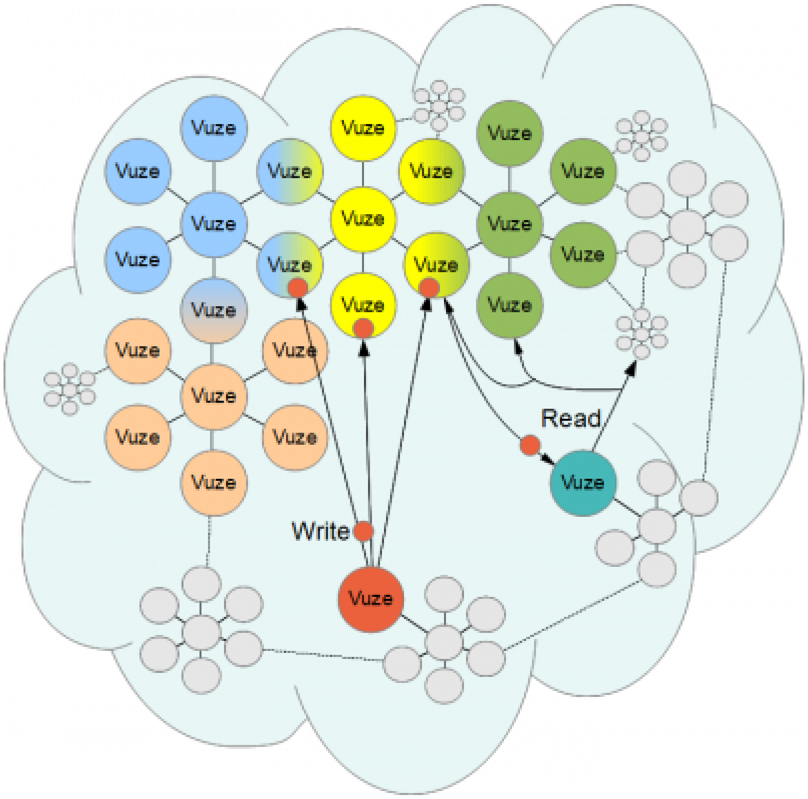
Distributed hash table

From VuzeWiki

Contents

Overview

The distributed database in all current Azureus builds (≥2.3.0.0) is based on a UDP based Distributed Hash Table (DHT). In particular Azureus uses a modified Kademlia implementation.



How it works

The DHT acts like a transparent, distributed Hash Table (thus the name) with node IDs based on the SHA-1 hash of the node's IP/Port combination.

It supports 4 basic operations:

- Ping - to ensure up to date routing tables
- Lookup node - to find nodes that are near to the desired key in the keyspace
- Get value - retrieve a list of values from those nodes
- Store value - store a single value or a list of values on these nodes

To handle the steady arrival and departure of nodes in the DHT every store is performed on those 20 nodes which are the nearest to the desired key. To handle possibly malicious nodes in the network every lookup does request the data from 20 nodes too.

Due to the nature of hash function fuzzy search algorithms required for keyword based searching are hard to implement because a small change in the input will result in a completely different output and thus another key and not a nearby one. Thus only precise lookups, i.e. based on unique content identifiers like torrent hash can be performed on a DHT without large overhead.

Additional Features

It is extended with several features not specified in the Kademlia whitepaper, here a tentative and incomplete list (since it's still under development and there are no official specs available):

- load and storage sized key diversification to prevent hotspots
- caching along path
- distinction between partial and exhaustive gets
- storage verification
- bootstrapping from nodes discovered in a (BitTorrent specific) swarm
- Vivaldi Coordinates to estimate the RTT between nodes (see Vivaldi View)
- anti-spoof mechanisms (prevent source address spoofing for stores to enforce 1 entry per IP limit)
- NAT hole punching
- encrypted data transfer to ensure nobody can fetch a .torrent without the matching torrent hash.

