Project 2 - TCP/IP Programming

Regina Sanchez

Oregon State University

CS372 - 400

Instructor Lewis

May 10th, 2024

Commands To Run Program

Commands

Hello! To start my program you have to first begin by having two terminals. In one terminal, enter “sudo python3 monitoring_service.py” and in the second terminal run “sudo python3 management_app.py”.

While the program is running, in the monitoring_service file, you can enter the commands of list_clients, list_all_queues, etc to see further information. Images of these outputs as well as images of the code will be displayed in the section below.

In the monitoring server, in the CLI you can enter list_clients or list_all_queues to get more information regarding the connections available.

To exit, enter control c in both files.

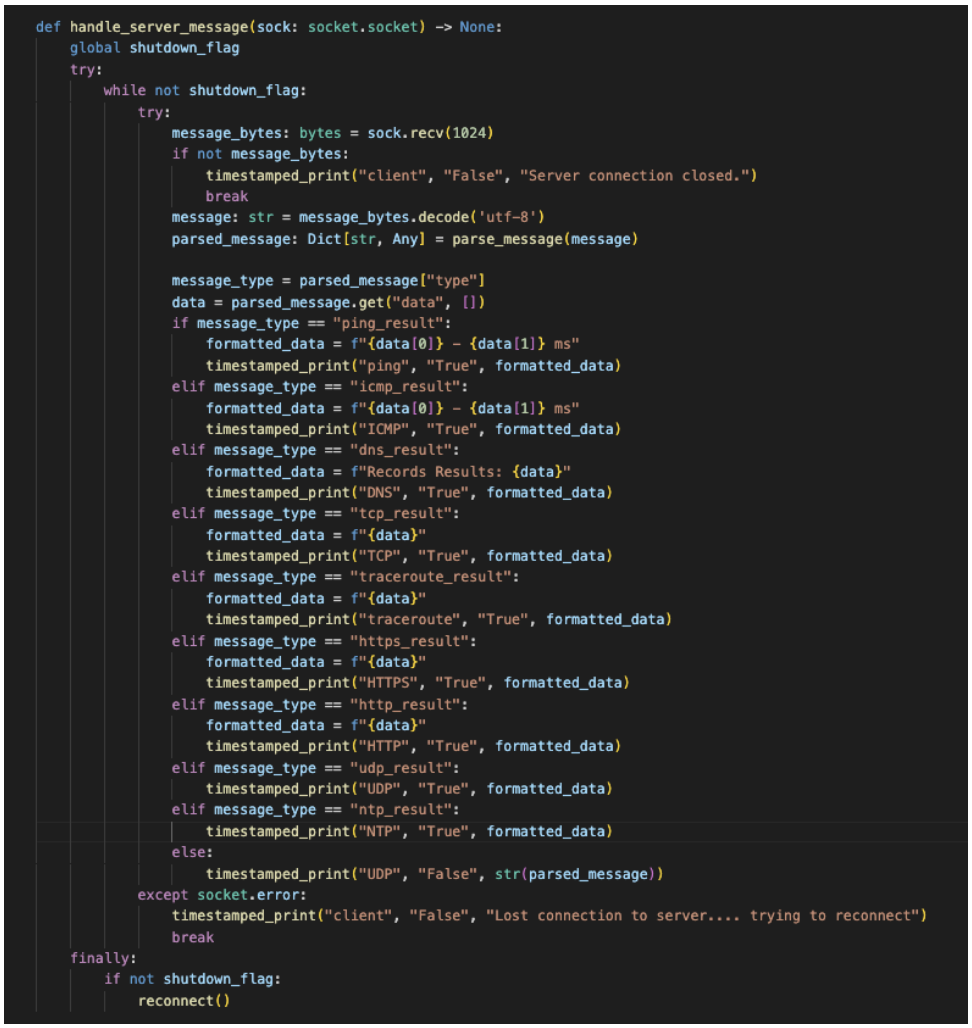
Task Management

For my tasks, I configured it so that the client file has a json file with all of the configurations and services wanted. From there, it sends the information to the server, also saying that it is connected to the server to send information. It will then state what service is being sent to the server. The server then prints all of the information received from the client in the terminal, as well as prints what port it is connected to for that management + monitor relationship. So overall, client is able to tell server what it wants, the server retrieves and sends the information back to the client, and the client then prints it.

```
ninasanchez@Ninas-MacBook-Pro asm2 % sudo python3 management_app.py
Password:
[2024-05-16 22:24:53]  [client] [True] [Connected to server]
[2024-05-16 22:24:53]  [client] [True] [sending information to server.. [ping]]
[2024-05-16 22:24:54]  [ping] [True] [['142.250.72.164', 0] - 71.30122184753418 ms]
[2024-05-16 22:25:03]  [client] [True] [sending information to server.. [TCP]]
[2024-05-16 22:25:04]  [TCP] [True] [Connected successfully]
[2024-05-16 22:25:07]  [client] [True] [sending information to server.. [HTTPS]]
[2024-05-16 22:25:08]  [HTTPS] [True] [{'is_up': True, 'status_code': 200, 'description': 'Server is up'}]
[2024-05-16 22:25:13]  [client] [True] [sending information to server.. [HTTP]]
[2024-05-16 22:25:14]  [HTTP] [True] [{'is_up': True, 'status_code': 200, 'description': 'Success'}]
[2024-05-16 22:25:23]  [client] [True] [sending information to server.. [ping]]
[2024-05-16 22:25:24]  [ping] [True] [['142.250.72.164', 0] - 74.41878318786621 ms]
[2024-05-16 22:25:33]  [client] [True] [sending information to server.. [UDP]]
[2024-05-16 22:25:36]  [UDP] [True] [['142.250.72.164', 0] - 74.41878318786621 ms]
[2024-05-16 22:25:40]  [client] [True] [sending information to server.. [DNS]]
[2024-05-16 22:25:46]  [DNS] [True] [Records Results: {'status': False}]
[2024-05-16 22:25:49]  [client] [True] [sending information to server.. [NTP]]
[2024-05-16 22:25:55]  [NTP] [True] [Records Results: {'status': False}]
[2024-05-16 22:26:01]  [client] [True] [sending information to server.. [ping]]
[2024-05-16 22:26:02]  [ping] [True] [['23.192.212.128', 0] - 50.01688003540039 ms]
[2024-05-16 22:26:13]  [client] [True] [sending information to server.. [ping]]
```

```
ninasanchez@Ninas-MacBook-Pro asm2 % sudo python3 monitoring_service.py
Password:
[Server] Status server listening on localhost:54321
[Server] accepted connection from ('127.0.0.1', 64386)
