Implementation of websockets I did cane up with.

There are many implementations of WebSockets everywhere but the core logic is the same,

Implementation 1

The logic is in **read_next_message**, I customized the implementation made by **Johan Hanssen Seferidis**.

```
import errno, socket, threading, hashlib, struct, socketserver, base64,

"""
+-+-+-+-+-------+-+-------------+-------------------------------+
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-------+-+-------------+-------------------------------+
|F|R|R|R| opcode|M| Payload len |    Extended payload length    |
|I|S|S|S|  (4)  |A|     (7)     |             (16/64)           |
|N|V|V|V|       |S|             |   (if payload len==126/127)   |
| |1|2|3|       |K|             |                               |
+-+-+-+-+-------+-+-------------+ - - - - - - - - - - - - - - - +
|     Extended payload length continued, if payload len == 127  |
+ - - - - - - - - - - - - - - - +-------------------------------+
|                     Payload Data continued ...                |
+---------------------------------------------------------------+
"""


FIN = 0x80
OPCODE = 0x0F
MASKED = 0x80
PAYLOAD_LEN = 0x7F
PAYLOAD_LEN_EXT16 = 0x7E
PAYLOAD_LEN_EXT64 = 0x7F

OPCODE_CONTINUATION = 0x0
OPCODE_TEXT = 0x1
OPCODE_BINARY = 0x2
OPCODE_CLOSE_CONN = 0x8
OPCODE_PING = 0x9
OPCODE_PONG = 0xA

CLOSE_STATUS_NORMAL = 1000
```

```
DEFAULT_CLOSE_REASON = bytes("", encoding="utf-8")


HANDSHAKE_STR = (
    "HTTP/1.1 101 Switching Protocols\r\n"
    "Upgrade: WebSocket\r\n"
    "Connection: Upgrade\r\n"
    "Sec-WebSocket-Accept: %(acceptstr)s\r\n\r\n"
)

FAILED_HANDSHAKE_STR = (
    "HTTP/1.1 426 Upgrade Required\r\n"
    "Upgrade: WebSocket\r\n"
    "Connection: Upgrade\r\n"
    "Sec-WebSocket-Version: 13\r\n"
    "Content-Type: text/plain\r\n\r\n"
    "This service requires use of the WebSocket protocol\r\n"
)

LOGGER = logging.getLogger(__name__)
logging.basicConfig()


def encode_to_UTF8(data):
    try:
        return data.encode("UTF-8")
    except UnicodeEncodeError as e:
        LOGGER.error("Could not encode data to UTF-8 -- %s" % e)
        return False
    except Exception as e:
        raise (e)


def try_decode_UTF8(data):
    try:

        return data.decode("utf-8")
    except UnicodeDecodeError as e:
        LOGGER.error("Could not decode data to UTF-8 -- %s" % e)
        return False
    except Exception as e:
        raise (e)


class AmeboServerAPI:
    def run_forever(self, threaded=False):
```

```
        return self._run_forever(threaded)

    def new_client(self, client, server):
        pass

    def client_left(self, client, server):
        pass

    def message_received(self, client, server, message):
        pass

    def set_fn_new_client(self, fn):
        self.new_client = fn

    def set_fn_client_left(self, fn):
        self.client_left = fn

    def set_fn_message_received(self, fn):
        self.message_received = fn

    def send_message(self, client, msg):
        self._unicast(client, msg)

    def send_message_to_all(self, msg):
        self._multicast(msg)

    def deny_new_connections(
        self, status=CLOSE_STATUS_NORMAL, reason=DEFAULT_CLOSE_REASON
    ):
        self._deny_new_connections(status, reason)

    def allow_new_connections(self):
        self._allow_new_connections()

    def shutdown_gracefully(

        self, status=CLOSE_STATUS_NORMAL, reason=DEFAULT_CLOSE_REASON
    ):
        self._shutdown_gracefully(status, reason)

    def shutdown_abruptly(self):
        self._shutdown_abruptly()

    def disconnect_clients_gracefully(
        self, status=CLOSE_STATUS_NORMAL, reason=DEFAULT_CLOSE_REASON
    ):
```

```
        self._disconnect_clients_gracefully(status, reason)

    def disconnect_clients_abruptly(self):
        self._disconnect_clients_abruptly()


class AmeboWebSocketServer(socketserver.ThreadingTCPServer, AmeboServerA

    allow_reuse_address = True
    request_queue_size = 10

    # daemon_threads = True  # comment to keep threads alive until finis
    def __init__(
        self,
        server_address: tuple[str, int],
        RequestHandlerClass: "AmeboWebSocketHandler",
        log_level=logging.WARNING,
    ) -> None:
        super().__init__(server_address, RequestHandlerClass)

        LOGGER.setLevel(log_level)

        self.clients_handlers: list[AmeboWebSocketHandler] = []
        self.id_counter = 0
        self.thread: threading.Thread = None
        self._deny_clients = False

    def start_server(self, threaded_serving: bool = True):
        cls_name = self.__class__.__name__
        try:
            if threaded_serving:
                self.thread = threading.Thread(target=self.serve_forever
                LOGGER.info(f"Starting {cls_name} on thread {self.thread

                self.thread.start()


            else:
                self.thread = threading.current_thread()
                LOGGER.info(f"Starting {cls_name} on main thread.")

