HTTP APPLICATION

CN\_FINALL

Ariel University

Connection networking

Maor Saadon 318532421

Duvi Amiram 305677494

**Contents**

1.1 About the System

1.1.1 Project overview…………………………………………………  
1.1.2 DHCP……………………………………………………………………..  
1.1.3 DNS……………………………………………………………………..  
1.1.4 HTTP APP………………………………………………..  
1.1.5 RUDP  
1.1.6 Diagram of our system ………………………….  
1.1.7 How to run the program

1.2 DHCP

1.2.1 Code Description……………………………………………………..…………  
1.2.2 Flowchart

1.3 DNS

1.3.1 Code Description……………………………………………………..…………  
1.3.2 Flowchart……………………………………………………………………

1.4 HTTP APP

1.4.1 Code Description……………………………………………………..…………  
1.4.2 Flowchart……………………………………………………………..

1.5 Client

1.5.1 Code Description……………………………………………………..…………

1.6 Wireshark……………………………………………………………………………….

1.7 Question………………………………………………………………………….

1.8 Bibliography………………………………………………………………………..……………..

**1.1 About the System**

1.1.1 Project overview

My project focuses on three servers and a client, with the servers being DNS, DHCP, and HTTP application servers. The DNS server will be responsible for translating domain names into IP addresses, while the DHCP server will be in charge of assigning IP addresses dynamically to the client devices. The HTTP application server will host a web application that the client can access through a web browser.

Additionally, the HTTP application server will have the role of performing redirects to another server and downloading files from there. The client device will be configured to interact with all three servers, ensuring seamless communication between them. The project aims to showcase how these servers can work together to provide efficient network services to client devices.

We will expand on the RUDP protocol later, but to answer the questions:

1. how do you overcome lost packages?

2. how the system overcomes latency problems?

Well as requested our system based on the RUDP protocol. A small explanation is given about the RUDP protocol:

RUDP (Reliable User Datagram Protocol) is a transport layer protocol that provides reliability on top of the User Datagram Protocol (UDP) by implementing features such as error detection, retransmission of lost packets, and congestion control. However, even with these features, RUDP may still experience packet loss due to various reasons such as network congestion, faulty network equipment, or temporary network interruptions.

1. To overcome lost packages when using RUDP, the protocol implements a mechanism for retransmitting lost packets. When a packet is sent by the sender, it is assigned a sequence number that is used to track the packets at the receiver. If a packet is lost or not acknowledged by the receiver, the sender will retransmit the packet after a certain amount of time (known as the retransmission timeout).

The retransmission timeout is dynamically adjusted based on the network conditions, and it is typically set to a value that allows the packet to be retransmitted before the receiver's timeout period expires. This helps to ensure that the packet is retransmitted as quickly as possible without causing unnecessary delay or congestion on the network.

In addition to retransmitting lost packets, RUDP also uses checksums to detect errors in the packets. If an error is detected, the receiver will request that the sender retransmit the packet.

Overall, by using a combination of retransmission, error detection, and congestion control mechanisms, RUDP is able to provide reliable data transmission even in the face of packet loss.

2. To overcome latency problems when using RUDP, the protocol implements several techniques that help to minimize the impact of latency on data transmission. These techniques include:

a. Retransmission timeout (RTO) adjustment: RUDP dynamically adjusts the RTO based on the network conditions, such as the round-trip time (RTT) of the packets. This helps to ensure that retransmissions are sent quickly enough to minimize the impact of latency.

b. Selective retransmission: RUDP allows the sender to retransmit only the lost packets, rather than retransmitting all the packets in a stream. This reduces the amount of data that needs to be transmitted and can help to reduce latency.

c. Pipelining: RUDP allows the sender to transmit multiple packets without waiting for an acknowledgement for each packet. This helps to keep the data flowing and can help to reduce the impact of latency.

d. Congestion control: RUDP uses congestion control mechanisms, such as slow start and congestion avoidance, to prevent the network from becoming congested and causing increased latency.

Overall, by using these techniques, RUDP is able to minimize the impact of latency on data transmission and provide reliable and efficient data transfer over the network.

1.1.2 DHCP

What is that DHCP?

A Dynamic Host Configuration Protocol (DHCP) server is a network server that automatically assigns IP addresses and other network configuration parameters to devices on a network. DHCP simplifies network administration by allowing devices to obtain network settings automatically instead of requiring manual configuration.

When a device, such as a computer or a smartphone, connects to a network, it sends a request to the DHCP server for an IP address. The DHCP server responds by assigning a unique IP address to the device, along with other network configuration information such as subnet mask, default gateway, and DNS server addresses.

DHCP servers are commonly used in local area networks (LANs) and are particularly useful in large organizations where manual IP address assignment would be time-consuming and error-prone. DHCP servers can also be configured to assign specific IP addresses to specific devices, which is useful for devices that require a static IP address, such as servers or network printers.

1.1.3 DNS

The Domain Name System (DNS) is a hierarchical distributed naming system for computers, services, or other resources connected to the Internet or a private network. DNS translates human-readable domain names, such as "google.com", into IP addresses that computers use to identify each other on the network.

When a user types a domain name into a web browser, the browser sends a request to a DNS resolver, which is typically provided by the user's Internet service provider (ISP). The resolver then queries one or more DNS servers to obtain the IP address associated with the domain name. Once the IP address is obtained, the browser can then connect to the web server associated with that IP address to retrieve the requested web page.

DNS is critical to the functioning of the Internet as it enables users to access websites and other online services using memorable domain names instead of having to remember the numerical IP addresses associated with those services. DNS is also used for other purposes such as email routing, network address translation (NAT), and other network services.

1.1.4 HTTP APP

HTTP stands for Hypertext Transfer Protocol, which is the primary protocol used for transferring data over the World Wide Web. An HTTP application, also known as a web application, is a software application that uses HTTP as its primary communication protocol to interact with users and other software applications over the internet.

HTTP applications are typically accessed using a web browser, which sends HTTP requests to the web server hosting the application. The server then responds to the requests by sending back HTTP responses, which may include HTML, CSS, JavaScript, images, or other types of data.

Web applications can perform a wide range of tasks, from simple tasks such as displaying web pages or handling user authentication to complex tasks such as online shopping, social networking, and collaborative document editing. Popular examples of web applications include online email services, social media platforms, online banking, and e-commerce websites.

HTTP applications can be built using a variety of programming languages, frameworks, and tools, depending on the specific requirements of the application. Some popular web development frameworks include Ruby on Rails, Django, and Node.js, while popular web development tools include HTML, CSS, JavaScript, and jQuery.

