Pyastre: A Python in the Dialplan

version 0.3
14 May 2005

1 Introduction

Pyastre is a module written in C and some helper classes in Python for the Asterisk PBX. It allows one to run a persistent python process within Asterisk, as well as to run python scripts from the dialplan.

Pyastre started out with frustration at the Asterisk Manager Interface. The AMI is a TCP protocol that makes it possible to get status information out of, and to send control messages to Asterisk. The authors of the AMI were, it seems, more concerned with making the PBX part of Asterisk work right than with designing a proper remote procedure protocol, so they made it very simple: some key-value pairs separated by newlines. It is a simplistic protocol, sometimes ambiguous, but most of the time it works.

The particular task at hand was to implement a "Summon" macro on a MoinMoin Wiki. The macro would take two user supplied arguments: two telephone numbers. When a button is pressed on the web page, those two phones ring and get bridged together. This could be done using the AMI, but at the time there was no Python library for doing this, and writing such a library for an underdesigned protocol wasn't very interesting.

There are, however several RPC protocols out there that are implemented in a variety of languages that are much more flexible. XMLRPC is a good example. Wouldn't it be nice to easily and painlessly make XMLRPC calls to control Asterisk?

Browsing through the Asterisk source code revealed a function called ast_originate that did pretty much what was needed. In fact the AMI uses this function under the hood. But how to expose this to the Wiki code most expediently?

And so PyAsterisk[1] was written to provide a persistent python process, and this process would run an XMLRPC server and expose ast_originate over the network. It is, of course, trivial to write an XMLRPC server in Python. Doing the same thing in C is quite a bit more complicated, and, it turns out, quite a lot less reuseable.

After running this for a while, the need came to store more data in call detail records than is available in the standard ones. Rather than hack the cdr_pgsql.c handler, and have to keep applying patches to new releases of Asterisk, better to write a programmable handler that can get information from wherever it wants and stuff it in the database, no? So a CDR handler was written that passes off the information to a python function for processing.

It was also about this time that it started making sense to use the Twisted framework. Twisted is a bunch of python modules for running servers and talking to databases. It also does so asynchronously – which is a big bonus since you don't want your python scripts tying up resources unnecessarily or, worse, causing something important with Asterisk to block.

So, now we had two entry points into the Python interpreter, running in different threads: from the outside world via whatever servers the persistent process was providing, and via the CDR handler. Adding another, to make the Python interpreter accessible from the call processing logic in the dialplan was then straightforward. This makes it easy to do things like make complicated routing decisions based on information in one (or even several!) databases without having to fork a new process and make a fresh database connection

[1] As PyAsterisk was being written, we were made aware of another project called py-asterisk to write a bunch of classes implementing the AMI in Python. In order to avoid ambiguity, PyAsterisk was renamed to Pyastre.

as is the case with the Application Gateway Interface or AGI, or to do complicated and error-prone operations in C where the bottleneck is not processing speed.

So then, there are three main parts to Pyastre: the Persistant Process, the Call Detail Record Handler and the Dialplan Application. There are also some convenient helper modules implemented along the way in Python. Respectively, these allow one to expose internal Asterisk API functions to 3rd party applications over the network and manage database connections, flexibly process and store state and logging information, and script the call processing behaviour of Asterisk itself.

2 Building and Installing Pyastre

To build Pyastre, Python is required. Version 2.4 is recommended, though it is known to work with 2.3. SWIG is also required. The zope.interface module is required for some of the supporting Python code. Installing the Twisted Framework is strongly suggested.

2.1 Preparing Asterisk

Unfortunately, since in Python parlance we are both extending and embedding the interpreter, the module that this produces, *_pyastre.so*, is not linked with libpython. The standard build infrastructure does not support doing this very easily. The workaround, so that the basic Python interpreter is made available to the module, is to link Asterisk with libpython. This involves changing the Asterisk Makefile and rebuilding it. The easiest way to do this is to find the line that looks like this:

```
LIBS+=-lssl
```

and change it to read:

```
LIBS+=-lssl -lpython2.4
```

and then recompile and re-install Asterisk.

