

---

# **btdht Documentation**

***Release 0.3.2***

**Valentin Samir**

**Mar 03, 2018**



---

## Contents

---

<b>1</b>	<b>btdht: A python implementation of the Bittorrent distributed hash table</b>	<b>3</b>
1.1	Dependencies . . . . .	3
1.2	Build dependencies . . . . .	4
1.3	Installation . . . . .	4
1.4	Usage examples . . . . .	4
<b>2</b>	<b>btdht package</b>	<b>7</b>
2.1	Submodules . . . . .	7
2.1.1	btdht.dht module . . . . .	7
2.1.2	btdht.utils module . . . . .	20
2.1.3	btdht.krcp module . . . . .	25
2.1.4	btdht.exceptions module . . . . .	29
2.2	Module contents . . . . .	30
<b>3</b>	<b>Indices and tables</b>	<b>31</b>
	<b>Python Module Index</b>	<b>33</b>



Contents:



---

## btdht: A python implementation of the Bittorrent distributed hash table

---

The aim of btdht is to provide a powerful implementation of the Bittorrent mainline DHT easily extended to build application over the DHT. The author currently uses it to crawl the dht and has been able to retrieve more than 200.000 torrents files a day.

The implementation is fully compliant with the [BEP5](#) and the kademlia paper<sup>1</sup> (with a predominance of the BEP5 over the paper) For example, this implementation uses a bucket-based approach for the routing table.

### Table of Contents

- *btdht: A python implementation of the Bittorrent distributed hash table*
  - *Dependencies*
  - *Build dependencies*
  - *Installation*
  - *Usage examples*

## 1.1 Dependencies

- python 2.7 or 3.4 or above
- [datrie](#)
- [netaddr](#)

---

<sup>1</sup> Maymounkov, P., & Mazieres, D. (2002, March). Kademlia: A peer-to-peer information system based on the xor metric. In International Workshop on Peer-to-Peer Systems (pp. 53-65). Springer Berlin Heidelberg.

## 1.2 Build dependencies

- A C compiler
- `cython`
- python header files

## 1.3 Installation

The recommended installation mode is to use a `virtualenv`.

To Install `btdht` using the last published release, run:

```
$ pip install btdht
```

Alternatively if you want to use the version of the git repository, you can clone it:

```
$ git clone https://github.com/nitmir/btdht
$ cd btdht
$ pip install -r requirements-dev.txt
```

Then, run `make install` to compile the sources and create a python package and install it with `pip`.

For installing or building on linux and unix systems, you will need a C compiler and the python headers (installing the packages `build-essential` and `python-dev` should be enough on debian like systems, you'll probably gonna need `make`, `gcc`, `python2-devel` and `redhat-rpm-config` on centos like systems).

On windows systems, we provide pre-built releases for python 2.7 and 3.5 so just running `pip install btdht` should be fine. If you want to build from the sources of the repository or, for another python version, you will also need a `C compiler`.

## 1.4 Usage examples

Search for the peers announcing the torrent `0403fb4728bd788fbc67e87d6feb241ef38c75a` ([Ubuntu 16.10 Desktop \(64-bit\)](#))

```
>>> import btdht
>>> import binascii
>>> dht = btdht.DHT()
>>> dht.start() # now wait at least 15s for the dht to bootstrap
init socket for 4c323257aa6c4c5c6ccae118db93ccce5bb05d92
Bootstrapping
>>> dht.get_peers(binascii.a2b_hex("0403fb4728bd788fbc67e87d6feb241ef38c75a"))
[
  ('81.171.107.75', 17744),
  ('94.242.250.86', 3813),
  ('88.175.164.228', 32428),
  ('82.224.107.213', 61667),
  ('85.56.118.178', 6881),
  ('78.196.28.4', 38379),
  ('82.251.140.70', 32529),
  ('78.198.108.3', 10088),
  ('78.235.153.136', 10619),
```

(continues on next page)



(continued from previous page)

```
( '88.189.113.32', 33192),  
( '81.57.9.183', 5514),  
( '82.251.17.155', 14721),  
( '88.168.207.178', 31466),  
( '82.238.89.236', 32970),  
( '78.226.209.88', 2881),  
( '5.164.219.48', 6881),  
( '78.225.252.39', 31002)  
]
```

Subsequent calls to `get_peers` may return more peers.

You may also inherit `btdht.DHT_BASE` and overload some of the `on_`msg`_(query|response)` functions. See the [doc](#) for a full overview of the `btdht` API.



## 2.1 Submodules

### 2.1.1 btdht.dht module

<i>DHT</i>	A DHT class ready for instantiation
<i>DHT_BASE</i>	The DHT base class
<i>Node</i>	A node of the dht in the routing table
<i>Bucket</i>	A bucket of nodes in the routing table
<i>RoutingTable</i>	A routing table for one or more <i>DHT_BASE</i> instances

**class** `btdht.dht.DHT`

Bases: `btdht.dht.DHT_BASE`

A DHT class ready for instantiation

#### Parameters

- **routing\_table** (`RoutingTable`) – An optional routing table, possibly shared between several dht instances. If not specified, a new routing table is instantiated.
- **bind\_port** (`int`) – An optional udp port to use for the dht instance. If not specified, the hosting system will choose an available port.
- **bind\_ip** (`str`) – The interface to listen to. The default is "0.0.0.0".
- **id** (`bytes`) – An optional 160 bits long (20 Bytes) id. If not specified, a random one is generated.
- **ignored\_ip** (`set`) – A set of ip address in dotted ("1.2.3.4") notation to ignore. The default is the empty set.
- **debuglvl** (`int`) – Level of verbosity, default to 0.
- **prefix** (`str`) – A prefix to use in logged messages. The default is "".

- **process\_queue\_size** (*int*) – Size of the queue of messages waiting to be processed by user defines functions (on\_‘msg’\_(query|response)). see the [register\\_message](#) method. The default to 500.
- **ignored\_net** (*list*) – An list of ip networks in cidr notation ("1.2.3.4/5") to ignore. The default is the value of the attribute `ignored_net`.
- **scheduler** (`btdht.utils.Scheduler`) – A optional [Scheduler](#) instance. If not specified, a new [Scheduler](#) is instantiated.

**Note:** try to use same `id` and `bind_port` over dht restart to increase the probability to remain in other nodes routing table

**class** `btdht.dht.DHT_BASE`

Bases: `object`

The DHT base class

#### Parameters

- **routing\_table** (`RoutingTable`) – An optional routing table, possibly shared between several dht instances. If not specified, a new routing table is instantiated.
- **bind\_port** (*int*) – And optional udp port to use for the dht instance. If not specified, the hosting system will choose an available port.
- **bind\_ip** (*str*) – The interface to listen to. The default is "0.0.0.0".
- **id** (*bytes*) – An optional 160 bits long (20 Bytes) id. If not specified, a random one is generated.
- **ignored\_ip** (*set*) – A set of ip address in dotted ("1.2.3.4") notation to ignore. The default is the empty set.
- **debuglvl** (*int*) – Level of verbosity, default to 0.
- **prefix** (*str*) – A prefix to use in logged messages. The default is "".
- **process\_queue\_size** (*int*) – Size of the queue of messages waiting to be processed by user defines functions (on\_‘msg’\_(query|response)). see the [register\\_message](#) method. The default to 500.
- **ignored\_net** (*list*) – An list of ip networks in cidr notation ("1.2.3.4/5") to ignore. The default is the value of the attribute `ignored_net`.
- **scheduler** (`btdht.utils.Scheduler`) – A optional [Scheduler](#) instance. If not specified, a new [Scheduler](#) is instantiated.

