

Rising Sun Pictures

Earth Daemon Plugins

October 26, 2008

Contents

1	Introduction	2
2	Exploration	2
3	Security Retrofitting	3
4	Plugin Framework Re-implementation	4
5	Plugin Creation	7
6	Installation	7
7	Uninstallation	8

List of Figures

1	Plugin standard code structure.	9
2	Plugin loading flow.	10
3	The re-implemented plugin framework.	11

1 Introduction

The Earth Daemon plugin, or just simply Earth plugin, was originally implemented in the existing Earth design. However, this feature was turned off. After three sprints, this feature was further explored.

It was found that the main engine of Earth, the File Monitor, was implemented as a plugin. However, the plugin feature was turned off, which was indicated by the flag `LOAD_FILE_MONITOR_AS_PLUGIN` being set to `FALSE`. The solution to this problem is to update the flag to `TRUE`. This potential solution had created a new problem, as there was no clear understanding of how plugins work in Earth.

The following sections will present the exploration conducted on the plugins framework, and the modification performed. Also, a quick guide to create and install a plugin.

2 Exploration

An exploration on getting a better understanding of the plugin framework was done. It was found that each plugin is a sub-class of the `EarthPlugin` class. This was discovered while re-reading the `file_monitor.rb` file. Figure 1 shows the basic structure of an Earth plugin. The codes shown in the mentioned figure are compulsory methods as the File Monitor uses them to retrieve information about the plugin. These compulsory methods will be revisited in later sections.

A further exploration was done to understand how a plugin is invoked from the daemon. It was found that each plugin is stored and retrieved from the database. The table that stores the plugins is the `plugins_descriptors` table. The following is the table's schema:

```
create_table "plugin_descriptors", :force => true do |t|
  t.string "name", :limit => 64, :null => false
  t.integer "version", :null => false
  t.binary "code", :null => false
  t.binary "sha1_signature", :null => false
end

add_index "plugin_descriptors", ["name"], :name => "plugin_descriptors_unique_name",
      :unique => true
```

This table will be revisited in later sections as well as further investigation found that the `code` and `sha1_signature` columns are not compatible with the PostgreSQL RubyGem connector. Figure 2 shows the flow of plugins being loaded in Earth.

It is now understood that the reason why the plugins are stored and loaded from database, instead of loading them from individual files. This is because each daemon is executed in different machines on the network, and the daemons relay files and directories information between the main database server and the file servers that they are monitoring. Thus, for a quick and clean plugin loading, the best solution is to have them loaded remotely and dynamically from a database, provided the plugin is not too complex.

3 Security Retrofitting

From the Figure 2, it is understood that plugins are to be dynamically and remotely loaded from the database, and there is a security processing overhead imposed on the load time. This is a mandatory overhead imposed as remote plugin calls can be susceptible to ‘man in the middle attack’, which a hacker can modify the codes during transmission for any malicious act. The challenge now is to have File Monitor loaded into the database. However, it was found that the `sign_plugin` and `create_cert` scripts were not updated to be compatible with Rails 2.0.2. The following codes were updated to `require_gem 'termios'`:

```
gem 'termios'
require 'termios'
```

The listed codes are the new syntax statements used to include the `termios` gem. After the certifications, signing and installation, it was later found that the plugin do not load properly when the flag was set to `true`. Further investigation was performed, and it was found that the both the `code` and `sha1_signature` columns have been ‘flatten’ to UTF codes, instead of the original characters used. Thus causing the verification to fail. It was also found that even without the verification, the plugin failed to load. As such, a mitigation action was performed to have this problem fixed. It was speculated that perhaps it is this problem that the flag was turned off. This is a known problem between PostgreSQL and Ruby, where the data being transferred are ‘lost in translation’ between UTF and plaintext when the data are stored in `binary` type. The following is the fix towards this problem:

```
class ChangePluginDescriptorCodeSignatureColumnsToText < ActiveRecord::Migration
  def self.up
    change_column :plugin_descriptors, :code, :text
    change_column :plugin_descriptors, :sha1_signature, :text
  end

  def self.down
    change_column :plugin_descriptors, :code, :binary
    change_column :plugin_descriptors, :sha1_signature, :binary
  end
end
```

This is the first step towards fixing the problem, which is to have the `binary` columns updated to `text` columns. The second step is to encode the codes and signatures using a reversible encoding, which plaintext is used as the encoded text. The solution is to use the Base64 encoding class that is bundled with Ruby. The following changes were added into the `plugin_manager.rb` file, which is the plugin management class:

```
# To load it into the database
b64_code = Base64.b64encode(code)
```

```

b64_signature = Base64.b64encode(signature)

# To load from the database
code = Base64.decode64(newPlugin.code)
signature = Base64.decode64(newPlugin.sha1_signature)

```

This will allow the code and signature to be stored and loaded from the database without losing integrity.

