Dynamic Configuration Server | DCS Evaluation Manual

Publication Date *09/20/06*

License Information

Table of Contents

List of Tables

List of Figures

1

Overview

This chapter briefly introduces the Nominum Domain Caching Server (DCS) engine and this manual.

Introduction

DCS is designed to be a very scalable, flexible DHCP server. It is designed to operate in a service provider network and integrate with the existing systems on the service provider network. With DCS, the administrator has the ability to customize the logic of the lease allocation process. There are several integration points where the logic can be configured. This is covered in more detail later in this document.

Target Audience

This guide is aimed at potential Nominum customers who are interested in evaluating DCS. A good working knowledge of DHCP is expected. Similarly, proficiency in the various supported operating systems and tools is assumed. The evaluator is assumed to have a general familiarity of the configuration of DCS based on the documentation.

About this Manual

This document is intended to serve as a guide to assist in the process of evaluating Nominum's Dynamic Configuration Server (DCS). While this document provides a look at many of DCS' features, it is not intended to be a definitive or comprehensive guide. Rather, it is intended to provide some guidance towards getting DCS up and running quickly and seeing some of the advantages that DCS provides over other DHCP servers.

Chapter 1, *Overview,* introduces the Nominum Domain Caching Server (DCS) engine and this manual.

Chapter 2, *Lab Setup,* describes the lab setup and the useful tools for evaluating DCS.

Chapter 3, *DCS Overview,* describes fundamental DCS concepts and how they relate to DHCP.

Chapter 4, *Testing DCS,* details suggested methodology to test DCS. It covers the basics such as doing the initial configuration and testing some of the more common configurations for DCS. These tests assume a network topology similar to the one described in section.

Chapter 5, *Embedded Python,* is a brief overview of the Python environment within DCS.

Chapter 6, *Client and Packet Objects,* describes the client and packet objects that are accessible by Embedded Python and have a great deal of information that is accessible to the Python functions.

Appendix A, *DCS Configuration Script,* contains a script that can be used to populate DCS with the configuration data described above. This is a simple shell script which calls the nom_tell utility.

Appendix B, *DCS Test Plan,* provides a concise plan for testing and evaluating DCS.

Typographical Conventions

Table 1-1 summarizes the typographical conventions used in this manual.

bold font	Resource names, command names, menu selections, keywords and system facilities.
italic font	Filenames, program names, domain names, UNIX pathnames and Uniform Resource Locators (URLs) such as *http://www.nominum.com*.
`constant width font`	Instances of resources, and excerpts from code and configuration files.
`constant width bold font`	User-provided input for interactive sessions or scripts.
`constant width italic`	Variables for which a context-sensitive substitution should be made.
%	The command-line prompt for interactive sessions.
#	The prompt for the **root** user.

Table 1-1 Typographical conventions

2

Lab Setup

This chapter describes the lab setup and the useful tools for evaluating DCS.

Lab Setup

In order to test all of the major features of DCS, it is important to have two machines to run DCS in failover mode. In addition, it is useful to have at least one router to forward DHCP traffic to the DCS servers. The diagram below shows a sample configuration that will allow for testing of all of the scenarios described below. In addition, if load testing is required, a traffic generation tool may also be connected to the network.

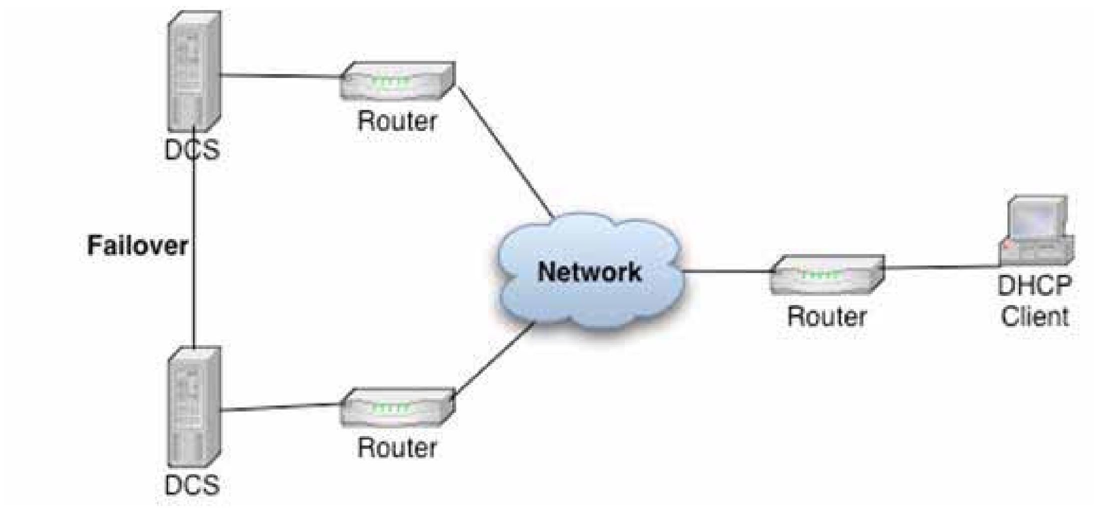

Figure 2-1 Lab Setup

Tools

In order to test DCS, several tools can be used. Some tools, such as nom_tell and the Engine Administration Console (EAC) are included with DCS or can be obtained from Nominum. In addition, other tools can be used; In order to test the throughput and latency of DCS, traffic generators by vendors such as Spirent and Agilent can be used. This assumes that the evaluator has access to these tools. Ad-hoc DHCP requests can be generated using DHCP clients such as Windows or Unix/Linux systems, Cable/DSL Modems or other devices.

Further, the following tools may come in handy:

- Tcpdump, snoop, ethereal or similar tools for capturing and inspecting packets;
- dhcperf for simulating loads in a test situation;
- Programmable UNIX shells such as sh, bash, csh;
- Scripting tools such as Python, Perl etc.

Evaluation Goals

Depending on the requirements of the individual organization, the goals of the evaluation may differ. In general, the following represent a reasonable subset of requirements to be tested during evaluation:

Configuration using the following methods:

- nom_tell
- Engine Administration Console (EAC).

DCS Features

- Allocation of IP addresses based on network topology
- DCHP Failover
- Events and Statistics
- Selection of DCS Pools based on criteria determined by the evaluator. Some common criteria are:
  - Mac Address
  - DCHP Relay information (option 82)

Before You Begin

Complete ISC DHCP configurations can be imported using the “isc2dcs” utility, while new configurations can be created using the DCS configuration file which is then compiled into DCS or by the Nominum Command Channel. Since DCS implements similar functionality in different ways, some complex ISC DHCP configuration files may not be completely migrated and may need some additional work to complete. Note that the configuration file compilation is a one time only activity. If no migration is necessary, there is no need to create a configuration file. All examples in this document use the Nominum Command Channel and assume that no configuration file was used. Please refer to the “Nominum Dynamic Configuration Server Administrator’s Guide” for more details. This evaluation guide assumes that you have already determined which networks and address ranges will be used for the testing. In order to test DHCP failover, two machines are required to run DCS. They do not need to be running the same operating system but must both be DCS supported platforms. Note: it is important to isolate the DCS test network from other, production networks so that DCS does not receive requests from, and potentially allocate addresses to, production clients.

Checklist:

Item	Required?
DCS Software	Yes
Supported Hardware/Software platform (X2 for failover testing)	Yes
Knowledge of the Lab network layout	Yes
ISC DHCP configuration file	No
DHCP Clients	Yes
DHCP Traffic Generator	No
DCHP Documentation	Yes

Table 2-1 Checklist

3

DCS Overview

This chapter describes fundamental DCS concepts and how they relate to DHCP.

