# Final Project Network

## TCP over UDP

**Names & Ids:**

Eyad Salama 7128
Hossam Shaybon 6788
Zeyad Zakaria 6764

# Introduction

The project requires the design and implementation of a system that simulates TCP packets using a UDP connection, while maintaining reliability and supporting the HTTP protocol on top of UDP. The solution should include mechanisms for error detection and correction, packet retransmission, and flow control, and should support HTTP 1.0 with HTTP headers.

The implementation should handle packet loss, duplication, and reordering while maintaining a reasonable level of performance, and the system should be evaluated on its effectiveness in achieving reliable data transfer over UDP and its support for the HTTP protocol.

The project requires the implementation of an HTTP server and client using a newly created class that should implement the transition. The HTTP server and client should support the GET and POST methods, and use the Stop-and-wait mechanism. The implementation should calculate the checksum of packets before sending them, and include them in the packet. The implementation should simulate packet loss and corruption, and implement methods that are specially created for this purpose. The implementation should also handle retransmission, duplicate packets, sequence number, handshake, flags like (SYN, SYNACK, ACK, FIN), and timeouts.

The project also offers a bonus for making a valid communication from any web browser to the newly implemented HTTP server and showing the traffic using Wireshark.

# Project Components:

docs
.env.example
Makefile
ParserHttp.py
ParserHttp_test.py
README.md
client.py
client_http.py
my_http.py
server.py
server_http.py
tcp_packet.py
udp_tcp_socket.py

# Codes Explanation:

### ParserHttp_test.py

This code includes three HTTP request strings (req1, req2, and req3) and three HTTP response strings (response1, response2, and response3). These are used to create HttpRequest and HttpResponse objects, respectively, from the ParserHttp module.

HttpRequest objects are created by passing a request string as an argument to the HttpRequest() constructor.

The HttpResponse objects are created by passing a response string to the HttpResponse() constructor.

The example usage at the bottom of the code shows how to create and print an HttpRequest and HttpResponse object.

The output of the example usage displays the HTTP method, URL, HTTP version, and headers of the HttpRequest object, and the HTTP version, status code, reason phrase, headers, and body of the HttpResponse object.

Lastly, the code prints the HttpResponse object again for good measure.

### client.py

This code establishes a client-server communication using a UDP socket to send data, and the TCP protocol is implemented over UDP. The purpose of the client is to send either a file or a message to the server.

Firstly, the code imports necessary modules like socket, pickle, dotenv, os, time, TCPPacket, and TCPOverUDPSocket. These modules are used for socket programming, environment variables, packet structuring, and TCP implementation over UDP.

Next, the code loads environment variables using the load_dotenv() function. This function loads variables stored in a .env file into the environment variables of the system.

Then, the code defines the variables port, address, timeout, and ADDR. port and address define the port number and address of the server respectively. timeout sets the time in seconds for which the client should wait for the server's response before timing out. ADDR is a tuple of the address and port.

The code initializes a TCPOverUDPSocket object udp_socket, sets its timeout to timeout, and connects it to the server address specified by ADDR. set_lossy is also called on udp_socket to set the mode of operation to either lossy or normal mode, depending on user input.

The code prompts the user to select whether to send a file or a message, and based on their input, either a file or a message is sent to the server.

If the user selects to send a file, the file alice.txt located in the docs directory is opened, its contents read, and sent to the server via udp_socket.send(). If the user selects to send a message, the code enters a loop and prompts the user to enter a message. The message is sent to the server using udp_socket.send(). If the message is "exit", the loop exits and the socket is closed.

Finally, the socket is closed using the close() method on udp_socket.

### client_http.py

This code is an example usage of a custom TCPOverUDPSocket class that is built to simulate TCP over UDP protocol.

It starts with importing necessary modules and libraries such as socket, dotenv, os, TCPPacket, ParserHttp, and TCPOverUDPSocket. It then loads environmental variables using load_dotenv() function and sets the port, address, and timeout values for the UDP socket.

Then it creates 4 different types of HTTP requests and 3 different types of HTTP responses, which will be used in the example usage later.

