



## Using Extension Points

---

You can write extensions to affect how Cisco CNS 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.

## Creating Extensions

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

You must complete these procedures 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's 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.

## Determining Tasks

The task to which to apply an extension is usually some modification of the DHCP server's processing so that it better meets the needs of your environment. You can apply an extension at each of these DHCP server's 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
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

These steps are described in detail in the “[DHCP Request Processing Using Extensions](#)” section on [page 17-9](#).

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 to be 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

There are often many solutions 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 the “[Determining Tasks](#)” section on [page 17-1](#), 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 return from the DHCP server. The second approach involves rewriting the packet, in this case using the **post-send-packet** extension point (see the “[Sending Packets](#)” section on [page 17-15](#)). 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 a bit easier for some 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 crashing the server.
- C/C++—This language provides the maximum possible performance and flexibility, including communicating with external processes. The complexity of the C/C++ API is greater and the possibility of a bug in the extension crashing the server is more likely than with TCL, because the extension operates in the same code space as the DHCP server itself.

# 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—there can be multiple extension functions in this file. 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 is a representation of 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* operation on the same named data item.

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 attempts to reference a dictionary that is not present at a particular extension point. See the “[Environment Dictionary](#)” section on page 17-16 and the “[Request and Response Dictionaries](#)” section on page 17-18.

## 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 the information that came in the request itself. Data is string-, integer-, IP address-, and blob-valued (a sequence of bytes, not zero terminated).
- Response dictionary—Information associated with the generation of a DHCP response packet to return to the DHCP client. Data is string-, integer-, IP address-, and blob-valued (a sequence of bytes, not zero terminated).
- Environment dictionary—Information passed between the DHCP server and extension. Data is string-valued only.

For a description of the dictionaries, see the “[Extension Dictionaries](#)” section on page 17-16.

You can also use the environment dictionary to communicate between two extensions running 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 names of the allowable data items are not fixed by the DHCP server. You can use it 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 **pre-dns-add-forward**), share the same environment dictionary. Thus, an extension may 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’s 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 **pre-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.

## Init-Entry Extension Point

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

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 requiring a change to the code to elicit different behavior.

You 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**.



### Note

---

The order in which extensions are called at their **init-entry** point is not assured. It can be different from reload to reload or release to release.

---

## Configuration Errors

There are many reasons why an extension can fail. For example:

- The file may not be found.
- The entry point or **init-entry** point may not appear in the file.
- The extension itself can return failure from an **init-entry** call.

By itself, an extension failure is not fatal and does not prevent the DHCP server from starting. However, the configuration for that extension point will fail. If the DHCP server fails to configure any of its extension points, the server will not start. Therefore, to debug the configuration process, you can configure the extension and the **init-entry** point without attaching it to an extension point. When you complete that process, you can attach your extension to an extension point.

## 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 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 are debugging extensions.

## 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.

## 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 it is called, the ASCII name of the extension point is passed in the environment dictionary under the key **extension-point**.

For some sample TCL extensions, see the Network Registrar directory:

- Solaris and Linux—`/opt/nwreg2/examples/dhcp/tcl`
- Windows—`\Program Files\Network Registrar\examples\dhcp\tcl`

