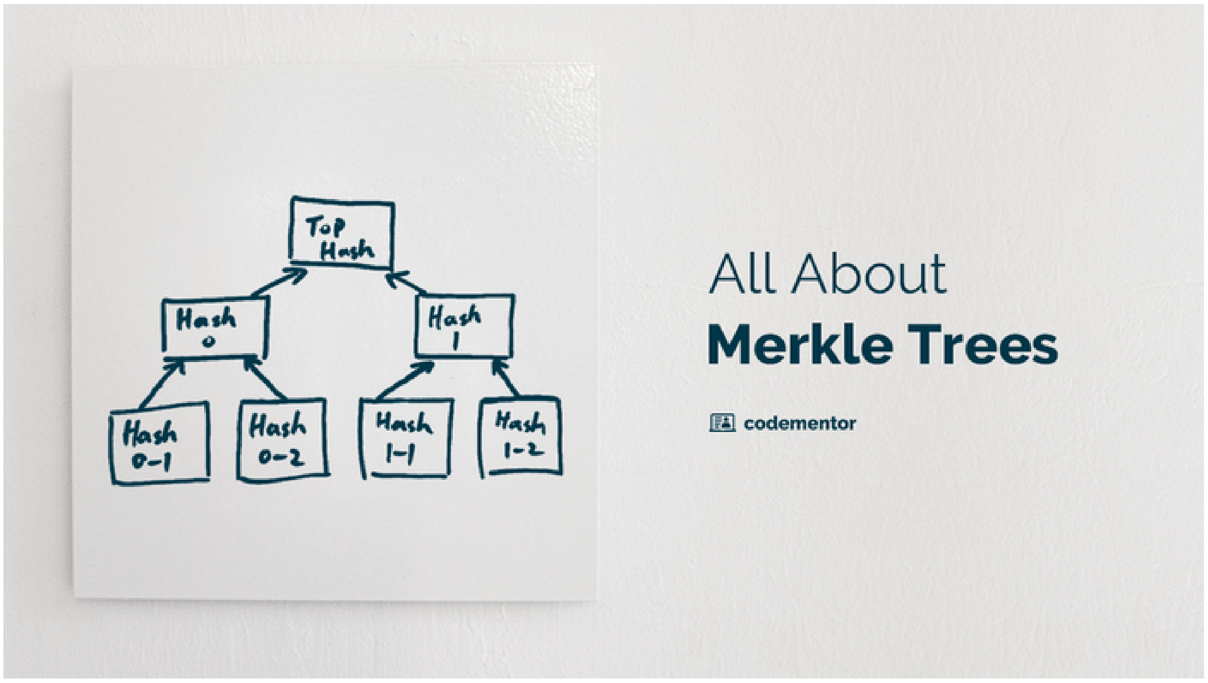
Last updated Oct 26, 2018

Merkle Trees: What They Are and the Problems They Solve



Quick question, can you think of something that's common amongst the implementation of Bitcoin, Git, IPFS, Ethereum, BitTorrent, and Cassandra?

Well, it's a technique to store data called the "merkle tree." Merkle trees are a useful component of so many technologies (mostly ones distributed in architecture), and the credit for coming up with such an awesome concept goes to Ralph Merkel. In 1979, Ralph Merkle patented the concept of hash trees, and in 1987, he published the paper "A Digital Signature Based on a Conventional Encryption Function" which used this concept.

Although his patent expired in 2002, this data structure became such a useful part of different technologies that it later came to be known as **merkle trees** (now you know where the name comes from!).

In this article, we'll first look at some of the eminent problems of the current state of the web, how decentralization is a potential solution for them, then we'll see where merkle trees come into the picture, how they are implemented, and the problems they solve. Feel free to jump to a specific section:

Let's get started!

Problems With the Current Web

The internet has been evolving for almost two decades, and sharing your content on the web wasn't as easy as it is now. With time, a lot of different services have been built and let us share our content effortlessly on the web. Be it be video sharing platforms like YouTube, thought sharing platforms like Twitter, or even create-your-own-site services like Wordpress.

