Manuel Schultheiß
Stefan Smarzly

Technische Universität München

Final Project Report

DHT Subproject for P2PSEC (IN2194) in SoSe 2015
September 6, 2015

Contents

1 Introduction

In this paper we present a Python based implementation of the distributed hash table Chord. It is designed as a public key storage for a peer-to-peer VoIP application.

2 Existing work

Our work is mainly based on three papers. First, there is the original paper from Stoica et al[2] which introduces the key aspects of Chord. This is the base for our implementation.

Then, there is the paper from Pamela Zave[1] which describes advanced methods for stabilization in a Chord network. It mainly suggests a way how to properly maintain additional successor lists of length r. A successor list is a list of immediate successors of a node according to its identifier address. If the currently selected successor fails, the next successor in the list allows to keep the Chord ring closed. This algorithm is crucial in an unsteady network with nodes joining and leaving within short amount of time. In our implementation, we set $r := 3$. In scenarios with a lot of nodes and high fluctuation, it is recommend to increase this value.

For redundant storage, we rely on the work of Waldvogel et al [3]. Waldvogel et al suggest generating replicas by mapping the key k to multiple locations. For replication the hash function h is defined by:

$$h(k, d) = H_{Sha256}(k||d)$$

d is the index of the replication. So if the replication degree is 3, we evaluate $h(k, 0)$, $h(k, 1)$ and $h(k, 2)$. The results of h(k, d) are the keys under which the content will be stored in the distributed hash table. However, when using small integers as key k, with the above function collisions can appear easily.

$$H_{Sha256}(11||1) = H_{Sha256}(111) = H_{Sha256}(1||11)$$

This type of collision may not occur when using only public keys as DHT keys, but as we wanted to keep the DHT able to save integer keys, we decided to eliminate this problem. Therefore a modification to the proposed hash function was applied. The new hash function is called d-times recursively:

$$h(k, d) = \begin{cases} H_{Sha256}(h(k, d-1)) & \text{if } d > 0 \\ H_{Sha256}(k) & \text{if } d \equiv 0. \end{cases}$$

3 Documentation

3.1 Dependencies

The project was tested on Fedora 21 and XUbuntu 14.04, but it should run on other distributions as well.

3.1.1 Setup on Fedora/RHEL/CentOS

```
yum install python3-pip
```

3.1.2 Setup on Debian/Ubuntu

```
apt-get install python3-pip
```

3.1.3 Python Libraries

On all systems, the following libraries have to be installed:

```
pip3 install jsonschema
pip3 install aiomas
```

3.2 Install and run Chord

3.2.1 Simple testing

We provided a test script for the GNOME and XFCE desktop environment. It does not need mininet, as the node addresses are mapped to different ports on the local machine. To start the test use either

```
gnome_start.sh
```

on a GNOME environment or

```
xfce_start.sh
```

on XFCE. You can add 3 nodes by calling

```
gnome_add3.sh
```

or

```
xfce_add3.sh
```

This command only works once.

Afterwards you can start 'dhtQuery.py' to add content to the DHT. You can store and lookup by an integer key here. Just follow the instructions. The Python script dhtQuery.py provides a console interface to store and retrieve data with an integer key.

3.2.2 Adding custom nodes

You can run nodes with custom properties using the console. For example:

```
python3 main.py -i 1333 -b 1337 -B 127.0.0.1
```

runs a node on port 1333 with bootstrap node `127.0.0.1/1337`. The parameters are:

Name	Type
-i	The node port
-I	The node IP address
-B	Bootstrap Node IP
-b	Bootstrap Node Port
-h	Path to Hostkey .pem file. Used to generate custom node id
-c	Path to config file. The above parameters will override the config properties

3.3 Configuration files

As an alternative to parameters you can use configuration files. The properties are described in figure 1.

Table 1: The properties of a config file.