**Note:** try to use same `id` and `bind_port` over dht restart to increase the probability to remain in other nodes routing table

**bind\_ip** = '0.0.0.0'

*str* interface the dht is binded to

**bind\_port** = None

*int* port the dht is binded to

**debuglvl** = 0

*int* the dht instance verbosity level

**last\_msg** = 0

last time we received any message

**last\_msg\_rep** = 0  
last time we receive a response to one of our messages

**ignored\_ip** = []  
set of ignored ip in dotted notation

**ignored\_net** = ['0.0.0.0/8', '10.0.0.0/8', '100.64.0.0/10', '127.0.0.0/8', '169.254.0.0/16']  
list of default ignored ip networks

**myid** = None  
utils.ID the dht instance id, 160bits long (20 Bytes)

**prefix** = ''  
str prefixing all debug message

**root** = None  
RoutingTable the used instance of the routing table

**sock** = None  
The current dht socket.socket

**stopped** = True  
the state (stopped ?) of the dht

**threads** = []  
list of the Thread of the dht instance

**token** = {}  
Token send with get\_peers response. Map between ip addresses and a list of random token. A new token by ip is generated at most every 5 min, a single token is valid 10 min. On reception of a announce\_peer query from ip, the query is only accepted if we have a valid token (generated less than 10min ago).

**mytoken** = {}  
Tokens received on get\_peers response. Map between ip addresses and received token from ip. Needed to send announce\_peer to that particular ip.

**transaction\_type** = {}  
Map between transaction id and messages type (to be able to match responses)

**to\_send** = <btdht.utils.PollableQueue instance>  
A PollableQueue of messages (data, (ip, port)) to send

**to\_schedule** = []  
A list of looping iterator to schedule, passed to \_scheduler

**zombie**  
True if dht is stopped but one thread or more remains alive, False otherwise

**save** (filename=None, max\_node=None)  
save the current list of nodes to filename.

#### Parameters

- **filename** (str) – An optional filename where to save the current list of nodes. If not provided, the file "dht\_`myid`.status" is used.
- **max\_node** (int) – An optional integer to limit the number of saved nodes. If not provided, all of the routing table nodes are saved.

**load** (filename=None, max\_node=None)  
load a list of nodes from filename.

#### Parameters

- **filename** (*str*) – An optional filename where to load the list of nodes. If not provided, the file "dht\_`myid`.status" is used.
- **max\_node** (*int*) – An optional integer to limit the number of loaded nodes. If not provided, all of the file nodes are loaded.

**start** (*start\_routing\_table=True, start\_scheduler=True*)

**Start the dht:**

- initialize some attributes
- initialize the dht socket (see :meth:~init\_socket)
- register this instance of the dht in the routing table (see [RoutingTable.register\\_dht\(\)](#))
- register this instance of the dht in the scheduler
- start the routing table if needed and `start_routing_table`` is ``True
- start the scheduler if needed and `start_scheduler` is True

**Parameters**

- **start\_routing\_table** (*bool*) – If True (the default) also start the routing table if needed
- **start\_scheduler** (*bool*) – If “True” (the default) also start the scheduler

**stop** ()

Stop the dht:

- Set the attribute `stopped` to True and wait for threads to terminate
- Close the dht socket

**Raises** [FailToStop](#) – if there is still some alive threads after 30 seconds, with the list of still alive threads as parameter.

**stop\_bg** ()

Launch the stop process of the dht and return immediately

**init\_socket** ()

Initialize the UDP socket of the DHT

**is\_alive** ()

Test if all threads of the dht are alive, stop the dht if one of the thread is dead

**Returns** True if all dht threads are alive, False otherwise and stop all remaining threads.

**Return type** bool

**debug** (*lvl, msg*)

Print msg prefixed with *prefix* if `lvl <= debuglvl`

**Parameters**

- **lvl** (*int*) – The debug level of the message to print
- **msg** (*str*) – The debug message to print

**Note:** duplicate messages are removed

**sleep** (*t*, *fstop*=None)

Sleep for *t* seconds. If the dht is requested to be stop, run *fstop*() and exit

**Parameters**

- *t* (*float*) – A time to sleep, in seconds
- *fstop* – A callable with no arguments, called before exiting

**Note:** Dont use it in the main thread otherwise it can exit before child threads. Only use it in child threads

) .. automethod:: build\_table .. automethod:: announce\_peer(info\_hash, port, delay=0, block=True) .. automethod:: get\_peers(hash, delay=0, block=True, callback=None, limit=10)

**get\_closest\_nodes** (*id*, *compact*=False)

return the current K closest nodes from *id* present in the routing table (K = 8)

**Parameters**

- *id* (*bytes*) – A 160bits (20 Bytes) long identifier for which we want the closest nodes in the routing table.
- *compact* (*bool*) – If True the nodes infos are returned in compact format. Otherwise, instances of *Node* are returned. The default is False.

**Returns** A list of *Node* if *compact* is False, a *bytes* of size multiple of 26 if *compact* is True.

**Return type** *list* if *compact* is False, a *bytes* otherwise.

**Note:** Contact information for peers is encoded as a 6-byte string. Also known as “Compact IP-address/port info” the 4-byte IP address is in network byte order with the 2 byte port in network byte order concatenated onto the end.

Contact information for nodes is encoded as a 26-byte string. Also known as “Compact node info” the 20-byte Node ID in network byte order and the compact IP-address/port info concatenated to the end.

**sendto** (*msg*, *addr*)

Program a msg to be send over the network

**Parameters**

- *msg* (*bytes*) – The message to send
- *addr* (*tuple*) – A couple (ip, port) to send the message to. ip is in dotted notation

**Notes:** The message is push to the *to\_send* queue.

**clean** ()

Function called every 15s to do some cleanning. It can safely be overload

**clean\_long** ()

Function called every 15min to do some cleanning. It can safely be overload

**register\_message** (*msg*)

Register a dht message to be processed by the following user defined functions

- *on\_error* ()
- *on\_ping\_query* ()

- `on_ping_response()`
- `on_find_node_query()`
- `on_find_node_response()`
- `on_get_peers_query()`
- `on_get_peers_response()`
- `on_announce_peer_query()`
- `on_announce_peer_response()`
- ...