DCS Concepts

Nominum DCS is a highly scalable DHCP server with support for DHCP failover and load sharing. There are six main configuration objects within DCS, these will be described in more detail below. DCS uses object based configuration. All configuration is applied to one or more objects within DCS. Objects are also hierarchical within DCS. As each object is described below, its place in the hierarchy will also be described. Any configuration applied to an object will be inherited by its subordinate objects unless specifically overridden by configuration at a lower level. The hierarchy model is shown below:

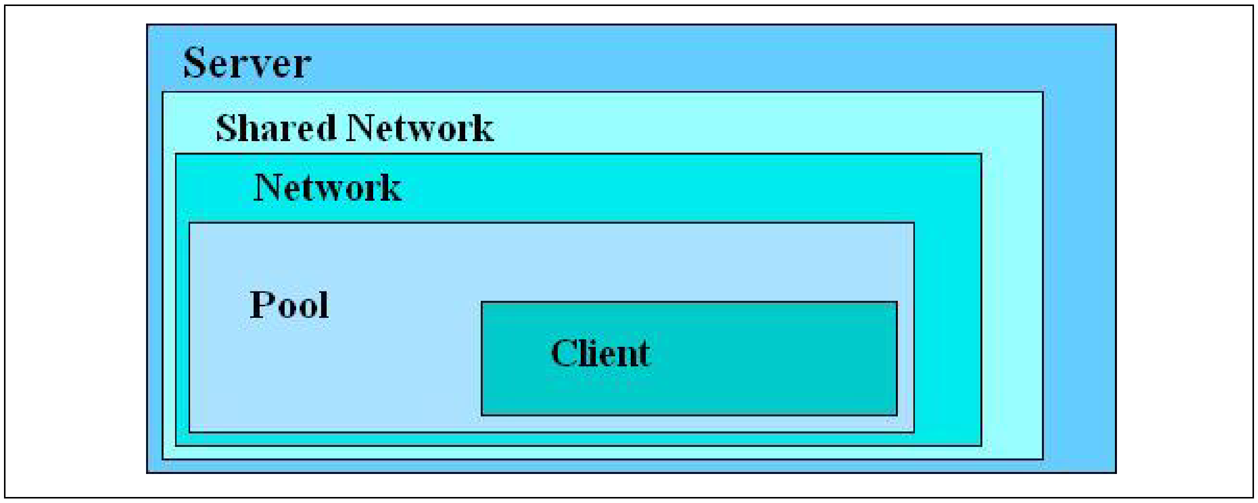

Figure 3-1 Hierarchy Model

Command Channel Structure

DCS uses the Nominum Command Channel protocol in a very structured, object-oriented way. Every command will have the following components:

- Service Name – This specifies which DCS server the command will be sent to. This service name is resolved in the /etc/channel.conf file.
- Command Type—This will always be ‘ccdb’
- Object Type (objtype)—The object that the command will affect (i.e. server, pool, network, etc.)
- Method—The type of operation to be performed (create, delete, list, get, etc.)
- Name—The name of the particular object to be affected. Name applied to all methods except ‘list’.

Other fields may be required depending on the object and the action being performed. For example, Section 5.2.1 gives examples of Command Channel operations using nom_tell.

Options and Parameters

Configuration applied to the objects described below are Options or Parameters. Options refers to DHCP Options only. These can be standard options that are pre-defined within DCS or custom options that have been described by the administrator. Parameters control non-DHCP option configuration of DCS or can be used to set defaults for some options. For example, Failover is configured using parameters and Parameters can be used to set up a default lease time.

The Server Object

The Server object contains all configuration that pertains to the server as a whole. The Server object is at the top of the object hierarchy and any configuration done at the Server level will apply to all objects on the DCS server. The Server object will normally be used to specify default values such as default Domain Names and default lease times. There are some configuration parameters that are specific to the DCS server as a whole such as the location of the Embedded Python function file.

The Shared Network Object

A Shared Network is a grouping of networks that are on the same physical interface on a router or layer 3 switch. In this situation, only the IP address of the primary interface of the router/switch will be placed into the GIADDR field of the DHCP request. A Shared Network tells DCS that all networks in the Shared Network are related to the same GIADDR. The Shared Network object is above the Network object in the hierarchy.

The Network Object

The Network object is used to define the network topology and will ensure that DCS only offers addresses to clients that are correct for the subnet that they are on. A network object must be created for any subnet that DCS will allocate IP addresses from. Options and parameters can be assigned to a Network and these will be inherited by objects below it.

Configuration Options

This section briefly examines some of DCS' configuration options with a few specific examples for reference. The decision regarding which options to implement is left up to each administrator. Remember that since DCS can make changes without restarting, removing (or adding) options has no effect on the performance or operation of the server although these changes can affect how DCS responds to future requests if the configuration is changed.

Most DHCP servers requires a configuration file to be able to operate. While DCS can use a configuration file to get started, all other configuration is done 'live' using the Nominum Command Channel. The Command Channel can be accessed by nom_tell, the Command Channel API or Nominum EAC. Please refer to the DCS documentation or EAC documentation for more details.

The Command Channel

Nominum provides a proprietary protocol called the command channel. An implementation of a command channel client, called nom_tell is also bundled with DCS. nom_tell commands take the form:

```
# nom_tell service what
```

Where service is the specific service to be managed (in this case it is always '`dcs`') and what provides details about what exactly it is we're trying to do. nom_tell commands can be used to get information about the server as it's running. However, it can also be used to change the operation and configuration of the server while it is running.

A basic use of nom_tell is to determine the version of DCS. The command:

```
# nom_tell dcs version
```

will return the current version of the server. The server can be stopped in the same manner.

nom_tell commands for DCS all take the same form. The user must specify the command class, the type of object, the method and the configuration information. This is described in more detail below.

While there are numerous nom_tell commands that are available to a DHCP administrator, we will focus on a few that are reasonably common. For a complete list of nom_tell commands, please refer to the "Nominum Dynamic Configuration Server Administrator's Manual".

Show Configuration

Nom_tell can be used to show the configuration of any object in DCS. The following most common objects' configuration can be displayed with nom_tell:

1. Server
2. Network
3. Shared Network
4. Pool
5. Client
6. Lease
7. Option Set
8. Functions

9. Events

10. Statistics

Nom_tell example

As an example of the robustness of DCS, below is an example of some of the operations we've already discussed. The following examples show the information that can be displayed:

List DCS Pools:

```
# /usr/local/nom/sbin/nom_tell dcs ccdb objtype=pool method=list
request:
{
    type => 'ccdb'
    objtype => 'pool'
    method => 'list'
}
response:
{
    type => 'ccdb'
    objtype => 'pool'
    method => 'list'
    list => (
        {
            name => 'example-public-pool'
        }
        {
            name => 'example2-public-pool'
        }
    )
    status => 'success'
}
```

To get information on a particular pool:

```
# /usr/local/nom/sbin/nom_tell dcs ccdb objtype=pool method=get \
   name=example-public-pool
request:
{
    type => 'ccdb'
    objtype => 'pool'
    method => 'get'
    name => 'example-public-pool'
}
response:
{
    type => 'ccdb'
    objtype => 'pool'
    method => 'get'
```

```
        name => 'example-public-pool'
        ranges => (('66.218.71.1' '66.218.71.250'))
        attributes => ('predefined')
        last_stored_time => '1149024313.347098'
        generation => '1'
        range_accounting => (
            {
                range_start => '66.218.71.1'
                range_end => '66.218.71.250'
                range_used => '0'
                range_active => '0'
                my_addr_free => '250'
                peer_addr_free => '0'
                bootp => '0'
            }
        )
        status => 'success'
    }
```

A Network's configuration can be displayed using nom_tell:

```
    # /usr/local/nom/sbin/nom_tell dcs ccdb objtype=network method=get \
       name=example.nominum.com
    request:
    {
        type => 'ccdb'
        objtype => 'network'
        method => 'get'
        name => 'example.nominum.com'
    }
    response:
    {
        type => 'ccdb'
        objtype => 'network'
        method => 'get'
        name => 'example.nominum.com'
        netaddr => '66.218.71.0'
        netmask => '255.255.255.0'
        pools => ('example-public-pool')
        generation => '0'
        optionsets => ('option_set_2')
        options => ('dhcp' ('subnet-mask' '255.255.255.0') ('routers'
       ('66.218.71.253')) ('domain-name' 'example.nominum.com') ('broad-
       cast-address' '66.218.71.255'))
        status => 'success'
    }
```

To see information on a DHCP client that DCS knows about type the following. This will retrieve a client name of client1. The client could also be looked up by its MAC address or DHCP Client Identifier.

```
# /usr/local/nom/sbin/nom_tell dcs ccdb objtype=client method=get \
   name=client1
request:
{
    type => 'ccdb'
    objtype => 'client'
    method => 'get'
    name => 'client1'
}
response:
{
    type => 'ccdb'
    objtype => 'client'
    method => 'get'
    name => 'client1'
    dhcp_client_identifier => <binary:01001122334455>
    predefined => '1'
    dhcp_client_id_synthesized => '1'
    hardware_address => {
        type => 'ethernet'
        bytes => '00:11:22:33:44:55'
    }
    reverse_dns => '0'
    reverse_rcode => '255'
    forward_dns => '0'
    forward_rcode => '255'
    bootp => '0'
    last_stored_time => '1149024923.012162'
    generation => '2'
    status => 'success'
}
```

4

Testing DCS

This chapter details suggested methodology to test DCS. It covers the basics such as doing the initial configuration and testing some of the more common configurations for DCS. These tests assume a network topology similar to the one described in section.

