A DESCRIPTION OF PROJECT CONCEPTS, MILESTONES, AND GOALS

PROJECT PROPOSAL

February 27, 2019

CMPT 434: Computer Networks

Christopher Mykota-Reid
Rowan MacLachlan

Derek Eager

Contents

0.1 Proposal Requirements:

For an implementation project, you should:

- Describe what concept or issue you intend to study.
- Describe the objectives of your investigation.
- Give a brief plan of what you will do to investigate the issue.
- Describe expected measurements or criteria by which you intend to evaluate whether you've succeeded at your objective.

0.2 Core Concepts

What does peer-to-peer file sharing[3], blockchain, crypto-currencies[4], private peer-to-peer voice and text messaging apps[2], distributed social-media platforms[5], and interplanetary internet models[1] have in common? If you guessed: "Well, they all rely upon or implement in some way a distributed data structure to achieve some of their core functionality," you would be correct. Most of the technologies from the above list implement or utilize a distributed hash table (or a similar distributed data structure, in the case of blockchain). These 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 of application makes it a suitable candidate for an implementation project.

0.3 Objectives

We hope that by implementing a very simple version of a DHT and by demonstrating its use on a small network we will become familiar with the issues overcome by the more large-scale applications of DHTs. As referenced above, these issues are myriad and often difficult to solve, but not impossible. By implementing an incremental plan to gradually roll out more complex features, new features, or re-implement sub-systems of our application, we will minimize risk so as to end up with a viable end-product for our presentation, while also taking full advantage of every learning opportunity which presents itself.

0.4 Directives

The rough plan for our implementation project will be as follows:

1. Collect and collate relevant technical (algorithm descriptions, source-code) and theoretical sources (papers) for the implementation of DHTs. This has already begun.
2. Divide the DHT into aspects. Identify concerns such as hashing, security, node discovery, redundancy, etc. This has already begun.
3. Once the separate issues and their complexities are understood, design a minimum-viable product for our implementation. This MVP should be modular and take advantage of as many pre-written libraries as are available, while ignoring any details of a DHT beyond its barest and least robust implementation. It should also take into consideration the test application we have in mind.[1]
4. Once the MVP has been designed, identify aspects of it which can be improved. This will include things such as data redundancy (storing hashed data at multiple nodes), caching, and node deletion and addition (how is content from deleted nodes rehashed, and how is load redistributed on node addition?).
5. Implement the MVP, keeping in mind the fact that a modular design will make feature improvement and addition easier than it would be in a poorly constructed monolithic design.
6. Once the MVP is implemented and we move onto more advanced features, we have to consider the metrics and factors by which we measure the effectiveness of our implementation. What are its drawbacks and restrictions? How efficient is it at finding data on the overweb (participating nodes on the internet)? How efficiently is data stored and updated?

0.5 Implementation

0.5.1 High-Level Overview and Tooling

RPC

The Kademlia paper implies that the protocol is implemented with datagrams[6]. 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,

[1]Although it may not result in the most general implementation, it is more important that the core concepts are applied *successfully* in our particular application.

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. **TODO: More here on why Kademlia specifies UDP?** Other Kademlia implementations use a python library asyncio, and 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.

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.

hashing

Kademlia specifies the use of the 160-bit SHA-1 hash for data, and 160-bit node IDs and 160 k-buckets, 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.

```
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()
```

0.6 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.

0.7 Additional Details

Here, we list some of the technical considerations we will have to resolve in our implementation project:

- How many nodes store the data of a single object? We can start with a single node on a network, but what do we want to achieve? What degree of redundancy can be achieved and what overhead does this add?
- Is an entire object stored on each node, or do we distribute a single object in incomplete parts through the network?
- 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?
- What kind of hash function do we use? Do we use a cryptographically secure hash function such as SHA-1 or SHA-2 or do we use an unsecure hash function? What consequences and benefits do different approaches confer?
- Do we hash the object's identifier or the object data? If we hash the identifier, how do we enforce unique identifiers?
- 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?
- 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?
- What about the keyspace partitioning? 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?
  To better grasp this issue, consider the case where we use a direct hashing method. If there are n nodes in the system, then we would normally store object o at the node $hash(o) \pmod n$. When someone wanted to look up object o, they would find it at node $hash(o) \pmod n$. However, if a node disappears and the number of active nodes decreases by one, we can no longer find the object o at $hash(o) \pmod n$. The other side of this issue is to address

how objects stores at a node which dissappears are remapped to active nodes. Two separate possibilities to resolve this issue are consistent hashing and rendezvous hashing.
Clearly, the hashing method and design we choose is a central component and will have large ramifications for the rest of the project. As best we can, we must keep the hashing technique decoupled from the rest of the system in our implementation so as to avoid creating unnecessary work.

- How does a DHT handle node discovery and linkage? Once a file name has been hashed to a node, how do we find that node in the network? There are a myriad of algorithms designed to do just that: Kademlia[6], Chord[7], CAN, Tapestry[9], and Pastry[10], to name a few.
  Kademlia uses an XOR function on a GUID (per-node unique identifier) to calculate the distance between two nodes, because the XOR satisfies the requirements of an ideal distance function:
  - The distance from A to itself is 0,
  - The distance from A to B is the same as the distance from B to A, and
  - It satisfies the triangle inequality: the sum of the distance from A to B to C is greater than the distance between any two of A, B, or C, unless they lie on a line.

  Applications using Kademlia or slightly modified versions of the Kademlia algorithm include torrent clients such as BitTorrent, other P2P distributed file systems such as IPFS and Gnutella as well as P2P chat, voice, and file-sharing programs like Tox.

Please see the bibliography for additional reading.

Bibliography

[1] **`https://ipfs.io/`**
The official site of the IPFS project.

[2] **`https://tox.chat/`**
The official site of the Tox project.

[3] **`https://en.wikipedia.org/wiki/BitTorrent`**
The BitTorrent wiki article (sorry.)

[4] **`https://github.com/ethereum/devp2p/blob/master/rlpx.md`**
A reference file on the ethereum github.

[5] **`https://arxiv.org/pdf/1508.05591.pdf`**
An article on the use of DHTs for distributed social network implementations.

[6] **`https://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf`**
The original **Kademlia** algorithm paper.

[7] **`https://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf`**
A paper on the DHT protocol **Chord**.

[8] **`http://www.srhea.net/papers/tapestry_jsac.pdf`**
A paper on the p2p overlay message routing algorithm **Tapestry**.

[9] **`https://people.mpi-sws.org/~gummadi/papers/p2ptechreport.pdf`**
A measurement study on peer-to-peer file sharing systems.

[10] **`https://www.cs.rice.edu/~druschel/publications/Pastry.pdf`**
A protocol for internet overlay (overweb) object location and routing in potentially very large WANs.

[11] **`https://medium.com/karachain/peer-to-peer-protocols-explained-3b1d947c4`**
An pop-science article on peer-to-peer protocols.