After that, an instance of the TCPOverUDPSocket class is created and set to work with the environmental variables previously set. It then connects to the specified address and port.

In the next part, it creates an instance of the HttpRequest class using req4 as a parameter. This request is then converted into a string message and sent to the server using the UDP socket. It then prints the message sent to the server. If a socket timeout exception is raised, the loop continues.

Finally, the UDP socket is closed and a message is printed that indicates the client is closed.

## server.py

This code sets up a server that listens for incoming connections from clients using a custom implementation of a TCP-like protocol over UDP. It starts by importing necessary modules such as `socket`, `dotenv`, `os`, and `pickle`. It then loads environment variables using `load_dotenv()` function and sets the server's address, port, and timeout values based on the environment variables or default values.

Next, it creates a `TCPOverUDPSocket` object which is an implementation of a TCP-like protocol over UDP. It binds the socket to the server's address and port, and sets a timeout for the socket.

The `handle_client()` function defines how the server will handle incoming client connections. When a new connection is established, the function prints a message indicating that a new connection has been made. It then listens for incoming messages from the client, and when a message is received, it prints the contents of the message using the `print_packet()` function. If a timeout occurs or the connection is closed, the function breaks out of the loop.

Finally, the main loop of the server listens for incoming client connections using `udp_socket.accept()` function, and when a connection is established, it calls `handle_client()` function to handle the connection. The server will continue to listen for incoming connections until it is terminated.

## server_http.py

This code defines a server that listens for incoming requests on a specific port and address using a UDP socket. When a request is received, the server handles the request and sends back an appropriate response.

The code first imports several modules, including the socket module for creating sockets, the dotenv module for loading environment variables from a .env file, the os module for accessing environment variables, the pickle module for serializing and deserializing Python objects, and the ParserHttp module for parsing HTTP requests.

The code then sets some environment variables for the server, including the port number, address, and timeout period.

The code defines several HTTP request and response messages as strings for testing purposes.

The code then creates a UDP socket using the TCPOverUDPSocket class and binds it to the specified address and port. The socket is also set to timeout after a specified period of inactivity.

The code defines a function handle_client() that receives a connection and address, prints a message indicating a new connection has been established, and then enters a loop to receive data from the client. If the connection status is closed, the loop is exited. If data is received, the code prints the received data, parses the HTTP request, and prints the parsed request. If there is a timeout, the connection is closed and the loop is exited.

The code then enters another loop that listens for incoming connections on the socket and calls handle_client() to handle each connection.

## tcp_packet.py

This code defines a class TCPPacket, which represents a TCP packet. The TCP packet contains various fields such as sequence number, acknowledgement number, data offset, flags, checksum, etc. The class also defines methods to set and get these fields.

The code defines a few constants such as DATA_DIVIDE_LENGTH, TCP_PACKET_SIZE, etc., and uses them to calculate the length of the packet and the size of the data.

The code also defines a method to calculate the checksum of the packet, which is used to verify the integrity of the packet during transmission.

The code also defines two static methods to convert a TCPPacket object to a byte string and vice versa, using the Python pickle library. This can be useful when sending the packet over a network.

The TCPPacket class has a few instance variables such as seq, ack, data_offset, flag_ns, flag_cwr, etc. The constructor initializes these variables to their default values. The class also defines methods to set and get these variables.

The TCPPacket class also has a few methods such as packet_type, set_flags, generate_starting_seq_num, etc. The packet_type method returns the type of the packet (e.g., SYN, SYN-ACK, ACK, etc.). The set_flags method is used to set the various flags (e.g., ACK, SYN, FIN) in the packet. The generate_starting_seq_num method generates a random starting sequence number for the packet.

Overall, this code defines a class that represents a TCP packet, and provides various methods to set and get the fields of the packet, as well as to convert the packet to and from a byte string.

## udp_tcp_socket.py

This code is for implementing TCP over a UDP socket. The code imports the necessary modules like socket, random, time, pickle, and threading. It then defines some constants and classes that will be used throughout the code.