Section	Field	Description	Example Value
No Section	HOSTKEY	Path to PEM keyfile	/home/test/key.pem
	LOG	Logfile	/home/test/test.log
	PORT	Port on local machine	1337
KX	PORT	Port for the KX module	20000
DHT	OVERLAY_HOSTNAME	Boostrap node hostname	bootstrapnode.com
	PORT	Port for the API	1234
	HOSTNAME	Own IP address	127.0.0.1
BOOTSTRAP	PORT	BOOTSRAP PORT	1234

3.3.1 Mininet

As the native virtual machine provided by mininet was terribly slow, we used a XUbuntu installation within a virtual machine (tested with Gnome Boxes on Fedora 21 and Virtual Box on Windows 10). To install mininet follow the instructions on `http://mininet.org/download/`. For Ubuntu you can use `apt-get` to install it. Furthermore you need to install the required packages as described above in section 3.1.2 and 3.1.3. For testing, run the Python script *'sudo python startMininet.sh'*. You may want to call *'sudo mn'* if there appear any errors regarding used ports (Enter *'quit'* to stop it afterwards). Also make sure you provide the right config files. For the 5 default nodes generated by the python script these are located in the `mnconfig/` folder. **You need to provide the right absolute path for the HOSTKEY here!**

3.4 Software Architecture

The application is started with *main.py*. It initializes a new node with parameters from console or config files. The logic for such a node is located in *Node.py* and its corresponding class *Node*. The *Node* class is responsible for remote procedure calls (RPC) and handles most of the logic such as stabilization, joins and look-ups in the DHT.

3.4.1 Helpers

We have separated certain tasks from the main *Node* class. The most important ones are *replica*, *storage* and *messageParser*. They are all located in the `helpers/` folder.

3.4.2 Validation

The validation of RPC calls was implemented by using the Python JSONSchema (`https://pypi.python.org/pypi/jsonschema`) Validator. This validator allows to create schemes defined by the JSON Schema standard (`http://json-schema.org/`). It defines how a

JSON object must look like. If there are errors such as missing attributes, an error message is returned. The schemes for our RPC calls are located in `helpers/validator.py`. The schemes ensure, that objects are passed with the correct type, length and name.

3.4.3 Unit Tests

To detect possible bugs as early as possible we use the integrated unit test framework of Python. If there is a file named *storage.py* the corresponding unit test is named *test_storage.py* for example. All unittests are referenced in *unittester.py*, from which you can run all unit tests at once. To do so, type `./unittester.py` into the console.