In our case, when a client, such as a web browser, sends a request to a server using the HTTP protocol, the server can respond with an HTTP redirect response code (such as 301 or 302) to indicate that the requested resource has been moved permanently or temporarily to a different location.

In the case of a file download, this could mean that the file is no longer hosted on the original server, but has been moved to a different server or location. The redirect response will include a new URL for the client to request the resource from the new location.

Once the client receives the redirect response, it will typically send a new request to the new URL, and the server at that location will respond with the requested file for download.

This process of redirecting to a new server or location can be useful in a variety of scenarios, such as when a website is moved to a new domain or when large files are hosted on a separate file server for better performance.

1.1.5 RUDP

RUDP (Reliable User Datagram Protocol) is a transport layer protocol designed to provide a reliable, connection-oriented service over unreliable, connectionless transport protocols such as UDP (User Datagram Protocol).

The main purpose of RUDP is to provide a reliable data transfer service to applications that require guaranteed delivery of data. It does this by adding reliability and flow control mechanisms to UDP.

RUDP provides reliable delivery of data by implementing mechanisms such as retransmission of lost or damaged packets, packet sequencing, and acknowledgment of received packets. RUDP also provides flow control to prevent the receiver from being overwhelmed by the sender's data.

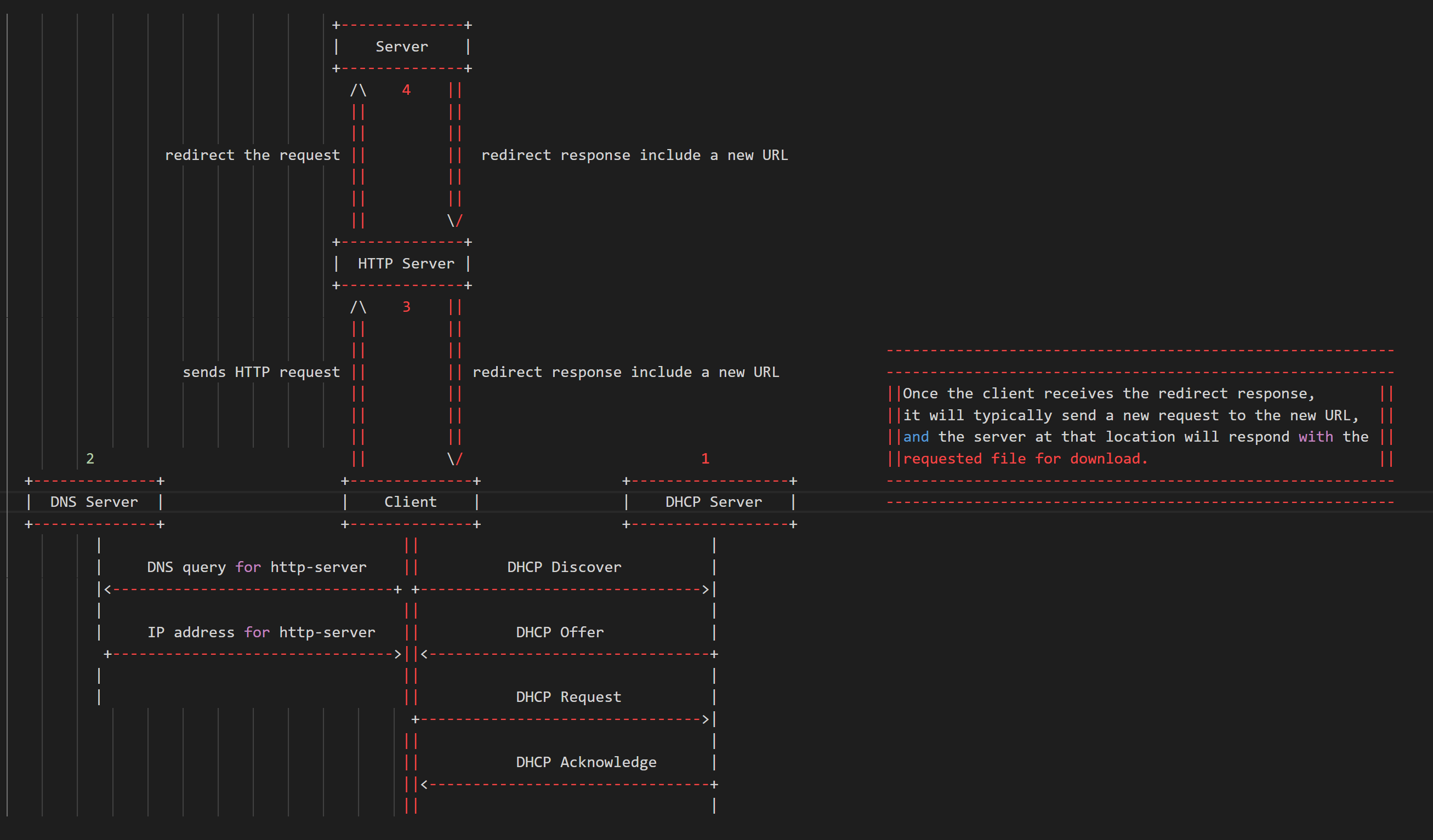
In addition to reliability and flow control, RUDP also supports congestion control. Congestion control mechanisms are used to avoid network congestion by controlling the rate at which data is transmitted. This helps to prevent network congestion, which can result in dropped packets and degraded performance.

RUDP has some advantages over other reliable transport protocols such as TCP (Transmission Control Protocol). One advantage is that RUDP is less complex than TCP, which makes it more suitable for use in embedded systems and other environments with limited resources. RUDP also has lower latency and overhead than TCP, which can be beneficial in applications that require real-time or low-latency data transfer.

However, RUDP also has some disadvantages compared to TCP. For example, RUDP does not provide congestion control as effectively as TCP, which can lead to network congestion and performance issues. Additionally, RUDP is not widely used and may not be supported by all network devices and applications.

Overall, RUDP is a reliable, connection-oriented protocol that can be a good alternative to TCP in certain situations. It provides mechanisms for reliable data transfer, flow control, and congestion control, while also being less complex and having lower latency than TCP. However, its effectiveness in preventing network congestion and its limited support in some environments should be taken into a count when considering its use.

