CSE351: Computer Networks

Implementing a Server Agent for the Dynamic Host Configuration Protocol (DHCP)

Name	ID
Omar Ahmed Salah Ahmed	**2100790**
Youssef Mahmoud Hassan	**21P0130**
Kareem Ahmed Sameer	**21P0096**
Basel Ashraf Fikry	**22P0122**

Submitted to:
Dr. Ayman Bahaa
Dr. Karim Emara
Eng. Noha Wahdan

Contents

Abstract

This document details the design, implementation, and analysis of a DHCP server agent that is compliant with RFC 2131 and RFC 2132. The server is capable of handling standard DHCP messages such as DISCOVER, REQUEST, OFFER, ACK, and NAK, and is designed to be interoperable with common client devices. The server also supports lease management, IP address assignment, and error handling, making it a complete DHCP server solution.

1 Introduction

The goal of this project is to analyze and implement a server agent for DHCP protocols that is compliant with the RFC. The project should work seamlessly with off-the-shelf clients such as typical browsers or Windows DHCP/DNS client agents. The project is divided into phases, building on each other to ensure compliance with DHCP standards.

1.1 Dynamic Host Configuration Protocol (DHCP)

1.1.1 RFC 2131

Dynamic Host Configuration Protocol (DHCP) defines DHCP, a protocol used to automate the assignment of IP addresses, subnet masks, gateways, and other network parameters. It allows devices on a network to request configuration information from a DHCP server dynamically.

1.1.2 RFC 2132

DHCP Options and BOOTP Vendor Extensions. This RFC details additional options that can be used with DHCP, extending the base protocol to support various configuration parameters such as subnet masks, routers, and domain names.

2 Server Configuration

2.1 Imported Modules

```python
import socket
import threading
import sys
import json
import uuid
import time
```

Listing 1: Imported Modules

2.2 Server and DHCP Configuration Parameters

```python
# Server Configuration
HOST = '0.0.0.0'
PORT = 65432

# DHCP Configuration
IP_POOL_START = '192.168.1.100'
IP_POOL_END = '192.168.1.200'
LEASE_TIME = 20
OFFER_TIMEOUT = 300
```

Listing 2: Server and DHCP Configuration

3 Data Structures

```
authenticated_clients = {}
ip_leases = {}
ip_offers = {}
lock = threading.Lock()
```

Listing 3: Data Structures

4 JSON Configuration

```
{
    "server": {
        "host": "0.0.0.0",
        "port": 65432
    },
    "dhcp": {
        "ip_pool_start": "192.168.1.100",
        "ip_pool_end": "192.168.1.200",
        "lease_time": 20,
        "offer_timeout": 300
    }
}
```

5 Overview of DHCP Protocol

5.1 Key RFCs

- RFC 2131
- RFC 2132

6 Modular Design of the DHCP Server

6.1 DHCP Message Handlers

This section explains how each DHCP message handler processes client requests to manage IP assignments and network configuration.

6.1.1 Handle DHCP Discover

The server listens for DHCP Discover messages and provides an available IP address, forming the initial OFFER to the client.

```python
def handle_dhcp_discover(server_socket, packet, ui=None):
    """Handle DHCP DISCOVER messages."""
    client_mac = packet['mac']
    xid = packet['xid']
    logging.info(f"DHCP DISCOVER from {client_mac}")
    print(f"[DHCP] DISCOVER received from MAC: {client_mac}")

    cleanup_expired_offers()  # Cleanup any expired offers

    # Check existing lease
    if client_mac in leases:
        available_ip = leases[client_mac]['ip']
        logging.debug(f"Found existing lease for {client_mac}: {available_ip}")
    else:
        # Find new available IP
        available_ip = None
        requested_ip = get_requested_ip(packet)

        with LOCK:
            # First try to give requested IP if available
            if requested_ip and requested_ip in ip_pool and ip_pool[requested_ip]
 == "available":
                available_ip = requested_ip

            # If not, find first available IP
            if not available_ip:
                for ip, status in ip_pool.items():
                    if status == "available":
                        available_ip = ip
                        break

            if available_ip:
                ip_pool[available_ip] = "offered"
                offered_ips[client_mac] = {
                    'ip': available_ip,
                    'timestamp': time.time()
                }
                save_json_data('ip_pool.json', ip_pool)
                save_json_data('offered_ips.json', offered_ips)

    if available_ip:
        print(f"[DHCP] Offering IP {available_ip} to MAC: {client_mac}")
        response = create_dhcp_packet(2, xid, available_ip, client_mac, DHCP_OFFER)
        if response:
            try:
                server_socket.sendto(response, (calculate_broadcast_address(),
 config['CLIENT_PORT']))
                logging.info(f"Sent DHCP OFFER: {available_ip} to {client_mac}")
                if ui:
                    ui.refresh_data()
            except Exception as e:
                logging.error(f"Failed to send DHCP OFFER: {e}")
    else:
        logging.error(f"No available IP addresses for {client_mac}")
        print("[DHCP] No available IP addresses to offer")
```