                self.serve_forever()
        except KeyboardInterrupt:
            LOGGER.info("Server terminated.")
            self.server_close()
        except Exception as e:
```

```
            LOGGER.error(str(e), exc_info=True)
            os.sys.exit(1)

    def _deny_new_connections(self, status, reason):
        self._deny_clients = {
            "status": status,
            "reason": reason,
        }

    def _allow_new_connections(self):
        self._deny_clients = False

    def _new_client_(self, handler: "AmeboWebSocketHandler"):
        if self._deny_clients:
            status = self._deny_clients["status"]
            reason = self._deny_clients["reason"]
            handler.send_close(status, reason)
            self._terminate_client(handler)
            return

        self.clients_handlers.append(handler)
        self.new_client(handler, self)

    def _client_left_(self, client_handler):
        self.client_left(client_handler, self)
        if client_handler in self.clients_handlers:
            self.clients_handlers.remove(client_handler)

    def _multicast(self, msg):
        for client in self.clients_handlers:
            client.send_message(msg)

    def _terminate_client_handler(self, handler: "AmeboWebSocketHandler"
        handler.keep_alive = False
        handler.finish()

        handler.connection.close()

    def _terminate_client_handlers(self):
        """
        Ensures request handler for each client is terminated correctly
        """
        for client in self.clients_handlers:
            self._terminate_client_handler(client)

    def _shutdown_gracefully(
```

```
        self, status=CLOSE_STATUS_NORMAL, reason=DEFAULT_CLOSE_REASON
    ):
        """
        Send a CLOSE handshake to all connected clients before terminati
        """
        self.keep_alive = False
        self._disconnect_clients_gracefully(status, reason)
        self.server_close()
        self.shutdown()

    def _shutdown_abruptly(self):
        """
        Terminate server without sending a CLOSE handshake
        """
        self.keep_alive = False
        self._disconnect_clients_abruptly()
        self.server_close()
        self.shutdown()

    def _disconnect_clients_gracefully(
        self, status=CLOSE_STATUS_NORMAL, reason=DEFAULT_CLOSE_REASON
    ):
        """
        Terminate clients gracefully without shutting down the server
        """
        for client in self.clients:
            client["handler"].send_close(status, reason)
        self._terminate_client_handlers()

    def _disconnect_clients_abruptly(self):
        """
        Terminate clients abruptly (no CLOSE handshake) without shutting
        """
        self._terminate_client_handlers()



class AmeboWebSocketHandler(socketserver.StreamRequestHandler):
    @classmethod
    def make_handshake_response(cls, key):
        return (
            "HTTP/1.1 101 Switching Protocols\r\n"
            "Upgrade: websocket\r\n"
            "Connection: Upgrade\r\n"
            "Sec-WebSocket-Accept: %s\r\n"
            "\r\n" % cls.calculate_response_key(key)
```

```
        )

    @classmethod
    def calculate_response_key(cls, key):
        GUID = "258EAFA5-E914-47DA-95CA-C5AB0DC85B11"
        hash = hashlib.sha1(key.encode() + GUID.encode())
        response_key = base64.b64encode(hash.digest()).strip()
        return response_key.decode("ASCII")

    def setup(self):
        super().setup()

        self.keep_alive = True
        self.handshake_done = False
        self.valid_client = False

    def handle(self):
        while self.keep_alive:
            if not self.handshake_done:
                self.handshake()
            elif self.valid_client:
                self.read_next_message()

    def read_bytes(self, num):
        return self.rfile.read(num)

    def read_http_headers(self):
        headers = {}
        # first line should be HTTP GET
        http_get = self.rfile.readline().decode().strip()
        assert http_get.upper().startswith("GET")
        # remaining should be headers
        while True:
            header = self.rfile.readline().decode().strip()
            if not header:

                break
            head, value = header.split(":", 1)
            headers[head.lower().strip()] = value.strip()
        return headers

    def handshake(self):
        headers = self.read_http_headers()

        try:
            assert headers["upgrade"].lower() == "websocket"
```

```
        except AssertionError:
            self.keep_alive = False

        try:
            key = headers["sec-websocket-key"]
        except KeyError:
            LOGGER.warning("Client tried to connect but was missing a ke
            self.keep_alive = False

        if not self.keep_alive:
            self.request.send(FAILED_HANDSHAKE_STR.encode("ascii"))
            return

        response = self.make_handshake_response(key)
        with self._send_lock:
            self.handshake_done = self.request.send(response.encode())
        self.valid_client = True
        self.server._new_client_(self)

    def read_next_message(self):
        try:
            b1, b2 = self.read_bytes(2)
        except socket.SocketError as e:  # to be replaced with Connectio
            if e.errno == errno.ECONNRESET:
                LOGGER.info("Client closed connection.")
