Report Number: WUCS-00-04

2000-01-01

Plugin Management for Active Network

Sumi Y. Choi

The purpose of this document is to present the overview of tte plugin management architecture and the description of the software developed for the scalable, high performance active network node project in Washington University, St. Louis. The plugin management is a user space daemon program that runs at the code(plugin) server and at the active network component of a router or a switch port processor. The running programs cooperate to load plugins from the code server to the active network component. This software is intended to be used among multiple platforms.

Plugin Management for Active Network

Sumi Y. Choi

WUCS-00-04

April 2000

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis MO 63130

Plugin Management for Active Network

Sumi Y. Choi
syc1@arl.wustl.edu

Feb. 17, 2000

Abstract

The purpose of this document is to present the overview of the plugin management architecture and the description of the software developed for the scalable, high performance active network node project in Washington University, St. Louis. The plugin management is a user space daemon program that runs at the code(plugin) server and at the active network component of a router or a switch port processor. The running programs cooperate to load plugins from the code server to the active network component. This software is intended to be used among multiple platforms.

1 Introduction

The core idea of the active network stems from the need for supporting new types of services, and protocols. The current network model inherently restricts these features because of the homogeneity implied in it.

Our active network platform has been developed so that one can achieve deployment of new protocols or the application-specific services efficiently. The focus of the platform is on the concept of *network plugin* or *plugin* and the processing component called *active element.* The plugin is binary code that can be recognized, dynamically downloaded and run on a datapath. The active element maintains the plugins and assigns each datagram with the proper plugin. Here, each datagram contains the identifier of the plugin to be executed on it. The active processing element finds the plugin using the identifier and then assigns the datagram to the plugin.

In case when a plugin references an unknown plugin, the active element is enable to cause a remote downloading of the plugin. The scheme used for the remote downloading is *distributed code cashing.*

In the distributed code cashing scheme, we view of the active network as a continuously changing entity, where various service vendors produce or update their services by distributing plugins. The plugins are stored in the entities called *code server.*

For the efficiency of the network datagram processing, the plugin downloading process is isolated from datagram process in the active element. The plugin management is the software which is responsible for the plugin downloading process and is located either in the user space of the active element or that of the code server in our current platform as shown in the Figure 1. This document is focused on the structure and the usage of the plugin management on active element and also on code server. We listed the terms used in this document in section 2 for a quick view. In section 3, we focus on the details of the plugin management in general by covering all the components. We discuss the code server and the key server in section 4 and 5, followed by the data path review in section 6. In section 7, we presents the usage of the softwares. Finally, we conclude with section 8.

2 Terms

The terms used in this document are listed here for future reference.

1. Plugin Management: userspace software for the plugin management
2. Active Plugin Loader: interface of the plugin management to the active element
3. Plugin Database Controller: plugin database and its controller in the plugin management
4. Plugin Requestor: plugin request protocol module in the plugin management
5. Security Gateway: authentication module in the plugin management
6. Policy Controller: policy verification module in the plugin management
7. Code Server: code server
8. Key Server: public key server for authentication
9. Active Element: active element.
   The active element is broader concept of EE, because it specifies the hardware structure as well as software. In this document, the active element can be considered as EE.

3 Plugin Management, *pmd*

The plugin management is the software component of the active network that is responsible for the remote download, the authentication, the policy verification and the store procedure of plugins. It has five component, each of which is assigned with one of the procedures.
As shown in the Figure 1, it interfaces with the external world, including the active element, the code server, and the active network administrator. The communication with the remote code server which is shown above the plugin management in Figure 1, will be managed by the plugin requester. All the other interfaces will be defined in the active plugin loader, which makes it the only system-dependent module in the plugin management. In this section, we will discuss the details of each components starting with the active plugin loader.

