PILS: A Generalized Plugin and Interface Loading System

Alan Robertson
International Business Machines Corporation
alanr@unix.sh OR alanr@us.ibm.com

Abstract

Many modern Linux application systems make extensive use of dynamically loadable object modules (plugins). However, most of these systems implement their plugin and interface management systems in a way that satisfies their own immediate needs, and is not generally directly usable by other projects.

PILS is an generalized and portable open source Plugin and Interface Loading System. PILS was developed as part of the Open Cluster Framework reference implementation, and is designed to be directly usable by a wide variety of other applications. PILS is available under the terms of the GNU Lesser General Public License (LGPL). Since it is written in C, and built with automake and libtool, it is portable to most modern operating systems. PILS manages both plugins (loadable objects), and the interfaces these plugins implement. PILS is designed to support any number of plugins implementing any number of interfaces.

This paper describes the philosophy and goals of PILS, presents an example of how to use PILS, and discusses a few implementation details of the PILS system.

1 Introduction

Many modern Linux application systems make extensive use of dynamically loaded object modules, oftentimes called plugins.

Plugins can be used for many purposes, and a complex program may use several different types of plugins for different purposes. A program which uses plugins can implement a variety of dynamic capabilities which were not explicitly planned for when the program was compiled. This situation is illustrated by Figure 1. The sample program has communications plugins, authentication plugins and other types of plugins. Such a program can take advantage of new types of communication systems, or authentication systems, etc. without recompiling or relinking the entire system. In some cases, the program can begin using newly-written code without even being restarted.

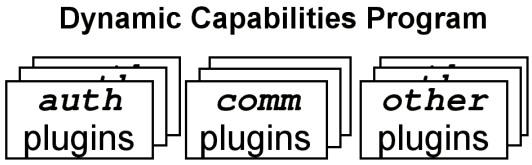

Figure 1: Plugin-enabled Program

This is ideal when one wishes to create a general platform for many different people and organizations to build on. The Open Cluster Framework (OCF) reference implementation is such a system. It is not known how many types of plugins the system may eventually need, nor how many different implementations of each there might eventually be. Plugins are ideal building blocks for such general systems.

On most Linux-like systems, the `dlopen(3)` [dlopen] suite of calls are sufficient to load and unload shared objects (`.so` files) and to find symbols. However, there is much more to managing such plugins than is provided by either `dlopen(3)` or libtool [libtool]. PILS provides the following capabilities which are not provided by either `dlopen(3)` or libtool:

- Determining what capabilities or interfaces are

implemented by a particular shared object

- Determining which plugins provide a particular interface
- Registering exported interfaces
- Importing interfaces for the use of the plugins
- Tracking the reference counts of interfaces

Additionally, the implementation of `dlopen(3)` varies from platform to platform, and is not available at all on some platforms. PILS uses libtool to take hide `dlopen(3)` idiosyncrasies.

PILS was written to provide basic capabilities for the Open Cluster Framework [OCF] reference implementation. OCF is intended to allow proprietary and closed software to coexist in the same framework, with contributions coming from many people, and to support plugins which were not compiled as part of the reference platform. As a result, the ideal model is to drop a suitable plugin into the correct directory, and have it simply work in every respect.

As a result, simple automatic determination of the type of plugin and its capabilities must be supported.

PILS also standardizes certain common functions, such as setting the debugging level, and logging functions through mandantory plugin interfaces. This standardization makes plugins more manageable and flexible than would otherwise be the case. PILS has similar goals to the Glib 2.0 C class loader, but does not require the plugins to use the GTK class hierarchy, and provides some additional features.

2 PILS Model and Terminology

Before presenting more about PILS, it is necessary to define some terminology which is used in this paper. Many of the terms which PILS uses do not have universally accepted meanings. For the purposes of this document the following definitions are assumed:

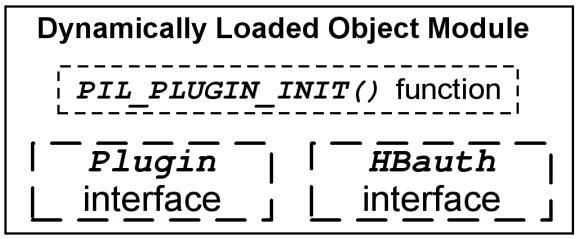

Figure 2: Dynamic Objects and Interfaces

- **Dynamically Loadable Object Module**. A dynamically loadable object module is an independent object file which can be linked at run time into a running program, executed, and then unloaded when desired. On Linux-like systems, dynamically loadable object modules are typically stored as shared object (**.so**) files. The relationship between a shared object file, its interfaces, and its `INIT` function is illustrated in Figure 2.

