Using Extension Points

You can write extensions to affect how Cisco Prime Network Registrar handles and responds to DHCP requests, and to change the behavior of a DHCP server that you cannot normally do using the user interfaces. This chapter describes the extension points to which you can attach extensions for DHCPv4 and DHCPv6.

- Using Extensions, on page 1
- Language-Independent API, on page 4
- Tcl Extensions, on page 7
- C/C++ Extensions, on page 8
- DHCP Request Processing Using Extensions, on page 11
- Extension Dictionaries, on page 21
- Request and Response Dictionaries, on page 25
- Extension Point Descriptions, on page 27

Using Extensions

You can alter and customize the operation of the Cisco Prime Network Registrar DHCP server by using extensions, functions that you can write in Tcl or C/C++.

Follow this process to create an extension for use in the DHCP server:

1. Determine the task to perform. What DHCP packet process do I want to modify?
2. Determine the approach to use. How do I want to modify the packet process?
3. Determine the extension point to which to attach the extension.
4. Choose the language (Tcl or C/C++).
5. Write (and possibly compile and link) the extension.
6. Add the extension to the DHCP server configuration.
7. Attach the extension to the extension point.
8. Reload the DHCP server so that it recognizes the extension.
9. Test and debug the results.

Note While upgrading Cisco Prime Network Registrar, it is recommended to recompile all the DHCP C/C++ extensions (dex extensions).

Creating, Editing, and Attaching Extensions

You can create, edit, and attach extensions.

You can associate multiple extensions per extension point. Each extension executes in the order specified by the sequence number used when the attachment was created. In the web UI, the sequence is the order in which the extensions appear per extension point on the List DHCP Extension Points page. In the CLI, you use the *sequence-number* value with the **dhcp attachExtension** command.

For more on multiple extensions per extension point, see Multiple Extension Considerations, on page 6.

Local Advanced Web UI

To create and attach extensions, do the following:

Step 1 From the **Deploy** menu, choose **Extensions** under the **DHCP** submenu to open the List/Add DHCP Extensions page.

Step 2 Click the **Add Extensions** icon to open the Add DHCP Server Extension dialog box.

Step 3 After you create an extension, you can attach it to one or more of the extension points on this page. To show the extension points where you can attach extensions, on the List/Add DHCP Extensions page, click **DHCP Extension Points** tab.

Step 4 If you attach more than one extension for each extension point, you can change the sequence in which they are processed by clicking the arrow keys to rearrange the entries. To remove the extension, click the **Delete** icon.

CLI Command

Use the **extension** command, which requires this syntax:

```
nrcmd> extension name create language extension-file entry-point
```

The *entry-point* is the name of the entry point in the *extension-file*. You can also set an optional *init-entry* attribute value for the initial entry point each time the DHCP server loads the file (see init-entry, on page 27). You can call this function from any extension point bound to this module. You can also list the extensions using **extension list**.

To attach and detach an extension, use **dhcp attachExtension** and **dhcp detachExtension** for the DHCP server, which require this syntax:

```
nrcmd> dhcp attachExtension extension-point extension-name [sequence-number]
nrcmd> dhcp detachExtension extension-point [sequence-number]
```

The *sequence-number* applies if you attach multiple extensions per extension point, in increasing sequence order ranging from 1 through 32. If omitted, it defaults to 1.

To view the currently registered extensions, use **dhcp listExtensions** command.

Determining Tasks

The task to which to apply an extension is usually some modification of the DHCP server processing so that it better meets the needs of your environment. You can apply an extension at each of these DHCP server processing points, from receiving a request to responding to the client:

1. Receive and decode the packet.
2. Look up, modify, and process any client-class.

3. Build a response type.
4. Determine the subnet (or link, in the case of DHCPv6).
5. Find any existing leases.
6. Serialize the lease requests.
7. Determine the lease acceptability for the client.
8. Gather and encode the response packet.
9. Update stable storage of the packet.
10. Return the packet.

A more complete list of these steps (along with the extension points to use at each step) appears in DHCP Request Processing Using Extensions, on page 11.

For example, you might have an unusual routing hub that uses BOOTP configuration. This device issues a BOOTP request with an Ethernet hardware type (1) and MAC address in the *chaddr* field. It then sends out another BOOTP request with the same MAC address, but with a hardware type of Token Ring (6). Specifying two different hardware types causes the DHCP server to allocate two IP addresses to the device. The DHCP server normally distinguishes between a MAC address with hardware type 1 and one with type 6, and considers them different devices. In this case, you might want to write an extension that prevents the DHCP server from handing out two different addresses to the same device.

Deciding on Approaches

Many solutions are often available to a single problem. When choosing the type of extension to write, you should first consider rewriting the input DHCP packet. This is a good approach, because it avoids having to know the internal processing of the DHCP server.

For the problem described in Determining Tasks, on page 2, you can write an extension to solve it in either of these ways:

- Drop the Token Ring (6) hardware type packet.
- Change the packet to an Ethernet packet and then switch it back again on exit.

Although the second way involves a more complex extension, the DHCP client could thereby use either reply from the DHCP server. The second approach involves rewriting the packet, in this case using the **post-packet-encode** extension point (see post-packet-encode, on page 38). Other approaches require other extensions and extension points.

Choosing Extension Languages

You can write extensions in Tcl or C/C++. The capabilities of each language, so far as the DHCP server is concerned, are similar, although the application programming interface (API) is slightly different to support the two very different approaches to language design:

- **Tcl**—Although scripting in Tcl might be somewhat easier than scripting in C/C++, it is interpreted and single-threaded, and might require more resources. However, you might be less likely than in C/C++ to introduce a serious bug, and there are fewer chances of a server failure. Cisco Prime Network Registrar currently supports Tcl version 8.6.
- **C/C++**—This language provides the maximum possible performance and flexibility, including communicating with external processes. However, the C/C++ API is more complex than the Tcl API. Also, the possibility of a bug in the extension causing a server failure is also more likely in C/C++.

Language-Independent API

The following concepts are independent of whether you write your extensions in Tcl or C/C++.

Routine Signature

You need to define the extension as a routine in a file, which can contain multiple extension functions. You then attach the extension to one or more of the DHCP server extension points. When the DHCP server reaches that extension point, it calls the routine that the extension defines. The routine returns with a success or failure. You can configure the DHCP server to drop a packet on an extension failure.

You can configure one file—Tcl source file or C/C++ .dll or .so file—as multiple extensions to the DHCP server by specifying a different entry point for each configured extension.

The server calls every routine entry point with at least three arguments, the three dictionaries—request, response, and environment. Each dictionary contains many data items, each being a key-value pair:

- The extension can retrieve data items from the DHCP server by performing a get method on a dictionary for a particular data item.
- The extension can alter data items by performing a put or remove operation on many of the same named data items.

Although you cannot use all dictionaries at every extension point, the calling sequence for all routines is the same for every extension point. The extension encounters an error if it tries to reference a dictionary that is not present at a particular extension point. (See Extension Dictionaries, on page 21.)

Dictionaries

You access data in the request, response, and server through a dictionary interface. Extension points include three types of dictionaries—request, response, and environment:

- **Request dictionary**—Information associated with the DHCP request, along with all that came in the request itself. Data is string-, integer-, IP address-, and blob-valued.
- **Response dictionary**—Information associated with generating a DHCP response packet to return to the DHCP client. Data is string-, integer-, IP address-, and blob-valued.
- **Environment dictionary**—Information passed between the DHCP server and extension.

For a description of the dictionaries, see Extension Dictionaries, on page 21.

You can also use the environment dictionary to communicate between an extension attached at different extension points. When encountering the first extension point at which some extension is configured, the DHCP server creates an environment dictionary. The environment dictionary is the only one in which the DHCP server does not fix the names of the allowable data items. You can use the environment dictionary to insert any string-valued data item.

Every extension point in the flow of control between the request and response for the DHCP client (all extension points except **lease-state-change**, depending on the cause of the change) share the same environment dictionary. Thus, an extension can determine that some condition exists, and place a sentinel in the environment dictionary so that a subsequent extension can avoid determining the same condition.

In the previous example, the extension at the **post-packet-decode** extension point determines that the packet was the interesting one—from a particular manufacturer device, BOOTP, and Token Ring—and then rewrites the hardware type from Token Ring to Ethernet. It also places a sentinel in the environment dictionary and

then, at a very simple extension at the **post-packet-encode** extension point, rewrites the hardware type back to Token Ring.

Utility Methods in Dictionaries

Each dictionary has associated utility methods with which you can reset the trace level for an extension and log values to an output file.

Configuration Errors

Extensions can fail for many reasons. For example:

- The server cannot find the file.
- The entry point or **init-entry** point does not appear in the file.
- The extension itself can return a failure from an **init-entry** call.

By itself, an extension failure is not fatal and does not prevent the DHCP server from starting. However, if you configure that failed extension at any extension point, the server will not start. Therefore, to debug the configuration process, you can configure the extension at the **init-entry** point (see init-entry, on page 27) without attaching it to an extension point. When you complete that process successfully, you can attach your extension to an extension point.

Communicating with External Servers

You can write extensions that communicate with external servers or databases to affect the client class or validate incoming DHCP client requests. Writing such extensions is a complex task, requiring considerable skill and debugging expertise. Such extensions must be multithreaded, and need to communicate very swiftly with the external server if the DHCP server performance is to remain at an acceptable level.

Performance degradations can result from extensions stalling the threads that are processing requests. A thread stalls while an extension communicates with an external server. If this interaction takes more than 50 to 100 milliseconds, this severely affects server performance. This might or might not impact you in the particular environment in which you deploy this extension.

One way to avoid having to communicate with an external server synchronously (that is, stalling the incoming DHCP client request processing to communicate with the external server) is to avoid performing this communication while processing the DHCP client request. This sounds obvious, and it also sounds, on the face of it, impossible. However, due to the nature of the DHCP client-server protocol, there is a way to decouple the access to the external server from the DHCP client request processing.

To avoid this bottleneck, use a caching mechanism as part of the extension. When the server calls the extension for a request, have it check a cache (with proper locking to avoid multithreading problems) for the client data. If the client is:

- In the cache (and did not expire), have the extension accept or reject the request depending on the data in the cache.
- Not in the cache, have the extension queue a request to the external server (preferably over UDP), and then drop the DHCP client request. By the time the client retransmits the request, the data should be in the cache.