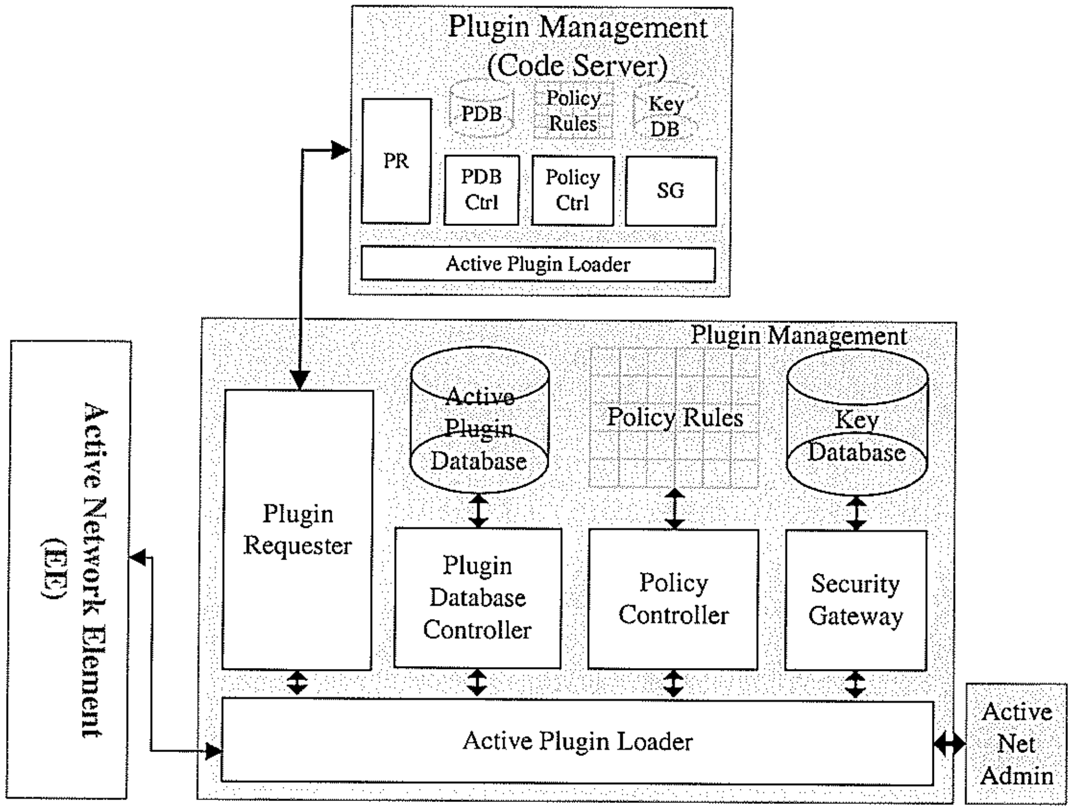


Figure 1: Plugin Management

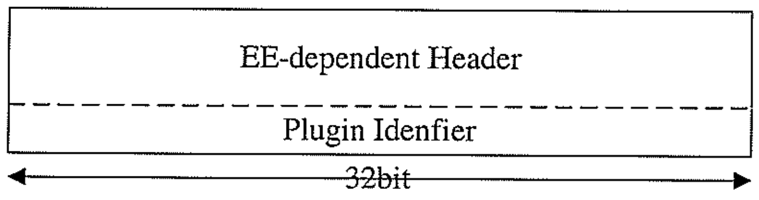


Figure 2: Plugin Request(from active element) Format

3.1 Active Plugin Loader

Active Plugin Loader is the central module of the plugin management with the interface to the active element, who requests for unknown plugins. So, the active plugin loader is the starting point of the plugin downloading process within the plugin management, and it organizes each steps of the data path. For the interface with the active element in our platform, we use the socket interface because the plugin management is built on the userspace of the active element. However, the interface is dependent on the active network platform and the active element or EE in the platform. Therefore, it should be configured by the active element or EE developer. The plugin request from the active element has a simple form which is composed of the active element specific header and the plugin id. The format is shown in Figure 2.

Besides the interface to the active element, the active plugin loader also has the interface to the plugin management administrator. The plugin management administrator is another software module that one can use to configure the plugin management from command line, such as the code server list, the policies for plugins, or the security constraints. It can be remotely located from the plugin management because it uses the remote procedure call(RPC) to run the interface routines in the active plugin loader.

3.2 Plugin Database Controller

The plugin database controller manages the database of plugins in the plugin management. Usually this database is used as a cache space for the plugins downloaded from code servers. After loaded into the active element, the plugin is also stored(cached) in the plugin database in the plugin management. Because we can have multiple active elements sharing a single plugin management in our future model of the active network component, the plugin management is likely to receive multiple requests for a single plugin. The cached plugin can be supplied for the subsequent request, once the plugin is loaded. Meanwhile, a loaded plugin can also get unloaded from the active element by the resource management in it. The datagrams coming with the identifier of the plugin after this point, will cause the active element to request the plugin again. This can also be a case for cache downloading.

This local cache downloading is a shortcut for the plugin downloading. However, there are times one might want to force plugin downloading from code servers. Upgrading plugins can be

Field	Size
Plugin ID	32 bit
EE ID	16 bit
Developer ID	32 bit
Time Stamp	32 bit
Plugin Size	32 bit
Codeserver Signature	64 bytes
Developer Signature	64 bytes
Plugin File Location	Fixed in code

Figure 3: Plugin Database

one reason. In that case, the plugin cache line in the plugin database of the plugin management has to be flushed. Another usage of the plugin database is to make it a code server's storage. The detail will be given in the code server section.

