quarry Documentation

Release 1.5.1

Barney Gale

Feb 24, 2020

Contents

Quarry is a Python library that implements the Minecraft protocol. It allows you to write special purpose clients, servers and proxies.

CHAPTER 1

Installation

Use `pip` to install quarry:

```
$ pip install quarry
```

CHAPTER 2

Features

- Supports Minecraft versions 1.7 through 1.15.2
- Supports Python 2.7 and 3.5+
- Built upon `twisted` and `cryptography`
- Exposes base classes and hooks for implementing your own client, server or proxy
- Implements many Minecraft data types, such as NBT, Anvil, chunk sections, command graphs and entity metadata
- Implements the design of the protocol - packet headers, modes, compression, encryption, login/session, etc.
- Implements all packets in “init”, “status” and “login” modes
- Does *not* implement most packets in “play” mode - it is left up to you to hook and implement the packets you’re interested in

CHAPTER 3

Documentation

3.1 Networking

3.1.1 Writing a Client

A client is generally made up of three parts:

- A *Profile* or *OfflineProfile* object, representing the Minecraft account to use.
- A subclass of *ClientFactory*. Client factories don't do a lot; simply pass a profile to its initializer and then call *connect()*. You may also want to subclass from twisted's ReconnectingClientFactory
- A subclass of *ClientProtocol*. This represents your connection to the server.

See also:

Factories and Protocols

Skeleton Client

By default quarry proceeds through the authentication process and then switches into the "play" protocol mode. The skeleton client below will receive world data from the server, but as it does not send any position updates it will be disconnected by the server after a few seconds. Please see the *Examples* for less silly clients.

```
from twisted.internet import defer, reactor
from quarry.net.client import ClientFactory, ClientProtocol
from quarry.auth import Profile


class ExampleClientProtocol(ClientProtocol):
    pass


class ExampleClientFactory(ClientFactory):
```

(continues on next page)

(continued from previous page)

```
    protocol = ExampleClientProtocol


@defer.inlineCallbacks
def main():
    print("logging in...")
    profile = yield Profile.from_credentials(
        "someone@somewhere.com", "p4ssw0rd")
    factory = ExampleClientFactory(profile)
    print("connecting...")
    factory = yield factory.connect("localhost", 25565)
    print("connected!")


if __name__ == "__main__":
    main()
    reactor.run()
```

Offline Profiles

Use an *OfflineProfile* if you only need to log into offline-mode servers:

```
from quarry.net.auth import OfflineProfile
profile = OfflineProfile("Notch")
```

class quarry.net.auth.**OfflineProfile**

__init__(*display_name='quarry'*)

classmethod from_display_name(*display_name*)
For compatibility with the `from_` methods on *Profile*, this method returns a `Deferred` that immediately fires with a constructed *OfflineProfile* object.

Online Profiles

Quarry also provides a number of methods for logging in to the Mojang session servers. Each of these returns a Deferred that will fire with a *Profile* object when login succeeds. Defining a callback and then calling `Profile.from_credentials(...).addCallback(myfunc)` is one approach, but it's usually cleaner to use inlineCallbacks, as in the first example.

class quarry.net.auth.**Profile**(*client_token*, *access_token*, *display_name*, *uuid*)

to_file(*profiles_path=None*)

classmethod from_credentials(*email*, *password*)

classmethod from_token(*client_token*, *access_token*, *display_name*, *uuid*)

classmethod from_file(*display_name=None*, *uuid=None*, *profiles_path=None*)

3.1.2 Writing a Server

A server is generally made up of two parts:

- A subclass of *ServerFactory*. Under normal circumstances only one ServerFactory is instantiated. This object represents your game server as a whole.
- A subclass of *ServerProtocol*. Each object represents a connection with a client.

See also:

Factories and Protocols

Skeleton Server

By default quarry takes clients through the authentication process and then switches into the "play" protocol mode. Normally at this point you would implement *player_joined()* to either disconnect the client or start the process of spawning the player. In the skeleton server below we don't do either, which leaves the client on the "Logging in..." screen. Please see the *Examples* for less pointless servers.

```
from twisted.internet import reactor
from quarry.net.server import ServerFactory, ServerProtocol

class ExampleServerProtocol(ServerProtocol):
    pass

class ExampleServerFactory(ClientFactory):
    protocol = ExampleServerProtocol


factory = ExampleServerFactory()
factory.listen('127.0.0.1', 25565)
reactor.run()
```

3.1.3 Writing a Proxy

A quarry proxy has five main parts:

Class	Superclass	Description
DownstreamFactory	*ServerFactory*	Spawns Downstream objects for connecting clients
Downstream	*ServerProtocol*	Connection with an external client
Bridge	PacketDispatcher	Forwards packets between the up/downstream
UpstreamFactory	*ClientFactory*	Spawns an Upstream
Upstream	*ClientProtocol*	Connection with an external server

In ASCII art:

```
+--------+       +---------------------------------+       +--------+
| mojang | ----> |              QUARRY             | ----> | mojang |
| client | <---- | downstream | bridge | upstream | <---- | server |
+--------+       +---------------------------------+       +--------+
```

Typically the *Bridge* and *DownstreamFactory* are customized.

When the user connects, the *DownstreamFactory* creates a Downstream object to communicate with the external client. If we're running in online-mode, we go through server-side auth with mojang.

Once the user is authed, we spawn a UpstreamFactory, which makes a connection to the external server and spawns an Upstream to handle it. If requested we go through client-side auth.

At this point both endpoints of the proxy are authenticated and switched to "play" mode. The *Bridge* assumes responsibility for passing packets between the endpoints. Proxy business logic is typically implemented by defining packet handlers in a *Bridge* subclass, much like in client and server *protocols*. Unlike clients and servers, the method name must include the packet direction before its name, e.g.:

```
# Hook server-to-client keep alive
def packet_upstream_tab_complete(self, buff):
    # Unpack the packet
    p_text = buff.unpack_string()

    # Do a custom thing
    if p_text.startswith("/msg"):
        return # Drop the packet

    # Forward the packet
    buff.restore()
    self.upstream.send_packet("tab_complete", buff.read())
```

If a packet is hooked but not explicitly forwarded it is effectively dropped. Unhooked packets are handled by *Bridge.packet_unhandled()*, which forwards packets by default.

Skeleton Proxy

The proxy below will do online-mode authentication with a client connecting on port 25565, then connect in offline mode to a server running on port 25566 and begin exchanging packets via the bridge.

```
from twisted.internet import reactor
from quarry.net.proxy import DownstreamFactory, Bridge


class ExampleBridge(Bridge):
    pass


def main(argv):
    factory = DownstreamFactory()
    factory.bridge_class = ExampleBridge
    factory.connect_host = "127.0.0.1"
    factory.connect_port = 25566
    factory.listen("127.0.0.1", 25565)
    reactor.run()


if __name__ == "__main__":
    import sys
    main(sys.argv[1:])
```

Downstream Factories

class quarry.net.proxy.**DownstreamFactory**

Subclass of *quarry.net.server.ServerFactory*. Additional attributes:

bridge_class = <class 'quarry.net.proxy.Bridge'>

connect_host = None

connect_port = None

Bridges

class quarry.net.proxy.**Bridge**(*downstream_factory*, *downstream*)
This class exchanges packets between the upstream and downstream.

upstream_factory_class = <class 'quarry.net.proxy.UpstreamFactory'>

log_level = 20

logger = None

downstream_factory = None

downstream = None

upstream_profile = None

upstream_factory = None

upstream = None

make_profile()
Returns the profile to use for the upstream connection. By default, use an offline profile with the same display name as the remote client.

connect()
Connect to the remote server.

downstream_ready()
Called when the downstream is waiting for forwarding to begin. By default, this method begins a connection to the remote server.

upstream_ready()
Called when the upstream is waiting for forwarding to begin. By default, enables forwarding.

downstream_disconnected()
Called when the connection to the remote client is closed.

upstream_disconnected()
Called when the connection to the remote server is closed.

enable_forwarding()
Enables forwarding. Packet handlers in the `Upstream` and `Downstream` cease to be called, and all packets are routed via the `Bridge`. This method is called by `upstream_ready()` by default.

enable_fast_forwarding()
Enables fast forwarding. Quarry passes network data between endpoints without decoding packets, and therefore all packet handlers cease to be called. Both parts of the proxy must be operating at the same compression threshold. This method is not called by default.

packet_received(*buff*, *direction*, *name*)
Called when a packet is received a remote. Usually this method dispatches the packet to a method named `packet_<direction>_<packet name>`, or calls *packet_unhandled()* if no such methods exists. You might want to override this to implement your own dispatch logic or logging.

packet_unhandled(*buff*, *direction*, *name*)
Called when a packet is received that is not hooked. The default implementation forwards the packet.

3.1.4 Factories and Protocols

Factories

Factories represent your minecraft server or client as a whole. Normally only one factory is created.

Client factories require a *Profile* object to be supplied to the initializer. Use the *ClientFactory.connect()* method to connect. If `force_protocol_version` is not defined, this method will make two connections to the server; the first is used to establish the server's protocol version.

class quarry.net.client.**ClientFactory**(*profile=None*)

protocol
alias of *ClientProtocol*

__init__(*profile=None*)

connect(*host*, *port=25565*)

force_protocol_version = None

get_buff_type(*protocol_version*)
Gets a buffer type for the given protocol version.