**Parameters** `msg` (*bytes*) – A dht message to register like `b'error'`, `b'ping'`, `b'find_node'`, `b'get_peers'` or `b'announce_peer'`

**Note:**

- on query reception, the function `on_“msg”_query` will be call with the query as parameter
- on response reception, the function `on_“msg”_response` will be called with the query and the response as parameters
- on error reception, the function `on_error` will be called with the error and the query as parameter
- The message kind is in the `q` key of any dht query message

**`on_announce_peer_response`** (*query, response*)

Function called on a `announce_peer` response reception. Can safely the overloaded

**Parameters**

- **`query`** (`krcp.BMessage`) – the sent query object
- **`response`** (`krcp.BMessage`) – the received response object

**Notes:** For this function to be called on `announce_peer` response reception, you need to call `register_message()` with the parameter `b'announce_peer'`

**`on_announce_peer_query`** (*query*)

Function called on a `announce` query reception. Can safely the overloaded

**Parameters** **`query`** (`krcp.BMessage`) – the received query object

**Notes:** For this function to be called on `announce_peer` query reception, you need to call `register_message()` with the parameter `b'announce_peer'`

**`on_find_node_query`** (*query*)

Function called on a `find_node` query reception. Can safely the overloaded

**Parameters** **`query`** (`krcp.BMessage`) – the received query object

**Notes:** For this function to be called on `find_node` query reception, you need to call `register_message()` with the parameter `b'find_node'`

**`on_find_node_response`** (*query, response*)

Function called on a `find_node` response reception. Can safely the overloaded



**Parameters**

- **query** (`krcp.BMessage`) – the sent query object
- **response** (`krcp.BMessage`) – the received response object

**Notes:** For this function to be called on `find_node` response reception, you need to call `register_message()` with the parameter `b'find_node'`

**on\_get\_peers\_query** (*query*)

Function called on a `get_peers` query reception. Can safely the overloaded

**Parameters** **query** (`krcp.BMessage`) – the received query object

**Notes:** For this function to be called on `get_peers` query reception, you need to call `register_message()` with the parameter `b'get_peers'`

**on\_get\_peers\_response** (*query, response*)

Function called on a `get_peers` response reception. Can safely the overloaded

**Parameters**

- **query** (`krcp.BMessage`) – the sent query object
- **response** (`krcp.BMessage`) – the received response object

**Notes:** For this function to be called on `get_peers` response reception, you need to call `register_message()` with the parameter `b'get_peers'`

**on\_ping\_query** (*query*)

Function called on a `ping` query reception. Can safely the overloaded

**Parameters** **query** (`krcp.BMessage`) – the received query object

**Notes:** For this function to be called on `ping` query reception, you need to call `register_message()` with the parameter `b'ping'`

**on\_ping\_response** (*query, response*)

Function called on a `ping` response reception. Can safely the overloaded

**Parameters**

- **query** (`krcp.BMessage`) – the sent query object
- **response** (`krcp.BMessage`) – the received response object

**Notes:** For this function to be called on `ping` response reception, you need to call `register_message()` with the parameter `b'ping'`

**on\_error** (*error, query=None*)

Function called then a query has be responded by an error message. Can safely the overloaded.

**Parameters**

- **error** (`krcp.Berror`) – An error instance
- **query** (`krcp.BMessage`) – An optional query raising the error message

**Notes:** For this function to be called on error reception, you need to call `register_message()` with the parameter `b'error'`

**class** btdht.dht.Node

Bases: `object`

A node of the dht in the routing table

**Parameters**

- **id** (*bytes*) – The 160 bits (20 Bytes) long identifier of the node
- **ip** (*str*) – The ip, in dotted notation of the node
- **port** (*int*) – The udp dht port of the node
- **last\_response** (*int*) – Unix timestamp of the last received response from this node
- **last\_query** (*int*) – Unix timestamp of the last received query from this node
- **failed** (*int*) – Number of consecutive queries sended to the node without responses

**Note:** A good node is a node has responded to one of our queries within the last 15 minutes. A node is also good if it has ever responded to one of our queries and has sent us a query within the last 15 minutes. After 15 minutes of inactivity, a node becomes questionable. Nodes become bad when they fail to respond to multiple queries in a row (3 query in a row in this implementation).

**port**

UDP port of the node

**last\_response**

Unix timestamp of the last received response from this node

**last\_query**

Unix timestamp of the last received query from this node

**failed**

Number of reponse pending (increase on sending query to the node, set to 0 on reception from the node)

**id**

160bits (20 Bytes) identifier of the node

**good**

True if the node is a good node. A good node is a node has responded to one of our queries within the last 15 minutes. A node is also good if it has ever responded to one of our queries and has sent us a query within the last 15 minutes.

**bad**

True if the node is a bad node (communication with the node is not possible). Nodes become bad when they fail to respond to 3 queries in a row.

**ip**

IP address of the node in dotted notation

**compact\_info** ()

Return the compact contact information of the node

**Notes:** Contact information for peers is encoded as a 6-byte string. Also known as “Compact IP-address/port info” the 4-byte IP address is in network byte order with the 2 byte port in network byte order concatenated onto the end. Contact information for nodes is encoded as a 26-byte string. Also known as “Compact node info” the 20-byte Node ID in network byte order has the compact IP-address/port info concatenated to the end.

**from\_compact\_infos** (*infos*)

This is a classmethod

Instancy nodes from multiple compact node information string

**Parameters** `infos` (*bytes*) – A string of size multiple of 26

**Returns** A list of *Node* instances

**Return type** *list*

**Notes:** Contact information for peers is encoded as a 6-byte string. Also known as “Compact IP-address/port info” the 4-byte IP address is in network byte order with the 2 byte port in network byte order concatenated onto the end. Contact information for nodes is encoded as a 26-byte string. Also known as “Compact node info” the 20-byte Node ID in network byte order has the compact IP-address/port info concatenated to the end.

**from\_compact\_info** (*info*)

This is a classmethod

Instancy a node from its compact node information string

**Parameters** `info` (*bytes*) – A string of length 26

**Returns** A node instance

**Return type** *Node*

**Notes:** Contact information for peers is encoded as a 6-byte string. Also known as “Compact IP-address/port info” the 4-byte IP address is in network byte order with the 2 byte port in network byte order concatenated onto the end. Contact information for nodes is encoded as a 26-byte string. Also known as “Compact node info” the 20-byte Node ID in network byte order has the compact IP-address/port info concatenated to the end.

**announce\_peer** (*dht, info\_hash, port*)

Send a announce\_peer query to the node

**Parameters**

- `dht` (*DHT\_BASE*) – The dht instance to use to send the message
- `info_hash` (*bytes*) – A 160bits (20 bytes) torrent id to announce
- `port` (*int*) – The tcp port where data for `info_hash` is available

**Raises** *NoTokenError* – if we have no valid token for `info_hash`. Try to call `get_peers()` on this `info_hash` first.

**find\_node** (*dht, target*)

Send a find\_node query to the node

**Parameters**

- `dht` (*DHT\_BASE*) – The dht instance to use to send the message
- `target` (*bytes*) – the 160bits (20 bytes) target node id

**get\_peers** (*dht, info\_hash*)

Send a get\_peers query to the node

**Parameters**

- `dht` (*DHT\_BASE*) – The dht instance to use to send the message
- `info_hash` (*bytes*) – a 160bits (20 bytes) torrent id

**ping** (*dht*)

Send a ping query to the node

**Parameters** **dht** (`DHT_BASE`) – The dht instance to use to send the message

**class** `btdht.dht.Bucket`

Bases: `list`

A bucket of nodes in the routing table

**Parameters**

- **id** (*bytes*) – A prefix identifier from 0 to 169 bits for the bucket
- **id\_length** (*int*) – number of significant bit in *id* (can also be seen as the length between the root and the bucket in the routing table)
- **init** (*iterable*) – some values to store initially in the bucket

**max\_size** = 8

Maximun number of element in the bucket

**last\_changed** = 0

Unix timestamp, last time the bucket had been updated

**id** = None

A prefix identifier from 0 to 160 bits for the bucket

**id\_length** = 0

Number of signifiant bit in *id*

**to\_refresh**

True if the bucket need refreshing

**random\_id** ()

**Returns** A random id handle by the bucket

**Return type** `bytes`

This is used to send `find_nodes` for randoms ids in a bucket

**add** (*dht*, *node*)

Try to add a node to the bucket.