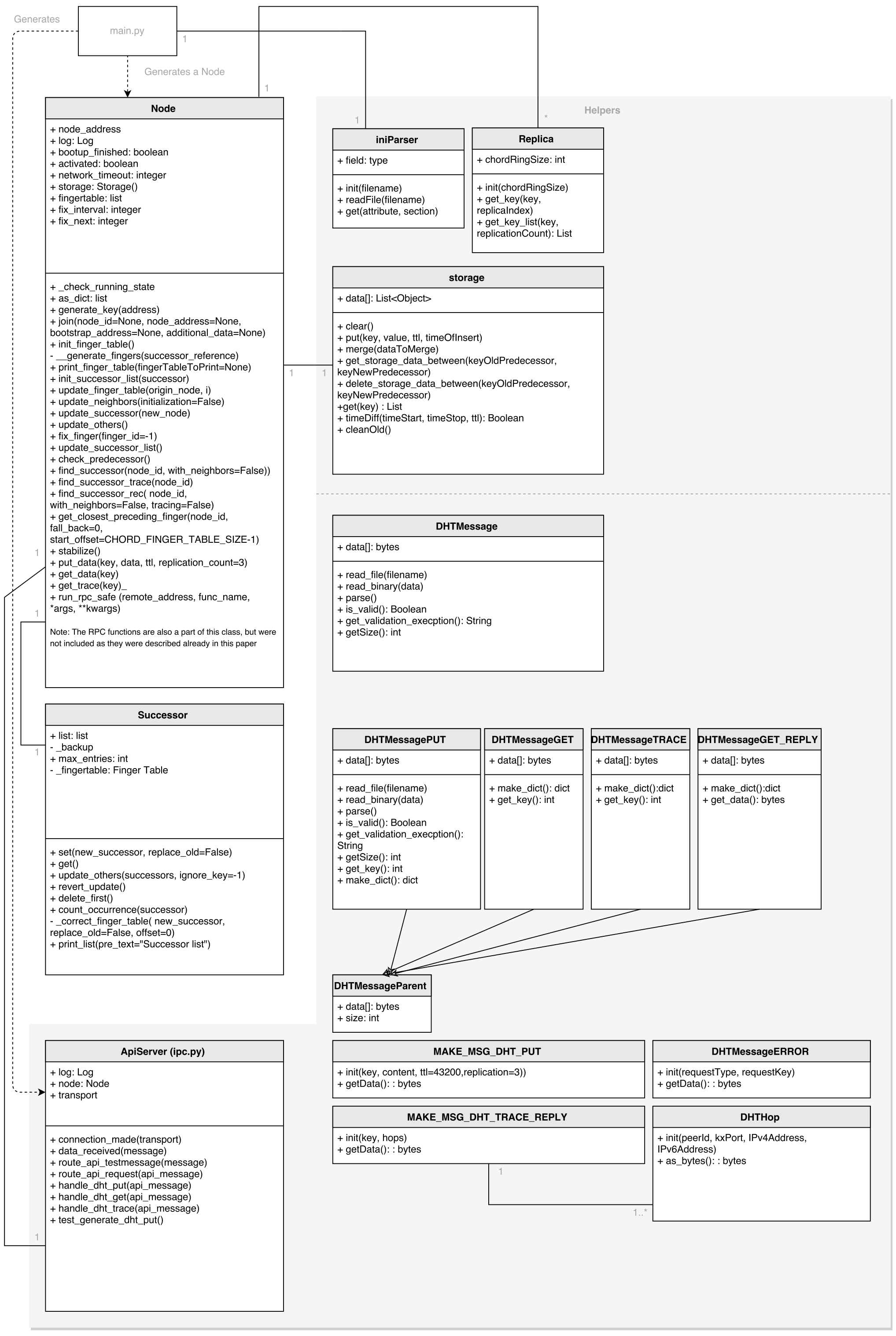
3.4.4 Class Documentation

To document the different classes, we used *sphinx*. You can find a pre built version in `docs/api` (Not the `docs` folder in the code directory!). It is used to generate the documentation right out of the source code and export it as HTML. To install *sphinx*, please refer to `http://sphinx-doc.org/latest/install.html`. With apt-get, type:

```
apt-get install python-sphinx
pip3 install sphinx
```

To generate the documentation, switch to the `code/doc/` directory and type `make html` in the console. The resulting web page is located in `code/doc/_build/html` afterwards.

