BTS Tools Documentation

Release 0.4.14

Nicolas Wack

Sep 18, 2017

Contents

The BTS Tools will help you build, run and monitor any graphene-based client (currently BitShares, Steem, Muse, PeerPlays).

Note: these tools were originally developed for the BitShares network, and later expanded to support any graphene-based network. This means that everywhere you will see BitShares mentioned in this documentation, it should be understood as BitShares, Steem, PeerPlays or Muse. Similarly, `bts` can be interchanged with `steem`, `ppy`, and `muse`.

There are 2 tools currently provided:

- a command line utility allowing to quickly build and run any graphene-based client
- a web application allowing to monitor a running instance of the client and send an email or push notification on failure

If you like these tools, please vote for witness wackou on the Steem, BitShares and Muse networks. Thanks!

To get started, just type the following in a shell:

```
$ pip3 install bts_tools
```

If you're not familiar with installing python packages or if you run into problems during installation, please visit the *Installing the tools* section for more details.

With the tools installed, you can refer to each section of the documentation for more information about how a certain aspect of the tools work.

Otherwise, if you prefer a more hands-on approach to setting up a delegate from scratch, please head to the following section: *How to setup a delegate - the easy tutorial [OBSOLETE]* [NOTE: deprecated]

CHAPTER 1

Documentation contents

Installing the tools

Installing dependencies for the tools

You will need some dependencies installed first before you can install the tools proper.

Linux

On Debian-derived OSes (Ubuntu, Mint, etc.), install with:

```
# apt-get install build-essential libyaml-dev python3-dev python3-pip lsof
```

Mac OSX

On OSX, you can install the dependencies like that:

```
$ brew install libyaml
```

Installing the tools

If the dependencies for the tools are properly installed, you should be able to install the bts_tools package with the following command:

```
$ pip3 install bts_tools
```

Note: You might need to run this as root on linux systems

Note: In general, when dealing with python packages, it is good practice to learn how to work with virtualenvs, as they make installing python packages more self-contained and avoid potential conflicts with python packages installed by the system. They do require to invest some time learning about them first, so only do it if you feel like you can dedicate that time to it. It is very recommended to do so, though, as it can potentially save you a few headaches in the future.

Please refer to the *Appendix: dealing with virtualenvs* section for more details.

Installing dependencies for building the BitShares command-line client

Even though the tools are properly installed and functional, you also need some dependencies for being able to compile the BitShares client.

The reference documentation for building the BitShares client can be found on the Graphene wiki

Linux

On Debian-derived systems, install them with:

```
# apt-get install build-essential git cmake libssl-dev libdb++-dev libncurses5-dev \
                  libreadline-dev libffi-dev libboost-all-dev
```

For Steem, you will also need the qt5 libs:

```
# apt-get install build-essential git cmake libssl-dev libdb++-dev libncurses5-dev \
                  libreadline-dev libffi-dev libboost-all-dev qt5-default qttools5-
↪dev-tools
```

Mac OSX

On OSX, you should install dependencies with brew instead of building your own, as the current libs in brew are recent enough to allow to compile the BitShares client. You will also need to force the install of `readline` system-wide and override OSX's native version, as it is antiquated.

```
$ brew install git cmake boost readline openssl autoconf automake libtool   # FIXME:␣
↪remove? berkeley-db
$ brew link --force readline
```

If you already had an "old" version of boost installed (< 1.55.0_2), please upgrade to a newer one:

```
$ brew upgrade boost
```

Installing dependencies for building the BitShares GUI client

To build the GUI client, you will need the same dependencies as for the command-line client, plus the following additional ones.

Linux

On Debian-derived systems, install them with:

```
# apt-get install qt5-default libqt5webkit5-dev qttools5-dev qttools5-dev-tools npm␣
↪nodejs-legacy
```

Mac OSX

TODO

Command-line tools

just run the `bts` script with the command you want to execute:

```
$ bts -h
usage: bts [-h] [-p PIDFILE] [-f]
           {version,clean_homedir,clean,build,build_gui,run,run_cli,run_gui,list,
↪monitor,deploy,deploy_node}
           [environment] [args [args ...]]

following commands are available:
  - version                : show version of the tools
  - clean_homedir          : clean home directory. WARNING: this will delete your␣
↪wallet!
  - save_blockchain_dir    : save a snapshot of the current state of the blockchain
  - restore_blockchain_dir : restore a snapshot of the current state of the blockchain
  - clean                  : clean build directory
  - build                  : update and build bts client
  - build_gui              : update and build bts gui client
  - run                    : run latest compiled bts client, or the one with the␣
↪given hash or tag
  - run_cli                : run latest compiled bts cli wallet (graphene)
  - run_gui                : run latest compiled bts gui client
  - list                   : list installed bts client binaries
  - monitor                : run the monitoring web app
  - deploy                 : deploy built binaries to a remote server
  - deploy_node            : full deploy of a seed or witness node on given ip␣
↪address. Needs ssh root access

Examples:
  $ bts build                 # build the latest bts client by default
  $ bts build v0.4.27         # build specific version
  $ bts build ppy-dev v0.1.8  # build a specific client/version
  $ bts run                   # run the latest compiled client by default
  $ bts run seed-test         # clients are defined in the config.yaml file

  $ bts build_gui   # FIXME: broken...
  $ bts run_gui     # FIXME: broken...



positional arguments:
  {version,clean_homedir,clean,build,build_gui,run,run_cli,run_gui,list,monitor,
↪deploy,deploy_node}
```

```
                        the command to run
  environment           the build/run environment (bts, pts, ...)
  args                  additional arguments to be passed to the given command

optional arguments:
  -h, --help            show this help message and exit
  -p PIDFILE, --pidfile PIDFILE
                        filename in which to write PID of child process
  -f, --forward-signals
                        forward unix signals to spawned witness client child process

You should also look into ~/.bts_tools/config.yaml to tune it to your liking.
```