This caching mechanism requires the extension to have a receiver thread (started and stopped in the **init-entry** extension point). This thread reads the socket and updates the cache with the response. This thread (or a

separate one) would also need to time out and remove old items from the cache. Using a single thread, however, might require setting a larger receive socket buffer size.

These techniques are only necessary if the load on the DHCP server is high and the speed of the external server is not high enough to support the required performance of the DHCP server load. However, this situation turns out to be all too common in practice. And, consider what can happen if the external server is unreachable (when connection timeouts are likely to be for minutes and not seconds).

Recognizing Extensions

The DHCP server only recognizes extensions when it initially configures itself at start or reload time. You can change an extension or the configuration for extensions in general. However, until you reload or restart the server, the changes have no effect. Forgetting to reload the DHCP server can be a frequent source of errors while debugging extensions.

The reason Cisco Prime Network Registrar requires a reload is to ensure minimum processing impact by preloading extensions and getting them ready at server configuration time. While this approach is useful in production mode, it might cause some frustration when you debug extensions.

Multiple Extension Considerations

You can register multiple extensions at any extension point. The DHCP server runs all the extensions attached to an extension point before resuming processing, the conditions being:

- An extension should not explicitly set a data item unless the extension explicitly requires that behavior. For example (as described for the drop environment dictionary data item in Table 31-5 on page 31-25), extensions can request dropping the client packet at most extension points.

  The server calls the first extension registered at an extension point with *drop* set to False. One or more extensions can set this to True or False. If all extensions were to explicitly set *drop* to either True or False, then the server would take whatever action the last extension to run requested.

  This might not be the desired behavior. Thus, for this data item, it is better for an extension to set *drop* to True only if it wants the packet to be dropped. That way, if all extensions play by this rule, the packet would be dropped if any of the extensions request it.

- An extension might want to return immediately if *drop* is True, as there may not be a need for the extension to do its processing if another one desires the packet to be discarded.
- If the environment dictionary is used to store items for use in later extension points, those data item names might want to use a prefix or suffix that is unique to that extension. This reduces the chance for data item name conflicts.
- At least one environment dictionary data item, the *user-defined-data* (see Table 5: General Environment Dictionary Data Items) that you can use to store data with a lease (for DHCPv4) or client (DHCPv6), requires special attention.

  This data item can be difficult for more than one extension to use, unless those extensions take special care to preserve and recognize each other’s values. Thus, it might not be possible for more than one extension to assume it can use this data item.

- You must specify whether the extensions should be run first, or last, if such a need exists. For example, you should run extensions that cause the server to drop certain packets first, because this reduces the processing burden on the server (assuming the remaining extensions return immediately if *drop* is true).

Tcl Extensions

If you choose to write your extensions in Tcl, you should understand the Tcl API, how to handle errors and Boolean variables, and how to initialize Tcl extensions. Cisco Prime Network Registrar uses Tcl version 8.6.

Note A single Tcl interpreter is used by the DHCP server. This can have severe performance implications. Tcl extensions are best used for prototyping more complex logic before reworking it to be a high performance multi-threaded dex extension or very simple operations that are quick.

Tcl Application Program Interface

Every Tcl extension has the same routine signature:

```
proc yourentry { request response environ } { # your-code }
```

To operate on the data items in any dictionary, you must treat these arguments as commands. Thus, to get the *giaddr* of the input packet, you would write:

```
set my_giaddr [ $request get giaddr ]
```

This sets the Tcl variable *my_giaddr* to the string value of the *giaddr* in the packet; for example, 10.10.1.5 or 0.0.0.0.

You could rewrite the *giaddr* in the input packet by using this Tcl statement:

```
$request put giaddr "1.2.3.4"
```

To configure one routine entry for multiple extension points and to alter its behavior depending on the extension point from which the server calls it, the DHCP server passes the ASCII name of the extension point in the environment dictionary under the key **extension-point**.

For some sample Tcl extensions, see the Cisco Prime Network Registrar directory; /opt/nwreg2/local/examples/dhcp/tcl by default.

Dealing with Tcl errors

You generate a Tcl error if you:

- Reference a dictionary that is not available.
- Reference a dictionary data item that is not available.
- Request a put operation on an invalid data item, for example, an invalid IP address.

In these cases, the extension immediately fails unless you surround the statement with a catch error statement:

```
catch { $request put giaddr "1.2.3.a" } error
```

Configuring Tcl Extensions

To configure a Tcl extension, write it and place it in the following extensions directory:

/var/nwreg2/local/extensions/dhcp/tcl

When the DHCP server configures an extension during startup, it reads the Tcl source file into an interpreter. Any syntax errors in the source file that would render Tcl interpreter unable to load the file would also fail the extension. Typically, the DHCP server generates an error traceback in the log file from Tcl to help you to find the error.

Handling Boolean Variables in Tcl

In the environment dictionary, the Boolean variables are string-valued and have a value of **true** or **false**. The DHCP server expects an extension to set the value to **true** or **false**. However, in the request or response dictionaries, Boolean values are single-byte numeric format, and **true** is **1** and **false** is **0**. While more efficient for the C/C++ extensions, this approach does make the Tcl API a bit more complex.

Init-Entry Extension Point in Tcl

Tcl extensions support the **init-entry** extension point (see init-entry, on page 27), and the arguments supplied in the *init-args* parameter to the command line appear in the environment dictionary associated with the key **arguments**.

A single Tcl interpreter is used by the DHCP server. This avoids issues with information flow and allows use of global variables to store information that could be used for a follow on client request, but does have a severe **impact on performance**.

Note that all Tcl extensions share the Tcl interpreter. If your Tcl extension initializes global variables or defines procedures, ensure that these do not conflict with some other Tcl extension global variables or procedure names.

C/C++ Extensions

All DHCP C/C++ extensions are **dex** extensions, short for DHCP Extension.

C/C++ API

The routine signature for both the **entry** and **init-entry** routines for the C/C++ API is:

```
typedef int (DEXAPI * DexEntryPointFunction)(
int iExtensionPoint,
dex_AttributeDictionary_t* pRequest,
dex_AttributeDictionary_t* pResponse,
dex_EnvironmentDictionary_t* pEnviron );
```

Along with pointers to three structures, the integer value of the extension point is one of the parameters of each routine.

The C/C++ API is specifically constructed so that you do not have to link your shared library with any Cisco Prime Network Registrar DHCP server files. You configure the entry to your routine when you configure the extension. The necessary call-back information for the operations to perform on the request, response, and environment dictionaries is in the structures that comprise the three dictionary parameters passed to your extension routine.

The DHCP server returns all binary information in network order, which is not necessarily properly aligned for the executing architecture.

Using Types in C/C++

Many C/C++ routines are available that use types, for example, **getByType()**. These routines are designed for use in performance-sensitive environments. The reasoning behind these routines is that the extension can acquire pointers to types once, for example, in the **init-entry** point, and thereafter use the pointers instead of string-valued names when calling the routines of the C/C++ API. Using types in this manner removes one hash table lookup from the extension processing flow of execution, which should improve (at least fractionally) the performance of any extension.

Building C/C++ Extensions

The directory /opt/nwreg2/local/examples/dhcp/dex contains sample C/C++ extension code, as well as a short makefile designed to build the sample extensions. To build your own extensions, you need to modify this file. It has sections for Microsoft Visual C++ and GNU C++. Simply move the comment lines to configure the file for your environment.

Your extension needs to reference the include file dex.h. This file contains the information your program needs to use the C/C++ API.

After you build the .so file (all dex extensions are shared libraries), you need to move them into the /var/nwreg2/local/extensions/dhcp/dex directory. You can then configure them.

Using Thread-Safe Extensions in C/C++

The DHCP server is multithreaded, so any C/C++ extensions written for it must be thread-safe. Multiple threads, and possibly multiple processors, must be capable of calling these extensions simultaneously at the same entry point. You should have considerable experience writing code for a multithreaded environment before designing C/C++ extensions for Cisco Prime Network Registrar.

Caution All C/C++ extensions must be thread-safe. If not, the DHCP server will not operate correctly and will fail in ways that are extremely difficult to diagnose. All libraries and library routines that these extensions use must also be thread-safe.

On several operating systems, you must ensure that the runtime functions used are really thread-safe. Check the documentation for each function. Special thread-safe versions are provided (often *functionname* _r) on several operating systems.

Be aware that if any thread makes a non-thread-safe call, it affects any of the threads that make up the safe or locked version of the call. This can cause memory corruptions, server failures, and so on.

Diagnosing these problems is extremely difficult, because the cause of these failures are rarely apparent. To cause a server failure, you need very high server loads or multiprocessor machines with many processes. You might need running times of several days. Often, problems in extension implementation might not appear until after sustained periods of heavy load.

Because some runtime or third-party libraries might make non-thread-safe calls that you cannot detect, check your executables as to what externals are being linked (**nm** on UNIX).

If the routines of a library call the routines without the _r suffixes, displayed in the following table, the library is not thread-safe, and you cannot use it. The interfaces to the thread-safe versions of these library routines can vary based on operating system.

asctime_r	getgrid_r	getnetent_r	getrpcbynumber_r	lgamma_r
ctermid_r	getgrnam_r	getprotobyname_r	getrpcent_r	localtime_r
ctime_r	gethostbyaddr_r	getprotobynumber_r	getservbyname_r	nis_sperror_r
fgetgrent_r	gethostbyname_r	getprotoent_r	getservbyport_r	rand_r
fgetpwent_r	gethostent_r	getpwnam_r	getservent_r	readdir_r
fgetspent_r	getlogin_r	getpwent_r	getspent_r	strtok_r
gamma_r	getnetbyaddr_r	getpwuid_r	getspnam_r	tmpnam_r
getgrent_r	getnetbyname_r	getrpcbyname_r	gmtime_r	ttyname_r

Configuring C/C++ Extensions

Because the .dll and .so files are active when the server is running, it is not a good idea to overwrite them. After you stop the server, you can overwrite the .dll and .so files with newer versions.