3.4.5 UML Diagram



Generates
main.py
Generates a Node
1
1
1
*
Helpers
Node
+ node_address
+ log: Log
+ bootup_finished: boolean
+ activated: boolean
+ network_timeout: integer
+ storage: Storage()
+ fingertable: list
+ fix_interval: integer
+ fix_next: integer
+ _check_running_state
+ as_dict: list
+ generate_key(address)
+ join(node_id=None, node_address=None, bootstrap_address=None, additional_data=None)
+ init_finger_table()
- __generate_fingers(successor_reference)
+ print_finger_table(fingerTableToPrint=None)
+ init_successor_list(successor)
+ update_finger_table(origin_node, i)
+ update_neighbors(initialization=False)
+ update_successor(new_node)
+ update_others()
+ fix_finger(finger_id=-1)
+ update_successor_list()
+ check_predecessor()
+ find_successor(node_id, with_neighbors=False))
+ find_successor_trace(node_id)
+ find_successor_rec(node_id, with_neighbors=False, tracing=False)
+ get_closest_preceding_finger(node_id, fall_back=0, start_offset=CHORD_FINGER_TABLE_SIZE-1)
+ stabilize()
+ put_data(key, data, ttl, replication_count=3)
+ get_data(key)
+ get_trace(key)_
+ run_rpc_safe (remote_address, func_name, *args, **kwargs)
Note: The RPC functions are also a part of this class, but were not included as they were described already in this paper
iniParser
+ field: type
+ init(filename)
+ readFile(filename)
+ get(attribute, section)
Replica
+ chordRingSize: int
+ init(chordRingSize)
+ get_key(key, replicaIndex)
+ get_key_list(key, replicationCount): List
storage
+ data[]: List<Object>
+ clear()
+ put(key, value, ttl, timeOfInsert)
+ merge(dataToMerge)
+ get_storage_data_between(keyOldPredecessor, keyNewPredecessor)
+ delete_storage_data_between(keyOldPredecessor, keyNewPredecessor)
+get(key) : List
+ timeDiff(timeStart, timeStop, ttl): Boolean
+ cleanOld()
1
1
DHTMessage
+ data[]: bytes
+ read_file(filename)
+ read_binary(data)
+ parse()
+ is_valid(): Boolean
+ get_validation_execption(): String
+ getSize(): int
1
1
Successor
+ list: list
- _backup
+ max_entries: int
- _fingertable: Finger Table
+ set(new_successor, replace_old=False)
+ get()
+ update_others(successors, ignore_key=-1)
+ revert_update()
+ delete_first()
+ count_occurrence(successor)
- _correct_finger_table(new_successor, replace_old=False, offset=0)
+ print_list(pre_text="Successor list")
1
DHTMessagePUT
+ data[]: bytes
+ read_file(filename)
+ read_binary(data)
+ parse()
+ is_valid(): Boolean
+ get_validation_execption(): String
+ getSize(): int
+ get_key(): int
+ make_dict(): dict
DHTMessageGET
+ data[]: bytes
+ make_dict(): dict
+ get_key(): int
DHTMessageTRACE
+ data[]: bytes
+ make_dict():dict
+ get_key(): int
DHTMessageGET_REPLY
+ data[]: bytes
+ make_dict():dict
+ get_data(): bytes
DHTMessageParent
+ data[]: bytes
+ size: int
ApiServer (ipc.py)
+ log: Log
+ node: Node
+ transport
+ connection_made(transport)
+ data_received(message)
+ route_api_testmessage(message)
+ route_api_request(api_message)
+ handle_dht_put(api_message)
+ handle_dht_get(api_message)
+ handle_dht_trace(api_message)
+ test_generate_dht_put()
1
MAKE_MSG_DHT_PUT
+ init(key, content, ttl=43200,replication=3))
+ getData(): : bytes
DHTMessageERROR
+ init(requestType, requestKey)
+ getData(): : bytes
MAKE_MSG_DHT_TRACE_REPLY
+ init(key, hops)
+ getData(): : bytes
DHTHop
+ init(peerId, kxPort, IPv4Address, IPv6Address)
+ as_bytes(): : bytes
1
1..*

3.5 Security and Stability

3.5.1 Churn

Our implementation can handle churn with trying to minimize the introduced latency. Therefore we use a technique called reactive recovery and periodic recovery as compared by Rhea et al. [4] to each other.

Reactive recovery is realized by using a successor list for each node. A node tries to find a replacement neighbor, which is the next item in the successor list, once it realizes its immediate neighbor has failed. Also during a look-up procedure (`find_successor()`), an alternative finger is chosen automatically as fallback if the node from the intended reference is not alive anymore. As at high churn rates rates, reactive recovery is inefficient regarding bandwidth, another technique used is periodic recovery. We call the `stabilize()` (in *Node.py*) routine in a fixed time interval here. The finger table, successor lists and predecessor is checked and repaired, if necessary.

Another technique would have been to use super-peers, which are nodes that are always online. But this does not go along with the distributed architecture of the project.

3.5.2 DDoS/DoS

Although (distributed) Denial of Service attacks should be handled by firewalls, increasing the replication level decreases the vulnerability against such attacks. Like this, the loss of single nodes does normally not destroy the occurrence of a certain data item. Some more ideas are mentioned in section 5.