## 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
```

## Handling Boolean Variables

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.

## Configuring TCL Extensions

To configure an extension, write it and place it in the extensions directory. For UNIX and Linux, this is the `/opt/nwreg2/extensions/dhcp/tcl` directory. For Windows, this is the `\Program Files\Network Registrar\extensions\dhcp\tcl` directory.

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.

## Init-Entry Extension Point in TCL

TCL extensions support the **init-entry** extension point, and the arguments supplied in the **init-args** parameter to the command line appear in the environment dictionary associated with the key **arguments**.

Multiple TCL interpreters may be running in the DHCP server, for performance purposes, each in its own TCL context. The server calls the TCL extension once at the **init-entry** point for every TCL context (interpreter) it runs. Ensure that your TCL extension's **init-entry** is robust, given multiple calls.

Information cannot flow between the TCL contexts, but the **init-entry** can initialize global TCL variables in each TCL interpreter that any TCL extension can access, regardless of the interpreter.

Note that the TCL interpreters are shared among all of the TCL extensions. If your TCL extension initializes global variables or defines procedures, ensure that these do not conflict with some other TCL extension's global variables or procedure names.

## C/C++ Extensions

All DHCP C/C++ extensions are called *dex* extensions, which is 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 was specifically constructed so that you do not have to link your shared library with any 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 be performed 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

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 directories `/opt/nwreg2/examples/dhcp/dex` (UNIX) and `\Program Files\Network Registrar\examples\dhcp\dex` (Windows) contains example C/C++ extension code, as well as a short makefile designed to build the example extensions. To build your own extensions, you need to modify this file. It has sections for Microsoft Visual C++ V5.0, GNU C++, and SunPro 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. When building C/C++ extensions on Windows, remember to add your entry points to the `.def` file.

After you build the `.dll` (Windows) or `.so` (UNIX) file (all dex extensions are shared libraries), you need to move them into the `/opt/nwreg2/extensions/dhcp/dex` directory (UNIX), or the `\Program Files\Network Registrar\extensions\dhcp\dex` directory (Windows). You can then configure them.

## Using Thread-Safe Extensions

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



### Caution

All C/C++ extensions *must* be thread-safe, or the DHCP server will not operate correctly, and will crash 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. Because Windows provides different versions of libraries for multithreaded applications that are threadsafe, this problem usually does not apply.

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

Diagnosing these problems is extremely difficult, because the cause of these failures are rarely apparent. To cause a crash, 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 may not appear until after sustained periods of heavy load.

Because some runtime or third-party libraries might be making nonthread-safe calls that you cannot detect, check your executables to see what externals are being linked (nm on UNIX).

If the single threaded version of any of the functions in the following list are called by the thread (versions without the *\_r* suffix by any thread on Solaris), do not use them. The interfaces to the thread-safe versions of these library routines may be different on different operating systems.

Thread-safe versions are:

asctime_r	gethostbyname_r	getservbyport_r
ctermid_r	gethostent_r	getservent_r
ctime_r	getnetbyaddr_r	getspent_r
fgetgrent_r	getnetbyname_r	getspnam_r
fgetpwent_r	getnetent_r	gmtime_r
fgetspent_r	getprotobyname_r	lgamma_r
gamma_r	getprotobyname_r	localtime_r
getgrid_r	getprotoent_r	nis_sperror_r
getgrnam_r	getpwent_r	rand_r
getlogin_r	getrpcbyname_r	readdir_r
getpwnam_r	getrpcbynumber_r	strtok_r
getpwuid_r	getrpcent_r	tmpnam_r
getgrent_r	getservbyname_r	ttyname_r
gethostbyaddr_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 the server is stopped, the *.dll* and *.so* files are not in use and you can overwrite them 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's memory, leading to a server crash. 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

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 of these 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. It is not, however, valid in the **pre-dns-add-forward** extension point, because this extension point is not part of the cycle of request-response processing, but is handled by a different subsystem.

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, you 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 once when configuring each extension and once when unconfiguring it. The dex.h file defines two extension point values that are 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's 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 any one time. You can use the saved dictionary pointer up to the time the extension returns from the **uninitialize** call. This way, the background threads can log messages during termination.

## DHCP Request Processing Using Extensions

The 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. The stages in processing a request in the DHCP server are:

1. Receive a packet from a DHCP client.
2. Decode the packet.
3. Look up client-classes, if any, using the client-class lookup identifier.
4. Modify the client-class, including any limitation identifiers.
5. Perform client-class processing, if any.
6. Build a response container from the request.
7. Determine the network from which the request arrived.
8. Find a lease already associated with this client, if any, or locate a new lease for the client.
9. Serialize all requests associated with this lease.
