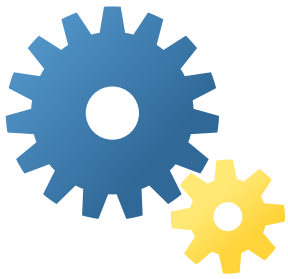


DHCPKit

DHCPKit Documentation

Release 1.0.8b1

S.J.M. Steffann

Jul 23, 2018

Contents

This package contains a flexible DHCPv6 server written in Python 3.4+. Its purpose is to provide a framework for DHCP services. It was written for ISPs to use in provisioning their customers according to their own business rules. It can be integrated into existing ISP management and provisioning tools. The flexibility and ability to integrate business rules will appeal to enterprises as well. Writing extensions to DHCPKit is very easy!

The official documentation[1] is hosted by Read the Docs[2].

[1] http://dhcpkit.readthedocs.io
[2] https://readthedocs.org

CHAPTER 1

Documentation

1.1 Applications

1.1.1 ipv6-dhcpd(8)

Synopsis

ipv6-dhcpd [-h] [-v] [-p PIDFILE] config

Description

This is the executable that runs the DHCP server code. Its functionality depends on the handler modules configured in the configuration file. These can implement anything from printing incoming packets to providing a fully functional stateful DHCP server.

Command line options

config
: is the *configuration file* (page 5).

-h, --help
: show the help message and exit.

-v, --verbosity
: increase output verbosity. This option can be provided up to five times to increase the verbosity level. If the `colorlog` package is installed logging will be in colour.

-p PIDFILE, **--pidfile** PIDFILE
: save the server's PID to this file

Security

Because it has to be able to bind to the DHCPv6 server UDP port (547) it has to be started as *root*. The process will give up *root* privileges after it reads the configuration file and opens the listening sockets.

1.1.2 ipv6-dhcpctl(8)

Synopsis

ipv6-dhcpctl [-h] [-v] [-c FILENAME] command

Description

A remote control utility that allows you to send commands to the DHCPv6 server.

Command line options

command
: is the command to send to the server. Use the *help* command to see what commands are available from your server.

-h, --help
: show the help message and exit.

-v, --verbosity
: increase output verbosity. This option can be provided up to five times to increase the verbosity level. If the `colorlog` package is installed logging will be in colour.

-c `FILENAME`**, --control-socket** `FILENAME`
: location of domain socket for server control. The default socket is /var/run/ipv6-dhcpd.sock which is also the default location where the server will create its control socket.

1.1.3 ipv6-dhcp-build-sqlite(1)

Synopsis

ipv6-dhcp-build-sqlite [-h] [-f] [-v] source destination

Description

This utility converts a *CSV file with assignments* (page 23) to a SQLite database for use with the *Static-sqlite* (page 24) handler.

Command line options

source
: is the source CSV file

destination
: is the destination SQLite file

-h, --help
: show the help message and exit.

-f, --force
: force removing old entries, even if that means deleting more than 30% of the contents of the database

-v, --verbosity
: increase output verbosity. This option can be provided up to five times to increase the verbosity level. If the `colorlog` package is installed logging will be in colour.

Concurrency

This utility implements some functionality to make it possible to run it against a SQLite database that is being concurrently used by a DHCPv6 server. It will release the write lock on the database every so often to allow the server to continue its processing of requests.

While updating the database this tool will check to see if another instance is writing newer entries to the same database. If this is detected it will abort to let the other instance finish its work.

Safety

To prevent the database being destroyed because of an invalid input file this tool compares the size of the number of entries read from the CSV file with the size of the database. If more than 30% of the database would be deleted because the corresponding entries dave disappeared from the CSV file the delete action is aborted and old entries are left in the database. Provide the `--force` (page 4) option to force removal of those entries.

1.2 IPv6 Server configuration

This describes the configuration file for DHCPKit. The syntax of this file is loosely based on the Apache configuration style. It is implemented using ZConfig[3].

The configuration file consists of *basic server settings* (page 6), *Listeners* (page 26) that receive messages from the network and some *Handlers* (page 13) that process the request and generate the response (possibly surrounded by *Filters* (page 10) that determine which handlers get applies to which request).

1.2.1 Configuration file format

This describes the configuration file for DHCPKit. The syntax of this file is loosely based on the Apache configuration style. It is implemented using ZConfig[4].

The configuration file consists of *basic server settings* (page 6), *Listeners* (page 26) that receive messages from the network and some *Handlers* (page 13) that process the request and generate the response (possibly surrounded by *Filters* (page 10) that determine which handlers get applies to which request).

Example

```
# Logging to console and syslog
<logging>
    <console>
        level debug-packets
    </console>
    <syslog>
        level info
    </syslog>
</logging>

# Run as user 'demo' with group 'nogroup'
user demo
group nogroup

# Listen to this unicast address (to receive messages from a relay)
<listen-unicast 2001:db8::1>
    interface en0
</listen-unicast>
```

(continues on next page)

[3] https://pypi.python.org/pypi/ZConfig
[4] https://pypi.python.org/pypi/ZConfig

(continued from previous page)

```
# Handlers that are only applied to this /48
<subnet 2001:db8:1::/48>
    # Ignore requests from this /64
    <subnet 2001:db8:1:2::/64>
        <ignore-request/>
    </subnet-group>

    # Everybody else: assign static address/prefix from this CSV
    <static-csv static.csv />
</subnet>
```

Configuration options

user The user name the server should run as.

Default: “nobody”

group The group name the server should run as.

Default: The primary group of the user.

pid-file Save the PID of the main process to this file.

Example: “/var/run/ipv6-dhcpd.pid”

Default: “/var/run/ipv6-dhcpd.pid”

control-socket Create a domain socket in this location to control the server.

Example: “/var/run/ipv6-dhcpd.sock”

Default: “/var/run/ipv6-dhcpd.sock”

control-socket-user User that owns the control-socket.

control-socket-group Group that owns the control-socket.

workers The number of worker processes that will be started.

Default: The number of CPUs detected in your system.

allow-rapid-commit Whether to allow DHCPv6 rapid commit if the client requests it.

Default: “no”

rapid-commit-rejections Whether to allow DHCPv6 rapid commit for responses that reject a request.

Default: “no”

server-id (section of type *Duid* (page 9)) The DUID to use as the server-identifier.

Example:

```
<duid-ll server-id>
    hardware-type 1
    link-layer-address 00:24:36:ef:1d:89
</duid-ll>
```

exception-window The length of the exceptions window.

Default: “10.0”

max-exceptions The number of exceptions that can occur in the exception window before the server stops itself. This prevents the server from spinning in circles when something unexpected goes wrong.

Default: “5”

Possible sub-section types

Logging **(page 7)** This section contains the logging configuration. It contains a list of log-handlers that specify where to send the log entries.

Statistics **(page 8)** By default the DHCPv6 server only keeps global statistics. Provide categories to collect statistics more granularly.

Listeners **(page 26) (multiple allowed)** Configuration sections that define listeners. These are usually the network interfaces that a DHCPv6 server listens on, like the well-known multicast address on an interface, or a unicast address where a DHCPv6 relay can send its requests to.

Filters **(page 10) (multiple allowed)** Configuration sections that specify filters. A filter limits which handlers get applied to which messages. Everything inside a filter gets ignored if the filter condition doesn't match. That way you can configure the server to only apply certain handlers to certain messages, for example to return different information options to different clients.

Handlers **(page 13) (multiple allowed)** Configuration sections that specify a handler. Handlers process requests, build the response etc. Some of them add information options to the response, others look up the client in a CSV file and assign addresses and prefixes, and others can abort the processing and tell the server not to answer at all.

You can make the server do whatever you want by configuring the appropriate handlers.

1.2.2 Overview of sections

Logging

This section contains the logging configuration. It contains a list of log-handlers that specify where to send the log entries.

Example

```
<logging>
    <console>
        level debug-handling
        color yes
    </console>

    <syslog />

    log-multiprocessing no
</logging>
```

Section parameters

log-multiprocessing Enable this if you want logging of process handling. Mostly useful for debugging server code.

Default: "no"

Possible sub-section types

Loghandler **(page 28) (multiple allowed)** Log-handlers output log entries to somewhere. If you want to send your logs somewhere configure one of these. There are log-handlers to show log entries on the console. Send them to a syslog process, server, etc.

Map-rule

A mapping rule for MAP implementations.

Example

```
<map-rule>
    ipv6-prefix 2001:db8:f000::/36
    ipv4-prefix 192.0.2.0/24
    contiguous-ports 64
    sharing-ratio 16
    forwarding-mapping yes
</map-rule>
```

Section parameters

ipv6-prefix (required) The IPv6 prefix containing MAP clients.

ipv4-prefix (required) The IPv4 prefix that the MAP clients will share.

contiguous-ports (required) The number of contiguous ports. This value must be a power of 2. It determines the number of bits after the PSID.

sharing-ratio (required) The number of customers sharing one IPv4 address. This value must be a power of 2. It determines the length of the PSID.

forwarding-mapping Whether this rule is a Forwarding Mapping Rule (FMR) or a Basic Mapping Rule (BMR).

Default: "no"

Statistics

By default the DHCPv6 server only keeps global statistics. Provide categories to collect statistics more granularly.

Example

```
<statistics>
    interface eth0
    subnet 2001:db8:0:1::/64
    subnet 2001:db8:0:2::/64
    relay 2001:db8:1:2::3
</statistics>
```

Section parameters

interface (multiple allowed) Collect statistics per server interface

Example: "interface eth0"

subnet (multiple allowed) Collect statistics per client subnet

Example: "subnet 2001:db8::/64"

relay (multiple allowed) Collect statistics per relay

Example: "relay 2001:db8::1:2"

1.2.3 Overview of section types

Duid

Configuration sections that specify a DUID.

Duid-en

A DUID based on an enterprise-number and an opaque identifier.

Example

```
<duid-en>
    enterprise-number 40208
    identifier 12:34:56:78:90:ab:cd:ef
</duid-en>
```

Section parameters

enterprise-number (required) This must be a Private Enterprise Number as maintained by IANA. See http://www.iana.org/assignments/enterprise-numbers.

identifier (required) This is a unique identifier assigned by the specified enterprise. The value must be provided as a hexadecimal string. Each octet may be separated with colons, but this is not required.

Example: "12:34:56:78:90:ab:cd:ef:ca:fe:be:ef"

Duid-ll

A DUID based on a link-layer address.

Example

```
<duid-ll>
    hardware-type 1
    link-layer-address 002436ef1d89
</duid-ll>

<duid-ll server-id>
    hardware-type 1
    link-layer-address 00:24:36:ef:1d:89
</duid-ll>
```

Section parameters

hardware-type (required) The hardware type must be a valid hardware type assigned by the IANA, as described in **RFC 826**[5]. Ethernet has type number 1.

link-layer-address (required) The link-layer address must be provided as a hexadecimal string. Each octet may be separated with colons, but this is not required.

Example: "00:24:36:ef:1d:89"

[5] https://tools.ietf.org/html/rfc826.html

Duid-llt

A DUID based on a link-layer address and a timestamp.

Example

```
<duid-llt>
    hardware-type 1
    link-layer-address 002436ef1d89
    timestamp 2016-12-31T23:59:59Z
</duid-llt>

<duid-llt server-id>
    hardware-type 1
    link-layer-address 00:24:36:ef:1d:89
    timestamp 2016-12-31T23:59:59Z
</duid-llt>
```

Section parameters

hardware-type (required) The hardware type must be a valid hardware type assigned by the IANA, as described in **RFC 826**[6]. Ethernet has type number 1.

link-layer-address (required) The link-layer address must be provided as a hexadecimal string. Each octet may be separated with colons, but this is not required.

Example: "00:24:36:ef:1d:89"

timestamp (required) The timestamp to include in the address. It must be provided in the ISO-8601 compatible format "%Y-%m-%dT%H:%M:%SZ".

Example: "2016-12-31T23:59:59Z"

Filters

Configuration sections that specify filters. A filter limits which handlers get applied to which messages. Everything inside a filter gets ignored if the filter condition doesn't match. That way you can configure the server to only apply certain handlers to certain messages, for example to return different information options to different clients.

Elapsed-time

Filter incoming messages based on the value of the *ElapsedTimeOption* (page 160) in the request. At least one time limit must be provided.

This filter can be used as a very simple mechanism for DHCPv6 server fail-over. You can configure one server without an elapsed-time filter and another server with a filter that ignores solicit messages when the elapsed time is less than a certain value. The first server will try to answer all request, but if it doesn't answer all requests for some reason then the client's elapsed time will increase until it passes the threshold of the second server, which will then stop ignoring it and respond.

[6] https://tools.ietf.org/html/rfc826.html

Example

```
<elapsed-time>
    less-than 30s

    <ignore-request>
        message-type solicit
    </ignore-request>
</elapsed-time>
```

Section parameters

more-than Only process messages where the elapsed time is more than the provided number of seconds. For ease of use these suffixes may be used: 's' (seconds), 'm' (minutes), 'h' (hours), or 'd' (days).

Example: "30s"

less-than Only process messages where the elapsed time is less than the provided number of seconds. For ease of use these suffixes may be used: 's' (seconds), 'm' (minutes), 'h' (hours), or 'd' (days).

Example: "1h"

Possible sub-section types

Filters **(page 10) (multiple allowed)** Configuration sections that specify filters. A filter limits which handlers get applied to which messages. Everything inside a filter gets ignored if the filter condition doesn't match. That way you can configure the server to only apply certain handlers to certain messages, for example to return different information options to different clients.

Handlers **(page 13) (multiple allowed)** Configuration sections that specify a handler. Handlers process requests, build the response etc. Some of them add information options to the response, others look up the client in a CSV file and assign addresses and prefixes, and others can abort the processing and tell the server not to answer at all.

You can make the server do whatever you want by configuring the appropriate handlers.

Marked-with

Filter incoming messages based on the mark set by i.e. the listener.

Example

```
<marked-with bla>
    <ignore-request/>
</marked-with>
```

Possible sub-section types

Filters **(page 10) (multiple allowed)** Configuration sections that specify filters. A filter limits which handlers get applied to which messages. Everything inside a filter gets ignored if the filter condition doesn't match. That way you can configure the server to only apply certain handlers to certain messages, for example to return different information options to different clients.

Handlers **(page 13) (multiple allowed)** Configuration sections that specify a handler. Handlers process requests, build the response etc. Some of them add information options to the response, others look up the client in a CSV file and assign addresses and prefixes, and others can abort the processing and tell the server not to answer at all.

You can make the server do whatever you want by configuring the appropriate handlers.

Subnet

Filter incoming messages based on the subnet that the link-address is in.

Example

```
<subnet 2001:db8:dead::/48>
    <ignore-request/>
</subnet-group>
```

Possible sub-section types

Filters **(page 10) (multiple allowed)** Configuration sections that specify filters. A filter limits which handlers get applied to which messages. Everything inside a filter gets ignored if the filter condition doesn't match. That way you can configure the server to only apply certain handlers to certain messages, for example to return different information options to different clients.

Handlers **(page 13) (multiple allowed)** Configuration sections that specify a handler. Handlers process requests, build the response etc. Some of them add information options to the response, others look up the client in a CSV file and assign addresses and prefixes, and others can abort the processing and tell the server not to answer at all.

You can make the server do whatever you want by configuring the appropriate handlers.

Subnet-group

Filter incoming messages based on the subnet that the link-address is in.

Example

```
<subnet-group>
    prefix 2001:db8:dead::/48
    prefix 2001:db8:beef::/48

    <ignore-request/>
</subnet-group>
```

Section parameters

prefix (required, multiple allowed) A prefix that the link-address of the relay or server interface can be in.

Example: "2001:db8:1:2::/64"

Possible sub-section types

Filters **(page 10) (multiple allowed)** Configuration sections that specify filters. A filter limits which handlers get applied to which messages. Everything inside a filter gets ignored if the filter condition doesn't match. That way you can configure the server to only apply certain handlers to certain messages, for example to return different information options to different clients.

Handlers **(page 13) (multiple allowed)** Configuration sections that specify a handler. Handlers process requests, build the response etc. Some of them add information options to the response, others look up the client in a CSV file and assign addresses and prefixes, and others can abort the processing and tell the server not to answer at all.

You can make the server do whatever you want by configuring the appropriate handlers.

Handlers

Configuration sections that specify a handler. Handlers process requests, build the response etc. Some of them add information options to the response, others look up the client in a CSV file and assign addresses and prefixes, and others can abort the processing and tell the server not to answer at all.

You can make the server do whatever you want by configuring the appropriate handlers.

Aftr-name

This sections add an AFTR tunnel endpoint name to the response sent to the client.

Example

```
<aftr-name>
    fqdn aftr.example.org
</aftr-name>
```

Section parameters

always-send Always send this option, even if the client didn't ask for it.

Default: "no"

fqdn (required) The FQDN of the AFTR tunnel endpoint.

Example: "aftr.example.com"

Copy-linklayer-id

A DHCPv6 server is not required to copy the client link-layer address option from a request to the response and echo it back to the relay. If you want to echo it back then include this handler to do so.

Example

```
<copy-linklayer-id/>
```

Copy-remote-id

A DHCPv6 server is not required to copy the remote-id option from a request to the response and echo it back to the relay. If you want to echo it back then include this handler to do so.

Example

```
<copy-remote-id/>
```

Copy-subscriber-id

A DHCPv6 server is not required to copy the subscriber-id option from a request to the response and echo it back to the relay. If you want to echo it back then include this handler to do so.

Example

```
<copy-subscriber-id/>
```

Domain-search-list

This sections adds domain names to the domain search list sent to the client. If there are multiple sections of this type then they will be combined into one set of domain names which is sent to the client.

Example

```
<domain-search-list>
    domain-name example.com
    domain-name example.net
    domain-name example.org
</domain-search-list>
```

Section parameters

always-send Always send this option, even if the client didn't ask for it.

Default: "no"

domain-name (required, multiple allowed) The domain name to add to the search list.

Example: "example.com"

Iana-timing-limits

Automatically set the T1 and T2 timers on IANA Options based on given limits.

Example

```
<iana-timing-limits>
    min-t1 30m
    max-t1 12h
    factor-t1 0.5

    min-t2 30m
    max-t2 1d
    factor-t2 0.8
</iana-timing-limits>
```

Section parameters

min-t1 Minimum value for T1. T1 is the time at which the client contacts the server from which the addresses were obtained to extend their lifetimes, specified in seconds after the current time.

Default: "0"

max-t1 Maximum value for T1. T1 is the time at which the client contacts the server from which the addresses were obtained to extend their lifetimes, specified in seconds after the current time.

Default: "INFINITY"

factor-t1 The default factor for calculating T1 if it hasn't been set already. This is specified as a fraction of the shortest lifetime of the addresses in the IANAOption.

Default: "0.5"

min-t2 Minimum value for T2. T2 is the time at which the client contacts any available server to extend the lifetimes of its addresses, specified in seconds after the current time.

Default: "0"

max-t2 Maximum value for T2. T2 is the time at which the client contacts any available server to extend the lifetimes of its addresses, specified in seconds after the current time.

Default: "INFINITY"

factor-t2 The default factor for calculating T2 if it hasn't been set already. This is specified as a fraction of the shortest lifetime of the addresses in the IANAOption.

Default: "0.8"

Iapd-timing-limits

Automatically set the T1 and T2 timers on IAPD Options based on given limits.

Example

```
<iapd-timing-limits>
    min-t1 30m
    max-t1 12h
    factor-t1 0.5

    min-t2 30m
    max-t2 1d
    factor-t2 0.8
</iapd-timing-limits>
```

Section parameters

min-t1 Minimum value for T1. T1 is the time at which the client contacts the server from which the prefixes were obtained to extend their lifetimes, specified in seconds after the current time.

Default: "0"

max-t1 Maximum value for T1. T1 is the time at which the client contacts the server from which the prefixes were obtained to extend their lifetimes, specified in seconds after the current time.

Default: "INFINITY"

factor-t1 The default factor for calculating T1 if it hasn't been set already. This is specified as a fraction of the shortest lifetime of the prefixes in the IAPDOption.

Default: "0.5"

min-t2 Minimum value for T2. T2 is the time at which the client contacts any available server to extend the lifetimes of its prefixes, specified in seconds after the current time.

Default: "0"

max-t2 Maximum value for T2. T2 is the time at which the client contacts any available server to extend the lifetimes of its prefixes, specified in seconds after the current time.

Default: "INFINITY"

factor-t2 The default factor for calculating T2 if it hasn't been set already. This is specified as a fraction of the shortest lifetime of the prefixes in the IAPDOption.

Default: "0.8"

Ignore-request

When this handler is activated it tells the server to immediately stop all processing and ignore the request. The server will not send any response to the client.

Example

```
<ignore-request>
    message-type solicit
</ignore-request>
```

Section parameters

message-type (multiple allowed) The name of a message type to ignore. Can be for example `solicit` or `information-request`.

Default: Ignore all messages

Inf-max-rt

This sections sets the INF_MAX_RT value that will be sent to the client. Specify the number of seconds to send as the section name. The value must be between 60 and 86400 seconds.

Example

```
<inf-max-rt>
    limit 43200
    always-send yes
</inf-max-rt>
```

Section parameters

always-send Always send this option, even if the client didn't ask for it.

Default: "no"

limit (required) Specify the number of seconds to send as INF_SOL_RT. The value must be between 60 and 86400 seconds.

Example: "21600"

Leasequery

Implement the Leasequery protocol (**RFC 5007**[7]) and Bulk Leasequery protocol (**RFC 5460**[8]).

Example

```
<leasequery>
    allow-from 2001:db8::ffff:1
    allow-from 2001:db8:1:2::/64

    sensitive-option sip-servers-domain-name-list
    sensitive-option sip-servers-address-list

    <lq-sqlite /var/lib/dhcpkit/leasequery.sqlite />
</leasequery>
```

Section parameters

allow-from (multiple allowed) Leasequeries are not used for normal operations. They can disclose information about clients on your network. Therefore you can specify from which clients to accept leasequeries.

Not specifying any trusted clients will allow leasequeries from everywhere. This is strongly not recommended.

Also note that this only limits which clients may use the leasequery protocol. Clients that are performing bulk leasequeries also need to set up a TCP connection to this server. This has to be explicitly allowed in the *Listen-tcp* (page 27) listener.

Example:

```
allow-from 2001:db8::ffff:1
allow-from 2001:db8:beef::/48
```

sensitive-option (multiple allowed) DHCPv6 servers SHOULD be configurable with a list of "sensitive options" that must not be returned to the requestor when specified in the OPTION_ORO of the OPTION_LQ_QUERY option in the LEASEQUERY message. Any option on this list MUST NOT be returned to a requestor, even if requested by that requestor.

[7] https://tools.ietf.org/html/rfc5007.html
[8] https://tools.ietf.org/html/rfc5460.html

Example:

```
sensitive-option recursive-name-servers
sensitive-option 23
```

Possible sub-section types

Leasequery_store **(page 25) (required)** Configuration sections that define Leasequery stores. Each leasequery section must configure exactly one store. Stores perform the storing of lease data at the end of a DHCPv6 request. They also handle the queries from Leasequery clients to search in that stored data.

Map-e

Configure MAP-E mappings to send to a client.

Example

```
<map-e>
    <map-rule>
        ipv6-prefix 2001:db8:f000::/36
        ipv4-prefix 192.0.2.0/24
        contiguous-ports 64
        sharing-ratio 16
        forwarding-mapping yes
    </map-rule>

    <map-rule>
        ipv6-prefix 2001:db8:9500::/40
        ipv4-prefix 198.51.100.0/24
        contiguous-ports 4
        sharing-ratio 256
    </map-rule>

    br-address 2001:db8::1
    br-address 2001:db8::2
</map-e>
```

Section parameters

always-send Always send this option, even if the client didn't ask for it.

Default: "no"

br-address (required, multiple allowed) The IPv6 address of the Border Relay (a.k.a. AFTR) to use for reaching IPv4 sites outside the configured mappings.

Possible sub-section types

Map-rule **(page 8) (required, multiple allowed)** A mapping rule for MAP implementations.

Map-t

Configure MAP-T mappings to send to a client.

Example

```
<map-t>
    <map-rule>
        ipv6-prefix 2001:db8:f000::/36
        ipv4-prefix 192.0.2.0/24
        contiguous-ports 64
        sharing-ratio 16
        forwarding-mapping yes
    </map-rule>

    <map-rule>
        ipv6-prefix 2001:db8:9500::/40
        ipv4-prefix 198.51.100.0/24
        contiguous-ports 4
        sharing-ratio 256
    </map-rule>

    default-mapping 2001:db8:0:1::/64
</map-t>
```

Section parameters

always-send Always send this option, even if the client didn't ask for it.

Default: "no"

default-mapping (required) The /64 prefix to use for reaching IPv4 sites outside the configured mappings.

Possible sub-section types

Map-rule **(page 8) (required, multiple allowed)** A mapping rule for MAP implementations.

Ntp-servers

This sections adds NTP servers to the response sent to the client. If there are multiple sections of this type then they will be combined into one set of NTP servers which is sent to the client.

Example

```
<ntp-servers>
    server-fqdn time-d.nist.gov
    server-address 2610:20:6F15:15::27
    multicast-address ff08::101
</ntp-servers>
```

Section parameters

always-send Always send this option, even if the client didn't ask for it.

Default: "no"

<multiple> (required, multiple allowed) The key is the type of NTP server reference and the data is the corresponding reference. Built-in NTP server reference types are 'server-fqdn', 'server-address' and 'multicast-address'.

Example: "server-fqdn time-d.nist.gov"

Preference

This handler adds a preference level to the response.

Example

```
<preference>
    level 255
</preference>
```

Section parameters

level (required) The preference level. Higher is better. Preference 255 tells the client that it doesn't have to wait for other DHCPv6 servers anymore and that it should request from this server immediately.

Rate-limit

The most common reason that clients keep sending requests is when they get an answer they don't like. The best way to slow them down is to just stop responding to them.

Example

```
<rate-limit>
    key remote-id
    rate = 5
    per = 30
</rate-limit>
```

Section parameters

key The key to use to distinguish between clients. By default the DUID is used, but depending on your environment a different key may be appropriate. Possible values are:

- duid
- interface-id
- remote-id
- subscriber-id
- linklayer-id

If the chosen key is not available in the incoming request then the rate limiter will automatically fall back to identification by DUID.

Default: "duid"

rate The number of messages that a client may send per time slot.

Default: "5"

per The duration of a time slot in seconds.

Default: "30"

burst The burst size allowed.

Default: The same as the rate.

Recursive-name-servers

This sections adds recursive name servers to the response sent to the client. If there are multiple sections of this type then they will be combined into one set of recursive name servers which is sent to the client.

Example

```
<recursive-name-servers>
    address 2001:4860:4860::8888
    address 2001:4860:4860::8844
</recursive-name-servers>
```

Section parameters

always-send Always send this option, even if the client didn't ask for it.

Default: "no"

address (required, multiple allowed) The IPv6 address of a recursive name server.

Example: "2001:db8:1::53"

Server-unicast

This handler tells clients that they may contact it using unicast.

Example

```
<server-unicast>
    address 2001:db8::1:2:3
</server-unicast>
```

Section parameters

address (required) The IPv6 unicast address that the client may send requests to

Sip-server-addresses

This sections adds SIP server addresses to the response sent to the client. If there are multiple sections of this type then they will be combined into one set of servers which is sent to the client.

Example

```
<sip-server-addresses>
    address 2001:db8::1
    address 2001:db8::2
</sip-server-addresses>
```

Section parameters

always-send Always send this option, even if the client didn't ask for it.

Default: "no"

address (required, multiple allowed) The IPv6 address of a SIP server.

Example: "2001:db8::1"

Sip-server-names

This sections adds SIP server domain names to the response sent to the client. If there are multiple sections of this type then they will be combined into one set of domain names which is sent to the client.

The option MAY contain multiple domain names, but these SHOULD refer to different NAPTR records, rather than different A records.

Example

```
<sip-server-names>
    domain-name example.org
</sip-server-names>
```

Section parameters

always-send Always send this option, even if the client didn't ask for it.

Default: "no"

domain-name (required, multiple allowed) The domain name to add to the list. This should refer to a NAPTR record.

Example: "example.com"

Sntp-servers

This sections adds SNTP servers to the response sent to the client. If there are multiple sections of this type then they will be combined into one set of SNTP servers which is sent to the client.

Example

```
<sntp-servers>
    address 2610:20:6F15:15::27
</sntp-servers>
```

Section parameters

always-send Always send this option, even if the client didn't ask for it.

Default: "no"

address (required, multiple allowed) IPv6 address of an SNTP server

Example: "2610:20:6F15:15::27"

Sol-max-rt

This sections sets the SOL_MAX_RT value that will be sent to the client.

Example

```
<sol-max-rt>
    limit 43200
    always-send yes
</sol-max-rt>
```

Section parameters

always-send Always send this option, even if the client didn't ask for it.

Default: "no"

limit (required) Specify the number of seconds to send as MAX_SOL_RT. The value must be between 60 and 86400 seconds.

Example: "21600"

Static-csv

This section specifies that clients get their address and/or prefix assigned based on the contents of a CSV file. The filename is given as the name of the section. Relative paths are resolved relative to the configuration file.

The CSV file must have a heading defining the field names, and the fields `id`, `address` and `prefix` must be present. All other columns are ignored.

The id can refer to the *DUID of the client* (page 160), the *Interface-ID* (page 167) provided by the DHCPv6 relay closest to the client or the *Remote-ID* (page 82) provided by the DHCPv6 relay closest to the client. It is specified in one of these formats:

duid:*hex-value* where `hex-value` is a hexadecimal string containing the DUID of the client.

interface-id:*value* where `value` is the value of the interface-id in hexadecimal notation.

interface-id-str:*value* where `value` is the value of the interface-id in ascii notation.

remote-id:*enterprise-number*:*value* where `enterprise-number` is an enterprise number[9] as registered with IANA and `value` is the value of the remote-id in hexadecimal notation.

remote-id-str:*enterprise-number*:*value* where `enterprise-number` is an enterprise number[10] as registered with IANA and `value` is the value of the remote-id in ascii notation.

subscriber-id:*value* where `value` is the value of the subscriber-id in hexadecimal notation.

subscriber-id-str:*value* where `value` is the value of the subscriber-id in ascii notation.

linklayer-id:*type*:*value* where `type` is a hardware type assigned by the IANA, as described in **RFC 826**[11] (ethernet has type number 1) and `value` is the value of the link-layer address in hexadecimal notation.

linklayer-id-str:*type*:*value* where `type` is a hardware type assigned by the IANA, as described in **RFC 826**[12] (ethernet has type number 1) and `value` is the value of the link-layer address in ascii notation.

[9] http://www.iana.org/assignments/enterprise-numbers
[10] http://www.iana.org/assignments/enterprise-numbers
[11] https://tools.ietf.org/html/rfc826.html
[12] https://tools.ietf.org/html/rfc826.html

The address column can contain an IPv6 address and the prefix column can contain an IPv6 prefix in CIDR notation. Both the address and prefix columns may have empty values.

For example:

```
id,address,prefix
duid:000100011d1d6071002436ef1d89,,2001:db8:0201::/48
interface-id:4661322f31,2001:db8:0:1::2:2,2001:db8:0202::/48
interface-id-str:Fa2/2,2001:db8:0:1::2:3,
remote-id:9:020023000001000a0003000100211c7d486e,2001:db8:0:1::2:4,2001:db8:0204::/
↪48
remote-id-str:40208:SomeRemoteIdentifier,2001:db8:0:1::2:5,2001:db8:0205::/48
```

Example

```
<static-csv data/assignments.csv>
    address-preferred-lifetime 1d
    address-valid-lifetime 7d
    prefix-preferred-lifetime 3d
    prefix-valid-lifetime 30d
</static-csv>
```

Section parameters

address-preferred-lifetime The preferred lifetime of assigned addresses. This is the time that the client should use it as the source address for new connections. After the preferred lifetime expires the address remains valid but becomes deprecated.

The value is specified in seconds. For ease of use these suffixes may be used: 's' (seconds), 'm' (minutes), 'h' (hours), or 'd' (days).

Default: "7d"

address-valid-lifetime The valid lifetime of assigned addresses. After this lifetime expires the client is no longer allowed to use the assigned address.

The value is specified in seconds. For ease of use these suffixes may be used: 's' (seconds), 'm' (minutes), 'h' (hours), or 'd' (days).

Default: "30d"

prefix-preferred-lifetime The preferred lifetime of assigned prefixes. This is the time that the client router should use as a preferred lifetime value when advertising prefixes to its clients.

The value is specified in seconds. For ease of use these suffixes may be used: 's' (seconds), 'm' (minutes), 'h' (hours), or 'd' (days).

Default: "7d"

prefix-valid-lifetime The valid lifetime of assigned prefixes. This is the time that the client router should use as a valid lifetime value when advertising prefixes to its clients.

The value is specified in seconds. For ease of use these suffixes may be used: 's' (seconds), 'm' (minutes), 'h' (hours), or 'd' (days).

Default: "30d"

Static-sqlite

This section specifies that clients get their address and/or prefix assigned based on the contents of a SQLite database. The filename of the database is given as the name of the section. Relative paths are resolved relative to

the configuration file.

The main advantages of using a SQLite database instead of a CSV file are:

- The CSV implementation reads all assignments into memory on startup, the SQLite implementation doesn't
- The SQLite file can be modified while the server is running, and the changes are used without the need to restart the server.

The SQLite database needs to have a table called `assignments` with TEXT columns `id`, `address` and `prefix`. Their contents use the same structure as the corresponding columns in the *CSV file* (page 23).

The *ipv6-dhcp-build-sqlite* command can be used to convert a CSV file into the right SQLite database format.

Example

```
<static-sqlite data/assignments.sqlite>
    address-preferred-lifetime 1d
    address-valid-lifetime 7d
    prefix-preferred-lifetime 3d
    prefix-valid-lifetime 30d
</static-csv>
```

Section parameters

address-preferred-lifetime The preferred lifetime of assigned addresses. This is the time that the client should use it as the source address for new connections. After the preferred lifetime expires the address remains valid but becomes deprecated.

The value is specified in seconds. For ease of use these suffixes may be used: 's' (seconds), 'm' (minutes), 'h' (hours), or 'd' (days).

Default: "7d"

address-valid-lifetime The valid lifetime of assigned addresses. After this lifetime expires the client is no longer allowed to use the assigned address.

The value is specified in seconds. For ease of use these suffixes may be used: 's' (seconds), 'm' (minutes), 'h' (hours), or 'd' (days).

Default: "30d"

prefix-preferred-lifetime The preferred lifetime of assigned prefixes. This is the time that the client router should use as a preferred lifetime value when advertising prefixes to its clients.

The value is specified in seconds. For ease of use these suffixes may be used: 's' (seconds), 'm' (minutes), 'h' (hours), or 'd' (days).

Default: "7d"

prefix-valid-lifetime The valid lifetime of assigned prefixes. This is the time that the client router should use as a valid lifetime value when advertising prefixes to its clients.

The value is specified in seconds. For ease of use these suffixes may be used: 's' (seconds), 'm' (minutes), 'h' (hours), or 'd' (days).

Default: "30d"

Leasequery_store

Configuration sections that define Leasequery stores. Each leasequery section must configure exactly one store. Stores perform the storing of lease data at the end of a DHCPv6 request. They also handle the queries from Leasequery clients to search in that stored data.

Lq-sqlite

This leasequery store will store observed leases seen in DHCPv6 reply messages in the SQLite database whose name is provided as the name of the section. It implements the query types from both the Leasequery (**RFC 5007**[13]) and Bulk Leasequery (**RFC 5460**[14]) protocol extensions.

Example

```
<lq-sqlite /var/lib/dhcpkit/leasequery.sqlite />
```

Listeners

Configuration sections that define listeners. These are usually the network interfaces that a DHCPv6 server listens on, like the well-known multicast address on an interface, or a unicast address where a DHCPv6 relay can send its requests to.

Listen-interface

This listener listens to the DHCPv6 server multicast address on the interface specified as the name of this section. This is useful to listen for clients on a directly connected LAN.

Example

```
<listen-interface eth0>
    listen-to-self yes
    reply-from fe80::1
    link-address 2001:db8::1
</listen-interface>
```

Section parameters

mark (multiple allowed) Every incoming request can be marked with different tags. That way you can handle messages differently based on i.e. which listener they came in on. Every listener can set one or more marks. Also see the *Marked-with* (page 11) filter.

Default: "unmarked"

listen-to-self Usually the server doesn't listen to requests coming from the local host. If you want the server to assign addresses to itself (also useful when debugging) then enable this.

Default: "no"

reply-from The link-local address to send replies from

Default: The first link-local address found on the interface

link-address A global unicast address used to identify the link to filters and handlers. It doesn't even need to exist.

Default: The first global unicast address found on the interface, or `::` otherwise

[13] https://tools.ietf.org/html/rfc5007.html
[14] https://tools.ietf.org/html/rfc5460.html

Listen-tcp

This listener listens for TCP connections on the unicast address specified as the name of the section. This is for BulkLeasequery support, but as an extension the server will also answer other types of messages.

Example

```
<listen-tcp>
    address 2001:db8::1:2

    allow-from 2001:db8::ffff:1
    allow-from 2001:db8:1:2::/64
</listen-tcp>
```

Section parameters

mark (multiple allowed) Every incoming request can be marked with different tags. That way you can handle messages differently based on i.e. which listener they came in on. Every listener can set one or more marks. Also see the *Marked-with* (page 11) filter.

Default: "unmarked"

address (required) Accept TCP connections on the specified address.

Example: "2001:db8::ffff:1"

max-connections Limit the number of accepted TCP connections. Servers MUST be able to limit the number of currently accepted and active connections.

Example: "20"

Default: "10"

allow-from (multiple allowed) TCP connections are not used for normal operations. They are used by Leasequery clients and other trusted clients for management purposes. Therefore you can specify from which clients to accept connections.

Not specifying any trusted clients will allow connections from everywhere. This is strongly not recommended.

Also note that this only limits which clients may set up a TCP connection to this server. The leasequery section contains a list of clients which are allowed to use the leasequery protocol. Clients that are allowed to connect over TCP should probably also be allowed to perform leasequeries.

Example:

```
allow-from 2001:db8::ffff:1
allow-from 2001:db8:beef::/48
```

Listen-unicast

This listener listens to the unicast address specified as the name of the section. This is useful when you configure a DHCP relay to forward requests to this server.

Example

```
<listen-unicast 2001:db8::1:2 />
```

Section parameters

mark (multiple allowed) Every incoming request can be marked with different tags. That way you can handle messages differently based on i.e. which listener they came in on. Every listener can set one or more marks. Also see the *Marked-with* (page 11) filter.

Default: "unmarked"

Loghandler

Log-handlers output log entries to somewhere. If you want to send your logs somewhere configure one of these. There are log-handlers to show log entries on the console. Send them to a syslog process, server, etc.

Console

Log to console.

Example

```
<console>
    level debug-handling
    color yes
</console>
```

Section parameters

level The log level. Only log messages with a priority of this level or higher are logged to this output. Possible values are:

"critical" Log critical errors that prevent the server from working

"error" Log errors that occur

"warning" Log warning messages that might indicate a problem

"info" Log informational messages

"debug" Log messages that are usually only useful when debugging issues

"debug-packets" Log the sending and receiving of packets

"debug-handling" Log everything about how a request is handled

Default: "warning"

color Whether to show log entries in colour

Default: auto-detect colour support

File

Log to a file. The name of the section is the filename of the log file.

Example

```
<file /var/log/dhcpkit/dhcpd.log>
    rotate daily
    keep 7
    level info
</file>
```

Section parameters

level The log level. Only log messages with a priority of this level or higher are logged to this output. Possible values are:

"critical" Log critical errors that prevent the server from working

"error" Log errors that occur

"warning" Log warning messages that might indicate a problem

"info" Log informational messages

"debug" Log messages that are usually only useful when debugging issues

"debug-packets" Log the sending and receiving of packets

"debug-handling" Log everything about how a request is handled

Default: "warning"

rotate Rotate the log file automatically. Valid options are:

"hourly" or "hour" Rotate the log file every hour

"daily" or "day" Rotate the log file every day

"weekly" or "week" Rotate the log file every week

"size" Rotate the log file based on size

Default: do not rotate based

size When rotating based on size a file size must be specified. You can use the suffixed "kb", "mb" or "gb" to make the value more readable.

keep When rotating log files you must specify how many files to keep.

Syslog

Log to local syslog. The name of the section is the destination, which can be a `hostname:port` or a unix socket file name. Relative names are resolved relative to the directory containing the configuration file.

Example

```
# This will try to auto-detect the syslog socket using the default level
<syslog />

# This logs explicitly to the specified socket using a non-default facility
<syslog /var/run/syslog>
    facility local3
    level info
</syslog>

# This logs explicitly to the specified socket using a non-default protocol
<syslog collector.example.com:514>
```

(continues on next page)

(continued from previous page)

```
    facility local1
    protocol tcp
</syslog>
```

Section parameters

level The log level. Only log messages with a priority of this level or higher are logged to this output. Possible values are:

"critical" Log critical errors that prevent the server from working

"error" Log errors that occur

"warning" Log warning messages that might indicate a problem

"info" Log informational messages

"debug" Log messages that are usually only useful when debugging issues

"debug-packets" Log the sending and receiving of packets

"debug-handling" Log everything about how a request is handled

Default: "warning"

facility Use the specified log facility. The available facilities are system-dependent but usually include "daemon", "local0" to "local7", "auth", "user" and "syslog".

Default: "daemon"

protocol Use a datagram ("dgram" or "udp") or stream ("stream" or "tcp") connection

Default: "dgram"

1.3 Developer's guide

Adapting dhcpkit to your needs might require some custom development. There are several areas where you can customise the server's behaviour:

1.3.1 Writing custom options

Implementing new options usually comes down to writing a new *Option* (page 168) class to store the option's content, validate the option's contents, and parse and generate the bytes that represent the option on the wire.

Class properties

Each option class must have a property that defines the option type code implemented by the class. The list of option codes[15] is maintained by IANA[16]. A common way of setting the option type code is by defining a constant for the code and then using that in the class definition for readability:

```
OPTION_DNS_SERVERS = 23

class RecursiveNameServersOption(Option):
    option_type = OPTION_DNS_SERVERS
```

[15] http://www.iana.org/assignments/dhcpv6-parameters/
[16] http://www.iana.org/

Constructor and properties

Because an option (any *ProtocolElement* (page 218)) is defined by its type and contents, the constructor must reflect that: all relevant properties must correspond to parameters of the option's constructor. This requirement makes it possible to automate comparison of protocol elements and to print their state in a readable `__str__()`[17] and parseable `__repr__()`[18] format.

An example is `RecursiveNameServersOption.__init__()`. As you can see `dns_servers` is both the name of the constructor parameter as the name of the state variable:

```
    def __init__(self, dns_servers: Iterable[IPv6Address] = None):
        self.dns_servers = list(dns_servers or [])
        """List of IPv6 addresses of resolving DNS servers"""
```

Validation

Next is the validation. Each option must be able to verify if its state is acceptable and can be encoded to bytes that can be sent on the wire.

Note: Additionally the validator may make sure that the information makes sense, but be aware that incoming messages that violate these checks will be rejected before even reaching the message handler, so make sure that is what you want.

An example is *RecursiveNameServersOption.validate()* (page 51) which checks that *dns_servers* (page 51) is a list of `IPv6Address`[19]:

```
    def validate(self):
        """
        Validate that the contents of this object conform to protocol specs.
        """
        if not isinstance(self.dns_servers, list):
            raise ValueError("DNS servers must be a list")

        for address in self.dns_servers:
            if not isinstance(address, IPv6Address):
                raise ValueError("DNS server must be an IPv6 address")
```

Parsing and generating binary representation

These are the most complex parts of an *Option* (page 168) implementation. The *load_from()* (page 219) method must be able to parse valid binary representations of the option. Its parameters are a string of bytes and an optional offset and length. It should start parsing at the specified offset and read up to the specified length from the buffer. The *load_from()* (page 219) method must return the number of bytes that it has used/parsed so that the caller knows which offset to give to any subsequent option parsers.

All options start with the same fields, which include the option type and the length of the option. That part is called the option header and is parsed with *parse_option_header()* (page 168). This will automatically make sure that the `length` the caller provided is enough to contain this option's data.

An option parser should make sure that all read data is verified and that all the data up to the option length is read and parsed. After parsing the data the properties of the object should correspond to the binary string's contents.

Here is the implementation of *RecursiveNameServersOption.load_from()* (page 51):

[17] https://docs.python.org/3.4/reference/datamodel.html#object.__str__
[18] https://docs.python.org/3.4/reference/datamodel.html#object.__repr__
[19] https://docs.python.org/3.4/library/ipaddress.html#ipaddress.IPv6Address

```
    def load_from(self, buffer: bytes, offset: int = 0, length: int = None) -> int:
        """
        Load the internal state of this object from the given buffer. The buffer may
        contain more data after the structured element is parsed. This data is ignored.

        :param buffer: The buffer to read data from
        :param offset: The offset in the buffer where to start reading
        :param length: The amount of data we are allowed to read from the buffer
        :return: The number of bytes used from the buffer
        """
        my_offset, option_len = self.parse_option_header(buffer, offset, length)
        header_offset = my_offset

        if option_len % 16 != 0:
            raise ValueError('DNS Servers Option length must be a multiple of 16')

        # Parse the addresses
        self.dns_servers = []
        max_offset = option_len + header_offset
        while max_offset > my_offset:
            address = IPv6Address(buffer[offset + my_offset:offset + my_offset + 16])
            self.dns_servers.append(address)
            my_offset += 16

        return my_offset
```

The reverse operation of *load_from()* (page 219) is *save()* (page 219). It should generate bytes to represent its properties. Here is the implementation of *RecursiveNameServersOption.save()* (page 51):

```
    def save(self) -> Union[bytes, bytearray]:
        """
        Save the internal state of this object as a buffer.

        :return: The buffer with the data from this element
        """
        buffer = bytearray()
        buffer.extend(pack('!HH', self.option_type, len(self.dns_servers) * 16))
        for address in self.dns_servers:
            buffer.extend(address.packed)

        return buffer
```

Note: Determining which option type is next in the incoming bytes, creating the right object for it and then loading its state with *load_from()* (page 219) from bytes is so common that there is a shortcut for that: *parse()* (page 219). This uses the option registry to determine the correct object class. See *Option.determine_class()* (page 168).

Note: *load_from()* (page 219) must be able to parse all valid binary representations of the option. Calling *save()* (page 219) should produce the original binary representation again. The following should be true:

```
# A RecursiveNameServersOption:
from dhcpkit.ipv6.options import Option
from dhcpkit.ipv6.extensions.dns import RecursiveNameServersOption

binary_representation = b'\x00\x17\x00 ' \
```

(continues on next page)

(continued from previous page)

```
    b' \x01H`H`\x00\x00\x00\x00\x00\x00\x00\x00\x88\x88' \
    b' \x01H`H`\x00\x00\x00\x00\x00\x00\x00\x00\x88D'

read_bytes, parsed_option = Option.parse(binary_representation)
assert type(parsed_option) == RecursiveNameServersOption
assert binary_representation == parsed_option.save()
```

Registering new options

New options must be registered so that the server knows which classes are available for parsing DHCP options. This is done by defining entry points in the setup script:

```
setup(
    name='dhcpkit_demo_extension',
    ...
    entry_points={
        'dhcpkit.ipv6.options': [
            '65535 = dhcpkit_demo_extension.package.module:MyOptionClass',
        ],
    },
)
```

Each protocol element also keeps track of which (sub)options it may contain. According to **RFC 3646#section-5**[20] the recursive name servers option may appear in Solicit, Advertise, Request, Renew, Rebind, Information-Request, and Reply messages. We need to let the classes for those messages know that they may contain this option:

```
SolicitMessage.add_may_contain(RecursiveNameServersOption, 0, 1)
AdvertiseMessage.add_may_contain(RecursiveNameServersOption, 0, 1)
RequestMessage.add_may_contain(RecursiveNameServersOption, 0, 1)
RenewMessage.add_may_contain(RecursiveNameServersOption, 0, 1)
RebindMessage.add_may_contain(RecursiveNameServersOption, 0, 1)
InformationRequestMessage.add_may_contain(RecursiveNameServersOption, 0, 1)
ReplyMessage.add_may_contain(RecursiveNameServersOption, 0, 1)
```

Here we have specified that the RecursiveNameServersOption has a `min_occurrence` of `0` and a `max_occurrence` of `1` in each of these message types. If no `min_occurrence` and `max_occurrence` are specified when calling *`add_may_contain()`* (page 218) they default to `0` and `infinite` respectively.

1.3.2 Writing custom handlers

Writing a custom handler is, together with *writing custom options* (page 30), the most common way of customising the DHCPKit server. Handlers process the incoming message and adapt the outgoing message. There are many things a handler could do. Some of the common use cases are:

- assigning addresses/prefixes to incoming *`IANAOption`* (page 163), *`IATAOption`* (page 165) and *`IAPDOption`* (page 75) requests (see e.g. *`CSVStaticAssignmentHandler`* (page 110))
- providing *`RecursiveNameServersOption`* (page 50) to clients (see *`RecursiveNameServersOptionHandler`* (page 92))
- limiting the maximum values for T1/T2 so that clients come back often enough for renewal of their addresses (see e.g. *`IANATimingLimitsHandler`* (page 112))

[20] https://tools.ietf.org/html/rfc3646.html#section-5

Basic handler structure

All handlers must be subclasses of *Handler* (page 121) or *RelayHandler* (page 121). Each handler must be registered as a server extension so that the server code is aware of their existence.

A handler usually implements its functionality by overriding the *handle()* (page 121) method (or *handle_relay()* (page 122) in the case of *RelayHandler* (page 121)). This method gets a *TransactionBundle* (page 144) as its only parameter `bundle`. The bundle contains all the information available about a request and the response. Handlers providing information (e.g. DNS information) commonly look at whether the client included an *OptionRequestOption* (page 169) in its *request* (page 146) and based on that information decide to add an extra option to the *response* (page 146).

Because there are several very common patterns here are some base classes you can use:

- *SimpleOptionHandler* (page 123) which adds a static option instance to responses
- `OverwritingOptionHandler` which overwrites all options of the same class with adds a static option instance
- *CopyOptionHandler* (page 122) which copies options from a certain class from the request to the response
- *CopyRelayOptionHandler* (page 124) which copies options from a certain class from each incoming *RelayForwardMessage* (page 155) to the corresponding *RelayReplyMessage* (page 155).

Loading handlers from the configuration file

There are two parts to creating new handlers that can be used in the configuration file. The first part is the XML definition of what the configuration section looks like. The second part is a factory function or object that will create the handler from the configuration.

Defining the configuration section

If you want your handler to be loadable from the configuration file you need to provide a `ZConfig` `component.xml` schema file that determines what your configuration section will look like. A configuration section definition can look like this:

```
<component xmlns="https://raw.githubusercontent.com/zopefoundation/ZConfig/master/
↪doc/schema.dtd"
           prefix="dhcpkit.ipv6.server.extensions.dns.config">

    <sectiontype name="recursive-name-servers"
                 extends="option_handler_factory_base"
                 implements="handler_factory"
                 datatype=".RecursiveNameServersOptionHandlerFactory">

        <description><![CDATA[
            This sections adds recursive name servers to the response sent to the
            client. If there are multiple sections of this type then they will be
            combined into one set of recursive name servers which is sent to the
            client.
        ]]></description>

        <example><![CDATA[
            <recursive-name-servers>
                address 2001:4860:4860::8888
                address 2001:4860:4860::8844
            </recursive-name-servers>
        ]]></example>

        <multikey name="address" attribute="addresses" required="yes"
```

(continues on next page)

(continued from previous page)

```
                datatype="ipaddress.IPv6Address">
            <description>
                The IPv6 address of a recursive name server.
            </description>
            <example>
                2001:db8:1::53
            </example>
        </multikey>
    </sectiontype>

    <sectiontype name="domain-search-list"
                 extends="option_handler_factory_base"
                 implements="handler_factory"
                 datatype=".DomainSearchListOptionHandlerFactory">

        <description><![CDATA[
            This sections adds domain names to the domain search list sent to the
            client. If there are multiple sections of this type then they will be
            combined into one set of domain names which is sent to the client.
        ]]></description>

        <example><![CDATA[
            <domain-search-list>
                domain-name example.com
                domain-name example.net
                domain-name example.org
            </domain-search-list>
        ]]></example>

        <multikey name="domain-name" attribute="domain_names" required="yes"
                  datatype="dhcpkit.common.server.config_datatypes.domain_name">
            <description>
                The domain name to add to the search list.
            </description>
            <example>
                example.com
            </example>
        </multikey>
    </sectiontype>
</component>
```

This component describes two section types: `recursive-name-servers` and `domain-search-list`. They both have `implements="handler_factory"` which makes them usable as a handler. The datatypes of the sections are relative to `prefix="dhcpkit.ipv6.server.extensions.dns.config"` because they start with a `.`.

The datatypes of `<key>` and `<multikey>` elements can be one of the ZConfig predefined types or anything that can be called like a function which accepts the string value of what the user put into the configuration file as its single parameter. Its return value is stored as the value. This behaviour also allows you to provide a class as the datatype. Its constructor will be called with a single string argument. In the example above you can see this for the `<multikey name="address" ...` where the datatype is the `IPv6Address`[21] class from `ipaddress`[22].

The `<description>` and `<example>` tags are used when generating documentation. The whole *configuration section* (page 5) of this manual is created based on such tags!

[21] https://docs.python.org/3.4/library/ipaddress.html#ipaddress.IPv6Address
[22] https://docs.python.org/3.4/library/ipaddress.html#module-ipaddress

Writing the handler factory

After parsing a section and converting its values using the specified datatypes, the datatype of the sectiontype will be called with a `ZConfig.SectionValue` object containing all the values as its only parameter. The return value of that datatype must be callable as a function, which acts as a factory for the actual handler.

Note: The reason that a factory is used is for privilege separation. The configuration file is read as the user that started the server process, usually `root`, while the factory is called with the privileges of the user and group that the server is configured to run as. This makes sure that e.g. all files created by a handler have the right ownership.