- **Plugin**. A plugin is a dynamically loadable object module which implements the `Plugin` interface described later. In addition to providing the `Plugin` interface, plugins typically implement one or more other interfaces.

- **Interface**. An interface is the set of exported and imported functions and data items which are shared by all implementations of these interfaces. For example, a communications interface might export functions to read and write packets, and import a function to lock a communications device. The exported functions are defined by a structure with pointers to the various functions (and optionally data items) which the plugin wishes to make public. The imported functions are similarly defined.

  Each interface type defines a unique set of imported and exported functions that are part of the interface which implementations of this interface must meet. PILS defines the **Plugin** interface, and allows others to be defined. PILS supports an arbitrary number of types of interfaces.

- **Exports**. The *exports* of an interface are the set of functions and/or data items which are provided by the plugin for the use of the system loading the plugin. These exported functions

are provided through a single pointer to a structure containing all the individual functions. A typical interface defintion is a C structure consisting of a number of pointers to functions in a structure. Here is a sample interface from the example we will present in detail later.

```
struct HBAuthOps {
 int (*auth)(struct HBauth_info* authinfo
 ,  const char*  data
 ,  char*        result
 ,  int          resultlen);
 int (*needskey) (void);
};
```

In this example, the HBauth authentication exports are defined as a `struct HBAuthOps`. This structure in turn contains two function pointers, the `auth()` function, and the `needskey()` function. All implementations of the `HBauth` interface export this exact set of functions.

- **Imports**. The *imports* of an interface are the set of functions and/or data items which are provided by the loader of a plugin for the use of the plugin. The plugin implementation is then able to use these interfaces to accomplish its purpose. Most plugin loading systems do not provide for importing capabilities into a plugin. The provision of imports to the plugin increases the reusability of plugins in other contexts, and minimizes the use of external symbols by plugins (which is problematic on some platforms). These imported functions are provided through a single pointer to a structure containing all the individual functions, similar to the `HBauth` example in the Exports definition.

- **Type**. The word *type* is used in two closely-related senses in this document. In the most proper sense, *type* refers to the type of an interface. All implementations which share the same interface name are constrained to implement the same interface. This interface name is called the type of the interface, and also the type of the implementation.

  The word type is also used to refer to the *type* of a plugin. Although, technically plugins don't inherently have distinct types, there is a convention that a plugin named *bar* in directory *foo* provides the *bar* implementation of interface type *foo*. This convention is assumed by software which automatically loads plugins in order to load the particular interfaces.

- **Implementation**. An *implementation* of an interface is a particular set of exported functions and data which conform to the definition of the type of interface which it implements. When a plugin is loaded, it registers its interface implementations. Any given plugin can register as many implementations of as many different types as it wishes. Normally, applications provide multiple implementations of an interface, and each is generally contained in a separate plugin. As a shorthand for referring to interfaces (and sometimes plugins), we use a simple pathname convention. The string "HBauth/md5" is a shorthand notation for the *md5* implementation of the *HBauth* interface. This is consistent with the way the implementations are arranged on disk - with all the plugins of a given type being in the same directory.

3 Basic PILS Capabilities

The basic capabilities which PILS provides include the following:

- Loading a plugin
- Managing Reference counts
- Unloading a plugin (by reference count)
- Registration of interface implementations
- Provision of interface imports

4 Loading a PILS plugin

The process of loading a PILS plugin goes through the following steps:

1. **Request** The application requests the loading of a particular interface of a particular type

using the `PILLoadPlugin()` function. Normally an application loads a particular plugin assuming that it provides an interface of the same type as the name of the directory in which it resides. Plugins which provide more than one interface are not fully supported at this time. More about this can be found in the Status and Future work section of this paper.