                self.keep_alive = 0
                return
            b1, b2 = 0, 0
        except ValueError as e:
            b1, b2 = 0, 0

        fin = b1 & FIN
        opcode = b1 & OPCODE
        masked = b2 & MASKED
        payload_length = b2 & PAYLOAD_LEN


        if opcode == OPCODE_CLOSE_CONN:
            LOGGER.info("Client asked to close connection.")
            self.keep_alive = 0
            return
        if not masked:
            LOGGER.warning("Client must always be masked.")
            self.keep_alive = 0
            return
        if opcode == OPCODE_CONTINUATION:
```

```
            LOGGER.warning("Continuation frames are not supported.")
            return
        elif opcode == OPCODE_BINARY:
            LOGGER.warning("Binary frames are not supported.")
            return
        elif opcode == OPCODE_TEXT:
            opcode_handler = self.message_received
        elif opcode == OPCODE_PING:
            opcode_handler = self.ping_received
        elif opcode == OPCODE_PONG:
            opcode_handler = self.pong_received
        else:
            LOGGER.warning("Unknown opcode %#x." % opcode)
            self.keep_alive = 0
            return

        if payload_length == 126:
            payload_length = struct.unpack(">H", self.rfile.read(2))[0]
        elif payload_length == 127:
            payload_length = struct.unpack(">Q", self.rfile.read(8))[0]

        masks = self.read_bytes(4)
        message_bytes = bytearray()
        for message_byte in self.read_bytes(payload_length):
            message_byte ^= masks[len(message_bytes) % 4]
            message_bytes.append(message_byte)
        opcode_handler(self, message_bytes.decode("utf8"))

    def send_text(self, message, opcode=OPCODE_TEXT):
        """
        Important: Fragmented(=continuation) messages are not supported
        their usage cases are limited - when we don't know the payload l
        """

        # Validate message

        if isinstance(message, bytes):
            message = try_decode_UTF8(
                message
            )  # this is slower but ensures we have UTF-8
            if not message:
                LOGGER.warning("Can't send message, message is not valid
                return False
        elif not isinstance(message, str):
            LOGGER.warning(
                "Can't send message, message has to be a string or bytes
```

```
                % type(message)
            )
            return False

        header = bytearray()
        payload = encode_to_UTF8(message)
        payload_length = len(payload)

        # Normal payload
        if payload_length <= 125:
            header.append(FIN | opcode)
            header.append(payload_length)

        # Extended payload
        elif payload_length >= 126 and payload_length <= 65535:
            header.append(FIN | opcode)
            header.append(PAYLOAD_LEN_EXT16)
            header.extend(struct.pack(">H", payload_length))

        # Huge extended payload
        elif payload_length < 18446744073709551616:
            header.append(FIN | opcode)
            header.append(PAYLOAD_LEN_EXT64)
            header.extend(struct.pack(">Q", payload_length))

        else:
            raise Exception("Message is too big. Consider breaking it in

        with self._send_lock:
            self.request.send(header + payload)

    def send_close(self, status=CLOSE_STATUS_NORMAL, reason=DEFAULT_CLOS
        """
        Send CLOSE to client


        Args:
            status: Status as defined in https://datatracker.ietf.org/do
            reason: Text with reason of closing the connection
        """
        if status < CLOSE_STATUS_NORMAL or status > 1015:
            raise Exception(f"CLOSE status must be between 1000 and 1015

        header = bytearray()
        payload = struct.pack("!H", status) + reason
        payload_length = len(payload)
```

```
        assert (
            payload_length <= 125
        ), "We only support short closing reasons at the moment"

        # Send CLOSE with status & reason
        header.append(FIN | OPCODE_CLOSE_CONN)
        header.append(payload_length)
        with self._send_lock:
            self.request.send(header + payload)

    def send_pong(self, message):
        self.send_text(message, OPCODE_PONG)

    def finish(self):
        self.server: AmeboWebSocketServer
        self.server._client_left_(self)

    def message_received(self, message):
        ...
