## NAME

Net::DNS::Nameserver – DNS server class

## SYNOPSIS

```
use Net::DNS::Nameserver;

my $nameserver = new Net::DNS::Nameserver(
    LocalAddr       => ['::1' , '127.0.0.1'],
    ZoneFile        => "filename"
    );

my $nameserver = new Net::DNS::Nameserver(
    LocalAddr       => '10.1.2.3',
    LocalPort       => 5353,
    ReplyHandler    => \&reply_handler
);
```

## DESCRIPTION

Net::DNS::Nameserver offers a simple mechanism for instantiation of customised DNS server objects intended to provide test responses to queries emanating from a client resolver.

It is not, nor will it ever be, a general-purpose DNS nameserver implementation.

See ‘‘EXAMPLE’’ for an example.

## METHODS

### new

```
$nameserver = new Net::DNS::Nameserver(
    LocalAddr       => ['::1' , '127.0.0.1'],
    ZoneFile        => "filename"
    );

$nameserver = new Net::DNS::Nameserver(
    LocalAddr       => '10.1.2.3',
    LocalPort       => 5353,
    ReplyHandler    => \&reply_handler,
    Verbose         => 1,
    Truncate        => 0
);
```

Returns a Net::DNS::Nameserver object, or undef if the object could not be created.

Each instance is configured using the following optional arguments:

```
LocalAddr           IP address on which to listen   Defaults to loopback address
LocalPort           Port on which to listen         Defaults to 53
ZoneFile            Name of file containing RRs
                    accessed using the default
                    reply-handling subroutine
ReplyHandler        Reference to customised
                    reply-handling subroutine
NotifyHandler       Reference to reply-handling
                    subroutine for queries with
                    opcode NOTIFY (RFC1996)
UpdateHandler       Reference to reply-handling
                    subroutine for queries with
                    opcode UPDATE (RFC2136)
Verbose             Report internal activity        Defaults to 0 (off)
Truncate            Truncates UDP packets that
```

```
                        are too big for the reply       Defaults to 1 (on)
    IdleTimeout         TCP clients are disconnected
                        if they are idle longer than
                        this duration                   Defaults to 120 (secs)
```

The LocalAddr attribute may alternatively be specified as a list of IP addresses to listen to. If the IO::Socket::IP library package is available on the system this may also include IPv6 addresses.

The ReplyHandler subroutine is passed the query name, query class, query type and optionally an argument containing the peerhost, the incoming query, and the name of the incoming socket (sockethost). It must either return the response code and references to the answer, authority, and additional sections of the response, or undef to leave the query unanswered. Common response codes are:

```
NOERROR     No error
FORMERR     Format error
SERVFAIL    Server failure
NXDOMAIN    Non-existent domain (name doesn't exist)
NOTIMP      Not implemented
REFUSED     Query refused
```

For advanced usage it may also contain a headermask containing an hashref with the settings for the `aa`, `ra`, and `ad` header bits. The argument is of the form `{ ad => 1, aa => 0, ra => 1 }`.

EDNS options may be specified in a similar manner using optionmask `{ $optioncode => $value, $optionname => $value }`.

See RFC 1035 and the IANA dns-parameters file for more information:

```
ftp://ftp.rfc-editor.org/in-notes/rfc1035.txt
http://www.isi.edu/in-notes/iana/assignments/dns-parameters
```

The nameserver will listen for both UDP and TCP connections. On Unix-like systems, the program will probably have to run as root to listen on the default port, 53. A non-privileged user should be able to listen on ports 1024 and higher.

UDP reply truncation functionality was introduced in VERSION 830. The size limit is determined by the EDNS0 size advertised in the query, otherwise 512 is used. If you want to do packet truncation yourself you should set `Truncate` to 0 and truncate the reply packet in the code of the ReplyHandler.

See "EXAMPLE" for an example.

**main_loop**

```
$ns->main_loop;
```

Start accepting queries. Calling main_loop never returns.