In order to run the test, the test network must match the networks, ranges and DHCP options below. If the lab environment is different, the examples below must be changed appropriately to conform to the actual environment.

Basic Functionality

The easiest way to get started with DCS is to start with a blank database and make all configuration using the Nominum Command Channel. To start DCS with a blank database all that is required is to start DCS from the system init scripts as follows:

```
# /etc/init.d/dcs start
```

This will start DCS with a blank configuration and create a database in the default location of:

```
# /var/nom/dcs/dcsd-conf.d
```

The simplest network configuration for testing DCS is to configure a few subnets, pools, option sets and ranges. These can be configured into DCS as follows. Note that all changes are made live on the server and are active immediately. All of the commands below are included in the file scripts.txt which is included with this document. Note that once an object has been created in DCS, can be referred to by its name in subsequent commands as can be seen below when creating shared networks.

Create Network:

```
# nom_tell dcs ccdb objtype=network method=create name=net1 \
   netaddr=10.1.1.0/24

# nom_tell dcs ccdb objtype=network method=create name=net2 \
   netaddr=10.1.2.0/24

# nom_tell dcs ccdb objtype=network method=create name=net3 \
   netaddr=10.1.3.0/24

# nom_tell dcs ccdb objtype=network method=create name=net4 \
   netaddr=10.1.4.0/24
```

Create Shared Network:

```
# nom_tell dcs ccdb objtype=sharednet method=create name=shared1

# nom_tell dcs ccdb objtype=sharednet method=add_network \
   name=shared1 network=net3

# nom_tell dcs ccdb objtype=sharednet method=add_network \
   name=shared1 network=net4
```

Create Pool

```
# nom_tell dcs ccdb objtype=pool method=create name=pool1

# nom_tell dcs ccdb objtype=pool method=create name=pool2

# nom_tell dcs ccdb objtype=pool method=create name=pool3
```

Add Range to a Pool

```
# nom_tell dcs ccdb objtype=pool method=add_range \
   name=pool1 range=('10.1.1.10', '10.1.1.100')

# nom_tell dcs ccdb objtype=pool method=add_range \
   name=pool2 range=('10.1.2.10', '10.1.2.100')

# nom_tell dcs ccdb objtype=pool method=add_range \
   name=pool3 range=('10.1.3.10', '10.1.3.100')

# nom_tell dcs ccdb objtype=pool method=add_range \
   name=pool3 range=('10.1.4.10', '10.1.4.100')
```

Create Option Sets:

```
# nom_tell dcs ccdb objtype=optionset method=create name=basicopts

# nom_tell dcs ccdb objtype=optionset method=setfield name=basicopts \
  options='(dhcp (subnet-mask 255.255.255.0) (domain-name company.com)
  (domain-name-servers (10.1.2.251 10.12.12.1)))'

# nom_tell dcs ccdb objtype=optionset method=create name=net1opts

# nom_tell dcs ccdb objtype=optionset method=setfield \
  name=net1opts options='('dhcp' ('routers' ('10.1.1.1')))'

# nom_tell dcs ccdb objtype=server method=add_optionset name=basicopts

# nom_tell dcs ccdb objtype=optionset method=create name=net2opts

# nom_tell dcs ccdb objtype=optionset method=setfield \
  name=net2opts options='(dhcp (routers (10.1.2.1)))'

# nom_tell dcs ccdb objtype=optionset method=create name=net3opts

# nom_tell dcs ccdb objtype=optionset method=setfield \
  name=net3opts options='(dhcp (routers (10.1.3.1)))'

# nom_tell dcs ccdb objtype=optionset method=create name=net4opts

# nom_tell dcs ccdb objtype=optionset method=setfield \
  name=net4opts options='(dhcp (routers (10.1.4.1)))'
```

Assign Option Sets to Networks:

```
# nom_tell dcs ccdb objtype=network method=add_optionset \
  name=net1 optionset=net1opts

# nom_tell dcs ccdb objtype=network method=add_optionset \
  name=net2 optionset=net2opts

# nom_tell dcs ccdb objtype=network method=add_optionset \
  name=net3 optionset=net3opts

# nom_tell dcs ccdb objtype=network method=add_optionset \
  name=net4 optionset=net4opts
```

Assign Server Parameters:

Server parameters can be used to set configuration that applies to the server itself or to set global defaults. Here we will specify a minimum, maximum and default lease times. Since DCS will normally respect the client's request for a specific lease time, some limits need to be set to protect against abnormally long or short lease times. For any requested lease time that is less than the minimum, DCS will assign the minimum lease time and for any requested lease that is over the maximum, DCS will assign the maximum lease time. The default is used if the client does not ask for a specific lease time. These values can be set as follows:

```
# nom_tell dcs ccdb objtype=server method=setfield max_lease_time=86400

# nom_tell dcs ccdb objtype=server method=setfield min_lease_time=3600

# nom_tell dcs ccdb objtype=server method=setfield \
  default_lease_time=7200
```

Once the configuration above has been entered, DHCP clients can be activated on the test subnet. For each client, verify that the correct IP address has been allocated and that the assigning server is the DCS server. Whenever DCS receives a DHCP packet, it will log this in syslog (if the 'info_msg_logging' parameter on the Server objet is set to true).

Once DCS has been configured and clients are requesting leases from DCS, there are several ways to verify that DCS is seeing the requests, see how it is answering them and querying DCS for lease allocations etc.

Whenever DCS receives a DHCP packet, it will log this in syslog. The full four-way handshake will be written to syslog as well as any other messages as to why DCS may not offer an address to a client. Below is an example of a DHCPREQUEST received by DCS:

dcsd[543]: [ID 356978 local3.info] info: DHCPREQUEST (renew or rebind) for 10.1.1.106 from 00:03:48:39:b4:c0 (01:00:03:48:39:b4:c0) via dmfe0, client name: client-3e67816f.5bd10.73a0, tid: 11, xid: 0x6269097c

It is possible to see how many leases have been assigned by DCS from a particular pool by examining the pool that the addresses are being assigned from As below:

```
# nom_tell dcs ccdb objtype=pool method=get name=pool1
```

Which will return the following output:

```
response:
{
    type => 'ccdb'
    objtype => 'pool'
    method => 'get'
    name => 'pool1'
    ranges => (('10.1.1.10' '10.1.1.100'))
    last_stored_time => '1150734810.712174'
    generation => '3'
    range_accounting => (
        {
            range_start => '10.1.1.10'
```

```
               range_end => '10.1.1.100'
               range_used => '0'
               range_active => '0'
               my_addr_free => '91'
               peer_addr_free => '0'
               bootp => '0'
           }
       )
       status => 'success'
   }
```

As can be seen above, there are zero addresses in use. The DCS server has 91 addresses available. Since the pool is not part of a failover relationship, there are no addresses being allocated by the failover peer (peer_addr_free).

To look for a particular DHCP client and lease in the DCS database, the client or lease can be looked up by its name or client Identifier. The client Identifier is normally the MAC type pre-pended to the MAC address (e.g. 01001122334455 for an Ethernet device) The lease object will contain the IP Address, DHCP Client Identifier, state, Lease Expiry Time, Reserved. The Lease Expiry time is in Epoch time (number of seconds since January 1st, 1970). The reserved bit will indicate whether the lease is associated with a fixed client.

The state can be one of the following values:

Code	**State**
1	free
2	active
3	expired
4	released
5	abandoned
6	reset
7	backup
8	freeorbackup

Table 4-1 State Codes

Advanced Functionality

In the advanced configuration section, we will set up DHCP Failover, Events and Statistics and Embedded Python.