**Parameters**

- **dht** (`DHT_BASE`) – The dht instance the node to add is from
- **node** (`Node`) – A node to add to the bucket

**Raises** `BucketFull` – if the bucket is full

**Notes:** The addition of a node to a bucket is done as follow: \* if the bucket is not full, just add the node \* if the bucket is full

- if there is some bad nodes in the bucket, remove a bad node and add the node
- if there is some questionnable nodes (neither good not bad), send a ping request to the oldest one, discard the node
- if all nodes are good in the bucket, discard the node

**get\_node** (*id*)

**Returns** A `Node` with `Node.id` equal to *id*

**Return type** *Node*

**Raises** *NotFound* – if no node is found within this bucket

**own** (*id*)

**Parameters** *id* (*bytes*) – A 60bit (20 Bytes) identifier

**Returns** *True* if *id* is handled by this bucket

**Return type** *bool*

**split** (*rt*, *dht*)

Split the bucket into two buckets

**Parameters**

- *rt* (*RoutingTable*) – The routing table handling the bucket
- *dht* (*DHT\_BASE*) – A dht using *rt* as routing table

**Returns** A couple of two bucket, the first one this the last significant bit of its id equal to 0, the second, equal to 1

**Return type** *tuple*

**Raises** *BucketNotFull* – If the bucket has not *max\_size* elements (and so the split is not needed)

**merge** (*bucket*)

Merge the bucket with *bucket*

**Parameters** *bucket* (*Bucket*) – a bucket to be merged with

**Returns** The merged bucket

**Return type** *Bucket*

**class** `btdht.dht.RoutingTable`

Bases: *object*

A routing table for one or more *DHT\_BASE* instances

**Parameters**

- *scheduler* (*utils.Scheduler*) – A scheduler instance
- *debuglvl* (*int*) – Level of verbosity, default to 0.

**debuglvl** = 0

*int* the routing table instance verbosity level

**trie** = *None*

The routing table storage data structure, an instance of *datrie.Trie*

**stoped** = *True*

The state (stoped ?) of the routing table

**need\_merge** = *False*

Is a merge sheduled ?

**threads** = []

*list* of the *Thread* of the routing table instance

**to\_schedule** = []

A class:*list* of couple (weightless thread name, weightless thread function)

**prefix** = ''

Prefix in logs and threads name

**zombie**

True if dht is stopped but one thread or more remains alive, False otherwise

**start** ()

start the routing table

**stop** ()

stop the routing table and wait for all threads to terminate

**stop\_bg** ()

stop the routing table and return immediately

**is\_alive** ()

Test if all routing table threads are alive. If a thread is found dead, stop the routingtable

**Returns** True if all routing table threads are alive, False otherwise

**Return type** bool

**register\_torrent** (id)

Register a torrent id (info\_hash) for being tracked by the routing table. This means that if a node need to be added to the bucket handling "id" and the bucket is full, then, this bucket will be split into 2 buckets

**Parameters** id (bytes) – A 160 bits (20 Bytes) torrent identifier

**Note:** torrent ids can automatically be release by a dht instance after a get\_peers. For keeping a torrent registered, use the method `register_torrent_longterm()`

**release\_torrent** (id)

Release a torrent id (info\_hash) and program the routing table to be merged

**Parameters** id (bytes) – A 160 bits (20 Bytes) torrent identifier

**register\_torrent\_longterm** (id)

Same as `register_torrent()` but garanty that the torrent wont be released automatically by the dht.

**Parameters** id (bytes) – A 160 bits (20 Bytes) torrent identifier

**release\_torrent\_longterm** (id)

For releasing torrent registered with the :meth'register\_torrent\_longterm' method

**Parameters** id (bytes) – A 160 bits (20 Bytes) torrent identifier

**register\_dht** (dht)

Register a dht instance to the routing table

**Parameters** dht (DHT\_BASE) – A dht instance

**Notes:** on start, all dht instances automatically register themself to their routing tables

**release\_dht** (dht)

Release a dht instance to the routing table, and shedule the routing table for a merge.

**Notes:** on stop, dht automatially release itself from the routing table

**empty** ()

Empty the routing table, deleting all buckets

**debug** (lvl, msg)

same as `DHT_BASE.debug()`

**stats()**

**Returns** A triple (number of nodes, number of good nodes, number of bad nodes)

**Return type** *tuple*

**height()**

**Returns** the height of the tree of the routing table

**Return type** *int*

**find(id)**

**Parameters** *id* (*bytes*) – A 160 bits (20 Bytes) identifier

**Returns** The bucket handling *id*

**Return type** *Bucket*

**Raises** *KeyError* – then a racing condition with merging and/or splitting a bucket is met. This should not happen

**Notes:** During a split or merge of bucket it is possible that the bucket handling *id* is not found. *find()* will retry at most 20 times to get the bucket. In most case, during those retries, the split and/or merge will end and the bucket handling *id* will be returned.

**get\_node(id)**

**Parameters** *id* (*bytes*) – A 160 bits (20 Bytes) identifier

**Returns** A node with *id id*

**Return type** *Node*

**Raises** *NotFound* – if no nodes is found

**get\_closest\_nodes(id, bad=False)**

Return the K closest nodes from *id* in the routing table

**Parameters**

- *id* (*bytes*) – A 160 bits (20 Bytes) identifier
- *bad* (*bool*) – Should we return bad nodes ? The default is *False*

**Notes:** If less than K (=8) good nodes is found, bad nodes will be included it solve the case there the connection where temporary lost and all nodes in the routing table marked as bad. In normal operation, we should always find K (=8) good nodes in the routing table.

**add(dht, node)**

Add a node the the routing table

**Parameters**

- *dht* (*DHT\_BASE*) – The dht instance “node” is from
- *node* (*Node*) – The node to add to the routing table

**split(dht, bucket)**

Split bucket in two

**Parameters**

- *dht* (*DHT\_BASE*) – A dht instance

- **bucket** (`Bucket`) – A bucket from the routing table to split

**Notes:** the routing table cover the entire 160bits space

**merge()**

Request a merge to be perform

## 2.1.2 btdht.utils module

<code>bencode</code>	bencode an arbitrary object
<code>bdecode</code>	bdecode an bytes string
<code>bdecode_rest</code>	bdecode an bytes string
<code>enumerate_ids</code>	Enumerate 2 to the power of <code>size</code> ids from <code>id</code>
<code>id_to_longid</code>	convert a random bytes to a unicode string of 1 and 0
<code>ip_in_nets</code>	Test if <code>ip</code> is in one of the networks of <code>nets</code>
<code>nbit</code>	Allow to retrieve the value of the <code>nth</code> bit of <code>s</code>
<code>nflip</code>	Allow to flip the <code>nth</code> bit of <code>s</code>
<code>nset</code>	Allow to set the value of the <code>nth</code> bit of <code>s</code>
<code>ID</code>	A 160 bit (20 Bytes) string implementing the XOR distance
<code>PollableQueue</code>	A queue that can be watch using <code>select.select()</code> .
<code>Scheduler</code>	Schedule weightless threads and DHTs io