[2024-05-16 22:24:53]  [Client] Received message from server: {"type": "execute", "data": {"name": "google", "host": "www.google.com", "port": 80, "service": "ping", "interval": 10}, "count": 10}
[2024-05-16 22:25:03]  [Client] Received message from server: {"type": "execute", "data": {"name": "google", "host": "www.google.com", "port": 80, "service": "TCP", "interval": 4}, "count": 4}
[2024-05-16 22:25:07]  [Client] Received message from server: {"type": "execute", "data": {"name": "yahoo", "host": "www.yahoo.com", "port": 80, "service": "HTTPS", "interval": 6}, "count": 6}
[2024-05-16 22:25:13]  [Client] Received message from server: {"type": "execute", "data": {"name": "google", "host": "www.google.com", "port": 80, "service": "HTTP", "interval": 10}, "count": 10}
[2024-05-16 22:25:23]  [Client] Received message from server: {"type": "execute", "data": {"name": "google", "host": "www.google.com", "port": 80, "service": "ping", "interval": 10}, "count": 10}
[2024-05-16 22:25:33]  [Client] Received message from server: {"type": "execute", "data": {"name": "8.8.8.8", "host": "google", "port": 53, "service": "UDP", "interval": 7}, "count": 7}
[2024-05-16 22:25:40]  [Client] Received message from server: {"type": "execute", "data": {"name": "8.8.8.8", "host": "google", "port": 53, "service": "DNS", "interval": 9}, "count": 9}
[2024-05-16 22:25:49]  [Client] Received message from server: {"type": "execute", "data": {"name": "pool.ntp.org", "host": "pool.ntp.org", "port": 53, "service": "NTP", "interval": 12}, "count": 12}
[2024-05-16 22:26:01]  [Client] Received message from server: {"type": "execute", "data": {"name": "walmart", "host": "www.walmart.com", "port": 80, "service": "ping", "interval": 11}, "count": 11}
[2024-05-16 22:26:13]  [Client] Received message from server: {"type": "execute", "data": {"name": "google", "host": "www.google.com", "port": 80, "service": "ping", "interval": 10}, "count": 10}
[2024-05-16 22:26:23]  [Client] Received message from server: {"type": "execute", "data": {"name": "google", "host": "www.google.com", "port": 80, "service": "TCP", "interval": 4}, "count": 4}
[2024-05-16 22:26:27]  [Client] Received message from server: {"type": "execute", "data": {"name": "yahoo", "host": "www.yahoo.com", "port": 80, "service": "HTTPS", "interval": 6}, "count": 6}
[2024-05-16 22:26:33]  [Client] Received message from server: {"type": "execute", "data": {"name": "google", "host": "www.google.com", "port": 80, "service": "HTTP", "interval": 10}, "count": 10}
[2024-05-16 22:26:43]  [Client] Received message from server: {"type": "execute", "data": {"name": "google", "host": "www.google.com", "port": 80, "service": "ping", "interval": 10}, "count": 10}
[2024-05-16 22:26:53]  [Client] Received message from server: {"type": "execute", "data": {"name": "8.8.8.8", "host": "google", "port": 53, "service": "UDP", "interval": 7}, "count": 7}
[2024-05-16 22:27:00]  [Client] Received message from server: {"type": "execute", "data": {"name": "8.8.8.8", "host": "google", "port": 53, "service": "DNS", "interval": 9}, "count": 9}
[2024-05-16 22:27:09]  [Client] Received message from server: {"type": "execute", "data": {"name": "pool.ntp.org", "host": "pool.ntp.org", "port": 53, "service": "NTP", "interval": 12}, "count": 12}
[2024-05-16 22:27:21]  [Client] Received message from server: {"type": "execute", "data": {"name": "walmart", "host": "www.walmart.com", "port": 80, "service": "ping", "interval": 11}, "count": 11}
```