6.1.2 Handle DHCP Request

When a client issues a DHCP request, the server validates the requested IP and either grants it via an ACK or denies it with a NAK.

```
def handle_dhcp_request(server_socket, packet, ui=None):
    """Handle DHCP REQUEST messages."""
    client_mac = packet['mac']
    xid = packet['xid']
    logging.info(f"DHCP REQUEST from {client_mac} with XID {xid}")
    print(f"[DHCP] REQUEST received from MAC: {client_mac} with XID {xid}")

    # Extract requested IP and server identifier
    requested_ip = get_requested_ip(packet)
    server_id = get_server_id(packet)

    logging.debug(f"Requested IP: {requested_ip}")
    logging.debug(f"Server Identifier: {server_id}")

    if not requested_ip or not server_id:
        logging.warning(f"Missing requested IP or server identifier in DHCP REQUEST from {cl
        send_dhcp_nak(server_socket, xid, client_mac)
        return

    # Validate server identifier matches the server's IP
    if server_id != config['SERVER_IP']:
        logging.warning(f"Server ID mismatch: Expected {config['SERVER_IP']}, got {server_id
        send_dhcp_nak(server_socket, xid, client_mac)
        return

    with LOCK:
        if client_mac in offered_ips:
            offered_ip = offered_ips[client_mac]['ip']
            logging.debug(f"Offered IP for {client_mac}: {offered_ip}")
            if requested_ip == offered_ip and ip_pool.get(requested_ip) == "offered":
                # Valid request
                send_dhcp_ack(server_socket, xid, client_mac, requested_ip)
                return
            else:
                logging.warning(f"Requested IP {requested_ip} does not match offered IP {off
        else:
            logging.warning(f"No offered IP found for MAC {client_mac}")
    # If validation fails, send DHCP NAK
    send_dhcp_nak(server_socket, xid, client_mac)
```

6.1.3 Handle DHCP Release

Once a client releases an IP, the server marks it as available, allowing new clients to be assigned this address.

```
def handle_dhcp_release(packet, ui=None):
    """Handle DHCP RELEASE messages."""
    client_mac = packet['mac']
    logging.info(f"DHCP RELEASE from {client_mac}")
    print(f"[DHCP] RELEASE received from MAC: {client_mac}")

    with LOCK:
        if client_mac in leases:
            ip = leases[client_mac]['ip']
            del leases[client_mac]
            ip_pool[ip] = "available"
            save_json_data('ip_pool.json', ip_pool)
            save_json_data('lease_database.json', leases)
            logging.info(f"Released IP {ip} from {client_mac}")
            print(f"[DHCP] IP {ip} released from MAC: {client_mac}")
            if ui:
                ui.refresh_data()
```

6.1.4 Handle DHCP Inform

The DHCP Inform message requests configuration parameters without leasing an IP. The server responds with these details in an ACK.

```
def handle_dhcp_inform(server_socket, packet, ui=None):
    """Handle DHCP INFORM messages."""
    client_mac = packet['mac']
    xid = packet['xid']
    logging.info(f"DHCP INFORM from {client_mac}")
    print(f"[DHCP] INFORM received from MAC: {client_mac}")

    # Send ACK with configuration information (but no IP assignment)
    response = create_dhcp_packet(2, xid, '0.0.0.0', client_mac, DHCP_ACK)
    if response:
        try:
            server_socket.sendto(response, (calculate_broadcast_address(), config['
   CLIENT_PORT']))
            logging.info(f"Sent DHCP ACK (INFORM) to {client_mac}")
            print(f"[DHCP] INFORM ACK sent to MAC: {client_mac}")
            if ui:
                ui.refresh_data()
        except Exception as e:
            logging.error(f"Failed to send DHCP ACK (INFORM): {e}")
```