The plugin database is implemented with the plugin database controller and the database storage. The plugin database controller contains the methods for maintaining the plugin database. The plugin database is a storage space in the file system, , and it contains a list of plugins classified by the plugin identification(pid) number and the execution environment identification number(eid), or the active element identification number. In addition to these information, there are a few other fields that are stored with each plugin in the plugin database, such as the location of the plugin object code in the file system, the signature of the plugin given by the developer of the plugin or the code server, etc. (The details about the signature will be given in section 3.4.) The format of the plugin database is shown in Figure 3.

3.3 Policy Controller

Policy Controller is responsible for applying the plugin management policies on the plugin downloading process. To protect the active element from possible hazard caused by plugins downloaded from remote sites, the plugin management maintains a list of policy rules. Each policy rules assigns `ACCEPT/DENY` policy on a group of plugins, specified with three fields, i.e. plugin identification number, developer identification number, and code server IP address. For example, one might always want to accept the plugins from a specific code server, say, *codeserver.com* that is poplar or block them from a specific developer, who has bad reputation(Assume this developer has id of 2). The first two lines of the Figure 5 shows the policy rules for these cases.

The policy rules are maintained in a database called *policy database*, that has the three field as in Figure 4. The policy rules in the database are listed in the order as added by the plugin management administrator or provided by the configuration file. For each plugin downloaded into the plugin management, the policy controller checks the list of the policy rules, one by one,

Field Name	Size
Plugin Identification Number	32 bit
Developer Identification Number	32 bit
Code Server IP address	32 bit

Figure 4: Policy Database

Plugin Id Number	Developer Id Number	Code Server Address	ACCEPT/DENY bit
*	0x00000002	*	0
	*	*codeserver.com*	1
0*x*80010005	*	*	0

Figure 5: Example Policy Database

whether they match the specification of the plugin. If the policy controller finds a matching rule, it applies the rule by returning the policy field.
In Figure 5, we show an example of the policy database.

According to the `ACCEPT/DENY` bit, the active plugin loader either abort the plugin downloading(`DENY`) or load it into the active element(`ACCEPT`).
Now, the policy discussed Φup to this point is the import policy, which is applied to the plugins coming into the plugin management. There is another type of policy that the policy controller manages, which is the export policy. The export policy is used by the plugin management in a code server to control the exporting of the plugin to the plugin management in active elements. The detail will be given in the code server section.

3.4 Security Gateway

In our model of the active processing, once plugins are accepted in the policy controller, we trust the origin(code server) and the vendor(developer) of the plugins and the plugin s are safe enough to run in the active element. Therefore, we assume that those code servers and the developers are well known to the public, and that they do not maintain malicious or deficient plugins. Maintaining this good quality as a code server or as a developer is a separate issue to be considered and out of the scope of this document.
However, there still is a need to secure and maintain the integrity of the plugins through the transmission between the plugin managements. For this purpose, we included the module called the security gateway in the plugin management. The security gateway is responsible for authenticating the origin of the plugin transmission and the vendor of the plugin. In our model, the origin is the code server that sends the plugin and the vendor is the developer that publishes the plugin. To perform this task, we use the signature. The plugins are transmitted with the signature of the code server or the developer of the plugin. We assume that the plugins are initially stored in the code server with the developer's signature. The signature of

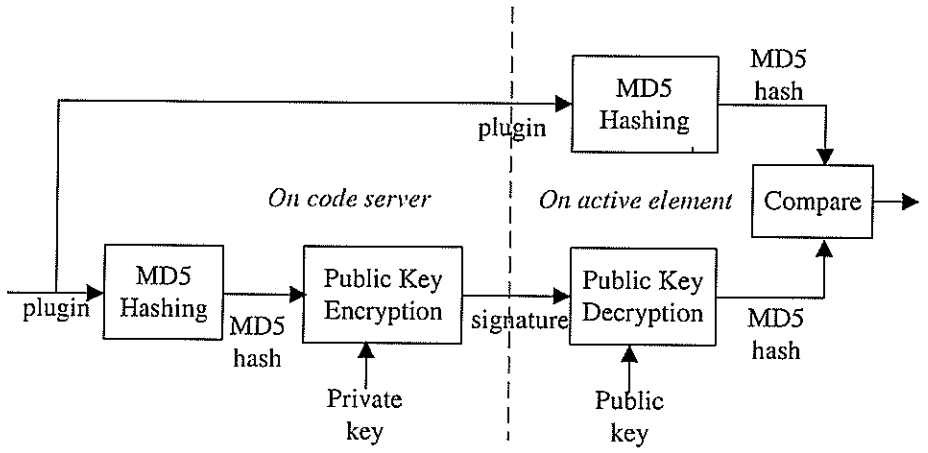


Figure 6: Security Gateway Processing

the code server can be generated at the time of loading or can be pre-computed. The plugin management should be able to authenticate the plugin using the signature and the identities of the code server and the developer.

