NAME

Net::DNS::Resolver – DNS resolver class

SYNOPSIS

```
use Net::DNS;

$resolver = new Net::DNS::Resolver();

# Perform a lookup, using the searchlist if appropriate.
$reply = $resolver->search( 'example.com' );

# Perform a lookup, without the searchlist
$reply = $resolver->query( 'example.com', 'MX' );

# Perform a lookup, without pre or post-processing
$reply = $resolver->send( 'example.com', 'MX', 'IN' );

# Send a prebuilt query packet
$query = new Net::DNS::Packet( ... );
$reply = $resolver->send( $query );
```

DESCRIPTION

Instances of the Net::DNS::Resolver class represent resolver objects. A program may have multiple resolver objects, each maintaining its own state information such as the nameservers to be queried, whether recursion is desired, etc.

METHODS

new

```
# Use the default configuration
$resolver = new Net::DNS::Resolver();

# Use my own configuration file
$resolver = new Net::DNS::Resolver( config_file => '/my/dns.conf' );

# Set options in the constructor
$resolver = new Net::DNS::Resolver(
    nameservers => [ '2001:DB8::1', 'ns.example.com' ],
    recurse     => 0,
    debug       => 1
    );
```

Returns a resolver object. If no arguments are supplied, `new()` returns an object having the default configuration.

On Unix and Linux systems, the default values are read from the following files, in the order indicated:

/etc/resolv.conf, *$HOME/.resolv.conf*, *./.resolv.conf*

The following keywords are recognised in resolver configuration files:

nameserver address

IP address of a name server that the resolver should query.

domain localdomain

The domain suffix to be appended to a short non-absolute name.

search domain ...

A space-separated list of domains in the desired search path.

options option:value ...

A space-separated list of key:value items.

Except for */etc/resolv.conf*, files will only be read if owned by the effective userid running the program. In addition, several environment variables may contain configuration information; see "ENVIRONMENT".

Note that the domain and searchlist keywords are mutually exclusive. If both are present, the resulting behaviour is unspecified. If neither is present, the domain is determined from the local hostname.

On Windows systems, an attempt is made to determine the system defaults using the registry. Systems with many dynamically configured network interfaces may confuse Net::DNS.

```
# Use my own configuration file
$resolver = new Net::DNS::Resolver( config_file => '/my/dns.conf' );
```

You can include a configuration file of your own when creating a resolver object. This is supported on both Unix and Windows.

If a custom configuration file is specified at first instantiation, all other configuration files and environment variables are ignored.

```
# Set options in the constructor
$resolver = new Net::DNS::Resolver(
    nameservers => [ '2001:DB8::1', 'ns.example.com' ],
    recurse     => 0
    );
```

Explicit arguments to `new()` override the corresponding configuration variables. The argument list consists of a sequence of (name=>value) pairs, each interpreted as an invocation of the corresponding method.

print

```
$resolver->print;
```

Prints the resolver state on the standard output.

query

```
$packet = $resolver->query( 'mailhost' );
$packet = $resolver->query( 'mailhost.example.com' );
$packet = $resolver->query( '2001:DB8::1' );
$packet = $resolver->query( 'example.com', 'MX' );
$packet = $resolver->query( 'annotation.example.com', 'TXT', 'IN' );
```

Performs a DNS query for the given name; the search list is not applied. If `defnames` is true, the default domain will be appended to unqualified names.

The record type and class can be omitted; they default to A and IN. If the name looks like an IP address (IPv4 or IPv6), then a query within in-addr.arpa or ip6.arpa will be performed.

Returns a Net::DNS::Packet object, or `undef` if no answers were found. The reason for failure may be determined using `errorstring()`.

If you need to examine the response packet, whether it contains any answers or not, use the `send()` method instead.

search

```
$packet = $resolver->search( 'mailhost' );
$packet = $resolver->search( 'mailhost.example.com' );
$packet = $resolver->search( '2001:DB8::1' );
$packet = $resolver->search( 'example.com', 'MX' );
$packet = $resolver->search( 'annotation.example.com', 'TXT', 'IN' );
```

Performs a DNS query for the given name, applying the searchlist if appropriate. The search algorithm is as follows:

If the name contains one or more non-terminal dots, perform an initial query using the unmodified name.

If the number of dots is less than `ndots`, and there is no terminal dot, try appending each suffix in the search list.

The record type and class can be omitted; they default to A and IN. If the name looks like an IP address (IPv4 or IPv6), then a query within in-addr.arpa or ip6.arpa will be performed.

Returns a Net::DNS::Packet object, or `undef` if no answers were found. The reason for failure may be determined using `errorstring()`.

