asyncdns Documentation

Release 0.1.2

Alastair Houghton

Nov 15, 2018

Contents

CHAPTER 1

What is this?

`asyncdns` is a pure Python asynchronous DNS resolver implementation written on top of asyncio. It doesn't require any external libraries, and it doesn't use threads or blocking functions.

CHAPTER 2

Usage

`asyncdns` doesn't have an equivalent to the widely used `gethostbyname()` or `getaddrinfo()` functions. Instead, you use it by constructing a `Query` object specifying the DNS query you wish to run, then pass it to a `Resolver` to actually perform the query.

There are a handful of built-in resolvers, but for demonstration purposes the easiest one to use is the `SmartResolver`, which automatically makes use of `/etc/hosts`, multicast DNS and regular DNS as appropriate.

For instance, do a simple lookup for an A record:

```
>>> import asyncdns, asyncio
>>> resolver = asyncdns.SmartResolver()
>>> loop = asyncio.get_event_loop()
>>> query = asyncdns.Query('www.example.com', asyncdns.A, asyncdns.IN)
>>> f = resolver.lookup(query)
>>> loop.run_until_complete(f)
>>> print(f.result())
;; No error (RD, RA)
; 1 answers:
www.example.com       54950   IN      A       93.184.216.34
; 0 authorities:
; 0 additional:
```

Note that you may or may not want to use `SmartResolver` in your code, depending on your requirements - it probably isn't a good idea using multicast DNS on an untrusted network, for instance.

Contents:

2.1 Queries

DNS queries are represented by *`Query`* objects, which hold a name to look up, a query type and a query class.

`class Query`(*name*, *q_type*, *q_class*)

Represents a DNS query.

Parameters

- **name** (*ipaddress.IPv6Address*) – The name to query.
- **q_type** (*int*) – The RR type you're querying for.
- **q_class** (*int*) – The RR class you're querying for.

name may be specified as a Python string (in which case, IDNA is applied if necessary); or as a Python `bytes` object (in which case the bytes are used literally, subject to the usual rules on DNS labels); or an IP address using the `ipaddress` module's `IPv4Address` or `IPv6Address` objects, in which case the address will be automatically turned into the appropriate form for a reverse lookup.

There are constants for most query types in the `asyncdns` module, e.g. `asyncdns.A`, `asyncdns.AAAA` and so on, but you can use the numeric value if required. Possible values are:

Constant	Value	Meaning
A	1	IPv4 address
NS	2	Nameserver
MD	3	Mail destination (obsolete)
MF	4	Mail forwarder (obsolete)
CNAME	5	Canonical name record - an alias
SOA	6	Start Of Authority
MB	7	Mailbox domain name (obsolete)
MG	8	Mail group member (obsolete)
MR	9	Mail rename (obsolete)
NUL	10	Null
WKS	11	Well Known Service description (obsolete)
PTR	12	Pointer - for inverse queries
HINFO	13	Host information (obsolete)
MINFO	14	Mailbox or list information (obsolete)
MX	15	Mail eXchanger
TXT	16	Free format text
RP	17	Responsible person
AFSDB	18	AFS database record
X25	19	X.121 address, as used on X.25 networks
ISDN	20	ISDN address
RT	21	Route record, for X.25 or ISDN
NSAP	22	OSI NSAP address
NSAPPTR	23	NSAP Pointer - for inverse queries
SIG	24	(Old) DNSSEC signature (obsolete)
KEY	25	(Old) DNSSEC key (obsolete)
PX	26	X.400 mail mapping information (obsolete)
GPOS	27	Geographical location (obsolete)
AAAA	28	IPv6 address
LOC	29	Geographical location
NXT	30	(Old) DNSSEC Next record (obsolete)
EID	31	Nimrod Endpoint Identifier (obsolete)
NIMLOC	32	Nimrod Locator (obsolete)
SRV	33	Service locator
ATMA	34	ATM address

