ABEL's Difficulty Smoothing Algorithm (DSA) - Whitepaper

Abelian Foundation

July 12, 2024

1 Background

1.1 Original Adjustment Algorithm

When the Abelian network was first launched, its Proof-of-Work (PoW) difficulty adjustment algorithm operated every 4000 blocks (we will refer to this as an epoch). At the end of every epoch - specifically at block heights 4000, 8000, 12000, and so on, it would approximate the network's mining power (hash rate) and then scale the upcoming mining difficulty up or down, so that miners would take an average of 256 seconds to mine a block in the next epoch.

In other words, the Abelian network aims to complete an epoch approximately every 12 days. If the latest epoch took over 12 days to complete, the network estimates the mining difficulty to be too high compared to the present hash rate, thus the next epoch's difficulty would be adjusted downwards. This would cause miners to take less time in mining a block, reducing the upcoming epoch's block time. On the contrary, if the latest epoch took less than 12 days to complete, the network would increase the upcoming epoch's difficulty, causing miners to take longer to mine each block and thereby increasing the epoch's completion time.

1.2 Limitations of Original Adjustment Algorithm

Up until June 2024, the Abelian network experienced significant fluctuations in hash rate from epoch to epoch, whereby miners would cyclically join the network during low-difficulty epochs and leave during high-difficulty epochs. This is because miners would switch between mining Abelian and other GPU-minable cryptocurrencies based on the networks' difficulty levels to optimize their mining profitability.

This cyclic mining behavior has caused drastic fluctuations in hash rates from epoch to epoch, which in turn caused significant fluctuations between average block times. For example, 2 epochs were completed in the period from April 25th to June 3rd, and the average block times in the first epoch were much higher than 256 seconds (peaks at approximately 1300 seconds), whereas the second epoch had a much lower average than 256 seconds (the minimum value is approximately 80 seconds).

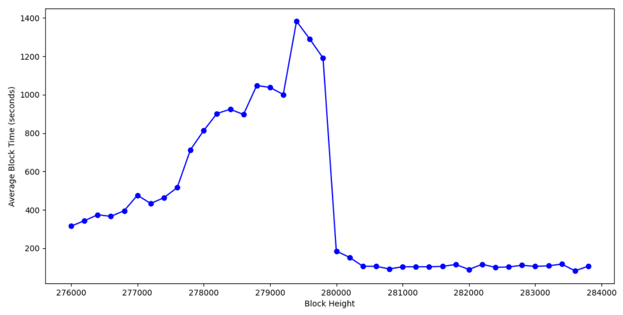

Figure 1: Average Block Time in the past 2 epochs

Such volatility is expected in a young PoW project, but it does pose some challenges to the network:

- **High Miner Influx**
  Block times would shorten, increasing the network's throughput. However, nodes with poor connection may struggle to synchronize, causing frequent soft-forks and wasting computational power.
- **High Miner Exodus**
  A drop in hash rate would result in an increase in a network's block time, thus increasing the network's transaction time and impacting the network's usability.

To address these issues, the Abelian Foundation established a task force in early 2024 to devise an improved difficulty adjustment algorithm, which will be introduced in the following section.

2 Difficulty Smoothing Algorithm (DSA)

The team has devised DSA as the new and improved difficulty adjustment algorithm.

2.1 How DSA Works

- **Slot-based Adjustment**
  DSA adjusts the network difficulty every 200 blocks (we call this a slot).
- **Weighted Average**
  DSA calculates the average block time for the latest 20 slots, and then computes the weighted average, giving the more recent slots a higher weight.
- **Adjustment Criteria**
  The upcoming slot's difficulty is increased if the weighted average block time is less than 256 seconds and is decreased if it is over 256 seconds.

2.2 Advantages of DSA

- **Frequent Adjustment**
  By adjusting every 200 blocks instead of 4000 blocks, the difficulty is updated approximately every 14 hours instead of 12 days.
- **Enhanced Responsiveness**
  The use of a weighted average that prioritizes recent slots over older ones ensures the network's responsiveness and adaptability to rapid fluctuations in the hash rate.
- **Improved Stability**
  Simulations show that using DSA would cause the block time to quickly converge and stabilize at the target of 256 seconds.

