BEP:	5
Title:	DHT Protocol
Version:	3dec52cb3ae103ce22358e3894b31cad47a6f22b
Last-Modified:	Tue Apr 2 16:51:45 2013 -0700
Author:	Andrew Loewenstern <drue@bittorrent.com>, Arvid Norberg <arvid@bittorrent.com>
Status:	Draft
Type:	Standards Track
Created:	31-Jan-2008
Post-History:	22-March-2013: Add "implied_port" to announce_peer message, to improve NAT support

BitTorrent uses a "distributed sloppy hash table" (DHT) for storing peer contact information for "trackerless" torrents. In effect, each peer becomes a tracker. The protocol is based on Kademila [1] and is implemented over UDP.

Please note the terminology used in this document to avoid confusion. A "peer" is a client/server listening on a TCP port that implements the BitTorrent protocol. A "node" is a client/server listening on a UDP port implementing the distributed hash table protocol. The DHT is composed of nodes and stores the location of peers. BitTorrent clients include a DHT node, which is used to contact other nodes in the DHT to get the location of peers to download from using the BitTorrent protocol.

Overview

Each node has a globally unique identifier known as the "node ID." Node IDs are chosen at random from the same 160-bit space as BitTorrent infohashes [2]. A "distance metric" is used to compare two node IDs or a node ID and an infohash for "closeness." Nodes must maintain a routing table containing the contact information for a small number of other nodes. The routing table becomes more detailed as IDs get closer to the node's own ID. Nodes know about many other nodes in the DHT that have IDs that are "close" to their own but have only a handful of contacts with IDs that are very far away from their own.

In Kademlia, the distance metric is XOR and the result is interpreted as an unsigned integer. `distance(A,B) = |A xor B|` Smaller values are closer.

When a node wants to find peers for a torrent, it uses the distance metric to compare the infohash of the torrent with the IDs of the nodes in its own routing table. It then contacts the nodes it knows about with IDs closest to the infohash and asks them for the contact information of peers currently downloading the torrent. If a contacted node knows about peers for the torrent, the peer contact information is returned with the response. Otherwise, the contacted node must respond with the contact information of the nodes in its routing table that are closest to the infohash of the torrent. The original node iteratively queries nodes that are closer to the target infohash until it cannot find any closer nodes. After the search is exhausted, the client then inserts the peer contact information for itself onto the responding nodes with IDs closest to the infohash of the torrent.

The return value for a query for peers includes an opaque value known as the "token." For a node to announce that its controlling peer is downloading a torrent, it must present the token received from the same queried node in a recent query for peers. When a node attempts to "announce" a torrent, the queried node checks the token against the querying node's IP address. This is to prevent malicious hosts from signing up other hosts for torrents. Since the token is merely returned by the querying node to the same node it received the token from, the implementation is not defined. Tokens must be accepted for a reasonable amount of time after they have been distributed. The BitTorrent implementation uses the SHA1 hash of the IP address concatenated onto a secret that changes every five minutes and tokens up to ten minutes old are accepted.

Routing Table

Every node maintains a routing table of known good nodes. The nodes in the routing table are used as starting points for queries in the DHT. Nodes from the routing table are returned in response to queries from other nodes.

Not all nodes that we learn about are equal. Some are "good" and some are not. Many nodes using the DHT are able to send queries and receive responses, but are not able to respond to queries from other nodes. It is important that each node's routing table must contain only known good nodes. 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. Nodes that we know are good are given priority over nodes with unknown status.

The routing table covers the entire node ID space from 0 to 2^{160}. The routing table is subdivided into "buckets" that each cover a portion of the space. An empty table has one bucket with an ID space range of min=0, max=2^{160}. When a node with ID "N" is inserted into the table, it is placed within the bucket that has min &lt;= N &lt; max. An empty table has only one bucket so any node must fit within it. Each bucket can only hold K nodes, currently eight, before becoming "full." When a bucket is full of known good nodes, no more nodes may be added unless our own node ID falls within the range of the bucket. In that case, the bucket is replaced by two new buckets each with half the range of the old bucket and the nodes from the old bucket are distributed among the two new ones. For a new table with only one bucket, the full bucket is always split into two new buckets covering the ranges 0..2^{159} and 2^{159}..2^{160}.

When the bucket is full of good nodes, the new node is simply discarded. If any nodes in the bucket are known to have become bad, then one is replaced by the new node. If there are any questionable nodes in the bucket have not been seen in the last 15 minutes, the least recently seen node is pinged. If the pinged node responds then the next least recently seen questionable node is pinged until one fails to respond or all of the nodes in the bucket are known to be good. If a node in the bucket fails to respond to a ping, it is suggested to try once more before discarding the node and replacing it with a new good node. In this way, the table fills with stable long running nodes.

