```
"""
COSC364 2022-S1 Assignment: RIP routing
Authors: MENG ZHANG (71682325), ZHENG CHAO (21671773)
File: rip_packet.py
"""
# RipPacket Class
class RipPacket:
    """
    A class for creating RIP update packet and provide methods to
    encode/decode outputing/incoming packets
    """
    # class attributes
    HEADER_LEN = 4
    ENTRY_LEN = 20


    def __init__(self, entries, router_id, command=2, version=2):
        """
        Parameters:
        entries: a list of rip entry objects
        router_id: the sender ID,  an integer between 1 and 64000
        (use the 16-bit wide all-zero field)
        command: an integer,
        i.e. 2 represents 'response'; 1 represents 'request'
        version: an integer, i.e. 1, 2(default)
        """
        # instance attributes
        self.command = command
        self.version = version
        self.router_id = router_id
        self.entries = entries


    @classmethod
    def decode_packet(cls, raw_packet):
        """
        Parameter:
        raw_packet: a packet of bytes
        i.e.
        HEADER:
        [command(1 byte), version(1), sender_id(2)]
        ENTRY:
        [afi(2 bytes), padding(2)
        dest(4)
        padding(4)
        padding(4)
        metric(4)]

        Return (True, RipPacket object) if raw_packet is valid,
        otherwise return (False, sender_id)
        """
        # Header: 4 bytes [0:4]
        command = raw_packet[0]
        version = raw_packet[1]
        sender_id = (raw_packet[2] << 8) + raw_packet[3]
        entries_num = int(len(raw_packet[4:]) / cls.ENTRY_LEN)
        # check header validity
        if not cls.is_valid_header(command, version,
                                   sender_id, entries_num):
            print("Broken packet:", "invalid header")
            return (False, sender_id)
        # Entries: n * 20 bytes [4:]
        # decode each entry
        entries = []
        for i in range(4, len(raw_packet), cls.ENTRY_LEN):
            raw_entry = raw_packet[i:i+cls.ENTRY_LEN]
```

```
            entry = RipEntry.decode_enty(raw_entry)
            # check entry validity
            # invalid entry is represented as None
            if entry is None:
                print("Broken packet:", "invalid entry")
                return (False, sender_id)
            entries.append(entry)
        rip_packet = RipPacket(entries, sender_id)
        return (True, rip_packet)


    def packet_bytes(self):
        """
        Return a rip packet in array of bytes
        """
        return self.header_bytes() + self.entries_bytes()


    def header_bytes(self):
        """
        Common header: 4 bytes in total
        [command(1 byte), version(1), sender_id(2)]

        command:2 (1 byte)
        version: 2 (1 byte)
        rouer_id: 1-64000 (2 bytes)

        Return a 4-byte rip header.
        """
        command_byte = self.command.to_bytes(1, byteorder='big')
        version_byte = self.version.to_bytes(1, byteorder='big')
        sender_id_bytes = self.router_id.to_bytes(2, byteorder='big')
        header = command_byte + version_byte + sender_id_bytes
        return header


    def entries_bytes(self):
        """
        Return a list of 20-byte rip entry
        """
        entries = bytes()
        for entry in self.entries:
            entries += entry.entry_bytes()
        return entries

    @classmethod
    def is_valid_header(self, command, version, router_id, entries_num):
        """
        check if a packet is valid
        """
        is_valid_command = command == 2
        if (not is_valid_command):
            print(f"invalid header command: {command}")
        is_valid_version = version == 2
        if (not is_valid_version):
            print(f"invalid header version: {version}")
        is_valid_id = 1 <= router_id <= 64000
        if (not is_valid_id):
            print(f"invalid header id: {router_id}")
        is_valid_entries_num = 1 <= entries_num <= 25
        if (not is_valid_entries_num):
            print(f"invalid header entries num: {entries_num}")
        return is_valid_command and\
               is_valid_version and\
               is_valid_id and\
               is_valid_entries_num
```

```python


# RipEntry Class
class RipEntry:
    """
    A class for creating entry objects in a rip packet
    """

    # class attributes
    PADDING_2BYTES = (0).to_bytes(2, byteorder='big')
    PADDING_4BYTES = (0).to_bytes(4, byteorder='big')


    def __init__(self, dest, metric, afi=2):
        """
        Parameters:
        dest: an integer, router_id of destination
        metric: an integer between 1 and 16 (inclusive)
        AFI: Address FAmily Identifier
        """
        self.dest = dest
        self.metric = metric
        self.afi = afi


    @classmethod
    def decode_enty(cls, raw_entry):
        """
        Parameter:
        raw_entry: an entry of bytes
        i.e.
        ENTRY:
        [afi(2 bytes), padding(2)
        dest(4)
        padding(4)
        padding(4)
        metric(4)]

        Return RipEntry object if raw_entry is valid,
        otherwise return None
        """
        # afi: 2 bytes [0:3]
        afi = (raw_entry[0] << 8) + raw_entry[1]
        # dest: 4 bytes but practically take 2 bytes [4:8]
        if (raw_entry[4] != 0 or
            raw_entry[5] != 0):
            print("Invalid dest of entry")
            return None
        dest = (raw_entry[6] << 8) + raw_entry[7]
        # metric 4 bytes but practically take 1 byte [16:]
        if (raw_entry[16] != 0 or
            raw_entry[17] != 0 or
            raw_entry[18] != 0):
            print("Invalid metric of entry")
            return None
        metric = raw_entry[19]
        entry = RipEntry(dest, metric, afi)
        if not entry.is_valid_entry():
            return None
        return entry


    def entry_bytes(self):
        """
        Rip entry: 20 bytes each
        [afi(2 bytes), padding(2)
        dest(4)
```

```
        padding(4)
        padding(4)
        metric(4)]

        afi: 2 (2 bytes)
        dest: 1-64000 (4 bytes)
        metric: 1-16 (4 bytes)
        padding: 0 (2 or 4 bytes)
        """
        afi_bytes = self.afi.to_bytes(2, byteorder='big')
        dest_bytes = self.dest.to_bytes(4, byteorder='big')
        metric_bytes = self.metric.to_bytes(4, byteorder='big')
        entry = afi_bytes + self.PADDING_2BYTES +\
                        dest_bytes +\
                        self.PADDING_4BYTES +\
                        self.PADDING_4BYTES +\
                        metric_bytes
        return entry

    def is_valid_entry(self):
        """
        check if an entry is valid
        """
        is_valid_dest = 1 <= self.dest <= 64000
        is_valid_metric = 0 <= self.metric <= 16
        is_valid_afi = self.afi == 2
        return is_valid_dest and is_valid_metric and is_valid_afi

    def set_metric_infinite(self):
        """
        set the metric to be infinite(16)
        """
        self.metric = 16

    def increment_metric(self):
        """
        add 1 to metric
        """
        self.metric += 1
```