No doubt, these services improved the accessibility of the web, but there are also some inherent concerns about this kind of structure of the web which started to arise. A few of them are:

- **Concentration of data:** The data we generate or the information we need to access on the internet keeps on accumulating in the data centers of a few giant companies (which provided all these content hosting services in return), and steadily, we realized that having control over our data is as important as the services these companies are offering.
- **Host-based addressing:** In the initial days of the web, when services like Facebook, Blogspot, Pinterest, etc. were absent, the only way to share what you want was to spin up a server, get a domain, and host your content. We can then share our content by sharing the address (something like my_site.com/my_content).

Now, we have social-media platforms where we share our content, and the addresses are now something like your_site.com/your_content and your_site.com/someone_elses_content which leads to undesired control, concentration, and even duplication of content (among the competitors). Hence, need of a better way of addressing is eminent, which can decouple the host from the content itself, i.e., we need *content-based addressing*.

- **Unable to utilize proximity benefits of a network:** The web servers of a hosted website might be serving from a far away part of the globe. Also, we're able to download one piece of content from only one server at a time. A much more efficient way for getting the content could be to obtain multiple pieces of it from multiple nearby computers in the network that already has parts of the content we seek.

Decentralization as a Solution

Different projects have come up in the last decade which have addressed these problems. A major theme in all of them was the distributed (peer-to-peer) nature of the architecture, which also leads to the distribution of control from one central point to the entire system. However, these systems have their own kind of challenges to solve. For example, if we look at distributed storage systems in general, there's one big problem that needs to be solved, and that is the problem of "verification."

How do I know that the information that I'm getting from some peer is genuine and hasn't been tampered with (or corrupted)?

The Need for Verification

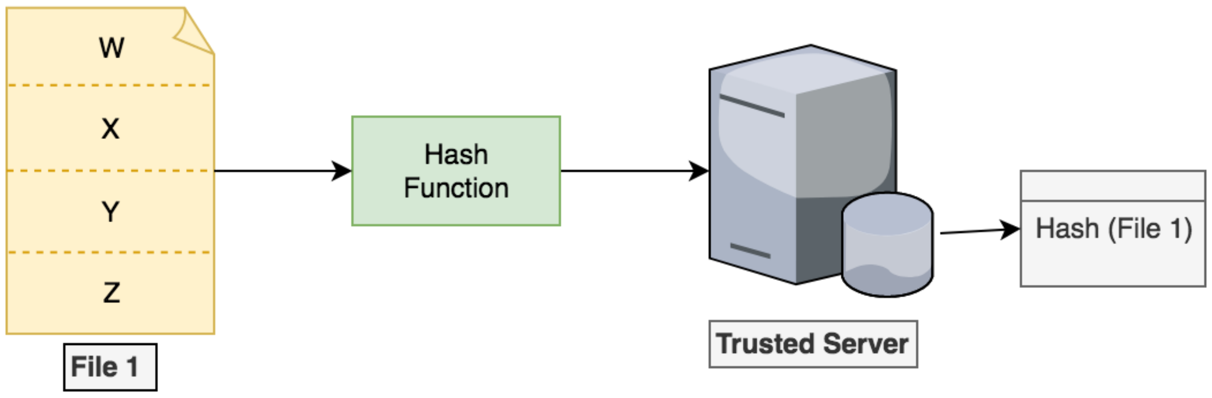
A mechanism was needed to verify that the participants of the network have not:

- Accidentally corrupted the data or
- Intentionally tampered the data.