DHCP Failover

Nominum DCS implements the draft standard for DHCP failover that has been defined by the IETF. This standard allows for ‘active-active’ failover where both servers in a failover relationship allocate addresses to clients and implement a load sharing algorithm. More detail on the failover draft standard can be obtained from the IETF or the DHCP Handbook.

In DCS, failover is done at the server, shared network, network or pool level. For pool level failover, each pool that is involved in a failover relationship is shared between two DCS servers. It is possible for a different pools on a DCS server to be part of different failover relationships with different DCS servers. For server level failover, the failover is done for all pools on the server with another DCS server. This allows for simplicity of configuration, the administrator can set up the failover relationship once for a pair of servers and not have to ensure that all pools are configured for failover or that new pools get added to a failover relationship.

In order to configure failover, a failover relationship must be created and the pool(s), shared network(s), networks(s) or server assigned to it. When creating a failover relationship, the two DCS servers must be defined. This can be done in DCS via the Command Channel as follows:

```
# nom_tell dcs ccdb objtype=failover method=create \
   name=failover1 local_fqdn=primary.nominum.com \
   peer_fqdn=secondary.nominum.com role=primary
```

The failover relationship can then be associated with the pool as follows:

```
# nom_tell dcs ccdb objtype=pool method=set_failover_relationship \
   name=poolname failover_endpoint=foo
```

Or set up for the server as follows:

```
# nom_tell dcs ccdb objtype=server \
   method=set_failover_relationship failover_endpoint=foo
```

Note that when setting up a failover relationship, it is important that the system clocks of the two DCS servers be as closely synchronized as possible. Where possible, NTP should be used to keep the clocks synchronized.

Once the failover relationship has been configured above, a client can be booted and observed to get an IP address from one of the DCS servers. Once the client has its IP address, unplug the network cable of the DCS server that allocated the lease, release and then renew the IP address of the client. The client should now get a lease from the other DCS server.

On the server that is still connected to the network, it is possible to check the state of the failover relationship using the following command:

```
# nom_tell dcs ccdb objtype=failover method=getfield name=foo
```

The output will have an entry for state which should have the value of 3 which is Communications Interrupted. The table below lists all possible values for the failover state.

State Code	**Failover State**
1	startup
2	normal
3	communications-interrupted
4	partner-down
5	potential-conflict
6	recover
7	pause
8	shutdown
9	recover-done
10	resolution-interrupted
11	conflict-done
12	recover-wait

Table 4-2 Failover state values

The connected DCS server can be put into the Partner-Down mode as follows:

```
# nom_tell dcs ccdb objtype=failover method=partner_down name=foo
```

Where ‘foo’ is the name of the failover relationship.

This tells the connected DCS server that its peer is not operational. Once this is done, release and renew the client, it will be seen that the client still gets an IP address from the connected DCS server but the lease time is now normal.

Reconnect the other DCS server to the network and wait a few moments. Check the state on each server and they should each show state 2 (Normal). Also, run the following commands on the server that was disconnected to see that the lease information from the other server has been updated via the failover protocol:

To retrieve the Lease object:

```
# nom_tell dcs ccdb objtype=lease method=get \
   dhcp_client_identifier=<identifier>
```

This will display the lease information for the client.

Dynamic DNS

DCS is able to automatically update a DNS server with the A and PTR records for a client after it has been assigned a lease. This is done using the Dynamic DNS protocol (DDNS). The configuration below assumes that a DNS server has been correctly set up to allow DDNS updates from the DCS server(s). The configuration of the DNS Server is outside the scope of this document.

By default, DCS is configured to not do DDNS updates. To enable DDNS, DCS must be configured to do DDNS and a parent domain name must be set for updates. DDNS can be turned on as follows:

```
# nom_tell dcs ccdb objtype=server method=setfield do_ddns=true
```

The parent domain name will be appended to every host name sent by the client and is configured as follows:

```
# nom_tell dcs ccdb objtype=server method=setfield \
  client_domain_name=company.com
```

This parameter can also be set on the Pool, Network, Shared Network, and Client. Settings that are on more specific objects will over ride parameters set at a higher level. For example, a Client Domain Name set at the pool level will over ride one set at the server level.

For security reasons, it is recommended that Transaction Signatures (TSIG) be used to secure DDNS updates. TSIG must also be configured on the DNS server in order for this to work which is outside the scope of this document. To configure TSIG signed updates in DCS, a TSIG key must be defined and the key must be associated with a DNS zone that updates will be sent to. The key can be set up as follows:

```
# nom_tell dcs ccdb objtype=key method=create algorithm='hmac-md5' \
  name='key23' secret='896aa6483d232eaaa1c29eed26f5429a'
```

The TSIG key can then be associated with a zone as follows:

```
# nom_tell dcs ccdb objtype=zone method=create name='example.com' \
  server='192.168.2.3' key='key23'
```

Once this is done, all DDNS updates to that zone will be signed with the key. It is recommended that a key generator be used to create the TSIG key and that they not be created by hand.

Statistics

Statistics allow DCS to capture data on the operation of DCS. A statistic consists of potentially four criteria. A statistic will have a type, interval, bucket and optionally a formula. There are two separate types of statistics; counter and quantity. A counter is simply incremented every time a new instance is noted and will only increase. Quantity statistics measure the size of an aggregation of data and the values may rise and fall.

Statistics allow for the monitoring of historical data using Buckets. Buckets are snapshots of the data over a certain amount of time. The user specifies the interval and the number of buckets. Each bucket will contain data for the length of time of the interval. For example, and interval of 100 with 50 buckets means 50 buckets of 100 seconds each. This means that the statistics will be kept for 5000 seconds (50*100). When the statistic is created, DCS starts to capture data and after the first interval expires, it moves this data into a new bucket and starts a new one. Once the maximum number of buckets has been created, the oldest bucket will be deleted to make room for the next bucket.

For quantity statistics, the number of instances is reset for each bucket. For counter statistics, the number is not reset for each bucket, the value for each bucket is simply the value of the statistic at the time the bucket is created.

DCS also has the ability to perform calculations on statistics. DCS can perform the following calculations on a statistic:

Type of Formula	Data Returned
None	Value of the current bucket
Average	Average of all the buckets
First Derivative	Value of the first derivative computed between the current and previous bucket
Difference	Difference between the values of the current and previous buckets

Table 4-3 Calculations on Statistics

Below is a simple example of a statistic. This statistic will track the number of packets received by DCS within a 10 minute period. It will keep track of ten buckets and will sample the value of the bucket at the end of the bucket. In most statistics it would be required to enter an object type to sample but since the packet_in statistic only applies to the server, this is not necessary.

```
# nom_tell dcs ccdb objtype=stats method=create \
   name=packets_in3 bucket_type=end interval=600 \
   statkind=packets_in sampled_object_type=server buckets=10
```

Once the statistic has been created, it will continue to track the data. This data can then be accessed over the command channel as follows:

```
# nom_tell dcs ccdb objtype=stats method=get name=packets_in3
```

The statistics types that can be gathered by DCS are listed in the table below:

Statistic Field Name	Description	Type	Scope
activeleases	Number of active leases	Quantity	Range
backupleases	Number of backup leases	Quantity	Range
bad_dhcp	Bad DHCP packets	Counter	Server
bndack	Number of BNDACKs a failover end-point recieved	Counter	Failover Endpoint
bndackrej	Number of BNDACKs that have been rejected by a failover endpoint	Counter	Failover Endpoint
bndupd	Number of BNDUPDs a Failover end-point recieved	Counter	Failover Endpoint
bootp	Bootp packets	Counter	Server
bootp_not_honored	Bootp packets not honored	Counter	Server
communications_notok	Number of times communications have gone 'not ok' in a failover relationship	Counter	Failover Endpoint
communications_ok	Number of times communications have gone 'ok' in a failover relationship	Counter	Failover Endpoint
declines	Number of declines	Counter	Server, SharedNetwork, Network, Pool
declines_not_honored	Number of declines not honored	Counter	Server
discover	Number of discovers	Counter	Server, SharedNetwork, Network, Pool
discover_no_leases	Discovers not honored because there were no available leases	Counter	Server
discover_no_permitted_pools	Discovers not honored because there were no permitted pools	Counter	Server
discover_not_honored	Number of discovers not honored	Counter	Server, SharedNetwork, Network
freeleases	Number of free leases	Quantity	Range
ignored	Number of ignored (due to already active transaction) packets	Counter	Server
inform	Number of informs	Counter	Server, SharedNetwork, Network, Pool