Debugging C/C++ Extensions

Because your C/C++ shared library runs in the same address space as the DHCP server, and receives pointers to information in the DHCP server, any bugs in your C/C++ extension can very easily corrupt the DHCP server memory, leading to a server failure. For this reason, use extreme care when writing and testing a C/C++ extension. Frequently, you should try the approach to an extension with a Tcl extension and then code the extension in C/C++ for increased performance.

Pointers into DHCP Server Memory in C/C++

The C/C++ extension interface routines return pointers into DHCP server memory in two formats:

- char* pointer to a series of bytes.
- Pointer to a structure called an abytes_t, which provides a pointer to a series of bytes with an associated length (defined in dex.h).

In both cases, the pointers into DHCP server memory are valid while the extension runs at that extension point. They are also valid for the rest of the extension points in the series processing this request. Thus, an abytes_t pointer returned in the **post-packet-decode** extension point is still valid in the **post-send-packet** extension point.

The pointers are valid for as long as the information placed in the environment dictionary is valid. However, there is one exception. One C/C++ routine, **getType**, returns a pointer to an abytes_t that references a type. These pointers are valid through the entire life of the extension. Typically, the server would call this routine in the **init-entry** extension point and save the pointers to the abytes_t structures that define the types in the static data of the shared library. Pointers to abytes_t structures returned by **getType** are valid from the **init-entry** call for initialization until the call for uninitialization.

Init-Entry Entry Point in C/C++

The DHCP server calls the **init-entry** extension point (see init-entry, on page 27) once when configuring each extension and once when unconfiguring it. The dex.h file defines two extension point values passed as the extension points for the configure and unconfigure calls: DEX_INITIALIZE for configure and DEX_UNINITIALIZE for unconfigure. The environment dictionary value of the extension-point data item is **initialize** or **uninitialize** in each call.

When calling the **init-entry** extension point for **initialize**, if the environment dictionary data item **persistent** contains the value **true**, you can save and use the environment dictionary pointer any time before the return from the **uninitialize** call. In this way, background threads can use the environment dictionary pointer to log messages in the server log file. Note that you must interlock all access to the dictionary to ensure that at most one thread processes a call to the dictionary at a time. You can use the saved dictionary pointer up to when the extension returns from the **uninitialize** call. This way, the background threads can log messages during termination.

DHCP Request Processing Using Extensions

The Cisco Prime Network Registrar DHCP server has extension points to which you can attach your own extensions. They have descriptive names that indicate where in the processing flow of control to use them.

Because the extension points are tied to the processing of input requests from DHCP clients, it is helpful to understand how the DHCP server handles requests. Request processing comes in three general stages:

1. Initial request processing (see Table 1: Initial Request Processing Using Extensions)
2. DHCPv4 or DHCPv6 processing (see Table 2: DHCPv4 or DHCPv6 Request Processing Using Extensions)
3. Final response processing (see Table 3: Final Response Processing Using Extensions)

Table 1: Initial Request Processing Using Extensions

Client Request Processing Stage	Extension Point Used
1. Receive a packet from a DHCP client.	**pre-packet-decode**
2. Decode the packet.	**post-packet-decode**
3. Determines the client-classes.	
4. Modifies the client-class.	**post-class-lookup**
5. Processes the client-classes, looking up clients.	**pre-client-lookup post-client-lookup**
6. Build a response container from the request.	

Table 2: DHCPv4 or DHCPv6 Request Processing Using Extensions

Client Request Processing Stage	Extension Point Used
1. In DHCPv4, find a lease already associated with this client, if any, or locate a new lease for the client.	
2. Serialize all requests associated with this client (processing continues when the request reaches the head of the serialization queue).	

Client Request Processing Stage	Extension Point Used
3. In DHCPv6, process the client request, generating leases if necessary. The server tries to provide the client with at least one preferred lease for each usable prefix per binding. You can generate leases and change lease states multiple times for a client request, but not for reserved leases.	**generate-lease** and **lease-state-change** (multiple calls are possible for both in DHCPv6)
4. Determine if the lease is (still) acceptable for this client (can occur multiple times in DHCPv6).	**check-lease-acceptable**
5. Initiate DNS Update operations as necessary (can occur multiple times in DHCPv6).	

Table 3: Final Response Processing Using Extensions

Client Response Processing Stage	Extension Point Used
1. Gather all the data to include in the response packet.	
2. Write to the lease database.	
3. Prepare the response packet for encoding.	**pre-packet-encode**
4. Encode the response packet for transmission to the client.	**post-packet-encode**
5. Send the packet to the client.	**post-send-packet**
6. Release all context for the client and request.	**environment-destructor**

These steps and additional opportunities for using extensions are explained in the following sections. The extension points are indicated in **bold**.

Enabling DHCPv6 Extensions

By default, extensions are assumed to support only DHCPv4. To write DHCPv6 extensions, you must implement an **init-entry** extension point that must:

1. Set the *dhcp-support* environment data item to **v4** (for DHCPv4 only, the preset value), **v6** (for DHCPv6 only), or **v4,v6** (for DHCPv4 and DHCPv6). This data item indicates to the server what the extension is willing to support.
2. Set the *extension-extensionapi-version* environment data item to **2**. (The *dhcp-support* data item is ignored unless the *extension-extension-api-version* is set to **2**.)

You might need to write separate extensions for DHCPv4 and DHCPv6, because of the differences in packet formats, DHCP protocol, and internal server data. However, the fundamentals of both kinds of extensions are very much the same.

The server calls these extension points at essentially the same places during processing, although it can call some DHCPv6 extension points multiple times due to the possibility of multiple lease requests per client.

Receiving Packets

The DHCP server receives DHCPv4 packets on port 67 and DHCPv6 packets on port 547 (the DHCP input ports) and queues them for processing. It attempts to empty the UDP input queue as quickly as possible and keeps all of the requests that it receives on an internal list for processing as soon as a free thread is available to process them. You can configure the length of this queue, and it will not grow beyond its maximum configured length.

Decoding Packets

When a free thread is available, the DHCP server allocates to it the task of processing an input request. The first action it takes is to decode the input packet to determine if it is a valid DHCP client packet. As part of this decoding process, the DHCP server checks all options to see if they are valid—if the lengths of the options make sense in the overall context of the request packet. It also checks all data in the DHCP request packet, but takes no action on any data in the packet at this stage.

Use the **pre-packet-decode** extension point to rewrite the input packet. After the DHCP server passes this extension point, it stores all information from the packet in several internal data structures to make subsequent processing more efficient.

Determining Client-Classes

If you configure an expression in the *client-class-lookup-id*, it is at this stage that the DHCP server evaluates the expression (see Using Expressions for a description of expressions). The result of the expression is either <null>, or something converted to a string. The value of the string must be either a client-class name or <none>. In the case of <none>, the server continues to process the packet in the same way as if there were no *client-class-lookup-id* configured. In the case of a <null> response or an error evaluating the *client-class-lookup-id*, the server logs an error message and drops the packet (unless an extension configured at the **post-class-lookup** extension point specifically instructs the server not to drop the packet). As part of the process of setting the client-class, the DHCP server evaluates any *limitation-id* configured for that client-class and stores it with the request.

Modifying Client-Classes

After the DHCP server evaluates the *client-class-lookup-id* and sets the client-class, it calls any extension attached to the **post-class-lookup** extension point. You can use that extension to change any data that the client-class caused to become associated with the request, including the *limitation-id*. The extension also learns if the evaluation of the *client-class-lookup-id* dropped the packet. The extension not only finds out if it needs to drop the packet, it instructs the server not to drop the packet if it wants the server not to do so.

Also, an extension running at the **post-class-lookup** extension point can set a new client-class for the request, and uses the data from that client-class instead of the current one. This is the only extension point where setting the client-class actually uses that client-class for the request.

Processing Client-Classes

If you enabled client-class processing, the DHCP server performs it at this stage.

Use the **pre-client-lookup** extension point to affect the client to look up, possibly by preventing the lookup or supplying data that overrides the existing data. After the DHCP server passes the **pre-client-lookup**

extension point, it looks up the client (unless the extension specifically prevents it) in the local database or in an LDAP database, if one was configured.

After the server looks up the client, it uses the data in the client entry to fill in additional internal data structures. The DHCP server uses data in the specified client-class entry to complete any data that the client entry does not specify. When the DHCP server retrieves all the data stored in the various places in the internal data structures for additional processing, it runs the next extension point.

Use the **post-client-lookup** extension point to review the operation of the client-class lookup process, such as examining the internal server data structures filled in from the client-class processing. You can also use the extension point to change any data before the DHCP server does additional processing.

Building Response Containers

At this stage, the DHCP server determines the request type and builds an appropriate response container based on the input. For example, if the request is a DHCPDISCOVER, the server creates a DHCPOFFER response to perform the processing. If the input request is a BOOTP request, the server creates a BOOTP response to perform the response processing.

For DHCPv6, a server creates an ADVERTISE or REPLY packet, depending on the request.

Determining Networks and Links

The DHCP server must determine the subnet from which every request originated and map that into a set of address pools, scopes, prefixes, or links that contain IP addresses.

For DHCPv4, internal to the DHCP server is the concept of a network, which, in this context, refers to a LAN segment or physical network. In the DHCP server, every scope or prefix belongs to a single network.

Some scopes or prefixes are grouped together on the same network because their network numbers and subnet masks are identical. Others are grouped because they are related through the primary-scope or prefix pointer.

The Cisco Prime Network Registrar DHCP server determines the network to use to process a DHCP client request in the following sequence:

1. Determining the source address, either the *giaddr* or, if the *giaddr* is zero, the address of the interface on which the request arrived.
2. Using this address to search for any scope or prefix that was configured in the server that is on the same subnet as this address. If the server does not find a scope or prefix, it drops the request.
3. After finding the scope or prefix, using its network in subsequent processing.

For DHCPv6 processing, see Determining Links and Prefixes.

Finding Leases

For DHCPv4, now that when the DHCP server establishes the network, it searches the hash table held at the network level to see if the network already knows the *client-id*. “Already knows,” in this context, means that this client previously received an offer or a lease on this network, and the lease was not offered to or leased by a different client since that time. Thus, a current lease or an available expired lease appears in the network level hash table. If the DHCP server finds a lease, it proceeds to the next step, which is to serialize all requests for the same IP address.