3.5.3 Authentication

As section 4 explains in more detail, we use Remote Procedure Calls provided by the library *aiomas*.

Every incoming request that is not a RPC, is ignored. By invoking a valid RPC to a remote node, a valid answer shows that it is a node of our DHT implementation. However, by itself, this does not provide any protection against modification of data by attackers.

Unintentional data corruption should be prevented by lower protocol layers. An additional hash of the data would not provide any security related improvement without a shared secret. However, the JSON fields are validated in terms of existence, length and data type.

4 Protocol Design

4.1 Original and new Approach

Our approach for communicating between multiple Chord instances changed considerably during the process of development. Initially, we started creating TCP server and clients based on Python's `asyncio` library which is a part of the standard library since version 3.4. For message serialization, deploying `JSON` was a valuable choice due to its human readability while debugging the communication between several nodes.

However, with more and more functionality being introduced, the communication part required to be completely abstracted into an independent module to maintain a structured and clear code base. We found the quite new library `aiomas`[1]. It provides an interface for remote procedure calls (*RPC*). In contrast to other modules, it is based on the event loop `asyncio` (the one we also utilized). Initial tries failed due to some documentation issues. Once working, we switched to this module for inter module communication as it's mostly a bad idea to reinvent the wheel. Albeit, we have some ideas how to improve the library as stated later.

4.2 Inter Node Communication

To exchange data with other nodes, RPCs allows to call functions on remote nodes as it would be local function calls. With *aiomas*, the elementary structure for RPC messages is represented by the following triplet:

type	id	payload

The message identifier *id* is an unique number (32 bit) incremented for each outgoing request made. Due to the asynchronous processing in *asyncio*, incoming TCP packets might not be related to the request sent before. Therefore, the message identifier associates outgoing requests to recognize the corresponding response. If a matching response is found, *aiomas* informs the future object linked to the blocking instruction. A message can have three different *types* (represented as *int*):

- REQUEST = 0
- RESULT = 1
- EXCEPTION = 2

Type *REQUEST* is for outgoing requests to other nodes. Type *RESULT* marks a response for a previously sent request. Type *EXCEPTION* contains a trace of the call stack if an exception is raised. We focus on the first two types. The last field *payload* embodies the actual data sent between the nodes. The data is serialized by Python's default JSON module. The binary representation is an UTF-8 encoding of the serialized JSON message.

Depending on the message type, the payload is different:

- **REQUEST** — For a request, payload contains the triplet `(path, args, kwargs)`. *path* specifies the function executed with the arguments *args* and *kwargs* in the RPC.

[1] `https://bitbucket.org/ssc/aiomas/overview`

- **RESULT** — A result message transports the return data of a RPC. Depending on the function, the underlying type can vary. In almost all scenarios, we return a Python dictionary (*dict*). It bundles several options accessible by a key string. For more details, refer to the individual RPC functions below.

4.2.1 rpc_dht_put_data

This call saves data on another node, given the key is managed by the contacted remote node.
Parameter Input:

bigint	key	The key
int	ttl	The time to live (seconds)
bytes	data	Arbitrary bytes

Replication values are not transferred as part of this message. Instead, the corresponding node invokes this RPC several times on different nodes to distribute the data. Section 2 briefly describes the mechanism.

Notice:

- *bigint* is also represented by type symbol *int* in Python. The intend is that this integer is not restricted to 32 or 64 bit.
- As bytes cannot be directly represented in JSON, *bytes* fields are replaced by UTF-8 strings containing the data encoded by *base64*.

Every notice also holds for all other RPCs.

Return Data:

int	status	The status. 0 for success, otherwise error code > 0.
string	message	(Optional) Descriptive reason when something went wrong.

4.2.2 rpc_dht_get_data

Fetches the stored data on a certain node given the key.
Parameter Input:

bigint	key	The key

Return Data:

int	status	The status. 0 for success, otherwise error code > 0
[]bytes	data	All data associated with the key (list)

If no data is stored for a particular key, an empty list is returned.

