Bravo Documentation

Release 1.8.1

Corbin Simpson & Derrick Dymock

December 12, 2012

CONTENTS

Bravo is an elegant, speedy, and extensible implementation of the Minecraft Alpha/Beta protocol. Only the server side is implemented. The following introductory topics provide a better look at the project, its goals, and current capabilities.

CHAPTER

ONE

A HIGH-LEVEL INTRODUCTION

Bravo is an open source, reverse-engineered implementation of Minecraft's server application. Two of the major building blocks are Python and Twisted, but you need not be familiar with either to run, administer, and play on a Bravo-based server.

1.1 Similar and different

While one of the goals of Bravo is to be roughly on par with the standard, "Notchian" Minecraft server, Bravo does change and improve things for the better, where appropriate. See *Differences vs. vanilla Minecraft Server* for more details.

Some of the more positive hilights include:

- More responsiveness with higher populations.
- Much less memory and bandwidth consumption.
- Better inventory system that avoids some bugs found in the standard server.

1.2 Current state

Bravo is currently in heavy development. While it is probably safe to run creative games, we lack some elements needed for Survival-Multiplayer. Take a look at *Features* to get an idea of where we currently stand.

We encourage the curious to investigate for themselves, and post any bugs, questions, or ideas you may have to our issue tracker.

1.3 Project licensing

Bravo is MIT/X11-licensed. A copy of the license is included in the `LICENSE` file in the repository or distribution. This extremely permissive license gives you all of the flexibility you could ever want.

1.4 Q & A

Why are you doing this? What's wrong with the official Alpha/Beta server?

Plenty. The biggest architectural mistake is the choice of dozens of threads instead of NIO and an asynchronous event-driven model, but there are other problems as well.

Are you implying that the official Alpha server is bad?

Yes. As previous versions of this FAQ have stated, Notch is a cool guy, but the official server is bad.

Are you going to make an open-source client? That would be awesome!

The server is free, but the client is not. Accordingly, we are not pursuing an open-source client at this time. If you want to play Alpha, you should pay for it. There's already enough Minecraft piracy going on; we don't feel like being part of the problem. That said, Bravo's packet parser and networking tools could be used in a client; the license permits it, after all.

Where did the docs go?

We contribute to the Minecraft Collective's wiki at http://mc.kev009.com/wiki/ now, since it allows us to share data faster. All general Minecraft data goes to that wiki. Bravo-specific docs are shipped in ReST form, and a processed Sphinx version is available online at http://www.docs.bravoserver.org/.

Why did you make design decision <X>?

There's an entire page dedicated to this in the documentation. Look at docs/philosophy.rst or *Philosophy*.

It doesn't install? Okay, maybe it installed, but I'm having issues!

On Freenode IRC (irc.freenode.net), #bravo is dedicated to Bravo development and assistance, and #mcdevs is a more general channel for all custom Minecraft development. You can generally get help from those channels. If you think you have found a bug, you can directly report it on the Github issue tracker as well.

Please, please, please read the installation instructions first, as well as the comments in bravo.ini.example. I did not type them out so that they could be ignored. :3

1.5 Credits

Who are you guys, anyway?

Corbin Simpson (MostAwesomeDude) is the main coder. Derrick Dymock (Ac-town) is the visionary and provider of network traffic dumps. Ben Kero and Mark Harris are the reluctant testers and bug-reporters. The Minecraft Coalition has been an invaluable forum for discussion.

CHAPTER

TWO

FEATURES

Bravo's extensible design means that there are many different plugins and features. Since most servers do not have an extensive or exhaustive list of the various plugins that they include, one is provided here for Bravo.

2.1 Standard features

These features are found in official, Mojang-sponsored, unmodified servers.

2.1.1 Console

Bravo provides a small, plain console suitable for piping input and output, as well as interactive sessions.

2.1.2 Login

Bravo supports the two login methods supported by the Mojang-sponsored client: offline authentication and online authentication.

2.1.3 Geometry

Bravo understands how to manipulate and transfer geometry. In addition, Bravo can read and write the Alpha NBT and Beta MCR disk formats.

2.1.4 Time

Bravo fully implements the in-game day and night. Bravo's days are exactly 20 minutes long.

2.1.5 Entities

Bravo understands the concept of entities, and is able to track the following kinds of entities:

- Mobs
- Paintings
- Pickups
- Players

- Tiles

Mobs

Bravo understands the following mobs:

- Chickens/ducks ("Chucks")
- Cows
- Creepers
- Ghasts
- Giant zombies
- Pigs
- Sheep
- Skeletons
- Slimes
- Spiders
- Squids
- Wolves
- Zombie pigmen
- Zombies

Tiles

Bravo understands the following tiles:

- Chests
- Furnaces
- Mob spawners
- Music blocks
- Signs

2.1.6 Inventory

Bravo provides server-side inventory handling.

2.1.7 Physics

Bravo simulates physics, including the behaviors of sand, gravel, water and lava, and redstone.

2.2 Extended features

Bravo provides many things not in other servers. While a strict comparison of other open-source servers is impossible due to the speedy rate at which they are changing, the features that separate Bravo from the Mojang-sponsored server are listed here.

2.2.1 Console

Bravo ships with a fancy console which supports readline-like editing features.

2.2.2 Time

Bravo implements an in-game year of 360 in-game days.

2.2.3 Plugins

Bravo supports several different types of plugins. For more information, see *Plugins*.