7 Implementation Details

7.1 Socket Communication

UDP sockets are used to receive incoming DHCP packets and send responses to clients, relying on broadcast and unicast communication.

```
def start_server(server_ui=None):
    """Start the DHCP server."""
    # Load saved data and configuration
    load_config()
    load_json_data()

    # Start lease manager thread
    lease_thread = threading.Thread(target=lease_manager, args=(server_ui,), daemon
=True)
    lease_thread.start()
    logging.info("Lease manager thread started")

    # Create and configure server socket
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
    server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

    try:
        # Binding to '' (all interfaces) on port SERVER_PORT
        # ensures we can receive broadcasts properly.
        server_socket.bind(('', config['SERVER_PORT']))
        logging.info(f"DHCP Server started on {config['SERVER_IP']}:{config['
SERVER_PORT']}")
        print(f"[DHCP] Server started on {config['SERVER_IP']}:{config['SERVER_PORT
']}")
        if server_ui:
            server_ui.update_status("Server running", "green")
    except Exception as e:
        logging.error(f"Failed to bind to port {config['SERVER_PORT']}: {e}")
        if server_ui:
            server_ui.update_status(f"Failed to start server: {e}", "red")
        return

    while True:
        try:
            data, addr = server_socket.recvfrom(2048)
            packet = parse_dhcp_packet(data)

            if not packet:
                continue

            if not validate_packet(packet):
                logging.warning("Received invalid DHCP packet")
                continue

            # Handle different DHCP message types (RFC2131)
```

```python
            if 53 in packet['options']:
                msg_type = packet['options'][53][0]
                if msg_type == 1:  # DHCP_DISCOVER
                    handle_dhcp_discover(server_socket, packet, server_ui)
                elif msg_type == 3:  # DHCP_REQUEST
                    handle_dhcp_request(server_socket, packet, server_ui)
                elif msg_type == 7:  # DHCP_RELEASE
                    handle_dhcp_release(packet, server_ui)
                elif msg_type == 8:  # DHCP_INFORM
                    handle_dhcp_inform(server_socket, packet, server_ui)
                elif msg_type == 4:  # DHCP_DECLINE
                    # Minimal handling of DECLINE: usually means client found IP conflict
                    logging.warning(f"DHCP DECLINE from {packet['mac']} - IP conflict suspected")
                    print(f"[DHCP] DECLINE from MAC: {packet['mac']} - possible IP conflict")
                else:
                    logging.warning(f"Unhandled DHCP message type: {msg_type}")

        except Exception as e:
            logging.error(f"Error in main server loop: {e}")
            if server_ui:
                server_ui.update_status(f"Error: {e}", "red")
```

7.2 Lease Management

Leases are tracked using data structures that record each client's IP assignment and lease expiration, ensuring timely checks for availability.

```python
def lease_manager(ui=None):
    """Background thread to manage lease expiration."""
    while True:
        try:
            time.sleep(60)  # Check every minute
            current_time = datetime.now()

            with LOCK:
                expired_leases = [
                    mac for mac, lease in leases.items()
                    if datetime.fromisoformat(lease['lease_expiration']) < current_time
                ]

                for mac in expired_leases:
                    ip = leases[mac]['ip']
                    del leases[mac]
                    ip_pool[ip] = "available"
                    logging.info(f"Lease expired for {mac}, IP {ip} released")
                    print(f"[DHCP] Lease expired for MAC: {mac}, IP {ip} is now available")

                if expired_leases:
            save_json_data('ip_pool.json', ip_pool)
            save_json_data('lease_database.json', leases)
            if ui:
                ui.refresh_data()

except Exception as e:
            logging.error(f"Error in lease manager: {e}")
```

