A REPORT ON A KADEMLIA IMPLEMENTATION ATTEMPT

KADEMLIA

February 27, 2019

CMPT 434: Computer Networks

Derek Eager

Christopher Mykota-Reid
Rowan MacLachlan

Contents

0.1 Introduction 3
0.2 Objectives 3
0.3 Implementation 4
0.3.1 Use 4
0.3.2 High-Level Overview and Tooling 4
0.3.3 Kademlia Remote Procedure Calls 8
0.4 Measurements and Criteria 8
0.5 Technical Difficulties 8

0.1 Introduction

DHTs are similar in principle to a regular hash table in that they provide the structure for fast retrieval of an object by referencing the location of the object's hash value. However, a distributed hash table is stored across nodes in a network, and this introduces a variety of complications involving node-lookup (finding the location of the stored data from its hash in the network), ensuring data redundancy (what happens if a node goes offline?), and data retrieval. Although this technology is not *new* by any standard, its continued use and development in various fields makes it a suitable candidate for an implementation project.

Kademlia provided an excellent learning opportunity with respect to practical network-related computer skills, the application of some course theory (implementing application-layer communication across different networks over UDP implies some difficulties with port-forwarding at the route — this is what NAT would help with,) and much deeper understanding of the Kademlia protocol as well as an understanding of some of the issues overlay networks (DHTs in particular) must deal with.

0.2 Objectives

As referenced above, the problems presented by a DHT implementation are myriad and often difficult to solve, but by no means impossible. By implementing an incremental plan to gradually roll out more complex features, taking full advantage of existing libraries, and using a high-level and expressive programming language, we managed to implement a DHT. While this DHT does not implement every particular directive of the Kademlia white paper exactly (nor is every successfully implemented feature necessarily bug-free), we managed to create an application that could be easily run and could, without issue, support a small network of up to 8 devices.

On one level, the implementation team is very happy with this result, considering the considerable hours invested in research, implementation, and testing. Although we failed to produce a working proto-type with enough left-over time to run large-scale tests (100's or more nodes), and so could not collect meaningful data on DHT performance, we feel as though the preliminary testing and class demo shows at least some acceptable level of achievement. While this may be dissappointing, it must be noted that many excellent Kademlia implementations already exist, most of which feature more finely-tuned, bugless, and efficient code. In this sense, simulation data would only have cemented an already clear understanding of our implementations shortcomings. Implementation decisions were almost always made by giving far more weight to deadlines than to writing performant code.

More importantly, we exceeded our personal academic goals — with respect to interest and learning, the project was a complete success.

0.3 Implementation

0.3.1 Use

Using the program is relatively simple. The project folder, once downloaded (either provided by the students or downloaded from the GIT repository here) has a simple structure. The source code is in the `kademlia` folder. So long as the user has a python3.7 (or greater) environment, everything should work.

- `git clone https://github.com/rowan-maclachlan/cmpt-434-proj`
- `cd cmpt-434-proj`
- `make init`. This will install the application dependencies.
- `python3 kad.py`. This will launch a single Kademlia node. To see application use, run the program without arguments.
- Use command `set`, `get`, `ping`, and `inspect` to interact with the DHT.

Please view the README for further details.

0.3.2 High-Level Overview and Tooling

RPC library

The Kademlia paper implies that the protocol is implemented with datagrams[1]. Typically, clients operating behind a NAT (Network Address Translator) do not need to worry about querying servers outside the NAT, as the Network Address Translator maintains a table correlating incoming messages to hosts on its network. However, on peer-to-peer systems, clients must also act as servers, and accept requests from other nodes on the overlay network despite being located behind a NAT. This requires the use of a NAT traversal technique such as Hole-Punching or SOCKS (Socket Secure) to overcome the difficulties inherent in routing requests through NATs. This makes application use on LANs behind a router address difficult without port-forwarding and configuration overhead. Because of this, our application was tested only on a LAN.

