dhcpd.conf(5) - Linux man page

Name

dhcpd.conf - dhcpd configuration file

Description

The dhcpd.conf file contains configuration information for *dhcpd,* the Internet Systems Consortium DHCP Server.

The dhcpd.conf file is a free-form ASCII text file. It is parsed by the recursive-descent parser built into dhcpd. The file may contain extra tabs and newlines for formatting purposes. Keywords in the file are case-insensitive. Comments may be placed anywhere within the file (except within quotes). Comments begin with the # character and end at the end of the line.

The file essentially consists of a list of statements. Statements fall into two broad categories - parameters and declarations.

Parameter statements either say how to do something (e.g., how long a lease to offer), whether to do something (e.g., should dhcpd provide addresses to unknown clients), or what parameters to provide to the client (e.g., use gateway 220.177.244.7).

Declarations are used to describe the topology of the network, to describe clients on the network, to provide addresses that can be assigned to clients, or to apply a group of parameters to a group of declarations. In any group of parameters and declarations, all parameters must be specified before any declarations which depend on those parameters may be specified.

Declarations about network topology include the *shared-network* and the *subnet* declarations. If clients on a subnet are to be assigned addresses dynamically, a *range* declaration must appear within the *subnet* declaration. For clients with statically assigned addresses, or for installations where only known clients will be served, each such client must have a *host* declaration. If parameters are to be applied to a group of declarations which are not related strictly on a per-subnet basis, the *group* declaration can be used.

For every subnet which will be served, and for every subnet to which the dhcp server is connected, there must be one *subnet* declaration, which tells dhcpd how to recognize that an address is on that subnet. A *subnet* declaration is required for each subnet even if no addresses will be dynamically allocated on that subnet.

Some installations have physical networks on which more than one IP subnet operates. For example, if there is a site-wide requirement that 8-bit subnet masks be used, but a department with a single physical ethernet network expands to the point where it has more than 254 nodes, it may be necessary to run two 8-bit subnets on the same ethernet until such time as a new physical network can be added. In this case, the *subnet* declarations for these two networks must be enclosed in a *shared-network* declaration.

Note that even when the *shared-network* declaration is absent, an empty one is created by the server to contain the *subnet* (and any scoped parameters included in the *subnet*). For practical purposes, this means that "stateless" DHCP clients, which are not tied to addresses (and therefore

subnets) will receive the same configuration as stateful ones.

Some sites may have departments which have clients on more than one subnet, but it may be desirable to offer those clients a uniform set of parameters which are different than what would be offered to clients from other departments on the same subnet. For clients which will be declared explicitly with *host* declarations, these declarations can be enclosed in a *group* declaration along with the parameters which are common to that department. For clients whose addresses will be dynamically assigned, class declarations and conditional declarations may be used to group parameter assignments based on information the client sends.

When a client is to be booted, its boot parameters are determined by consulting that client's *host* declaration (if any), and then consulting any *class* declarations matching the client, followed by the *pool*, *subnet* and *shared-network* declarations for the IP address assigned to the client. Each of these declarations itself appears within a lexical scope, and all declarations at less specific lexical scopes are also consulted for client option declarations. Scopes are never considered twice, and if parameters are declared in more than one scope, the parameter declared in the most specific scope is the one that is used.

When dhcpd tries to find a *host* declaration for a client, it first looks for a *host* declaration which has a *fixed-address* declaration that lists an IP address that is valid for the subnet or shared network on which the client is booting. If it doesn't find any such entry, it tries to find an entry which has no *fixed-address* declaration.

Examples

A typical dhcpd.conf file will look something like this:

```
global parameters...

subnet 204.254.239.0 netmask 255.255.255.224 {
  subnet-specific parameters...
  range 204.254.239.10 204.254.239.30;
}

subnet 204.254.239.32 netmask 255.255.255.224 {
  subnet-specific parameters...
  range 204.254.239.42 204.254.239.62;
}

subnet 204.254.239.64 netmask 255.255.255.224 {
  subnet-specific parameters...
  range 204.254.239.74 204.254.239.94;
}

group {
  group-specific parameters...
  host zappo.test.isc.org {
    host-specific parameters...
  }
  host beppo.test.isc.org {
    host-specific parameters...
```

```
  }
  host harpo.test.isc.org {
    host-specific parameters...
  }
}
```

Figure 1

Notice that at the beginning of the file, there's a place for global parameters. These might be things like the organization's domain name, the addresses of the name servers (if they are common to the entire organization), and so on. So, for example:

```
option domain-name "isc.org";

    option domain-name-servers ns1.isc.org, ns2.isc.org;
```

Figure 2

As you can see in Figure 2, you can specify host addresses in parameters using their domain names rather than their numeric IP addresses. If a given hostname resolves to more than one IP address (for example, if that host has two ethernet interfaces), then where possible, both addresses are supplied to the client.

The most obvious reason for having subnet-specific parameters as shown in Figure 1 is that each subnet, of necessity, has its own router. So for the first subnet, for example, there should be something like:

```
    option routers 204.254.239.1;
```

Note that the address here is specified numerically. This is not required - if you have a different domain name for each interface on your router, it's perfectly legitimate to use the domain name for that interface instead of the numeric address. However, in many cases there may be only one domain name for all of a router's IP addresses, and it would not be appropriate to use that name here.

In Figure 1 there is also a *group* statement, which provides common parameters for a set of three hosts - zappo, beppo and harpo. As you can see, these hosts are all in the test.isc.org domain, so it might make sense for a group-specific parameter to override the domain name supplied to these hosts:

```
    option domain-name "test.isc.org";
```

Also, given the domain they're in, these are probably test machines. If we wanted to test the DHCP leasing mechanism, we might set the lease timeout somewhat shorter than the default:

```
    max-lease-time 120;

    default-lease-time 120;
```

You may have noticed that while some parameters start with the *option* keyword, some do not. Parameters starting with the *option* keyword correspond to actual DHCP options, while parameters that do not start with the option keyword either control the behavior of the DHCP server (e.g., how long a lease dhcpd will give out), or specify client parameters that are not optional in the DHCP protocol (for example, server-name and filename).

In Figure 1, each host had *host-specific parameters*. These could include such things as the *hostname* option, the name of a file to upload (the *filename* parameter) and the address of the server from which to upload the file (the *next-server* parameter). In general, any parameter can appear anywhere that parameters are allowed, and will be applied according to the scope in which the parameter appears.

Imagine that you have a site with a lot of NCD X-Terminals. These terminals come in a variety of models, and you want to specify the boot files for each model. One way to do this would be to have host declarations for each server and group them by model:

```
group {
  filename "Xncd19r";
  next-server ncd-booter;

 host ncd1 { hardware ethernet 0:c0:c3:49:2b:57; }
  host ncd4 { hardware ethernet 0:c0:c3:80:fc:32; }
  host ncd8 { hardware ethernet 0:c0:c3:22:46:81; }
}

group {
  filename "Xncd19c";
  next-server ncd-booter;

 host ncd2 { hardware ethernet 0:c0:c3:88:2d:81; }
  host ncd3 { hardware ethernet 0:c0:c3:00:14:11; }
}

group {
  filename "XncdHMX";
  next-server ncd-booter;

 host ncd1 { hardware ethernet 0:c0:c3:11:90:23; }
  host ncd4 { hardware ethernet 0:c0:c3:91:a7:8; }
  host ncd8 { hardware ethernet 0:c0:c3:cc:a:8f; }
}
```

Address Pools

The **pool** declaration can be used to specify a pool of addresses that will be treated differently than another pool of addresses, even on the same network segment or subnet. For example, you may want to provide a large set of addresses that can be assigned to DHCP clients that are registered to your DHCP server, while providing a smaller set of addresses, possibly with short lease times, that are available for unknown clients. If you have a firewall, you may be able to arrange for addresses from one pool to be allowed access to the Internet, while addresses in another pool are not, thus encouraging users to register their DHCP clients. To do this, you would set up a pair of pool declarations:

```
subnet 10.0.0.0 netmask 255.255.255.0 {
  option routers 10.0.0.254;

 # Unknown clients get this pool.
```

```
  pool {
    option domain-name-servers bogus.example.com;
    max-lease-time 300;
    range 10.0.0.200 10.0.0.253;
    allow unknown-clients;
  }

 # Known clients get this pool.
  pool {
    option domain-name-servers ns1.example.com, ns2.example.com;
    max-lease-time 28800;
    range 10.0.0.5 10.0.0.199;
    deny unknown-clients;
  }
}
```

It is also possible to set up entirely different subnets for known and unknown clients - address pools exist at the level of shared networks, so address ranges within pool declarations can be on different subnets.

As you can see in the preceding example, pools can have permit lists that control which clients are allowed access to the pool and which aren't. Each entry in a pool's permit list is introduced with the *allow* or *deny* keyword. If a pool has a permit list, then only those clients that match specific entries on the permit list will be eligible to be assigned addresses from the pool. If a pool has a deny list, then only those clients that do not match any entries on the deny list will be eligible. If both permit and deny lists exist for a pool, then only clients that match the permit list and do not match the deny list will be allowed access.

Dynamic Address Allocation

Address allocation is actually only done when a client is in the INIT state and has sent a DHCPDISCOVER message. If the client thinks it has a valid lease and sends a DHCPREQUEST to initiate or renew that lease, the server has only three choices - it can ignore the DHCPREQUEST, send a DHCPNAK to tell the client it should stop using the address, or send a DHCPACK, telling the client to go ahead and use the address for a while.

If the server finds the address the client is requesting, and that address is available to the client, the server will send a DHCPACK. If the address is no longer available, or the client isn't permitted to have it, the server will send a DHCPNAK. If the server knows nothing about the address, it will remain silent, unless the address is incorrect for the network segment to which the client has been attached and the server is authoritative for that network segment, in which case the server will send a DHCPNAK even though it doesn't know about the address.

There may be a host declaration matching the client's identification. If that host declaration contains a fixed-address declaration that lists an IP address that is valid for the network segment to which the client is connected. In this case, the DHCP server will never do dynamic address allocation. In this case, the client is *required* to take the address specified in the host declaration. If the client sends a DHCPREQUEST for some other address, the server will respond with a DHCPNAK.

When the DHCP server allocates a new address for a client (remember, this only happens if the

client has sent a DHCPDISCOVER), it first looks to see if the client already has a valid lease on an IP address, or if there is an old IP address the client had before that hasn't yet been reassigned. In that case, the server will take that address and check it to see if the client is still permitted to use it. If the client is no longer permitted to use it, the lease is freed if the server thought it was still in use - the fact that the client has sent a DHCPDISCOVER proves to the server that the client is no longer using the lease.