Building and running the BitShares command-line client

To build and run the command-line client, you can use the following two commands:

```
$ bts build
$ bts run
```

By default, `bts build` will build the latest version of the BitShares client (available on the master branch). If you want to build a specific version, you can do so by specifying either the tag or the git hash.:

```
$ bts build 0.5.3
$ bts build 8c908f8
```

After the command-line client is successfully built, it will be installed in the `bin_dir` directory as defined in the `build_environments` section of the `config.yaml` file. The last built version will also be symlinked as `witness_node` in that directory, and this is the binary that a call to `bts run` will execute.

You can see a list of all binaries available by typing:

```
$ bts list
```

Passing additional arguments to "bts run"

You can pass additional arguments to "bts run" and the tools will forward them to the actual invocation of the bts client. This can be useful for options that you only use from time to time, eg: re-indexing the blockchain, or clearing the peer database. If they are args that start with a double dash (eg: –my-option), then you need to also prepend those with an isolated double dash, ie:

```
$ bts run -- --resync-blockchain --clear-peer-database
```

otherwise, the "–resync-blockchain" and "–clear-peer-database" would be considered to be an option for the bts script, and not an argument that should be forwarded.

Building and running the BitShares GUI client

[FIXME: currently broken]

To build and run the GUI client, the procedure is very similar to the one for the command-line client:

```
$ bts build_gui
$ bts run_gui
```

There is one major difference though: the GUI client will not be installed anywhere and will always be run from the build directory. This is done so in order to be as little intrusive as possible (ie: not mess with a wallet you already have installed) and as install procedures are not as clear-cut as for the command-line client.

Monitoring web app

Launch the monitoring web app locally

The main entry point to the monitoring app is the `~/.bts_tools/config.yaml` file. You should edit it first and set the values to correspond to your witness's configuration. See the *Format of the config.yaml file* page for details.

If this file doesn't exist yet, run the tools once (for instance: `bts -h`) and it will create a default one.

To run the debug/development monitoring web app, just do the following:

```
$ bts monitor
```

and it will launch on `localhost:5000`.

Setting up on a production server

For production deployments, it is recommended to put it behind a WSGI server, in which case the entry point is `bts_tools.wsgi:application`.

Do not forget to edit the `~/.bts_tools/config.yaml` file to configure it to suit your needs.

Example

We will run the monitoring tools using nginx as frontend. Install using:

```
# apt-get install nginx uwsgi uwsgi-plugin-python3
```

The tools will have to be run from a virtualenv, so let's create it:

```
$ mkvirtualenv -p `which python3` bts_tools
$ pip3 install bts_tools
```

Edit the following configuration files:

`/etc/uwsgi/apps-available/bts_tools.ini` (need symlink to `/etc/uwsgi/apps-enabled/bts_tools.ini`)

```
[uwsgi]
uid = myuser
gid = mygroup
chmod-socket = 666
plugin = python34
virtualenv = /home/myuser/.virtualenvs/bts_tools
enable-threads = true
lazy-apps = true
workers = 1
```

```
module = bts_tools.wsgi
callable: application
```

Note: The important, non-obvious, fields to set in the uwsgi config file are the following:

- set `enable-threads = true`, otherwise you won't get the monitoring thread properly launched
- set `lazy-apps = true`, otherwise the stats object will not get properly shared between the master process and the workers, and you won't get any monitoring data
- set `workers = 1`, otherwise you will get multiple instances of the worker thread active at the same time

The `virtualenv` field also needs to be setup if you installed the tools inside one, otherwise you can leave it out.

`/etc/nginx/sites-available/default` (need symlink to `/etc/nginx/sites-enabled/default`)