10. When this request reaches the head of the serialization queue, determine if this lease is (still) acceptable for this client.
11. Gather all the data to include in the response packet.
12. Encode the response packet for transmission to the DHCP client.
13. Update stable storage, if necessary.
14. Send the packet to the DHCP client.

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

## Receiving Packets

The DHCP server receives packets on port 67 (the DHCP input port) 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 of the 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 of the information in the packet at this stage.

Use the **post-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 there is an expression configured in the *client-class-lookup-id*, it is at this stage that the expression is evaluated (see the “[Enhanced DHCP Request Processing Using Expressions](#)” section on page 13-14 for a description of expressions). The result of the expression is either <null>, or something that is 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, any *limitation-id* that is configured for that client-class is evaluated and stored with the request.

## Modifying Client-Classes

After the *client-class-lookup-id* is evaluated and the client-class set, any extension attached to the **post-class-lookup** extension point is called. 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 receives information about whether the evaluation of the *client-class-lookup-id* caused the packet to be dropped. The extension not only finds out if the packet is to be dropped, it instructs the server not to drop the packet if it wants to do so.

Also, an extension running at the **post-class-lookup** extension point can set a new client-class for the request, and information from that client-class is used instead of the information from the current client class. This is the only extension point where setting the client-class actually causes that client-class to be used 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 that is looked 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 information not specified by the client entry. When the DHCP server retrieves all the data and stored it in the various places in the server’s 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 it 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.

## Determining Networks

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

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 belongs to a single network. Some scopes 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 pointer.

The Network Registrar DHCP server determines the network to use to process a DHCP client request by:

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

## Finding Leases

Now that the DHCP server established the network, it searches the hash table held at the network level to see if this client's *client-id* is *already known* to this network. Already known, 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 will appear 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 in the network. If it finds a reserved lease, the server checks whether the scope and lease are both acceptable. The following must be true of the reserved lease and the scope that contains it:

- The lease must be available (not leased to another DHCP client).
- The scope must support the request type (BOOTP or DHCP).
- The scope must not be de-activated.
- The lease must not be de-activated.
- The scope-selection tags must contain all of the client's *selection-criteria* and none of the client's *selection-criteria-excluded*.
- The scope must not be renew-only.

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 associated with this network in round-robin order, looking for one that is acceptable for this client and also has available addresses. An acceptable scope has these characteristics:

- If the client has *selection-criteria* associated with it, the scope-selection tags must contain all of the client's inclusion criteria.
- If the client has *selection-criteria-excluded* associated with it, the scope-selection tags must contain none of the client's exclusion criteria.

- The scope must support the client's request type—If the client's request is a DHCPREQUEST, the scope must be enabled for DHCP. Likewise, if the request is a BOOTP request, the scope must be enabled for BOOTP and dynamic BOOTP.
- It must not be renew-only.
- It must not be de-activated.
- It must have an available address.

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

## Serializing Lease Requests

Because multiple DHCP requests can be in process simultaneously for one client and lease, they must be serialized at the level of the lease. They are queued on the lease and processed in order of queuing.

## Determining Lease Acceptability

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 may have changed since the lease was granted and needs to be checked 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 de-activated.
- The scope must not be de-activated.
- If the request is BOOTP, the scope must support BOOTP. If the request is DHCP, the scope must support DHCP.
- If the client has any *selection-criteria*, the scope-selection tags must contain all of the client's inclusion criteria.
- If the client has any *selection-criteria-excluded*, the scope-selection tags must contain none of the client's exclusion criteria.
- If the client previously associated with this lease is not the current client, the scope must not be renew-only.

If the reserved lease meets all of this 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 de-activated.
- The scope must not be de-activated.
- If the request is BOOTP, the scope must support BOOTP. If the request is DHCP, the scope must support DHCP.
- If the client does not have a reservation for this lease, and the request is BOOTP, the scope must support dynamic BOOTP.

- If the client does not have a reservation for this lease, no other client must have a reservation for this lease either.
- If the client has any *selection-criteria*, the scope-selection tags must contain all of the client's inclusion criteria.
- If the client has any *selection-criteria-excluded*, the scope-selection tags must contain none of the client's exclusion criteria.