If no existing lease is found, or if the client is forbidden to receive the existing lease, then the server will look in the list of address pools for the network segment to which the client is attached for a lease that is not in use and that the client is permitted to have. It looks through each pool declaration in sequence (all *range* declarations that appear outside of pool declarations are grouped into a single pool with no permit list). If the permit list for the pool allows the client to be allocated an address from that pool, the pool is examined to see if there is an address available. If so, then the client is tentatively assigned that address. Otherwise, the next pool is tested. If no addresses are found that can be assigned to the client, no response is sent to the client.

If an address is found that the client is permitted to have, and that has never been assigned to any client before, the address is immediately allocated to the client. If the address is available for allocation but has been previously assigned to a different client, the server will keep looking in hopes of finding an address that has never before been assigned to a client.

The DHCP server generates the list of available IP addresses from a hash table. This means that the addresses are not sorted in any particular order, and so it is not possible to predict the order in which the DHCP server will allocate IP addresses. Users of previous versions of the ISC DHCP server may have become accustomed to the DHCP server allocating IP addresses in ascending order, but this is no longer possible, and there is no way to configure this behavior with version 3 of the ISC DHCP server.

Ip Address Conflict Prevention

The DHCP server checks IP addresses to see if they are in use before allocating them to clients. It does this by sending an ICMP Echo request message to the IP address being allocated. If no ICMP Echo reply is received within a second, the address is assumed to be free. This is only done for leases that have been specified in range statements, and only when the lease is thought by the DHCP server to be free - i.e., the DHCP server or its failover peer has not listed the lease as in use.

If a response is received to an ICMP Echo request, the DHCP server assumes that there is a configuration error - the IP address is in use by some host on the network that is not a DHCP client. It marks the address as abandoned, and will not assign it to clients.

If a DHCP client tries to get an IP address, but none are available, but there are abandoned IP addresses, then the DHCP server will attempt to reclaim an abandoned IP address. It marks one IP address as free, and then does the same ICMP Echo request check described previously. If there is no answer to the ICMP Echo request, the address is assigned to the client.

The DHCP server does not cycle through abandoned IP addresses if the first IP address it tries to reclaim is free. Rather, when the next DHCPDISCOVER comes in from the client, it will attempt a new allocation using the same method described here, and will typically try a new IP address.

Dhcp Failover

This version of the ISC DHCP server supports the DHCP failover protocol as documented in draft-ietf-dhc-failover-07.txt. This is not a final protocol document, and we have not done interoperability testing with other vendors' implementations of this protocol, so you must not assume that this implementation conforms to the standard. If you wish to use the failover protocol, make sure that both failover peers are running the same version of the ISC DHCP server.

The failover protocol allows two DHCP servers (and no more than two) to share a common address pool. Each server will have about half of the available IP addresses in the pool at any given time for allocation. If one server fails, the other server will continue to renew leases out of the pool, and will allocate new addresses out of the roughly half of available addresses that it had when communications with the other server were lost.

It is possible during a prolonged failure to tell the remaining server that the other server is down, in which case the remaining server will (over time) reclaim all the addresses the other server had available for allocation, and begin to reuse them. This is called putting the server into the PARTNER-DOWN state.

You can put the server into the PARTNER-DOWN state either by using the **omshell (1)** command or by stopping the server, editing the last peer state declaration in the lease file, and restarting the server. If you use this last method, be sure to leave the date and time of the start of the state blank:

```
failover peer name state {
my state partner-down;
peer state state at date;
}
```

When the other server comes back online, it should automatically detect that it has been offline and request a complete update from the server that was running in the PARTNER-DOWN state, and then both servers will resume processing together.

It is possible to get into a dangerous situation: if you put one server into the PARTNER-DOWN state, and then *that* server goes down, and the other server comes back up, the other server will not know that the first server was in the PARTNER-DOWN state, and may issue addresses previously issued by the other server to different clients, resulting in IP address conflicts. Before putting a server into PARTNER-DOWN state, therefore, make *sure* that the other server will not restart automatically.

The failover protocol defines a primary server role and a secondary server role. There are some differences in how primaries and secondaries act, but most of the differences simply have to do with providing a way for each peer to behave in the opposite way from the other. So one server must be configured as primary, and the other must be configured as secondary, and it doesn't matter too much which one is which.

Failover Startup

When a server starts that has not previously communicated with its failover peer, it must establish communications with its failover peer and synchronize with it before it can serve clients. This can happen either because you have just configured your DHCP servers to perform failover for the first time, or because one of your failover servers has failed catastrophically and lost its database.

The initial recovery process is designed to ensure that when one failover peer loses its database

and then resynchronizes, any leases that the failed server gave out before it failed will be honored. When the failed server starts up, it notices that it has no saved failover state, and attempts to contact its peer.

When it has established contact, it asks the peer for a complete copy its peer's lease database. The peer then sends its complete database, and sends a message indicating that it is done. The failed server then waits until MCLT has passed, and once MCLT has passed both servers make the transition back into normal operation. This waiting period ensures that any leases the failed server may have given out while out of contact with its partner will have expired.

While the failed server is recovering, its partner remains in the partner-down state, which means that it is serving all clients. The failed server provides no service at all to DHCP clients until it has made the transition into normal operation.

In the case where both servers detect that they have never before communicated with their partner, they both come up in this recovery state and follow the procedure we have just described. In this case, no service will be provided to DHCP clients until MCLT has expired.

Configuring Failover

In order to configure failover, you need to write a peer declaration that configures the failover protocol, and you need to write peer references in each pool declaration for which you want to do failover. You do not have to do failover for all pools on a given network segment. You must not tell one server it's doing failover on a particular address pool and tell the other it is not. You must not have any common address pools on which you are not doing failover. A pool declaration that utilizes failover would look like this:

```
pool {

    failover peer "foo";

    pool specific parameters

};
```

Dynamic BOOTP leases are not compatible with failover, and, as such, you need to disallow BOOTP in pools that you are using failover for.

The server currently does very little sanity checking, so if you configure it wrong, it will just fail in odd ways. I would recommend therefore that you either do failover or don't do failover, but don't do any mixed pools. Also, use the same master configuration file for both servers, and have a separate file that contains the peer declaration and includes the master file. This will help you to avoid configuration mismatches. As our implementation evolves, this will become less of a problem. A basic sample dhcpd.conf file for a primary server might look like this:

```
failover peer "foo" {
  primary;
  address anthrax.rc.vix.com;
  port 647;
  peer address trantor.rc.vix.com;
  peer port 847;
  max-response-delay 60;
```

```
  max-unacked-updates 10;
  mclt 3600;
  split 128;
  load balance max seconds 3;
}

include "/etc/dhcpd.master";
```

The statements in the peer declaration are as follows:

The *primary* and *secondary* statements

[**primary** | **secondary**];

This determines whether the server is primary or secondary, as described earlier under DHCP FAILOVER.

The *address* statement
address *address*;

The **address** statement declares the IP address or DNS name on which the server should listen for connections from its failover peer, and also the value to use for the DHCP Failover Protocol server identifier. Because this value is used as an identifier, it may not be omitted.

The *peer address* statement
peer address *address*;

The **peer address** statement declares the IP address or DNS name to which the server should connect to reach its failover peer for failover messages.

The *port* statement
port *port-number*;

The **port** statement declares the TCP port on which the server should listen for connections from its failover peer.

The *peer port* statement
peer port *port-number*;

The **peer port** statement declares the TCP port to which the server should connect to reach its failover peer for failover messages. The port number declared in the **peer port** statement may be the same as the port number declared in the **port** statement.

The *max-response-delay* statement
max-response-delay *seconds*;

The **max-response-delay** statement tells the DHCP server how many seconds may pass without receiving a message from its failover peer before it assumes that connection has failed. This number should be small enough that a transient network failure that breaks the connection will not result in the servers being out of communication for a long time, but large enough that the server isn't constantly making and breaking connections. This parameter must be specified.

The *max-unacked-updates* statement
max-unacked-updates *count*;

The **max-unacked-updates** statement tells the remote DHCP server how many BNDUPD messages it can send before it receives a BNDACK from the local system. We don't have enough operational experience to say what a good value for this is, but 10 seems to work. This parameter must be specified.

The *mclt* statement
mclt *seconds*;

The **mclt** statement defines the Maximum Client Lead Time. It must be specified on the primary, and may not be specified on the secondary. This is the length of time for which a lease may be renewed by either failover peer without contacting the other. The longer you set this, the longer it will take for the running server to recover IP addresses after moving into PARTNER-DOWN state. The shorter you set it, the more load your servers will experience when they are not communicating. A value of something like 3600 is probably reasonable, but again bear in mind that we have no real operational experience with this.

The *split* statement
split *index*;

The split statement specifies the split between the primary and secondary for the purposes of load balancing. Whenever a client makes a DHCP request, the DHCP server runs a hash on the client identification, resulting in value from 0 to 255. This is used as an index into a 256 bit field. If the bit at that index is set, the primary is responsible. If the bit at that index is not set, the secondary is responsible. The **split** value determines how many of the leading bits are set to one. So, in practice, higher split values will cause the primary to serve more clients than the secondary. Lower split values, the converse. Legal values are between 0 and 255, of which the most reasonable is 128.

The *hba* statement
hba *colon-separated-hex-list*;

The hba statement specifies the split between the primary and secondary as a bitmap rather than a cutoff, which theoretically allows for finer-grained control. In practice, there is probably no need for such fine-grained control, however. An example hba statement:

```
hba ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:
    00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00;
```

This is equivalent to a **split 128;** statement, and identical. The following two examples are also equivalent to a **split** of 128, but are not identical:

```
 hba aa:aa:aa:aa:aa:aa:aa:aa:aa:aa:aa:aa:aa:aa:aa:aa:
     aa:aa:aa:aa:aa:aa:aa:aa:aa:aa:aa:aa:aa:aa:aa:aa;

hba 55:55:55:55:55:55:55:55:55:55:55:55:55:55:55:55:
     55:55:55:55:55:55:55:55:55:55:55:55:55:55:55:55;
```

They are equivalent, because half the bits are set to 0, half are set to 1 (0xa and 0x5 are 1010 and 0101 binary respectively) and consequently this would roughly divide the clients equally between

the servers. They are not identical, because the actual peers this would load balance to each server are different for each example.

You must only have **split** or **hba** defined, never both. For most cases, the fine-grained control that **hba** offers isn't necessary, and **split** should be used.