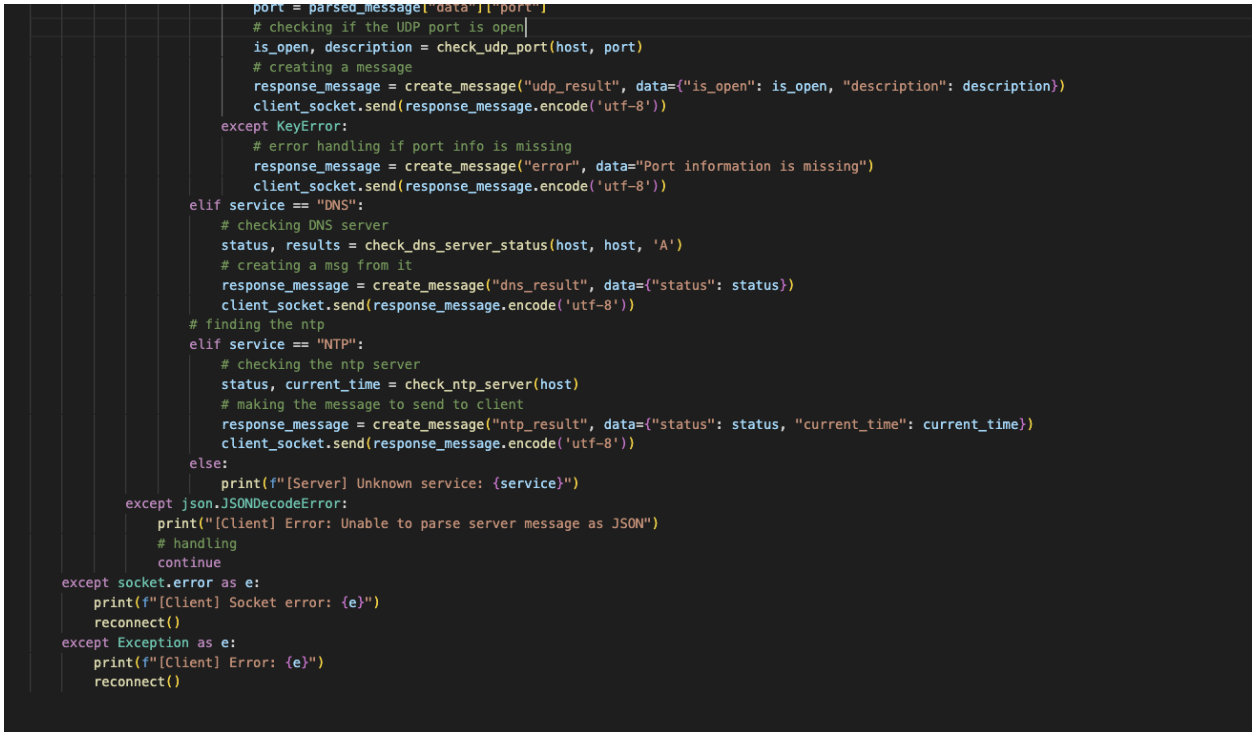
In this image, I am handling the received information from the server file. It essentially is able to firstly find what information is being sent back to the client, using a type to find the service's results. Then it prints the information to the user.



```python
def handle_server_message(sock: socket.socket) -> None:
    global shutdown_flag
    try:
        while not shutdown_flag:
            try:
                message_bytes: bytes = sock.recv(1024)
                if not message_bytes:
                    timestamped_print("client", "False", "Server connection closed.")
                    break
                message: str = message_bytes.decode('utf-8')
                parsed_message: Dict[str, Any] = parse_message(message)

                message_type = parsed_message["type"]
                data = parsed_message.get("data", [])
                if message_type == "ping_result":
                    formatted_data = f"{data[0]} - {data[1]} ms"
                    timestamped_print("ping", "True", formatted_data)
                elif message_type == "icmp_result":
                    formatted_data = f"{data[0]} - {data[1]} ms"
                    timestamped_print("ICMP", "True", formatted_data)
                elif message_type == "dns_result":
                    formatted_data = f"Records Results: {data}"
                    timestamped_print("DNS", "True", formatted_data)
                elif message_type == "tcp_result":
                    formatted_data = f"{data}"
                    timestamped_print("TCP", "True", formatted_data)
                elif message_type == "traceroute_result":
                    formatted_data = f"{data}"
                    timestamped_print("traceroute", "True", formatted_data)
                elif message_type == "https_result":
                    formatted_data = f"{data}"
                    timestamped_print("HTTPS", "True", formatted_data)
                elif message_type == "http_result":
                    formatted_data = f"{data}"
                    timestamped_print("HTTP", "True", formatted_data)
                elif message_type == "udp_result":
                    timestamped_print("UDP", "True", formatted_data)
                elif message_type == "ntp_result":
                    timestamped_print("NTP", "True", formatted_data)
                else:
                    timestamped_print("UDP", "False", str(parsed_message))
            except socket.error:
                timestamped_print("client", "False", "Lost connection to server.... trying to reconnect")
                break
    finally:
        if not shutdown_flag:
            reconnect()
```

