Proof of Work, Explained (Part 2 - A Hash Bash for Techies)

Author	josh
Date	2017-11-30 01:41:31

Overview

In the last article, we looked at the overall idea of proof of work and its applications. That article covered the origins of this concept, how it works at a high level, and some of its applications. Now, let's take a look at the technical inner workings of these algorithms.

In a nutshell, proof of work involves the use of hash functions. These one way functions form the basis for a difficult to solve, but easily verifiable computational puzzle as a way to prove that one did some amount of desired computing work.

Proof of Work, the Technical Perspective

Hash Functions

First, we need to understand a bit about hash functions and why they form the basis for proof of work. A hash function is a *one-way* function that takes some input of any size and outputs a consistently sized set of bits. The two most important characteristics of hash functions are that they:

- Are *one-way* - you cannot take an output and find the input without brute force guessing
- Have unique outputs for every possible input (if the hash function is a good one!)

These two properties are critical for proof of work. First, the one-way nature of these makes it so that brute-force is required to find some desired output. Second, the desired one-to-one input/output property makes it so we can easily verify the solution once we have one.

An Overview of the Algorithm

Hashing and Binary and Difficulty Targets, Oh My!

Proof of work builds on top of the properties of hash functions by realizing that as a stream of bits, hash outputs *actually represent binary numbers*. For example, an 8 bit hash 00001000 represents the decimal number "8". Now remember that hash outputs can only be matched to a particular input by using brute force to guess.

Using these properties, proof of work takes a pretty ingenious approach to making a user do some amount of predetermined work - it makes them look for a hash that, interpreted as a number, is *less than* some target value!

This is where the idea of difficulty comes in. Let's say you want the user to find some input where the hash value, when representing an 8 bit number, has two zeros in the front (00101010, for example). Now imagine you want the user to find an input that gives a hash with four zeros in front (00001011). Which one takes more guesses to compute? It turns out that the smaller the "difficulty target" value, the more guesses (and more computing time) it takes to find an input that gives the desired hash output. This is the fundamental basis for proof of work. It can be statistically predicted that a certain difficulty target will take roughly some amount of guesses (and therefore computing time) to find. So the smaller the difficulty target, the harder the puzzle.

The `nonce` value

Now since hash outputs map one-to-one with some input, how can we prevent the worker from just using a dictionary to find an input that meets the difficulty? Here is what makes proof of work truly *proof* - we always create a hash input unique to the problem we're trying to solve, using an applicable message and a random guess we call a *nonce*.

See, for our proof of work to truly require the desired amount of work, we always start with a unique message for the problem. In the case of Bitcoin, our message is what is called the "block header" - a chunk of data containing information about the transactions included in the current block. In an anti-spam application, this message would be something like a forum post or the contents of an email message. Since this message is unique, a dictionary cannot be used to guess a hash that meets the difficulty target.

The worker then has to take the `message` plus a random number guess called the `nonce`, and run that combined string through the hashing function. If the hash output isn't less than the target number, then the worker increments that random number guess concatenated to the message and tries again - and again and again until the right output is found.

Verifying the nonce

Once a nonce is found, another node or server can very easily verify that the solution (the nonce) is correct. All the verifying party has to do is take the original message plus the nonce value found by the worker and run that through the hash function. Since a hash input will always give the same output, it only takes one step to verify that the worker's nonce is in fact a correct solution to the proof of work problem.

A Practical Example

Let's take a look at an example anti-spam proof of work problem. Let's say a user wants to contact a site owner with the message "Hello", and the site owner wants the user to do some proof of work before sending that email. The site owner specifies a difficulty target of 2^240. This difficulty target can be any 8 bit number in this case, but a power of two is easy to work when building an application. This system uses the compute-intensive SHA-256 hashing algorithm for its proof of work. Here's what the steps would look like:

Worker (Client)

1. Uses "Hello" as the `message`
2. Starts guessing a `nonce` with 0 - the hash input is the string "Hello0"

3. The `SHA-256` output of "Hello0" (in hexadecimal format) is `80878c5b013ba72c0d2b7e8f65868649cbdb1e7e7a8c8a07537d6b3619e4e32f`
4. Clearly, this output is greater than the `difficulty target` of 2^240, which would have three prepending 0's in hexadecimal: `0001000000000000000000000000000000000000000000000000000000000000`
5. Increment `nonce` to 1, and try again. This continues until an appropriate `nonce` is found
6. The client finally finds a `nonce` that works with the value `9172`. The `SHA-256` hash of "Hello9172" is `00001f2e9f8f74117b4178eb04b368c807f906ae2a07bece562266cbc9adff3c`, which is less than the `difficulty target` of `0001000000000000000000000000000000000000000000000000000000000000` (2^240)
7. Since the client has a `nonce` guess that meets the `difficulty target` for this unique message, it now has proof that it did all that computing work!

Verifying Party (Server)

1. Take the `message` for this problem, "Hello", plus the client's found `nonce`, "9172" and pass "Hello9172" through the `SHA-256` hash function
2. Since hash functions produce the same output for any input, we get the same output the client found: `00001f2e9f8f74117b4178eb04b368c807f906ae2a07bece562266cbc9adff3c`.
3. Since the above output is indeed less than the `difficulty target` 2^240, the server has now verified that the client did the desired amount of computing work to find the `nonce`. The message can now be sent.

Proof of Work - Hashing for a Cause

These algorithms put the properties of hashing algorithms to new and innovative uses, particularly in the incredible space of cryptocurrencies. Proof of work takes the one-to-one input/output and irreversible properties of hash functions and uses them to create difficult to solve, easy to verify computing problems. This simple but interesting bit of math and computer science powers new approaches to interesting challenges. Proof of work can be used to help prevent spam in a new and unique way - by making large-volume spam uneconomical for its propagators. Arguably at its most revolutionary, proof of work powers the transaction verification and currency issuance components of cryptocurrencies like Bitcoin and Litecoin, allowing for an entirely new form of money free from centralized institutions.