Prosper Mining Pool

A mining pool implementation for the Pegnet network.

Document Updated
October 25, 2019

Definitions

Some terms that relate to mining pools that will ease the reading of the document.

Hash-Rate

An amount of lxr hashing operations per second.

Target

8 byte hexadecimal value that comes from the top 8 bytes of the lxr hash of the `(oprhash+nonce)`. On bitcoin, the lower the target the better, on Pegnet it is the opposite, the higher the better. This is often stored as a uint64 for easy comparisons, but displayed in hex.

Difficulty

Difficulty is a floating point number that can describe the amount of work needed to generate a given target. The difficulty is proportional to the hashrate, meaning a target with difficulty 2 requires twice as many hashes (on average) as a target with difficulty 1. All difficulty numbers are related to a base difficulty 1, which is arbitrarily chosen. The reason we want to calculate Difficulty, is that it is easier to work with than large Targets.

pDiff

`Pool-Difficulty` is the base difficulty for determining the difficulty of work submitted to a mining pool. pDiff is currently set at the target `0xfffe000000000000`, which is about 6.5K h/s for 5s of work. So the target `0xfffe000000000000` has a difficulty of 1. All difficulty reported by the pool is in relation to pDiff. Meaning a difficulty of 100 is 100x the work of pDiff.

Share

A share is a proof of work found by a miner that has a high enough Target that the pool will accept it. When the pool accepts the share, it does not mean it will submit it to the Factom network, as the share target is always lower than the network's. The accepting of shares is to account for a miner's work.

Job

A block is a generic Pegnet block with a set of winners and rewards. A Job is the last pegnet block on the network that has not been built on top of yet, and therefore we can mine in the hopes of winning the block.

Reference Miner *pegnet term*

Refers to the public pegnet miner: https://github.com/pegnet/PegNet

Channel *golang term*

In GoLang a channel is a mechanism to send a message from 1 internal process to another. Many of the processes in the pool will sit idle unless they receive an external message over a channel. This design choice creates an event based system.

User Vs Miner *pool term*

A miner is a single `prosper-miner` process connected to the pool. A miner belongs to a user, and a User can have many miners. The User is the account associated with an entity that has signed up for the pool and operates miners.

Overview

The mining pool implementation, called “Prosper”, is intended to facilitate pooled mining on the Pegnet network. The Pegnet network has some unique properties compared to other PoW coins, such as having a fixed block time, and a variable target. This means that every block is 10 minutes long, and miners do not know if their PoW is enough until the end of the block. There are some other differences that will appear in the design decisions of the pool.

This document will not only detail design decisions, but also provide what additional improvements and work could be done with more time.

Running the Pool

Pool running instructions are provided in the repository with markdown files. The pool consists of a single daemon and a postgres instance. The pool daemon is a GoLang daemon that can be shipped as an executable, however for the primitive authentication UI, the static files need to also be provided. It is suggested to use the provided docker-compose.yml that will use the pool’s docker file to launch the daemon. The daemon requires access to a running factomd and a postgres instance.

ADMIN.md in the root directory has information about how to make an admin user, how to create new invite codes to join the pool, and how to make the PEG payouts. The amount of admin functions is currently limited, and requires access to the pool to execute. The best method to use the pool’s admin cli calls is to do so from within the container.

Tooling

The pool binary also operates as a cli tool to help configuration and some db interaction.

In addition to the pool binary, a small executable called `payout-cli` is also provided to construct and submit the PEG payments on the network. Documentation for using this is in the ADMIN.md