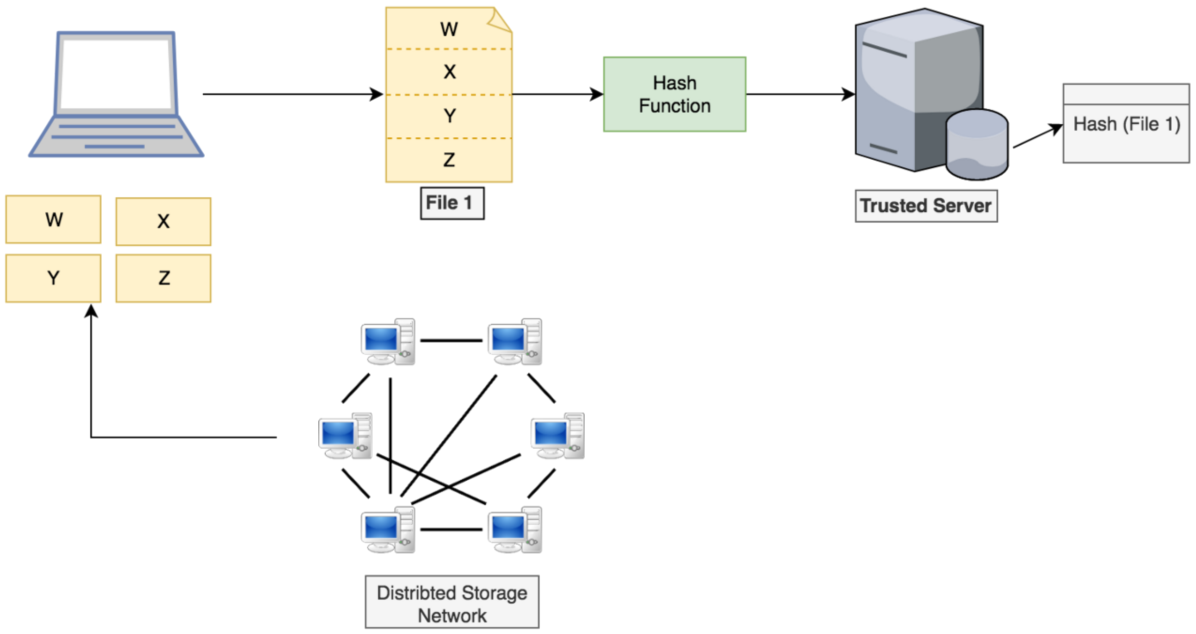
Let's try to think for a bit about how can we solve this problem in distributed systems. To someone familiar with Computer Science, one common thought in tackling this problem would be to use simple cryptographic hash functions, so let's start with that only.

Cryptographic Hash Functions as a Possible Solution

Let's say we want to store a large file in a distributed manner. We can compute the hash of the entire file (before storing it in a decentralized way), store this hash at a "trusted" location, and use it for verification in the future.



So after computing the hash, we distribute the file to some peers in the network. Whenever any peer wants to get the entire file, he/she downloads the chunks from the peers who have the file, combines them, computes the hash of the entire file, and verifies if the hash exists in the trusted server.



If you think two different files can have the same hash, yes it's possible to get a hash collision, but almost impossible (a 256-bit hashing algorithm can generate 2^256 combinations, and this number is comparable to the number of atoms in the entire universe!). Also, since the hash only depends on the content being hashed, it solves the problem of content-based addressing, which means we can use it to locate files on our network. The information to resolve the peers for a particular file can be stored in a Distributed Hash Table maintained by the untrusted peers.

This approach would work, but it isn't efficient, due to the following reasons:

1. **Verification only after completion**: The peer has to wait for the entire data (from several peers) to arrive before he/she can verify its authenticity. If there are thousands of peers scattered all over the world (like in a torrent network), some chunks will arrive earlier and some chunks won't. So waiting for obtaining the entire data before verifying if it is correct in the first place, is not an efficient approach.
2. **No way to figure out the culprit**: If the verification fails (the hash provided by the trusted server doesn't match the hash of the file downloaded by the peer), there's no way to find out which peer(s) were responsible for sending the incorrect chunk(s).
3. **Too much overhead in syncing**: Let's say we want to update the contents of our existing file. Even if there's one character change in some chunk, the entire file needs to be hashed again, and the generated hash needs to be communicated to the trusted server.
4. **Too much "trust" on our trusted server**: What if the trusted server gets compromised? Then we can never know if it's the peers lying or the trusted server lying, or both of them lying in tandem.