Implementation Specifications for the DHT should be included here

Packet format, RPCs, routing tables, diversification.... stuff

(work in progress)

Packet format

Protocol versions

Each packet has a field that specifies protocol version. Some values used in specific packets are present only when protocol version meets certain condition. Symbolic names used in the conditions as well as their values are in the following table.

Beware: changes in protocol versions after `VIVALDI_FINDVALUE` are not completely documented here.

Regarding backward compatibility: contacts' versions are stored, so any node should know the version of a contact it is going to message. However, there is a minimum acceptable protocol version (currently `2502`, which is the same as `VIVALDI_FINDVALUE`).

Protocol versions

Constant name	Value
`DIV_AND_CONT`	6
`ANTI_SPOOF`	7
`ANTI_SPOOF2`	8
`FIX_ORIGINATOR`	9
`NETWORKS`	9
`VIVALDI`	10
`REMOVE_DIST_ADD_VER`	11
`XFER_STATUS`	12
`SIZE_ESTIMATE`	13
`VENDOR_ID`	14
`BLOCK_KEYS`	14
`GENERIC_NETPOS`	15
`VIVALDI_FINDVALUE`	16
`ANON_VALUES`	17
`CVS_FIX_OVERLOAD_V1`	18
`CVS_FIX_OVERLOAD_V2`	19
`MORE_STATS`	20
`CVS_FIX_OVERLOAD_V3`	21
`MORE_NODE_STATUS`	22
`LONGER_LIFE`	23
`REPLICATION_CONTROL`	24
`RESTRICT_ID_PORTS`	32
`RESTRICT_ID_PORTS2`	33
`RESTRICT_ID_PORTS2X`	34
`RESTRICT_ID_PORTS2Y`	35
`RESTRICT_ID_PORTS2Z`	36
`RESTRICT_ID3`	50

Serialisation

Serialisation

Type	Width	Note
`byte`	1 B	
`short`	2 B	big endian
`int`	4 B	big endian
`long`	8 B	big endian
`boolean`	1 B	`false` = 0; `true` = 1
address	7 B or 19 B	first byte indicates length of the IP address (4 for IPv4, 16 for IPv6); next comes the address in network byte order; the last value is port number as `short`
contact	9 B or 21 B	first byte indicates contact type, which must be UDP (1); second byte indicates the contact's protocol version; the rest is an address

Transport Value

Name	Type	Protocol version	Note
`VERSION`	`int`	≥`REMOVE_DIST_ADD_VER`	Version of the value. (details later)
`CREATED`	`long`	always	Creation time. Milliseconds since the epoch.
`VALUE_BYTES_COUNT`	`short`	always	Number of bytes in the value.
`VALUE_BYTES`	`bytes`	always	The bytes of the value.
`ORIGINATOR`	*contact*	always	The node that created the value.
`FLAGS`	`byte`	always	value specific flags - see below
`LIFE_HOURS`	`byte`	≥`LONGER_LIFE`	Hours for the value to live. (Details of how it's handled TODO)
`REPLICATION_FACTOR`	`byte`	≥`REPLICATION_CONTROL`	Per-value # of replicas to maintain.

Flags contains a somewhat random collection of bits, revealing the evolving nature of the beast.

- 0x00 - the value represents a 'single value', this has no bits set and conveys the default semantics for values.
- 0x01 - from when the DHT was only used for tracking peers: signifies a downloading peer.
- 0x02 - as above but signifies a seeding peer.
- 0x04 - multi-value - when storing values larger than the maximum permitted value size (512 bytes) the value is fragmented internally and stored at alternative key locations derived from the initial value's key. This flag is set on all but the last fragmented value. It is supported at the plugin interface's 'Distributed Database' level.
- 0x08 - stats marker. When set stats regarding the value will be returned instead of the value itself (for diagnosing DHT load issues)
- 0x10 - anonymous marker. When set the originator of the stored value will not be returned in query responses.
- 0x20 - precious marker. Indicates that the value should be replicated more frequently as an attempt to increase resilience.