Connecting a Miner to the Pool

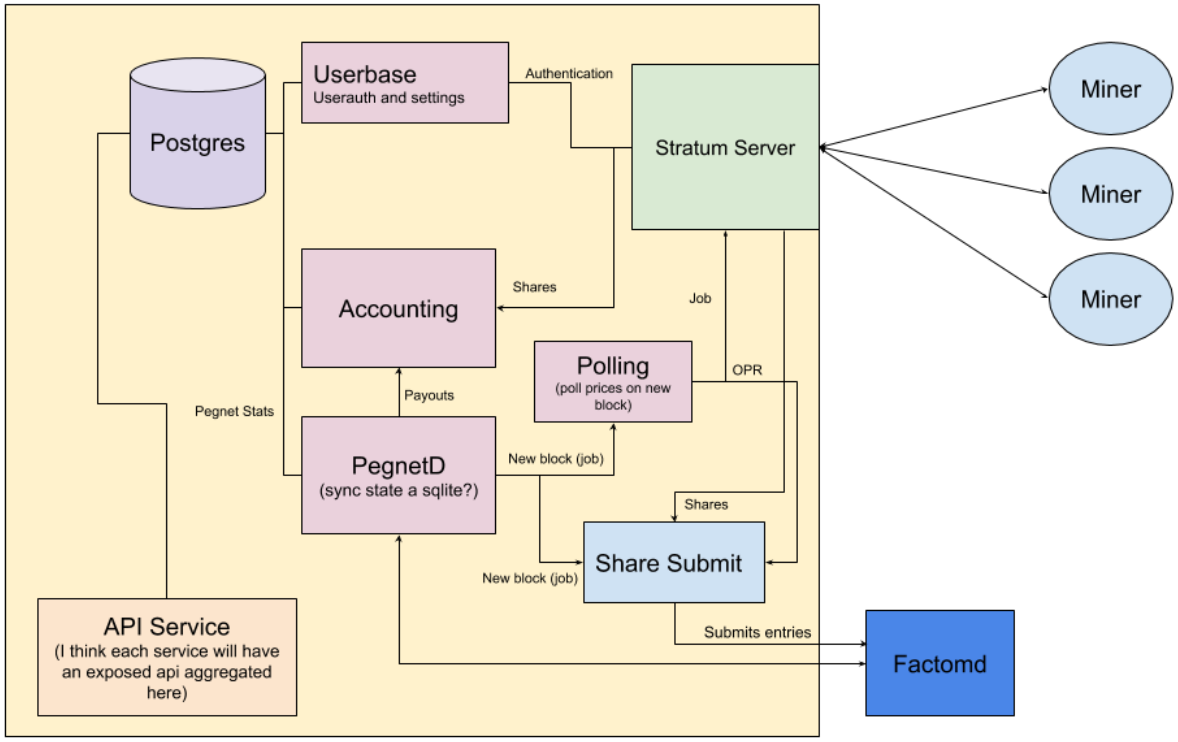
Currently the pool operates on an invite only basis. All miners need an invite code to register for the first time, then from there adding additional miners is trivial. The miner documentation is in the README.md in the `prosper-miner` directory.

The miner is a GoLang compiled executable. Downloads are provided in the Releases on the github repo. However, a change should be made to the miner before going to production. Currently the miners default mining pool server is localhost. The correct server address should be compiled in by default, such that a new miner does not need to specify this flag.

Technical Design

The pool consists of multiple modules. Almost all communication from module to module is done via a GoLang channel. Not shown below is an “engine” module in the code that links everything together. There are 2 sources for events in the system: PegnetD and the external Miners. PegnetD sends new job alerts throughout the system each time a new block is found. Miners also generate events when they find new shares to submit to the pool.

This documentation will detail some design decisions and functionality of each module. If any future development work is to be done on this software, this section is a recommended read.



*Design of the Prosper Pool. Anything in the yellow box is in the same GoLang process **except** the Postgres instance.*

Postgres

Postgres is used as the database for persistent storage. It is used to keep the PegnetD sync state, accounting payments/owed, and the user’s authentication information. All data in the database related to the blockchain is saved when a block is closed. There is no data in the database about a working block, meaning if the pool is shut down, the reboot will begin from the last closed block’s information. More on this in the Accounting section.

Userbase *improvements suggested*

Just stores users and their passwords with a salted bcrypt hash. Using https://github.com/qor/auth to save time and effort constructing a userbase plugin. If a more custom solution is needed, this can be replaced.

There is currently no method of changing user settings. Future work should expand on this.

Polling

A direct copy of the polling in the reference miner to grab asset price quotes from datasources.

PegnetD

Inputs: Polls factomd for block updates
Outputs: New & Old blocks, where new blocks are considered Jobs
SideEffects: Updates Postgres with pegnet syncing information

The pool runs an internal lightweight PegnetD daemon. It only tracks block rewards and mining activity, not transactions or conversions. The internal daemon only has 1 purpose, to find new blocks. When you run the pool, the pegnet daemon will begin syncing and flag each block is syncs as “new” or “old”. Only “new” blocks are started as a mining job, as there is no purpose in mining an old block.

Important: Unlike the reference miner, this pegnetd does not watch for minutes from factomd. This is an intentional design choice, as factomd followers have been seen to not always follow minutes properly. A polling mechanism on the last saved block is used to determine when a block is complete, so a follower following by dbstates can be used by the pool, where a normal reference miner would fail.