In a full path, an active element's plugin management requests a code server for a plugin, the code server transmits back with the plugin, the developer's signature, and its signature on the plugin. The plugin management on the active element authenticate the plugin based on the signatures, the code server address, and the developer id. To vary the weight of focus on the authentication complexity and the security, one can have options to have no authentication, code server authentication, or maximum authentication.

There are several ways to do the authentication, well known as cryptographic standards. In our implementation, we use the RSA encryption and the MD5 hashing scheme. The MD5 is used for generating a 64 byte plugin specific hash, which is the result of digesting the plugin and then the RSA encryption is used for the encryption of the hash. The encrypted hash is the signature. For the RSA encryption, we maintain a key pair of public key and private key per each entity, such as code server or developer.

As implied, the authentication is done with the cooperation of the sender and the receiver, or the plugin managements on both ends. The Figure 6 shows the example of the authentication between the two plugin managements each on code server and the active element. The left side of the Figure 6 shows the steps for generating the signature on the sender side. The actual authentication happens on the right side, which is the receiver. The receiver goes through two separate stages, one for the decryption of the received signature to get the MD5 hash, and the other for regenerating the MD5 hash from the received plugin. At the end, it compares two MD5 hashes, and only when they are the same, the authentication is successful.

For the plugin management on a active element to be able to authenticate different entities related to the plugins, it needs the public keys of the entities besides the plugin and the signatures transmitted from a code server. According to the RSA encryption, the private key

is securely maintained by the owner of the private and public key pair, while the public key should be known to the other ends.
Therefore, a key distributing service is needed for the plugin management to obtain the keys. There considered a couple of ways to distribute the public keys. One is to receive them from the owners. The other is to maintain a key server for the public key distribution service. Because there are an issue of identifying the owner of a public key, particularly when the owner is a developer, we decided to follow the second option. The other thing about the second option is that we could make use of the existing implementation. The DNS security extension provides the public key maintenance, and it is implemented in BIND ver 8.2. In the DNS security extension, they have a field for the cryptographic key for well known algorithms including RSA, and their resolver library support the query of a public key just as the domain name query.

3.5 Plugin Requester

For the transmission of the plugin request and the plugin response, we have the plugin request protocol. It defines the formats for the plugin request and the plugin response, and is implemented upon TCP/IP. The packet formats are shown in the Figure 8. Depending on the length of the plugin, the response could be fragmented. The whole protocol is built upon the socket layer over TCP/IP.
The plugin requester is the module that runs the plugin request protocol. To support the protocol, the plugin requester is required to have a method to locate code servers. Once it knows where to contact, it uses the plugin request protocol to send request and to download a plugin. We came up with several methods for locating code servers.
One option is to use unicast. To do this, the plugin requester simply maintains a list of code server addresses and it contacts the code servers on the list, one by one, in the order of the priorities assigned to them, starting with the most preferred one. The plugin requester sends out requests to a code server every *re-request interval* seconds until it receives a response. It gives up with the code server after *request-cancel timeout period* seconds, and try the next code server. This two parameters are configured from the configuration file at the time of initial setup.(See section 7.2) This option is desirable for the local network where the network resources are mostly stable, and the types of plugin services are restricted and expected to be found in one of the stable code servers.
Another option is to use multicast. The plugin requester maintains a list of multicast addresses instead of unicast addresses. The works the same way as unicast from the point of the plugin requester. However, it has an effect of sending a single request to the group of code servers. This scheme requires a global notion of public multicast groups for code servers. It implies the development of a distributed storage with code servers, which is out of the scope of this document. One of the drawbacks of the multicast solution is that it may waste network load by causing duplicate responses from code servers, because once the request is multicasted, there