The *load balance max seconds* statement
load balance max seconds *seconds*;

This statement allows you to configure a cutoff after which load balancing is disabled. The cutoff is based on the number of seconds since the client sent its first DHCPDISCOVER or DHCPREQUEST message, and only works with clients that correctly implement the *secs* field - fortunately most clients do. We recommend setting this to something like 3 or 5. The effect of this is that if one of the failover peers gets into a state where it is responding to failover messages but not responding to some client requests, the other failover peer will take over its client load automatically as the clients retry.

The Failover pool balance statements.
max-lease-misbalance *percentage*; **max-lease-ownership** *percentage*; **min-balance** *seconds*; **max-balance** *seconds*;

This version of the DHCP Server evaluates pool balance on a schedule, rather than on demand as leases are allocated. The latter approach proved to be slightly klunky when pool misbalanced reach total saturation...when any server ran out of leases to assign, it also lost its ability to notice it had run dry.

In order to understand pool balance, some elements of its operation first need to be defined. First, there are 'free' and 'backup' leases. Both of these are referred to as 'free state leases'. 'free' and 'backup' are 'the free states' for the purpose of this document. The difference is that only the primary may allocate from 'free' leases unless under special circumstances, and only the secondary may allocate 'backup' leases.

When pool balance is performed, the only plausible expectation is to provide a 50/50 split of the free state leases between the two servers. This is because no one can predict which server will fail, regardless of the relative load placed upon the two servers, so giving each server half the leases gives both servers the same amount of 'failure endurance'. Therefore, there is no way to configure any different behaviour, outside of some very small windows we will describe shortly.

The first thing calculated on any pool balance run is a value referred to as 'lts', or "Leases To Send". This, simply, is the difference in the count of free and backup leases, divided by two. For the secondary, it is the difference in the backup and free leases, divided by two. The resulting value is signed: if it is positive, the local server is expected to hand out leases to retain a 50/50 balance. If it is negative, the remote server would need to send leases to balance the pool. Once the lts value reaches zero, the pool is perfectly balanced (give or take one lease in the case of an odd number of total free state leases).

The current approach is still something of a hybrid of the old approach, marked by the presence of the **max-lease-misbalance** statement. This parameter configures what used to be a 10% fixed value in previous versions: if lts is less than free+backup * **max-lease-misbalance** percent, then the server will skip balancing a given pool (it won't bother moving any leases, even if some leases "should" be moved). The meaning of this value is also somewhat overloaded, however, in that it also governs the estimation of when to attempt to balance the pool (which may then also be

skipped over). The oldest leases in the free and backup states are examined. The time they have resided in their respective queues is used as an estimate to indicate how much time it is probable it would take before the leases at the top of the list would be consumed (and thus, how long it would take to use all leases in that state). This percentage is directly multiplied by this time, and fit into the schedule if it falls within the **min-balance** and **max-balance** configured values. The scheduled pool check time is only moved in a downwards direction, it is never increased. Lastly, if the lts is more than double this number in the negative direction, the local server will 'panic' and transmit a Failover protocol POOLREQ message, in the hopes that the remote system will be woken up into action.

Once the lts value exceeds the **max-lease-misbalance** percentage of total free state leases as described above, leases are moved to the remote server. This is done in two passes.

In the first pass, only leases whose most recent bound client would have been served by the remote server - according to the Load Balance Algorithm (see above **split** and **hba** configuration statements) - are given away to the peer. This first pass will happily continue to give away leases, decrementing the lts value by one for each, until the lts value has reached the negative of the total number of leases multiplied by the **max-lease-ownership** percentage. So it is through this value that you can permit a small misbalance of the lease pools - for the purpose of giving the peer more than a 50/50 share of leases in the hopes that their clients might some day return and be allocated by the peer (operating normally). This process is referred to as 'MAC Address Affinity', but this is somewhat misnamed: it applies equally to DHCP Client Identifier options. Note also that affinity is applied to leases when they enter the state be moved from free to backup if the secondary already has more than its share.

The second pass is only entered into if the first pass fails to reduce the lts underneath the total number of free state leases multiplied by the **max-lease-ownership** percentage. In this pass, the oldest leases are given over to the peer without second thought about the Load Balance Algorithm, and this continues until the lts falls under this value. In this way, the local server will also happily keep a small percentage of the leases that would normally load balance to itself.

So, the **max-lease-misbalance** value acts as a behavioural gate. Smaller values will cause more leases to transition states to balance the pools over time, higher values will decrease the amount of change (but may lead to pool starvation if there's a run on leases).

The **max-lease-ownership** value permits a small (percentage) skew in the lease balance of a percentage of the total number of free state leases.

Finally, the **min-balance** and **max-balance** make certain that a scheduled rebalance event happens within a reasonable timeframe (not to be thrown off by, for example, a 7 year old free lease).

Plausible values for the percentages lie between 0 and 100, inclusive, but values over 50 are indistinguishable from one another (once lts exceeds 50% of the free state leases, one server must therefore have 100% of the leases in its respective free state). It is recommended to select a **max-lease-ownership** value that is lower than the value selected for the **max-lease-misbalance** value. **max-lease-ownership** defaults to 10, and **max-lease-misbalance** defaults to 15.

Plausible values for the **min-balance** and **max-balance** times also range from 0 to (2^32)-1 (or the limit of your local time_t value), but default to values 60 and 3600 respectively (to place balance events between 1 minute and 1 hour).

Client Classing

Clients can be separated into classes, and treated differently depending on what class they are in. This separation can be done either with a conditional statement, or with a match statement within the class declaration. It is possible to specify a limit on the total number of clients within a particular class or subclass that may hold leases at one time, and it is possible to specify automatic subclassing based on the contents of the client packet.

To add clients to classes based on conditional evaluation, you can specify a matching expression in the class statement:

```
class "ras-clients" {
  match if substring (option dhcp-client-identifier, 1, 3) = "RAS";
}
```

Note that whether you use matching expressions or add statements (or both) to classify clients, you must always write a class declaration for any class that you use. If there will be no match statement and no in-scope statements for a class, the declaration should look like this:

```
class "ras-clients" {
}
```

Subclasses

In addition to classes, it is possible to declare subclasses. A subclass is a class with the same name as a regular class, but with a specific submatch expression which is hashed for quick matching. This is essentially a speed hack - the main difference between five classes with match expressions and one class with five subclasses is that it will be quicker to find the subclasses. Subclasses work as follows:

```
class "allocation-class-1" {
  match pick-first-value (option dhcp-client-identifier, hardware);
}

class "allocation-class-2" {
  match pick-first-value (option dhcp-client-identifier, hardware);
}

subclass "allocation-class-1" 1:8:0:2b:4c:39:ad;
subclass "allocation-class-2" 1:8:0:2b:a9:cc:e3;
subclass "allocation-class-1" 1:0:0:c4:aa:29:44;

subnet 10.0.0.0 netmask 255.255.255.0 {
  pool {
    allow members of "allocation-class-1";
    range 10.0.0.11 10.0.0.50;
  }
  pool {
    allow members of "allocation-class-2";
    range 10.0.0.51 10.0.0.100;
  }
```

```
}
```

The data following the class name in the subclass declaration is a constant value to use in matching the match expression for the class. When class matching is done, the server will evaluate the match expression and then look the result up in the hash table. If it finds a match, the client is considered a member of both the class and the subclass.

Subclasses can be declared with or without scope. In the above example, the sole purpose of the subclass is to allow some clients access to one address pool, while other clients are given access to the other pool, so these subclasses are declared without scopes. If part of the purpose of the subclass were to define different parameter values for some clients, you might want to declare some subclasses with scopes.

In the above example, if you had a single client that needed some configuration parameters, while most didn't, you might write the following subclass declaration for that client:

```
subclass "allocation-class-2" 1:08:00:2b:a1:11:31 {
  option root-path "samsara:/var/diskless/alphapc";
  filename "/tftpboot/netbsd.alphapc-diskless";
}
```

In this example, we've used subclassing as a way to control address allocation on a per-client basis. However, it's also possible to use subclassing in ways that are not specific to clients - for example, to use the value of the vendor-class-identifier option to determine what values to send in the vendor-encapsulated-options option. An example of this is shown under the VENDOR ENCAPSULATED OPTIONS head in the **dhcp-options(5)** manual page.

Per-class Limits On Dynamic Address Allocation

You may specify a limit to the number of clients in a class that can be assigned leases. The effect of this will be to make it difficult for a new client in a class to get an address. Once a class with such a limit has reached its limit, the only way a new client in that class can get a lease is for an existing client to relinquish its lease, either by letting it expire, or by sending a DHCPRELEASE packet. Classes with lease limits are specified as follows:

```
class "limited-1" {
  lease limit 4;
}
```

This will produce a class in which a maximum of four members may hold a lease at one time.

Spawning Classes

It is possible to declare a *spawning class*. A spawning class is a class that automatically produces subclasses based on what the client sends. The reason that spawning classes were created was to make it possible to create lease-limited classes on the fly. The envisioned application is a cable-modem environment where the ISP wishes to provide clients at a particular site with more than one IP address, but does not wish to provide such clients with their own subnet, nor give them an unlimited number of IP addresses from the network segment to which they are connected.

Many cable modem head-end systems can be configured to add a Relay Agent Information option

to DHCP packets when relaying them to the DHCP server. These systems typically add a circuit ID or remote ID option that uniquely identifies the customer site. To take advantage of this, you can write a class declaration as follows:

```
class "customer" {
  spawn with option agent.circuit-id;
  lease limit 4;
}
```

Now whenever a request comes in from a customer site, the circuit ID option will be checked against the class's hash table. If a subclass is found that matches the circuit ID, the client will be classified in that subclass and treated accordingly. If no subclass is found matching the circuit ID, a new one will be created and logged in the **dhcpd.leases** file, and the client will be classified in this new class. Once the client has been classified, it will be treated according to the rules of the class, including, in this case, being subject to the per-site limit of four leases.

The use of the subclass spawning mechanism is not restricted to relay agent options - this particular example is given only because it is a fairly straightforward one.

COMBINING MATCH, MATCH IF AND SPAWN WITH

In some cases, it may be useful to use one expression to assign a client to a particular class, and a second expression to put it into a subclass of that class. This can be done by combining the **match if** and **spawn with** statements, or the **match if** and **match** statements. For example:

```
class "jr-cable-modems" {
  match if option dhcp-vendor-identifier = "jrcm";
  spawn with option agent.circuit-id;
  lease limit 4;
}

class "dv-dsl-modems" {
  match if opton dhcp-vendor-identifier = "dvdsl";
  spawn with option agent.circuit-id;
  lease limit 16;
}
```

This allows you to have two classes that both have the same **spawn with** expression without getting the clients in the two classes confused with each other.