Server factories are used to customize server-wide behaviour. Use *listen()* to listen for connections. A set of all associated *ServerProtocol* objects is available as *players()*.

class quarry.net.server.**ServerFactory**

protocol
alias of *ServerProtocol*

motd = 'A Minecraft Server'

max_players = 20

icon_path = None

online_mode = True

prevent_proxy_connections = True

compression_threshold = 256

auth_timeout = 30

__init__()

players = None

listen(*host*, *port=25565*)

force_protocol_version = None

get_buff_type(*protocol_version*)
Gets a buffer type for the given protocol version.

Protocols

Protocols represent a connection to a remote minecraft server or client. For most common usages, clients have only one protocol active at any given time. In protocols you can define packet handlers or override methods in order to respond to events.

class quarry.net.protocol.**Protocol**
Minecraft protocol implementation common to both clients and servers. You should not subclass from this class, but rather subclass from one of the three classes below.

The methods/attributes given below relate specifically to quarry; the rest are given in the *Connection*, *Authentication* and *Packets* sections further on.

factory = None
A reference to the factory

logger = None
The logger for this protocol.

ticker = None
A reference to a *Ticker* instance.

class quarry.net.server.**ServerProtocol**(*factory*, *remote_addr*)
This class represents a connection with a client

class quarry.net.client.**ClientProtocol**(*factory*, *remote_addr*)
This class represents a connection to a server

class quarry.net.client.**SpawningClientProtocol**(*factory*, *remote_addr*)

Connection

Override the *connection_made()*, *connection_lost()* and *connection_timed_out()* methods to handle connection events. The remote's IP address is available as the *remote_addr* attribute.

In servers, *connect_host* stores the hostname the client reported that it connected to; this can be used to implement virtual hosting.

Protocol.**connection_made**()
Called when the connection is established

Protocol.**connection_lost**(*reason=None*)
Called when the connection is lost

Protocol.**connection_timed_out**()
Called when the connection has been idle too long

Protocol.**remote_addr = None**
The IP address of the remote.

ServerProtocol.**connect_host = None**

ServerProtocol.**connect_port = None**

Protocol.**close**(*reason=None*)
Closes the connection

Protocol.**closed = False**

Authentication

Override the *auth_ok()* and *auth_failed()* methods to handle an authentication outcome. In servers, the player's display name can be obtained as *display_name*, with *display_name_confirmed* being set to `True` when authentication is successful. In clients, the in-use profile is available as `self.factory.profile`.

Override the *player_joined()* and *player_left()* methods to respond to a player entering "play" mode (via the authentication process) or quitting the game from "play" mode. You can check the player's current status via *in_game*

Protocol.**auth_ok**(*data*)
Called when auth with mojang succeeded (online mode only)

Protocol.**auth_failed**(*err*)
Called when auth with mojang failed (online mode only)

ServerProtocol.**display_name = None**

ServerProtocol.**display_name_confirmed = False**

Protocol.**player_joined**()
Called when the player joins the game

Protocol.**player_left**()
Called when the player leaves the game

Protocol.**in_game = False**

Packets

Call *send_packet()* to send a packet:

```
# Add a diamond sword to the first hotbar slot
window_id = 0
slot_id = 36
item_id = 276

self.send_packet("set_slot",
    self.buff_type.pack('bh', window_id, slot_id) +
    self.buff_type.pack_slot(item_id))
```

To construct the payload, call static methods on *Buffer*. A reference to this class is available as `self.buff_type`.

To receive a packet, implement a method in your subclass of *ClientProtocol* or *ServerProtocol* with a name like `packet_<packet name>`:

```
def packet_update_health(self, buff):
    health = buff.unpack('f')
    food = buff.unpack_varint()
    saturation = buff.unpack('f')
```

See also:

Packet Names.

You are passed a *Buffer* instance, which contains the payload of the packet. If you hook a packet, you should ensure you read the entire payload.

Packet dispatching can be customized. If you override *packet_unhandled()* you can handle any packets without a matching `packet_<packet name>` handler. If you override *packet_received()*, you can replace the entire `packet_<packet name>` dispatching.

Protocol.**send_packet**(*name*, **data*)
Sends a packet to the remote.

Protocol.**buff_type = None**
Usually a reference to a *Buffer* class. This is useful when constructing a packet payload for use in *send_packet()*

Protocol.**packet_received**(*buff*, *name*)
Called when a packet is received from the remote. Usually this method dispatches the packet to a method named packet_<packet name>, or calls *packet_unhandled()* if no such methods exists. You might want to override this to implement your own dispatch logic or logging.

Protocol.**packet_unhandled**(*buff*, *name*)
Called when a packet is received that is not hooked. The default implementation silently discards the packet.

Protocol.**log_packet**(*prefix*, *name*)
Logs a packet at debug level

Protocol.**get_packet_name**(*ident*)

Protocol.**get_packet_ident**(*name*)

Ticking

To register delayed or repeating callbacks, call methods on the *Ticker* object available as self.ticker.

class quarry.net.ticker.**Ticker**(*logger*)

tick = 0
The current tick

interval = 0.05
Interval between ticks, in seconds

max_lag = 40
Maximum number of delayed ticks before they're all skipped

start()
Start running the tick loop.

stop()
Stop running the tick loop.

add_loop(*interval*, *callback*)
Repeatedly run a callback.

Parameters

- **interval** – The interval in ticks
- **callback** – The callback to run

Returns An instance providing a stop() method

add_delay(*delay*, *callback*)
Run a callback after a delay.

Parameters

- **delay** – The delay in ticks
- **callback** – The callback to run

Returns An instance providing stop() and restart() methods

remove(*task*)
Removes a task, effectively cancelling it.

Parameters **task** – The task to remove

remove_all()
Removes all registered tasks, effectively cancelling them.

3.1.5 Packet Names

See the Minecraft Coalition Wiki for a details on every packet.

Minecraft 1.15.2

- acknowledge_player_digging (downstream)
- advancement_tab (upstream)
- advancements (downstream)
- animation (downstream, upstream)
- attach_entity (downstream)
- block_action (downstream)
- block_break_animation (downstream)
- block_change (downstream)
- block_metadata_request (upstream)
- block_metadata_response (downstream)
- boss_bar (downstream)
- camera (downstream)
- change_game_state (downstream)
- chat_message (downstream, upstream)
- chunk_data (downstream)
- click_window (upstream)
- client_settings (upstream)
- client_status (upstream)
- close_window (downstream, upstream)
- collect_item (downstream)
- combat_event (downstream)
- confirm_transaction (downstream, upstream)
- craft_recipe_request (upstream)
- craft_recipe_response (downstream)
- crafting_book_data (upstream)
- creative_inventory_action (upstream)

- declare_commands (downstream)
- declare_recipes (downstream)
- destroy_entities (downstream)
- disconnect (downstream)
- display_scoreboard (downstream)
- edit_book (upstream)
- effect (downstream)
- enchant_item (upstream)
- entity (downstream)
- entity_action (upstream)
- entity_effect (downstream)
- entity_equipment (downstream)
- entity_head_look (downstream)
- entity_look (downstream)
- entity_look_and_relative_move (downstream)
- entity_metadata (downstream)
- entity_metadata_request (upstream)
- entity_properties (downstream)
- entity_relative_move (downstream)
- entity_sound_effect (downstream)
- entity_status (downstream)
- entity_teleport (downstream)
- entity_velocity (downstream)
- explosion (downstream)
- face_player (downstream)
- handshake (upstream)
- held_item_change (downstream, upstream)
- join_game (downstream)
- keep_alive (downstream, upstream)
- lock_difficulty (upstream)
- login_disconnect (downstream)
- login_encryption_request (downstream)
- login_encryption_response (upstream)
- login_plugin_request (downstream)
- login_plugin_response (upstream)
- login_set_compression (downstream)

- login_start (upstream)
- login_success (downstream)
- map (downstream)
- multi_block_change (downstream)
- name_item (upstream)
- named_sound_effect (downstream)
- open_book (downstream)
- open_horse_window (downstream)
- open_sign_editor (downstream)
- open_window (downstream)
- particle (downstream)
- pick_item (upstream)
- player (upstream)
- player_abilities (downstream, upstream)
- player_block_placement (upstream)
- player_digging (upstream)
- player_list_header_footer (downstream)
- player_list_item (downstream)
- player_look (upstream)
- player_position (upstream)
- player_position_and_look (downstream, upstream)
- plugin_message (downstream, upstream)
- remove_entity_effect (downstream)
- resource_pack_send (downstream)
- resource_pack_status (upstream)
- respawn (downstream)
- scoreboard_objective (downstream)
- select_advancement_tab (downstream)
- select_trade (upstream)
- server_difficulty (downstream)
- set_beacon_effect (upstream)
- set_cooldown (downstream)
- set_difficulty (upstream)
- set_experience (downstream)
- set_passengers (downstream)
- set_slot (downstream)

- sound_effect (downstream)
- spawn_experience_orb (downstream)
- spawn_global_entity (downstream)
- spawn_mob (downstream)
- spawn_object (downstream)
- spawn_painting (downstream)
- spawn_player (downstream)
- spawn_position (downstream)
- spectate (upstream)
- statistics (downstream)
- status_ping (upstream)
- status_pong (downstream)
- status_request (upstream)
- status_response (downstream)
- steer_boat (upstream)
- steer_vehicle (upstream)
- stop_sound (downstream)
- tab_complete (downstream, upstream)
- tags (downstream)
- teams (downstream)
- teleport_confirm (upstream)
- time_update (downstream)
- title (downstream)
- trade_list (downstream)
- unload_chunk (downstream)
- unlock_recipes (downstream)
- update_block_entity (downstream)
- update_command_block (upstream)
- update_command_block_minecart (upstream)
- update_health (downstream)
- update_jigsaw_block (upstream)
- update_light (downstream)
- update_score (downstream)
- update_sign (upstream)
- update_structure_block (upstream)
- update_view_distance (downstream)

- update_view_position (downstream)
- use_entity (upstream)
- use_item (upstream)
- vehicle_move (downstream, upstream)
- window_items (downstream)
- window_property (downstream)
- world_border (downstream)

3.2 Data Types

3.2.1 Buffers

Quarry implements Minecraft's data types by way of the *Buffer* class.