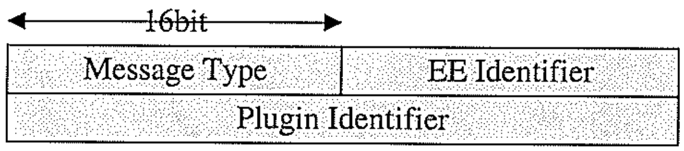
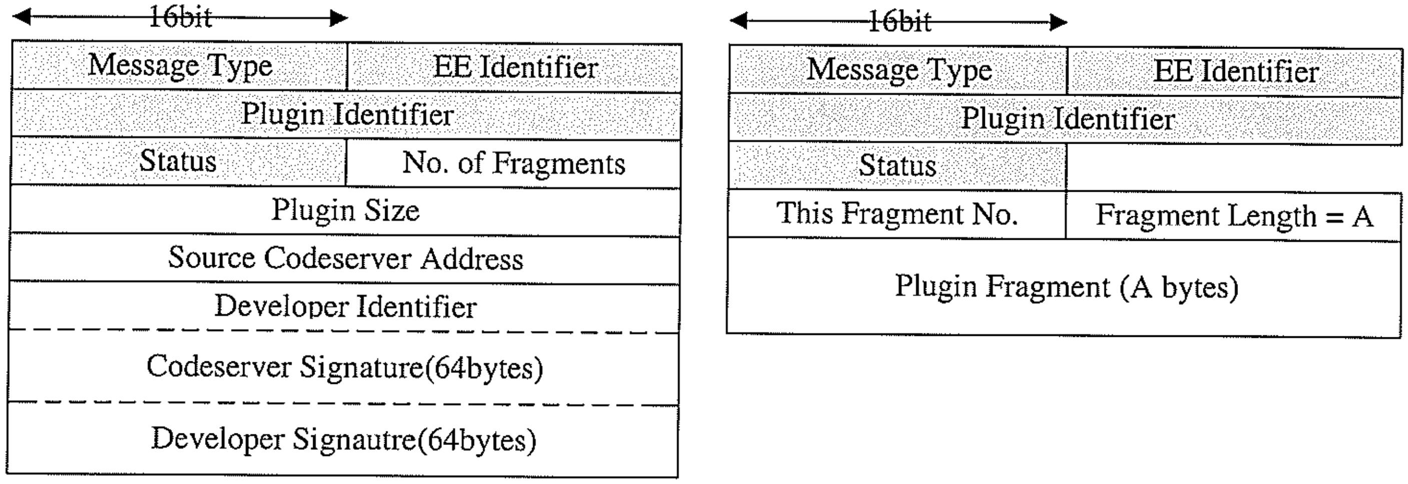


Figure 7: Plugin Request Format(From PR(active) to PR(code server))



(a) First Plugin Packet

(b) Subsequent Plugin Packet

Figure 8: Plugin Response Format

is no control for optimizing the number of responses.
Last option is to use anycast. An anycast address represents a group of destinations as a multicast address does. However, unlike the multicast, it does the selective addressing. In other words, if a datagram is sent out addressed to an anycast address, the network figures out the *best* one out of the group that the anycast address represents. Then, the datagram is sent to the one picked by the network. Here, we eliminate the disadvantage of high network load caused by multicast. Anycast is a convenient mechanism for the service discovery in general. However, it has not been detailed for implementation.
The multicast solution and the anycast solution are better for the network where hierarchy or configuration of the code storages are dynamically changing. The combination of any of the three solutions are also possible. Currently, we only implements unicast, where the addresses for code servers are fixed at the time of configuration or by the administrator.

The Figure 8 shows the plugin request format and the plugin response format that are used for the plugin request protocol. The plugin request format here is different than the one used internally between the active element core and the active element plugin management. This plugin request is used between plugin managements for code server downloading.

4 Code Server

As briefly introduced in the section 1, the code server is the entity that maintains a database of plugins, and it responses to plugin requests that comes from the remote active elements. The plugins in the database are specified with the plugin identification number and the execution environment number. Therefore, various active element(execution environment) can share a single code storage.

In our implementation, we recycle the plugin management software to use it for the code server. Because of the ability to store plugins, to provide additional safety features, and to support the plugin request protocol, the plugin management can be used either for the active element and for the code server with some tune-up parameters, which is specified in the configuration file.

The key difference of the plugin management for the code server as to that for the active element is in the interfaces to the external world. While the plugin management for the active element has the interface to the active element, the plugin management for the code server is the code server itself. So, it does not have any other interfaces other than the communication channel through the plugin request protocol.

Another difference is in the security gateway. The code server itself is one of the entity who can have its own key pair. If the security option says it has to maintain code server authentication, the key pair should be generated, and in addition, the public key has to be registered with the key server. The key pair is also read locally by the plugin management software when the code server gets executed. This information should also be appeared on the configuration file.

From an abstract view, we consider a group of code servers as a distributed database. The code servers can form a tree structure, or nested structure so that each code server can also download plugins from each other. Here, the code server list maintained in the plugin requester is used for the next level code server connection. Therefore, the plugin request can be forwarded to the next level, if it cannot be satisfied at the current level. Now, the plugin request and plugin reply can go through several code servers causing them to forward back to the active element who originally requested the plugin.

It is still not clear how to find a best hierarchy of the code servers and how to distribute the plugins throughout the hierarchy. This is out of the scope of this document, and also the more crucial issue at the current moment is to experiment with the shared plugin storage on a network of the heterogeneous network platforms.

5 Key Server

The key server is the entity that maintains a database of public keys for the security gateway, and distributes the keys as requested. As mentioned, we provides three options for authentication, either, *No authentication*, *code server authentication*, or *developer authentication*. Either the second or third case, we use the RSA scheme which requires public key distribution.