1.1.6 Diagram of our system



The flow of the system is as follows:

1. The client sends a DHCP request to the DHCP server to obtain an IP address.

2. The DHCP server assigns an IP address to the client.

3. The client sends a DNS query to the DNS server to resolve the hostname of the HTTP server.

4. The DNS server responds to the client with the IP address of the HTTP server.

5. The client sends an HTTP request to the HTTP server to download a file.

6. The HTTP server responds to the client with the file.

7. The client receives the file.

1.1.7 How to run the program

**1.2 DHCP**

1.2.1 Code Description

Before we will jump into the code, we want to explain about D.O.R.A.

DORA is an acronym for the four-step process used by DHCP to assign IP addresses to devices on a network.   
Here are the steps involved:

**1. Discover:** The client device sends out a broadcast message on the network, requesting an IP address. The broadcast message includes the client's MAC address and other information about the client.

**2. Offer:** The DHCP server receives the broadcast message and responds with an IP address offer. The server sends a broadcast message back to the client, including an available IP address and other configuration information, such as the subnet mask, DNS servers, and default gateway.

**3. Request:** The client receives the IP address offer from the DHCP server and decides whether to accept it. If the client accepts the offer, it sends a broadcast message to the server, indicating that it wants to use the offered IP address.

**4. Acknowledge:** The DHCP server receives the broadcast message from the client and sends a final message to the client, acknowledging that the IP address has been assigned to the client. This message includes the lease time for the IP address, which indicates how long the client can use the address before it needs to renew the lease.

Overall, the DORA process ensures that IP addresses are assigned in a coordinated and efficient manner, minimizing the risk of IP address conflicts, and making it easier for network administrators to manage network resources.

DHCP server:

**1.** **Import necessary libraries:**

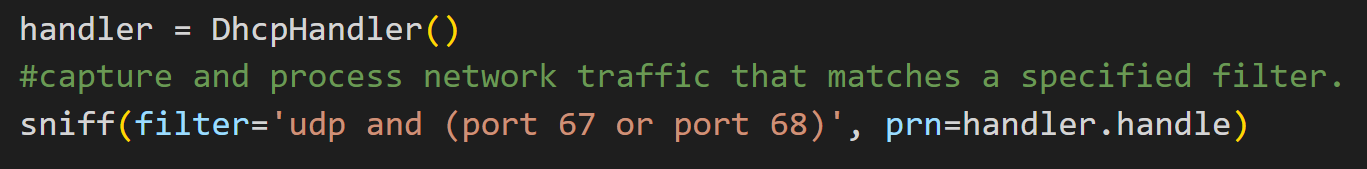
The first step of the code is to import the necessary libraries, namely **scapy.all**, **scapy.layers.dhcp**, **scapy.layers.inet**, and **scapy.layers.l2**. These libraries are used throughout the code to handle and manipulate network packets.

Text

Description automatically generated

**2. Instantiate a DhcpHandler object & Use scapy to sniff network traffic:**

The next step is to create an instance of the DhcpHandler class. And then we will use scapy's **sniff** function to capture and process network traffic that matches a specified filter. The **sniff** function takes two arguments: a filter expression and a callback function (handler.handle in this case). The filter expression is used to specify the types of packets that should be captured and processed by the handle method. In this case, the filter expression specifies that UDP packets with a source or destination port of 67 or 68 should be captured and processed.



**3. Initialize instance variables:**

The **DhcpHandler** class has an **\_\_init\_\_** method that initializes two instance variables. The **ip\_pool** variable is a list of IP addresses that can be assigned to clients. The **ip\_assignments** variable is a dictionary that maps assigned IP addresses to the MAC addresses of the clients that have been assigned those addresses.

Text

Description automatically generated

**4. Define a method to get the next available IP address:**

The **DhcpHandler** class has a **get\_next\_available\_ip** method that returns the next available IP address from the **ip\_pool** list. If there are no more available IP addresses, the method returns None.

Text

Description automatically generated

**5. Define a method to handle DHCP packets:**

The **DhcpHandler** class has a **handle** method that is responsible for handling DHCP packets. The method takes a packet object as an argument and checks whether the packet is a DHCP Discover or a DHCP Request.

If the packet is a DHCP Discover, the method extracts the client's MAC address from the packet, assigns the next available IP address to the client, constructs a DHCP offer packet, and sends the offer packet to the client.

Text

Description automatically generated

If the packet is a DHCP Request, the method extracts the client's MAC address and requested IP address from the packet, assigns the requested IP address to the client, constructs a DHCP ACK packet, and sends the ACK packet to the client.

Text

Description automatically generated

1.2.2 Flowchart

Here's a simple flowchart that outlines the basic steps involved in a DHCP lookup process between a DHCP client and server:

**1.** Get MAC address of client interface

**2.** Send DHCP Discover packet to broadcast address

**3.** Wait for DHCP Offer packet from server

**4.** If Offer received:

a. Send DHCP Request packet with offered IP address and server ID

b. Wait for DHCP Acknowledgement packet from server

**5.** If Acknowledgement received:

a. Configure network interface with assigned IP address and lease time

b. End

Here's a simple diagram that illustrate the communication between a DHCP server and a DHCP client:

A picture containing text

Description automatically generated

In this diagram, the DHCP client sends a DHCP Discover message to the broadcast address. The DHCP server receives the message and responds with a DHCP Offer message. The client then sends a DHCP Request message requesting the offered IP address, and the server sends a DHCP Acknowledge message to confirm that the IP address has been assigned to the client.

**1.3 DNS**

1.3.1 Code Description

The DNS server receives DNS requests from clients, looks up the requested domain name in its DNS zones, and returns the corresponding IP address to the client. The code is divided into several functions, each of which performs a specific task.

Here's a step-by-step explanation of the code:

**1.** Import the socket module. This module provides low-level network communication functionality, including the ability to create and interact with sockets.

Text

Description automatically generated with low confidence

**2.** Define a class named DNSserver. This class has two instance variables: 'addresses' is a dictionary that maps domain names to IP addresses, and 'ip\_domain' is a variable that stores the resolved IP address for a given domain name.

Text

Description automatically generated

**3.** Define a method named **'receive\_dns\_query'** that listens for incoming DNS queries and handles them. The method does the following:

**a.** Creates a UDP socket using the socket.socket() method with parameters 'socket.AF\_INET' for IPv4 and 'socket.SOCK\_DGRAM' for UDP.

**b.** Binds the socket to a local address and port using the bind() method with the parameter tuple ('127.0.0.1', 53), which means the socket will listen for incoming connections on all network interfaces and port 53 (the standard port for DNS).

**c.** Starts an infinite loop that waits for incoming packets using the recvfrom() method of the socket. The method blocks until a packet is received. When a packet is received, the method calls the 'handle\_query\_response' method to construct a DNS response packet, and sends the response packet back to the client using the sendto() method of the socket.

