PowerPool Documentation

Release 0.6.5-dev

Isaac Cook

September 17, 2015

Contents

CHAPTER 1

Features

- Lightweight, asynchronous, gevent based internals.
- Built in HTTP statistics/monitoring server.
- Flexible statistics collection engine.
- Multiple coinserver (RPC server) support for redundancy. Support for coinserver prioritization.
- Redis driven share logging allows multiple servers to log shares and statistics to a central source for easy scaling out.
- SHA256, X11, scrypt, and scrypt-n support
- Support for merge mining multiple auxilury (merge mined) blockchains
- Modular architecture makes customization simple(r)
- Support for sending statistics via statsd

Uses Redis to log shares and statistics for miners. Work generation and (bit|lite|alt)coin data structure serialization is performed by Cryptokit and connects to bitcoind using GBT for work generation (or getauxblock for merged work). Currently only Python 2.7 is supported.

Built to power the SimpleMulti mining pool.

CHAPTER 2

Indices and tables

2.1 Getting Setup

PowerPool is a Python application designed to be run on Ubuntu Linux, but will likely run in just about any Linux. If you're brave you might be able to get it running on Window, but I wouldn't recommend it since it's untested and unsupported.

2.1.1 Requirements

- **Redis** - For share/block logging and hashrate recording
- **Coinserver** - PowerPool builds mining jobs by running `getblocktemplate` or `getauxblock` on a Bitcoin Core, or bitcoin core like node. These docs will always refer to this as a "coinserver".
- **Miner** - To test out mining we recommend getting a cpuminer since it's easy to setup

2.1.2 Installation

```
mkvirtualenv pp  # if you've got virtualenvwrapper...
# Install all of powerpools dependencies
pip install -r requirements.txt
# Install powerpool
pip install -e .
# Install the hashing algorithm modules
pip install vtc_scrypt  # for scryptn support
pip install drk_hash  # for x11 support
pip install ltc_scrypt  # for scrypt support
```

Now copy `config.yml.example` to `config.yml`. Fill out all required fields and you should be good to go for testing.

```
pp config.yml
```

And now your stratum server is (or should be...) running. Point a miner at it on `localhost:3333` (or more specifically, `stratum+tcp://localhost:3333` and do some mining. View server health on the monitor port at `http://localhost:3855`. Various events will be recorded into Redis in a format that SimpleCoin is familiar with. See Simple Coin for a reference implementation of a frontend that is compaitble with PowerPool.

2.1.3 Production Use

There's no official guide at this point, but some general recommendations for new pool ops. Realize that unfortunately running a well optimized pool is complicated, so do your reading and don't become a hidden cost for your miners by being uneducated.

- Increase the number of connections on your coinserver with `maxconnections` configuration parameter. This helps you get notified of new blocks more quickly, leading to lower orphan rates.
- Recompile your coinserver from source with an increased `MAX_OUTBOUND_CONNECTIONS` in `net.cpp`. This will cause blocks that you solve to propogate to the network more rapidly.
- Increase `rpcthreads` configuration on coinservers. Generally you want at least few threads for the frontend (simplecoin_multi), and a few threads for each powerpool that connects to the server. If you are running polling instead of push block the rpcserver can become thread starved and block sumits, etc might fail.
- Setup Nagios to monitor your coinservers. This will help you know when they're getting slow or thread starved.
- Change your `stop-writes-on-bgsave-error` configuration to `no` for Redis, in case you run out of disk space. However you should setup a Nagios to make sure this isn't a normal occurance.
- Run PowerPool with `PYTHONOPTIMIZE=2` enviroment variable to skip all debugging computations/logging.
- Use a service like Nagios or Sensu to monitor your Stratum server ports with the `check_stratum.py` script in the contrib folder. Your miners appreicate good uptime.
- Use upstart or init.d to manage starting/stopping powerpool as a service. There is an example upstart config in the contrib folder.
- Use a firewall to block public access to your debugging port (`3855` by default..), since it contains sensative information.
- Read and understand the config.yml.example. It should be thoroughly commented and up to date, and if it's not open a ticket for us.

2.2 Setting up push block notification

To check for new blocks Powerpool defaults to polling each of the coinservers you configure. It just runs the rpc call 'getblockcount' 5x/second (configurable) to see if the block height has changed. If it has changed, it runs getblocktemplate to grab the new info.

Since polling creates a 100ms delay (on average) for detecting new blocks one optimization is to configure the coinservers to push PowerPool a notification when they accept a new block. Since this reduces the delay to <1ms you'll end up with fewer orphans. The impact of the faster speed is more pronounced with currencies that have shorter block times.