Of course, we can do better. What if instead of just storing the hash of the complete file, we also maintain the hashes of the individual chunks in our "trusted" server. In that way, we can verify the chunks as we collect them from untrusted peers?

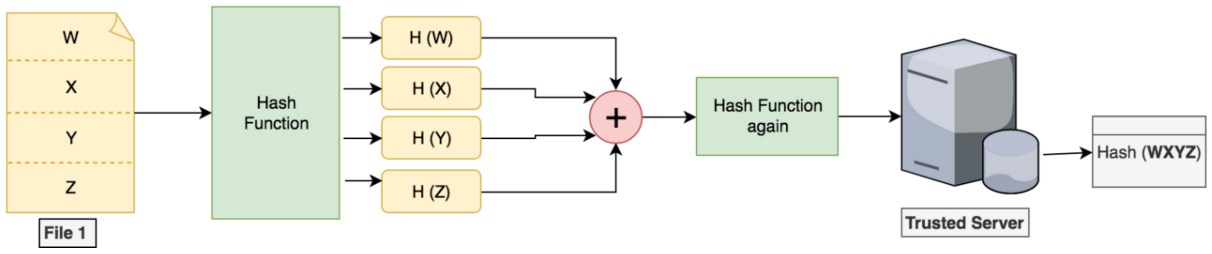
Yes, that'd solve "1" and "2" as we'll be able to verify the chunks individually as soon as we download them from the peers. This would also somewhat solve "3" since the newly updated chunks can be hashed individually and this information can be communicated to the trusted server, but:

- The trusted server has to store hashes corresponding to all the chunks corresponding to all the files, which means slightly more storage requirement than the previous case ("slightly" because we're just storing hashes which might be around 8 bytes per chunk).
- The chance of hash collision increases slightly.

- To verify every chunk downloaded, we need to rely on the trusted server. And we still can't do anything if the trusted server gets compromised.

Hash Chaining as a Possible Solution

Okay, what if we do this?



If we combine all the individual hashes and hash them again to obtain a **root hash**, then we just need to store that root hash at a trusted location. We can use this root hash for content-based addressing and verification, therefore:

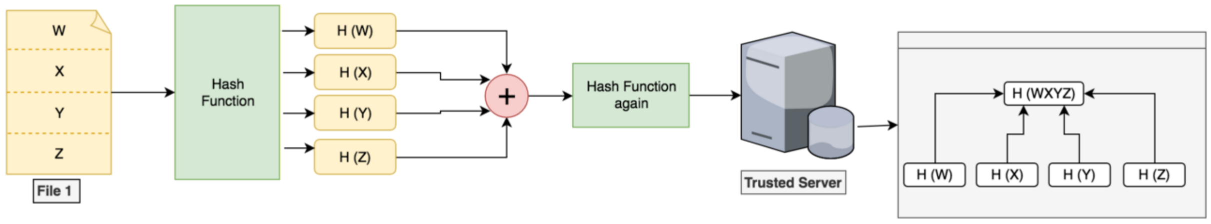
- In the beginning only, the peer can collect hashes of all the chunks from other peers on the network (who hold the chunks of the file needed).
- Once any chunk is downloaded, the peer can compute its hash and then compute the root hash and check if it exists in the trusted server.
- And now, since we know the root hash exists in the server, we don't need to worry about bothering our trusted server and can verify the future chunks locally, which would reduce overdependence on the trusted server.

All good, but we still can't do anything if the server is compromised and gives us false acknowledgment of the hash's existence.

As it turns out, there's a way in which the server can prove that it is correct. Here's how...