Text

Description automatically generated

**4.** Define a method named 'handle\_query\_response' that constructs a DNS response packet for a given DNS query packet. The method does the following:

**a.** Calls the 'extract\_domain' method to extract the domain name from the query packet.

**b.** Checks if the domain name is in the 'addresses' dictionary. If it is, the method retrieves the IP address from the dictionary. If it's not, the method resolves the IP address using the gethostbyname() method of the socket module.

**c.** Constructs the DNS response packet by concatenating byte strings that represent the different fields of the packet. The response packet includes the ID, QR flag, Opcode, AA flag, TC flag, RD flag, RA flag, Z flag, RCODE, QDCOUNT, ANCOUNT, NSCOUNT, ARCOUNT, the question section of the query packet, and the answer section that contains the IP address of the domain name.

**d.** Returns the response packet as a byte string.

Text

Description automatically generated

**5.** Define a method named 'extract\_domain' that extracts the domain name from the question section of a DNS query packet. The method does the following:

**a.** Initializes an empty string named 'domain' to store the domain name.

**b.** Sets the variable 'pos' to the starting position of the question section.

**c.** Iterates over the question section byte by byte, starting at position 'pos'. The first byte is the length of the first label (i.e., the first part of the domain name delimited by dots). The next 'length' bytes represent the label, which is decoded from ASCII to a string and appended to 'domain'. The process is repeated until the byte with value 0 is encountered, which marks the end of the domain name.

**d.** The trailing dot is removed from the domain name, and the method returns the resulting string.

Text

Description automatically generated

**6.** In the main block of the code, the following steps are executed:

**a.** An instance of the DNSserver class named 'dns\_server' is created.

**b.** The 'receive\_dns\_query' method of the 'dns\_server' instance is called to start listening for incoming DNS queries and handling them.

Text

Description automatically generated

1.3.2 Flowchart

Here's a simple flowchart that outlines the basic steps involved in a DNS lookup process between a DNS client and server:

**1.** DNS client sends a query for a domain name to the local DNS resolver.

**2.** If the local resolver has the IP address for the domain name in its cache, it returns the IP address to the client.

**3.** If the local resolver does not have the IP address in its cache, it sends the query to the root DNS server.

**4.** The root server responds with the IP address of the top-level domain server (such as .com or .org).

**5.** The local resolver sends the query to the top-level domain server.

**6.** The top-level domain server responds with the IP address of the authoritative DNS server for the domain name (such as ns1.example.com).

**7.** The local resolver sends the query to the authoritative DNS server.

**8.** The authoritative DNS server responds with the IP address for the domain name.

**9.** The local resolver caches the IP address and returns it to the client.

Here's a simple diagram that illustrate the communication between a DNS server and a DNS client:

A picture containing text

Description automatically generated

In this diagram, the DNS client sends a query to the DNS server using User Datagram Protocol (UDP) on port 53. The DNS server receives the query and processes it, then sends a response back to the client on UDP port 53. The client receives the response and uses the information provided to complete its request.

Note that in some cases, a DNS server may also communicate with other DNS servers to resolve a query. This communication typically occurs over Transmission Control Protocol (TCP) rather than UDP and may involve multiple rounds of querying and response exchanges. However, for the purposes of this diagram, we have simplified the communication to focus on the basic interaction between a DNS client and a DNS server.

**1.4 HTTP APP**

1.4.1 Code Description

HTTP – RUDP:

This is an implementation of a Reliable UDP (RUDP) HTTP server. The RUDP protocol is built on top of the User Datagram Protocol (UDP), but provides reliability features such as packet acknowledgement and retransmission. Our server uses the Cubic Congestion Control algorithm to manage the flow of packets and avoid network congestion. The server waits for a client to initiate a connection by sending a SYN packet. Once a connection is established, the server can send data packets to the client, and the client can send acknowledgement packets back to the server. The server uses a sliding window mechanism to manage the flow of packets and avoid congestion. The server can also receive file size information from the client, and can send requests to the client to ask for missing packets.

There is some explanation about our code:

**1. deconstruct\_packet(packet):** This is a helper function that takes a packet as an argument and returns a dictionary with the packet's type, sequence number, source address, and data. It first unpacks the packet using struct.unpack() with the format string FORMAT, which is defined as !II and represents two unsigned integers. The first unsigned integer corresponds to the sequence number, and the second corresponds to the packet type. The function then returns a dictionary with the packet's information.

Text

Description automatically generated

**2.** **class RUDPServer:**

**a. \_\_init\_\_(self, ip, port):** This is the constructor method for the RUDPServer class. It creates a UDP socket using the socket module and sets several properties, including the socket's blocking mode, timeout value, and initial congestion window size. It also sets other properties, such as the outgoing sequence number, slow start threshold, and maximum window size.

Text

Description automatically generated

Text

Description automatically generated

**b.** **bind(self):** This method binds the socket to the specified address and port.

Text

Description automatically generated

**c. accept\_connection(self):** This method waits for a SYN packet from a client and responds with a SYN ACK packet to establish a connection. It uses the deconstruct\_packet() helper function to extract the packet's information, and it sets the connected and target\_address properties to indicate that the server is now connected to a client.

**Text

Description automatically generated**

**d. receive\_packet(self):** This method waits for and receives a packet from the client. It uses the deconstruct\_packet() helper function to extract the packet's information and returns it.

Text

Description automatically generated

**e**. **cubic\_algo(self, T):** This method implements the Cubic Congestion Control algorithm to adjust the congestion window size. It takes a time T as an argument and returns the new congestion window size. The algorithm uses the maximum window size (w\_max), a scaling factor (C), and a parameter (beta) to calculate the new window size.

Text

Description automatically generated

**f**. **send\_data(self, address):** This method sends data packets to the client using the congestion control algorithm. It first checks if the server is in the slow start phase or the congestion avoidance phase and adjusts the congestion window size accordingly. If the new window size is smaller than the old window size, the method updates the slow start threshold and sets the last window reduction time. Finally, it sends the packet using the sendto() method and removes the packet from the packets\_to\_send dictionary.

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

**g**. **construct\_payload method:** This method takes the file data as input and constructs the data packets to be sent to the client. It iterates over the chunks of data and packs them into data packets along with sequence numbers.

Text

Description automatically generated

**h. send\_packet\_count method:** This method sends a file size info packet to the client containing the number of packets to be sent. It then waits for an ACK packet from the client acknowledging the file size info packet.