Our primary investigation showed other Kademlia implementations using the core python library asyncio to manage callback procedures for asyncronous code execution. While the semantics of asynchronous RPC might make implementation *more* difficult than for a simple synchronous I/O implementation (the python socket module directly, for example), implementing asynchronous UDP RCP with asyncio is very simple. In fact, we used a 3rd party library called rpcudp which overlays functionality for remote procedure calls onto a datagram communication protocol - all asynchronously![2]

As described on the asyncio Read-The-Docs page: "asyncio is often a perfect fit for IO-bound and high-level structured network code." This is exactly the Kademlia use-case.[3]

Hashing

Kademlia specifies the use of the 160-bit SHA-1 hash for data, and 160-bit node IDs and 160 `k-buckets` 0.3.2, to accompany that. This parameter of ID and hash length, however, is not mandated. The python library hashlib provides an array of hashing functions, cryptographic and otherwise, including the SHA-1 hash. This library can be used behind a small wrapper, and hash/ID size can be trimmed to the length specified in the simulation parameters. This value could initially be set quite small to increase ease-of-testing and provide more comprehensible and tractable output during development stages. It can easily be changed later.

```python
def hash_function(data):
    """
    Hash the data to a byte array of length p.params[B] / 8

    TODO: How do we truncate this large integer?  mod?  or mask?
        what would we mod by?

    data : binary data

    int
        A hash of length p.params[B] / 8 in hexadecimal string form.
    """

    return int(hashlib.sha1(data).hexdigest(), 16) & get_mask()
```

As shown here, the returned digest is AND'ed with a mask — this mask is the length of the system-wide hash/ID bit length value — called 'B' in the Kademlia paper. This mask then truncates the sha1 hash of the data to a value between 0 and $2^b - 1$. This allows the value of 'B' to be effortlessly changed for more practical use of the network or for larger-scale simulations.

Routing Table Design

Study of different Kademlia implementations[4][5][6][7] reveal that most implementations implement their routing table in a way very similar to that described in the Kademlia white paper[1]. This involves creating nodes with only a single `k-bucket` 0.3.2. As the k-bucket reaches capacity, it is then split into two and its contents distributed among the two resultant buckets according to their proximity to the node that owns the routing table. In this sense, a routing table may not actually have as many `k-bucket`s in memory as there are bits in the keyspace. Naturally, this is far more space efficient than the implementation we opted for.

As the maximum number of `k-bucket`s on any particular node is simply the length of the ID space, they can all be created at once, as such:

```
self.buckets = [ KBucket(k) for _ in range(b) ]
```

Instead of attempting to insert nodes into a `k-bucket` which may or may not be full and which may result in further operations on the data structure, nodes can be inserted directly into the `k-bucket` corresponding to their distance from the routing table's owner, as such:

```
return self.get_bucket(contact.getId()).add(contact)
```

where the method `RoutingTable.get_bucket(id)` calculates the correct bucket index by referencing the most significant bit of the distance between the ID of the routing table's owner and the ID of the contact we are trying to add to the routing table, like so:

```
distance = self.id ^ id
index = 0
while distance > 1:
    # Count the index of the largest bit in the distance
    # The distance will be zero after 'bit' many shifts.
    distance = distance >> 1
    index += 1
return self.buckets[index]
```

Naturally, `k-bucket`s can be retrieved in a similar manner.

Although this approach is less space efficent than creating `k-bucket`s only as needed, the overhead is fixed and relatively small. Ultimately, this decision was made because it was believed to be simpler.

Bucket Design

The `k-bucket` is a container that holds contact information for other nodes on the Kademlia network. Each `k-bucket` is created with a maximum capacity of `k`, a system-wide parameter that controls for such things as `k-bucket` size. In our implementation, it is simply a light wrapper for a list into which contacts are stored.

As described by the white paper, `k-bucket`s, once full, need to make evaluations about whether new contacts are inserted in the `k-bucket` or not after it has reached capacity. When a new node is discovered but its `k-bucket` is full, it is only added if the node at the head of the list is unresponsive. Otherwise, the new node is simply discarded, and the responsive node is moved from the head to the tail of the list.

There are a variety of reasons for this behaviour. First, the Kademlia authors illustrate that nodes which have existed on the network for a long time are statistically more likely to continue being active on the network than new nodes. Therefore, the integrity of the network as a whole is improved by prioritizing these old nodes. Secondly, this behaviour makes the network automatically resistant to Denial-of-Service attacks: the network cannot be brought down by flooding it with new nodes, because the old network will not replace legitimate nodes that already exist in its routing table unless they become unresponsive.

Unfortunately, our implementation does not track node responsiveness, as the minimum-viable-product was simpler if new nodes are simply added (because they are known to be active) and old nodes are kicked out of the routing table.

```python
if contact in self.contacts:
    # If this contact already appears in the list, move it to the back
    # of the list
    self.contacts.remove(contact)
    self.contacts.append(contact)
elif not self.full():
    # If the contact doesn't appear in the list, append it.
    self.contacts.append(contact)
else:
    # If the list is full, remove the oldest contact before appending
    # the new one.
    self.contacts.popleft()
    self.contacts.append(contact)
return True
```

Because most operation on the `k-bucket` involve adding and removing nodes from the beginning and end of the list, a special-use data structure was employed to optimize the efficiency of these operations. Python provides a structure called a ‘deque’ (pronounced ‘deck’) that implements constant-time additions and removals from the head and tail of the list.