Storing More Information In the Trusted Server

What if we store both the individual chunk hashes and the root hash in the server? At the cost of slightly more storage in the trusted server, we would finally get to solve "4" by following this process for verification:



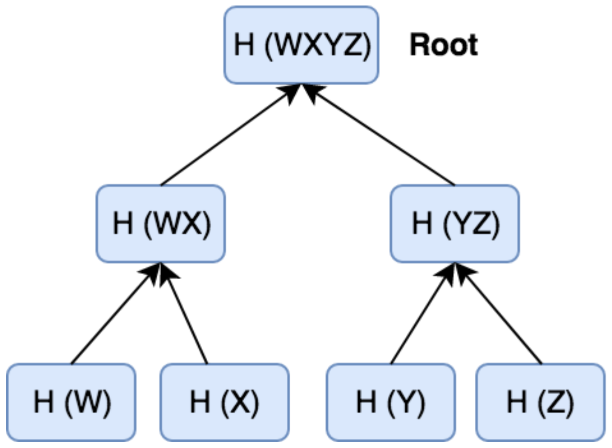
- Instead of sending an acknowledgment of existence, the server will send the entire proof to the peer requesting verification. By entire proof, we mean all the hashes of the fellow chunks and the root chunk.
- Since the server is now sending a proof the client can verify, the client now computes the root hash by concatenating:
  - The hash of the current chunk that he/she wants to verify
  - The hashes of all the fellow chunks received from the trusted server

    Then finally computing root hash, and verifying it against the root hash used to locate the file on the network.

Alright, good news, we're very close to the idea behind merkle trees!

So, What Are Merkle Trees?

What if we store a tree at the trusted server, that looks something like this,



The leaves of the tree correspond to hashes of the data chunks of a file, and the parents of these leaves being hashes of the concatenation.

These structures are called merkle trees (also hash trees).

Why Are Merkle Trees Awesome?

You might wonder, how is this approach more efficient than the previous hash chaining one? What advantage does a tree of hashes give to us?

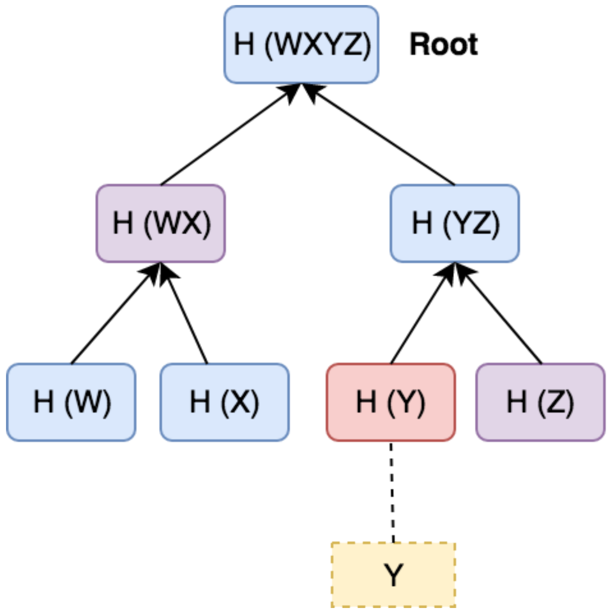
The root hash is used for content-based addressing. The merkle tree-like organization allows us to perform the following tasks in a very storage- and computation-efficient way:

Data Verification

Here's how the data verification happens in merkle trees,

1. We download some chunk of data from the untrusted network.
2. We ask the server to provide the proof that this chunk is in the tree.
3. The server returns the appropriate hashes.
4. Using this information, you compute the root hash and verify it against the root hash with which you accessed the file.

For example, if the peer wants to verify that "Y" chunk exists in the file and is untampered, the server returns the info `H (Z)` and `H (WX)` , which is also known as the audit trail.



We can then compute:

- `H (YZ)` from `H (Y)` that we already have, and `H (Z)` that trusted server provided to us.

- `H (WXYZ)` from `H (YZ)` we just computed, and `H (WX)` that trusted server provided to us.

