The Distributed Hash Tree

Wiebe-Marten Wijnja

May 1, 2016

NOTICE

This is a draft version, version 0.5.0 of this paper. Read with care.

If you have any questions about this subject, or if you spot any mistakes, ambigueties or omissions in the paper, feel free to contact me at w-m@wmcode.nl

1 Abstract

By exploiting the nature of one-way hashing functions and digital signing algorithms, a distributed data structure in two layers is described. This tree-like structure is tamper-evident, allows in-band discovery of appended data entries, and can enforce access control measures to read and/or append to certain branches of this thee.

Distributed Hash Trees (*DH3*s) might be utilized to implement systems like distributed content indexes, distributed version-control-systems and distributed bulletin-boards. Because of its distributed nature, it is virtually impossible to remove information stored inside, which makes it resilient to both natural and man-made disasters. Also, it is very possible for multiple applications to share the same DH3-network.

The paper first describes the data structure in general terms, then it dives in some of the details and solutions for some of the problems that might come up. Finally, to ensure network safety, a method to prevent Sybil attacks on distributed systems is described.

2 Distributed Content Discovery

When creating a new kind of system where multiple entities (humans or computers) communicate with each other, the most obvious and often easiest approach is to create a *centralized* system: One entity is the leader whose authority is final. This central entity is the one that distributes the work and that keeps track of the actual state of information.

While easy to create, centralized approaches have some very obvious drawbacks:

- There is a single point of failure: When the leader stops working properly, the whole system breaks down.
- Other entities need to have complete trust in the leader; It is often impossible for other entities to tell when information is falsified.

With the advent of computers and networking, it has become possible to create distributed systems. While they come with the drawback of taking more resources[1]

[1]Depending on the type of system, these resources might be any of time, computation or storage space

to store and share the same information as a centralized system. They do, however, remove the earlier-mentioned drawbacks:

- Because data is shared and there is overlap in the stored data, the system still continues when a part of the nodes shut down.
- To ensure proper collaboration, sent and received data is verified by all nodes. A malicious node can easily be removed from the network without destabilizing it.

Storing data in a distributed network of computers is done by using a system called a **Distributed Hash Table**(*DHT*). This is a (key→value)-store where one can look up the value that is stored in the network by sending a query containing the (fixed-length e.g. shorter) key. This way, when something has been stored in the network, it is possible for any node in the network to efficiently retrieve it.

Many variants of Distributed Hash Table have been made. They vary in their implementation details, but all of them have:

- A method to iteratively find the node(s) most likely to hold the value, given a certain key.
- A method to request a value from a node.
- A method to store a value at a certain node.

Distributed Hash Tables vary in the amount of nodes that information is duplicated on, the way that determines what nodes should hold what (key→value) pairs and the way that keys are generated.

One thing that Distributed Hash Tables have a problem with, is finding out where information is stored, and also updating earlier-stored information: Most DHTs use *Content Addressible Keys*. That is, the key that identifies a value is generated by taking this value as input; usually using a digest function that changes this varying-length value into a fixed-length key.

Advantages:

- The only way for a concurrency conflict (nodes disagreeing about what value is stored under a certain key) to happen is when the value that was stored was identical.[2]
- It is impossible to change a value without invalidating it, as the key can be recomputed from the value by anyone. This makes the stored values immutable.

Disadvantages:

- It is impossible to append a value and let others know that there was an update, other than using an out-of-band method to share the new key with the other party(/parties).

The idea for the Distributed Hash Tree arose from the necessity of a method to share and update information while not being dependent on an out-of-band method to share the keys.

The hard part was finding out how predictable keys could be used instead of Content-Addressible keys while still keeping immutable and preventing concurrency conflicts.

[2]Of course, the pigeonhole-principle means that conflicting keys where the values are not identical can theoretically still happen, but with a reasonably large keyspace the probability of this happening is astronomically small.

3 Predictable Keys

Content Addressible Keys are impossible to predict. There are, however, other ways of key generation that *are* predictable. Using a method called **Iterative Hashing** it is possible to procedurally determine the next location where a new value that builds on the current value will be stored.

For this, a Cryptographic One-Way Hashing function such as SHA-2 or Keccak can be used.

