NAME

AnyEvent::DNS – fully asynchronous DNS resolution

SYNOPSIS

```
use AnyEvent::DNS;

my $cv = AnyEvent->condvar;
AnyEvent::DNS::a "www.google.de", $cv;
# ... later
my @addrs = $cv->recv;
```

DESCRIPTION

This module offers both a number of DNS convenience functions as well as a fully asynchronous and high-performance pure-perl stub resolver.

The stub resolver supports DNS over IPv4 and IPv6, UDP and TCP, optional EDNS0 support for up to 4kiB datagrams and automatically falls back to virtual circuit mode for large responses.

CONVENIENCE FUNCTIONS

AnyEvent::DNS::a `$domain`, `$cb->(@addrs)`
: Tries to resolve the given domain to IPv4 address(es).

AnyEvent::DNS::aaaa `$domain`, `$cb->(@addrs)`
: Tries to resolve the given domain to IPv6 address(es).

AnyEvent::DNS::mx `$domain`, `$cb->(@hostnames)`
: Tries to resolve the given domain into a sorted (lower preference value first) list of domain names.

AnyEvent::DNS::ns `$domain`, `$cb->(@hostnames)`
: Tries to resolve the given domain name into a list of name servers.

AnyEvent::DNS::txt `$domain`, `$cb->(@hostnames)`
: Tries to resolve the given domain name into a list of text records. Only the first text string per record will be returned. If you want all strings, you need to call the resolver manually:

```
resolver->resolve ($domain => "txt", sub {
   for my $record (@_) {
      my (undef, undef, undef, @txt) = @$record;
      # strings now in @txt
   }
});
```

AnyEvent::DNS::srv `$service`, `$proto`, `$domain`, `$cb->(@srv_rr)`
: Tries to resolve the given service, protocol and domain name into a list of service records.

  Each `$srv_rr` is an array reference with the following contents: `[$priority, $weight, $transport, $target]`.

  They will be sorted with lowest priority first, then randomly distributed by weight as per RFC 2782.

  Example:

```
AnyEvent::DNS::srv "sip", "udp", "schmorp.de", sub { ...
# @_ = ( [10, 10, 5060, "sip1.schmorp.de" ] )
```

AnyEvent::DNS::any `$domain`, `$cb->(@rrs)`
: Tries to resolve the given domain and passes all resource records found to the callback. Note that this uses a DNS `ANY` query, which, as of RFC 8482, are officially deprecated.

AnyEvent::DNS::ptr `$domain`, `$cb->(@hostnames)`
: Tries to make a PTR lookup on the given domain. See `reverse_lookup` and `reverse_verify` if you want to resolve an IP address to a hostname instead.

AnyEvent::DNS::reverse_lookup `$ipv4_or_6`, `$cb->`(@hostnames)
: Tries to reverse-resolve the given IPv4 or IPv6 address (in textual form) into its hostname(s). Handles V4MAPPED and V4COMPAT IPv6 addresses transparently.

AnyEvent::DNS::reverse_verify `$ipv4_or_6`, `$cb->`(@hostnames)
: The same as `reverse_lookup`, but does forward-lookups to verify that the resolved hostnames indeed point to the address, which makes spoofing harder.

  If you want to resolve an address into a hostname, this is the preferred method: The DNS records could still change, but at least this function verified that the hostname, at one point in the past, pointed at the IP address you originally resolved.

  Example:

```
AnyEvent::DNS::reverse_verify "2001:500:2f::f", sub { print shift };
# => f.root-servers.net
```

LOW-LEVEL DNS EN-/DECODING FUNCTIONS

`$AnyEvent::DNS::EDNS0`
: This variable decides whether dns_pack automatically enables EDNS0 support. By default, this is disabled (`0`), unless overridden by `$ENV{PERL_ANYEVENT_EDNS0}`, but when set to `1`, AnyEvent::DNS will use EDNS0 in all requests.

`$pkt` = AnyEvent::DNS::dns_pack `$dns`
: Packs a perl data structure into a DNS packet. Reading RFC 1035 is strongly recommended, then everything will be totally clear. Or maybe not.

  Resource records are not yet encodable.