2.3 Software Updates

DSA has been in use since block height 284000, and users must upgrade their software to at least `v0.13.0 (Abec)` or `v4.0.0 (Desktop Wallet)` or `Abelian Pro (Android & iOS)` to use the network past this height. At present, all mining pools, exchanges, and major individual Abelian nodes have completed their upgrade process.

3 Technical Details

In this section, we will first give the definitions of some of the parameters used by DSA before giving a summary of DSA. After that, we will show how DSA works in-depth by demonstrating the calculation of `bits` and `target` for slot 284800-284999.

3.1 Parameters

The Abelian Explorer provides an API that we can use to query a block's details. For example, we can examine block 284800 at:

https://api.abelian.info/v1/block/284800

Let us examine the following parameters in particular:

```
time: 1718093710
bits: 1a69bad4
(seal)hash: 0000000000002169f2e9c6a24f6f2868a889abd194d28595b20d22bc766e5635
```

The `time` parameter refers to the time at which the block was mined (in Unix timestamp). When converted to a human-readable format, we see that block 284800 was mined at `11 June 2024 08:15:10 (GMT)`.

The `bits` parameter represents the `target` parameter of a block in compact form. We can convert `bits` to `target` like so:

$$\begin{tikzcd}
& \text{1a69bad4} \arrow[dl] \arrow[dr] & \\
\text{1a} \arrow[d] & & \text{69bad4} \arrow[d] \\
2^{8(\texttt{0x1a}-3)} \arrow[dr] & & \texttt{0x69bad4} \arrow[dl] \\
& 2^{8\cdot(\texttt{0x1a}-3)}\cdot\ \texttt{0x69bad4} \arrow[d] & \\
& \texttt{00000000000069bad40000000000000000000000000000000000000000000000} &
\end{tikzcd}$$

The `target` is an indicator of how small a block's `hash` must be for it to be valid. For example,

`hash` and `target` for block 284800, respectively:

`0000000000002169f2e9c6a24f6f2868a889abd194d28595b20d22bc766e5635`

$<$

`00000000000069bad40000000000000000000000000000000000000000000000`

When attempting to mine a block, a miner would first compute the `hash` value of its header, and then check if it's less than the block's `target`. If the check succeeds, they would have successfully mined a block.

Since the Abelian network uses a 256-bit hash, there are 2^{256} possible `hash` values, and the miner succeeds in mining a block for `target` possible values. This means the probability of successfully mining a block would be

$$p = \frac{\texttt{target}}{2^{256}}$$

and therefore the expected number of attempts until a miner succeeds would be

$$\frac{1}{p} = \frac{2^{256}}{\texttt{target}} \approx \frac{2^{256}}{\texttt{target}+1} \text{ to avoid division by 0}$$

We define `work` (for a single block) to be this number of expected attempts, and we define the `hash rate` as the amount of `work` done per second (across the entire network).

3.2 DSA Overview

At the end of each slot (every 200 blocks), DSA would first calculate a weighted average `hash rate` over the past 20 slots (giving the more recent slots a higher weight), then using the weighted `hash rate` to come up with a new `target` so that the upcoming slot's average block time would be 256 seconds, assuming the next slot's hash rate matches the predicted weighted hash rate.

First, given a `target` and `slot time` (number of seconds between the mining time of the first and last block in the slot), we can calculate a slot's `hash rate` like so:

$$\begin{aligned}\texttt{hash rate} &= \frac{\texttt{total work done}}{\texttt{slot time}} \\ &= \frac{\texttt{(work done per block)} \cdot \texttt{(number of blocks per slot)}}{\texttt{slot time}} \\ &= \frac{\left(\frac{2^{256}}{\texttt{target}+1}\right) \cdot 199}{\texttt{slot time}}\end{aligned}$$