Continued on next page

Table 1 – continued from previous page

NAPTR	35	Naming Authority Pointer - regex rewriting
KX	36	Key exchanger record
CERT	37	Certificate record
A6	38	Intended to replace AAAA (obsolete)
DNAME	39	Alias for a name *and all subnames*
SINK	40	Kitchen sink (joke, obsolete)
OPT	41	EDNS option (PSEUDO-RR)
APL	42	Address Prefix List
DS	43	Delegation Signer record
SSHFP	44	SSH public key fingerprint
IPSECKEY	45	IPsec key
RRSIG	46	DNSSEC signature
NSEC	47	Next Secure record - to prove non-existence
DNSKEY	48	DNSSEC key record
DHCID	49	DHCP identifier
NSEC3	50	Next Secure record (v3)
NSEC3PARAM	51	NSEC3 parameter record
TLSA	52	TLSA cetificate association
HIP	55	Host Identity Protocol record
CDS	59	Child DS record
CDNSKEY	60	Child DNSKEY
OPENPGPKEY	61	OpenPGP public key
SPF	99	SPF record (obsolete)
UINFO	100	Reserved
UID	101	Reserved
GID	102	Reserved
UNSPEC	103	Reserved
TKEY	249	Transaction key
TSIG	250	Transaction signature
IXFR	251	Incremental zone transfer (PSEUDO-RR)
AXFR	252	Authoritative zone transfers (PSEUDO-RR)
MAILB	253	Used to get MB/MG/MR/MINFO records (obsolete)
MAILA	254	Used to retrieve MD or MF records (obsolete)
ANY	255	Return all record types (PSEUDO-RR)
URI	256	Maps a hostname to a URI
CAA	257	Certificate Authority Authorization
TA	32768	DNSSEC Trust Authorities
DLV	32769	DNSSEC Lookaside Validation record

The query class will almost always be `asyncdns.IN`. Possible values are:

Constant	Value	Meaning
IN	1	Internet
CH	3	Chaos
HS	4	Hesiod
NONE	254	
ANY	255	

__lt__(*other*)

__eq__(*other*)

__ne__(*other*)

__gt__(*other*)

__ge__(*other*)

__le__(*other*)

Query provides comparison and ordering operators.

__hash__()

Query is also hashable, so it can be used as a key in a `dict` or `set`.

__repr__()

Returns a debug representation.

2.2 Replies

Replies are represented by *Reply* objects, which hold the flags, RCODE, and three sets of returned RRs (answers, authorities and additional).

class Reply

flags

The flags returned by the server. These are as follows:

Constant	Value	Meaning
AA	0x0400	Authoritative Answer
TC	0x0200	Truncated Response
RD	0x0100	Recursion Desired
RA	0x0080	Recursion Allowed
Z	0x0040	Reserved
AD	0x0020	Authentic Data (DNSSEC)
CD	0x0010	Checking Disabled (DNSSEC)

rcode

The RCODE returned by the server. Possible values are:

Constant	Value	Meaning
NOERROR	0	Successful query
FORMERR	1	Format failure
SERVFAIL	2	Server failure
NXDOMAIN	3	Non-existent domain
NOTIMP	4	Not implemented
REFUSED	5	Query refursed
YXDOMAIN	6	Name exists when it should not
YXRRSET	7	RR set exists when it should not
NXRRSET	8	RR set that should exist does not
NOTAUTH	9	Server not authoritative OR Not authorized
NOTZONE	10	Name not contained in zone
BADVERS	16	Bad OPT version
BADSIG	16	TSIG signature failure
BADKEY	17	Key not recognized
BADTIME	18	Signature out of time window
BADMODE	19	Bad TKEY mode
BADNAME	20	Duplicate key name
BADALG	21	Algorithm not supported
BADTRUNC	22	Bad truncation
BADCOOKIE	23	Bad/missing server cookie

answers
A list of *rr.RR* returned by the server in the Answers section of the reply.

authorities
A list of *rr.RR* returned by the server in the Authorities section of the reply.

additional
A list of additional *rr.RR* returned by the server.

