VoIP functionality and merging it with your own is not for the faint of heart. Nor can you find VoIP functionality in abundance on-line. The sheer difficulty of reusing this kind of code severely discourages trading. Standard PBX dial plan configurations are typically meaningless without the half-dozen other configuration files that put them into perspective.

Not only does Adhearsion allow others easily to integrate with their VoIP applications, it facilitates the integration. The Adhearsion Web site hosts a nice community where users can submit, tag, browse and rate extensions. All of these can be freely downloaded and copied to your helpers/ directory where Adhearsion will find it and automatically absorb its features into the framework. Extensions can vary from adding a new method to the dial plan DSL to entire Web servers that run in a separate thread.

Let's say in a fit of lunacy you decide to write a VoIP calculator application that speaks back an answer. Because you have many different mathematical operations from which to choose, you decide to implement a factorial method and expose it to the entire framework. This requires simply creating the file helpers/factorial.rb and adding the following code to it:

```
# helpers/factorial.rb
def your_factorial num
   (1..num).inject { |sum,n| n+sum }
end
```

When you start Adhearsion, you'll have the ability to use this in your dial plan, your Micromenus, Adhearsion's distributed computing servers, and any other nook or cranny of the framework. But, you say this doesn't cut it.

Like any dynamically typed language, this simply takes too long for very, very large numbers. Wouldn't it be nice if we could write this extension in C? Well, we can.

The terrific third-party library RubyInline takes a string of C/C++ code, read from a file or otherwise, and automatically compiles with the Ruby source headers, caches, and dynamically loads it into the Ruby interpreter.   The library even finds any method signatures and creates a matching Ruby method. Static return types automatically convert to Ruby equivalents (int to Fixnum, double to Float) when the native code finishes its business. With this library, Adhearsion allows more efficient extensions to the framework with languages other than Ruby.

Because RubyInline requires the Ruby development headers and a configured compiling environment, it doesn't come in as an Adhearsion dependency. If you have GCC and its peripheral development requirements, do a **`gem install RubyInline`**, and throw this code in the file helpers/factorial.alien.c:

```
int fast_factorial(int num) {
    int sum = 0, counter = 1;
    while(counter <= num) {
        sum += counter++;
    }
    return sum;
}
```

Like other extensions, Adhearsion automatically finds this file and hooks its functionality into the framework. If the C code requires special compile instructions or include statements, you easily can add these to the helper's config file, which all helpers can optionally have. These config files exist in the config/helpers directory with the same name as the helper to which they belong but with the YAML .yml extension.

Let your imagination run away with you. If you come up with a great new idea for a VoIP system, the Adhearsion extension architecture serves as a great launchpad to materialize your concepts easily.

Resources *9519s2.qrk*

Adhearsion Web Site: http://adhearsion.com

PwnYourPhone, Official Adhearsion Video Podcast: http://PwnYourPhone.com

Official Adhearsion Blog: http://jicksta.com

VoIP-Info Wiki: http://voip-info.org

Adhearsion Wiki: http://docs.adhearsion.com

Codemecca Web Site: http://codemecca.com

RubyGems Web Site: http://rubyforge.org/projects/rubygems