If the DHCP server does not find a lease, and if this is a BOOTP or DHCPDISCOVER request, the server looks for a reserved lease from a scope or prefix in the network.

If it finds a reserved lease, the server checks whether the scope or prefix and lease are both acceptable. The following must be true regarding the reserved lease and the scope or prefix that contains it:

- The lease must be available (not leased to another DHCP client).
- The scope or prefix must support the request type (BOOTP or DHCP).
- The scope or prefix must not be in a deactivated state.
- The lease must not be in a deactivated state.
- The selection tags must contain all of the client *selection-criteria* and none of the client *selection-criteria-excluded*.
- The scope or prefix must not be in a renew-only state.

If the reserved lease is acceptable, the server proceeds to the next step, which is to serialize all requests for the IP address. Having failed to find an existing or reserved lease for this client, the server now attempts to find any available IP addresses for this client.

The general process the DHCP server uses is to scan all of the scopes or prefixes associated with this network in round-robin order, looking for one that is acceptable for the client and also has available addresses. An acceptable scope or prefix has the following characteristics:

- If the client has *selection-criteria* associated with it, the selection tags must contain all of the client inclusion criteria.
- If the client has *selection-criteria-excluded* associated with it, the selection tags must contain none of the client exclusion criteria.
- The scope or prefix must support the client request type—If the client request is a DHCPREQUEST, you must enable the scope or prefix for DHCP. Likewise, if the request is a BOOTP request, you must enable the scope or prefix for BOOTP and dynamic BOOTP.
- It must not be in a renew-only state.
- It must not be in deactivated state.
- It must have an available address.

If the server does not find an acceptable scope or prefix, it logs a message and drops the packet.

For DHCPv6 processing, see Determining Links and Prefixes.

Serializing Lease Requests

Because multiple DHCP requests can be in process simultaneously for one client and lease, you must serialize DHCPv4 requests at the lease level. The server queues them on the lease and processes them in the order of queueing.

For DHCPv6, the server serializes on the client (per link) and not on the lease.

Determining Lease Acceptability

For DHCPv4, the DHCP server now determines if the lease is (still) acceptable for the client. In the case where this is a newly acquired lease for a first-time client, it will be acceptable. However, in the case where the server processes a renewal for an existing lease, the acceptability criteria might have changed since the server granted the lease, and you need to check its acceptability again.

If the client has a reservation that is different from the current lease, the server first determines if the reserved lease is acceptable. The criteria for release acceptability are:

- The reserved lease must be available.

- The reserved lease must not be in a deactivated state.
- The scope or prefix must not be in a deactivated state.
- If the request is BOOTP, the scope or prefix must support BOOTP.
- If the request is DHCP, the scope or prefix must support DHCP.
- If the client has any *selection-criteria*, the selection tags must contain all of the client inclusion criteria.
- If the client has any *selection-criteria-excluded*, the selection tags must contain none of the client exclusion criteria.
- If the client previously associated with this lease is not the current client, the scope or prefix must not be in a renew-only state.

If the reserved lease meets all of these criteria, the DHCP server considers the current lease unacceptable. If there is no reserved lease for this client, or the reserved lease did not meet the criteria for acceptability, the DHCP server examines the current lease for acceptability.

The criteria for acceptability are:

- The lease must not be in a deactivated state.
- The scope or prefix must not be in a deactivated state.
- If the request is BOOTP, the scope or prefix must support BOOTP. If the request is DHCP, the scope or prefix must support DHCP.
- If the client does not have a reservation for this lease, and the request is BOOTP, the scope or prefix must support dynamic BOOTP.
- If the client does not have a reservation for this lease, no other client can either.
- If the client has any *selection-criteria*, the selection tags must contain all of the client inclusion criteria.
- If the client has any *selection-criteria-excluded*, the selection tags must contain none of the client exclusion criteria.
- If the client previously associated with this lease is not the current client, the scope or prefix must not be in a renew-only state.

Tip At this point in the DHCP server processing, you can use the **check-lease-acceptable** extension point. You can use it to change the results of the acceptability test. Do this only with extreme care.

Upon determining that a lease is unacceptable, the DHCP server takes different actions, depending on the particular DHCP request currently being processed.

- **DHCPDISCOVER**—The DHCP server releases the current lease and attempts to acquire a different, acceptable lease for this client.
- **DHCPREQUEST SELECTING**—The DHCP server sends a NACK to the DHCP client because the lease is invalid. The client should then immediately issue a DISCOVER request to acquire a new DHCPOFFER.
- **DHCPRENEW**, **DHCPREBIND**—The DHCP server sends a NACK to the DHCP client to attempt to force the DHCP client into the INIT phase (attempt to force the DHCP client into issuing a DHCPDISCOVER request). The lease is still valid until the client actually issues the request.
- **BOOTP**—The DHCP server releases the current lease and attempts to acquire a different, acceptable lease for this client.

Caution Take extreme care with the **check-lease-acceptable** extension point. If the answer the extension point returns does not match the acceptability checks in the search for an available lease performed in a DHCPDISCOVER or dynamic BOOTP request, an infinite server loop can result (either immediately or on the next DHCPDISCOVER or BOOTP request). In this case, the server would acquire a newly available lease, determine that it was not acceptable, try to acquire a newly available lease, and determine that it was not acceptable, in a continuous loop.

DHCPv6 Leasing

The DHCP server processes IPv6 lease requests by scanning the client request for IA_NA, IA_TA, and IA_PD options (see DHCPv6 Bindings). For each of these options, the server considers any leases that the client explicitly requests. If the lease already exists for the client and binding (IA option and IAID), the server determines if the lease is still acceptable. For leases that the client requests that do not already exist for the client, the server tries to give that lease to the client if:

- Another client or binding is not already using the lease.
- The prefix for the lease has the client-request flag set in its *allocation-algorithms* attribute.
- The lease is usable and on a usable prefix (see the DHCPv6 Prefix Usability, on page 17).

Next, the server tries to assure that clients are using reservations and that a client has a usable lease with a nonzero preferred lifetime on each usable prefix on the link. Thus, the server processes each of these bindings as follows:

1. Adds any client reservations (not already in use) to the binding, provided the reservation flag is set in the prefix *allocation-algorithms* attribute. The server uses the first binding of the appropriate type for the reservation; that is, it uses address leases for IA_NA bindings and prefix leases for IA_PD bindings.
2. If the client has no lease with a nonzero preferred lifetime on each prefix that the client can use, the server tries to allocate a lease to the client. The prefix *allocation-algorithms* flags control how the server allocates the lease.

DHCPv6 Prefix Usability

A usable prefix:

- Is not deactivated.
- Did not expire.
- Allows leases of the binding type.
- Matches the client selection criteria, if any.
- Does not match the client selection exclusion criteria, if any.

DHCPv6 Lease Usability

A usable lease is:

- Not unavailable.
- Not revoked.
- Not deactivated.

- Not reserved for a different client.
- Not subject to *inhibit-all-renews* or *inhibit-renews-at-reboot*.
- Renewable if being renewed (IA_TA leases are not renewable).
- Leasable with a nonzero valid lifetime.

DHCPv6 Lease Allocation

When the server needs to allocate a new lease on a prefix, it calls any extensions registered at the **generate-lease** extension point if the prefix extension flag is set in the *allocation-algorithms* attribute. (See generate-lease, on page 34.) The extensions can supply the address (IA_NA or IA_TA binding) or prefix (IA_PD binding) to be assigned, request that the server use its normal allocation algorithm (if enabled in *allocation-algorithms*), or request the server to skip assigning a lease on this prefix. The server might call the extension again if it supplied an address or prefix that is not valid or is already in use.

If extensions are not allowed, there are no extensions registered, or the extension requests the normal allocation algorithm of the server, the server allocates a randomly generated address or finds the first best-fit available prefix (as controlled by the prefix *allocation-algorithms* attribute) and creates the lease.

Once the server has a lease and does an acceptability check on it (see DHCPv6 Lease Usability, on page 17), the server calls any extensions registered at the **check-lease-acceptable** extension point to allow the extension to alter the acceptability of the lease. (See check-lease-acceptable, on page 36.) You would typically only use this extension point to change an acceptable result into a unacceptable one; however, the server allows a unacceptable result to be changed to an acceptable one, although this is strongly discouraged because of possibly adverse consequences. If the lease is not acceptable, the server likely tries to allocate another lease; thus, use care to avoid an infinite loop. In some cases, you might need the **check-lease-acceptable** and **generate-lease** extension points for full control of the leases a client gets: **generate-lease** can request the server to skip allocation of the lease.

The server calls the **check-lease-acceptable** extension point for each client request for each lease.

Gathering Response Packet Data

In this stage of processing, the DHCP server collects all the data to send back in the DHCP response and determines the address and port to which to send the response. You can use the **pre-packet-encode** extension point to change the data sent back to the DHCP client in the response, or to change the address to which to send the DHCP response. (See pre-packet-encode, on page 38.)

Caution Any packets dropped at the **pre-packet-encode** extension point, whether they be DHCP or BOOTP packets, still show the address to be leased in the Cisco Prime Network Registrar lease state database, for as long as the remaining lease time. Because of this, it is advisable to drop packets at an earlier point.

Encoding Response Packets

In this stage, the DHCP encodes the information in the response data structure into a network packet. If this DHCP client requires DNS activity, the DHCP server queues a DNS work request to the DNS processing subsystem in the DHCP server. That request runs whenever it can, but generally not before sending the packet to the client. (See pre-packet-encode, on page 38.)

Updating Stable Storage

At this stage, the DHCP server ensures that the on-disk copy of the information is up to date with respect to the IP address before proceeding. For DHCPv6, this can involve multiple leases.

Sending Packets

Use the **post-send-packet** extension point (see post-send-packet, on page 38) for any processing that you want to perform outside of the serious time constraints of the DHCP request-response cycle. After the server sends the packet to the client, it calls this extension point.