7.3 IP Address Pool

A pool of IP addresses is maintained to supply new leases. Addresses transition between "available," "offered," and "in_use" states.

```
def get_requested_ip(packet):
    """Extract the requested IP address from DHCP options."""
    options = packet.get('options', {})
    if OPTION_REQUESTED_IP in options:
        return socket.inet_ntoa(options[OPTION_REQUESTED_IP])
    return None

def get_server_id(packet):
    """Extract the server identifier from DHCP options."""
    options = packet.get('options', {})
    if OPTION_SERVER_IDENTIFIER in options:
        return socket.inet_ntoa(options[OPTION_SERVER_IDENTIFIER])
    return None

def send_dhcp_ack(server_socket, xid, mac, ip):
    """Send DHCPACK to the client."""
    response = create_dhcp_packet(BOOTREPLY, xid, ip, mac, DHCP_ACK)
    if response:
        try:
            server_socket.sendto(response, (calculate_broadcast_address(), config['
    CLIENT_PORT']))
            logging.info(f"Sent DHCP ACK: {ip} to {mac}")
            # Update lease and IP pool
            ip_pool[ip] = "in_use"
            leases[mac] = {
                "ip": ip,
                "lease_expiration": (datetime.now() + timedelta(seconds=config['
    LEASE_TIME'])).isoformat()
            }
            del offered_ips[mac]
            save_json_data('ip_pool.json', ip_pool)
            save_json_data('lease_database.json', leases)
            save_json_data('offered_ips.json', offered_ips)
            if ui:
                ui.refresh_data()
        except Exception as e:
            logging.error(f"Failed to send DHCP ACK: {e}")

def send_dhcp_nak(server_socket, xid, mac):
    """Send DHCPNAK to the client."""
    response = create_dhcp_packet(BOOTREPLY, xid, '0.0.0.0', mac, DHCP_NAK)
    if response:
        try:
            server_socket.sendto(response, (calculate_broadcast_address(), config['CLIENT_PORT']))
            logging.info(f"Sent DHCP NAK to {mac}")
        except Exception as e:
            logging.error(f"Failed to send DHCP NAK: {e}")
```

7.4 Background Lease Management Thread

The lease manager thread periodically scans the lease database, releasing expired leases and updating the stored data accordingly.

```
def lease_manager(ui=None):
    """Background thread to manage lease expiration."""
    while True:
        try:
            time.sleep(60)  # Check every minute
            current_time = datetime.now()

            with LOCK:
                expired_leases = [
                    mac for mac, lease in leases.items()
                    if datetime.fromisoformat(lease['lease_expiration']) < current_time
                ]

                for mac in expired_leases:
                    ip = leases[mac]['ip']
                    del leases[mac]
                    ip_pool[ip] = "available"
                    logging.info(f"Lease expired for {mac}, IP {ip} released")
                    print(f"[DHCP] Lease expired for MAC: {mac}, IP {ip} is now available")

                if expired_leases:
                    save_json_data('ip_pool.json', ip_pool)
                    save_json_data('lease_database.json', leases)
                    if ui:
                        ui.refresh_data()

        except Exception as e:
            logging.error(f"Error in lease manager: {e}")
```

8 Code Walkthrough

```
def handle_dhcp_discover(server_socket, packet, ui=None):
    client_mac = packet['mac']
    xid = packet['xid']
    cleanup_expired_offers()
    available_ip = find_available_ip(client_mac, packet)
    if available_ip:
        response = create_dhcp_packet(2, xid, available_ip, client_mac, DHCP_OFFER)
        server_socket.sendto(response, (broadcast_address, config['CLIENT_PORT']))
```

9 Extended DHCP Implementation

This section provides additional code snippets and constants for a more comprehensive DHCP implementation.

9.1 DHCP Message and BOOTP Constants

```
# ------------------ DHCP MESSAGE TYPE CONSTANTS ------------------ #
DHCP_DISCOVER = 1
DHCP_OFFER = 2
DHCP_REQUEST = 3
DHCP_DECLINE = 4
DHCP_ACK = 5
DHCP_NAK = 6
DHCP_RELEASE = 7
DHCP_INFORM = 8

# ------------------ BOOTP MESSAGE TYPE CONSTANTS ------------------ #
BOOTREQUEST = 1
BOOTREPLY = 2
```

Listing 4: DHCP and BOOTP Message Type Constants

9.2 DHCP Options Constants

```
OPTION_SUBNET_MASK = 1
OPTION_ROUTER = 3
OPTION_DNS_SERVER = 6
OPTION_DOMAIN_NAME = 15
OPTION_REQUESTED_IP = 50
OPTION_MESSAGE_TYPE = 53
OPTION_SERVER_IDENTIFIER = 54
OPTION_PARAMETER_REQUEST_LIST = 55
OPTION_LEASE_TIME = 51
OPTION_RENEWAL_TIME = 58
OPTION_REBINDING_TIME = 59
OPTION_VENDOR_CLASS_IDENTIFIER = 60
OPTION_END = 255
```

Listing 5: DHCP Option Codes

9.3 Server Default Configuration

```
DEFAULT_CONFIG = {
    "SERVER_IP": "192.168.1.1",
    "SERVER_PORT": 67,
    "CLIENT_PORT": 68,
    "NETWORK_MASK": "255.255.255.0",
    "DEFAULT_GATEWAY": "192.168.1.1",
    "DNS_SERVER": "8.8.8.8",
    "ADDITIONAL_DNS_SERVERS": ["8.8.4.4"],
    "TIME_SERVER": "192.168.1.2",
    "NAME_SERVER": "192.168.1.3",
    "DOMAIN_NAME": "example.com",
    "VENDOR_ID": "PythonDHCP",
    "LEASE_TIME": 86400,
    "RENEWAL_TIME": 43200,
    "REBINDING_TIME": 75600
}
```