Finally, we can compare `H (WXYZ)` computed to the original root hash used to locate the file on the untrusted network. If the hashes match, the proof validates that the chunk exists in the tree and is not tampered/corrupted. If in case the proof fails, we can chuck this peer and ask for the same chunk from another peer who has the file that we're looking for. This process is also known as audit proof.

And here's the good part,

- A very little information was required from the trusted server to verify the data. If the number of data chunks are doubled, the additional information required for verification would be just two more hashes, and the verification on the client side would require two more hash computations.
- Since the size of data verification packet is small, it helps us to save bandwidth.

This a big improvement over our previous hash chain approach where the trusted server had to send all the hashes of the fellow chunks, and the peer used to combine all of them to compute the root hash. We can even go a step further, and develop a mechanism to get the audit trail from the untrusted peer-to-peer network.

Consistency Verification

Consistency verification is desired in systems maintaining immutable (and hence append-only) log of data. It is used to verify that the entire log is untampered, which means verifying that the newer version at any time frame includes all the data of the older version and in the same order, i.e. no data at any stage has been into the history of the log.

Data Synchronization

Merkle trees can be used in synchronizing data across multiple nodes (peers) in a distributed system. With merkle trees, we don't need to compare the entire data to figure out what changed — we can just do a hash comparison of the trees. Once we figure out which leaves have been changed, the corresponding data chunk can be sent over the network and synced across all the nodes.

Merkle Trees in Action

Implementing the Basic Data Structure

What could be a better way of understanding merkle trees than just implementing one? We'll use Python as the programming language, since it's easy to understand for anyone with basic programming experience.

```python
from hashlib import sha256


class MerkleNode:
    """
    Stores the hash and the parent.
    """
    def __init__(self, hash):
        self.hash = hash
        self.parent = None


class MerkleTree:
    """
    Stores the leaves and the root hash of the tree.
    """
    def __init__(self, data_chunks):
        leaves = []

        for chunk in data_chunks:
            node = MerkleNode(self.compute_hash(chunk))
            leaves.append(node)

        self.root = self.build_merkle_tree(leaves)

    def build_merkle_tree(self, leaves):
        """
        Builds the Merkle tree from a list of leaves. In case of an odd number
        """
        num_leaves = len(leaves)
        if num_leaves == 1:
            return leaves[0]

        parents = []
        i = 0
        while i < num_leaves:
            left_child = leaves[i]
            right_child = leaves[i + 1] if i + 1 < num_leaves else left_child

            parents.append(self.create_parent(left_child, right_child))

            i += 2

        return self.build_merkle_tree(parents)

    def create_parent(self, left_child, right_child):
        """
        Creates the parent node from the children, and updates
        their parent field.
        """
        parent = MerkleNode(
            self.compute_hash(left_child.hash + right_child.hash))
        left_child.parent, right_child.parent = parent, parent

        print("Left child: {}, Right child: {}, Parent: {}".format(
            left_child.hash, right_child.hash, parent.hash))
        return parent

    @staticmethod
    def compute_hash(data):
        data = data.encode('utf-8')
        return sha256(data).hexdigest()
```

Our merkle tree is a full binary tree and in case of an odd number of leaves, we'll duplicate the last leaf to achieve this. The `build_merkle_tree` method creates the tree recursively in a bottom-up fashion. Let's check if the class that we've implemented so far works,

```
>>> file = "01234567" # some file
>>> chunks = list(file) # dividing the file into chunks
>>> chunks
['0', '1', '2', '3', '4', '5', '6', '7']
>>> merkle_tree = MerkleTree(chunks) # creating a merkle tree bottom-up
Left child: 5feceb66ffc86f38d952786c6d696c79c2dbc239dd4e91b46729d73a27fb57e9, R
Left child: d4735e3a265e16eee03f59718b9b5d03019c07d8b6c51f90da3a666eec13ab35, R
Left child: 4b227777d4dd1fc61c6f884f48641d02b4d121d3fd328cb08b5531fcacdabf8a, R
Left child: e7f6c011776e8db7cd330b54174fd76f7d0216b612387a5ffcfb81e6f0919683, R
Left child: fa13bb36c022a6943f37c638126a2c88fc8d008eb5a9fe8fcde17026807feae4, R
Left child: 67d62ee831ff99506ce1cd9435351408c3a845fca2dc0f34d085cdb51a37ec40, R
Left child: 862532e6a3c9aafc2016810598ed0cc3025af5640db73224f586b6f1138385f4, R

>>> print(merkle_tree.root.hash) # Let's see the root hash
e11a20bae8379fdc0ed560561ba33f30c877e0e95051aed5acebcb9806f6521f
```