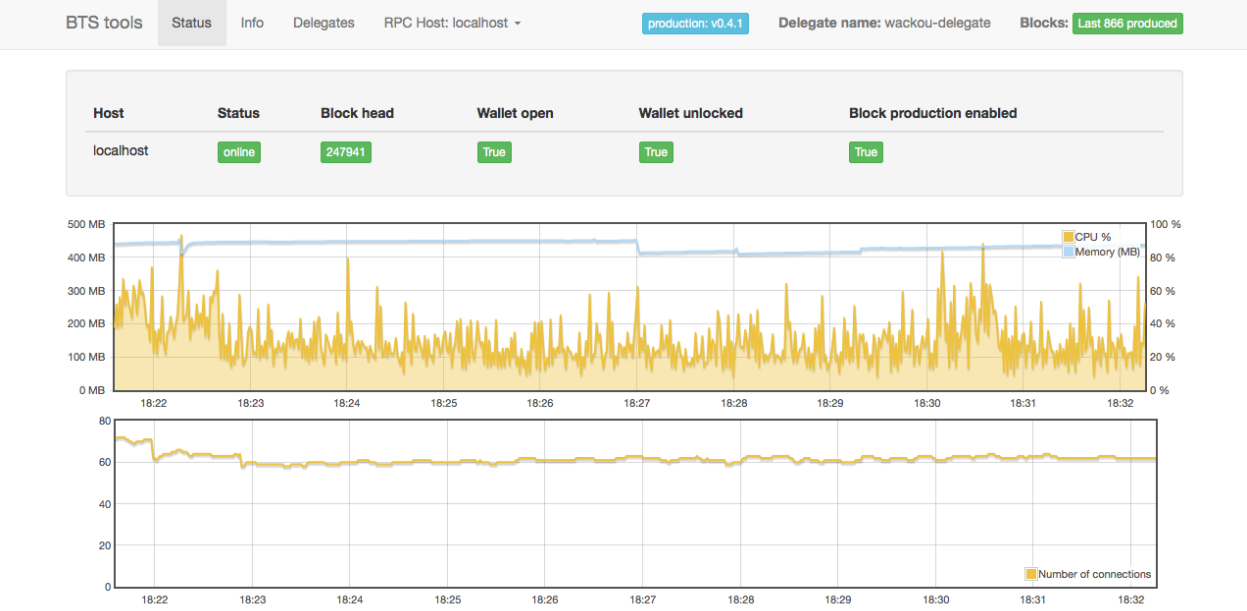
```
server {
        listen 80;
        server_name myserver.com;
        charset utf-8;
        location / { try_files $uri @bts_tools; }
        location @bts_tools {
                # optional password protection
                #auth_basic "Restricted";
                #auth_basic_user_file /home/myuser/.htpasswd;
                include uwsgi_params;
                uwsgi_pass unix:/run/uwsgi/app/bts_tools/socket;
        }
}
```

After having changed those files, you should:

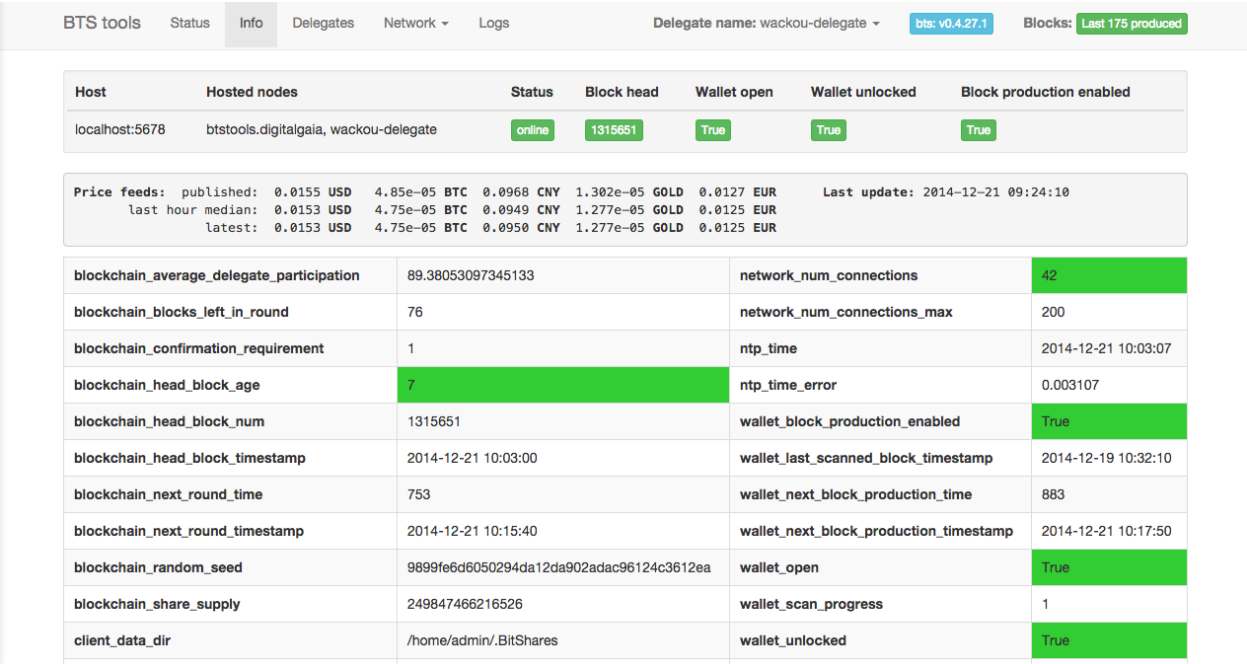
```
# service uwsgi restart
# service nginx restart
```

