The Design and Implementation of a Distributed Hash Table for a Peer-to-Peer Network

DHT – Distributed Hash Table

Helsinki 27th May 2004

Software Engineering Project

UNIVERSITY OF HELSINKI
Department of Computer Science

Course

581260 Software Engineering Project (6 cr)

Project Group

Marko Räihä
Risto Saarelma
Antti Salonen
Tuomas Toivonen
Tomi Tukiainen
Simo Viitanen

Client

Jussi Lindgren

Project Masters

Juha Taina
Turjo Tuohiniemi

Homepage

`http://www.cs.helsinki.fi/group/dht/`

Change Log

Version	Date	Modifications
1.0	2004-05-27	Released version

Contents

1 Introduction **1**

2 Architecture **2**

2.1 Environment 2

2.1.1 GNUnet 2

2.1.2 Kademlia 2

2.2 Modules 2

2.2.1 DHT API 2

2.2.2 Client-server communications 2

2.2.3 Kademlia 2

2.2.4 Datalayer 3

2.2.5 Tools 3

2.2.6 RPC 3

3 Implementation **4**

3.1 Modules 4

3.1.1 DHT API 4

3.1.2 Client-server 4

3.1.3 Kademlia 5

3.1.4 Datalayer 6

3.1.5 RPC 7

3.2 Compilation 8

4 Suggestions **9**

4.1 Bugs 9

4.2 Wishlist 9

4.2.1 DHT API 9

4.2.2 Client-server communications 10

4.2.3 Kademlia 10

4.2.4 Datalayer 10

5 Conclusion **11**

References **12**

1 Introduction

In this white paper we describe the design and implementation of a distributed hash table using a peer-to-peer networking approach. We used the GNUnet framework for core peer-to-peer networking functionality. As the distributed hash table algorithm we used Kademlia.

The work was performed as part of the *Software Engineering Project*[1] course at the department of computer science[2], University of Helsinki, Finland. The project team consisted of six students. Each participant worked on the project for approximately one and half man months during the spring 2004 term. In addition to the program source code and this white paper in English, the team produced a set of project and specification documents in Finnish. An archive of all the materials is available on the project's web site [RSS+04].

In chapter two we describe the overall architecture of the distributed hash table implementation. Rationale for using GNUnet and Kademlia as basic building blocks is given as is an overview of each implemented component. In chapter three we provide component by component implementation details. The chapter is intended as a gentle introduction to the actual source code. In the fourth chapter we describe for example missing features and suggestions for future improvements.

The project team would like to acknowledge the input of Marianne Korpela and Jussi Lindgren. As the project instructor Marianne helped our at times unruly group stay focused. Jussi, as the project customer, was responsive and knowledgeable, a fact we could have taken more advantage of.

[1] `http://www.cs.helsinki.fi/group/ohtu/`
[2] `http://www.cs.helsinki.fi/`

2 Architecture

2.1 Environment

2.1.1 GNUnet

GNUnet is described at the GNUnet development team's website [GNU04].

2.1.2 Kademlia

Kademlia is described by Maymounkov and Mazières [MM02].

2.2 Modules

2.2.1 DHT API

To access the distributed hash table (DHT) a client application uses an application programming interface (API) implemented in C. The API provides a set of synchronous functions. Each function will invoke the API library and communicate with the local GNUnet daemon resident DHT module over a TCP socket. The standard GNUnet client/server protocol is used in communicating with the daemon.

The DHT API offers five primitives for applications:

create Used to create a new hash table. The creating node automatically joins the new hash table.

join Used to join an existing hash table. After a node has joined to a hash table, it will be possible for the node to perform searches on the table and store new data to the table.

leave Used to leave a hash table that the node has previously joined.

insert Used to store a new <key,value> -pair to a joined hash table.

list Used to retrieve a list of all hash tables the node is joined to.

fetch Used to retrieve a <value> using a <key> from a joined hash table.

2.2.2 Client-server communications

Standard GNUnet client-server communication framework is used.

2.2.3 Kademlia

TBD.

2.2.4 Datalayer

Datalayer is the module representing storage and lookups for locally stored data. Data is stored as a mapping from a hash to a value.