Dynamic Dns Updates

The DHCP server has the ability to dynamically update the Domain Name System. Within the configuration files, you can define how you want the Domain Name System to be updated. These updates are RFC 2136 compliant so any DNS server supporting RFC 2136 should be able to accept updates from the DHCP server.

Two DNS update schemes are currently implemented, and another is planned. The two that are currently available are the ad-hoc DNS update mode and the interim DHCP-DNS interaction draft update mode. If and when the DHCP-DNS interaction draft and the DHCID draft make it through the IETF standards process, there will be a third mode, which will be the standard DNS update

method. The DHCP server must be configured to use one of the two currently-supported methods, or not to do dns updates. This can be done with the *ddns-update-style* configuration parameter.

The Ad-hoc Dns Update Scheme

The ad-hoc Dynamic DNS update scheme is **now deprecated** and **does not work.** In future releases of the ISC DHCP server, this scheme will not likely be available. The interim scheme works, allows for failover, and should now be used. The following description is left here for informational purposes only.

The ad-hoc Dynamic DNS update scheme implemented in this version of the ISC DHCP server is a prototype design, which does not have much to do with the standard update method that is being standardized in the IETF DHC working group, but rather implements some very basic, yet useful, update capabilities. This mode **does not work** with the *failover protocol* because it does not account for the possibility of two different DHCP servers updating the same set of DNS records.

For the ad-hoc DNS update method, the client's FQDN is derived in two parts. First, the hostname is determined. Then, the domain name is determined, and appended to the hostname.

The DHCP server determines the client's hostname by first looking for a *ddns-hostname* configuration option, and using that if it is present. If no such option is present, the server looks for a valid hostname in the FQDN option sent by the client. If one is found, it is used; otherwise, if the client sent a host-name option, that is used. Otherwise, if there is a host declaration that applies to the client, the name from that declaration will be used. If none of these applies, the server will not have a hostname for the client, and will not be able to do a DNS update.

The domain name is determined from the *ddns-domainname* configuration option. The default configuration for this option is:

```
option server.ddns-domainname = config-option domain-name;
```

So if this configuration option is not configured to a different value (over-riding the above default), or if a domain-name option has not been configured for the client's scope, then the server will not attempt to perform a DNS update.

The client's fully-qualified domain name, derived as we have described, is used as the name on which an "A" record will be stored. The A record will contain the IP address that the client was assigned in its lease. If there is already an A record with the same name in the DNS server, no update of either the A or PTR records will occur - this prevents a client from claiming that its hostname is the name of some network server. For example, if you have a fileserver called "fs.sneedville.edu", and the client claims its hostname is "fs", no DNS update will be done for that client, and an error message will be logged.

If the A record update succeeds, a PTR record update for the assigned IP address will be done, pointing to the A record. This update is unconditional - it will be done even if another PTR record of the same name exists. Since the IP address has been assigned to the DHCP server, this should be safe.

Please note that the current implementation assumes clients only have a single network interface. A client with two network interfaces will see unpredictable behavior. This is considered a bug, and will be fixed in a later release. It may be helpful to enable the *one-lease-per-client* parameter so

that roaming clients do not trigger this same behavior.

The DHCP protocol normally involves a four-packet exchange - first the client sends a DHCPDISCOVER message, then the server sends a DHCPOFFER, then the client sends a DHCPREQUEST, then the server sends a DHCPACK. In the current version of the server, the server will do a DNS update after it has received the DHCPREQUEST, and before it has sent the DHCPACK. It only sends the DNS update if it has not sent one for the client's address before, in order to minimize the impact on the DHCP server.

When the client's lease expires, the DHCP server (if it is operating at the time, or when next it operates) will remove the client's A and PTR records from the DNS database. If the client releases its lease by sending a DHCPRELEASE message, the server will likewise remove the A and PTR records.

The Interim Dns Update Scheme

The interim DNS update scheme operates mostly according to several drafts that are being considered by the IETF and are expected to become standards, but are not yet standards, and may not be standardized exactly as currently proposed. These are:

draft-ietf-dhc-ddns-resolution-??.txt
draft-ietf-dhc-fqdn-option-??.txt
draft-ietf-dnsext-dhcid-rr-??.txt

Because our implementation is slightly different than the standard, we will briefly document the operation of this update style here.

The first point to understand about this style of DNS update is that unlike the ad-hoc style, the DHCP server does not necessarily always update both the A and the PTR records. The FQDN option includes a flag which, when sent by the client, indicates that the client wishes to update its own A record. In that case, the server can be configured either to honor the client's intentions or ignore them. This is done with the statement *allow client-updates;* or the statement *ignore client-updates;*. By default, client updates are allowed.

If the server is configured to allow client updates, then if the client sends a fully-qualified domain name in the FQDN option, the server will use that name the client sent in the FQDN option to update the PTR record. For example, let us say that the client is a visitor from the "radish.org" domain, whose hostname is "jschmoe". The server is for the "example.org" domain. The DHCP client indicates in the FQDN option that its FQDN is "jschmoe.radish.org.". It also indicates that it wants to update its own A record. The DHCP server therefore does not attempt to set up an A record for the client, but does set up a PTR record for the IP address that it assigns the client, pointing at jschmoe.radish.org. Once the DHCP client has an IP address, it can update its own A record, assuming that the "radish.org" DNS server will allow it to do so.

If the server is configured not to allow client updates, or if the client doesn't want to do its own update, the server will simply choose a name for the client from either the fqdn option (if present) or the hostname option (if present). It will use its own domain name for the client, just as in the ad-hoc update scheme. It will then update both the A and PTR record, using the name that it chose for the client. If the client sends a fully-qualified domain name in the fqdn option, the server uses only the leftmost part of the domain name - in the example above, "jschmoe" instead of "jschmoe.radish.org".

Further, if the *ignore client-updates;* directive is used, then the server will in addition send a response in the DHCP packet, using the FQDN Option, that implies to the client that it should perform its own updates if it chooses to do so. With *deny client-updates;*, a response is sent which indicates the client may not perform updates.

Also, if the *use-host-decl-names* configuration option is enabled, then the host declaration's *hostname* will be used in place of the *hostname* option, and the same rules will apply as described above.

The other difference between the ad-hoc scheme and the interim scheme is that with the interim scheme, a method is used that allows more than one DHCP server to update the DNS database without accidentally deleting A records that shouldn't be deleted nor failing to add A records that should be added. The scheme works as follows:

When the DHCP server issues a client a new lease, it creates a text string that is an MD5 hash over the DHCP client's identification (see draft-ietf-dnsext-dhcid-rr-??.txt for details). The update adds an A record with the name the server chose and a TXT record containing the hashed identifier string (hashid). If this update succeeds, the server is done.

If the update fails because the A record already exists, then the DHCP server attempts to add the A record with the prerequisite that there must be a TXT record in the same name as the new A record, and that TXT record's contents must be equal to hashid. If this update succeeds, then the client has its A record and PTR record. If it fails, then the name the client has been assigned (or requested) is in use, and can't be used by the client. At this point the DHCP server gives up trying to do a DNS update for the client until the client chooses a new name.

The interim DNS update scheme is called interim for two reasons. First, it does not quite follow the drafts. The current versions of the drafts call for a new DHCID RRtype, but this is not yet available. The interim DNS update scheme uses a TXT record instead. Also, the existing ddns-resolution draft calls for the DHCP server to put a DHCID RR on the PTR record, but the *interim* update method does not do this. It is our position that this is not useful, and we are working with the author in hopes of removing it from the next version of the draft, or better understanding why it is considered useful.

In addition to these differences, the server also does not update very aggressively. Because each DNS update involves a round trip to the DNS server, there is a cost associated with doing updates even if they do not actually modify the DNS database. So the DHCP server tracks whether or not it has updated the record in the past (this information is stored on the lease) and does not attempt to update records that it thinks it has already updated.

This can lead to cases where the DHCP server adds a record, and then the record is deleted through some other mechanism, but the server never again updates the DNS because it thinks the data is already there. In this case the data can be removed from the lease through operator intervention, and once this has been done, the DNS will be updated the next time the client renews.

Dynamic Dns Update Security

When you set your DNS server up to allow updates from the DHCP server, you may be exposing it to unauthorized updates. To avoid this, you should use TSIG signatures - a method of cryptographically signing updates using a shared secret key. As long as you protect the secrecy of

this key, your updates should also be secure. Note, however, that the DHCP protocol itself provides no security, and that clients can therefore provide information to the DHCP server which the DHCP server will then use in its updates, with the constraints described previously.

The DNS server must be configured to allow updates for any zone that the DHCP server will be updating. For example, let us say that clients in the sneedville.edu domain will be assigned addresses on the 10.10.17.0/24 subnet. In that case, you will need a key declaration for the TSIG key you will be using, and also two zone declarations - one for the zone containing A records that will be updates and one for the zone containing PTR records - for ISC BIND, something like this:

```
key DHCP_UPDATER {
  algorithm hmac-md5;
  secret pRP5FapFoJ95JEL06sv4PQ==;
};

zone "example.org" {

     type master;

     file "example.org.db";

     allow-update { key DHCP_UPDATER; };

};

zone "17.10.10.in-addr.arpa" {

     type master;

     file "10.10.17.db";

     allow-update { key DHCP_UPDATER; };

};
```

You will also have to configure your DHCP server to do updates to these zones. To do so, you need to add something like this to your dhcpd.conf file:

```
key DHCP_UPDATER {
  algorithm hmac-md5;
  secret pRP5FapFoJ95JEL06sv4PQ==;
};

zone EXAMPLE.ORG. {
  primary 127.0.0.1;
  key DHCP_UPDATER;
}

zone 17.127.10.in-addr.arpa. {
  primary 127.0.0.1;
  key DHCP_UPDATER;
}
```

The *primary* statement specifies the IP address of the name server whose zone information is to be updated.

Note that the zone declarations have to correspond to authority records in your name server - in the above example, there must be an SOA record for "example.org." and for "17.10.10.in-addr.arpa.". For example, if there were a subdomain "foo.example.org" with no separate SOA, you could not write a zone declaration for "foo.example.org." Also keep in mind that zone names in your DHCP configuration should end in a "."; this is the preferred syntax. If you do not end your zone name in a ".", the DHCP server will figure it out. Also note that in the DHCP configuration, zone names are not encapsulated in quotes where there are in the DNS configuration.