    def ping_received(self, message):
        ...
    def pong_received(self, message):
        ...
```

Implementation 2

```
 **Dave P.** Logic in **handlePacket**



from http.server import BaseHTTPRequestHandler, HTTPServer, SimpleHTTPRe

import hashlib, base64, socket, struct, ssl, errno, codecs, signal, io,

from collections import deque

from select import select



__all__ = ["WebSocket", "WebSocketServer", "SSLWebSocketServer"]
```

```
class HTTPRequest(BaseHTTPRequestHandler):
    def __init__(self, request_text):
        self.rfile = io.BytesIO(request_text)
        self.raw_requestline = self.rfile.readline()
        self.error_code = self.error_message = None
        self.parse_request()




_VALID_STATUS_CODES = [
    1000,
    1001,
    1002,
    1003,
    1007,
    1008,
    1009,
    1010,
    1011,
    3000,
    3999,
    4000
```

```
    4000,
    4999,
]

HANDSHAKE_STR = (
    "HTTP/1.1 101 Switching Protocols\r\n"
    "Upgrade: WebSocket\r\n"
    "Connection: Upgrade\r\n"
    "Sec-WebSocket-Accept: %(acceptstr)s\r\n\r\n"
)

FAILED_HANDSHAKE_STR = (
    "HTTP/1.1 426 Upgrade Required\r\n"
    "Upgrade: WebSocket\r\n"
    "Connection: Upgrade\r\n"
    "Sec-WebSocket-Version: 13\r\n"
    "Content-Type: text/plain\r\n\r\n"
    "This service requires use of the WebSocket protocol\r\n"
)

GUID_STR = "258EAFA5-E914-47DA-95CA-C5AB0DC85B11"

STREAM = 0x0
```

```
TEXT = 0x1

BINARY = 0x2

CLOSE = 0x8

PING = 0x9

PONG = 0xA


HEADERB1 = 1

HEADERB2 = 3


LENGTHSHORT = 4

LENGTHLONG = 5

MASK = 6

PAYLOAD = 7


MAXHEADER = 65536

MAXPAYLOAD = 33554432




class WebSocket:

    def __init__(self, server, sock, address):

        self.server = server

        self.client = sock

        self.address = address
```

```
        self.handshaked = False
        self.headerbuffer = bytearray()
        self.headertoread = 2048

        self.fin = 0
        self.data = bytearray()
        self.opcode = 0
        self.hasmask = 0

        self.maskarray = None
        self.length = 0
        self.lengtharray = None
        self.index = 0
        self.request = None
        self.usingssl = False

        self.frag_start = False
        self.frag_type = BINARY
        self.frag_buffer = None
        self.frag_decoder = codecs.getincrementaldecoder("utf-8")(errors
        self.closed = False
        self.sendq = deque()
```

```
        self.state = HEADERB1

        # restrict the size of header and payload for security reasons
        self.maxheader = MAXHEADER
        self.maxpayload = MAXPAYLOAD

    def handleMessage(self):
        """
        Called when websocket frame is received.
        To access the frame data call self.data.

        If the frame is Text then self.data is a unicode object.
        If the frame is Binary then self.data is a bytearray object.
        """
        pass

    def handleConnected(self):
        """
        Called when a websocket client connects to the server.
        """
        pass

    def handleClose(self):
```

```
    def handleClose(self):
        """
        Called when a websocket server gets a Close frame from a client.