Text

Description automatically generated

**i.** **close\_connection method:** This method sends a CLOSE\_CONNECTION packet to the client and waits for an ACK packet acknowledging the CLOSE\_CONNECTION packet. It then closes the server socket.

Text

Description automatically generated

**3.** **downloadmanager function:** This is the main function that creates a RUDPServer object, binds it to a specific IP address and port, and waits for connection requests from clients. It listens for HTTP GET requests, extracts the file name and host name from the request, constructs a URL from these values, and sends an HTTP GET request to retrieve the file data. It then constructs the data packets to be sent to the client and sends them using the send\_data method of the RUDPServer object. Finally, it waits for more HTTP GET requests from clients and repeats the process until no requests are received for a specified amount of time. When no requests are received for the specified amount of time, it closes the connection with the client using the close\_connection method.

Text

Description automatically generatedText

Description automatically generatedText

Description automatically generated

1.4.2 Flowchart

**RUDP Protocol:**

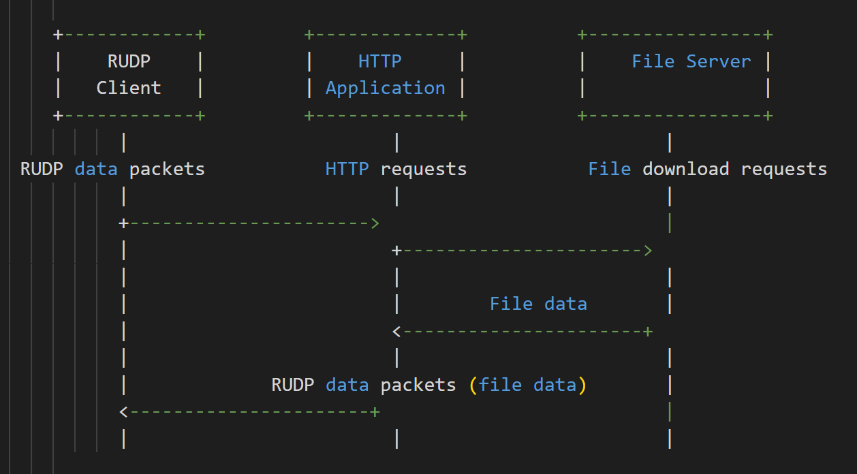
**1.** The client sends a data packet using the RUDP protocol to the HTTP application, which contains the request for an HTTP resource.

**2.** The HTTP application receives the RUDP packet from the client and processes the request.

**3.** The HTTP application sends a request to the file server to download the requested file.

**4.** The file server sends the requested file as a series of RUDP data packets to the HTTP application.

**5.** The HTTP application receives the RUDP data packets from the file server and sends them to the client as RUDP packets.



**TCP Protocol:**

**1.** The client sends a TCP segment to the HTTP application containing the request for an HTTP resource.

**2.** The HTTP application receives the TCP segment from the client and processes the request.

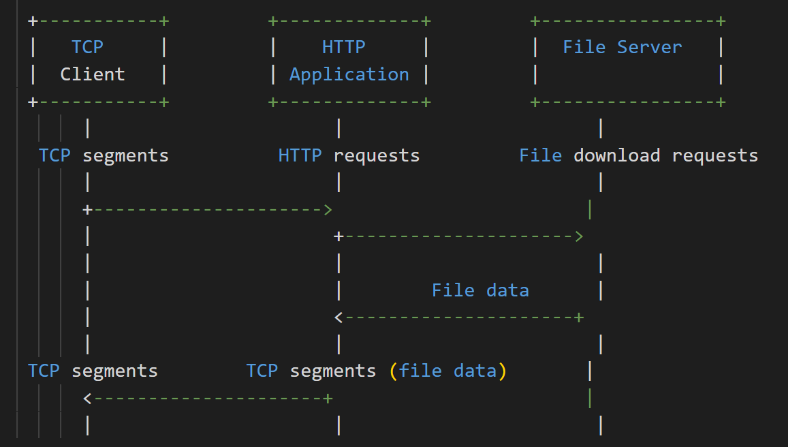
**3.** The HTTP application sends a request to the file server to download the requested file.

**4.** The file server sends the requested file as a series of TCP segments to the HTTP application.

**5.** The HTTP application receives the TCP segments from the file server and sends them to the client as TCP segments.

A picture containing diagram

Description automatically generated



In both cases, the client initiates the request and sends it to the HTTP application. The HTTP application processes the request and communicates with the file server to download the requested file. The file server sends the file back to the HTTP application, which then sends it back to the client. The main difference between RUDP and TCP protocols is in the way the data packets are transmitted, with RUDP providing a simpler, faster transmission method at the cost of reliability, while TCP provides a more reliable transmission method at the cost of increased complexity and potentially slower transmission speeds.

**1.5 Client**

1.5.1 Code Description

This code is distributed to several different clients:

**1. Class DHCPClient:** This class represents a DHCP client that sends Discover and Request packets to a DHCP server to obtain an IP address.

**a. The init function** initializes the client's MAC address and sets the IP address and DNS server IP address to None.

Text

Description automatically generated

**b. The handle\_offer\_packet function** is called when the client receives an offer packet from a DHCP server. It sends a request packet in response to the offer.

Text

Description automatically generated

**c. The handle\_ack\_packet function** is called when the client receives an ACK packet from a DHCP server. It extracts the DNS server IP address from the packet and sets it as a class variable.

Graphical user interface, text

Description automatically generated

**d. The send\_discover\_packet function** constructs and sends a DHCP Discover packet over the network, and then waits for a DHCP offer packet to arrive. When a packet is received, it calls handle\_offer\_packet to send a DHCP request packet.

Text

Description automatically generated

**e. The send\_request\_packet** function constructs and sends a DHCP Request packet to the DHCP server, and then waits for a DHCP ACK packet to arrive. When a packet is received, it calls handle\_ack\_packet to extract the DNS server IP address.

**Text

Description automatically generated**

**2. Class DNSClient :** This class is a simple client that can be used to query a DNS server and obtain the IP address for a given domain name. It contains two functions:

Text

Description automatically generated

**a. query(domain):** This function takes a domain name as input and returns the corresponding IP address by sending a DNS query packet to the DNS server and parsing the response. It first creates a UDP socket and constructs a DNS query packet using the create\_dns\_packet() function. It then sends the packet to the DNS server using the sendto() method and waits for a response using the recvfrom() method. It parses the response packet by extracting the header and answer sections, and then extracts the IP address from the answer section. Finally, it closes the socket and returns the IP address.