Or equivalently,

$$\begin{aligned}\texttt{hash rate} &= \frac{\texttt{total work done}}{\texttt{slot time}} \\ &= \frac{\texttt{(total work until slot's last block)} - \texttt{(total work until slot's first block)}}{\texttt{(time of slot's last block)} - \texttt{(time of slot's first block)}}\end{aligned}$$

Once we calculate the `hash rate` of the past 20 slots ($\texttt{hash rate}_{i-20}$, $\texttt{hash rate}_{i-19}$, ..., $\texttt{hash rate}_{i-1}$), we then proceed with calculating the weighted average hash rate:

$$\alpha_i = \frac{1}{400} + \frac{i}{200} \text{ for } 0 \leq i < 20$$

$$\texttt{avg hash rate} = \sum_{j=0}^{19} \alpha_i \cdot \texttt{hash rate}_{(i-20)+j}$$

We then compare the ratio between `avg hash rate` and the most recent slot's hash rate ($\texttt{hash rate}_{i-1}$). If the ratio is below $\frac{1}{4}$ or above 4, we clamp it between these values.

Note: Assuming the hash rate of the next slot doesn't change from the most recent slot, and if the newly-calculated difficulty would increase the average block time to over $4 \cdot 256 = 1024$ seconds, or decrease it to below $\frac{256}{4} = 64$ seconds, this would clamp the average block time between these values.

$$\texttt{ratio} = \frac{\texttt{avg hash rate}}{\texttt{hash rate}_{i-1}}$$

$$\texttt{target hash rate} = \begin{cases} 4 \cdot \texttt{hash rate}_{i-1} & \text{if } 4 < \texttt{ratio} \\ \texttt{avg hash rate} & \text{if } \frac{1}{4} \leq \texttt{ratio} \leq 4 \\ \frac{1}{4} \cdot \texttt{hash rate}_{i-1} & \text{if } \texttt{ratio} < \frac{1}{4} \end{cases}$$

Finally, we 'predict' the next slot's `hash rate` would be `target hash rate`, and we would calculate the new `target` value accordingly. Given the relationship between `hash rate` and `target`:

$$\texttt{hash rate} = \frac{\left(\frac{2^{256}}{\texttt{target}+1}\right) \cdot 199}{\texttt{slot time}}$$

We wish for the upcoming `slot time` to be $199 \cdot 256$ seconds, therefore our new target would be:

$$\texttt{target hash rate} = \frac{\left(\frac{2^{256}}{\texttt{target}+1}\right) \cdot 199}{199 \cdot 256}$$

$$\implies \texttt{target} = \frac{2^{256}}{256 \cdot \texttt{target hash rate}} - 1$$

And finally, we would convert our `target` into compact form as `bits`, and use `bits`'s value in all future calculations and comparisons.

3.3 Illustrative Example

We demonstrate DSA by calculating the `target` for the slot 284800-284999.

First, we retrieve the parameters of the previous 20 slots from the Explorer's API and use them to calculate each slot's hash rate. For example, we see that slot 280800-280999 has the following parameters:

```
"bits":"1b00c247"
"time":1717092234 (Block 280800)
"time":1717110521 (Block 280999)
```

We convert `bits` into `target`:

$$\texttt{target} = \texttt{0x0c247} \cdot 2^{8\cdot(\texttt{0x1b}-3)}$$

And we calculate `slot time`:

$$\texttt{slot time} = 1717110521 - 1717092234 = 18287$$

Once we have `target` and `slot time`, we calculate `hash rate`:

$$\texttt{hash rate} = \frac{\left(\frac{2^{256}}{\left(\texttt{0x0c247}\cdot 2^{8\cdot(\texttt{0x1b}-3)}\right)+1}\right)\cdot 199}{18287} \approx 4036158491504$$

We repeat these steps for each of the past 20 slots:

Slot	Blocks	`bits`	`slot time`	`hash rate`
1st	280800-280999	1b00c247	18287	4036158491504
2nd	281000-281199	1b00c247	20504	3599747870373
3rd	281200-281399	1b00c247	20578	3586802912534
4th	281400-281599	1b00c247	20555	3590816362643
5th	281600-281799	1b00c247	20800	3548520689141
6th	281800-281999	1b00c247	22897	3223532791812
7th	282000-282199	1b00c247	17615	4190135131089
8th	282200-282399	1b00c247	23081	3197835030290
9th	282400-282599	1b00c247	19818	3724353130191
10th	282600-282799	1b00c247	20385	3620761851073
11th	282800-282999	1b00c247	22046	3347964725307
12th	283000-283199	1b00c247	20818	3545452509085
13th	283200-283399	1b00c247	21408	3447740579882
14th	283400-283599	1b00c247	23258	3173498595499
15th	283600-283799	1b00c247	16298	4528729312439
16th	283800-283999	1b00c247	21015	3512216527915

17th	284000-284199	1a4e4839	68246	2684065158980
18th	284200-284399	1a501029	168718	1061546137207
19th	284400-284599	1a55f13a	198210	841783374579
20th	284600-284799	1a5d1e3b	231375	665552551807

Table 1: Hash rates of the past 20 slots

Once we have the past 20 slots' hash rates, we calculate the weighted `avg hash rate`:

$$\texttt{avg hash rate} = \alpha_0 \cdot \texttt{hash rate}_{i-20} + \alpha_1 \cdot \texttt{hash rate}_{i-19} + \cdots + \alpha_{19} \cdot \texttt{hash rate}_{i-1}$$

Slot	α	`hash rate`	$\alpha\cdot$ `hash rate`
1st	0.0025	4036158491504	10090396228.7600
2nd	0.0075	3599747870373	26998109027.7975
3rd	0.0125	3586802912534	44835036406.6750
4th	0.0175	3590816362643	62839286346.2525
5th	0.0225	3548520689141	79841715505.6725
6th	0.0275	3223532791812	88647151774.8300
7th	0.0325	4190135131089	136179391760.3925
8th	0.0375	3197835030290	119918813635.8750
9th	0.0425	3724353130191	158285008033.1175
10th	0.0475	3620761851073	171986187925.9675
11th	0.0525	3347964725307	175768148078.6175
12th	0.0575	3545452509085	203863519272.3875
13th	0.0625	3447740579882	215483786242.6250
14th	0.0675	3173498595499	214211155196.1825
15th	0.0725	4528729312439	328332875151.8275
16th	0.0775	3512216527915	272196780913.4125
17th	0.0825	2684065158980	221435375615.8500
18th	0.0875	1061546137207	92885287005.6125
19th	0.0925	841783374579	77864962148.5575
20th	0.0975	665552551807	64891373801.1825
Total			2766554360071.5950

Table 2: Calculating the `avg hash rate`

We then compute the `ratio`:

$$\texttt{ratio} = \frac{2766554360071.5950}{665552551807} \approx 4.16$$

Since $\texttt{ratio} = 4.16 > 4$, we clamp `target hash rate` to $4 \cdot \texttt{hash rate}_{i-1} = 2662210207228$.

Finally, we can use `target hash rate` to compute the new `target`:

$$\begin{aligned}\texttt{target} &= \frac{2^{256}}{256 \cdot 2662210207228} - 1 \\ &= \texttt{00000000000069bad4c31f3cc44ce91d9a9b6b3744a15dae9ee647cf1cddd557}\end{aligned}$$

Now that we have the `target`, we can find the `bits` value by repeatedly dividing the `mantissa` by 256 and incrementing the `exponent` by 1 starting from 3, until `mantissa` < `0x00800000`.