Getting the audit trail

Let's extend the code to return the audit trails as well for any data chunk (or leaf in the merkle tree). Since the first thing to do is to check if the leaf (hash corresponding to the chunk) exists in the tree or not, we've introduced `left_child` and `right_child` fields in our `MerkleNode` class. Once the leaf is located, we append the other child of the parent of the audit trail, and recursively build the audit trail with the `generate_audit_trail` method.

```
class MerkleNode:
    """
    Stores the hash and the parent.
    """
    def __init__(self, hash):
        self.hash = hash
        self.parent = None
        self.left_child = None
        self.right_child = None


class MerkleTree:
    ###
```

```
def create_parent(self, left_child, right_child):
    parent = MerkleNode(
        self.compute_hash(left_child.hash + right_child.hash))

    parent.left_child, parent.right_child = left_child, right_child
    left_child.parent, right_child.parent = parent, parent

    return parent
###

def get_audit_trail(self, chunk_hash):
    """
    Checks if the leaf exists, and returns the audit trail
    in case it does.
    """
    for leaf in self.leaves:
        if leaf.hash == chunk_hash:
            print("Leaf exists")
            return self.generate_audit_trail(leaf)
    return False

def generate_audit_trail(self, merkle_node, trail=[]):
    """
    Generates the audit trail in a bottom-up fashion
    """
    if merkle_node == self.root:
        trail.append(merkle_node.hash)
        return trail

    # check if the merkle_node is the left child or the right child
    is_left = merkle_node.parent.left_child == merkle_node
    if is_left:
        # since the current node is left child, right child is
        # needed for the audit trail. We'll need this info later
        # for audit proof.
        trail.append((merkle_node.parent.right_child.hash, not is_left))
        return self.generate_audit_trail(merkle_node.parent, trail)
    else:
        trail.append((merkle_node.parent.left_child.hash, is_left))
        return self.generate_audit_trail(merkle_node.parent, trail)
```

Let's try to test it by computing the audit trail for one of the chunks,

```
>>> file = "01234567"
>>> chunks = list(file)
>>> merkle_tree = MerkleTree(chunks)
>>> print(merkle_tree.root.hash)
e11a20bae8379fdc0ed560561ba33f30c877e0e95051aed5acebcb9806f6521f
>>> chunk_hash = MerkleTree.compute_hash("2")
>>> chunk_hash
d4735e3a265e16eee03f59718b9b5d03019c07d8b6c51f90da3a666eec13ab35
>>> audit_trail = merkle_tree.get_audit_trail(chunk_hash)
>>> audit_trail
[('4e07408562bedb8b60ce05c1decfe3ad16b72230967de01f640b7e4729b49fce', False), (
```

Our function returns the audit trail, along with the root hash. Usually, the root hash is something which is already known to the client (since it's used for content-based addressing), but for simplicity, we're appending it to the end of the audit trail.