Table 4-4 DCS statistic types

Statistic Field Name	Description	Type	Scope
inform_not_honored	Number of informs not honored	Counter	Server, SharedNetwork, Network, Pool
notdhcpbootp	Number of packets that were not DHCP or BOOTP	Counter	Server
packets_in	Number of packets (of any type)	Counter	Server
packets_out	Number of packets out (of any type)	Counter	Server
releases	Number of DHCPRELEASES the server has received	Counter	Server, SharedNetwork, Network, Pool
reqrir	Number of requests (renew/rebind/initreboot)	Counter	Server, SharedNetwork, Network, Pool
reqrir_not_honored	Number of requests (renew/rebind/initreboot) not honored	Counter	Server, SharedNetwork, Network, Pool
reqsel	Number of requests (selecting)	Counter	Server, SharedNetwork, Network, Pool
reqsel_not_honored	Number of requests (selecting) not honored	Counter	Server, SharedNetwork, Network, Pool
request_not_honored	Number of requests not honored	Counter	Server
transaction	Number of transactions	Counter	Server
transaction_duration	Transaction duration	Quantity	Server
transaction_expire	Number of times a transaction has expired due to timeout	Counter	Server
transaction_queue_depth	Depth of the transaction queue	Quantity	Server
usedleases	Number of in-use (not free or backup) leases	Quantity	Range

Table 4-4 DCS statistic types

Events

Events provide a way for DCS to send notices of certain occurrences in DCS to external processes or files. Events are sent over specific channels that are configured at create time. This flexibility allows different events to be handled differently. Events can be written to a file and also be sent over a network socket to an external listening application. There are four classes of Events for DCS; Lease, Failover, Server and Trigger.

When setting up an Event, there are two required steps. First, the Event channel must be set up with the facility it will be sent to (file or socket). The second step is to subscribe to a specific Event. Below is a simple example of setting up an event to write an entry to a file every time a lease is assigned by DCS:

First we set up the Event Channel:

```
# nom_tell  dcs ccdb objtype=eventchannel name=leaseevent \
   method=create channel_type=file path_name='/tmp/newleases.txt' \
   persistent=true
```

Note that unless the persistent=true section is included, the Event Channel will be deleted if DCS is restarted.

Next we subscribe to an Event:

```
# nom_tell dcs ccdb objtype=eventchannel name=leaseevent method=sub-
   scribe events=lease.allocated
```

The following Events can be subscribed to in DCS:

lease.discover	lease.released	server.event-overflow
lease.offer	failover.binding-active	server.reload
lease.request	failover.binding-expired	server.newclient
lease.allocated	failover.binding-released	server.keepalive
lease.confirm	failover.binding-abandoned	server.shutdown
lease.denied	failover.binding-reset	lease.bootprequest
lease.decline	failover.binding-backup	lease.bootpreply
lease.renew	failover.binding-free	trigger.notify
lease.renewed	failover.binding-update	failover.failover-recvstate
lease.expired	failover.failover-poolreq	lease.bootp-not-honored
lease.abandoned	failover.failover-connect	server.database-changed
lease.reclaimed	lease.discover-not-honored	
lease.inform	failover.failover-poolresp	
server.transaction-expired	failover.failover-state	
failover.binding-rejected	failover.failover-disconnect	

Table 4-5 DCS subscribable events

Triggers

A trigger on a statistic will allow DCS to proactively send a notification when a particular threshold is reached. A trigger is sent using an Event. Triggers are of two types; Threshold and 'Every'. A threshold trigger is activated when a statistic exceeds a defined value while an 'Every' trigger is activated every time DCS moves from one bucket to another.

A Trigger can be combined with an Event to set up powerful notification mechanisms. For the example below, we can see how to write an event to a file when the an address range reaches more than 90% utilization:

First we create a statistic to keep track of the number of used leases:

```
# nom_tell dcs ccdb objtype=stats method=create name=pool1_util2 \
  bucket_type=end interval=600 statkind=usedleases \
  sampled_object_type=poolrange sampled_object_name=public-pool \
  range='(10.1.1.100 10.1.1.150)' buckets=10
```

Next we create the Event Channel:

```
# nom_tell dcs ccdb objtype=eventchannel name=pool1_lease_util \
  method=create channel_type=file path_name='/tmp/events' \
  persistent=true
```

Now we create the Trigger:

```
# nom_tell dcs ccdb objtype=trigger method=create name=pool1_over90pct \
  event_channel_name=pool1_lease_util trigger_type=threshold \
  threshold_value=90% threshold_over=True
```

This will cause DCS to write to a file whenever the range of addresses becomes more than 90% utilized. It is important to note that DCS does not write this event to the file immediately. For performance reasons, DCS writes the event to a buffer that is 16K in size. The contents of the buffer are not written to the file until the buffer is full. For real time notification, the event must be sent to an external application over a network socket using the Command Channel over TCP.

5

Embedded Python

DCS makes use of the Python language. The following is a brief overview of the Python environment within DCS.

Overview

Embedded Python allows the user to control how leases are allocated by DCS. It can be used to leverage any part of a DHCP request, as well as other data within DCS, to determine policies to deal with the client. This may include denying the client an address, allocation of specific addresses or DHCP options or applying logic when renewing an IP address.

Policies in DCS are enforced by using Pools. A Pool may contain one or more ranges of IP addresses. Pools can also have Attributes assigned to them. When a DHCP request is received, a client will potentially be able to get an IP address in one or more Pools. These 'possible' Pools are determined by the network topology. This is determined by the gateway address (GIADDR) contained in the DHCP request.

Once DCS has determined which Pools are possible for a client, it must decide which Pools are 'permitted' based on the logic configured in DCS. The 'permitted' list is determined by a Filter function in DCS which uses Attributes to make its policy decisions.

When a packet is received by DCS, DCS will call all functions that have been defined by the administrator. These functions are defined in more detail below.

Embedded Python Functions

Within Embedded Python, there are eight types of functions that can be used. They are described below. They are listed in the order in which DCS will execute them. Note that if there is more than one function of a particular type, the order in which they will be executed by DCS can not be guaranteed. The last three types of functions; Lease In Use, Lease No Longer Used and Lease Bind Update Received are not part of the initial processing of a packet but are called after the client has received its address or been denied.

Packet Receipt Function

The Packet receipt function is called as soon as a packet is received by DCS and before DCS either retrieves a matching client from the database or creates a client object for it. The only input into the Packet Receipt function is the packet object. The Packet Receipt function is normally used to decide whether to continue processing the packet or discard it. Attributes are not normally set in this type of function because there is no client object yet associated with the transaction and attributes cannot be associated with packets. A sample application for a Packet is to search for something in the packet and decide if it should be dropped. For example, if the administrator wanted to drop all requests from Microsoft RAS servers, the Packet Receipt function could search for the string ‘RAS’ in the DHCP Client Identifier option in the packet and discard packets that have it.

Client Lookup Function

A Client Lookup function is called after all Packet Receipt functions have been called but before any attributes are assigned. A Client Lookup function can be used to modify a client associated with the DHCP request before any subsequent processing such as attribute assignment. In addition to the client, the function can also modify or create other objects such as pools, networks etc.

NOTE Use caution when attempting to modify or create objects such as networks or pools in embedded python. Such network topological and policy elements are usually defined statically and do not make sense to be generated on the fly.

Association Function

The association function is used to associate attributes (text strings) with a client. These attributes can then be used in other functions to make decisions on how a request is processed. All subsequent functions can use the attributes assigned here.

Attribute Complete Function

These functions are executed after all attribute assignments have been done but before any other processing. This function can modify or create other objects such as pools, networks etc.

Filter Function

The Filter function decides which pools a client may get an address from (if any). The filter function can use any attributes that have been associated with the client or any part of the client or packet objects to make this determination. As has been described above, a Filter function will normally be used to modify the list of permitted pools that a client can receive an address from.

Lease In Use function

A lease In Use function is called anytime a lease changes state from any status to ACTIVE.

Lease No Longer Used function

This function is called anytime a lease changes from the ACTIVE state to any other state. That is, if you want to do something after a lease expires or is released, this is where the user should do it. This may be used to detect when clients leave the network (although it is not possible to know precisely when a client left. The time the client left will be any time between the last lease renewal and the lease expiry.

