Reliable HTTP over UDP (RUDP) - README Documentation

Berdj Youssef George 8004
Zyad Tarek Ahmed 7962
Jowayria Alaa Eldin 8328
Nada Hanafi Ahmed 8099

Requirement Fulfillment Mapping

Each of the following project requirements is fulfilled in the corresponding source files:

- TCP-like reliable transmission using Stop-and-Wait: rudp_http_client.py, rudp_http_server.py
- Handshake and teardown with TCP flags (SYN, SYNACK, ACK, FIN): rudp_http_client.py, rudp_http_server.py
- CRC32 checksum for error detection: rudp_simulations.py
- Simulate corrupted packets: rudp_simulations.py
- Configurable packet loss and corruption: rudp_config.py, rudp_socket.py
- Handle retransmissions, duplicates, sequence numbers, and timeouts: rudp_http_client.py
- Support HTTP 1.0 GET and POST methods: rudp_http_client.py, rudp_http_server.py
- Handle HTTP headers and request parsing: rudp_http_server.py
- Return HTTP status 200 OK and 404 Not Found: rudp_http_server.py
- Write test cases for GET and POST: test_rudp.py

Execution Order and Code Commentary

This section describes the order in which the code executes and explains the primary responsibilities of each module.
1. test_rudp.py - Entry Point for Tests
start_server(): Uses subprocess to run the server.
run_client_get(): Executes a GET request using subprocess.
run_client_post(): Uses HTTPRUDPClient to perform a POST request.
__main__: Launches the server in a thread, waits 3 seconds, and runs both test cases.
2. rudp_http_server.py - HTTP Server Over RUDP
Initializes a UDP socket and handles client sessions.
parse_http_request(): Parses method, path, headers, and body from request.
build_http_response(): Constructs response with status line and headers.
serve_loop(): Waits for packets, manages handshake, processes GET/POST, handles teardown.
3. rudp_http_client.py - HTTP Client Over RUDP
__init__(): Creates UDP socket and stores server address.
handshake(): Manages connection setup with SYN, SYNACK, and ACK.
send_http_request(): Sends HTTP requests and handles retry logic.
teardown(): Closes connection using FIN and waits for ACK.
4. rudp_socket.py - Socket Selector Based on Simulation
choose_simulation(): Allows user to select simulation scenario.
create_socket_from_config(): Returns socket class instance based on configuration.
5. rudp_simulations.py - RUDP Logic and Simulation
BaseRUDP: Defines core packet format and CRC32 error checking.
CleanRUDPSocket: Implements basic send/receive logic.
LossRUDPSocket: Simulates packet loss.
CorruptRUDPSocket: Simulates packet corruption.
LossCorruptRUDPSocket: Simulates both loss and corruption.
6. rudp_config.py - Simulation Configuration
Stores drop_rate and corrupt_rate simulation parameters.

Functionality Limitations

Limitations of the current RUDP implementation include:
- Stop-and-Wait ARQ only; no sliding window or congestion control.
- No handling of sequence number wraparound or advanced logic.
- Only GET and POST HTTP methods implemented.
- Supports only 200 OK and 404 Not Found HTTP status codes.

Design Trade-Offs

- Stop-and-Wait protocol: Simple to implement but less efficient in high-latency environments.
- CRC32 for checksum: Fast and effective, though not cryptographically secure.
- Separate simulation types (loss/corrupt): Modular simulation, not adaptive to real-world conditions.
- Manual retry loops in client: Effective but lacks exponential backoff.
- One thread per test: Easy to manage but does not allow performance benchmarking.

How to Run

To execute the system:
- Set simulation parameters in rudp_config.py or via input when prompted.
- Run test_rudp.py to start the server and send GET/POST requests.
- Use Wireshark or Postman for traffic analysis and debugging if needed.

RUDP over UDP: Code Walkthrough with Comments

Berdj Youssef George 8004
Zyad Tarek Ahmed 7962
Jowayria Alaa Eldin 8328
Nada Hanafi Ahmed 8099

#Python files with action-specific descriptions

# rudp_config.py

```
# Configuration dictionary specifying simulation type and packet behavior for RUDP
config = {'type': '4', 'drop_rate': 0.2, 'corrupt_rate': 0.2}
```