Datalayer supports multiple tables and in addition to storing nodes it supports four different lookups. These include:

by key All data stored to a mapping from a single key.

by own flag All data stored by the local node.

by expiration All data with expiration time exceeding .

by xor metric distance All data units that are closer to another unit in xor metric topology.

In addition datalayer supports persistence module for the data.

2.2.5 Tools

For testing and demonstration purposes a set of command line tools have been developed. Each tool invokes one of the DHT API functions. If the tools are used in a sequence, the output of one tool can be used as the parameters for other tools. For example, the output of *dht-join* tool is a hash table handle which can then be used as a parameter to eg. *dht-create* or *dht-leave*.

2.2.6 RPC

The RPC module is a generic Remote Procedure Call (RPC) mechanism on top of GNUnet's message-based communication. By generic it is meant that it is not specific to the DHT application. Any node may either register a callback function in order to receive RPC's, or node may make an RPC to a remote node. RPC's take a set of request and response parameters, whose values may be of arbitrary length.

The RPC module serializes the request (or response) parameters into a data packet which is then divided into segments that can be transmitted as GNUnet messages to another node. On top of GNUnet's unreliable communication semantics, the RPC module attemps to provide reliable communication, so the messages sent are acknowledged by the receiver and if necessary, retransmitted by the sender.

Directory	Description
gtkui	A (partially implemented) GTK+ user interface to the DHT API.
include	The DHT API headers for use by client applications.
module	Implementation of the DHT as a GNUnet module.
tools	Command line tools for testing the DHT module and API.
util	Utility functions used by the DHT module.

Table 1: Subdirectories containing the DHT source code.

3 Implementation

The DHT module has been implemented to form part of the GNUnet source tree. All DHT related code can be found in the directory *.../src/applications/dht/* (relative to the root of the GNUnet source tree). Table 1 presents an overview of the subdirectories containing the DHT source code.

3.1 Modules

3.1.1 DHT API

Interface *dht_api.h, api_errorcodes.h, api_structs.h, client-server.h*

Implementation *dht-api.c*

The DHT API provides a C language application programming interface to client applications that wish to use the distributed hash table. The API is synchronous. Each function only returns after a definitive set of return values have been prepared (eg. all values have been fetched). The API uses the GNUnet client-server -protocol to communicate with the GNUnet daemon resident DHT implementation. It is thus possible to connect over TCP to remote GNUnet daemons as well.

Full documentation (in Doxygen compatible comments) of the API is given in the *dht_api.h* header. As an example the source code of testing tools in directory *.../tools/* can be perused. Also *.../tools/Makefile.am* can be used as a example on how to incorporate new client applications to the GNU autoconf framework for compilation.

3.1.2 Client-server

Interface *client-server.h*

Implementation *client-server.c*

Undocumented. See Doxygen comments in code.

3.1.3 Kademlia

Interface *dht.h*

Implementation *dht.c, kademlia.h, kademlia.c*

Kademlia is the module that implements the actual distributed hash table functionality. The interface in dht.h provides functions for joining and leaving tables in the network, creating new tables, inserting <key, value> pairs and fetching <key, value> pairs from the dht network based on key.

The file kademlia.c contains the internal implementation of the dht system using the Kademlia algorithm. The Kademlia implementation stores local information using the datalayer module and queries for and receives remote information using the rpc module. The Kademlia implementation maintains a fixed-size local routing table in each node. Routing information is updated when the implementation communicates with other nodes. Using the routing tables, the implementation should be able to find a specific dht node in O(log n) time.

The most important functions in the interface are the following:

- create(tableConfig, metaData, tableId) - creates a new table and writes its identifier into tableId
- join(nodeAddress, tableId) - joins the table with a given identifier that is hosted in the given node
- leave(tableId) - leaves the table with the given identifier
- insert(tableId, key, value) - inserts a <key, value> pair in a table
- fetch(tableId, key, result) - searches for <key, value> pairs with the given key and writes them in the result set
- tables(nodeAddress, tableSet) - lists all tables a given node has joined

Nodes cannot join the dht network by themselves because they are not initially aware of any nodes which are running the dht service. To have the node join the dht network, the user must be aware of at least one other node already on the network. The node will then be able to join the network by joining into one or more tables in the other node. After the node has joined some tables, the Kademlia implementation will take over and begin routing queries, updating its routing information and storing <key, value> pairs as necessary.