In this image, I am handling the information recieved from the client. It is able to parse the messages, then execute the service wanted, and sends the message back to the client. The image is able to reflect the tasks being completed.

```
def handle_client_connection(client_socket: socket.socket, addr: str) -> None:
    global shutdown_flag
    try:
        while not shutdown_flag:
            message_bytes = client_socket.recv(1024)
            if not message_bytes:
                # empty message received --> continue listening
                continue
            message = message_bytes.decode('utf-8')
            timestamped_print("[Client] Received message from server:", message)
            try:
                parsed_message = json.loads(message)
                # process parsed_message as needed
                if parsed_message["type"] == "execute": # this is how they are read in and filtered
                    service = parsed_message["data"]["service"]
                    if service == "ping" or service == "ICMP":
                        # perform ping using the  ping function
                        host = parsed_message["data"]["host"]
                        addr, ping_time = ping(host)
                        results = ping(host)
                        if addr and ping_time is not None:
                            # response_message = create_to_send(results)
                            # response_message = create_message("ping_result", data=f"Address: {addr}, Ping Time: {ping_time} ms")
                            response_message = create_message("ping_result", data=results)
                        else:
                            response_message = create_message("ping_result", data="Ping request timed out.")
                        client_socket.send(response_message.encode('utf-8'))
                    # getting the TCP results
                    elif service == "TCP":
                        # performing the TCP action
                        try:
                            tcp_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                            tcp_socket.settimeout(5)
                            tcp_socket.connect((parsed_message["data"]["host"], parsed_message["data"]["port"]))
                            response_message = create_message("tcp_result", data="Connected successfully")
                        except socket.error as e:
                            response_message = create_message("tcp_result", data=str(e))
                        finally:
                            tcp_socket.close()
                        client_socket.send(response_message.encode('utf-8'))
                    # doing the traceroute
                    elif service == "traceroute":
                        print("traceroute")
                        traceroute_ans = traceroute(host)
                        response_message = create_message("traceroute_result", data=traceroute_ans)
                        client_socket.send(response_message.encode('utf-8'))
                    # doing the http checks
                    elif service == "HTTP":
                        # checking if the HTTP server is up
                        is_up, status_code, description = check_server_http(f"http://{host}")
                        # create a message based on the result
                        response_message = create_message("http_result", data={"is_up": is_up, "status_code": status_code, "description": description})
                        client_socket.send(response_message.encode('utf-8'))
                    # doing the https check
                    elif service == "HTTPS":
                        # check if the HTTPS server is up
                        is_up, status_code, description = check_server_https(f"https://{host}")
                        # create a message based on the result
                        response_message = create_message("https_result", data={"is_up": is_up, "status_code": status_code, "description": description})
                        client_socket.send(response_message.encode('utf-8'))
                    # doing the udp check
                    elif service == "UDP":
                        try:
                            # extract the port from the message data
                            port = parsed_message["data"]["port"]
```



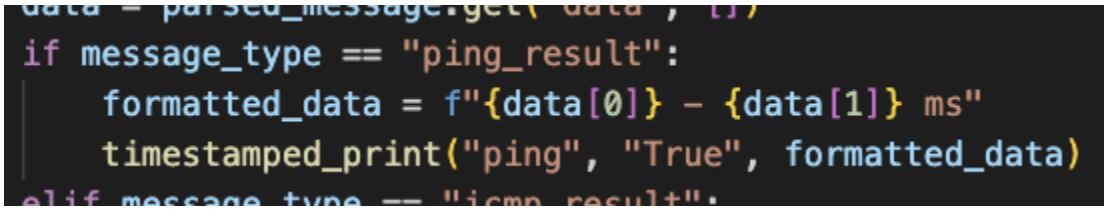
```
                    port = parsed_message["data"]["port"]
                    # checking if the UDP port is open
                    is_open, description = check_udp_port(host, port)
                    # creating a message
                    response_message = create_message("udp_result", data={"is_open": is_open, "description": description})
                    client_socket.send(response_message.encode('utf-8'))
                except KeyError:
                    # error handling if port info is missing
                    response_message = create_message("error", data="Port information is missing")
                    client_socket.send(response_message.encode('utf-8'))
            elif service == "DNS":
                # checking DNS server
                status, results = check_dns_server_status(host, host, 'A')
                # creating a msg from it
                response_message = create_message("dns_result", data={"status": status})
                client_socket.send(response_message.encode('utf-8'))
            # finding the ntp
            elif service == "NTP":
                # checking the ntp server
                status, current_time = check_ntp_server(host)
                # making the message to send to client
                response_message = create_message("ntp_result", data={"status": status, "current_time": current_time})
                client_socket.send(response_message.encode('utf-8'))
            else:
                print(f"[Server] Unknown service: {service}")
        except json.JSONDecodeError:
            print("[Client] Error: Unable to parse server message as JSON")
            # handling
            continue
    except socket.error as e:
        print(f"[Client] Socket error: {e}")
        reconnect()
    except Exception as e:
        print(f"[Client] Error: {e}")
        reconnect()
```

Monitoring Service monitoring actions

For my program, my client reads in a json file. It then checks the interval time and sends the information to my server file. The server file checks for the wanted information, retrieves it, and sends it back to the client. Once the client has received the information, it will print it out in the terminal. All of my services follow the same system.