- If the client previously associated with this lease is not the current client, the scope must not be renew-only.

At this point in the DHCP server processing, you can use the **check-lease-acceptable** extension point. You can use this 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 DHCP OFFER.
- 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.

One reason for taking extreme care with the **check-lease-acceptable** extension point is that, if the answer returned by the extension point 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, 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.

## 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 the DHCP response should be sent.



### Caution

Any packets that you drop at the **pre-packet-encode** extension point, whether they be DHCP or BOOTP packets, still show the address to be leased in the 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 the “[Processing DNS Requests](#)” section.

## 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.

## Sending Packets

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 DHCP server sends the packet to the DHCP client, it calls this extension point.

If it takes a long time to connect to the external environment, the extension should use a separate thread for improved performance. The DHCP server has only a limited number of threads for request processing, and if some or (worse) all of them are stalled in an extension waiting on some external condition, the DHCP server’s performance suffers. There are no hard guidelines for how long is too long, but in general, if the extension completes within two or three seconds it should not impact the performance of the DHCP server. More than three seconds would definitely be too long. Thus, you should structure the extension to add a request to an internal queue in the extension code and immediately return. Use a separate thread owned by the extension to process this queue.

You can create a separate thread in the initialization call to the C/C++ extension’s *init-entry* routine. Remember to destroy the thread on the corresponding **init-entry uninitialization** call.

When adding a request to an internal queue for processing later, copy the data returned by dictionary **get** requests, because any pointers returned in C/C++ are typically invalid by the time the thread processing the queue runs.

## Processing DNS Requests

The DHCP server does this to processes DNS work item requests:

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.
2. At this point, use the **pre-dns-add-forward** extension point to alter the name used for the DNS forward (A record) request.
3. Attempts 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 the next task, which is to update the reverse record.
4. Attempts to add the name, asserting that the DHCP server should supply it—The DHCP server attempts to add the host name, asserting that it exists and that it has the same TXT record as the one that was sent. If this succeeds, the server proceeds to the next task, which is to update the reverse record. If it fails, the server checks if it exhausted its naming retries. If it did, it exits and logs an error. If not, it returns to the first step, which is to build up a name for the A record.

- 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 **lease-state-change** extension point is called whenever a lease changes state. The existing state is in response dictionary lease-state data item. The new state is in environment dictionary under *new-state*. The new-state never equals the existing state (the extension is not called if this is the case). This extension should be considered read-only—you should not make modifications to any dictionary items, because it is called in a wide variety of places in the server. This extension point is used only for tracking changes to a lease's state.

## Extension Dictionaries

Every extension is defined as a routine with three arguments. These arguments represent the request dictionary, response dictionary, and environment dictionary. Not every dictionary is available to every extension. Table 17-1 shows the extensions points and the dictionaries that are available to them.

**Table 17-1 Extensions Points and Dictionaries**

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

Each of the three dictionaries consists of name-value pairs. The environment dictionary, which is available to every extension point, is the simplest dictionary. It consists of a set of name-value pairs in which the name and the value are both strings. 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 may place values in the environment dictionary to control the flow or data after the extension completed its processing.

The environment dictionary is unique in that an extension may put any name/value pair it wishes in it. Although you will not get an error for using undocumented name-values, the server does not recognize 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 may put data into the environment dictionary, and then an extension run at the **pre-packet-encode** extension point might read that information from the dictionary.

**Note**

The extension point **pre-dns-add-forward** has an environment dictionary that is not the same as other extension points use. The extension point **init-entry** also has a unique environment dictionary only.

## Environment Dictionary Data Items

These data items are always valid in the environment dictionary:

- **drop** (read/write)—If the **drop** value is equal to the string *true* when the extension exits, then the DHCP server drops the input packet and logs a message in the log file.
- **extension-name** (read-only)—Name with which the extension was configured. The same piece of code can be configured as several different extensions and at several different extension points. This allows one piece of code to do different things, depending on how it is configured. 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** (read-only)—Maps to a comma-separated string representing the configuration for this extension point. It allows an extension to determine the environment in which it is running. All the extensions that you configure for this extension point must be listed in sequential order and separated by commas. Any position in the sequence without an extension configured must be represented by adjacent commas. For example, to configure **tcfirst** as the first position in the sequence and **dexscript** as the fifth position in the sequence, you would use **tcfirst,,,dexscript**.