Although this is an improvement, its worth mentioning that it is pretty minor. We're talking about shaving off ~100ms or so, which should reduce orphan percentages by ~0.01% - 0.1%, depending on block times. Miners often connect with far more latency than this. The biggest reason to do this is to reduce the rpc load on your coinservers if there are multiple powerpool instances connected to them.

2.2.1 How push block works

Standard Bitcoin/Litecoin based coinservers have a built in config option to allow executing a script right after a new block is discovered. We want to run a script that notifies our PowerPool process(es) to check for a new block.

To accomplish this PowerPool has built in support for receiving a UDP datagram on its datagram port. The basic system flow looks like this:

```
Coinserver -> Learns of new block
Coinserver -> Executes blocknotify script (Alertblock)
Alertblock -> Parses the passed in .push file
Alertblock -> Sends a UDP datagram based on that .push file
PowerPool -> Receives UDP datagram
PowerPool -> Runs `getblocktemplate` on the Coinserver
```

Note: Using a pushblock script to deliver a UDP datagram to PowerPool can be accomplished in many different ways. We're going to walk through how we've set it up on our own servers, but please note if your server configuration/architechture differs much from ours you may have to adapt this guide.

2.2.2 Open the datagram port

The `datagram` option in powerpool's config is disabled by default because access to that port will allow users to remotely execute any command in your powerpool instance. It must be enabled in your powerpool config for any of this to do anything.

Warning: In production the datagram port should **always** be behind a firewall, as it is basically root access to your mining server.

2.2.3 Modify the coinserver's config

This is the part that tells the coinserver what script to run when it learns of a new block.

```
blocknotify=/usr/bin/alertblock /path/to/my.push
```

You'll want something similar to this in each coinserver's config. Make sure to restart it after.

2.2.4 Alertblock script

Now that the coin server is trying to run `/usr/bin/alertblock`, you'll need to make that Alertblock script.

Open your text editor of choice and save this to `/usr/bin/alertblock`. You'll also need to make it executable with `chmod +x /usr/bin/alertblock`.

```
#!/bin/bash
cat $1 | xargs -P 0 -d '\n' -I ARGS bash -c 'a="ARGS"; args=($a); echo "${args[@]:2}" | nc -4u -w0 -
# For testing the command
#cat $1 | xargs -P 0 -td '\n' -I ARGS bash -xc 'a="ARGS"; args=($a); echo "${args[@]:2}" | nc -4u -w
```

Note: Unfortunately netcat has a non-uniform implementation across different Linux platforms. Some platforms will require you to use "ncat" instead of "nc" in the above script.

2.2.5 Block .push script

Now your Alertblock script will be looking for a `/path/to/my.push` file. The data in this file is interpreted by the alertblock script. It looks at each line and tries to send a UDP packet based on the info. The .push file might contain something like this:

```
127.0.0.1 6855 VTC getblocktemplate signal=1 __spawn=1
```

The `127.0.0.1` and `6855` are the address and port to send the datagram to. The remaining stuff are the contents of the datagram. PowerPool's datagram format is basically:

```
<name of component> <function to run on component> \*<positional arguments for component> \*\*<keywo
```

The port (6855) should be the monitor port for the stratum process you want to send the notification to. The (VTC) should match the name of the coinserver component in the powerpool instance, normally the currency code.

If you need to push to multiple monitor ports just do something like:

```
127.0.0.1 6855 VTC getblocktemplate signal=1 __spawn=1
127.0.0.1 6856 VTC getblocktemplate signal=1 __spawn=1
```

For merge mined coins you'll want something slightly different:

```
127.0.0.1 6855 DOGE _check_new_jobs signal=1 _single_exec=True __spawn=1
```

2.2.6 Powerpool config

Now we need to update PowerPool's config to not poll, as it is no longer needed, and makes the coinserver's logs a lot harder to use. All that needs to be done is set the `poll` key to `False` for each currency you have push block setup for.

```
VTC:
    poll: False
    type: powerpool.jobmanagers.MonitorNetwork
    algo: scryptn
    currency: VTC
    etc...
```

2.2.7 Confirm it is working

You'll want to double check push block notifications are actually working as planned. The easiest way is to visit PowerPool's monitoring endpoint and look for the `last_signal` key. It should be updated each time PowerPool is notified of a block via push block.

Warning: If the server has poll turned off and is not getting push block notifications, you will get a LOT of orphans. In the future we would have polling automatically enable on failed push block, but right now it will just not update more than every 15 seconds!

2.2.8 Motivations

If this whole process seems complex that's because it is. Unfortunately it needs improvement. The reason for all this is that we can change which powerpool servers recieve push block notifications without needing to restart any powerpool servers or coinservers. A hardcoded implementation is simpler to setup, although more brittle, and requires service interruptions to add/remove instances and coins, which we don't want.