Listing 6: DHCP Server Default Configuration

9.4 Implementation Snippet: Globals and Logging

```
# ------------------ GLOBALS ------------------ #
leases = {}
ip_pool = {}
offered_ips = {}
config = {}

CONFIG_FILE = 'dhcp_config.json'

# ------------------ LOGGING CONFIGURATION ------------------ #
import logging
logging.basicConfig(
    level=logging.INFO,
    format='[%(asctime)s] [%(levelname)s] %(message)s',
    handlers=[
        logging.FileHandler("dhcp_server.log"),
        logging.StreamHandler(sys.stdout)
    ]
)

LOCK = threading.RLock()
```

Listing 7: Global Variables and Logging

9.5 Implementation Snippet: Configuration Handling

```
def load_config():
    """Load server configuration from JSON file or initialize with defaults."""
    global config
    try:
        with open(CONFIG_FILE, "r") as file:
            loaded_config = json.load(file)
            logging.info("Configuration loaded.")

            config = DEFAULT_CONFIG.copy()
            config.update(loaded_config)

            # Identify missing keys
            missing_keys = [k for k in DEFAULT_CONFIG if k not in loaded_config]
            for key in missing_keys:
                logging.warning(f"'{key}' not found in config. Using default: {
    DEFAULT_CONFIG[key]}")

            save_config()
    except FileNotFoundError:
        config = DEFAULT_CONFIG.copy()
        save_config()
        logging.info("Default configuration initialized.")
    except json.JSONDecodeError:
        logging.error("Error decoding configuration file. Using default settings.")
        config = DEFAULT_CONFIG.copy()
        save_config()

def save_config():
    """Save server configuration to JSON file."""
    with LOCK:
        try:
            with open(CONFIG_FILE, 'w') as file:
                json.dump(config, file, indent=4)
                logging.info("Configuration saved.")
        except Exception as e:
            logging.error(f"Failed to save configuration: {e}")
```

Listing 8: Loading and Saving Configuration

9.6 Implementation Snippet: JSON Data Loading and Saving

```
def load_json_data():
    """Load data from JSON files with error handling."""
    global ip_pool, leases, offered_ips

    def load_file(filename, default_value):
        try:
            with open(filename, "r") as file:
                data = json.load(file)
                logging.info(f"Loaded {filename}")
                return data
        except FileNotFoundError:
            logging.info(f"{filename} not found. Initializing with default data.")
            save_json_data(filename, default_value)
            return default_value
        except json.JSONDecodeError:
            logging.error(f"Error decoding {filename}. Using default data.")
            return default_value

    default_pool = {f"192.168.1.{i}": "available" for i in range(100, 200)}
    ip_pool = load_file('ip_pool.json', default_pool)
    leases = load_file('lease_database.json', {})
    offered_ips = load_file('offered_ips.json', {})

def save_json_data(file_path, data):
    """Save data to JSON file with error handling."""
    with LOCK:
        try:
            with open(file_path, 'w') as file:
                json.dump(data, file, indent=4)
                logging.info(f"Saved data to {file_path}")
        except Exception as e:
            logging.error(f"Failed to save {file_path}: {e}")
```

Listing 9: JSON Data Handling

9.7 Implementation Snippet: IP and MAC Utilities

```
import ipaddress

def ip_to_int(ip_str):
    """Convert IP address string to integer."""
    return int(ipaddress.IPv4Address(ip_str))

def int_to_ip(ip_int):
    """Convert integer to IP address string."""
    return str(ipaddress.IPv4Address(ip_int))

def get_mac_str(data, offset=28):
    """Extract MAC address from packet data."""
    if len(data) < offset + 6:
        logging.debug("Not enough data for MAC address.")
        return '00:00:00:00:00:00'
    return ":".join(["{:02x}".format(x) for x in data[offset:offset+6]])
```