The easiest way to write a handler factory is to create a subclass of *HandlerFactory* (page 121) and create the *Handler* (page 121) in the implementation of the *create()* (page 42) method. Because *HandlerFactory* (page 121) is a subclass of *ConfigSection* (page 42) you can use its functionality to assist with processing configuration sections. If some of the values in the configuration are optional and the default value has to be determined at runtime you can modify *section* (page 42) in *clean_config_section()* (page 42). If the configuration values need extra validation then do so in *validate_config_section()* (page 43). For convenience you can access the configuration values both as *self.section.xyz* and as *self.xyz*.

If you want your section to have a "name" like in:

```
<static-csv data/assignments.csv>
    prefix-preferred-lifetime 3d
    prefix-valid-lifetime 30d
</static-csv>
```

You can set the *name_datatype* (page 42) to the function or class that should be used to parse the name.

This is a complete example that uses both the name and other section values:

```
class CSVStaticAssignmentHandlerFactory(HandlerFactory):
    """
    Factory for a handler that reads assignments from a CSV file
    """

    name_datatype = staticmethod(existing_file)

    def create(self) -> CSVStaticAssignmentHandler:
        """
        Create a handler of this class based on the configuration in the config␣
↪section.

        :return: A handler object
        """

        # Get the lifetimes
        address_preferred_lifetime = self.address_preferred_lifetime
        address_valid_lifetime = self.address_valid_lifetime
        prefix_preferred_lifetime = self.prefix_preferred_lifetime
        prefix_valid_lifetime = self.prefix_valid_lifetime

        return CSVStaticAssignmentHandler(
            self.name,
            address_preferred_lifetime, address_valid_lifetime,
            prefix_preferred_lifetime, prefix_valid_lifetime
        )
```

Handler initialisation

Handlers are initialised in two steps. The first step is when the factory creates the handler object. This happens in the main server process just before the worker processes are spawned. Those worker processes get a copy of the

handlers when the worker is being initialised. This is done by pickling[23] the *MessageHandler* (page 139) and all the filters and handlers it contains. The advantage is that workers don't need to initialise everything themselves (especially if that initialisation can take a long time, like when parsing a CSV file) but it also means that things that cannot be pickled can therefore not be initialised when creating the handler. Therefore handlers have a separate *worker_init()* (page 121) method that is called inside each worker. Initialisation that need to happen in each worker process (for example opening database connections) can be done there.

Registering new handlers

New handlers must be registered so that the server knows which sections are available when parsing the server configuration. This is done by defining entry points in the setup script:

```
setup(
    name='dhcpkit_demo_extension',
    ...
    entry_points={
        'dhcpkit.ipv6.server.extensions': [
            'handler-name = dhcpkit_demo_extension.package',
        ],
    },
)
```

If the package contains a file called `component.xml` then that file is used as an extension to the configuration file syntax.

More advanced message handling

If necessary a handler can do *pre()* (page 121) and *post()* (page 121) processing. Pre processing can be useful in cases where an incoming request has to be checked to see if it should be handled at all or whether processing should be aborted. Post processing can be used for cleaning up, checking that all required options are included in the response, committing leases to persistent storage, etc.

The post processing stage is especially important to handlers that assign resources. In the *handle()* (page 121) method the handler puts its assignments in the response. That doesn't mean that that response is actually sent to the client. Another handler might change the response or abort the processing later.

Handlers that have to store state should do that during post processing after verifying the response. If rapid commit is active the response might even have changed from an *AdvertiseMessage* (page 152) to a *ReplyMessage* (page 157). Handlers that store data based on whether a resource was only advertised or whether it was actually assigned must look at the response being sent to determine that.

Handling rapid commit

Usually rapid commit is handled by its own built-in handler. If a handler does not want a rapid commit to happen it can set the *allow_rapid_commit* (page 145) attribute of the transaction bundle to False. The built-in handler will take that into account when deciding whether it performs a rapid commit or not.

Rules for handlers that assign resources

Options meant for assigning addresses and prefixes like *IANAOption* (page 163), *IATAOption* (page 165) and *IAPDOption* (page 75) are a bit more complex to handle. The way handlers are designed in dhcpkit is that each such option can be handled by one handler. A handler that assigns addresses should use the *bundle.get_unhandled_options* (page 145) method to find those options in the request that haven't been handled yet:

[23] https://docs.python.org/3.4/library/pickle.html#module-pickle

After handling an option the handler must mark that option as handled by calling *bundle.mark_handled* (page 145) with the handled option as parameter. This will let handlers that are executed later know which options still need to be handled.

When handling *ConfirmMessage* (page 153), *ReleaseMessage* (page 157) and *DeclineMessage* (page 154) the handler should behave as follows:

- It should mark as handled the options that it is responsible for
- If the confirm/release/decline is successful it should not modify the response
- If the confirm/release/decline is **not** successful it should put the appropriate options and/or status code in the response
- If a previous handler has already put a negative status code in the response then that status code should be left intact

The *built-in message handler* (page 139) will automatically apply handlers that check for any unhandled options and set the status code if it hasn't been set by any other handler.

Aborting message handling

There are cases where a handler decides that the current request should not be handled by this server at all. One example is when a handler determines that the *ServerIdOption* (page 174) in the request refers to a difference *DUID* (page 148) than that of the server. In those cases the handler can throw a *CannotRespondError* (page 121) exception. This will stop all processing and prevent a response from being sent to the client.

A handler should not abort in the post processing phase. When post processing starts all handlers should be able to assume that the response is finished and that they can rely on the response being sent.

Example of a Handler

This is the implementation of *RecursiveNameServersOptionHandler* (page 92). As you can see most of the code is for processing the configuration data so that this handler can be added through the configuration file as described in the *Recursive-name-servers* (page 21) manual page.

```
class RecursiveNameServersOptionHandler(SimpleOptionHandler):
    """
    Handler for putting RecursiveNameServersOption in responses
    """

    def __init__(self, dns_servers: Iterable[IPv6Address], always_send: bool =␣
↪False):
        option = RecursiveNameServersOption(dns_servers=dns_servers)
        option.validate()

        super().__init__(option, always_send=always_send)

    def __str__(self):
        return "{} with {}".format(self.__class__.__name__, ', '.join(map(str,␣
↪self.option.dns_servers)))

    def combine(self, existing_options: Iterable[RecursiveNameServersOption]) ->␣
↪RecursiveNameServersOption:
        """
        Combine multiple options into one.

        :param existing_options: The existing options to include name servers from
        :return: The combined option
        """
        addresses = []
```

(continues on next page)

(continued from previous page)

```
        # Add from existing options first
        for option in existing_options:
            for address in option.dns_servers:
                if address not in addresses:
                    addresses.append(address)

        # Then add our own
        for address in self.option.dns_servers:
            if address not in addresses:
                addresses.append(address)

        # And return a new option with the combined addresses
        return RecursiveNameServersOption(dns_servers=addresses)
```

Example of a RelayHandler

This is the implementation of *InterfaceIdOptionHandler* (page 125) which copies *InterfaceIdOption* (page 167) from incoming relay messages to outgoing relay messages. The implementation is very simple:

```
class InterfaceIdOptionHandler(CopyRelayOptionHandler):
    """
    The handler for InterfaceIdOptions in relay messages
    """

    def __init__(self):
        super().__init__(InterfaceIdOption)
```

If you are developing software to work with DHCPKit you probably want to know all the modules, classes and functions you can use. Below is a complete list of everything in the package:

1.3.3 dhcpkit package

DHCPKit internals

Subpackages

dhcpkit.common package

Common components that might be usable for both IPv4 and IPv6

Subpackages

dhcpkit.common.logging package

Common logging related functionality

Submodules

dhcpkit.common.logging.verbosity module

Basic console logging based on verbosity

dhcpkit.common.logging.verbosity.**set_verbosity_logger**(*logger: logging.Logger*, *verbosity: int*, *existing_console: logging.Handler = None*)

Install a console based logger based based on the given verbosity.

Parameters

- **logger** – The logger to add the handlers to
- **verbosity** – The verbosity level given as command line argument
- **existing_console** – The existing console handler

dhcpkit.common.server package

Common components that might be usable for both an IPv4 and an IPv6 server

Subpackages

dhcpkit.common.server.logging package

Common logging component

Submodules

dhcpkit.common.server.logging.config_datatypes module

Datatypes useful for the logging component

dhcpkit.common.server.logging.config_datatypes.**logging_level**(*value: str*) → int

Convert the strings representing logging levels to their numerical value

Parameters **value** – The string representing the logging level

Returns Numerical logging level

dhcpkit.common.server.logging.config_datatypes.**rotation_style**(*value: str*) → str

Determine the rotation style.

Parameters **value** – String representation of rotation style

Returns Normalised rotation style

dhcpkit.common.server.logging.config_datatypes.**syslog_facility**(*value: str*) → int

Convert the strings representing syslog facilities to their numerical value

Parameters **value** – The string representing the syslog facility

Returns Numerical syslog facility

dhcpkit.common.server.logging.config_datatypes.**udp_or_tcp**(*value: str*) → int

Convert the strings “udp” and “tcp” to SOCK_DGRAM and SOCK_STREAM respectively

Parameters **value** – The string “udp” or “tcp”

Returns SOCK_DGRAM or SOCK_STREAM

dhcpkit.common.server.logging.config_elements module

The basic configuration objects for logging

class dhcpkit.common.server.logging.config_elements.**ConsoleHandlerFactory**(*section: ZConfig.matcher.SectionV*

Bases: *dhcpkit.common.server.config_elements.ConfigElementFactory* (page 42)

Factory for a logging handler that logs to the console, optionally in colour.

create() → logging.StreamHandler
Create a console handler

Returns The logging handler

validate_config_section()
Validate the colorlog setting

class dhcpkit.common.server.logging.config_elements.**FileHandlerFactory**(*section: ZConfig.matcher.SectionValue*

Bases: *dhcpkit.common.server.config_elements.ConfigElementFactory* (page 42)

Factory for a logging handler that logs to a file, optionally rotating it.

create() → logging.StreamHandler
Create a console handler

Returns The logging handler

static name_datatype(*v*)

validate_config_section()
Validate if the combination of settings is valid

class dhcpkit.common.server.logging.config_elements.**Logging**(*section: ZConfig.matcher.SectionValue*)

Bases: *dhcpkit.common.server.config_elements.ConfigSection* (page 42)

Class managing the configured logging handlers.

configure(*logger: logging.Logger*, *verbosity: int = 0*) → int
Add all configured handlers to the supplied logger. If verbosity > 0 then make sure we have a console logger and force the level of the console logger based on the verbosity.

Parameters

- **logger** – The logger to add the handlers to
- **verbosity** – The verbosity level given as command line argument

Returns The lowest log level that is going to be handled

validate_config_section()
Check for duplicate handlers

class dhcpkit.common.server.logging.config_elements.**SysLogHandlerFactory**(*section: ZConfig.matcher.SectionVa*

Bases: *dhcpkit.common.server.config_elements.ConfigElementFactory* (page 42)

Factory for a logging handler that logs to syslog.

clean_config_section()
Fill in the name automatically if not given

create() → logging.handlers.SysLogHandler
Create a syslog handler

Returns The logging handler

default_destinations = ('/dev/log', '/var/run/syslog', 'localhost:514')

static name_datatype(*value*)

Submodules

dhcpkit.common.server.config_datatypes module

Extra datatypes for the IPv6 DHCP server

dhcpkit.common.server.config_datatypes.**domain_name**(*value: str*) → str
Validate and clean a domain name.

Parameters **value** – Domain name

Returns Validated and cleaned domain name

dhcpkit.common.server.config_datatypes.**user_name**(*value: str*) → pwd.struct_passwd
Validate the given user name

Parameters **value** – User name

Returns Resolved user

dhcpkit.common.server.config_datatypes.**group_name**(*value: str*) → grp.struct_group
Validate the given group name

Parameters **value** – Group name

Returns Resolved group

dhcpkit.common.server.config_elements module

The basic configuration objects

class dhcpkit.common.server.config_elements.**ConfigElementFactory**(*section: ZConfig.matcher.SectionValue*)
Bases: *dhcpkit.common.server.config_elements.ConfigSection* (page 42)

Base class for factories to create elements from configuration

create(**args*, ***kwargs*) → object
Override this method to create the element.

Returns The element

class dhcpkit.common.server.config_elements.**ConfigSection**(*section: ZConfig.matcher.SectionValue*)
Bases: object[24]

Basic configuration section

clean_config_section()
Clean up the config, calculating defaults etc.

name = None
The parsed value of the section name

name_datatype = None
The datatype of the name of this section. Sections with datatype None cannot have a name

[24] https://docs.python.org/3.4/library/functions.html#object

section = None
The *SectionValue* we received as input from the parser

to_str(*indent: int = 0*) → str
Return a readable string representation of this element. Because it is not in the real configuration file format we don't attempt to make it look like one. We intentionally make it look different.

Parameters **indent** – How much indentation at the start of this element

Returns The formatted representation

validate_config_section()
Validate if the information in the config section is acceptable

Submodules

dhcpkit.common.privileges module

Common code to handle privileges

dhcpkit.common.privileges.**drop_privileges**(*user: pwd.struct_passwd, group: grp.struct_group, permanent: bool = True*)
Drop root privileges and change to something more safe.

Parameters

- **user** – The tuple with user info
- **group** – The tuple with group info
- **permanent** – Whether we want to drop just the euid (temporary), or all uids (permanent)

dhcpkit.common.privileges.**restore_privileges**()
Restore root privileges

dhcpkit.ipv6 package

Constants relevant for the IPv6 DHCP protocol

Subpackages

dhcpkit.ipv6.client package

DHCPv6 client related code

Submodules

dhcpkit.ipv6.client.test_leasequery module

A simple DHCPv6 client to send/receive messages from a DHCPv6 server

class dhcpkit.ipv6.client.test_leasequery.**ClientSocket**
Bases: object[25]

Base class for client sockets

[25] https://docs.python.org/3.4/library/functions.html#object

recv() → Tuple
Receive a DHCPv6 message

Returns The message

send(*message: dhcpkit.ipv6.messages.Message*) → ipaddress.IPv6Address
Send a DHCPv6 message

Parameters **message** – The message

set_timeout(*timeout: float*)
Set the timeout on the socket

Parameters **timeout** – Timeout in seconds

class dhcpkit.ipv6.client.test_leasequery.**TCPClientSocket**(*options*)
Bases: *dhcpkit.ipv6.client.test_leasequery.ClientSocket* (page 43)

Client socket for TCP connections

recv() → Tuple
Receive a DHCPv6 message

Returns The message

send(*message: dhcpkit.ipv6.messages.Message*) → ipaddress.IPv6Address
Send a DHCPv6 message

Parameters **message** – The message

set_timeout(*timeout: float*)
Set the timeout on the socket

Parameters **timeout** – Timeout in seconds

class dhcpkit.ipv6.client.test_leasequery.**UDPClientSocket**(*options*)
Bases: *dhcpkit.ipv6.client.test_leasequery.ClientSocket* (page 43)

Client socket for UDP connections

recv() → Tuple
Receive a DHCPv6 message

Returns The message

send(*message: dhcpkit.ipv6.messages.Message*) → ipaddress.IPv6Address
Send a DHCPv6 message

Parameters **message** – The message

set_timeout(*timeout: float*)
Set the timeout on the socket

Parameters **timeout** – Timeout in seconds

dhcpkit.ipv6.client.test_leasequery.**create_client_address_query**(*options*) → dhcpkit.ipv6.extensions.leasequery.LQQ
Create query option for address query.

Parameters **options** – Options from the main argument parser

Returns The Leasequery

dhcpkit.ipv6.client.test_leasequery.**create_client_id_query**(*options*) → dhcpkit.ipv6.extensions.leasequery.LQQueryOp
Create query option for client-id query.

Parameters **options** – Options from the main argument parser

Returns The Leasequery

dhcpkit.ipv6.client.test_leasequery.**create_link_address_query**(*options*) → dhcpkit.ipv6.extensions.leasequery.LQQue

Create query option for link-address query.

Parameters **options** – Options from the main argument parser

Returns The Leasequery

dhcpkit.ipv6.client.test_leasequery.**create_relay_id_query**(*options*) → dhcpkit.ipv6.extensions.leasequery.LQQueryOpti

Create query option for relay-id query.

Parameters **options** – Options from the main argument parser

Returns The Leasequery

dhcpkit.ipv6.client.test_leasequery.**create_remote_id_query**(*options*) → dhcpkit.ipv6.extensions.leasequery.LQQueryOp

Create query option for remote-id query.

Parameters **options** – Options from the main argument parser

Returns The Leasequery

dhcpkit.ipv6.client.test_leasequery.**handle_args**(*args: Iterable*)

Handle the command line arguments.

Parameters **args** – Command line arguments

Returns The arguments object

dhcpkit.ipv6.client.test_leasequery.**main**(*args: Iterable*) → int

The main program

Parameters **args** – Command line arguments

Returns The program exit code

dhcpkit.ipv6.client.test_leasequery.**parse_duid**(*duid_str: str*) → dhcpkit.ipv6.duids.DUID

Parse a string representing a DUID into a real DUID

Parameters **duid_str** – The string representation

Returns The DUID object

dhcpkit.ipv6.client.test_leasequery.**run**() → int

Run the main program and handle exceptions

Returns The program exit code

dhcpkit.ipv6.extensions package

Module containing extensions to the basic DHCPv6 RFC.

Submodules

dhcpkit.ipv6.extensions.bulk_leasequery module

Implementation of the Bulk Leasequery protocol extension as specified in **RFC 5460**[26].

[26] https://tools.ietf.org/html/rfc5460.html

class dhcpkit.ipv6.extensions.bulk_leasequery.**LeasequeryDataMessage**(*transaction_id: bytes = b'x00x00x00', options: Iterable = None*)

Bases: *dhcpkit.ipv6.messages.ClientServerMessage* (page 152)

The LEASEQUERY-DATA message carries data about a single DHCPv6 client's leases and/or PD bindings on a single link. The purpose of the message is to reduce redundant data when there are multiple bindings to be sent. The LEASEQUERY-DATA message MUST be preceded by a LEASEQUERY-REPLY message. The LEASEQUERY-REPLY carries the query's status, the Leasequery's Client-ID and Server-ID options, and the first client's binding data if the query was successful.

LEASEQUERY-DATA MUST ONLY be sent in response to a successful LEASEQUERY, and only if more than one client's data is to be sent. The LEASEQUERY-DATA message's transaction-id field MUST match the transaction-id of the LEASEQUERY request message. The Server-ID, Client-ID, and OPTION_STATUS_CODE options SHOULD NOT be included: that data should be constant for any one Bulk Leasequery reply, and should have been conveyed in the LEASEQUERY-REPLY message.

from_server_to_client = True

message_type = 17

class dhcpkit.ipv6.extensions.bulk_leasequery.**LeasequeryDoneMessage**(*transaction_id: bytes = b'x00x00x00', options: Iterable = None*)

Bases: *dhcpkit.ipv6.messages.ClientServerMessage* (page 152)

The LEASEQUERY-DONE message indicates the end of a group of related Leasequery replies. The LEASEQUERY-DONE message's transaction-id field MUST match the transaction-id of the LEASEQUERY request message. The presence of the message itself signals the end of a stream of reply messages. A single LEASEQUERY-DONE MUST BE sent after all replies (a successful LEASEQUERY-REPLY and zero or more LEASEQUERY-DATA messages) to a successful Bulk Leasequery request that returned at least one binding.

A server may encounter an error condition after it has sent the initial LEASEQUERY-REPLY. In that case, it SHOULD attempt to send a LEASEQUERY-DONE with an OPTION_STATUS_CODE option indicating the error condition to the requestor. Other DHCPv6 options SHOULD NOT be included in the LEASEQUERY-DONE message.

from_server_to_client = True

message_type = 16

class dhcpkit.ipv6.extensions.bulk_leasequery.**RelayIdOption**(*duid: dhcpkit.ipv6.duids.DUID = None*)

Bases: *dhcpkit.ipv6.options.Option* (page 168)

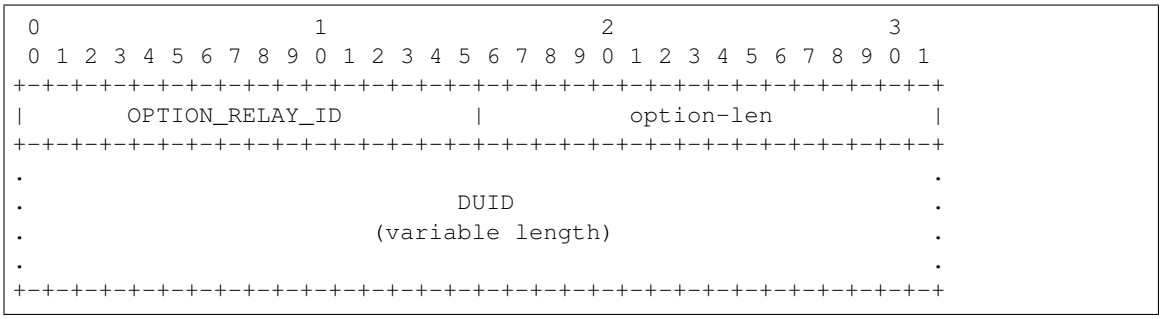
RFC 5460#section-5.4.1[27]

[27] https://tools.ietf.org/html/rfc5460.html#section-5.4.1

The Relay-ID option carries a DUID [RFC3315]. A relay agent MAY include the option in Relay-Forward messages it sends. Obviously, it will not be possible for a server to respond to QUERY_BY_RELAY_ID queries unless the relay agent has included this option. A relay SHOULD be able to generate a DUID for this purpose, and capture the result in stable storage. A relay SHOULD also allow the DUID value to be configurable: doing so allows an administrator to replace a relay agent while retaining the association between the relay and existing DHCPv6 bindings.

A DHCPv6 server MAY associate Relay-ID options from Relay-Forward messages it processes with prefix delegations and/or lease bindings that result. Doing so allows it to respond to QUERY_BY_RELAY_ID Leasequeries.

The format of the Relay-ID option is shown below:



option-code OPTION_RELAY_ID (53).

option-len Length of DUID in octets.

DUID The DUID for the relay agent.

duid = None
The DUID of the relay agent

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 53

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()
Validate that the contents of this object conform to protocol specs.

dhcpkit.ipv6.extensions.client_fqdn module

Implementation of the Client FQDN option as specified in **RFC 4704**[28].

[28] https://tools.ietf.org/html/rfc4704.html

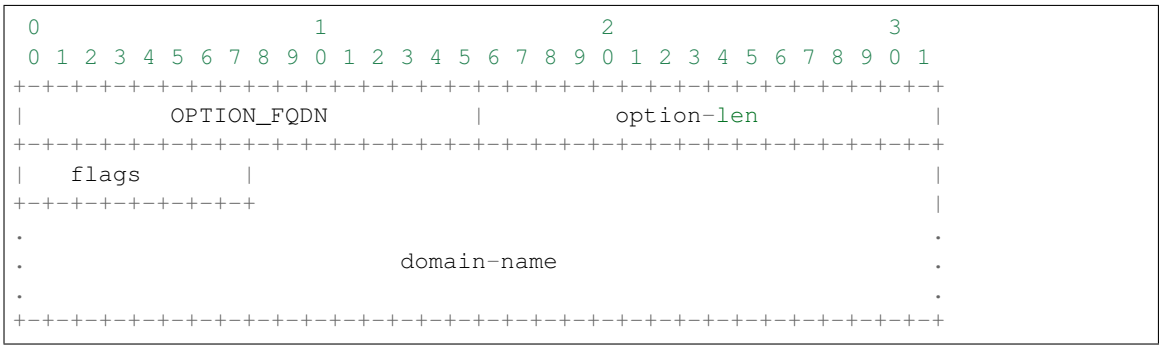
class dhcpkit.ipv6.extensions.client_fqdn.**ClientFQDNOption**(*flags: int = 0*, *domain_name: str = None*)

Bases: *dhcpkit.ipv6.options.Option* (page 168)

To update the IPv6-address-to-FQDN mapping, a DHCPv6 server needs to know the FQDN of the client for the addresses for the client's IA_NA bindings. To allow the client to convey its FQDN to the server, this document defines a new DHCPv6 option called "Client FQDN". The Client FQDN option also contains Flags that DHCPv6 clients and servers use to negotiate who does which updates.

The code for this option is 39. Its minimum length is 1 octet.

The format of the DHCPv6 Client FQDN option is shown below:



option-code OPTION_CLIENT_FQDN (39).

option-len 1 + length of domain name.

flags flag bits used between client and server to negotiate who performs which updates.

domain-name the partial or fully qualified domain name (with length option-len - 1).

The Client FQDN option MUST only appear in a message's options field and applies to all addresses for all IA_NA bindings in the transaction.

4.1. The Flags Field

The format of the Flags field is:

The "S" bit indicates whether the server SHOULD or SHOULD NOT perform the AAAA RR (FQDN-to-address) DNS updates. A client sets the bit to 0 to indicate that the server SHOULD NOT perform the updates and 1 to indicate that the server SHOULD perform the updates. The state of the bit in the reply from the server indicates the action to be taken by the server; if it is 1, the server has taken responsibility for AAAA RR updates for the FQDN.

The "O" bit indicates whether the server has overridden the client's preference for the "S" bit. A client MUST set this bit to 0. A server MUST set this bit to 1 if the "S" bit in its reply to the client does not match the "S" bit received from the client.

The "N" bit indicates whether the server SHOULD NOT perform any DNS updates. A client sets this bit to 0 to request that the server SHOULD perform updates (the PTR RR and possibly the AAAA RR based on the "S" bit) or to 1 to request that the server SHOULD NOT perform any DNS updates. A server sets the "N" bit to indicate whether the server SHALL (0) or SHALL NOT (1) perform DNS updates. If the "N" bit is 1, the "S" bit MUST be 0.

The remaining bits in the Flags field are reserved for future assignment. DHCPv6 clients and servers that send the Client FQDN option MUST clear the MBZ bits, and they MUST ignore these bits.

4.2. The Domain Name Field

The Domain Name part of the option carries all or part of the FQDN of a DHCPv6 client. The data in the Domain Name field MUST be encoded as described in Section 8 of [5]. In order to determine whether the FQDN has changed between message exchanges, the client and server MUST NOT alter the Domain Name field contents unless the FQDN has actually changed.

A client MAY be configured with a fully qualified domain name or with a partial name that is not fully qualified. If a client knows only part of its name, it MAY send a name that is not fully qualified, indicating that it knows part of the name but does not necessarily know the zone in which the name is to be embedded.

To send a fully qualified domain name, the Domain Name field is set to the DNS-encoded domain name including the terminating zero-length label. To send a partial name, the Domain Name field is set to the DNS-encoded domain name without the terminating zero-length label.

A client MAY also leave the Domain Name field empty if it desires the server to provide a name.

Servers SHOULD send the complete fully qualified domain name in Client FQDN options.

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

no_server_dns_update
Extract the N flag

Returns Whether the N flag is set

option_type = 39

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

server_aaaa_override
Extract the O flag

Returns Whether the O flag is set

server_aaaa_update
Extract the S flag

Returns Whether the S flag is set

validate()
Validate that the contents of this object conform to protocol specs.

dhcpkit.ipv6.extensions.dns module

Implementation of DNS options as specified in **RFC 3646**[29].

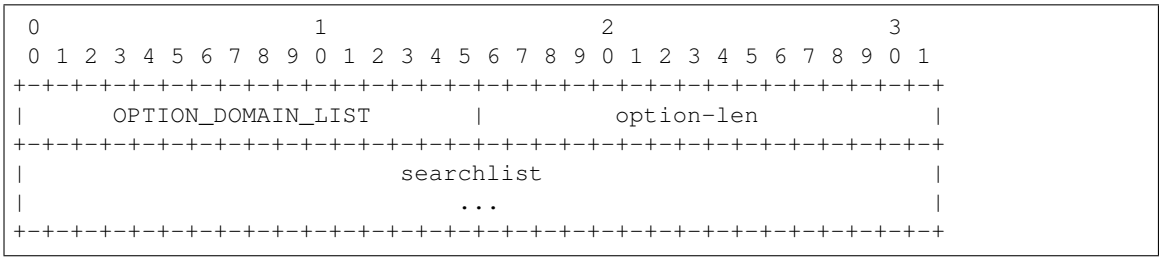
class dhcpkit.ipv6.extensions.dns.**DomainSearchListOption**(*search_list: Iterable = None*)
Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 3646#section-4[30]

[29] https://tools.ietf.org/html/rfc3646.html
[30] https://tools.ietf.org/html/rfc3646.html#section-4

The Domain Search List option specifies the domain search list the client is to use when resolving hostnames with DNS. This option does not apply to other name resolution mechanisms.

The format of the Domain Search List option is:



option-code OPTION_DOMAIN_LIST (24).

option-len Length of the 'searchlist' field in octets.

searchlist The specification of the list of domain names in the Domain Search List.

The list of domain names in the 'searchlist' MUST be encoded as specified in section "Representation and use of domain names" of **RFC 3315**[31].

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 24

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

search_list = None
List of domain names to use as a search list

validate()
Validate that the contents of this object conform to protocol specs.

class dhcpkit.ipv6.extensions.dns.**RecursiveNameServersOption**(*dns_servers: Iterable = None*)

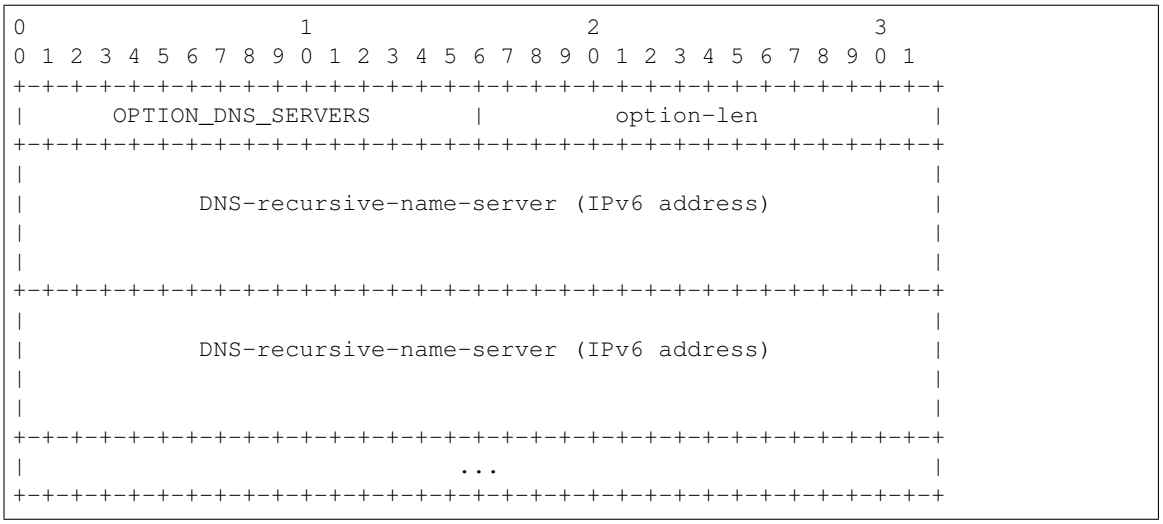
Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 3646#section-3[32]

The DNS Recursive Name Server option provides a list of one or more IPv6 addresses of DNS recursive name servers to which a client's DNS resolver MAY send DNS queries [1]. The DNS servers are listed in the order of preference for use by the client resolver.

The format of the DNS Recursive Name Server option is:

[31] https://tools.ietf.org/html/rfc3315.html
[32] https://tools.ietf.org/html/rfc3646.html#section-3



option-code OPTION_DNS_SERVERS (23).

option-len Length of the list of DNS recursive name servers in octets; must be a multiple of 16.

DNS-recursive-name-server IPv6 address of DNS recursive name server.

dns_servers = None
List of IPv6 addresses of resolving DNS servers

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 23

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()
Validate that the contents of this object conform to protocol specs.

dhcpkit.ipv6.extensions.dslite module

Implementation of DS-Lite AFTR Name option as specified in **RFC 6334**[33].

class dhcpkit.ipv6.extensions.dslite.**AFTRNameOption**(*fqdn: str = ''*)
Bases: *dhcpkit.ipv6.options.Option* (page 168)

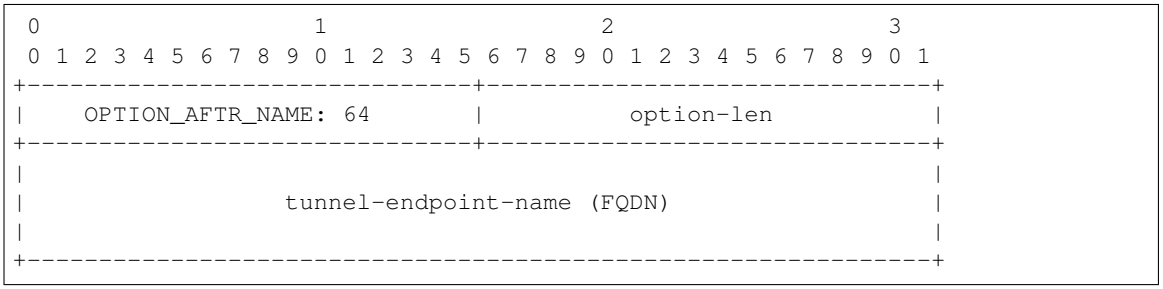
RFC 6334#section-3[34]

[33] https://tools.ietf.org/html/rfc6334.html
[34] https://tools.ietf.org/html/rfc6334.html#section-3

The AFTR-Name option consists of option-code and option-len fields (as all DHCPv6 options have), and a variable-length tunnel-endpoint- name field containing a fully qualified domain name that refers to the AFTR to which the client MAY connect.

The AFTR-Name option SHOULD NOT appear in any DHCPv6 messages other than the following: Solicit, Advertise, Request, Renew, Rebind, Information-Request, and Reply.

The format of the AFTR-Name option is shown in the following figure:



OPTION_AFTR_NAME 64

option-len Length of the tunnel-endpoint-name field in octets.

tunnel-endpoint-name A fully qualified domain name of the AFTR tunnel endpoint.

fqdn = None
Domain name of the AFTR tunnel endpoint

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 64

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()
Validate that the contents of this object conform to protocol specs.

dhcpkit.ipv6.extensions.leasequery module

Implementation of the Leasequery protocol extension as specified in **RFC 5007**[35].

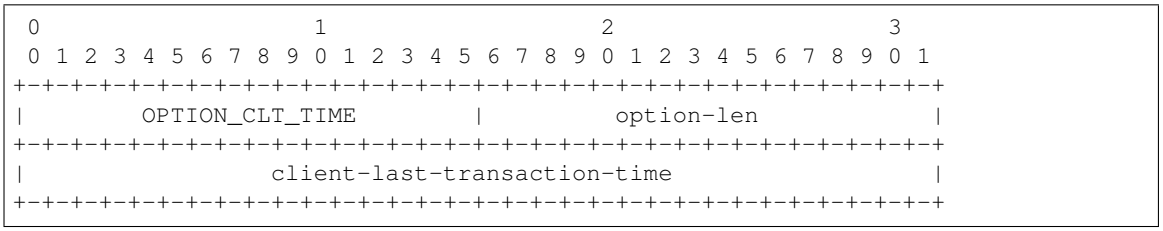
class dhcpkit.ipv6.extensions.leasequery.**CLTTimeOption**(*clt_time: int = 0*)
Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 5007#section-4.1.2.3[36]

[35] https://tools.ietf.org/html/rfc5007.html
[36] https://tools.ietf.org/html/rfc5007.html#section-4.1.2.3

The Client Last Transaction Time option is encapsulated in an OPTION_CLIENT_DATA and identifies how long ago the server last communicated with the client, in seconds.

The format of the Client Last Transaction Time option is shown below:



option-code OPTION_CLT_TIME (46)

option-len 4

client-last-transaction-time The number of seconds since the server last communicated with the client (on that link).

The client-last-transaction-time is a positive value and reflects the number of seconds since the server last communicated with the client (on that link).

clt_time = None
The number of seconds since the server last communicated with the client

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 46

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()
Validate that the contents of this object conform to protocol specs.

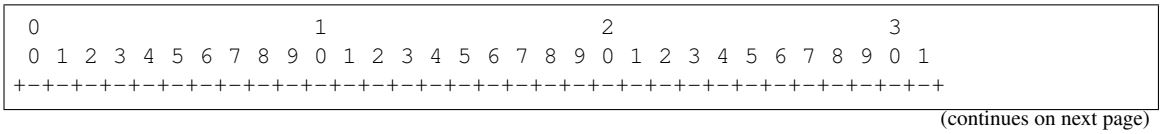
class dhcpkit.ipv6.extensions.leasequery.**ClientDataOption**(*options: Iterable = None*)

Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 5007#section-4.1.2.2[37]

The Client Data option is used to encapsulate the data for a single client on a single link in a LEASEQUERY-REPLY message.

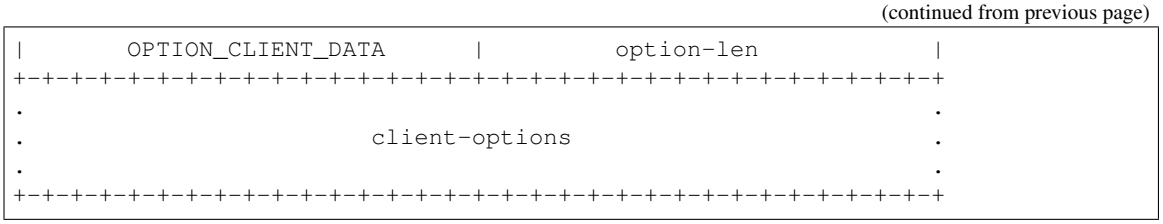
The format of the Client Data option is shown below:



(continues on next page)

[37] https://tools.ietf.org/html/rfc5007.html#section-4.1.2.2

(continued from previous page)



option-code OPTION_CLIENT_DATA (45)

option-len Length, in octets, of the encapsulated client-options field.

client-options The options associated with this client.

The encapsulated client-options include the OPTION_CLIENTID, OPTION_IAADDR, OPTION_IAPREFIX, and OPTION_CLT_TIME options and other options specific to the client and requested by the requestor in the OPTION_ORO in the OPTION_LQ_QUERY's query-options. The server MUST return all of the client's statefully assigned addresses and delegated prefixes, with a non-zero valid lifetime, on the link.

get_option_of_type(**args*) → Union
Get the first option that is a subclass of the given class.

Parameters **args** – The classes to look for

Returns The option or None

get_options_of_type(**args*) → List
Get all options that are subclasses of the given class.

Parameters **args** – The classes to look for

Returns The list of options

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 45

options = None
The options associated with this client

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()
Validate that the contents of this object conform to protocol specs.

class dhcpkit.ipv6.extensions.leasequery.**LQClientLink**(*link_addresses: Iterable = None*)

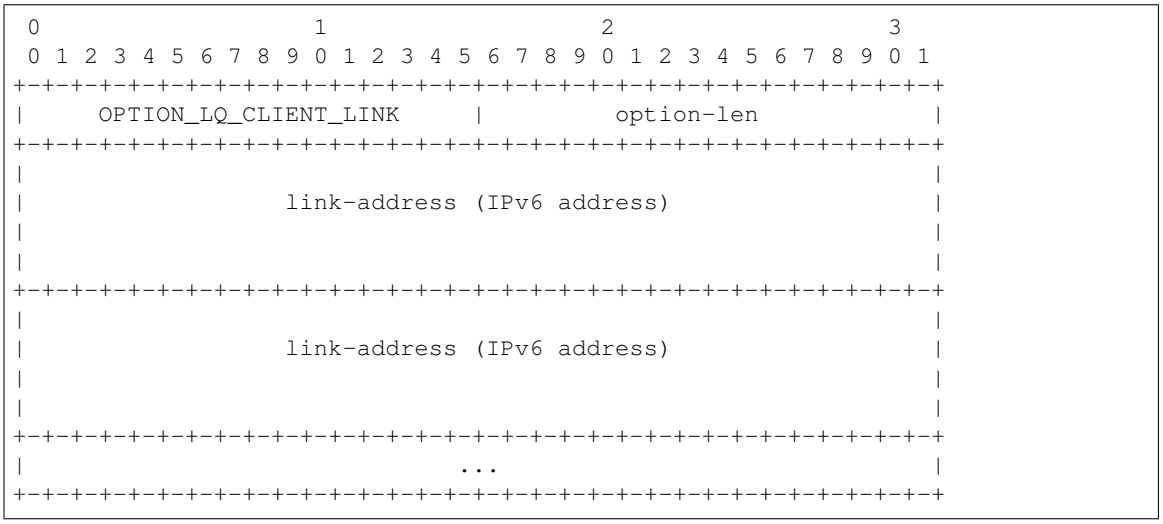
Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 5007#section-4.1.2.5[38]

[38] https://tools.ietf.org/html/rfc5007.html#section-4.1.2.5

The Client Link option is used only in a LEASEQUERY-REPLY message and identifies the links on which the client has one or more bindings. It is used in reply to a query when no link-address was specified and the client is found to be on more than one link.

The format of the Client Link option is shown below:



option-code OPTION_LQ_CLIENT_LINK (48)

option-len Length of the list of links in octets; must be a multiple of 16.

link-address A global address used by the server to identify the link on which the client is located.

A server may respond to a query by client-id, where the 0::0 link- address was specified, with this option if the client is found to be on multiple links. The requestor may then repeat the query once for each link-address returned in the list, specifying the returned link- address. If the client is on a single link, the server SHOULD return the client's data in an OPTION_CLIENT_DATA option.

link_addresses = None
Global addresses used by the server to identify the link on which the client is located

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 48

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()
Validate that the contents of this object conform to protocol specs.

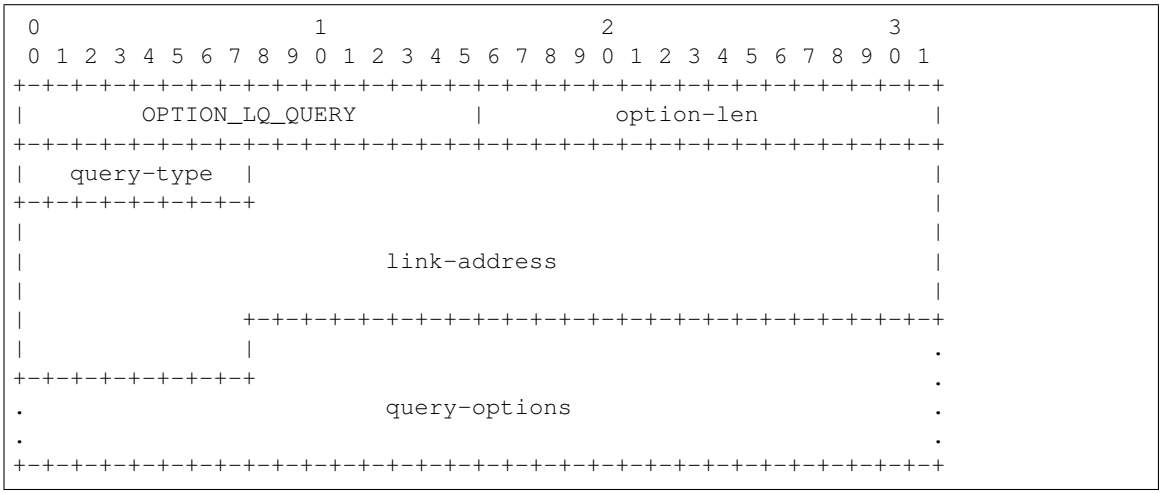
class dhcpkit.ipv6.extensions.leasequery.**LQQueryOption**(*query_type: int = 0*, *link_address: ipaddress.IPv6Address = None*, *options: Iterable = None*)

Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 5007#section-4.1.2.1[39]

The Query option is used only in a LEASEQUERY message and identifies the query being performed. The option includes the query type, link- address (or 0::0), and option(s) to provide data needed for the query.

The format of the Query option is shown below:



option-code OPTION_LQ_QUERY (44)

option-len 17 + length of query-options field.

link-address A global address that will be used by the server to identify the link to which the query applies, or 0::0 if unspecified.

query-type The query requested (see below).

query-options The options related to the query.

The query-type and required query-options are:

QUERY_BY_ADDRESS (1) The query-options MUST contain an OPTION_IAADDR option [2]. The link-address field, if not 0::0, specifies an address for the link on which the client is located if the address in the OPTION_IAADDR option is of insufficient scope. Only the information for the client that has a lease for the specified address or was delegated a prefix that contains the specified address is returned (if available).

QUERY_BY_CLIENTID (2) The query-options MUST contain an OPTION_CLIENTID option [2]. The link-address field, if not 0::0, specifies an address for the link on which the client is located. If the link-address field is 0::0, the server SHOULD search all of its links for the client.

The query-options MAY also include an OPTION_ORO option [2] to indicate the options for each client that the requestor would like the server to return. Note that this OPTION_ORO is distinct and separate from an OPTION_ORO that may be in the requestor's LEASEQUERY message.

If a server receives an OPTION_LQ_QUERY with a query-type it does not support, the server SHOULD return an UnknownQueryType status-code. If a server receives a supported query-type but the query-options is missing a required option, the server SHOULD return a MalformedQuery status-code.

This checking of mandatory options is done in the server code, not in *validate()* (page 57).

[39] https://tools.ietf.org/html/rfc5007.html#section-4.1.2.1

display_query_type() → dhcpkit.protocol_element.ElementDataRepresentation
Nicer representation of query types :return: Representation of query type

get_option_of_type(**args*) → Union
Get the first option that is a subclass of the given class.

Parameters **args** – The classes to look for

Returns The option or None

get_options_of_type(**args*) → List
Get all options that are subclasses of the given class.

Parameters **args** – The classes to look for

Returns The list of options

link_address = None
A global address that will be used by the server to identify the link to which the query applies

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 44

options = None
The options related to the query

query_type = None
The query requested

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()
Validate that the contents of this object conform to protocol specs.

class dhcpkit.ipv6.extensions.leasequery.**LQRelayDataOption**(*peer_address: ipaddress.IPv6Address = None*, *relay_message: dhcpkit.ipv6.messages.RelayForwardMessage = None*)

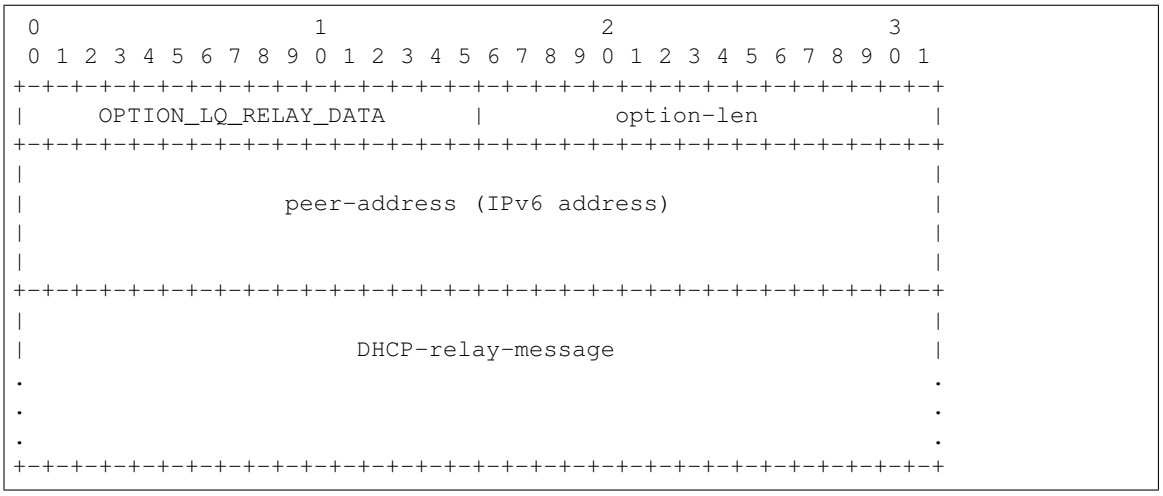
Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 5007#section-4.1.2.4[40]

The Relay Data option is used only in a LEASEQUERY-REPLY message and provides the relay agent information used when the client last communicated with the server.

The format of the Relay Data option is shown below:

[40] https://tools.ietf.org/html/rfc5007.html#section-4.1.2.4



option-code OPTION_LQ_RELAY_DATA (47)

option-len 16 + length of DHCP-relay-message.

peer-address The address of the relay agent from which the relayed message was received by the server.

DHCP-relay-message The last complete relayed message, excluding the client's message OPTION_RELAY_MSG, received by the server.

This option is used by the server to return full relay agent information for a client. It MUST NOT be returned if the server does not have such information, either because the client communicated directly (without relay agent) with the server or if the server did not retain such information.

If returned, the DHCP-relay-message MUST contain a valid (perhaps multi-hop) RELAY-FORW message as the most recently received by the server for the client. However, the (innermost) OPTION_RELAY_MSG option containing the client's message MUST have been removed.

This option SHOULD only be returned if requested by the OPTION_ORO of the OPTION_LQ_QUERY.

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 47

peer_address = None
The address of the relay agent from which the relayed message was received by the server.

relay_message = None
The options related to the query

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()
Validate that the contents of this object conform to protocol specs.

class dhcpkit.ipv6.extensions.leasequery.**LeasequeryMessage**(*transaction_id: bytes = b'x00x00x00', options: Iterable = None*)

Bases: *dhcpkit.ipv6.messages.ClientServerMessage* (page 152)

The LEASEQUERY and LEASEQUERY-REPLY messages use the Client/Server message formats. A requestor sends a LEASEQUERY message to any available server to obtain information on a client's leases. The options in an OPTION_LQ_QUERY determine the query.

from_client_to_server = True

message_type = 14

class dhcpkit.ipv6.extensions.leasequery.**LeasequeryReplyMessage**(*transaction_id: bytes = b'x00x00x00', options: Iterable = None*)

Bases: *dhcpkit.ipv6.messages.ClientServerMessage* (page 152)

The LEASEQUERY and LEASEQUERY-REPLY messages use the Client/Server message formats. A server sends a LEASEQUERY-REPLY message containing client data in response to a LEASEQUERY message.

from_server_to_client = True

message_type = 15

dhcpkit.ipv6.extensions.linklayer_id module

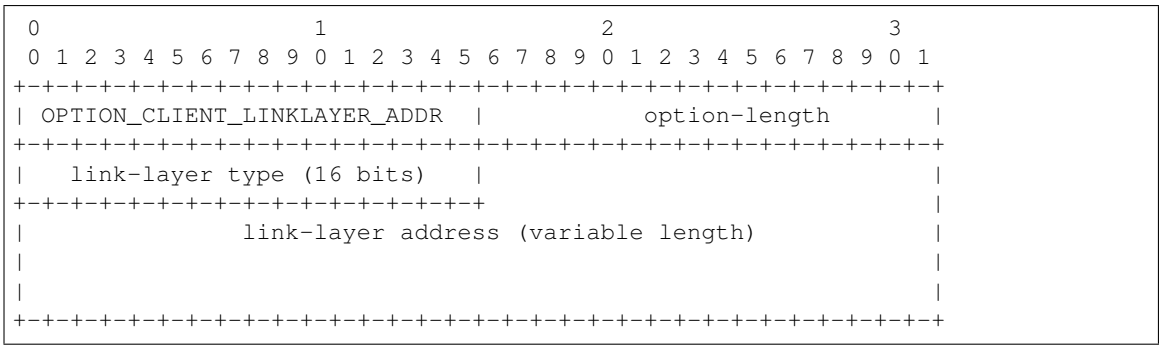
Implementation of the Client LinkLayer Address relay option as specified in **RFC 6939**[41].

class dhcpkit.ipv6.extensions.linklayer_id.**LinkLayerIdOption**(*link_layer_type: int = 0, link_layer_address: bytes = b''*)

Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 6939#section-4[42]

The format of the DHCPv6 Client Link-Layer Address option is shown below.



option-code OPTION_CLIENT_LINKLAYER_ADDR (79)

option-length 2 + length of link-layer address

[41] https://tools.ietf.org/html/rfc6939.html
[42] https://tools.ietf.org/html/rfc6939.html#section-4

link-layer type Client link-layer address type. The link-layer type MUST be a valid hardware type assigned by the IANA, as described in **RFC 826**[43]

link-layer address Client link-layer address

display_link_layer_type() → dhcpkit.protocol_element.ElementDataRepresentation
Nicer representation of hardware types :return: Representation of hardware type

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 79

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()
Validate that the contents of this object conform to protocol specs.

dhcpkit.ipv6.extensions.map module

Implementation of MAP options as specified in **RFC 7598**[44].

class dhcpkit.ipv6.extensions.map.**S46BROption**(*br_address: ipaddress.IPv6Address = None*)
Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 7598#section-4.2[45]

The S46 BR option (OPTION_S46_BR) is used to convey the IPv6 address of the Border Relay. Figure 3 shows the format of the OPTION_S46_BR option.

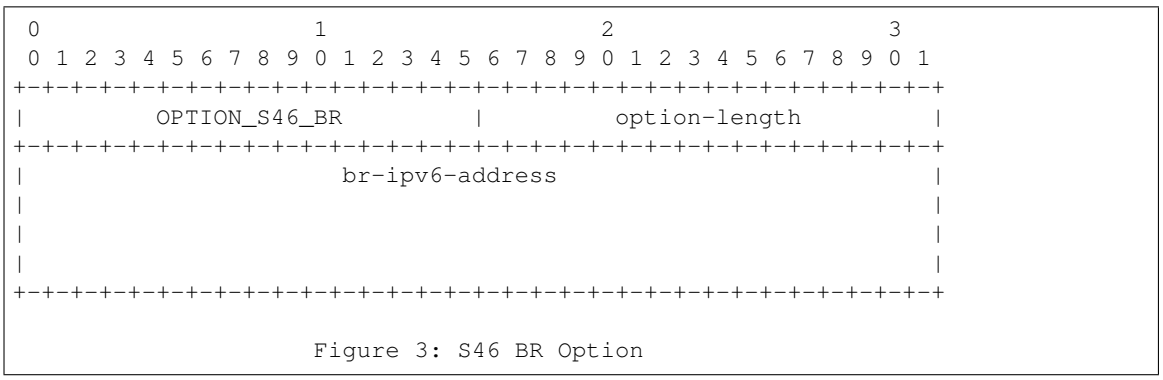


Figure 3: S46 BR Option

option-code OPTION_S46_BR (90)

option-length 16

br-ipv6-address a fixed-length field of 16 octets that specifies the IPv6 address for the S46 BR.