- **extension-point** (read-only)—Name of the extension point. It is made available to an extension so that one extension may run at several extension points and determine from which point it is called. For example, **post-packet-decode**.
- **extension-sequence** (read-only)—String that is the sequence number of this extension at this extension point.
- **log-drop-message** (read/write)—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. For all extension points that have a request dictionary, the data items that begin with **log** and **verbose-logging** can be set at any time. The DHCP server reads them as needed.
- **trace-level** (write-only)—Setting this to a number makes that number the current setting of the *extension-trace-level* server attribute for all extensions processing this request.

## 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. The DHCP property **initial-environment-dictionary** can be set 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.

The values defined using the **initial-environment-dictionary** approach may be read 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 that is 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

These dictionaries have a fixed set of accessible names. However, not all the names are accessible 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 encounters an error.

The request dictionary is available at the beginning of the processing of a request. After the DHCP 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 that is configurable in the server. In some cases, however, you can use an extension to change configured information, but only for the duration of the processing for just that single request.

## Decoded DHCP Packet Data Items

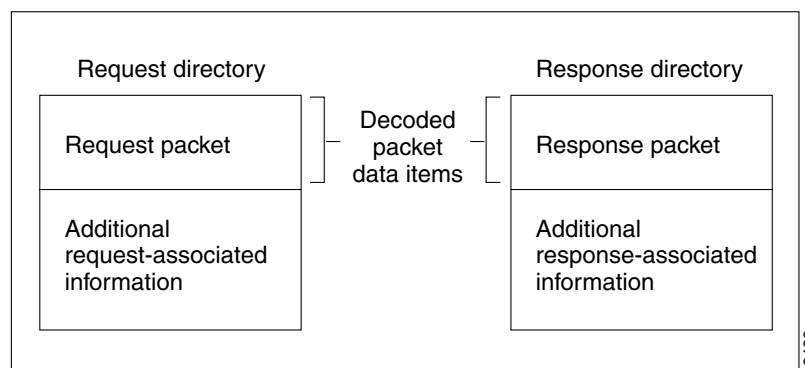
The DHCP protocol is a request-response UDP-based protocol and, thus, the stimulation for a DHCP server operation is usually a DHCP request from a client. The result is usually a DHCP response to be sent back to that client.

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.

In addition to this DHCP packet-based information, there is additional information that the DHCP server uses when processing DHCP requests. This information is associated with either the DHCP request or the DHCP response as part of the architecture of the DHCP server. Much of this information 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 information 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 that are 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. Figure 17-1 shows the relationship between the request and the response dictionaries.

**Figure 17-1 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 *ciaddr* and *giaddr*, 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. In the description of each extension point, the specific data items available to that extension point are listed. 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, no data is returned 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 the data is specifically supposed to be appended to existing data.

Some DHCP options may have multiple values. For example, the routers option may have one or more IP addresses associated with it. These multiple values are accessed using indexed operations on the option name.



**Note**

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 is handled specially in two ways:

- It is available as a multiple-valued option of bytes under the name **dhcp-parameter-request-list**.

- It is also available as a blob-valued (a sequence of bytes) option under the name **dhcp-parameter-request-list-blob**.

You can get or put it 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 it is accessed in the request dictionary, this option 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 are put into the response, until it is replaced by a new list.

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.

### post-packet-decode

The dictionaries available are Request and Environment.

The **post-packet-decode** extension point is the first one the DHCP server encounters when a request arrives. 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 this extension point to rewrite the input packet.

This 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 that you have to be aware of are very limited.

The **post-packet-decode** extension point is the only one at which you can make modifications to the decoded input packet and ensure that all the modifications are recognized. If the extension decides that the packet should be dropped, and further processing terminated, it may do so by using the **drop** data item in the environment dictionary.

All of the decoded packet data items are specified in [Table 17-2](#) lists the data items that are available in the **post-packet-decode** request dictionary.

**Table 17-2** *post-packet-decode Request Dictionary Data Items*

Request Dictionary Data Item	Value	Operation
client-ipaddress	IP address	read/write
client-port	int	read/write

Table 17-2 *post-packet-decode Request Dictionary Data Items (continued)*

Request Dictionary Data Item	Value	Operation
os-type	string	read/write
release-by-ip	int	read/write
transaction-time	int	read-only

## post-class-lookup

The dictionaries available are Request and Environment.

The **post-class-lookup** extension point is only called if there is a *client-class-lookup-id*, otherwise it is similar to the **post-packet-decode** extension point. It is called after evaluating the *client-class-lookup-id* and setting the client-class information for this client. This extension point has the same dictionaries as the **post-packet-decode** extension point.

On input to this extension point, the environment dictionary has the **drop** data item set to **true** or **false**. This can be changed by extension to cause the packet to be dropped or not, and the change is recognized. This also looks at **log-drop-message** to decide whether to log the drop.