You should choose your own secret key, of course. The ISC BIND 8 and 9 distributions come with a program for generating secret keys called dnssec-keygen. The version that comes with BIND 9 is likely to produce a substantially more random key, so we recommend you use that one even if you are not using BIND 9 as your DNS server. If you are using BIND 9's dnssec-keygen, the above key would be created as follows:

```
dnssec-keygen -a HMAC-MD5 -b 128 -n USER DHCP_UPDATER
```

If you are using the BIND 8 dnskeygen program, the following command will generate a key as seen above:

```
dnskeygen -H 128 -u -c -n DHCP_UPDATER
```

You may wish to enable logging of DNS updates on your DNS server. To do so, you might write a logging statement like the following:

```
logging {
        channel update_debug {
        file "/var/log/update-debug.log";
        severity
        debug 3;
        print-category
        yes;
        print-severity
        yes;
        print-time
        yes;
        };
        channel security_info
        {
        file
```

```
        "/var/log/named-auth.info";

        severity

        info;

        print-category

        yes;

        print-severity

        yes;

        print-time

        yes;

        };

        category update { update_debug; };

        category security { security_info; };

};
```

You must create the /var/log/named-auth.info and /var/log/update-debug.log files before starting the name server. For more information on configuring ISC BIND, consult the documentation that accompanies it.

REFERENCE: EVENTS

There are three kinds of events that can happen regarding a lease, and it is possible to declare statements that occur when any of these events happen. These events are the commit event, when the server has made a commitment of a certain lease to a client, the release event, when the client has released the server from its commitment, and the expiry event, when the commitment expires.

To declare a set of statements to execute when an event happens, you must use the **on** statement, followed by the name of the event, followed by a series of statements to execute when the event happens, enclosed in braces. Events are used to implement DNS updates, so you should not define your own event handlers if you are using the built-in DNS update mechanism.

The built-in version of the DNS update mechanism is in a text string towards the top of server/dhcpd.c. If you want to use events for things other than DNS updates, and you also want DNS updates, you will have to start out by copying this code into your dhcpd.conf file and modifying it.

REFERENCE: DECLARATIONS

The *include* statement

```
include "filename";
```

The *include* statement is used to read in a named file, and process the contents of that file as though it were entered in place of the include statement.

The *shared-network* statement

```
shared-network name {
  [ parameters ]
  [ declarations ]
}
```

The *shared-network* statement is used to inform the DHCP server that some IP subnets actually share the same physical network. Any subnets in a shared network should be declared within a *shared-network* statement. Parameters specified in the *shared-network* statement will be used when booting clients on those subnets unless parameters provided at the subnet or host level override them. If any subnet in a shared network has addresses available for dynamic allocation, those addresses are collected into a common pool for that shared network and assigned to clients as needed. There is no way to distinguish on which subnet of a shared network a client should boot.

Name should be the name of the shared network. This name is used when printing debugging messages, so it should be descriptive for the shared network. The name may have the syntax of a valid domain name (although it will never be used as such), or it may be any arbitrary name, enclosed in quotes.

The *subnet* statement

```
subnet subnet-number netmask netmask {
  [ parameters ]
  [ declarations ]
}
```

The *subnet* statement is used to provide dhcpd with enough information to tell whether or not an IP address is on that subnet. It may also be used to provide subnet-specific parameters and to specify what addresses may be dynamically allocated to clients booting on that subnet. Such addresses are specified using the *range* declaration.

The *subnet-number* should be an IP address or domain name which resolves to the subnet number of the subnet being described. The *netmask* should be an IP address or domain name which resolves to the subnet mask of the subnet being described. The subnet number, together with the netmask, are sufficient to determine whether any given IP address is on the specified subnet.

Although a netmask must be given with every subnet declaration, it is recommended that if there is any variance in subnet masks at a site, a subnet-mask option statement be used in each subnet declaration to set the desired subnet mask, since any subnet-mask option statement will override the subnet mask declared in the subnet statement.

The *subnet6* statement

```
subnet6 subnet6-number {
  [ parameters ]
  [ declarations ]
```

```
}
```

The *subnet6* statement is used to provide dhcpd with enough information to tell whether or not an IPv6 address is on that subnet6. It may also be used to provide subnet-specific parameters and to specify what addresses may be dynamically allocated to clients booting on that subnet.

The *subnet6-number* should be an IPv6 network identifier, specified as ip6-address/bits.

The *range* statement

```
range [ dynamic-bootp ] low-address [ high-address];
```

For any subnet on which addresses will be assigned dynamically, there must be at least one *range* statement. The range statement gives the lowest and highest IP addresses in a range. All IP addresses in the range should be in the subnet in which the *range* statement is declared. The *dynamic-bootp* flag may be specified if addresses in the specified range may be dynamically assigned to BOOTP clients as well as DHCP clients. When specifying a single address, *high-address* can be omitted.

The *range6* statement

```
range6 low-address high-address;
range6 subnet6-number;
range6 subnet6-number temporary;
range6 address temporary;
```

For any IPv6 subnet6 on which addresses will be assigned dynamically, there must be at least one *range6* statement. The *range6* statement can either be the lowest and highest IPv6 addresses in a *range6*, or use CIDR notation, specified as ip6-address/bits. All IP addresses in the *range6* should be in the subnet6 in which the *range6* statement is declared.

The *temporay* variant makes the prefix (by default on 64 bits) available for temporary (RFC 4941) addresses. A new address per prefix in the shared network is computed at each request with an IA_TA option. Release and Confirm ignores temporary addresses.

Any IPv6 addresses given to hosts with *fixed-address6* are excluded from the *range6*, as are IPv6 addresses on the server itself.

The *prefix6* statement

```
prefix6 low-address high-address / bits;
```

The *prefix6* is the *range6* equivalent for Prefix Delegation (RFC 3633). Prefixes of *bits* length are assigned between *low-address* and *high-address*.

Any IPv6 prefixes given to static entries (hosts) with *fixed-prefix6* are excluded from the *prefix6*.

This statement is currently global but it should have a shared-network scope.

The *host* statement

```
host hostname {
  [ parameters ]
```

```
    [ declarations ]
}
```

The **host** declaration provides a scope in which to provide configuration information about a specific client, and also provides a way to assign a client a fixed address. The host declaration provides a way for the DHCP server to identify a DHCP or BOOTP client, and also a way to assign the client a static IP address.

If it is desirable to be able to boot a DHCP or BOOTP client on more than one subnet with fixed addresses, more than one address may be specified in the *fixed-address* declaration, or more than one **host** statement may be specified matching the same client.

If client-specific boot parameters must change based on the network to which the client is attached, then multiple **host** declarations should be used. The **host** declarations will only match a client if one of their *fixed-address* statements is viable on the subnet (or shared network) where the client is attached. Conversely, for a **host** declaration to match a client being allocated a dynamic address, it must not have any *fixed-address* statements. You may therefore need a mixture of **host** declarations for any given client...some having *fixed-address* statements, others without.

hostname should be a name identifying the host. If a *hostname* option is not specified for the host, *hostname* is used.

Host declarations are matched to actual DHCP or BOOTP clients by matching the dhcp-client-identifier option specified in the *host* declaration to the one supplied by the client, or, if the *host* declaration or the client does not provide a dhcp-client-identifier option, by matching the *hardware* parameter in the *host* declaration to the network hardware address supplied by the client. BOOTP clients do not normally provide a *dhcp-client-identifier*, so the hardware address must be used for all clients that may boot using the BOOTP protocol.

DHCPv6 servers can use the *host-identifier option* parameter in the *host* declaration, and specify any option with a fixed value to identify hosts.

Please be aware that **only** the *dhcp-client-identifier* option and the hardware address can be used to match a host declaration, or the *host-identifier option* parameter for DHCPv6 servers. For example, it is not possible to match a host declaration to a *host-name* option. This is because the host-name option cannot be guaranteed to be unique for any given client, whereas both the hardware address and *dhcp-client-identifier* option are at least theoretically guaranteed to be unique to a given client.

The *group* statement

```
group {
    [ parameters ]
    [ declarations ]
}
```

The group statement is used simply to apply one or more parameters to a group of declarations. It can be used to group hosts, shared networks, subnets, or even other groups.

REFERENCE: ALLOW AND DENY

The *allow* and *deny* statements can be used to control the response of the DHCP server to various

sorts of requests. The allow and deny keywords actually have different meanings depending on the context. In a pool context, these keywords can be used to set up access lists for address allocation pools. In other contexts, the keywords simply control general server behavior with respect to clients based on scope. In a non-pool context, the *ignore* keyword can be used in place of the *deny* keyword to prevent logging of denied requests.

Allow Deny And Ignore In Scope

The following usages of allow and deny will work in any scope, although it is not recommended that they be used in pool declarations.

The *unknown-clients* **keyword**

allow unknown-clients; deny unknown-clients; ignore unknown-clients;

The **unknown-clients** flag is used to tell dhcpd whether or not to dynamically assign addresses to unknown clients. Dynamic address assignment to unknown clients is **allow**ed by default. An unknown client is simply a client that has no host declaration.

The use of this option is now *deprecated*. If you are trying to restrict access on your network to known clients, you should use **deny unknown-clients;** inside of your address pool, as described under the heading ALLOW AND DENY WITHIN POOL DECLARATIONS.

The *bootp* **keyword**

allow bootp; deny bootp; ignore bootp;

The **bootp** flag is used to tell dhcpd whether or not to respond to bootp queries. Bootp queries are **allow**ed by default.

This option does not satisfy the requirement of failover peers for denying dynamic bootp clients. The **deny dynamic bootp clients;** option should be used instead. See the ALLOW AND DENY WITHIN POOL DECLARATIONS section of this man page for more details.

The *booting* **keyword**

allow booting; deny booting; ignore booting;

The **booting** flag is used to tell dhcpd whether or not to respond to queries from a particular client. This keyword only has meaning when it appears in a host declaration. By default, booting is **allow**ed, but if it is disabled for a particular client, then that client will not be able to get an address from the DHCP server.

The *duplicates* **keyword**

allow duplicates; deny duplicates;

Host declarations can match client messages based on the DHCP Client Identifier option or based on the client's network hardware type and MAC address. If the MAC address is used, the host declaration will match any client with that MAC address - even clients with different client identifiers. This doesn't normally happen, but is possible when one computer has more than one operating system installed on it - for example, Microsoft Windows and NetBSD or Linux.

The **duplicates** flag tells the DHCP server that if a request is received from a client that matches the MAC address of a host declaration, any other leases matching that MAC address should be discarded by the server, even if the UID is not the same. This is a violation of the DHCP protocol, but can prevent clients whose client identifiers change regularly from holding many leases at the same time. By default, duplicates are **allow**ed.