CHAPTER

THREE

DIFFERENCES VS. VANILLA MINECRAFT SERVER

Bravo was written from the ground up and doesn't inherit code from any other Minecraft project. This means that it sometimes behaves very differently, in subtle and obvious ways, from other servers.

The "Notchian" server is the server authored by Notch and distributed by Mojang as a companion to the Mojang-sponsored client.

3.1 Responsiveness

Bravo is occasionally perceived to be "lighter" or "snappier" compared to the Notchian server. Reports of feeling like players are moving faster than normal are also common. The root cause is simple: Bravo is quicker to respond to clients than the Notchian server. This is normal, expected, and not currently planned to be fixed.

3.2 Chunks

The Notchian server maintains a floating pattern above players, centered on the chunk the player is standing in. This pattern is always a square of chunks, 21 chunks to a side. This results in a total of 441 chunks being deployed to the client at any one time. All 441 chunks are deployed before the client is permitted to interact with the world.

Bravo does something slightly different; while Bravo also has a floating pattern above each of its players, the pattern is a circle with the same diameter as the Notchian server's square. This effectively results in a circle of 315 chunks deployed to the client; a savings of nearly 30% in memory and bandwidth for chunks. Additionally, only the 50 closest chunks are deployed before the client is spawned and permitted to interact with the world.

3.3 Inventory

The Notchian viewpoint of items in the inventory is as a list of slots. Each slot holds an item, identified by a single number, and can hold 1 to 64 instances of that item. Some items can be damaged. Some items are completely different depending on their damage.

Bravo views item identifiers as a composite key of a primary and secondary identifier. In this scheme, items with identical primary keys and different secondary keys are properly segregated, and item damage is stored as the secondary key, keeping items with differing amounts of damage from occupying the same slot. This avoids an entire class of bugs, where items can be stacked and unstacked to change the amount of damage on them, which have historically plagued the Notchian codebase.

3.4 Minecarts

Bravo permits minecart tracks to be placed on glass.

CHAPTER

FOUR

PHILOSOPHY

4.1 Design Decisions

A **design decision** is a core component of building a large piece of software. Roughly stated, it is a choice to use a certain language, library, or methodology when constructing software. Design decisions can be metaphysical, and affect other design decisions. This is merely a way of talking formally and reasonably about choices made in producing Bravo.

This section is largely dedicated to members of the community that have decided that things in Bravo are done incorrectly. While we agree with the need of the community to constructively criticize itself, some things are not worth debating again.

4.1.1 Python

Python is occasionally seen as slow compared to statically typed languages. Some benchmarks certainly are very unflattering to Python, but we feel that there are several advantages to Python which are too important to sacrifice:

- Rapid prototyping
- Algorithmic simplicity
- Simple types
- Twisted

Additionally, with the advent of PyPy, the question of whether a full-fledged Python application is too slow for consumer hardware is rapidly fading.

No Extension Modules

There are several good reasons to not ship "extension modules," pieces of code written in Fortran, C, or C++ which are compiled and dynamically linked against the CPython extension API. Some of them are:

Portability Python and C have different scopes of portability, and the scope of the C API for Python is limited practically to CPython. Each module we depend on externally has the potential to reduce the number of platforms we can support.

Maintainability C is not maintainable on the same scale as Python, even with (and, some would argue, especially with) the extremely structured syntax required to interface with the C API for Python. Cython is maintainable, but does not solve the other problems.

Dependencies Somebody has to provide binary versions of the modules for all the people without compilers. Practically, this does mean that Win32 users need to have binaries provided for them, as long as our thin veneer of Win32 compatibility holds up.

Forward-compatibility Frankly, extension modules are forever incompatible with the spirit of PyPy, and require, at bare minimum, a recompile and prayer before they'll cooperate. This is another hurdle to jump over in the ongoing quest to make PyPy a supported Python interpreter for the entire package.

Frankly, most extension modules aren't worth this trouble. Extension modules which are well-tested, ubiquitous, and actively maintained, are generally going to be favored more than extensions which break, are hard to obtain or compile, or are derelict.

At the moment, the only extension modules required are in the numpy package, which has benefits far outweighing the above complaints.

I am expressly vetoing noise. In addition to the above complaints, its API doesn't even provide an equivalent to the pure-Python code in Bravo's core which it would supposedly supplant.

Twisted

Apparently, in this day and age, people are still of the opinion that Twisted is too big and not necessary for speedy, relatively bug-free networking. Nothing written here will convince these people; so, instead, I offer this promise: If anybody contributes a patch which makes Bravo not depend on Twisted, does not degrade its performance measureably, and does not break any part of Bravo, then I will acknowledge and apply it.

4.1.2 No Threads

Threads are evil. They are not an effective concurrency model in most cases. Tests done with offloading various parts of Bravo's CPU-bound tasks to threads have shown that threads are a liability in most cases, enforcing locking overhead while providing little to no actual benefit in terms of speed and latency.

However, as a concession to the CPU-centric nature of geometry generation, Bravo will offload all geometry generation to separate processes when Ampoule is available and enabled in its configuration file, which does yield massive improvements to server interactivity.

4.1.3 Extreme Extensibility

Bravo is remarkably extensible. Pieces of functionality that are considered essential or "core" are treated as plugins and dynamically loaded on server startup. Actual services are dynamically started and stopped as needed. Bravo's core does not even provide Minecraft services by default.

The reason for this extreme plugin approach is that Bravo was designed to be easily totally convertible; in theory, a proper set of configuration files and external plugins can completely change Bravo's behavior.

4.2 Versioning

Bravo's version numbers are not very complex. Here's a quick breakdown.