When quarry reads a packet, it stores its payload in a buffer object and passes the buffer to a packet handler. The packet handler then unpacks the payload, which usually made up of multiple fields of differing types. You can read from the front of the buffer via the *Buffer.read()* method or any of the unpack_*() methods listed below

Buffers also provide a number of static methods that pack data into a byte string. These are named like pack_*().

When *unpacking* data you work with a buffer *object*, whereas when packing data you work with a buffer *type*. A reference to the buffer type is available from *Protocol* objects as self.buff_type.

class quarry.types.buffer.**Buffer**(*data=None*)

registry = <quarry.types.registry.OpaqueRegistry object>
An object that encodes/decodes IDs, such as blocks and items.

add(*data*)
Add some bytes to the end of the buffer.

discard()
Discards the entire buffer contents.

classmethod pack(*fmt*, **fields*)
Pack *fields* into a struct. The format accepted is the same as for struct.pack().

classmethod pack_array(*fmt*, *array*)
Packs *array* into a struct. The format accepted is the same as for struct.pack().

classmethod pack_block(*block*, *packer=None*)
Packs a block.

classmethod pack_chat(*message*)
Pack a Minecraft chat message.

classmethod pack_chunk_section(*blocks*, *block_lights=None*, *sky_lights=None*)
Packs a chunk section. The supplied argument should be an instance of quarry.types.chunk.BlockArray.

classmethod pack_command_node(*node*, *nodes*)
Packs a command node.

classmethod pack_command_node_properties(*parser*, *properties*)
Packs the properties of an argument command node.

classmethod pack_commands(*root_node*)
Packs a command graph.

classmethod pack_direction(*direction*)
Packs a direction.

classmethod pack_entity_metadata(*metadata*)
Packs entity metadata.

classmethod pack_ingredient(*ingredient*)
Packs a crafting recipe ingredient alternation.

classmethod pack_json(*obj*)
Serialize an object to JSON and pack it to a Minecraft string.

classmethod pack_nbt(*tag=None*)
Packs an NBT tag

classmethod pack_optional(*packer*, *val*)
Packs a boolean indicating whether *val* is None. If not, `packer(val)` is appended to the returned string.

classmethod pack_optional_varint(*number*)
Packs an optional varint.

classmethod pack_packet(*data*, *compression_threshold=-1*)
Unpacks a packet frame. This method handles length-prefixing and compression.

classmethod pack_particle(*kind*, *data=None*)
Packs a particle.

classmethod pack_pose(*pose*)
Packs a pose.

classmethod pack_position(*x*, *y*, *z*)
Packs a Position.

classmethod pack_recipe(*name*, *type*, ***recipe*)
Packs a crafting recipe.

classmethod pack_rotation(*x*, *y*, *z*)
Packs a rotation.

classmethod pack_slot(*item=None*, *count=1*, *tag=None*)
Packs a slot.

classmethod pack_string(*text*)
Pack a varint-prefixed utf8 string.

classmethod pack_uuid(*uuid*)
Packs a UUID.

classmethod pack_varint(*number*, *max_bits=32*)
Packs a varint.

classmethod pack_villager(*kind*, *profession*, *level*)
Packs villager data.

read(*length=None*)
Read *length* bytes from the beginning of the buffer, or all bytes if *length* is `None`

restore()
Restores the buffer contents to its state when *save()* was last called.

save()
Saves the buffer contents.

unpack(*fmt*)
Unpack a struct. The format accepted is the same as for `struct.unpack()`.

unpack_array(*fmt*, *length*)
Unpack an array struct. The format accepted is the same as for `struct.unpack()`.

unpack_block(*unpacker=None*)
Unpacks a block.

unpack_chat()
Unpack a Minecraft chat message.

unpack_chunk_section(*overworld=True*)
Unpacks a chunk section. Returns a sequence of length 4096 (16x16x16).

unpack_command_node()
Unpacks a command node.

unpack_command_node_properties(*parser*)
Unpacks the properties of an `argument` command node.

unpack_commands(*resolve_redirects=True*)
Unpacks a command graph.

If *resolve_redirects* is `True` (the default), the returned structure may contain contain circular references, and therefore cannot be serialized to JSON (or similar). If it is `False`, all node redirect information is stripped, resulting in a directed acyclic graph.

unpack_direction()
Unpacks a direction.

unpack_entity_metadata()
Unpacks entity metadata.

unpack_ingredient()
Unpacks a crafting recipe ingredient alternation.

unpack_json()
Unpack a Minecraft string and interpret it as JSON.

unpack_nbt()
Unpacks NBT tag(s).

unpack_optional(*unpacker*)
Unpacks a boolean. If it's True, return the value of `unpacker()`. Otherwise return None.

unpack_optional_varint()
Unpacks an optional varint.

unpack_packet(*cls*, *compression_threshold=-1*)
Unpacks a packet frame. This method handles length-prefixing and compression.

unpack_particle()
Unpacks a particle. Returns an `(kind, data)` pair.

unpack_pose()
Unpacks a pose.

unpack_position()
Unpacks a position.

unpack_recipe()
Unpacks a crafting recipe.

unpack_rotation()
Unpacks a rotation

unpack_slot()
Unpacks a slot.

unpack_string()
Unpack a varint-prefixed utf8 string.

unpack_uuid()
Unpacks a UUID.

unpack_varint(*max_bits=32*)
Unpacks a varint.

unpack_villager()
Unpacks villager data.

Protocol Versions

Some data types vary between Minecraft versions. Quarry automatically sets the `buff_type` attribute of `Protocol` instance to an appropriate buffer class when the protocol version becomes known.

Minecraft 1.7

Support for Minecraft 1.7+ is implemented in the `Buffer1_7` class.

Minecraft 1.9

Support for Minecraft 1.9+ is implemented in the `Buffer1_9` class.

Changes from 1.7:

- `pack_chunk_section()` and `unpack_chunk_section()` added.
- `pack_entity_metadata()` and `unpack_entity_metadata()` modified.

Minecraft 1.13

Support for Minecraft 1.13+ is implemented in the `Buffer1_13` class.

Changes from 1.9:

- `pack_commands()` and `unpack_commands()` added.
- `pack_particle()` and `unpack_particle()` added.
- `pack_recipes()` and `unpack_recipes()` added.
- `pack_chunk_section_palette()` and `unpack_chunk_section_palette()` modified.
- `pack_slot()` and `unpack_slot()` modified.
- `pack_entity_metadata()` and `unpack_entity_metadata()` modified.

Minecraft 1.13.2

Support for Minecraft 1.13.2+ is implemented in the `Buffer1_13_2` class.

Changes from 1.13:

- `pack_slot()` and `unpack_slot()` modified.

Minecraft 1.14

Support for Minecraft 1.14+ is implemented in the `Buffer1_14` class.

Changes from 1.13.2:

- `pack_villager()` and `unpack_villager()` added.
- `pack_optional_varint()` and `unpack_optional_varint()` added.
- `pack_pose()` and `unpack_pose()` added.
- `pack_chunk_section()` and `unpack_chunk_section()` modified.
- `pack_position()` and `unpack_position()` modified.
- `pack_entity_metadata()` and `unpack_entity_metadata()` modified.
- `pack_particle()` and `unpack_particle()` modified.
- `pack_recipes()` and `unpack_recipes()` modified.

3.2.2 Registry

Quarry can be told to encode/decode block, item and other information by setting the *registry* attribute on the in-use buffer. This can be set directly or by deriving a subclass and customizing *get_buff_type()*. The registry affects the following methods:

- *unpack_slot()* and *pack_slot()*
- *unpack_block()* and *pack_block()*
- *unpack_entity_metadata()* and *pack_entity_metadata()*
- *unpack_chunk_section()* and *pack_chunk_section()*
- *unpack_villager()* and *pack_villager()*
- *unpack_particle()* and *pack_particle()*

All registry objects have the following methods:

`Registry.`**`encode`**(*kind*, *obj*)
Encodes a thing to an integer ID.