3.1.4 Datalayer

Interface *datalayer.h, datalayer_persistance.h*

Implementation *datalayer.c, datalayer_persistance.h*

Datalayer is the module representing all local lookups for <key,value> -pairs. Interface of the datalayer is presented in the file datalayer.h. It also presents the structs needed by the datalayer. These structs include DHT_DataStoreUnit, DHT_LocalStoreResultSet, DHT_LocalTableSet and DHT_TableDef.

DHT_DataStoreUnit is the basic entity for storing units. It contains hash, data, creation- and expiration time and flags. So datastore stores only mapping between a hash and a data. How these are constructed is not datalayer's concern.

DHT_TableDef is the struct which defines all that datalayer needs to know about one table. This includes table id, metadata and config information. Table id is the TableId identifier of the table. Metadata is strictly for user interfaces and config information is stored because it needs to be copied to new nodes joining the table.

DHT_LocalStoreResultSet and DHT_LocalTableSet are structs representing results. They both present an abstract iteration pattern for the search results with an abstract typedef for both: LocalStoreTableIter and LocalStoreResultIter. Implementation of these iterators is not visible for the user of the datastore. The implementations are defined in file datastore.c.

Datalayer is split in two layers: search layer and persistence layer. Search layer, as the name indicates, is responsible for implementing searches for the data. It implements four different searches:

- localStoreGet(tableId, key) - searches all <key,value>-pairs stored to the specified table with the specified key
- localStoreGetOwnData(tableId) - searches all <key,value>-pairs stored to the specified table with the flag myData on
- localStoreGetExpired(tableId, TIME_T now) - searches all <key,value>-pairs stored to the specified table with the expiration time older than the now parameter
- localStoreGetCloserTo(tableId, myID, newID) - searches all <key,value>-pairs stored to the specified table with the key hash closer to the new id than my id. Closeness of the hashes is implemented in Kademlia's XOR metric.

Persistence layer is responsible for persisting the datastore. At the moment it is not implemented and the datastore only stores the data units to the search data structures of search layer.

3.1.5 RPC

Interface *module/rpc.h*

Implementation *module/rpc.c*

The RPC module provides an generic RPC mechanism on top of GNUnet's message-based communication primitives. Unlike GNUnet's unreliable communication semantics, the RPC module attempts to provide a reliable communication mechanism through acknowledgement and retransmission of messages in case they get lost.

The interface is very straight-forward:

- RPC_init - Registers peer-to-peer message handlers, etc.
- RPC_register - Registers an RPC locally, identified by a string name and implemented by a callback function
- RPC_execute - Executes an RPC on another node (possibly itself)

In addition, the interface includes the type RPC_Param, which represents a set of parameters. The parameter names are strings and their values arbitrary data. No limit has been implemented for the size of a parameter's value. A set of functions is provided to access parameter sets and the same type is used for passing both request and response parameters. The parameter sets are accessed in an indexed fashion, and perhaps unconventionally, there can be several parameters with the same name in a set.

Every successful RPC call in a GNUnet network includes the caller sending a request which is acknowledged by the receiver, and after the receiver has executed the RPC locally it returns a response which is again acknowledged by the caller. The RPC module serializes the request (or response) and its parameters into a data packet, which is then divided into numbered segments which are eventually transmitted with GNUnet messages. The RPC module defines three new GNUnet message types for transmitting request data (REQ), response data (RES) and acknowledgements (ACK).

Every REQ or RES message is acknowlegded by the receiver and if an acknowledgement is not received, retransmitted by the sender. The RPC module was designed to use a sliding window mechanism similar to TCP, with the send window's size being increased until problems arise. However, with the current implementation the size remains 1, although the module has otherwise been implemented so that the send window could be of arbitrary size. In order to utilize the bandwidth of the network between the sender and the receiver, some sort of slow-start mechanism should be implemented.