Ping

For ping, below I have images of the code used to get the wanted output. The first image shows the client printing out information after it is received from the server. The second image shows the server code used to get the ping information:



```
if message_type == "ping_result":
    formatted_data = f"{data[0]} - {data[1]} ms"
    timestamped_print("ping", "True", formatted_data)
```

```python
def ping(host: str, ttl: int = 64, timeout: int = 1, sequence_number: int = 1) -> Tuple[Any, float] | Tuple[Any, None]:
    """
    Send an ICMP Echo Request to a specified host and measure the round-trip time.

    This function creates a raw socket to send an ICMP Echo Request packet to the given host.
    It then waits for an Echo Reply, measuring the time taken for the round trip. If the
    specified timeout is exceeded before receiving a reply, the function returns None for the ping time.

    Args:
    host (str): The IP address or hostname of the target host.
    ttl (int): Time-To-Live for the ICMP packet. Determines how many hops (routers) the packet can pass through.
    timeout (int): The time in seconds that the function will wait for a reply before giving up.
    sequence_number (int): The sequence number for the ICMP packet. Useful for matching requests with replies.

    Returns:
    Tuple[Any, float] | Tuple[Any, None]: A tuple containing the address of the replier and the total ping time in milliseconds.
    If the request times out, the function returns None for the ping time. The address part of the tuple is also None if no reply is received.
    """

    # Create a raw socket with the Internet Protocol (IPv4) and ICMP.
    # socket.AF_INET specifies the IPv4 address family.
    # socket.SOCK_RAW allows sending raw packets (including ICMP).
    # socket.IPPROTO_ICMP specifies the ICMP protocol.
    with socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_ICMP) as sock:
        # Set the Time-To-Live (TTL) for the ICMP packet.
        sock.setsockopt(socket.IPPROTO_IP, socket.IP_TTL, ttl)

        # Set the timeout for the socket's blocking operations (e.g., recvfrom).
        sock.settimeout(timeout)

        # Create an ICMP Echo Request packet.
        # icmp_type=8 and icmp_code=0 are standard for Echo Request.
        # sequence_number is used to match Echo Requests with Replies.
        packet: bytes = create_icmp_packet(icmp_type=8, icmp_code=0, sequence_number=sequence_number)

        # Send the ICMP packet to the target host.
        # The second argument of sendto is a tuple (host, port).
        # For raw sockets, the port number is irrelevant, hence set to 1.
        sock.sendto(packet, (host, 1))

        # Record the current time to measure the round-trip time later.
        start: float = time.time()

        try:
            # Wait to receive data from the socket (up to 1024 bytes).
            # This will be the ICMP Echo Reply if the target host is reachable.
            data, addr = sock.recvfrom(1024)

            # Record the time when the reply is received.
            end: float = time.time()

            # Calculate the round-trip time in milliseconds.
            total_ping_time = (end - start) * 1000

            # Return the address of the replier and the total ping time.
            return addr, total_ping_time
        except socket.timeout:
            # If no reply is received within the timeout period, return None for the ping time.
            return None, None
```

```
[2024-05-16 21:53:25]  [ping] [True] [['172.217.12.132', 0] — 77.82101631164551 ms]
```

HTTP

For HTTP, the first image shows the code in the client used to read in the message from the server and print it to the terminal. The second image shows the code used, in the server file, to

retrieve the wanted information. The third image shows the terminal output back on the client output.

```
83                elif message_type == "http_result":
84                    formatted_data = f"{data}"
85                    timestamped_print("HTTP", "True", formatted_data)
```

```python
def check_server_http(url: str) -> Tuple[bool, Optional[int], str]:
    """
    Check if an HTTP server is up by making a request to the provided URL.

    This function attempts to connect to a web server using the specified URL.
    It returns a tuple containing a boolean indicating whether the server is up,
    the HTTP status code returned by the server, and a description.

    :param url: URL of the server (including http://)
    :return: Tuple (True/False, status code, description)
             True if server is up (status code < 400), False otherwise
    """
    try:
        # Making a GET request to the server
        response: requests.Response = requests.get(url)

        # The HTTP status code is a number that indicates the outcome of the request.
        # Here, we consider status codes less than 400 as successful,
        # meaning the server is up and reachable.
        # Common successful status codes are 200 (OK), 301 (Moved Permanently), etc.
        is_up: bool = response.status_code < 400

        # Returning a tuple: (True/False, status code, description)
        # True if the server is up, False if an exception occurs (see except block)
        return is_up, response.status_code, "Success"

    except requests.RequestException as e:
        # This block catches any exception that might occur during the request.
        # This includes network problems, invalid URL, etc.
        # If an exception occurs, we assume the server is down.
        # Returning False for the status, None for the status code,
        # and the exception description as the description.
        return False, None, str(e)
```

```
[2024-05-16 22:05:35]  [HTTP] [True] [{'is_up': True, 'status_code': 200, 'description': 'Success'}]
```

HTTPS

For HTTPS, the first image shows the code in the client used to read in the message from the server and print it to the terminal. The second image shows the code used, in the server file, to retrieve the wanted information. The third image shows the terminal output back on the client output.