4 Plugin Framework Re-implementation

After the first rounds of retrofitting, another challenge arise, which is to allow a plugin to be plug-able as well. In the other words to have Earth support plugins of plugins. In the beginning, the idea of re-writing Earth to introduce an Application Programmable Interface (API) was used. However, this idea was gradually dropped due to the very monolithic design. Thus, a new idea was explored, which is to include extensions into the plugins. Figure 3 shows the re-implemented plugin framework, from the `earthd`'s point of view.

The current plugin framework do not support such a framework. Thus, a new extension module was developed, which is included when the daemon is launched. Listing 1 presents the extension module. The following line can be added into the plugin intended for extension:

```

extension_point(<the extension point name>,self.class.to_s,:file => new_file)

```

Listing 1: The extensions.rb.

```

module Extensions

  # The +extension_point+ method is used to create an extension point
  # inside a plugin file
  # It instantiates all plugins which are linked to a specific extension
  # point
  # The extension point (in the extension-points table) should be created
  # while installing the plugin using the +add_extension_point+ method
  # Note: except the extension point at the Earthd which is created when
  #       we create the extension-points table
  def extension_point(extension_point_name, host_plugin, *args)
    plugins = []
    #args are the paremetrs should be passed to the plug-in
    #so, put them in the plug-in session
    args.each do |arg|
      $plugin_session = arg
    end
  end
end

```

```

plugin_manager = PluginManager.new

#temporary method to get the class name
host_plugin = extract_plugin_name(host_plugin)

# bring all the plug-ins for this extension point using: host_plugin
# name and the extension_point name
extension_point = Earth::ExtensionPoint.find(:first ,
                                              :conditions => {
                                              :host_plugin => host_plugin ,
                                              :name => extension_point_name
                                              })

#debugger
plugins = extension_point.plugin_descriptors unless extension_point.nil?
for p in plugins do
  # instantiate the plugin class
  #TODO DELETE ME
  #plugin = get_class_from_name(p.name)

  plugin = plugin_manager.load_plugin(p.name, p.version)

  #execute a specific method
  eval 'plugin' + '.' + p.method unless p.method.nil?
end

#clear the plugin_session
$plugin_session = {}
end

# the +add_extension_point+ method creates a new extension point in the
# extension_points table
# this method should be used while installing new plugin
def add_extension_point(name, host_plugin, description)
  Earth::ExtensionPoint.create :name => name, :host_plugin => host_plugin ,
                              :description => description
end

#TODO method comments
def get_class_from_name(name)
  eval name + '.new'
end

```

```

private

def extract_plugin_name(name)
  tokens = name.split('::')
  tokens.last unless tokens.nil?
end

end

```

Therefore, to re-implement the plugin framework to use extension, a new table was added into the database, and two new columns were added to the `plugin_descriptors` table as well:

```

create_table :extension_points do |t|
  t.column :name, :string, :null => false
  #TODO this could be changed to be a foreign key to the plug-in descriptors
  #currently, just leave it as a class name (String)
  t.column :host_plugin, :string, :null => false
  t.column :description, :text, :null => false
end

#add a foreign key to the plug-in descriptor table
add_column :plugin_descriptors, :extension_point_id, :integer

#TODO this is temporarily until we read the code from the database
change_column :plugin_descriptors, :code, :text, :null => true
change_column :plugin_descriptors, :sha1_signature, :text, :null => true

#add the first extension point which is located at Earthd
ext = Earth::ExtensionPoint.create :name => "main_loop",
                                   :host_plugin => "Earthd",
                                   :description => "extension point at the
                                                  main loop in the earthd script"

#add file_monitor as a plugin for Earthd
ext.reload

```

The final step for re-implementation is to make the daemon extendable by adding the `main_loop` point, which is shown in the following:

```

while true
  logger.info("Inside main loop")

  # create an extension point here

```

```

# to extend the earthd functionality on each iteration
# if no plugins are installed, earthd will do nothing
extension_point("main_loop", self.class.to_s,
               :logger => logger,
               :cache => cache,
               :only_initial_update => false,
               :force_update_time => @config["override_update_interval"])
end

```

The plugin framework is now re-implemented.