Lease Bind Update Received function

This function is called whenever DCS receives an update on a lease from its failover peer.

Examples

The following examples are the two most common usages of Embedded Python.

Example – Restriction by MAC Address

There are two ways to limit which clients can get addresses from which Pools in DCS. The first is to create an entry in the DCS User database for each MAC address and have DCS search this list for every lease request. This is not efficient since the longer the list, the longer the search will take.

The preferred method is to create a client object manually (either via the DCS configuration file, nom_tell or the CC API). Clients that are created manually have the ‘predefined’ bit set on the client object. The filter functions can then be set to only allow predefined clients to get an address. This method is more scalable since DCS simply has to look to see if a client object exists rather than parsing through a list of MAC addresses in the User database. Looking up clients in DCS is substantially quicker than looking up objects in the user database

A client can be predefined as follows using the Command Channel:

```
# nom_tell dcs ccdb objtype=client method=create name=client1 \
  hardware_address='{type=>ethernet bytes=>00:11:22:33:44:55}'
```

A sample of a DCS Configuration file and the embedded Python configuration file is listed below:

DCS Configuration

In order to use an Embedded Python function in DCS, the Python function file must exist with the proper logic (As shown below), the DCS server must be told which functions to use, where to find the function file, and any attributes must be assigned to the appropriate pools. These can all be done using the Command Channel as shown below:

Specify the location of the Python function file and the file name:

```
# nom_tell dcs ccdb objtype=server method=setfield \
  userpath='/usr/local/nom/etc'
# nom_tell dcs ccdb objtype=server method=setfield userfile='dcsuser.py'
```

Create the Assoctiation Function and the Filter function. Note that the name must correspond to the function name in the function file:

```
# nom_tell dcs ccdb objtype=attr_assoc_func name=assoc_attrib \
  method=create
# nom_tell dcs ccdb objtype=filter_func method=create name=filter_pools
```

Create the Attribute and add it to the pool:

```
# nom_tell dcs ccdb objtype=attribute method=create \
  attribute=attrib1 name=known
# nom_tell dcs ccdb objtype=pool method=add_attribute \
  name=pool1 attribute=known
```

To restrict access to known clients on all pools, the add_attribute command can be run for each pool in the DCS server.

DCS Embedded Python Functions

```
import sys
# All the DCS-specific stuff is in this module
import pydcs

# DCS hands the AssociationFunc the client object and the packet
# that is being handled at present.
# with clients
def assoc_attrib(client, packet):
   known_attr = pydcs.Attribute.find(name = 'known')
   # If this client has the predefined boolean set, meaning it was
   # defined in the config file or over the Command Channel, give it
   # the "Known Clients" attribute
   if (client.predefined == True):
         client.add_attribute(known_attr)
   # Else, do nothing

# DCS hands the FilterFunc the current client, packet, list of
# possible pools, and the (starting empty) list of pools the client
# is permitted to.
def filter_pools(client,packet,possible,permitted):
   # iterate through all the pools DCS thinks are allowed
   is_known = client.find_attribute(name = 'known')
   for pool in possible:
         # If the client is known ...
         if is_known:
               # ... and the pool has a known-clients attribute,
               # let the client into this pool.
               if pool.find_attribute(name = 'known'):
                     permitted.append(pool)
         # if the client is unknown,
         else:
               if pool.find_attribute(name = 'known') == None:
                     permitted.append(pool)
```

Example – Pool Selection based on Relay agent

This example is used to allocate clients whose Circuit ID is known to one pool while unknown clients get an address on the other pool. It is assumed that the userfile and userpath have been set up as described in the previous example.

DCS Configuration

Create the Association Function and the Filter function. Note that the name must correspond to the function name in the function file:

```
# nom_tell dcs ccdb objtype=attr_assoc_func name=associate_attributes \
   method=create
# nom_tell dcs ccdb objtype=filter_func method=create name=filter_pools
```

Create the Attribute and add it to one of the pools:

```
# nom_tell dcs ccdb objtype=attribute method=create \
   attribute=attrib1 name=known-clients
```

```
# nom_tell dcs ccdb objtype=pool method=add_attribute \
   name=pool1 attribute=known-clients
```

Create the Circuit IDs in the DCS User Database:

```
# nom_tell dcs ccdb objtype=user_data method=create \
   name='01:02:03:04:05:06:07:08' value='customer1'
# nom_tell dcs ccdb objtype=user_data method=create \
   name='01:02:03:04:05:06:07:09' value='customer2'
```

Embedded Python Functions

```
# This Python file is only executable from DCS
# This will look up Circuit ID information from the User Database
# to determine if the incoming request is from a known client

# import some necessary modules
import pydcs # This has all the DCS specific stuff
import sys   # system stuff, just in case


def associate_attributes(client,packet):
   # Check for the relay agent option, assume if there is, that there's
   #    a circuit id

   # If there aren't any agent options, do nothing.
   if client.agent_options != None:
         agent_opts = client.agent_options['options']
         circuit_id = agent_opts['agent-circuit-id']
   # If we don't have any relay agent options, don't do anything else.
   else:
         return # Leave the function

   # Look up to see if the User Data exists
   user_data = None
   user_data = pydcs.UserData.find(name = circuit_id)



   # Assign the client an attribute since we know about it already
   attr = pydcs.Attribute.find(name = 'known-clients')

   # If we don't find the attribute, give up.  We can't do anything
   # without this attribute
   if attr == None:
         return

   # If the user data exists (client is known), and we succeeded
   # at getting the attribute out of the database, add the attribute
   # to the client object
```

```
      if (user_data != None):
                  client.add_attribute(attr)
      # else, we should remove the known clients attribute from the client
      # if it's there, since clearly this client has roved.
      else:
            if client.find_attribute('known-clients'):
                  client.del_attribute(attr)

   def filter_pools(client,packet,permitted,possible):
      # If the client is known, add the known-clients pools to the
      # permitted list
      if client.find_attribute('known-clients'):
            for pool in possible:
                  if pool.find_attribute('known-clients'):
                        permitted.append(pool)
      # If the client is unknown, skip all the known-clients pools
      else:
            for pool in possible:
                  if pool.find_attribute('known-clients') == None:
                        permitted.append(pool)
```

6

Client and Packet Objects

The client and packet objects that are accessible by Embedded Python have a great deal of information that is accessible to the Python functions. These are described in more detail below:

The Packet Object

Packet fields	Data Type	Description
chaddr	binary data of a fixed size	Client Hardware Address
ciaddr	IPv4 address	Client's IP address, set by the client when it has confirmed that the IP address is valid
dhcp_msg_type	integer	
file	string	Name of the file for the client to request from the next hop server
flags	integer	Flags field
giaddr	IPv4 address	Relay Agent IP address
hlen	integer	Length of hardware address in bytes
hops	integer	Number of relay agents that have forwarded the packet
htype	integer	Hardware type (Ethernet is 1)
max_message_size	integer	
op	integer	Message operation code (1 for messages sent by client and 2 for messages sent by the server)
options	list of options	Requested DHCP Options
options_length	integer	Length of Options field
secs	integer	
siaddr	IPv4 address	IP address of the next hop server
sname	string	Name of the next server to use in the configuration process
total_length	integer	Total length of packet
xid	integer	Transaction identifier
yiaddr	IPv4 address	Requested IP

Table 6-1 The Packet Object

Methods for accessing packet data:

To access the client hardware address:

```
mac_addr = packet.chaddr
```

To access the remote circuit id:

```
option_82_data = packet.agent_options[‘options’]
remote_id = option_82_data[‘agent-remote-id’]
```

To access the hostname:

```
dhcp_options = packet.options[‘options’]
client_name = dhcp_options[‘dhcp_client_identifier’]
```

To drop a request from a Microsoft RAS server, a Packet Receipt Function can be used. The function would be as follows:

```
def drop_ras(packet)

         dhcp_options = packet.options[‘options’]
   client_name = dhcp_options[‘dhcp_client_identifier’]

   if client_name.search(‘ RAS’)
    raise pydcs.cancelled
```