Major version numbers indicate the core structure of Bravo. A major version bump probably means that lots of modules changed names, or that something significant was added. In practice, this probably means that an entirely new set of protocols was added. (The next major version bump will probably be for InfiniCraft support.)

Minor version numbers are for changes to interfaces or any other change which means that external code relying on Bravo's API will have to be updated.

Patchlevel version numbers aren't currently used, but probably will signify that the release is a bugfix-only release with no significant change in functionality.

The hope of all of this is that, given a series of releases with the same major and minor, plugins do not have to be changed.

CHAPTER

FIVE

ADMINISTRATOR TOPICS

The following topics are meant for those wishing to run a Bravo server. Topics such as installation, configuration, and troubleshooting are covered here. No software development background is necessary.

5.1 How to administer Bravo

While Bravo is not a massively complex piece of software on its own, the plugins and features that are available in Bravo can be overwhelming and daunting. This page is a short but comprehensive overview for new administrators looking to set up and run Bravo instances.

5.1.1 Configuration

Bravo uses a single configuration file, bravo.ini, for all of its settings. The file is in standard INI format. Note that this is not the extended INI format of Windows 32-bit configuration settings, nor the format of PHP's configuration files. Specifically, bravo.ini is parsed and written using Python's `ConfigParser` class.

An example configuration file is provided as `bravo.ini.example`, and is a good starting point for new configurations.

`bravo.ini` should live in one of three locations:

1. /etc/bravo
2. ~/.bravo
3. The working directory

All three locations will be checked, in that order, and more-recently-loaded configurations will override configurations in previous directories. For sanity purposes, it is highly encouraged to either use `/etc/bravo` if running as root, or `~/.bravo` if running as a normal user.

The configuration file is divided up into **sections**. Each section starts with a name, like `[section name]`, and only ends when another section starts, or at the end of the file.

A note on lists

Bravo uses long lists of named plugins, and has special facilities for handling them.

If an option takes a list of choices, then the choices should be comma-separated. They may be on the same line, or multiple lines; spaces do not matter much. (As an aside, spaces matter *inside* plugin names, but Bravo's plugin collection uses only underscores, not spaces, so this should not matter. If it does, bug your plugin authors to fix their code.)

Additionally, to simplify plugin naming, many plugin configuration options support **wildcards**. Currently, the "*" wildcard is supported. A "*" anywhere in an option list will be internally expanded to *all* of the available choices for that option.

The special notation "-" before a name will forcibly remove that name from a list.

Putting everything together, an example set of configurations might look like this:

```
some_option = first, second, third
some_newline_option = first, second,
    third, fourth
some_wildcard_option = *
some_picky_option = *, -fifth
another_picky_option = -fifth, -sixth, *
```

General settings

These settings apply to all of Bravo. This section is named `[bravo]`.

fancy_console Whether to enable the fancy console in standalone mode. This setting will be overridden if the fancy console cannot be set up; e.g. on Win32 systems.

ampoule Whether asynchronous chunk generators will be used. This can result in massive improvements to Bravo's latency and responsiveness, and defaults to enabled. This setting will be overridden if Ampoule cannot be found.

World settings

These settings only apply to a specific world. Worlds are created by starting the section of the configuration with "world"; an example world section might start with `[world example]`.

port Which port to run on. Must be a number between 0 and 65535. Note that ports below 1024 are typically privileged and cannot be bound by non-root users.

host The hostname to bind to. Defaults to no hostname, which is usually correct for most people. If you don't know what this is, you don't need it.

url The path to the folder to use for loading and saving world data. Must be a valid URL.

authenticator Which authentication plugin to use.

serializer Which serializer to use for saving worlds. Currently, the "alpha" and "beta" serializers are provided for MC Alpha and MC Beta compatibility, respectively.

build_hooks Which *Build hooks* to enable. This is a list of plugins; see above.

dig_hooks Which *Dig hooks* to enable. This is a list of plugins.

generators Which *Terrain generators* to use. This is a list of plugins.

seasons Which *Seasons* to enable. This, too, is a list of plugins.

automatons Which *Automatons* to enable. Another list of plugins.

5.1.2 Plugin Data Files

Plugins have a standardized per-world storage. Only a few of the plugins that ship with Bravo use this storage. Each plugin has complete autonomy over its data files, but the file name varies depending on the serializer used to store the world. For example, when using the Alpha and Beta world serializers, the file name is <plugin>.dat, where <plugin> is the name of the plugin.

Bravo worlds have per-world IP ban lists. The IP ban lists are stored under the plugin name "banned_ips", with one IP address per line.

Warps and homes are stored in hey0 CSV format, in "warps" and "homes".

5.2 Plugins

Bravo is highly configurable and extensible. The plugins shipped with Bravo are listed here, for convenience.

5.2.1 Authenticators

Offline

Offline authentication does no checking against the official minecraft.net server, so it can be used when minecraft.net is down or the network is unreachable for any reason. On the downside, it provides no actual security.

Online

Online authentication is the traditional authentication with minecraft.net.

Password

Password authentication is an experimental authentication scheme which directly authenticates clients against a server without consulting minecraft.net or any other central authority. However, it is not correctly implemented in the Notchian client, and will not work with that client.

5.2.2 Terrain generators

The following terrain generators may be added to the `generators` setting in your `bravo.ini` under the `[world]` section. The order in which these appear in the list is not important.

Beaches

Generates simple beaches.

Beaches are areas of sand around bodies of water. This generator will form beaches near all bodies of water regardless of size or composition; it will form beaches at large seashores and frozen lakes. It will even place beaches on one-block puddles.