2.2 Building Pyastre

Once you have the source distribution of Pyastre, uncompress it as usual. The main installation step is the same as for most Python programs and modules:

```
Compiling the Pyastre module

% python setup.py install
```

2.3 Loading Pyastre

The previous step will have produced a C module in

```
$PYTHON_LIBDIR/site-python/_pyastre.so
```

This is the Asterisk module. Asterisk has to be told to load it. There are two ways to do this. One is to add a line with the full path to */etc/asterisk/modules.conf*.

```
Loading the module in modules.conf

load => $PYTHON_LIBDIR/site-python/_pyastre.so
```

The other is to make a symlink to this file in */usr/lib/asterisk/modules*, but this will only work if you have *autoload => yes* in modules.conf.

3 Writing Scripts for Pyastre

Typically Python scripts for Pyastre will live in the directory */etc/asterisk/site-python*, though they may live anywhere in the $PYTHONPATH. There is only one special script that must exist, and it is called *softswitch.py*. It contains the Python side of the glue to the interpreter embedded in Asterisk.

softswitch.py must contain three functions: *run*, *stop* and *cdr*. *run* is called with no arguments when Pyastre is loaded. It is the main thread and should not return until stop is called upon unload. *stop* is also called with no arguments. *cdr* is called with an opaque call detail record object. There exists a helper class to convert this opaque object into something useful.

```
## A Skeletal softswitch.py with Twisted

from twisted.internet import reactor
def run():
    reactor.run()

def stop():
    reactor.stop()

def cdr(cdrobj):
    from pyastre.cdr import CallDetailRecord
    c = CallDetailRecord(cdrobj)
    ### do something with c
```

Typical usage also will have either the top-level code in *softswitch.py* or the *run* method make connections to databases and prepare other persistant resources that may be needed by scripts, etc.

3.1 Persistant Process

It is crucially important to remember that the persistent Python process in Pyastre, what can be thought of as the main thread of the program, does not exit. If it exits, the Python Interpreter will not be available to the other threads that call into it (such as the Dialplan Application and the Call Detail Record handler).

If you don't use Twisted, you will have to find some way of preventing softswitch.run from returning. One idiom that has been used is:

```
## Kludge to keep the persistant process running

from threading import Event
kludgeE = Event()

def run():
    kludgeE.wait()

def stop():
    kludgeE.set()
```

This is not very elegant, but it will do the job if all you want to do is something simple that doesn't require any sort of persistance. In most cases it is much better to use Twisted.

3.2 Call Detail Record Handler

The CDR handler should collect what information it needs and return as quickly as possible in order to free up channel resources. If possible it should queue the data for storage rather than store it immediately.

As this handler was written to get at and save more information than is normally possible, naturally there is a convenient technique for collecting data during life of a call. Asterisk has a concept of *channel variables* which can be set from the dialplan. Say for example you wanted to store the colour of a caller's shoes, perhaps based on some sort of IVR system. You might collect this information from the user and then do something like:

```
;; Seting a Channel Variable in extensions.conf

exten => 1234,1,SetVar(SHOES=RED)
```

This information can then be retrieved in the CDR handler, except that a CDR object is passed to the handler, not the channel itself. Fortunately the CDR object contains the channel name and Pyastre exposes a function to use this to get at the channel. Once you have the channel object, the variables are accessible using dictionary syntax.

```
## CDR Handler Example

def cdr(cdrobj):
    from pyastre.cdr import CallDetailRecord
    from pyastre.channel import Channel
    from asterisk import ast_channel_byname

    c = CallDetailRecord(cdrobj)
    ch = Channel(ast_channel_byname(c))

    c.shoes = ch["SHOES"]
```

TODO: modify the CallDetailRecord class to automatically make available the associated Channel class.

3.3 Dialplan Application

Writing a dialplan application is very similar to writing a MoinMoin Wiki macro. For example,

```
;; Calling Python from the Dialplan

exten => 1234,1,Python(myscript,arg1,arg2,arg3)
```

requires that there is a module somewhere in the $PYTHONPATH called *myscript* which has an *execute* method. Usually this module will exist in */etc/asterisk/site-python/myscript.py*, and a trivial example might look like this:

```
## myscript.py

def execute(c, args):
    ## c is an opaque channel object representing the current channel
    from pyastre.channel import Channel
    chan = Channel(c)

    ## as with a normal Asterisk application, we need to parse the
    ## | separated arguments
    args = args.split('|')[1:]

    ## write something to the console
    from pyastre.utils import verbose
    verbose("%s" % args)
```