The TCPOverUDPSocket class is defined, which has the following methods:

1. __init__(): This is the constructor method that initializes a TCPOverUDPSocket object. It sets the status to open, creates a socket object using the AF_INET and SOCK_DGRAM parameters, and sets the SO_REUSEADDR option for the socket. It also initializes the address and port to None.
2. __repr__(): This method returns a string representation of the object.
3. __str__(): This method returns a string that contains the status, socket, address, and port of the object.
4. set_lossy(): This method sets the lossy parameter of the object.
5. bind(): This method binds the socket to a particular address.
6. settimeout(): This method sets the timeout for the socket.
7. send(): This method sends data over the network by dividing it into packets and sending each packet separately.
8. __divide_data(): This method divides the given data into smaller packets of a fixed length.
9. send_pkt(): This method sends a packet over the network.
10. __send_normal_pkt(): This method sends a packet normally without any loss or corruption.
11. __send_lossy_pkt(): This method sends a packet with the possibility of being lost or corrupted.
12. __wait_for_ack_data(): This method waits for the acknowledgment data for the packet that was sent.
13. rcv(): This method waits for the data to arrive.

The print_packet() method is also defined, which prints a packet with a colored background depending on its type.

## my_http.py

This code sets up a simple HTTP server that listens for incoming connections, reads the requests from the client, generates a response based on the request, and sends the response back to the client.

The code first imports the socket module and the parse_qs function from the urllib.parse module.

Next, it creates a TCP socket using the socket.socket function and sets some options on it using the setsockopt method. It then binds the socket to a local address and port using the bind method, and starts listening for incoming connections using the listen method.

The code enters into a while loop and waits for incoming connections. When a connection is received, it accepts the connection using the accept method, reads the client's request using the recv method, and parses the request method, path, and version using the split method.

If the method is GET, it checks the path to determine what response to generate. If the path is '/' or contains a 'name' parameter, it generates a response that greets the specified name. If the path is '/form', it generates a response that contains an HTML form. If the path is not recognized, it generates a 404 Not Found response.

If the method is POST, it checks the path to determine what response to generate. If the path is '/post', it extracts the name from the request data and generates a response that greets the specified name. If the path is not recognized, it generates a 404 Not Found response.

If the method is not recognized, it generates a 400 Bad Request response.

Finally, it sends the response back to the client using the sendall method and closes the connection using the close method.

## .env.example

The code appears to be defining some constants:

- `PORT=8080`: This sets the value of the constant `PORT` to `8080`. It's likely used to define the port on which the program will listen for incoming connections.
- `ADDRESSS=localhost`: This sets the value of the constant `ADDRESSS` to `localhost`. It's likely used to define the IP address of the server.
- `FORMAT=utf8`: This sets the value of the constant `FORMAT` to `utf8`. It's likely used to define the character encoding format used to send and receive data.
- `TIMEOUT=1`: This sets the value of the constant `TIMEOUT` to `1`. It's likely used to define the maximum time (in seconds) that the server will wait for a client to send a request. If no request is received within the specified timeout period, the server will terminate the connection.

## Makefile

This is a makefile which contains a set of commands that can be executed to build or run a software project. In this makefile, there are several targets specified:

- `main`: When this target is executed, it runs the `main.py` file using Python 3.
- `s`: This target runs the `server.py` file using Python 3.
- `c`: This target runs the `client.py` file using Python 3.
- `httpc`: This target runs the `client_http.py` file using Python 3.
- `https`: This target runs the `server_http.py` file using Python 3.

To use the makefile, you can navigate to the directory where the makefile is located and run the following command:

```
make <target>
```

where `<target>` is the name of the target you want to run. For example, if you want to run the `main` target, you can run the following command:

```
make main
```

This will execute the command specified under the `main` target, which is to run the `main.py` file using Python 3. Similarly, you can run the other targets by specifying their names after the `make` command.