4.2.3 rpc_get_node_info

Returns information about the node and its environment, such as successors and predecessor.

Parameter Input:

bool	successor_list = False	Return the list of successor
bool	additional_data = False	Return additional data

Return Data:

—	Information about the node and its immediate neighboring nodes (see 4.2.4)	
[]dict	successor_list	(If requested) List of available immediate successors
dict	additional_data	(If requested) Additional data initially supplied to this Chord instance, e.g., the KX port

Notice: In all cases, a node is represented by its node identifier *node_id* and its network address *node_address* bundled by a dictionary. The address string is built according to *aioma's* agent terminology, e.g., `tcp://127.0.0.1:5555/0`. The trailing number is actually not needed in the implementation. It allows to spawn several nodes on the same host's port.

4.2.4 Node Info

This section only describes a part of a RPC referencing it. The following fields store information about a node and its environment.

bigint	node_id	Node ID
string	node_address	Network address (IPv4 or IPv6)
dict	successor	Successor node.
dict	predecessor	Predecessor node.

A node is represented by its node identifier *node_id* and its network address *node_address* bundled by a dictionary. This is the information stored in *successor* and *predecessor*.

4.2.5 rpc_find_successor_rec

This function retrieves information about the node responsible for the given key. It employs a recursive mechanism to shorten the distance to the target node along a path of intermediate nodes.

Parameter Input:

int	node_id	Node ID (key) whose responsible successor is interesting
bool	with_neighbors	Adds information about neighboring node of the target node
bool	tracing	Activates trace messages, i.e., recording the responsible node on each hop

Adding information about the target node's neighbors is useful if its immediate predecessor is interesting. This can be done with the *with_neighbors* option. The option *tracing* enables further information about immediate nodes along the route to the target node.

Return Data:

—	Information about the found node responsible for node_id (see 4.2.4)	
dict	additional_data	(Tracing requested) Additional data provided by the successor hop
[]dict	trace	(Tracing requested) List with visited nodes on the route to the target so far. Inverse creation of the list starts once the goal is reached.

The return message needs some explanation. It is returned along the same path of nodes which was set up during the recursive calls of this function to find the responsible node for the given key (*node_id*). If the responsible node can be located and is alive, information about this node is forwarded back to the initial requester. If not, a *status* field informs about the error albeit a descriptive error *message*.

If the requesting node activates trace logging (*tracing*), two further fields are part of the return data. The trace log is stored as a list in the *trace* field. One special property about list assembly is that it is done while forwarding the response of the last node (hop) along the established path. Consequently, the last hop is the first entry in the list. Each list entry contains the following fields:

***trace* list entry**	bigint	node_id	Node ID
	string	node_address	Network address (IPv4 or IPv6)
	dict	additional_data	Additional data, e.g., KX port

When analyzing the protocol, there is another characteristic: the entry of a certain node is always added by its predecessor in the call hierarchy. This guarantees that a node cannot pretend to be a different one, for instance. Of course, an attacker node can invalidate the entries in the accumulated trace log. However, its own entry is a valid one if it has gotten an honest predecessor.

4.2.6 rpc_update_finger_table

This function allows a remote peer to update the finger table entries of its predecessors that should link to this node. It is part of the **reactive update**. This is useful for a

fast stabilization of the base ring if a new node joins the network.
Parameter Input:

dict	origin_node	node_id and node_address of the node the initial request originates from
int	i	Index of finger entry which should be replaced with origin_node

Return Data:

int	status	The status. Always 0 in the current implementation.

Although the status is 0, it does not mean necessarily that the update request was accepted. Wrong entries are also corrected during the periodic updates.

4.2.7 rpc_update_successor

This function allows a new node to speed up its integration between two neighboring nodes in the Chord ring. It is part of the **reactive update**. It provides a hint to its predecessor that the immediate successor changed. In our implementation, the informed node will ask its current successor first about its predecessor view. If it links to this new node, it will be accepted by the node as valid immediate successor that replaces the old entry.
Parameter Input:

dict	node_hint	A node which might be eligible as a new successor

Return Data: None

There is no return data, because it is just a hint to accelerate the stabilization. It is up to the recipient whether it updates its first finger (immediate successor) to the suggested node. Otherwise, the periodic checks will fix it after some time.