5 Plugin Creation

To create a plugin, all one needed to do is to follow the coding scheme as shown in Figure 1. To illustrate how can this be done, the following is a sample code of a blank plugin called Mr Bogus:

```

class BogusPlugin < EarthPlugin
  def self.plugin_name
    "EarthBogusPlugin"
  end

  def self.plugin_version
    1
  end

  def initialize
    #bring the parameters from the plug-in session
    @logger = get_param(:logger)
    @logger.debug("Mr Bogus... he is the hero!!");
  end
end

```

This plugin will be loaded by the daemon and prints out `Mr Bogus... he is the hero!!` in the log file, which can be found in the `temp/earthd.log` file. As compared to the original listing in Figure 1, some of the methods were dropped as this plugin is too simple to have those methods. However, the listed methods above is important and must not be dropped.

6 Installation

The following are the steps to have a plugin installed:

1. Run `script/create_cert` to generate a certificate of the host system. (One will need to create directories `config/certificates` and `config/keys` in order this script to work.

2. Run `script/earth_plugins sign <plugin_file>` to create a `*.sha1` signature file.
3. (Optional) Run `script/console` to check whether the signature is valid. (NOTE: At this stage, one should have created 3 files: `config/certificates/test_cert.pem`, `config/keys/test_key.pem` and `<plugin>.rb.sha1`)
 - (a) Run `signature = File.read('‘<plugin>.rb.sha1’')`
 - (b) Run `code = File.read('‘<plugin>.rb’')`
 - (c) Run `cert = OpenSSL::X509::Certificate.new(
 File::read('‘config/certificates/test_cert.pem’'))`
 - (d) Run `cert.public_key.verify(OpenSSL::Digest::SHA1.new, signature, code)`. (One should get `true` as the result after running this line.)
4. Run `script/earth_plugins install <plugin_file> <extension_point> <attached_plugin>` to install. The extension point is `main_loop` if the plugin is attached to the daemon, which is `Earthd`.

At this stage, the plugin should be installed, and the `earth_plugins` script will print out all the Base64 encoding on screen without any error. To verify the plugin is indeed installed, one can invoke the following command, which will list all the installed plugins in the database:

```
script/earth_plugins list info
```

7 Uninstallation

To uninstall a plugin, the following command can be used:

```
script/earth_plugins uninstall <plugin_name>
```

One can obtain the plugin name from the list command, as stated in the previous section.

```

class MyPlugin < EarthPlugin

  # required for all plugins
  @status_info = "Mr Bogus is waking up..."
  @logger = nil

  def status_info
    @status_info
  end

  def status_info=(status_string)
    @status_info = status_string
  end

  def self.plugin_name
    "MyPlugin"
  end

  def self.plugin_version
    007
  end

  def logger=(logger)
    @logger = logger
  end

  def logger
    @logger || RAILS_DEFAULT_LOGGER
  end

  ...
end

```

Figure 1: Plugin standard code structure.