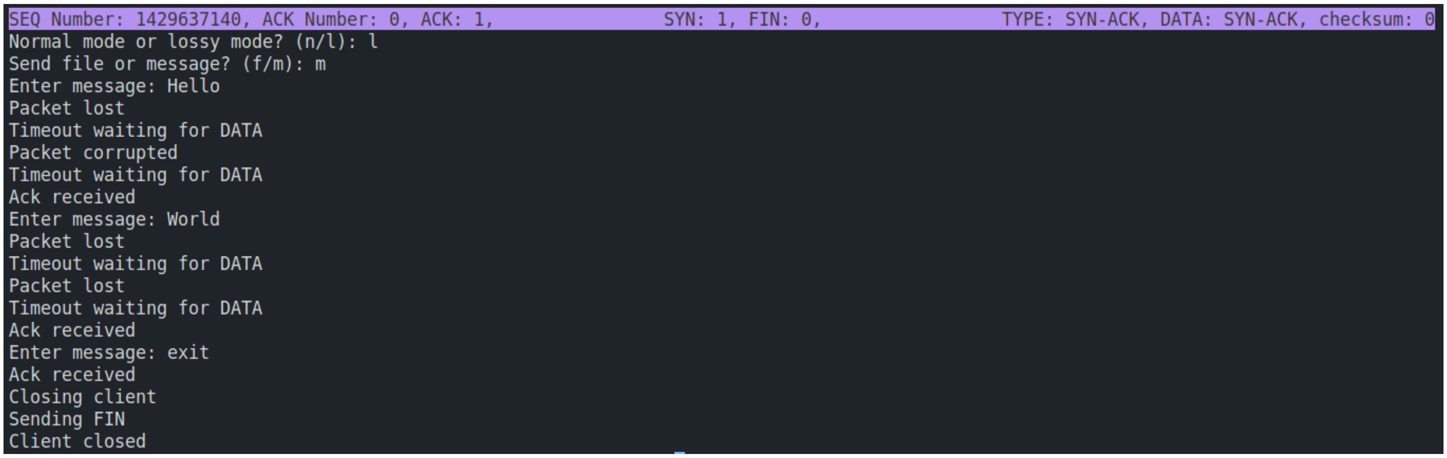
## docs

Contain validation images and Alice.txt file which used in Implementation

# Running Tests

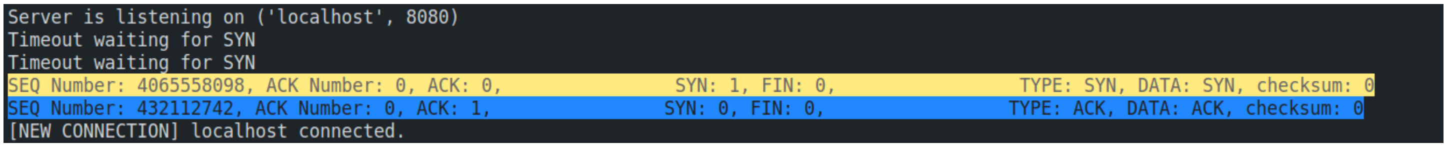
## Sending a message in loosy mode

### Client side

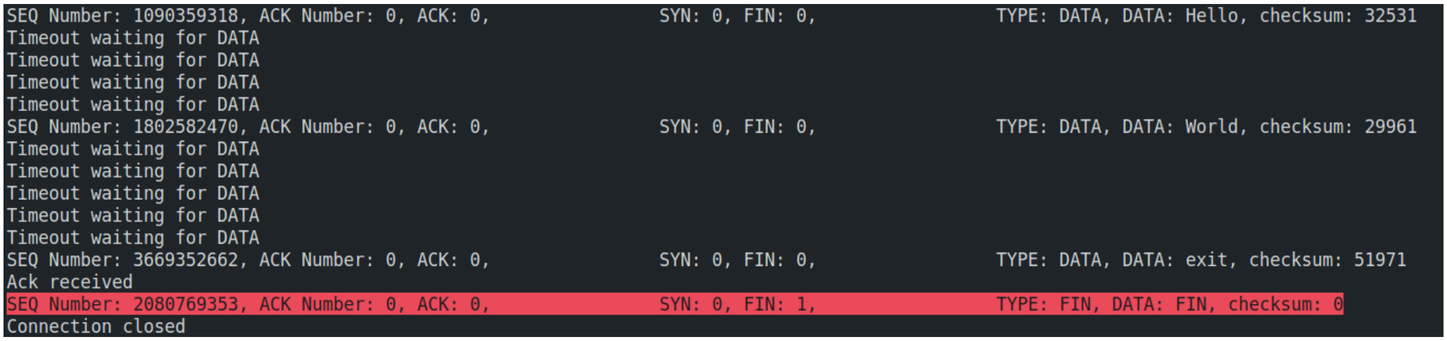


```
SEQ Number: 1429637140, ACK Number: 0, ACK: 1,                 SYN: 1, FIN: 0,                  TYPE: SYN-ACK, DATA: SYN-ACK, checksum: 0
Normal mode or lossy mode? (n/l): l
Send file or message? (f/m): m
Enter message: Hello
Packet lost
Timeout waiting for DATA
Packet corrupted
Timeout waiting for DATA
Ack received
Enter message: World
Packet lost
Timeout waiting for DATA
Packet lost