Boring

Generates boring slabs of flat stone.

Grass

Grows grass on exposed dirt.

Caves

Carves caves and seams out of terrain.

Cliffs

Generates sheer cliffs.

Complex

Generates islands of stone and other ridiculous things.

Erosion

Erodes stone surfaces into dirt.

Float

Rips chunks out of the map, to create surreal chunks of floating land.

Safety

Generates terrain features essential for the safety of clients, such as the indestructible bedrock at Y = 0.

Warning: Removing this generator will permit players to dig through the bottom of the world.

Simplex

Generates organic-looking, continuously smooth terrain.

Saplings

Plants saplings at relatively silly places around the map.

Note: This generator only places saplings, and is not responsible for the growth of trees over time. The `trees` automaton should be used for ensuring that trees will grow.

Ore

Places ores and clay.

Watertable

Creates a flat water table half-way up the map (Y = 64).

5.2.3 Automatons

Automatons are simple tasks which examine and update the world as the world loads and displays data to players. They are able to do periodic or delayed work to keep the world properly. (The mental image of small robotic gardeners roving across the hills and valleys trimming grass and dusting trees is quite compelling and adorable!)

Automatons marked with (Beta) provide Beta compatibility and should probably be enabled.

- **lava**: Enable physics for placed lava springs. (Beta)
- **trees**: Turn planted saplings into trees. (Beta)
- **water**: Enable physics for placed water springs. (Beta)

5.2.4 Seasons

Bravo's years are 360 days long, with each day being 20 minutes long. For those who would like seasons, the following seasons be added to the `seasons` setting in your `bravo.ini` under the `[world]` section.

Winter

Causes water to freeze, and snow to be placed on certain block types. Winter starts on the first day of the year.

Spring

Thaws frozen water and removes snow as that was placed during Winter. Spring starts on the 90th day of the the year.

5.2.5 Hooks

Hooks are small pluggable pieces of code used to add event-driven functionality to Bravo.

Build hooks

Hooks marked with (Beta) provide Beta compatibility and should probably be enabled.

- **alpha_sand_gravel**: Make sand and gravel fall as if affected by gravity. (Beta)
- **bravo_snow**: Make snow fall as if affected by gravity.
- **build**: Enable placement of blocks from inventory onto the terrain. (Beta)
- **build_snow**: Adjust things built on top of snow to replace the snow. (Beta)
- **redstone**: Enable physics for placed redstone. (Beta)
- **tile**: Register tiles. Required for signs, furnaces, chests, etc. (Beta)
- **tracks**: Align minecart tracks. (Beta)

Dig hooks

- **alpha_sand_gravel**: Make sand and gravel fall as if affected by gravity. (Beta)
- **alpha_snow**: Destroy snow when it is dug or otherwise disturbed. (Beta)
- **bravo_snow**: Make snow fall as if affected by gravity.
- **give**: Spawn pickups for blocks and items destroyed by digging. (Beta)
- **lava**: Enable physics for lava. (Beta)
- **redstone**: Enable physics for redstone. (Beta)
- **torch**: Destroy torches that are not attached to walls or floors. (Beta)
- **tracks**: Align minecart tracks. (Beta)
- **water**: Enable physics for water. (Beta)

5.3 Troubleshooting

5.3.1 Configuring

When I connect to the server, the client gets an "End of Stream" error and the server log says something about "ConsoleRPCProtocol".

You are connecting to the wrong port.

Bravo always runs an RPC console by default. This console isn't directly accessible from clients. In order to connect a client, you must configure a world and connect to that world. See the example bravo.ini configuration file for an example of how to configure a world.

My world is snowy. I didn't want this.

In bravo.ini, change your `seasons` list to exclude winter. A possible incantation might be the following:

```
seasons = *, -winter
```

5.3.2 Errors

I get lots of RuntimeErrors from Exocet while loading things like

`bravo.parameters`, `xml.sax`, and `twisted.internet`. Those are harmless.

Exocet is very, very strict about imports, and in fact, it is stricter than the standard importer. This means that Exocet will warn about modules which try to do weird or tricky things during imports. The warnings might be annoying, but they aren't indicative of anything going wrong.

I have an error involving construct!

Install Construct.

I have an error involving JSON!

If you update to a newer Bravo, you won't need JSON support.

I have an error involving IRC/AMP/ListOf!

Your Twisted is too old. You really do need Twisted 10.1 or newer.

I have an error ''TypeError: an integer is required'' when starting Bravo!

Is your Twisted 10.1 or older? This error could be caused by your Twisted not being 10.2 or newer.

I am running as root on a Unix system and twistd cannot find ''bravo.service''. What's going on?

For security reasons, twistd doesn't look in non-system directories as root. If you insist on running as root, try an incantation like the following, setting `PYTHONPATH`:

```
# PYTHONPATH=. twistd -n bravo
```

5.3.3 Help!

If you are having a hard time figuring something out, encountered a bug, or have ideas, feel free to reach out to the community in one of several different ways:

- **IRC:** #Bravo on FreeNode
- Post to our issue tracker.
- Speak up over our mailing list.

5.4 Web Service

Bravo comes with a simple web service which can be used to monitor the status of your server.

5.4.1 Configuration

Only one web service can be defined; it uses the configuration key `[web]` and has only one parameter, `port`, specifying the port on which to listen. An example configuration snippet might look like this:

```
[web]
port = 8080
```

CHAPTER

SIX

DEVELOPER TOPICS

The following topics are of general use to those wishing to modify or understand the Bravo source code. These topics are completely unecessary for those who are only interested in running or administering a Bravo server.