The extension point can also set the **client-class-name** in the environment dictionary, which causes the named client-class to be set for this packet, regardless of the previous client-class. This only has an effect if the **drop** environment dictionary data item is set to **false** on exiting the extension point.

## pre-client-lookup

The dictionaries available are Request and Environment.

You can only use the **pre-client-lookup** extension point 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 is looked up during client-class processing.
- Specify individual data items to override those found from the client entry or the client-class it specifies.
- Instruct the server to skip the client lookup altogether. In this case, the only client data used is one that was supplied by the extension in the environment dictionary.

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

## Environment Dictionary for pre-client-lookup

These items are available at **pre-client-lookup** for client-class control:

- **client-specifier** (read/write)—Name of the client the client-class processing code looks up, in MCD or LDAP. If you change it at this extension point, then the DHCP server will look up whatever client is specified.
- **default-client-class-name** (read/write)—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 specific client entry could not be located.

The **default-client-class-name** data item then assumes precedence over the *class-name* associated with the default client.

- **release-by-ip** (read/write)—Instructs the server to release the lease by the IP address if the lease cannot be retrieved by the *client-id* (derived from the DHCPRELEASE request).
- **skip-client-lookup** (read/write)—If you set this item to **true**, the DHCP server skips the normal client lookup that it would have performed immediately upon exit from this extension. In this case, the only data items used to describe this client are those placed in the environment dictionary.

## Client-Class Data Input to pre-client-lookup

If you set the following 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, then the server uses what is in the client value, or key:

- **host-name** (read/write)—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's request.
- **domain-name** (read/write)—Use this domain name for the client's DNS operations in preference to the one specified in the scope. Note that the DNS server shown as the primary server for the domain in the scope must also be the primary server for the domain you specified. If there is no override for the domain name in the client or client-class entry, the DHCP server uses the domain name from the scope. If the client entry or the extension contains the word **none**, the DHCP server uses the domain name from the scope.
- **policy-name** (read/write)—Use this policy as the policy specified for the client entry, overriding any policy specified by that client entry.
- **action** (read/write)—Convert this string to a number and use the result as the action. The numbers you can use are **0x1** (for exclude) and **0x2** (for one-shot).
- **selection-criteria** (read/write)—List of comma-separated strings, each specifying (for this input packet) a scope-selection criteria for this client. Any scope this client uses must have all of these scope-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's scope-selection criteria, independent of whether they were stored in the local or LDAP database. If you set this to **none**, the DHCP server does not use scope-selection tags for this packet. If you set this to a null string, the DHCP server treats it as if it were not set and uses the scope-selection criteria from the client or client-class entry.

- **selection-criteria-excluded** (read/write)—List of comma-separated strings, each specifying (for this input) an exclusion criteria for this client. Any scope this client uses must not have any of these selection tags.

Use this data item to override any specified client or client-class entries. If you do, the DHCP server does not use the client entry's scope-exclusion criteria, independent of whether they were stored in the local or LDAP database. If you set this to **none**, the DHCP server does not use any scope-exclusion tags for this packet. If you set this to a null string, the DHCP server treats it as if it were not set and uses the scope-exclusion criteria from the client or client-class entry.

- **client-class-name** (read/write)—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 there were no client-class name specified in this client entry.

- **authenticate-until** (read/write)—Absolute time, measured in seconds, from January 1, 1970. Use to indicate the time at which the client's authentication expires. When the client's authentication expires, the DHCP server uses the values in the client's *unauthenticated-client-class* option instead of its client-class to fill in missing data items in the client entry.
- **unauthenticated-client-class-name** (read/write)—Name of the client-class to use if the client is not authenticated. If you want to indicate that no **unauthenticated-client-class-name** is specified, use an illegal client-class name as the value of this data item. The value **none** is fine, but any name that is not a client-class name will do. The DHCP server logs an error that the client-class is not present.

## Request Dictionary for pre-client-lookup

You can use all of the decoded packet data items specified in [Table 17-3](#). The request information data items available for the **pre-client-lookup** extension point are **client-ipaddress**, **client-port**, and **transaction-time** (see [Table 17-2 on page 17-20](#)). [Table 17-3](#) describes the client information data items, and [Table 17-4](#) describes the client understanding data items, available at this extension point.

**Table 17-3** *pre-client-lookup Request Dictionary Client Information Data Items*

Client Information Data Item	Value	Operation
client-id	blob	read/write
client-id-created-from-mac-address	int	read-only
client-mac-address	blob	read/write
client-os-type	string	read/write
mac-address	blob	read/write

**Table 17-4** *pre-client-lookup Request Dictionary Client Understanding Data Items*

Client Understanding Data Item	Value	Operation