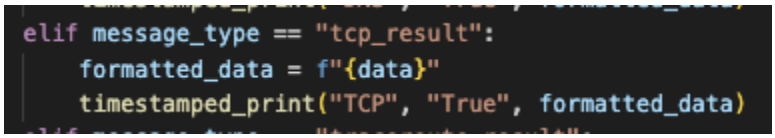
```
80                  elif message_type == "https_result":
81                      formatted_data = f"{data}"
82                      timestamped_print("HTTPS", "True", formatted_data)
```

```
30  def check_server_https(url: str, timeout: int = 5) -> Tuple[bool, Optional[int], str]:
31      """
32      Check if an HTTPS server is up by making a request to the provided URL.
33
34      This function attempts to connect to a web server using the specified URL with HTTPS.
35      It returns a tuple containing a boolean indicating whether the server is up,
36      the HTTP status code returned by the server, and a descriptive message.
37
38      :param url: URL of the server (including https://)
39      :param timeout: Timeout for the request in seconds. Default is 5 seconds.
40      :return: Tuple (True/False for server status, status code, description)
41      """
42      try:
43          # Setting custom headers for the request. Here, 'User-Agent' is set to mimic a web browser.
44          headers: dict = {'User-Agent': 'Mozilla/5.0'}
45
46          # Making a GET request to the server with the specified URL and timeout.
47          # The timeout ensures that the request does not hang indefinitely.
48          response: requests.Response = requests.get(url, headers=headers, timeout=timeout)
49
50          # Checking if the status code is less than 400. Status codes in the 200-399 range generally indicate success.
51          is_up: bool = response.status_code < 400
52
53          # Returning a tuple: (server status, status code, descriptive message)
54          return is_up, response.status_code, "Server is up"
55
56      except requests.ConnectionError:
57          # This exception is raised for network-related errors, like DNS failure or refused connection.
58          return False, None, "Connection error"
59
60      except requests.Timeout:
61          # This exception is raised if the server does not send any data in the allotted time (specified by timeout).
62          return False, None, "Timeout occurred"
63
64      except requests.RequestException as e:
65          # A catch-all exception for any error not covered by the specific exceptions above.
66          # 'e' contains the details of the exception.
67          return False, None, f"Error during request: {e}"
68
```

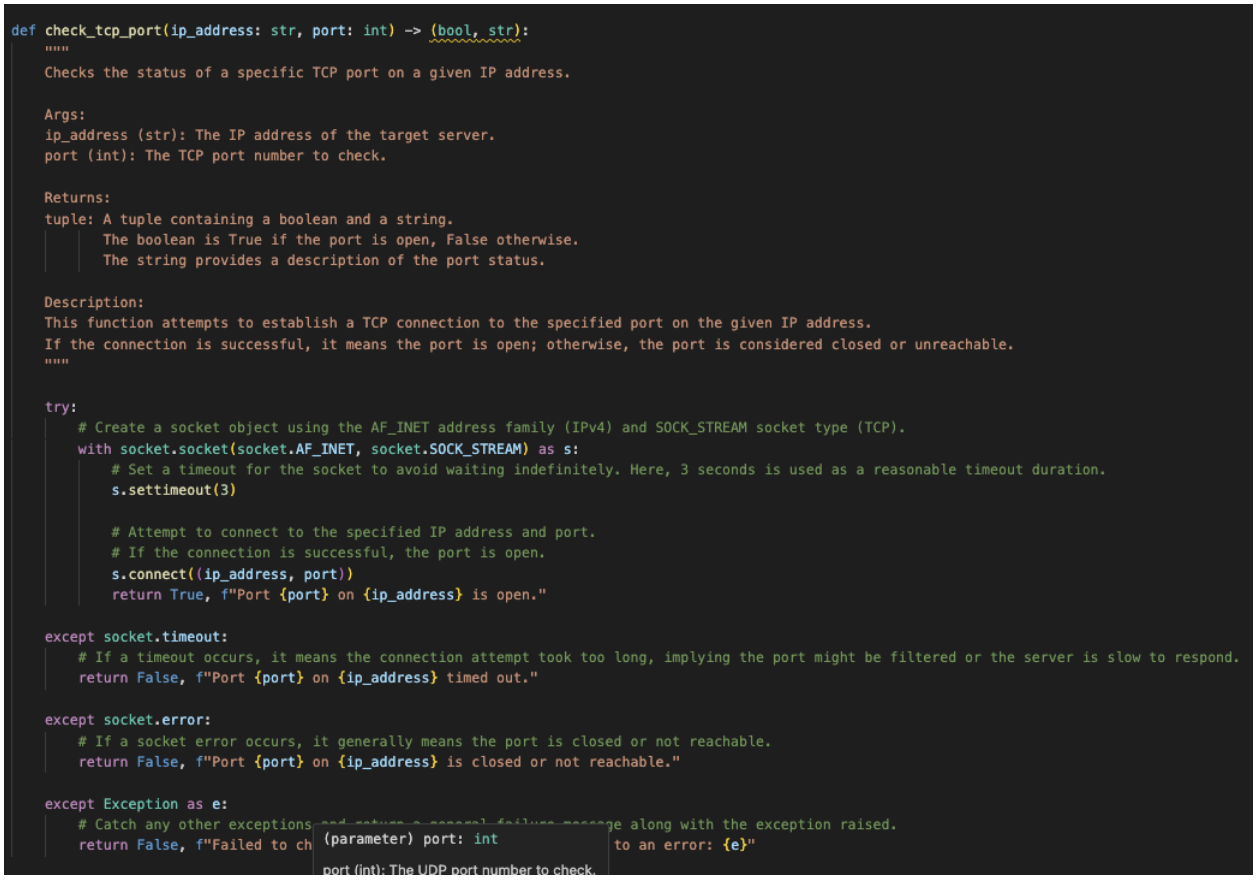
```
[2024-05-16 22:04:10]  [HTTPS] [True] [{'is_up': True, 'status_code': 200, 'description': 'Server is up'}]
[2024-05-16 22:04:15]  [client] [True] [sending information to server... [HTTP]]
```