`btdht.utils.bencode(obj)`

bencode an arbitrary object

**Parameters** `obj` – A combination of dict, list, bytes or int

**Returns** Its bencoded representation

**Return type** `bytes`

**Notes:** This method is just a wrapper around `_bencode()`

`btdht.utils.bdecode(s)`

bdecode an bytes string

**Parameters** `s` – A bencoded bytes string

**Returns** Its bencoded representation

**Return type** A combination of `dict`, `list`, `bytes` or `int`

**Raises** `BcodeError` – If failing to decode `s`

**Notes:** This method is just a wrapper around `_bdecode()`

`btdht.utils.bdecode_rest(s)`

bdecode an bytes string

**Parameters** `s` – A bencoded bytes string

**Returns** A couple: (bdecoded representation, rest of the string). If only one bencoded object is given as argument, then the ‘rest of the string’ will be empty

**Return type** `tuple` ( combination of `dict`, `list`, `bytes` or `int`, bytes)

**Raises** `BcodeError` – If failing to decode `s`



`btdht.utils.enumerate_ids(size, id)`

Enumerate 2 to the power of `size` ids from `id`

#### Parameters

- **size** (*int*) – A number of bit to flip in `id`
- **id** (*bytes*) – A 160 bit (20 Bytes) long `id`

**Returns** A list of `id` and 2 to the power of `size` (minus one) ids the furthest from each other

**Return type** `list`

For instance: if `id=("\0" * 20) (~0 * 160)`, `enumerate_ids(4, id)` will return a list with

- `'\x00\x00\x00\x00\x00...' (~00000000...)`
- `'\x80\x00\x00\x00\x00...' (~10000000...)`
- `'@\x00\x00\x00\x00...' (~0100000000...)`
- `'\xc0\x00\x00\x00\x00...' (~11000000...)`

The can be see as the tree:



The root is `id`, at each level `n`, we set the `n`th bit to 1 left and 0 right, `size` if the level we return.

This function may be useful to launch multiple DHT instance with ids the most distributed on the 160 bit space.

`btdht.utils.id_to_longid(id, l=20)`

convert a random bytes to a unicode string of 1 and 0

For instance: `"\0" -> "000000000"`

#### Parameters

- **id** (*bytes*) – A random string
- **size** (*int*) – The length of `id`

**Returns** The corresponding base 2 unicode string

**Return type** `unicode`

`btdht.utils.ip_in_nets(ip, nets)`

Test if `ip` is in one of the networks of `nets`

#### Parameters

- **ip** (*str*) – An ip, in dotted notation
- **nets** (*list*) – A list of `netaddr.IPNetwork`

**Returns** `True` if `ip` is in one of the listed networks, `False` otherwise

**Return type** `bool`

`btdht.utils.nbit(s, n)`

Allow to retrieve the value of the `n`th bit of `s`

**Parameters**

- **s** (*bytes*) – A byte string
- **n** (*int*) – A bit number (n must be smaller than 8 times the length of s)

**Returns** The value of the nth bit of s (0 or 1)

**Return type** *int*

`btdht.utils.nflip(s, n)`

Allow to flip the nth bit of s

**Parameters**

- **s** (*bytes*) – A byte string
- **n** (*int*) – A bit number (n must be smaller than 8 times the length of s)

**Returns** The same string except for the nth bit was flip

**Return type** *bytes*

`btdht.utils.nset(s, n, i)`

Allow to set the value of the nth bit of s

**Parameters**

- **s** (*bytes*) – A byte string
- **n** (*int*) – A bit number (n must be smaller than 8 times the length of s)
- **i** (*int*) – A bit value (0 or 1)

**Returns** s where the nth bit was set to i

**Return type** *bytes*

**class** `btdht.utils.ID`

Bases: *object*

A 160 bit (20 Bytes) string implementing the XOR distance

**Parameters** **id** – An optional initial value (*bytes* or *ID*). If not specified, a random 160 bit value is generated.

**value = None**

*bytes*, Actual value of the *ID*

**classmethod** `to_bytes(id)`

**Parameters** **id** – A *bytes* or *ID*

**Returns** The value of the id

**Return type** *bytes*

**startswith** (*s*)

S.startswith(prefix[, start[, end]]) -> bool

Return True if S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.

**\_\_getitem\_\_** (*i*)

x.\_\_getitem\_\_(y) <==> x[y]

`__xor__ (other)`

Perform a XOR bit by bit between the current id and `other`

**Parameters** `other` – A `bytes` or `ID`

**Returns** The resulted XORed bit by bit string

**Return type** `bytes`

**class** `btdht.utils.PollableQueue`

Bases: `Queue.Queue`

A queue that can be watch using `select.select()`

**Parameters** `maxsize (int)` – The maximum size on the queue. If `maxsize` is  $\leq 0$ , the queue size is infinite.

**sock = None**

A `socket.socket` object ready for read then here is something to pull from the queue

**empty()**

Return True if the queue is empty, False otherwise (not reliable!).

**full()**

Return True if the queue is full, False otherwise (not reliable!).

**get (block=True, timeout=None)**

Remove and return an item from the queue.

If optional args 'block' is true and 'timeout' is None (the default), block if necessary until an item is available. If 'timeout' is a non-negative number, it blocks at most 'timeout' seconds and raises the Empty exception if no item was available within that time. Otherwise ('block' is false), return an item if one is immediately available, else raise the Empty exception ('timeout' is ignored in that case).

**get\_nowait()**

Remove and return an item from the queue without blocking.

Only get an item if one is immediately available. Otherwise raise the Empty exception.

**join()**

Blocks until all items in the Queue have been gotten and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer thread calls `task_done()` to indicate the item was retrieved and all work on it is complete.

When the count of unfinished tasks drops to zero, `join()` unblocks.

**put (item, block=True, timeout=None)**

Put an item into the queue.

If optional args 'block' is true and 'timeout' is None (the default), block if necessary until a free slot is available. If 'timeout' is a non-negative number, it blocks at most 'timeout' seconds and raises the Full exception if no free slot was available within that time. Otherwise ('block' is false), put an item on the queue if a free slot is immediately available, else raise the Full exception ('timeout' is ignored in that case).

**put\_nowait (item)**

Put an item into the queue without blocking.

Only enqueue the item if a free slot is immediately available. Otherwise raise the Full exception.

**qsize()**

Return the approximate size of the queue (not reliable!).

**task\_done()**

Indicate that a formerly enqueued task is complete.

Used by Queue consumer threads. For each get() used to fetch a task, a subsequent call to task\_done() tells the queue that the processing on the task is complete.

If a join() is currently blocking, it will resume when all items have been processed (meaning that a task\_done() call was received for every item that had been put() into the queue).

Raises a ValueError if called more times than there were items placed in the queue.

**class** btdht.utils.Scheduler

Bases: `object`

Schedule weightless threads and DHTs io

A weightless threads is a python callable returning an iterator that behave as describe next. The first returned value must be an integer describing the type of the iterator. 0 means time based and all subsequent yield must return the next timestamp at which the iterator want to be called. 1 means queue based. The next call to the iterator must return an instance of `PollableQueue`. All subsequent yield value are then ignored. The queue based iterator will be called when something is put on its queue.