Timeout waiting for DATA
Ack received
Enter message: exit
Ack received
Closing client
Sending FIN
Client closed
```

### Server side



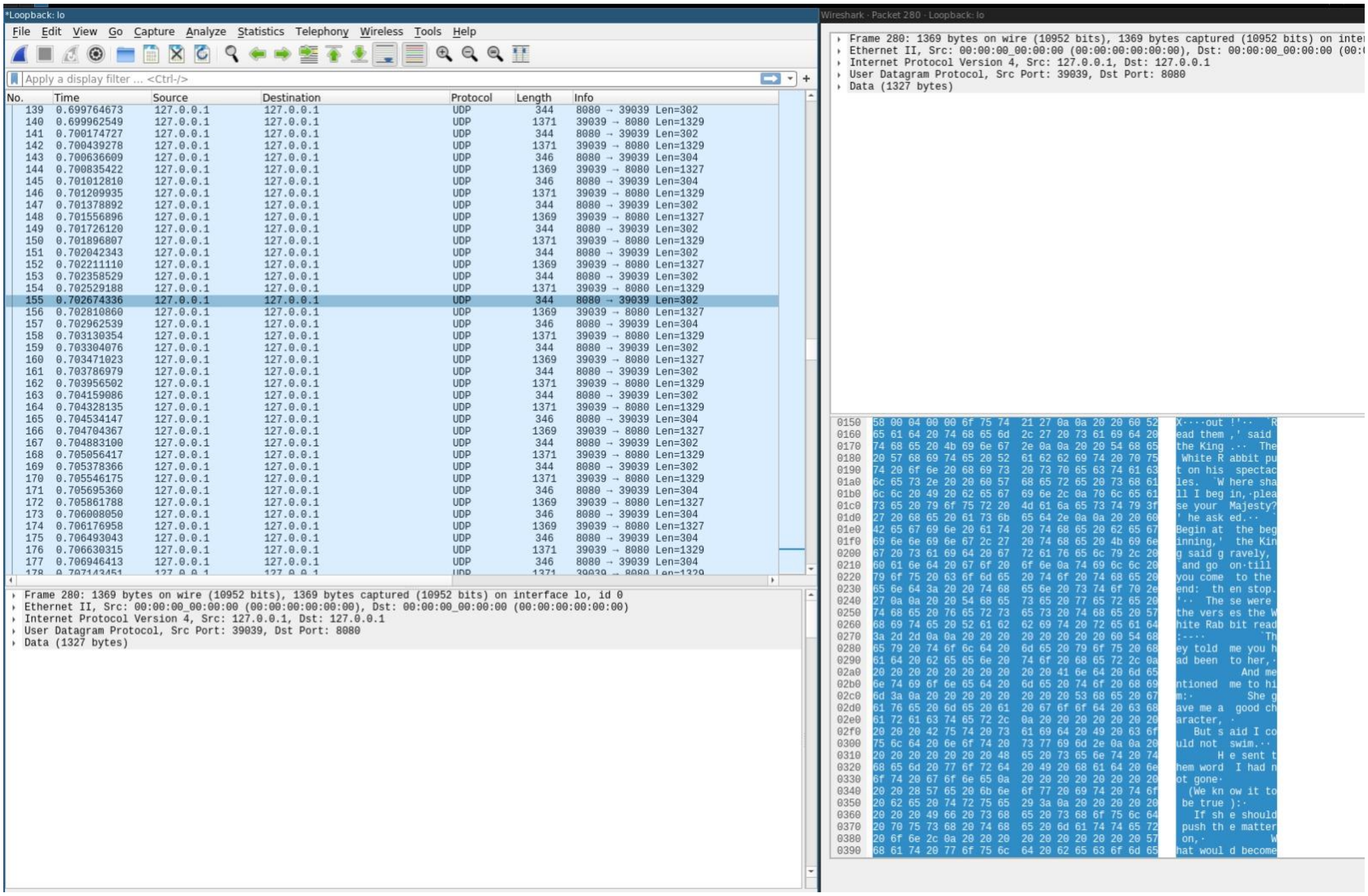
```
Server is listening on ('localhost', 8080)
Timeout waiting for SYN
Timeout waiting for SYN
SEQ Number: 4065558098, ACK Number: 0, ACK: 0,               SYN: 1, FIN: 0,                 TYPE: SYN, DATA: SYN, checksum: 0
SEQ Number: 432112742, ACK Number: 0, ACK: 1,                SYN: 0, FIN: 0,                 TYPE: ACK, DATA: ACK, checksum: 0
[NEW CONNECTION] localhost connected.
```



```
SEQ Number: 1090359318, ACK Number: 0, ACK: 0,               SYN: 0, FIN: 0,                 TYPE: DATA, DATA: Hello, checksum: 32531
Timeout waiting for DATA
Timeout waiting for DATA
Timeout waiting for DATA
Timeout waiting for DATA
SEQ Number: 1802582470, ACK Number: 0, ACK: 0,               SYN: 0, FIN: 0,                 TYPE: DATA, DATA: World, checksum: 29961
Timeout waiting for DATA
Timeout waiting for DATA
Timeout waiting for DATA
Timeout waiting for DATA
Timeout waiting for DATA
SEQ Number: 3669352662, ACK Number: 0, ACK: 0,               SYN: 0, FIN: 0,                 TYPE: DATA, DATA: exit, checksum: 51971
Ack received
SEQ Number: 2080769353, ACK Number: 0, ACK: 0,               SYN: 0, FIN: 1,                 TYPE: FIN, DATA: FIN, checksum: 0
Connection closed
```