6.1 Extending Bravo

Bravo is designed to be highly extensible. This document is a short guide to the basics of writing extension code for Bravo.

6.1.1 Asynchronous Ideas

Bravo, being built on Twisted, has inherited most of the concepts of asynchronous control flow from Twisted, and uses them liberally. Nearly every plugin method is permitted to return a Deferred in place of their actual return value.

6.1.2 Exocet and You

Bravo uses a library called Exocet to help it with plugin discovery. Exocet is a remarkably powerful library which customizes the way imports are done. Instead of importing plugins by name, or package, Exocet can be asked to **load** a plugin. When Exocet loads plugins, all import statements in the plugin are transformed to go through Exocet, so Exocet (and by extension, Bravo) can modify what your plugins import.

So, what does this mean for you, the plugin author? Well, there are a few things to keep in mind...

Blacklisting

Exocet can blacklist imports, preventing them from actually happening and keeping your plugin from loading. Bravo uses this ability to blacklist a smattering of standard library modules from plugins.

Some of these blacklisted modules are chosen for security reasons, while others are chosen because they will cause slow or buggy behavior. If you think you absolutely need one of these modules, consider carefully whether the listed reason for it being on the blacklist is relevant and reasonable.

The following modules are blacklisted becuse they can be used to crash the server:

- ctypes

The following modules are blacklisted because they can be used to examine the internals of the server or bypass Exocet's protections:

- gc

- imp
- inspect

The following modules are blacklisted because they conflict with, or are slow compared to, Twisted's own systems:

- asyncore
- multiprocessing
- socket
- subprocess
- thread
- threading

Parameters

Exocet supports parameterization of imports. Specifically, imports of modules which don't actually exist can be rewritten to provide faked, or **synthetic**, modules. For an example, consider the following snippet of code:

```
from bravo.parameters import example
```

This snippet brings the `example` name into the global namespace for the module, obviously, but what might not be obvious is that bravo.parameters doesn't actually exist! It is a synthetic module created by the plugin loader.

A word of warning: If the plugin loader decides not to offer any parameters to plugins, then your plugin will not load at that time. This is important because it means that you probably should not try to do things like `from bravo import parameters`. Import exactly the names you need to import; don't have imports which do nothing.

Of course, if you want to have a name available, but it is ultimately optional, the following is legal and works fine:

```
try:
    from bravo.parameters import example
except ImportError:
    example = None
```

The following parameters might be available:

- `factory`: The factory owning this instance of the plugin.

6.1.3 The Flexibility of Commands

Bravo's command interface is designed to feel like a regular class instead of a specialized plugin, while still providing lots of flexibility to authors. Let's look at a simple plugin:

```
class Hello(object):
    """
    Say hello to the world.
    """

    implements(IChatCommand)

    def chat_command(self, username, parameters):
        greeting = "Hello, %s!" % username
        yield greeting

    name = "hello"
```

```
    aliases = tuple()
    usage = ""
```

This command is a simple greeter which merely echoes a salutation to its caller. It is an `IChatCommand`, so it only works in the in-game chat, but that should not be a problem, since there is an internal, invisible adaptation from `IChatCommand` to `IConsoleCommand`. This means that chat commands are also valid console commands, without any action on your part! Pretty cool, huh?

So, how does this plugin actually work? Well, nearly every line of this plugin is required. The first thing you'll notice is that this plugin has a class docstring. Docstrings on commands are required; the docstring is used to provide help text. As with all chat commands, this plugin `implements(IChatCommand)`, which lets it be discovered as a command.

The plugin implements the required `chat_command(username, parameters)`, which will be called when a player uses the command. An interesting thing to note is that this plugin yields its return value; commands may return any iterable of lines, including a generator!

Finally, the plugin finishes with more required interface attributes: a name which will be used to call the command, a (possibly empty) list of aliases which can also be used to call the command, and a (possibly empty) usage string.

6.2 Noise

Bravo, like all Minecraft terrain generators, relies heavily on randomness to generate its terrain. In order to understand some of the design decisions in the terrain generator, it is required to understand noise and its various properties.

6.2.1 Probability

Noise's probability distribution is not even, equal, or normal. It *is* symmetric about 0, meaning that the absolute value of noise has all of the same relative probabilities as the entire range of noise.

When binned into a histogram with 100 bins, a few bins become very large.

Bin	Probability
0.00	2.6150%
0.49	2.2262%
0.59	1.8274%
0.43	1.8248%
0.42	1.7888%
0.58	1.5939%
0.48	1.5194%
0.41	1.5118%
0.18	1.4715%
0.24	1.4366%
0.54	1.4072%
0.22	1.3825%
0.50	1.3786%
0.44	1.3696%
0.26	1.3680%

6.3 Core

These modules comprise the core functionality of Bravo.

6.3.1 `blocks` – Block descriptions

The `blocks` module contains descriptions of blocks.

class `bravo.blocks.`**`Block`**(*slot*, *name*, *secondary=0*, *drop=None*, *replace=0*, *ratio=1*, *quantity=1*, *dim=16*, *breakable=True*, *orientation=None*)

Bases: `object`

A model for a block.

There are lots of rules and properties specific to different types of blocks. This class encapsulates those properties in a singleton-style interface, allowing many blocks to be referenced in one location.

The basic idea of this class is to provide some centralized data and information about blocks, in order to abstract away as many special cases as possible. In general, if several blocks all have some special behavior, then it may be worthwhile to store data describing that behavior on this class rather than special-casing it in multiple places.