[43] https://tools.ietf.org/html/rfc826.html
[44] https://tools.ietf.org/html/rfc7598.html
[45] https://tools.ietf.org/html/rfc7598.html#section-4.2

BR redundancy can be implemented by using an anycast address for the BR IPv6 address. Multiple OPTION_S46_BR options MAY be included in the container; this document does not further explore the use of multiple BR IPv6 addresses.

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 90

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()
Validate that the contents of this object conform to protocol specs.

class dhcpkit.ipv6.extensions.map.**S46ContainerOption**(*options: Iterable = None*)
Bases: *dhcpkit.ipv6.options.Option* (page 168)

Common code for MAP-E, MAP-T and LW4over6 containers

get_option_of_type(**args*) → Union
Get the first option that is a subclass of the given class.

Parameters **args** – The classes to look for

Returns The option or None

get_options_of_type(**args*) → List
Get all options that are subclasses of the given class.

Parameters **args** – The classes to look for

Returns The list of options

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 0

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()
Validate that the contents of this object conform to protocol specs.

class dhcpkit.ipv6.extensions.map.**S46DMROption**(*dmr_prefix: ipaddress.IPv6Network = None*)

Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 7598#section-4.3[46]

The S46 DMR option (OPTION_S46_DMR) is used to convey values for the Default Mapping Rule (DMR). Figure 4 shows the format of the OPTION_S46_DMR option used for conveying a DMR.

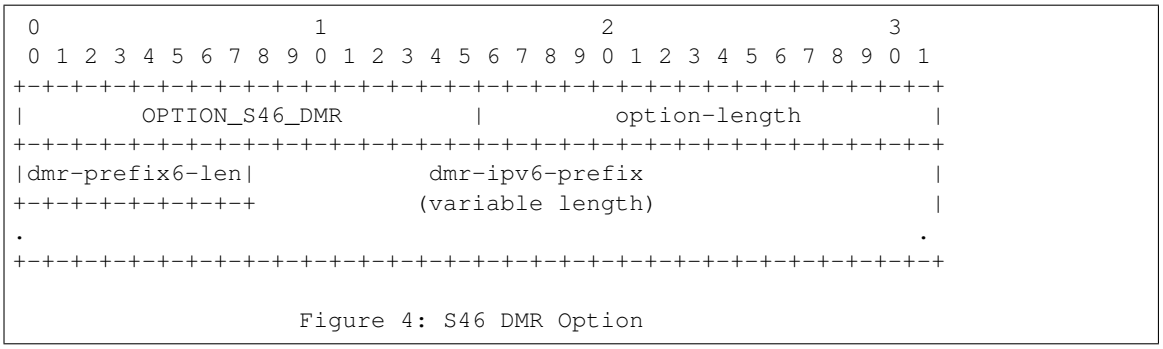


Figure 4: S46 DMR Option

option-code OPTION_S46_DMR (91)

option-length 1 + length of dmr-ipv6-prefix specified in octets.

dmr-prefix6-len 8 bits long; expresses the bitmask length of the IPv6 prefix specified in the dmr-ipv6-prefix field. Allowed values range from 0 to 128.

dmr-ipv6-prefix a variable-length field specifying the IPv6 prefix or address for the BR. This field is right-padded with zeros to the nearest octet boundary when dmr-prefix6-len is not divisible by 8.

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int

Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 91

save() → Union

Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()

Validate that the contents of this object conform to protocol specs.

class dhcpkit.ipv6.extensions.map.**S46LWContainerOption**(*options: Iterable = None*)

Bases: *dhcpkit.ipv6.extensions.map.S46ContainerOption* (page 61)

RFC 7598#section-5.3[47]

The S46 Lightweight 4over6 Container option (OPTION_S46_CONT_LW) specifies the container used to group all rules and optional port parameters for a specified domain.

[46] https://tools.ietf.org/html/rfc7598.html#section-4.3
[47] https://tools.ietf.org/html/rfc7598.html#section-5.3

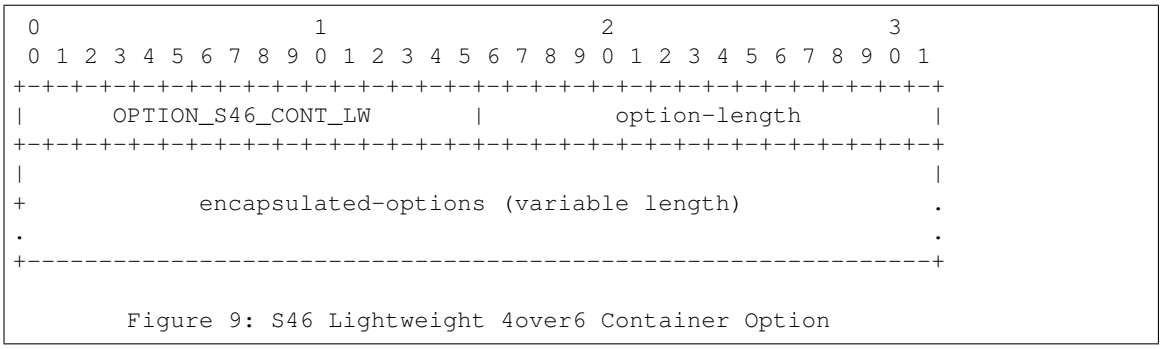


Figure 9: S46 Lightweight 4over6 Container Option

option-code OPTION_S46_CONT_LW (96)

option-length length of encapsulated options, expressed in octets.

encapsulated-options options associated with this Softwire46 Lightweight 4over6 domain.

The encapsulated-options field conveys options specific to the OPTION_S46_CONT_LW option. Currently, there are two options specified: OPTION_S46_V4V6BIND and OPTION_S46_BR. There MUST be at most one OPTION_S46_V4V6BIND option and at least one OPTION_S46_BR option.

option_type = 96

class dhcpkit.ipv6.extensions.map.**S46MapEContainerOption**(*options: Iterable = None*)

Bases: *dhcpkit.ipv6.extensions.map.S46ContainerOption* (page 61)

RFC 7598#section-5.1[48]

The S46 MAP-E Container option (OPTION_S46_CONT_MAPE) specifies the container used to group all rules and optional port parameters for a specified domain.

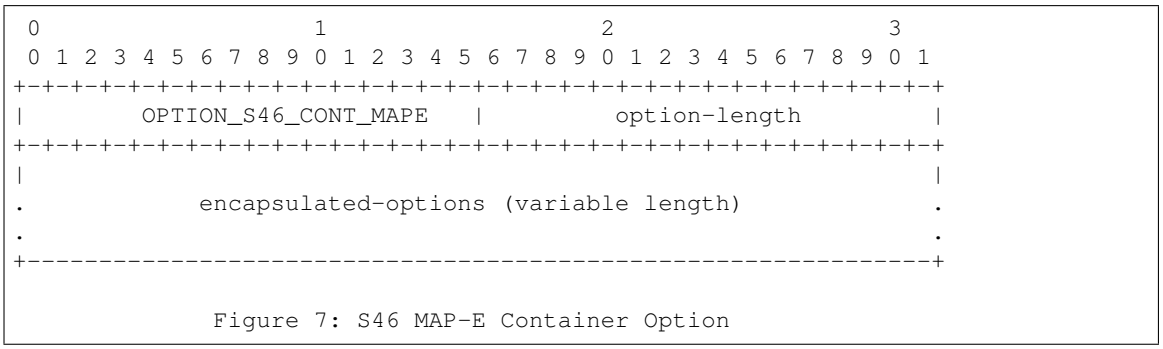


Figure 7: S46 MAP-E Container Option

option-code OPTION_S46_CONT_MAPE (94)

option-length length of encapsulated options, expressed in octets.

encapsulated-options options associated with this Softwire46 MAP-E domain.

The encapsulated-options field conveys options specific to the OPTION_S46_CONT_MAPE option. Currently, there are two encapsulated options specified: OPTION_S46_RULE and OPTION_S46_BR. There MUST be at least one OPTION_S46_RULE option and at least one OPTION_S46_BR option.

Other options applicable to a domain may be defined in the future. A DHCPv6 message MAY include multiple OPTION_S46_CONT_MAPE options (representing multiple domains).

option_type = 94

class dhcpkit.ipv6.extensions.map.**S46MapTContainerOption**(*options: Iterable = None*)

Bases: *dhcpkit.ipv6.extensions.map.S46ContainerOption* (page 61)

[48] https://tools.ietf.org/html/rfc7598.html#section-5.1

RFC 7598#section-5.2[49]

The S46 MAP-T Container option (OPTION_S46_CONT_MAPT) specifies the container used to group all rules and optional port parameters for a specified domain.

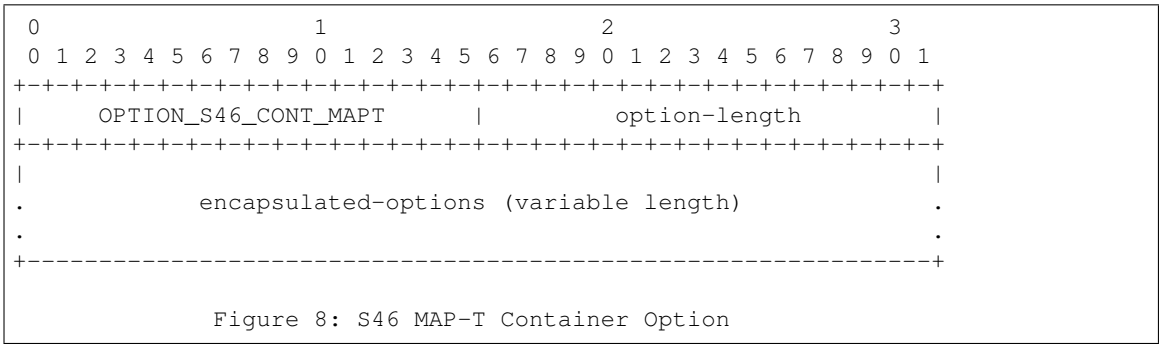


Figure 8: S46 MAP-T Container Option

option-code OPTION_S46_CONT_MAPT (95)

option-length length of encapsulated options, expressed in octets.

encapsulated-options options associated with this Softwire46 MAP-T domain.

The encapsulated-options field conveys options specific to the OPTION_S46_CONT_MAPT option. Currently, there are two options specified: the OPTION_S46_RULE and OPTION_S46_DMR options. There MUST be at least one OPTION_S46_RULE option and exactly one OPTION_S46_DMR option.

option_type = 95

class dhcpkit.ipv6.extensions.map.**S46PortParametersOption**(*offset: int = 0*, *psid_len: int = 0*, *psid: int = 0*)

Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 7598#section-4.5[50]

The S46 Port Parameters option (OPTION_S46_PORTPARAMS) specifies optional port set information that MAY be provided to CEs.

See Section 5.1 of [RFC7597] for a description of the MAP algorithm and detailed explanation of all of the parameters.

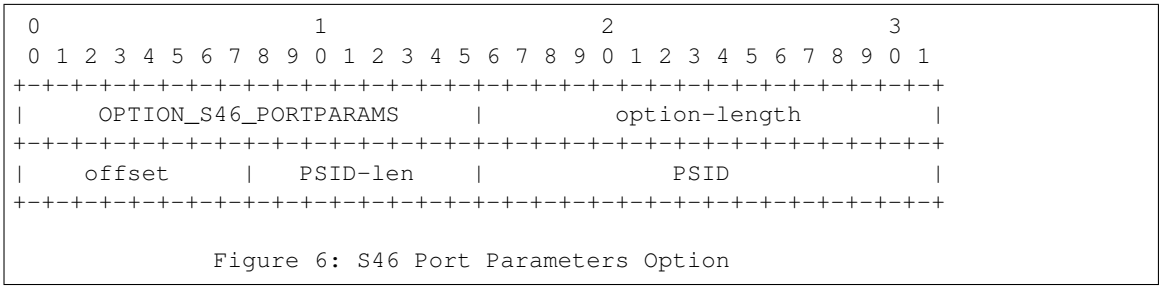


Figure 6: S46 Port Parameters Option

option-code OPTION_S46_PORTPARAMS (93)

option-length 4

offset Port Set Identifier (PSID) offset. 8 bits long; specifies the numeric value for the S46 algorithm's excluded port range/offset bits (a-bits), as per Section 5.1 of [RFC7597]. Allowed values are between 0 and 15. Default values for this field are specific to the softwire mechanism being implemented and are defined in the relevant specification document.

[49] https://tools.ietf.org/html/rfc7598.html#section-5.2
[50] https://tools.ietf.org/html/rfc7598.html#section-4.5

PSID-len 8 bits long; specifies the number of significant bits in the PSID field (also known as 'k'). When set to 0, the PSID field is to be ignored. After the first 'a' bits, there are k bits in the port number representing the value of the PSID. Consequently, the address-sharing ratio would be 2^k.

PSID 16 bits long. The PSID value algorithmically identifies a set of ports assigned to a CE. The first k bits on the left of this field contain the PSID binary value. The remaining (16 - k) bits on the right are padding zeros.

When receiving the OPTION_S46_PORTPARAMS option with an explicit PSID, the client MUST use this explicit PSID when configuring its softwire interface. The OPTION_S46_PORTPARAMS option with an explicit PSID MUST be discarded if the S46 CE isn't configured with a full IPv4 address (e.g., IPv4 prefix).

The OPTION_S46_PORTPARAMS option is contained within an OPTION_S46_RULE option or an OPTION_S46_V4V6BIND option.

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 93

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()
Validate that the contents of this object conform to protocol specs.

class dhcpkit.ipv6.extensions.map.**S46RuleOption**(*flags: int = 0*, *ea_len: int = 0*, *ipv4_prefix: ipaddress.IPv4Network = None*, *ipv6_prefix: ipaddress.IPv6Network = None*, *options: Iterable = None*)

Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 7598#section-4.1[51]

Figure 1 shows the format of the S46 Rule option (OPTION_S46_RULE) used for conveying the Basic Mapping Rule (BMR) and Forwarding Mapping Rule (FMR).

This option follows behavior described in Sections 17.1.1 and 18.1.1 of [RFC3315]. Clients can send those options, encapsulated in their respective container options, with specific values as hints for the server. See Section 5 for details. Depending on the server configuration and policy, it may accept or ignore the hints. Clients MUST be able to process received values that are different than the hints it sent earlier.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        OPTION_S46_RULE        |          option-length          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     flags     |     ea-len    |  prefix4-len  | ipv4-prefix   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

(continues on next page)

[51] https://tools.ietf.org/html/rfc7598.html#section-4.1

(continued from previous page)

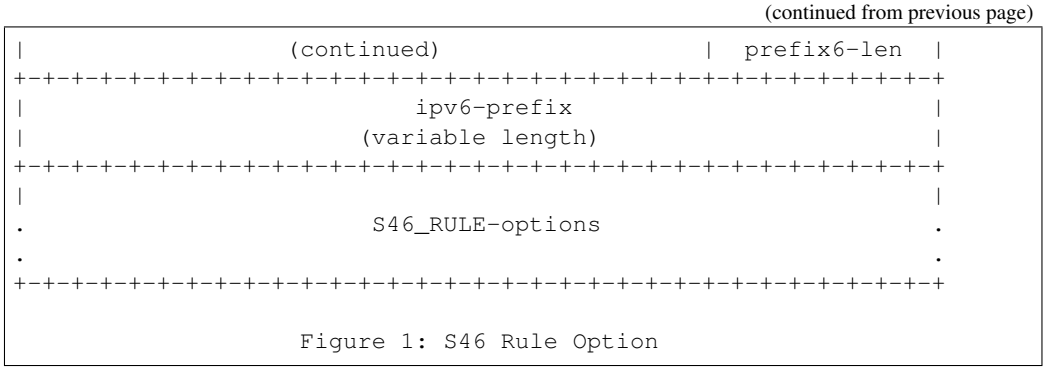


Figure 1: S46 Rule Option

option-code OPTION_S46_RULE (89)

option-length length of the option, excluding option-code and option-length fields, including length of all encapsulated options; expressed in octets.

flags 8 bits long; carries flags applicable to the rule. The meanings of the specific bits are explained in Figure 2.

ea-len 8 bits long; specifies the Embedded Address (EA) bit length. Allowed values range from 0 to 48.

prefix4-len 8 bits long; expresses the prefix length of the Rule IPv4 prefix specified in the ipv4-prefix field. Allowed values range from 0 to 32.

ipv4-prefix a fixed-length 32-bit field that specifies the IPv4 prefix for the S46 rule. The bits in the prefix after prefix4-len number of bits are reserved and MUST be initialized to zero by the sender and ignored by the receiver.

prefix6-len 8 bits long; expresses the length of the Rule IPv6 prefix specified in the ipv6-prefix field. Allowed values range from 0 to 128.

ipv6-prefix a variable-length field that specifies the IPv6 domain prefix for the S46 rule. The field is padded on the right with zero bits up to the nearest octet boundary when prefix6-len is not evenly divisible by 8.

S46_RULE-options a variable-length field that may contain zero or more options that specify additional parameters for this S46 rule. This document specifies one such option: OPTION_S46_PORTPARAMS.

The format of the S46 Rule Flags field is:

Figure 2: S46 Rule Flags

Reserved 7 bits; reserved for future use as flags.

F-flag 1-bit field that specifies whether the rule is to be used for forwarding (FMR). If set, this rule is used as an FMR; if not set, this rule is a BMR only and MUST NOT be used for forwarding.

Note: A BMR can also be used as an FMR for forwarding if the F-flag is set. The BMR is determined by a longest-prefix match of the Rule IPv6 prefix against the End-user IPv6 prefix(es).

It is expected that in a typical mesh deployment scenario there will be a single BMR, which could also be designated as an FMR using the F-flag.

fmr
Extract the F flag

Returns Whether the F flag is set

get_option_of_type(**args*) → Union
Get the first option that is a subclass of the given class.

Parameters **args** – The classes to look for

Returns The option or None

get_options_of_type(**args*) → List
Get all options that are subclasses of the given class.

Parameters **args** – The classes to look for

Returns The list of options

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 89

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()
Validate that the contents of this object conform to protocol specs.

class dhcpkit.ipv6.extensions.map.**S46V4V6BindingOption**(*ipv4_address: ipaddress.IPv4Address = None*, *ipv6_prefix: ipaddress.IPv6Network = None*, *options: Iterable = None*)

Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 7598#section-4.4[52]

The S46 IPv4/IPv6 Address Binding option (OPTION_S46_V4V6BIND) MAY be used to specify the full or shared IPv4 address of the CE. The IPv6 prefix field is used by the CE to identify the correct prefix to use for the tunnel source.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      OPTION_S46_V4V6BIND      |          option-length          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          ipv4-address                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|bindprefix6-len|             bind-ipv6-prefix                   |
+-+-+-+-+-+-+-+-+             (variable length)                 |
.                                                               .
```

(continues on next page)

[52] https://tools.ietf.org/html/rfc7598.html#section-4.4

(continued from previous page)

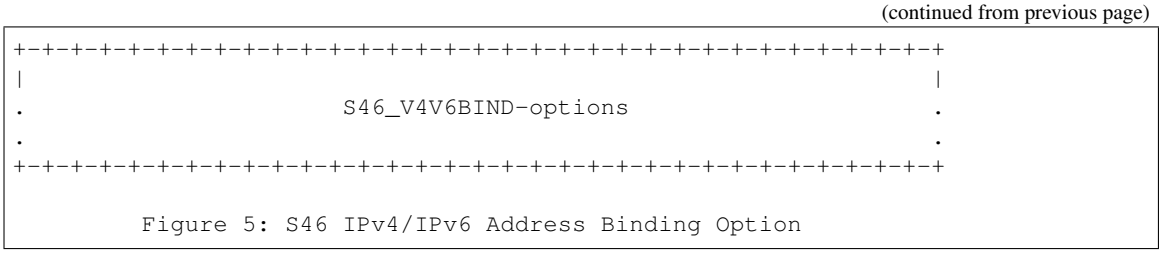


Figure 5: S46 IPv4/IPv6 Address Binding Option

option-code OPTION_S46_V4V6BIND (92)

option-length length of the option, excluding option-code and option-length fields, including length of all encapsulated options; expressed in octets.

ipv4-address a fixed-length field of 4 octets specifying an IPv4 address.

bindprefix6-len 8 bits long; expresses the bitmask length of the IPv6 prefix specified in the bind-ipv6-prefix field. Allowed values range from 0 to 128.

bind-ipv6-prefix a variable-length field specifying the IPv6 prefix or address for the S46 CE. This field is right-padded with zeros to the nearest octet boundary when bindprefix6-len is not divisible by 8.

S46_V4V6BIND-options a variable-length field that may contain zero or more options that specify additional parameters. This document specifies one such option: OPTION_S46_PORTPARAMS.

get_option_of_type(**args*) → Union
Get the first option that is a subclass of the given class.

Parameters **args** – The classes to look for

Returns The option or None

get_options_of_type(**args*) → List
Get all options that are subclasses of the given class.

Parameters **args** – The classes to look for

Returns The list of options

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 92

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()
Validate that the contents of this object conform to protocol specs.

dhcpkit.ipv6.extensions.ntp module

Implementation of NTP options as specified in **RFC 5908**[53].

[53] https://tools.ietf.org/html/rfc5908.html

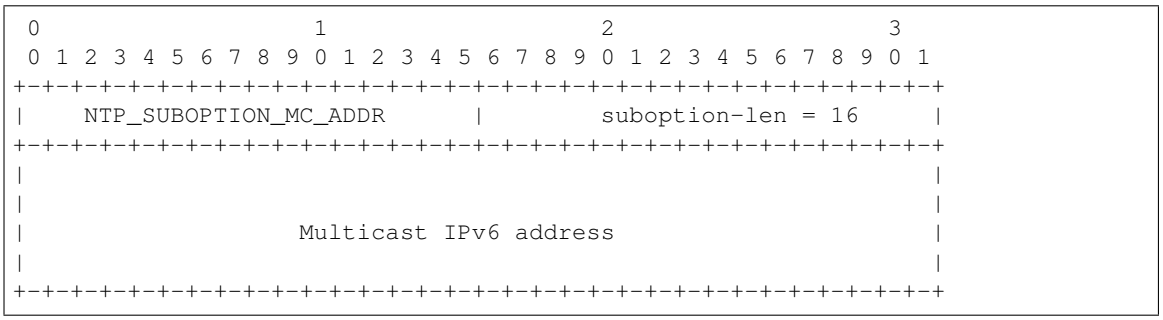
class dhcpkit.ipv6.extensions.ntp.**NTPMulticastAddressSubOption**(*address: ipaddress.IPv6Address = None*)

Bases: *dhcpkit.ipv6.extensions.ntp.NTPSubOption* (page 73)

RFC 5908#section-4.2[54]

This suboption is intended to appear inside the OPTION_NTP_SERVER option. It specifies the IPv6 address of the IPv6 multicast group address used by NTP on the local network.

The format of the NTP Multicast Address Suboption is:



Multicast IPv6 address An IPv6 address.

suboption-code NTP_SUBOPTION_MC_ADDR (2).

suboption-len

16.

address = None
IPv6 multicast group address

static config_datatype(*value: str*) → ipaddress.IPv6Address
Convert string data from the configuration to an IPv6address.

Parameters **value** – String from config file

Returns Parsed IPv6 address

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

suboption_type = 2

validate()
Validate that the contents of this object conform to protocol specs.

[54] https://tools.ietf.org/html/rfc5908.html#section-4.2

value
Return a simple string representation of the value of this sub-option.

Returns The value of this option as a string

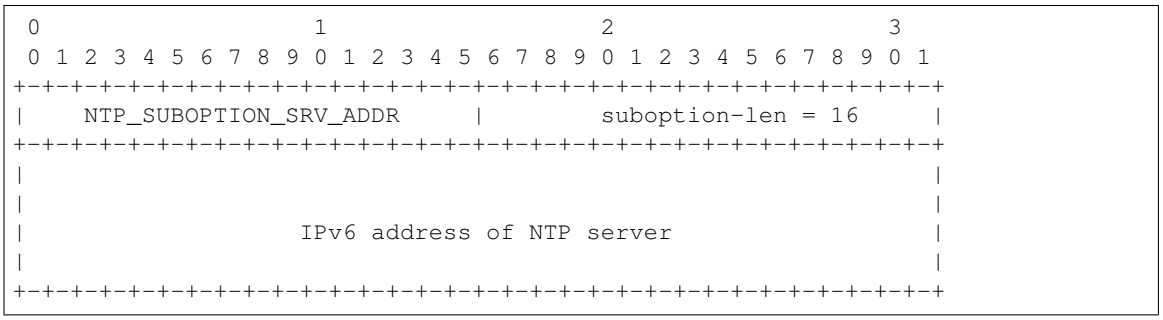
class dhcpkit.ipv6.extensions.ntp.**NTPServerAddressSubOption**(*address: ipaddress.IPv6Address = None*)

Bases: *dhcpkit.ipv6.extensions.ntp.NTPSubOption* (page 73)

RFC 5908#section-4.1[55]

This suboption is intended to appear inside the OPTION_NTP_SERVER option. It specifies the IPv6 unicast address of an NTP server or SNTP server available to the client.

The format of the NTP Server Address Suboption is:



IPv6 address of the NTP server An IPv6 address.

suboption-code NTP_SUBOPTION_SRV_ADDR (1).

suboption-len

16.

address = None
IPv6 address of an NTP server

static config_datatype(*value: str*) → ipaddress.IPv6Address
Convert string data from the configuration to an IPv6address.

Parameters **value** – String from config file

Returns Parsed IPv6 address

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

[55] https://tools.ietf.org/html/rfc5908.html#section-4.1

suboption_type = 1

validate()
Validate that the contents of this object conform to protocol specs.

value
Return a simple string representation of the value of this sub-option.

Returns The value of this option as a string

class dhcpkit.ipv6.extensions.ntp.**NTPServerFQDNSubOption**(*fqdn: str = ''*)
Bases: *dhcpkit.ipv6.extensions.ntp.NTPSubOption* (page 73)

RFC 5908#section-4.3[56]

This suboption is intended to appear inside the OPTION_NTP_SERVER option. It specifies the FQDN of an NTP server or SNTP server available to the client.

The format of the NTP Server FQDN Suboption is:

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    NTP_SUBOPTION_SRV_FQDN     |         suboption-len         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
|                       FQDN of NTP server                      |
:                                                               :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

suboption-code NTP_SUBOPTION_SRV_FQDN (3).

suboption-len Length of the included FQDN field.

FQDN Fully-Qualified Domain Name of the NTP server or SNTP server. This field MUST be encoded as described in **RFC 3315**[57], Section 8. Internationalized domain names are not allowed in this field.

static config_datatype(*value: str*) → str
Convert string data from the configuration to, well, a string. But a validated string!

Parameters **value** – String from config file

Returns Parsed fqdn

fqdn = None
Domain name of an NTP server

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

save() → Union
Save the internal state of this object as a buffer.

[56] https://tools.ietf.org/html/rfc5908.html#section-4.3
[57] https://tools.ietf.org/html/rfc3315.html

Returns The buffer with the data from this element

suboption_type = 3

validate()
Validate that the contents of this object conform to protocol specs.

value
Return a simple string representation of the value of this sub-option.

Returns The value of this option as a string

class dhcpkit.ipv6.extensions.ntp.**NTPServersOption**(*options: Iterable = None*)
Bases: *dhcpkit.ipv6.options.Option* (page 168)

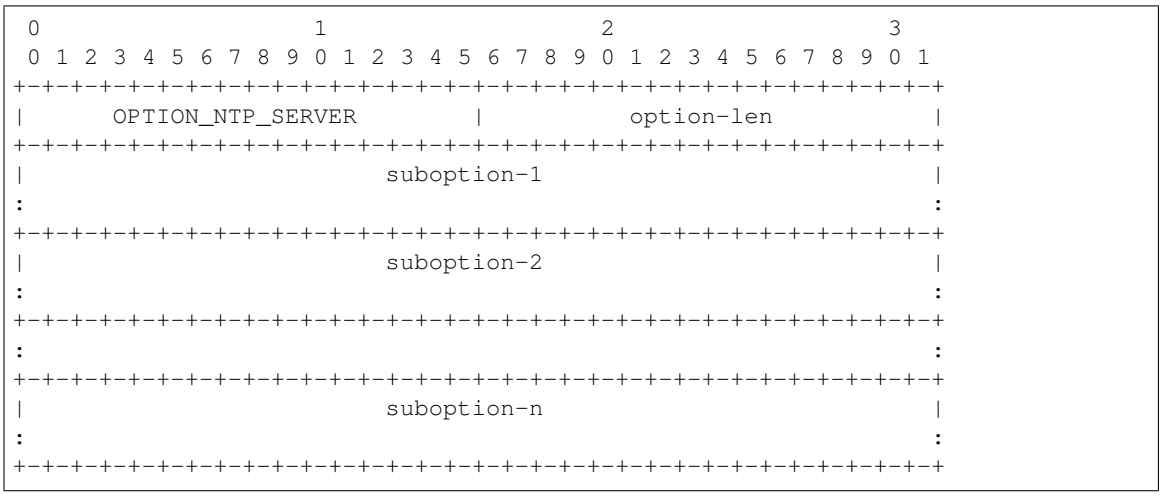
RFC 5908#section-4[58]

This option serves as a container for server location information related to one NTP server or Simple Network Time Protocol (SNTP) **RFC 4330**[59] server. This option can appear multiple times in a DHCPv6 message. Each instance of this option is to be considered by the NTP client or SNTP client as a server to include in its configuration.

The option itself does not contain any value. Instead, it contains one or several suboptions that carry NTP server or SNTP server location. This option MUST include one, and only one, time source suboption. The currently defined time source suboptions are NTP_OPTION_SRV_ADDR, NTP_OPTION_SRV_MC_ADDR, and NTP_OPTION_SRV_FQDN. It carries the NTP server or SNTP server location as a unicast or multicast IPv6 address or as an NTP server or SNTP server FQDN. More time source suboptions may be defined in the future. While the FQDN option offers the most deployment flexibility, resiliency as well as security, the IP address options are defined to cover cases where a DNS dependency is not desirable.

If the NTP server or SNTP server location is an IPv6 multicast address, the client SHOULD use this address as an NTP multicast group address and listen to messages sent to this group in order to synchronize its clock.

The format of the NTP Server Option is:



option-code OPTION_NTP_SERVER (56).

option-len Total length of the included suboptions.

This document does not define any priority relationship between the client's embedded configuration (if any) and the NTP or SNTP servers discovered via this option. In particular, the client is allowed to simultaneously use its own configured NTP servers or SNTP servers and the servers discovered via DHCP.

[58] https://tools.ietf.org/html/rfc5908.html#section-4
[59] https://tools.ietf.org/html/rfc4330.html

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 56

options = None
List of NTP server sub-options

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()
Validate that the contents of this object conform to protocol specs.

class dhcpkit.ipv6.extensions.ntp.**NTPSubOption**
Bases: *dhcpkit.protocol_element.ProtocolElement* (page 218)

RFC 5908[60]

config_datatype = None

classmethod determine_class(*buffer: bytes*, *offset: int = 0*) → type
Return the appropriate subclass from the registry, or UnknownNTPSubOption if no subclass is registered.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading

Returns The best known class for this suboption data

parse_suboption_header(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → Tuple
Parse the option code and length from the buffer and perform some basic validation.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer and the value of the suboption-len field

suboption_type = 0

value
Return a simple string representation of the value of this sub-option.

Returns The value of this option as a string

[60] https://tools.ietf.org/html/rfc5908.html

class dhcpkit.ipv6.extensions.ntp.**UnknownNTPSubOption**(*suboption_type: int = 0*, *suboption_data: bytes = b''*)

Bases: *dhcpkit.ipv6.extensions.ntp.NTPSubOption* (page 73)

Container for raw NTP sub-option content for cases where we don't know how to decode it.

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

suboption_data = None
Data for this sub-option

validate()
Validate that the contents of this object conform to protocol specs.

value
Return a simple string representation of the value of this sub-option.

Returns The value of this option as a string

dhcpkit.ipv6.extensions.ntp_suboption_registry module

The NTP suboption registry

class dhcpkit.ipv6.extensions.ntp_suboption_registry.**NTPSuboptionRegistry**
Bases: *dhcpkit.registry.Registry* (page 220)

Registry for NTP Suboptions

entry_point = 'dhcpkit.ipv6.options.ntp.suboptions'

get_name(*item: object*) → str
Get the name for the by_name mapping.

Parameters **item** – The item to determine the name of

Returns The name to use as key in the mapping

dhcpkit.ipv6.extensions.pd_exclude module

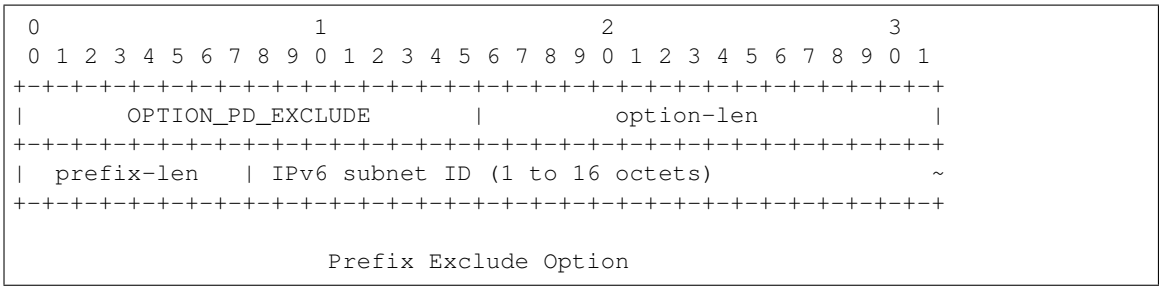
Implementation of the DHCPv6-PD-Exclude option as specified in **RFC 4833**[61].

class dhcpkit.ipv6.extensions.pd_exclude.**PDExcludeOption**(*prefix_length: int = 64*, *subnet_id: bytes = None*)

Bases: *dhcpkit.ipv6.options.Option* (page 168)

[61] https://tools.ietf.org/html/rfc4833.html

RFC 6603#section-4.2[62]



Prefix Exclude Option

option-code: OPTION_PD_EXCLUDE (67).

option-len: 1 + length of IPv6 subnet ID in octets. A valid option-len is between 2 and 17.

prefix-len: The length of the excluded prefix in bits. The prefix-len MUST be between 'OPTION_IAPREFIX prefix-length'+1 and 128.

IPv6 subnet ID: A variable-length IPv6 subnet ID up to 128 bits.

The IPv6 subnet ID contains prefix-len minus 'OPTION_IAPREFIX prefix- length' bits extracted from the excluded prefix starting from the bit position 'OPTION_IAPREFIX prefix-length'. The extracted subnet ID MUST be left-shifted to start from a full octet boundary, i.e., left- shift of 'OPTION_IAPREFIX prefix-length' mod 8 bits. The subnet ID MUST be zero-padded to the next full octet boundary.

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 67

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()
Validate that the contents of this object conform to protocol specs.

dhcpkit.ipv6.extensions.prefix_delegation module

Implementation of Prefix Delegation options as specified in RFC 3633[63].

class dhcpkit.ipv6.extensions.prefix_delegation.**IAPDOption**(*iaid: bytes = b'x00x00x00x00'*, *t1: int = 0*, *t2: int = 0*, *options: Iterable = None*)

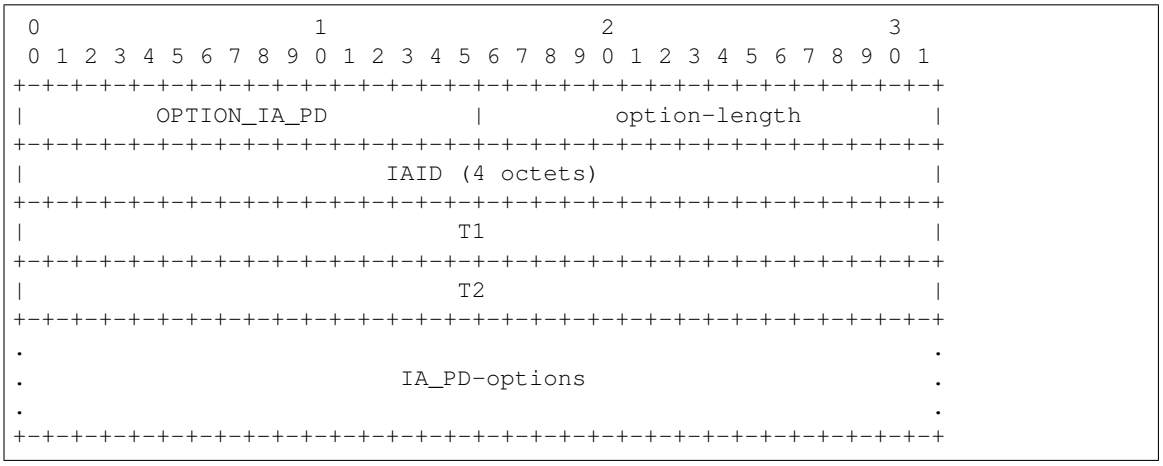
Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 3633#section-9[64]

[62] https://tools.ietf.org/html/rfc6603.html#section-4.2
[63] https://tools.ietf.org/html/rfc3633.html
[64] https://tools.ietf.org/html/rfc3633.html#section-9

The IA_PD option is used to carry a prefix delegation identity association, the parameters associated with the IA_PD and the prefixes associated with it.

The format of the IA_PD option is:



option-code OPTION_IA_PD (25).

option-length 12 + length of IA_PD-options field.

IAID The unique identifier for this IA_PD; the IAID must be unique among the identifiers for all of this requesting router's IA_PDs.

T1 The time at which the requesting router should contact the delegating router from which the prefixes in the IA_PD were obtained to extend the lifetimes of the prefixes delegated to the IA_PD; T1 is a time duration relative to the current time expressed in units of seconds.

T2 The time at which the requesting router should contact any available delegating router to extend the lifetimes of the prefixes assigned to the IA_PD; T2 is a time duration relative to the current time expressed in units of seconds.

IA_PD-options Options associated with this IA_PD.

The IA_PD-options field encapsulates those options that are specific to this IA_PD. For example, all of the IA_PD Prefix Options carrying the prefixes associated with this IA_PD are in the IA_PD-options field.

An IA_PD option may only appear in the options area of a DHCP message. A DHCP message may contain multiple IA_PD options.

The status of any operations involving this IA_PD is indicated in a Status Code option in the IA_PD-options field.

Note that an IA_PD has no explicit "lifetime" or "lease length" of its own. When the valid lifetimes of all of the prefixes in a IA_PD have expired, the IA_PD can be considered as having expired. T1 and T2 are included to give delegating routers explicit control over when a requesting router should contact the delegating router about a specific IA_PD.

In a message sent by a requesting router to a delegating router, values in the T1 and T2 fields indicate the requesting router's preference for those parameters. The requesting router sets T1 and T2 to zero if it has no preference for those values. In a message sent by a delegating router to a requesting router, the requesting router MUST use the values in the T1 and T2 fields for the T1 and T2 parameters. The values in the T1 and T2 fields are the number of seconds until T1 and T2.

The delegating router selects the T1 and T2 times to allow the requesting router to extend the lifetimes of any prefixes in the IA_PD before the lifetimes expire, even if the delegating router is unavailable for some short period of time. Recommended values for T1 and T2 are .5 and .8 times the shortest preferred lifetime of the prefixes in the IA_PD that the delegating router is willing to extend, respectively. If the time at which the prefixes in an IA_PD are to be renewed is to be left to the discretion of the requesting router, the delegating router sets T1 and T2 to 0.

If a delegating router receives an IA_PD with T1 greater than T2, and both T1 and T2 are greater than 0, the delegating router ignores the invalid values of T1 and T2 and processes the IA_PD as though the delegating router had set T1 and T2 to 0.

If a requesting router receives an IA_PD with T1 greater than T2, and both T1 and T2 are greater than 0, the client discards the IA_PD option and processes the remainder of the message as though the delegating router had not included the IA_PD option.

get_option_of_type(**args*) → Union
Get the first option that is a subclass of the given class.

Parameters **args** – The classes to look for

Returns The option or None

get_options_of_type(**args*) → List
Get all options that are subclasses of the given class.

Parameters **args** – The classes to look for

Returns The list of options

get_prefixes() → List
Get all prefixes from IAPrefixOptions

Returns list if prefixes

iaid = None
The unique identifier for this IA_PD

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 25

options = None
The list of options contained in this IAPDOption

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

t1 = None
The time at which the client contacts the server to renew its prefixes

t2 = None
The time at which the client contacts any available server to rebind its prefixes

validate()
Validate that the contents of this object conform to protocol specs.

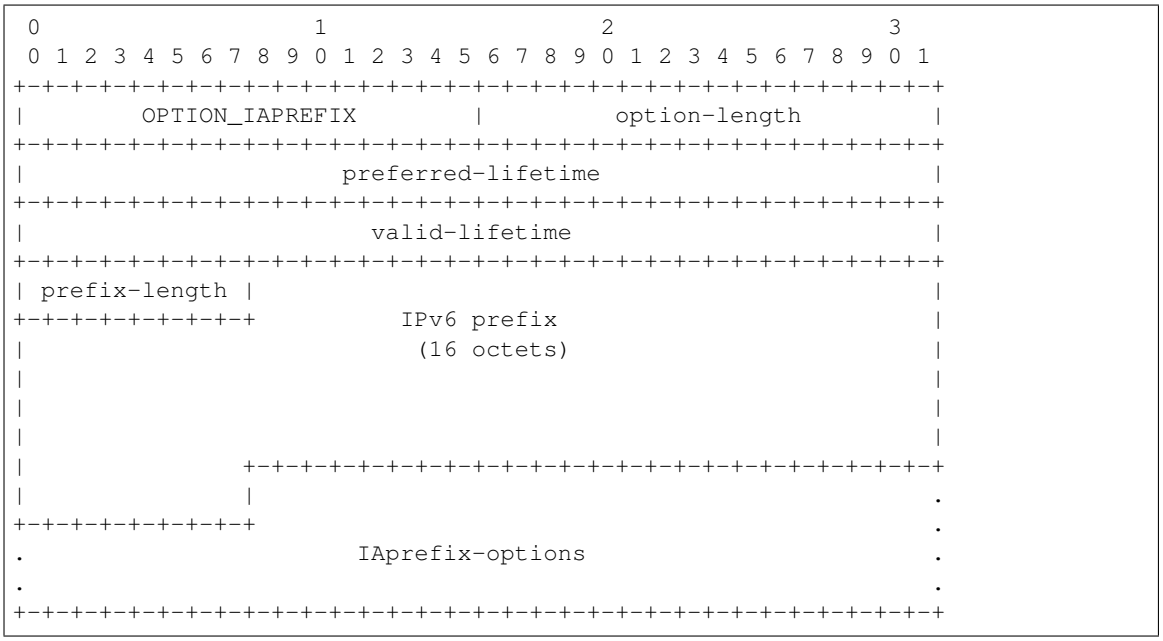
class dhcpkit.ipv6.extensions.prefix_delegation.**IAPrefixOption**(*prefix: ipaddress.IPv6Network = None*, *preferred_lifetime: int = 0*, *valid_lifetime: int = 0*, *options: Iterable = None*)

Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 3633#section-10[65]

The IA_PD Prefix option is used to specify IPv6 address prefixes associated with an IA_PD. The IA_PD Prefix option must be encapsulated in the IA_PD-options field of an IA_PD option.

The format of the IA_PD Prefix option is:



option-code OPTION_IAPREFIX (26).

option-length 25 + length of IAprefix-options field.

preferred-lifetime The recommended preferred lifetime for the IPv6 prefix in the option, expressed in units of seconds. A value of 0xFFFFFFFF represents infinity.

valid-lifetime The valid lifetime for the IPv6 prefix in the option, expressed in units of seconds. A value of 0xFFFFFFFF represents infinity.

prefix-length Length for this prefix in bits.

IPv6-prefix An IPv6 prefix.

IAprefix-options Options associated with this prefix.

In a message sent by a requesting router to a delegating router, the values in the fields can be used to indicate the requesting router's preference for those values. The requesting router may send a value of zero to indicate no preference. A requesting router may set the IPv6 prefix field to zero and a given value in the prefix-length field to indicate a preference for the size of the prefix to be delegated.

[65] https://tools.ietf.org/html/rfc3633.html#section-10

In a message sent by a delegating router the preferred and valid lifetimes should be set to the values of AdvPreferredLifetime and AdvValidLifetime as specified in section 6.2.1, "Router Configuration Variables" of **RFC 2461**[66] [4], unless administratively configured.

A requesting router discards any prefixes for which the preferred lifetime is greater than the valid lifetime. A delegating router ignores the lifetimes set by the requesting router if the preferred lifetime is greater than the valid lifetime and ignores the values for T1 and T2 set by the requesting router if those values are greater than the preferred lifetime.

The values in the preferred and valid lifetimes are the number of seconds remaining for each lifetime.

An IA_PD Prefix option may appear only in an IA_PD option. More than one IA_PD Prefix Option can appear in a single IA_PD option.

The status of any operations involving this IA_PD Prefix option is indicated in a Status Code option in the IAprefix-options field.

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 26

options = None
The list of options related to this IAPrefixOption

preferred_lifetime = None
The preferred lifetime of this IPv6 prefix

prefix = None
The IPv6 prefix

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

valid_lifetime = None
The valid lifetime of this IPv6 prefix

validate()
Validate that the contents of this object conform to protocol specs.

dhcpkit.ipv6.extensions.relay_echo_request module

Implementation of the Echo Request option as specified in **RFC 4994**[67].

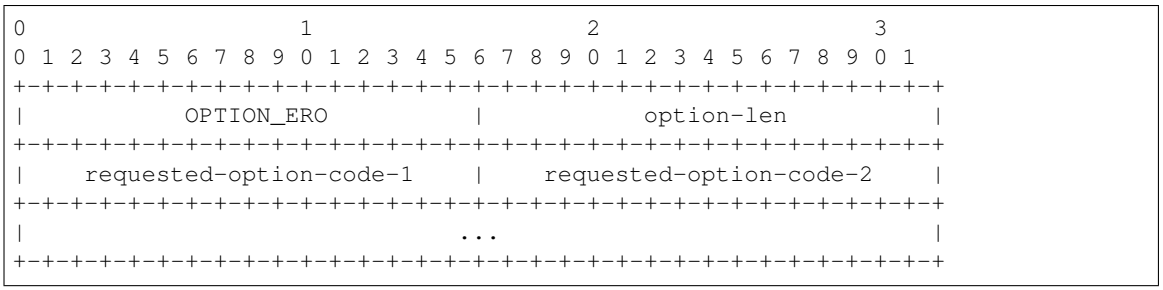
class dhcpkit.ipv6.extensions.relay_echo_request.**EchoRequestOption**(*requested_options: Iterable = None*)

Bases: *dhcpkit.ipv6.options.Option* (page 168)

[66] https://tools.ietf.org/html/rfc2461.html
[67] https://tools.ietf.org/html/rfc4994.html

The relay agent adds options in the Relay Forward message that the server uses to guide its decision making with regard to address assignment, prefix delegation, and configuration parameters. The relay agent also knows which of these options that it will need to efficiently return replies to the client. It uses the relay agent Echo Request option to inform the server of the list of relay agent options that the server must echo back.

The format of the DHCPv6 Relay Agent Echo Request option is shown below:



option-code OPTION_ERO (43).

option-len 2 * number of requested options.

requested-option-code-n The option code for an option requested by the relay agent.

display_requested_options() → List
Provide a nicer output when displaying the requested options.

Returns A list of option names

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 43

requested_options = None
The list of option type numbers that the relay wants to receive back

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()
Validate that the contents of this object conform to protocol specs.

dhcpkit.ipv6.extensions.remote_id module

Implementation of Remote-ID option as specified in **RFC 4649**[68].

[68] https://tools.ietf.org/html/rfc4649.html

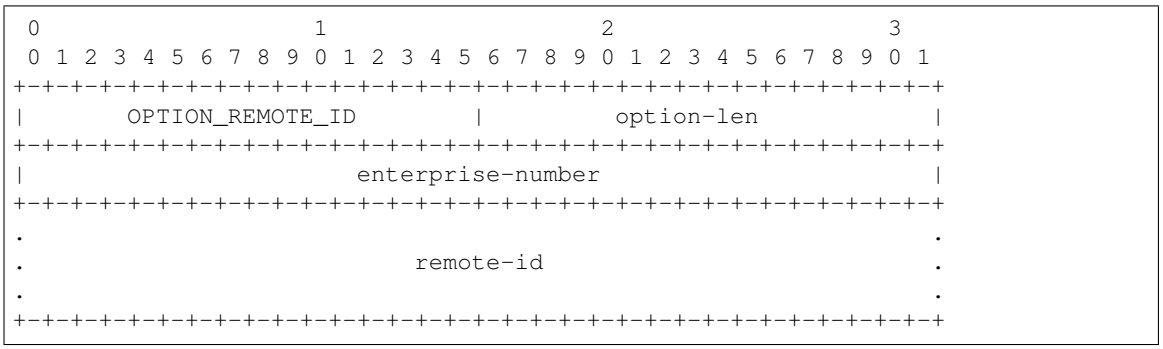
class dhcpkit.ipv6.extensions.remote_id.**RemoteIdOption**(*enterprise_number: int = 0*, *remote_id: bytes = b''*)

Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 4649#section-3[69]

This option may be added by DHCPv6 relay agents that terminate switched or permanent circuits and have mechanisms to identify the remote host end of the circuit.

The format of the DHCPv6 Relay Agent Remote-ID option is shown below:



option-code OPTION_REMOTE_ID (37).

option-len 4 + the length, in octets, of the remote-id field. The minimum option-len is 5 octets.

enterprise-number The vendor's registered Enterprise Number as registered with IANA [5].

remote-id The opaque value for the remote-id.

The definition of the remote-id carried in this option is vendor specific. The vendor is indicated in the enterprise-number field. The remote-id field may be used to encode, for instance:

- a "caller ID" telephone number for dial-up connection
- a "user name" prompted for by a Remote Access Server
- a remote caller ATM address
- a "modem ID" of a cable data modem
- the remote IP address of a point-to-point link
- a remote X.25 address for X.25 connections
- an interface or port identifier

Each vendor must ensure that the remote-id is unique for its enterprise-number, as the octet sequence of enterprise-number followed by remote-id must be globally unique. One way to achieve uniqueness might be to include the relay agent's DHCP Unique Identifier (DUID) [1] in the remote-id.

enterprise_number = None
The enterprise number[70] as registered with IANA

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading

[69] https://tools.ietf.org/html/rfc4649.html#section-3
[70] http://www.iana.org/assignments/enterprise-numbers

- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 37

remote_id = None
The remote-id as bytes

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()
Validate that the contents of this object conform to protocol specs.

dhcpkit.ipv6.extensions.sip_servers module

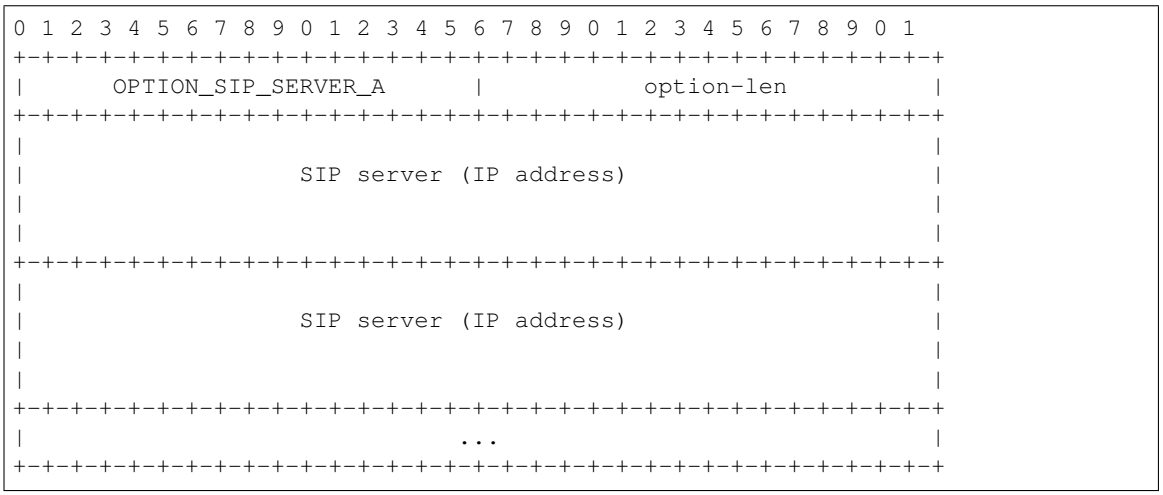
Implementation of SIP options as specified in **RFC 3319**[71].

class dhcpkit.ipv6.extensions.sip_servers.**SIPServersAddressListOption**(*sip_servers: Iterable = None*)

Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 3319#section-3.2[72]

This option specifies a list of IPv6 addresses indicating SIP outbound proxy servers available to the client. Servers MUST be listed in order of preference.



option-code OPTION_SIP_SERVER_A (22).

option-length Length of the 'options' field in octets; must be a multiple of 16.

SIP server IPv6 address of a SIP server for the client to use. The servers are listed in the order of preference for use by the client.

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

[71] https://tools.ietf.org/html/rfc3319.html
[72] https://tools.ietf.org/html/rfc3319.html#section-3.2

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 22

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

sip_servers = None
List of IPv6 addresses of SIP servers

validate()
Validate that the contents of this object conform to protocol specs.

class dhcpkit.ipv6.extensions.sip_servers.**SIPServersDomainNameListOption**(*domain_names: Iterable = None*)

Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 3319#section-3.1[73]

The option length is followed by a sequence of labels, encoded according to Section 3.1 of **RFC 1035**[74] [5], quoted below:

> "Domain names in messages are expressed in terms of a sequence of labels. Each label is represented as a one octet length field followed by that number of octets. Since every domain name ends
>
> with the null label of the root, a domain name is terminated by a length byte of zero. The high order two bits of every length octet must be zero, and the remaining six bits of the length field limit the label to 63 octets or less. To simplify implementations, the total length of a domain name (i.e., label octets and label length octets) is restricted to 255 octets or less."
>
> **RFC 1035**[75] encoding was chosen to accommodate future internationalized domain name mechanisms.

The option MAY contain multiple domain names, but these SHOULD refer to different NAPTR records, rather than different A records. The client MUST try the records in the order listed, applying the mechanism described in Section 4.1 of **RFC 3263**[76] [3] for each. The client only resolves the subsequent domain names if attempts to contact the first one failed or yielded no common transport protocols between client and server or denote a domain administratively prohibited by client policy. Domain names MUST be listed in order of preference.

> Use of multiple domain names is not meant to replace NAPTR or SRV records, but rather to allow a single DHCP server to indicate outbound proxy servers operated by multiple providers.

The DHCPv6 option has the format shown here:

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

(continues on next page)

[73] https://tools.ietf.org/html/rfc3319.html#section-3.1
[74] https://tools.ietf.org/html/rfc1035.html
[75] https://tools.ietf.org/html/rfc1035.html
[76] https://tools.ietf.org/html/rfc3263.html

(continued from previous page)

```
|      OPTION_SIP_SERVER_D      |         option-length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  SIP Server Domain Name List                  |
|                               ...                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

option-code OPTION_SIP_SERVER_D (21).

option-length Length of the 'SIP Server Domain Name List' field in octets; variable.