If, as part of its configuration, the application needs to ask the user which particular implementation of a particular plugin should be loaded, the application can use the `PILListPlugins()` function to return a list of plugins of the given type. If it wishes to validate whether a particular plugin exists, it can use the `PILPluginExists()` call

2. **Load Shared Object** The PILS system then asks the libtool `lt_dlopen()` function to load the shared object into memory. `lt_dlopen()` then uses the native library loading system (commonly `dlopen(3)` to load the object into memory.

3. **Initialize Shared Object** Each plugin has a single initialization function which is then called to initialize the plugin. The name of this function is computed on the basis of its type and its name. This function name is created by the `PIL_PLUGIN_INIT` macro.

4. **Register Plugin** When the plugin's initialization (`PIL_PLUGIN_INIT`) function is called, it is is passed the Imports portion of the Plugin interface as a parameter. This Imports structure includes the following functions:

   - `register_plugin()` A function to call to register oneself as a plugin
   - `register_interface()` A function to call to register an exported interface
   - `log()` The preferred logging function - to be used by all the interfaces in the plugin.

   Once the plugin initialization function is called, the plugin then calls `register_plugin()` to register itself as a plugin. The register_plugin() function is where the exports portion of the Plugin interface is provided to the system. These standard exported Plugin functions include:

   - `pluginversion()` Returns the version of the plugin as a string.
   - `getdebuglevel()` Returns the current plugin debugging level.
   - `setdebuglevel()` Sets the current plugin debugging level to its parameter.
   - `close()` Prepare to be unloaded from memory.

5. **Register Interfaces** After registering itself as a plugin, the plugin calls `register_interface()` to register each interface it implements.

   Each type of interface has its own import and export requirements. Pointers to these structures are exchanged in the `register_interface()` call. When the `register_interface()` call is made, the `InterfaceMgr` managing this interface type then makes the interface available to be called. The generic `InterfaceMgr` does this by adding an entry to a `GHashTable` for that interface type. At this point, all the public interfaces of the plugin are available to be called.

5 Interface Managers

When interfaces are loaded, a plugin of type `InterfaceMgr` is invoked to manage the registered interfaces and make them available to the calling program. PILS provides the capability for each different type of interface to export its capabilities in a unique fashion, because each interface may have different policies and mechanisms for using them and making them accessible to the application. PILS allows these `InterfaceMgr`s to be plugins because they implement a single interface, and managing them as plugins is consistent with the design philosophy of the remainder of PILS. `InterfaceMgr` interfaces are managed much like other interface types, with the exception that the `InterfaceMgr/InterfaceMgr` interface manager is not dynamically loaded, but is linked to a set of built-in functions which are required to load other interface managers.

Any given type of interface can be managed either using a type-specific `InterfaceMgr` (named "`InterfaceMgr/`*type*"), or the generic interface manager. A type-specific interface manager may register the plugin with some other database or registry according to the needs of the application. The

generic `InterfaceMgr` registers all the plugins it manages in a set of `GHashTable`s. One `GHashTable` is maintained for each interface type it manages. Many applications will find that the generic interface manager meets most common needs. This process sounds somewhat tedious but in practice most of this tedium is hidden and it is reasonably easy to use.

6 Sample Plugin

In this section, code for providing a sample plugin are provided and explained. This example code is based on the linux-ha [linux-ha] authentication plugins. In this case, authentication operations are exported as a `struct HBAuthOps`, which defines two exported functions, one for calculating a signature value, and another specifying whether or not the signature method requires a key. This is the same set of exported functions described earlier. In this example, these functions are called `md5_auth_calc()` and `md5_auth_needskey()` respectively.

The first thing to do is set a few #defines which are used by later macros. PIL_PLUGINTYPE defines the interface being implemented, and PIL_PLUGIN and PIL_PLUGIN_S define the name of our implementation.

```
#define PIL_PLUGINTYPE HBauth
#define PIL_PLUGIN      md5
#define PIL_PLUGIN_S    "md5"
/* Our plugin is called "HBauth/md5.so" */
```

Next, declare the set of operations to be exported to the world. In this case, an authentication plugin only needs to export two functions, so declare them, and set up the appropriate structure to point to them, so they can be exported later on.

```
static int
md5_auth_calc(const struct HBauth_info *t
,       const char * text, char * result
,       int resultlen);
static int md5_auth_needskey(void);

/* Authentication plugin operations  */
static struct HBAuthOps md5ops =
{       md5_auth_calc
,       md5_auth_needskey
};
```