  Examples:

```
# very simple request, using lots of default values:
{ rd => 1, qd => [ [ "host.domain", "a"] ] }

# more complex example, showing how flags etc. are named:

{
   id => 10000,
   op => "query",
   rc => "nxdomain",

   # flags
   qr => 1,
   aa => 0,
   tc => 0,
   rd => 0,
   ra => 0,
   ad => 0,
   cd => 0,

   qd => [@rr], # query section
   an => [@rr], # answer section
   ns => [@rr], # authority section
   ar => [@rr], # additional records section
}
```

`$dns` = AnyEvent::DNS::dns_unpack `$pkt`
: Unpacks a DNS packet into a perl data structure.

  Examples:

```
# an unsuccessful reply
{
  'qd' => [
            [ 'ruth.plan9.de.mach.uni-karlsruhe.de', '*', 'in' ]
          ],
  'rc' => 'nxdomain',
  'ar' => [],
  'ns' => [
            [
              'uni-karlsruhe.de',
              'soa',
              'in',
              600,
              'netserv.rz.uni-karlsruhe.de',
              'hostmaster.rz.uni-karlsruhe.de',
              2008052201, 10800, 1800, 2592000, 86400
            ]
          ],
  'tc' => '',
  'ra' => 1,
  'qr' => 1,
  'id' => 45915,
  'aa' => '',
  'an' => [],
  'rd' => 1,
  'op' => 'query',
  '__' => '<original dns packet>',
}

# a successful reply

{
  'qd' => [ [ 'www.google.de', 'a', 'in' ] ],
  'rc' => 0,
  'ar' => [
            [ 'a.l.google.com', 'a', 'in', 3600, '209.85.139.9' ],
            [ 'b.l.google.com', 'a', 'in', 3600, '64.233.179.9' ],
            [ 'c.l.google.com', 'a', 'in', 3600, '64.233.161.9' ],
          ],
  'ns' => [
            [ 'l.google.com', 'ns', 'in', 3600, 'a.l.google.com' ],
            [ 'l.google.com', 'ns', 'in', 3600, 'b.l.google.com' ],
          ],
  'tc' => '',
  'ra' => 1,
  'qr' => 1,
  'id' => 64265,
  'aa' => '',
  'an' => [
            [ 'www.google.de', 'cname', 'in', 3600, 'www.google.com' ],
            [ 'www.google.com', 'cname', 'in', 3600, 'www.l.google.com' ],
            [ 'www.l.google.com', 'a', 'in', 3600, '66.249.93.104' ],
            [ 'www.l.google.com', 'a', 'in', 3600, '66.249.93.147' ],
          ],
```

```
      'rd' => 1,
      'op' => 0,
      '__' => '<original dns packet>',
    }
```

Extending DNS Encoder and Decoder

This section describes an *experimental* method to extend the DNS encoder and decoder with new opcode, rcode, class and type strings, as well as resource record decoders.

Since this is experimental, it can change, as anything can change, but this interface is expe ctedc to be relatively stable and was stable during the whole existence of `AnyEvent::DNS` so far.

Note that, since changing the decoder or encoder might break existing code, you should either be sure to control for this, or only temporarily change these values, e.g. like so:

```
my $decoded = do {
   local $AnyEvent::DNS::opcode_str{7} = "yxrrset";
   AnyEvent::DNS::dns_unpack $mypkt
};
```

`%AnyEvent::DNS::opcode_id`, `%AnyEvent::DNS::opcode_str`
: Two hashes that map lowercase opcode strings to numerical id's (For the encoder), or vice versa (for the decoder). Example: add a new opcode string `notzone`.

```
$AnyEvent::DNS::opcode_id{notzone} = 10;
$AnyEvent::DNS::opcode_str{10} = 'notzone';
```

`%AnyEvent::DNS::rcode_id`, `%AnyEvent::DNS::rcode_str`
: Same as above, for for rcode values.

`%AnyEvent::DNS::class_id`, `%AnyEvent::DNS::class_str`
: Same as above, but for resource record class names/values.

`%AnyEvent::DNS::type_id`, `%AnyEvent::DNS::type_str`
: Same as above, but for resource record type names/values.