Each bucket should maintain a "last changed" property to indicate how "fresh" the contents are. When a node in a bucket is pinged and it responds, or a node is added to a bucket, or a node in a bucket is replaced with another node, the bucket's last changed property should be updated. Buckets that have not been changed in 15 minutes should be "refreshed." This is done by picking a random ID in the range of the bucket and performing a find_nodes search on it. Nodes that are able to receive queries from other nodes usually do not need to refresh buckets often. Nodes that are not able to receive queries from other nodes usually will need to refresh all buckets periodically to ensure there are good nodes in their table when the DHT is needed.

Upon inserting the first node into its routing table and when starting up thereafter, the node should attempt to find the closest nodes in the DHT to itself. It does this by issuing find_node messages to closer and closer nodes until it cannot find any closer. The routing table should be saved between invocations of the client software.

BitTorrent Protocol Extension

The BitTorrent protocol has been extended to exchange node UDP port numbers between peers that are introduced by a tracker. In this way, clients can get their routing tables seeded automatically through the download of regular torrents. Newly installed clients who attempt to download a trackerless torrent on the first try will not have any nodes in their routing table and will need the contacts included in the torrent file.

Peers supporting the DHT set the last bit of the 8-byte reserved flags exchanged in the BitTorrent protocol handshake. Peer receiving a handshake indicating the remote peer supports the DHT should send a PORT message. It begins with byte 0x09 and has a two byte payload containing the UDP port of the DHT node in network byte order. Peers that receive this message should attempt to ping the node on the received port and IP address of the remote peer. If a response to the ping is recieved, the node should attempt to insert the new contact information into their routing table according to the usual rules.

Torrent File Extensions

A trackerless torrent dictionary does not have an "announce" key. Instead, a trackerless torrent has a "nodes" key. This key should be set to the K closest nodes in the torrent generating client's routing table. Alternatively, the key could be set to a known good node such as one operated by the person generating the torrent. Please do not automatically add "router.bittorrent.com" to torrent files or automatically add this node to clients routing tables.

```
nodes = [["<host>", <port>], ["<host>", <port>], ...]
nodes = [["127.0.0.1", 6881], ["your.router.node", 4804]]
```

KRPC Protocol

The KRPC protocol is a simple RPC mechanism consisting of bencoded dictionaries sent over UDP. A single query packet is sent out and a single packet is sent in response. There is no retry. There are three message types: query, response, and error. For the DHT protocol, there are four queries: ping, find_node, get_peers, and announce_peer.

A KRPC message is a single dictionary with two keys common to every message and additional keys depending on the type of message. Every message has a key "t" with a string value representing a transaction ID. This transaction ID is generated by the querying node and is echoed in the response, so responses may be correlated with multiple queries to the same node. The transaction ID should be encoded as a short string of binary numbers, typically 2 characters are enough as they cover 2^16 outstanding queries. The other key contained in every KRPC message is "y" with a single character value describing the type of message. The value of the "y" key is one of "q" for query, "r" for response, or "e" for error.

Contact Encoding

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.

Queries

Queries, or KRPC message dictionaries with a "y" value of "q", contain two additional keys; "q" and "a". Key "q" has a string value containing the method name of the query. Key "a" has a dictionary value containing named arguments to the query.

Responses

Responses, or KRPC message dictionaries with a "y" value of "r", contain one additional key "r". The value of "r" is a dictionary containing named return values. Response messages are sent upon successful completion of a query.

Errors

Errors, or KRPC message dictionaries with a "y" value of "e", contain one additional key "e". The value of "e" is a list. The first element is an integer representing the error code. The second element is a string containing the error message. Errors are sent when a query cannot be fulfilled. The following table describes the possible error codes:

Code	Description
201	Generic Error
202	Server Error
203	Protocol Error, such as a malformed packet, invalid arguments, or bad token
204	Method Unknown

Example Error Packets:

```
generic error = {"t":"aa", "y":"e", "e":[201, "A Generic Error Ocurred"]}
bencoded = d1:eli201e23:A Generic Error Ocurrede1:t2:aa1:y1:ee
```

DHT Queries

All queries have an "id" key and value containing the node ID of the querying node. All responses have an "id" key and value containing the node ID of the responding node.

ping

The most basic query is a ping. "q" = "ping" A ping query has a single argument, "id" the value is a 20-byte string containing the senders node ID in network byte order. The appropriate response to a ping has a single key "id" containing the node ID of the responding node.

```
arguments:  {"id" : "<querying nodes id>"}

response: {"id" : "<queried nodes id>"}
```

Example Packets

```
ping Query = {"t":"aa", "y":"q", "q":"ping", "a":{"id":"abcdefghij0123456789"}}
bencoded = d1:ad2:id20:abcdefghij0123456789e1:q4:ping1:t2:aa1:y1:qe

Response = {"t":"aa", "y":"r", "r": {"id":"mnopqrstuvwxyz123456"}}
bencoded = d1:rd2:id20:mnopqrstuvwxyz123456e1:t2:aa1:y1:re
```

find_node

Find node is used to find the contact information for a node given its ID. "q" == "find_node" A find_node query has two arguments, "id" containing the node ID of the querying node, and "target" containing the ID of the node sought by the queryer. When a node receives a find_node query, it should respond with a key "nodes" and value of a string containing the compact node info for the target node or the K (8) closest good nodes in its own routing table.

```
arguments:  {"id" : "<querying nodes id>", "target" : "<id of target node>"}

response: {"id" : "<queried nodes id>", "nodes" : "<compact node info>"}
```