TCP

For TCP, the first image shows the code in the client used to read in the message from the server and print it to the terminal. The second image shows the code used, in the server file, to retrieve the wanted information. The third image shows the terminal output back on the client output.



```
elif message_type == "tcp_result":
    formatted_data = f"{data}"
    timestamped_print("TCP", "True", formatted_data)
```

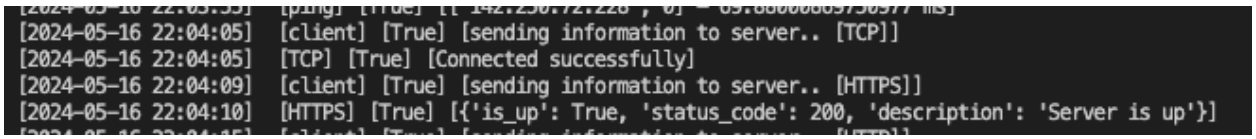


```
def check_tcp_port(ip_address: str, port: int) -> (bool, str):
    """
    Checks the status of a specific TCP port on a given IP address.

    Args:
    ip_address (str): The IP address of the target server.
    port (int): The TCP port number to check.

    Returns:
    tuple: A tuple containing a boolean and a string.
           The boolean is True if the port is open, False otherwise.
           The string provides a description of the port status.

    Description:
    This function attempts to establish a TCP connection to the specified port on the given IP address.
    If the connection is successful, it means the port is open; otherwise, the port is considered closed or unreachable.
    """

    try:
        # Create a socket object using the AF_INET address family (IPv4) and SOCK_STREAM socket type (TCP).
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            # Set a timeout for the socket to avoid waiting indefinitely. Here, 3 seconds is used as a reasonable timeout duration.
            s.settimeout(3)

            # Attempt to connect to the specified IP address and port.
            # If the connection is successful, the port is open.
            s.connect((ip_address, port))
            return True, f"Port {port} on {ip_address} is open."

    except socket.timeout:
        # If a timeout occurs, it means the connection attempt took too long, implying the port might be filtered or the server is slow to respond.
        return False, f"Port {port} on {ip_address} timed out."

    except socket.error:
        # If a socket error occurs, it generally means the port is closed or not reachable.
        return False, f"Port {port} on {ip_address} is closed or not reachable."

    except Exception as e:
        # Catch any other exceptions and return a general failure message along with the exception raised.
        return False, f"Failed to ch
```

(parameter) port: int

port (int): The UDP port number to check.

to an error: {e}"



```
[2024-05-16 22:04:05]  [client] [True] [sending information to server.. [TCP]]
[2024-05-16 22:04:05]  [TCP] [True] [Connected successfully]
[2024-05-16 22:04:09]  [client] [True] [sending information to server.. [HTTPS]]
[2024-05-16 22:04:10]  [HTTPS] [True] [{'is_up': True, 'status_code': 200, 'description': 'Server is up'}]
```