`mantissa`	`exponent`
00000000000069bad4c31f3cc44ce91d9a9b6b3744a15dae9ee647cf1cddd557	0x3
0000000000000069bad4c31f3cc44ce91d9a9b6b3744a15dae9ee647cf1cddd5	0x4
000000000000000069bad4c31f3cc44ce91d9a9b6b3744a15dae9ee647cf1cdd	0x5
00000000000000000069bad4c31f3cc44ce91d9a9b6b3744a15dae9ee647cf1c	0x6
0000000000000000000069bad4c31f3cc44ce91d9a9b6b3744a15dae9ee647cf	0x7
000000000000000000000069bad4c31f3cc44ce91d9a9b6b3744a15dae9ee647	0x8
00000000000000000000000069bad4c31f3cc44ce91d9a9b6b3744a15dae9ee6	0x9
0000000000000000000000000069bad4c31f3cc44ce91d9a9b6b3744a15dae9e	0xa
000000000000000000000000000069bad4c31f3cc44ce91d9a9b6b3744a15dae	0xb
00000000000000000000000000000069bad4c31f3cc44ce91d9a9b6b3744a15d	0xc
0000000000000000000000000000000069bad4c31f3cc44ce91d9a9b6b3744a1	0xd
000000000000000000000000000000000069bad4c31f3cc44ce91d9a9b6b3744	0xe
00000000000000000000000000000000000069bad4c31f3cc44ce91d9a9b6b37	0xf
0000000000000000000000000000000000000069bad4c31f3cc44ce91d9a9b6b	0x10
000000000000000000000000000000000000000069bad4c31f3cc44ce91d9a9b	0x11
00000000000000000000000000000000000000000069bad4c31f3cc44ce91d9a	0x12
0000000000000000000000000000000000000000000069bad4c31f3cc44ce91d	0x13
000000000000000000000000000000000000000000000069bad4c31f3cc44ce9	0x14
00000000000000000000000000000000000000000000000069bad4c31f3cc44c	0x15
0000000000000000000000000000000000000000000000000069bad4c31f3cc4	0x16
000000000000000000000000000000000000000000000000000069bad4c31f3c	0x17
00000000000000000000000000000000000000000000000000000069bad4c31f	0x18
0000000000000000000000000000000000000000000000000000000069bad4c3	0x19
000000000000000000000000000000000000000000000000000000000069bad4	0x1a

Table 3: Mantissa & Bits

We find the `bits` value to be `1a69bad4`, and we will use the `bits`'s corresponding `target` value in future calculations (for example, calculating the `difficulty`):

`00000000000069bad40000000000000000000000000000000000000000000000`

Note - we can calculate the `difficulty` value as shown in the Explorer's API like so:

$$\texttt{difficulty} = \frac{M}{\texttt{target}}$$

where M is the maximum possible difficulty (whose `bits` is `1d017c38`).

For example, for slot 284800-284999,

$$\begin{aligned}\texttt{difficulty} &= \frac{\texttt{0x017c38} \cdot 2^{8\cdot(\texttt{0x1d}-3)}}{\texttt{00000000000069bad400000000000000000000000000000000000000000000000}} \\ &\approx 235676.38093908\end{aligned}$$

4 Effect on Network

Ever since the Abelian network started using DSA, the network block times have quickly converged to around the target of 256 seconds per block.

From June 24th to July 5th (at the time of writing), we see that the average block time doesn't fluctuate much when compared to before, with the minimum average block time being 166 seconds, and the maximum being 334 seconds.

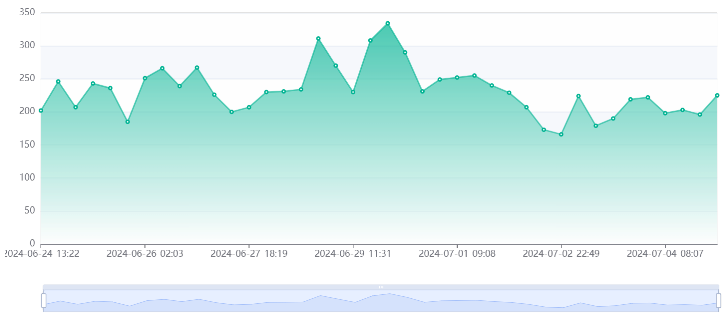

Figure 2: Average Block Time; Obtained from https://www.abelpool.io/en

We conclude by claiming that DSA has achieved its goals in improving the stability of the Abelian network, and in maintaining the average block time at the target of 256 seconds.