Now, define a couple of shutdown functions for managing the unloading of the interface and plugin. These two are provided separately, one or both of them may not have anything to do.

```
/* Shut down the plugin */
static void
md5closepi(PILPlugin* pi)
{
}

/* Shut down the interface */
static PIL_rc
md5closeintf(PILInterface* pi, void* pp)
{
        return PIL_OK;
}
```

The plugin needs to invoke a magic boilerplate macro which provides some common defaults for a number of things that the plugin requires. Next comes declarations about the information to be exchanged when the plugin and interface are registered.

```
PIL_PLUGIN_BOILERPLATE("1.0", Debug, md5closepi);

static const PILPluginImports*  PluginImports;
static PILPlugin*               OurPlugin;
static PILInterface*            OurInterface;
static void*                    OurImports;
static void*                    interfprivate;
```

Next comes the plugin initialization and registration function which gets called when the plugin is loaded. The `PIL_PLUGIN_INIT` macro gives the initialization function a name based on the plugin type and name, to avoid symbol clashes.

```
PIL_rc
PIL_PLUGIN_INIT(PILPlugin* us
,       const PILPluginImports* imports)
{
        /* Save away imports for later */
        PluginImports = imports;
        OurPlugin = us;

        /* Register ourselves as a plugin */
        imports->register_plugin(us
```

```
        ,    &OurPIExports);

        /*  Register an HAauth/md5 interface */
        return imports->register_interface(us
        ,       "HBauth" , "md5"
        ,       &md5ops
        ,       md5closeintf
        ,       &OurInterface
        ,       &OurImports
        ,       interfprivate);
}
```

This is the end of all the PILS-specific code. The real work of the plugin follows. Note the use of the imported `log()` function. This allows the plugin to use the same logging method as the application which loads it uses. The plugin neither knows nor cares how logging is to be done in the particular application in which it has been loaded.

```
/* Real work (should) happen here... */

static int
md5_auth_calc(const struct HBauth_info *t
,       const char * text, char * result
,       int resultlen)
{
        /* UhOh, No Code yet! <8-O */

        OurImports->log(PIL_FATAL
        ,   "UhOh! forgot to write code!");

        /*NOTREACHED*/
        /* Compute md5 authentication  */
        return 0;
}
static int
md5_auth_needskey(void)
{
        /* md5 authentication requires a key */
        return 1;
}
```

7 Plugin Usage Code

Plugin code doesn't need to be aware of which `InterfaceMgr` is managing it, but code that needs to access the loaded functions must be aware of how to interact with the interface manager, in order to be able to find the exported interfaces. For this example, the generic `InterfaceMgr` code is assumed.

First, declare variables to hold a reference to the plugin system, the loaded authentication functions, and some authentication information which the HBauth system needs.

```
/* Sample code ignores errors ;-) */

PILPluginUniv* PluginLoadingSystem = NULL;
GHashTable*    AuthFuncs = NULL;
char                result[64];
struct HBAuthOps*    Auth;

struct HBauth_info   authinfo =
{NULL, "md5", "TopSecretKey!"};

PILGenericIfMgmtRqst RegisterRqsts[]= {
 {"HBauth", &AuthFuncs, NULL, NULL, NULL}
 { NULL, NULL, NULL, NULL, NULL}
};
```

Next, initialize the plugin system, telling it where to look to find plugins.

```
PluginLoadingSystem = NewPILPluginUniv
  ("/usr/lib/heartbeat/plugins");
```

Load the generic plugin manager, telling it (through `RegisterRqsts`) to update `Authfuncs` whenever an HBauth plugin is registered or unregistered.

```
PILLoadPlugin(PluginLoadingSystem
,   "InterfaceMgr", "generic"
,   &RegisterRqsts));
```

At this point, the plugin system is completely ready to go, and plugins can be loaded and unload at will.

```
PILLoadPlugin(PluginLoadingSystem
,   "HBauth", "md5", NULL);
```

Now, the plugin is loaded, and can be accessed. The generic plugin loader stashed a pointer to the interface the `AuthFuncs GHashTable`.

```
/* Get the interface for the "md5" plugin */
Auth = g_hash_table_lookup(AuthFuncs,"md5");

/* Compute signature and put it in 'result' */
Auth->auth(&authinfo, "ImportantStuffToSign"
,      result, sizeof(result));
```