4.2.8 rpc_update_predecessor

Calling this function on a remote peer allows to update its predecessor reference to the caller node. It is called as part of the **reactive update** and **periodic update**.
Parameter Input:

dict	remote_node	A node which might be eligible as a new predecessor

Return Data:

bigint	node_id	Node ID
string	node_address	Network address (IPv4 or IPv6)
dict	old_predecessor	(Optional) Information about the predecessor before the update
[]bytes	storage	(Optional) List of DHT keys with associated byte strings the new predecessor is responsible for

If the update succeeds, *node_id* and *node_address* contain the information of the requesting node. In this case, the other two fields are present. *old_predecessor* presents information about the original predecessor. This is useful for a reactive update: the joining node directly informs its predecessor that it got a new immediate successor. This allows to accelerate the stabilization. As this update might cause a race condition for concurrent access, the periodic update establishes a correct placement in the Chord ring if two nodes tried to be the predecessor of a certain node.

storage contains a list of data (keys with associated data, ttl, etc.) the current node is responsible now. The node stores these data items and returns it on request.

4.3 Application Programming Interface

For local communication with other modules of the overall project we use the binary protocol as specified in the course documentation. You can test PUT and GET methods with ./dhtQuery.py.

4.3.1 DHT PUT

For DHT PUT, the following fields are needed:

1. **size:** The total size of the message in bit
2. **id:** 500 for MSG_DHT_PUT
3. **key:** The key under wich the content is stored
4. **TTL:** Time to live
5. **replication:** Degree of replication
6. **content:** The content

Integer fields (size, TTL, replication) are encoded in big endian byteorder.

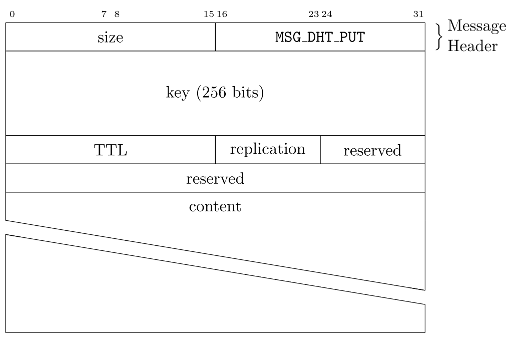


Figure 1: DHT PUT Scheme

4.3.2 DHT GET

For DHT PUT, the following fields are needed:

1. **size:** The total size of the message in bit
2. **id:** 501 for MSG_DHT_GET
3. **key:** The key under wich the content is stored

Integer fields (key and size) are encoded in big endian byteorder.

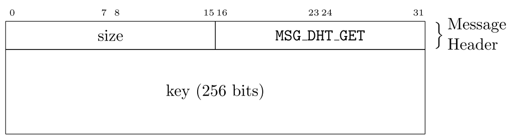


Figure 2: DHT GET Scheme

4.3.3 DHT GET REPLY

For DHT PUT, the following fields are needed:

1. **size:** The total size of the message in bit
2. **id:** 502 for MSG_DHT_GET_REPLY
3. **key:** The key under wich the content is stored
4. **content:** The content

Integer fields (key and size) are encoded in big endian byteorder.

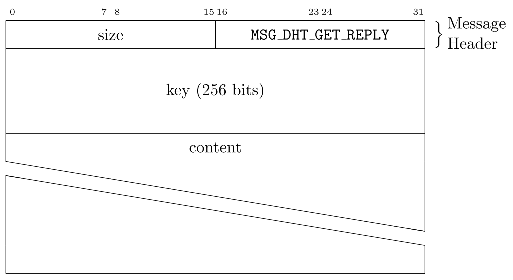


Figure 3: DHT GET Scheme

4.3.4 DHT TRACE

For DHT PUT, the following fields are needed:

1. **size:** The total size of the message in bit
2. **id:** 501 for MSG_DHT_TRACE
3. **key:** The key under wich the content is stored

Integer fields (key and size) are encoded in big endian byteorder.