- 0x40 - local flag only - 'put and forget'. The value will be stored but will not be republished.
- 0x80 - obfuscate lookup - rather than looking up the key directly an approximate key and value are used so that intermediate nodes in the lookup process don't learn the real key. Once a node is found that stored the approximate value a direct lookup with the real key is performed against the same node.

Headers

Note that connection IDs in requests are guaranteed to have their MSB set to 1. Requests always start with the action, which always has the MSB clear. Therefore, the MSB of an incoming packet should be used to distinguish requests from replies.

Request header

Name	Type	Protocol version	Note
`CONNECTION_ID`	`long`	always	random number with most significant bit set to 1
`ACTION`	`int`	always	type of the packet
`TRANSACTION_ID`	`int`	always	unique number used through the communication; it is randomly generated at the start of the application and increased by 1 with each sent packet
`PROTOCOL_VERSION`	`byte`	always	version of protocol used in this packet
`VENDOR_ID`	`byte`	≥`VENDOR_ID`	ID of the DHT implementator; 0 = Azureus, 1 = ShareNet, 255 = unknown
`NETWORK_ID`	`int`	≥`NETWORKS`	ID of the network; 0 = stable version; 1 = CVS version
`LOCAL_PROTOCOL_VERSION`	`byte`	≥`FIX_ORIGINATOR`	maximum protocol version this node supports; if this packet's protocol version is <`FIX_ORIGINATOR` then the value is stored at the end of the packet
`NODE_ADDRESS`	*address*	always	address of the local node
`INSTANCE_ID`	`int`	always	application's helper number; randomly generated at the start
`TIME`	`long`	always	time of the local node; stored as number of milliseconds since Epoch (http://en.wikipedia.org/wiki/Unix_time)

Reply header

Name	Type	Protocol version	Note
ACTION	int	always	type of the packet
TRANSACTION_ID	int	always	must be equal to TRANSACTION_ID from the request
CONNECTION_ID	long	always	must be equal to CONNECTION_ID from the request
PROTOCOL_VERSION	byte	always	version of protocol used in this packet
VENDOR_ID	byte	≥VENDOR_ID	same meaning as in the request
NETWORK_ID	int	≥NETWORKS	same meaning as in the request
INSTANCE_ID	int	always	instance id of the node that replies to the request

PING

Request PING has ACTION field equal to 1024. Body of the packet is empty.

ACTION of PING reply is equal to 1025. If protocol version is ≥VIVALDI then packet's body carries network coordinates.

STORE

Request STORE ACTION = 1026.

Request STORE

Name	Type	Protocol version	Note
SPOOF_ID	int	≥ANTI_SPOOF	Spoof ID of the target node; it must be the same number as previously retrived through FIND_NODE reply.
KEYS_COUNT	byte	always	Number of keys that follow.
KEYS	*keys*	always	Keys that the target node should store.
VALUE_GROUPS_COUNT	byte	always	Number of groups of values this packet contains.
VALUES	*value groups*	always	Groups of values, one for each key; values are stored in the same order as keys.

Reply STORE ACTION = 1027.

Reply STORE

Name	Type	Protocol version	Note
DIVERSIFICATIONS_LENGTH	byte	≥DIV_AND_CONT	Number of diversifications this packet contains.
DIVERSIFICATIONS	byte[]	≥DIV_AND_CONT	Array with diversifications; they are stored in the same order as keys and values from the request.

FIND_NODE

Request FIND_NODE ACTION = 1028.

Request FIND_NODE

Name	Type	Protocol version	Note
ID_LENGTH	byte	always	Length of the following ID.
ID	byte[]	always	ID to search
NODE_STATUS	int	≥MORE_NODE_STATUS	Node status (TODO: describe)
DHT_SIZE	int	≥MORE_NODE_STATUS	Estimated size of the DHT; Unknown value can be indicated as zero.