The *declines* **keyword**

allow declines; deny declines; ignore declines;

The DHCPDECLINE message is used by DHCP clients to indicate that the lease the server has offered is not valid. When the server receives a DHCPDECLINE for a particular address, it normally abandons that address, assuming that some unauthorized system is using it. Unfortunately, a malicious or buggy client can, using DHCPDECLINE messages, completely exhaust the DHCP server's allocation pool. The server will reclaim these leases, but while the client is running through the pool, it may cause serious thrashing in the DNS, and it will also cause the DHCP server to forget old DHCP client address allocations.

The **declines** flag tells the DHCP server whether or not to honor DHCPDECLINE messages. If it is set to **deny** or **ignore** in a particular scope, the DHCP server will not respond to DHCPDECLINE messages.

The *client-updates* **keyword**

allow client-updates; deny client-updates;

The **client-updates** flag tells the DHCP server whether or not to honor the client's intention to do its own update of its A record. This is only relevant when doing *interim* DNS updates. See the documentation under the heading THE INTERIM DNS UPDATE SCHEME for details.

The *leasequery* **keyword**

allow leasequery; deny leasequery;

The **leasequery** flag tells the DHCP server whether or not to answer DHCPLEASEQUERY packets. The answer to a DHCPLEASEQUERY packet includes information about a specific lease, such as when it was issued and when it will expire. By default, the server will not respond to these packets.

Allow And Deny Within Pool Declarations

The uses of the allow and deny keywords shown in the previous section work pretty much the same way whether the client is sending a DHCPDISCOVER or a DHCPREQUEST message - an address will be allocated to the client (either the old address it's requesting, or a new address) and then that address will be tested to see if it's okay to let the client have it. If the client requested it, and it's not okay, the server will send a DHCPNAK message. Otherwise, the server will simply not respond to the client. If it is okay to give the address to the client, the server will send a DHCPACK message.

The primary motivation behind pool declarations is to have address allocation pools whose allocation policies are different. A client may be denied access to one pool, but allowed access to another pool on the same network segment. In order for this to work, access control has to be done

during address allocation, not after address allocation is done.

When a DHCPREQUEST message is processed, address allocation simply consists of looking up the address the client is requesting and seeing if it's still available for the client. If it is, then the DHCP server checks both the address pool permit lists and the relevant in-scope allow and deny statements to see if it's okay to give the lease to the client. In the case of a DHCPDISCOVER message, the allocation process is done as described previously in the ADDRESS ALLOCATION section.

When declaring permit lists for address allocation pools, the following syntaxes are recognized following the allow or deny keywords:

known-clients;

If specified, this statement either allows or prevents allocation from this pool to any client that has a host declaration (i.e., is known). A client is known if it has a host declaration in *any* scope, not just the current scope.

unknown-clients;

If specified, this statement either allows or prevents allocation from this pool to any client that has no host declaration (i.e., is not known).

members of "class**";**

If specified, this statement either allows or prevents allocation from this pool to any client that is a member of the named class.

dynamic bootp clients;

If specified, this statement either allows or prevents allocation from this pool to any bootp client.

authenticated clients;

If specified, this statement either allows or prevents allocation from this pool to any client that has been authenticated using the DHCP authentication protocol. This is not yet supported.

unauthenticated clients;

If specified, this statement either allows or prevents allocation from this pool to any client that has not been authenticated using the DHCP authentication protocol. This is not yet supported.

all clients;

If specified, this statement either allows or prevents allocation from this pool to all clients. This can be used when you want to write a pool declaration for some reason, but hold it in reserve, or when you want to renumber your network quickly, and thus want the server to force all clients that have been allocated addresses from this pool to obtain new addresses immediately when they next renew.

after *time***;**

If specified, this statement either allows or prevents allocation from this pool after a given date. This

can be used when you want to move clients from one pool to another. The server adjusts the regular lease time so that the latest expiry time is at the given time+min-lease-time. A short min-lease-time enforces a step change, whereas a longer min-lease-time allows for a gradual change. *time* is either second since epoch, or a UTC time string e.g. 4 2007/08/24 09:14:32 or a string with time zone offset in seconds e.g. 4 2007/08/24 11:14:32 -7200

REFERENCE: PARAMETERS

The *adaptive-lease-time-threshold* statement

adaptive-lease-time-threshold *percentage*;

When the number of allocated leases within a pool rises above the *percentage* given in this statement, the DHCP server decreases the lease length for new clients within this pool to *min-lease-time* seconds. Clients renewing an already valid (long) leases get at least the remaining time from the current lease. Since the leases expire faster, the server may either recover more quickly or avoid pool exhaustion entirely. Once the number of allocated leases drop below the threshold, the server reverts back to normal lease times. Valid percentages are between 1 and 99.

The *always-broadcast* statement
always-broadcast *flag*;

The DHCP and BOOTP protocols both require DHCP and BOOTP clients to set the broadcast bit in the flags field of the BOOTP message header. Unfortunately, some DHCP and BOOTP clients do not do this, and therefore may not receive responses from the DHCP server. The DHCP server can be made to always broadcast its responses to clients by setting this flag to 'on' for the relevant scope; relevant scopes would be inside a conditional statement, as a parameter for a class, or as a parameter for a host declaration. To avoid creating excess broadcast traffic on your network, we recommend that you restrict the use of this option to as few clients as possible. For example, the Microsoft DHCP client is known not to have this problem, as are the OpenTransport and ISC DHCP clients.

The *always-reply-rfc1048* statement
always-reply-rfc1048 *flag*;

Some BOOTP clients expect RFC1048-style responses, but do not follow RFC1048 when sending their requests. You can tell that a client is having this problem if it is not getting the options you have configured for it and if you see in the server log the message "(non-rfc1048)" printed with each BOOTREQUEST that is logged.

If you want to send rfc1048 options to such a client, you can set the **always-reply-rfc1048** option in that client's host declaration, and the DHCP server will respond with an RFC-1048-style vendor options field. This flag can be set in any scope, and will affect all clients covered by that scope.

The *authoritative* statement
authoritative;

not authoritative;

The DHCP server will normally assume that the configuration information about a given network segment is not known to be correct and is not authoritative. This is so that if a naive user installs a DHCP server not fully understanding how to configure it, it does not send spurious DHCPNAK

messages to clients that have obtained addresses from a legitimate DHCP server on the network.

Network administrators setting up authoritative DHCP servers for their networks should always write **authoritative;** at the top of their configuration file to indicate that the DHCP server *should* send DHCPNAK messages to misconfigured clients. If this is not done, clients will be unable to get a correct IP address after changing subnets until their old lease has expired, which could take quite a long time.

Usually, writing **authoritative;** at the top level of the file should be sufficient. However, if a DHCP server is to be set up so that it is aware of some networks for which it is authoritative and some networks for which it is not, it may be more appropriate to declare authority on a per-network-segment basis.

Note that the most specific scope for which the concept of authority makes any sense is the physical network segment - either a shared-network statement or a subnet statement that is not contained within a shared-network statement. It is not meaningful to specify that the server is authoritative for some subnets within a shared network, but not authoritative for others, nor is it meaningful to specify that the server is authoritative for some host declarations and not others.

The *boot-unknown-clients* statement
boot-unknown-clients *flag*;

If the *boot-unknown-clients* statement is present and has a value of *false* or *off*, then clients for which there is no *host* declaration will not be allowed to obtain IP addresses. If this statement is not present or has a value of *true* or *on*, then clients without host declarations will be allowed to obtain IP addresses, as long as those addresses are not restricted by *allow* and *deny* statements within their *pool* declarations.

The *db-time-format* statement
db-time-format [*default* | *local*] ;

The DHCP server software outputs several timestamps when writing leases to persistent storage. This configuration parameter selects one of two output formats. The *default* format prints the day, date, and time in UTC, while the *local* format prints the system seconds-since-epoch, and helpfully provides the day and time in the system timezone in a comment. The time formats are described in detail in the **dhcpd.leases**(5) manpage.

The *ddns-hostname* statement
ddns-hostname *name*;

The *name* parameter should be the hostname that will be used in setting up the client's A and PTR records. If no ddns-hostname is specified in scope, then the server will derive the hostname automatically, using an algorithm that varies for each of the different update methods.

The *ddns-domainname* statement
ddns-domainname *name*;

The *name* parameter should be the domain name that will be appended to the client's hostname to form a fully-qualified domain-name (FQDN).

The *ddns-rev-domainname* statement
ddns-rev-domainname *name*; The *name* parameter should be the domain name that will be

appended to the client's reversed IP address to produce a name for use in the client's PTR record. By default, this is "in-addr.arpa.", but the default can be overridden here.

The reversed IP address to which this domain name is appended is always the IP address of the client, in dotted quad notation, reversed - for example, if the IP address assigned to the client is 10.17.92.74, then the reversed IP address is 74.92.17.10. So a client with that IP address would, by default, be given a PTR record of 10.17.92.74.in-addr.arpa.

The *ddns-update-style* parameter
ddns-update-style *style*;

The *style* parameter must be one of **ad-hoc**, **interim** or **none**. The *ddns-update-style* statement is only meaningful in the outer scope - it is evaluated once after reading the dhcpd.conf file, rather than each time a client is assigned an IP address, so there is no way to use different DNS update styles for different clients. The default is **none**.

The *ddns-updates* **statement**
ddns-updates *flag*;

The *ddns-updates* parameter controls whether or not the server will attempt to do a DNS update when a lease is confirmed. Set this to *off* if the server should not attempt to do updates within a certain scope. The *ddns-updates* parameter is on by default. To disable DNS updates in all scopes, it is preferable to use the *ddns-update-style* statement, setting the style to *none*.

The *default-lease-time* statement
default-lease-time *time*;

Time should be the length in seconds that will be assigned to a lease if the client requesting the lease does not ask for a specific expiration time. This is used for both DHCPv4 and DHCPv6 leases (it is also known as the "valid lifetime" in DHCPv6).

The *delayed-ack* and *max-ack-delay* statements
delayed-ack *count*; **max-ack-delay** *microseconds*;

Count should be an integer value from zero to 2^16-1, and defaults to 28. The count represents how many DHCPv4 replies maximum will be queued pending transmission until after a database commit event. If this number is reached, a database commit event (commonly resulting in fsync() and representing a performance penalty) will be made, and the reply packets will be transmitted in a batch afterwards. This preserves the RFC2131 direction that "stable storage" be updated prior to replying to clients. Should the DHCPv4 sockets "go dry" (select() returns immediately with no read sockets), the commit is made and any queued packets are transmitted.

Similarly, *microseconds* indicates how many microseconds are permitted to pass inbetween queuing a packet pending an fsync, and performing the fsync. Valid values range from 0 to 2^32-1, and defaults to 250,000 (1/4 of a second).