`Registry.`**`decode`**(*kind*, *val*)
Decodes a thing from an integer ID.

`Registry.`**`encode_block`**(*obj*)
Encodes a block to an integer ID.

`Registry.`**`decode_block`**(*val*)
Decodes a block from an integer ID.

Registry.**is_air_block**(*obj*)
Returns true if the given object is considered air for lighting purposes.

Quarry supports the following registry types:

class quarry.types.registry.**OpaqueRegistry**(*max_bits*)
Registry that passes IDs through unchanged. This is the default.

class quarry.types.registry.**BitShiftRegistry**(*max_bits*)
Registry implementing the Minecraft 1.7 - 1.12 bit-shift format for blocks.

Blocks decode to a `(block_id, metadata)` pair. Items pass through unchanged.

class quarry.types.registry.**LookupRegistry**(*blocks*, *registries*)
Registry implementing a dictionary lookup, recommended for 1.13+.

Blocks decode to a `dict` where the only guaranteed key is `u'name'`. Items decode to a `str` name.

Use the `from_jar()` or `from_json()` class methods to load data from the official server.

classmethod from_jar(*jar_path*)
Create a `LookupRegistry` from a Minecraft server jar file. This method generates JSON files by running the Minecraft server like so:

```
java -cp minecraft_server.jar net.minecraft.data.Main --reports
```

It then feeds the generated JSON files to `from_json()`.

classmethod from_json(*reports_path*)
Create a `LookupRegistry` from JSON files generated by the official server.

3.2.3 Chat Messages

Minecraft chat is implemented in the *Message* class.

class quarry.types.chat.**Message**(*value*)
Represents a Minecraft chat message.

classmethod from_buff(*buff*)

to_bytes()

classmethod from_string(*string*)

to_string(*strip_styles=True*)
Minecraft uses a JSON format to represent chat messages; this method retrieves a plaintext representation, optionally including styles encoded using old-school chat codes (U+00A7 plus one character).

classmethod strip_chat_styles(*text*)

3.2.4 Blocks and Chunks

Minecraft uses tightly-packed arrays to store data like light levels, heightmaps and block data. Quarry can read and write these formats in both Chunk Data packets and `.mca` files. Two classes are available for working with this data:

class quarry.types.chunk.**PackedArray**(*storage*, *sector_width*, *value_width*, *fresh*)
This class provides support for an array where values are tightly packed into a number of bits (such as 4 bits for light or 9 bits for height).

All operations associated with fixed-size mutable sequences are supported, such as slicing.

Internally data is stored as a bit array with contiguous values, starting at the leftmost bits. Serializing to/from bytes is achieved by performing bitwise reversals of values and sectors; these reversals are deferred until access to packed values is needed.

Several constructors are available for specific uses of packed arrays:

- Light data used 4-bit values and 8-bit sectors
- Height data uses 9-bit values and 64-bit sectors
- Block data uses 64-bit sectors

storage = None
The `bitstring.BitArray` object used for storage.

sector_width = None
The width in bits of sectors. Used in (de)serialization.

value_width = None
The width in bits of values.

fresh = None
Whether this array is new and empty

twiddled = None
Whether this array is contiguous (assumes non-empty, non-aligned)

classmethod empty(*length*, *sector_width*, *value_width*)
Creates an empty array.

classmethod empty_light()
Creates an empty array suitable for storing light data.

classmethod empty_block()
Creates an empty array suitable for storing block data.

classmethod empty_height()
Creates an empty array suitable for storing height data.

classmethod from_bytes(*bytes*, *sector_width*, *value_width=None*)
Deserialize a packed array from the given bytes.

classmethod from_light_bytes(*bytes*)
Deserialize a packed array from the given light data bytes.

classmethod from_block_bytes(*bytes*, *value_width=None*)
Deserialize a packed array from the given block data bytes.

classmethod from_height_bytes(*bytes*)
Deserialize a packed array from the given height data bytes.

to_bytes()
Serialize this packed array to bytes.

init_storage()
Initializes the storage by performing bitwise reversals.

You should not need to call this method.

purge(*value_width*)
Re-initialize the storage to use a different value width, **destroying stored data in the process**.

You should not need to call this method.

is_empty()
Returns true if this packed array is entirely zeros.

class quarry.types.chunk.**BlockArray**(*storage*, *palette*, *registry*, *non_air=-1*)
This class provides support for block arrays. It wraps a *PackedArray* object and implements block encoding/decoding, palettes, and counting of non-air blocks for lighting purposes. It stores precisely 4096 (16x16x16) values.

All operations associated with fixed-size mutable sequences are supported, such as slicing.

A palette is used when there are fewer than 256 unique values; the value width varies from 4 to 8 bits depending on the size of the palette, and is automatically adjusted upwards as necessary. Use *repack()* to reclaim space by eliminating unused entries.

When 256 or more unique values are present, the palette is unused and values are stored directly.

storage = None
The *PackedArray* object used for storage.

palette = None
List of encoded block values. Empty when palette is not used.

registry = None
The *Registry* object used to encode/decode blocks

classmethod empty(*registry*, *non_air=-1*)
Creates an empty block array.

classmethod from_bytes(*bytes*, *palette*, *registry*, *non_air=-1*, *value_width=None*)
Deserialize a block array from the given bytes.

classmethod from_nbt(*section*, *registry*, *non_air=-1*)
Creates a block array that uses the given NBT section tag as storage for block data and the palette. Minecraft 1.13+ only.

to_bytes()
Serialize this block array to bytes.

is_empty()
Returns true if this block array is entirely air.

non_air
The number of non-air blocks

repack(*reserve=None*)
Re-packs internal data to use the smallest possible bits-per-block by eliminating unused palette entries. This operation is slow as it walks all blocks to determine the new palette.

Packets

On the client side, you can unpack a Chunk Data packet as follows:

```
def packet_chunk_data(self, buff):
    x, z, full = buff.unpack('ii?')
    bitmask = buff.unpack_varint()
    heightmap = buff.unpack_nbt()  # added in 1.14
    biomes = buff.unpack_array('I', 1024) if full else None  # changed in 1.15
    sections_length = buff.unpack_varint()
    sections = buff.unpack_chunk(bitmask)
    block_entities = [buff.unpack_nbt() for _ in range(buff.unpack_varint())]
```

On the server side:

```
def send_chunk(self, x, z, full, heightmap, sections, biomes, block_entities):
    sections_data = self.bt.pack_chunk(sections)
    self.send_packet(
        'chunk_data',
        self.bt.pack('ii?', x, z, full),
        self.bt.pack_chunk_bitmask(sections),
        self.bt.pack_nbt(heightmap),  # added in 1.14
        self.bt.pack_array('I', biomes) if full else b'',  # changed in 1.15
        self.bt.pack_varint(len(sections_data)),
        sections_data,
        self.bt.pack_varint(len(block_entities)),
        b''.join(self.bt.pack_nbt(entity) for entity in block_entities))
```

The variables used in these examples are as follows:

Variable	Value type
x	int
z	int
full	bool
bitmask	int
heightmap	TagRoot[TagCompound[TagLongArray[PackedArray]]]
sections	List[Optional[BlockArray]]
biomes	List[int]
block_entities	List[TagRoot]

Regions

Quarry can load and save data from the `.mca` format via the *RegionFile* class. NBT tags such as `"BlockStates"`, `"BlockLight"`, `"SkyLight"` and heightmaps such as `"MOTION_BLOCKING"` make their values available as *PackedArray* objects.

Use *BlockArray.from_nbt()* with a *LookupRegistry* to create a block array backed by NBT data. Modifications to the block array will automatically be reflected in the NBT data, and vice versa.

Putting these pieces together, the following function could be used to set a block in an existing region file:

```
import os.path

from quarry.types.nbt import RegionFile
from quarry.types.registry import LookupRegistry
from quarry.types.chunk import BlockArray


def set_block(server_path, x, y, z, block):
    rx, x = divmod(x, 512)
    rz, z = divmod(z, 512)
    cx, x = divmod(x, 16)
    cy, y = divmod(y, 16)
    cz, z = divmod(z, 16)

    jar_path = os.path.join(server_path, "minecraft_server.jar")
    region_path = os.path.join(server_path, "world", "region", "r.%d.%d.mca" % (rx,␣
↪rz))
```

(continues on next page)

(continued from previous page)

```
    registry = LookupRegistry.from_jar(jar_path)
    with RegionFile(region_path) as region:
        chunk, section = region.load_chunk_section(cx, cy, cz)
        blocks = BlockArray.from_nbt(section, registry)
        blocks[256 * y + 16 * z + x] = block
        region.save_chunk(chunk)


set_block("/path/to/server", 10, 80, 40, {'name': 'minecraft:bedrock'})
```

3.2.5 NBT

Quarry implements the Named Binary Tag (NBT) format. The following tag types are available from the *quarry.types.nbt* module:

Class	Value type
`TagByte`	`int`
`TagShort`	`int`
`TagInt`	`int`
`TagLong`	`int`
`TagFloat`	`float`
`TagDouble`	`float`
`TagByteArray`	*PackedArray* with 8-bit sectors
`TagIntArray`	*PackedArray* with 32-bit sectors
`TagLongArray`	*PackedArray* with 64-bit sectors
`TagString`	`str` (py3) or `unicode` (py2)
`TagList`	`list` of tags.
`TagCompound`	`dict` of names and tags.
`TagRoot`	`dict` containing a single name and tag.