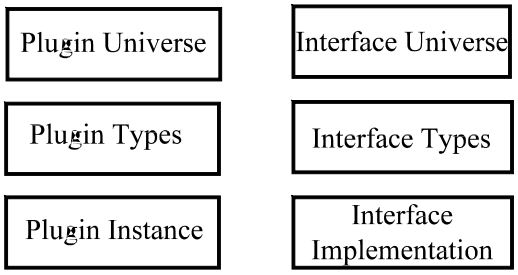

Figure 3: Layers of Abstractions in PILS

When the authentication plugin is no longer needed, decrement the reference count, and the plugin will automatically be unloaded.

```
Auth = NULL;

PILIncrIFRefCount(PluginLoadingSystem
,       "HBauth", "md5", -1);
```

Although the code to prepare the application is somewhat more complex than the example of the plugin itself, most of this code won't be repeated for each plugin or each plugin type.

8 PILS Implementation Overview

8.1 PILS Data Relationships

PILS is written using the Glib [Glib] library, and makes extensive use of `GHashTables`. There are basically two related abstraction stacks which PILS maintains: the Plugin Universe, and the Interface Universe. These are, in effect, parallel representations of related information, or for the more pun-minded, parallel universes. This is illustrated by Figure 3.

Each universe consists of a set of types, and each type contains a set of instances of the fundamental object (a Plugin or an Interface). Most of the work is keeping the relationships between the two layers and these two universes synchronized, so that it is known what interfaces were instantiated from any given plugin, and which plugin any particular interface was loaded from. There are a number of reference counts, and more complexity than one might expect.

8.2 InterfaceMgr: Managing Interfaces

From the perspective of PILS, the most interesting part of the world consists of interfaces. `InterfaceMgr` is PILS' name for the type of interface which is presented by plugins which manage interfaces.

Interfaces are where the variation and interesting behaviors are generally implemented. So, PILS implements an interface for the management of interfaces. This interface management function of PILS is believed to be unique. In PILS, the interface which manages interfaces is called the `InterfaceMgr` interface. So the `InterfaceMgr` interface which manages other the `InterfaceMgr` interfaces is the `InterfaceMgr/InterfaceMgr` implementation. The `InterfaceMgr/InterfaceMgr` is built-in (not dynamically loaded) since it is necessary for bootstrapping the dynamic loading management system. It loads and manages the other interface managers (including the generic interface manager).

Normally, the name of an interface manager is the same as the name of the type of interface it manages. Not so for the generic interface manager, which can manage any number of types of interfaces. When it is loaded, it is passed a parameter to tell it which types of interfaces it should manage. Since it can register any number of implementations, it then registers itself as the manager for each of these interface types it was passed when it started up.

9 Security Considerations

There are a few additional security considerations associated with plugin-enabled programs. Programs which use plugins to provide capabilities have more files and directories which need to be properly secured in order to ensure the application is not compromised. All programs have files and

directories which must be properly secured in order for them to be secure, but software which uses plugins typically have a few more such directories and files. In addition to the location of the binary, and the normal libraries, the location of the plugins and the plugins themselves (and there may be many of them) must also be properly secured.

With an improperly secured system, and plugins which meet well-known interfaces, it is a very simple matter to create a plugin which meets the well-known interface, but which opens a wide security hole which can go completely undetected for a long period of time.

Plugins which provide security functions and which provide extremely simple interfaces (like the authentication example presented earlier) make extraordinarily tempting targets for intruders. It is prudent to assume that attackers *will* exploit such interfaces if they are improperly secured.

Plugins run in the address space of the loading program, so they can easily do any thing which the program itself has permissions to do. There is a difficult issue of trust associated with a collection of plugins which come from different sources.

It is necessary for software which uses plugins (whether from PILS or some other source) to ensure that they install their software in properly secured locations. Although some of the security enhancements described later will help this problem somewhat, the need to properly install and administer systems is still fundamental.

10 Status and Future Work

The current implementation of PILS is functional, and is currently used by the Linux-HA project and part of the OCF reference framework. It is currently generally available as part of the Linux-HA distribution [ha-dist], but is not currently available as a separate subpackage (but this will probably have occured by the publication of this paper). The source to PILS can also be directly viewed in CVS [ha-cvs].