Example Packets

```
find_node Query = {"t":"aa", "y":"q", "q":"find_node", "a": {"id":"abcdefghij0123456789", "target":"mnopqrstuvwxyz12345
bencoded = d1:ad2:id20:abcdefghij01234567896:target20:mnopqrstuvwxyz123456e1:q9:find_node1:t2:aa1:y1:qe

Response = {"t":"aa", "y":"r", "r": {"id":"0123456789abcdefghij", "nodes": "def456..."}}
bencoded = d1:rd2:id20:0123456789abcdefghij5:nodes9:def456...e1:t2:aa1:y1:re
```

get_peers

Get peers associated with a torrent infohash. "q" = "get_peers" A get_peers query has two arguments, "id" containing the node ID of the querying node, and "info_hash" containing the infohash of the torrent. If the queried node has peers for the infohash, they are returned in a key "values" as a list of strings. Each string containing "compact" format peer information for a single peer. If the queried node has no peers for the infohash, a key "nodes" is returned containing the K nodes in the queried nodes routing table closest to the infohash supplied in the query. In either case a "token" key is also included in the return value. The token value is a required argument for a future announce_peer query. The token value should be a short binary string.

```
arguments:  {"id" : "<querying nodes id>", "info_hash" : "<20-byte infohash of target torrent>"}

response: {"id" : "<queried nodes id>", "token" :"<opaque write token>", "values" : ["<peer 1 info string>", "<peer 2 i

or: {"id" : "<queried nodes id>", "token" :"<opaque write token>", "nodes" : "<compact node info>"}
```

Example Packets:

```
get_peers Query = {"t":"aa", "y":"q", "q":"get_peers", "a": {"id":"abcdefghij0123456789", "info_hash":"mnopqrstuvwxyz12
bencoded = d1:ad2:id20:abcdefghij01234567899:info_hash20:mnopqrstuvwxyz123456e1:q9:get_peers1:t2:aa1:y1:qe

Response with peers = {"t":"aa", "y":"r", "r": {"id":"abcdefghij0123456789", "token":"aoeusnth", "values": ["axje.u", "
bencoded = d1:rd2:id20:abcdefghij01234567895:token8:aoeusnth6:valuesl6:axje.u6:idhtnmee1:t2:aa1:y1:re

Response with closest nodes = {"t":"aa", "y":"r", "r": {"id":"abcdefghij0123456789", "token":"aoeusnth", "nodes": "def4
bencoded = d1:rd2:id20:abcdefghij01234567895:nodes9:def456...5:token8:aoeusnthe1:t2:aa1:y1:re
```

announce_peer

Announce that the peer, controlling the querying node, is downloading a torrent on a port. announce_peer has four arguments: "id" containing the node ID of the querying node, "info_hash" containing the infohash of the torrent, "port" containing the port as an integer, and the "token" received in response to a previous get_peers query. The queried node must verify that the token was previously sent to the same IP address as the querying node. Then the queried node should store the IP address of the querying node and the supplied port number under the infohash in its store of peer contact information.

There is an optional argument called `implied_port` which value is either 0 or 1. If it is present and non-zero, the `port` argument should be ignored and the source port of the UDP packet should be used as the peer's port instead. This is useful for peers behind a NAT that may not know their external port, and supporting uTP, they accept incoming connections on the same port as the DHT port.

```
arguments:  {"id" : "<querying nodes id>",
  "implied_port": <0 or 1>,
  "info_hash" : "<20-byte infohash of target torrent>",
  "port" : <port number>,
  "token" : "<opaque token>"}

response: {"id" : "<queried nodes id>"}
```

Example Packets:

```
announce_peers Query = {"t":"aa", "y":"q", "q":"announce_peer", "a": {"id":"abcdefghij0123456789", "implied_port": 1, "i
bencoded = d1:ad2:id20:abcdefghij01234567899:info_hash20:<br />
mnopqrstuvwxyz1234564:porti6881e5:token8:aoeusnthe1:q13:announce_peer1:t2:aa1:y1:qe
```

```
Response = {"t":"aa", "y":"r", "r": {"id":"mnopqrstuvwxyz123456"}}
bencoded = d1:rd2:id20:mnopqrstuvwxyz123456e1:t2:aa1:y1:re
```

References

[1] Peter Maymounkov, David Mazieres, "Kademlia: A Peer-to-peer Information System Based on the XOR Metric", *IPTPS 2002*. http://www.cs.rice.edu/Conferences/IPTPS02/109.pdf

[2] Use SHA1 and plenty of entropy to ensure a unique ID.

Copyright