**loop_once**

```
$ns->loop_once( [TIMEOUT_IN_SECONDS] );
```

Start accepting queries, but returns. If called without a parameter, the call will not return until a request has been received (and replied to). Otherwise, the parameter specifies the maximum time to wait for a request. A zero timeout forces an immediate return if there is nothing to do.

Handling a request and replying obviously depends on the speed of ReplyHandler. Assuming a fast ReplyHandler, loop_once should spend just a fraction of a second, if called with a timeout value of 0.0 seconds. One exception is when an AXFR has requested a huge amount of data that the OS is not ready to receive in full. In that case, it will remain in a loop (while servicing new requests) until the reply has been sent.

In case loop_once accepted a TCP connection it will immediately check if there is data to be read from the socket. If not it will return and you will have to call **loop_once()** again to check if there is any data waiting on the socket to be processed. In most cases you will have to count on calling "loop_once" twice.

A code fragment like:

```
$ns->loop_once(10);
while( $ns->get_open_tcp() ){
    $ns->loop_once(0);
}
```

Would wait for 10 seconds for the initial connection and would then process all TCP sockets until none is left.

**get_open_tcp**

In scalar context returns the number of TCP connections for which state is maintained. In array context it returns IO::Socket objects, these could be useful for troubleshooting but be careful using them.

## EXAMPLE

The following example will listen on port 5353 and respond to all queries for A records with the IP address 10.1.2.3. All other queries will be answered with NXDOMAIN. Authority and additional sections are left empty. The `$peerhost` variable catches the IP address of the peer host, so that additional filtering on its basis may be applied.

```
#!/usr/bin/perl

use strict;
use warnings;
use Net::DNS::Nameserver;

sub reply_handler {
    my ( $qname, $qclass, $qtype, $peerhost, $query, $conn ) = @_;
    my ( $rcode, @ans, @auth, @add );

    print "Received query from $peerhost to " . $conn->{sockhost} . "\n";
    $query->print;

    if ( $qtype eq "A" && $qname eq "foo.example.com" ) {
            my ( $ttl, $rdata ) = ( 3600, "10.1.2.3" );
            my $rr = new Net::DNS::RR("$qname $ttl $qclass $qtype $rdata");
            push @ans, $rr;
            $rcode = "NOERROR";
    } elsif ( $qname eq "foo.example.com" ) {
            $rcode = "NOERROR";

    } else {
            $rcode = "NXDOMAIN";
    }

    # mark the answer as authoritative (by setting the 'aa' flag)
    my $headermask = {aa => 1};

    # specify EDNS options  { option => value }
    my $optionmask = {};

    return ( $rcode, \@ans, \@auth, \@add, $headermask, $optionmask );
}


my $ns = new Net::DNS::Nameserver(
    LocalPort    => 5353,
    ReplyHandler => \&reply_handler,
    Verbose      => 1
```

```
    ) || die "couldn't create nameserver object\n";


$ns->main_loop;
```

## BUGS

Limitations in perl 5.8.6 makes it impossible to guarantee that replies to UDP queries from Net::DNS::Nameserver are sent from the IP-address they were received on. This is a problem for machines with multiple IP-addresses and causes violation of RFC2181 section 4. Thus a UDP socket created listening to INADDR_ANY (all available IP-addresses) will reply not necessarily with the source address being the one to which the request was sent, but rather with the address that the operating system chooses. This is also often called ''the closest address''. This should really only be a problem on a server which has more than one IP-address (besides localhost − any experience with IPv6 complications here, would be nice). If this is a problem for you, a work-around would be to not listen to INADDR_ANY but to specify each address that you want this module to listen on. A separate set of sockets will then be created for each IP-address.

## COPYRIGHT



## LICENSE



## SEE ALSO

perl, Net::DNS, Net::DNS::Resolver, Net::DNS::Packet, Net::DNS::Update, Net::DNS::Header, Net::DNS::Question, Net::DNS::RR, RFC 1035