Screenshots

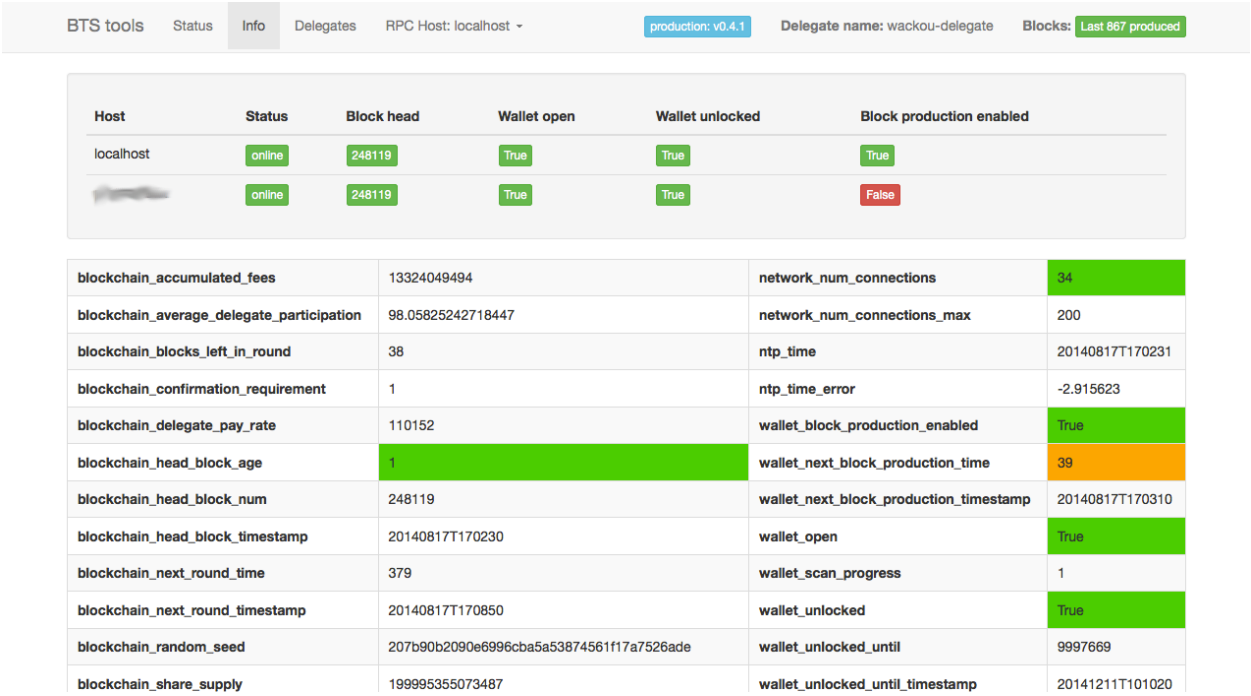
Monitoring the status of your running bts client binary:



You can host multiple delegates accounts in the same wallet, and check feed info:



Monitoring multiple instances (ie: running on different hosts) at the same time, to have an overview while running backup nodes and re-compiling your main node:



Working with various Graphene clients

The BTS Tools have originally been developed for managing BitShares witnesses (delegates at the time), but thanks to Graphene providing a common substrate for blockchains they are now able to handle all the following blockchains:

- `bts`: BitShares
- `steem`: Steem
- `ppy`: PeerPlays
- `muse`: Muse

Support for the various Graphene clients is built-in directly in the `bts` cmdline tool, and you only need to specify the corresponding build environment, e.g.:

```
$ bts build steem     # build the Steem client
$ bts run muse        # run the Muse client
```

If you don't specify a environment, it will use the `bts` environment by default. As a convenience feature, the following aliases to the `bts` command-line tool are provided:

- `steem`: uses the Steem environment by default
- `ppy`: uses the PeerPlays environment by default
- `muse` : uses the Muse environment by default

This means that:

```
$ steem build
```

and

```
$ bts build steem
```

are exactly equivalent.

Working with other types of nodes

Originally, the focus of the tools has been on maintaining witness nodes on the BitShares network, but they now support more types of specialized nodes.

Concretely, you can now manage the following types of nodes:

- witness
- feed_publisher
- seed
- backbone [FIXME: not implemented yet!]

You can define which type of node to run using the `roles` directive of the client configuration (see *Core concepts / architecture*)

Seed nodes are public, do not need an open wallet to run and usually have a high number of network connections.

Feed publishers will check feed prices on external feed providers and provide a feed aggregate to publish on the blockchain.

Backbone nodes are public, do not need an open wallet to run, and do not perform peer exchange (in order to hide IP addresses of the witnesses connected to them). They also try to maintain at all time an open connection to all other backbone nodes in order to have the backbone being a fully-connected graph of its nodes.

Command-line arguments and monitoring plugins are automatically defined depending on the type of node (seed or backbone), so config should be straightforward.

Core concepts / architecture

In order to better understand how the tools are structured and work internally, here are a few key concepts that should be understood:

- there are a number of blockchains for which we know a git repo that we can use to build a client. these are called "build environments", and currently can be any one of [bts, steem, ppy, muse]
- when a binary is compiled, you want to run an instance of it. This is called a "client" and each has its own data dir, network ports, etc. so you can run more than 1 simultaneously.
- furthermore, each client can assume 1 or more "roles", which describe its function to the network and will tune the client in order to better fulfill it. Currently: [witness, feed_publisher, seed] # planned: API node

Build environments

These are the types of clients that you can build. `bts`, `steem`, `ppy`, etc.

The full list can be found here: https://github.com/wackou/bts_tools/blob/master/bts_tools/templates/config/build_environments.yaml

Clients

A client definition contains all the information needed to launch a witness client, a cli_wallet that connects to it, and have the bts_tools web app monitor all of it.

A client has a name and defines at least the following properties:

- **type**: the type of build you want to run. Needs to be a valid build env (ie: bts, steem, ...)
- **data_dir**: *[required]* the data dir (blockchain, wallet, etc.) of the bts client
- **api_access**: the location of the api_access.json file. It contains authentication data for the RPC communication and needs to be created according to this spec
- **witness_user**: user for authentication with the witness client
- **witness_password**: password for authentication with the witness client

A list of default clients can be found here: https://github.com/wackou/bts_tools/blob/master/bts_tools/templates/config/clients.yaml

To run a specific client, type:

```
$ bts run client_name
```

If you don't specify a client on the command-line (ie: `bts run`), the tools will run the client using the `bts` run environment by default.