SIP Server Domain Name List The domain names of the SIP outbound proxy servers for the client to use. The domain names are encoded as specified in Section 8 ("Representation and use of domain names") of the DHCPv6 specification [1].

domain_names = None
List of domain names of SIP servers

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 21

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()
Validate that the contents of this object conform to protocol specs.

dhcpkit.ipv6.extensions.sntp module

Implementation of SNTP option as specified in **RFC 4075**[77].

class dhcpkit.ipv6.extensions.sntp.**SNTPServersOption**(*sntp_servers: Iterable = None*)

Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 4075#section-4[78]

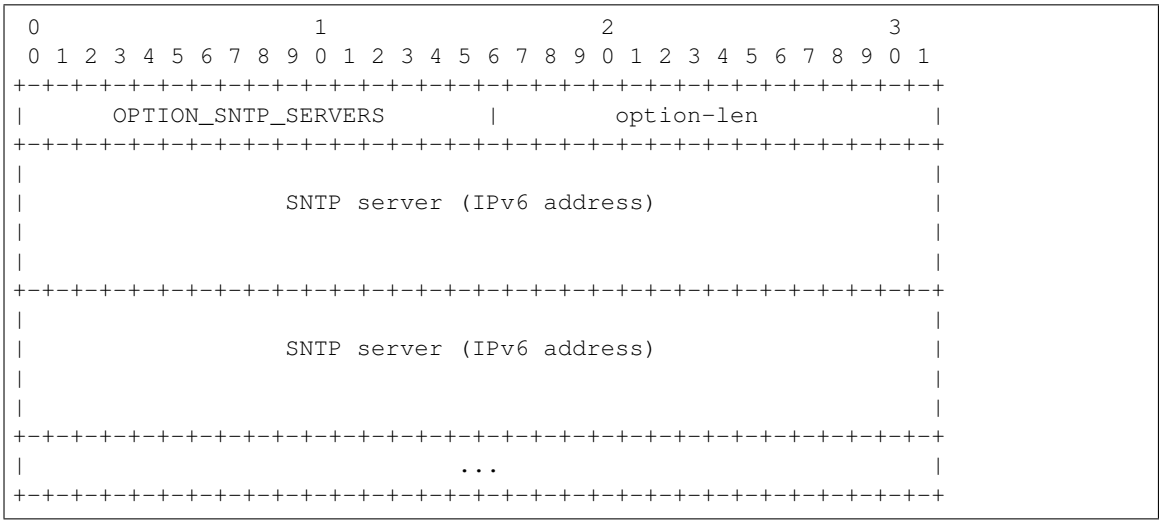
The Simple Network Time Protocol servers option provides a list of one or more IPv6 addresses of SNTP [3] servers available to the client for synchronization. The clients use these SNTP servers to synchronize their system time to that of the standard time servers. Clients MUST treat the list of SNTP servers as an ordered list. The server MAY list the SNTP servers in decreasing order of preference.

The option defined in this document can only be used to configure information about SNTP servers that can be reached using IPv6. The DHCP option to configure information about IPv4 SNTP servers can be found

[77] https://tools.ietf.org/html/rfc4075.html
[78] https://tools.ietf.org/html/rfc4075.html#section-4

in RFC 2132[79] [4]. Mechanisms for configuring IPv4/IPv6 dual- stack applications are being considered, but are not specified in this document.

The format of the Simple Network Time Protocol servers option is as shown below:



option-code OPTION_SNTP_SERVERS (31).

option-len Length of the 'SNTP server' fields, in octets; it must be a multiple of 16.

SNTP server IPv6 address of SNTP server.

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 31

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

sntp_servers = None
List of IPv6 addresses of SNTP servers

validate()
Validate that the contents of this object conform to protocol specs.

dhcpkit.ipv6.extensions.sol_max_rt module

Implementation of SOL-MAX-RT and INF-MAX-RT options as specified in RFC 7083[80].

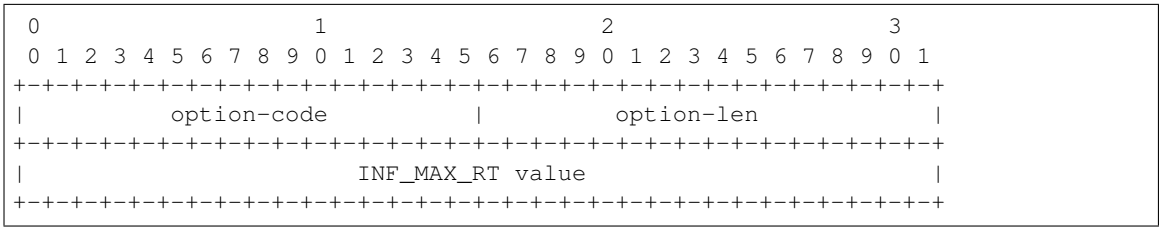
class dhcpkit.ipv6.extensions.sol_max_rt.**InfMaxRTOption**(*inf_max_rt: int = 0*)
Bases: *dhcpkit.ipv6.options.Option* (page 168)

[79] https://tools.ietf.org/html/rfc2132.html
[80] https://tools.ietf.org/html/rfc7083.html

RFC 7083#section-5[81]

A DHCPv6 server sends the INF_MAX_RT option to a client to override the default value of INF_MAX_RT. The value of INF_MAX_RT in the option replaces the default value defined in Section 3. One use for the INF_MAX_RT option is to set a longer value for INF_MAX_RT, which reduces the Information-request traffic from a client that has not received a response to its Information-request messages.

The format of the INF_MAX_RT option is:



option-code OPTION_INF_MAX_RT (83).

option-len

4.

INF_MAX_RT value Overriding value for INF_MAX_RT in seconds; MUST be in range: 60 <= "value" <= 86400 (1 day).

inf_max_rt = None
The new value for INF_MAX_RT for the client

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 83

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()
Validate that the contents of this object conform to protocol specs.

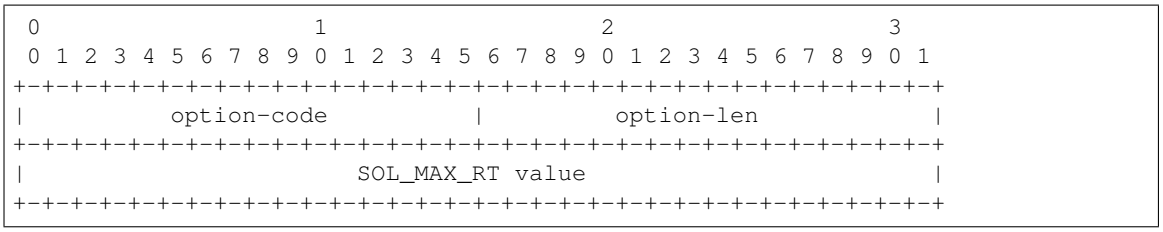
class dhcpkit.ipv6.extensions.sol_max_rt.**SolMaxRTOption**(*sol_max_rt: int = 0*)
Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 7083#section-4[82]

A DHCPv6 server sends the SOL_MAX_RT option to a client to override the default value of SOL_MAX_RT. The value of SOL_MAX_RT in the option replaces the default value defined in Section 3. One use for the SOL_MAX_RT option is to set a longer value for SOL_MAX_RT, which reduces the Solicit traffic from a client that has not received a response to its Solicit messages.

[81] https://tools.ietf.org/html/rfc7083.html#section-5
[82] https://tools.ietf.org/html/rfc7083.html#section-4

The format of the SOL_MAX_RT option is:



option-code OPTION_SOL_MAX_RT (82).

option-len

4.

SOL_MAX_RT value Overriding value for SOL_MAX_RT in seconds; MUST be in range: 60 <= "value" <= 86400 (1 day).

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 82

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

sol_max_rt = None
The new value of SOL_MAX_RT for the client

validate()
Validate that the contents of this object conform to protocol specs.

dhcpkit.ipv6.extensions.subscriber_id module

Implementation of Subscriber-ID option as specified in **RFC 4580**[83].

class dhcpkit.ipv6.extensions.subscriber_id.**SubscriberIdOption**(*subscriber_id: bytes = b''*)

Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 4580#section-2[84]

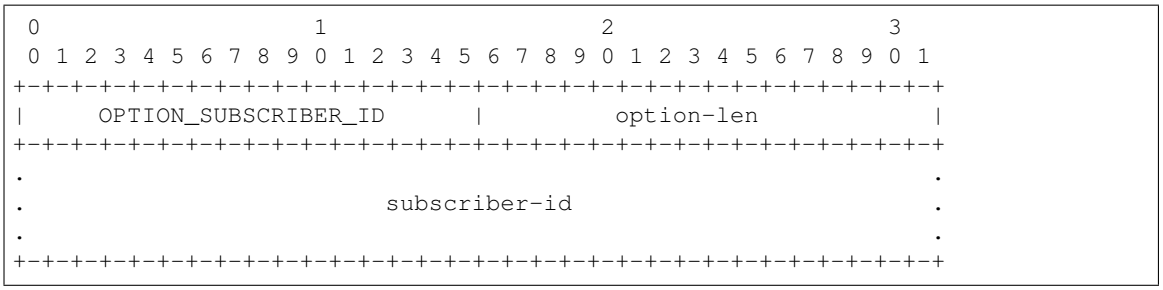
The subscriber-id information allows the service provider to assign/ activate subscriber-specific actions; e.g., assignment of specific IP addresses, prefixes, DNS configuration, trigger accounting, etc. This option is de-coupled from the access network's physical structure, so a subscriber that moves from one access-point to another, for example, would not require reconfiguration at the service provider's DHCPv6 servers.

[83] https://tools.ietf.org/html/rfc4580.html
[84] https://tools.ietf.org/html/rfc4580.html#section-2

The subscriber-id information is only intended for use within a single administrative domain and is only exchanged between the relay agents and DHCPv6 servers within that domain. Therefore, the format and encoding of the data in the option is not standardized, and this specification does not establish any semantic requirements on the data. This specification only defines the option for conveying this information from relay agents to DHCPv6 servers.

However, as the DHCPv4 Subscriber-ID suboption [3] specifies Network Virtual Terminal (NVT) American Standard Code for Information Interchange (ASCII) [4] encoded data, in environments where both DHCPv4 [5] and DHCPv6 are being used, it may be beneficial to use that encoding.

The format of the DHCPv6 Relay Agent Subscriber-ID option is shown below:



option-code OPTION_SUBSCRIBER_ID (38)

option-len length, in octets, of the subscriber-id field. The minimum length is 1 octet.

subscriber-id The subscriber's identity.

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 38

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

subscriber_id = None
The subscriber-id as bytes

validate()
Validate that the contents of this object conform to protocol specs.

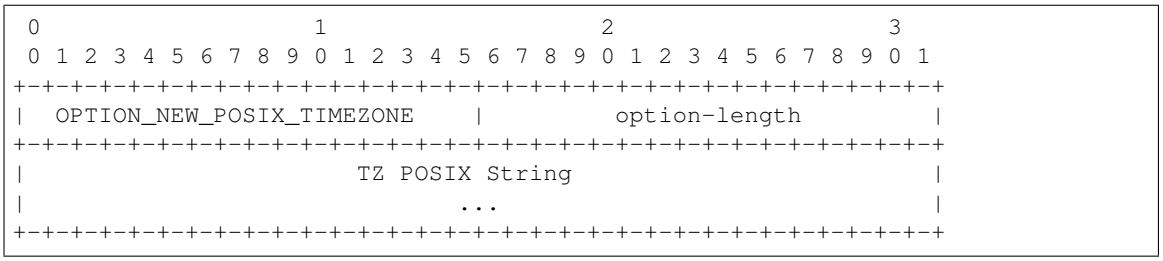
dhcpkit.ipv6.extensions.timezone module

Implementation of timezone options as specified in **RFC 4833**[85].

class dhcpkit.ipv6.extensions.timezone.**PosixTimezoneOption**(*timezone: str = None*)

Bases: *dhcpkit.ipv6.options.Option* (page 168)

[85] https://tools.ietf.org/html/rfc4833.html



option-code: OPTION_NEW_POSIX_TIMEZONE(41)

option-length: the number of octets of the TZ POSIX String Index described below:

TZ POSIX string is a string suitable for the TZ variable as specified by IEEE 1003.1 in Section 8.3, with the exception that a string may not begin with a colon (":"). This string is NOT terminated by an ASCII NULL.

Here is an example: EST5EDT4,M3.2.0/02:00,M11.1.0/02:00

In this case, the string is interpreted as a timezone that is normally five hours behind UTC, and four hours behind UTC during DST, which runs from the second Sunday in March at 02:00 local time through the first Sunday in November at 02:00 local time. Normally the timezone is abbreviated "EST" but during DST it is abbreviated "EDT".

Clients and servers implementing other timezone options MUST support this option for basic compatibility.

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 41

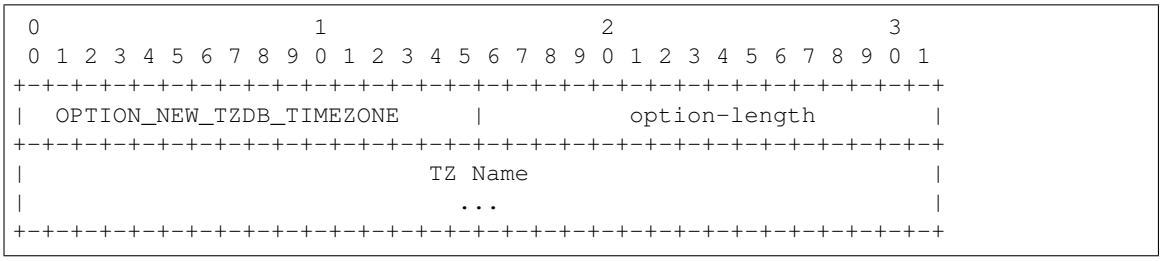
save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()
Validate that the contents of this object conform to protocol specs.

class dhcpkit.ipv6.extensions.timezone.**TZDBTimezoneOption**(*timezone: str = None*)

Bases: *dhcpkit.ipv6.options.Option* (page 168)



option-code: OPTION_NEW_TZDB_TIMEZONE(42)

option-length: the number of octets of the TZ Database String Index described below.

TZ Name is the name of a Zone entry in the database commonly referred to as the TZ database. Specifically, in the database's textual form, the string refers to the name field of a zone line. In order for this option to be useful, the client must already have a copy of the database. This string is NOT terminated with an ASCII NULL.

An example string is: Europe/Zurich.

Clients must already have a copy of the TZ Database for this option to be useful. Configuration of the database is beyond the scope of this document. A client that supports this option SHOULD prefer this option to POSIX string if it recognizes the TZ Name that was returned. If it doesn't recognize the TZ Name, the client MUST ignore this option.

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 42

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()
Validate that the contents of this object conform to protocol specs.

dhcpkit.ipv6.server package

The IPv6 DHCP server

Subpackages

dhcpkit.ipv6.server.duids package

Implementation of DUIDs in the configuration, used for example to configure the server-id.

Subpackages

dhcpkit.ipv6.server.duids.duid_en package

Configuration section for EnterpriseDUID

Submodules

dhcpkit.ipv6.server.duids.duid_en.config module

Configuration section for EnterpriseDUID

dhcpkit.ipv6.server.duids.duid_en.config.**duid_en**(*section*) → dhcpkit.ipv6.duids.EnterpriseDUID

Create a EnterpriseDUID from the data provided in the config section.

Parameters **section** – The section data

Returns The DUID object

dhcpkit.ipv6.server.duids.duid_ll package

Configuration section for LinkLayerDUID

Submodules

dhcpkit.ipv6.server.duids.duid_ll.config module

Configuration section for LinkLayerDUID

dhcpkit.ipv6.server.duids.duid_ll.config.**duid_ll**(*section*) → dhcpkit.ipv6.duids.LinkLayerDUID

Create a LinkLayerDUID from the data provided in the config section.

Parameters **section** – The section data

Returns The DUID object

dhcpkit.ipv6.server.duids.duid_llt package

Configuration section for LinkLayerTimeDUID

Submodules

dhcpkit.ipv6.server.duids.duid_llt.config module

Configuration section for LinkLayerTimeDUID

dhcpkit.ipv6.server.duids.duid_llt.config.**duid_llt**(*section*) → dhcpkit.ipv6.duids.LinkLayerTimeDUID

Create a LinkLayerDUID from the data provided in the config section.

Parameters **section** – The section data

Returns The DUID object

dhcpkit.ipv6.server.extensions package

Extensions to the basic DHCPv6 server

Subpackages

dhcpkit.ipv6.server.extensions.dns package

Handlers for the options defined in dhcpkit.ipv6.extensions.dns

class dhcpkit.ipv6.server.extensions.dns.**DomainSearchListOptionHandler**(*search_list: Iterable*, *always_send: bool = False*)

Bases: *dhcpkit.ipv6.server.handlers.basic.SimpleOptionHandler* (page 123)

Handler for putting RecursiveNameServersOption in responses

combine(*existing_options: Iterable*) → dhcpkit.ipv6.extensions.dns.DomainSearchListOption
Combine multiple options into one.

Parameters **existing_options** – The existing options to include domain names from

Returns The combined option

class dhcpkit.ipv6.server.extensions.dns.**RecursiveNameServersOptionHandler**(*dns_servers: Iterable*, *always_send: bool = False*)

Bases: *dhcpkit.ipv6.server.handlers.basic.SimpleOptionHandler* (page 123)

Handler for putting RecursiveNameServersOption in responses

combine(*existing_options: Iterable*) → dhcpkit.ipv6.extensions.dns.RecursiveNameServersOption
Combine multiple options into one.

Parameters **existing_options** – The existing options to include name servers from

Returns The combined option

Submodules

dhcpkit.ipv6.server.extensions.dns.config module

Configuration elements for the dns option handlers

class dhcpkit.ipv6.server.extensions.dns.config.**DomainSearchListOptionHandlerFactory**(*sect ZCo fig.r*

Bases: *dhcpkit.ipv6.server.handlers.HandlerFactory* (page 121)

Create the handler for the domain search list.

create() → dhcpkit.ipv6.server.extensions.dns.DomainSearchListOptionHandler
Create a handler of this class based on the configuration in the config section.

Returns A handler object

class dhcpkit.ipv6.server.extensions.dns.config.**RecursiveNameServersOptionHandlerFactory**

Bases: *dhcpkit.ipv6.server.handlers.HandlerFactory* (page 121)

Create the handler for recursive name servers.

create() → dhcpkit.ipv6.server.extensions.dns.RecursiveNameServersOptionHandler
Create a handler of this class based on the configuration in the config section.

Returns A handler object

dhcpkit.ipv6.server.extensions.dslite package

Handlers for the options defined in dhcpkit.ipv6.extensions.dslite

class dhcpkit.ipv6.server.extensions.dslite.**AFTRNameOptionHandler**(*fqdn: str*, *always_send: bool = False*)

Bases: *dhcpkit.ipv6.server.handlers.basic.SimpleOptionHandler* (page 123)

Handler for putting an AFTRNameOption in responses

Submodules

dhcpkit.ipv6.server.extensions.dslite.config module

Configuration elements for the DS-Lite server option handlers

class dhcpkit.ipv6.server.extensions.dslite.config.**AFTRNameOptionHandlerFactory**(*section: ZConfig.matcher.*

Bases: *dhcpkit.ipv6.server.handlers.HandlerFactory* (page 121)

Create the handler for the AFTR tunnel endpoint.

create() → dhcpkit.ipv6.server.extensions.dslite.AFTRNameOptionHandler
Create a handler of this class based on the configuration in the config section.

Returns A handler object

dhcpkit.ipv6.server.extensions.leasequery package

Implementation of the Leasequery and Bulk Leasequery extensions.

class dhcpkit.ipv6.server.extensions.leasequery.**LeasequeryHandler**(*store: dhcpkit.ipv6.server.extensions.leaseq*, *allow_from: Iterable = None*, *sensitive_options: Iterable = None*)

Bases: *dhcpkit.ipv6.server.handlers.Handler* (page 121)

Handle leasequery requests and analyse replies that we send out to store any observed leases.

analyse_post(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Watch outgoing replies and store observed leases in the store.

Parameters **bundle** – The transaction bundle containing the outgoing reply

static generate_data_messages(*transaction_id: bytes*, *leases: Iterator*) → Iterator
Generate a leasequery data message for each of the leases, followed by a leasequery done message.

Parameters

- **transaction_id** – The transaction ID to use in the messages
- **leases** – An open iterator for the data we still need to return

Returns Leasequery messages to send to the client

handle(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Perform leasequery if requested.

Parameters **bundle** – The transaction bundle

pre(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Make sure we allow this client to make leasequery requests.

Parameters **bundle** – The transaction bundle

worker_init()
Make sure the store gets a chance to initialise itself.

class dhcpkit.ipv6.server.extensions.leasequery.**LeasequeryStore**
Bases: object[86]

Base class for leasequery stores

build_relay_data_option_from_relay_data(*relay_data: bytes*) → Union
The relay data includes the outer relay message, which is generated inside the server to keep track of where we got the request from. When returning relay data to the leasequery client we build the LQRelayDataOption using this internal relay message only including the real relay messages we received.

Parameters **relay_data** – The raw relay data

Returns The LQRelayDataOption if applicable

static decode_duid(*duid_str: str*) → dhcpkit.ipv6.duids.DUID
Decode DUID from a string.

Parameters **duid_str** – The DUID string

Returns The DUID object

static decode_options(*data: bytes*) → Iterable
Decode a list of options from bytes.

Parameters **data** – The bytes

Returns The list of options

static decode_relay_messages(*data: bytes*) → Union
Decode a chain of relay messages from bytes.

Parameters **data** – The bytes

Returns The relay message

static decode_remote_id(*remote_id_str: str*) → dhcpkit.ipv6.extensions.remote_id.RemoteIdOption
Decode remote id from a string.

Parameters **remote_id_str** – The remote-id string

Returns The remote-id option

static encode_duid(*duid: dhcpkit.ipv6.duids.DUID*) → str
Encode DUID as a string.

Parameters **duid** – The DUID object

[86] https://docs.python.org/3.4/library/functions.html#object

Returns The string representing the DUID

encode_options(*options: Iterable*) → bytes
Encode a list of options as bytes.

Parameters **options** – The list of options

Returns The bytes

encode_relay_messages(*relay_chain: Union*) → bytes
Encode a chain of relay messages as bytes.

Parameters **relay_chain** – The incoming relay messages

Returns The bytes

static encode_remote_id(*remote_id_option: dhcpkit.ipv6.extensions.remote_id.RemoteIdOption*) → str
Encode remote id as a string.

Parameters **remote_id_option** – The remote-id option

Returns The string representing the remote-id

static filter_options(*options: Iterable*, *unwanted_option_types: Iterable*) → Iterable
Remove unwanted data from the options.

Parameters

- **options** – The options to filter
- **unwanted_option_types** – List of option types to filter out

Returns The filtered options

filter_requested_options(*options: Iterable*, *requested_options: Iterable*)
Only return options that are requested by the leasequery client.

Parameters

- **options** – The original list of options
- **requested_options** – The list of requested options

Returns The filtered list

filter_sensitive_options(*options: Iterable*) → Iterable
Remove sensitive data from the options.

Parameters **options** – The options to filter

Returns The filtered options

filter_storable_options(*options: Iterable*) → Iterable
Only include storable data from the options.

Parameters **options** – The options to filter

Returns The filtered options

find_leases(*query: dhcpkit.ipv6.extensions.leasequery.LQQueryOption*) → Tuple
Find all leases that match the given query.

Parameters **query** – The query

Returns The number of leases and an iterator over tuples of link-address and corresponding client data

get_address_leases(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*) → Iterator
Search through the reply and return all addresses given to the client.

Parameters **bundle** – The transaction bundle

Returns The address options

get_prefix_leases(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*) → Iterator
Search through the reply and return all prefixes given to the client.

Parameters **bundle** – The transaction bundle

Returns The prefix options

static get_relay_ids(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*) → Iterator
Go through all the relay messages and return all relay-ids found as lowercase hex strings

Parameters **bundle** – The transaction bundle

Returns The relay-ids as hex strings

get_remote_ids(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*) → Iterator
Go through all the relay messages and return all remote-ids found as lowercase hex strings

Parameters **bundle** – The transaction bundle

Returns The remote-ids as hex strings

static is_accepted(*element: Union*) → bool
Check if there is no status code that signals rejection.

Parameters **element** – The element to look in

Returns Whether the status is ok

remember_lease(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Remember the leases in the given transaction bundle so they can be queried later.

Parameters **bundle** – The transaction to remember

worker_init(*sensitive_options: Iterable*)
Separate initialisation that will be called in each worker process that is created. Things that can't be forked (think database connections etc) have to be initialised here.

Parameters **sensitive_options** – The options that are not allowed to be stored

class dhcpkit.ipv6.server.extensions.leasequery.**UnansweredLeasequeryHandler**
Bases: *dhcpkit.ipv6.server.handlers.Handler* (page 121)

When there are leasequeries that haven't been handled at the end of the handling phase that means that no handler understood the query.

post(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Check for unhandled leasequeries.

Parameters **bundle** – The transaction bundle

dhcpkit.ipv6.server.extensions.leasequery.**create_cleanup_handlers**() → List
Create handlers to handle unhandled queries

Returns Handlers to add to the handler chain

Submodules

dhcpkit.ipv6.server.extensions.leasequery.config module

Config processing for a handler to echo a LinkLayerIdOption back to the relay

class dhcpkit.ipv6.server.extensions.leasequery.config.**LeasequeryHandlerFactory**(*section*)
Bases: *dhcpkit.ipv6.server.handlers.HandlerFactory* (page 121)

Config processing for a handler to echo a LinkLayerIdOption back to the relay

create()
Create a leasequery handler.

Returns A leasequery handler

class dhcpkit.ipv6.server.extensions.leasequery.config.**LeasequerySqliteStoreFactory**(*sectio*
Bases: *dhcpkit.common.server.config_elements.ConfigElementFactory* (page 42)

Factory for LeasequerySqliteStore

create()
Create a leasequery store.

Returns A leasequery store

static name_datatype(*v*)

dhcpkit.ipv6.server.extensions.leasequery.config.**sensitive_option_name**(*value: str*) → int
If the argument is a number then check if it is a 16-bit unsigned integer and return it. Otherwise see if we have an option implementation with the given name, and return its option-type code.

Parameters **value** – The name or number of a DHCPv6 option

Returns The number of the option

dhcpkit.ipv6.server.extensions.leasequery.sqlite module

SQLIte based implementation of a leasequery store

class dhcpkit.ipv6.server.extensions.leasequery.sqlite.**LeasequerySqliteStore**(*filename: str*)
Bases: *dhcpkit.ipv6.server.extensions.leasequery.LeasequeryStore* (page 94)

A leasequery store using a SQLite database.

create_tables()
Create the tables required for this leasequery implementation

db = None
Workers store the database connection here

find_client_by_address(*query: dhcpkit.ipv6.extensions.leasequery.LQQueryOption*) → List
Get the row ids of the clients we want to return.

Parameters **query** – The query

Returns A list of row ids

find_client_by_client_id(*query: dhcpkit.ipv6.extensions.leasequery.LQQueryOption*) → List
Get the row ids of the clients we want to return.

Parameters **query** – The query

Returns A list of row ids

find_client_by_link_address(*query: dhcpkit.ipv6.extensions.leasequery.LQQueryOption*) → List
Get the row ids of the clients we want to return.

Parameters **query** – The query

Returns A list of row ids

find_client_by_relay_id(*query: dhcpkit.ipv6.extensions.leasequery.LQQueryOption*) → List
Get the row ids of the clients we want to return.

Parameters **query** – The query

Returns A list of row ids

find_client_by_remote_id(*query: dhcpkit.ipv6.extensions.leasequery.LQQueryOption*) → List
Get the row ids of the clients we want to return.

Parameters **query** – The query

Returns A list of row ids

find_leases(*query: dhcpkit.ipv6.extensions.leasequery.LQQueryOption*) → Tuple
Find all leases that match the given query.

Parameters **query** – The query

Returns The number of leases and an iterator over tuples of link-address and corresponding client data

generate_client_data_options(*client_row_ids: Iterable*, *requested_options: Iterable*) → Iterable
Create a generator for the data of the specified client rows/

Parameters

- **client_row_ids** – The list of client rows what we are interested in
- **requested_options** – Option types explicitly requested by the leasequery client

Returns The client data options for those rows

get_client_row_id(*client_id_str: str*, *link_address_long: str*, *create: bool = True*) → Union
Get the client's row id, creating the client row if necessary.

Parameters

- **client_id_str** – The DUID of the client as a string
- **link_address_long** – The fully expanded link address
- **create** – Should we create this record if it doesn't exist?

Returns The row id

open_database() → sqlite3.Connection
Open the database with the right settings.

Returns The database connection

remember_lease(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Remember the leases in the given transaction bundle so they can be queried later.

Parameters **bundle** – The transaction to remember

replace_relay_ids(*client_row_id: int*, *relay_ids: Iterable*)
Replace the existing relay-id records with the relay-ids provided.

Parameters

- **client_row_id** – The id of the client record
- **relay_ids** – The new relay-ids

replace_remote_ids(*client_row_id: int*, *remote_ids: Iterable*)
Replace the existing remote-id records with the remote-ids provided.

Parameters

- **client_row_id** – The id of the client record
- **remote_ids** – The new remote-ids

sqlite_filename = None
Name of the database file

update_address_leases(*client_row_id: int*, *address_leases: Iterator*)
Update address leases in the database and remove expired ones.

Parameters

- **client_row_id** – The id of the client record
- **address_leases** – The updated leases to record

update_last_interaction(*client_row_id: int*, *options: Iterable*, *relay_chain: Union*)
Keep track of when we last communicated with this client.

Parameters

- **client_row_id** – The row id of the client
- **options** – Options of the last response
- **relay_chain** – The incoming relay messages

update_prefix_leases(*client_row_id: int*, *prefix_leases: Iterator*)
Update prefix leases in the database and remove expired ones.

Parameters

- **client_row_id** – The id of the client record
- **prefix_leases** – The updated leases to record

worker_init(*sensitive_options: Iterable*)
Worker initialisation: open database connection

Parameters **sensitive_options** – The type-numbers of options that are not allowed to be stored

dhcpkit.ipv6.server.extensions.linklayer_id package

Handlers for the options defined in dhcpkit.ipv6.extensions.linklayer

class dhcpkit.ipv6.server.extensions.linklayer_id.**CopyLinkLayerIdOptionHandler**
Bases: *dhcpkit.ipv6.server.handlers.basic_relay.CopyRelayOptionHandler* (page 124)

The handler for LinkLayerIdOption in relay messages

Submodules

dhcpkit.ipv6.server.extensions.linklayer_id.config module

Config processing for a handler to echo a LinkLayerIdOption back to the relay

class dhcpkit.ipv6.server.extensions.linklayer_id.config.**CopyLinkLayerIdOptionHandlerFac**

Bases: *dhcpkit.ipv6.server.handlers.HandlerFactory* (page 121)

Config processing for a handler to echo a LinkLayerIdOption back to the relay

create() → dhcpkit.ipv6.server.extensions.linklayer_id.CopyLinkLayerIdOptionHandler
Create a handler of this class based on the configuration in the config section.

Returns A handler object

dhcpkit.ipv6.server.extensions.map package

Handlers for the options defined in dhcpkit.ipv6.extensions.map

class dhcpkit.ipv6.server.extensions.map.**MapEOptionHandler**(*br_addresses: Iterable*, *rules: Iterable*, *always_send: bool = False*)

Bases: *dhcpkit.ipv6.server.handlers.basic.SimpleOptionHandler* (page 123)

Handler for putting an S46MapEContainerOption in responses

class dhcpkit.ipv6.server.extensions.map.**MapTOptionHandler**(*dmr_prefix: ipaddress.IPv6Network*, *rules: Iterable*, *always_send: bool = False*)

Bases: *dhcpkit.ipv6.server.handlers.basic.SimpleOptionHandler* (page 123)

Handler for putting an S46MapTContainerOption in responses

Submodules

dhcpkit.ipv6.server.extensions.map.config module

Configuration elements for the MAP server option handlers

class dhcpkit.ipv6.server.extensions.map.config.**MapEOptionHandlerFactory**(*section: ZConfig.matcher.SectionVa*

Bases: *dhcpkit.ipv6.server.handlers.HandlerFactory* (page 121)

Create a handler for putting an S46MapEContainerOption in responses

create() → dhcpkit.ipv6.server.extensions.map.MapEOptionHandler
Create a handler for putting an S46MapEContainerOption in responses

Returns A handler object

class dhcpkit.ipv6.server.extensions.map.config.**MapRule**(*section: ZConfig.matcher.SectionValue*)

Bases: *dhcpkit.common.server.config_elements.ConfigElementFactory* (page 42)

Representation of a single MAP rule

a_bits
a bits: offset of the PSID bits

Returns Number of bits

create() → dhcpkit.ipv6.extensions.map.S46RuleOption
Create a MAP rule option based on the configuration.

Returns The mapping rule

ea_len
Calculate the number of Embedded Address bits.

Returns Number of bits

k_bits
k bits: length in bits of the PSID (2^k == sharing_ratio)

Returns Number of bits

m_bits
m bits: number of bits after the PSID (2^m == contiguous ports)

Returns Number of bits

validate_config_section()
Check whether the combination of parameters make sense.

class dhcpkit.ipv6.server.extensions.map.config.**MapTOptionHandlerFactory**(*section: ZConfig.matcher.SectionVa*

Bases: *dhcpkit.ipv6.server.handlers.HandlerFactory* (page 121)

Create a handler for putting an S46MapTContainerOption in responses

create() → dhcpkit.ipv6.server.extensions.map.MapTOptionHandler
Create a handler for putting an S46MapTContainerOption in responses

Returns A handler object

dhcpkit.ipv6.server.extensions.map.config.**power_of_two**(*value: str*) → int
Validate whether this is an integer that is a power of two.

Parameters **value** – The config string

Returns The integer value

dhcpkit.ipv6.server.extensions.ntp package

Handlers for the options defined in dhcpkit.ipv6.extensions.ntp

class dhcpkit.ipv6.server.extensions.ntp.**NTPServersOptionHandler**(*sub_options: Iterable, always_send: bool = False*)

Bases: *dhcpkit.ipv6.server.handlers.basic.SimpleOptionHandler* (page 123)

Handler for putting NTPServersOption in responses

combine(*existing_options: Iterable*) → dhcpkit.ipv6.extensions.ntp.NTPServersOption
Combine multiple options into one.

Parameters **existing_options** – The existing options to include NTP servers from

Returns The combined option

Submodules

dhcpkit.ipv6.server.extensions.ntp.config module

Configuration elements for the NTP option handlers

class dhcpkit.ipv6.server.extensions.ntp.config.**NTPServersOptionHandlerFactory**(*section: ZConfig.matcher.S*

Bases: *dhcpkit.ipv6.server.handlers.HandlerFactory* (page 121)

Create the handler for NTP servers.

clean_config_section()
Convert the data to the right types

create() → dhcpkit.ipv6.server.extensions.ntp.NTPServersOptionHandler
Create a handler of this class based on the configuration in the config section.

Returns A handler object

validate_config_section()
Make sure the keys refer to actual NTP sub-options

dhcpkit.ipv6.server.extensions.rate_limit package

Handler to rate limit clients that keep rapidly sending requests.

class dhcpkit.ipv6.server.extensions.rate_limit.**RateLimitHandler**(*key=<function duid_key>*, *rate: int = 5*, *per: int = 30*, *burst: int = None*)

Bases: *dhcpkit.ipv6.server.handlers.Handler* (page 121)

Handler to rate limit clients that keep rapidly sending requests.

The most common reason that clients keep sending requests is when they get an answer they don't like. The best way to slow them down is to just stop responding to them.

pre(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Check the rate of incoming requests from this client and stop processing when a client sends too many requests.

Parameters **bundle** – The transaction bundle

Submodules

dhcpkit.ipv6.server.extensions.rate_limit.config module

Config processing for a handler to rate limit clients

class dhcpkit.ipv6.server.extensions.rate_limit.config.**RateLimitHandlerFactory**(*section: ZConfig.matcher.S*

Bases: *dhcpkit.ipv6.server.handlers.HandlerFactory* (page 121)

Config processing for a handler to rate limit clients

create() → dhcpkit.ipv6.server.extensions.rate_limit.RateLimitHandler
Create a handler of this class based on the configuration in the config section.

Returns A handler object

dhcpkit.ipv6.server.extensions.rate_limit.config.**duration**(*config_duration: str*) → int
Convert the config duration to an integer.

Parameters **config_duration** – The duration as a string

Returns The duration as an integer

dhcpkit.ipv6.server.extensions.rate_limit.config.**key_function**(*key_name: str*) → function
Map from name to key extraction function.

Parameters **key_name** – The name of the function

Returns The specified function

dhcpkit.ipv6.server.extensions.rate_limit.config.**rate**(*configured_rate: str*) → int

Convert the config rate to an integer.

Parameters **configured_rate** – The number of messages as a string

Returns The number of messages as an integer

dhcpkit.ipv6.server.extensions.rate_limit.key_functions module

Functions to extract a key from a transaction bundle

dhcpkit.ipv6.server.extensions.rate_limit.key_functions.**duid_key**(*bundle: dhcpkit.ipv6.server.transaction_bundle* → str

Get the DUID from the request in the transaction bundle

Parameters **bundle** – The transaction bundle

Returns The DUID in hex notation

dhcpkit.ipv6.server.extensions.rate_limit.key_functions.**interface_id_key**(*bundle: dhcpkit.ipv6.server.transac* → str

Get the Interface-ID from the request in the transaction bundle, with a fallback to the DUID if no Interface-ID is found.

Parameters **bundle** – The transaction bundle

Returns The Interface-ID (or DUID) in hex notation

dhcpkit.ipv6.server.extensions.rate_limit.key_functions.**linklayer_id_key**(*bundle: dhcpkit.ipv6.server.transac* → str

Get the LinkLayer-ID from the request in the transaction bundle, with a fallback to the DUID if no LinkLayer-ID is found.

Parameters **bundle** – The transaction bundle

Returns The LinkLayer-ID (or DUID) in hex notation

dhcpkit.ipv6.server.extensions.rate_limit.key_functions.**remote_id_key**(*bundle: dhcpkit.ipv6.server.transaction* → str

Get the Remote-ID from the request in the transaction bundle, with a fallback to the DUID if no Remote-ID is found.

Parameters **bundle** – The transaction bundle

Returns The Remote-ID (or DUID) in hex notation

dhcpkit.ipv6.server.extensions.rate_limit.key_functions.**subscriber_id_key**(*bundle: dhcpkit.ipv6.server.transa* → str

Get the Subscriber-ID from the request in the transaction bundle, with a fallback to the DUID if no Subscriber-ID is found.

Parameters **bundle** – The transaction bundle

Returns The Subscriber-ID (or DUID) in hex notation

dhcpkit.ipv6.server.extensions.rate_limit.manager module

A custom manager that manages the shared rate limit counters

class dhcpkit.ipv6.server.extensions.rate_limit.manager.**RateLimitCounters**(*rate: int*, *per: int*, *burst: int = None*)

Bases: object[87]

Counters for rate limiting of DHCPv6 requests

check_request(*key: str*) → bool

Check whether this request is within limits. This method uses the algorithm described on http://stackoverflow.com/questions/667508/whats-a-good-rate-limiting-algorithm#668327

Parameters **key** – The key for this client

Returns Whether we should allow this

class dhcpkit.ipv6.server.extensions.rate_limit.manager.**RateLimitManager**(*address=None*, *authkey=None*, *serializer='pickle'*, *ctx=None*)

Bases: multiprocessing.managers.BaseManager[88]

A custom manager that manages the shared rate limit counters

RateLimitCounters(**args*, ***kwds*)

start(*initializer=None*, *initargs=()*)

Start the rate limit counter manager

dhcpkit.ipv6.server.extensions.rate_limit.manager.**init_manager_process**(*parent_logger*, *initializer=None*, *initargs=()*)

Migrate the logger of the parent to the child. It will be a queue logger anyway.

Parameters

- **parent_logger** – The logger from the parent
- **initializer** – Optional extra initializer
- **initargs** – Optional initializer arguments

[87] https://docs.python.org/3.4/library/functions.html#object
[88] https://docs.python.org/3.4/library/multiprocessing.html#multiprocessing.managers.BaseManager

dhcpkit.ipv6.server.extensions.remote_id package

Handlers for the options defined in dhcpkit.ipv6.extensions.remote_id

class dhcpkit.ipv6.server.extensions.remote_id.**CopyRemoteIdOptionHandler**
Bases: *dhcpkit.ipv6.server.handlers.basic_relay.CopyRelayOptionHandler* (page 124)

The handler for RemoteIdOptions in relay messages

Submodules

dhcpkit.ipv6.server.extensions.remote_id.config module

Config processing for a handler to echo a RemoteIdOption back to the relay

class dhcpkit.ipv6.server.extensions.remote_id.config.**CopyRemoteIdOptionHandlerFactory**(*s*
f

Bases: *dhcpkit.ipv6.server.handlers.HandlerFactory* (page 121)

Config processing for a handler to echo a RemoteIdOption back to the relay

create() → dhcpkit.ipv6.server.extensions.remote_id.CopyRemoteIdOptionHandler
Create a handler of this class based on the configuration in the config section.

Returns A handler object

dhcpkit.ipv6.server.extensions.sip_servers package

Handlers for the options defined in dhcpkit.ipv6.extensions.sip_serves

class dhcpkit.ipv6.server.extensions.sip_servers.**SIPServersAddressListOptionHandler**(*sip_s*
Iterable, *always_bool* = *False*

Bases: *dhcpkit.ipv6.server.handlers.basic.SimpleOptionHandler* (page 123)

Handler for putting SIPServersAddressListOptions in responses

combine(*existing_options: Iterable*) → dhcpkit.ipv6.extensions.sip_servers.SIPServersAddressListOption
Combine multiple options into one.

Parameters **existing_options** – The existing options to include NTP servers from

Returns The combined option

class dhcpkit.ipv6.server.extensions.sip_servers.**SIPServersDomainNameListOptionHandler**(*a*
I
e
a
a
v
b
=
F

Bases: *dhcpkit.ipv6.server.handlers.basic.SimpleOptionHandler* (page 123)

Handler for putting SIPServersDomainNameListOptions in responses

combine(*existing_options: Iterable*) → dhcpkit.ipv6.extensions.sip_servers.SIPServersDomainNameListOption
Combine multiple options into one.

Parameters **existing_options** – The existing options to include NTP servers from

Returns The combined option

Submodules

dhcpkit.ipv6.server.extensions.sip_servers.config module

Configuration elements for the SIP server option handlers

class dhcpkit.ipv6.server.extensions.sip_servers.config.**SIPServersAddressListOptionHandl**

Bases: *dhcpkit.ipv6.server.handlers.HandlerFactory* (page 121)

Create the handler for SIP servers.

create() → dhcpkit.ipv6.server.extensions.sip_servers.SIPServersAddressListOptionHandler
Create a handler of this class based on the configuration in the config section.

Returns A handler object

class dhcpkit.ipv6.server.extensions.sip_servers.config.**SIPServersDomainNameListOptionHa**

Bases: *dhcpkit.ipv6.server.handlers.HandlerFactory* (page 121)

Create the handler for SIP servers.

create() → dhcpkit.ipv6.server.extensions.sip_servers.SIPServersDomainNameListOptionHandler
Create a handler of this class based on the configuration in the config section.

Returns A handler object

dhcpkit.ipv6.server.extensions.sntp package

Handlers for the options defined in dhcpkit.ipv6.extensions.sntp

class dhcpkit.ipv6.server.extensions.sntp.**SNTPServersOptionHandler**(*sntp_servers: Iterable*, *always_send: bool = False*)

Bases: *dhcpkit.ipv6.server.handlers.basic.SimpleOptionHandler* (page 123)

Handler for putting SNTPServersOptions in responses

combine(*existing_options: Iterable*) → dhcpkit.ipv6.extensions.sntp.SNTPServersOption
Combine multiple options into one.

Parameters **existing_options** – The existing options to include NTP servers from

Returns The combined option

Submodules

dhcpkit.ipv6.server.extensions.sntp.config module

Configuration elements for the dns option handlers

class dhcpkit.ipv6.server.extensions.sntp.config.**SNTPServersOptionHandlerFactory**(*section: ZConfig.matche*

Bases: *dhcpkit.ipv6.server.handlers.HandlerFactory* (page 121)

Create the handler for SNTP servers.

create() → dhcpkit.ipv6.server.extensions.sntp.SNTPServersOptionHandler
Create a handler of this class based on the configuration in the config section.

Returns A handler object

dhcpkit.ipv6.server.extensions.sol_max_rt package

Handlers for the options defined in dhcpkit.ipv6.extensions.sol_max_rt

class dhcpkit.ipv6.server.extensions.sol_max_rt.**InfMaxRTOptionHandler**(*inf_max_rt: int*, *always_send: bool = False*)

Bases: *dhcpkit.ipv6.server.handlers.basic.OverwriteOptionHandler* (page 123)

Handler for putting InfMaxRTOption in responses

class dhcpkit.ipv6.server.extensions.sol_max_rt.**SolMaxRTOptionHandler**(*sol_max_rt: int*, *always_send: bool = False*)

Bases: *dhcpkit.ipv6.server.handlers.basic.OverwriteOptionHandler* (page 123)

Handler for putting SolMaxRTOption in responses

Submodules

dhcpkit.ipv6.server.extensions.sol_max_rt.config module

Configuration elements for the SOL_MAX_RT option handlers

class dhcpkit.ipv6.server.extensions.sol_max_rt.config.**InfMaxRTOptionHandlerFactory**(*sectio ZCon fig.ma*

Bases: *dhcpkit.ipv6.server.handlers.HandlerFactory* (page 121)

Create the handler for the InfMaxRTOption.

create() → dhcpkit.ipv6.server.extensions.sol_max_rt.InfMaxRTOptionHandler
Create a handler of this class based on the configuration in the config section.

Returns A handler object

class dhcpkit.ipv6.server.extensions.sol_max_rt.config.**SolMaxRTOptionHandlerFactory**(*sectio*
ZCon
fig.ma

Bases: *dhcpkit.ipv6.server.handlers.HandlerFactory* (page 121)

Create the handler for the SolMaxRTOption.

create() → dhcpkit.ipv6.server.extensions.sol_max_rt.SolMaxRTOptionHandler
Create a handler of this class based on the configuration in the config section.

Returns A handler object

dhcpkit.ipv6.server.extensions.sol_max_rt.config.**max_rt**(*value: str*) → int
Convert the name of the section to the number of seconds, and validate the range

Parameters **value** – Config section name

Returns Number of seconds

dhcpkit.ipv6.server.extensions.static_assignments package

An extension to get static assignments from CSV files, Shelves or an SQLite database

class dhcpkit.ipv6.server.extensions.static_assignments.**Assignment**(*address*, *pre-*
fix)

Bases: tuple[89]

address
Alias for field number 0

prefix
Alias for field number 1

class dhcpkit.ipv6.server.extensions.static_assignments.**StaticAssignmentHandler**(*address_pr*
int,
ad-
dress_valid
int,
pre-
fix_preferre
int,
pre-
fix_valid_li
int)

Bases: *dhcpkit.ipv6.server.handlers.Handler* (page 121)

An option handler that gives a static address and/or prefix to clients

static find_iana_option_for_address(*options: Iterable*, *address: ipad-*
dress.IPv6Address) → Union
Find an IANAOption that contains the given address

Parameters

- **options** – The list of options to search
- **address** – The address to look for

Returns The matching option, if any

static find_iapd_option_for_prefix(*options: Iterable*, *prefix: ipad-*
dress.IPv6Network) → Union
Find an IAPDOption that contains the given prefix

Parameters

[89] https://docs.python.org/3.4/library/stdtypes.html#tuple

- **options** – The list of options to search
- **prefix** – The prefix to look for

Returns The matching option, if any

get_assignment(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*) → dhcpkit.ipv6.server.extensions.static_assignments.Assignment
Subclasses override this method to determine the assignment for the request in the bundle. This MUST return an Assignment object, even if no addresses are provided in it.

Parameters **bundle** – The transaction bundle

Returns The assignment

handle(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
The handling is so complex that we just delegate the implementation to separate methods.

Parameters **bundle** – The transaction bundle

handle_confirm(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Handle a client requesting confirmation

Parameters **bundle** – The request bundle

handle_release_decline(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Handle a client releasing or declining resources. Doesn't really need to do anything because assignments are static. Just mark the right options as handled.

Parameters **bundle** – The request bundle

handle_renew_rebind(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Handle a client renewing/rebinding addresses

Parameters **bundle** – The request bundle

handle_request(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Handle a client requesting addresses (also handles SolicitMessage)

Parameters **bundle** – The request bundle

Submodules

dhcpkit.ipv6.server.extensions.static_assignments.config module

Configuration elements for the static assignment handlers

class dhcpkit.ipv6.server.extensions.static_assignments.config.**CSVStaticAssignmentHandle**

Bases: *dhcpkit.ipv6.server.handlers.HandlerFactory* (page 121)

Factory for a handler that reads assignments from a CSV file

create() → dhcpkit.ipv6.server.extensions.static_assignments.csv.CSVStaticAssignmentHandler
Create a handler of this class based on the configuration in the config section.

Returns A handler object

static name_datatype(*v*)

class dhcpkit.ipv6.server.extensions.static_assignments.config.**SqliteStaticAssignmentHar**

Bases: *dhcpkit.ipv6.server.handlers.HandlerFactory* (page 121)

Factory for a handler that reads assignments from a SQLite database

create() → dhcpkit.ipv6.server.extensions.static_assignments.sqlite.SqliteStaticAssignmentHandler
Create a handler of this class based on the configuration in the config section.

Returns A handler object

static name_datatype(*v*)

dhcpkit.ipv6.server.extensions.static_assignments.csv module

An option handler that assigns addresses based on DUID from a CSV file

class dhcpkit.ipv6.server.extensions.static_assignments.csv.**CSVStaticAssignmentHandler**(*f*
s
a
a
i
a
a
i
p
f
i
p
f
i

Bases: *dhcpkit.ipv6.server.extensions.static_assignments.StaticAssignmentHandler* (page 108)

Assign addresses and/or prefixes based on the contents of a CSV file

get_assignment(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*) → dhcpkit.ipv6.server.extensions.static_assignments.Assignment
Look up the assignment based on DUID, Interface-ID of the relay closest to the client and Remote-ID of the relay closest to the client, in that order.

Parameters **bundle** – The transaction bundle

Returns The assignment, if any

static parse_csv_file(*csv_filename: str*) → List
Read the assignments from the file specified in the configuration

Parameters **csv_filename** – The filename of the CSV file

Returns An list of identifiers and their assignment

read_csv_file(*csv_filename: str*) → Mapping
Read the assignments from the file specified in the configuration

Parameters **csv_filename** – The filename of the CSV file

Returns A dictionary mapping identifiers to assignments

dhcpkit.ipv6.server.extensions.static_assignments.sqlite module

An option handler that assigns addresses based on DUID from a SQLite database

class dhcpkit.ipv6.server.extensions.static_assignments.sqlite.**SqliteStaticAssignmentHar**

Bases: *dhcpkit.ipv6.server.extensions.static_assignments.StaticAssignmentHandler* (page 108)

Assign addresses and/or prefixes based on the contents of a Shelf file

get_assignment(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*) → dhcpkit.ipv6.server.extensions.static_assignments.Assignment

Look up the assignment based on DUID, Interface-ID of the relay closest to the client and Remote-ID of the relay closest to the client, in that order.

Parameters **bundle** – The transaction bundle

Returns The assignment, if any

worker_init()

Open the SQLite database in each worker

dhcpkit.ipv6.server.extensions.static_assignments.sqlite.**build_sqlite**() → int

Function to be called from the command line to convert a CSV based assignments file to a sqlite database.
:return: exit code

dhcpkit.ipv6.server.extensions.subscriber_id package

Handlers for the options defined in dhcpkit.ipv6.extensions.subscriber_id

class dhcpkit.ipv6.server.extensions.subscriber_id.**CopySubscriberIdOptionHandler**

Bases: *dhcpkit.ipv6.server.handlers.basic_relay.CopyRelayOptionHandler* (page 124)

The handler for SubscriberIdOptions in relay messages

Submodules

dhcpkit.ipv6.server.extensions.subscriber_id.config module

Config processing for a handler to echo a SubscriberIdOption back to the relay

class dhcpkit.ipv6.server.extensions.subscriber_id.config.**CopySubscriberIdOptionHandlerF**

Bases: *dhcpkit.ipv6.server.handlers.HandlerFactory* (page 121)

Config processing for a handler to echo a SubscriberIdOption back to the relay

create() → dhcpkit.ipv6.server.extensions.subscriber_id.CopySubscriberIdOptionHandler

Create a handler of this class based on the configuration in the config section.

Returns A handler object

dhcpkit.ipv6.server.extensions.timing_limits package

Handlers that limit the t1/t2 values in replies

class dhcpkit.ipv6.server.extensions.timing_limits.**IANATimingLimitsHandler**(*min_t1: int = 0*, *max_t1: int = 4294967295*, *factor_t1: Union = 0.5*, *min_t2: int = 0*, *max_t2: int = 4294967295*, *factor_t2: Union = 0.8*)

Bases: *dhcpkit.ipv6.server.extensions.timing_limits.TimingLimitsHandler* (page 113)

A handler that limits the t1/t2 values in an IANAOption

static extract_preferred_lifetime(*option: dhcpkit.ipv6.options.Option*) → Union

Extract the preferred lifetime from the given (sub)option. Returns None if this option doesn't contain a preferred lifetime.