# Sending a file in normal mode

## Wireshark


*Loopback: lo
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Wireshark · Packet 280 · Loopback: lo

```
No.   Time         Source      Destination   Protocol  Length  Info
 139  0.699764673  127.0.0.1   127.0.0.1     UDP        344    8080 → 39039 Len=302
 140  0.699962549  127.0.0.1   127.0.0.1     UDP       1371    39039 → 8080 Len=1329
 141  0.700174727  127.0.0.1   127.0.0.1     UDP        344    8080 → 39039 Len=302
 142  0.700439278  127.0.0.1   127.0.0.1     UDP       1371    39039 → 8080 Len=1329
 143  0.700636609  127.0.0.1   127.0.0.1     UDP        346    8080 → 39039 Len=304
 144  0.700835422  127.0.0.1   127.0.0.1     UDP       1369    39039 → 8080 Len=1327
 145  0.701012810  127.0.0.1   127.0.0.1     UDP        346    8080 → 39039 Len=304
 146  0.701209935  127.0.0.1   127.0.0.1     UDP       1371    39039 → 8080 Len=1329
 147  0.701378892  127.0.0.1   127.0.0.1     UDP        344    8080 → 39039 Len=302
 148  0.701556896  127.0.0.1   127.0.0.1     UDP       1369    39039 → 8080 Len=1327
 149  0.701726120  127.0.0.1   127.0.0.1     UDP        344    8080 → 39039 Len=302
 150  0.701896807  127.0.0.1   127.0.0.1     UDP       1371    39039 → 8080 Len=1329
 151  0.702042343  127.0.0.1   127.0.0.1     UDP        344    8080 → 39039 Len=302
 152  0.702211110  127.0.0.1   127.0.0.1     UDP       1369    39039 → 8080 Len=1327
 153  0.702358529  127.0.0.1   127.0.0.1     UDP        344    8080 → 39039 Len=302
 154  0.702529188  127.0.0.1   127.0.0.1     UDP       1371    39039 → 8080 Len=1329
 155  0.702674336  127.0.0.1   127.0.0.1     UDP        344    8080 → 39039 Len=302
 156  0.702810860  127.0.0.1   127.0.0.1     UDP       1369    39039 → 8080 Len=1327
 157  0.702962539  127.0.0.1   127.0.0.1     UDP        346    8080 → 39039 Len=304
 158  0.703130354  127.0.0.1   127.0.0.1     UDP       1371    39039 → 8080 Len=1329
 159  0.703304076  127.0.0.1   127.0.0.1     UDP        344    8080 → 39039 Len=302
 160  0.703471023  127.0.0.1   127.0.0.1     UDP       1369    39039 → 8080 Len=1327
 161  0.703786979  127.0.0.1   127.0.0.1     UDP        344    8080 → 39039 Len=302
 162  0.703956502  127.0.0.1   127.0.0.1     UDP       1371    39039 → 8080 Len=1329
 163  0.704159086  127.0.0.1   127.0.0.1     UDP        344    8080 → 39039 Len=302
 164  0.704328135  127.0.0.1   127.0.0.1     UDP       1371    39039 → 8080 Len=1329
 165  0.704534147  127.0.0.1   127.0.0.1     UDP        346    8080 → 39039 Len=304
 166  0.704704367  127.0.0.1   127.0.0.1     UDP       1369    39039 → 8080 Len=1327
 167  0.704883100  127.0.0.1   127.0.0.1     UDP        344    8080 → 39039 Len=302
 168  0.705056417  127.0.0.1   127.0.0.1     UDP       1371    39039 → 8080 Len=1329
 169  0.705378366  127.0.0.1   127.0.0.1     UDP        344    8080 → 39039 Len=302
 170  0.705546175  127.0.0.1   127.0.0.1     UDP       1371    39039 → 8080 Len=1329
 171  0.705695360  127.0.0.1   127.0.0.1     UDP        346    8080 → 39039 Len=304
 172  0.705861788  127.0.0.1   127.0.0.1     UDP       1369    39039 → 8080 Len=1327
 173  0.706008050  127.0.0.1   127.0.0.1     UDP        346    8080 → 39039 Len=304
 174  0.706176958  127.0.0.1   127.0.0.1     UDP       1369    39039 → 8080 Len=1327
 175  0.706493043  127.0.0.1   127.0.0.1     UDP        346    8080 → 39039 Len=304
 176  0.706630315  127.0.0.1   127.0.0.1     UDP       1371    39039 → 8080 Len=1329
 177  0.706946413  127.0.0.1   127.0.0.1     UDP        346    8080 → 39039 Len=304