Processing DNS Requests

Here is a simplified view of what the DHCP server does to add names to DNS:

1. **Builds up a name to use for the A record**—The DHCP server creates the name that it will use in the forward (A record) DNS request. For DHCPv6, these are AAAA records. The DNS name comes from a variety of sources including the client-requested-host-name and client-domain-name data items, which are usually populated from options in the DHCP request, and the DNS update configuration (including the host-name-generator/v6-host-name-generator expressions).
2. **Tries to add the name, asserting that none exists yet**—At this stage, the prerequisites for the DNS name update request indicate that the name should not exist. If it succeeds, the DHCP server proceeds to update the reverse record.
3. **Tries to add the name, asserting that the server should supply it**—The server tries to add the hostname, asserting that the host exists and has the same TXT record (or DHCID record for DHCPv6) as the one that was sent.

    - If this succeeds, the server proceeds to the next step.
    - If it fails, the server checks if it exhausted its naming retries, in which case it exits and logs an error.
    - If it did not exhaust its naming entries, it returns to the first step, which is to build up a name for the A record.

    For DHCPv6, the server uses DHCID records instead of TXT records. Also, DHCPv6 clients can have multiple leases and the forward zones can be the same or potentially different.

4. **Updates the reverse record**—Now that the DHCP server knows which name to associate with the reverse (PTR) record, it can update the reverse record with no prerequisites, because it can assume it is the owner of the record. If the update fails, the DHCP server logs an error.

Tracing Lease State Changes

The server calls the **lease-state-change** extension point whenever (and only when) a lease changes state. The existing state is in the response dictionary **lease-state** data item. The new state is in the environment dictionary under **new-state**. The **new-state** never equals the existing state (if it did, the server would not call the extension). You should consider this extension to be read-only and not make modifications to any dictionary items, because the server calls it in many different places. Use this extension point only for tracking changes to a lease state.

Controlling Active Leasequery Notifications

The server determines whether a lease is queued for active leasequery notifications based on the *leasequery-send-all* attribute of dhcp-listener. If this attribute is enabled, the DHCP server always sends a notification to an active leasequery client. If disabled or unset, the DHCP server only sends notifications which are necessary to maintain accurate state in the active leasequery client.

To allow customer written extensions to control the sending of a lease (such as only on specific state changes), a new data item, *active-leasequery-control*, has been added to both the request and response dictionaries. These data items have three values:

- 0—unspecified (the server determines whether to send the notification)
- 1—send (the server will send the notification)
- 2—do not send (the server will not send the notification)

The *active-leasequery-control* data item is initialized as 0, unspecified.

Note These data items may be written and read, but the value that is read is only the value that might have been previously written.

These data items can force the DHCP server to take specific actions after being written, but reading them without previously writing them will always return 0, unspecified. These data items will not let you determine the choices that the DHCP server makes when it comes to deciding whether or not to send a message to an active leasequery client concerning the changes (if any) made to a lease that is being processed. Thus, these data items are technically read/write, but reading them will only allow you to determine what you may have previously written into them.

These data items are examined (the response dictionary is examined first, then the request) when the lease is written to the internal lease state database as that is when the lease is also queued for active leasequery notification. This occurs after the check-lease-acceptable and lease-state-change extensions points, but prior to the pre-packet-encode extension point. Therefore, any changes made to these attributes at or after the pre-packet-encode extension point will be ignored.

Whether a lease is queued for active leasequery notification is determined as follows:

Response's active-leasequery-control	Request's active-leasequery-control	Leasequery-send-all	Action
0—unspecified	0—unspecified	False or unset	Conditional (see *leasequery-send-all* attribute description)
0—unspecified	0—unspecified	True	Sent
0—unspecified	1— send	Ignored	Sent
0—unspecified	2—don't send	Ignored	Not Sent
1— send	Ignored	Ignored	Sent
2—don't send	Ignored	Ignored	Not Sent

Note The *active-leasequery-control* of response and request is examined prior to any examination of the *leasequery-send-all* attribute.

If either of these dictionary data items has a value other than unspecified, that value will override any value configured in the *leasequery-send-all* attribute of the dhcp listener.

Note You cannot control the sending of active leasequery information by writing a single extension that runs only at the *lease-state-change* extension point, because that extension point is only called when there is a change in state of a lease.

Lease state changes may not occur when you might expect them to. For example, if a lease is leased, and that same client goes through a DISCOVER/OFFER/REQUEST/ACK cycle, the *lease-state-change* extension point is not called since the lease does not actually go through a state change internally and it remains leased throughout the cycle. Thus, to gain absolute control over the transmission of information to active leasequery clients, you have to initialize the *active-leasequery-control* attribute in request processing, and then possibly alter it or override it by operating on the response dictionary value at the *lease-state-change* extension point.

Extension Dictionaries

Every extension is a routine with three arguments. These arguments represent the request dictionary, response dictionary, and environment dictionary. Not every dictionary is available to every extension. The following table shows the extensions points and the dictionaries that are available to them.

Table 4: Extension Points and Relevant Dictionaries

Extension Point	Dictionary
init-entry	Environment
pre-packet-decode	Request, Environment
post-packet-decode	Request, Environment
pre-client-lookup	Request, Environment
post-client-lookup	Request, Environment
post-class-lookup	Request, Environment
generate-lease	Request, Response, Environment
lease-state-change	Response, Environment
check-lease-acceptable	Request, Response, Environment
pre-packet-encode	Request, Response, Environment
post-packet-encode	Request, Response, Environment

Extension Point	Dictionary
post-send-packet	Request, Response, Environment
environment-destructor	Environment

Note When the server sends DHCPv6 Reconfigure messages, it can call the **pre-packet-encode**, **post-packet-encode**, and **post-send-packet** extension points without a request.

For the request and response dictionaries, you can use the **isValid** method to probe if the dictionary is available for an extension point.

Each of the three dictionaries consists of name-value pairs. The environment dictionary, which is available to every extension point, is the simplest dictionary. The request and response dictionaries are more complex and their data is typed. Thus, when you set a value in one of these dictionaries, you need to match the data type to the value. You can use the dictionaries for getting, putting, and removing values.

Environment Dictionary

The environment dictionary is available at all extension points. It is strictly a set of name-value pairs in which both the name and the value are strings.

The DHCP server uses the environment dictionary to communicate with extensions in different ways at different extension points. At some extension points, the server places information in the environment dictionary for the extension to modify. In others, the extension can place values in the environment dictionary to control the flow or data after the extension finishes its processing.

The environment dictionary is unique in that an extension can put any name-value pair in it. Although you do not get an error for using undocumented name-value pairs, the server does not recognize them. These name-value pairs can be useful for your extension points to communicate data among them.

The DHCP server creates the environment dictionary when a DHCP request arrives and the dictionary remains with that request through the processing. Thus, an extension that runs at the **post-packet-decode** extension point can put data into the environment dictionary, and then an extension run at the **pre-packet-encode** extension point might read that data from the dictionary.

Note The **init-entry** extension point has a unique environment dictionary.

General Environment Dictionary Data Items

The data items in the following table are valid in the environment dictionary at all extension points. (See the individual extension point sections for environment dictionary data items specific to each one.)

The data items are input, output, or both:

- Input—The DHCP server sets the value and inputs it to the extension.
- Output—The value is output to the DHCP server, which reads it, and acts upon it. Note that as there can be multiple extensions at an extension point, it is possible that an earlier extension running at an extension point has set this and hence this can be an “input” to a later extension run at that extension point. When

the table indicates that it is not an "input", it means that the DHCP server did not explicitly set this before calling the extension(s) at that extension point.

Table 5: General Environment Dictionary Data Items

Environment Data Item	Description
drop (input[1]/output)	If the *drop* value is equal to the string **true** when the extension exits, the DHCP server drops the input packet and logs a message in the log file. Initially set to **false**. Available at most extension points, but not all (such as **generate-lease**). **Note** For recommendations on how to use *drop* for multiple extensions per extension point, see Multiple Extension Considerations, on page 6.
extension-name (input)	Name with which the extension was configured. You can configure the same piece of code as several different extensions and at several different extension points. This allows one piece of code to do different things, depending on how you configure it. The code can also use this string to find itself in the extension-name-sequence string, for which it needs to know its own name.
extension-name-sequence (input)	Provides a comma-separated string representing the configured extensions for this extension point. It allows an extension to determine which extensions can run before and after it. The *extension-name* data item provides the currently running extension. For example, if you configure **tclfirst** as the first extension and **dexscript** as the fifth, the *extension-name-sequence* would contain **"tclfirst,,,,dexscript"**.
extension-point (input)	Name of the extension point. For example, **post-packet-decode**.
extension-sequence (input)	String that is the sequence number of the extension at the extension point.
giaddr-override (output)	This data item may be set by a **pre-packet-decode**, **post-packet-decode**, **pre-client-lookup**, **post-client-lookup**, and **post-class-lookup** extension to specify an IPv4 address or scope name to be used in determining the network location of the client (instead of the giaddr or received interface's address). This is only used for DHCPv4 requests (ignored for DHCPv6). If a scope name is specified, it is only used to determine the client's location and does not mean that the client will necessarily get a lease from that scope.
link-address-override (output)	This data item may be set by a **pre-packet-decode**, **post-packet-decode**, **pre-client-lookup**, **post-client-lookup**, and **post-class-lookup** extension to specify an IPv6 address or prefix name to be used in determining the network location of the client (instead of the Relay-Forw's link-address or received interface's address). This is only used for DHCPv6 requests (ignored for DHCPv4). If a prefix name is specified, it is only used to determine the client's location and does not mean that the client will necessarily get a lease from that prefix. See Determining Links and Prefixes.

Environment Data Item	Description
log-drop-message (output)	If the *drop* value is equal to the string **true,** and the *log-drop-message* value is equal to the string **false** when the extension exits, then the DHCP server drops the input packet, but does not log a message in the log file. Does not apply to **init-entry**.
release-by-ip (output)	For this to be effective, it must be set by an extension called at the **pre-packet-decode**, **post-packet-decode**, **pre-client-lookup**, **post-client-lookup**, or **post-class-lookup** extension point. It applies to DHCPRELEASE requests only. If set to **true**, instructs the server to release the lease by the IP address if it cannot retrieve the lease by the *client-id* as derived from the DHCPRELEASE request.
trace-level (output)	Setting this to a number makes that number the current setting of the *extension-trace-level* server attribute for all extensions processing this request.
user-defined-data (output)	Set with the *user-defined-data* attribute of a lease stored with the lease before request processing. You can have it written to disk before (but not with) a **pre-packet-encode**. If set to null, the server ignores the *user-defined-data* from the lease. You cannot remove a previous value by using a null string value. Appropriate for responses only. When the server writes the *user-defined-data* to a lease, the read-only *client-user-defined-data* response dictionary data item assumes its value. **Note** Be careful in using this data item in multiple extensions for an extension point. See Multiple Extension Considerations, on page 6.