ⓐ Observe that when $key_0 = hash(secret)$,a new key to store the next value can be inferred, by hashing the new key again: $key_1 = hash(hash(secret)) = hash(key_0)$. This procedure can be repeated any $n \in \mathbb{N}$ number of times, providing locations to store any number of values:

$key_0 = hash(secret)$
$key_1 = hash(hash(secret)) = hash(key_0)$
$key_n = hash(key_{n-1})$

The only information needed to move to the next index in this direction is the current hash.

ⓑ Instead, a secret value named the $salt$ can be concatenated[3] to each current key before hashing it. This procedure can also be repeated any number of n times:

$key_0 = hash(secret \parallel salt)$
$key_1 = hash(hash(secret \parallel salt) \parallel salt) = hash(key_0 \parallel salt)$
$key_n = hash(key_{n-1} \parallel salt)$

To move to the next index in this direction, both the current hash as well as the salt have to be known. Also observe that this procedure might be made more complicated by concatenating multiple salts to the hash.

There are other methods of iterative hashing as well, but no more than two variants are needed for the Distributed Hash Tree. As can be seen, depending on the iterative hashing **direction** that is used, parties need to have access to certain information. In the easiest case, only the current key is enough information. To restrict access to certain (key→value) pairs, passwords only known by the proper parties can be used as (part of the) salt and be concatenated to the current key at each iterative hashing step.

The Distributed Hash Tree is called that way because of the tree-like structure that is formed by following the iterative hashing directions from the initial (root) (key→value) pair to the (current) endpoints, also known as **leaves** of the tree.

4 Preventing Concurrency Conflicts

However, when two parties try to introduce a new value at the next iteratively hashed key at the same time, a conflict will occur: Half of the network will propagate the value of Alice, while the other half of the network will propagate the value of Bob. This is undesireable as the network now no longer has a concensus about the actual state of events.

To prevent this from happening, a second hashing-direction is introduced, the so called **conflict-resolving direction**. By having nodes iteratively hash in this conflict-resolving direction whenever they are expected to store a value at an already-taken

[3]the concatenation operation is denoted as '$\parallel$'. Other operations such as $\oplus$ (exclusive or) might also be used for a similar effect, providing even more directions with the same level of secrecy.

key, we can ensure that both the value of ALice and the value of Bob are stored in the network. Network concensus is restored. Do note however, that the *order* of Alice's and Bob's value is something that Nodes might disagree on, and therefore should not be relied upon. *(We will see below why this is not a large problem.)*

By reading not only the key itself but also iterating over the conflict key (reading the conflict key of the conflict key, etc... until a non-filled space is found[4]), we can ensure that we gather all values that were added as an answer on the current (key→value) pair.

5 Reintroducing Immutability, and also introducing Non-Repudiation

However, there still is no way to ensure immutability of the values that are transmitted over the network. A solution needs to be found to prevent a malfunctioning (or malicious) node from modifying an uploaded value before storing and propagating it to the rest of the network.

This can be done by using a digital signing algorithm such as the Elliptic Curve Digital Signing Algorithm (ECDSA) to have the uploader sign the data in a specific value.

But because most values in the network depend on previous values, this is not enough: We need to ensure that we have a Merkle-tree-like structure to ensure that the current value will be invalidated when the value it references (or the value that value might reference, etc.) is modified.

This is done by introducing the second (inner) layer, the **tree** layer.

5.1 The Tree layer vs the Table layer

The first (outer) layer is the **table** layer, which is the outside wrapper that has been described earlier: On this layer, the DHT stores (key→value) pairs. This layer uses the concept of iterative hashing to ensure that all values are stored and concurrency problems do not arise. A newly-added (key→value) pair will always point to the last (key→value) pair that was part of the same project. This layer is therefore structured as a two-dimensional list, and not as a tree.

The **tree** layer is a whole different beast: This actually creates a tree-like structure. The fields of this layer are all contained inside the *value* part of the table layer. The fields are as follows:

parent-reference tree-hash of parent node. (if this node is a root node, this field is NULL)

data Actual data that is stored.

signature An ECDSA-signature of **parent-reference** and **data**.

tree-hash The tree-hash identifier of this tree node. Subsequent nodes can refer to this. Calculated by hashing **signature**.