Listing 10: IP and MAC Processing

9.8 Implementation Snippet: Network Helpers

```
def calculate_broadcast_address():
    """Calculate broadcast address based on server configuration."""
    try:
        network = ipaddress.IPv4Network(f"{config['SERVER_IP']}/{config['
    NETWORK_MASK']}", strict=False)
        return str(network.broadcast_address)
    except ValueError as e:
        logging.error(f"Error calculating broadcast address: {e}")
        return '255.255.255.255'

def is_ip_in_range(ip):
    """Check if IP is within the configured network range."""
    try:
        network = ipaddress.IPv4Network(f"{config['SERVER_IP']}/{config['
    NETWORK_MASK']}", strict=False)
        return ipaddress.IPv4Address(ip) in network
    except ValueError:
        return False
```

Listing 11: Network Utility Functions

9.9 Implementation Snippet: Offer Cleanup and Validation

```
def cleanup_expired_offers():
    """Remove expired offers from offered_ips."""
    with LOCK:
        current_time = time.time()
        expired = [
            mac for mac in offered_ips
            if offered_ips[mac].get('timestamp', 0) + 60 < current_time
        ]
        for mac in expired:
            ip = offered_ips[mac].get('ip')
            if ip and ip in ip_pool:
                ip_pool[ip] = 'available'
            del offered_ips[mac]
        if expired:
            save_json_data('ip_pool.json', ip_pool)
            save_json_data('offered_ips.json', offered_ips)

def validate_packet(packet):
    """Validate DHCP packet structure and contents."""
    required_fields = ['op', 'htype', 'hlen', 'xid', 'mac']
    return all(field in packet for field in required_fields)
```

Listing 12: Offer Cleanup and Packet Validation

9.10 Implementation Snippet: DHCP Option Helpers

```
def get_option_name(option_code):
    """Convert DHCP option code to readable name."""
    options = {
        1: "Subnet Mask",
        3: "Router",
        4: "Time Server",
        6: "DNS Servers",
        15: "Domain Name",
        42: "NTP Servers",
        44: "NetBIOS Name Server",
        46: "NetBIOS Node Type",
        51: "Lease Time",
        53: "DHCP Message Type",
        54: "DHCP Server Identifier",
        58: "Renewal Time (T1)",
        59: "Rebinding Time (T2)",
        60: "Vendor Class Identifier",
    }
    return options.get(option_code, f"Option {option_code}")

def format_options_debug(options):
    """Format DHCP options for debug logging."""
    return ", ".join(f"{get_option_name(k)}={v}" for k, v in options.items())
```

Listing 13: DHCP Option Handling

9.11 Implementation Snippet: Parsing and Server Startup

```
import struct

def parse_dhcp_packet(data):
    """Parse a DHCP packet with error handling and options parsing."""
    if len(data) < 240:
        logging.debug("Packet too short to be valid.")
        return None

    packet = {}
    try:
        (
            packet['op'],
            packet['htype'],
            packet['hlen'],
            packet['hops'],
            packet['xid'],
            packet['secs'],
            packet['flags'],
            packet['ciaddr'],
            packet['yiaddr'],
            packet['siaddr'],
            packet['giaddr']
        ) = struct.unpack('!BBBBIHHIIII', data[:28])

        packet['chaddr'] = data[28:28+16]
        packet['mac'] = get_mac_str(data, 28)

        packet['sname'] = data[44:108]
        packet['file'] = data[108:236]

        magic_cookie = data[236:240]
        if magic_cookie != b'\x63\x82\x53\x63':
            logging.debug("Invalid DHCP magic cookie.")
            return None

        packet['options'] = {}
        options_data = data[240:]
        idx = 0

        while idx < len(options_data):
            opt_type = options_data[idx]
            if opt_type == 255:  # END
                break
            if opt_type == 0:    # PAD
                idx += 1
                continue
            if idx + 1 >= len(options_data):
                logging.debug("Truncated option field.")
                break
            opt_len = options_data[idx + 1]
            if idx + 2 + opt_len > len(options_data):
                logging.debug(f"Option {opt_type} truncated.")
                break
            option_value = options_data[idx + 2: idx + 2 + opt_len]
            packet['options'][opt_type] = option_value
            idx += 2 + opt_len

    except Exception as e:
        logging.error(f"Error parsing DHCP packet: {e}")
        return None

    return packet

def start_dhcp_server():
    """Main function to start the DHCP server."""
    load_config()
    load_json_data()
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    server_socket.bind((config['SERVER_IP'], config['SERVER_PORT']))
    logging.info(f"DHCP Server started on {config['SERVER_IP']}:{config['
   SERVER_PORT']}")

    while True:
        data, addr = server_socket.recvfrom(1024)
        packet = parse_dhcp_packet(data)
        if packet:
            handle_dhcp_packet(server_socket, packet, addr)
```

Listing 14: Packet Parsing and Server Startup