**zombie**

**Returns** `True` if the scheduler is stoped but its threads are still running

**Return type** `bool`

**start** (*name\_prefix="scheduler"*)

start the scheduler

**Parameters** **name\_prefix** (*str*) – Prefix to the scheduler threads names

**stop()**

stop the scheduler

**Raises** `FailToStop` – if we fail to stop one of the scheduler threads after 30 seconds

**stop\_bg()**

Lauch the stop process of the dht and return immediately

**is\_alive()**

Test if the scheduler main thread is alive

**Returns** `True` the scheduler main thread is alive, `False` otherwise

**Return type** `bool`

**thread\_alive** (*name*)

Test is a weightless threads named *name* is currently schedule

**Parameters** **name** (*str*) – The name of a thread

**Returns** `True` if a thread of name *name* if found

**Return type** `bool`

**add\_dht** (*dht*)

Add a dht instance to be schedule by the scheduler

**Parameters** **dht** (`dht.DHT_BASE`) – A dht instance

**del\_dht** (*dht*)

Remove a dht instance from the scheduler

**Parameters** **dht** (`dht.DHT_BASE`) – A dht instance

**add\_thread** (*name*, *function*, *user=False*)

Schedule the call of weightless threads

#### Parameters

- **name** (*str*) – The name of the thread to add. Must be unique in the *Scheduler* instance
- **function** – A weightless threads, i.e a callable returning an iterator
- **user** (*bool*) – If *True* the weightless threads is schedule in a secondary thread. The default is *False* and the weightless threads is processed in the main scheduler thread. This is usefull to put controled weightless threads and the main thread, and all the other (like the user defined on\_“msg“\_(query/response)) function to the secondary one.

**del\_thread** (*name*, *stop\_if\_empty=True*)

Remove the weightless threads named *name*

#### Parameters

- **name** (*str*) – The name of a thread
- **stop\_if\_empty** (*bool*) – If *True* (the default) and the scheduler has nothing to schedules, the scheduler will be stopped.

## 2.1.3 btdht.krcp module

**class** btdht.krcp.**BError**

Bases: *exceptions.Exception*

A base class exception for all bittorrent DHT protocol error exceptions

#### Parameters

- **t** (*bytes*) – The value of the key *t* of the query for with the error is returned
- **e** (*list*) – A couple [error code, error message]

**e** = *None*

A list. The first element is an *int* representing the error code. The second element is a string containing the error message

**t** = *None*

string value representing a transaction ID, must be set to the query transaction ID for which an error is raises.

**y** = *'e'*

The *y* key of the error message. For an error message, it is always *b"e"*

**encode** ()

Bencode the error message

**Returns** The bencoded error message ready to be send

**Return type** *bytes*

**class** btdht.krcp.**GenericError**

Bases: *btdht.krcp.BError*

A Generic Error, error code 201

#### Parameters

- **t** (*bytes*) – The value of the key *t* of the query for with the error is returned
- **msg** (*bytes*) – An optionnal error message

**class** btdht.krcp.MethodUnknownError

Bases: *btdht.krcp.BError*

Method Unknown, error code 204

**Parameters**

- **t** (*bytes*) – The value of the key **t** of the query for with the error is returned
- **msg** (*bytes*) – An optionnal error message

**class** btdht.krcp.ProtocolError

Bases: *btdht.krcp.BError*

A Protocol Error, such as a malformed packet, invalid arguments, or bad token, error code 203

**Parameters**

- **t** (*bytes*) – The value of the key **t** of the query for with the error is returned
- **msg** (*bytes*) – An optionnal error message

**class** btdht.krcp.ServerError

Bases: *btdht.krcp.BError*

A Server Error, error code 202

**Parameters**

- **t** (*bytes*) – The value of the key **t** of the query for with the error is returned
- **msg** (*bytes*) – An optionnal error message

**class** btdht.krcp.BMessage

Bases: *object*

A bittorrent DHT message. This class is able to bdecode a bittorrent DHT message. It expose then the messages keys **t**, **y**, **q**, **errno**, **errmsg** and **v** as attributes, and behave itself as a dictionary for the **a** or **r** keys that contains a secondary dictionary (see Notes).

**Parameters**

- **addr** (*tuple*) – An optionnal couple (ip, port) of the sender of the message
- **debug** (*bool*) – True for enabling debug message. The default is `False`

**Notes:** A query message is always of the following form with **y** == `b'q'`:

```
{
    "t": t,
    "y": y,
    "q": q,
    "a": { ... }
}
```

A response message is always of the following form with **y** == `b'r'`:

```
{
    "t": t,
    "y": y,
    "r": { ... }
}
```

An error message is always in response of a query message and of the following form with **y** == `b'e'`:

```
{
    "t": t,
    "y": y,
    "e": [errno, errmsg]
}
```

The `t` key is a random string generated with every query. It is used to match a response to a particular query.

The `y` key is used to differentiate the type of the message. Its value is `b'q'` for a query, `b'r'` for a response, and `b'e'` for an error message.

The `q` is only present on query message and contains the name of the query (`ping`, `get_peers`, `announce_peer`, `find_node`)

`errno` and `errmsg` are only defined if the message is an error message. They are respectively the error number (`int`) and the error describing message of the error.

The `v` key is set by some DHT clients to the name and version of the client and is totally optional in the protocol.

#### **addr**

The couple (ip, port) source of the message

#### **errmsg**

The error message of the message if the message is an error message

#### **errno**

The error number of the message if the message is an error message

#### **q**

The `q` key of the message, should only be defined if the message is a query (`y` is `"q"`). It contains the name of the RPC method the query is asking for. Can be `b'ping'`, `b'find_node'`, `b'get_peers'`, `b'announce_peer'`, ...

#### **t**

The `t` key, a random string, transaction id used to match queries and responses together.

#### **v**

The `v` key of the message. This attribute is not described in the BEP5 that describes the bittorrent DHT protocol. It is used as a version flag. Many bittorrent clients set it to the name and version of the client.

#### **y**

The `y` key of the message. Possible values are `"q"` for a query, `"r"` for a response and `"e"` for an error.

#### **\_\_getitem\_\_(key)**

Allow to fetch infos from the secondary dictionary:

```
self[b"id"] -> b"..."
```

**Parameters** `key` (*bytes*) – The name of an attribute of the secondary dictionary to retrieve.

**Returns** The value stored for `key` if found

**Raises** `KeyError` – if `key` is not found

#### **Notes:**

Possible keys are:

- id
- target
- info\_hash
- token
- nodes
- implied\_port
- port
- values

`__delitem__` (*key*)

Allow to unset attributes from the secondary dictionary:

```
del self[b'id']
```

:param *key*: The name of an attribute of the secondary dictionary to unset :return: True if *key* is found and successfully unset :raise `KeyError`: if *key* is not found

`__setitem__` (*key*, *value*)

Allow to set attributes from the secondary dictionary:

```
self[b'id'] = b"..."
```

#### Parameters

- **key** (*bytes*) – The name of an attribute of the secondary dictionary to set
- **value** – The value to set

#### Raises

- **KeyError** – if *key* is not one of id, target, info\_hash, token, nodes, implied\_port, port, values.
- **ValueError** – if *value* is not well formatted (length, type, ...)