client-wants-nulls-in-strings	int	read/write
import-packet	int	read/write
reply-to-client-address	int	read/write

## post-client-lookup

The dictionaries available are Request and Environment.

You can use the **post-client-lookup** 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 host name in the packet returned from the client-class processing from an extension running at the **post-client-lookup** extension point, set the host name to the **client-requested-host-name** data item in the request dictionary. This causes 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 some extension running at the **pre-packet-encode** extension point, where it might load different options into the response packet or take other actions.

## Environment Dictionary for post-client-lookup

These data items are available at **post-client-lookup**:

- **client-specifier** (read-only)—Name of the client that the client-class processing looked up.
- **cnr-ldap-query-failed** (read-only)—The DHCP server sets this attribute to ease recovery from LDAP server failures. In this manner, a post-client-lookup script can respond to 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:
  - The flag is set to **false**.
  - The *cnr-ldap-query-failed* attribute does not appear in the environment dictionary.

## Request Dictionary for post-client-lookup

You can use all the decoded packet data items described in [Table 17-5](#). The request information data items available for the **post-client-lookup** extension point are **client-ipaddress**, **client-port**, and **transaction-time** (see [Table 17-2 on page 17-20](#)). The client information data items are the same as those for the **pre-client-lookup** extension point (see [Table 17-3 on page 17-23](#)). [Table 17-5](#) describes the client understanding data items.

**Table 17-5** *post-client-lookup Request Dictionary Client Understanding Data Items*

Client Understanding Data Item	Value	Operation
client-class-name	string	read-only
client-class-policy	string	read/write
client-domain-name	string	read/write
client-host-name	string	read/write
client-policy	string	read/write
client-requested-host-name	string	read/write
client-wants-nulls-in-strings	int	read/write
import-packet	int	read/write
reply-to-client-address	int	read/write
selection-criteria	string	read/write
selection-criteria-excluded	string	read/write

## check-lease-acceptable

The dictionaries available are Request, Response, and Environment.

This **check-lease-acceptable** extension point comes immediately after the server determined 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 the [“Determining Lease Acceptability” section on page 17-13](#).

The **acceptable** data item is available in the environment dictionary at this extension point. This is a read/write data item that the DHCP server initializes depending on whether 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.

All the request dictionary data items available for **pre-packet-encode** are available for a **check-lease-acceptable** request.

All the response dictionary data items available for **pre-packet-encode** are available for **check-lease-acceptable** response, with the addition of the **client-os-type** data item. This is a read/write data item that you can read and change in a extension. However, you can set the **client-os-type** data item only by changing the **os-type** data item in the **post-packet-decode** request dictionary.

## lease-state-change

The dictionaries available are Response and Environment.

The **lease-state-change** extension point is called whenever a lease changes state. The existing state is in the **lease-state** response dictionary lease information data item (see [Table 17-7](#)). The new state is in the environment dictionary data item **new-state**. The extension point is never called if the new state matches the existing one.

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

### Response Dictionary for lease-state-change

The **lease-state-change** extension point has the same response dictionary data items as the pre-packet-encode extension point, although only the client information and lease information data items are very useful (see [Table 17-6](#) and [Table 17-7](#)). For this extension point, do not access any options in the response dictionary or any encoded packet data items (such as *giaddr* and *ciaddr*), because there is often no response packet transmitted.

**Table 17-6** *lease-state-change Response Dictionary Client Information Data Items*

Client Information Data Item	Value	Operation
client-domain-name	string	read/write
client-expiration-time	blob	read-only
client-host-name	string	read/write
client-id	blob	read/write
client-id-created-from-mac-address	int	read-only
client-mac-address	blob	read/write
client-requested-host-name	string	read/write
domain-name-changed	int	read/write
host-name-changed	int	read/write
host-name-in-dns	int	read/write
last-transaction-time	int	read-only
mac-address	blob	read/write

**Table 17-6** *lease-state-change Response Dictionary Client Information Data Items (continued)*

Client Information Data Item	Value	Operation
reverse-name-in-dns	int	read/write

**Table 17-7** *lease-state-change Response Dictionary Lease Information Data Items*

Lease Information Data Item	Value	Operation
lease-deactivated	int	read-only
lease-ipaddress	IP address	read-only
lease-reserved	int	read-only
lease-state	string	read-only
start-time-of-state	int	read-only

## Environment Dictionary for lease-state-change

The current state is in the **lease-state** lease information data item in the response dictionary (see [Table 17-7 on page 17-26](#)), and the state being changed to is in the environment dictionary under the **new-state** data item. This is all read-only.

In the initialization entry point for a script attached to this extension point, if on exit the environment dictionary contains an attribute **exiting-state**, the value of that attribute is saved and the script is only called when a lease is transitioning *away* from that state. Thus, you might set **exiting-state** to **leased**, and the extension point would only be called when the existing state was leased and the new state was something else. You would do this because you caught other lease transitions in other ways (such as in the **post-send-packet** extension point) and you did not want to incur overhead for calling this extension point except for transitions that you really care about.