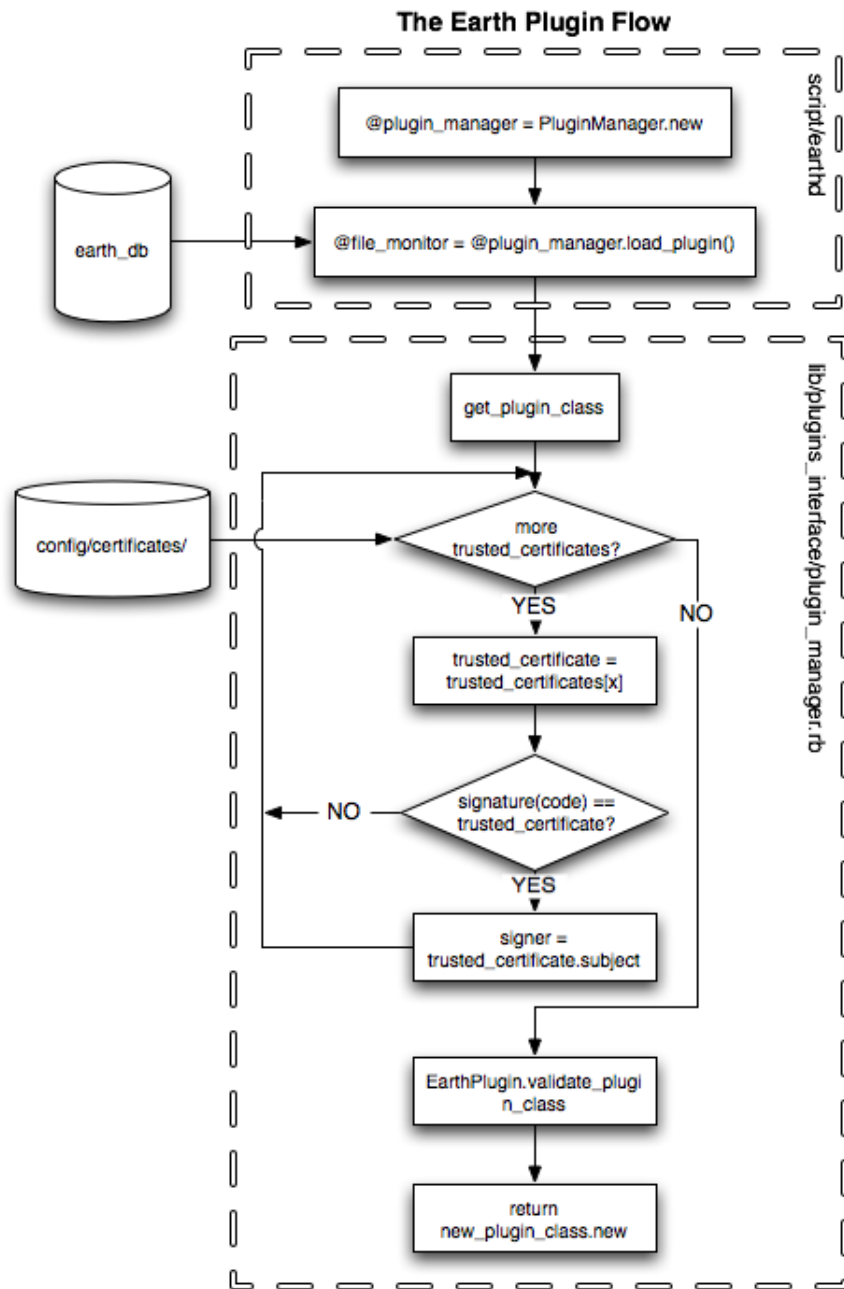


Figure 2: Plugin loading flow.

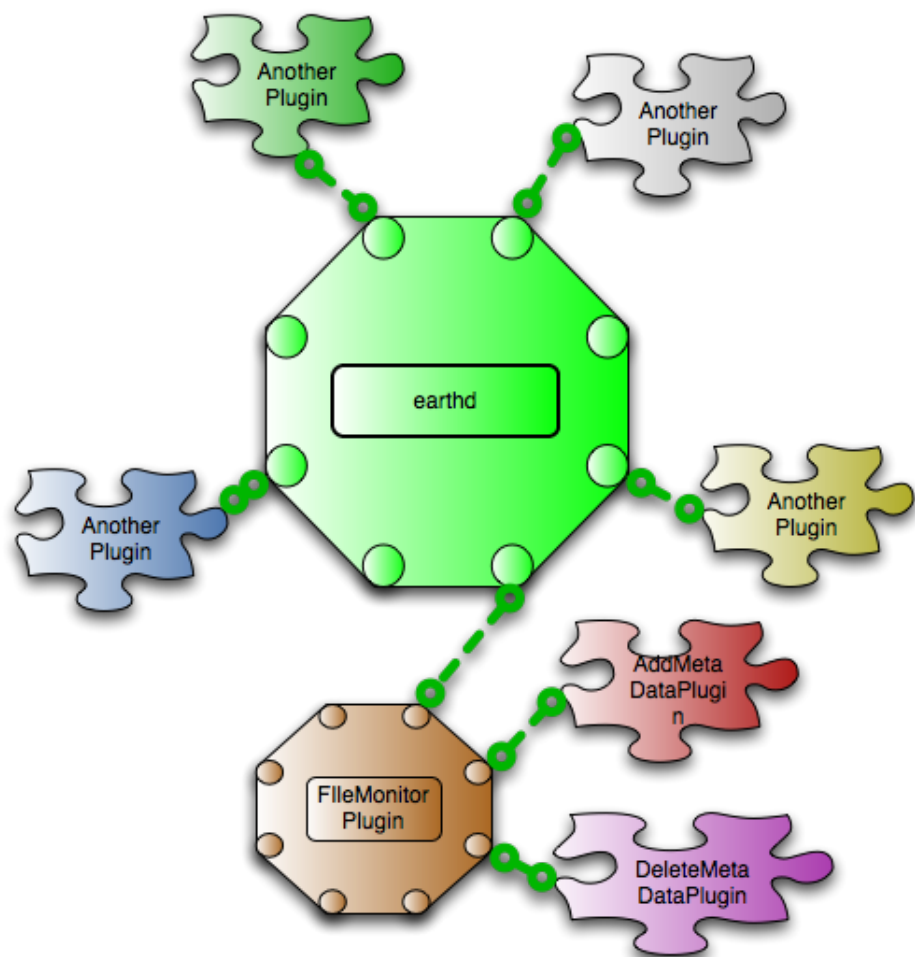


Figure 3: The re-implemented plugin framework.