If you need to examine the response packet, whether it contains any answers or not, use the `send()` method instead.

send

```
$packet = $resolver->send( $query );

$packet = $resolver->send( 'mailhost.example.com' );
$packet = $resolver->send( '2001:DB8::1' );
$packet = $resolver->send( 'example.com', 'MX' );
$packet = $resolver->send( 'annotation.example.com', 'TXT', 'IN' );
```

Performs a DNS query for the given name. Neither the searchlist nor the default domain will be appended.

The argument list can be either a pre-built query Net::DNS::Packet or a list of strings. The record type and class can be omitted; they default to A and IN. If the name looks like an IP address (IPv4 or IPv6), then a query within in-addr.arpa or ip6.arpa will be performed.

Returns a Net::DNS::Packet object whether there were any answers or not. Use `$packet->header->ancount` or `$packet->answer` to find out if there were any records in the answer section. Returns `undef` if no response was received.

axfr

```
@zone = $resolver->axfr();
@zone = $resolver->axfr( 'example.com' );
@zone = $resolver->axfr( 'example.com', 'IN' );

$iterator = $resolver->axfr();
$iterator = $resolver->axfr( 'example.com' );
$iterator = $resolver->axfr( 'example.com', 'IN' );

$rr = $iterator->();
```

Performs a zone transfer using the resolver nameservers list, attempted in the order listed.

If the zone is omitted, it defaults to the first zone listed in the resolver search list.

If the class is omitted, it defaults to IN.

When called in list context, `axfr()` returns a list of Net::DNS::RR objects. The redundant SOA record that terminates the zone transfer is not returned to the caller.

In deferrence to RFC1035(6.3), a complete zone transfer is expected to return all records in the zone or nothing at all. When no resource records are returned by `axfr()`, the reason for failure may be determined using `errorstring()`.

Here is an example that uses a timeout and TSIG verification:

```
$resolver->tcp_timeout( 10 );
$resolver->tsig( 'Khmac-sha1.example.+161+24053.private' );
@zone = $resolver->axfr( 'example.com' );

foreach $rr (@zone) {
    $rr->print;
```

```
}
```

When called in scalar context, `axfr()` returns an iterator object. Each invocation of the iterator returns a single Net::DNS::RR or `undef` when the zone is exhausted.

An exception is raised if the zone transfer can not be completed.

The redundant SOA record that terminates the zone transfer is not returned to the caller.

Here is the example above, implemented using an iterator:

```
$resolver->tcp_timeout( 10 );
$resolver->tsig( 'Khmac-sha1.example.+161+24053.private' );
$iterator = $resolver->axfr( 'example.com' );

while ( $rr = $iterator->() ) {
    $rr->print;
}
```

bgsend

```
$handle = $resolver->bgsend( $packet ) || die $resolver->errorstring;

$handle = $resolver->bgsend( 'mailhost.example.com' );
$handle = $resolver->bgsend( '2001:DB8::1' );
$handle = $resolver->bgsend( 'example.com', 'MX' );
$handle = $resolver->bgsend( 'annotation.example.com', 'TXT', 'IN' );
```

Performs a background DNS query for the given name and returns immediately without waiting for the response. The program can then perform other tasks while awaiting the response from the nameserver.

The argument list can be either a Net::DNS::Packet object or a list of strings. The record type and class can be omitted; they default to A and IN. If the name looks like an IP address (IPv4 or IPv6), then a query within in−addr.arpa or ip6.arpa will be performed.

Returns an opaque handle which is passed to subsequent invocations of the `bgbusy()` and `bgread()` methods. Errors are indicated by returning `undef` in which case the reason for failure may be determined using `errorstring()`.

The response Net::DNS::Packet object is obtained by calling `bgread()`.

BEWARE: Programs should make no assumptions about the nature of the handles returned by `bgsend()` which should be used strictly as described here.

bgread

```
$handle = $resolver->bgsend( 'www.example.com' );
$packet = $resolver->bgread($handle);
```

Reads the response following a background query. The argument is the handle returned by `bgsend()`.

Returns a Net::DNS::Packet object or `undef` if no response was received before the timeout interval expired.

bgbusy

```
$handle = $resolver->bgsend( 'foo.example.com' );

while ($resolver->bgbusy($handle)) {
    ...
}

$packet = $resolver->bgread($handle);
```

Returns true while awaiting the response or for the transaction to time out. The argument is the handle returned by `bgsend()`.