 178  0.707143451  127.0.0.1   127.0.0.1     UDP       1371    39039 → 8080 Len=1329
```

```
› Frame 280: 1369 bytes on wire (10952 bits), 1369 bytes captured (10952 bits) on interface lo, id 0
› Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
› Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
› User Datagram Protocol, Src Port: 39039, Dst Port: 8080
› Data (1327 bytes)
```

```
› Frame 280: 1369 bytes on wire (10952 bits), 1369 bytes captured (10952 bits) on inter
› Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:0
› Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
› User Datagram Protocol, Src Port: 39039, Dst Port: 8080
› Data (1327 bytes)
```

```
0150  58 00 04 00 00 6f 75 74  21 27 0a 0a 20 20 60 52   X····out !'··  `R
0160  65 61 64 20 74 68 65 6d  2c 27 20 73 61 69 64 20   ead them ,' said 
0170  74 68 65 20 4b 69 6e 67  2e 0a 0a 20 20 54 68 65   the King .··  The
0180  20 57 68 69 74 65 20 52  61 62 62 69 74 20 70 75    White R abbit pu
0190  74 20 6f 6e 20 68 69 73  20 73 70 65 63 74 61 63   t on his  spectac
01a0  6c 65 73 2e 20 20 60 57  68 65 72 65 20 73 68 61   les.  `W here sha
01b0  6c 6c 20 49 20 62 65 67  69 6e 2c 0a 70 6c 65 61   ll I beg in,·plea
01c0  73 65 20 79 6f 75 72 20  4d 61 6a 65 73 74 79 3f   se your  Majesty?
01d0  27 20 68 65 20 61 73 6b  65 64 2e 0a 0a 20 20 60   ' he ask ed.··  `
01e0  42 65 67 69 6e 20 61 74  20 74 68 65 20 62 65 67   Begin at  the beg
01f0  69 6e 6e 69 6e 67 2c 27  20 74 68 65 20 4b 69 6e   inning,'  the Kin
0200  67 20 73 61 69 64 20 67  72 61 76 65 6c 79 2c 20   g said g ravely, 
0210  60 61 6e 64 20 67 6f 20  6f 6e 0a 74 69 6c 6c 20   `and go  on·till 
0220  79 6f 75 20 63 6f 6d 65  20 74 6f 20 74 68 65 20   you come  to the 
0230  65 6e 64 3a 20 20 74 68  65 6e 20 73 74 6f 70 2e   end:  th en stop.
0240  27 0a 0a 20 20 54 68 65  73 65 20 77 65 72 65 20   '··  The se were 
0250  74 68 65 20 76 65 72 73  65 73 20 74 68 65 20 57   the vers es the W
0260  68 69 74 65 20 52 61 62  62 69 74 20 72 65 61 64   hite Rab bit read
0270  3a 2d 2d 0a 0a 20 20 20  20 20 20 20 20 60 54 68   :--··          `Th
0280  65 79 20 74 6f 6c 64 20  6d 65 20 79 6f 75 20 68   ey told  me you h
0290  61 64 20 62 65 65 6e 20  74 6f 20 68 65 72 2c 0a   ad been  to her,·
02a0  20 20 20 20 20 20 20 20  20 20 41 6e 64 20 6d 65              And me
02b0  6e 74 69 6f 6e 65 64 20  6d 65 20 74 6f 20 68 69   ntioned  me to hi
02c0  6d 3a 0a 20 20 20 20 20  20 20 20 53 68 65 20 67   m:·         She g
02d0  61 76 65 20 6d 65 20 61  20 67 6f 6f 64 20 63 68   ave me a  good ch
02e0  61 72 61 63 74 65 72 2c  0a 20 20 20 20 20 20 20   aracter, ·       
02f0  20 20 20 42 75 74 20 73  61 69 64 20 49 20 63 6f      But s aid I co
0300  75 6c 64 20 6e 6f 74 20  73 77 69 6d 2e 0a 0a 20   uld not  swim.·· 
0310  20 20 20 20 20 20 20 48  65 20 73 65 6e 74 20 74          H e sent t
0320  68 65 6d 20 77 6f 72 64  20 49 20 68 61 64 20 6e   hem word  I had n
0330  6f 74 20 67 6f 6e 65 0a  20 20 20 20 20 20 20 20   ot gone·         
0340  20 20 28 57 65 20 6b 6e  6f 77 20 69 74 20 74 6f     (We kn ow it to
0350  20 62 65 20 74 72 75 65  29 3a 0a 20 20 20 20 20    be true ):·     
0360  20 20 20 49 66 20 73 68  65 20 73 68 6f 75 6c 64      If sh e should
0370  20 70 75 73 68 20 74 68  65 20 6d 61 74 74 65 72    push th e matter
0380  20 6f 6e 2c 0a 20 20 20  20 20 20 20 20 20 20 57    on,·          W
0390  68 61 74 20 77 6f 75 6c  64 20 62 65 63 6f 6d 65   hat woul d become
```