`decode` (*data*, *datalen*)

Bdecode a bencoded message and set the current *BMessage* attributes accordingly

#### Parameters

- **data** (*bytes*) – The bencoded message
- **datalen** (*int*) – The length of data

**Returns** The remaining of *data* after the first bencoded message of *data* has been bdecoded (it may be the empty string if *data* contains exactly one bencoded message with no garbage at the end).

#### Raises

- **DecodeError** – If we fail to decode the message
- **ProtocolError** – If the message is decoded but some attributes are missing or badly formatted (length, type, ...).
- **MissingT** – If the message does not have a `b"t"` key. Indeed, according to the BEP5, every message (queries, responses, errors) should have a `b"t"` key.



**encode()**

Bencodes the current message if necessary

**Returns** The bencoded message

**Return type** `bytes`

**get** (*key*, *default=None*)

**Parameters**

- **key** (*bytes*) – The name of an attribute of the secondary dictionary to retrieve.
- **default** – Value to return in case *key* is not found. The default is `None`

**Returns** The value of *key* if found, else the value of *default*.

**response** (*dht*)

If the message is a query, return the response message to send

**Parameters** *dht* (`dht.DHT_BASE`) – The *dht* instance from which the message is originated

**Returns** A *BMessage* to send as response to the query

**Raises**

- **ProtocolError** – if the query is malformed. To send as response to the querier
- **MethodUnknownError** – If the RPC DHT method asked in the query is unknown. To send as response to the querier

## 2.1.4 btdht.exceptions module

**exception** `btdht.exceptions.BucketFull`

Bases: `exceptions.Exception`

Raised then trying to add a node to a *Bucket* that already contains *Bucket.max\_size* elements.

**exception** `btdht.exceptions.BucketNotFull`

Bases: `exceptions.Exception`

Raises then trying to split a *Bucket* that contains less than *Bucket.max\_size* elements.

**exception** `btdht.exceptions.NoTokenError`

Bases: `exceptions.Exception`

Raised then trying to announce to a node we download an *info\_hash* using *Node.announce\_peer* but we do not know any valid token. The error should always be catch and never seen by btdht users.

**exception** `btdht.exceptions.FailToStop`

Bases: `exceptions.Exception`

Raises then we are trying to stop threads but failing at it

**exception** `btdht.exceptions.TransactionIdUnknown`

Bases: `exceptions.Exception`

Raised then receiving a response with an unknown *t* key

**exception** `btdht.exceptions.MissingT`

Bases: `exceptions.ValueError`

Raised while decoding of a *dht* message if that message of no key *t*

**exception** `btdht.exceptions.DecodeError`

Bases: `exceptions.ValueError`

Raised while decoding a dht message

**exception** `btdht.exceptions.BcodeError`

Bases: `exceptions.Exception`

Raised by `btdht.utils.bdecode()` and `btdht.utils.bencode()` functions

**exception** `btdht.exceptions.NotFound`

Bases: `exceptions.Exception`

Raised when trying to get a node that do not exists from a *Bucket*

## 2.2 Module contents

## CHAPTER 3

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`



### **b**

- `btdht`, [30](#)
- `btdht.dht`, [7](#)
- `btdht.exceptions`, [29](#)
- `btdht.krcp`, [25](#)
- `btdht.utils`, [20](#)



## Symbols

`__delitem__()` (btdht.krcp.BMessage method), 28  
`__getitem__()` (btdht.krcp.BMessage method), 27  
`__getitem__()` (btdht.utils.ID method), 22  
`__setitem__()` (btdht.krcp.BMessage method), 28  
`__xor__()` (btdht.utils.ID method), 22

## A

`add()` (btdht.dht.Bucket method), 16  
`add()` (btdht.dht.RoutingTable method), 19  
`add_dht()` (btdht.utils.Scheduler method), 24  
`add_thread()` (btdht.utils.Scheduler method), 24  
`addr` (btdht.krcp.BMessage attribute), 27  
`announce_peer()` (btdht.dht.Node method), 15

## B

`bad` (btdht.dht.Node attribute), 14  
`BcodeError`, 30  
`bdecode()` (in module btdht.utils), 20  
`bdecode_rest()` (in module btdht.utils), 20  
`bencode()` (in module btdht.utils), 20  
`BError` (class in btdht.krcp), 25  
`bind_ip` (btdht.dht.DHT\_BASE attribute), 8  
`bind_port` (btdht.dht.DHT\_BASE attribute), 8  
`BMessage` (class in btdht.krcp), 26  
`btdht` (module), 30  
`btdht.dht` (module), 7  
`btdht.exceptions` (module), 29  
`btdht.krcp` (module), 25  
`btdht.utils` (module), 20  
`Bucket` (class in btdht.dht), 16  
`BucketFull`, 29  
`BucketNotFull`, 29

## C

`clean()` (btdht.dht.DHT\_BASE method), 11  
`clean_long()` (btdht.dht.DHT\_BASE method), 11  
`compact_info()` (btdht.dht.Node method), 14

## D

`debug()` (btdht.dht.DHT\_BASE method), 10  
`debug()` (btdht.dht.RoutingTable method), 18  
`debuglvl` (btdht.dht.DHT\_BASE attribute), 8  
`debuglvl` (btdht.dht.RoutingTable attribute), 17  
`decode()` (btdht.krcp.BMessage method), 28  
`DecodeError`, 29  
`del_dht()` (btdht.utils.Scheduler method), 24  
`del_thread()` (btdht.utils.Scheduler method), 25  
`DHT` (class in btdht.dht), 7  
`DHT_BASE` (class in btdht.dht), 8

## E

`e` (btdht.krcp.BError attribute), 25  
`empty()` (btdht.dht.RoutingTable method), 18  
`empty()` (btdht.utils.PollableQueue method), 23  
`encode()` (btdht.krcp.BError method), 25  
`encode()` (btdht.krcp.BMessage method), 28  
`enumerate_ids()` (in module btdht.utils), 20  
`errmsg` (btdht.krcp.BMessage attribute), 27  
`errno` (btdht.krcp.BMessage attribute), 27

## F

`failed` (btdht.dht.Node attribute), 14  
`FailToStop`, 29  
`find()` (btdht.dht.RoutingTable method), 19  
`find_node()` (btdht.dht.Node method), 15  
`from_compact_info()` (btdht.dht.Node method), 15  
`from_compact_infos()` (btdht.dht.Node method), 14  
`full()` (btdht.utils.PollableQueue method), 23

## G

`GenericError` (class in btdht.krcp), 25  
`get()` (btdht.krcp.BMessage method), 29  
`get()` (btdht.utils.PollableQueue method), 23  
`get_closest_nodes()` (btdht.dht.DHT\_BASE method), 11  
`get_closest_nodes()` (btdht.dht.RoutingTable method), 19  
`get_node()` (btdht.dht.Bucket method), 16  
`get_node()` (btdht.dht.RoutingTable method), 19

`get_nowait()` (btdht.utils.PollableQueue method), 23  
`get_peers()` (btdht.dht.Node method), 15  
`good` (btdht.dht.Node attribute), 14