Unlike the CDR handler, the Dialplan Application is intended to be synchronous. This means that if you intend to use an asynchronous framework such as Twisted, you need to arrange that the execute method does not return until the work is finished, so that the call processing doesn't continue prematurely. The way to do this is the same as you would when using Twisted from within a MoinMoinMacro:

```
## Twisted from the Dialplan

def _work(chan, event, *args):
    deferred = some.twisted.async.call(...)
    d.addCallback(lambda x: event.set())
    d.addErrback(lambda x: event.set())

def execute(c, args):
    chan = Channel(c)
    args = args.split('|')[1:]

    from threading import Event
    e = Event()
    reactor.callInThread(_work, chan, e, *args)
    e.wait()
```

4 Extending Pyastre

What happens if you need to make a call into the Asterisk C API and there's no Python function to do it? Unfortunately from this perspective Pyastre is incomplete; it does not contain all of the hooks to every C function in Asterisk. Some of the more common and useful ones are there, but most are not. To add a Python hook, you need to recompile Pyastre.

In the source code distribution of Pyastre there is a directory called *_pyastre/* which is where C and SWIG source code lives. It is a very good idea to read the SWIG documentation do see how this works, but suppose there is a very simple function, *ast_foo(struct ast_channel *)* which returns an integer. Including a reference to this might be as simple as adding the line

```
int ast_foo(struct ast_channel *);
```

to the *pyastre.i* file in the appropriate place.

If you need something more complicated than just calling into the C API, you may need to inline a wrapper function. This is often the case when the C function would block – and we certainly don't want the whole Python process and all of the interpreters to block when we call a function. A good example of this is *pbx_exec()*. In pyastre.i you will find an inline section that defines *ast_exec()* as

```
/* ast_exec() -- reentrant wrapper around pbx_exec */
%inline %{
    int ast_exec(struct ast_channel *c, struct ast_app *app,
                 char *args, int flags) {
        int ret;
        Py_BEGIN_ALLOW_THREADS
            ret = pbx_exec(c, app, args, flags);
        Py_END_ALLOW_THREADS;
        return ret;
    }
%}
```

In this case the Py_BEGIN/END_ALLOW_THREADS pair allows other Python threads to run while we are waiting for pbx_exec to return, so the whole thing doesn't block while waiting for us.

Once the desired modifications have been made, simply rebuild Pyastre and restart Asterisk.

5 Your Rights

Writing Free Software for telephony applications is a licensing minefield. The root of many of the problems is the prevalence of patent-encumbered codecs that are nevertheless in wide use and must be supported if there is any hope of interoperating with other equipment in the field. I have thought long and hard about the most appropriate license for this software to be released into that environment in order to perserve as many rights for the user and programmer as possible and yet to allow the use of certain necessary bits of non-free software.

5.1 This Manual

This manual is released under the terms of the GNU Free Documentation License (FDL). A copy of the GNU FDL is available in the *doc/* subdirectory of the source distribution.

While the license itself contains the canonical terms of distribution and modification and should be deferred to in all cases where there is a conflict or ambiguity between it and this text, loosely this means that:

- You may read, copy and distribute this manual without modification.
- You may modify this manual so long as the original copyright notice remains intact.
- You may distribute modified versions of this manual so long as you make available the modified Texinfo source and so long as the modified version is also distributed under terms of the GNU FDL.
- You may translate this manual into any language.
- You may distribute translated versions of this manual under terms of the GNU FDL.
- Any distribution of a modified or translated version of this manual must contain either the Texinfo source or instructions on where to obtain the Texinfo source so that others may likewise modify, translate and redistribute it.

5.2 The Software

Originally, the intent was to distribute the Pyastre software under the GNU General Public License (GPL). Because of the issues relating to patent-encumbered codecs alluded to above, it quickly became apparent that this was impractical. Rather than add clauses to the GPL, it was decided to release it under the GNU Lesser or Library GPL (LGPL), which is designed specifically for the situation where it is impractical to forbid linking of Free Software with Non-Free Software.

As with the documentation, in the case of any ambiguity or conflict between this text and the LGPL, the text of the LGPL shall prevail.

Loosely the distribution terms of the LGPL mean

- You may use, compile, copy and distribute the software.
- You may modify the software to suit your purposes.
- You may distribute modified copies of the software as long as the source code is made available, and as long as the modified copies are likewise released under the terms of the LGPL.

You are encouraged to notify the author of any changes you make to the software so that, if appropriate, they may be incorporated into new versions.

Table of Contents