Parameters

- **slot** (*int*) – The index of this block. Must be globally unique.
- **name** (*str*) – A common name for this block.
- **secondary** (*int*) – The metadata/damage/secondary attribute for this block. Defaults to zero.
- **drop** (*tuple*) – The type of block that should be dropped when an instance of this block is destroyed. Defaults to the block value, to drop instances of this same type of block. To indicate that this block does not drop anything, set to air (0, 0).
- **replace** (*int*) – The type of block to place in the map when instances of this block are destroyed. Defaults to air.
- **ratio** (*float*) – The probability of this block dropping a block on destruction.
- **quantity** (*int*) – The number of blocks dropped when this block is destroyed.
- **dim** (*int*) – How much light dims when passing through this kind of block. Defaults to 16 = opaque block.
- **breakable** (*bool*) – Whether this block is diggable, breakable, bombable, explodeable, etc. Only a few blocks actually genuinely cannot be broken, so the default is True.
- **orientation** (*tuple*) – The orientation data for a block. See `orientable()` for an explanation. The data should be in standard face order.

`face`(*metadata*)

Retrieve the face for given metadata corresponding to an orientation, or None if the metadata is invalid for this block.

This method only returns valid data for orientable blocks; check `orientable()` first.

`orientable`()

Whether this block can be oriented.

Orientable blocks are positioned according to the face on which they are built. They may not be buildable on all faces. Blocks are only orientable if their metadata can be used to directly and uniquely determine the face against which they were built.

Ladders are orientable, signposts are not.

Return type bool

Returns True if this block can be oriented, False if not.

orientation(*face*)
Retrieve the metadata for a certain orientation, or None if this block cannot be built against the given face.

This method only returns valid data for orientable blocks; check `orientable()` first.

class bravo.blocks.**Item**(*slot*, *name*, *secondary=0*)
Bases: object

An item.

bravo.blocks.**armor_boots** = (301, 305, 309, 313, 317)
List of slots of boots.

bravo.blocks.**armor_chestplates** = (299, 303, 307, 311, 315)
List of slots of chestplates.

Note that slot 303 (chainmail chestplate) is a chestplate, even though it is not normally obtainable.

bravo.blocks.**armor_helmets** = (86, 298, 302, 306, 310, 314)
List of slots of helmets.

Note that slot 86 (pumpkin) is a helmet.

bravo.blocks.**armor_leggings** = (300, 304, 308, 312, 316)
List of slots of leggings.

bravo.blocks.**blocks** = {0: Block((0, 0) 'air': unbreakable, transparent), 1: Block((1, 0) 'stone': drops 1 (key (4, 0), rate 1(
A dictionary of Block objects.

This dictionary can be indexed by slot number or block name.

bravo.blocks.**items** = {'wooden-door': Item((324, 0) 'wooden-door'), 'compass': Item((345, 0) 'compass'), 'blaze-rod': Ite
A dictionary of Item objects.

This dictionary can be indexed by slot number or block name.

bravo.blocks.**parse_block**(*block*)
Get the key for a given block/item.

bravo.blocks.**unstackable** = (268, 269, 270)
List of fuel blocks and items maped to burn time

6.3.2 chunk – Chunk data structures

The chunk module holds the data structures required to track and update block data in chunks.

6.3.3 entity – Entities

The entity module contains entity classes.

6.3.4 factories – Twisted factories

The factories package hosts factories for various protocols.

MineCraft Beta factories

InfiniCraft factories

6.3.5 `ibravo` – Interfaces

The `ibravo` module holds the interfaces required to implement plugins and hooks.

Interface Bases

These are the base interface classes for Bravo. Plugin developers probably will not inherit from these; they are used purely to express common plugin functionality.

Plugins

Hooks

6.3.6 `inventory` – Inventories

The `inventory` module contains all kinds of windows and window parts like inventory, crafting and storage slots.

Generally to create a window you must create a `Window` object (of specific class derived from `Window`) and pass arguments like: window ID, player's inventory, slot's or tile's inventory, coordinates etc.

Generic construction (never use in your code :)

```
window = Window( id, Inventory(), Workbench(), ...)
```

Please note that player's inventory window is a special case. It is created when user logins and stays always opened. You probably will never have to create it.

```
def authenticated(self):
    BetaServerProtocol.authenticated(self)

    # Init player, and copy data into it.
    self.player = yield self.factory.world.load_player(self.username)
    ...
    # Init players' inventory window.
    self.inventory = InventoryWindow(self.player.inventory)
    ...
```

Every windows have own class. For instanse, to create a workbench window:

```
i = WorkbenchWindow(self.wid, self.player.inventory)
```

Furnace:

```
bigx, smallx, bigz, smallz, y = coords
furnace = self.chunks[x, y].tiles[(smallx, y, smallz)]
window = FurnaceWindow(self.wid, self.player.inventory, furnace.inventory, coords)
```

class `bravo.inventory.`**`Inventory`**

Bases: `bravo.inventory.SerializableSlots`

The class represents Player's inventory

add(*item*, *quantity*)
Attempt to add an item to the inventory.

Parameters **item** (*tuple*) – a key representing the item

Returns quantity of items that did not fit inventory

consume(*item*, *index*)
Attempt to remove a used holdable from the inventory.

A return value of `False` indicates that there were no holdables of the given type and slot to consume.