Parameters **option** – The option to extract the preferred lifetime from

Returns The preferred lifetime, if any

static filter_options(*options: Iterable*) → List

Extract the IANAOptions that we want to set the t1/t2 values of.

Parameters **options** – The options in the response message

Returns The relevant options of the response message

Return type list[90][*IANAOption* (page 163)]

[90] https://docs.python.org/3.4/library/stdtypes.html#list

class dhcpkit.ipv6.server.extensions.timing_limits.**IAPDTimingLimitsHandler**(*min_t1: int = 0*, *max_t1: int = 4294967295*, *factor_t1: Union = 0.5*, *min_t2: int = 0*, *max_t2: int = 4294967295*, *factor_t2: Union = 0.8*)

Bases: *dhcpkit.ipv6.server.extensions.timing_limits.TimingLimitsHandler* (page 113)

A handler that limits the t1/t2 values in an IANAOption

static extract_preferred_lifetime(*option: dhcpkit.ipv6.options.Option*) → Union

Extract the preferred lifetime from the given (sub)option. Returns None if this option doesn't contain a preferred lifetime.

Parameters **option** – The option to extract the preferred lifetime from

Returns The preferred lifetime, if any

static filter_options(*options: Iterable*) → List

Extract the IAPDOptions that we want to set the t1/t2 values of.

Parameters **options** – The options in the response message

Returns The relevant options of the response message

Return type list[91][*IAPDOption* (page 75)]

[91] https://docs.python.org/3.4/library/stdtypes.html#list

class dhcpkit.ipv6.server.extensions.timing_limits.**TimingLimitsHandler**(*min_t1: int = 0, max_t1: int = 4294967295, factor_t1: Union = 0.5, min_t2: int = 0, max_t2: int = 4294967295, factor_t2: Union = 0.8*)

Bases: *dhcpkit.ipv6.server.handlers.Handler* (page 121)

A handler that limits the t1/t2 values in an option

static extract_preferred_lifetime(*option: dhcpkit.ipv6.options.Option*) → Union
Extract the preferred lifetime from the given (sub)option. Returns None if this option doesn't contain a preferred lifetime.

Parameters **option** – The option to extract the preferred lifetime from

Returns The preferred lifetime, if any

static filter_options(*options: Iterable*) → List
Extract the options that we want to set the t1/t2 values of.

Parameters **options** – The options in the response message

Returns The relevant options of the response message

Return type list[92][*IANAOption* (page 163)]

handle(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Make sure the T1/T2 values are within the set limits.

Parameters **bundle** – The transaction bundle

Submodules

dhcpkit.ipv6.server.extensions.timing_limits.config module

Configuration elements for the IANA/IAPD timing limits.

[92] https://docs.python.org/3.4/library/stdtypes.html#list

class dhcpkit.ipv6.server.extensions.timing_limits.config.**IANATimingLimitsHandlerFactory**

Bases: *dhcpkit.ipv6.server.handlers.HandlerFactory* (page 121)

Create the IANATimingLimitsHandler.

create() → dhcpkit.ipv6.server.extensions.timing_limits.IANATimingLimitsHandler
Create a handler of this class based on the configuration in the config section.

Returns A handler object

validate_config_section()
Check if all the values are valid in combination

class dhcpkit.ipv6.server.extensions.timing_limits.config.**IAPDTimingLimitsHandlerFactory**

Bases: *dhcpkit.ipv6.server.handlers.HandlerFactory* (page 121)

Create the IAPDTimingLimitsHandler.

create() → dhcpkit.ipv6.server.extensions.timing_limits.IAPDTimingLimitsHandler
Create a handler of this class based on the configuration in the config section.

Returns A handler object

validate_config_section()
Check if all the values are valid in combination

dhcpkit.ipv6.server.extensions.timing_limits.config.**factor_value**(*value: str*) → Union
Cast the string NONE to the None value, otherwise convert to a float

Parameters **value** – The string to parse

Returns The float value or None

dhcpkit.ipv6.server.extensions.timing_limits.config.**time_value**(*value: str*) → int
Cast the string INFINITY to the infinity value, otherwise convert to an integer

Parameters **value** – The string to parse

Returns The integer value

Submodules

dhcpkit.ipv6.server.extensions.bulk_leasequery module

Server extension to handle bulk leasequery properly

class dhcpkit.ipv6.server.extensions.bulk_leasequery.**RefuseBulkLeasequeryOverUDPHandler**
Bases: *dhcpkit.ipv6.server.handlers.Handler* (page 121)

A handler that refuses bulk leasequery over UDP.

The new queries introduced in this specification cannot be used with the UDP Leasequery protocol. Servers that implement this specification and also permit UDP queries MUST NOT accept Bulk Leasequery query-types in UDP Leasequery messages. Such servers MUST respond with an error status code of STATUS_NOT_ALLOWED.

pre(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Make sure that bulk leasequery options are not coming in over UDP.

Parameters **bundle** – The transaction bundle

class dhcpkit.ipv6.server.extensions.bulk_leasequery.**RequireBulkLeasequeryOverTCPHandler**
Bases: *dhcpkit.ipv6.server.handlers.Handler* (page 121)

A handler that makes sure only bulk leasequery is accepted over TCP.

Only LEASEQUERY, LEASEQUERY-REPLY, LEASEQUERY-DATA, and LEASEQUERY-DONE messages are allowed over the Bulk Leasequery connection. No other DHCPv6 messages are supported. The Bulk Leasequery connection is not an alternative DHCPv6 communication option for clients seeking DHCPv6 service.

pre(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Make sure that bulk leasequery options are not coming in over UDP.

Parameters **bundle** – The transaction bundle

dhcpkit.ipv6.server.extensions.bulk_leasequery.**create_setup_handlers**() → List
Create handlers to clean up stuff in the transaction bundle

Returns Handlers to add to the handler chain

dhcpkit.ipv6.server.extensions.prefix_delegation module

Server extension to handle prefix delegation options properly

class dhcpkit.ipv6.server.extensions.prefix_delegation.**UnansweredIAPDOptionHandler**(*authoritative: bool = True*)
Bases: *dhcpkit.ipv6.server.handlers.Handler* (page 121)

A handler that answers to all unanswered IAPDOptions

Parameters **authoritative** – Whether this handler is authorised to tell clients to stop using prefixes

handle(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Make sure that every *IAPDOption* (page 75) is answered.

Parameters **bundle** – The transaction bundle

dhcpkit.ipv6.server.extensions.prefix_delegation.**create_cleanup_handlers**() → List
Create handlers to clean up stuff in the transaction bundle

Returns Handlers to add to the handler chain

dhcpkit.ipv6.server.extensions.relay_echo_request module

Implementation of Echo Request option handling as specified in **RFC 4994**[93].

class dhcpkit.ipv6.server.extensions.relay_echo_request.**RelayEchoRequestOptionHandler**
Bases: *dhcpkit.ipv6.server.handlers.RelayHandler* (page 121)

When a server creates a Relay-Reply, it SHOULD perform ERO processing after processing the ORO and other options processing. For each option in the ERO:

1. If the option is already in the Relay-Reply, the server MUST ignore that option and continue to process any remaining options in the ERO.
2. If the option was not in the received Relay-Forward, the server MUST ignore that option and continue to process any remaining options in the ERO.

[93] https://tools.ietf.org/html/rfc4994.html

3. Otherwise, the server MUST copy the option, verbatim, from the received Relay-Forward to the Relay-Reply, even if the server does not otherwise recognize that option.

handle_relay(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*, *relay_message_in: dhcpkit.ipv6.messages.RelayForwardMessage*, *relay_message_out: dhcpkit.ipv6.messages.RelayReplyMessage*)
Handle the options for each relay message pair.

Parameters

- **bundle** – The transaction bundle
- **relay_message_in** – The incoming relay message
- **relay_message_out** – Thr outgoing relay message

dhcpkit.ipv6.server.extensions.relay_echo_request.**create_cleanup_handlers**() → List
Create handlers to clean up stuff in the transaction bundle

Returns Handlers to add to the handler chain

dhcpkit.ipv6.server.filters package

Filters to apply to transaction bundles

class dhcpkit.ipv6.server.filters.**Filter**(*filter_condition: object*, *sub_filters: Iterable = None*, *sub_handlers: Iterable = None*)
Bases: object[94]

Base class for filters

filter_description
A short description of this filter for log messages.

Returns The description

get_handlers(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*) → List
Get all handlers that are going to be applied to the request in the bundle.

Parameters **bundle** – The transaction bundle

Returns The list of handlers to apply

match(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*) → bool
Check whether the given message matches our filter condition.

Parameters **bundle** – The transaction bundle

Returns Whether our filter condition matches

worker_init()
Separate initialisation that will be called in each worker process that is created. Things that can't be forked (think database connections etc) have to be initialised here.

class dhcpkit.ipv6.server.filters.**FilterFactory**(*section: ZConfig.matcher.SectionValue*)
Bases: *dhcpkit.common.server.config_elements.ConfigElementFactory* (page 42)

Base class for filter factories

create()
Create the filter and feed it with the sub-filters and sub-handlers.

Returns The filter

[94] https://docs.python.org/3.4/library/functions.html#object

filter_class
Get the class of filter to create

Returns The class of filter

filter_condition
Return the filter condition, the name of the section by default :return: The filter condition

Subpackages

dhcpkit.ipv6.server.filters.elapsed_time package

Filtering on the elapsed time field in the request

Submodules

dhcpkit.ipv6.server.filters.elapsed_time.config module

Filter on elapsed time indicated by the client

class dhcpkit.ipv6.server.filters.elapsed_time.config.**ElapsedTimeFilter**(*filter_condition: object*, *sub_filters: Iterable = None*, *sub_handlers: Iterable = None*)

Bases: *dhcpkit.ipv6.server.filters.Filter* (page 117)

Filter on marks that have been placed on the incoming message

filter_description
A short description of this filter for log messages.

Returns The description

match(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*) → bool
Check if the elapsed time is within the configured limits

Parameters **bundle** – The transaction bundle

Returns Whether the elapsed time is within the limits

class dhcpkit.ipv6.server.filters.elapsed_time.config.**ElapsedTimeFilterFactory**(*section: ZConfig.matcher.S*

Bases: *dhcpkit.ipv6.server.filters.FilterFactory* (page 117)

Create a MarkedWithFilter

filter_class
alias of *ElapsedTimeFilter* (page 118)

filter_condition
The filter condition is based on the configured time limits.

Returns A list of time limits

validate_config_section()
Check that at least one filter condition is provided, and that if multiple conditions are provided they are compatible with each other.

class dhcpkit.ipv6.server.filters.elapsed_time.config.**TimeLimit**(*operator*, *limit*)

Bases: tuple[95]

limit
Alias for field number 1

operator
Alias for field number 0

dhcpkit.ipv6.server.filters.marks package

Filtering on marks

Submodules

dhcpkit.ipv6.server.filters.marks.config module

Filter on marks that have been placed on the incoming message

class dhcpkit.ipv6.server.filters.marks.config.**MarkedWithFilter**(*filter_condition: object*, *sub_filters: Iterable = None*, *sub_handlers: Iterable = None*)

Bases: *dhcpkit.ipv6.server.filters.Filter* (page 117)

Filter on marks that have been placed on the incoming message

match(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*) → bool
Check if the configured mark is in the set

Parameters **bundle** – The transaction bundle

Returns Whether the configured mark is present

class dhcpkit.ipv6.server.filters.marks.config.**MarkedWithFilterFactory**(*section: ZConfig.matcher.SectionValue*)

Bases: *dhcpkit.ipv6.server.filters.FilterFactory* (page 117)

Create a MarkedWithFilter

filter_class
alias of *MarkedWithFilter* (page 119)

static name_datatype(*value*)

[95] https://docs.python.org/3.4/library/stdtypes.html#tuple

dhcpkit.ipv6.server.filters.subnets package

Filtering on link-address subnet

Submodules

dhcpkit.ipv6.server.filters.subnets.config module

Filter on subnet that the link address is in

class dhcpkit.ipv6.server.filters.subnets.config.**SubnetFilter**(*filter_condition: object*, *sub_filters: Iterable = None*, *sub_handlers: Iterable = None*)

Bases: *dhcpkit.ipv6.server.filters.Filter* (page 117)

Filter on subnet that the link address is in

filter_description
A short description of this filter for log messages.

Returns The description

match(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*) → bool
Check if the link-address is in the subnet

Parameters **bundle** – The transaction bundle

Returns Whether the link-address matches

class dhcpkit.ipv6.server.filters.subnets.config.**SubnetFilterFactory**(*section: ZConfig.matcher.SectionValue*)

Bases: *dhcpkit.ipv6.server.filters.FilterFactory* (page 117)

Create a subnet filter

filter_class
alias of *SubnetFilter* (page 120)

filter_condition
Return the filter condition, the list of prefixes :return: The filter condition

static name_datatype(*value*)

class dhcpkit.ipv6.server.filters.subnets.config.**SubnetGroupFilterFactory**(*section: ZConfig.matcher.SectionV*

Bases: *dhcpkit.ipv6.server.filters.FilterFactory* (page 117)

Create a subnet filter

filter_class
alias of *SubnetFilter* (page 120)

filter_condition
Return the filter condition, the list of prefixes :return: The filter condition

dhcpkit.ipv6.server.handlers package

Handlers to apply to transaction bundles

exception dhcpkit.ipv6.server.handlers.**CannotRespondError**
Bases: *dhcpkit.ipv6.server.handlers.HandlerException* (page 121)

This exception signals that we cannot reply to this client.

class dhcpkit.ipv6.server.handlers.**Handler**
Bases: object[96]

Base class for handlers

analyse_post(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Analyse the response that is going out after all handlers have been applied.

Parameters **bundle** – The transaction bundle

analyse_pre(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Analyse the request that came in before handlers can change it.

Parameters **bundle** – The transaction bundle

handle(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Handle the data in the bundle. Subclasses should do their main work here.

Parameters **bundle** – The transaction bundle

post(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Post-process the data in the bundle. Subclasses can e.g. clean up state. Subclasses assigning addresses should check whether the bundle.response is an AdvertiseMessage or a ReplyMessage. The class can change between handle() and post() when the server is using rapid-commit.

Parameters **bundle** – The transaction bundle

pre(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Pre-process the data in the bundle. Subclasses can update bundle state here or abort processing of the request by raising a CannotRespondError.

Parameters **bundle** – The transaction bundle

worker_init()
The __init__ method will be called in the master process. After initialisation the master process will create worker processes using the multiprocessing module. Things that can't be pickled and transmitted to the worker processes (think database connections etc) have to be initialised separately. Each worker process will call worker_init() to do so. Filters that don't need per-worker initialisation can do everything in __init__().

exception dhcpkit.ipv6.server.handlers.**HandlerException**
Bases: Exception[97]

Base class for handler exceptions

class dhcpkit.ipv6.server.handlers.**HandlerFactory**(*section: ZConfig.matcher.SectionValue*)
Bases: *dhcpkit.common.server.config_elements.ConfigElementFactory* (page 42)

Base class for handler factories

class dhcpkit.ipv6.server.handlers.**RelayHandler**
Bases: *dhcpkit.ipv6.server.handlers.Handler* (page 121)

A base class for handlers that work on option in the relay messages chain.

[96] https://docs.python.org/3.4/library/functions.html#object
[97] https://docs.python.org/3.4/library/exceptions.html#Exception

handle(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Handle the data in the bundle by checking the relay chain and calling *handle_relay()* (page 122) for each relay message.

Parameters **bundle** – The transaction bundle

handle_relay(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*, *relay_message_in: dhcpkit.ipv6.messages.RelayForwardMessage*, *relay_message_out: dhcpkit.ipv6.messages.RelayReplyMessage*)
Handle the options for each relay message pair.

Parameters

- **bundle** – The transaction bundle
- **relay_message_in** – The incoming relay message
- **relay_message_out** – Thr outgoing relay message

exception dhcpkit.ipv6.server.handlers.**ReplyWithLeasequeryError**(*status_code: int = 0*, *status_message: str = ''*)
Bases: *dhcpkit.ipv6.server.handlers.ReplyWithStatusError* (page 122)

This exception signals a leasequery error to the client.

error_description = 'Leasequery error'

exception dhcpkit.ipv6.server.handlers.**ReplyWithStatusError**(*status_code: int = 0*, *status_message: str = ''*)
Bases: *dhcpkit.ipv6.server.handlers.HandlerException* (page 121)

This exception signals an error to the client.

error_description = 'Error'

exception dhcpkit.ipv6.server.handlers.**UseMulticastError**
Bases: *dhcpkit.ipv6.server.handlers.HandlerException* (page 121)

This exception signals that a STATUS_USE_MULTICAST should be returned to the client.

Submodules

dhcpkit.ipv6.server.handlers.basic module

Basic handlers for options

class dhcpkit.ipv6.server.handlers.basic.**CopyOptionHandler**(*option_class: Type*, ***, *always_send: bool = False*)
Bases: *dhcpkit.ipv6.server.handlers.Handler* (page 121)

This handler just copies a type of option from the request to the response

Parameters

- **option_class** – The option class to copy
- **always_send** – Always send this option, even if the OptionRequestOption doesn't ask for it

always_send = None
Whether an *OptionRequestOption* (page 169) in the request should be ignored

handle(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Copy the option from the request to the response.

Parameters **bundle** – The transaction bundle

option_class = None
The class of the option from the request to the response

class dhcpkit.ipv6.server.handlers.basic.**OverwriteOptionHandler**(*option: dhcpkit.ipv6.options.Option*, *, *always_send: bool = False*)

Bases: *dhcpkit.ipv6.server.handlers.Handler* (page 121)

Overwriting handler for simple static options.

Parameters

- **option** – The option instance to use
- **always_send** – Always send this option, even if the OptionRequestOption doesn't ask for it

always_send = None
Whether an *OptionRequestOption* (page 169) in the request should be ignored

handle(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Overwrite the option in the response in the bundle.

Parameters **bundle** – The transaction bundle

option = None
The option to add to the response

option_class = None
The class of the option

class dhcpkit.ipv6.server.handlers.basic.**SimpleOptionHandler**(*option: dhcpkit.ipv6.options.Option*, *, *append: bool = False*, *always_send: bool = False*)

Bases: *dhcpkit.ipv6.server.handlers.Handler* (page 121)

Standard handler for simple static options

Parameters

- **option** – The option instance to add to the response
- **append** – Always add, even if an option of this class already exists
- **always_send** – Always send this option, even if the OptionRequestOption doesn't ask for it

always_send = None
Always send this option, even if the *OptionRequestOption* (page 169) doesn't ask for it

append = None
Always add, even if an option of this class already exists

combine(*existing_options: Iterable*) → Union
If an option of this type already exists this method can combine the existing option with our own option to create a combined option.

Parameters **existing_options** – The existing options

Returns The combined option which will replace all existing options, or None to leave the existing options

handle (*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Add the option to the response in the bundle.

Parameters **bundle** – The transaction bundle

option = None
The option instance to add to the response

option_class = None
The class of the option

dhcpkit.ipv6.server.handlers.basic_relay module

Basic handlers for relay options

class dhcpkit.ipv6.server.handlers.basic_relay.**CopyRelayOptionHandler**(*option_class: type*)
Bases: *dhcpkit.ipv6.server.handlers.RelayHandler* (page 121)

This handler just copies a type of option from the incoming relay messages to the outgoing relay messages

Parameters **option_class** – The option class to copy

handle_relay (*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle, relay_message_in: dhcpkit.ipv6.messages.RelayForwardMessage, relay_message_out: dhcpkit.ipv6.messages.RelayReplyMessage*)
Copy the options for each relay message pair.

Parameters

- **bundle** – The transaction bundle
- **relay_message_in** – The incoming relay message
- **relay_message_out** – Thr outgoing relay message

option_class = None
The class of the option from the *RelayForwardMessage* (page 155) to the *RelayReplyMessage* (page 155)

dhcpkit.ipv6.server.handlers.client_id module

Handlers for the basic **RFC 3315**[98] options

class dhcpkit.ipv6.server.handlers.client_id.**ClientIdHandler**
Bases: *dhcpkit.ipv6.server.handlers.basic.CopyOptionHandler* (page 122)

The handler for ClientIdOptions

dhcpkit.ipv6.server.handlers.ignore module

A simple handler that tells the server to ignore the request.

class dhcpkit.ipv6.server.handlers.ignore.**IgnoreRequestHandler**(*message_types: Iterable = None*)
Bases: *dhcpkit.ipv6.server.handlers.Handler* (page 121)

A simple handler that tells the server to stop processing the request and ignore it

[98] https://tools.ietf.org/html/rfc3315.html

pre(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Stop processing

Parameters **bundle** – The transaction bundle

class dhcpkit.ipv6.server.handlers.ignore.**IgnoreRequestHandlerFactory**(*section: ZConfig.matcher.SectionValue*)

Bases: *dhcpkit.ipv6.server.handlers.HandlerFactory* (page 121)

Create an IgnoreRequestHandler

create() → dhcpkit.ipv6.server.handlers.Handler
Create an IgnoreRequestHandler

dhcpkit.ipv6.server.handlers.interface_id module

Option handlers for the basic **RFC 3315**[99] options

class dhcpkit.ipv6.server.handlers.interface_id.**InterfaceIdOptionHandler**
Bases: *dhcpkit.ipv6.server.handlers.basic_relay.CopyRelayOptionHandler* (page 124)

The handler for InterfaceIdOptions in relay messages

dhcpkit.ipv6.server.handlers.preference module

Option handlers that inserts a PreferenceOption in replies

class dhcpkit.ipv6.server.handlers.preference.**PreferenceOptionHandler**(*preference: int*)

Bases: *dhcpkit.ipv6.server.handlers.basic.SimpleOptionHandler* (page 123)

The handler for PreferenceOption which adds a preference option to appropriate responses

class dhcpkit.ipv6.server.handlers.preference.**PreferenceOptionHandlerFactory**(*section: ZConfig.matcher.Sect*

Bases: *dhcpkit.ipv6.server.handlers.HandlerFactory* (page 121)

Create an IgnoreRequestHandler

create() → dhcpkit.ipv6.server.handlers.preference.PreferenceOptionHandler
Create an IgnoreRequestHandler

dhcpkit.ipv6.server.handlers.rapid_commit module

Handler that implements rapid-commit on the server.

class dhcpkit.ipv6.server.handlers.rapid_commit.**RapidCommitHandler**(*rapid_commit_rejections: bool*)

Bases: *dhcpkit.ipv6.server.handlers.Handler* (page 121)

Upgrade AdvertiseMessage to ReplyMessage when client asks for rapid-commit

handle(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Don't do anything, all the processing happens in *post()* (page 125).

Parameters **bundle** – The transaction bundle

post(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Upgrade the response from a AdvertiseMessage to a ReplyMessage if appropriate :param bundle: The transaction bundle

[99] https://tools.ietf.org/html/rfc3315.html

rapid_commit_rejections = None
Do rapid-commit when an IA_NA, IA_TA or IA_PD request gets refused. We have seen at lease one vice (Fritz!Box) that gets confused when a rapid-commit message tells it there are no addresses available. Turning this setting off works around that problem by not doing a rapid-commit when something gets refused.

dhcpkit.ipv6.server.handlers.server_id module

Handlers for the basic **RFC 3315**[100] options

exception dhcpkit.ipv6.server.handlers.server_id.**ForOtherServerError**
Bases: *dhcpkit.ipv6.server.handlers.CannotRespondError* (page 121)

A specific case of being unable to respond: this message is for another server

class dhcpkit.ipv6.server.handlers.server_id.**ServerIdHandler**(*duid: dhcpkit.ipv6.duids.DUID*)
Bases: *dhcpkit.ipv6.server.handlers.basic.OverwriteOptionHandler* (page 123)

The handler for ServerIdOption. Checks whether any server-id in the request matches our own and puts our server-id in the response message to let the client know who is answering.

option = None

pre(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Check if there is a ServerId in the request

Parameters **bundle** – The transaction bundle

dhcpkit.ipv6.server.handlers.status_option module

Some messages need a status code in the response. These handlers insert that status code if no other handler did.

class dhcpkit.ipv6.server.handlers.status_option.**AddMissingStatusOptionHandler**
Bases: *dhcpkit.ipv6.server.handlers.Handler* (page 121)

The handler that makes sure that replies to confirm messages have a status code. When we reach the end without any status code being set we assume success. Other option handlers set the status to something else if they cannot confirm their part.

handle(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Update the status of the reply to *ConfirmMessage* (page 153), *ReleaseMessage* (page 157) and *DeclineMessage* (page 154).

Parameters **bundle** – The transaction bundle

dhcpkit.ipv6.server.handlers.unanswered_ia module

Option handlers that cleans up unanswered requests

class dhcpkit.ipv6.server.handlers.unanswered_ia.**UnansweredIAOptionHandler**(*authoritative: bool = True*)
Bases: *dhcpkit.ipv6.server.handlers.Handler* (page 121)

A handler that answers to all unanswered IANAOptions and IATAOptions

Parameters **authoritative** – Whether this handler is authorised to tell clients to stop using prefixes

[100] https://tools.ietf.org/html/rfc3315.html

handle(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Make sure that every *IANAOption* (page 163) and *IATAOption* (page 165) is answered.

Parameters **bundle** – The transaction bundle

dhcpkit.ipv6.server.handlers.unicast module

A simple handler that tells the client to use multicast to reach this server.

class dhcpkit.ipv6.server.handlers.unicast.**RejectUnwantedUnicastHandler**
Bases: *dhcpkit.ipv6.server.handlers.Handler* (page 121)

A simple handler that tells the client to use multicast to reach this server.

pre(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Reject unicast messages

Parameters **bundle** – The transaction bundle

class dhcpkit.ipv6.server.handlers.unicast.**ServerUnicastOptionHandler**(*address: ipaddress.IPv6Address*)
Bases: *dhcpkit.ipv6.server.handlers.basic.SimpleOptionHandler* (page 123)

A simple handler that tells the client that it may use unicast to contact this server.

pre(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Set flag to let the server know that unicast is ok, otherwise RejectUnwantedUnicastHandler will reject it later.

Parameters **bundle** – The transaction bundle

class dhcpkit.ipv6.server.handlers.unicast.**ServerUnicastOptionHandlerFactory**(*section: ZConfig.matcher.Sect*
Bases: *dhcpkit.ipv6.server.handlers.HandlerFactory* (page 121)

Create a ServerUnicastOptionHandler

create() → dhcpkit.ipv6.server.handlers.unicast.ServerUnicastOptionHandler
Create a RequireMulticastHandler

dhcpkit.ipv6.server.handlers.utils module

Utility functions for handlers

dhcpkit.ipv6.server.handlers.utils.**force_status**(*options: List, new_status_code: dhcpkit.ipv6.options.StatusCodeOption*)
If there is a StatusCodeOption with a different status code in the options list then replace it. Leave any option with the right status code. Add the given StatusCodeOption if there is none.

Parameters

- **options** – The list of options to manipulate
- **new_status_code** – The wanted StatusCodeOption

dhcpkit.ipv6.server.listeners package

Code to keep the receiving and sending sockets together. When receiving traffic on a link-local multicast address the reply should be sent from a link-local address on the receiving interface. This class makes it easy to keep those together.

exception dhcpkit.ipv6.server.listeners.**ClosedListener**
Bases: *dhcpkit.ipv6.server.listeners.ListeningSocketError* (page 129)

Signal that the socket isn't done receiving yet

exception dhcpkit.ipv6.server.listeners.**IgnoreMessage**
Bases: *dhcpkit.ipv6.server.listeners.ListeningSocketError* (page 129)

Signal that this message should be ignored

class dhcpkit.ipv6.server.listeners.**IncomingPacketBundle**(*, *message_id: str = '??????'*, *data: bytes = b''*, *source_address: ipaddress.IPv6Address = None*, *link_address: ipaddress.IPv6Address = None*, *interface_index: int = -1*, *received_over_multicast: bool = False*, *received_over_tcp: bool = False*, *marks: Iterable = None*, *relay_options: Iterable = None*)

Bases: object[101]

A class that is very efficient to pickle because this is what will be sent to worker processes.

Using a class instead of a namedtuple makes it easier to extend it in the future. To make this possible all properties should have a default value, and the constructor must be called with keyword arguments only.

exception dhcpkit.ipv6.server.listeners.**IncompleteMessage**
Bases: *dhcpkit.ipv6.server.listeners.IgnoreMessage* (page 128)

Signal that the socket isn't done receiving yet

class dhcpkit.ipv6.server.listeners.**Listener**
Bases: object[102]

A class to represent something listening for incoming requests.

fileno() → int
The fileno of the listening socket, so this object can be used by select()

Returns The file descriptor

recv_request() → Tuple
Receive incoming messages

Returns The incoming packet data and a replier object

class dhcpkit.ipv6.server.listeners.**ListenerCreator**
Bases: object[103]

A class to represent something that creates something to listen for incoming requests.

create_listener() → Union
Receive incoming messages

[101] https://docs.python.org/3.4/library/functions.html#object
[102] https://docs.python.org/3.4/library/functions.html#object
[103] https://docs.python.org/3.4/library/functions.html#object

Returns The incoming packet data and a replier object

fileno() → int
The fileno of the listening socket, so this object can be used by select()

Returns The file descriptor

exception dhcpkit.ipv6.server.listeners.**ListenerError**
Bases: Exception[104]

Base class for listener errors

exception dhcpkit.ipv6.server.listeners.**ListeningSocketError**
Bases: *dhcpkit.ipv6.server.listeners.ListenerError* (page 129)

Signal that the listening socket could not be created.

class dhcpkit.ipv6.server.listeners.**Replier**
Bases: object[105]

A class to send replies to the client

can_send_multiple = False

send_reply(*outgoing_message: dhcpkit.ipv6.messages.RelayReplyMessage*) → bool
Send a reply to the client

Parameters **outgoing_message** – The message to send, including a wrapping RelayReplyMessage

Returns Whether sending was successful

dhcpkit.ipv6.server.listeners.**increase_message_counter**()
Increase the message counter and return the new value

Returns The new value of the message counter

Subpackages

dhcpkit.ipv6.server.listeners.multicast_interface package

Implementation of a listener on a local multicast network interface

Submodules

dhcpkit.ipv6.server.listeners.multicast_interface.config module

Implementation of a listener on a local multicast network interface

class dhcpkit.ipv6.server.listeners.multicast_interface.config.**MulticastInterfaceUDPList**

Bases: *dhcpkit.ipv6.server.listeners.factories.UDPListenerFactory* (page 131)

Factory for the implementation of a listener on a local multicast network interface

create(*old_listeners: Iterable = None*) → dhcpkit.ipv6.server.listeners.udp.UDPListener
Create a listener of this class based on the configuration in the config section.

Parameters **old_listeners** – A list of existing listeners in case we can recycle them

Returns A listener object

[104] https://docs.python.org/3.4/library/exceptions.html#Exception
[105] https://docs.python.org/3.4/library/functions.html#object

name_datatype
alias of builtins.str

validate_config_section()
Validate the interface information

dhcpkit.ipv6.server.listeners.unicast package

Factory for the implementation of a listener on a unicast address of a local network interface

Submodules

dhcpkit.ipv6.server.listeners.unicast.config module

Factory for the implementation of a listener on a unicast address of a local network interface

class dhcpkit.ipv6.server.listeners.unicast.config.**UnicastUDPListenerFactory**(*section: ZConfig.matcher.Sect*

Bases: *dhcpkit.ipv6.server.listeners.factories.UDPListenerFactory* (page 131)

Factory for the implementation of a listener on a unicast address of a local network interface

create(*old_listeners: Iterable = None*) → dhcpkit.ipv6.server.listeners.udp.UDPListener
Create a listener of this class based on the configuration in the config section.

Parameters **old_listeners** – A list of existing listeners in case we can recycle them

Returns A listener object

name_datatype
alias of ipaddress.IPv6Address[106]

validate_config_section()
Validate the interface information

dhcpkit.ipv6.server.listeners.unicast_tcp package

Factory for the implementation of a TCP listener on a unicast address of a local network interface

Submodules

dhcpkit.ipv6.server.listeners.unicast_tcp.config module

Factory for the implementation of a TCP listener on a unicast address of a local network interface

class dhcpkit.ipv6.server.listeners.unicast_tcp.config.**UnicastTCPListenerFactory**(*section: ZConfig.matche*

Bases: *dhcpkit.ipv6.server.listeners.factories.TCPListenerFactory* (page 131)

Factory for the implementation of a listener on a unicast address of a local network interface

create(*old_listeners: Iterable = None*) → dhcpkit.ipv6.server.listeners.tcp.TCPConnectionListener
Create a listener of this class based on the configuration in the config section.

Parameters **old_listeners** – A list of existing listeners in case we can recycle them

Returns A listener object

[106] https://docs.python.org/3.4/library/ipaddress.html#ipaddress.IPv6Address

validate_config_section()
Validate the interface information

Submodules

dhcpkit.ipv6.server.listeners.factories module

Factory base classes for listener factories

class dhcpkit.ipv6.server.listeners.factories.**ListenerFactory**(*section: ZConfig.matcher.SectionValue*)
Bases: *dhcpkit.common.server.config_elements.ConfigElementFactory* (page 42)

Base class for listener factories

listen_port = 547

match_socket(*sock: socket.socket*, *address: ipaddress.IPv6Address*, *interface: int = 0*) → bool
Determine if we can recycle this socket

Parameters

- **sock** – An existing socket
- **address** – The address we want
- **interface** – The interface number we want

Returns Whether the socket is suitable

sock_proto = None

sock_type = None

class dhcpkit.ipv6.server.listeners.factories.**TCPListenerFactory**(*section: ZConfig.matcher.SectionValue*)
Bases: *dhcpkit.ipv6.server.listeners.factories.ListenerFactory* (page 131)

Base class for TCP listener factories

sock_proto = 6

sock_type = 1

class dhcpkit.ipv6.server.listeners.factories.**UDPListenerFactory**(*section: ZConfig.matcher.SectionValue*)
Bases: *dhcpkit.ipv6.server.listeners.factories.ListenerFactory* (page 131)

Base class for UDP listener factories

sock_proto = 17

sock_type = 2

dhcpkit.ipv6.server.listeners.tcp module

Code to keep the receiving and sending sockets together. When receiving traffic on a link-local multicast address the reply should be sent from a link-local address on the receiving interface. This class makes it easy to keep those together.

class dhcpkit.ipv6.server.listeners.tcp.**TCPConnection**(*interface_name: str*, *connected_socket: socket.socket*, *write_lock: <bound method BaseContext.Lock of <multiprocessing.context.DefaultContext object at 0x7f3bfa0ad470>>*, *global_address: ipaddress.IPv6Address*, *marks: Iterable = None*)

Bases: *dhcpkit.ipv6.server.listeners.Listener* (page 128)

A TCP connection listener for DHCPv6 messages

fileno() → int
The fileno of the listening socket, so this object can be used by select()

Returns The file descriptor

packet_from_buffer()
Create a packet and replier from the data in the buffer

Returns The incoming packet data and a replier object

recv_data_into_buffer(*amount: int*) → int
Receive data into the buffer and do proper error handling

Parameters **amount** – How much data do we want?

Returns How much data did we receive?

recv_request() → Tuple
Receive incoming messages

Returns The incoming packet data and a replier object

class dhcpkit.ipv6.server.listeners.tcp.**TCPConnectionListener**(*interface_name: str*, *listen_socket: socket.socket*, *global_address: ipaddress.IPv6Address = None*, *marks: Iterable = None*, *max_connections: int = 10*, *allow_from: Iterable = None*)

Bases: *dhcpkit.ipv6.server.listeners.ListenerCreator* (page 128)

Wrapper for a listening TCP socket. This is not a listener in the DHCPKit sense of the concept. DHCPKit listeners receive DHCPv6 messages, which is done on an established connection.

create_listener() → Union
Accept incoming connection

Returns The connection object

fileno() → int
The fileno of the listening socket, so this object can be used by select()

Returns The file descriptor

class dhcpkit.ipv6.server.listeners.tcp.**TCPReplier**(*reply_socket: socket.socket*, *reply_lock: <bound method BaseContext.Lock of <multiprocessing.context.DefaultContext object at 0x7f3bfa0ad470>>*)

Bases: *dhcpkit.ipv6.server.listeners.Replier* (page 129)

A class to send replies to the client

can_send_multiple = True

send_reply(*outgoing_message: dhcpkit.ipv6.messages.RelayReplyMessage*) → bool
Send a reply to the client

Parameters **outgoing_message** – The message to send, including a wrapping RelayReplyMessage

Returns Whether sending was successful

dhcpkit.ipv6.server.listeners.udp module

UDP implementations of listeners and repliers

class dhcpkit.ipv6.server.listeners.udp.**UDPListener**(*interface_name: str*, *listen_socket: socket.socket*, *reply_socket: socket.socket = None*, *global_address: ipaddress.IPv6Address = None*, *marks: Iterable = None*)

Bases: *dhcpkit.ipv6.server.listeners.Listener* (page 128)

A wrapper for a normal socket that bundles a socket to listen on with a (potentially different) socket to send replies from.

fileno() → int
The fileno of the listening socket, so this object can be used by select()

Returns The file descriptor

recv_request() → Tuple
Receive incoming messages

Returns The incoming packet data and a replier object

class dhcpkit.ipv6.server.listeners.udp.**UDPReplier**(*reply_socket: socket.socket*)

Bases: *dhcpkit.ipv6.server.listeners.Replier* (page 129)

A class to send replies to the client

send_reply(*outgoing_message: dhcpkit.ipv6.messages.RelayReplyMessage*) → bool
Send a reply to the client

Parameters **outgoing_message** – The message to send, including a wrapping RelayReplyMessage

Returns Whether sending was successful

Submodules

dhcpkit.ipv6.server.config_datatypes module

Extra datatypes for the server configuration

dhcpkit.ipv6.server.config_datatypes.**message_type**(*value: str*) → Type
Parse the value as the name of a DHCPv6 message type

Parameters **value** – The name of the message type

Returns The message class

dhcpkit.ipv6.server.config_datatypes.**unicast_address**(*value: str*) → ipaddress.IPv6Address
Parse an IPv6 address and make sure it is a unicast address

Parameters **value** – The address as string

Returns The parsed IPv6 address

dhcpkit.ipv6.server.config_elements module

The basic configuration objects

class dhcpkit.ipv6.server.config_elements.**MainConfig**(*section: ZConfig.matcher.SectionValue*)
Bases: *dhcpkit.common.server.config_elements.ConfigSection* (page 42)

The top level configuration element

clean_config_section()
Clean up the config, making sure we have user, group and DUID

create_message_handler() → dhcpkit.ipv6.server.message_handler.MessageHandler
Create a message handler based on this configuration.

Returns The message handler

class dhcpkit.ipv6.server.config_elements.**StatisticsConfig**(*section: ZConfig.matcher.SectionValue*)
Bases: *dhcpkit.common.server.config_elements.ConfigSection* (page 42)

Configuration of the statistics gatherer

dhcpkit.ipv6.server.config_parser module

Configuration file definition and parsing

dhcpkit.ipv6.server.config_parser.**get_config_loader**() → ZConfig.loader.ConfigLoader
Get the config loader with all extensions

Returns The fully extended config loader

dhcpkit.ipv6.server.config_parser.**load_config**(*config_filename: str*) → dhcpkit.ipv6.server.config_elements.MainConfig
Load the given configuration file.

Parameters **config_filename** – The configuration file

Returns The parsed config

dhcpkit.ipv6.server.control_socket module

A socket to control the DHCPKit server

class dhcpkit.ipv6.server.control_socket.**ControlConnection**(*sock: socket.socket*)

Bases: object[107]

A connection of the remote control socket

acknowledge(*feedback: str = None*)
Acknowledge the command

close()
Close the socket nicely

fileno() → int
The fileno of the listening socket, so this object can be used by select()

Returns The file descriptor

get_commands() → List
Receive data until the next newline and return the result

Returns A list of commands

reject()
Reject the command

send(*output: str*, *eol=b'\n'*)
Send data over the socket

Parameters

- **output** – The data to send
- **eol** – The end-of-line character

class dhcpkit.ipv6.server.control_socket.**ControlSocket**(*socket_path: str*)

Bases: object[108]

Remote control of the DHCPKit server

accept() → Union
Accept a new connection

Returns The new connection

close()
Close the socket nicely

fileno() → int
The fileno of the listening socket, so this object can be used by select()

Returns The file descriptor

dhcpkit.ipv6.server.dhcpctl module

The remote control app for the server process

exception dhcpkit.ipv6.server.dhcpctl.**CommunicationError**

Bases: *dhcpkit.ipv6.server.dhcpctl.ControlClientError* (page 135)

There was a problem communicating

[107] https://docs.python.org/3.4/library/functions.html#object
[108] https://docs.python.org/3.4/library/functions.html#object

exception dhcpkit.ipv6.server.dhcpctl.**ControlClientError**
Bases: Exception[109]

Base class for DHCPKit Control Client errors

class dhcpkit.ipv6.server.dhcpctl.**DHCPKitControlClient**(*control_socket: str*)
Bases: object[110]

A class for communicating with a DHCPKit DHCPv6 server

execute_command(*command: str*, *optional: bool = False*) → Iterable
Send a command and parse the response

Parameters

- **command** – The command
- **optional** – Whether we care about this command being properly executed

Returns The output

receive_line(*optional: bool = False*) → Union
Receive one line of output from the server

Parameters **optional** – Whether we care about this command being properly executed

Returns The received line

send_command(*command: str*, *optional: bool = False*)
Send a command to the server

Parameters

- **command** – The command
- **optional** – Whether we care about this command being properly executed

exception dhcpkit.ipv6.server.dhcpctl.**UnknownCommandError**
Bases: *dhcpkit.ipv6.server.dhcpctl.ControlClientError* (page 135)

The server doesn't understand the command we sent

exception dhcpkit.ipv6.server.dhcpctl.**WrongServerError**
Bases: *dhcpkit.ipv6.server.dhcpctl.ControlClientError* (page 135)

The socket we connected to doesn't seem to be a DHCPKit server

dhcpkit.ipv6.server.dhcpctl.**handle_args**(*args: Iterable*)
Handle the command line arguments.

Parameters **args** – Command line arguments

Returns The arguments object

dhcpkit.ipv6.server.dhcpctl.**main**(*args: Iterable*)
The main program loop

Parameters **args** – Command line arguments

Returns The program exit code

dhcpkit.ipv6.server.dhcpctl.**run**() → int
Run the main program and handle exceptions

Returns The program exit code

[109] https://docs.python.org/3.4/library/exceptions.html#Exception
[110] https://docs.python.org/3.4/library/functions.html#object

dhcpkit.ipv6.server.extension_registry module

The server extension registry

class dhcpkit.ipv6.server.extension_registry.**ServerExtensionRegistry**
Bases: *dhcpkit.registry.Registry* (page 220)

Registry for DHCPKit IPv6 Server Extensions

entry_point = 'dhcpkit.ipv6.server.extensions'

dhcpkit.ipv6.server.generate_config_docs module

A script to generate .rst documentation based on the config schema

dhcpkit.ipv6.server.generate_config_docs.**create_file**(*name*, *args*)
Create a file, or a file-like dummy if dry-run is enabled

Parameters

- **name** – The relative file/path name
- **args** – The command like arguments

Returns A file-like object

dhcpkit.ipv6.server.generate_config_docs.**handle_args**(*args: Iterable*)
Handle the command line arguments.

Parameters **args** – Command line arguments

Returns The arguments object

dhcpkit.ipv6.server.generate_config_docs.**heading**(*text: str*, *underline: str*) → str
Create a heading using the specified underline character.

Parameters

- **text** – The text to use as the heading
- **underline** – The character to underline with

Returns The heading in rst format

dhcpkit.ipv6.server.generate_config_docs.**key_doc**(*info: Union*) → List
Generate documentation for a key.

Parameters **info** – The information object for this key

Returns The documentation for that key

dhcpkit.ipv6.server.generate_config_docs.**link_destination**(*name: str*) → str
Create an rst link.

Parameters **name** – The destination to link to

Returns The reStructuredText link

dhcpkit.ipv6.server.generate_config_docs.**link_to**(*text: str*, *link: str = None*) → str
Make the text a reference link.

Parameters

- **text** – The text to link
- **link** – The link destination, if different from the text

Returns The texts as a reference link

dhcpkit.ipv6.server.generate_config_docs.**main**(*args: Iterable*) → int
Generate .rst documentation based on the config schema

Parameters **args** – Command line arguments

Returns Program exit code

dhcpkit.ipv6.server.generate_config_docs.**nicer_type_name**(*name: str*) → str
Make a nicer name for a type.

Parameters **name** – The ugly name

Returns The nicer name

dhcpkit.ipv6.server.generate_config_docs.**normalise_link_name**(*link: str*) → str
Convert i.e. "filter_factory" to "filters"

Parameters **link** – The original link name

Returns The normalised link name

dhcpkit.ipv6.server.generate_config_docs.**reindent**(*text: str*, *new_indent: str = ''*) → str
Fix the indentation.

Parameters

- **text** – The original text with unknown indentation
- **new_indent** – The string to indent with

Returns The text with fixed indentation

dhcpkit.ipv6.server.generate_config_docs.**run**() → int
Run the main program and handle exceptions

Returns The program exit code

dhcpkit.ipv6.server.generate_config_docs.**sectiontype_doc**(*section: ZConfig.info.SectionType*) → List
Extract the documentation for the given section.

Parameters **section** – The section to extract documentation from

Returns A list of strings with documentation

dhcpkit.ipv6.server.generate_config_docs.**write_lines**(*file*, *lines: Iterable*)
Write a set of lines to the file

Parameters

- **file** – The file, or None
- **lines** – The lines to write

dhcpkit.ipv6.server.main module

The main server process

dhcpkit.ipv6.server.main.**create_control_socket**(*args*, *config: dhcpkit.ipv6.server.config_elements.MainConfig*) → dhcpkit.ipv6.server.control_socket.ControlSocket
Create a control socket when configured to do so.

Parameters

- **args** – The command line arguments
- **config** – The server configuration

Returns The name of the created control socket

dhcpkit.ipv6.server.main.**create_pidfile**(*args*, *config: dhcpkit.ipv6.server.config_elements.MainConfig*) → Union

Create a PID file when configured to do so.

Parameters

- **args** – The command line arguments
- **config** – The server configuration

Returns The name of the created PID file

dhcpkit.ipv6.server.main.**error_callback**(*exception*)

Show exceptions that occur while handling messages

Parameters **exception** – The exception that occurred

dhcpkit.ipv6.server.main.**handle_args**(*args: Iterable*)

Handle the command line arguments.

Parameters **args** – Command line arguments

Returns The arguments object

dhcpkit.ipv6.server.main.**main**(*args: Iterable*) → int

The main program loop

Parameters **args** – Command line arguments

Returns The program exit code

dhcpkit.ipv6.server.main.**run**() → int

Run the main program and handle exceptions

Returns The program exit code

dhcpkit.ipv6.server.main.**stop_logging_thread**()

Stop the logging thread from the global

dhcpkit.ipv6.server.message_handler module

The code to handle a message

class dhcpkit.ipv6.server.message_handler.**MessageHandler**(*server_id: dhcpkit.ipv6.duids.DUID*, *sub_filters: Iterable = None*, *sub_handlers: Iterable = None*, *allow_rapid_commit: bool = False*, *rapid_commit_rejections: bool = False*)

Bases: object[111]

Message processing class

construct_leasequery_status_reply(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*, *option: dhcpkit.ipv6.options.StatusCodeOption*) → dhcpkit.ipv6.extensions.leasequery.LeasequeryReplyMessage

Construct a leasequery reply message signalling a status code to the client.

Parameters

- **bundle** – The transaction bundle containing the incoming request

[111] https://docs.python.org/3.4/library/functions.html#object

- **option** – The status code option to include in the reply

Returns A leasequery reply with only the bare necessities and a status code

construct_plain_status_reply(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*, *option: dhcpkit.ipv6.options.StatusCodeOption*) → dhcpkit.ipv6.messages.ReplyMessage

Construct a reply message signalling a status code to the client.

Parameters

- **bundle** – The transaction bundle containing the incoming request
- **option** – The status code option to include in the reply

Returns A reply with only the bare necessities and a status code

construct_use_multicast_reply(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*) → Union

Construct a message signalling to the client that they should have used multicast.

Parameters **bundle** – The transaction bundle containing the incoming request

Returns The proper answer to tell a client to use multicast

static get_cleanup_handlers() → List

Build a list of cleanup handlers and cache it

Returns The list of handlers

get_handlers(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*) → List

Get all handlers that are going to be applied to the request in the bundle.

Parameters **bundle** – The transaction bundle

Returns The list of handlers to apply

get_setup_handlers() → List

Build a list of setup handlers and cache it

Returns The list of handlers

handle(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*, *statistics: dhcpkit.ipv6.server.statistics.StatisticsSet*)

The main dispatcher for incoming messages.

Parameters

- **bundle** – The transaction bundle
- **statistics** – Container for shared memory with statistics counters

static init_response(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)

Create the message object in bundle.response

Parameters **bundle** – The transaction bundle

worker_init()

Separate initialisation that will be called in each worker process that is created. Things that can't be forked (think database connections etc) have to be initialised here.

dhcpkit.ipv6.server.nonblocking_pool module

A multiprocessing pool that doesn't block when full. If we don't do this then the queue fills up with old messages and the workers keep answering those while the client has probably already given up, instead of answering recent messages.

class dhcpkit.ipv6.server.nonblocking_pool.**NonBlockingPool**(*processes=None*, *initializer=None*, *initargs=()*, *maxtasksperchild=None*, *context=None*)

Bases: multiprocessing.pool.Pool[112]

A multiprocessing pool that doesn't block when full

apply_async(*func: Callable*, *args: Tuple = ()*, *kwds: Dict = None*, *callback: Callable = None*, *error_callback: Callable = None*)

Asynchronous version of *apply()* method.

dhcpkit.ipv6.server.pygments_plugin module

Extensions to Pygments to correctly parse DHCPKit config files

class dhcpkit.ipv6.server.pygments_plugin.**DHCPKitConfLexer**(***options*)

Bases: pygments.lexer.RegexLexer

Lexer for configuration files following the DHCPKit config file format.

aliases = ['dhcpkitconf', 'dhcpkit']

flags = 10

name = 'DHCPKitConf'

tokens = {'root': [('\\s+', Token.Text), ('(#.*?)$', Token.Comment), ('(<[^\\s>]+)

dhcpkit.ipv6.server.queue_logger module

Adapt the QueueListener so that it respects the log levels of the handlers. Based on the Python 3.5 implementation.

class dhcpkit.ipv6.server.queue_logger.**QueueLevelListener**(*queue*, **handlers*, *respect_handler_level=False*)

Bases: logging.handlers.QueueListener[113]

QueueListener that respects log levels

addHandler(*handler*)

Add the specified handler to this logger.

dequeue(*block*)

Dequeue a record and return it, optionally blocking. Return the sentinel on EOF because otherwise there are strange errors after a reload.

handle(*record*)

Handle a record.

This just loops through the handlers offering them the record to handle.

removeHandler(*handler*)

Remove the specified handler from this logger.

class dhcpkit.ipv6.server.queue_logger.**WorkerQueueHandler**(*queue: multiprocessing.queues.Queue*)

Bases: logging.handlers.QueueHandler[114]

[112] https://docs.python.org/3.4/library/multiprocessing.html#multiprocessing.pool.Pool
[113] https://docs.python.org/3.4/library/logging.handlers.html#logging.handlers.QueueListener
[114] https://docs.python.org/3.4/library/logging.handlers.html#logging.handlers.QueueHandler

A logging handler that queues messages and doesn't cause exceptions when the queue is full.

enqueue(*record*)
Enqueue a record.

Try three times rapidly, then just drop it.

prepare(*record*)
Prepares a record for queuing. The object returned by this method is enqueued. This implementation adds the log_id if it is set.

dhcpkit.ipv6.server.statistics module

Statistics about the server in shared memory

class dhcpkit.ipv6.server.statistics.**ServerStatistics**
Bases: object[115]

A set of statistics about the DHCPv6 server

export() → Dict
Export the counters

Returns The counters in a processable format

get_update_set(*interface_name: str = None*, *bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle = None*) → dhcpkit.ipv6.server.statistics.StatisticsSet
Return all statistics objects that need to be updated.