Text

Description automatically generated

**b. create\_dns\_packet(domain):** This function takes a domain name as input and constructs a DNS query packet. It first initializes an empty byte string packet, and then adds the various fields of the DNS header, such as the ID, QR flag, opcode, question count, answer count, and so on. It then constructs the question section of the packet by splitting the domain name into its component parts, encoding each part as ASCII and adding it to the packet, along with the appropriate length and termination bytes. Finally, it adds the QTYPE and QCLASS fields to the packet and returns it.

A screenshot of a computer

Description automatically generated with medium confidence

**3.** **deconstruct\_packet(packet):** This static function takes in a packet and returns a dictionary containing the packet type, sequence number, source address, and data. It uses the struct module to unpack the data from the packet.

Text

Description automatically generated

**4. class RUDPClient:**

**a. \_\_init\_\_() :** This function is the constructor for the RUDPClient class. It initializes the following instance variables:

**self.sock:** a socket object for communication.

**self.received\_packets:** a dictionary to hold received data payloads.

**self.outgoing\_seq:** a sequence number for outgoing packets.

**self.server\_address:** the server's address in the form of a tuple (ip, port).

**self.all\_data\_received:** a flag indicating whether all data has been received.

**self.request\_sent:** a flag indicating whether a request has been sent.

Text

Description automatically generated

Text

Description automatically generated**b.** **connect(server\_ip, server\_port):** This function attempts to establish a connection with the server. It does this by creating a SYN packet and sending it to the server. It then waits for a SYN-ACK response from the server, sends an ACK packet in response, and returns True if the connection is established. If it fails to establish a connection within ATTEMPT\_LIMIT attempts, it returns False.

**c. .send\_request(request):** This function sends an HTTP request to the server. It does this by creating a REQUEST packet containing the request and sending it to the server. It then waits for a response from the server by calling self.receive\_data(). If it fails to send the request after 10 attempts, it closes the socket and prints an error message.

Text

Description automatically generated

**d. receive\_data():** This function receives data from the server. It loops until all data has been received. It receives packets from the server, verifies the packet type, and stores the payload in the self.received\_packets dictionary. If the packet is a DATA\_PACKET, it sends an ACK packet in response. If the packet is a FILE\_SIZE\_INFO packet, it stores the total number of packets to be received. If the packet is a CLOSE\_CONNECTION packet, it sets self.all\_data\_received to True and breaks out of the loop. If the packet is a REQUEST\_ACK packet, it sets self.request\_sent to True. If a timeout occurs, it continues looping until all data has been received.

Text

Description automatically generated

Text

Description automatically generated

**7. ack(seq):** This function sends an ACK packet in response to a received packet. It creates an ACK packet containing the sequence number of the packet to acknowledge, sends it to the source address of the packet, and increments self.outgoing\_seq.

Text

Description automatically generated

**5. client\_request(url, file\_name) function:** This is the main function that performs the whole process of sending a DHCP request to obtain an IP address, querying a DNS server for the IP address of the downloadmanager.com domain, establishing a connection with the application server using RUDP protocol, sending a GET request for a specific file, receiving and downloading the file, and saving it to a file with the given file name.

**a. dhcp\_client.send\_discover\_packet() function:** This function sends a DHCP discovery packet to the network to initiate the process of obtaining an IP address. DHCP (Dynamic Host Configuration Protocol) is used to dynamically assign IP addresses to devices on a network.

Text

Description automatically generated

**b. dns\_client.query(domain\_name) function:** This function sends a DNS (Domain Name System) query to the DNS server to obtain the IP address of the given domain name. DNS is used to translate domain names into IP addresses that can be used to establish network connections.

Text

Description automatically generated

**c. rudp\_c.connect(app\_server\_ip, port\_number) function:** This function establishes a connection with the application server using RUDP (Reliable UDP) protocol, which is a protocol that provides reliable data transfer over an unreliable network. It takes the IP address of the application server and a port number as input parameters.

**d. rudp\_c.send\_request(request\_data) function:** This function sends an HTTP GET request to the application server to request a specific file. The request data is passed as a byte array.

**f. rudp\_c.receive\_data() function**: This function receives the response data from the application server and downloads the file. It collects all the packets received from the server, reorders them if necessary, and saves the file to disk.

**Text

Description automatically generated**

Each of these functions performs a specific task that is essential to the overall process of requesting and downloading a file from an application server.

**6. Class HTMLFormServer:** It uses the Flask web framework to create a web server that serves an HTML form to clients and handles their form submissions.

The **\_\_init\_\_** method of the class initializes the Flask app and creates a route for handling form submissions. The route is specified as '/', which is the root path of the server. It supports both GET and POST requests.

When the server receives a POST request, it extracts the values of the hostName and fileName form fields and passes them to the client\_request function. It then prints the values to the console and returns a "Form submitted successfully" message to the client.

When the server receives a GET request, it serves an HTML file that contains a form with two fields: hostName and fileName. The form is submitted using the POST method when the user clicks the "Submit Request" button.

The **run** method of the class starts the Flask app and listens for incoming requests on the specified host and port. By default, the host is set to 'localhost' and the port is set to 5000.

**Text

Description automatically generated**

**Text

Description automatically generated**

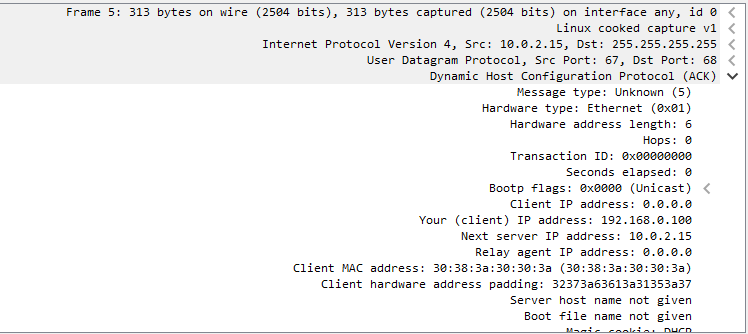
**1.6 Wireshark**

**Table

Description automatically generated**0% packet lost :

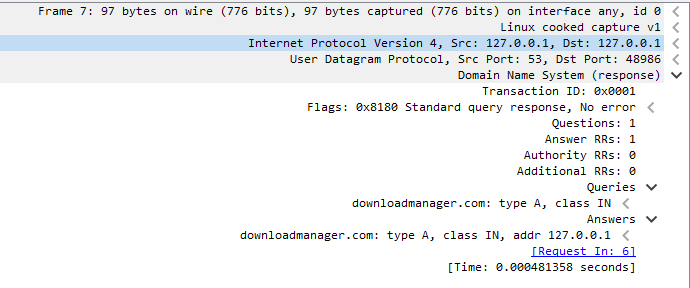
In lines 1-4 we can see the DORA - DHCP process like we explain above.