The valid lease states are:

- available
- offered
- leased
- expired
- unavailable
- released
- other-available
- pending-available

There is no strict state transition table that can be drawn for these states. In a failover environment, the server that receives a binding update message typically sets the state to whatever its partner tells it to be, without requiring any specific state transitions.

## pre-packet-encode

The dictionaries available are Request, Response, and Environment.

## Request Dictionary for pre-packet-encode

You can use all the decoded packet data items described in [Table 17-8](#). The request information data items available for the **pre-packet-encode** extension point are **client-ipaddress**, **client-port**, and **transaction-time** (see [Table 17-2 on page 17-20](#)). [Table 17-8](#) describes the client information data items. The client understanding data items are the same as for the **post-client-lookup** extension point (see [Table 17-5 on page 17-24](#)).

**Table 17-8** *pre-packet-encode Client Information Data Items*

Client Information Data Item	Value	Operation
client-id	blob	read/write
client-id-created-from-mac-address	int	read-only
client-macaddress	blob	read/write
mac-address	blob	read/write

## Response Dictionary for pre-packet-encode

You can use all the decoded packet data item described in [Table 17-9](#). The **pre-packet-encode** extension point has these response dictionary data items:

- General—See [Table 17-9](#)
- Client information—See [Table 17-10](#)
- Lease information—See [Table 17-11 on page 17-28](#)
- Scope address information—See [Table 17-12 on page 17-28](#)
- Scope acceptability information—See [Table 17-13 on page 17-28](#)
- Scope DNS information—See [Table 17-14 on page 17-29](#)

**Table 17-9** *pre-packet-encode Response Dictionary Data Items*

Response Dictionary Data Item	Value	Operation
auto-configure	int	read/write
reply-ipaddress	IP address	read/write
reply-port	int	read/write
scope-ping-clients	int	read-only
scope-renew-only	int	read-only
scope-renew-only-expire-time	int	read-only
scope-selection-tags	string	read-only
scope-send-ack-first	int	read-only
transaction-time	int	read-only

**Table 17-10 pre-packet-encode Client Information Data Items**

Client Information Data Item	Value	Operation
client-domain-name	string	read/write
client-expiration-time	blob	read-only
client-host-name	string	read/write
client-id	blob	read/write
client-id-created-from-mac-address	int	read-only
client-mac-address	blob	read/write
client-requested-host-name	string	read/write
domain-name-changed	int	read/write
host-name-changed	int	read/write
host-name-in-dns	int	read/write
last-transaction-time	int	read-only
mac-address	blob	read/write
reverse-name-in-dns	int	read/write

**Table 17-11 pre-packet-encode Lease Information Data Items**

Lease Information Data Item	Value	Operation
lease-deactivated	int	read-only
lease-ipaddress	IP address	read-only
lease-reserved	int	read-only
lease-state	string	read-only
lease-start-time-of-state	int	read-only

**Table 17-12 pre-packet-encode Scope Address Information Data Items**

Scope Address Information Data Item	Value	Operation
scope-network-number	IP address	read-only
scope-primary-network-number	IP address	read-only
scope-primary-subnet-mask	IP address	read-only
scope-subnet-mask	IP address	read-only

**Table 17-13 pre-packet-encode Scope Acceptability Information Data Items**

Scope Acceptability Information Data Item	Value	Operation
scope-allow-bootp	int	read-only
scope-allow-dhcp	int	read-only
scope-allow-dynamic-bootp	int	read-only

**Table 17-13 pre-packet-encode Scope Acceptability Information Data Items (continued)**

Scope Acceptability Information Data Item	Value	Operation
scope-available-leases	int	read-only
scope-deactivated	int	read-only

**Table 17-14 pre-packet-encode Scope DNS Information Data Items**

Scope DNS Information Data Item	Value	Operation
scope-dns-forward-server-address	IP address	read-only
scope-dns-forward-zone-name	string	read-only
scope-dns-number-of-host-bytes	int	read-only
scope-dns-reverse-server-address	IP address	read-only
scope-dns-reverse-zone-name	string	read-only
scope-update-dns-enabled	int	read-only
scope-update-dns-for-bootp	int	read-only

## pre-dns-add-forward

The dictionary available is Environment.

You can use the **pre-dns-add-forward** extension point to choose the name and affect the DNS retries during update operations. These data items are available in the environment dictionary at this extension point:

- **host-name** (read/write)—Host name that the DHCP server tries next when updating the DNS server. You can use an extension to read and change the name.
- **txt-string** (read/write)—TXT record string the DHCP server writes to DNS. By default this is the client-id rendered as a blob, but it can be anything. It identifies this client as the owner of this name, so correct operation relies on a one-to-one mapping between the text string and the client.
- **domain-name** (read-only)—Domain name the DHCP server uses for DNS updates.
- **renaming-retries** (read-only)—Current renaming retry count.
- **maximum-renaming-retries** (read/write)—Maximum number of renaming retries.
- **last-name-number** (read/write)—Last number that was used to disambiguate a DNS name.
- **last-name-number-length** (read/write)—Length of that number, that is, the number of characters it used in the name when rendered as a decimal number.
- **ignore-prerequisites** (read/write)—If you set it to **true**, the DHCP server ignores the setting of the prerequisites on the DNS A record update, which will cause the *last* client to attempt to get a name to succeed. The default behavior of the DHCP server is for the *first* client to use a name to get the name and for later clients to get a disambiguated name.
- **continue** (read/write)—The DHCP server will continue to update DNS if any renaming retries are left. If you set this to **false**, the DHCP server stops attempting to update DNS, even if renaming retries remain.