The DNS security extension is used to provide the features of the key server. It has the capability to store well known public keys, including the RSA public keys.
In the normal DNS server, the necessary fields per a name is the IP address(type A). In our key server, we add the key field(type KEY) in addition to the IP address. Unless the key server is also used as a real DNS server, a separate key server domain is required for the plugin management.
As mentioned, we have a couple of entities for our key server, code server and developer.
Adding a record to the key server can be slightly different for the code server and the developer.
If the entity is a code server, the record for this code server already exists in the local DNS server. We take the same name without the subnet domain name and the IP address field and the public key field have to be created to complete the record for this entity. (The tool for generating the RSA public key is included in our package. See section 7.3)
If the entity is a developer and it's record is already in some DNS server, then the procedure for the code server is applied here, too.
If the entity is a developer and it is a new entity to be introduced, all three items, the name, the IP address, and the public key, have to be created.
The IP address for a name in the key server is not to be used for the domain name lookup and should comply with the subnet where the key server is located.
In our plugin management version, we include BIND ver 8.2 to support the key query and the key generation. For generating the RSA keys and the signatures, we support the command line tools, located in `bin` directory, each named `pmkeygen` and `pmsiggen`.
Although we included all the necessary tools in our package, we recommend you download the DNS package from `http://www.isc.org/products/BIND/`.

6 Overview of the data path

6.0.1 Plugin Request

The Figure 9 shows the data path that the plugin request is received from the active element and sent out to the code server. The interface with the active element is open, and can be defined as needed. The default case the socket interface.

6.0.2 Plugin Response

The Figure 10 shows the data path that the plugin response is received and processed. It touches through all the modules in the plugin management and finally ends at the active element, where the plugin request is issued.