Reply FIND_NODE ACTION = 1029

Reply FIND_NODE

Name	Type	Protocol version	Note
SPOOF_ID	int	≥ANTI_SPOOF	Spoof ID of the requesting node; it should be constructed from information known about requesting contact and not easily guessed by others.
NODE_TYPE	int	≥XFER_STATUS	Type of the replying node; Possible values are 0 for bootstrap node, 1 for ordinary node and ffffffffh for unknown type.
DHT_SIZE	int	≥SIZE_ESTIMATE	Estimated size of the DHT; Unknown value can be indicated as zero.
NETWORK_COORDINATES	*network coordinates*	≥VIVALDI	Network coordinates of replying node.
CONTACTS_COUNT	short	always	Number of carried contacts.
CONTACTS	*contacts*	always	List with contacts.

FIND_VALUE

Request FIND_VALUE ACTION = 1030.

Request FIND_VALUE

Name	Type	Note
KEY	*key*	Key for which the values are requested.
FLAGS	byte	Flags for the operation; possible values are: - SINGLE_VALUE = 00h - DOWNLOADING = 01h - SEEDING = 02h - MULTI_VALUE = 04h - STATS = 08h If STATS are used then some stats for the value are returned instead of value itself. They are serialised as follows: 0 (byte) - version, number of stored values for the key (int), total size of stored values (int), reads per minute (int), diversification type (byte).
MAX_VALUES	byte	Maximum number of returned values.

Reply FIND_VALUE ACTION = 1031.

Reply FIND_VALUE

Name	Type	Condition	Note
HAS_CONTINUATION	boolean	protocol version ≥DIV_AND_CONT	Indicates whether there is at least one other packet with values.
HAS_VALUES	boolean	always	Indicates whether this packet carries values or contacts.
CONTACTS_COUNT	short	HAS_VALUES == false	Number of stored contacts.
CONTACTS	*contacts*	HAS_VALUES == false	Stored contacts that are close to the searched key.
NETWORK_COORDINATES	*network coordinates*	HAS_VALUES == false && protocol version ≥VIVALDI_FINDVALUE	Network coordinates of the replying node.
DIVERSIFICATION_TYPE	byte	HAS_VALUES == true && protocol version ≥DIV_AND_CONT	Type of key's diversification.
VALUES	*value group*	HAS_VALUES == true	Values that match searched key.

ERROR

This message type is used only when replying. It's action number is equal to 1032.

Reply ERROR

Name	Type	Condition	Note
`ERROR_TYPE`	`int`	always	Type of the error. Possible values are: - `WRONG_ADDRESS = 1` - originator's address stored in the request is incorrect - `KEY_BLOCKED = 2` - the requested key has been blocked
`SENDER_ADDRESS`	*address*	`ERROR_TYPE == WRONG_ADDRES`	Real originator's address.
`KEY_BLOCK_REQUEST_LENGTH`	`byte`	`ERROR_TYPE == KEY_BLOCKED`	Length of the following request.
`KEY_BLOCK_REQUEST`	`byte[]`	`ERROR_TYPE == KEY_BLOCKED`	Request that blocks/unlocks the key.
`SIGNATURE_LENGTH`	`short`	`ERROR_TYPE == KEY_BLOCKED`	Length of the following signature.
`SIGNATURE`	`byte[]`	`ERROR_TYPE == KEY_BLOCKED`	Signature of the request.

KEY_BLOCK

Request KEY_BLOCK `ACTION` = 1036.

Request KEY_BLOCK

Name	Type	Note
`SPOOF_ID`	`int`	Spoof ID obtained through FIND_NODE request.
`KEY_BLOCK_REQUEST_LENGTH`	`byte`	Length of the following request.
`KEY_BLOCK_REQUEST`	`byte[]`	Request that blocks/unlocks the key.
`SIGNATURE_LENGTH`	`short`	Length of the following signature.
`SIGNATURE`	`byte[]`	Signature of the request.

Reply KEY_BLOCK `ACTION` = 1037. Body of the reply is empty.

Retrieved from "https://wiki.vuze.com/mediawiki/index.php?title=Distributed_hash_table&oldid=14796"

Category: Technical Information

- This page was last modified on 25 October 2012, at 18:11.
- This page has been accessed 64,930 times.