2.2.1 RRs

RRs are represented by subclasses of *rr.RR*; a handful of common RR types have special subclasses that decode the RDATA field in the DNS reply for you. If you are using some other type of RR, you can create your own subclass and register it using *rr.RR.register()*, or you can just decode the data in your own code.

class rr.**RR**(*name*, *rr_type*, *rr_class*, *ttl*)
The base class of all RRs. You won't get a raw *rr.RR* in a Reply - RRs that we don't understand are mapped to *rr.Unknown*.

name
The associated domain name, in the form given in the DNS packet (a `bytes`).

unicode_name
The associated domain name, after IDNA processing (a `str`)

rr_type
The RR type (see `query` for a list).

rr_class
The RR class (see `query` for a list).

ttl
The remaining time to live for this RR, in seconds. Note that this field is only updated

register(*rr_type*, *rr_class*, *pyclass*)
Register a subclass of *rr.RR*; when we decode a response from the DNS server, we will create an instance of the specified class to represent RRs of the specified type and class.

Parameters

- **rr_type** (*int*) – The RR type to map.
- **rr_class** (*int*) – The RR class to map, or ANY if the mapping should operate for any class.
- **pyclass** – The Python class we should use for RRs of the specified type and class.

decode(*name*, *rr_type*, *rr_class*, *ttl*, *packet*, *ptr*, *rdlen*)
Decode an RR from a DNS packet, returning a new *rr.RR* instance representing it. The implementation in *rr.RR* looks up the correct Python class and calls its *decode()* method; if it doesn't find a class registered for the RR type with which it's presented, it will use *rr.Unknown*.

Parameters

- **name** (*bytes*) – The domain name.
- **rr_type** (*int*) – The RR type.
- **rr_class** (*int*) – The RR class.
- **ttl** (*int*) – The remaining time to live for this RR.
- **packet** (*bytes*) – The entire DNS response packet.
- **ptr** (*int*) – The current offset within the DNS packet.
- **rdlen** (*int*) – The length of the RR's data, starting from ptr.

class rr.**A**(*name*, *ttl*, *address*)

address
The IPv4 address (an ipaddress.IPv4Address).

class rr.**AAAA**(*name*, *ttl*, *address*)

address
The IPv6 address (an ipaddress.IPv6Address).

class rr.**CNAME**(*name*, *ttl*, *address*)

cname
The aliased name, in the form given in the DNS packet (a bytes).

unicode_cname
The aliased name after IDNA processing (a str)

class rr.**HINFO**(*name*, *ttl*, *cpu*, *os*)

cpu
The CPU model (as a string).

os
The operating system (as a string).

Note that the RFC does not specify the encoding of either string, so for maximum robustness we decode the data as ISO Latin 1. In most cases we would expect the two fields to be ASCII; if they are not, each code point in the resulting string with have the same value as the byte in the byte string.

class rr.**MB**(*name*, *ttl*, *host*)

host
The host specified in the record.

unicode_host
The host name after IDNA processing.

class rr.**MF**(*name*, *ttl*, *host*)

host
The host specified in the record.

unicode_host
The host name after IDNA processing.

class rr.**MG**(*name*, *ttl*, *mailbox*)

mailbox
The mailbox specified in the record.

unicode_mailbox
The mailbox name after IDNA processing.

class rr.**MINFO**(*name*, *ttl*, *mailbox*)

rmailbox

emailbox
The mailboxes specified in the record.

unicode_rmailbox

unicode_emailbox
The mailbox names after IDNA processing.

class rr.**MR**(*name*, *ttl*, *mailbox*)

mailbox
The mailbox specified in the record.

unicode_mailbox
The mailbox name after IDNA processing.

class rr.**MX**(*name*, *ttl*, *preference*, *exchange*)

preference
The mail exchanger priority from the DNS record.

exchange
The mail exchanger hostname as found in the DNS packet.