Parameters

- **interface_name** – The name of the interface that we received the packet on
- **bundle** – The transaction bundle to base the selection on

Returns The set to call count methods on

set_categories(*category_settings*)
Create space for the given interfaces

Parameters **category_settings** – Configuration setting for categories

class dhcpkit.ipv6.server.statistics.**Statistics**
Bases: object[116]

A set of statistics about DHCPv6

count_do_not_respond()
Call the counting method on all statistics objects

count_for_other_server()
Call the counting method on all statistics objects

count_handling_error()
Call the counting method on all statistics objects

count_incoming_packet()
Call the counting method on all statistics objects

count_malformed_query()
Call the counting method on all statistics objects

[115] https://docs.python.org/3.4/library/functions.html#object
[116] https://docs.python.org/3.4/library/functions.html#object

count_message_in(*key*)
Update the counter for the given key

count_message_out(*key*)
Update the counter for the given key

count_not_allowed()
Call the counting method on all statistics objects

count_other_error()
Call the counting method on all statistics objects

count_outgoing_packet()
Call the counting method on all statistics objects

count_unknown_query_type()
Call the counting method on all statistics objects

count_unparsable_packet()
Call the counting method on all statistics objects

count_use_multicast()
Call the counting method on all statistics objects

export() → Dict
Export the counters

Returns The counters in a processable format

class dhcpkit.ipv6.server.statistics.**StatisticsSet**(*statistics_set: Iterable = None*)

Bases: object[117]

A set of statistics objects that are updated together. The metaclass will create all methods for us.

count_do_not_respond()
Call the counting method on all statistics objects

count_for_other_server()
Call the counting method on all statistics objects

count_handling_error()
Call the counting method on all statistics objects

count_incoming_packet()
Call the counting method on all statistics objects

count_malformed_query()
Call the counting method on all statistics objects

count_message_in(*key*)
Call the counting method on all statistics objects

count_message_out(*key*)
Call the counting method on all statistics objects

count_not_allowed()
Call the counting method on all statistics objects

count_other_error()
Call the counting method on all statistics objects

count_outgoing_packet()
Call the counting method on all statistics objects

count_unknown_query_type()
Call the counting method on all statistics objects

[117] https://docs.python.org/3.4/library/functions.html#object

count_unparsable_packet()
Call the counting method on all statistics objects

count_use_multicast()
Call the counting method on all statistics objects

dhcpkit.ipv6.server.statistics.**create_count_dict_method**(*method_name: str*)
Create a counting method for the StatisticsSet class

Parameters **method_name** – The name of the method to call on Statistics objects

Returns The generated method

dhcpkit.ipv6.server.statistics.**create_count_method**(*method_name: str*)
Create a counting method for the StatisticsSet class

Parameters **method_name** – The name of the method to call on Statistics objects

Returns The generated method

dhcpkit.ipv6.server.statistics.**create_update_dict_method**(*counter_name*)
Create a counting method for a counter in a dictionary on the Statistics class

Parameters **counter_name** – The name of the counter to update

Returns The generated method

dhcpkit.ipv6.server.statistics.**create_update_method**(*counter_name*)
Create a counting method for a simple counter on the Statistics class

Parameters **counter_name** – The name of the counter to update

Returns The generated method

dhcpkit.ipv6.server.transaction_bundle module

An object to hold everything related to a request/response transaction

class dhcpkit.ipv6.server.transaction_bundle.**MessagesList**(*first_message: dhcpkit.ipv6.messages.ClientServerMessage = None, subsequent_messages: Iterator = None*)

Bases: object[118]

A weird iterator wrapper. This allows handlers to manipulate the first message while not needing to load all of the subsequent messages in memory.

class dhcpkit.ipv6.server.transaction_bundle.**TransactionBundle**(*incoming_message: dhcpkit.ipv6.messages.Message, received_over_multicast: bool, received_over_tcp: bool = False, allow_rapid_commit: bool = False, marks: Iterable = None*)

[118] https://docs.python.org/3.4/library/functions.html#object

Bases: object[119]

A bundle with all data about a transaction. This makes it much easier to pass around multiple pieces of information.

add_mark(*mark: str*)
Add this mark to the set.

Parameters **mark** – The mark to add

allow_rapid_commit = None
Allow rapid commit? May be set to True on creation, may be set to False by handlers, not vice versa

allow_unicast = None
Allow the client use unicast to contact the server. Set to True by handlers

create_outgoing_relay_messages()
Create a plain chain of RelayReplyMessages for the current response

get_unhandled_options(*option_types: Type*) → List
Get a list of all Options in the request that haven't been marked as handled

Returns The list of unanswered Options

handled_options = None
A list of options from the request that have been handled, only applies to IA type options

handler_data = None
A place for handlers to store data related to this transaction

incoming_message = None
The incoming message including the relay chain

incoming_relay_messages = None
The chain of relay messages starting with the one closest to the client

link_address
Find the link address that identifies where this request is coming from. For TCP connections we use the remote endpoint of the connection instead.

mark_handled(*option: dhcpkit.ipv6.options.Option*)
Mark the given option as handled. Not all options are specifically handled. This is mostly useful for options like IANAOption, IATAOption and IAPDOption.

Parameters **option** – The option to mark as handled

marks = None
A set of marks that have been applied to this message

outgoing_message
Wrap the response in a relay chain if necessary. Only works when there is a single response.

outgoing_messages
Wrap the responses in a relay chain if necessary and iterate over them.

> **Warning:** Be careful when iterating over outgoing messages. When iterating over multiple responses the original relay messages will be updated to contain the next response when proceeding the the next one!

outgoing_relay_messages = None
This is where the user puts the reply relay chain by calling *create_outgoing_relay_messages()* (page 145)

[119] https://docs.python.org/3.4/library/functions.html#object

received_over_multicast = None
A flag indicating whether the client used multicast to contact the server

received_over_tcp = None
A flag indicating whether the client used TCP to contact the server

relays
Get a list of all the relays that this message went through

request = None
The incoming request without the relay messages

response
Backwards-compatibility handling for when we only supported one response. TCP connections can support more than one response, but for normal DHCPv6 a single response is all we need is a single one, so make this use-case easy and backwards-compatible.

Returns The first response

responses = None
This is where we keep our responses, potentially more than one

dhcpkit.ipv6.server.utils module

Utility functions for the DHCP server

dhcpkit.ipv6.server.utils.**determine_local_duid**() → dhcpkit.ipv6.duids.LinkLayerDUID
Calculate our own DUID based on one of our MAC addresses

Returns The server DUID

dhcpkit.ipv6.server.worker module

Worker process for handling requests using multiprocessing.

dhcpkit.ipv6.server.worker.**current_message_handler = None**

Type MessageHandler

dhcpkit.ipv6.server.worker.**get_interface_name_from_options**(*options: Iterable*)
Get the interface name from the given options and decode it as unicode

Parameters **options** – A list of options

Returns The interface name

dhcpkit.ipv6.server.worker.**handle_message**(*incoming_packet: dhcpkit.ipv6.server.listeners.IncomingPacketBundle*, *replier: dhcpkit.ipv6.server.listeners.Replier*)
Handle a single incoming request. This is supposed to be called in a separate worker thread that has been initialised with setup_worker().

Parameters

- **incoming_packet** – The raw incoming request
- **replier** – The object that will send replies for us

Returns The packet to reply with and the destination

dhcpkit.ipv6.server.worker.**logger = None**

Type logging.Logger

dhcpkit.ipv6.server.worker.**logging_handler = None**

Type WorkerQueueHandler

dhcpkit.ipv6.server.worker.**parse_incoming_request**(*incoming_packet: dhcpkit.ipv6.server.listeners.IncomingPacketBundle*) → dhcpkit.ipv6.server.transaction_bundle.TransactionBundle

Parse the incoming packet and add a RelayServerMessage around it containing the meta-data received from the listener.

Parameters **incoming_packet** – The received packet

Returns The parsed message in a transaction bundle

dhcpkit.ipv6.server.worker.**setup_worker**(*message_handler: dhcpkit.ipv6.server.message_handler.MessageHandler*, *logging_queue: <bound method BaseContext.Queue of <multiprocessing.context.DefaultContext object at 0x7f3bfa0ad470>>*, *lowest_log_level: int*, *statistics: dhcpkit.ipv6.server.statistics.ServerStatistics*, *master_pid: int*)

This function will be called after a new worker process has been created. Its purpose is to set the global variables in this specific worker process so that they can be reused across multiple requests. Otherwise we would have to pickle them each and every time, and because they are static that would be a waste.

Parameters

- **message_handler** – The message handler for the incoming requests
- **logging_queue** – The queue where we can deposit log messages so the main process can log them
- **lowest_log_level** – The lowest log level that is going to be handled by the main process
- **statistics** – Container for shared memory with statistics counters
- **master_pid** – The PID of the master process, in case we have critical errors while initialising

dhcpkit.ipv6.server.worker.**shared_statistics = None**

Type ServerStatistics

dhcpkit.ipv6.server.worker.**verify_response**(*outgoing_message: dhcpkit.ipv6.messages.Message*)

generate the outgoing packet and check the RelayServerMessage around it.

Parameters **outgoing_message** – The reply message

Submodules

dhcpkit.ipv6.duid_registry module

The DUID registry

class dhcpkit.ipv6.duid_registry.**DUIDRegistry**

Bases: *dhcpkit.registry.Registry* (page 220)

Registry for DHCPKit IPv6 DUIDs

entry_point = 'dhcpkit.ipv6.duids'

get_name(*item: object*) → str

Get the name for the by_name mapping.

Parameters **item** – The item to determine the name of

Returns The name to use as key in the mapping

dhcpkit.ipv6.duids module

Classes and constants for the DUIDs defined in **RFC 3315**[120]

class dhcpkit.ipv6.duids.**DUID**

Bases: *dhcpkit.protocol_element.ProtocolElement* (page 218)

RFC 3315#section-9.1[121]

A DUID consists of a two-octet type code represented in network byte order, followed by a variable number of octets that make up the actual identifier. A DUID can be no more than 128 octets long (not including the type code).

classmethod determine_class(*buffer: bytes*, *offset: int = 0*) → type

Return the appropriate subclass from the registry, or UnknownDUID if no subclass is registered.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading

Returns The best known class for this duid data

duid_type = 0

parse_duid_header(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int

Parse the DUID type and perform some basic validation.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

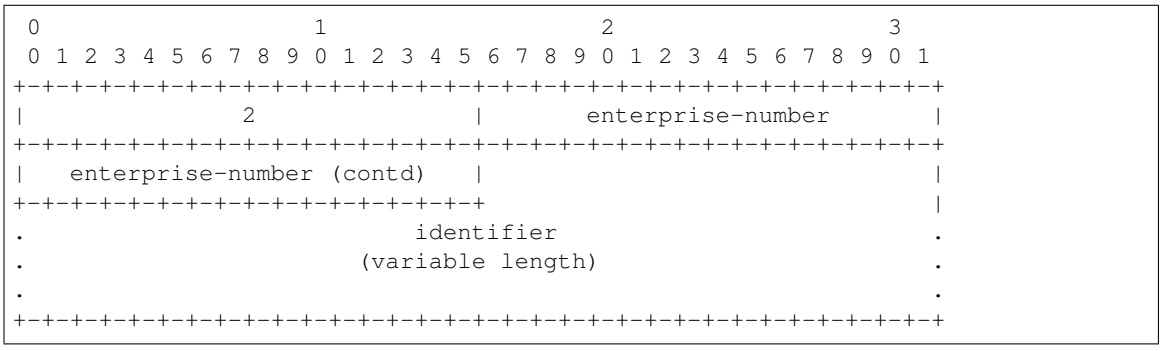
Returns The number of bytes used from the buffer

class dhcpkit.ipv6.duids.**EnterpriseDUID**(*enterprise_number: int = 0*, *identifier: bytes = b''*)

Bases: *dhcpkit.ipv6.duids.DUID* (page 148)

RFC 3315#section-9.3[122]

This form of DUID is assigned by the vendor to the device. It consists of the vendor's registered Private Enterprise Number as maintained by IANA [6] followed by a unique identifier assigned by the vendor. The following diagram summarizes the structure of a DUID-EN:



[120] https://tools.ietf.org/html/rfc3315.html
[121] https://tools.ietf.org/html/rfc3315.html#section-9.1
[122] https://tools.ietf.org/html/rfc3315.html#section-9.3

The source of the identifier is left up to the vendor defining it, but each identifier part of each DUID-EN MUST be unique to the device that is using it, and MUST be assigned to the device at the time it is manufactured and stored in some form of non-volatile storage. The generated DUID SHOULD be recorded in non-erasable storage. The enterprise-number is the vendor's registered Private Enterprise Number as maintained by IANA [6]. The enterprise-number is stored as an unsigned 32 bit number.

An example DUID of this type might look like this:

```
+---+---+---+---+---+---+---+---+
| 0 | 2 | 0 | 0 | 0 |  9| 12|192|
+---+---+---+---+---+---+---+---+
|132|221| 3 | 0 | 9 | 18|
+---+---+---+---+---+---+
```

This example includes the two-octet type of 2, the Enterprise Number (9), followed by eight octets of identifier data (0x0CC084D303000912).

duid_type = 2

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()
Validate that the contents of this object conform to protocol specs.

class dhcpkit.ipv6.duids.**LinkLayerDUID**(*hardware_type: int = 0*, *link_layer_address: bytes = b''*)
Bases: *dhcpkit.ipv6.duids.DUID* (page 148)

RFC 3315#section-9.4[123]

This type of DUID consists of two octets containing the DUID type 3, a two octet network hardware type code, followed by the link-layer address of any one network interface that is permanently connected to the client or server device. For example, a host that has a network interface implemented in a chip that is unlikely to be removed and used elsewhere could use a DUID-LL. The hardware type MUST be a valid hardware type assigned by the IANA, as described in **RFC 826**[124] [14]. The hardware type is stored in network byte order. The link-layer address is stored in canonical form, as described in **RFC 2464**[125] [2]. The following diagram illustrates the format of a DUID-LL:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               3               |    hardware type (16 bits)    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
.                                                               .
.             link-layer address (variable length)              .
```

(continues on next page)

[123] https://tools.ietf.org/html/rfc3315.html#section-9.4
[124] https://tools.ietf.org/html/rfc826.html
[125] https://tools.ietf.org/html/rfc2464.html

(continued from previous page)

```
.                                                               .
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The choice of network interface can be completely arbitrary, as long as that interface provides a unique link-layer address and is permanently attached to the device on which the DUID-LL is being generated. The same DUID-LL SHOULD be used in configuring all network interfaces connected to the device, regardless of which interface's link-layer address was used to generate the DUID.

DUID-LL is recommended for devices that have a permanently-connected network interface with a link-layer address, and do not have nonvolatile, writable stable storage. DUID-LL MUST NOT be used by DHCP clients or servers that cannot tell whether or not a network interface is permanently attached to the device on which the DHCP client is running.

display_hardware_type() → dhcpkit.protocol_element.ElementDataRepresentation
Nicer representation of hardware types :return: Representation of hardware type

display_link_layer_address() → Union
Nicer representation of link-layer address if we know the hardware type :return: Representation of link-layer address

duid_type = 3

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()
Validate that the contents of this object conform to protocol specs.

class dhcpkit.ipv6.duids.**LinkLayerTimeDUID**(*hardware_type: int = 0*, *time: int = 0*, *link_layer_address: bytes = b''*)

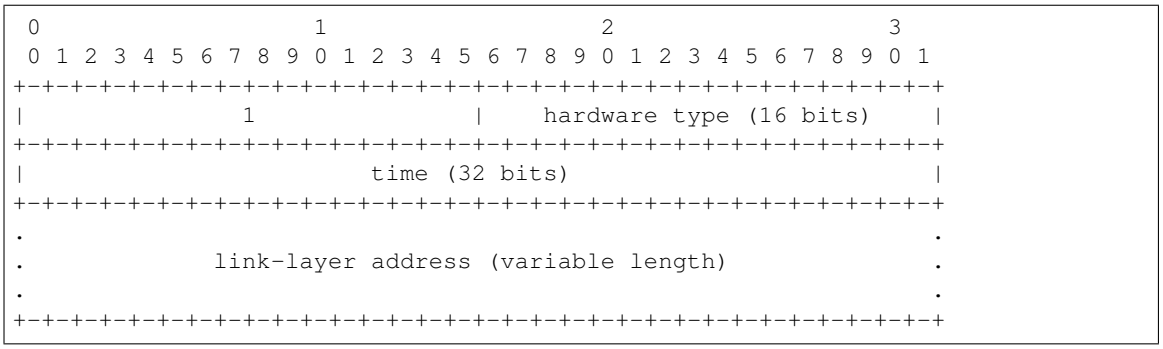
Bases: *dhcpkit.ipv6.duids.DUID* (page 148)

RFC 3315#section-9.2[126]

This type of DUID consists of a two octet type field containing the value 1, a two octet hardware type code, four octets containing a time value, followed by link-layer address of any one network interface that is connected to the DHCP device at the time that the DUID is generated. The time value is the time that the DUID is generated represented in seconds since midnight (UTC), January 1, 2000, modulo 2^32. The hardware type MUST be a valid hardware type assigned by the IANA as described in **RFC 826**[127] [14]. Both the time and the hardware type are stored in network byte order. The link-layer address is stored in canonical form, as described in **RFC 2464**[128] [2].

The following diagram illustrates the format of a DUID-LLT:

[126] https://tools.ietf.org/html/rfc3315.html#section-9.2
[127] https://tools.ietf.org/html/rfc826.html
[128] https://tools.ietf.org/html/rfc2464.html



The choice of network interface can be completely arbitrary, as long as that interface provides a globally unique link-layer address for the link type, and the same DUID-LLT SHOULD be used in configuring all network interfaces connected to the device, regardless of which interface's link-layer address was used to generate the DUID-LLT.

Clients and servers using this type of DUID MUST store the DUID-LLT in stable storage, and MUST continue to use this DUID-LLT even if the network interface used to generate the DUID-LLT is removed. Clients and servers that do not have any stable storage MUST NOT use this type of DUID.

Clients and servers that use this DUID SHOULD attempt to configure the time prior to generating the DUID, if that is possible, and MUST use some sort of time source (for example, a real-time clock) in generating the DUID, even if that time source could not be configured prior to generating the DUID. The use of a time source makes it unlikely that two identical DUID-LLTs will be generated if the network interface is removed from the client and another client then uses the same network interface to generate a DUID-LLT. A collision between two DUID-LLTs is very unlikely even if the clocks have not been configured prior to generating the DUID.

This method of DUID generation is recommended for all general purpose computing devices such as desktop computers and laptop computers, and also for devices such as printers, routers, and so on, that contain some form of writable non-volatile storage.

Despite our best efforts, it is possible that this algorithm for generating a DUID could result in a client identifier collision. A DHCP client that generates a DUID-LLT using this mechanism MUST provide an administrative interface that replaces the existing DUID with a newly-generated DUID-LLT.

display_hardware_type() → dhcpkit.protocol_element.ElementDataRepresentation
Nicer representation of hardware types :return: Representation of hardware type

display_link_layer_address() → Union
Nicer representation of link-layer address if we know the hardware type :return: Representation of link-layer address

duid_type = 1

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()
Validate that the contents of this object conform to protocol specs.

class dhcpkit.ipv6.duids.**UnknownDUID**(*duid_type: int = 0*, *duid_data: bytes = b''*)
Bases: *dhcpkit.ipv6.duids.DUID* (page 148)

Container for raw DUID content for cases where we don't know how to decode the DUID.

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

dhcpkit.ipv6.message_registry module

The option registry

class dhcpkit.ipv6.message_registry.**MessageRegistry**
Bases: *dhcpkit.registry.Registry* (page 220)

Registry for DHCPKit IPv6 Options

entry_point = 'dhcpkit.ipv6.messages'

get_name(*item: object*) → str
Get the name for the by_name mapping.

Parameters **item** – The item to determine the name of

Returns The name to use as key in the mapping

dhcpkit.ipv6.messages module

Classes and constants for the message types defined in **RFC 3315**[129]

class dhcpkit.ipv6.messages.**AdvertiseMessage**(*transaction_id: bytes = b'x00x00x00'*, *options: Iterable = None*)
Bases: *dhcpkit.ipv6.messages.ClientServerMessage* (page 152)

A server sends an Advertise message to indicate that it is available for DHCP service, in response to a Solicit message received from a client.

from_server_to_client = True

message_type = 2

class dhcpkit.ipv6.messages.**ClientServerMessage**(*transaction_id: bytes = b'x00x00x00'*, *options: Iterable = None*)
Bases: *dhcpkit.ipv6.messages.Message* (page 154)

[129] https://tools.ietf.org/html/rfc3315.html

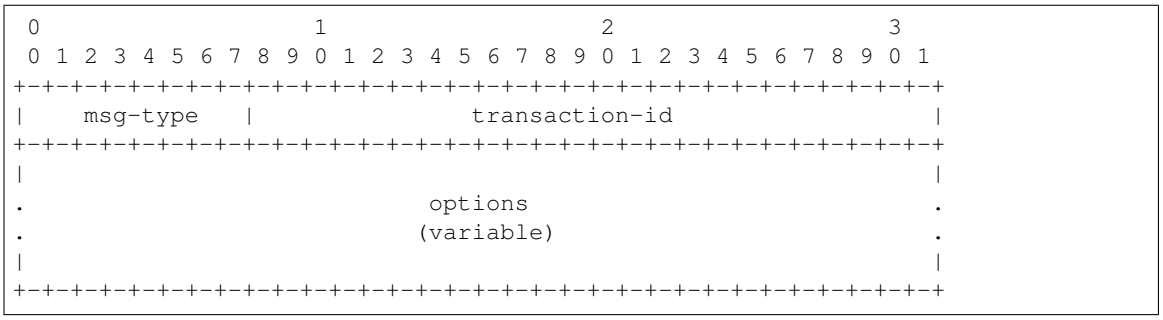
RFC 3315#section-6[130]

All DHCP messages sent between clients and servers share an identical fixed format header and a variable format area for options.

All values in the message header and in options are in network byte order.

Options are stored serially in the options field, with no padding between the options. Options are byte-aligned but are not aligned in any other way such as on 2 or 4 byte boundaries.

The following diagram illustrates the format of DHCP messages sent between clients and servers:



msg-type Identifies the DHCP message type; the available message types are listed in section 5.3.

transaction-id The transaction ID for this message exchange.

options Options carried in this message; options are described in section 22.

get_option_of_type(**args*) → Union
Get the first option that is a subclass of the given class.

Parameters **args** – The classes to look for

Returns The option or None

get_options_of_type(**args*) → List
Get all options that are subclasses of the given class.

Parameters **args** – The classes to look for

Returns The list of options

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()
Validate that the contents of this object conform to protocol specs.

[130] https://tools.ietf.org/html/rfc3315.html#section-6

class dhcpkit.ipv6.messages.**ConfirmMessage**(*transaction_id: bytes = b'x00x00x00'*, *options: Iterable = None*)

Bases: *dhcpkit.ipv6.messages.ClientServerMessage* (page 152)

A client sends a Confirm message to any available server to determine whether the addresses it was assigned are still appropriate to the link to which the client is connected.

from_client_to_server = True

message_type = 4

class dhcpkit.ipv6.messages.**DeclineMessage**(*transaction_id: bytes = b'x00x00x00'*, *options: Iterable = None*)

Bases: *dhcpkit.ipv6.messages.ClientServerMessage* (page 152)

A client sends a Decline message to a server to indicate that the client has determined that one or more addresses assigned by the server are already in use on the link to which the client is connected.

from_client_to_server = True

message_type = 9

class dhcpkit.ipv6.messages.**InformationRequestMessage**(*transaction_id: bytes = b'x00x00x00'*, *options: Iterable = None*)

Bases: *dhcpkit.ipv6.messages.ClientServerMessage* (page 152)

A client sends an Information-request message to a server to request configuration parameters without the assignment of any IP addresses to the client.

from_client_to_server = True

message_type = 11

class dhcpkit.ipv6.messages.**Message**

Bases: *dhcpkit.protocol_element.ProtocolElement* (page 218)

The base class for DHCP messages.

classmethod determine_class(*buffer: bytes*, *offset: int = 0*) → type

Return the appropriate subclass from the registry, or UnknownClientServerMessage if no subclass is registered.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading

Returns The best known class for this message data

from_client_to_server = False

from_server_to_client = False

message_type = 0

class dhcpkit.ipv6.messages.**RebindMessage**(*transaction_id: bytes = b'x00x00x00'*, *options: Iterable = None*)

Bases: *dhcpkit.ipv6.messages.ClientServerMessage* (page 152)

A client sends a Rebind message to any available server to extend the lifetimes on the addresses assigned to the client and to update other configuration parameters; this message is sent after a client receives no response to a Renew message.

from_client_to_server = True

message_type = 6

class dhcpkit.ipv6.messages.**ReconfigureMessage**(*transaction_id: bytes = b'x00x00x00', options: Iterable = None*)

Bases: *dhcpkit.ipv6.messages.ClientServerMessage* (page 152)

A server sends a Reconfigure message to a client to inform the client that the server has new or updated configuration parameters, and that the client is to initiate a Renew/Reply or Information-request/Reply transaction with the server in order to receive the updated information.

from_server_to_client = True

message_type = 10

class dhcpkit.ipv6.messages.**RelayForwardMessage**(*hop_count: int = 0, link_address: ipaddress.IPv6Address = None, peer_address: ipaddress.IPv6Address = None, options: Iterable = None*)

Bases: *dhcpkit.ipv6.messages.RelayServerMessage* (page 155)

A relay agent sends a Relay-forward message to relay messages to servers, either directly or through another relay agent. The received message, either a client message or a Relay-forward message from another relay agent, is encapsulated in an option in the Relay-forward message.

from_client_to_server = True

message_type = 12

wrap_response(*response: dhcpkit.ipv6.messages.ClientServerMessage*) → dhcpkit.ipv6.messages.RelayReplyMessage

The incoming message was wrapped in this RelayForwardMessage. Let this RelayForwardMessage then create a RelayReplyMessage with the correct options and wrap the reply .

Parameters **response** – The response that is going to be sent to the client

Returns The RelayReplyMessage wrapping the response

Return type *RelayReplyMessage* (page 155)

class dhcpkit.ipv6.messages.**RelayReplyMessage**(*hop_count: int = 0, link_address: ipaddress.IPv6Address = None, peer_address: ipaddress.IPv6Address = None, options: Iterable = None*)

Bases: *dhcpkit.ipv6.messages.RelayServerMessage* (page 155)

A server sends a Relay-reply message to a relay agent containing a message that the relay agent delivers to a client. The Relay-reply message may be relayed by other relay agents for delivery to the destination relay agent.

The server encapsulates the client message as an option in the Relay-reply message, which the relay agent extracts and relays to the client.

from_server_to_client = True

message_type = 13

class dhcpkit.ipv6.messages.**RelayServerMessage**(*hop_count: int = 0, link_address: ipaddress.IPv6Address = None, peer_address: ipaddress.IPv6Address = None, options: Iterable = None*)

Bases: *dhcpkit.ipv6.messages.Message* (page 154)

RFC 3315#section-7[131]

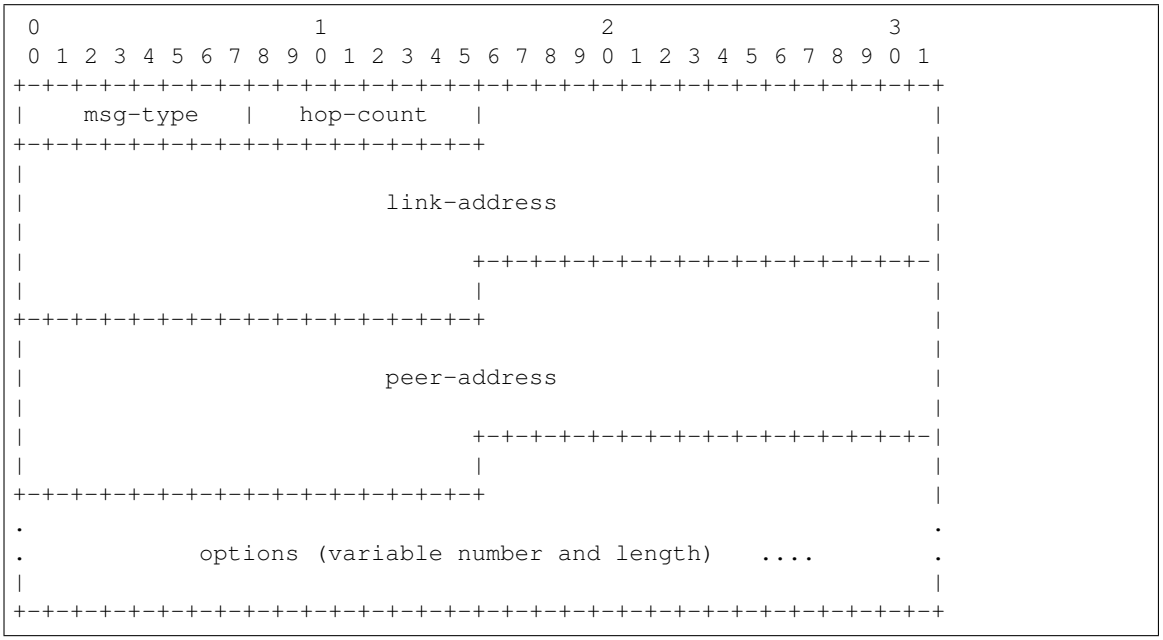
Relay agents exchange messages with servers to relay messages between clients and servers that are not connected to the same link.

[131] https://tools.ietf.org/html/rfc3315.html#section-7

All values in the message header and in options are in network byte order.

Options are stored serially in the options field, with no padding between the options. Options are byte-aligned but are not aligned in any other way such as on 2 or 4 byte boundaries.

There are two relay agent messages, which share the following format:



get_option_of_type(**args*) → Union
Get the first option that is a subclass of the given class.

Parameters **args** – The classes to look for

Returns The option or None

get_options_of_type(**args*) → List
Get all options that are subclasses of the given class.

Parameters **args** – The classes to look for

Returns The list of options

inner_message
Utility method to easily get the innermost message from the RelayMessageOption inside this RelayServerMessage.

Returns The message, if found

inner_relay_message
Utility method to easily get the innermost relay message from the RelayMessageOption inside this RelayServerMessage.

Returns The message, if found

Return type *RelayServerMessage* (page 155) or None[132]

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from

[132] https://docs.python.org/3.4/library/constants.html#None

- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

relayed_message
Utility method to easily get the relayed message from the RelayMessageOption inside this RelayServerMessage.

Returns The message, if found

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()
Validate that the contents of this object conform to protocol specs.

class dhcpkit.ipv6.messages.**ReleaseMessage**(*transaction_id: bytes = b'x00x00x00'*, *options: Iterable = None*)
Bases: *dhcpkit.ipv6.messages.ClientServerMessage* (page 152)

A client sends a Release message to the server that assigned addresses to the client to indicate that the client will no longer use one or more of the assigned addresses.

from_client_to_server = True

message_type = 8

class dhcpkit.ipv6.messages.**RenewMessage**(*transaction_id: bytes = b'x00x00x00'*, *options: Iterable = None*)
Bases: *dhcpkit.ipv6.messages.ClientServerMessage* (page 152)

A client sends a Renew message to the server that originally provided the client's addresses and configuration parameters to extend the lifetimes on the addresses assigned to the client and to update other configuration parameters.

from_client_to_server = True

message_type = 5

class dhcpkit.ipv6.messages.**ReplyMessage**(*transaction_id: bytes = b'x00x00x00'*, *options: Iterable = None*)
Bases: *dhcpkit.ipv6.messages.ClientServerMessage* (page 152)

A server sends a Reply message containing assigned addresses and configuration parameters in response to a Solicit, Request, Renew, Rebind message received from a client. A server sends a Reply message containing configuration parameters in response to an Information-request message. A server sends a Reply message in response to a Confirm message confirming or denying that the addresses assigned to the client are appropriate to the link to which the client is connected. A server sends a Reply message to acknowledge receipt of a Release or Decline message.

from_server_to_client = True

message_type = 7

class dhcpkit.ipv6.messages.**RequestMessage**(*transaction_id: bytes = b'x00x00x00'*, *options: Iterable = None*)
Bases: *dhcpkit.ipv6.messages.ClientServerMessage* (page 152)

A client sends a Request message to request configuration parameters, including IP addresses, from a specific server.

from_client_to_server = True

message_type = 3

class dhcpkit.ipv6.messages.**SolicitMessage**(*transaction_id: bytes = b'x00x00x00'*, *options: Iterable = None*)

Bases: *dhcpkit.ipv6.messages.ClientServerMessage* (page 152)

SOLICIT (1) A client sends a Solicit message to locate servers.

from_client_to_server = True

message_type = 1

class dhcpkit.ipv6.messages.**UnknownMessage**(*message_type: int = 0*, *message_data: bytes = b''*)

Bases: *dhcpkit.ipv6.messages.Message* (page 154)

Container for raw message content for cases where we don't know how to decode the message.

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int

Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

save() → Union

Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()

Validate that the contents of this object conform to protocol specs.

dhcpkit.ipv6.option_registry module

The option registry

class dhcpkit.ipv6.option_registry.**OptionRegistry**

Bases: *dhcpkit.registry.Registry* (page 220)

Registry for DHCPKit IPv6 Options

entry_point = 'dhcpkit.ipv6.options'

get_name(*item: object*) → str

Get the name for the by_name mapping.

Parameters **item** – The item to determine the name of

Returns The name to use as key in the mapping

dhcpkit.ipv6.options module

Classes and constants for the options defined in **RFC 3315**[133]

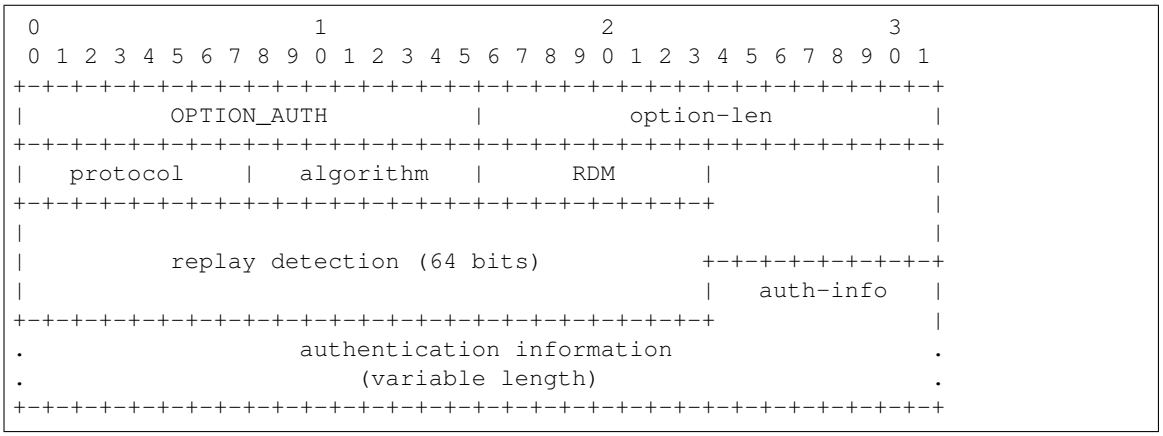
[133] https://tools.ietf.org/html/rfc3315.html

class dhcpkit.ipv6.options.**AuthenticationOption**(*protocol: int = 0, algorithm: int = 0, rdm: int = 0, replay_detection: bytes = b'x00x00x00x00x00x00x00x00', auth_info: bytes = b''*)

Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 3315#section-22.11[134]

The Authentication option carries authentication information to authenticate the identity and contents of DHCP messages. The use of the Authentication option is described in section 21. The format of the Authentication option is:



option-code OPTION_AUTH (11).

option-len 11 + length of authentication information field.

protocol The authentication protocol used in this authentication option.

algorithm The algorithm used in the authentication protocol.

RDM The replay detection method used in this authentication option.

Replay detection The replay detection information for the RDM.

authentication information The authentication information, as specified by the protocol and algorithm used in this authentication option.

load_from(*buffer: bytes, offset: int = 0, length: int = None*) → int

Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 11

save() → Union

Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

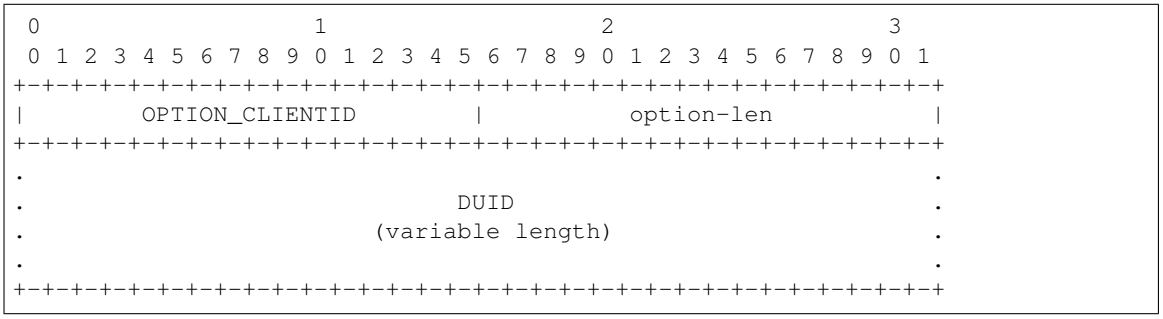
[134] https://tools.ietf.org/html/rfc3315.html#section-22.11

validate()
Validate that the contents of this object conform to protocol specs.

class dhcpkit.ipv6.options.**ClientIdOption**(*duid: dhcpkit.ipv6.duids.DUID = None*)
Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 3315#section-22.2[135]

The Client Identifier option is used to carry a DUID (see section 9) identifying a client between a client and a server. The format of the Client Identifier option is:



option-code OPTION_CLIENTID (1).

option-len Length of DUID in octets.

DUID The DUID for the client.

duid = None
The DUID of the client

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 1

save() → Union
Save the internal state of this object as a buffer.

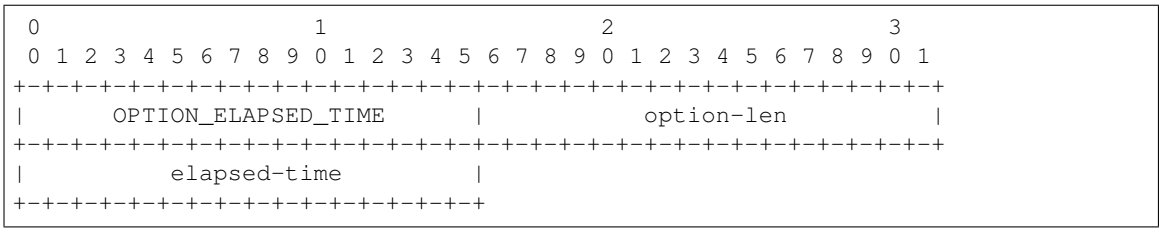
Returns The buffer with the data from this element

validate()
Validate that the contents of this object conform to protocol specs.

class dhcpkit.ipv6.options.**ElapsedTimeOption**(*elapsed_time: int = 0*)
Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 3315#section-22.9[136]

[135] https://tools.ietf.org/html/rfc3315.html#section-22.2
[136] https://tools.ietf.org/html/rfc3315.html#section-22.9



option-code OPTION_ELAPSED_TIME (8).

option-len

2.

elapsed-time The amount of time since the client began its current DHCP transaction. This time is expressed in hundredths of a second (10^-2 seconds).

A client MUST include an Elapsed Time option in messages to indicate how long the client has been trying to complete a DHCP message exchange. The elapsed time is measured from the time at which the client sent the first message in the message exchange, and the elapsed-time field is set to 0 in the first message in the message exchange. Servers and Relay Agents use the data value in this option as input to policy controlling how a server responds to a client message. For example, the elapsed time option allows a secondary DHCP server to respond to a request when a primary server has not answered in a reasonable time. The elapsed time value is an unsigned, 16 bit integer. The client uses the value 0xffff to represent any elapsed time values greater than the largest time value that can be represented in the Elapsed Time option.

elapsed_time = None
The amount of time since the client began its current DHCP transaction

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 8

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()
Validate that the contents of this object conform to protocol specs.

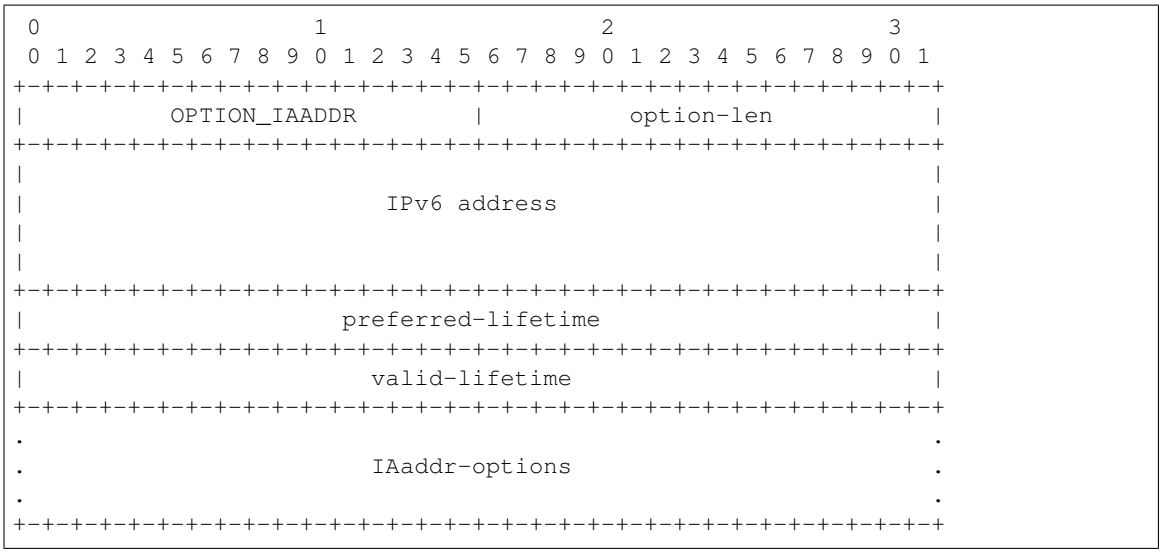
class dhcpkit.ipv6.options.**IAAddressOption**(*address: ipaddress.IPv6Address = None*, *preferred_lifetime: int = 0*, *valid_lifetime: int = 0*, *options: Iterable = None*)
Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 3315#section-22.6[137]

The IA Address option is used to specify IPv6 addresses associated with an IA_NA or an IA_TA. The IA Address option must be encapsulated in the Options field of an IA_NA or IA_TA option. The Options field encapsulates those options that are specific to this address.

[137] https://tools.ietf.org/html/rfc3315.html#section-22.6

The format of the IA Address option is:



option-code OPTION_IAADDR (5).

option-len 24 + length of IAaddr-options field.

IPv6 address An IPv6 address.

preferred-lifetime The preferred lifetime for the IPv6 address in the option, expressed in units of seconds.

valid-lifetime The valid lifetime for the IPv6 address in the option, expressed in units of seconds.

IAaddr-options Options associated with this address.

In a message sent by a client to a server, values in the preferred and valid lifetime fields indicate the client's preference for those parameters. The client may send 0 if it has no preference for the preferred and valid lifetimes. In a message sent by a server to a client, the client MUST use the values in the preferred and valid lifetime fields for the preferred and valid lifetimes. The values in the preferred and valid lifetimes are the number of seconds remaining in each lifetime.

A client discards any addresses for which the preferred lifetime is greater than the valid lifetime. A server ignores the lifetimes set by the client if the preferred lifetime is greater than the valid lifetime and ignores the values for T1 and T2 set by the client if those values are greater than the preferred lifetime.

Care should be taken in setting the valid lifetime of an address to 0xffffffff ("infinity"), which amounts to a permanent assignment of an address to a client.

An IA Address option may appear only in an IA_NA option or an IA_TA option. More than one IA Address Option can appear in an IA_NA option or an IA_TA option.

The status of any operations involving this IA Address is indicated in a Status Code option in the IAaddr-options field.

address = None
The IPv6 address

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading

- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 5

options = None
The list of options related to this IAAddressOption

preferred_lifetime = None
The preferred lifetime of this IPv6 address

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

valid_lifetime = None
The valid lifetime of this IPv6 address

validate()
Validate that the contents of this object conform to protocol specs.

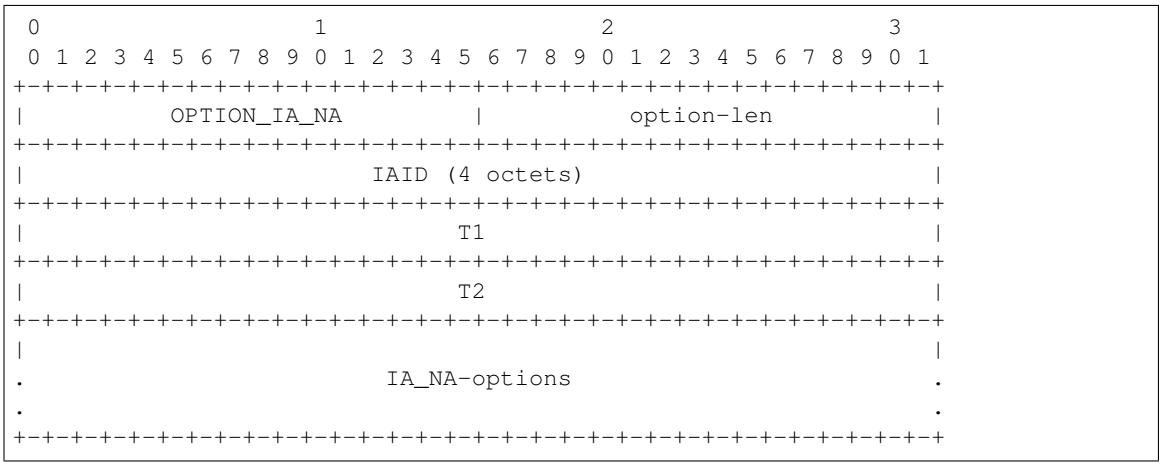
class dhcpkit.ipv6.options.**IANAOption**(*iaid: bytes = b'x00x00x00x00'*, *t1: int = 0*, *t2: int = 0*, *options: Iterable = None*)
Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 3315#section-22.4[138]

The Identity Association for Non-temporary Addresses option (IA_NA option) is used to carry an IA_NA, the parameters associated with the IA_NA, and the non-temporary addresses associated with the IA_NA.

Addresses appearing in an IA_NA option are not temporary addresses (see section 22.5).

The format of the IA_NA option is:



option-code OPTION_IA_NA (3).

option-len 12 + length of IA_NA-options field.

IAID The unique identifier for this IA_NA; the IAID must be unique among the identifiers for all of this client's IA_NAs. The number space for IA_NA IAIDs is separate from the number space for IA_TA IAIDs.

T1 The time at which the client contacts the server from which the addresses in the IA_NA were obtained to extend the lifetimes of the addresses assigned to the IA_NA; T1 is a time duration relative to the current time expressed in units of seconds.

T2 The time at which the client contacts any available server to extend the lifetimes of the addresses assigned to the IA_NA; T2 is a time duration relative to the current time expressed in units of seconds.

[138] https://tools.ietf.org/html/rfc3315.html#section-22.4

IA_NA-options Options associated with this IA_NA.

The IA_NA-options field encapsulates those options that are specific to this IA_NA. For example, all of the IA Address Options carrying the addresses associated with this IA_NA are in the IA_NA-options field.

An IA_NA option may only appear in the options area of a DHCP message. A DHCP message may contain multiple IA_NA options.

The status of any operations involving this IA_NA is indicated in a Status Code option in the IA_NA-options field.

Note that an IA_NA has no explicit "lifetime" or "lease length" of its own. When the valid lifetimes of all of the addresses in an IA_NA have expired, the IA_NA can be considered as having expired. T1 and T2 are included to give servers explicit control over when a client recontacts the server about a specific IA_NA.

In a message sent by a client to a server, values in the T1 and T2 fields indicate the client's preference for those parameters. The client sets T1 and T2 to 0 if it has no preference for those values. In a message sent by a server to a client, the client MUST use the values in the T1 and T2 fields for the T1 and T2 parameters, unless those values in those fields are 0. The values in the T1 and T2 fields are the number of seconds until T1 and T2.

The server selects the T1 and T2 times to allow the client to extend the lifetimes of any addresses in the IA_NA before the lifetimes expire, even if the server is unavailable for some short period of time. Recommended values for T1 and T2 are .5 and .8 times the shortest preferred lifetime of the addresses in the IA that the server is willing to extend, respectively. If the "shortest" preferred lifetime is 0xffffffff ("infinity"), the recommended T1 and T2 values are also 0xffffffff. If the time at which the addresses in an IA_NA are to be renewed is to be left to the discretion of the client, the server sets T1 and T2 to 0.

If a server receives an IA_NA with T1 greater than T2, and both T1 and T2 are greater than 0, the server ignores the invalid values of T1 and T2 and processes the IA_NA as though the client had set T1 and T2 to 0.

If a client receives an IA_NA with T1 greater than T2, and both T1 and T2 are greater than 0, the client discards the IA_NA option and processes the remainder of the message as though the server had not included the invalid IA_NA option.

Care should be taken in setting T1 or T2 to 0xffffffff ("infinity"). A client will never attempt to extend the lifetimes of any addresses in an IA with T1 set to 0xffffffff. A client will never attempt to use a Rebind message to locate a different server to extend the lifetimes of any addresses in an IA with T2 set to 0xffffffff.

get_addresses() → List
Get all addresses from IAAddressOptions

Returns list if addresses

get_option_of_type(**args*) → Union
Get the first option that is a subclass of the given class.

Parameters **args** – The classes to look for

Returns The option or None

get_options_of_type(**args*) → List
Get all options that are subclasses of the given class.

Parameters **args** – The classes to look for

Returns The list of options

iaid = None
The unique identifier for this IA_NA

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 3

options = None
The list of options contained in this IANAOption

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

t1 = None
The time at which the client contacts the server to renew its addresses

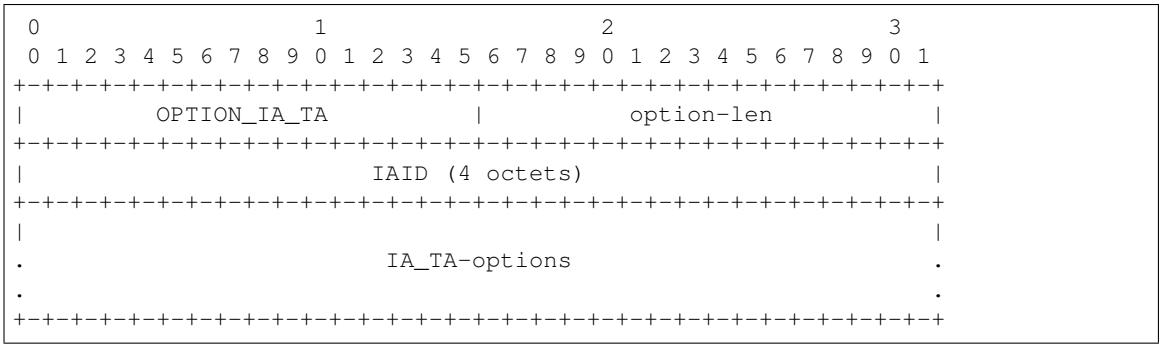
t2 = None
The time at which the client contacts any available server to rebind its addresses

validate()
Validate that the contents of this object conform to protocol specs.

class dhcpkit.ipv6.options.**IATAOption**(*iaid: bytes = b'x00x00x00x00'*, *options: Iterable = None*)
Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 3315#section-22.5[139]

The Identity Association for the Temporary Addresses (IA_TA) option is used to carry an IA_TA, the parameters associated with the IA_TA and the addresses associated with the IA_TA. All of the addresses in this option are used by the client as temporary addresses, as defined in **RFC 3041**[140] [12]. The format of the IA_TA option is:



option-code OPTION_IA_TA (4).

option-len 4 + length of IA_TA-options field.

IAID The unique identifier for this IA_TA; the IAID must be unique among the identifiers for all of this client's IA_TAs. The number space for IA_TA IAIDs is separate from the number space for IA_NA IAIDs.

IA_TA-options Options associated with this IA_TA.

The IA_TA-Options field encapsulates those options that are specific to this IA_TA. For example, all of the IA Address Options carrying the addresses associated with this IA_TA are in the IA_TA-options field.

[139] https://tools.ietf.org/html/rfc3315.html#section-22.5
[140] https://tools.ietf.org/html/rfc3041.html

Each IA_TA carries one "set" of temporary addresses; that is, at most one address from each prefix assigned to the link to which the client is attached.

An IA_TA option may only appear in the options area of a DHCP message. A DHCP message may contain multiple IA_TA options.

The status of any operations involving this IA_TA is indicated in a Status Code option in the IA_TA-options field.

Note that an IA has no explicit "lifetime" or "lease length" of its own. When the valid lifetimes of all of the addresses in an IA_TA have expired, the IA can be considered as having expired.

An IA_TA option does not include values for T1 and T2. A client MAY request that the lifetimes on temporary addresses be extended by including the addresses in a IA_TA option sent in a Renew or Rebind message to a server. For example, a client would request an extension on the lifetime of a temporary address to allow an application to continue to use an established TCP connection.

The client obtains new temporary addresses by sending an IA_TA option with a new IAID to a server. Requesting new temporary addresses from the server is the equivalent of generating new temporary addresses as described in **RFC 3041**[141]. The server will generate new temporary addresses and return them to the client. The client should request new temporary addresses before the lifetimes on the previously assigned addresses expire.

A server MUST return the same set of temporary address for the same IA_TA (as identified by the IAID) as long as those addresses are still valid. After the lifetimes of the addresses in an IA_TA have expired, the IAID may be reused to identify a new IA_TA with new temporary addresses.

This option MAY appear in a Confirm message if the lifetimes on the temporary addresses in the associated IA have not expired.

get_addresses() → List
Get all addresses from IAAddressOptions

Returns list if addresses

get_option_of_type(**args*) → Union
Get the first option that is a subclass of the given class.

Parameters **args** – The classes to look for

Returns The option or None

get_options_of_type(**args*) → List
Get all options that are subclasses of the given class.

Parameters **args** – The classes to look for

Returns The list of options

iaid = None
The unique identifier for this IA_TA

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 4

[141] https://tools.ietf.org/html/rfc3041.html

options = None
The list of options contained in this IATAOption

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

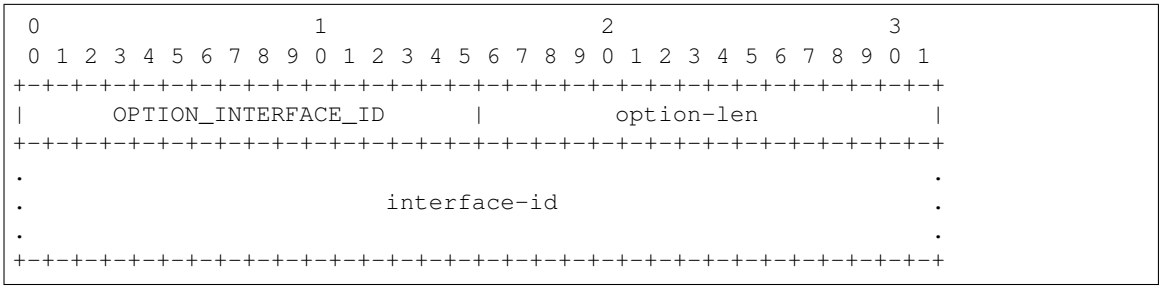
validate()
Validate that the contents of this object conform to protocol specs.

class dhcpkit.ipv6.options.**InterfaceIdOption**(*interface_id: bytes = b''*)
Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 3315#section-22.18[142]

The relay agent MAY send the Interface-id option to identify the interface on which the client message was received. If a relay agent receives a Relay-reply message with an Interface-id option, the relay agent relays the message to the client through the interface identified by the option.

The format of the Interface ID option is:



option-code OPTION_INTERFACE_ID (18).

option-len Length of interface-id field.

interface-id An opaque value of arbitrary length generated by the relay agent to identify one of the relay agent's interfaces.

The server MUST copy the Interface-Id option from the Relay-Forward message into the Relay-Reply message the server sends to the relay agent in response to the Relay-Forward message. This option MUST NOT appear in any message except a Relay-Forward or Relay-Reply message.