UDP

For UDP, the first image shows the code in the client used to read in the message from the server and print it to the terminal. The second image shows the code used, in the server file, to retrieve the wanted information. The third image shows the terminal output back on the client output.

```
    elif message_type == "udp_result":
        timestamped_print("UDP", "True", formatted_data)
    else:
```

```
def check_udp_port(ip_address: str, port: int, timeout: int = 3) -> (bool, str):
    """
    Checks the status of a specific UDP port on a given IP address.

    Args:
    ip_address (str): The IP address of the target server.
    port (int): The UDP port number to check.
    timeout (int): The timeout duration in seconds for the socket operation. Default is 3 seconds.

    Returns:
    tuple: A tuple containing a boolean and a string.
           The boolean is True if the port is open (or if the status is uncertain), False if the port is definitely closed.
           The string provides a description of the port status.

    Description:
    This function attempts to send a UDP packet to the specified port on the given IP address.
    Since UDP is a connectionless protocol, the function can't definitively determine if the port is open.
    It can only confirm if the port is closed, typically indicated by an ICMP 'Destination Unreachable' response.
    """

    try:
        # Create a socket object using the AF_INET address family (IPv4) and SOCK_DGRAM socket type (UDP).
        with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
            # Set a timeout for the socket to avoid waiting indefinitely.
            s.settimeout(timeout)

            # Send a dummy packet to the specified IP address and port.
            # As UDP is connectionless, this does not establish a connection but merely sends the packet.
            s.sendto(b'', (ip_address, port))

            try:
                # Try to receive data from the socket.
                # If an ICMP 'Destination Unreachable' message is received, the port is considered closed.
                s.recvfrom(1024)
                return False, f"Port {port} on {ip_address} is closed."

            except socket.timeout:
                # If a timeout occurs, it's uncertain whether the port is open or closed, as no response is received.
                return True, f"Port {port} on {ip_address} is open or no response received."

    except Exception as e:
        # Catch any other exceptions and return a general failure message along with the exception raised.
        return False, f"Failed to check UDP port {port} on {ip_address} due to an error: {e}"
```

```
[2024-05-16 22:15:02]  [client] [True] [sending information to server.. [UDP]]
[2024-05-16 22:15:05]  [UDP] [True] [['142.250.188.228', 0] - 116.45627021789551 ms]
[2024-05-16 22:15:09]  [client] [True] [sending information to server.. [DNS]]
```

DNS

For DNS, the first image shows the code in the client used to read in the message from the server and print it to the terminal. The second image shows the code used, in the server file, to retrieve the wanted information. The third image shows the terminal output back on the client output.

```
elif message_type == "dns_result":
    formatted_data = f"Records Results: {data}"
    timestamped_print("DNS", "True", formatted_data)
elif message_type == "tcp_result":
```

```
def check_dns_server_status(server, query, record_type) -> (bool, str):
    """
    Check if a DNS server is up and return the DNS query results for a specified domain and record type.

    :param server: DNS server name or IP address
    :param query: Domain name to query
    :param record_type: Type of DNS record (e.g., 'A', 'AAAA', 'MX', 'CNAME')
    :return: Tuple (status, query_results)
    """
    try:
        # Set the DNS resolver to use the specified server
        resolver = dns.resolver.Resolver()
        resolver.nameservers = [socket.gethostbyname(server)]

        # Perform a DNS query for the specified domain and record type
        query_results = resolver.resolve(query, record_type)
        results = [str(rdata) for rdata in query_results]

        return True, results

    except (dns.exception.Timeout, dns.resolver.NoNameservers, dns.resolver.NoAnswer, socket.gaierror) as e:
        # Return False if there's an exception (server down, query failed, or record type not found)
        return False, str(e)
```

```
[2024-05-16 22:15:14]  [DNS] [True] [Records Results: {'status': False}]
```

NTP

For NTP, the first image shows the code in the client used to read in the message from the server and print it to the terminal. The second image shows the code used, in the server file, to retrieve the wanted information. The third image shows the terminal output back on the client output.

```
elif message_type == "ntp_result":
    timestamped_print("NTP", "True", formatted_data)
else:
```

```
def check_ntp_server(server: str) -> Tuple[bool, Optional[str]]:
    """
    Checks if an NTP server is up and returns its status and time.

    Args:
    server (str): The hostname or IP address of the NTP server to check.

    Returns:
    Tuple[bool, Optional[str]]: A tuple containing a boolean indicating the server status
                                (True if up, False if down) and the current time as a string
                                if the server is up, or None if it's down.
    """
    # Create an NTP client instance
    client = ntplib.NTPClient()

    try:
        # Request time from the NTP server
        # 'version=3' specifies the NTP version to use for the request
        response = client.request(server, version=3)

        # If request is successful, return True and the server time
        # 'ctime' converts the time in seconds since the epoch to a readable format
        return True, ctime(response.tx_time)
    except (ntplib.NTPException, gaierror):
        # If an exception occurs (server is down or unreachable), return False and None
        return False, None
```

```
[2024-05-16 22:27:09]  [client] [True] [sending informa
[2024-05-16 22:27:14]  [NTP] [True] [Records Results: {
```

Resilience and Recovery

For my server file, you can type in “list_clients” and “list_all_queues” to get further information about connections that are occurring. The information can let you know the port number, the client queue as well as that client's unique ID.

```
server command: list_clients
connected clients: 1
 1: ('127.0.0.1', 62723)
[Client] Received message from server: {"type": "execute", "data": {"name": "google", "host": "www.google.com", "port": 80, "service": "ping", "interval": 10, "server_number": 1}, "count": 10}
server command: list_all_queues

 All client queues:
    Client ID 1 (('127.0.0.1', 62723)): 0 items
[Client] Received message from server: {"type": "execute", "data": {"name": "google", "host": "www.google.com", "port": 80, "service": "TCP", "interval": 4, "server_number": 2}, "count": 4}
[Client] Received message from server: {"type": "execute", "data": {"name": "google", "host": "www.google.com", "port": 80, "service": "ping", "interval": 10, "server_number": 1}, "count": 10}
server command:
```

This image shows an exmaple of what a disconnection would look like. This is where the client is disconnected from the server and is able to reconnect and continue on with tasks.