The RPC module is inherently multi-threaded. In addition to the interface function RPC_execute, control can be returned to the RPC module from GNUnet core with several callbacks. As GNUnet may call the callbacks concurrently, the callback functions and RPC_execute use a mutex to control access to shared data structures. The following callback functions are registered with GNUnet.

- handleRPCMessageReq - Called when a REQ message is received
- handleRPCMessageRes - Called when a RES message is received
- handleRPCMessageAck - Called when an ACK message is received
- retransmitTimerExpire - Called when the retransmission timer of sent message expires
- ackTimerExpire - Called when the acknowledgement timer of a received message expires

3.2 Compilation

The DHT module is implemented as part of the GNUnet source tree and is integrated with the GNU autoconf based compilation framework used by GNUnet. See autoconf documentation and *Makefile.am* files in the various DHT source code directories for more information.

Operation	Single node	Multiple nodes
dht-create	OK	??
dht-fetch	OK	ERROR
dht-insert	ERROR	ERROR
dht-join	OK	OK
dht-leave	OK	??
dht-list	OK	ERROR
dht-inserted-drop	??	ERROR
dht-inserted-list	ERROR	N/A

Table 2: Summary of test results for each DHT API operation. OK signifies successful test case, ERROR a failure. N/A means that the operation doesn't apply to the single or multiple nodes case. Question marks point to cases that haven't been tested.

4 Suggestions

4.1 Bugs

Some of the bugs and missing features identified at the time of project closure are listed in the GNUnet bug tracking system, Mantis[3].

Table 2 summarises test results for each of the DHT API functions. The tests were performed using the command line tools. Success or failure indication was derived from debug logs and output of the tools. A more complete report of test cases and their results is available in Finnish at the project site [RSS+04].

4.2 Wishlist

4.2.1 DHT API

- Add support for multiple threads; ie. make the API asynchronous.

- The DHT implementation inside the GNUnet daemon doesn't differentiate between clients. For example, two unrelated users can connect to tha same daemon and issue operations. User A can join a table, but before executing fetches or inserts, user B could have unsubscribed (leave) the node from the previously joined table. There would appear to be no easy workaround to this. The decision to implement all DHT functionality inside GNUnet daemon instead of using an end-to-end model is inherent in the design. Perhaps from the perspective of the Kademlia algorithm we should have modelled each client application as a node of the peer-to-peer network. Functionality could thus have been pushed to the edge.

[3] Mantis is available at `http://www.ovmj.org/~mantis/`.

4.2.2 Client-server communications

- Implement better checks for incoming messages.
- Limit size of reply messages.
- Implement process status queries (csStatus-function).

4.2.3 Kademlia

- **Joining a table** — Joining could be improved with at least following:
  - Receiver:
    - Send the DHT_TableConfig to the sender.
    - Send information about n random nodes in the same table.
  - Sender
    - Send join to multiple nodes (the Byzantine General Problem).
- **Timed tasks (crons)** — All cron jobs maintaining the dht. These include:
  - Republishing own data (every 24 hours).
  - Dropping expired data (every now and then).
  - Distibuting data to the closest nodes in table (newly joined nodes) (every hour).

4.2.4 Datalayer

- Enhanced data structures:
  - Hashtable for mapping.
  - Use the general Vector data structure (used also by RPC and Kademlia).
- Implement the persistence layer properly:
  - One file per table.
  - A daemon thread that writes the units to file.

5 Conclusion

As a whole the project can be considered a moderate success. Almost all of the designed functionality was implemented. However, not all of the implementation was adequately tested and several critical bugs have been identified. Also, the design centralises functionality in the DHT module instead of pushing it to the client applications on the network edges. This may not be the most opportune design charasteristic.

References

GNU04 GNUnet, Website of the GNUnet project. URL `http://www.ovmj.org/GNUnet/`. 2004.

MM02 Maymounkov, P. and Mazières, D., Kademlia: A peer-to-peer information system based on the xor metric. *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*.

RSS+04 Räihä, M., Saarelma, R., Salonen, A., Toivonen, T., Tukiainen, T. and Viitanen, S., The Distributed Hash Table project: Archive of project documentation and source code. URL `http://www.cs.helsinki.fi/group/dht/`. 2004.