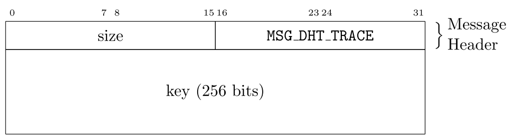


Figure 4: DHT TRACE Scheme

4.3.5 DHT TRACE REPLY

For a trace reply message, the following fields are needed:

1. **size:** The total size of the message in bit
2. **id:** 501 for MSG_DHT_TRACE
3. **key:** The key under wich the content is stored

Now for each hop, there is additional information added:

1. **KX Port**
2. **IPv4 Address**
3. **IPv6 Address**

The IP address is encoded in Python with

```
ipaddress.ip_address(strIP).packed
```

where *strIP* is the IP address in string format. Integer fields (key, KX Port and size) are encoded in big endian byte-order.

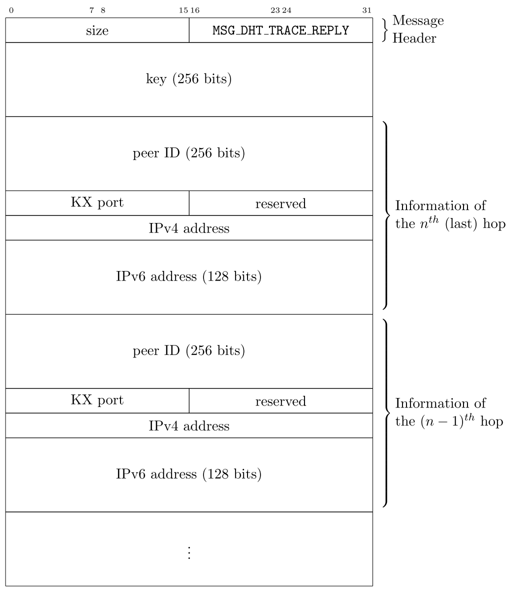


Figure 5: DHT TRACE REPLY Scheme

5 Future Work

Future work could introduce more security features. To add another level of protection against DDoS attacks, we could remember the IP addresses of the origin servers to limit their number of allowed requests. DoS attacks are quite a problem in the DHT, as functions like `find_successor` are executed recursively. This means one request to a Chord node usually affects at least 1, but up to $O(logn)$ nodes in the network. An attacker can cause serious damage using only a few resources this way. Another useful feature would be to detect high churn rates automatically to switch between reactive and periodic recovery.

6 Work Justification

As in the beginning we worked together on one computer, where one person was coding and the other person observing, towards the end, we have split up the work a bit more. Stefan cared more about the Chord logic itself, where Manuel focused on the helper classes (for replication, storage etc.) and the API. The overall effort spent on the project, was quite high compared to other courses.

References

[1] Pamela Zave, How To Make Chord Correct, *http://www2.research.att.com/~pamela/zave_podc.pdf*, February 2014

[2] Ion Stoica et al, Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications, *http://pdos.csail.mit.edu/papers/ton:chord/paper-ton.pdf*, IEEE/ACM TRANSACTIONS ON NETWORKING, VOL. 11, NO. 1, FEBRUARY 2003

[3] Waldvogel et al, Dynamic Replica Management in Distributed Hash Tables, *http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.2.5845&rep=rep1&type=pdf*,

[4] Rhea et al, Handling Churn in a DHT, *https://www.usenix.org/legacy/publications/library/proceedings/usenix04/tech/general/rhea/rhea_html/usenix-cr.html*,

[5] Keith Needels and Minseok Kwon, Secure Routing in Peer-to-Peer Distributed Hash Tables, SAC'09 March 8-12, 2009, *http://www.cs.rit.edu/~jmk/papers/secchord.pdf*,

7 Notation

|| Concatenation