Accounting *improvements suggested*

Inputs: Rewards from PegnetD to know if we have won any PEG. Jobs from pegnetd to know which shares are valid. Shares from stratum to calculate user payouts.
Outputs:
SideEffects: Updates Postgres with accounting information on block boundaries

The Accountant is in charge of everything PEG. From PegnetD, the accountant is aware of how much PEG the pool is winning. To know how to pay out the PEG, all accepted shares from the miners are sent to the accountant, where the total work per **user** & per **miner** is kept in memory until the job is complete. Only the per user work is saved to disk at the end of the block.

All payments owed is calculated using a Proportional Payout strategy. The proportional payout strategy takes all the miner's work for the 10min period, takes the reward of that 10minute period, and pays the miners proportionally based on the work for that block. The pro of this strategy is that it is easy, however the con is that if your rewards are not consistent per block, then the miner ratios might be not 100% fair. An **improvement** would be to implement a scoring basing system like found in SlushPool: https://slushpool.com/help/reward-system/. The proportional approach was chosen because of time constraints.

Important: When the pool is turned off, all in memory information is lost. So all miner work for that in flight block is lost. The work on the pegnet network is NOT lost however because of *rolling submissions*. This means is you restart the pool, you could lose up to 10min of information for how to pay miners. This can be resolved by using the previous and next block's payout proportions to estimate what the miner's proportion work would have been.

Additionally, only work per user is stored. It might be beneficial to also store work per miner to provide more insight for the user's individual miners.

Stratum Server *improvements suggested*

Inputs: New jobs from pegnetd, new shares from miners.
Outputs: Shares from miners to the accounting and submissions.
SideEffects: Uses the userbase for miner authentication

Stratum is a popular pool communication specification for miners to communicate with a pool. The stratum server handles the set of miners connected over tcp. It forwards all jobs to the miners, and accepts all shares.

Currently the stratum server does not use **vardiff**. Meaning each miner is given the same minimum target, pDiff, to aim for and above. A vardiff approach considers the miner's hashrates when deciding the target they should aim for. This is so all miners will consume about the same amount of network bandwidth to the pool. A consequence of not using vardiff is that a miner with higher hashrate will consume more bandwidth. If bandwidth is an issue, pDiff can be raised. **Vardiff should be implemented in the future**. Because of time constraints it could not be added. A vardiff implementation will help bandwidth, but will break the current proportional payout strategy. The scoring based approach will be necessary before this improvement is implemented.

Important: Vardiff should be implemented as the pool scales. Another security concern The pool is not checking the PoW hash on shares at this time. If a miner edits their code, they can lie about their amount of work. This was chosen as the lxrhash is slow, and checking all work from all miners could seriously impact the pool's performance. A solution is to randomly sample miner's work, and finding an improper PoW will be punished by discrediting all their prior shares. This would reduce the overhead costs by whatever the sample rate is, and prevent a malicious miner.

Share Submit

Inputs: Shares from the miners, and jobs from pegnetd. Polls factomd for minutes if factomd is syncing by minutes.
Outputs: OPRs into the Factom Blockchain
SideEffects:

The share submit handles submitting oprs to the Factom network. It is the gatekeeper to determine if a share is worthy to be made into an entry.

Unlike the reference miner, the Pool does not necessarily track the minutes in Factomd. This is to handle node syncing issues, and still allow the pool to perform at some capacity. The reference miner submits at minute 9, whereas the pool uses a **rolling submission** strategy. The pool will compute the minimum target the network is reasonably likely to accept using the math found here: https://github.com/pegnet/pegnet/blob/master/utilities/simulate/DifficultyTarget.md. Unlike the reference miner, the pool uses an exponential moving average of the last 36 blocks (6hrs) to prevent any spiking. If the pool has a dominant portion of the network's hashrate, this strategy might result in over submitting records, and a new approach will be needed.

The downside to this strategy is that from minute 0 to minute 1, the old job is still being mined, but no longer accepted by the network. To accommodate this, a small process was added to detect this window if the node is syncing minutes, and will reject shares. If the node is not syncing by minutes, the window cannot be determined, and the pool will submit during this window. This was chosen as it is preferable to submit for 9 good minutes and 1 bad, then to not submit at all.

Important: Above describes that is is possible the pool will submit invalid OPRs to the network if the factomd is having syncing issues. The reference miner will idle in the case, the pool makes a best effort approach.

API + Web *improvements suggested*

There is currently little interaction to the pool provided. The web interface is primitive, and needs to be expanded. The API currently only has 1 endpoint that was used for some debugging. This section needs to be vastly improved.

The static files for the authentication library are also not ideally placed, as it uses the $GOPATH environment variable to find a specific repo.