## H

`heigh()` (btdht.dht.RoutingTable method), 19

## I

`id` (btdht.dht.Bucket attribute), 16  
`id` (btdht.dht.Node attribute), 14  
`ID` (class in btdht.utils), 22  
`id_length` (btdht.dht.Bucket attribute), 16  
`id_to_longid()` (in module btdht.utils), 21  
`ignored_ip` (btdht.dht.DHT\_BASE attribute), 9  
`ignored_net` (btdht.dht.DHT\_BASE attribute), 9  
`init_socket()` (btdht.dht.DHT\_BASE method), 10  
`ip` (btdht.dht.Node attribute), 14  
`ip_in_nets()` (in module btdht.utils), 21  
`is_alive()` (btdht.dht.DHT\_BASE method), 10  
`is_alive()` (btdht.dht.RoutingTable method), 18  
`is_alive()` (btdht.utils.Scheduler method), 24

## J

`join()` (btdht.utils.PollableQueue method), 23

## L

`last_changed` (btdht.dht.Bucket attribute), 16  
`last_msg` (btdht.dht.DHT\_BASE attribute), 8  
`last_msg_rep` (btdht.dht.DHT\_BASE attribute), 8  
`last_query` (btdht.dht.Node attribute), 14  
`last_response` (btdht.dht.Node attribute), 14  
`load()` (btdht.dht.DHT\_BASE method), 9

## M

`max_size` (btdht.dht.Bucket attribute), 16  
`merge()` (btdht.dht.Bucket method), 17  
`merge()` (btdht.dht.RoutingTable method), 20  
`MethodUnknownError` (class in btdht.krcp), 26  
`MissingT`, 29  
`myid` (btdht.dht.DHT\_BASE attribute), 9  
`mytoken` (btdht.dht.DHT\_BASE attribute), 9

## N

`nbit()` (in module btdht.utils), 21  
`need_merge` (btdht.dht.RoutingTable attribute), 17  
`nflip()` (in module btdht.utils), 22  
`Node` (class in btdht.dht), 13  
`NotFound`, 30  
`NoTokenError`, 29  
`nset()` (in module btdht.utils), 22

## O

`on_announce_peer_query()` (btdht.dht.DHT\_BASE method), 12

`on_announce_peer_response()` (btdht.dht.DHT\_BASE method), 12  
`on_error()` (btdht.dht.DHT\_BASE method), 13  
`on_find_node_query()` (btdht.dht.DHT\_BASE method), 12  
`on_find_node_response()` (btdht.dht.DHT\_BASE method), 12  
`on_get_peers_query()` (btdht.dht.DHT\_BASE method), 13  
`on_get_peers_response()` (btdht.dht.DHT\_BASE method), 13  
`on_ping_query()` (btdht.dht.DHT\_BASE method), 13  
`on_ping_response()` (btdht.dht.DHT\_BASE method), 13  
`own()` (btdht.dht.Bucket method), 17

## P

`ping()` (btdht.dht.Node method), 15  
`PollableQueue` (class in btdht.utils), 23  
`port` (btdht.dht.Node attribute), 14  
`prefix` (btdht.dht.DHT\_BASE attribute), 9  
`prefix` (btdht.dht.RoutingTable attribute), 17  
`ProtocolError` (class in btdht.krcp), 26  
`put()` (btdht.utils.PollableQueue method), 23  
`put_nowait()` (btdht.utils.PollableQueue method), 23

## Q

`q` (btdht.krcp.BMessage attribute), 27  
`qsize()` (btdht.utils.PollableQueue method), 23

## R

`random_id()` (btdht.dht.Bucket method), 16  
`register_dht()` (btdht.dht.RoutingTable method), 18  
`register_message()` (btdht.dht.DHT\_BASE method), 11  
`register_torrent()` (btdht.dht.RoutingTable method), 18  
`register_torrent_longterm()` (btdht.dht.RoutingTable method), 18  
`release_dht()` (btdht.dht.RoutingTable method), 18  
`release_torrent()` (btdht.dht.RoutingTable method), 18  
`release_torrent_longterm()` (btdht.dht.RoutingTable method), 18  
`response()` (btdht.krcp.BMessage method), 29  
`root` (btdht.dht.DHT\_BASE attribute), 9  
`RoutingTable` (class in btdht.dht), 17

## S

`save()` (btdht.dht.DHT\_BASE method), 9  
`Scheduler` (class in btdht.utils), 24  
`sendto()` (btdht.dht.DHT\_BASE method), 11  
`ServerError` (class in btdht.krcp), 26  
`sleep()` (btdht.dht.DHT\_BASE method), 10  
`sock` (btdht.dht.DHT\_BASE attribute), 9  
`sock` (btdht.utils.PollableQueue attribute), 23  
`split()` (btdht.dht.Bucket method), 17



[split\(\)](#) (btdht.dht.RoutingTable method), 19  
[start\(\)](#) (btdht.dht.DHT\_BASE method), 10  
[start\(\)](#) (btdht.dht.RoutingTable method), 18  
[start\(\)](#) (btdht.utils.Scheduler method), 24  
[startswith\(\)](#) (btdht.utils.ID method), 22  
[stats\(\)](#) (btdht.dht.RoutingTable method), 18  
[stop\(\)](#) (btdht.dht.DHT\_BASE method), 10  
[stop\(\)](#) (btdht.dht.RoutingTable method), 18  
[stop\(\)](#) (btdht.utils.Scheduler method), 24  
[stop\\_bg\(\)](#) (btdht.dht.DHT\_BASE method), 10  
[stop\\_bg\(\)](#) (btdht.dht.RoutingTable method), 18  
[stop\\_bg\(\)](#) (btdht.utils.Scheduler method), 24  
[stoped](#) (btdht.dht.DHT\_BASE attribute), 9  
[stoped](#) (btdht.dht.RoutingTable attribute), 17

## T

[t](#) (btdht.krcp.BError attribute), 25  
[t](#) (btdht.krcp.BMessage attribute), 27  
[task\\_done\(\)](#) (btdht.utils.PollableQueue method), 23  
[thread\\_alive\(\)](#) (btdht.utils.Scheduler method), 24  
[threads](#) (btdht.dht.DHT\_BASE attribute), 9  
[threads](#) (btdht.dht.RoutingTable attribute), 17  
[to\\_bytes\(\)](#) (btdht.utils.ID class method), 22  
[to\\_refresh](#) (btdht.dht.Bucket attribute), 16  
[to\\_schedule](#) (btdht.dht.DHT\_BASE attribute), 9  
[to\\_schedule](#) (btdht.dht.RoutingTable attribute), 17  
[to\\_send](#) (btdht.dht.DHT\_BASE attribute), 9  
[token](#) (btdht.dht.DHT\_BASE attribute), 9  
[transaction\\_type](#) (btdht.dht.DHT\_BASE attribute), 9  
[TransactionIdUnknown](#), 29  
[trie](#) (btdht.dht.RoutingTable attribute), 17

## V

[v](#) (btdht.krcp.BMessage attribute), 27  
[value](#) (btdht.utils.ID attribute), 22

## Y

[y](#) (btdht.krcp.BError attribute), 25  
[y](#) (btdht.krcp.BMessage attribute), 27

## Z

[zombie](#) (btdht.dht.DHT\_BASE attribute), 9  
[zombie](#) (btdht.dht.RoutingTable attribute), 18  
[zombie](#) (btdht.utils.Scheduler attribute), 24