Truncated UDP packets will be retried transparently using TCP while continuing to assert busy to the caller.

debug

```
print 'debug flag: ', $resolver->debug, "\n";
$resolver->debug(1);
```

Get or set the debug flag. If set, calls to `search()`, `query()`, and `send()` will print debugging information on the standard output. The default is false.

defnames

```
print 'defnames flag: ', $resolver->defnames, "\n";
$resolver->defnames(0);
```

Get or set the defnames flag. If true, calls to `query()` will append the default domain to resolve names that are not fully qualified. The default is true.

dnsrch

```
print 'dnsrch flag: ', $resolver->dnsrch, "\n";
$resolver->dnsrch(0);
```

Get or set the dnsrch flag. If true, calls to `search()` will apply the search list to resolve names that are not fully qualified. The default is true.

domain

```
$domain = $resolver->domain;
$resolver->domain( 'domain.example' );
```

Gets or sets the resolver default domain.

igntc

```
print 'igntc flag: ', $resolver->igntc, "\n";
$resolver->igntc(1);
```

Get or set the igntc flag. If true, truncated packets will be ignored. If false, the query will be retried using TCP. The default is false.

nameserver, nameservers

```
@nameservers = $resolver->nameservers();
$resolver->nameservers( '2001:DB8::1', '192.0.2.1' );
$resolver->nameservers( 'ns.domain.example.' );
```

Gets or sets the nameservers to be queried.

Also see the IPv6 transport notes below

persistent_tcp

```
print 'Persistent TCP flag: ', $resolver->persistent_tcp, "\n";
$resolver->persistent_tcp(1);
```

Get or set the persistent TCP setting. If true, Net::DNS will keep a TCP socket open for each host:port to which it connects. This is useful if you are using TCP and need to make a lot of queries or updates to the same nameserver.

The default is false unless you are running a SOCKSified Perl, in which case the default is true.

persistent_udp

```
print 'Persistent UDP flag: ', $resolver->persistent_udp, "\n";
$resolver->persistent_udp(1);
```

Get or set the persistent UDP setting. If true, a Net::DNS resolver will use the same UDP socket for all queries within each address family.

This avoids the cost of creating and tearing down UDP sockets, but also defeats source port randomisation.

port

```
print 'sending queries to port ', $resolver->port, "\n";
$resolver->port(9732);
```

Gets or sets the port to which queries are sent. Convenient for nameserver testing using a non-standard port. The default is port 53.

recurse

```
print 'recursion flag: ', $resolver->recurse, "\n";
$resolver->recurse(0);
```

Get or set the recursion flag. If true, this will direct nameservers to perform a recursive query. The default is true.

retrans

```
print 'retrans interval: ', $resolver->retrans, "\n";
$resolver->retrans(3);
```

Get or set the retransmission interval The default is 5 seconds.

retry

```
print 'number of tries: ', $resolver->retry, "\n";
$resolver->retry(2);
```

Get or set the number of times to try the query. The default is 4.

searchlist

```
@searchlist = $resolver->searchlist;
$resolver->searchlist( 'a.example', 'b.example', 'c.example' );
```

Gets or sets the resolver search list.

srcaddr

```
$resolver->srcaddr('2001::DB8::1');
```

Sets the source address from which queries are sent. Convenient for forcing queries from a specific interface on a multi-homed host. The default is to use any local address.

srcport

```
$resolver->srcport(5353);
```

Sets the port from which queries are sent. The default is 0, meaning any port.

tcp_timeout

```
print 'TCP timeout: ', $resolver->tcp_timeout, "\n";
$resolver->tcp_timeout(10);
```

Get or set the TCP timeout in seconds. The default is 120 seconds (2 minutes).

udp_timeout

```
print 'UDP timeout: ', $resolver->udp_timeout, "\n";
$resolver->udp_timeout(10);
```

Get or set the **bgsend()** UDP timeout in seconds. The default is 30 seconds.

udppacketsize

```
print "udppacketsize: ", $resolver->udppacketsize, "\n";
$resolver->udppacketsize(2048);
```

Get or set the UDP packet size. If set to a value not less than the default DNS packet size, an EDNS extension will be added indicating support for large UDP datagrams.

usevc

```
print 'usevc flag: ', $resolver->usevc, "\n";
$resolver->usevc(1);
```

Get or set the usevc flag. If true, queries will be performed using virtual circuits (TCP) instead of datagrams (UDP). The default is false.

replyfrom

```
print 'last response was from: ', $resolver->replyfrom, "\n";
```

Returns the IP address from which the most recent packet was received in response to a query.

errorstring

```
print 'query status: ', $resolver->errorstring, "\n";
```

Returns a string containing error information from the most recent DNS protocol interaction. `errorstring()` is meaningful only when interrogated immediately after the corresponding method call.

dnssec

```
print "dnssec flag: ", $resolver->dnssec, "\n";
$resolver->dnssec(0);
```

The dnssec flag causes the resolver to transmit DNSSEC queries and to add a EDNS0 record as required by RFC2671 and RFC3225. The actions of, and response from, the remote nameserver is determined by the settings of the AD and CD flags.