Parameters

- **item** (*tuple*) – a key representing the type of the item
- **slot** (*int*) – which slot was selected

Returns whether the item was successfully removed

select_armor(*index*, *alternate*, *shift*, *selected=None*)
Handle a slot selection on an armor slot.

Returns tuple (True/False, new selection)

class `bravo.inventory.`**SerializableSlots**
Bases: `object`

Base class for all slots configurations

6.3.7 `furnace` – Furnace Tile

The `Furnace` tile has method `changed(factory, coords)` where `coords` is tuple `(bigx, smallx, bigz, smallz, y)` - coordinates of the furnace which inventory was updated.

```
from bravo.parameters import factory
...
# inform content of furnace was probably changed
d = factory.world.request_chunk(bigx, bigz)
@d.addCallback
def on_change(chunk):
    furnace = self.get_furnace_tile(chunk, (x, y, z))
    if furnace is not None:
        furnace.changed(factory, coords)
...
```

`Furnace.changed()` method checks if current furnace shall start to burn: it must have source item, fuel and must have valid recipe. If it meets the requirements `Furnace` schedules `burn()` method with LoopingCall for every .5 second.

At every `burn()` call it:

1. increases cooktime timer and checks if item shall be crafted on this iteration;
2. decreases fuel counter and burns next fuel item if needed;
3. if there is no need to burn next fuel item because crafted slot is full or source slot is empty it stops the LoopingCall;
4. sends progress bars updates to all players that have this furnace's window opened.

6.3.8 `location` – Locations

The `location` module contains objects for tracking and analyzing locations.

class `bravo.location.`**`Location`**
Bases: `object`

The position and orientation of an entity.

`distance`(*other*)
Return the distance between this location and another location.

Distance is measured in blocks.

`in_front_of`(*distance*)
Return a `Location` a certain number of blocks in front of this position.

The orientation of the returned location is undefined.

Parameters **distance** (*int*) – the number of blocks by which to offset this position

`save_to_packet`()
Returns a position/look/grounded packet.

6.3.9 `packets` – Packet serializers

The `packets` module contains descriptions for all packets in the wire protocol, as well as several utility functions for encoding data to and decoding data from packets.

6.3.10 `plugin` – Plugin loader

6.3.11 `protocols` – Twisted protocols

The `protocols` package hosts protocol classes for various wire protocols.

MineCraft Beta protocols

InfiniCraft protocols

6.3.12 `stdio` – Console support

The `stdio` module provides a non-blocking, interactive console for administration, diagnostics, and debugging of running servers.

6.3.13 `world` – Worlds

6.4 Auxiliary

Modules which do not contribute directly to the functionality of Bravo.

6.4.1 `simplex` – Simplex noise generation

`bravo.simplex.`**`dot2`**(*u*, *v*)
Dot product of two 2-dimensional vectors.

`bravo.simplex.`**`dot3`**(*u*, *v*)
Dot product of two 3-dimensional vectors.

`bravo.simplex.`**`octaves2`**(*x*, *y*, *count*)
Generate fractal octaves of noise.

Summing increasingly scaled amounts of noise with itself creates fractal clouds of noise.

Parameters

- **x** (*int*) – X coordinate
- **y** (*int*) – Y coordinate
- **count** (*int*) – number of octaves

Returns Scaled fractal noise

`bravo.simplex.`**`octaves3`**(*x*, *y*, *z*, *count*)
Generate fractal octaves of noise.

Parameters

- **x** (*int*) – X coordinate
- **y** (*int*) – Y coordinate
- **z** (*int*) – Z coordinate
- **count** (*int*) – number of octaves

Returns Scaled fractal noise

`bravo.simplex.`**`offset2`**(*x*, *y*, *xoffset*, *yoffset*, *octaves=1*)
Generate an offset noise difference field.

Parameters

- **x** (*int*) – X coordinate
- **y** (*int*) – Y coordinate
- **xoffset** (*int*) – X offset
- **yoffset** (*int*) – Y offset

Returns Difference of noises

`bravo.simplex.`**`reseed`**(*seed*)
Reseed the simplex gradient field.

`bravo.simplex.`**`set_seed`**(*seed*)
Set the current seed.

`bravo.simplex.`**`simplex2`**(*x*, *y*)
Generate simplex noise at the given coordinates.

This particular implementation has very high chaotic features at normal resolution; zooming in by a factor of 16x to 256x is going to yield more pleasing results for most applications.

The gradient field must be seeded prior to calling this function; call `reseed()` first.

Parameters

- **x** (*int*) – X coordinate
- **y** (*int*) – Y coordinate

Returns simplex noise

Raises Exception the gradient field is not seeded

`bravo.simplex.`**`simplex3`**(*x*, *y*, *z*)
Generate simplex noise at the given coordinates.

This is a 3-dimensional flavor of `simplex2()`; all of the same caveats apply.

The gradient field must be seeded prior to calling this function; call `reseed()` first.

Parameters

- **x** (*int*) – X coordinate
- **y** (*int*) – Y coordinate
- **z** (*int*) – Z coordinate

Returns simplex noise

Raises Exception the gradient field is not seeded or you broke the function somehow

6.4.2 `utilities` – Helper functions

The `utilities` package is the standard home for shared functions which many modules may use. The spirit of `utilities` is also to isolate sections of critical code so that unit tests can be used to ensure a minimum of bugginess.

Automaton Helpers

Chat Formatting

Colorizers.

`bravo.utilities.chat.`**`sanitize_chat`**(*s*)
Verify that the given chat string is safe to send to Notchian recepients.

`bravo.utilities.chat.`**`username_alternatives`**(*n*)
Permute a username through several common alternative-finding algorithms.