        """
        pass


    def _handlePacket(self):
        if self.opcode in [PONG, PING]:
            if len(self.data) > 125:
                raise Exception("control frame length can not be > 125")


        elif self.opcode not in [CLOSE, STREAM, TEXT, BINARY]:
            # unknown or reserved opcode so just close
            raise Exception("unknown opcode")


        if self.opcode == CLOSE:
            status = 1000
            reason = ""
            length = len(self.data)


            if length == 0:
                pass
            elif length >= 2:
```

```
            status = struct.unpack_from("!H", self.data[:2])[0]
            reason = self.data[2:]

            if status not in _VALID_STATUS_CODES:
                status = 1002

            if len(reason) > 0:
                try:
                    reason = reason.decode("utf8", errors="strict")
                except:
                    status = 1002
        else:
            status = 1002

        self.close(status, reason)
        return

    elif self.fin == 0:
        if self.opcode != STREAM:
            if self.opcode == PING or self.opcode == PONG:
                raise Exception("control messages can not be fragmen

            self.frag_type = self.opcode
```

```
            self.frag_type = self.opcode
            self.frag_start = True
            self.frag_decoder.reset()

            if self.frag_type == TEXT:
                self.frag_buffer = []
                utf_str = self.frag_decoder.decode(self.data, final=
                if utf_str:
                    self.frag_buffer.append(utf_str)
            else:
                self.frag_buffer = bytearray()
                self.frag_buffer.extend(self.data)

        else:
            if self.frag_start is False:
                raise Exception("fragmentation protocol error")

            if self.frag_type == TEXT:
                utf_str = self.frag_decoder.decode(self.data, final=
                if utf_str:
                    self.frag_buffer.append(utf_str)
            else:
                self.frag_buffer.extend(self.data)
```

```
        else:
            if self.opcode == STREAM:
                if self.frag_start is False:
                    raise Exception("fragmentation protocol error")

                if self.frag_type == TEXT:
                    utf_str = self.frag_decoder.decode(self.data, final=

                    self.frag_buffer.append(utf_str)
                    self.data = "".join(self.frag_buffer)
                else:
                    self.frag_buffer.extend(self.data)
                    self.data = self.frag_buffer

                self.handleMessage()

                self.frag_decoder.reset()
                self.frag_type = BINARY
                self.frag_start = False
                self.frag_buffer = None

            elif self.opcode == PING:
                self.sendMessage(False, PONG, self.data)
```

```
                self._sendMessage(False, PONG, self.data)

            elif self.opcode == PONG:
                pass

            else:
                if self.frag_start is True:
                    raise Exception("fragmentation protocol error")

                if self.opcode == TEXT:
                    try:
                        self.data = self.data.decode("utf8", errors="str
                    except Exception as exp:
                        raise Exception("invalid utf-8 payload")

                self.handleMessage()

    def _handleData(self):
        # do the HTTP header and handshake
        if self.handshaked is False:

            data = self.client.recv(self.headertoread)
            if not data:
```

```
                raise Exception("remote socket closed")


            else:
                # accumulate
                self.headerbuffer.extend(data)

                if len(self.headerbuffer) >= self.maxheader:
                    raise Exception("header exceeded allowable size")


                # indicates end of HTTP header
                if b"\r\n\r\n" in self.headerbuffer:
                    self.request = HTTPRequest(self.headerbuffer)

                    # handshake rfc 6455
                    try:
                        key = self.request.headers["Sec-WebSocket-Key"]
                        k = key.encode("ascii") + GUID_STR.encode("ascii
                        k_s = base64.b64encode(hashlib.sha1(k).digest())
                        hStr = HANDSHAKE_STR % {"acceptstr": k_s}
                        self.sendq.append((BINARY, hStr.encode("ascii"))
                        self.handshaked = True
                        self.handleConnected()
                    except Exception as e:
```

```
                except Exception as e:
                    hStr = FAILED_HANDSHAKE_STR
                    self._sendBuffer(hStr.encode("ascii"), True)
                    self.client.close()
                    raise Exception("handshake failed: %s", str(e))

        # else do normal data
        else:
            data = self.client.recv(16384)

            if not data:
                raise Exception("remote socket closed")

            for d in data:
                self._parseMessage(d)

    def close(self, status=1000, reason=""):
        """
        Send Close frame to the client. The underlying socket is only cl
        when the client acknowledges the Close frame.

        status is the closing identifier.
        reason is the reason for the close.