This way, we ensure that it is impossible for anyone to modify the data without invalidating both the field and all its descendants, as well as clearly showing that the signature is not correct.

[4]To even further enhance the stability of the system, one might continue reading in the conflict-resolving direction until n consecutive empty spaces are found. This ensures that the tree will still be stable if a single value is removed from it.

It also creates non-repudiation: If someone uploads something that is signed by them, they cannot afterwards claim that they did not do so.

5.2 What signing keys to use

Determining what public/private keys are valid to sign values with is application-specific: If a DH3 is required that everyone can read but only one person is allowed to write to, client applications reading the tree should reject all public keys bar the one of this person.

It is completely possible to add the public key of the signer to the **data** field, in cases where anyone might upload. It is also possible to have one key that is allowed to add to certain branches of the tree, and other keys that are not allowed there but are allowed in different parts. It is also possible to have one key that adds tree nodes containing whitelisted keys for other operations, etc.

Thus, complex application-specific authentication logic is possible.

6 Reading data from the Distributed Hash Tree

As explained before, to read data from the DH3, one needs to know the key of a (key→value) pair. From this key, it is then possible to iteratively hash and find all subsequent locations that something might be stored. For each of the locations generated in this fashion, it is also necessary to iteratively check the conflict-resolving direction, to ensure that data that was added at nearly the same time is not missed.

One can iterate until no values can be found. To make the process faster in the future (as both hashing and looking data up in the DHT are expensive/slow operations), it is recommended to cache the (key→value) pairs that were found, as well as keep a list of all 'loose ends' of this two-dimensional table. In the future, lookup can then start at these loose ends.

Internally, a tree can be built by adding children to a node whenever they refer to an earlier node by its **tree-hash**. What to then do with the resulting nodes is application-specific.

7 Safe Servers & preventing Sybil attacks

The network is split up in Servers. [5] These servers implement the adapted form of the Kademlia Distributed Hash Table, that, when asked to add a value to the DHT, does not return an error code when a location is already filled. Instead, it stores the value at the first empty spot in the conflict-resolving direction, and returns that key to the requester.

Servers themselves do not care for the *tree*-layer, only storing and returning the encapsulated *table*-layer results.

A Server has a Server-ID: This is a hash key just like keys used to store values [6]

To ensure that the network is safe for Sybil attacks, the procedure in creating a Server-ID is slightly more complicated than 'just picking a random ID':

[5] These are also sometimes called *nodes*, but this is confusing in the DH3-context as a *node* also refers to an element in the resulting tree.

[6] Kademlia refers to these as the Node ID – again, this nomenclature is avoided in this paper to prevent confusion with tree nodes.

metadata Each Server has a set of arbitrary metadata. This contains its address, software version, maintainer email and maybe in the future more information. This information can be edited at will by the Server maintainer.

bcrypt digest Each Server then computes the Bcrypt digest hash of this metadata. A suitably high iteration-count is chosen to ensure that this is a time-costly operation. The resulting hash is public information, and the server is required to send it when requested.

Server-ID This Bcrypt digest is then hashed using the chosen hashing function to create a key in the keyspace of the DHT. This will be the Server's Server-ID.

Because of the Bcrypt step that takes a lot of time, it is no longer feasible to brute-force your way to a desired Server-ID. In the future, when computers get faster, the required Bcrypt minimum iteration count can be increased (and Servers with less than that therefore rejected by their peers).

To check if a Server functions properly, another Server or Client that connects to this Server should check:

1. If the bcrypt digest matches the given metadata.
2. If hashing the bcrypt digest indeed returns the given Server ID.

This is also a costly operation, because the same number of Bcrypt iterations needs to be performed. Therefore, Servers and Clients should keep track of a list of Servers they have validated previously.

8 Conclusion

Presented was the Distributed Hash Tree, a new data structure built on top of a slightly modified Kademlia Distributed Hash Table. In the Distributed Hash Tree, it is possible to store data that is non-repudiable, immutable and, if necessary, confidential. Removing or invalidating a value in the network once it has been uploaded to there is nearly impossible. This makes the Distributed Hash Tree very resilient against natural or man-made disasters.

Using the Distributed Hash Tree, it is possible to propagate content to other parties without needing an out-of-band communication method.

Finally, a method to prevent Sybil attacks has been described.