Servers MAY use the Interface-ID for parameter assignment policies. The Interface-ID SHOULD be considered an opaque value, with policies based on exact match only; that is, the Interface-ID SHOULD NOT be internally parsed by the server. The Interface-ID value for an interface SHOULD be stable and remain unchanged, for example, after the relay agent is restarted; if the Interface-ID changes, a server will not be able to use it reliably in parameter assignment policies.

interface_id = None
The interface-ID that the relay received the incoming message on

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

[142] https://tools.ietf.org/html/rfc3315.html#section-22.18

Returns The number of bytes used from the buffer

option_type = 18

save() → Union
Save the internal state of this object as a buffer.

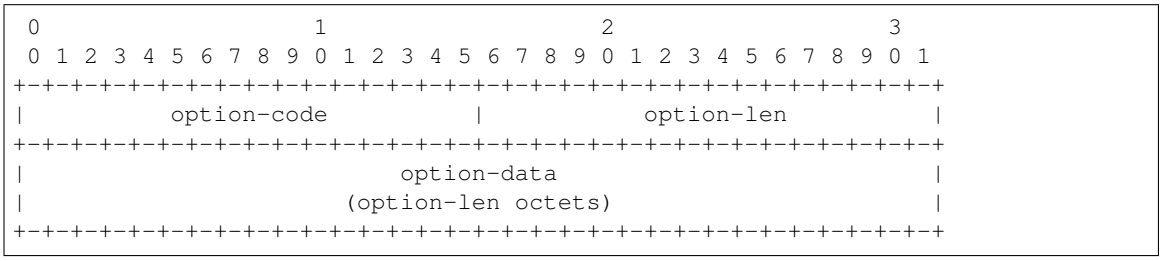
Returns The buffer with the data from this element

validate()
Validate that the contents of this object conform to protocol specs.

class dhcpkit.ipv6.options.**Option**
Bases: *dhcpkit.protocol_element.ProtocolElement* (page 218)

RFC 3315#section-22.1[143]

The format of DHCP options is:



option-code An unsigned integer identifying the specific option type carried in this option.

option-len An unsigned integer giving the length of the option-data field in this option in octets.

option-data The data for the option; the format of this data depends on the definition of the option.

DHCPv6 options are scoped by using encapsulation. Some options apply generally to the client, some are specific to an IA, and some are specific to the addresses within an IA. These latter two cases are discussed in sections 22.4 and 22.6.

classmethod determine_class(*buffer: bytes*, *offset: int = 0*) → type
Return the appropriate subclass from the registry, or UnknownOption if no subclass is registered.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading

Returns The best known class for this option data

option_type = 0

parse_option_header(*buffer: bytes*, *offset: int = 0*, *length: int = None*, *min_length: int = 0*, *max_length: int = 65535*) → Tuple
Parse the option code and length from the buffer and perform some basic validation.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer
- **min_length** – The minimum length this option can have
- **max_length** – The maximum length this option can have

[143] https://tools.ietf.org/html/rfc3315.html#section-22.1

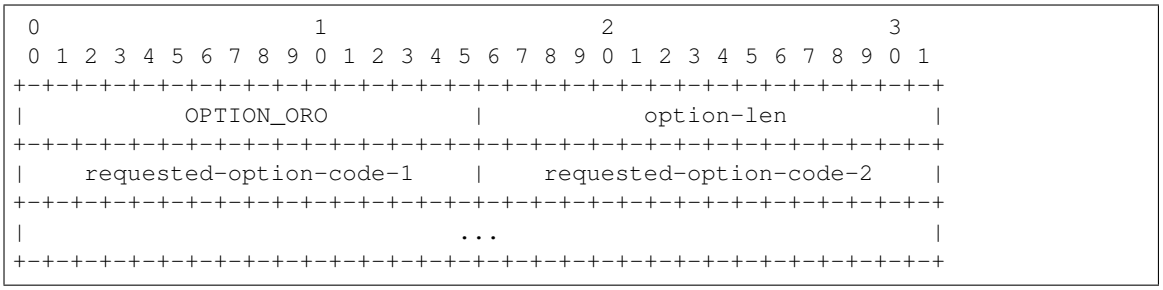
Returns The number of bytes used from the buffer and the value of the option-len field

class dhcpkit.ipv6.options.**OptionRequestOption**(*requested_options: Iterable = None*)

Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 3315#section-22.7[144]

The Option Request option is used to identify a list of options in a message between a client and a server. The format of the Option Request option is:



option-code OPTION_ORO (6).

option-len 2 * number of requested options.

requested-option-code-n The option code for an option requested by the client.

A client MAY include an Option Request option in a Solicit, Request, Renew, Rebind, Confirm or Information-request message to inform the server about options the client wants the server to send to the client. A server MAY include an Option Request option in a Reconfigure option to indicate which options the client should request from the server.

display_requested_options() → List

Provide a nicer output when displaying the requested options.

Returns A list of option names

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int

Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 6

requested_options = None

The list of option type numbers that the client is interested in

save() → Union

Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()

Validate that the contents of this object conform to protocol specs.

[144] https://tools.ietf.org/html/rfc3315.html#section-22.7

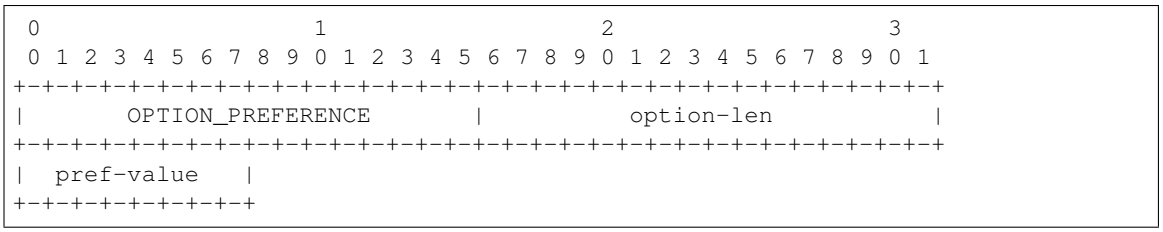
class dhcpkit.ipv6.options.**PreferenceOption**(*preference: int = 0*)

Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 3315#section-22.8[145]

The Preference option is sent by a server to a client to affect the selection of a server by the client.

The format of the Preference option is:



option-code OPTION_PREFERENCE (7).

option-len

1.

pref-value The preference value for the server in this message.

A server MAY include a Preference option in an Advertise message to control the selection of a server by the client. See section 17.1.3 for the use of the Preference option by the client and the interpretation of Preference option data value.

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int

Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 7

preference = None

The preference that the client should treat this server with

save() → Union

Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()

Validate that the contents of this object conform to protocol specs.

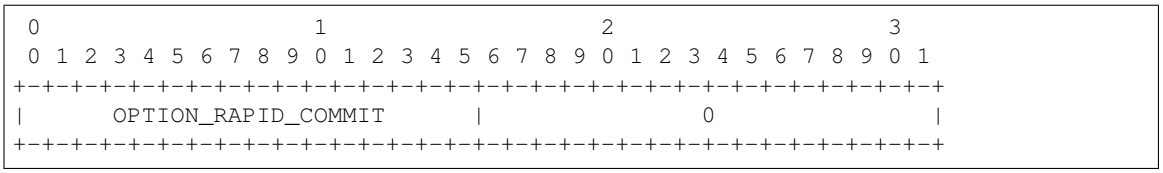
class dhcpkit.ipv6.options.**RapidCommitOption**

Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 3315#section-22.14[146]

The Rapid Commit option is used to signal the use of the two message exchange for address assignment. The format of the Rapid Commit option is:

[145] https://tools.ietf.org/html/rfc3315.html#section-22.8
[146] https://tools.ietf.org/html/rfc3315.html#section-22.14



option-code OPTION_RAPID_COMMIT (14).

option-len

0.

A client MAY include this option in a Solicit message if the client is prepared to perform the Solicit-Reply message exchange described in section 17.1.1.

A server MUST include this option in a Reply message sent in response to a Solicit message when completing the Solicit-Reply message exchange.

DISCUSSION:

Each server that responds with a Reply to a Solicit that includes a Rapid Commit option will commit the assigned addresses in the Reply message to the client, and will not receive any confirmation that the client has received the Reply message. Therefore, if more than one server responds to a Solicit that includes a Rapid Commit option, some servers will commit addresses that are not actually used by the client.

The problem of unused addresses can be minimized, for example, by designing the DHCP service so that only one server responds to the Solicit or by using relatively short lifetimes for assigned addresses.

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 14

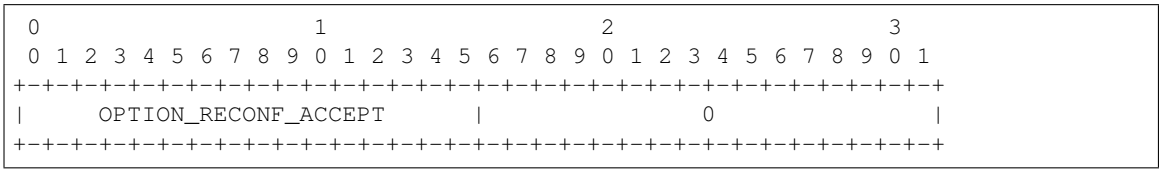
save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

class dhcpkit.ipv6.options.**ReconfigureAcceptOption**
Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 3315#section-22.20[147]

A client uses the Reconfigure Accept option to announce to the server whether the client is willing to accept Reconfigure messages, and a server uses this option to tell the client whether or not to accept Reconfigure messages. The default behavior, in the absence of this option, means unwillingness to accept Reconfigure messages, or instruction not to accept Reconfigure messages, for the client and server messages, respectively. The following figure gives the format of the Reconfigure Accept option:



[147] https://tools.ietf.org/html/rfc3315.html#section-22.20

option-code OPTION_RECONF_ACCEPT (20).

option-len

0.

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 20

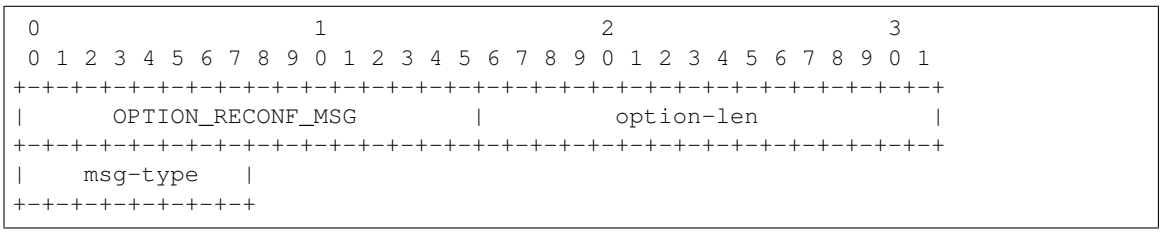
save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

class dhcpkit.ipv6.options.**ReconfigureMessageOption**(*message_type: int = 0*)
Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 3315#section-22.19[148]

A server includes a Reconfigure Message option in a Reconfigure message to indicate to the client whether the client responds with a Renew message or an Information-request message. The format of this option is:



option-code OPTION_RECONF_MSG (19).

option-len

1.

msg-type 5 for Renew message, 11 for Information-request message.

The Reconfigure Message option can only appear in a Reconfigure message.

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

[148] https://tools.ietf.org/html/rfc3315.html#section-22.19

message_type = None
The message type that the client should respond with

option_type = 19

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()
Validate that the contents of this object conform to protocol specs.

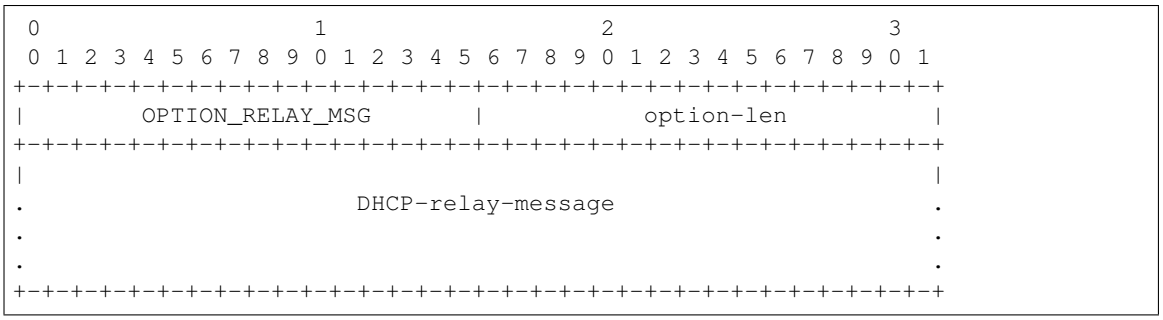
class dhcpkit.ipv6.options.**RelayMessageOption**(*relayed_message: dhcpkit.ipv6.messages.Message = None*)

Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 3315#section-22.10[149]

The Relay Message option carries a DHCP message in a Relay-forward or Relay-reply message.

The format of the Relay Message option is:



option-code OPTION_RELAY_MSG (9)

option-len Length of DHCP-relay-message

DHCP-relay-message In a Relay-forward message, the received message, relayed verbatim to the next relay agent or server; in a Relay-reply message, the message to be copied and relayed to the relay agent or client whose address is in the peer-address field of the Relay-reply message

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 9

relayed_message = None
The relayed DHCP message

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

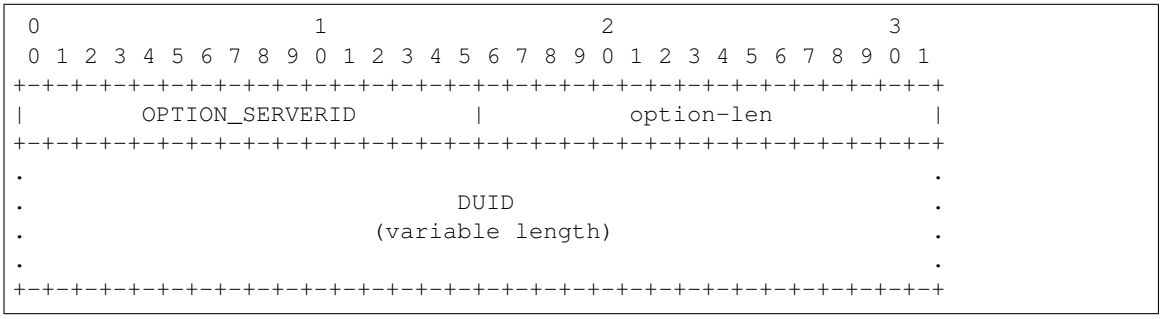
[149] https://tools.ietf.org/html/rfc3315.html#section-22.10

validate()
Validate that the contents of this object conform to protocol specs.

class dhcpkit.ipv6.options.**ServerIdOption**(*duid: dhcpkit.ipv6.duids.DUID = None*)
Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 3315#section-22.3[150]

The Server Identifier option is used to carry a DUID (see section 9) identifying a server between a client and a server. The format of the Server Identifier option is:



option-code OPTION_SERVERID (2).

option-len Length of DUID in octets.

DUID The DUID for the server.

duid = None
The DUID of the server

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 2

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

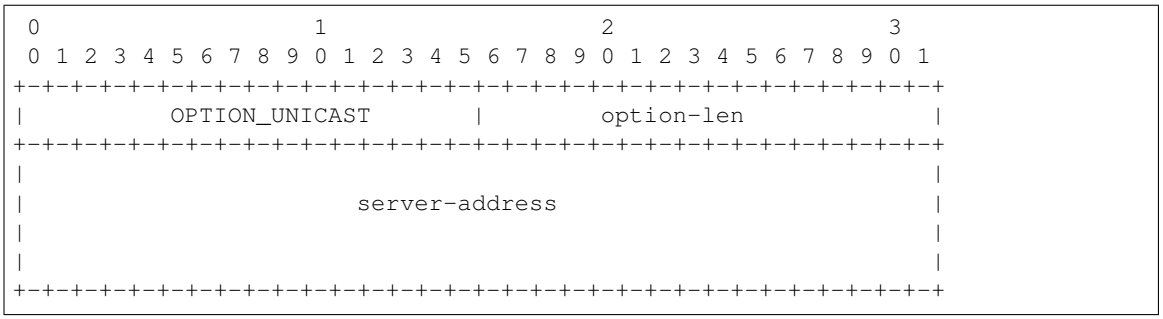
validate()
Validate that the contents of this object conform to protocol specs.

class dhcpkit.ipv6.options.**ServerUnicastOption**(*server_address: ipaddress.IPv6Address = None*)
Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 3315#section-22.12[151]

The server sends this option to a client to indicate to the client that it is allowed to unicast messages to the server. The format of the Server Unicast option is:

[150] https://tools.ietf.org/html/rfc3315.html#section-22.3
[151] https://tools.ietf.org/html/rfc3315.html#section-22.12



option-code OPTION_UNICAST (12).

option-len

16.

server-address The IP address to which the client should send messages delivered using unicast.

The server specifies the IPv6 address to which the client is to send unicast messages in the server-address field. When a client receives this option, where permissible and appropriate, the client sends messages directly to the server using the IPv6 address specified in the server-address field of the option.

When the server sends a Unicast option to the client, some messages from the client will not be relayed by Relay Agents, and will not include Relay Agent options from the Relay Agents. Therefore, a server should only send a Unicast option to a client when Relay Agents are not sending Relay Agent options. A DHCP server rejects any messages sent inappropriately using unicast to ensure that messages are relayed by Relay Agents when Relay Agent options are in use.

Details about when the client may send messages to the server using unicast are in section 18.

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 12

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

server_address = None
The global unicast address that the client may contact this server on

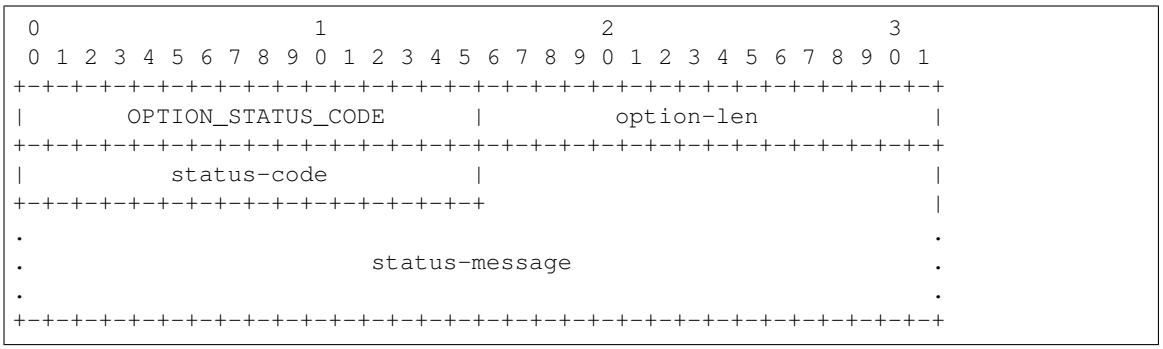
validate()
Validate that the contents of this object conform to protocol specs.

class dhcpkit.ipv6.options.**StatusCodeOption**(*status_code: int = 0*, *status_message: str = ''*)
Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 3315#section-22.13[152]

[152] https://tools.ietf.org/html/rfc3315.html#section-22.13

This option returns a status indication related to the DHCP message or option in which it appears. The format of the Status Code option is:



option-code OPTION_STATUS_CODE (13).

option-len 2 + length of status-message.

status-code The numeric code for the status encoded in this option. The status codes are defined in section 24.4.

status-message A UTF-8 encoded text string suitable for display to an end user, which MUST NOT be null-terminated.

A Status Code option may appear in the options field of a DHCP message and/or in the options field of another option. If the Status Code option does not appear in a message in which the option could appear, the status of the message is assumed to be Success.

display_status_code() → dhcpkit.protocol_element.ElementDataRepresentation
Nicer representation of status codes :return: Representation of status code

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 13

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

status_code = None
The status code

status_message = None
The status message suitable for display to an end user

validate()
Validate that the contents of this object conform to protocol specs.

class dhcpkit.ipv6.options.**UnknownOption**(*option_type: int = 0*, *option_data: bytes = b''*)
Bases: *dhcpkit.ipv6.options.Option* (page 168)

Container for raw option content for cases where we don't know how to decode the option.

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_data = None
The option data as bytes

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()
Validate that the contents of this object conform to protocol specs.

class dhcpkit.ipv6.options.**UserClassOption**(*user_classes: Iterable = None*)
Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 3315#section-22.15[153]

The User Class option is used by a client to identify the type or category of user or applications it represents.

The format of the User Class option is:

```
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       OPTION_USER_CLASS       |          option-len           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
.                                                               .
.                          user-class-data                      .
.                                                               .
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

option-code OPTION_USER_CLASS (15).

option-len Length of user class data field.

user-class-data The user classes carried by the client.

The information contained in the data area of this option is contained in one or more opaque fields that represent the user class or classes of which the client is a member. A server selects configuration information for the client based on the classes identified in this option. For example, the User Class option can be used to configure all clients of people in the accounting department with a different printer than clients of people in the marketing department. The user class information carried in this option MUST be configurable on the client.

The data area of the user class option MUST contain one or more instances of user class data. Each instance of the user class data is formatted as follows:

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-...-+-+-+-+-+-+-+-+
|        user-class-len         |          opaque-data          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-...-+-+-+-+-+-+-+-+
```

[153] https://tools.ietf.org/html/rfc3315.html#section-22.15

The user-class-len is two octets long and specifies the length of the opaque user class data in network byte order.

A server interprets the classes identified in this option according to its configuration to select the appropriate configuration information for the client. A server may use only those user classes that it is configured to interpret in selecting configuration information for a client and ignore any other user classes. In response to a message containing a User Class option, a server includes a User Class option containing those classes that were successfully interpreted by the server, so that the client can be informed of the classes interpreted by the server.

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 15

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

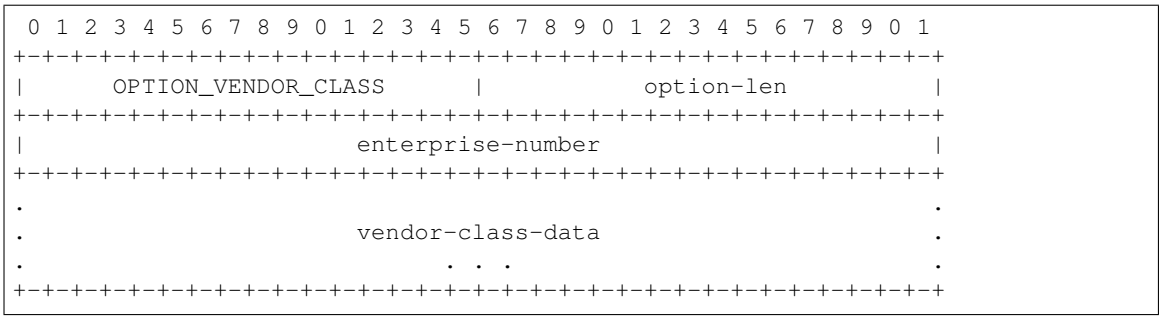
user_classes = None
The list of user classes

validate()
Validate that the contents of this object conform to protocol specs.

class dhcpkit.ipv6.options.**VendorClassOption**(*enterprise_number: int = 0*, *vendor_classes: Iterable = None*)
Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 3315#section-22.16[154]

This option is used by a client to identify the vendor that manufactured the hardware on which the client is running. The information contained in the data area of this option is contained in one or more opaque fields that identify details of the hardware configuration. The format of the Vendor Class option is:



option-code OPTION_VENDOR_CLASS (16).

option-len 4 + length of vendor class data field.

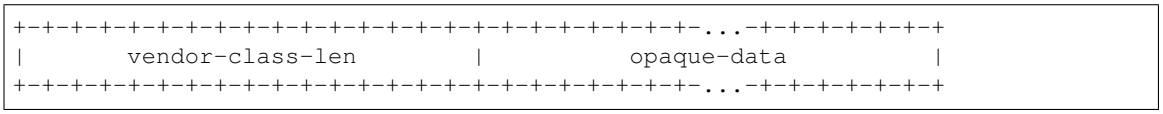
enterprise-number The vendor's registered Enterprise Number as registered with IANA [6].

vendor-class-data The hardware configuration of the host on which the client is running.

[154] https://tools.ietf.org/html/rfc3315.html#section-22.16

The vendor-class-data is composed of a series of separate items, each of which describes some characteristic of the client's hardware configuration. Examples of vendor-class-data instances might include the version of the operating system the client is running or the amount of memory installed on the client.

Each instance of the vendor-class-data is formatted as follows:



The vendor-class-len is two octets long and specifies the length of the opaque vendor class data in network byte order.

enterprise_number = None
The enterprise number

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 16

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()
Validate that the contents of this object conform to protocol specs.

vendor_classes = None
The list of vendor classes for this enterprise

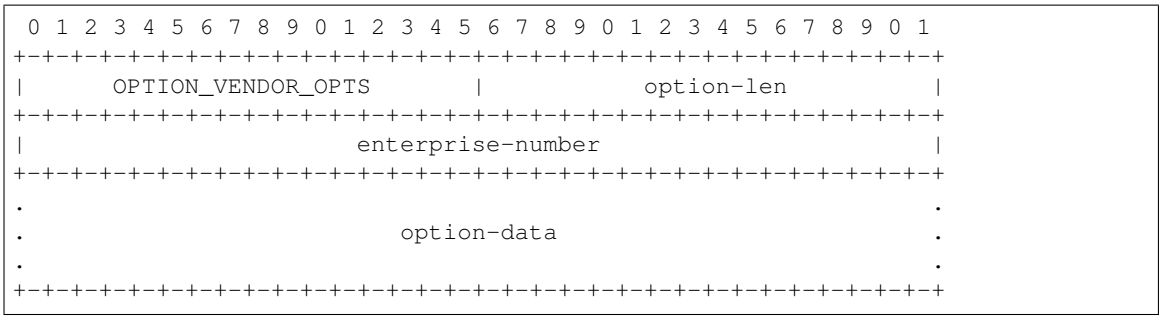
class dhcpkit.ipv6.options.**VendorSpecificInformationOption**(*enterprise_number: int = 0*, *vendor_options: Iterable = None*)

Bases: *dhcpkit.ipv6.options.Option* (page 168)

RFC 3315#section-22.17[155]

This option is used by clients and servers to exchange vendor-specific information.

The format of the Vendor-specific Information option is:



[155] https://tools.ietf.org/html/rfc3315.html#section-22.17

option-code OPTION_VENDOR_OPTS (17)

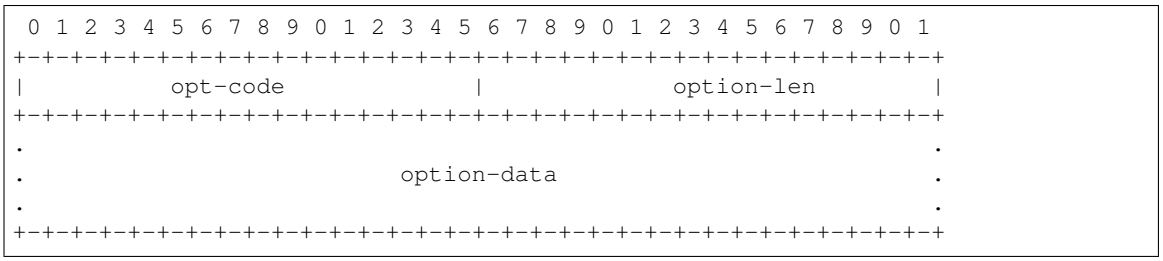
option-len 4 + length of option-data field

enterprise-number The vendor's registered Enterprise Number as registered with IANA [6].

option-data An opaque object of option-len octets, interpreted by vendor-specific code on the clients and servers

The definition of the information carried in this option is vendor specific. The vendor is indicated in the enterprise-number field. Use of vendor-specific information allows enhanced operation, utilizing additional features in a vendor's DHCP implementation. A DHCP client that does not receive requested vendor-specific information will still configure the host device's IPv6 stack to be functional.

The encapsulated vendor-specific options field MUST be encoded as a sequence of code/length/value fields of identical format to the DHCP options field. The option codes are defined by the vendor identified in the enterprise-number field and are not managed by IANA. Each of the encapsulated options is formatted as follows:



opt-code The code for the encapsulated option.

option-len An unsigned integer giving the length of the option-data field in this encapsulated option in octets.

option-data The data area for the encapsulated option.

Multiple instances of the Vendor-specific Information option may appear in a DHCP message. Each instance of the option is interpreted according to the option codes defined by the vendor identified by the Enterprise Number in that option.

enterprise_number = None
The enterprise number

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 17

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()
Validate that the contents of this object conform to protocol specs.

vendor_options = None
The list of vendor options for this enterprise where each option is a tuple containing a code and the data

dhcpkit.ipv6.utils module

Utility functions for IPv6 DHCP

dhcpkit.ipv6.utils.**address_in_prefixes**(*address: ipaddress.IPv6Address*, *prefixes: Iterable*) → bool
Check whether the given address is part of one of the given prefixes

Parameters

- **address** – The IPv6 address to check
- **prefixes** (*list*[156] *[IPv6Network]*) – The list of IPv6 prefixes

Returns Whether the address is part of one of the prefixes

dhcpkit.ipv6.utils.**is_global_unicast**(*address: ipaddress.IPv6Address*) → bool
Check if an address is a global unicast address according to **RFC 4291**[157].

Parameters **address** – The address to check

Returns Whether it is a global unicast address

dhcpkit.ipv6.utils.**prefix_overlaps_prefixes**(*prefix: ipaddress.IPv6Network*, *prefixes: Iterable*) → bool
Check whether the given address is part of one of the given prefixes

Parameters

- **prefix** – The IPv6 prefix to check
- **prefixes** (*list*[158] *[IPv6Network]*) – The list of IPv6 prefixes

Returns Whether the address is part of one of the prefixes

dhcpkit.ipv6.utils.**split_relay_chain**(*message: dhcpkit.ipv6.messages.Message*) → Tuple
Separate the relay chain from the actual request message.

Parameters **message** – The incoming message

Returns The request and the chain of relay messages starting with the one closest to the client

dhcpkit.tests package

All the unit tests go here

class dhcpkit.tests.**DeepCopyMagicMock**(**args*, ***kw*)
Bases: unittest.mock.MagicMock[159]

A magic mock class that deep-copies the method arguments to check the state of mutable objects at call time

[156] https://docs.python.org/3.4/library/stdtypes.html#list
[157] https://tools.ietf.org/html/rfc4291.html
[158] https://docs.python.org/3.4/library/stdtypes.html#list
[159] https://docs.python.org/3.4/library/unittest.mock.html#unittest.mock.MagicMock

Subpackages

dhcpkit.tests.common package

Tests for common code (for when we implement IPv4 as well)

Subpackages

dhcpkit.tests.common.logging package

Test whether the common logging functions work as intended

Submodules

dhcpkit.tests.common.logging.test_verbosity module

Test whether the common logging verbosity functions work as intended

class dhcpkit.tests.common.logging.test_verbosity.**VerbosityLoggerTestCase**(*methodName='runT*
Bases: unittest.case.TestCase

test_create_handler()

test_existing_handler()

test_logger_level()

dhcpkit.tests.common.privileges package

Tests for common privileges code

Submodules

dhcpkit.tests.common.privileges.test_privileges module

Test whether the common privilege functions work as intended

class dhcpkit.tests.common.privileges.test_privileges.**PrivilegeTestCase**(*methodName='runTest*
Bases: unittest.case.TestCase

static get_nobody() → pwd.struct_passwd

static get_somebody() → pwd.struct_passwd

setUp()
Hook method for setting up the test fixture before exercising it.

tearDown()
Hook method for deconstructing the test fixture after testing it.

test_drop_privileges_not_necessary()

test_drop_privileges_with_restore()

test_restore_privileges_as_effective_other()

test_restore_privileges_as_non_root()

test_restore_privileges_as_root()

dhcpkit.tests.common.server package

Tests for common server components

Submodules

dhcpkit.tests.common.server.test_config_datatypes module

Tests for datatypes for use in configuration files

class dhcpkit.tests.common.server.test_config_datatypes.**DomainNameTestCase**(*methodName='run*
Bases: unittest.case.TestCase

test_label_too_long()

test_name_too_long()

test_valid()

dhcpkit.tests.ipv6 package

Tests for the IPv6 DHCPv6 implementation go here

Subpackages

dhcpkit.tests.ipv6.extensions package

Tests for extensions to the base DHCPv6 protocol

Subpackages

dhcpkit.tests.ipv6.extensions.leasequery package

Tests for the Leasequery extension

Submodules

dhcpkit.tests.ipv6.extensions.leasequery.test_client_data_option module

Test the ClientDataOption implementation

class dhcpkit.tests.ipv6.extensions.leasequery.test_client_data_option.**ClientDataOptionT**
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_get_option_of_type()

test_get_options_of_type()

test_parse_wrong_type()

dhcpkit.tests.ipv6.extensions.leasequery.test_clt_time_option module

Test the CLTTimeOption implementation

class dhcpkit.tests.ipv6.extensions.leasequery.test_clt_time_option.**CLTTimeOptionTestCas**
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_parse_wrong_type()

test_validate_clt_time()

dhcpkit.tests.ipv6.extensions.leasequery.test_leasequery_message module

Test the LeasequeryMessage implementation

class dhcpkit.tests.ipv6.extensions.leasequery.test_leasequery_message.**LeasequeryMessage**
Bases: *dhcpkit.tests.ipv6.messages.test_message.MessageTestCase* (page 193)

setUp()
Hook method for setting up the test fixture before exercising it.

dhcpkit.tests.ipv6.extensions.leasequery.test_lq_client_link_option module

Test the LQClientLink implementation

class dhcpkit.tests.ipv6.extensions.leasequery.test_lq_client_link_option.**ClientDataOpti**
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_parse_wrong_type()

test_validate_link_addresses()

dhcpkit.tests.ipv6.extensions.leasequery.test_lq_query_option module

Test the LQQueryOption implementation

class dhcpkit.tests.ipv6.extensions.leasequery.test_lq_query_option.**LQQueryOptionTestCas**
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_display()

test_get_option_of_type()

test_get_options_of_type()

test_parse_wrong_type()

test_validate_link_address()

test_validate_query_type()

dhcpkit.tests.ipv6.extensions.leasequery.test_lq_relay_data_option module

Test the LQRelayDataOption implementation

class dhcpkit.tests.ipv6.extensions.leasequery.test_lq_relay_data_option.**ClientDataOptic**
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_parse_wrong_type()

test_test_wrong_message()

test_validate_peer_address()

Submodules

dhcpkit.tests.ipv6.extensions.test_bulk_leasequery module

Test the RelayIdOption implementation

class dhcpkit.tests.ipv6.extensions.test_bulk_leasequery.**RelayIdOptionTestCase**(*methodName*
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_validate_duid()

dhcpkit.tests.ipv6.extensions.test_client_fqdn module

Test the Client FQDN option implementations

class dhcpkit.tests.ipv6.extensions.test_client_fqdn.**ClientFQDNOptionTestCase**(*methodName=*
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_n_flag()

test_o_flag()

test_s_flag()

test_validate_domain_name()

dhcpkit.tests.ipv6.extensions.test_dns module

Test the DNS options implementations

class dhcpkit.tests.ipv6.extensions.test_dns.**DomainSearchListOptionTestCase**(*methodName='ru*
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_validate_search_list()

class dhcpkit.tests.ipv6.extensions.test_dns.**RecursiveNameServersOptionTestCase**(*methodNam*
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_validate_addresses()

dhcpkit.tests.ipv6.extensions.test_dslite module

Test the DS-Lite options implementations

class dhcpkit.tests.ipv6.extensions.test_dslite.**AFTRNameOptionTestCase**(*methodName='runTest'*)
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_validate_fqdn()

dhcpkit.tests.ipv6.extensions.test_echo_request_option module

Test the EchoRequestOption implementation

class dhcpkit.tests.ipv6.extensions.test_echo_request_option.**EchoRequestOptionTestCase**(*n*
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_display_requested_options()

test_validate_requested_options()

dhcpkit.tests.ipv6.extensions.test_linklayer_id module

Test the RemoteIdOption implementation

class dhcpkit.tests.ipv6.extensions.test_linklayer_id.**LinkLayerIdOptionTestCase**(*methodNam*
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_display()

test_link_layer_address()

test_link_layer_type()

dhcpkit.tests.ipv6.extensions.test_map module

Test the Prefix Delegation option implementation

class dhcpkit.tests.ipv6.extensions.test_map.**S46BROptionTestCase**(*methodName='runTest'*)
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_validate_br_address()

class dhcpkit.tests.ipv6.extensions.test_map.**S46DMROptionTestCase**(*methodName='runTest'*)
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_ipv6_prefix_length()

test_bad_option_length()

test_validate_dmr_prefix()

class dhcpkit.tests.ipv6.extensions.test_map.**S46LWContainerOptionTestCase**(*methodName='runT*
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

class dhcpkit.tests.ipv6.extensions.test_map.**S46MapEContainerOptionTestCase**(*methodName='ru*
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_get_option_of_type()

test_get_options_of_type()

class dhcpkit.tests.ipv6.extensions.test_map.**S46MapTContainerOptionTestCase**(*methodName='ru*
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_get_option_of_type()

test_get_options_of_type()

class dhcpkit.tests.ipv6.extensions.test_map.**S46PortParametersOptionTestCase**(*methodName='*
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_validate_combined_offset_psid_len()

test_validate_offset()

test_validate_psid()

test_validate_psid_len()

class dhcpkit.tests.ipv6.extensions.test_map.**S46RuleOptionTestCase**(*methodName='runTest'*)
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_ipv4_prefix_length()

test_bad_ipv6_prefix_length()

test_bad_option_length()

test_flags()

test_get_option_of_type()

test_get_options_of_type()

test_validate_ea_len()

test_validate_flags()

test_validate_ipv4_prefix()

test_validate_ipv6_prefix()

class dhcpkit.tests.ipv6.extensions.test_map.**S46V4V6BindingOptionTestCase**(*methodName='runT*
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_ipv6_prefix_length()

test_bad_option_length()

test_get_option_of_type()

test_get_options_of_type()

test_validate_ipv4_address()

test_validate_ipv6_prefix()

dhcpkit.tests.ipv6.extensions.test_ntp module

Test the NTP option implementation

class dhcpkit.tests.ipv6.extensions.test_ntp.**NTPMulticastAddressSubOptionTestCase**(*methodN*
Bases: *dhcpkit.tests.ipv6.extensions.test_ntp.NTPSubOptionTestCase* (page 189)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_config_datatype()

test_validate_address()

test_validate_value()

class dhcpkit.tests.ipv6.extensions.test_ntp.**NTPServerAddressSubOptionTestCase**(*methodName*
Bases: *dhcpkit.tests.ipv6.extensions.test_ntp.NTPSubOptionTestCase* (page 189)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_config_datatype()

test_validate_address()

test_validate_value()

class dhcpkit.tests.ipv6.extensions.test_ntp.**NTPServerFQDNSubOptionTestCase**(*methodName='ru*
Bases: *dhcpkit.tests.ipv6.extensions.test_ntp.NTPSubOptionTestCase* (page 189)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_config_datatype()

test_validate_fqdn()

test_validate_value()

class dhcpkit.tests.ipv6.extensions.test_ntp.**NTPServersOptionTestCase**(*methodName='runTest'*)
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_parse_wrong_type()

class dhcpkit.tests.ipv6.extensions.test_ntp.**NTPSubOptionTestCase**(*methodName='runTest'*)
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

parse_option()

setUp()
Hook method for setting up the test fixture before exercising it.

test_load_from_wrong_buffer()

class dhcpkit.tests.ipv6.extensions.test_ntp.**UnknownNTPSubOptionTestCase**(*methodName='runTes*
Bases: *dhcpkit.tests.ipv6.extensions.test_ntp.NTPSubOptionTestCase* (page 189)

test_validate_suboption_data()

test_validate_suboption_type()

test_validate_value()

dhcpkit.tests.ipv6.extensions.test_pd_exclude module

Test the PDExcludeOption implementation

class dhcpkit.tests.ipv6.extensions.test_pd_exclude.**PDExcludeOptionTestCase**(*methodName='ru*
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_prefix_length()

test_subnet_id()

dhcpkit.tests.ipv6.extensions.test_prefix_delegation module

Test the Prefix Delegation option implementation

class dhcpkit.tests.ipv6.extensions.test_prefix_delegation.**IAPDOptionTestCase**(*methodName=*
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_get_option_of_type()

test_get_options_of_type()

test_get_prefixes()

test_sort()

test_validate_iaid()

test_validate_t1()

test_validate_t2()

class dhcpkit.tests.ipv6.extensions.test_prefix_delegation.**IAPrefixOptionTestCase**(*methodN*
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_validate_address()

test_validate_preferred_lifetime()

test_validate_valid_lifetime()

dhcpkit.tests.ipv6.extensions.test_remote_id module

Test the RemoteIdOption implementation

class dhcpkit.tests.ipv6.extensions.test_remote_id.**RemoteIdOptionTestCase**(*methodName='runT*
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_enterprise_number()

test_remote_id()

dhcpkit.tests.ipv6.extensions.test_sip_servers module

Test the SIP options implementations

class dhcpkit.tests.ipv6.extensions.test_sip_servers.**SIPServersAddressListOptionTestCase**
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_validate_sip_servers()

class dhcpkit.tests.ipv6.extensions.test_sip_servers.**SIPServersDomainNameListOptionTestC**
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_validate_domain_names()

dhcpkit.tests.ipv6.extensions.test_sntp module

Test the SNTP options implementations

class dhcpkit.tests.ipv6.extensions.test_sntp.**SNTPServersOptionTestCase**(*methodName='runTest*
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_validate_sntp_servers()

dhcpkit.tests.ipv6.extensions.test_sol_max_rt module

Test the SolMaxRTOption and InfMaxRTOption option implementations

class dhcpkit.tests.ipv6.extensions.test_sol_max_rt.**InfMaxRTOptionTestCase**(*methodName='run*
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_validate_inf_max_rt()

class dhcpkit.tests.ipv6.extensions.test_sol_max_rt.**SolMaxRTOptionTestCase**(*methodName='run*
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_validate_sol_max_rt()

dhcpkit.tests.ipv6.extensions.test_subscriber_id module

Test the SubscriberIdOption implementation

class dhcpkit.tests.ipv6.extensions.test_subscriber_id.**SubscriberIdOptionTestCase**(*methodN*
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_subscriber_id()

dhcpkit.tests.ipv6.extensions.test_timezone module

Test the DNS options implementations

class dhcpkit.tests.ipv6.extensions.test_timezone.**PosixTimezoneOptionTestCase**(*methodName=*
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_validate_timezone()

class dhcpkit.tests.ipv6.extensions.test_timezone.**TZDBTimezoneOptionTestCase**(*methodName='*
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_validate_timezone()

dhcpkit.tests.ipv6.messages package

Tests for message types go here

Submodules

dhcpkit.tests.ipv6.messages.test_advertise_message module

Test the AdvertiseMessage implementation

class dhcpkit.tests.ipv6.messages.test_advertise_message.**AdvertiseMessageTestCase**(*methodN*
Bases: *dhcpkit.tests.ipv6.messages.test_client_server_message.ClientServerMessageTestCase* (page 192)

setUp()
Hook method for setting up the test fixture before exercising it.

dhcpkit.tests.ipv6.messages.test_client_server_message module

Test the ClientServerMessage implementation

class dhcpkit.tests.ipv6.messages.test_client_server_message.**ClientServerMessageTestCase**
Bases: *dhcpkit.tests.ipv6.messages.test_message.MessageTestCase* (page 193)

parse_packet()

setUp()
Hook method for setting up the test fixture before exercising it.

test_get_option_of_type()

test_get_options_of_type()

test_load_from_wrong_buffer()

test_validate_IAID_uniqueness()

test_validate_transaction_id()

dhcpkit.tests.ipv6.messages.test_confirm_message module

Test the RequestMessage implementation

class dhcpkit.tests.ipv6.messages.test_confirm_message.**RequestMessageTestCase**(*methodName=*
Bases: *dhcpkit.tests.ipv6.messages.test_client_server_message.ClientServerMessageTestCase* (page 192)

setUp()
Hook method for setting up the test fixture before exercising it.

dhcpkit.tests.ipv6.messages.test_message module

Test the Message implementation

class dhcpkit.tests.ipv6.messages.test_message.**MessageTestCase**(*methodName='runTest'*)
Bases: unittest.case.TestCase

check_unsigned_integer_property(*property_name: str*, *size: int*)
Perform basic verification of validation of an unsigned integer

Parameters

- **property_name** – The property under test
- **size** – The number of bits of this integer field

parse_packet()

setUp()
Hook method for setting up the test fixture before exercising it.

test_length()

test_parse()

test_save_fixture()

test_save_parsed()

test_validate()

dhcpkit.tests.ipv6.messages.test_relay_forward_message module

Test the RelayForwardMessage implementation

class dhcpkit.tests.ipv6.messages.test_relay_forward_message.**RelayedSolicitMessageTestCa**
Bases: *dhcpkit.tests.ipv6.messages.test_relay_server_message.RelayServerMessageTestCase* (page 193)

setUp()
Hook method for setting up the test fixture before exercising it.

test_wrap_response()

dhcpkit.tests.ipv6.messages.test_relay_reply_message module

Test the RelayReplyMessage implementation

class dhcpkit.tests.ipv6.messages.test_relay_reply_message.**RelayedAdvertiseMessageTestCa**
Bases: *dhcpkit.tests.ipv6.messages.test_relay_server_message.RelayServerMessageTestCase* (page 193)

setUp()
Hook method for setting up the test fixture before exercising it.

dhcpkit.tests.ipv6.messages.test_relay_server_message module

Test the RelayServerMessage implementation

class dhcpkit.tests.ipv6.messages.test_relay_server_message.**RelayServerMessageTestCase**(*m*
Bases: *dhcpkit.tests.ipv6.messages.test_message.MessageTestCase* (page 193)

parse_packet()

setUp()
Hook method for setting up the test fixture before exercising it.

test_empty_inner_message()

test_empty_relayed_message()

test_get_relayed_message()

test_inner_message()

test_inner_relay_message()

test_missing_inner_message()

test_set_relayed_message()

test_validate_hop_count()

test_validate_link_address()

test_validate_peer_address()

dhcpkit.tests.ipv6.messages.test_reply_message module

Test the ReplyMessage implementation

class dhcpkit.tests.ipv6.messages.test_reply_message.**ReplyMessageTestCase**(*methodName='runT*
Bases: *dhcpkit.tests.ipv6.messages.test_client_server_message.ClientServerMessageTestCase* (page 192)

setUp()
Hook method for setting up the test fixture before exercising it.

dhcpkit.tests.ipv6.messages.test_request_message module

Test the RequestMessage implementation

class dhcpkit.tests.ipv6.messages.test_request_message.**RequestMessageTestCase**(*methodName=*
Bases: *dhcpkit.tests.ipv6.messages.test_client_server_message.ClientServerMessageTestCase* (page 192)

setUp()
Hook method for setting up the test fixture before exercising it.

dhcpkit.tests.ipv6.messages.test_solicit_message module

Test the SolicitMessage implementation

class dhcpkit.tests.ipv6.messages.test_solicit_message.**SolicitMessageTestCase**(*methodName=*
Bases: *dhcpkit.tests.ipv6.messages.test_client_server_message.ClientServerMessageTestCase* (page 192)

setUp()
Hook method for setting up the test fixture before exercising it.

dhcpkit.tests.ipv6.messages.test_unknown_message module

Test the UnknownMessage implementation

class dhcpkit.tests.ipv6.messages.test_unknown_message.**UnknownMessageTestCase**(*methodName=*
Bases: *dhcpkit.tests.ipv6.messages.test_message.MessageTestCase* (page 193)

parse_packet()

setUp()
Hook method for setting up the test fixture before exercising it.

test_validate_data()

test_validate_message_type()

dhcpkit.tests.ipv6.options package

All the different options are tested here

Submodules

dhcpkit.tests.ipv6.options.test_authentication_option module

Test the ElapsedTimeOption implementation

class dhcpkit.tests.ipv6.options.test_authentication_option.**AuthenticationOptionTestCase**
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_auth_info()

test_bad_option_length()

test_replay_detection()

test_validate_algorithm()

test_validate_protocol()

test_validate_rdm()

dhcpkit.tests.ipv6.options.test_client_id_option module

Test the ClientIdOption implementation

class dhcpkit.tests.ipv6.options.test_client_id_option.**ClientIdOptionTestCase**(*methodName=*
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_validate_duid()

dhcpkit.tests.ipv6.options.test_elapsed_time_option module

Test the ElapsedTimeOption implementation

class dhcpkit.tests.ipv6.options.test_elapsed_time_option.**ElapsedTimeOptionTestCase**(*metho*
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_validate_elapsed_time()

dhcpkit.tests.ipv6.options.test_ia_address_option module

Test the IAAddressOption implementation

class dhcpkit.tests.ipv6.options.test_ia_address_option.**IAAddressOptionTestCase**(*methodNam*
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_validate_address()

test_validate_preferred_lifetime()

test_validate_valid_lifetime()

dhcpkit.tests.ipv6.options.test_ia_na_option module

Test the IANAOption implementation

class dhcpkit.tests.ipv6.options.test_ia_na_option.**IANAOptionTestCase**(*methodName='runTest'*)
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_get_addresses()

test_get_option_of_type()

test_get_options_of_type()

test_sort()

test_validate_iaid()

test_validate_t1()

test_validate_t2()

dhcpkit.tests.ipv6.options.test_ia_ta_option module

Test the IATAOption implementation

class dhcpkit.tests.ipv6.options.test_ia_ta_option.**IATAOptionTestCase**(*methodName='runTest'*)
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_get_addresses()

test_get_option_of_type()

test_get_options_of_type()

test_sort()

test_validate_iaid()

dhcpkit.tests.ipv6.options.test_interface_id_option module

Test the InterfaceIdOption implementation

class dhcpkit.tests.ipv6.options.test_interface_id_option.**UnknownOptionTestCase**(*methodNam*
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_interface_id()

dhcpkit.tests.ipv6.options.test_option module

Test the basic option implementation

class dhcpkit.tests.ipv6.options.test_option.**OptionTestCase**(*methodName='runTest'*)
Bases: unittest.case.TestCase

check_integer_property_range(*property_name: str*, *min_value: int = None*, *max_value: int = None*)
Perform basic verification of validation of an integer range

Parameters

- **property_name** – The property under test
- **min_value** – The minimum value allowed
- **max_value** – The maximum value allowed

check_unsigned_integer_property(*property_name: str*, *size: int = None*)
Perform basic verification of validation of an unsigned integer

Parameters

- **property_name** – The property under test
- **size** – The number of bits of this integer field

parse_option()

setUp()
Hook method for setting up the test fixture before exercising it.

test_length()

test_load_from_wrong_buffer()

test_overflow()

test_parse()

test_save_fixture()

test_save_parsed()

test_validate()

dhcpkit.tests.ipv6.options.test_option_length module

Test the implementation of option length checking

class dhcpkit.tests.ipv6.options.test_option_length.**LengthTestingOption**(*data: bytes = b''*)

Bases: *dhcpkit.ipv6.options.Option* (page 168)

Fake DHCPv6 option for testing length checks

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int

Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

option_type = 65535

save() → Union

Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

class dhcpkit.tests.ipv6.options.test_option_length.**RelayMessageOptionTestCase**(*methodName*

Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()

Hook method for setting up the test fixture before exercising it.

tearDown()

Hook method for deconstructing the test fixture after testing it.

test_bad_option_length()

dhcpkit.tests.ipv6.options.test_option_request_option module

Test the OptionRequestOption implementation

class dhcpkit.tests.ipv6.options.test_option_request_option.**OptionRequestOptionTestCase**

Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()

Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_display_requested_options()

test_validate_requested_options()

dhcpkit.tests.ipv6.options.test_preference_option module

Test the PreferenceOption implementation

class dhcpkit.tests.ipv6.options.test_preference_option.**PreferenceOptionTestCase**(*methodNa*
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_validate_preference()

dhcpkit.tests.ipv6.options.test_rapid_commit_option module

Test the RapidCommitOption implementation

class dhcpkit.tests.ipv6.options.test_rapid_commit_option.**RapidCommitOptionTestCase**(*meth*
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

dhcpkit.tests.ipv6.options.test_reconfigure_accept_option module

Test the ReconfigureAcceptOption implementation

class dhcpkit.tests.ipv6.options.test_reconfigure_accept_option.**ReconfigureAcceptOption**
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

dhcpkit.tests.ipv6.options.test_reconfigure_message_option module

Test the ReconfigureMessageOption implementation

class dhcpkit.tests.ipv6.options.test_reconfigure_message_option.**ReconfigureMessageOptio**
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_message_type()

dhcpkit.tests.ipv6.options.test_relay_message_option module

Test the RelayMessageOption implementation

class dhcpkit.tests.ipv6.options.test_relay_message_option.**NonRelayableMessage**(*message_typ*
int
=
0,
mes-
sage_data:
bytes
=
b”)

Bases: *dhcpkit.ipv6.messages.UnknownMessage* (page 158)

A message that can not be relayed

class dhcpkit.tests.ipv6.options.test_relay_message_option.**RelayMessageOptionTestCase**(*me*

Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()

Hook method for setting up the test fixture before exercising it.

test_bad_message_length()

test_validate_relayed_message()

class dhcpkit.tests.ipv6.options.test_relay_message_option.**WeirdLengthMessage**(*transaction_id*
bytes
=
b’x00x00x00’,
op-
tions:
It-
er-
able
=
None)

Bases: *dhcpkit.ipv6.messages.ClientServerMessage* (page 152)

An option that returns an incorrect length, to test error handling

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*)

Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

message_type = 254

dhcpkit.tests.ipv6.options.test_server_id_option module

Test the ServerIdOption implementation

class dhcpkit.tests.ipv6.options.test_server_id_option.**ServerIdOptionTestCase**(*methodName=*

Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()

Hook method for setting up the test fixture before exercising it.