0.3.3 Kademlia Remote Procedure Calls

Store

Find Value

Find Node

Ping

Finding Nodes on the Network

0.4 Measurements and Criteria

As referenced in the implementation plan, there are quantitative measurements available when considering the success or failure of our implementation. Comparing these values to those achieved by more mature DHT implementations will provide valuable insight into the challenges posed by DHTs, and should be included in our final report. However, the most valuable metric will be the realization of a working MVP. This may be something as simple as the observed and correct distributed storage of one or more files on the personal computers of the researchers (Chris and Rowan) or a more glorious success - or, perhaps, the real success is the friends we made along the way.

0.5 Technical Difficulties

Here, we list some of the technical considerations and their resolutions:

- How many nodes store the data of a single object? What degree of redundancy can be achieved and what overhead does this add?

  The Kademlia paper invokes this concern, but does not provide any hard-numbers. According to the authors, this number must be high enough that even on very difficult networks, no more than this number of nodes in the same key-space go offline within an hour of each other. The authors provide a value of 20, but we tested with a far smaller value so that correct behaviour could be more easily observed.

- Is an entire object stored on each node, or do we distribute a single object in incomplete parts through the network?

  Our implementation accepted only `String` types as keys and values. Regardless of the message type (node list, node id, key, or value) the entire message is always transmitted in a single UDP packet. This limits message size to a theoretical value of 65,507 bytes minus the overhead incurred by the rpcudp library, but our implementation did not account for messages sizes in any way. Presumably, the user could crash their node by trying to store a massive value.

- How large is the address space: how large are the hash keys? 128 bits? 160 bits? We may not need to have a large address space for our implementation, but just what are the consequences of this choice?

  As described in 0.3.2, key-space size was easily adjustable because of system design. However, the decision was made during implementation to keep the value small. This made it easy to manually verify correct network behaviour by calculating distances by hand, as well as put more stress onto some system internals such as 0.3.2 and 0.3.2 which behave differently depending on the density of nodes around the keyspace.

- Do we hash the object's identifier or the object data? If we hash the identifier, how do we enforce unique identifiers?

  Unique identifiers (keys) are not enforced. If some aspect of uniqueness was taken from the node attempting the store, their ID or IP/port could be used to create a unique hash. However, This would make it difficult for other nodes to request that data (as the hash was created with some other node's identifying information.) A better solution would be to request the data mapped to by the hash-key before attempting the store, and only allow the store if the data was not found. While this solution is not technically difficult, it was deemed unimportant in relation to other features.

- Any hash function will have collisions. How do we manage collisions? Do we chain hash contents at the site of storage or do we dissallow colliding files?

  Support for chaining hash-collisions was not implemented. Data storage was managed simply by Python's built-in dictionary type. If there was a collision (highly possible on a test-network with a smaller than usual key-space) the previous value would be overwritten at the node performing the store. This could be avoided by the use of a more powerful container type, but it would introduce other questions and corresponding technical issues.

  When requesting a value from a key which maps to a node containing chained occurences of that key, which value should be returned? There's no way for the node recieving the request to know which value the node making the request is after, so should it return both values? This would throw a wrench in the current implementation of the `get` command, which does not expect more than a single value response.

- How does the *CAP* theorem (or *Brewer* Theorem) inform our implementation? What does it mean for it to be impossible for a distributed system to provide all of Consistency, Availability, and Partition Tolerance?

  As described by the theorem, providing both consistency and availability on well-managed internal networks (our test-case) is generally not an issue. Even so, the Kademlia protocol does an excellent job at specifying the responsibilities of nodes to keep their routing tables up-to-date and ensure that the key-values they are responsible for exist with a high degree of redundancy. In the event of highly-congested networks or networks with partitions, the Kademlia protocol (with some small changes to our existing implementation) will naturally route around high-latency links. This is achieved by the continuous re-ordering of active and responsive nodes in the routing table, and the option of storing contacts within k-buckets ordered on round-trip-time, which can be easily measured.

- How did we manage the issues presented by key-space remapping?

  Is it possible for us to avoid the issue of key-space re-mapping (changing the node location of data) when adding or removing nodes from the network?