Note: Unlike some other NBT libraries, a tag's name is stored by its *parent* - either a `TagRoot` or a `TagCompound`. A tag when considered alone is always nameless.

Tags

All tag types have the following attributes and methods:

`Tag.__init__`(*value*)
: Creates a tag object from the given value.

classmethod `Tag.from_bytes`(*bytes*)
: Creates a tag object from data at the beginning of the supplied byte string.

classmethod `Tag.from_buff`(*buff*)
: Creates a tag object from data at the beginning of the supplied *Buffer* object.

`Tag.to_obj`()
: Returns a friendly representation of the tag using only basic Python datatypes. This is a lossy operation, as Python has fewer data types than NBT.

`Tag.`**`to_bytes`**`()`
Returns a packed version of the tag as a byte string.

`Tag.`**`value`**
The value of the tag.

When working with NBT in relation to a *`Protocol`*, the *`Buffer.unpack_nbt()`* and *`Buffer.pack_nbt()`* methods may be helpful.

Files

You can open an NBT file using the `NBTFile` class.

class `quarry.types.nbt.`**`NBTFile`**(*root_tag*)

root_tag = None

classmethod load(*path*)

save(*path*)

You can open Minecraft 1.13+ world files (`.mca`) using the `RegionFile` class, which can also function as a context manager. See *Blocks and Chunks* for information on loading block and light data.

class `quarry.types.nbt.`**`RegionFile`**(*path*)
Experimental support for the Minecraft world storage format (`.mca`).

close()
Closes the region file.

save_chunk(*chunk*)
Saves the given chunk, which should be a `TagRoot`, to the region file.

load_chunk(*chunk_x*, *chunk_z*)
Loads the chunk at the given co-ordinates from the region file. The co-ordinates should range from 0 to 31. Returns a `TagRoot`.

load_chunk_section(*chunk_x*, *chunk_y*, *chunk_z*)
Loads the chunk section at the given co-ordinates from the region file. The co-ordinates should range from 0 to 31. Returns a `TagRoot`.

Debugging

Call `repr(tag)` or `alt_repr(tag)` for a human-readable representation of a tag.

`quarry.types.nbt.`**`alt_repr`**(*tag*, *level=0*)
Returns a human-readable representation of a tag using the same format as used the NBT specification.

3.2.6 UUIDs

Minecraft UUIDs are implemented in the *`UUID`* class.

class `quarry.types.uuid.`**`UUID`**(*hex=None*, *bytes=None*, *bytes_le=None*, *fields=None*, *int=None*, *version=None*)

classmethod from_hex(*hex*)

classmethod from_bytes(*bytes*)

classmethod from_offline_player(*display_name*)

classmethod random()

to_hex(*with_dashes=True*)

to_bytes()

3.3 Examples

The quarry source tree includes a few example uses of the `quarry` module:

```
# List examples
$ python -m examples

# Run an example
$ python -m examples.client_messenger
```

If you have `quarry` in your python search path, you can run the example files directly.

3.3.1 Clients

Pinger

```
"""
Pinger example client

This example client connects to a server in "status" mode to retrieve some
information about the server. The information returned is what you'd normally
see in the "Multiplayer" menu of the official client.
"""

from twisted.internet import reactor
from quarry.net.client import ClientFactory, ClientProtocol


class PingProtocol(ClientProtocol):

    def status_response(self, data):
        for k, v in sorted(data.items()):
            if k != "favicon":
                self.logger.info("%s --> %s" % (k, v))

        reactor.stop()


class PingFactory(ClientFactory):
    protocol = PingProtocol
    protocol_mode_next = "status"


def main(argv):
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument("host")
```

(continues on next page)

(continued from previous page)

```
    parser.add_argument("-p", "--port", default=25565, type=int)
    args = parser.parse_args(argv)

    factory = PingFactory()
    factory.connect(args.host, args.port)
    reactor.run()

if __name__ == "__main__":
    import sys
    main(sys.argv[1:])
```

Player Lister

```
"""
Player lister example client

This client requires a Mojang account for online-mode servers. It logs in to
the server and prints the players listed in the tab menu.
"""

from twisted.internet import reactor, defer
from quarry.net.client import ClientFactory, ClientProtocol
from quarry.net.auth import ProfileCLI


class PlayerListProtocol(ClientProtocol):
    def setup(self):
        self.players = {}

    def packet_player_list_item(self, buff):
        # 1.7.x
        if self.protocol_version <= 5:
            p_player_name = buff.unpack_string()
            p_online = buff.unpack('?')
            p_ping = buff.unpack('h')

            if p_online:
                self.players[p_player_name] = {
                    'name': p_player_name,
                    'ping': p_ping
                }
            elif p_player_name in self.players:
                del self.players[p_player_name]
        # 1.8.x
        else:
            p_action = buff.unpack_varint()
            p_count = buff.unpack_varint()
            for i in range(p_count):
                p_uuid = buff.unpack_uuid()
                if p_action == 0:  # ADD_PLAYER
                    p_player_name = buff.unpack_string()
                    p_properties_count = buff.unpack_varint()
                    p_properties = {}
                    for j in range(p_properties_count):
                        p_property_name = buff.unpack_string()
```

(continues on next page)

(continued from previous page)

```
                    p_property_value = buff.unpack_string()
                    p_property_is_signed = buff.unpack('?')
                    if p_property_is_signed:
                        p_property_signature = buff.unpack_string()

                    p_properties[p_property_name] = p_property_value
                p_gamemode = buff.unpack_varint()
                p_ping = buff.unpack_varint()
                p_has_display_name = buff.unpack('?')
                if p_has_display_name:
                    p_display_name = buff.unpack_chat()
                else:
                    p_display_name = None

                self.players[p_uuid] = {
                    'name': p_player_name,
                    'properties': p_properties,
                    'gamemode': p_gamemode,
                    'ping': p_ping,
                    'display_name': p_display_name
                }

            elif p_action == 1:  # UPDATE_GAMEMODE
                p_gamemode = buff.unpack_varint()

                if p_uuid in self.players:
                    self.players[p_uuid]['gamemode'] = p_gamemode
            elif p_action == 2:  # UPDATE_LATENCY
                p_ping = buff.unpack_varint()

                if p_uuid in self.players:
                    self.players[p_uuid]['ping'] = p_ping
            elif p_action == 3:  # UPDATE_DISPLAY_NAME
                p_has_display_name = buff.unpack('?')
                if p_has_display_name:
                    p_display_name = buff.unpack_chat()
                else:
                    p_display_name = None

                if p_uuid in self.players:
                    self.players[p_uuid]['display_name'] = p_display_name
            elif p_action == 4:  # REMOVE_PLAYER
                if p_uuid in self.players:
                    del self.players[p_uuid]

    def packet_chunk_data(self, buff):
        buff.discard()

        # convert self.players into a more readable format
        printable_players = []
        for data in self.players.values():
            printable_players.append((data['name'], data['ping']))

        for display_name, ping in sorted(printable_players):
            self.logger.info("%4sms %s" % (ping, display_name))

        reactor.stop()
```

(continues on next page)

(continued from previous page)

```
class PlayerListFactory(ClientFactory):
    protocol = PlayerListProtocol


@defer.inlineCallbacks
def run(args):
    # Log in
    profile = yield ProfileCLI.make_profile(args)

    # Create factory
    factory = PlayerListFactory(profile)

    # Connect!
    factory.connect(args.host, args.port)


def main(argv):
    parser = ProfileCLI.make_parser()
    parser.add_argument("host")
    parser.add_argument("-p", "--port", default=25565, type=int)
    args = parser.parse_args(argv)

    run(args)
    reactor.run()


if __name__ == "__main__":
    import sys
    main(sys.argv[1:])
```

Chat Logger

```
"""
Chat logger example client

This client stays in-game after joining. It prints chat messages received from
the server and slowly rotates (thanks c45y for the idea).
"""

from twisted.internet import reactor, defer
from quarry.net.client import ClientFactory, SpawningClientProtocol
from quarry.net.auth import ProfileCLI


class ChatLoggerProtocol(SpawningClientProtocol):
    def packet_chat_message(self, buff):
        p_text = buff.unpack_chat()

        # 1.7.x
        if self.protocol_version <= 5:
            pass
        # 1.8.x
        else:
```

(continues on next page)

(continued from previous page)

```
            p_position = buff.unpack('B')

        self.logger.info(":: %s" % p_text)


class ChatLoggerFactory(ClientFactory):
    protocol = ChatLoggerProtocol


@defer.inlineCallbacks
def run(args):
    # Log in
    profile = yield ProfileCLI.make_profile(args)

    # Create factory
    factory = ChatLoggerFactory(profile)

    # Connect!
    factory.connect(args.host, args.port)


def main(argv):
    parser = ProfileCLI.make_parser()
    parser.add_argument("host")
    parser.add_argument("-p", "--port", default=25565, type=int)
    args = parser.parse_args(argv)

    run(args)
    reactor.run()

if __name__ == "__main__":
    import sys
    main(sys.argv[1:])
```