[1] For all but **post-client-lookup** and **post-class-lookup**, *drop* is only an output. For **post-client-lookup** and **post-class-lookup**, the server sets *drop* to **false** if the specified *client-class* exists; and **true** if the *client-class* does not exist (and hence the server will not continue processing this packet unless the extension changes *drop* to **false**).

Initial Environment Dictionary

You can configure an extension with *init-args* and **init-entry**. Alternatively, you can specify configuration information for an extension to read out of the environment dictionary. You can set the DHCP property *initial-environment-dictionary* with a series of attribute-value pairs, and each pair is available in every environment dictionary. Using this capability, you can specify a variety of configuration and customizing information. Any extension can simply read this data directly out of the environment dictionary, without having to store it in some static data area, as is required with the *init-args* or **init-entry** approach.

You can read the values defined using the *initial-environment-dictionary* approach from any environment dictionary. You can also define new values for any attributes that appear in the *initial-environment-dictionary*. These new values are then available for the life of that environment dictionary (usually the life of the request packet being processed). However, this does not change the contents of any other environment dictionary. Any new environment dictionary (associated with a different request) sees the attribute-value pairs defined by the *initial-environment-dictionary* property of the DHCP server.

In addition, these *initial-environment-dictionary* attribute-value pairs do not appear in an enumeration of the values of the environment dictionary. They are only available if you request an attribute value not currently

defined in the environment dictionary. The attribute-value pairs do not actually appear in the environment dictionary. Thus, if you define a new value for one of the attributes, that new value does appear in the environment dictionary. If you later delete the value, the original one is again available if you should request it.

Request and Response Dictionaries

The request and response dictionaries have a fixed set of accessible names. However, you cannot access all names from every extension point. These dictionaries make internal server data structures available to the extension for read-write or, in some cases, read-only access. Each data item has a particular data type. If you omit the correct data type (for C/C++ extensions) on a put operation, or if the DHCP server cannot convert it to the correct data type (for Tcl extensions), the extension will fail.

The request dictionary is available at the beginning of the processing of a request. After the server creates a response, both the request and response dictionaries are available. It is an error to access a response dictionary before it is available.

In general, you cannot use an extension to change information data in the server. In some cases, however, you can use an extension to change configured data, but only for the duration of the processing for just that single request.

Appendix B contains details on the options and data items available for the received client request (the Request Dictionary) and for the response sent (the Response dictionary).

Decoded DHCP Packet Data Items

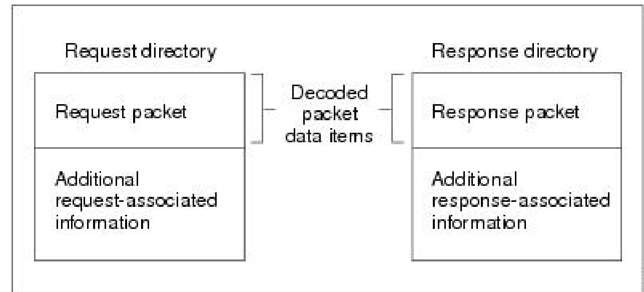
The DHCP protocol is a request-response UDP-based protocol and, thus, the stimulation for a DHCP server operation is usually a DHCP request from a client. The result is usually a DHCP response sent back to that client.

The DHCP extension facility makes the information input in the DHCP request available to extensions at most of the extension points, and the information to be sent as a response to a DHCP request available at the **pre-packet-encode** extension point (see pre-packet-encode, on page 38).

In addition to this DHCP packet-based information, there is additional data that the DHCP server uses when processing DHCP requests. This data becomes associated with either the DHCP request or the DHCP response as part of the architecture of the server. Much of this data is also made available to extensions, and much of it can be both read and written—in many cases altering the processing algorithms of the DHCP server.

The request and response dictionaries, therefore, contain two classes of data in each dictionary. They contain decoded packet data items as well as other request or response associated data items. The decoded packet data items are those data items directly contained in or derived from the DHCP request or DHCP response. Access to the decoded packet data items allows you to read and, in some cases, rewrite the DHCP request and DHCP response packet. The following figure shows the relationship between the request and the response dictionaries.

Figure 1: Extensions Request and Response Dictionaries



You can access information from the DHCP request packet, such as the *giaddr*, *ciaddr*, and all the incoming DHCP options by using the decoded packet data items in the request dictionary. Similarly, you can set the *giaddr* and *ciaddr*, and add and remove DHCP options in the outgoing DHCP response by accessing the decoded packet data items in the response dictionary.

It is important to realize that access to the packet information provided by the decoded packet data items is not all available to you. The specific data items available to that extension point are listed in the description of each extension point. Because the decoded packet data items are always accessible as a group, they are listed as a group.

You access DHCP options by name. If the option is not present, the server returns no data for that option. If you place an option into the decoded request or decoded response, it replaces any option with the same name already in the decoded request or decoded response, unless, in the put operation, you want the data specifically appended to existing data.

Some DHCP options can have multiple values. For example, the routers option can have one or more IP addresses associated with it. Access to these multiple values is by indexed operations on the option name.

Tip A **clear** operation on the request or response dictionary removes all the options in the decoded packet.

Using Parameter List Option

There is one option, *dhcp-parameter-request-list*, that the DHCP server specially handles in two ways, available as either a:

- Multiple-valued option of bytes under the name *dhcp-parameter-request-list*.
- Blob (sequence of bytes) option under the name *dhcp-parameter-request-list-blob*.

You can get or put the option using either name. The DHCP server handles the *dhcp-parameter-request-list* (and its *-blob* variant as well) differently in the response dictionary than in the request dictionary. When you access this option in the request dictionary, it is just another DHCP option in the request dictionary. In the response dictionary, however, special processing takes place.

You can use the *dhcp-parameter-request-list* option in the response dictionary to control the order of the options returned to the DHCP or BOOTP client. When you put the option in the response dictionary, the DHCP server reorders the existing options so that the ones listed in the option are first and in the order that they appear in the list. Then, the remaining options appear in their current order after the last ones that were in the list. The DHCP server retains the list, and uses it to order any future options that it puts into the response, until it replaces the list with a new one.

When an extension does a get operation for the *dhcp-parameter-request-list* in the response dictionary, it does not look in the decoded response packet to find an option. Instead, the DHCP server synthesizes one that contains the list of all options currently in the decoded response packet.

Extension Point Descriptions

The following sections describe each extension point, their actions, and data items. For all the extension points, you can read the **extension-point** and set the *trace-level* data item values in the environment dictionary. For most extension points, you can also tell the server to drop the packet.

init-entry

The **init-entry** extension point is an additional one that the DHCP server calls when it configures or unconfigures the extension, which occurs when starting, stopping, or reloading the server. This entry point has the same signature as the others for the extension, but you can use only the environment dictionary. You do not configure the **init-entry** extension with **dhcp attachExtension** in the CLI, but you do so implicitly by defining an **init-entry** on an already configured extension.

Note You must supply an **init-entry** extension point to enable extension points for DHCPv6 (or disable them for DHCPv4).

In addition to configuring an **init-entry** with the name of the entry point, you can also configure a string of arguments that the DHCP server loads in the environment dictionary under the string *arguments* before calling the **init-entry** point. Using *arguments*, you can create a customized extension by giving it different initialization arguments and thus not require a change to the code to elicit different behavior.

Note The order in which the server calls extensions at the **init-entry** extension point can be different from reload to reload, or release to release.

Caution An extension, when called to uninitialize, must terminate any threads it creates and clean up after itself before returning. Once the extension returns, the DHCP server unloads the extension from memory, which could result in a server failure if a thread an extension created is left running.

Environment Dictionary for init-entry

See the following table for the environment dictionary data items specific to **init-entry**.

Table 6: init-entry Environment Dictionary Data Items

Environment Dictionary Data Item	Description
dhcp-support (input/output)	Version or versions of DHCP for which the server should call the registered extension points for the extension. Can be **v4**, **v6**, or **v4,v6**.
exiting-state (output)	For an extension attached to the **lease-state-change** extension point, if specified, the **lease-state-change** extension point is called only if the current state of the lease is the state specified by *exiting-state*.The extension is only called when the specified state is exited. If not specified, and the extension is attached to the **lease-state-change** extension point, the extension will be called for all state changes. If specified, the *exiting-state* must specify a valid lease state: available, offered, leased, expired, unavailable, released, other-available, pending-available, revoked. **Note:** There is no strict state transition table. In a failover environment, the server that receives a binding update message sets the state to whatever its partner informs it to be, without requiring specific state transitions.
extension-extensionapi-version (output)	Minimum version of the extension API required by the extension. Set it to **2** as that is the current API version.
init-args (input)	Configure arguments by setting *init-args* on an existing extension point. These arguments are present for both the configure and unconfigure calls of the **init-entry** entry point. The extension point name for the configure call is **initialize**, and for the unconfigure call is **uninitialize**.
server-dhcp-support (input)	The server sets this data item to indicate what the server is configured to support. Can be **v4**, **v6**, or **v4,v6**, depending on the DHCP server *dhcp-support* attribute setting (which requires setting expert attribute visibility=3) and whether any prefixes are configured: • If *dhcp-support* =**both** and prefixes are not configured, then *server-dhcp-support* is set to **v4**. • If *dhcp-support* =**both** and one or more prefixes are configured, then *server-dhcp-support* is set to **v4,v6**. • If *dhcp-support* =**v4**, then *server-dhcp-support* is set to **v4**. • If *dhcp-support* =**v6** and one or more prefixes are configured, then *server-dhcp-support* is set to **v6**.
server-extensionapi-version (input)	Version of the server extension API. The value is **2**.

pre-packet-decode

The dictionaries available for **pre-packet-decode** are request and environment.

This extension point is the first one the DHCP server encounters when a request arrives. The server calls it after receiving a packet but before it decodes the packet (at the **post-packet-decode** extension point). An

extension can use this extension point to examine a packet and alter it before the server decodes it, or cause the server to drop it.