Inside the ACK packet we can see our new IP:



In lines 5-6 we can see the connection with the DNS server.

We can see in the query response the answer :



In lines 7-9 this is the open connection between the app server and the client. – we can see that the packet len = 8.

In lines 10 -11 we can see the http redirect to another server.

In lines 12 – 28 we can see the closing connection between the app server and the client. – we can see that the packet len = 8.

**Table

Description automatically generated with low confidence**10% packet lost:

we can see that is the same process like in the 0% packets lost, but if you notice there is more packet that has been sent, because when the packet lost the server didn’t get the ACK on this packets and in the RUDP protocol we handle this situation by sending this packets that has been lost again and again until the server get back an ack about the specific packet (we check this with the seq number).

**1.7 Question**

**1.** List at least four major differences between the TCP and QUIC protocols

TCP (Transmission Control Protocol) and QUIC (Quick UDP Internet Connections) are two transport layer protocols used to transfer data over the internet. Here are four major differences between these two protocols:

**1.** Reliability: TCP is a reliable protocol, which means it guarantees the delivery of data and detects lost packets by retransmitting them. QUIC is also reliable, but it uses a different mechanism to achieve reliability. QUIC uses a combination of forward error correction and retransmissions to ensure the delivery of data.

**2.** Connection setup: TCP requires a three-way handshake to establish a connection between the client and server. This process involves a series of messages between the client and server before data transfer can begin. QUIC, on the other hand, uses a single round-trip to establish a connection. This makes QUIC faster than TCP for establishing connections.

**3.** Congestion control: TCP uses a congestion control algorithm that adjusts the flow of data based on the network conditions. QUIC uses a similar congestion control algorithm, but it is more aggressive and can adapt to changing network conditions more quickly than TCP.

**4.** Packet loss recovery: When a packet is lost during transmission, TCP uses a retransmission mechanism to resend the lost packet. QUIC, on the other hand, uses a more efficient mechanism that allows it to recover from multiple lost packets at once, reducing latency and improving performance. This mechanism is called "retransmission with loss detection."

**2.** List at least two main differences between Cubic and Vegas

Cubic and Vegas are two different congestion control algorithms used in TCP. Here are two main differences between Cubic and Vegas:

**1.** Approach: Cubic and Vegas use different approaches to control congestion. Cubic uses a window-based approach that increases the congestion window size slowly at the beginning and then ramps up quickly to utilize available bandwidth. Vegas, on the other hand, uses a delay-based approach that measures the round-trip time (RTT) of packets and adjusts the sending rate based on the observed delay.

**2.** Responsiveness: Cubic and Vegas different in their responsiveness to changes in the network conditions. Cubic is less responsive than Vegas and takes longer to react to changes in the available bandwidth. This makes Cubic more suitable for long-lived flows with steady-state traffic. Vegas, on the other hand, is more responsive and can quickly adapt to changes in network conditions, making it more suitable for short-lived flows and bursty traffic.

3. Explain what the BGP protocol is, how it differs from OSPF and does it work according to short routes.

BGP (Border Gateway Protocol) is a routing protocol used to exchange routing information between different networks on the Internet. It is primarily used by Internet Service Providers (ISPs) to route traffic between different autonomous systems (AS).

Unlike OSPF (Open Shortest Path First), which is an Interior Gateway Protocol (IGP) used within a single autonomous system (AS), BGP is an Exterior Gateway Protocol (EGP) used to exchange routing information between different autonomous systems.

One of the key differences between BGP and OSPF is the way they calculate routes. OSPF uses the shortest path algorithm to calculate the shortest path between two routers within the same AS, whereas BGP uses a policy-based routing approach to determine the best path for traffic to take between different ASes. BGP allows network administrators to define policies that influence the path selection process, based on factors such as AS path length, network performance, and cost.

BGP does not necessarily always work according to the shortest routes. The selection of the best path in BGP is based on various factors, including AS path length, local preference, MED (Multi-Exit Discriminator), and other policy-based attributes. Therefore, BGP may choose a longer path if it meets certain policy criteria, such as avoiding a particular network or preferring a certain type of connection. The primary goal of BGP is to select the best path according to the policies defined by the network administrator.

4. Given the code you developed in this project, please add the data to this table based on your project's message process. Explain how the messages will change if there is a NAT between the user and the servers and whether you will use the QUIC protocol.

If there is a NAT between the HTTP app and the client, and the app is redirecting the request of the client and fetching the file from another server, and then sending it back to the client, the table might look like this:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Application** | **Port Src** | **Port Des** | **Ip Src** | **Ip Des** | **Mac Src** | **Mac Des** |
| HTTP | 50000 | 80 | 192.168.1.100 | NAT's Public IP | HTTP app's MAC | NAT's MAC |
| HTTP | 50001 | 80 | 192.168.1.100 | File server's IP | HTTP app's MAC | File server's MAC |
| HTTP | 50002 | 880 | 192.168.1.100 | Client's IP | HTTP app's MAC | NAT's MAC |

The HTTP app is running on a device with IP address 192.168.1.100 and is listening on port 50000. The client is behind a NAT device with a public IP address, which is used as the destination IP address in the first message. The NAT device translates the source IP address to its own public IP address and forwards the message to the HTTP server.

The HTTP server sends a redirect message back to the client, which contains the IP address of the file server. The HTTP app then sends an HTTP request to the file server to download the file, using a different source port to avoid conflicts with the NAT device's stateful translation.

Once the file is downloaded, the HTTP app sends the file back to the client, using a different source port again to avoid conflicts with the NAT device's stateful translation.

The MAC addresses in this table are specific to the devices involved and will be different for other devices on the network.

Diagram

Description automatically generated

If there is a NAT between the user and the servers, the messages processing will change in the following ways:

**1.** The NAT device will translate the private IP address of the user to a public IP address that is used in the communication with the servers. This means that the IP addresses in the messages exchanged between the user and the servers will be different than the actual IP addresses of the user and the servers.

**2.** The NAT device will also translate the source port number of the user's messages to a different port number that is used in the communication with the servers. This means that the port numbers in the messages exchanged between the user and the servers will be different than the actual port numbers used by the user.

If the HTTP app is using the QUIC (Quick UDP Internet Connections) protocol, it can help overcome some of the issues caused by NAT devices. QUIC is designed to work well in situations where network conditions are suboptimal, including situations with high latency, packet loss, and NAT devices.