        """
```

```
        try:
            if self.closed is False:
                close_msg = bytearray()
                close_msg.extend(struct.pack("!H", status))
                if isinstance(reason, str):
                    close_msg.extend(reason.encode("utf-8"))
                else:
                    close_msg.extend(reason)


                self._sendMessage(False, CLOSE, close_msg)


        finally:
            self.closed = True


    def _sendBuffer(self, buff, send_all=False):
        size = len(buff)
        tosend = size
        already_sent = 0


        while tosend > 0:
            try:
                # i should be able to send a bytearray
                sent = self.client.send(buff[already_sent:])
```

```
                sent = self.client.send(buff[already_sent:])
                if sent == 0:
                    raise RuntimeError("socket connection broken")

                already_sent += sent
                tosend -= sent

            except socket.error as e:
                # if we have full buffers then wait for them to drain an
                if e.errno in [errno.EAGAIN, errno.EWOULDBLOCK]:
                    if send_all:
                        continue
                    return buff[already_sent:]
                else:
                    raise e

        return None

    def sendFragmentStart(self, data):
        """
        Send the start of a data fragment stream to a websocket client.
        Subsequent data should be sent using sendFragment().
        A fragment stream is completed when sendFragmentEnd() is called.
```

```
        If data is a unicode object then the frame is sent as Text.
        If the data is a bytearray object then the frame is sent as Bina
        """
        opcode = BINARY
        if isinstance(data, str):
            opcode = TEXT
        self._sendMessage(True, opcode, data)


    def sendFragment(self, data):
        """
        see sendFragmentStart()

        If data is a unicode object then the frame is sent as Text.
        If the data is a bytearray object then the frame is sent as Bina
        """
        self._sendMessage(True, STREAM, data)


    def sendFragmentEnd(self, data):
        """
        see sendFragmentEnd()

        If data is a unicode object then the frame is sent as Text.
```

```
        If data is a unicode object then the frame is sent as Text.
        If the data is a bytearray object then the frame is sent as Bina
        """
        self._sendMessage(False, STREAM, data)

    def sendMessage(self, data):
        """
        Send websocket data frame to the client.

        If data is a unicode object then the frame is sent as Text.
        If the data is a bytearray object then the frame is sent as Bina
        """
        opcode = BINARY
        if isinstance(data, str):
            opcode = TEXT
        self._sendMessage(False, opcode, data)

    def _sendMessage(self, fin, opcode, data):

        payload = bytearray()

        b1 = 0
        b2 = 0
```

```
        if fin is False:
            b1 |= 0x80
        b1 |= opcode

        if isinstance(data, str):
            data = data.encode("utf-8")

        length = len(data)

        payload.append(b1)

        if length <= 125:
            b2 |= length
            payload.append(b2)

        elif length >= 126 and length <= 65535:
            b2 |= 126
            payload.append(b2)
            payload.extend(struct.pack("!H", length))

        else:
            b2 |= 127
            payload.append(b2)
            payload.extend(struct.pack("!Q", length))
```

```
            payload.extend(struct.pack("!Q", length))

        if length > 0:
            payload.extend(data)

        self.sendq.append((opcode, payload))

    def _parseMessage(self, byte):
        # read in the header
        if self.state == HEADERB1:

            self.fin = byte & 0x80
            self.opcode = byte & 0x0F
            self.state = HEADERB2

            self.index = 0
            self.length = 0
            self.lengtharray = bytearray()
            self.data = bytearray()

            rsv = byte & 0x70
            if rsv != 0:
                raise Exception("RSV bit must be 0")
```

```
        elif self.state == HEADERB2:
            mask = byte & 0x80
            length = byte & 0x7F

            if self.opcode == PING and length > 125:
                raise Exception("ping packet is too large")

            if mask == 128:
                self.hasmask = True
            else:
                self.hasmask = False

            if length <= 125:
                self.length = length

                # if we have a mask we must read it
                if self.hasmask is True:
                    self.maskarray = bytearray()
                    self.state = MASK
                else:
                    # if there is no mask and no payload we are done
                    if self.length <= 0:
```

```
                if self.length <= 0:
                    try:
                        self._handlePacket()
                    finally:
                        self.state = HEADERB1
                        self.data = bytearray()

                # we have no mask and some payload
                else:
                    # self.index = 0
                    self.data = bytearray()
                    self.state = PAYLOAD