Although PILS is a powerful tool providing a rich set of capabilities, the area of plugin management is broad and quite interesting, and PILS is in the early stages of its evolution. As a result, there are a number of needs which PILS does not fully satisfy. Since it is an open source project, all interested parties are invited to contribute these or other enhancements. The following is a list of features which are under consideration for the inclusion in future versions.

- **Aliases.** Add an alias capability. Although each plugin can provide more than one interface, the current implementation of the "tell me all the plugins which implement interface X", assumes that each plugin actually only implements its main interface. To remedy this limitation, it is desirable to add an alias capability.

  On many Linux filesystems, symbolic links could be used, but it is believed that even symbolic links would require some additional implementation effort and then they would be limited to being stored on filesystems which implement symbolic links.

- **PATH support.** Allow PATH-like searches for the location of plugins.

- **Porting.** Complete and verify the ports to other operating systems.

- **Default InterfaceMgr.** Add the ability for PILS to set an automatic default `InterfaceMgr`, rather than expecting the application to declare in advance which plugins they wish the current generic manager to manage.

- **InterfaceMgr management.** Extend the InterfaceMgr paradigm to add a new function to ask a particular InterfaceMgr to manage a particular Interface type.

- **InterfaceMgr interface** Add the ability to add new interfaces to manage after an interface manager is loaded. This is mainly for the generic interface manager, but may also be necessary for plugins whose interfaces have become inaccessable but whose plugin is still loaded. Note that this may overlap or interact with the previous item.

- **Security awareness.** Enhance PILS security awareness. For example, verify who owns plugins, plugin directories, and so forth.
- **Signed plugins.** Add support for cryptographic signatures for plugins.
- **Plugin licenses.** Add the license and licenseURL functions as standard member functions for plugins.
- **Independence.** If sufficient interest is shown, it would be good to make PILS a completely independent open source project. In any case, PILS needs to be a completely independent package which can be installed without any of the rest of the OCF software.
- **Gtype support.** Extend PILS to load GTK C types. If this were reasonable, then it might be a better solution than the C class loader which is scheduled to be released with the 2.0 version of Glib.
- **C++ class support.** Support loading of C++ classes.
- **Non-native languages.** Generalize the idea of plugins in such a way that PILS could also load plugins written in arbitrary languages like Perl, Python, or Java.
- **Interface version management.** There is currently no built-in capability or convention for managing interface versions (including the plugin interface).

11 Acknowledgments

Special thanks go to Neal McBurnett who contributed both to the early design stages of PILS, and also to the original design stages of the HBauth plugin which is used in the examples. Thanks also go out to Cliff White, Ramachandra Pai and Xiaoxiang Liu who spent time reviewing and critiquing the paper. The author also wishes to thank the developers of the Glib library, who have created an extraordinarily useful and functional C library. PILS development would have been much more difficult without Glib. The author notes that PILS is not a commercial product and this paper represents the views of the author, and does not necessarily represent the views of his employer.

12 Trademarks

References

[ha-cvs] *PILS CVS Repository*, Robertson, et al, `http://cvs.linux-ha.org/viewcvs/viewcvs.cgi/linux-ha/lib/pils/`, `http://cvs.linux-ha.org/viewcvs/viewcvs.cgi/linux-ha/include/pils/`

[dlopen] *Linux dlopen(3) manual page*, Linux community. `http://www.freebsd.org/cgi/man.cgi?query=dlopen&apropos=0&sektion=0&format=html&manpath=SuSE+Linux%2Fi386+7.3`

[Glib] *Glib Reference Manual*, Gnome Project. `http://developer.gnome.org/doc/API/glib/index.html`

[ha-dist] *High-Availability Linux Distribution*, Robertson, et al, High-Availability Linux Project. `http://linux-ha.org/download/`

[libtool] *Libtool Reference Manual*, Free Software Foundation. `http://www.gnu.org/software/libtool/manual.html`

[linux-ha] *High-Availability Linux Home Page*, Robertson, et al, High-Availability Linux Project. `http://linux-ha.org/`

[OCF] *Open Cluster Framework Project Home Page*, Robertson, et al, `http://opencf.org/`