Messenger

```
"""
Messenger example client

Bridges minecraft chat (in/out) with stdout and stdin.
"""

import os
import sys

from twisted.internet import defer, reactor, stdio
from twisted.protocols import basic
from quarry.net.auth import ProfileCLI
from quarry.net.client import ClientFactory, SpawningClientProtocol


class StdioProtocol(basic.LineReceiver):
    delimiter = os.linesep.encode('ascii')
    in_encoding  = getattr(sys.stdin,  "encoding", 'utf8')
    out_encoding = getattr(sys.stdout, "encoding", 'utf8')
```

(continues on next page)

(continued from previous page)

```
    def lineReceived(self, line):
        self.minecraft_protocol.send_chat(line.decode(self.in_encoding))

    def send_line(self, text):
        self.sendLine(text.encode(self.out_encoding))


class MinecraftProtocol(SpawningClientProtocol):
    spawned = False

    def packet_chat_message(self, buff):
        p_text = buff.unpack_chat().to_string()

        # 1.7.x
        if self.protocol_version <= 5:
            p_position = 0
        # 1.8.x
        else:
            p_position = buff.unpack('B')

        if p_position in (0, 1) and p_text.strip():
            self.stdio_protocol.send_line(p_text)

    def send_chat(self, text):
        self.send_packet("chat_message", self.buff_type.pack_string(text))


class MinecraftFactory(ClientFactory):
    protocol = MinecraftProtocol
    log_level = "WARN"

    def buildProtocol(self, addr):
        minecraft_protocol = super(MinecraftFactory, self).buildProtocol(addr)
        stdio_protocol = StdioProtocol()

        minecraft_protocol.stdio_protocol = stdio_protocol
        stdio_protocol.minecraft_protocol = minecraft_protocol

        stdio.StandardIO(stdio_protocol)
        return minecraft_protocol


@defer.inlineCallbacks
def run(args):
    # Log in
    profile = yield ProfileCLI.make_profile(args)

    # Create factory
    factory = MinecraftFactory(profile)

    # Connect!
    factory.connect(args.host, args.port)


def main(argv):
    parser = ProfileCLI.make_parser()
```

(continues on next page)

(continued from previous page)

```
    parser.add_argument("host")
    parser.add_argument("port", nargs='?', default=25565, type=int)
    args = parser.parse_args(argv)

    run(args)
    reactor.run()


if __name__ == "__main__":
    main(sys.argv[1:])
```

3.3.2 Servers

Downtime Server

```
"""
Example "downtime" server

This server kicks players with the MOTD when they try to connect. It can be
useful for when you want players to know that your usual server is down for
maintenance.
"""

from twisted.internet import reactor
from quarry.net.server import ServerFactory, ServerProtocol


class DowntimeProtocol(ServerProtocol):
    def packet_login_start(self, buff):
        buff.discard()
        self.close(self.factory.motd)


class DowntimeFactory(ServerFactory):
    protocol = DowntimeProtocol


def main(argv):
    # Parse options
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument("-a", "--host", default="", help="address to listen on")
    parser.add_argument("-p", "--port", default=25565, type=int, help="port to listen␣
↪on")
    parser.add_argument("-m", "--message", default="We're down for maintenance",
                        help="message to kick users with")
    args = parser.parse_args(argv)

    # Create factory
    factory = DowntimeFactory()
    factory.motd = args.message

    # Listen
    factory.listen(args.host, args.port)
```

(continues on next page)

(continued from previous page)

```
    reactor.run()


if __name__ == "__main__":
    import sys
    main(sys.argv[1:])
```

Auth Server

```
"""
Example "auth" server

This server authenticates players with the mojang session server, then kicks
them. Useful for server websites that ask users for a valid Minecraft account.
"""

from twisted.internet import reactor
from quarry.net.server import ServerFactory, ServerProtocol


class AuthProtocol(ServerProtocol):
    def player_joined(self):
        # This method gets called when a player successfully joins the server.
        #   If we're in online mode (the default), this means auth with the
        #   session server was successful and the user definitely owns the
        #   display name they claim to.

        # Call super. This switches us to "play" mode, marks the player as
        #   in-game, and does some logging.
        ServerProtocol.player_joined(self)

        # Define your own logic here. It could be an HTTP request to an API,
        #   or perhaps an update to a database table.
        display_name = self.display_name
        ip_addr = self.remote_addr.host
        self.logger.info("[%s authed with IP %s]" % (display_name, ip_addr))

        # Kick the player.
        self.close("Thanks, you are now registered!")


class AuthFactory(ServerFactory):
    protocol = AuthProtocol
    motd = "Auth Server"


def main(argv):
    # Parse options
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument("-a", "--host", default="", help="address to listen on")
    parser.add_argument("-p", "--port", default=25565, type=int, help="port to listen␣
↪on")
    args = parser.parse_args(argv)
```

(continues on next page)

(continued from previous page)

```
    # Create factory
    factory = AuthFactory()

    # Listen
    factory.listen(args.host, args.port)
    reactor.run()


if __name__ == "__main__":
    import sys
    main(sys.argv[1:])
```

Chat Room Server

```
"""
Example "chat room" server

This server authenticates players, then spawns them in an empty world and does
the bare minimum to keep them in-game. Players can speak to eachother using
chat.

Supports Minecraft 1.15. Earlier versions will not work as the packet formats
differ.
"""

from twisted.internet import reactor
from quarry.net.server import ServerFactory, ServerProtocol


class ChatRoomProtocol(ServerProtocol):
    def player_joined(self):
        # Call super. This switches us to "play" mode, marks the player as
        #   in-game, and does some logging.
        ServerProtocol.player_joined(self)

        # Send "Join Game" packet
        self.send_packet("join_game",
            self.buff_type.pack("iBqiB",
                0,                                  # entity id
                3,                                  # game mode
                0,                                  # dimension
                0,                                  # hashed seed
                0),                                 # max players
            self.buff_type.pack_string("flat"),     # level type
            self.buff_type.pack_varint(1),          # view distance
            self.buff_type.pack("??",
                False,                              # reduced debug info
                True))                              # show respawn screen

        # Send "Player Position and Look" packet
        self.send_packet("player_position_and_look",
            self.buff_type.pack("dddff?",
                0,                          # x
                255,                        # y
                0,                          # z
```

(continues on next page)

(continued from previous page)

```
                0,                          # yaw
                0,                          # pitch
                0b00000),                   # flags
            self.buff_type.pack_varint(0)) # teleport id

        # Start sending "Keep Alive" packets
        self.ticker.add_loop(20, self.update_keep_alive)

        # Announce player joined
        self.factory.send_chat(u"\u00a7e%s has joined." % self.display_name)

    def player_left(self):
        ServerProtocol.player_left(self)

        # Announce player left
        self.factory.send_chat(u"\u00a7e%s has left." % self.display_name)

    def update_keep_alive(self):
        # Send a "Keep Alive" packet

        # 1.7.x
        if self.protocol_version <= 338:
            payload =  self.buff_type.pack_varint(0)

        # 1.12.2
        else:
            payload = self.buff_type.pack('Q', 0)

        self.send_packet("keep_alive", payload)

    def packet_chat_message(self, buff):
        # When we receive a chat message from the player, ask the factory
        # to relay it to all connected players
        p_text = buff.unpack_string()
        self.factory.send_chat("<%s> %s" % (self.display_name, p_text))


class ChatRoomFactory(ServerFactory):
    protocol = ChatRoomProtocol
    motd = "Chat Room Server"

    def send_chat(self, message):
        for player in self.players:
            player.send_packet("chat_message",player.buff_type.pack_chat(message) +␣
↪player.buff_type.pack('B', 0) )


def main(argv):
    # Parse options
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument("-a", "--host", default="", help="address to listen on")
    parser.add_argument("-p", "--port", default=25565, type=int, help="port to listen␣
↪on")
    args = parser.parse_args(argv)

    # Create factory
```

(continues on next page)

(continued from previous page)

```
    factory = ChatRoomFactory()

    # Listen
    factory.listen(args.host, args.port)
    reactor.run()


if __name__ == "__main__":
    import sys
    main(sys.argv[1:])
```

3.3.3 Proxies

Chat Hider Proxy

```
"""
"Quiet mode" example proxy

Allows a client to turn on "quiet mode" which hides chat messages
"""

from twisted.internet import reactor
from quarry.net.proxy import DownstreamFactory, Bridge


class QuietBridge(Bridge):
    quiet_mode = False

    def packet_upstream_chat_message(self, buff):
        buff.save()
        chat_message = self.read_chat(buff, "upstream")
        self.logger.info(" >> %s" % chat_message)

        if chat_message.startswith("/quiet"):
            # Switch mode
            self.quiet_mode = not self.quiet_mode

            action = self.quiet_mode and "enabled" or "disabled"
            msg = "Quiet mode %s" % action
            self.downstream.send_packet("chat_message",
                                        self.write_chat(msg, "downstream"))

        elif self.quiet_mode and not chat_message.startswith("/"):
            # Don't let the player send chat messages in quiet mode
            msg = "Can't send messages while in quiet mode"
            self.downstream.send_packet("chat_message",
                                        self.write_chat(msg, "downstream"))

        else:
            # Pass to upstream
            buff.restore()
            self.upstream.send_packet("chat_message", buff.read())

    def packet_downstream_chat_message(self, buff):
```