Now, let's implement the verification method, which will be used on the client side to verify the authenticity of any chunk from the audit trail received from the trusted server.

```python
def verify_audit_trail(chunk_hash, audit_trail):
    """
    Performs the audit-proof from the audit_trail received
    from the trusted server.
    """
    proof_till_now = chunk_hash
    for node in audit_trail[:-1]:
        hash = node[0]
        is_left = node[1]
        if is_left:
            # the order of hash concatenation depends on whether the
            # the node is a left child or right child of its parent
            proof_till_now = MerkleTree.compute_hash(hash + proof_till_now)
        else:
            proof_till_now = MerkleTree.compute_hash(proof_till_now + hash)
        print(proof_till_now)

    # verifying the computed root hash against the actual root hash
    return proof_till_now == audit_trail[-1]
```

Okay, let's verify the `chunk_hash` with its `audit_trail` received previously:

```
>>> verify_audit_trail(chunk_hash, audit_trail)
True
```

Works as expected! Let's consider another scenario where the data chunk is tampered,

```
>>> tampered_chunk = "20" # tamepering the chunk from 2 to 20
>>> tampered_chunk_hash = MerkleTree.compute_hash(tampered_chunk)
>>> audit_trail = merkle_tree.get_audit_trail(tampered_chunk_hash) # getting th
False
```

As we can see, the `tampered_chunk_hash` wasn't present in the merkle tree, so we got `False` in response, which is great. Now let's check what would happen if the trusted server is compromised and a wrong audit trail is sent to the peer.

```
>>> some_chunk = "3"
>>> chunk_hash = MerkleTree.compute_hash(some_chunk)
>>> audit_trail = merkle_tree.get_audit_trail(tampered_chunk_hash)
>>> audit_trail
[('d4735e3a265e16eee03f59718b9b5d03019c07d8b6c51f90da3a666eec13ab35', True), ('
>>> audit_trail[0][0] = "0"*64 # tampering the audit trail, we'll assume this i
>>> tampered_audit_trail = audit_trail
>>> tampered_audit_trail
[('0000000000000000000000000000000000000000000000000000000000000000', True), ('
>>> verify_audit_trail(chunk_hash, tampered_audit_trail)
False
```

In this case, the audit-proof failed as expected, which shows that trusted server can't lie to the peer, and hence our intial problem with the trusted server being compromised is solved.

Alright! We've come very far in understanding and implementing the merkle tree. The real world applications of merkle trees use tailored implementations of the data structure, but the core concepts behind them are the same as discussed in this post so far. Let's have a look at some real-world use cases of merkle trees.

Use Cases for Merkle Trees

In general, what can we use merkle trees for? The short answer is, merkle trees can be an integral part of the systems which require:

- Data verification
- Consistency verification
- Data Synchronization

So things like:

Cryptocurrencies

Many cryptocurrencies (including Bitcoin) store the transaction data in a merkle tree structure. Merkle trees help in consistency verification, i.e., making sure that the newer version of the ledger includes all the transactions from the previous version in the same order.

Version Control Systems

Version control systems like Git and Mercurial use specialized merkle trees to manage versions of files and even directories. One advantage of using merkle trees in version control systems is we can simply compare hashes of files and directories between two commits to know if they've been modified or not, which is quite fast. This post discusses in detail how the entire process works.

Certificate Authorities

Certificate Authorities use merkle tree in their mechanisms to maintain certificate transparency logs that are verifiable. The log keeps on growing as new certificates are appended to it, and the transparency is verified using a log proof mechanism.

Database systems

No-SQL distributed database systems like Apache Cassandra and Amazon DynamoDB use merkle trees to detect inconsistencies between data replicas. This process of repairing the data by comparing all replicas and updating each one of them to the newest version is also called **anti-entropy repair**. The process is also described in Cassandra's documentation.

Conclusion

In this post, we took an in-depth look at how merkle trees work and how it compares to similar solutions, how merkle trees can be implemented, why they are an integral part of so many technologies, how you can use merkle trees to solve issues that require data verification, consistency verification, and data synchronization, and finally, what use cases merkle trees are being utilized in.

Published Oct 25, 2018

Engineering & Technical Insights

Like this article? Share it with your friends!

Satwik Kansal

Python | Data Science | Distributed Applications

https://satwikkansal.xyz Hey there! I'm the author of "What the f*ck Python?". My skills of expertise include Data Science, and Distributed applications (Large scale systems, Blockchain, etc). I actively work with Open Source ...

Related Articles

29 AngularJS Interview Questions