unicode_exchange
The mail exchanger hostname after IDNA processing.

class rr.**NUL**(*name*, *ttl*, *data*)

data
The RDATA from the record.

class rr.**NS**(*name*, *ttl*, *host*)

host
The hostname of the nameserver.

unicode_host
The hostname of the nameserver after IDNA processing.

class rr.**PTR**(*name*, *ttl*, *dname*)

address
The IPv4 or IPv6 address, decoded from *name*, or `None` if no address could be decoded.

dname
The name pointed to by this record.

unicode_host
The name poitned to by this record, after IDNA processing.

rr.SOA(name, ttl, mname, rname, serial, refresh, retry, expire, minimum)

mname
The name of the primary mailserver for the zone.

unicode_mname
Same as above, but after IDNA processing.

rname
The mailbox name of the person responsible for the zone.

unicode_rname
As above, but after IDNA processing.

serial
The zone's serial number; this is used to detect changes to a zone (it must be incremented every time a zone is changed).

refresh
The number of seconds for which a secondary nameserver may assume the zone data has not changed - controls how often the secondary checks the zone serial number.

retry
The number of seconds a secondary should wait to retry a refresh if the primary nameserver is busy.

expire
The number of seconds a secondary nameserver can cache the data before it is no longer authoritative.

minimum
The minimum time to live for RRs in the zone.

class rr.**TXT**(*name*, *ttl*, *text*)

text
The stored text. Since no encoding is specified, this is decoded as ISO Latin 1 (since that is the most robust option).

class rr.**WKS**(*name*, *ttl*, *address*, *protocol*, *bitmap*)

address
The IPv4 address for this record.

protocol
The IP protocol number for this record (typically 6, for TCP, or 17, for UDP).

bitmap
A bytes holding the port bitmap.

class rr.**Unknown**(*name*, *ttl*, *rr_type*, *rr_class*, *ttl*, *data*)
This subclass of *rr.RR* is used when we don't know how to decode the RR found in the data packet.

data
The RDATA from the record.

2.3 What are Resolvers?

In asyncdns, Resolvers are the objects that are responsible for taking *Query* objects and returning *Reply* objects corresponding to those queries.

Resolvers don't derive from a single base class, as some of them work quite differently to others. Instead, they all implement the following two methods:

close()
Cancel all in-progress lookups and shut down the resolver.

lookup(*query*)

Parameters **query** – The *Query* to process.

Retval An asyncio.Future that will complete with a *Reply*.

Resolvers are guaranteed to cancel lookups that are in progress when the resolver itself is destroyed. Active lookups do not keep a resolver alive.

Individual resolvers may support additional parameters for their *lookup()* method, but those parameters are generally specific to the workings of the resolver in question.

2.3.1 Resolver

class Resolver
The core DNS resolver. This class holds all of the code to perform normal DNS queries, including recursive resolution, and maintains its own request cache, so that repeatedly querying for the same record won't result in unnecessary network traffic or delay.

lookup(query, servers=None, should_cache=True, recursive=False, prefer_ipv6=False, force_tcp=False)
Perform a DNS lookup.

Parameters

- **query** – The *Query* to resolve.

- **servers** – See discussion below.
- **should_cache** – Setting this to False disables the *Resolver* cache.
- **recursive** – Whether to perform recursive lookups.
- **prefer_ipv6** – When doing recursive lookup, prefer servers that talk over IPv6.
- **force_tcp** – Prevents the resolver from using UDP for queries that are short enough to fit.

Retval An `asyncio.Future` that will complete with a *Reply*.

The `servers` parameter can be:

- An (*address*, *port*) tuple.
- A `list` of (*address*, *port*) tuples, which will be used randomly.
- An iterable of some sort that yields (*address*, *port*) tuples. Note that if the iterable raises StopIteration, any in-progress queries will fail with the StopIteration exception.
- `None`, in which case the resolver will be recursive (regardless of the setting of the `recursive` parameter) and will start with the global root servers. We recommend not using this feature unless absolutely necessary, as it puts additional load on the root servers and it's usually better to talk to your own nameserver or use one provided by your ISP or infrastructure platform.