`%AnyEvent::DNS::dec_rr`
: This hash maps resource record type values to code references. When decoding, they are called with `$_` set to the undecoded data portion and `$ofs` being the current byte offset. of the record. You should have a look at the existing implementations to understand how it works in detail, but here are two examples:

  Decode an A record. A records are simply four bytes with one byte per address component, so the decoder simply unpacks them and joins them with dots in between:

```
$AnyEvent::DNS::dec_rr{1} = sub { join ".", unpack "C4", $_ };
```

  Decode a CNAME record, which contains a potentially compressed domain name.

```
package AnyEvent::DNS; # for %dec_rr, $ofsd and &_dec_name
$dec_rr{5} = sub { local $ofs = $ofs - length; _dec_name };
```

THE AnyEvent::DNS RESOLVER CLASS

This is the class which does the actual protocol work.

AnyEvent::DNS::resolver
: This function creates and returns a resolver that is ready to use and should mimic the default resolver for your system as good as possible. It is used by AnyEvent itself as well.

  It only ever creates one resolver and returns this one on subsequent calls – see `$AnyEvent::DNS::RESOLVER`, below, for details.

  Unless you have special needs, prefer this function over creating your own resolver object.

  The resolver is created with the following parameters:

```
   untaint           enabled
   max_outstanding   $ENV{PERL_ANYEVENT_MAX_OUTSTANDING_DNS} (default 10)
```

os_config will be used for OS-specific configuration, unless $ENV{PERL_ANYEVENT_RESOLV_CONF} is specified, in which case that file gets parsed.

$AnyEvent::DNS::RESOLVER
: This variable stores the default resolver returned by AnyEvent::DNS::resolver, or undef when the default resolver hasn't been instantiated yet.

  One can provide a custom resolver (e.g. one with caching functionality) by storing it in this variable, causing all subsequent resolves done via AnyEvent::DNS::resolver to be done via the custom one.

$resolver = new AnyEvent::DNS key => value...
: Creates and returns a new resolver.

  The following options are supported:

  server => [...]
  : A list of server addresses (default: v127.0.0.1 or ::1) in network format (i.e. as returned by AnyEvent::Socket::parse_address – both IPv4 and IPv6 are supported).

  timeout => [...]
  : A list of timeouts to use (also determines the number of retries). To make three retries with individual time-outs of 2, 5 and 5 seconds, use [2, 5, 5], which is also the default.

  search => [...]
  : The default search list of suffixes to append to a domain name (default: none).

  ndots => $integer
  : The number of dots (default: 1) that a name must have so that the resolver tries to resolve the name without any suffixes first.

  max_outstanding => $integer
  : Most name servers do not handle many parallel requests very well. This option limits the number of outstanding requests to $integer (default: 10), that means if you request more than this many requests, then the additional requests will be queued until some other requests have been resolved.

  reuse => $seconds
  : The number of seconds (default: 300) that a query id cannot be re-used after a timeout. If there was no time-out then query ids can be reused immediately.

  untaint => $boolean
  : When true, then the resolver will automatically untaint results, and might also ignore certain environment variables.

$resolver->parse_resolv_conf ($string)
: Parses the given string as if it were a *resolv.conf* file. The following directives are supported (but not necessarily implemented).

  #- and ;-style comments, nameserver, domain, search, sortlist, options (timeout, attempts, ndots).

  Everything else is silently ignored.

$resolver->os_config
: Tries so load and parse */etc/resolv.conf* on portable operating systems. Tries various egregious hacks on windows to force the DNS servers and searchlist out of the system.

  This method must be called at most once before trying to resolve anything.

`$resolver->`timeout ($timeout, ...)
: Sets the timeout values. See the `timeout` constructor argument (and note that this method expects the timeout values themselves, not an array-reference).

`$resolver->`max_outstanding ($nrequests)
: Sets the maximum number of outstanding requests to `$nrequests`. See the `max_outstanding` constructor argument.

`$resolver->`request ($req, `$cb->`($res))
: This is the main low-level workhorse for sending DNS requests.

  This function sends a single request (a hash-ref formatted as specified for `dns_pack`) to the configured nameservers in turn until it gets a response. It handles timeouts, retries and automatically falls back to virtual circuit mode (TCP) when it receives a truncated reply. It does not handle anything else, such as the domain searchlist or relative names – use `->resolve` for that.

  Calls the callback with the decoded response packet if a reply was received, or no arguments in case none of the servers answered.

`$resolver->`resolve ($qname, `$qtype`, `%options`, `$cb->`(@rr))
: Queries the DNS for the given domain name `$qname` of type `$qtype`.