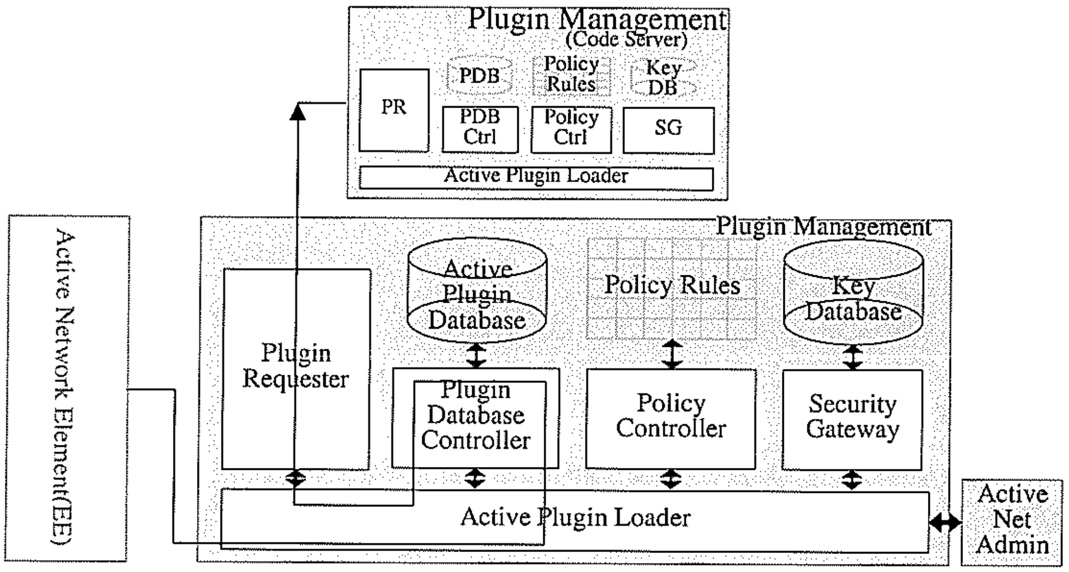


Figure 9: Plugin Request

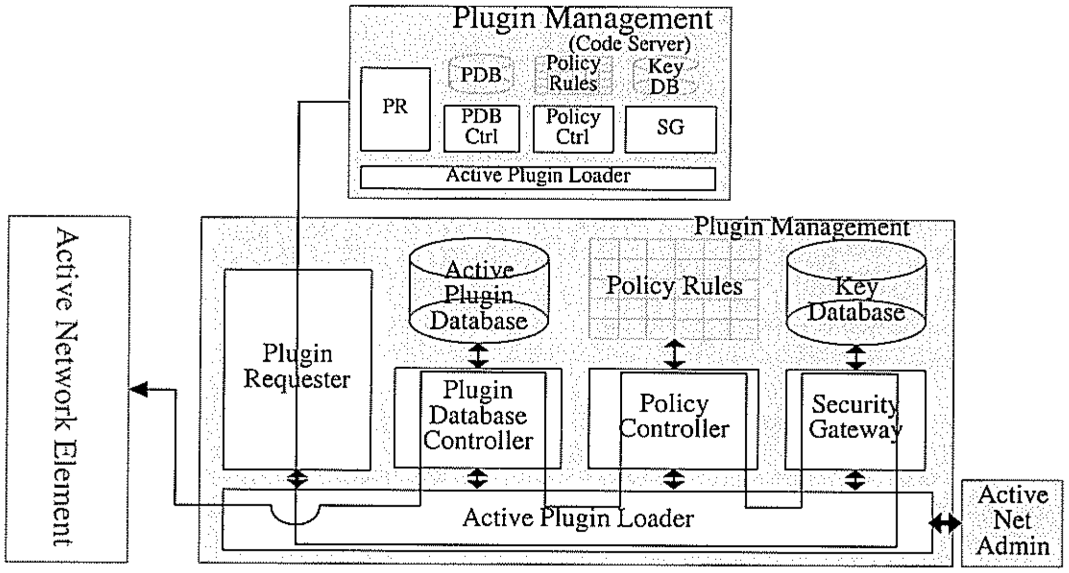


Figure 10: Plugin Response

7 Manual

7.1 Compilation

From the root directory of the plugin management(untared path), do the following.

```
make clear
make depend
make install
```

This stores the binary files in the `bin` directory. Note that only `Linux` and `NetBSD` are supported at this moment.

7.2 Configuration

The slight different configurations are applied to the plugin management depending on whether it runs for the active element or for the code server. A configuration file starts with comments, which are the lines with prefix '#'. The comments are in italic fonts in the figure. A sample configuration file is shown Figure 11. A valid configuration line starts with one of more keyword, which is in bold in the figure. The syntax for each line is given in the commented lines. The configuration files including this one can be found in `demo` directory. In this document, we only describe the semantics of each line. (The following notation are used `pid`=plugin id, `eid`=ee id, `did`=developer id)

- `path [local path]`
  sets the local directory for pmd
- `keyserver [name]`
  sets the key server
- `keydomain [domain name]`
  sets the domain name used by key server
- `codeserver [name]`
  sets the code server
- `security [security code]`
  sets the security option used by security gateway
- `policy [pid] [eid] [did] [policy(0/1]`
  sets a policy, entered in order of appearance
- `plugin import [pid] [eid]`
  load a plugin into plugin cache
- `plugin export [pid] [eid]`
  load a plugin and send it through active element interface
- `plugin import [pid] [eid] [did] [plugin file] <[developer signature file]>`
  load from local file

```
#
# Configuration file for pmd EE/Active Element
#
# 1. Path for db
# Syntax : path [local path for pm]
path ann-gantry

# 2. Keyserver
# Syntax : keyserver [key server name]
# keyserver gussie.arl.wustl.edu

# 3. Domain of the key server
# Syntax : keydomain [domain used for the key server]
# keydomain ann.arl.wustl.edu

# 4. PM's public key
# Syntax : cownkey [key name] [key id] [path]
# cownkey _KEYNAME_. 38144 /c/syc/pm/demo/keys

# 5. Code server
# Syntax : cserver [code server name] [priority]
cserver nmvc1.arl.wustl.edu 1

# 6. EE id
# Syntax : ee [ee id]
ee 0x0001

# 7. Security Gateway Option
# Syntax :
#      security no (no security check)
#      security developer (check developer signature)
#      security max (code server + developer signature)
#      security codeserver (check code server signature)
# Default : security codeserver
security no
```

```
#
# Continued
#
# 8. Policy
# Syntax : policy [pid] [did] [code server] [allow(1)/deny(0)]
policy 0x80010001 * * 1
policy * * clouseau.arl.wustl.edu 1

# 9. Default plugin to load
# Syntax : plugin import [pid] [eid]
#          plugin export [pid] [eid]
# plugin export 0x80010001 0x0001
# plugin import 0x80020001 0x0001

# 10. Inter-request interval for plugin. \
# The requests are sent out every [this] seconds
# Syntax : plugin_nextreq [interval]
# nrequest 5

# 11. Timeout period for plugin request
# Syntax : plugin_cancelreq [timeout period]
# crequest 20

# 12. Set code server key
# Syntax : ckey [code server name]
# ckey nmvc1.arl.wustl.edu

# 13. Set developer key
# Syntax : dkey [developer name]
# dkey dev1.arl.wustl.edu
```

Figure 11: Sample Configuration File

- `plugin export [pid] [eid] [did] [plugin file] <[developer signature file]>`
  load from local file
- `nrequest [interval]`
  sets the inter-request period
- `crequest [timeout period]`
  sets the timeout period for plugin request

In the sample files, only mandatory lines are remained and all the others are commented out. These lines are the minimum requirements.