2.3 Components

2.3.1 Component Base

class powerpool.lib.**Component**

Abstract base class documenting the component architecture expectations. Each major part of powerpool inherits from this class.

_configure(*config*)

Applies defaults and checks requirements of component configuration

_incr(*counter*, *amount=1*)

_lookup(*key*)

defaults = {}

gl_methods = []

key = None

name

one_min_stats = []

one_sec_stats = []

start()

Called when the application is starting.

status

Should return a json convertable data structure to be shown in the web interface.

stop()

Called when the application is trying to exit. Should not block.

update_config(*updated_config*)

A call performed when the configuration file gets reloaded at runtime. self.raw_config will have bee pre-populated by the manager before call is made.

Since configuration values of certain components can't be reloaded at runtime it's good practice to log a warning when a change is detected but can't be implemented. Not currently used, but set aside for sunnier days.

2.3.2 PowerPool (manager)

class powerpool.main.**PowerPool**(*config*)

This is a singelton class that manages starting/stopping of the server, along with all statistical counters rotation schedules. It takes the raw config and distributes it to each module, as well as loading dynamic modules.

It also handles logging facilities by being the central logging registry. Each module can "register" a logger with the main object, which attaches it to configured handlers.

_tick_stats()

A greenlet that handles rotation of statistics

defaults = {'loggers': [{'type': 'StreamHandler', 'level': 'NOTSET'}], 'server_number': 0, 'datagram': {'host': '127.0

dump_objgraph()

Dump garbage collection information on SIGUSR1 to aid debugging memory leaks

exit (*signal=None*)
Handle an exit request

classmethod from_raw_config (*raw_config*, *args*)

gl_methods = ['_tick_stats']

handle (*data*, *address*)

log_event (*event*)
Sends an event to statsd

manager = None

register_logger (*name*)

register_stat_counters (*comp*, *min_counters*, *sec_counters=None*)
Creates and adds the stat counters to internal tracking dictionaries. These dictionaries are iterated to perform stat rotation, as well as accessed to perform stat logging

start ()

status
For display in the http monitor

2.3.3 Stratum Server

Handles spawning one or many stratum servers (which bind to a single port each), as well as spawning corresponding agent servers as well. It holds data structures that allow lookup of all StratumClient objects.

class powerpool.stratum_server.**StratumServer** (*config*)
A single port binding of our stratum server.

_spawn = None

add_client (*client*)

defaults = {'minimum_manual_diff': 64, 'reporter': None, 'server_seed': 0, 'vardiff': {'spm_target': 20, 'tiers': [8, 16

handle (*sock*, *address*)
A new connection appears on the server, so setup a new StratumClient object to manage it.

new_job (*event*)
Gets called whenever there's a new job generated by our jobmanager.

one_min_stats = ['stratum_connects', 'stratum_disconnects', 'agent_connects', 'agent_disconnects', 'reject_low_share

remove_client (*client*)
Manages removing the StratumClient from the luts

set_user (*client*)
Add the client (or create) appropriate worker and address trackers

start (**args*, ***kwargs*)

status
For display in the http monitor

stop (**args*, ***kwargs*)

class powerpool.stratum_server.**StratumClient** (*sock*, *address*, *logger*, *manager*, *server*, *reporter*, *algo*, *config*)
Object representation of a single stratum connection to the server.

DUP_SHARE = 1

DUP_SHARE_ERR = 22

LOW_DIFF_ERR = 23

LOW_DIFF_SHARE = 2

STALE_SHARE = 3

STALE_SHARE_ERR = 21

VALID_SHARE = 0

_incr (**args*)

_push (*job*, *flush=False*, *block=True*)
Abbreviated push update that will occur when pushing new block notifications. Mico-optimized to try and cut stale share rates as much as possible.

details
Displayed on the single client view in the http status monitor

error_counter = {24: 'not_authed_err', 25: 'not_subbed_err', 20: 'unk_err'}

errors = {20: 'Other/Unknown', 21: 'Job not found (=stale)', 22: 'Duplicate share', 23: 'Low difficulty share', 24: 'Una

last_share_submit_delta

push_difficulty ()
Pushes the current difficulty to the client. Currently this only happens uppon initial connect, but would be used for vardiff

push_job (*flush=False*, *timeout=False*)
Pushes the latest job down to the client. Flush is whether or not he should dump his previous jobs or not. Dump will occur when a new block is found since work on the old block is invalid.