Calling the `dnssec()` method with a non-zero value will also set the UDP packet size to the default value of 2048. If that is too small or too big for your environment, you should call the `udppacketsize()` method immediately after.

```
$resolver->dnssec(1);                 # DNSSEC using default packetsize
$resolver->udppacketsize(1250);       # lower the UDP packet size
```

A fatal exception will be raised if the `dnssec()` method is called but the Net::DNS::SEC library has not been installed.

adflag

```
$resolver->dnssec(1);
$resolver->adflag(1);
print "authentication desired flag: ", $resolver->adflag, "\n";
```

Gets or sets the AD bit for dnssec queries. This bit indicates that the caller is interested in the returned AD (authentic data) bit but does not require any dnssec RRs to be included in the response. The default value is false.

cdflag

```
$resolver->dnssec(1);
$resolver->cdflag(1);
print "checking disabled flag: ", $resolver->cdflag, "\n";
```

Gets or sets the CD bit for dnssec queries. This bit indicates that authentication by upstream nameservers should be suppressed. Any dnssec RRs required to execute the authentication procedure should be included in the response. The default value is false.

tsig

```
$resolver->tsig( $tsig );

$resolver->tsig( 'Khmac-sha1.example.+161+24053.private' );

$resolver->tsig( 'Khmac-sha1.example.+161+24053.key' );

$resolver->tsig( 'Khmac-sha1.example.+161+24053.key',
            fudge => 60
            );

$resolver->tsig( $key_name, $key );

$resolver->tsig( undef );
```

Set the TSIG record used to automatically sign outgoing queries, zone transfers and updates. Automatic

signing is disabled if called with undefined arguments.

The default resolver behaviour is not to sign any packets. You must call this method to set the key if you would like the resolver to sign and verify packets automatically.

Packets can also be signed manually; see the Net::DNS::Packet and Net::DNS::Update manual pages for examples. TSIG records in manually-signed packets take precedence over those that the resolver would add automatically.

ENVIRONMENT

The following environment variables can also be used to configure the resolver:

RES_NAMESERVERS

```
# Bourne Shell
RES_NAMESERVERS="2001:DB8::1 192.0.2.1"
export RES_NAMESERVERS

# C Shell
setenv RES_NAMESERVERS "2001:DB8::1 192.0.2.1"
```

A space-separated list of nameservers to query.

RES_SEARCHLIST

```
# Bourne Shell
RES_SEARCHLIST="a.example.com b.example.com c.example.com"
export RES_SEARCHLIST

# C Shell
setenv RES_SEARCHLIST "a.example.com b.example.com c.example.com"
```

A space-separated list of domains to put in the search list.

LOCALDOMAIN

```
# Bourne Shell
LOCALDOMAIN=example.com
export LOCALDOMAIN

# C Shell
setenv LOCALDOMAIN example.com
```

The default domain.

RES_OPTIONS

```
# Bourne Shell
RES_OPTIONS="retrans:3 retry:2 inet6"
export RES_OPTIONS

# C Shell
setenv RES_OPTIONS "retrans:3 retry:2 inet6"
```

A space-separated list of resolver options to set. Options that take values are specified as `option:value`.

IPv4 TRANSPORT

The `force_v4()`, `force_v6()`, `prefer_v4()`, and `prefer_v6()` methods with non-zero argument may be used to configure transport selection.

The behaviour of the `nameserver()` method illustrates the transport selection mechanism. If, for example, IPv4 transport has been forced, the `nameserver()` method will only return IPv4 addresses:

```
$resolver->nameservers( '192.0.2.1', '192.0.2.2', '2001:DB8::3' );
$resolver->force_v4(1);
print join ' ', $resolver->nameservers();
```

will print

```
192.0.2.1 192.0.2.2
```

CUSTOMISED RESOLVERS

Net::DNS::Resolver is actually an empty subclass. At compile time a super class is chosen based on the current platform. A side benefit of this allows for easy modification of the methods in Net::DNS::Resolver. You can simply add a method to the namespace!

For example, if we wanted to cache lookups:

```
package Net::DNS::Resolver;

my %cache;

sub search {
    $self = shift;

    $cache{"@_"} ||= $self->SUPER::search(@_);
}
```

COPYRIGHT

Copyright (c)1997–2000 Michael Fuhr.

Portions Copyright (c)2002–2004 Chris Reinhardt.

Portions Copyright (c)2005 Olaf M. Kolkman, NLnet Labs.

Portions Copyright (c)2014,2015 Dick Franks.

All rights reserved.

LICENSE

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of the author not be used in advertising or publicity pertaining to distribution of the software without specific prior written permission.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

SEE ALSO

perl, Net::DNS, Net::DNS::Packet, Net::DNS::Update, Net::DNS::Header, Net::DNS::Question, Net::DNS::RR, **resolver** (5), RFC 1034, RFC 1035