Coordinate Handling

Utilities for coordinate handling and munging.

`bravo.utilities.coords.`**`adjust_coords_for_face`**(*coords*, *face*)
Adjust a set of coords according to a face.

The face is a standard string descriptor, such as "+x".

The "noop" face is supported.

`bravo.utilities.coords.`**`polar_round_vector`**(*vector*)
Rounds a vector towards zero

`bravo.utilities.coords.`**`split_coords`**(*x*, *z*)
Split a pair of coordinates into chunk and subchunk coordinates.

Parameters

- **x** (*int*) – the X coordinate
- **z** (*int*) – the Z coordinate

Returns a tuple of the X chunk, X subchunk, Z chunk, and Z subchunk

`bravo.utilities.coords.`**`taxicab2`**(*x1*, *y1*, *x2*, *y2*)
Return the taxicab distance between two blocks.

`bravo.utilities.coords.`**`taxicab3`**(*x1*, *y1*, *z1*, *x2*, *y2*, *z2*)
Return the taxicab distance between two blocks, in three dimensions.

Data Packing

More affectionately known as "bit-twiddling."

Decorators

General decorators for a variety of purposes.

`bravo.utilities.decos.`**`timed`**(*f*)
Print out timing statistics on a given callable.

Intended largely for debugging; keep this in the tree for profiling even if it's not currently wired up.

Geometry

Simple pixel graphics helpers.

`bravo.utilities.geometry.`**`gen_close_point`**(*point1*, *point2*)
Retrieve the first integer set of coordinates on the line from the first point to the second point.

The set of coordinates corresponding to the first point will not be retrieved.

`bravo.utilities.geometry.`**`gen_line_covered`**(*point1*, *point2*)
This is Bresenham's algorithm with a little twist: *all* the blocks that intersect with the line are yielded.

`bravo.utilities.geometry.`**`gen_line_simple`**(*point1*, *point2*)
An adaptation of Bresenham's line algorithm in three dimensions.

This function returns an iterable of integer coordinates along the line from the first point to the second point. No points are omitted.

Scheduling

Spatial Hashes

class `bravo.utilities.spatial.`**`Block2DSpatialDict`**
Bases: `bravo.utilities.spatial.SpatialDict`

Class for tracking blocks in the XZ-plane.

`key_for_bucket`(*key*)
Partition keys into chunk-sized buckets.

`keys_near`(*key*, *radius*)
Get all bucket keys "near" this key.

This method may return a generator.

class bravo.utilities.spatial.**Block3DSpatialDict**
Bases: bravo.utilities.spatial.SpatialDict

Class for tracking blocks in the XZ-plane.

key_for_bucket(*key*)
Partition keys into chunk-sized buckets.

keys_near(*key*, *radius*)
Get all bucket keys "near" this key.

This method may return a generator.

class bravo.utilities.spatial.**SpatialDict**
Bases: object, UserDict.DictMixin

A spatial dictionary, for accelerating spatial lookups.

This particular class is a template for specific spatial dictionaries; in order to make it work, subclass it and add `key_for_bucket()`.

iteritemsnear(*key*, *radius*)
A version of `iteritems()` that filters based on the distance from a given key.

The key does not need to actually be in the dictionary.

iterkeys()
Yield all the keys.

iterkeysnear(*key*, *radius*)
Yield all of the keys within a certain radius of this key.

itervaluesnear(*key*, *radius*)
Yield all of the values within a certain radius of this key.

keys()
Get a list of all keys in the dictionary.

Trigonometry

bravo.utilities.maths.**clamp**(*x*, *low*, *high*)
Clamp or saturate a number to be no lower than a minimum and no higher than a maximum.

Implemented as its own function simply because it's so easy to mess up when open-coded.

bravo.utilities.maths.**morton2**(*x*, *y*)
Create a Morton number by interleaving the bits of two numbers.

This can be used to map 2D coordinates into the integers.

Inputs will be masked off to 16 bits, unsigned.

bravo.utilities.maths.**rotated_cosine**(*x*, *y*, *theta*, *lambd*)
Evaluate a rotated 3D sinusoidal wave at a given point, angle, and wavelength.

The function used is:

$$f(x, y) = -\cos((x\cos\theta - y\sin\theta)/\lambda)/2 + 1$$

This function has a handful of useful properties; it has a local minimum at f(0, 0) and oscillates infinitely betwen 0 and 1.

Parameters

- **x** (*float*) – X coordinate
- **y** (*float*) – Y coordinate
- **theta** (*float*) – angle of rotation
- **lambda** (*float*) – wavelength

Returns float of f(x, y)

6.5 Tools

A handful of utilities are distributed with Bravo, in the tools directory.

6.5.1 Chunkbench

Chunkbench is a script that tests terrain generation speed.

6.5.2 Jsondump

Jsondump pretty-prints a JSON file.

6.5.3 NBTdump

NBTdump pretty-prints an NBT file.

6.5.4 Noiseview

Noiseview creates a picture of simplex noise, using Bravo's builtin noise generator.

6.5.5 parser-cli

parser-cli parses and pretty-prints raw Alpha packets.

CHAPTER

SEVEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

b

bravo.blocks, **??**
bravo.inventory, **??**
bravo.location, **??**
bravo.simplex, **??**
bravo.utilities.chat, **??**
bravo.utilities.coords, **??**
bravo.utilities.decos, **??**
bravo.utilities.geometry, **??**
bravo.utilities.maths, **??**
bravo.utilities.spatial, **??**