The Client Object

Client Parameter	Data Type
agent_options	list of options
attributes	list of strings
boot_server_address	domain name or IPv4 address
bootp	boolean
client_domain_name	string
client_fqdn	string
client_name	string
client_updates	boolean
default_lease_time	integer
default_options	list of options
dhcp_client_id_synthesized	boolean
dhcp_client_identifier	binary data
dns_name	string
dns_update_override	boolean
do_ddns	boolean
filename	string
fixed_address	list of strings
forward_dns	boolean
forward_rcode	integer
fqdn_is_multiple_labels	boolean
generation	integer
get_lease_hostnames	boolean
hardware_address	MAC address
inaddr_domain	string
last_ddnsupdate_time	floating-point seconds since epoch
last_stored_time	floating-point seconds since epoch
last_transaction_time	integer seconds since epoch
lease	IPv4 address
max_lease_time	integer

Table 6-2 Client Parameter Data Types

Client Parameter	**Data Type**
min_lease_time	integer
name	string
notifies	list of event channels, each has an associated integer notify count
optionsets	list of strings
parameter_request_list	list of strings
picked_net	string
picked_pool	string
picked_sharednet	string
predefined	boolean
replace_fqdn_domain	boolean
reverse_dns	boolean
reverse_rcode	integer
sent_options	list of options
server_identifier	domain name or IPv4 address

Table 6-2 Client Parameter Data Types *(continued)*

DHCP Options

Option Number	Option Name
1	subnet-mask
2	time-offset
3	routers
4	time-servers
5	name-servers
6	domain-name-servers
7	log-servers
8	cookie-servers
9	lpr-servers
10	impress-servers
11	resource-location-servers
12	host-name
13	boot-size
14	merit-dump
15	domain-name
16	swap-server
17	root-path
18	extensions-path
19	ip-forwarding
20	non-local-source-routing
21	policy-filter
22	max-datagram-reassembly-size
23	default-ip-ttl
24	path-mtu-aging-timeout
25	path-mtu-plateau-table
26	interface-mtu
27	all-subnets-local
28	broadcast-address
29	perform-mask-discovery

Table 6-3 DHCP Options

Option Number	Option Name
30	mask-supplier
31	router-discovery
32	router-solicitation-address
33	static-routes
34	trailer-encapsulation
35	arp-cache-timeout
36	ieee802.3-encapsulation
37	default-tcp-ttl
38	tcp-keepalive-interval
39	tcp-keepalive-garbage
40	nis-domain
41	nis-servers
42	ntp-servers
43	vendor-encapsulated-options
44	netbios-name-servers
45	netbios-dd-server
46	netbios-node-type
47	netbios-scope
48	font-servers
49	x-display-manager
50	requested-address
51	lease-time
52	option-overload
53	message-type
54	server-identifier
55	parameter-request-list
56	message
57	max-message-size
58	renewal-time
59	rebinding-time
60	vendor-class-identifier

Table 6-3 DHCP Options *(continued)*

Option Number	Option Name
61	dhcp-client-identifier
62	nwip-domain-name
63	nwip-suboptions
64	nis+-domain
65	nis+-servers
66	tftp-server-name
67	bootfile-name
68	mobile-ip-home-agents
69	smtp-servers
70	pop3-servers
71	nntp-servers
72	www-servers
73	finger-servers
74	irc-servers
75	streettalk-servers
76	streettalk-directory-assistance-servers
77	user-class
78	slp-directory-agent
79	slp-service-scope
81	fqdn
82	relay-agent-information
85	nds-servers
86	nds-tree-name
87	nds-context
98	uap-servers
116	autoconfiguration
117	name-service-search-order
118	subnet-selection
119	domain-search-list
121	classless-static-routes

Table 6-3 DHCP Options *(continued)*

A

DCS Configuration Script

This appendix contains a script that can be used to populate DCS with the configuration data described above. This is a simple shell script which calls the nom_tell utility.

```
/usr/local/nom/sbin/nom_tell dcs ccdb objtype=network method=create
   name=net1 netaddr=10.1.1.0/24
/usr/local/nom/sbin/nom_tell dcs ccdb objtype=pool method=create name=
   pool1
/usr/local/nom/sbin/nom_tell dcs ccdb objtype=pool method=add_range
   name=pool1 range=('10.1.1.10', '10.1.1.100')
/usr/local/nom/sbin/nom_tell dcs ccdb objtype=optionset method=create
   name=basicopts
/usr/local/nom/sbin/nom_tell dcs ccdb objtype=optionset method=setfield
   name=basicopts options=('dhcp'  ('subnet-mask' [255.255.255.0])
   ('domain-name' 'company.com') ('domain-name-servers' ('10.1.2.251'
   '10.12.12.1')))'
/usr/local/nom/sbin/nom_tell dcs ccdb objtype=optionset method=create
   name=net1opts
/usr/local/nom/sbin/nom_tell dcs ccdb objtype=optionset method=setfield
   name=net1opts options='('dhcp' ('routers' ('10.1.1.1')))'
/usr/local/nom/sbin/nom_tell dcs ccdb objtype=server method=
   add_optionset optionset=basicopts
/usr/local/nom/sbin/nom_tell dcs ccdb objtype=network method=
   add_optionset name=net1 optionset=net1opts


/usr/local/nom/sbin/nom_tell dcs ccdb objtype=optionset method=create
   name=net2opts
```

```
/usr/local/nom/sbin/nom_tell dcs ccdb objtype=optionset method=setfield
   name=net2opts options='('dhcp' ('routers' ('10.1.2.1')))'

/usr/local/nom/sbin/nom_tell dcs ccdb objtype=optionset method=create
   name=net3opts
/usr/local/nom/sbin/nom_tell dcs ccdb objtype=optionset method=setfield
   name=net3opts options='('dhcp' ('routers' ('10.1.3.1')))'

/usr/local/nom/sbin/nom_tell dcs ccdb objtype=optionset method=create
   name=net4opts
/usr/local/nom/sbin/nom_tell dcs ccdb objtype=optionset method=setfield
   name=net4opts options='('dhcp' ('routers' ('10.1.4.1')))'



/usr/local/nom/sbin/nom_tell dcs ccdb objtype=network method=
   add_optionset name=net2 optionset=net2opts

/usr/local/nom/sbin/nom_tell dcs ccdb objtype=network method=create
   name=net3 netaddr=10.1.3.0/24
/usr/local/nom/sbin/nom_tell dcs ccdb objtype=network method=
   add_optionset name=net3 optionset=net3opts

/usr/local/nom/sbin/nom_tell dcs ccdb objtype=network method=create
   name=net4 netaddr=10.1.4.0/24
/usr/local/nom/sbin/nom_tell dcs ccdb objtype=network method=
   add_optionset name=net4 optionset=net4opts

# Additional Pools
/usr/local/nom/sbin/nom_tell dcs ccdb objtype=pool method=create name=
   pool2
/usr/local/nom/sbin/nom_tell dcs ccdb objtype=pool method=add_range
   name=pool2 range=('10.1.2.10', '10.1.2.100')
/usr/local/nom/sbin/nom_tell dcs ccdb objtype=pool method=create name=
   pool3
/usr/local/nom/sbin/nom_tell dcs ccdb objtype=pool method=add_range
   name=pool3 range=('10.1.3.10', '10.1.3.100')
/usr/local/nom/sbin/nom_tell dcs ccdb objtype=pool method=add_range
   name=pool3 range=('10.1.4.10', '10.1.4.100')

#Shared Network
/usr/local/nom/sbin/nom_tell dcs ccdb objtype=sharednet method=create
   name=shared1
/usr/local/nom/sbin/nom_tell dcs ccdb objtype=sharednet method=
   add_network name=shared1 network=net3
/usr/local/nom/sbin/nom_tell dcs ccdb objtype=sharednet method=
   add_network name=shared1 network=net4
```

B

DCS Test Plan

This Appendix provides a concise plan for testing and evaluating DCS.

Overview

This document is intended to serve as a guide to assist in the process of evaluating Nominum's Dynamic Configuration Server (DCS). This document will cover various test cases to run with DCS to certify it for use in a service provider network. This document can be used in conjunction with the DCS Evaluation Guide and the DCS documentation to provide detailed instructions on the configuration of the scenarios below.

NOTE We have provided blank space in each table in which you can note the results of your tests.

Target Audience

This guide is aimed at potential Nominum customers who are interested in deploying DCS. A good working knowledge of DHCP and detailed knowledge of either the existing or anticipated DHCP deployment.