Two key data items in the request dictionary are for use with **pre-packet-decode** are *client-packet* and *packet*. These can be used to examine the received packet, modify the packet, and write it back.

Caution The request dictionary *client-packet* and *packet* data items used for **pre-packet-decode** are available at any extension point that has a request dictionary. However, you should not directly alter or replace the packet at any extension point other than **pre-packet-decode**, because doing so can have unexpected side effects. For example, the server might never pick up the changes to the packet, or options data can change unexpectedly during processing.

An extension that uses **getBytes** with client-packet or packet directly alters the bytes of packet by writing into the returned buffer. However, an extension must use **put** or **putBytes** to adjust the length of the packet (and the operation can fail if the packet is too big). For DHCPv6, adjusting the length of the client portion of the packet, if relayed, requires updating the lengths in the Relay Message options in the packet.

It is up to an extension to handle parsing the packet to locate what it needs and properly alter the packet, if that is the intent.

Because the server has not yet decoded the received packet, most request dictionary data items are not available (as the server normally fills them in from the received packet). Thus, this extension point must extract data directly from the packet. The extension must also properly handle incorrectly formatted packets.

If you enable *incoming-packet-detail* logging, the server logs the received packet after calling the extensions registered at this extension point. If DHCP server debug tracing is configured with V is 3 or more, the server also logs the packet before calling the extensions registered for this extension point, if at least one extension is registered.

Caution This extension gets access to the received packet before it has been validated in any way. Therefore, the extension must be written to handle completely or partially invalid DHCP packets.

post-packet-decode

The dictionaries available for **post-packet-decode** are request and environment.

Extension Description

This extension point immediately follows the decoding of the input packet and precedes any processing on the data in the packet. The primary activity for an extension at this point is to read information from an input packet and do something with it. For example, you might use it to rewrite the input packet.

The **post-packet-decode** extension point is one of the easiest extension points to use. If you can express the change in server behavior as a rewrite of the input DHCP or BOOTP packet, you should use this extension point. Because the packet was decoded, but not processed in any way, the number of side effects are very limited.

The **post-packet-decode** extension point is the only one at which you can modify the decoded input packet and ensure that the server recognizes all the modifications. You can have the extension drop the packet and terminate further processing by using the *drop* data item in the environment dictionary.

Overriding Client Identifiers

To override client identifiers (IDs), you can set an expression value for the *override-client-id* attribute for a client-class or use the *override-client-id* data item at the **post-packet-decode** extension point. The extension method maps the client to a different identifier than the server.

There is a variant of the extension data item where you can get or put the override client ID as a string: *override-client-id-string*. You can also request the data type of the override client ID through the read-only *override-client-id-data-type* data item.

Different values are returned based on how you put and get the *override-client-id* or its *override-client-id-string* variant (see the following table for some examples).

Table 7: Puts and Gets of Client ID Overrides

Action	Data Item Used	Put Value	Get Value
put	*override-client-id*	01:02:03:04	
putBytes	*override-client-id*	01 02 03 04	
get	*override-client-id*		01:02:03:04 (blob)
getBytes	*override-client-id*		01 02 03 04 (raw bytes)
get [**Bytes**]	*override-client-id-string*		01:02:03:04 (blob-as-string)
get [**Bytes**]	*override-client-id-data-type*		blob

Table 8: Puts and Gets of Client ID Overrides

Action	Data Item Used	Put Value	Get Value
put [**Bytes**]	*override-client-id-string*	01:02:03:04 test	
get [**Bytes**]	*override-client-id-string*		01:02:03:04 test (string)
get [**Bytes**]	*override-client-id*		30:31:3a:30:32:3a:30:33:3a:30:34:74:65:73:74 (blob of “01:02:03:04 test”)
get [**Bytes**]	*override-client-id-data-type*		nstr

The equivalent *client-override-client-id* data items (that you can use in later extension points where the response dictionary is valid) function the same way, although they are read-only.

Note When using *[v6-]override-client-id* expressions, leasequery by *client-id* requests may need to specify the *override-client-id* attribute to correctly retrieve the information on the lease(s) for the client.

Caution This extension is called after the server has parsed the packet syntactically, but before any validation has been applied. Therefore, the extension must be written to handle potentially invalid packets.

Environment Dictionary for post-packet-decode

See the following table for the environment dictionary data items specific to **post-packet-decode**.

Table 9: post-packet-decode Environment Dictionary Data Items

Environment Dictionary Data Item	Description
cnr-forward-dhcp-request (input) *cnr-request-forward-address-list* (output)	Both these data items are for DHCPv4 only. If *cnr-forward-dhcp-request* is set to **true** when the extension returns, the *cnr-request-forward-address-list* must contain the (comma separated) list of IPv4 addresses (and optionally port number) to which the server should forward the request. Once forwarded, the server drops the request. Each entry in the comma separated list may be *ipv4-address* or *ipv4-address*:*port-number* (if no port number specified, the default dhcp server port is used). For more information, see Setting DHCP Forwarding.

post-class-lookup

The dictionaries available for **post-class-lookup** are request and environment.

The server calls this extension point only if there is a *client-class-lookup-id*; otherwise, it is similar to a **post-packet-decode**. The server calls the **post-class-lookup** extension point after evaluating the *client-class-lookup-id* and setting the *client-class* data for this client.

On input to this extension point, the environment dictionary has the *drop* data item set to **true** or **false**. You can change this setting by extension to drop the packet (or not drop it), and the server recognizes the change. The server also looks at the *log-drop-message* to decide whether to log the drop.

Environment Dictionary for post-class-lookup

See the following table for the environment dictionary data item specific to **post-class-lookup**.

Table 10: post-class-lookup Environment Dictionary Data Item

Environment Dictionary Data Item	Description
client-class-name (output)	Sets the named *client-class* for the packet, regardless of the previous *client-class*. This setting has an effect only if the *drop* environment dictionary data item value is **false** on exiting the extension point.

pre-client-lookup

The dictionaries available for **pre-client-lookup** are request and environment.

You can use this extension point only if you enabled client-class processing for your DHCP server. This extension point allows an extension to perform any or all of these actions:

- Modify the client that the server looks up during client-class processing.

- Specify individual data items to override those found from the client entry or client-class it specifies.
- Instruct the server to skip the client lookup altogether. In this case, the only client data used is data that the extension supplied in the environment dictionary.

Although the request dictionary is available to make decisions about the operation of an extension running at this extension point, the environment dictionary controls all the operations.

Environment Dictionary for pre-client-lookup

The environment dictionary data items in the table below are the control data items available at **pre-client-lookup** for clients and client-classes.

If you set the environment dictionary data items in Table 12: pre-client-lookup Environment Dictionary Override Data Items, their values override those determined from the client lookup (either in the internal database or from LDAP). If you do not add anything to the dictionary, the server uses what is available in the client lookup.

Table 11: pre-client-lookup Environment Dictionary Control Data Items

Environment Dictionary Data Item	Description
client-specifier (input/output)	Name of the client the client-class processing code looks up, in CNRDB or LDAP. If you change the name at this extension point, the DHCP server looks up the client you specify.
default-client-class-name (output)	Instructs the server to use the value associated with the *default-client-class-name* option as the *class-name* if: • The *client-specifier* data item was not specified in the **pre-client-lookup** script. • The server could not locate the specific client entry. The *default-client-class-name* data item then assumes precedence over the *class-name* associated with the default client.
skip-client-lookup (input/output)	The value is determined by the server configuration. If set to **true**, the DHCP server skips the normal client lookup that it would have performed immediately upon exit from this extension. The only data items used to describe this client are those placed in the environment dictionary (see the table below).

Table 12: pre-client-lookup Environment Dictionary Override Data Items

Environment Data Item	Description
action (output)	Convert this string to a number and use the result as the action. The numbers you can use are **0** (for none) and **1** (for exclude).

Environment Data Item	Description
authenticate-until (output)	Absolute time, measured in seconds, from January 1, 1970. Use to indicate the time at which the client authentication expires. When the client authentication expires, the DHCP server uses the values in the client *unauthenticated-client-class* option instead of its client-class to fill in missing data items in the client entry.
client-class-name (output)	Use the client-class specified by this data item to fill in the missing information in the client entry. If there is no client-class corresponding to the name specified, the DHCP server logs a warning and continues processing. If you specify **none**, the DHCP server acts as if the client entry did not include the client-class name.
domain-name (output)	Use this domain name for the client DNS operations in preference to the one specified in the DNS update configuration. The DNS server shown as the primary server for the domain in the scope or prefix must also be the primary server for the domain you specified. If the domain name is not overridden in the client or client-class entry, the DHCP server uses the domain name from the scope or prefix. If the client entry or the extension contains the word **none**, the DHCP server uses the domain name from the scope or prefix.
host-name (output)	Use this for the client in preference to the host-name options specified in the input packet, or any data from the client or client-class entry. If you set this to **none**, the DHCP server does not use any information from the client or client-class entry, but uses the name from the client request.
policy-name (output)	Use this policy as the policy specified for the client entry, overriding any policy specified by that client entry.
selection-criteria (output)	List of comma-separated strings, each specifying (for this input packet) the selection criteria for the client. Any scope or prefix the client uses must have all of these selection tags. Use this data item to override any criteria specified in the client or client-class entry. If you do, the DHCP server does not use the client entry selection criteria, independent of whether they were stored in the local or LDAP database. If you set this data item to **none**, the DHCP server does not use selection tags for the packet. If you set this to a null string, the DHCP server treats it as if it were not set and uses the selection criteria from the client or client-class entry.
unauthenticated-client-class-name (output)	Name of the client-class to use if the server does not authenticate the client. If you want to indicate without specifying the *unauthenticated-client-class-name*, use an invalid client-class name as the value of this data item. You can use the value **none** or any name that is not a client-class name. The DHCP server logs an error that the client-class is not present.

post-client-lookup

The dictionaries available for **post-client-lookup** are request and environment.

You can use this extension point to examine the results of the entire client-class processing operation, and take an action based on those results. You might want to use it to rewrite some of the results, or to drop the packet. If you want to override the hostname in the packet returned from the client-class processing from an extension running at the **post-client-lookup** extension point, set the hostname to the *client-requested-host-name* data item in the request dictionary. This causes Cisco Prime Network Registrar to look to the server as though the packet came in with whatever string you specified in that data item.