  A `$qtype` is either a numerical query type (e.g. `1` for A records) or a lowercase name (you have to look at the source to see which aliases are supported, but all types from RFC 1035, `aaaa`, `srv`, `spf` and a few more are known to this module). A `$qtype` of "*" is supported and means "any" record type.

  The callback will be invoked with a list of matching result records or none on any error or if the name could not be found.

  CNAME chains (although illegal) are followed up to a length of 10.

  The callback will be invoked with arraryefs of the form `[$name, $type, $class, $ttl, @data]`, where `$name` is the domain name, `$type` a type string or number, `$class` a class name, `$ttl` is the remaining time-to-live and `@data` is resource-record-dependent data, in seconds. For `a` records, this will be the textual IPv4 addresses, for `ns` or `cname` records this will be a domain name, for `txt` records these are all the strings and so on.

  All types mentioned in RFC 1035, `aaaa`, `srv`, `naptr` and `spf` are decoded. All resource records not known to this module will have the raw `rdata` field as fifth array element.

  Note that this resolver is just a stub resolver: it requires a name server supporting recursive queries, will not do any recursive queries itself and is not secure when used against an untrusted name server.

  The following options are supported:

  search => [$suffix...]
  : Use the given search list (which might be empty), by appending each one in turn to the `$qname`. If this option is missing then the configured `ndots` and `search` values define its value (depending on `ndots`, the empty suffix will be prepended or appended to that `search` value). If the `$qname` ends in a dot, then the searchlist will be ignored.

  accept => [$type...]
  : Lists the acceptable result types: only result types in this set will be accepted and returned. The default includes the `$qtype` and nothing else. If this list includes `cname`, then CNAME-chains will not be followed (because you asked for the CNAME record).

  class => "class"
  : Specify the query class ("in" for internet, "ch" for chaosnet and "hs" for hesiod are the only ones making sense). The default is "in", of course.

  Examples:

```
   # full example, you can paste this into perl:
   use Data::Dumper;
   use AnyEvent::DNS;
   AnyEvent::DNS::resolver->resolve (
      "google.com", "*", my $cv = AnyEvent->condvar);
   warn Dumper [$cv->recv];

   # shortened result:
   # [
   #   [ 'google.com', 'soa', 'in', 3600, 'ns1.google.com', 'dns-admin.google.
   #     2008052701, 7200, 1800, 1209600, 300 ],
   #   [
   #     'google.com', 'txt', 'in', 3600,
   #     'v=spf1 include:_netblocks.google.com ~all'
   #   ],
   #   [ 'google.com', 'a', 'in', 3600, '64.233.187.99' ],
   #   [ 'google.com', 'mx', 'in', 3600, 10, 'smtp2.google.com' ],
   #   [ 'google.com', 'ns', 'in', 3600, 'ns2.google.com' ],
   # ]

   # resolve a records:
   $res->resolve ("ruth.plan9.de", "a", sub { warn Dumper [@_] });

   # result:
   # [
   #   [ 'ruth.schmorp.de', 'a', 'in', 86400, '129.13.162.95' ]
   # ]

   # resolve any records, but return only a and aaaa records:
   $res->resolve ("test1.laendle", "*",
      accept => ["a", "aaaa"],
      sub {
         warn Dumper [@_];
      }
   );

   # result:
   # [
   #   [ 'test1.laendle', 'a', 'in', 86400, '10.0.0.255' ],
   #   [ 'test1.laendle', 'aaaa', 'in', 60, '3ffe:1900:4545:0002:0240:0000:000
   # ]
```

`$resolver->wait_for_slot ($cb->($resolver))`

Wait until a free request slot is available and call the callback with the resolver object.

A request slot is used each time a request is actually sent to the nameservers: There are never more than `max_outstanding` of them.

Although you can submit more requests (they will simply be queued until a request slot becomes available), sometimes, usually for rate-limiting purposes, it is useful to instead wait for a slot before generating the request (or simply to know when the request load is low enough so one can submit requests again).

This is what this method does: The callback will be called when submitting a DNS request will not result in that request being queued. The callback may or may not generate any requests in response.

Note that the callback will only be invoked when the request queue is empty, so this does not play well

if somebody else keeps the request queue full at all times.

AUTHOR

Marc Lehmann <schmorp@schmorp.de>
http://anyevent.schmorp.de