            elif length == 126:
                self.lengtharray = bytearray()
                self.state = LENGTHSHORT

            elif length == 127:
                self.lengtharray = bytearray()
                self.state = LENGTHLONG

        elif self.state == LENGTHSHORT:
            self.lengtharray.append(byte)
```

```
        if len(self.lengtharray) > 2:
            raise Exception("short length exceeded allowable size")

        if len(self.lengtharray) == 2:
            self.length = struct.unpack_from("!H", self.lengtharray)

            if self.hasmask is True:
                self.maskarray = bytearray()
                self.state = MASK
            else:
                # if there is no mask and no payload we are done
                if self.length <= 0:
                    try:
                        self._handlePacket()
                    finally:
                        self.state = HEADERB1
                        self.data = bytearray()

                # we have no mask and some payload
                else:
                    # self.index = 0
                    self.data = bytearray()
```

```
                    self.data = bytearray()
                    self.state = PAYLOAD

        elif self.state == LENGTHLONG:

            self.lengtharray.append(byte)

            if len(self.lengtharray) > 8:
                raise Exception("long length exceeded allowable size")

            if len(self.lengtharray) == 8:
                self.length = struct.unpack_from("!Q", self.lengtharray)

                if self.hasmask is True:
                    self.maskarray = bytearray()
                    self.state = MASK
                else:
                    # if there is no mask and no payload we are done
                    if self.length <= 0:
                        try:
                            self._handlePacket()
                        finally:
                            self.state = HEADERB1
```

```
                    self.data = bytearray()


                # we have no mask and some payload
                else:
                    # self.index = 0
                    self.data = bytearray()
                    self.state = PAYLOAD


# MASK STATE
elif self.state == MASK:
    self.maskarray.append(byte)

    if len(self.maskarray) > 4:
        raise Exception("mask exceeded allowable size")


    if len(self.maskarray) == 4:
        # if there is no mask and no payload we are done
        if self.length <= 0:
            try:
                self._handlePacket()
            finally:
                self.state = HEADERB1
                self.data = bytearray()
```

```
                    self.data = bytearray()

                # we have no mask and some payload
                else:
                    # self.index = 0
                    self.data = bytearray()
                    self.state = PAYLOAD

            # PAYLOAD STATE
            elif self.state == PAYLOAD:
                if self.hasmask is True:
                    self.data.append(byte ^ self.maskarray[self.index % 4])
                else:
                    self.data.append(byte)

                # if length exceeds allowable size then we except and remove
                if len(self.data) >= self.maxpayload:
                    raise Exception("payload exceeded allowable size")

                # check if we have processed length bytes; if so we are done
                if (self.index + 1) == self.length:
                    try:
                        self._handlePacket()
```

```
                finally:
                    # self.index = 0
                    self.state = HEADERB1
                    self.data = bytearray()
            else:
                self.index += 1



class WebSocketServer:
    def __init__(self, host, port, websocketclass, selectInterval=0.1):
        self.websocketclass = websocketclass

        if host == "":
            host = None

        if host is None:
            fam = socket.AF_INET6
        else:
            fam = 0

        hostInfo = socket.getaddrinfo(
            host, port, fam, socket.SOCK_STREAM, socket.IPPROTO_TCP, soc
        )
```

```
        )

        self.serversocket = socket.socket(
            hostInfo[0][0], hostInfo[0][1], hostInfo[0][2]
        )
        self.serversocket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEA
        self.serversocket.bind(hostInfo[0][4])
        self.serversocket.listen(5)
        self.selectInterval = selectInterval
        self.connections = {}

        self.listeners = [self.serversocket]


    def _decorateSocket(self, sock):
        return sock


    def _constructWebSocket(self, sock, address):
        return self.websocketclass(self, sock, address)


    def close(self):
        self.serversocket.close()

        for conn in self.connections.values():
            conn.close()
            self._handleClose(conn)
```

```
    def _handleClose(self, client):
        client.client.close()
        # only call handleClose when we have a successful websocket conn
        if client.handshaked:
            try:
                client.handleClose()
            except:

                pass


    def serveonce(self):
        writers = []
        for fileno in self.listeners:
            if fileno == self.serversocket:
                continue