(continues on next page)

(continued from previous page)

```
        chat_message = self.read_chat(buff, "downstream")
        self.logger.info(" :: %s" % chat_message)

        if self.quiet_mode and chat_message.startswith("<"):
            # Ignore message we're in quiet mode and it looks like chat
            pass

        else:
            # Pass to downstream
            buff.restore()
            self.downstream.send_packet("chat_message", buff.read())

    def read_chat(self, buff, direction):
        buff.save()
        if direction == "upstream":
            p_text = buff.unpack_string()
            return p_text
        elif direction == "downstream":
            p_text = str(buff.unpack_chat())

            # 1.7.x
            if self.upstream.protocol_version <= 5:
                p_position = 0

            # 1.8.x
            else:
                p_position = buff.unpack('B')

            if p_position in (0, 1):
                return p_text

    def write_chat(self, text, direction):
        if direction == "upstream":
            return self.buff_type.pack_string(text)
        elif direction == "downstream":
            data = self.buff_type.pack_chat(text)

            # 1.7.x
            if self.downstream.protocol_version <= 5:
                pass

            # 1.8.x
            else:
                data += self.buff_type.pack('B', 0)

            return data


class QuietDownstreamFactory(DownstreamFactory):
    bridge_class = QuietBridge
    motd = "Proxy Server"


def main(argv):
    # Parse options
    import argparse
    parser = argparse.ArgumentParser()
```

(continues on next page)

(continued from previous page)

```
    parser.add_argument("-a", "--listen-host", default="", help="address to listen on
↪")
    parser.add_argument("-p", "--listen-port", default=25565, type=int, help="port to␣
↪listen on")
    parser.add_argument("-b", "--connect-host", default="127.0.0.1", help="address to␣
↪connect to")
    parser.add_argument("-q", "--connect-port", default=25565, type=int, help="port␣
↪to connect to")
    args = parser.parse_args(argv)

    # Create factory
    factory = QuietDownstreamFactory()
    factory.connect_host = args.connect_host
    factory.connect_port = args.connect_port

    # Listen
    factory.listen(args.listen_host, args.listen_port)
    reactor.run()


if __name__ == "__main__":
    import sys
    main(sys.argv[1:])
```

3.4 Changelog

3.4.1 master

Nothing yet.

3.4.2 v1.5.1

- Added support for Minecraft 1.15.2

3.4.3 v1.5

- Added support for Minecraft 1.15 and 1.15.1
- Dropped support for Python 3.4
- Added `TagRoot.from_body()` constructor.
- Added `Message.__repr__()` method.
- Revised implementation of chunk data
  - Added dependency on `bitstring`
  - Added a new `PackedArray` class for working with tightly-packed arrays. This replaces the `LightArray` class, and additionally supports heightmaps and raw block data. This particular implementation ensures values are contiguous in memory, which speeds up gets/sets at the expense of a de/serialization process that involves two passes of bitwise reversals.
  - Reworked `BlockArray` to use `PackedArray` internally.

  - Changed the value type of NBT arrays from a `list` of `int` to a `PackedArray`. A heuristic is used to determine the value width.
  - Revised `Buffer1_14.un/pack_chunk_section()` to include arguments for block/skylight data, for consistency with earlier `Buffer` classes.
  - Added `Buffer1_9.un/pack_chunk()` methods.
  - Added `Buffer1_9.un/pack_chunk_section_array()` methods.
  - Added `Buffer1_9.pack_chunk_bitmask()` method.

3.4.4 v1.4

- Added support for Minecraft 1.14.3 and 1.14.4
- Fixed support for Minecraft 1.7

3.4.5 v1.3

- Added support for Minecraft 1.14 - 1.14.2
  - BREAKING CHANGE! `BlockMap` objects are replaced by `Registry` objects with greater responsibilities, reflecting the increase in information generated by the official server when run with `--reports`. Villager and particle data is now decoded when using a `LookupRegistry` in a buffer. Other information (for example, mob names from IDs) can be decoded in packet handlers.
  - `BlockArray` objects now track the number of non-air blocks, which is conveyed in `chunk_data` packets.
  - Added methods for packing/unpacking optional varints, rotation, direction, villager and pose data.

3.4.6 v1.2

- Added support for Minecraft 1.13.2
- Fixed support for server icons (thanks @dries007) and added caching.

3.4.7 v1.1.1

- Various bugfixes.

3.4.8 v1.1

- Added support for Minecraft 1.13.
  - Added 1.13 packet enumeration.
  - The wire format of chunk sections, entity metadata and slots has changed. Slots no longer contain a 'damage' field.
  - Added methods for packing/unpacking particles and command graphs.
  - Clients now respond to `login_plugin_request` messages with a `login_plugin_response` indicating that the client didn't understand the request. Like other quarry packet handlers, this method can be overridden in a subclass to implement a custom authentication flow.

- Slot/blocks/chunks/regions improvements:
  - Added `quarry.types.block` module, containing three classes for handling block and item IDs:
    - `OpaqueBlockMap` passes IDs through unchanged
    - `BitShiftBlockMap` decodes blocks by bit-shifting - this format is used in Minecraft 1.7 through 1.12. Item IDs pass through unchanged.
    - `LookupBlockMap` decodes by looking up in a dictionary. This class has `from_jar()` and `from_json()` methods for loading this dictionary from the official server (1.13+).

    `Buffer` types gain a `block_map` attribute. By default this is an `OpaqueBlockMap(13)`. The buffer's block map is consulted by methods that deal with slots, entity metadata and chunk data.

    `BlockArray` objects must now be given a block map on initialization, and will pass getitem/setitem values through the map.
  - Added `quarry.types.nbt.RegionFile` class, which supports reading and writing NBT chunk data to `.mca` files.
  - `BlockArray` and `LightArray` now support a `from_nbt()` class method. This creates an array that is a view on to an NBT compound tag representing a section (as might be retrieved via a `RegionFile`). Supports Minecraft 1.13+ only.
  - `BlockArray.palette` is now an empty list rather than `None` when a palette is not in use
  - Added `Buffer.pack_block()` and `Buffer.unpack_block()` methods.
  - Slot dictionaries now use an `'item'` key to store the item identifier, rather than `'id'`. An empty slot is now represented with an `'item'` value of `None` rather than `-1`.
- Added `quarry.types.nbt.TagLongArray` class.
- Added `quarry.types.nbt.TagRoot.body` property to retrieve the child tag.
- Added `quarry.types.nbt._Tag.from_bytes()` method.
- Added `quarry.types.uuid.UUID.random()` constructor.
- Added `Protocol.get_packet_name()` and `Protocol.get_packet_ident()` methods. These can be overridden to support custom packet name lookup behaviour.
- Moved `PacketDispatcher.dump_packet()` to `Buffer.hexdump()`.
- Fixed unpacking of byte entity metadata.
- Fixed NBT handling of 1-length arrays.
- Fixed `SpawningClientProtocol` not responding to keep-alives.
- Fixed unicode handling in chat unpacking.

3.4.9 v1.0

- Changes to `quarry.types.buffer`:
  - Split `Buffer` into `Buffer1_7` and `Buffer_1_9`, and select an appropriate buffer type by protocol version. This is done in anticipation of revisions to the slot and entity metadata formats in 1.13.
  - Moved some packet framing logic from `Protocol` into `Buffer.pack_packet()` and `Buffer.unpack_packet()`
  - Added `Buffer.pack_optional()` and `Buffer.unpack_optional()`, which handle boolean-prefixed optional data.

  - Added `Buffer.pack_array()` and `Buffer.unpack_array()` convenience methods.
  - Made `Buffer.pack_entity_metadata()` and `Buffer.unpack_entity_metadata()` work with a dictionary rather than a list of tuples. Also corrected a couple of issues with re-packing data.
  - Removed the `signed` argument from `Buffer.pack_varint()` and `Buffer.unpack_varint()`. All varints are now signed.
- Changes to `quarry.types.chunk`:
  - Made `BlockArray` setitem/getitem accept/return an opaque ID, rather than a 2-tuple of `(block_id, metadata)`. In Minecraft 1.13 it's no longer possible to convert between the two with bitshifting.
  - Added `BlockArray.empty()` and `LightArray.empty()` methods to initialize empty (zero-filled) block/light arrays.
  - Added `BlockArray.is_empty()` method, which can be used by servers to check whether a chunk section should be sent.
- Changes to `quarry.types.nbt`:
  - Added `TagCompound.update()` method, which performs a "deep" update of an NBT tree.
- Changes to `quarry.net`:
  - Added `Proxy.disable_forwarding()`
  - `ClientFactory.connect()` no longer accepts `protocol_mode_next` and `protocol_version` arguments.
  - `ServerFactory.force_protocol_version` has moved to `Factory.force_protocol_version`, and is now observed by clients.
  - `ClientProtocol.protocol_mode_next` has moved to `ClientFactory.protocol_mode_next`, and now defaults to "login".
  - Removed `Protocol.compression_enabled`. Uncompressed connections are now indicated by `Protocol.compression_threshold == -1`.
  - Modified `Profile.validate()` to not automatically attempt to refresh invalid profiles. This should be an explicit user choice.
  - Added `Profile.to_file()`, which saves to a JSON file containing a subset of the information available in `~/.minecraft/launcher_profiles.json`.
  - Fixed restarting a stopped `Ticker`.
- Fixed `client_messenger` chat unpacking.
- Fixed the `entity_properties` and `advancements` packets being swapped.