test_validate_duid()

dhcpkit.tests.ipv6.options.test_server_unicast_option module

Test the ServerUnicastOption implementation

class dhcpkit.tests.ipv6.options.test_server_unicast_option.**ServerUnicastOptionTestCase**
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_server_address()

dhcpkit.tests.ipv6.options.test_status_code_option module

Test the StatusCodeOption implementation

class dhcpkit.tests.ipv6.options.test_status_code_option.**StatusCodeOptionTestCase**(*methodN*
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_display()

test_status_code()

test_status_message()

dhcpkit.tests.ipv6.options.test_unknown_option module

Test the UnknownOption implementation

class dhcpkit.tests.ipv6.options.test_unknown_option.**UnknownOptionTestCase**(*methodName='run*
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_validate_data()

test_validate_option_type()

test_validate_type()

dhcpkit.tests.ipv6.options.test_user_class_option module

Test the UserClassOption implementation

class dhcpkit.tests.ipv6.options.test_user_class_option.**UserClassOptionTestCase**(*methodNam*
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_user_classes()

dhcpkit.tests.ipv6.options.test_vendor_class_option module

Test the VendorClassOption implementation

class dhcpkit.tests.ipv6.options.test_vendor_class_option.**VendorClassOptionTestCase**(*meth*
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_enterprise_number()

test_vendor_classes()

dhcpkit.tests.ipv6.options.test_vendor_specific_information_option module

Test the VendorSpecificInformationOption implementation

class dhcpkit.tests.ipv6.options.test_vendor_specific_information_option.**VendorSpecificI**
Bases: *dhcpkit.tests.ipv6.options.test_option.OptionTestCase* (page 197)

setUp()
Hook method for setting up the test fixture before exercising it.

test_bad_option_length()

test_enterprise_number()

test_vendor_options()

dhcpkit.tests.ipv6.server package

Tests for IPv6 server components

Subpackages

dhcpkit.tests.ipv6.server.handlers package

Tests for server handlers

Submodules

dhcpkit.tests.ipv6.server.handlers.test_echo_request_option_handler module

Tests for a relay message handler

class dhcpkit.tests.ipv6.server.handlers.test_echo_request_option_handler.**RelayHandlerTe**
Bases: unittest.case.TestCase

setUp()
Hook method for setting up the test fixture before exercising it.

test_absent_option_echo_request()

test_empty_echo_request()

test_remote_id_echo_request()

test_unnecessary_echo_request()

dhcpkit.tests.ipv6.server.handlers.test_handler module

Basic handler testing

class dhcpkit.tests.ipv6.server.handlers.test_handler.**HandlerTestCase**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_str()

class dhcpkit.tests.ipv6.server.handlers.test_handler.**TestHandler**
Bases: *dhcpkit.ipv6.server.handlers.Handler* (page 121)

A handler that doesn't do anything

dhcpkit.tests.ipv6.server.handlers.test_relay_handler module

Tests for a relay message handler

class dhcpkit.tests.ipv6.server.handlers.test_relay_handler.**RelayHandlerTestCase**(*methodNa*
Bases: unittest.case.TestCase

test_str()

class dhcpkit.tests.ipv6.server.handlers.test_relay_handler.**TestRelayHandler**
Bases: *dhcpkit.ipv6.server.handlers.RelayHandler* (page 121)

A relay handler that doesn't do anything

handle_relay(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*, *relay_message_in: dhcpkit.ipv6.messages.RelayForwardMessage*, *relay_message_out: dhcpkit.ipv6.messages.RelayReplyMessage*)
Handler implementation that doesn't do anything

Submodules

dhcpkit.tests.ipv6.server.test_message_handler module

Testing of the message handler

class dhcpkit.tests.ipv6.server.test_message_handler.**BadExceptionHandler**
Bases: *dhcpkit.ipv6.server.handlers.Handler* (page 121)

A handler that raises a bogus exception

pre(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Raise UseMulticastError on multicast messages... This is intentionally wrong.

class dhcpkit.tests.ipv6.server.test_message_handler.**DummyExtension**
Bases: object[160]

A server extension that adds the DummyMarksHandler at both setup and cleanup

static create_cleanup_handlers()
Add the DummyMarksHandler at cleanup

static create_setup_handlers()
Add the DummyMarksHandler at setup

class dhcpkit.tests.ipv6.server.test_message_handler.**DummyMarksHandler**(*mark: str*)
Bases: *dhcpkit.ipv6.server.handlers.Handler* (page 121)

A handler that sets marks in each of the phases of message handling

[160] https://docs.python.org/3.4/library/functions.html#object

handle(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Add a mark to show we have been here

post(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Add a mark to show we have been here

pre(*bundle: dhcpkit.ipv6.server.transaction_bundle.TransactionBundle*)
Add a mark to show we have been here

class dhcpkit.tests.ipv6.server.test_message_handler.**MessageHandlerTestCase**(*methodName='ru*
Bases: unittest.case.TestCase

setUp()
Hook method for setting up the test fixture before exercising it.

test_accept_unicast_message()

test_badly_rejected_multicast_message()

test_confirm_message()

test_empty_confirm_message()

test_empty_message()

test_ignorable_multicast_message()

test_not_implemented_message()

test_rapid_solicit_message()

test_reject_unicast_message()

test_request_message()

test_solicit_message()

test_very_rapid_solicit_message()

test_worker_init()

dhcpkit.tests.ipv6.server.test_transaction_bundle module

Test transaction bundle

class dhcpkit.tests.ipv6.server.test_transaction_bundle.**TransactionBundleTestCase**(*methodN*
Bases: unittest.case.TestCase

setUp()
Hook method for setting up the test fixture before exercising it.

test_auto_create_outgoing_relay_messages()

test_bad_response()

test_direct_outgoing_message()

test_get_unhandled_options()

test_incoming_relay_messages()

test_link_address()

test_mark_handled()

test_marks()

test_no_outgoing_message()

test_outgoing_message()

test_request()

test_shallow_bundle()

test_str()

test_unanswered_iana_options()

test_unanswered_iapd_options()

test_unanswered_iata_options()

test_unknown_message()

test_wrong_way()

Submodules

dhcpkit.tests.ipv6.test_duids module

Test the included DUID types

class dhcpkit.tests.ipv6.test_duids.**EnterpriseDUIDTestCase**(*methodName='runTest'*)
Bases: *dhcpkit.tests.ipv6.test_duids.UnknownDUIDTestCase* (page 205)

setUp()
Hook method for setting up the test fixture before exercising it.

test_validate_enterprise_number()

test_validate_identifier()

test_validate_length()

test_wrong_parser()

class dhcpkit.tests.ipv6.test_duids.**LinkLayerDUIDTestCase**(*methodName='runTest'*)
Bases: *dhcpkit.tests.ipv6.test_duids.UnknownDUIDTestCase* (page 205)

setUp()
Hook method for setting up the test fixture before exercising it.

test_display_ethernet()

test_display_other()

test_validate_hardware_type()

test_validate_length()

test_validate_link_layer()

test_wrong_parser()

class dhcpkit.tests.ipv6.test_duids.**LinkLayerTimeDUIDTestCase**(*methodName='runTest'*)
Bases: *dhcpkit.tests.ipv6.test_duids.UnknownDUIDTestCase* (page 205)

setUp()
Hook method for setting up the test fixture before exercising it.

test_display_ethernet()

test_display_other()

test_validate_hardware_type()

test_validate_length()

test_validate_link_layer()

test_validate_time()

test_wrong_parser()

class dhcpkit.tests.ipv6.test_duids.**UnknownDUIDTestCase**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()
Hook method for setting up the test fixture before exercising it.

test_hash()

test_parse()

test_parse_with_larger_buffer()

test_save()

dhcpkit.tests.ipv6.test_utils module

Test the IPv6 utility functions

class dhcpkit.tests.ipv6.test_utils.**IPv6UtilsTestCase**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_address_in_prefixes()

test_is_global_unicast()

test_prefix_overlaps_prefixes()

dhcpkit.tests.utils package

Tests for generic utility functions go here

Submodules

dhcpkit.tests.utils.test_camelcase module

Test the camelcase conversion functions

class dhcpkit.tests.utils.test_camelcase.**CamelCaseTestCase**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_camelcase_to_dash()

test_camelcase_to_underscore()

dhcpkit.tests.utils.test_domain_name module

Test the encoding and parsing of domain names

class dhcpkit.tests.utils.test_domain_name.**DomainNameListTestCase**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()
Hook method for setting up the test fixture before exercising it.

test_encode_good()

test_parse_good()

class dhcpkit.tests.utils.test_domain_name.**DomainNameTestCase**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()
Hook method for setting up the test fixture before exercising it.

test_encode_good()

test_encode_idn()

test_encode_idn_oversized_label()

test_encode_oversized_domain()

test_encode_oversized_label()

test_encode_relative()

test_parse_buffer_overflow()

test_parse_good()

test_parse_idn()

test_parse_idn_oversized_label()

test_parse_oversized_domain()

test_parse_oversized_label()

test_parse_oversized_relative_domain()

test_parse_relative()

test_parse_unending()

class dhcpkit.tests.utils.test_domain_name.**ValidateDomainLabelTestCase**(*methodName='runTest'*)

Bases: unittest.case.TestCase

test_validate_correct_labels()

test_validate_empty_label()

test_validate_invalid_label()

test_validate_oversized_label()

dhcpkit.tests.utils.test_normalise_hex module

Test the camelcase conversion functions

class dhcpkit.tests.utils.test_normalise_hex.**NormaliseHexTestCase**(*methodName='runTest'*)

Bases: unittest.case.TestCase

test_bad_hex()

test_hex()

test_hex_with_colons()

Submodules

dhcpkit.tests.test_protocol_element module

Test whether the basic stuff of ProtocolElement works as intended

class dhcpkit.tests.test_protocol_element.**AnythingContainerElement**(*elements: Iterable*)

Bases: *dhcpkit.tests.test_protocol_element.ContainerElementBase* (page 207)

Container that may contain as many as it wants

class dhcpkit.tests.test_protocol_element.**BadDemoElement**
Bases: *dhcpkit.tests.test_protocol_element.DemoElementBase* (page 207)

Sub-element to test with

class dhcpkit.tests.test_protocol_element.**ContainerElementBase**(*elements: Iterable*)
Bases: *dhcpkit.tests.test_protocol_element.DemoElementBase* (page 207)

A simple element that contains DemoElements

validate()
Validate the contents of this element

class dhcpkit.tests.test_protocol_element.**DemoElement**
Bases: *dhcpkit.tests.test_protocol_element.DemoElementBase* (page 207)

Sub-element to test with

class dhcpkit.tests.test_protocol_element.**DemoElementBase**
Bases: *dhcpkit.protocol_element.ProtocolElement* (page 218)

A simple element to test with

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Intentionally left empty. Specific implementations must be tested separately.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

save() → Union
Intentionally left empty. Specific implementations must be tested separately.

Returns The buffer with the data from this element

class dhcpkit.tests.test_protocol_element.**ElementOccurrenceTestCase**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_anything_0()

test_anything_1()

test_anything_2()

test_bad()

test_class_based()

test_compare()

test_element_class_case_less_specific()

test_element_class_case_more_specific()

test_element_class_forbidden()

test_element_class_missing()

test_element_class_superclasses_less_specific()

test_element_class_superclasses_more_specific()

test_exactly_one_0()

test_exactly_one_1()

test_exactly_one_2()

test_exactly_two_1()

test_exactly_two_2()

test_exactly_two_3()

test_max_one_0()

test_max_one_1()

test_max_one_2()

test_min_one_0()

test_min_one_1()

test_min_one_2()

test_nothing_0()

test_nothing_1()

test_nothing_2()

test_repr()

test_str_no_parameters()

test_str_one_parameter()

test_str_one_parameter_display()

test_str_one_parameter_display_hidden()

test_str_one_parameter_display_hidden_string()

test_str_three_parameters()

test_str_two_parameters()

test_str_two_parameters_display()

test_str_two_parameters_display_hidden()

test_str_two_parameters_display_hidden_string()

class dhcpkit.tests.test_protocol_element.**ExactlyOneContainerElement**(*elements: Iterable*)

Bases: *dhcpkit.tests.test_protocol_element.ContainerElementBase* (page 207)

Container that must contain exactly one sub-element

class dhcpkit.tests.test_protocol_element.**ExactlyTwoContainerElement**(*elements: Iterable*)

Bases: *dhcpkit.tests.test_protocol_element.ContainerElementBase* (page 207)

Container that must contain exactly two sub-elements

class dhcpkit.tests.test_protocol_element.**HardCodedContainerElement**(*elements: Iterable*)

Bases: *dhcpkit.tests.test_protocol_element.ContainerElementBase* (page 207)

Container that will have its _may_contain class property overwritten in the test

class dhcpkit.tests.test_protocol_element.**JSONEncodingTestCase**(*methodName='runTest'*)

Bases: unittest.case.TestCase

test_str_no_parameters()

test_str_one_parameter()

test_str_three_parameters()

test_str_two_parameters()

class dhcpkit.tests.test_protocol_element.**MaxOneContainerElement**(*elements: Iterable*)
Bases: *dhcpkit.tests.test_protocol_element.ContainerElementBase* (page 207)

Container that must contain at most one sub-element

class dhcpkit.tests.test_protocol_element.**MinOneContainerElement**(*elements: Iterable*)
Bases: *dhcpkit.tests.test_protocol_element.ContainerElementBase* (page 207)

Container that must contain at least one sub-element

class dhcpkit.tests.test_protocol_element.**NothingContainerElement**(*elements: Iterable*)
Bases: *dhcpkit.tests.test_protocol_element.ContainerElementBase* (page 207)

Container that may contain as many as it wants

class dhcpkit.tests.test_protocol_element.**OneParameterDemoElement**(*one*)
Bases: *dhcpkit.tests.test_protocol_element.DemoElementBase* (page 207)

Sub-element to test with

class dhcpkit.tests.test_protocol_element.**OneParameterDisplayDemoElement**(*one*)
Bases: *dhcpkit.tests.test_protocol_element.DemoElementBase* (page 207)

Sub-element to test with

display_one()
Nicer display for property one

class dhcpkit.tests.test_protocol_element.**OneParameterDisplayHiddenDemoElement**(*one*)
Bases: *dhcpkit.tests.test_protocol_element.DemoElementBase* (page 207)

Sub-element to test with

display_one = **HIDDEN**

class dhcpkit.tests.test_protocol_element.**OneParameterDisplayHiddenStringDemoElement**(*one*
Bases: *dhcpkit.tests.test_protocol_element.DemoElementBase* (page 207)

Sub-element to test with

display_one = 'HIDDEN**'**

class dhcpkit.tests.test_protocol_element.**ProtocolElementTestCase**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_determine_class()

class dhcpkit.tests.test_protocol_element.**ThreeParameterDemoElement**(*one: int, two: str, three: Iterable*)
Bases: *dhcpkit.tests.test_protocol_element.DemoElementBase* (page 207)

Sub-element to test with

class dhcpkit.tests.test_protocol_element.**TwoParameterDemoElement**(*one: int*, *two: dhcpkit.tests.test_protocol_element.D*

Bases: *dhcpkit.tests.test_protocol_element.DemoElementBase* (page 207)

Sub-element to test with

class dhcpkit.tests.test_protocol_element.**TwoParameterDisplayDemoElement**(*one: int*, *two: dhcpkit.tests.test_protocol_*

Bases: *dhcpkit.tests.test_protocol_element.DemoElementBase* (page 207)

Sub-element to test with

display_one()
Nicer display for property one

class dhcpkit.tests.test_protocol_element.**TwoParameterDisplayHiddenDemoElement**(*one: int*, *two: dhcpkit.tests.test_*

Bases: *dhcpkit.tests.test_protocol_element.DemoElementBase* (page 207)

Sub-element to test with

display_one = **HIDDEN**

class dhcpkit.tests.test_protocol_element.**TwoParameterDisplayHiddenStringDemoElement**(*one*
int,
two
dhc
kit.t

Bases: *dhcpkit.tests.test_protocol_element.DemoElementBase* (page 207)

Sub-element to test with

display_one = 'HIDDEN**'**

class dhcpkit.tests.test_protocol_element.**UnknownProtocolElementTestCase**(*methodName='runTes*
Bases: unittest.case.TestCase

test_load_from()

test_save()

dhcpkit.tests.test_registry module

Test whether the basic stuff of Registry works as intended

class dhcpkit.tests.test_registry.**ElementOccurrenceTestCase**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_bad_entry()

test_duplicate_entries()

test_registry_loading()

test_version_mismatch()

class dhcpkit.tests.test_registry.**TestRegistry**
Bases: *dhcpkit.registry.Registry* (page 220)

A registry that doesn't exist to test with

entry_point = 'dhcpkit.tests.registry'

dhcpkit.typing package

This provides a backwards-compatibility layer for the Python typing system as described in PEP484

Submodules

dhcpkit.typing.py352_typing module

class dhcpkit.typing.py352_typing.**Any**
Bases: dhcpkit.typing.py352_typing.Final

Special type indicating an unconstrained type.

- Any object is an instance of Any.
- Any class is a subclass of Any.
- As a special case, Any and object are subclasses of each other.

class dhcpkit.typing.py352_typing.**Callable**
Bases: dhcpkit.typing.py352_typing.Final

Callable type; Callable[[int], str] is a function of (int) -> str.

The subscription syntax must always be used with exactly two values: the argument list and the return type. The argument list must be a list of types; the return type must be a single type.

There is no syntax to indicate optional or keyword arguments, such function types are rarely used as callback types.

class dhcpkit.typing.py352_typing.**Generic**
Bases: object[161]

Abstract base class for generic types.

A generic type is typically declared by inheriting from an instantiation of this class with one or more type variables. For example, a generic mapping type might be defined as:

```
class Mapping(Generic[KT, VT]):
    def __getitem__(self, key: KT) -> VT:
        ...
    # Etc.
```

This class can then be used as follows:

```
def lookup_name(mapping: Mapping[KT, VT], key: KT, default: VT) -> VT:
    try:
        return mapping[key]
    except KeyError:
        return default
```

class dhcpkit.typing.py352_typing.**Optional**
Bases: dhcpkit.typing.py352_typing.Final

Optional type.

Optional[X] is equivalent to Union[X, type(None)].

[161] https://docs.python.org/3.4/library/functions.html#object

class dhcpkit.typing.py352_typing.**Tuple**
Bases: dhcpkit.typing.py352_typing.Final

Tuple type; Tuple[X, Y] is the cross-product type of X and Y.

Example: Tuple[T1, T2] is a tuple of two elements corresponding to type variables T1 and T2. Tuple[int, float, str] is a tuple of an int, a float and a string.

To specify a variable-length tuple of homogeneous type, use Sequence[T].

class dhcpkit.typing.py352_typing.**Type**
Bases: type[162], *dhcpkit.typing.py352_typing.Generic* (page 212)

A special construct usable to annotate class objects.

For example, suppose we have the following classes:

```
class User: ...  # Abstract base for User classes
class BasicUser(User): ...
class ProUser(User): ...
class TeamUser(User): ...
```

And a function that takes a class argument that's a subclass of User and returns an instance of the corresponding class:

```
U = TypeVar('U', bound=User)
def new_user(user_class: Type[U]) -> U:
    user = user_class()
    # (Here we could write the user object to a database)
    return user

joe = new_user(BasicUser)
```

At this point the type checker knows that joe has type BasicUser.

class dhcpkit.typing.py352_typing.**TypeVar**(**args*, ***kwds*)
Bases: dhcpkit.typing.py352_typing.TypingMeta

Type variable.

Usage:

```
T = TypeVar('T')  # Can be anything
A = TypeVar('A', str, bytes)  # Must be str or bytes
```

Type variables exist primarily for the benefit of static type checkers. They serve as the parameters for generic types as well as for generic function definitions. See class Generic for more information on generic types. Generic functions work as follows:

def repeat(x: T, n: int) -> Sequence[T]: ''Return a list containing n references to x.'' return [x]*n

def longest(x: A, y: A) -> A: ''Return the longest of two strings.'' return x if len(x) >= len(y) else y

The latter example's signature is essentially the overloading of (str, str) -> str and (bytes, bytes) -> bytes. Also note that if the arguments are instances of some subclass of str, the return type is still plain str.

At runtime, isinstance(x, T) will raise TypeError. However, issubclass(C, T) is true for any class C, and issubclass(str, A) and issubclass(bytes, A) are true, and issubclass(int, A) is false. (TODO: Why is this needed? This may change. See #136.)

Type variables may be marked covariant or contravariant by passing covariant=True or contravariant=True. See PEP 484 for more details. By default type variables are invariant.

Type variables can be introspected. e.g.:

[162] https://docs.python.org/3.4/library/functions.html#type

T.__name__ == 'T' T.__constraints__ == () T.__covariant__ == False T.__contravariant__ = False A.__constraints__ == (str, bytes)

class dhcpkit.typing.py352_typing.**Union**
Bases: dhcpkit.typing.py352_typing.Final

Union type; Union[X, Y] means either X or Y.

To define a union, use e.g. Union[int, str]. Details:

- The arguments must be types and there must be at least one.
- None as an argument is a special case and is replaced by type(None).
- Unions of unions are flattened, e.g.:

```
Union[Union[int, str], float] == Union[int, str, float]
```

- Unions of a single argument vanish, e.g.:

```
Union[int] == int  # The constructor actually returns int
```

- Redundant arguments are skipped, e.g.:

```
Union[int, str, int] == Union[int, str]
```

- When comparing unions, the argument order is ignored, e.g.:

```
Union[int, str] == Union[str, int]
```

- When two arguments have a subclass relationship, the least derived argument is kept, e.g.:

```
class Employee: pass
class Manager(Employee): pass
Union[int, Employee, Manager] == Union[int, Employee]
Union[Manager, int, Employee] == Union[int, Employee]
Union[Employee, Manager] == Employee
```

- Corollary: if Any is present it is the sole survivor, e.g.:

```
Union[int, Any] == Any
```

- Similar for object:

```
Union[int, object] == object
```

- To cut a tie: Union[object, Any] == Union[Any, object] == Any.
- You cannot subclass or instantiate a union.
- You cannot write Union[X][Y] (what would it mean?).
- You can use Optional[X] as a shorthand for Union[X, None].

class dhcpkit.typing.py352_typing.**AbstractSet**
Bases: collections.abc.Sized[163], *dhcpkit.typing.py352_typing.Iterable* (page 214), *dhcpkit.typing.py352_typing.Container* (page 214)

class dhcpkit.typing.py352_typing.**Awaitable**
Bases: *dhcpkit.typing.py352_typing.Generic* (page 212)

class dhcpkit.typing.py352_typing.**AsyncIterator**
Bases: *dhcpkit.typing.py352_typing.AsyncIterable* (page 214)

class dhcpkit.typing.py352_typing.**AsyncIterable**
Bases: *dhcpkit.typing.py352_typing.Generic* (page 212)

[163] https://docs.python.org/3.4/library/collections.abc.html#collections.abc.Sized

class dhcpkit.typing.py352_typing.**ByteString**
Bases: *dhcpkit.typing.py352_typing.Sequence* (page 215)

class dhcpkit.typing.py352_typing.**Container**
Bases: *dhcpkit.typing.py352_typing.Generic* (page 212)

class dhcpkit.typing.py352_typing.**Hashable**
Bases: object[164]

class dhcpkit.typing.py352_typing.**ItemsView**
Bases: *dhcpkit.typing.py352_typing.MappingView* (page 215), *dhcpkit.typing.py352_typing.AbstractSet* (page 214), *dhcpkit.typing.py352_typing.Generic* (page 212)

class dhcpkit.typing.py352_typing.**Iterable**
Bases: *dhcpkit.typing.py352_typing.Generic* (page 212)

class dhcpkit.typing.py352_typing.**Iterator**
Bases: *dhcpkit.typing.py352_typing.Iterable* (page 214)

class dhcpkit.typing.py352_typing.**KeysView**
Bases: *dhcpkit.typing.py352_typing.MappingView* (page 215), *dhcpkit.typing.py352_typing.AbstractSet* (page 214)

class dhcpkit.typing.py352_typing.**Mapping**
Bases: collections.abc.Sized[165], *dhcpkit.typing.py352_typing.Iterable* (page 214), *dhcpkit.typing.py352_typing.Container* (page 214), *dhcpkit.typing.py352_typing.Generic* (page 212)

class dhcpkit.typing.py352_typing.**MappingView**
Bases: collections.abc.Sized[166], *dhcpkit.typing.py352_typing.Iterable* (page 214)

class dhcpkit.typing.py352_typing.**MutableMapping**
Bases: *dhcpkit.typing.py352_typing.Mapping* (page 214)

class dhcpkit.typing.py352_typing.**MutableSequence**
Bases: *dhcpkit.typing.py352_typing.Sequence* (page 215)

class dhcpkit.typing.py352_typing.**MutableSet**
Bases: *dhcpkit.typing.py352_typing.AbstractSet* (page 214)

class dhcpkit.typing.py352_typing.**Sequence**
Bases: collections.abc.Sized[167], *dhcpkit.typing.py352_typing.Iterable* (page 214), *dhcpkit.typing.py352_typing.Container* (page 214)

class dhcpkit.typing.py352_typing.**Sized**
Bases: object[168]

class dhcpkit.typing.py352_typing.**ValuesView**
Bases: *dhcpkit.typing.py352_typing.MappingView* (page 215)

class dhcpkit.typing.py352_typing.**Reversible**
Bases: dhcpkit.typing.py352_typing._Protocol

class dhcpkit.typing.py352_typing.**SupportsAbs**
Bases: dhcpkit.typing.py352_typing._Protocol

class dhcpkit.typing.py352_typing.**SupportsFloat**
Bases: dhcpkit.typing.py352_typing._Protocol

[164] https://docs.python.org/3.4/library/functions.html#object
[165] https://docs.python.org/3.4/library/collections.abc.html#collections.abc.Sized
[166] https://docs.python.org/3.4/library/collections.abc.html#collections.abc.Sized
[167] https://docs.python.org/3.4/library/collections.abc.html#collections.abc.Sized
[168] https://docs.python.org/3.4/library/functions.html#object

class dhcpkit.typing.py352_typing.**SupportsInt**
Bases: dhcpkit.typing.py352_typing._Protocol

class dhcpkit.typing.py352_typing.**SupportsRound**
Bases: dhcpkit.typing.py352_typing._Protocol

class dhcpkit.typing.py352_typing.**Dict**
Bases: dict[169], *dhcpkit.typing.py352_typing.MutableMapping* (page 215)

class dhcpkit.typing.py352_typing.**DefaultDict**
Bases: collections.defaultdict[170], *dhcpkit.typing.py352_typing.MutableMapping* (page 215)

class dhcpkit.typing.py352_typing.**List**
Bases: list[171], *dhcpkit.typing.py352_typing.MutableSequence* (page 215)

class dhcpkit.typing.py352_typing.**Set**
Bases: set[172], *dhcpkit.typing.py352_typing.MutableSet* (page 215)

dhcpkit.typing.py352_typing.**NamedTuple**(*typename*, *fields*)
Typed version of namedtuple.

Usage:

```
Employee = typing.NamedTuple('Employee', [('name', str), 'id', int)])
```

This is equivalent to:

```
Employee = collections.namedtuple('Employee', ['name', 'id'])
```

The resulting class has one extra attribute: _field_types, giving a dict mapping field names to types. (The field names are in the _fields attribute, which is part of the namedtuple API.)

class dhcpkit.typing.py352_typing.**Generator**
Bases: *dhcpkit.typing.py352_typing.Iterator* (page 214), *dhcpkit.typing.py352_typing.Generic* (page 212)

class dhcpkit.typing.py352_typing.**AnyStr**
Bases: dhcpkit.typing.py352_typing.Final

dhcpkit.typing.py352_typing.**cast**(*typ*, *val*)
Cast a value to a type.

This returns the value unchanged. To the type checker this signals that the return value has the designated type, but at runtime we intentionally don't check anything (we want this to be as fast as possible).

dhcpkit.typing.py352_typing.**get_type_hints**(*obj*, *globalns=None*, *localns=None*)
Return type hints for a function or method object.

This is often the same as obj.__annotations__, but it handles forward references encoded as string literals, and if necessary adds Optional[t] if a default value equal to None is set.

BEWARE – the behavior of globalns and localns is counterintuitive (unless you are familiar with how eval() and exec() work). The search order is locals first, then globals.

- If no dict arguments are passed, an attempt is made to use the globals from obj, and these are also used as the locals. If the object does not appear to have globals, an exception is raised.
- If one dict argument is passed, it is used for both globals and locals.
- If two dict arguments are passed, they specify globals and locals, respectively.

[169] https://docs.python.org/3.4/library/stdtypes.html#dict
[170] https://docs.python.org/3.4/library/collections.html#collections.defaultdict
[171] https://docs.python.org/3.4/library/stdtypes.html#list
[172] https://docs.python.org/3.4/library/stdtypes.html#set

dhcpkit.typing.py352_typing.**NewType**(*name*, *tp*)
NewType creates simple unique types with almost zero runtime overhead. NewType(name, tp) is considered a subtype of tp by static type checkers. At runtime, NewType(name, tp) returns a dummy function that simply returns its argument. Usage:

```
UserId = NewType('UserId', int)

def name_by_id(user_id: UserId) -> str:
    ...

UserId('user')          # Fails type check

name_by_id(42)          # Fails type check
name_by_id(UserId(42))  # OK

num = UserId(5) + 1     # type: int
```

dhcpkit.typing.py352_typing.**no_type_check**(*arg*)
Decorator to indicate that annotations are not type hints.

The argument must be a class or function; if it is a class, it applies recursively to all methods defined in that class (but not to methods defined in its superclasses or subclasses).

This mutates the function(s) in place.

dhcpkit.typing.py352_typing.**no_type_check_decorator**(*decorator*)
Decorator to give another decorator the @no_type_check effect.

This wraps the decorator with something that wraps the decorated function in @no_type_check.

dhcpkit.typing.py352_typing.**overload**(*func*)
Decorator for overloaded functions/methods.

In a stub file, place two or more stub definitions for the same function in a row, each decorated with @overload. For example:

```
@overload
def utf8(value: None) -> None: ...
@overload
def utf8(value: bytes) -> bytes: ...
@overload
def utf8(value: str) -> bytes: ...
```

In a non-stub file (i.e. a regular .py file), do the same but follow it with an implementation. The implementation should *not* be decorated with @overload. For example:

```
@overload
def utf8(value: None) -> None: ...
@overload
def utf8(value: bytes) -> bytes: ...
@overload
def utf8(value: str) -> bytes: ...

def utf8(value):
    # implementation goes here
    pass
```

dhcpkit.typing.py352_typing.**Text**
alias of builtins.str

Submodules

dhcpkit.display_strings module

Dictionaries with names of common elements, like hardware types. Just for display purposes.

dhcpkit.protocol_element module

The base class *ProtocolElement* (page 218) provides the basic structure for each element of the DHCP protocol. This base class provides several functions:

- **Parsing:** Each subclass can parse a stream of bytes from a protocol packet and construct an instance that contains all the data from the byte stream as properties.
- **Identification:** Each category of ProtocolElement can determine which subclass is the most specific implementation for the data being parsed. For example when letting the Message class `parse` a message it will look at the message type code in the byte steam and determine which specific subclass should parse the data (i.e. SolicitMessage, RequestMessage, ReplyMessage etc). Each category of ProtocolElement has its own registry that keeps track of which type code corresponds to which subclass.
- **Saving:** Each instance can save its contents to a stream of bytes as required by the protocol.
- **Validation:** Each element can validate if its contents are valid. As protocol elements often contain other protocol elements (a message has options, an option might have sub-options etc) there are standard tools for defining which protocol element may contain which other protocol elements and optionally define a minimum and maximum occurrence. Some elements may not occur more than once, some elements must occur at least once, etc.
- **Representation:** The default implementation provides __str__ and __repr__ methods so that protocol elements can be printed for debugging and represented as a parseable Python string.

class dhcpkit.protocol_element.**AutoConstructorParams**
Bases: *dhcpkit.protocol_element.AutoMayContainTree* (page 218)

Meta-class that stores the list of parameters for __init__ so that we don't have to use inspect every time we want to know.

class dhcpkit.protocol_element.**AutoMayContainTree**
Bases: `type`[173]

Meta-class that automatically creates a _may_contain class property that is a ChainMap that links all parent _may_contain class properties.

class dhcpkit.protocol_element.**ElementDataRepresentation**(*element_representation: str*)
Bases: `object`[174]

Class that represents data in a nicer way when printing it with `ProtocolElement.__str__`.

class dhcpkit.protocol_element.**JSONProtocolElementEncoder**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)
Bases: `json.encoder.JSONEncoder`

A JSONEncoder that can handle ProtocolElements

[173] https://docs.python.org/3.4/library/functions.html#type
[174] https://docs.python.org/3.4/library/functions.html#object

default(*o*)
Return a data structure that JSON can handle

Parameters **o** – The object to convert

Returns A serializable data structure

class dhcpkit.protocol_element.**ProtocolElement**
Bases: object[175]

A StructuredElement is a specific kind of class that represents a protocol message or option. Structured elements have the following extra requirements:

- The constructor parameters and the internal state properties must be identical So if an object has a property *timeout* which is an integer then the constructor must accept a named parameter called *timeout* which is stored in that property. The constructor must have appropriate default values if possible. Empty objects, lists, dictionaries etc are represented by a default value of None.
- The full internal state of the object must be loadable from a bytes object with the *load_from()* (page 219) method
- The full internal state of the object must be storable as a bytes object with the *save()* (page 219) method

classmethod add_may_contain(*klass: type*, *min_occurrence: int = 0*, *max_occurrence: int = 2147483647*)
Add the given class to the list of permitted sub-element classes, optionally with a minimum and maximum occurrence count.

Parameters

- **klass** – The class to add
- **min_occurrence** – Minimum occurrence for validation
- **max_occurrence** – Maximum occurrence for validation

classmethod determine_class(*buffer: bytes*, *offset: int = 0*) → type
Return the appropriate class to parse this element with.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading

Returns The best known class for this data

classmethod get_element_class(*element: object*) → Union
Get the class this element is classified as, for occurrence counting.

Parameters **element** – Some element

Returns The class it classifies as

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

[175] https://docs.python.org/3.4/library/functions.html#object

classmethod may_contain(*element: object*) → bool
Shortcut-method to verify that objects of this class may contain element

Parameters **element** – Sub-element to verify

Returns Whether this class may contain element or not

classmethod parse(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → Tuple
Constructor for a new element of which the state is automatically loaded from the given buffer. Both the number of bytes used from the buffer and the instantiated element are returned. The class of the returned element may be a subclass of the current class if the parser can determine that the data in the buffer contains a subtype.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer and the resulting element

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

validate()
Subclasses may overwrite this method to validate their state. Subclasses are expected to raise a ValueError if validation fails.

validate_contains(*elements: Iterable*)
Utility method that subclasses can use in their validate method for verifying that all sub-elements are allowed to be contained in this element. Will raise ValueError if validation fails.

Parameters **elements** – The list of sub-elements

class dhcpkit.protocol_element.**UnknownProtocolElement**(*data: bytes = b''*)
Bases: *dhcpkit.protocol_element.ProtocolElement* (page 218)

Representation of a protocol element about which nothing is known.

load_from(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → int
Load the internal state of this object from the given buffer. The buffer may contain more data after the structured element is parsed. This data is ignored.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer

save() → Union
Save the internal state of this object as a buffer.

Returns The buffer with the data from this element

dhcpkit.registry module

Base class for pkg_resources based registries

class dhcpkit.registry.**Registry**
Bases: collections.UserDict[176]

[176] https://docs.python.org/3.4/library/collections.html#collections.UserDict

Base class for registries

by_name = None
An alternative name-based mapping

entry_point = 'dhcpkit.NONE'
The name of the entry_point group

get_name(*item: object*) → str
Get the name for the by_name mapping.

Parameters **item** – The item to determine the name of

Returns The name to use as key in the mapping

dhcpkit.utils module

Utility functions

dhcpkit.utils.**camelcase_to_dash**(*camelcase: str*) → str
Convert a name in CamelCase to non-camel-case

Parameters **camelcase** – CamelCased string

Returns non-camel-cased string

dhcpkit.utils.**camelcase_to_underscore**(*camelcase: str*) → str
Convert a name in CamelCase to non_camel_case

Parameters **camelcase** – CamelCased string

Returns non_camel_cased string

dhcpkit.utils.**encode_domain**(*domain_name: str*, *allow_relative: bool = False*) → bytearray
Encode a single domain name as a sequence of bytes

Parameters

- **domain_name** – The domain name
- **allow_relative** – Assume that domain names that don't end with a period are relative and encode them as such

Returns The encoded domain name as bytes

dhcpkit.utils.**encode_domain_list**(*domain_names: Iterable*) → bytearray
Encode a list of domain names to a sequence of bytes

Parameters **domain_names** – The list of domain names

Returns The encoded domain names as bytes

dhcpkit.utils.**normalise_hex**(*hex_data: Union*, *include_colons: bool = False*) → str
Normalise a string containing hexadecimal data

Parameters

- **hex_data** – Hexadecimal data, either with or without colon separators per byte
- **include_colons** – Whether to include colon separators per byte in the output

Returns Hexadecimal data in lowercase without colon separators

dhcpkit.utils.**parse_domain_bytes**(*buffer: bytes*, *offset: int = 0*, *length: int = None*, *allow_relative: bool = False*) → Tuple
Extract a single domain name.

Parameters

- **buffer** – The buffer to read data from

- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer
- **allow_relative** – Allow domain names that do not end with a zero-length label

Returns The number of bytes used from the buffer and the extracted domain name

dhcpkit.utils.**parse_domain_list_bytes**(*buffer: bytes*, *offset: int = 0*, *length: int = None*) → Tuple

Extract a list of domain names.

Parameters

- **buffer** – The buffer to read data from
- **offset** – The offset in the buffer where to start reading
- **length** – The amount of data we are allowed to read from the buffer

Returns The number of bytes used from the buffer and the extracted domain names

dhcpkit.utils.**validate_domain_label**(*label: str*)

Check if a given string is a valid domain label

Parameters **label** – The domain label

1.4 Changes per version

1.4.1 1.0.8 - Unreleased

New features

Fixes

Changes for users

Changes for developers

1.4.2 1.0.7 - 2017-06-25

Fixes

- Fix Debian and RPM dependencies
- Fix unnecessary dependency on newer version of pkg_resources

1.4.3 1.0.6 - 2017-06-25

Fixes

- Fix calculations of maximum domain name length
- Deal with the release of ZConfig 3.2.0, which broke our ZConfig 3.1.0 previous hacks

Changes for users

- Switch to `idna` with better IDNA implementation (it implements **RFC 5891**[177] instead of the obsolete **RFC 3490**[178])

Changes for developers

- `normalise_hex()` now accepts bytes as input, for easier byte printing

1.4.4 1.0.5 - 2017-06-21

New features

- Provide RPM repositories at https://repo.dhcpkit.org/
- Add PD-Exclude option implementation
- Add Client FQDN option implementation
- Add Timezone options implementation
- Add Relay Echo-Request option implementation
- Add Relay Echo-Request option handler implementation

Fixes

- Fixed Python type annotations in many places

Changes for users

- Show LDRA relays when printing *`TransactionBundle`* (page 144)
- The DHCPv6 server will try to keep going when not running as root, it might sometimes work, for example when connecting to a VPP instance instead of the usual opening listening sockets on low ports

Changes for developers

- Improve DNS handling: be explicit about difference between absolute and relative domain names
- New *`UnknownProtocolElement`* (page 219) available for more generic protocol parsing
- `All_DHCP_Relay_Agents_and_Servers` and `All_DHCP_Servers` are now instances of IPv6Address instead of strings
- Added an *`IgnoreMessage`* (page 128) exception type so listeners can signal to the server that the received data is to be ignored
- The *`IncompleteMessage`* (page 128) exception is now a subclass of *`IgnoreMessage`* (page 128)

[177] https://tools.ietf.org/html/rfc5891.html
[178] https://tools.ietf.org/html/rfc3490.html

1.4.5 1.0.4 - 2016-12-17

New features

- Provide debian/ubuntu packaging with init/init.d/systemd scripts
- Provide an extensive default configuration
- Allow the control socket location to be overridden from the command line

Fixes

- Improve ipv6-dhcpd to ipv6-dhcpctl communication when shutting down
- Add missing copyright statement for RFC 7598
- Fix Sphinx RFC reference

Changes for users

- Allow unicast listening on ::1 for testing purposes

Changes for developers

- Make python dependencies more accurate
- Make building process for documentation more stable

1.4.6 1.0.3 - 2016-11-17

Fixes

- Fix leasequery statistics output, also fixes Observium stats

1.4.7 1.0.2 - 2016-11-16

Why?

- Re-release with updated changelog and status

1.4.8 1.0.0 - 2016-11-16

New features

- Add rate limit handler to ignore obnoxious clients
- Add implementation for the Leasequery and Bulk Leasequery protocols

Fixes

- Ignore MAC address *00:00:00:00:00:00* when searching for a server-id
- Fix finding the inner relay message in a RelayForwardMessage

Changes for users

- Improve logging for ignored messages
- The default log level now only logs errors, not warnings
- Improve exception handling and logging for errors during worker initialisation

Changes for developers

- Sending replies has been moved from the main process to the worker processes
- Therefore `OutgoingPacketBundle` does no longer exist
- Constants for status codes have been renamed to be more consistent, the old names have been deprecated and will be removed in the future
- Tests have been moved under the dhcpkit module to be easier to import from other extensions (for example when they need a solicit message and packet to test with)
- Added Leasequery and Bulk Leasequery messages, options and status codes
- Leasequery needs RelayForwardMessages without a contained message, so allow that now
- Code for privilege management have been moved to dhcpkit.common
- Code for console logging has been moved to dhcpkit.common.logging
- Replies are now sent directly from worker processes, not first handed back to the master and then sent from there
- Refactor listeners and message handling to allow for TCP listeners and leasequery extensions
- Open sockets with SO_REUSEADDR so we can restart quickly without having to wait for TIME_WAIT
- Allow for multiple responses in transaction bundle, especially useful for TCP connections

1.4.9 0.9.5 - 2016-08-11

New features

- 2.5x speed improvement.

Changes for developers

- *`ProtocolElement.parse()`* (page 219) and the *`load_from()`* (page 219) methods it uses no longer call *`ProtocolElement.validate()`* (page 219) because every (nested) element validating everything all the time is rather inefficient. Now callers are supposed to call *`ProtocolElement.validate()`* (page 219) themselves (if they want to).
- We no longer use `abc`[179] and `ABCMeta`[180]. It turned out that all the run-time validation it did caused a ±20% slow down.

1.4.10 0.9.4 - 2016-08-04

New features

- Added support for the **RFC 6939**[181] client link-layer address relay option

[179] https://docs.python.org/3.4/library/abc.html#module-abc
[180] https://docs.python.org/3.4/library/abc.html#abc.ABCMeta
[181] https://tools.ietf.org/html/rfc6939.html

- Added support for the **RFC 4580**[182] subscriber-id relay option
- Added support for the **RFC 6334**[183] DS-Lite AFTR tunnel endpoint name option
- Added support for the **RFC 7598**[184] MAP options
- Added support for `linklayer_id` (page 59) and `subscriber_id` (page 87) in *Static-csv* (page 23) and *Static-sqlite* (page 24)

Fixes

- Fix error where command line log-level argument was ignored.
- Fix error that caused every message to be interpreted as received-over-multicast
- Don't block when the inbound queue is full, just drop the message and continue
- Fixed an interface-id parsing bug in *Static-csv* (page 23) and *Static-sqlite* (page 24)
- Allow UnknownOption in all options, otherwise we reject messages with options that contain unknown sub-options

1.4.11 0.9.3 - 2016-07-27

Fixes

- Not all systems have a `wheel` group anymore, so don't use that as a default group for the control socket.
- Linux doesn't support SIGINFO, and its functionality has become redundant with the new control socket functionality, so remove SIGINFO handling.

Changes for users

- Critical errors are now always shown on *stderr*. Otherwise the server could crash without the user seeing the reason.

1.4.12 0.9.2 - 2016-07-27

Fixes

- A packaging error slipped through the checks, and it turns out that crucial XML files weren't packaged in previous 0.9.x versions. This has now been fixed.

1.4.13 0.9.1 - 2016-07-27

New features

- It is now possible to use IDNs everywhere in DHCPKit, including configuration files.
- Implement a domain socket to control the server process.
- Added *ipv6-dhcpctl(8)* (page 4) to control the server process through the domain socket.
- Added a configuration section `<statistics>` to specify categories that you would like statistics on. Currently it is possible to gather statistics per interface, client subnet or relay.
- Added `stats` and `stats-json` commands for *ipv6-dhcpctl*.

[182] https://tools.ietf.org/html/rfc4580.html
[183] https://tools.ietf.org/html/rfc6334.html
[184] https://tools.ietf.org/html/rfc7598.html

Changes for users

- Create PID file /var/run/ipv6-dhcpd.pid by default.
- Create domain socket /var/run/ipv6-dhcpd.sock control the server by default.

Changes for developers

- Added support for Internationalized Domain Names (IDN) in *parse_domain_bytes()* (page 221) and *encode_domain()* (page 220).
- Created ForOtherServerError as a subclass of CannotRespondError, to enable more accurate logging, and to make it possible to gather better statistics.
- Replaced `IncomingPacketBundle.interface_id bytes` with `interface_name str`, providing `interface_id` for backwards compatibility.
- Added *relays* (page 146) property to more easily enumerate all the relays a message went through.
- Moved responsibility of creating the *TransactionBundle* (page 144) from the *MessageHandler* (page 139) to *worker* (page 146). It gives a cleaner API and helps with statistics counting.
- Added *statistics* (page 142) and updated *worker* (page 146) and *MessageHandler* (page 139) to update relevant counters.

1.4.14 0.9.0 - 2016-07-16

- A complete rewrite of the DHCPv6 server with a new configuration style.

1.5 About this project

1.5.1 Background

There are plenty of good DHCPv6 servers, but all of them were made for "standard" dynamic environments. During a project at Solcon[185] I found out that something as simple as getting a DHCPv6 server to do some static prefix delegations to a predetermined set of customers (we were doing a pilot) didn't work with existing tools. I'm constantly on the lookout for potential blocks to IPv6 deployment to solve, and here was one. Thus, DHCPKit was born.

1.5.2 Sponsors

The first implementation of DHCPKit was partially sponsored by Solcon[186], and I am very grateful for their support.

After the first version was running in production I decided to take this project further. My goals were:

- Write better documentation
- Improve performance
- Better quality assurance
- Implement more DHCPv6 options
- Add more interfaces, e.g. with RADIUS
- Provide a more flexible configuration file format

[185] http://www.solcon.nl/
[186] http://www.solcon.nl/

- Integrate with monitoring systems

I applied for a grant from the SIDN Fund[187] to implement all of this. I received the grant in 2016 and am currently working to achieve these goals.

1.5.3 List of users

Here is a list of organisations, projects and individuals that have notified me that they are using DHCPKit and want to be listed here:

- Solcon[188]

If you are using DHCPKit please let me know by sending an email to dhcpkit@steffann.nl. Please also let me know whether you want to be mentioned on this page - I will not add any names here without explicit consent.

1.5.4 Participating

DHCPKit is released under the GPLv3 license so you are free to use and adapt DHCPKit. If you distribute modified or extended versions of DHCPKit you must honour the license and make your changes available under a compatible license. If you would like to see extra features and/or options implemented and don't feel like writing the code yourself, please contact me on dhcpkit@steffann.nl.

1.6 Applicable copyright licences

1.6.1 DHCPKit License

Copyright (c) 2015-2016, S.J.M. Steffann

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

1.6.2 RFC Licenses

This project includes text copies of text from relevant RFC documents published by The Internet Society and the IETF Trust to aid in the implementation of the standards described in those RFC documents. Each file containing copies of text from RFC documents provides a reference to the original text. The following licenses apply:

RFC 3315[189], **RFC 3319**[190], **RFC 3633**[191], **RFC 3646**[192] Copyright (C) The Internet Society (2003). All Rights Reserved.

> This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may

[187] https://www.sidnfonds.nl/excerpt/
[188] http://www.solcon.nl/
[189] https://tools.ietf.org/html/rfc3315.html
[190] https://tools.ietf.org/html/rfc3319.html
[191] https://tools.ietf.org/html/rfc3633.html
[192] https://tools.ietf.org/html/rfc3646.html

not be modified in any way, such as by removing the copyright notice or references to the Internet Society or other Internet organizations, except as needed for the purpose of developing Internet standards in which case the procedures for copyrights defined in the Internet Standards process must be followed, or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by the Internet Society or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

RFC 3898[193] Copyright (C) The Internet Society (2004).

This document is subject to the rights, licenses and restrictions contained in BCP 78, and except as set forth therein, the authors retain all their rights.

This document and the information contained herein are provided on an "AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

RFC 4075[194], **RFC 4242**[195], **RFC 4280**[196] Copyright (C) The Internet Society (2005).

This document is subject to the rights, licenses and restrictions contained in BCP 78, and except as set forth therein, the authors retain all their rights.

This document and the information contained herein are provided on an "AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

RFC 4580[197], **RFC 4649**[198], **RFC 4704**[199] Copyright (C) The Internet Society (2006).

This document is subject to the rights, licenses and restrictions contained in BCP 78, and except as set forth therein, the authors retain all their rights.

This document and the information contained herein are provided on an "AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

RFC 4776[200] Copyright (C) The IETF Trust (2006).

This document is subject to the rights, licenses and restrictions contained in BCP 78, and except as set forth therein, the authors retain all their rights.

This document and the information contained herein are provided on an "AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY, THE IETF TRUST, AND THE INTERNET ENGINEERING TASK FORCE

[193] https://tools.ietf.org/html/rfc3898.html
[194] https://tools.ietf.org/html/rfc4075.html
[195] https://tools.ietf.org/html/rfc4242.html
[196] https://tools.ietf.org/html/rfc4280.html
[197] https://tools.ietf.org/html/rfc4580.html
[198] https://tools.ietf.org/html/rfc4649.html
[199] https://tools.ietf.org/html/rfc4704.html
[200] https://tools.ietf.org/html/rfc4776.html

DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

RFC 4833[201], **RFC 4994**[202], **RFC 5007**[203] Copyright (C) The IETF Trust (2007).

This document is subject to the rights, licenses and restrictions contained in BCP 78, and except as set forth therein, the authors retain all their rights.

This document and the information contained herein are provided on an "AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY, THE IETF TRUST AND THE INTERNET ENGINEERING TASK FORCE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

RFC 5192[204], **RFC 5223**[205] Copyright (C) The IETF Trust (2008).

This document is subject to the rights, licenses and restrictions contained in BCP 78, and except as set forth therein, the authors retain all their rights.

This document and the information contained herein are provided on an "AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY, THE IETF TRUST AND THE INTERNET ENGINEERING TASK FORCE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

RFC 5460[206], **RFC 5678**[207] Copyright (c) 2009 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the BSD License.

RFC 5417[208] Copyright (c) 2009 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents in effect on the date of publication of this document (http://trustee.ietf.org/license-info). Please review these documents carefully, as they describe your rights and restrictions with respect to this document.

This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November 10, 2008. The person(s) controlling the copyright in some of this material may not have granted the IETF Trust the right to allow modifications of such material outside the IETF Standards Process. Without obtaining an adequate license from the person(s) controlling the copyright in such materials, this document may not be modified outside the IETF Standards Process, and derivative works of it may not be created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.

[201] https://tools.ietf.org/html/rfc4833.html
[202] https://tools.ietf.org/html/rfc4994.html
[203] https://tools.ietf.org/html/rfc5007.html
[204] https://tools.ietf.org/html/rfc5192.html
[205] https://tools.ietf.org/html/rfc5223.html
[206] https://tools.ietf.org/html/rfc5460.html
[207] https://tools.ietf.org/html/rfc5678.html
[208] https://tools.ietf.org/html/rfc5417.html

[209] https://tools.ietf.org/html/rfc5970.html
[210] https://tools.ietf.org/html/rfc5986.html
[211] https://tools.ietf.org/html/rfc6011.html
[212] https://tools.ietf.org/html/rfc5908.html
[213] https://tools.ietf.org/html/rfc6153.html
[214] https://tools.ietf.org/html/rfc6334.html
[215] https://tools.ietf.org/html/rfc6422.html
[216] https://tools.ietf.org/html/rfc6440.html
[217] https://tools.ietf.org/html/rfc6225.html
[218] https://tools.ietf.org/html/rfc6603.html
[219] https://tools.ietf.org/html/rfc6731.html
[220] https://tools.ietf.org/html/rfc6607.html
[221] https://tools.ietf.org/html/rfc6610.html
[222] https://tools.ietf.org/html/rfc6784.html

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November 10, 2008. The person(s) controlling the copyright in some of this material may not have granted the IETF Trust the right to allow modifications of such material outside the IETF Standards Process. Without obtaining an adequate license from the person(s) controlling the copyright in such materials, this document may not be modified outside the IETF Standards Process, and derivative works of it may not be created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.

RFC 6939[223], **RFC 6977**[224], **RFC 7037**[225], **RFC 7083**[226] Copyright (c) 2013 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

RFC 7291[227], **RFC 7341**[228] Copyright (c) 2014 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

RFC 7078[229] Copyright (c) 2014 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November 10, 2008. The person(s) controlling the copyright in some of this material may not have granted the IETF Trust the right to allow modifications of such material outside the IETF Standards Process. Without obtaining an adequate license from the person(s) controlling the copyright in such materials, this document may not be modified outside the IETF Standards Process, and derivative works of it may not be created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.

RFC 7598[230] Copyright (c) 2015 IETF Trust and the persons identified as the document authors. All rights reserved.

[223] https://tools.ietf.org/html/rfc6939.html
[224] https://tools.ietf.org/html/rfc6977.html
[225] https://tools.ietf.org/html/rfc7037.html
[226] https://tools.ietf.org/html/rfc7083.html
[227] https://tools.ietf.org/html/rfc7291.html
[228] https://tools.ietf.org/html/rfc7341.html
[229] https://tools.ietf.org/html/rfc7078.html
[230] https://tools.ietf.org/html/rfc7598.html

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

1.6.3 Python License

This software includes the Python 3.5.2 version of the `typing` package. The following license applies:

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 3.5.2 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 3.5.2 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright © 2001-2016 Python Software Foundation; All Rights Reserved" are retained in Python 3.5.2 alone or in any derivative version prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 3.5.2 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 3.5.2.
4. PSF is making Python 3.5.2 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 3.5.2 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.5.2 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.5.2, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By copying, installing or otherwise using Python 3.5.2, Licensee agrees to be bound by the terms and conditions of this License Agreement.

Python Module Index

d

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W