Roles

The roles are the main mechanism with which you control what type of monitoring plugins are launched for a given client.

Roles also contain additional information (e.g.: witness name, id and signing key) that allow the client to be monitored more efficiently (the web UI will show whether a witness signing key is currently active, for instance)

A client can assume one or more roles, and these are the roles that you can use:

- the **witness** role monitors a valid witness on the network, whether it is signing blocks (ie: its signing key is active) and activates the following monitoring plugins: `missed`, `voted_in`, `wallet_state`, `network_connections`, `fork`
- the **feed_publisher** role activates the `feeds` monitoring plugin that checks feed prices on external feed providers and publishes an aggregate price on the blockchain. Note: for feed publishing to work, you need to have an unlocked wallet in order to be able to publish the transaction.
- the **seed** role activates the `seed`, `network_connections` and `fork` monitoring plugin which will make the client monitor network health and increase its number of connections to higher than normal in order to better serve as an entry point to the network

Monitoring plugins

As a reference, here are the monitoring plugins that can be activated via the roles:

- `seed`: will set the number of desired/max connections as specified in the `monitoring` config section
- `feeds`: check price feeds and publish them
- `missed`: check for missed blocks for a witness
- `network_connections`: check that number of active connections to the network is higher than a threshold

- `payroll`: periodically distribute delegate pay amongst the configured accounts in the `monitoring` section.
- `wallet_state`: check when wallet is opened/closed and locked/unlocked
- `fork`: tries to detect whether client is being moved to a minority fork
- `voted_in`: check when a witness is voted in/out
- `cpu_ram_usage`: always active, monitor CPU and RAM usage of the witness client
- `free_disk_space`: always active, check whether the amount of free disk space falls below a threshold

Format of the config.yaml file

The `config.yaml` file is the file where you configure the tools to match your environment. It is located in `~/.bts_tools/config.yaml`. If you properly fill in all the values in it, there is no need for you to edit anything in the data-dir of the witness client (ie: no config.ini editing) and the tools can manage everything from the command line. This allows you to keep all the information in one single file, much easier to manage.

Instead of going over all the parameters in an arbitrary order and explaining them, we will just show a working example file that is fully commented and serves as the reference documentation.

Please make sure that you fully understand some *Core concepts / architecture* before proceeding.

Also know that there are a lot of other options under the hood, but in order to not be overwhelming, those have default values and are contained in a bigger config.yaml file. At startup, your config file is merged into it before being used by the application. If you want to see the full configuration file generated from it, it is located in `~/.bts_tools/full_config.yaml`

And now, without further ado, here is the reference `config.yaml` file:

config.yaml

```
---
# For a more detailed description of the format of this file, visit:
# https://bts-tools.readthedocs.io/en/latest/config_yaml.html
#

hostname: xxxxxxxx  # [OPTIONAL] label used as source of notification messages

# the logging levels for the different submodules
# can be any of DEBUG, INFO, WARNING, ERROR
logging:
    bts_tools.cmdline: DEBUG
    bts_tools.feeds: INFO

#
# These are the types of clients that you can build. `bts`, `steem`, `ppy`, etc.
# Mostly the default should work and you shouldn't need to change those values.
# Still, if you want to add other git remote repositories to pull from or
# extra flags to use for compiling, this is the place to do so.
#
# The full list of build_environments can be found here:
# https://github.com/wackou/bts_tools/blob/master/bts_tools/templates/config/build_
↪environments.yaml
#
build_environments:
```

```
    cmake_args: []                              # [OPTIONAL] shared among all build␣
↪environments
    make_args: ['-j4']                          # [OPTIONAL] shared among all build␣
↪environments
    bts:
        cmake_args: []                          # [OPTIONAL] build environment specific
        witness_filename: witness_node          # [OPTIONAL] filename for the compiled␣
↪witness binary
        wallet_filename: cli_wallet             # [OPTIONAL] filename for the compiled␣
↪wallet binary
    steem:
        cmake_args: ['-DLOW_MEMORY_NODE=ON']  # [OPTIONAL] see https://github.com/
↪steemit/steem/blob/master/doc/building.md
    muse:
        boost_root: ~/boost1.60_install       # [OPTIONAL] if boost was installed␣
↪manually
        cmake_args: []  #'-DBUILD_MUSE_TEST=ON']


#
# list of clients (witness accounts / seed nodes) that are being monitored
#
# the following file tries to keep a balanced distribution of the options between␣
↪each client
# in order to make for easier reading, but (unless otherwise noted) all the options␣
↪declared
# in a client also apply in the other ones
#
clients:
    bts:
        type: bts                           # [REQUIRED] build environment used for␣
↪compiling the binary [bts, muse, steem, ppy, etc.]
        data_dir: ~/.BitShares2             # [REQUIRED] the data dir of the client is␣
↪where the blockchain data and the wallet file are stored
        api_access: ~/api_access.json       # [REQUIRED] api_access.json should be␣
↪created according to https://github.com/BitShares/bitshares-2#accessing-restricted-
↪apis
        seed_nodes:                         # [OPTIONAL]
        - bts-seed1.abit-more.com:62015
        - seed.bitsharesnodes.com:1776
        p2p_port: 1778                      # [OPTIONAL] network port to be used for␣
↪p2p communication of the witness node
        track_accounts:                     # [OPTIONAL] [bts only] reduce RAM usage by␣
↪specifying the only accounts that should be tracked
        - 1.2.1       # witness-account
        - 1.2.111226  # bittwenty
        - 1.2.126782  # bittwenty.feed
        witness_host: localhost             # [OPTIONAL] defaults to 'localhost', but␣
↪can be a remote host if desired
        witness_port: 8090                  # [OPTIONAL] some defaults are provided for␣
↪each different chain
        witness_user: xxxxxxxx              # [REQUIRED] as in api_access.json
        witness_password: xxxxxxxx          # [REQUIRED] as in api_access.json
        wallet_host: localhost              # [OPTIONAL] defaults to 'localhost', but␣
↪can be a remote host if desired
        wallet_port: 8093                   # [OPTIONAL] some defaults are provided for␣
↪each different chain
        wallet_password: xxxxxxxx           # [OPTIONAL] only needed for feed␣
↪publishing  # FIXME: not true, currently unused param
```

```
        notification: email, telegram     # [OPTIONAL] type of notification channel␣
↪to be used: [email, boxcar, telegram]

        # roles are optional, and you can have as many as you want for a given client
        # a client that doesn't define any role will not be monitored by the web␣
↪interface,
        # but can still be used for building and running a client
        roles:
        -
            role: witness
            name: xxxxxxxx           # [REQUIRED] name of the witness account on the␣
↪blockchain
            witness_id: 1.6.xxx      # [REQUIRED]
            signing_key: 5xxxxxxxx   # [REQUIRED] private key used by this witness␣
↪for signing
        -
            role: feed_publisher
            name: xxxxxxxx           # [REQUIRED] name of the account that publishes␣
↪the feed on the blockchain
        -
            role: seed
            name: seed01             # [REQUIRED] name has no relevance for seed␣
↪nodes, except for identifying them in the UI

    # most of the clients try to have sensible defaults as much as possible, but you
    # need to specify at least: type, data_dir, api_access, witness_user, witness_
↪password
    muse:
        type: muse
        data_dir: ~/.Muse
        witness_user: xxxxxxxx       # defined in api_access.json
        witness_password: xxxxxxxx   # defined in api_access.json
        roles:
        -
            role: seed
            name: seed-muse

    steem:
        type: steem
        data_dir: ~/.Steem
        shared_file_size: 80G                                          # [OPTIONAL] size␣
↪for the shared memory file
        run_args: ['--replay-blockchain']                              # [OPTIONAL]␣
↪additional args for running the witness client
        run_cli_args: ['--rpc-http-allowip', '127.0.0.1']              # [OPTIONAL]␣
↪additional args for running the cli wallet
        plugins: ['witness']                                           # [OPTIONAL]␣
↪defaults to ['witness', 'account_history']
        seed_nodes: ["52.74.152.79:2001", "212.47.249.84:40696", "104.199.118.92:2001
↪", "gtg.steem.house:2001"]
        # FIXME: implement me!
        override_default_seed_nodes: true  # [OPTIONAL] [default=false] if true, the␣
↪client will only use the given seed nodes, otherwise it adds them to the list of␣
↪built-in seed nodes

        # Steemd (and Mused) can now accept the contents of the api_access.json␣
↪directly as argument on the command line,
        # hence the field api_access.json is not required anymore here (and you don't␣
↪need to create the file either),
```

```
        # as bts_tools will generate the proper arguments from the user and password␣
↪given here.
        witness_user: xxxxxxxx
        witness_password: xxxxxxxx
        notification: telegram
        roles:
        -
            role: witness  # for steem only, the 'witness_id' field is not required,␣
↪only 'name' and 'signing_key'
            name: xxxxxxxx
            signing_key: 5xxxxxxxx

    ppy:
        type: ppy
        data_dir: ~/.PeerPlays
        seed_nodes: ['213.184.225.234:59500']
        p2p_port: 9777
        witness_host: localhost
        witness_port: 8590
        witness_user: xxxxxxxx
        witness_password: xxxxxxxx
        wallet_host: localhost
        wallet_port: 8593
        api_access: ~/api_access.json
        roles:
        -
            role: seed
            name: seed01ppy


#
# configuration of the monitoring plugins
# most default values should work, see reference here:
# https://github.com/wackou/bts_tools/blob/master/bts_tools/templates/config/
↪monitoring.yaml
#
monitoring:
    seed:
        desired_number_of_connections: 200
        maximum_number_of_connections: 400

    feeds:
        # some assets are not published by default as they are experimental or have␣
↪some requirements
        # eg: need to be an approved witness to publish
        # if you want to publish them you need to say so explicitly here
        enabled_assets: [RUBLE, ALTCAP, HERO]

        # explicitly disable some assets (eg: due to black swan)
        #disabled_assets: [RUB, SEK, GRIDCOIN, KRW, SGD, HKD, BTWTY]

        check_time_interval: 300            # interval at which the external feed␣
↪providers should be queried, in seconds
        median_time_span: 1800              # leave default

        # if you have at least 1 feed_publisher role defined in your clients, then
        # you need to uncomment at least one of the next 2 lines
        #publish_time_interval: 2400        # use this to publish feeds at fixed time␣
↪intervals (in seconds)
```

```
        #publish_time_slot: 08              # use this to publish every hour at a␣
↪fixed number of minutes (in minutes)

        steem_dollar_adjustment: 1.0

#
# configuration of the notification channels
#
notification:
    email:
        smtp_server: smtp.example.com
        smtp_user: user
        smtp_password: secret-password
        identity: "BTS Monitor <bts_monitor@example.com>"
        recipient: me@example.com

    boxcar:
        tokens: [xxxxxxxx, xxxxxxxx]

    telegram:
        token: xxxxxxxx         # create your Telegram bot at @BotFather (https://
↪telegram.me/botfather)
        recipient_id: xxxxxxxx  # get your telegram id at @MyTelegramID_bot (https://
↪telegram.me/mytelegramid_bot)


#
# List of credentials for external services used by bts_tools
#
credentials:
    bitcoinaverage: # developer account recommended, otherwise need to lower
↪'monitoring.feeds.check_time_interval'
        secret_key: xxxxxxxx
        public_key: xxxxxxxx

    geoip2:
        user: xxxxxxxx
        password: xxxxxxxx

    currencylayer:
        access_key: xxxxxxxx

    quandl:
        api_key: xxxxxxxx
```