read (**args*, ***kwargs*)

recalc_vardiff ()

send_error (*num=20*, *id_val=1*)
Utility for transmitting an error to the client

send_success (*id_val=1*)
Utility for transmitting success to the client

share_type_strings = {0: 'acc', 1: 'dup', 2: 'low', 3: 'stale'}

submit_job (*data*, *t*)
Handles recieving work submission and checking that it is valid , if it meets network diff, etc. Sends reply to stratum client.

summary
Displayed on the all client view in the http status monitor

2.3.4 Jobmanager

This module generates mining jobs and sends them to workers. It must provide current jobs for the stratum server to be able to push. The reference implementation monitors an RPC daemon server.

class powerpool.jobmanagers.monitor_aux_network.**MonitorAuxNetwork** (*config*)

_check_new_jobs (**args*, ***kwargs*)

defaults = {'signal': None, 'enabled': False, 'send': True, 'currency': 2345987234589723495872345L, 'rpc_ping_int':

found_block(*address*, *worker*, *header*, *coinbase_raw*, *job*, *start*)

gl_methods = ['_monitor_nodes', '_check_new_jobs']

one_min_stats = ['work_restarts', 'new_jobs']

start()

status

class powerpool.jobmanagers.monitor_network.**MonitorNetwork**(*config*)

_check_new_jobs(**args*, ***kwargs*)

_poll_height(**args*, ***kwargs*)

config = {'diff1': 176682010483171717894350283372783149619681025973119641754912509768237056OL, 'pool_address

defaults = {'diff1': 176682010483171717894350283372783149619681025973119641754912509768237056OL, 'pool_addr

found_block(*raw_coinbase*, *address*, *worker*, *hash_hex*, *header*, *job*, *start*)
Submit a valid block (hopefully!) to the RPC servers

generate_job(*push=False*, *flush=False*, *new_block=False*, *network='main'*)
Creates a new job for miners to work on. Push will trigger an event that sends new work but doesn't force a restart. If flush is true a job restart will be triggered.

getblocktemplate(*new_block=False*, *signal=False*)

new_merged_work(*event*)

one_min_stats = ['work_restarts', 'new_jobs', 'work_pushes']

start()

status
For display in the http monitor

2.3.5 Reporters

The reporter is responsible for transmitting shares, mining statistics, and new blocks to some external storage. The reference implementation is the CeleryReporter which aggregates shares into batches and logs them in a way designed to interface with SimpleCoin. The reporter is also responsible for tracking share rates for vardiff. This makes sense if you want vardiff to be based off the shares per second of an entire address, instead of a single connection.

class powerpool.reporters.base.**Reporter**
An abstract base class to document the Reporter interface.

add_block(*address*, *height*, *total_subsidy*, *fees*, *hex_bits*, *hash*, *merged*, *worker*, *algo*)
Called when a share is submitted with a hash that is valid for the network.

agent_send(*address*, *worker*, *typ*, *data*, *time*)
Called when valid data is recieved from a PPAgent connection.

log_share(*client*, *diff*, *typ*, *params*, *job=None*, *header_hash=None*, *header=None*, *start=None*, ***kwargs*)
Logs a share to external sources for payout calculation and statistics

class powerpool.reporters.redis_reporter.**RedisReporter**(*config*)

_queue_add_block(*address*, *height*, *total_subsidy*, *fees*, *hex_bits*, *hex_hash*, *currency*, *algo*, *merged=False*, *worker=None*, ***kwargs*)

_queue_agent_send(*address*, *worker*, *typ*, *data*, *stamp*)

_queue_log_one_minute(*address*, *worker*, *algo*, *stamp*, *typ*, *amount*)

_queue_log_share(*address*, *shares*, *algo*, *currency*, *merged=False*)

agent_send(**args*, ***kwargs*)

defaults = {'pool_report_configs': {}, 'redis': {}, 'attrs': {}, 'chain': 1}

gl_methods = ['_queue_proc', '_report_one_min']

log_share(*client*, *diff*, *typ*, *params*, *job=None*, *header_hash=None*, *header=None*, ***kwargs*)

one_sec_stats = ['queued']

status

2.3.6 Monitor

class powerpool.monitor.**ServerMonitor**(*config*)

Provides a few useful json endpoints for viewing server health and performance.

client(*comp_key*, *username*)

clients_0_5()

Legacy client view emulating version 0.5 support

clients_comp(*comp_key*)

comp(*comp_key*)

comp_config(*comp_key*)

counters()

debug()

defaults = {'DEBUG': False, 'JSONIFY_PRETTYPRINT_REGULAR': False, 'port': 3855, 'JSON_SORT_KEYS': F

general()

general_0_5()

Legacy 0.5 emulating view

handler_class

alias of CustomWSGIHandler

start(**args*, ***kwargs*)

stop(**args*, ***kwargs*)

Symbols

A

C

D

S

U

V