  Thankfully, this is handled very nicely by the Kademlia protocol. The first issue it resolves is re-mapping existing data to a new node, and the second issue is caused by nodes leaving the network. Image a node $n1$ joining the Kademlia network. It does this by pinging another node on the network that it knows about, $n2$. When $n2$ hears about $n1$ (and $n1$'s ID), $n2$ issues `store` requests to $n1$ for every piece of data $n2$ has in its routing table for which the following conditions are met:

  – $n1$'s ID is closer to the data (by XOR) than the furthest among our k-many closest already known contacts to the data.

  – We are closest to the data of any of our already known contacts.

  This ensures that new nodes are holding data they are responsible for, and it also minizes the number of extraneous stores being sent over the network. This is perhaps difficult to understand but section 2.5 of the Kademlia paper describes it well. Or, consider the code below:

```
def handle_node(self, contact):
...
    # See Kademlia paper section 2.5 on how to incorporate new nodes.
    # We may have to store all values we have which are closer to the
    # new node than they are to us.
    for key, value in self.data.items():
        # find neighbours close to the key value
        nearest_contacts = self.table.find_nearest_neighbours(key)
        # If there are fewer than k neighbours, store the key-value to the
        # new node
        if len(nearest_contacts) < self.table.k:
            log.debug(f"Few contacts, storing data to new contact...")
            # schedule the task in the event loop, continue to next data
            asyncio.create_task(self.try_store_value(contact, key, value))
```

```
                continue
            # If there are k neighbours, only store the key−value if the
                new
            # node is closer to the key the our neighbour furthest from
               the
            # key, and if we are closer to the new node than any of our
            # neighbours.
            nearest_contact = nearest_contacts[0]
            # are we nearer to this key than nearest_contact is?
            were_nearest = \
                    self.this_node.distance(contact.getId()) < \
                    nearest_contact.distance(contact.getId())
            furthest_contact = nearest_contacts[−1]
            # Is contact closer to the key than the furthest contact?
            close_enough_to_store = \
                    contact.distance(key) < furthest_contact.distance(
                       key)
            if close_enough_to_store and were_nearest:
                log.debug(f"Storing {data} to new contact...")
                asyncio.create_task(self.try_store_value(contact, key,
                   value))

    self.table.add(contact)

    return True
```

The second issue, that of nodes leaving the network, is handled by their being a degree of redundancy for data. Data are replicated by a factor of *k* on the network, and so a single node leaving has no practical effect.

Please see the bibliography for additional reading.

Bibliography

[1] Petar Maymounkov and David Mazieres. “Kademlia: A peer-to-peer information system based on the xor metric”. In: *International Workshop on Peer-to-Peer Systems*. Springer. 2002, pp. 53–65.

[2] Brian Muller. *RPC over UDP Python module*. URL: `https://github.com/bmuller/rpcudp` (visited on 04/02/2019).

[3] *Python asyncio module documentation*. URL: `https://docs.python.org/3/library/asyncio.html` (visited on 04/02/2019).

[4] Brian Muller. *Kademlia Implementation*. URL: `https://github.com/bmuller/kademlia` (visited on 04/02/2019).

[5] Isaac Zafuta. *Kademlia Implementation*. URL: `https://github.com/isaaczafuta/pydht` (visited on 04/02/2019).

[6] flosch. *Kademlia Implementation*. URL: `https://github.com/flosch/libdht` (visited on 04/02/2019).

[7] cjgu. *Kademlia Implementation*. URL: `https://github.com/cjgu/asyncio-kademlia` (visited on 04/02/2019).

[8] Stefan Saroiu, P Krishna Gummadi, and Steven D Gribble. “Measurement study of peer-to-peer file sharing systems”. In: *Multimedia computing and networking 2002*. Vol. 4673. International Society for Optics and Photonics. 2001, pp. 156–171.

[9] Ion Stoica et al. “Chord: a scalable peer-to-peer lookup protocol for internet applications”. In: *IEEE/ACM Transactions on Networking (TON)* 11.1 (2003), pp. 17–32.

[10] Antony Rowstron and Peter Druschel. “Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems”. In: *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer. 2001, pp. 329–350.

[11] Ben Y Zhao et al. “Tapestry: A resilient global-scale overlay for service deployment”. In: *IEEE Journal on selected areas in communications* 22.1 (2004), pp. 41–53.

[12] *BitTorrent*. URL: `https://en.wikipedia.org/wiki/BitTorrent` (visited on 03/21/2019).

[13] *Kademlia Specification Guide*. URL: `http://xlattice.sourceforge.net/components/protocol/kademlia/specs.html` (visited on 04/02/2019).