            client = self.connections[fileno]
            if client.sendq:
                writers.append(fileno)

        rList, wList, xList = select(
            self.listeners, writers, self.listeners, self.selectInterval
        )
```

```
        for ready in wList:
            client = self.connections[ready]
            try:
                while client.sendq:
                    opcode, payload = client.sendq.popleft()
                    remaining = client._sendBuffer(payload)
                    if remaining is not None:
                        client.sendq.appendleft((opcode, remaining))

                        break
                    else:
                        if opcode == CLOSE:
                            raise Exception("received client close")

            except Exception as n:
                self._handleClose(client)
                del self.connections[ready]
                self.listeners.remove(ready)

        for ready in rList:
            if ready == self.serversocket:
                sock = None
                try:
```

```
                sock, address = self.serversocket.accept()
                newsock = self._decorateSocket(sock)
                newsock.setblocking(0)
                fileno = newsock.fileno()
                self.connections[fileno] = self._constructWebSocket(
                    newsock, address
                )
                self.listeners.append(fileno)

            except Exception as n:
                if sock is not None:
                    sock.close()
        else:
            if ready not in self.connections:
                continue
            client = self.connections[ready]
            try:
                client._handleData()
            except Exception as n:
                self._handleClose(client)
                del self.connections[ready]
                self.listeners.remove(ready)


    for failed in xList:
```

```
        for failed in xlist:
            if failed == self.serversocket:
                self.close()
                raise Exception("server socket failed")
            else:
                if failed not in self.connections:
                    continue
                client = self.connections[failed]
                self._handleClose(client)

                del self.connections[failed]
                self.listeners.remove(failed)


    def serveforever(self):
        while True:
            self.serveonce()



class SSLWebSocketServer(WebSocketServer):
    def __init__(
        self,
        host,
        port,
        websocketclass,
```

```
        certfile=None,
        keyfile=None,
        version=ssl.PROTOCOL_TLSv1,
        selectInterval=0.1,
        ssl_context=None,
    ):

        WebSocketServer.__init__(self, host, port, websocketclass, selec

        if ssl_context is None:
            self.context = ssl.SSLContext(version)
            self.context.load_cert_chain(certfile, keyfile)
        else:
            self.context = ssl_context

    def close(self):
        super(SSLWebSocketServer, self).close()

    def _decorateSocket(self, sock):
        sslsock = self.context.wrap_socket(sock, server_side=True)
        return sslsock

    def _constructWebSocket(self, sock, address):
```

```
    def _constructWebSocket(self, sock, address):
        ws = self.websocketclass(self, sock, address)
        ws.usingssl = True
        return ws

    def serveforever(self):
        super(SSLWebSocketServer, self).serveforever()



# ------------------------------------------


class Echo(WebSocket):
    def handleMessage(self):
        self.sendMessage(f"Echo -->{self.data}")



class Chat(WebSocket):
    clients: list["Chat"] = []

    def handleMessage(self):
        for client in Chat.clients:
            if client != self:
```

```
                client.sendMessage(self.address[0] + " - " + self.data)


    def handleConnected(self):
        print(self.address, "connected")
        for client in Chat.clients:
            client.sendMessage(self.address[0] + " - connected")
        Chat.clients.append(self)


    def handleClose(self):
        Chat.clients.remove(self)
        print(self.address, "closed")
        for client in Chat.clients:
            client.sendMessage(self.address[0] + " - disconnected")



def make_https():
    httpd = HTTPServer(("", 443), SimpleHTTPRequestHandler)
    httpd.socket = ssl.wrap_socket(
        httpd.socket,
        server_side=True,
        certfile="./cert.pem",
        keyfile="./key.pem",
        ssl_version=ssl.PROTOCOL_TLSv1
```

```
        ssl_version=ssl.PROTOCOL_TLSv1,
    )
    httpd.serve_forever()


if __name__ == "__main__":
    host = ""
    port = 8000
    use_ssl = False

    websocket_class = Echo
    # websocket_class = Chat
    keyfile = ""
    version = ssl.PROTOCOL_TLSv1

    if use_ssl:
        websocket_server = SSLWebSocketServer(
            host,
            port,
            websocket_class,
            keyfile=keyfile,
            version=version,
            selectInterval=0.1,
            ssl_context=None,
```

```
        )
    else:
        websocket_server = WebSocketServer(host, port, websocket_class)


    def close_sig_handler(signal, frame):
        websocket_server.close()
        sys.exit()


    signal.signal(signal.SIGINT, close_sig_handler)
    websocket_server.serveforever()
```