Please note that as delayed-ack is currently experimental, the delayed-ack feature is not compiled in by default, but must be enabled at compile time with './configure --enable-delayed-ack'.

The *do-forward-updates* statement
do-forward-updates *flag*;

The *do-forward-updates* statement instructs the DHCP server as to whether it should attempt to update a DHCP client's A record when the client acquires or renews a lease. This statement has no effect unless DNS updates are enabled and **ddns-update-style** is set to **interim**. Forward updates are enabled by default. If this statement is used to disable forward updates, the DHCP server will never attempt to update the client's A record, and will only ever attempt to update the client's PTR record if the client supplies an FQDN that should be placed in the PTR record using the fqdn option. If forward updates are enabled, the DHCP server will still honor the setting of the **client-updates** flag.

The *dynamic-bootp-lease-cutoff* statement
dynamic-bootp-lease-cutoff *date***;**

The *dynamic-bootp-lease-cutoff* statement sets the ending time for all leases assigned dynamically to BOOTP clients. Because BOOTP clients do not have any way of renewing leases, and don't know that their leases could expire, by default dhcpd assigns infinite leases to all BOOTP clients. However, it may make sense in some situations to set a cutoff date for all BOOTP leases - for example, the end of a school term, or the time at night when a facility is closed and all machines are required to be powered off.

Date should be the date on which all assigned BOOTP leases will end. The date is specified in the form:

W YYYY/MM/DD HH:MM:SS

W is the day of the week expressed as a number from zero (Sunday) to six (Saturday). YYYY is the year, including the century. MM is the month expressed as a number from 1 to 12. DD is the day of the month, counting from 1. HH is the hour, from zero to 23. MM is the minute and SS is the second. The time is always in Coordinated Universal Time (UTC), not local time.

The *dynamic-bootp-lease-length* statement
dynamic-bootp-lease-length *length***;**

The *dynamic-bootp-lease-length* statement is used to set the length of leases dynamically assigned to BOOTP clients. At some sites, it may be possible to assume that a lease is no longer in use if its holder has not used BOOTP or DHCP to get its address within a certain time period. The period is specified in *length* as a number of seconds. If a client reboots using BOOTP during the timeout period, the lease duration is reset to *length*, so a BOOTP client that boots frequently enough will never lose its lease. Needless to say, this parameter should be adjusted with extreme caution.

The *filename* statement
filename "*filename***";**

The *filename* statement can be used to specify the name of the initial boot file which is to be loaded by a client. The *filename* should be a filename recognizable to whatever file transfer protocol the client can be expected to use to load the file.

The *fixed-address* declaration
fixed-address address [**,** *address* ...]**;**

The *fixed-address* declaration is used to assign one or more fixed IP addresses to a client. It should only appear in a *host* declaration. If more than one address is supplied, then when the

client boots, it will be assigned the address that corresponds to the network on which it is booting. If none of the addresses in the *fixed-address* statement are valid for the network to which the client is connected, that client will not match the *host* declaration containing that *fixed-address* declaration. Each *address* in the *fixed-address* declaration should be either an IP address or a domain name that resolves to one or more IP addresses.

The *fixed-address6* declaration
fixed-address6 ip6-address ;

The *fixed-address6* declaration is used to assign a fixed IPv6 addresses to a client. It should only appear in a *host* declaration.

The *get-lease-hostnames* statement
get-lease-hostnames *flag***;**

The *get-lease-hostnames* statement is used to tell dhcpd whether or not to look up the domain name corresponding to the IP address of each address in the lease pool and use that address for the DHCP *hostname* option. If *flag* is true, then this lookup is done for all addresses in the current scope. By default, or if *flag* is false, no lookups are done.

The *hardware* statement
hardware *hardware-type hardware-address***;**

In order for a BOOTP client to be recognized, its network hardware address must be declared using a *hardware* clause in the *host* statement. *hardware-type* must be the name of a physical hardware interface type. Currently, only the **ethernet** and **token-ring** types are recognized, although support for a **fddi** hardware type (and others) would also be desirable. The *hardware-address* should be a set of hexadecimal octets (numbers from 0 through ff) separated by colons. The *hardware* statement may also be used for DHCP clients.

The *host-identifier option* statement
host-identifier option *option-name option-data***;**

This identifies a DHCPv6 client in a *host* statement. *option-name* is any option, and *option-data* is the value for the option that the client will send. The *option-data* must be a constant value.

The *infinite-is-reserved* statement
infinite-is-reserved *flag***;**

ISC DHCP now supports 'reserved' leases. See the section on RESERVED LEASES below. If this *flag* is on, the server will automatically reserve leases allocated to clients which requested an infinite (0xffffffff) lease-time.

The default is off.

The *lease-file-name* statement
lease-file-name *name***;**

Name should be the name of the DHCP server's lease file. By default, this is /var/lib/dhcpd/dhcpd.leases. This statement **must** appear in the outer scope of the configuration file - if it appears in some other scope, it will have no effect. Furthermore, it has no effect if overridden by the **-lf** flag or the **PATH_DHCPD_DB** environment variable.

The *limit-addrs-per-ia* statement
limit-addrs-per-ia *number*;

By default, the DHCPv6 server will limit clients to one IAADDR per IA option, meaning one address. If you wish to permit clients to hang onto multiple addresses at a time, configure a larger *number* here.

Note that there is no present method to configure the server to forcibly configure the client with one IP address per each subnet on a shared network. This is left to future work.

The *dhcpv6-lease-file-name* statement
dhcpv6-lease-file-name *name*;

Name is the name of the lease file to use if and only if the server is running in DHCPv6 mode. By default, this is /var/lib/dhcpd/dhcpd6.leases. This statement, like *lease-file-name,* **must** appear in the outer scope of the configuration file. It has no effect if overridden by the **-lf** flag or the **PATH_DHCPD6_DB** environment variable. If *dhcpv6-lease-file-name* is not specified, but *lease-file-name* is, the latter value will be used.

The *local-port* statement
local-port *port*;

This statement causes the DHCP server to listen for DHCP requests on the UDP port specified in *port*, rather than on port 67.

The *local-address* statement
local-address *address*;

This statement causes the DHCP server to listen for DHCP requests sent to the specified *address*, rather than requests sent to all addresses. Since serving directly attached DHCP clients implies that the server must respond to requests sent to the all-ones IP address, this option cannot be used if clients are on directly attached networks...it is only realistically useful for a server whose only clients are reached via unicasts, such as via DHCP relay agents.

Note: This statement is only effective if the server was compiled using the USE_SOCKETS #define statement, which is default on a small number of operating systems, and must be explicitly chosen at compile-time for all others. You can be sure if your server is compiled with USE_SOCKETS if you see lines of this format at startup:

Listening on Socket/eth0

Note also that since this bind()s all DHCP sockets to the specified address, that only one address may be supported in a daemon at a given time.

The *log-facility* statement
log-facility *facility*;

This statement causes the DHCP server to do all of its logging on the specified log facility once the dhcpd.conf file has been read. By default the DHCP server logs to the daemon facility. Possible log facilities include auth, authpriv, cron, daemon, ftp, kern, lpr, mail, mark, news, ntp, security, syslog, user, uucp, and local0 through local7. Not all of these facilities are available on all systems, and there may be other facilities available on other systems.

In addition to setting this value, you may need to modify your *syslog.conf* file to configure logging of the DHCP server. For example, you might add a line like this:

```
local7.debug /var/log/dhcpd.log
```

The syntax of the *syslog.conf* file may be different on some operating systems - consult the *syslog.conf* manual page to be sure. To get syslog to start logging to the new file, you must first create the file with correct ownership and permissions (usually, the same owner and permissions of your /var/log/messages or /usr/adm/messages file should be fine) and send a SIGHUP to syslogd. Some systems support log rollover using a shell script or program called newsyslog or logrotate, and you may be able to configure this as well so that your log file doesn't grow uncontrollably.

Because the *log-facility* setting is controlled by the dhcpd.conf file, log messages printed while parsing the dhcpd.conf file or before parsing it are logged to the default log facility. To prevent this, see the README file included with this distribution, which describes how to change the default log facility. When this parameter is used, the DHCP server prints its startup message a second time after parsing the configuration file, so that the log will be as complete as possible.

The *max-lease-time* statement
max-lease-time *time***;**

Time should be the maximum length in seconds that will be assigned to a lease. The only exception to this is that Dynamic BOOTP lease lengths, which are not specified by the client, are not limited by this maximum.

The *min-lease-time* statement
min-lease-time *time***;**

Time should be the minimum length in seconds that will be assigned to a lease.

The *min-secs* statement
min-secs *seconds***;**

Seconds should be the minimum number of seconds since a client began trying to acquire a new lease before the DHCP server will respond to its request. The number of seconds is based on what the client reports, and the maximum value that the client can report is 255 seconds. Generally, setting this to one will result in the DHCP server not responding to the client's first request, but always responding to its second request.

This can be used to set up a secondary DHCP server which never offers an address to a client until the primary server has been given a chance to do so. If the primary server is down, the client will bind to the secondary server, but otherwise clients should always bind to the primary. Note that this does not, by itself, permit a primary server and a secondary server to share a pool of dynamically-allocatable addresses.

The *next-server* statement
next-server *server-name***;**

The *next-server* statement is used to specify the host address of the server from which the initial boot file (specified in the *filename* statement) is to be loaded. *Server-name* should be a numeric IP address or a domain name. If no *next-server* statement applies to a given client, the address 0.0.0.0 is used.

The *omapi-port* statement
omapi-port *port*;

The *omapi-port* statement causes the DHCP server to listen for OMAPI connections on the specified port. This statement is required to enable the OMAPI protocol, which is used to examine and modify the state of the DHCP server as it is running.

The *one-lease-per-client* statement
one-lease-per-client *flag*;

If this flag is enabled, whenever a client sends a DHCPREQUEST for a particular lease, the server will automatically free any other leases the client holds. This presumes that when the client sends a DHCPREQUEST, it has forgotten any lease not mentioned in the DHCPREQUEST - i.e., the client has only a single network interface *and* it does not remember leases it's holding on networks to which it is not currently attached. Neither of these assumptions are guaranteed or provable, so we urge caution in the use of this statement.

The *pid-file-name* statement
pid-file-name *name*;

Name should be the name of the DHCP server's process ID file. This is the file in which the DHCP server's process ID is stored when the server starts. By default, this is /var/run/dhcpd.pid. Like the *lease-file-name* statement, this statement must appear in the outer scope of the configuration file. It has no effect if overridden by the **-pf** flag or the **PATH_DHCPD_PID** environment variable.