Appendix: dealing with virtualenvs

When dealing with python packages, it is possible to install them as root and make them available for the entire system. This is not always recommended as it can sometimes cause conflicts with the packages installed by your OS.

In the python world a solution to deal with that problem has emerged and allows to create sandboxes in which to install python packages, so that they do not interfere with those of the system. These sandboxes are called virtualenvs, short for "virtual environments".

Although very powerful, the usage of the bare virtualenv functionality can sometimes be cumbersome, so it is very recommended to use another project instead that gives you an easier API to work with: virtualenvwrapper

The main commands that virtualenvwrapper provides are the following:

- `mkvirtualenv` creates a new virtualenv (`rmvirtualenv` deletes it)
- `workon` allows to "activate" a virtualenv, meaning all packages that you install after that will be installed inside this virtualenv, and they will take precedence over those of the system (basically, they will be active). (use `deactivate` to stop using it)

Example

If you want to create a new virtualenv with python3 being used as interpreter of choice, you would run the following:

```
$ mkvirtualenv -p `which python3` bts_tools
```

Note that after creating it, the virtualenv is already active, so you don't need to call `workon bts_tools` right after creating it. You will have to do it next time you reboot or open a shell, though.

If you then run the following:

```
$ pip install bts_tools
```

it will install the tools inside the virtualenv, and won't interfere with your system.

Appendix: tips and tricks

This is a collection of various tips and tricks that didn't fit in any particular section, but that you probably want to know, or at least want to know that they exist :)

Use clang as a compiler on linux instead of gcc

When running on debian/ubuntu, the best solution (the "native" one) is to change your default compiler, like that:

```
sudo apt-get install clang
sudo update-alternatives --config c++
```

Alternatively, you can set the following in your `config.yaml` file:

```
CONFIGURE_OPTS = ['CC=/usr/bin/clang', 'CXX=/usr/bin/clang++']
```

Compiling Steem for low memory consumption

Add the following to your `config.yaml` file:

```
build_environments:
    steem:
        cmake_args: ['-DLOW_MEMORY_NODE=ON']
```

How to setup a delegate - the easy tutorial [OBSOLETE]

This guide will try to show you how to setup all the infrastructure needed to build, run and monitor a delegate easily and effortlessly. We will start from scratch, and end up with a fully functional delegate client running and being monitored for crashes and missed blocks.

Note also that this guide will not only show you the mininum number of steps required to get it working once, but it will try to guide you into using best practices and useful tools that make maintenance of the delegate over time a seamless experience. (For the curious, that means using virtualenvs, tmux, etc... If you have no idea what these are, don't worry, we'll get to it)

Once everything it setup properly, building the latest version of the client, running it, and launching the monitoring webapp that publishes feeds and sends you notifications is just a matter of:

```
$ bts build
$ bts run
$ bts monitor
```

In details, these are the following steps that this guide will cover:

Setup the base OS and build environment

Linux

We will do the install on Debian Jessie. It should be very similar on Ubuntu, however some packages might have slightly different names. Adapting it for Ubuntu is left as an exercise to the reader.

Note about the text editor: there are countless wars about which editor is the best, but ultimately it is up to you to pick the one you like best, so no details will be provided when needing to edit text files, it will just be mentioned that you need to do it.

Install base up-to-date OS

Installing the base OS will depend on your VPS provider, so check their documentation for that. The tutorial will use Debian Jessie as base distro, so you should install it directly whenever possible. You can download the most recent release of the Debian Jessie installer here, preferably the netinst version.

Install required dependencies

The first step is to install the dependencies necessary for installing the tools and for compiling the BitShares client (still as root):

```
# apt-get install build-essential git cmake libssl-dev libdb++-dev libncurses5-dev␣
↪libreadline-dev \
                 python3-dev python3-pip libyaml-dev libboost-all-dev ntp
```

Note that we also install the `ntp` client here, this is needed to keep your server's time correctly adjusted, which is a requirement for a witness wanting to sign blocks (given that the time slot for a block is 3 seconds, you need to be pretty much spot on when it's your turn to sign a block)

Mac OSX

It is possible, and hence recommended, to build the BitShares client using only libraries pulled out of homebrew, as you don't have to compile and maintain dependencies yourself.

```
brew install git cmake boost berkeley-db readline openssl libyaml
brew link --force readline
```

If you already had an "old" version of boost installed, please upgrade to a newer one:

```
$ brew upgrade boost
```

Also make sure that you are running a constantly up-to-date version of cmake, you might encounter weird configuration errors otherwise (ie: cmake not finding properly installed dependencies, etc.)

```
$ brew upgrade cmake
```

Install the bts_tools package

The first step in your quest for being a delegate is to install the `bts_tools` python package. Make sure you have installed the dependencies as described in the previous section, and run (as root[1]):

```
# pip3 install bts_tools
```

That's it, the tools are installed and you should now be setup for building the BitShares client! To see all that the tools provide, try running:

```
$ bts -h
```

which should show the online help for the tools. You should definitely get accustomed to the list of commands that are provided.