QUIC uses UDP instead of TCP as the underlying transport protocol. This means that it can avoid some of the issues caused by TCP's congestion control mechanisms, which can result in degraded performance in situations with high latency and packet loss.

QUIC also supports encryption and multiplexing, which can help improve security and performance. Encryption can help protect against eavesdropping and tampering, while multiplexing can help improve performance by allowing multiple requests and responses to be sent over a single connection.

In summary, if there is a NAT between the user and the servers, using the QUIC protocol can help improve performance and overcome some of the issues caused by NAT devices. However, the specific impact on message processing will depend on the details of the network topology and the configuration of the NAT device.

If the HTTP app is using the QUIC protocol, the table might look like this:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Application** | **Port Src** | **Port Des** | **Ip Src** | **Ip Des** | **Mac Src** | **Mac Des** |
| HTTP | 50000 | 443 | 192.168.1.100 | NAT's Public IP | HTTP app's MAC | NAT's MAC |
| HTTP | 50001 | 443 | 192.168.1.100 | File server's IP | HTTP app's MAC | File server's MAC |
| HTTP | 50002 | 443 | 192.168.1.100 | Client's IP | HTTP app's MAC | NAT's MAC |

The HTTP app is using the QUIC protocol on port 443 (the default port for HTTPS) instead of the HTTP protocol on port 80. The client's message is still sent to the NAT device's public IP address, but this time the destination port is 443, which is the default port for HTTPS and QUIC.

The NAT device translates the source IP address to its public IP address and forwards the message to the HTTP app, which is listening on port 50000. The HTTP app sends a redirect message back to the client, which contains the IP address of the file server and the QUIC protocol's Connection ID.

The HTTP app then establishes a QUIC connection with the file server using a different source port (50001 in this case) and sends an HTTP request over this connection to download the file. The response from the file server is also sent over the QUIC connection, which uses the same Connection ID as the redirect message.

Once the file is downloaded, the HTTP app sends the file back to the client over the same QUIC connection, using a different source port (50002 in this case) to avoid conflicts with the NAT device's stateful translation.

In summary, if the HTTP app is using the QUIC protocol, the table will look similar to the previous table, but with the port numbers changed to 443 and the protocol changed to QUIC. The use of the QUIC protocol can help improve performance and overcome some of the issues caused by NAT devices, as discussed in the previous answer.

5. Explain the differences between the ARP protocol and DNS

ARP (Address Resolution Protocol) and DNS (Domain Name System) are two protocols used in computer networks for different purposes. Here are the main differences between ARP and DNS:

**1.** Purpose: ARP is used to map a physical (MAC) address to an IP address, while DNS is used to map a domain name to an IP address.

**2.** Scope: ARP operates at the data link layer, which means it is used to resolve addresses within a local network segment. DNS operates at the application layer, which means it can be used to resolve addresses across different networks and the internet.

**3.** Functionality: ARP is a simple protocol that sends a broadcast message to the network to request the MAC address of a specific IP address. Once the MAC address is obtained, it is cached in the system's ARP cache for future use. DNS, on the other hand, is a more complex protocol that involves a series of recursive queries to different DNS servers to resolve a domain name to an IP address.

**4.** Implementation: ARP is implemented as a part of the network interface driver in most operating systems and requires no additional software. DNS requires a DNS resolver to be installed on the system, which communicates with one or more DNS servers to resolve domain names.

In summary, ARP is used to map physical addresses to IP addresses within a local network segment, while DNS is used to map domain names to IP addresses across different networks and the internet. ARP operates at the data link layer, while DNS operates at the application layer, and DNS is a more complex protocol that involves recursive queries to multiple servers, while ARP is a simple protocol that sends a broadcast message to obtain the MAC address of a specific IP address.

**1.7 Bibliography**

DNS server :

<https://www.ietf.org/rfc/rfc1035.txt>

<https://www.youtube.com/@howCode>

<https://thepacketgeek.com/>

DHCP server:

<https://www.thepythoncode.com/article/dhcp-listener-using-scapy-in-python>

<https://www.youtube.com/watch?v=e6-TaH5bkjo>

<https://he.wikipedia.org/wiki/Dynamic_Host_Configuration_Protocol>

<https://wiki.wireshark.org/DHCP>

<https://mislove.org/teaching/cs4700/spring11/handouts/project1-primer.pdf>

HTTP server:

<https://requests.readthedocs.io/en/latest/>

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Redirections>

<https://stackoverflow.com/questions/27241804/sending-a-file-over-tcp-sockets-in-python>

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview>

TCP protocol:

<https://www.geeksforgeeks.org/what-is-transmission-control-protocol-tcp/>

QUIC protocol:

<https://peering.google.com/#/learn-more/quic>

TCP – VEGAS:

<https://www.geeksforgeeks.org/basic-concept-of-tcp-vegas/>

<https://pandorafms.com/blog/tcp-congestion-control/>

BGP protocol:

<https://en.wikipedia.org/wiki/Border_Gateway_Protocol>

OSPF protocol:

<https://en.wikipedia.org/wiki/Open_Shortest_Path_First>

NAT:

<https://www.comptia.org/content/guides/what-is-network-address-translation>

<https://en.wikipedia.org/wiki/Network_address_translation>

ARP:

<https://www.fortinet.com/resources/cyberglossary/what-is-arp>

for the debugging. Used only to compare different versions **of our own code**:

<https://www.diffnow.com/samples>

<https://stackoverflow.com/questions/10116518/im-getting-key-error-in-python>

Html :

<https://blog.finxter.com/generating-html-documents-in-python/>

<https://www.geeksforgeeks.org/how-to-write-to-an-html-file-in-python/>

cc :

<https://en.wikipedia.org/wiki/CUBIC_TCP>

rudp:

<https://en.wikipedia.org/wiki/Reliable_User_Datagram_Protocol>

<https://www.geeksforgeeks.org/reliable-user-datagram-protocol-rudp/>

General :

<https://www.google.com/webhp?hl=iw&sa=X&ved=0ahUKEwizqcOCxtP9AhXbTKQEHaLwC1QQPAgI>

man in wsl

<https://www.youtube.com/watch?v=NGLeprazvkM&ab_channel=QuantGuild>

We consulted **verbally** with Zohar Simhon and Matan Weiss about the project.

We also consulted with a tutor named Ruet Rabenuo, a developer on Microsoft, about how to construct the html GUI, and generally about all parts of the code an about how to correctly and conveniently build the app. **We did not receive any code from her.**