Lab Setup

In order to test all of the major features of DCS, it is important to have two machines to run DCS in failover mode. In addition, it is useful to have at least one router to forward DHCP traffic to the DCS servers. The diagram below shows a sample configuration that will allow for testing of all of the scenarios described below. In addition, if load testing is required, a traffic generation tool may also be connected to the network.

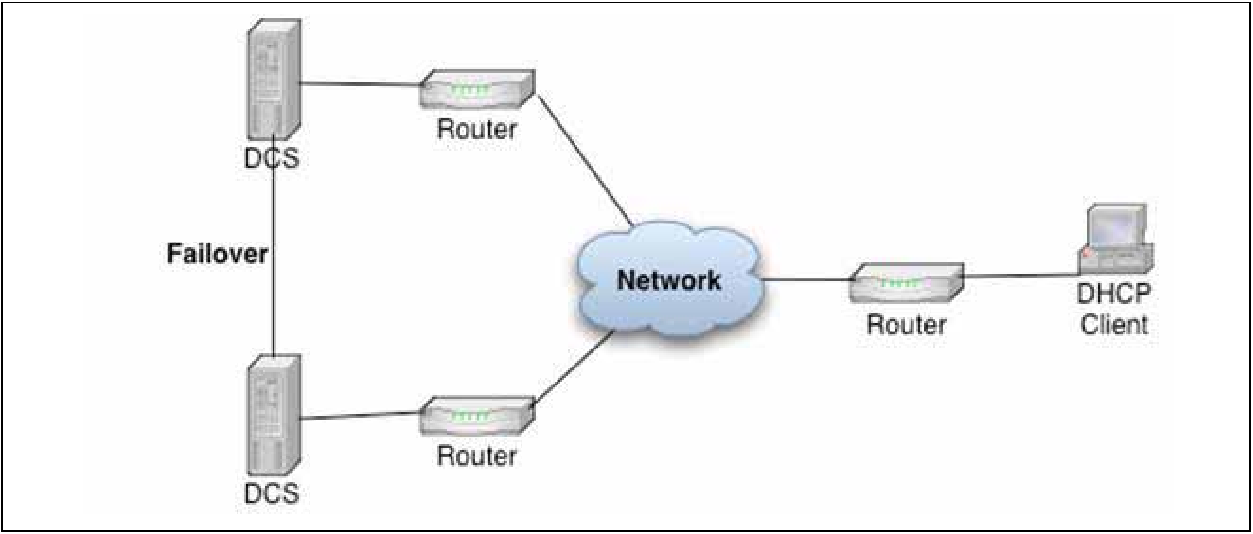

Figure B-1 Lab Setup

Test Cases

The following test cases are designed to prove the suitability of Nominum DCS for use in a broadband network. For full detail on setting up the DCS configuration necessary for these tests, please refer to the DCS Tutorial or the DCS documentation. All tests assume a network topology as described in the above section.

Basic Operations

Objective

Determine that DCS is able to bring up a single client.

Configuration

- This test can be done using a PC client, DSL modem or DHCP traffic generator.

- There must be at least two DHCP pools, one that is set aside for Known clients and one that is set aside for Unknown clients.
- There must be a BRAS between the client and DCS to append the Circuit ID to the DHCP request.
- Shorter leases are desirable in testing to reduce wait time for t clients to renew or time out. However, very short leases (up to 5 minutes) are not recommended. Using 10 minute leases should provide a reasonable balance between the need not to spend too much time waiting, and the need to use workable lease lengths.

Test #1 – Client's Circuit ID is known

- Add the client's Circuit ID to the DCS database
- Verify client gets an IP address in the 'known' pool
- Release and renew the client
- Remove the client from the network and verify that the lease is made available for other clients by DCS by looking at the pool statistics
- Monitor the IP allocation process using syslog on the DCS server or via a traffic analyzer

P/F	Comments

Table B-1 Single Client Test #1 - Clients Curcuit ID is known

Test #2 – Client's Circuit ID is not known

- Ensure the client's Circuit ID is not in the DCS database
- Verify client gets an IP address in the 'unknown' pool
- Release and renew the client
- Remove the client from the network and verify that the lease is made available for other clients by DCS

- Monitor the IP allocation process using syslog on the DCS server or via a traffic analyzer

P/F	Comments

Table B-2 Single Client Test #1 - Clients Curcuit ID is not known

Interoperability with Network Devices

Objective

Determine that DCS is able to configure all types of devices expected on the network.

Configuration

- This test can be done using at least one of each type of device to be tested.
- There must be at least two DHCP pools, one that is set aside for Known clients and one that is set aside for Unknown clients.
- There must be a BRAS between the client and DCS to append the Circuit ID to the DHCP request.
- Shorter leases are desirable in testing to reduce wait time for t clients to renew or time out. However, very short leases (up to 5 minutes) are not recommended. Using 10 minute leases should provide a reasonable balance between the need not to spend too much time waiting, and the need to use workable lease lengths..

CPE	P/F	Comments

Table B-3 Interoperability with Network Devices Test

Load and Capacity

Objective

Determine that DCS is able to deal with large numbers of DHCP clients. This test is designed to test both normal, high load, scenarios as well as to simulate large numbers of clients booting up after a power failure.

Configuration

- The address ranges in the pools must be large enough to support the large numbers of clients that will be simulated. Valid test results should be possible with 50,000 to 100,000 clients. It is suggested that the pools should contain ranges of approximately 16,000 addresses each
- When discriminating between known and unknown clients, the Circuit IDs must be pre-generated and provisioned into DCS to ensure that clients that are 'known' get the proper addresses.

Test	**P/F**	**# Subscribers**	**Leases/Sec**	**Comments**
DHCP w/o Circuit ID checking				
DHCP with Circuit ID checking				

Table B-4 Load and Capacity Test

Denial of Service

Objective

This test will verify DCS' abilities to operate while under a Denial of Service attack.

Configuration

- This test can be done using a DHCP traffic generator to simulate the DHCP traffic.
- Two scenarios should be attempted; A large number of requests with a small number of MAC addresses, and a large number of requests with a large number of MAC addresses.

- The address ranges in the pools must be large enough to support the large numbers of clients that will be simulated. Valid test results should be possible with 50,000 to 100,000 clients. It is suggested that the pools should contain ranges of approximately 16,000 addresses each.

Small population of Mac Addresses

P/F	Requests/ Second	% CPU	Leases/ Second	Comments
DHCP w/o Circuit ID checking				
DHCP with Circuit ID checking				

Table B-5 Denial of Service Test: Small population of Mac Addresses

Large population of Mac Addresses

P/F	Requests/ Second	% CPU	Leases/ Second	Comments
DHCP w/o Circuit ID checking				
DHCP with Circuit ID checking				

Table B-6 Denial of Service Test: Large population of Mac Addresses

Long Running Test

Objective

This test is designed to verify that DCS can continue to operate over longer periods of time as normal to high loads.

Configuration

Test #1 Long Running test

- The address ranges in the pools must be large enough to support the large numbers of clients that will be simulated. Valid test results should be possible with 50,000 to 100,000 clients. It is suggested that the pools should contain ranges of approximately 16,000 addresses each
- When discriminating between known and unknown clients, the Circuit IDs must be pre-generated to ensure that clients that are 'known' get the proper addresses.

Time	CPU Utilization	Memory Size

Table B-7 Long Running Test

Test #2 On the fly configuration

- Add new networks ranges and pools to DCS while the above test is running. This will verify that DCS can be reconfigured while DCS is responding to DHCP requests

P/F	Comments

Table B-8 On the fly configuration test

DCHP Failover

Objective

This test is designed to validate DCS' implementation of DHCP failover. This will demonstrate that DCS can continue to provide service in the event of the failure of a server.

Configuration

- This test can be performed using a PC client or DSL modem
- The two DCS servers must me configured to be failover peers for the subnets that the test clients will originate from.

Suggested test procedure

- Boot client with both DCS servers in operation
- Unplug network cable from DCS server that allocated the address
- Verify the Failover status of the active server
- Release and renew the client
- Verify the active server allocates the an IP address
- Reconnect the other DCS server, verify that the two DCS servers resynchronize

Scenario	P/F	Comments
Physically disconnect one DCS server from the network		
Reattach DCS server to network		
Shutdown one DCS server		
Restart DCS Server		

Table B-9 DHCP Failover Test

Index