Build and run the BitShares client

To build the BitShares client, just type the following:

```
$ bts build
```

This will take some time, but you should end up with a BitShares binary ready to be executed. To make sure this worked, and see all the versions available on your system, type:

```
$ bts list
```

This should also show you the default version of the client that will be run.

To run it, you just need to:

```
$ bts run
```

[1] This installs the tools system-wide, and is the simplest way of doing it. However, if you have time to invest in learning about them, it is highly recommended to look into python virtualenvs and how to deal with them. You can find a quick overview about them here: *Appendix: dealing with virtualenvs*

After the first run, this will have created the `~/.BitShares` directory (`~/Library/Application Support/BitShares` on OSX) and you should go there, edit the `config.json` file, and fill in the user and password for the RPC connection.

At this point, you want to create a wallet, an account and register it as a witness. Please refer to the BitShares documentation for instructions.

Pro Tip: running the client in tmux

Running the client inside your shell after having logged in to your VPS is what you want to do in order to be able to run it 24/7. However, you want the client to still keep running even after logging out. The solution to this problem is to use what is called a terminal multiplexer, such as screen or tmux. Don't worry about the complicated name, what a terminal multiplexer allows you to do is to run a shell to which you can "attach" and "detach" at will, and which will keep running in the background. When you re-attach to it, you will see your screen as if you had never disconnected.

Here we will use `tmux`, but the process with `screen` is extremely similar (although a few keyboard shortcuts change).

The first thing to do is to launch `tmux` itself, simply by running the following in your shell:

```
$ tmux
```

You should now see the same shell prompt, but a status bar should have appeared at the bottom of your screen, meaning you are now running "inside" tmux.

Note: The keyboard shortcuts are somewhat arcane, but this is the bare minimum you have to remember:

when outside of tmux:

- `tmux` : create a new tmux session
- `tmux attach` : re-attach to a running session

when inside of tmux:

- `ctrl+b d` : detach the session - do this before disconnecting from your server
- `ctrl+b [` : enter "scrolling mode" - you can scroll back the screen (normal arrows and sliders from your terminal application don't work with tmux...) Use `q` to quit this mode

---

So let's try attaching/detaching our tmux session now: as you just ran 'tmux', you are now inside it type `ctrl-b d`, and you should now be back to your shell before launching it

```
$ tmux attach  # this re-attaches to our session
$ bts run      # we run the bitshares client inside tmux
```

type `ctrl-b d`, you are now outside of tmux, and doesn't see anything from the bts client

```
$ tmux attach  # this re-attaches your session, and you should see the bts client␣
↪still in action
```

To get more accustomed to tmux, it is recommended to find tutorials on the web, this one for instance seems to do a good job of showing the power of tmux while not being too scary...

Run the monitoring webapp

This is the good part :)

Now that you know how to build and run the delegate client, let's look into setting up the monitoring of the client. Say you want to monitor the delegate called `mydelegate`. The possible events that we can monitor and the actions that we can take also are the following:

- monitor when the client comes online / goes offline (crash), and send notifications when that happens (email or iOS)
- monitor when the client loses network connection
- monitor when the client misses a block
- publish feeds
- ensure that version number is the same as the one published on the blockchain, and if not, publish a new version

These can be set independently for each delegate that you monitor, and need to be specified in the `nodes` attribute of the `config.yaml` file.

A node specifies the following properties:

- the type of client that it runs (BitShares, PTS, ...) In our case here, this will be `"bts"`.
- the type of the node will be `"delegate"` (could be `"seed"` too, but we're setting up a delegate here).
- the name here will be set to `"mydelegate"` (replace with your delegate's name)
- the `"monitoring"` variable will contain: `[version, feeds, email]`. As online status, network connections and missed blocks are always monitored for a delegate node, you only need to specify whether you want to receive the notifications by email, boxcar or Telegram, in this case here we want `email`. You will also need to configure the `email` section in the `config.yaml` in order to be able to send them out.

This gives the following:

```
nodes:
    -
        client: bts
        type: delegate
        name: mydelegate
        monitoring: [version, feeds, email]
```

Once you have properly edited the `~/.bts_tools/config.yaml` file, it is just a matter of running:

```
$ bts monitor
```

and you can now go to http://localhost:5000/ in order to see it.

Install the tools behind Nginx + uWSGI

Although outside of the scope of this tutorial, if you want to set up your delegate properly and have the web interface accessible from the outside, it is recommended to put it behind an Nginx server. Please refer here for an example: *Setting up on a production server* (skip the part about the virtualenv if it doesn't apply to you)

Note that there are some choices of software that are quite opinionated in this guide, however this should not be considered as the only way to do things, but rather just a way that the author thinks makes sense and found convenient for himself.