You also can use this extension point to place some data items in the environment dictionary to affect the processing of an extension running at the **pre-packet-encode** extension point (see pre-packet-encode, on page 38), where it might load different options into the response packet or take other actions.

Environment Dictionary for post-client-lookup

See the following table for the environment dictionary data items specific to **post-client-lookup**.

Table 13: post-client-lookup Environment Dictionary Data Items

Environment Dictionary Data Item	Description
client-specifier (input)	Name of the client that the client-class processing looked up.
cnr-ldap-query-failed (input)	The DHCP server sets this attribute to ease recovery from LDAP server failures so that a post-client-lookup script can respond to an LDAP server failure. The DHCP server, after a client lookup, sets this flag to **true** if the LDAP query failed because of an LDAP server error. If the server received a response from the LDAP server, one of two conditions occurs: • It sets the flag to **false**. • The *cnr-ldap-query-failed* attribute does not appear in the environment dictionary.

generate-lease

The dictionaries available for **generate-lease** are request, response, and environment. This extension point is available for DHCPv6 only.

You can use this extension point to generate a DHCPv6 address or prefix and allow the extension to control the address or prefix. When the extension returns a *generated-address* value, the server relaxes many restrictions on the address or prefix returned as it assumes the extension is in control for leasing activities. This includes failover constraints (hence, an odd address can be assigned by the backup, an even address can be assigned by the main, and a delegated prefix that is in other-available can be assigned). Extensions are thus responsible for managing the address or prefix delegation space.

The server calls **generate-lease** only if the prefix is configured to allow extensions to be called during address allocation or prefix delegation—the extension flag must be set in the *allocation-algorithms* attribute for the prefix. When the server calls the generate-lease extension:

- The server sets the prefix context for the response dictionary to the prefix on which the lease is to be created. (Calling **setObject** with DEX_PREFIX and DEX_INITIAL will return to this context.)
- No lease context exists, because the server has not yet created a lease. However, lease-binding data items, in particular *lease-binding-type* and *lease-binding-iaid* are available. (Calling **setObject** with DEX_LEASE and DEX_INITIAL returns to this context and also sets the prefix, because a lease context sets three contexts: lease, binding, and prefix.)
- The server sets the *skip-lease* environment dictionary data item to false.
- The server sets the (read-only) *attempts* environment dictionary data item with the number of times (starting with 1) it called the extension to create this lease.
- For prefix delegation, the following environment dictionary data items are available:
    - **prefix-length**—Prefix length (requested or default prefix length).
    - **default-prefix-length**—Default prefix length (from policies).
    - **longest-prefix-length**—Longest allowable prefix (from policies).
    - **shortest-prefix-length**—Shortest allowable prefix (from policies).

When the extension returns, it can:

- Request an explicit address (for stateful address assignment) by setting the address on the *generated-address* environment dictionary data item. If the address is not available for the client (that is, if the address is already in use) or is not contained by the prefix, the server might call this extension again.
- Request an explicit prefix (for prefix delegation assignment) by setting the prefix on the *generated-prefix* environment dictionary data item. If the prefix is not available for the client or is not contained by the prefix, the server might call this extension again. The prefix is not available for the client under the following conditions:
    - if the prefix is already in use
    - if it is contained in a shorter prefix that has already been delegated
    - if a longer prefix contained in it has already been delegated by the server

  The server will not reject the prefix if it is shorter or longer than allowed by the policy.

    - Cause the server not to assign a lease for this prefix by setting the *skip-lease* environment dictionary data item to true. The server will advance to the next prefix (if any).
    - Allow normal address assignment or prefix delegation to occur by not setting any of the above.

The server calls the extension point at most 500 times for each lease (this limit is the same one that currently applies when the server randomly generates leases). The server calls an extension multiple times only if the extension supplies an unusable address or delegated prefix (that is not in range for the prefix or already exists).

Note You cannot request the server to drop the packet at this extension point.

Environment Dictionary for generate-lease

See the following table for the environment dictionary data items specific to **generate-lease**.

Table 14: generate-lease Environment Dictionary Data Items

Environment Dictionary Data Item	Description
attempts (input)	Number of times that the server calls this extension for a single lease.
default-prefix-length (input)	Specifies the default prefix length to be used for allocating a delegated prefix. Set to the *default-prefix-length* (from the policy hierarchy).
generated-address (output)	Address the extension wants the server to use for the lease.
generated-prefix (output)	Delegated DHCPv6 prefix the extension wants the server to use for the lease.
limit-to-prefix-length (input)	Set to **true** if the server is requesting the extension to limit any generated prefix to the client's requested prefix length of *prefix-length*; **false** otherwise. If the client requested a prefix length, the server first calls the **generate-lease** extension attempting to get a delegated prefix of that length. Note that the server bounds the client's requested length to be between the *shortest-prefix-length* and *longest-prefix-length*.
longest-prefix-length (input)	Specifies the longest prefix length to be used for allocating a delegated prefix. Set to the (expert mode) *longest-prefix-length* (from the policy hierarchy) – defaults to *default-prefix-length* if not configured.
prefix-length (input)	Set to the requested or default prefix length.
shortest-prefix-length (input)	Specifies the shortest prefix length to be used for allocating a delegated prefix. Set to the (expert mode) *shortest-prefix-length* (from the policy hierarchy) – defaults to *default-prefix-length* if not configured.
skip-lease (output)	Set to **true** if the extension does not want the server to generate the lease.

check-lease-acceptable

The dictionaries available for **check-lease-acceptable** are request, response, and environment.

This extension point comes immediately after the server determines whether the current lease is acceptable for this client. You can use this extension to examine the results of that operation, and to cause the routine to return different results. See Determining Lease Acceptability, on page 15.

Environment Dictionary for check-lease-acceptable

See the following table for the environment dictionary data item specific to **check-lease-acceptable**.

Table 15: check-lease-acceptable Environment Dictionary Data Item

Environment Dictionary Data Item	Description
acceptable (input)	A read/write data item that the DHCP server initializes depending on if the lease is acceptable for this client. You can read and change this result in an extension. Setting the acceptable data item to **true** indicates that it is acceptable; setting it to **false** indicates that it is unacceptable.
default-prefix-length (input)	Specifies the default prefix length to be used for allocating a delegated prefix. Set to the *default-prefix-length* (from the policy hierarchy).
longest-prefix-length (input)	Specifies the longest prefix length to be used for allocating a delegated prefix. Set to the (expert mode) *longest-prefix-length* (from the policy hierarchy) – defaults to *default-prefix-length* if not configured.
prefix-length (input)	Specifies the client's requested prefix length if the client provided one or 0 if not.
shortest-prefix-length (input)	Specifies the shortest prefix length to be used for allocating a delegated prefix. Set to the (expert mode) *shortest-prefix-length* (from the policy hierarchy) – defaults to *default-prefix-length* if not configured.

lease-state-change

The dictionaries available for **lease-state-change** are response and environment.

The existing state is in the *lease-state* response dictionary data item. The new state is in the environment dictionary data item *new-state*. The server never calls the extension point if the new state matches the existing one.

Use this extension point mainly for read-only purposes, although you can place data in the environment dictionary so that other extension points can get it later.

The **lease-state-change** can also have a different environment dictionary, such as for lease expirations.

Environment Dictionary for lease-state-change

See the following table for the environment dictionary data items specific to **lease-state-change**.

Table 16: lease-state-change Environment Dictionary Data Items

Environment Dictionary Data Item	Description
new-start-time-of-state (input)	The start time of the new state. The previous state's start time is in the *lease-start-time-of-state* information data item in the response dictionary.
new-state (input)	The state the lease is being changed to. The current state is in the *lease-state* lease information data item in the response dictionary.

pre-packet-encode

The dictionaries available for **pre-packet-encode** are request, response, and environment.

Note For DHCPv6 Reconfigure messages, there is no request dictionary (because Reconfigure is a server-initiated message). Thus, enabled extensions should check the response *msg-type* for ADVERTISE or REPLY or use **isValid** on the request to ensure that the Reconfigure message exists.

post-packet-encode

The dictionaries available for **post-packet-encode** are request, response, and environment.

Note For DHCPv6 Reconfigure messages, there is no request dictionary (because Reconfigure is a server-initiated message). Thus, enabled extensions should check the response *msg-type* for ADVERTISE or REPLY or use **isValid** on the request to ensure that the request dictionary exists.

The server calls this extension point after encoding a packet, but before sending it to the client. The server can thereby examine and alter the packet before it sends the packet to the client, or the extension can cause the server to drop the packet (although the server might have made changes to its internal and on-disk data that will not be backed out if the packet is dropped).

The *client-packet* and *packet* data items were added to the response dictionary with similar behavior as described for the request dictionary in pre-packet-decode, on page 28. Note that this extension point is the only one that can request the response *client-packet* or *packet* data items, because no packet exists at any other extension point. Also, the server does not process the changes made to the packet; the server simply sends the altered packet to the client.

If you enable outgoing-packet-detail logging, the server logs the packet after calling the extensions registered at this extension point. If DHCP server debug tracing is configured with X>=3, the server also logs the packet before calling the extensions registered for this extension point, but only if at least one extension is registered.

post-send-packet

Use the **post-send-packet** extension point for any processing that you want to perform outside of the serious time constraints of the DHCP request-response cycle. After the server sends the packet to the client, it calls this extension point.

Note For DHCPv6 Reconfigure messages, there is no request dictionary (because Reconfigure is a server-initiated message). Thus, enabled extensions should check the response *msg-type* for ADVERTISE or REPLY or use **isValid** on the request to ensure that the request dictionary exists.

environment-destructor

The **environment-destructor** extension point allows an extension to clean up any context that it might be holding. The only dictionary available for this extension point is environment.

The environment dictionary is available for all extension points called for a single client request. Because some extensions may need to maintain context information between the multiple extension points called for a single client request, and because the server might drop requests at several places during processing, an extension cannot reliably release context that it might have created for that request. The environment-destructor extension point now makes it possible to reliably remove this context when processing of a request has completed, for whatever reason.

Note The server calls all extensions attached to the **environment-destructor** extension point, even if the server did not call each extension at any other attachment point.