`asyncdns` provides two useful iterables, `RandomServer` and `RoundRobinServer`, both of which provide an infinite stream of tuples given a list of server addresses.

flush_cache()
Flushes the resolver's cache.

2.3.2 HostsResolver

class HostsResolver
Resolves names using the contents of `/etc/hosts` (or, on Windows, `\Windows\System32\drivers\etc\hosts`).

lookup(*query*)

Parameters **query** – The *Query* to resolve.

Retval An `asyncio.Future` that will complete with a *Reply*.

This method only supports A, AAAA and PTR queries. In addition to names listed in `/etc/hosts`, it knows about the `.in-addr.arpa` and `.ip6.arpa` pseudo-zones.

The *HostsResolver* will automatically re-read `/etc/hosts` if it has changed, but only if the last time it was read was more than 30 seconds ago.

2.3.3 MulticastResolver

class MulticastResolver
Resolves queries using Multicast DNS (aka MDNS). You don't need to have Apple's mdnsResponder software installed to use this - it will work on any system that can run Python and that supports IP multicast.

lookup(*query*, *use_ipv6=False*, *unicast_reply=False*)

Parameters

- **query** – The *Query* to resolve.

- **use_ipv6** – Whether to multicast using IPv6 or not. The default is to use IPv4.
- **unicast_reply** – Whether to request that the reply be sent via unicast. This is intended to reduce multicast traffic.

Retval An `asyncio.Future` that will complete with a *Reply*.

2.3.4 SystemResolver

SystemResolver is actually a "class cluster", in that there are separate implementations for Darwin/Mac OS X/macOS, Windows, and generic UNIX/Linux. The idea of *SystemResolver* is that it works like *Resolver*, but uses the system configured nameservers (and will automatically update its list of nameservers should the system configuration change).

There are some limitations here: the UNIX/Linux generic implementation works by reading `/etc/resolv.conf`, so any other configuration mechanism that might be in use will be ignored, while the Windows version uses Windows APIs that appear to be limited to returning IPv4 nameservers only. On Windows, there doesn't seem to be a mechanism to spot changes to the configuration, so we re-read it at most once every 30 seconds; on UNIX/Linux, we watch the timestamp on `/etc/resolv.conf`, again, at most once every 30 seconds. Some people have suggested using `res_ninit()` on UNIX rather than directly reading `/etc/resolv.conf`; that's certainly a possibility, but if `/etc/resolv.conf` isn't being used to configure the nameservers, we'd end up in the same situation as on Windows, where we have no way to tell if the server settings have been updated.

class SystemResolver

lookup(query, servers=None, should_cache=True, recursive=False, prefer_ipv6=False, force_tcp=False)

Perform a DNS lookup.

Parameters

- **query** – The *Query* to resolve.
- **should_cache** – Setting this to False disables the *Resolver* cache.
- **recursive** – Whether to perform recursive lookups.
- **prefer_ipv6** – When doing recursive lookup, prefer servers that talk over IPv6.
- **force_tcp** – Prevents the resolver from using UDP for queries that are short enough to fit.

Retval An `asyncio.Future` that will complete with a *Reply*.

2.3.5 SmartResolver

`SmartResolver` is a convenience class that accepts a query and determines which of the other resolvers to use to process it. Specifically:

- It first tries *HostsResolver*, which means the hosts file can override resolution the way people expect.
- If that fails and the query is for a name ending `.local`, it uses *MulticastResolver*.
- Otherwise, it uses *SystemResolver*.

N.B. Pay attention to the security implications of using *MulticastResolver* here; if you are using a server platform where multicast isn't appropriately restricted, this could open up a security hole that causes you to send data to an attacker's system instead of the one you wanted to.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Index

S

T

U