The *dhcpv6-pid-file-name* statement

dhcpv6-pid-file-name *name*;

Name is the name of the pid file to use if and only if the server is running in DHCPv6 mode. By default, this is /var/lib/dhcpd/dhcpd6.pid. This statement, like *pid-file-name,* **must** appear in the outer scope of the configuration file. It has no effect if overridden by the **-pf** flag or the **PATH_DHCPD6_PID** environment variable. If *dhcpv6-pid-file-name* is not specified, but *pid-file-name* is, the latter value will be used.

The *ping-check* statement
ping-check *flag*;

When the DHCP server is considering dynamically allocating an IP address to a client, it first sends an ICMP Echo request (a *ping*) to the address being assigned. It waits for a second, and if no ICMP Echo response has been heard, it assigns the address. If a response *is* heard, the lease is abandoned, and the server does not respond to the client.

This *ping check* introduces a default one-second delay in responding to DHCPDISCOVER messages, which can be a problem for some clients. The default delay of one second may be configured using the ping-timeout parameter. The ping-check configuration parameter can be used to control checking - if its value is false, no ping check is done.

The *ping-timeout* statement
ping-timeout *seconds*;

If the DHCP server determined it should send an ICMP echo request (a *ping*) because the

ping-check statement is true, ping-timeout allows you to configure how many seconds the DHCP server should wait for an ICMP Echo response to be heard, if no ICMP Echo response has been received before the timeout expires, it assigns the address. If a response *is* heard, the lease is abandoned, and the server does not respond to the client. If no value is set, ping-timeout defaults to 1 second.

The *preferred-lifetime* statement
preferred-lifetime *seconds*;

IPv6 addresses have 'valid' and 'preferred' lifetimes. The valid lifetime determines at what point at lease might be said to have expired, and is no longer useable. A preferred lifetime is an advisory condition to help applications move off of the address and onto currently valid addresses (should there still be any open TCP sockets or similar).

The preferred lifetime defaults to the renew+rebind timers, or 3/4 the default lease time if none were specified.

The *remote-port* statement
remote-port *port*;

This statement causes the DHCP server to transmit DHCP responses to DHCP clients upon the UDP port specified in *port*, rather than on port 68. In the event that the UDP response is transmitted to a DHCP Relay, the server generally uses the **local-port** configuration value. Should the DHCP Relay happen to be addressed as 127.0.0.1, however, the DHCP Server transmits its response to the **remote-port** configuration value. This is generally only useful for testing purposes, and this configuration value should generally not be used.

The *server-identifier* statement
server-identifier *hostname*;

The server-identifier statement can be used to define the value that is sent in the DHCP Server Identifier option for a given scope. The value specified **must** be an IP address for the DHCP server, and must be reachable by all clients served by a particular scope.

The use of the server-identifier statement is not recommended - the only reason to use it is to force a value other than the default value to be sent on occasions where the default value would be incorrect. The default value is the first IP address associated with the physical network interface on which the request arrived.

The usual case where the *server-identifier* statement needs to be sent is when a physical interface has more than one IP address, and the one being sent by default isn't appropriate for some or all clients served by that interface. Another common case is when an alias is defined for the purpose of having a consistent IP address for the DHCP server, and it is desired that the clients use this IP address when contacting the server.

Supplying a value for the dhcp-server-identifier option is equivalent to using the server-identifier statement.

The *server-duid* statement
server-duid *LLT* [*hardware-type timestamp hardware-address*] ;

server-duid *EN enterprise-number enterprise-identifier* ;

server-duid *LL* [*hardware-type hardware-address*] **;**

The server-duid statement configures the server DUID. You may pick either LLT (link local address plus time), EN (enterprise), or LL (link local).

If you choose LLT or LL, you may specify the exact contents of the DUID. Otherwise the server will generate a DUID of the specified type.

If you choose EN, you must include the enterprise number and the enterprise-identifier.

The default server-duid type is LLT.

The *server-name* statement

server-name *name* **;**

The *server-name* statement can be used to inform the client of the name of the server from which it is booting. *Name* should be the name that will be provided to the client.

The *site-option-space* statement

site-option-space *name* **;**

The *site-option-space* statement can be used to determine from what option space site-local options will be taken. This can be used in much the same way as the *vendor-option-space* statement. Site-local options in DHCP are those options whose numeric codes are greater than 224. These options are intended for site-specific uses, but are frequently used by vendors of embedded hardware that contains DHCP clients. Because site-specific options are allocated on an ad hoc basis, it is quite possible that one vendor's DHCP client might use the same option code that another vendor's client uses, for different purposes. The *site-option-space* option can be used to assign a different set of site-specific options for each such vendor, using conditional evaluation (see **dhcp-eval (5)** for details).

The *stash-agent-options* statement

stash-agent-options *flag***;**

If the *stash-agent-options* parameter is true for a given client, the server will record the relay agent information options sent during the client's initial DHCPREQUEST message when the client was in the SELECTING state and behave as if those options are included in all subsequent DHCPREQUEST messages sent in the RENEWING state. This works around a problem with relay agent information options, which is that they usually not appear in DHCPREQUEST messages sent by the client in the RENEWING state, because such messages are unicast directly to the server and not sent through a relay agent.

The *update-conflict-detection* statement

update-conflict-detection *flag***;**

If the *update-conflict-detection* parameter is true, the server will perform standard DHCID multiple-client, one-name conflict detection. If the parameter has been set false, the server will skip this check and instead simply tear down any previous bindings to install the new binding without question. The default is true.

The *update-optimization* statement

update-optimization *flag***;**

If the *update-optimization* parameter is false for a given client, the server will attempt a DNS update for that client each time the client renews its lease, rather than only attempting an update when it appears to be necessary. This will allow the DNS to heal from database inconsistencies more easily, but the cost is that the DHCP server must do many more DNS updates. We recommend leaving this option enabled, which is the default. This option only affects the behavior of the interim DNS update scheme, and has no effect on the ad-hoc DNS update scheme. If this parameter is not specified, or is true, the DHCP server will only update when the client information changes, the client gets a different lease, or the client's lease expires.

The *update-static-leases* statement

update-static-leases *flag*;

The *update-static-leases* flag, if enabled, causes the DHCP server to do DNS updates for clients even if those clients are being assigned their IP address using a *fixed-address* statement - that is, the client is being given a static assignment. This can only work with the *interim* DNS update scheme. It is not recommended because the DHCP server has no way to tell that the update has been done, and therefore will not delete the record when it is not in use. Also, the server must attempt the update each time the client renews its lease, which could have a significant performance impact in environments that place heavy demands on the DHCP server.

The *use-host-decl-names* statement

use-host-decl-names *flag*;

If the *use-host-decl-names* parameter is true in a given scope, then for every host declaration within that scope, the name provided for the host declaration will be supplied to the client as its hostname. So, for example,

```
group {
  use-host-decl-names on;

 host joe {
    hardware ethernet 08:00:2b:4c:29:32;
    fixed-address joe.fugue.com;
  }
}

is equivalent to

 host joe {
    hardware ethernet 08:00:2b:4c:29:32;
    fixed-address joe.fugue.com;
    option host-name "joe";
  }
```

An *option host-name* statement within a host declaration will override the use of the name in the host declaration.

It should be noted here that most DHCP clients completely ignore the host-name option sent by the DHCP server, and there is no way to configure them not to do this. So you generally

have a choice of either not having any hostname to client IP address mapping that the client will recognize, or doing DNS updates. It is beyond the scope of this document to describe how to make this determination.

The *use-lease-addr-for-default-route* statement

use-lease-addr-for-default-route *flag*;

If the *use-lease-addr-for-default-route* parameter is true in a given scope, then instead of sending the value specified in the routers option (or sending no value at all), the IP address of the lease being assigned is sent to the client. This supposedly causes Win95 machines to ARP for all IP addresses, which can be helpful if your router is configured for proxy ARP. The use of this feature is not recommended, because it won't work for many DHCP clients.

The *vendor-option-space* statement

vendor-option-space *string*;

The *vendor-option-space* parameter determines from what option space vendor options are taken. The use of this configuration parameter is illustrated in the **dhcp-options(5)** manual page, in the *VENDOR ENCAPSULATED OPTIONS* section.

Setting Parameter Values Using Expressions

Sometimes it's helpful to be able to set the value of a DHCP server parameter based on some value that the client has sent. To do this, you can use expression evaluation. The **dhcp-eval(5)** manual page describes how to write expressions. To assign the result of an evaluation to an option, define the option as follows:

```
  my-parameter = expression ;
```

For example:

```
ddns-hostname = binary-to-ascii (16, 8, "-",
                                 substring (hardware, 1, 6));
```

Reserved Leases

It's often useful to allocate a single address to a single client, in approximate perpetuity. Host statements with **fixed-address** clauses exist to a certain extent to serve this purpose, but because host statements are intended to approximate 'static configuration', they suffer from not being referenced in a littany of other Server Services, such as dynamic DNS, failover, 'on events' and so forth.

If a standard dynamic lease, as from any range statement, is marked 'reserved', then the server will only allocate this lease to the client it is identified by (be that by client identifier or hardware address).

In practice, this means that the lease follows the normal state engine, enters ACTIVE state when the client is bound to it, expires, or is released, and any events or services that would normally be supplied during these events are processed normally, as with any other dynamic lease. The only difference is that failover servers treat reserved leases as special when they enter the FREE or BACKUP states - each server applies the lease into the state it may allocate from - and the leases

are not placed on the queue for allocation to other clients. Instead they may only be 'found' by client identity. The result is that the lease is only offered to the returning client.

Care should probably be taken to ensure that the client only has one lease within a given subnet that it is identified by.

Leases may be set 'reserved' either through OMAPI, or through the 'infinite-is-reserved' configuration option (if this is applicable to your environment and mixture of clients).

It should also be noted that leases marked 'reserved' are effectively treated the same as leases marked 'bootp'.

REFERENCE: OPTION STATEMENTS

DHCP option statements are documented in the **dhcp-options(5)** manual page.

REFERENCE: EXPRESSIONS

Expressions used in DHCP option statements and elsewhere are documented in the **dhcp-eval(5)** manual page.

See Also

dhcpd(8), **dhcpd.leases**(5), **dhcp-options**(5), **dhcp-eval**(5), RFC2132, RFC2131.

Author

dhcpd.conf(5) was written by Ted Lemon under a contract with Vixie Labs. Funding for this project was provided by Internet Systems Consortium. Information about Internet Systems Consortium can be found at **https://www.isc.org.**

Referenced By

dhclient.conf(5), **dhclient.leases**(5), **dhcpctl**(3), **dhcpd-eval**(5), **dhcpd-options**(5), **ltsp-config**(8), **omapi**(3), **omshell**(1)