7.3 Running Plugin Management

In the `bin` directory, there are binaries for the plugin management, signature generator, key generator, and plugin administrator

- Plugin Management on active element
  `pmd -a -f <configuration file>`
- Plugin Management on code server
  `pmd -c -f <configuration file>`
- Key Generator
  `pmkeygen <key name>`
  This generates a 512 RSA key pair.
  The public key is stored in the file named `K<key name>.+001+<key id>.key`, which can be directly inserted as another field for DNS database. The private key is stored in the file name `K<key name>.+001+<key id>.private`.
- Signature Generator
  `pmsiggen -n <key name> -i <key id> -p <plugin filename>`
  This generates a 64 bytes signature for the plugin file. The key pairs should be in the same directory. The signature file is named `<plugin filename>.sig`.
- Sample EE interface
  `pmeereq <pid>`
  This sends a plugin request to the plugin management. (Only applicable for `netBSD`.

7.4 Modifying EE Interface

The ee interface is defined in `pmsrc/pmd/pm_eeinterface.c`. Currently, the system type of DAN is defined for our active network interface. Additional modifications should go to the correct system type slot. The reference source files are as follows::

```
pm_eeinterface.h :  interface header file
pm_eeinterface.c :  interface source file
```

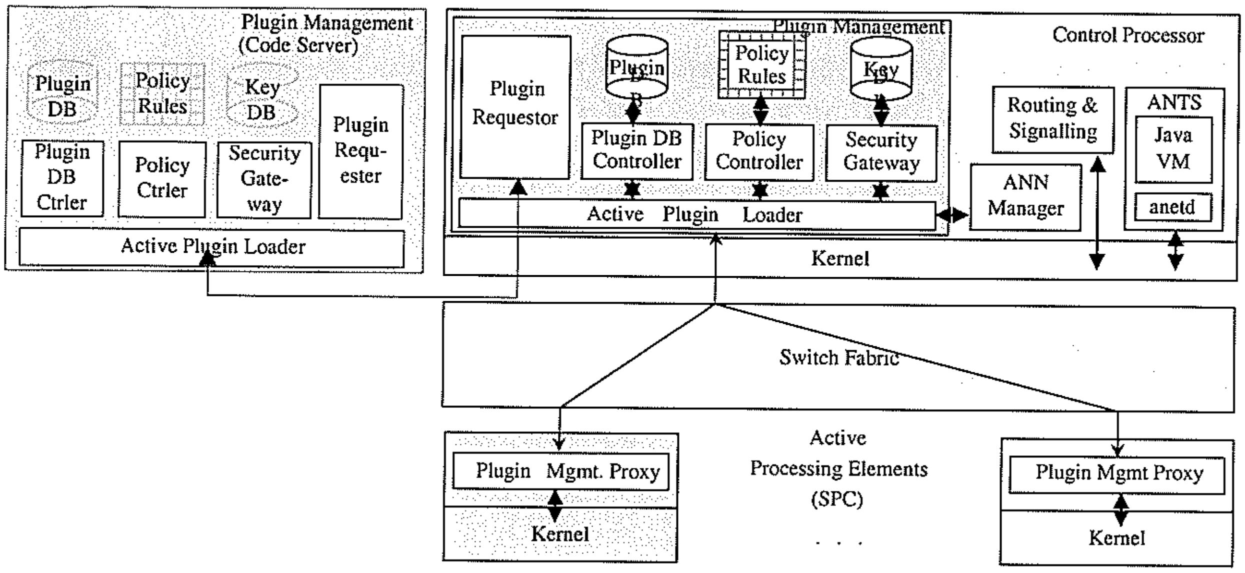


Figure 12: Plugin Management Proxy and the Plugin Management

8 Summary

The plugin management is the user space module that support the remote code loading process on the active network platform. For the active network environment with different platforms, the plugin management can also be used as a tool for managing a shared code storage.
To support these features, we developed the plugin request protocol module, the authentication module, the policy control module, and the plugin database module as part of the plugin managment.
In our testbed, we use kernel interface that directly load plugins into the kernel of the active element operating system. As a next step, we are planning to have the plugin management in the control processor of a network switch, and the active element on the port cards of the switch. Because we will have multiple active elements(for a multiport switch), each active element will have a small module that communicates with the plugin management. This module, which is called plugin managment proxy, will issue plugin requests through the active plugin loader in the plugin management. The new model is shown in Figure 12. We expect to obtain better performance by detaching the plugin management process from the critical path of the packet process. At the same time, active elements can achieve the distributed effect with cost-effectiveness by sharing the plugin management. Meanwhile, we expect the plugin management to be used in various environments, that require the remote code loading feature.