3.4.10 v0.9.1

- Dropped support for Python 3.3.
- Fixed Python 3.4+ compatibility issues.
- Made `SpawningClientProtocol` send `player_position_and_look` rather than `player_position`.
- Fixed ticker logger being `None`.

3.4.11 v0.9

- Added support for Minecraft 1.12.2.
- Added documentation for proxies
- Added a "fast forwarding" mode for proxies that skips packing/unpacking of packets.
- Re-arranged some proxy internals.
- Replaced `quarry.net.tasks` with `quarry.net.ticker`. An instance of the `Ticker` class is available as `self.ticker` from protocols. This object has `add_delay()` and `add_loop()` methods for setting up delayed and repeating tasks respectively. The interface similar to the previous `Tasks` object, except that timings are now given in ticks rather than seconds. The new tick loop is closer to the vanilla minecraft tick loop: delayed ticks are run faster the usual, and when too many ticks are queued they are skipped altogether.
- Added `quarry.types.chat` module for handling Minecraft chat. Chat packing/unpacking methods in `Buffer` now accept/return an instance of the `Message` class.
- Added `Buffer.pack_slot()` method.
- Added `Buffer.pack_entity_metadata()` and `Buffer.unpack_entity_metadata()` methods.
- Added `ServerFactory.prevent_proxy_connections` attribute, defaulting to `True`, that prevents clients from connecting via a proxy. Note that this specifically affects online mode, and works by comparing the IP of the connecting client with the IP recorded as making the authentication request with the Mojang session server.

3.4.12 v0.8

- Added support for Minecraft 1.12.1. Thanks to Christian Hogan for the patch.

3.4.13 v0.7

- Added support for Minecraft 1.12
- Several breaking changes! Read on for more.
- Removed the `quarry.utils` package. Its contents have been distributed as follows:
  - The `buffer`, `chunk`, `nbt` and `uuid` (renamed from `types`) modules have moved into a new `quarry.types` package.
  - The `auth`, `crypto`, `http` and `tasks` modules have moved into the `quarry.net` package.
  - The `error` module was removed. `ProtocolError` is now part of `quarry.net.protocol`.
- Revised the NBT implementation
  - `TagByteArray` and `TagIntArray` have more straightforward signatures for `__init__` and `from_buff`
  - `TagList` now stores its contents as a list of tags, rather than a list of tag *values*. It no longer accepts an `inner_kind` initialiser parameter, as this is derived from the type of the first supplied tag.
  - `NamedTag` is removed.
  - `TagCompound` now stores its value as a `dict` of names and tags, rather than a `list` of `NamedTag` objects.

  - `TagRoot` is introduced as the top-level tag. This is essentially a `TagCompound` containing a single record.
  - Added a new `alt_repr` function that prints a tag using the same representation as in the NBT specification.
  - Improved performance.
  - Added some tests.
- Substantially expanded documentation.
- Added a new `server_chat_room` example. This server spawns a player in an empty world and allows player to chat to eachother.
- Made `Protocol.send_packet()` accept any number of `data` arguments, which are concatenated together.
- Made `Buffer.__init__()` accept a string argument, which is equivalent to creating an empty buffer and calling `add()`.
- Added `Buffer.pack_chunk_section()` and `Buffer.unpack_chunk_section()`. These methods work with new `quarry.types.chunk` types: `LightArray` (4 bits per block) and `BlockArray` (4-8 or 13 bits per block, with an optional palette).
- Added `Buffer.pack_position()`, which packs co-ordinates into a `long` and complements `Buffer.unpack_position()`.
- Added a `Bridge.make_profile()` method, which is called to provide a profile to the `UpstreamFactory`. The default implementation generates an offline profile with the same display name as the `Downstream`.

3.4.14 v0.6.3

- Fix bundle

3.4.15 v0.6.2

- Added support for Minecraft 1.11.2
- Added a default implementation for the "disconnect" packet, which now does the same thing as "login_disconnect", i.e. logs a warning and closes the connection.

3.4.16 v0.6.1

- Fix bundle

3.4.17 v0.6

- Added support for Minecraft 1.11
- BREAKING CHANGES!
  - Throughout the codebase, references to `username` have changed to `display_name` for consistency with Mojang's terminology.
  - `Factory.run()` and `Factory.stop()` have been removed for being misleading about the role of factories. Use twisted's `reactor.run()` instead.

  - `quarry.mojang` has been renamed to `quarry.auth` and substantially rewritten.
  - Offline profiles are now represented by `OfflineProfile` objects.
  - Online profiles have a number of new static creator methods: - `from_credentials()` accepts an email address and password - `from_token()` accepts a client and access token, display name and UUID - `from_file()` loads a profile from the Mojang launcher.
  - A new `ProfileCLI` class provides a couple of useful methods for creating profiles from command-line arguments.
  - Profiles must now be provided to the `ClientFactory` initializer, rather than set as a class variable. When a profile is not given, an offline profile is used. In proxies, the initialiser for `UpstreamFactory` must be re-implemented if the proxy connects to the backing server in online mode.
  - `Factory.auth_timeout` has moved to `ServerFactory.auth_timeout`. Clients now use `Profile.timeout` when calling `/join` endpoint.
- `ClientFactory.connect` returns a deferred that will fire after after `reactor.connectTCP` is called for the last time. Usually there is a small time delay before this happens while quarry queries the server's version.
- Clients will refresh a profile if `/join` indicates a token is invalid, then retry the `/join` once.
- Added a new `SpawningClientProtocol` class that implements enough packets to keep a player in-game
- Added a new `client_messenger` example. This bridges minecraft chat (in/out) with stdout and stdin.

3.4.18 v0.5

- Added `Buffer.unpack_nbt()` and `Buffer.pack_nbt()` methods for working with the NBT (Named Binary Tag) format.
- Added `Buffer.unpack_position()` method. This unpacks a 26/12/26-packed position.
- Added `strip_styles` parameter to `Buffer.unpack_chat()`. If set to *false*, text is returned including old-style style escapes (U+00A7 plus a character)
- A stopping client factory no longer invalidates its profile.
- Added Python 3 compatibility to `PacketDispatcher.dump_packet()`
- Fix tests for `Buffer.unpack_chat()`

3.4.19 v0.4

- Added support for Minecraft 1.10
- Added support for Minecraft 1.9.3 and 1.9.4
- Improved the varint implementation - it now supports signed and magnitude-limited numbers. Also added some sensible defaults to various bits of quarry that use varints.
- Made `Buffer.unpack_chat()` not add curly braces to "translate" objects without accompanying "with" objects.
- Made `Buffer.unpack_chat()` strip old-style (u00A7) chat escapes.

3.4.20 v0.3.1

- Added support for Minecraft 1.9.1 and 1.9.2
- Fixed protocol error in example chat logger when connecting to 1.9 servers

3.4.21 v0.3

- Added support for Minecraft 1.9
- Compression is now supported in servers
- Servers will now reject new connections when full
- Servers will now report a forced protocol version in status responses, rather than repeating the client's version.
- The point at which a proxy will connect to the upstream server is now customisable.
- Renamed "maps" packet to "map"
- Renamed "sign editor open" packet to "open sign editor"
- Renamed `ServerFactory.favicon_path` to `ServerFactory.favicon`
- Renamed `quarry.util` to `quarry.utils`
- Removed `protocol_mode` parameter from some proxy callbacks
- Added many new docstrings; made documentation use Sphinx's `autodoc`
- Fixed exception handling when looking up a packet name. Thanks to PangeaCake for the fix.
- Fixed issue where an exception was raised when generating an offline-mode UUID in Python 3. Thanks to PangeaCake for the report.
- Fixed issue with compression in proxies when the upstream server set the compression threshold after passthrough had been enabled. Thanks to PangeaCake for the report.
- (tests) `quarry.utils.buffer` and `quarry.utils.types` are now covered.

3.4.22 v0.2.3

- (documentation) Fixed changelog for v0.2.2

3.4.23 v0.2.2

- Fixed proxies
- (documentation) Added changelog

3.4.24 v0.2.1

- (documentation) Fixed front page

3.4.25 v0.2

- Tentative Python 3 support
- Removed `@register`. Packet handlers are now looked up by method name
- Packets are now addressed by name, rather than mode and ident
- `Protocol.recv_addr` renamed to `Protocol.remote_addr`
- Client profile is automatically invalidated when `ClientFactory` stops
- (internals) `PacketDispatcher` moved from `quarry.util` to `quarry.net`
- (examples) Chat logger now closely emulates vanilla client behaviour when sending "player"
- (documentation) It now exists!

3.4.26 v0.1

- Initial release

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

q

Index

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

Q

V