# rudp_http_client.py

```
# Client class implementing HTTP over Reliable UDP
import socket
import time
from rudp_socket import rudp_socket, SYN, ACK, FIN
MAX_RETRIES = 5
TIMEOUT = 2.0
class HTTPRUDPClient:
    # Initialize client socket, address, and connection state
    def __init__(self, host='localhost', port=8080):
        self.rudp = rudp_socket
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.sock.settimeout(TIMEOUT)
        self.server_address = (host, port)
        self.seq = 0
        self.connected = False

    # Send a packet with optional data and flags using configured rudp socket
    def send_packet(self, data=b"", flags=0):
        self.rudp.send(self.sock, data, self.server_address, seq=self.seq, flags=flags)

    # Receive a packet and return unpacked information
    def recv_packet(self):
        return self.rudp.recv(self.sock)

    # Perform 3-way handshake with SYN -> wait for SYN+ACK -> respond with ACK
    def handshake(self):
        print("[CLIENT] Starting handshake")
        for attempt in range(MAX_RETRIES):
            self.send_packet(flags=SYN)
            try:
                seq, flags, payload, valid, addr = self.recv_packet()
                if valid and (flags & SYN) and (flags & ACK):
                    print("[CLIENT] Received SYN+ACK")
                    self.seq += 1
                    self.send_packet(flags=ACK)
                    self.connected = True
                    return True
            except socket.timeout:
                print(f"[CLIENT] Handshake attempt {attempt + 1} timed out.")
        print("[CLIENT] Handshake failed")
        return False

    # Send FIN to initiate connection teardown and wait for ACK
    def teardown(self):
        if not self.connected:
            return
        print("[CLIENT] Sending FIN")
        for attempt in range(MAX_RETRIES):
            self.send_packet(flags=FIN)
            try:
                seq, flags, payload, valid, addr = self.recv_packet()
                if valid and (flags & ACK):
                    print("[CLIENT] Received ACK for FIN, connection closed")
                    self.connected = False
                    return
            except socket.timeout:
                print(f"[CLIENT] FIN attempt {attempt + 1} timed out.")
        print("[CLIENT] FIN handshake failed")

    # Create and send a raw HTTP request over RUDP and process the response
    def send_http_request(self, method="GET", path="/", headers=None, body=""):
        if not self.connected and not self.handshake():
            return
        if headers is None:
            headers = {}
        headers["Connection"] = "close"

        request_line = f"{method} {path} HTTP/1.0\r\n"
        headers_lines = "".join(f"{k}: {v}\r\n" for k, v in headers.items())
        http_request = request_line + headers_lines + "\r\n" + body

        http_data = http_request.encode('utf-8')
        print(f"[CLIENT] Sending HTTP {method} request with retransmission")

        for attempt in range(MAX_RETRIES):
            self.send_packet(data=http_data)
            try:
                seq, flags, payload, valid, addr = self.recv_packet()
                if valid:
                    response_text = payload.decode('utf-8', errors='replace')
                    print("[CLIENT] HTTP Response received:\n" + response_text)
                    break
                else:
                    print(f"[CLIENT] Received corrupted HTTP response. Retrying...")
            except socket.timeout:
                print(f"[CLIENT] Timeout waiting for response (attempt {attempt + 1})")
        else:
            print("[CLIENT] Failed to receive HTTP response after retries.")
        self.teardown()

if __name__ == "__main__":
    client = HTTPRUDPClient()
    client.send_http_request(method="GET", path="/")
    # client.send_http_request(method="POST", path="/submit", headers={"Content-Type":
"text/plain"}, body="Hello Server")
```

```
#  rudp_http_server.py
from rudp_socket import rudp_socket, SYN, ACK, FIN
class HTTPRUDPServer:
    # Initialize UDP socket, bind address, and session tracker
    def __init__(self, host='localhost', port=8080):
        self.rudp = rudp_socket
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.sock.bind((host, port))
        self.sessions = {}  # session map per client address

    # Send a RUDP packet to a client
    def send_packet(self, data, addr, seq=0, flags=0):
        self.rudp.send(self.sock, data, addr, seq=seq, flags=flags)

    # Parse incoming HTTP request from raw payload
    def parse_http_request(self, data):
        try:
            text = data.decode('utf-8')
            lines = text.split('\r\n')
            request_line = lines[0].split()
            method = request_line[0]
            path = request_line[1]
            headers = {}
            i = 1
            while i < len(lines) and lines[i]:
                if ':' in lines[i]:
                    key, val = lines[i].split(':', 1)
                    headers[key.strip().lower()] = val.strip()
                i += 1
            body = '\r\n'.join(lines[i+1:]) if i+1 < len(lines) else ""
            return method, path, headers, body
        except Exception:
            return None, None, {}, ""

    # Build HTTP response string with headers and body
    def build_http_response(self, status_code=200, body="", headers=None):
        reason = {200: "OK", 404: "Not Found"}.get(status_code, "OK")
        if headers is None:
            headers = {}
        headers_text = "".join(f"{k}: {v}\r\n" for k,v in headers.items())
        response = f"HTTP/1.0 {status_code} {reason}\r\n{headers_text}\r\n{body}"
        return response.encode('utf-8')

    # Main loop to receive and respond to RUDP requests
    def serve_loop(self):
        print("[SERVER] Listening for connections...")
        while True:
            try:
                seq, flags, payload, valid, addr = self.rudp.recv(self.sock)
            except Exception as e:
                print("[SERVER] Error receiving packet:", e)
                continue
            if not valid:
                print(f"[SERVER] Dropped corrupted packet from {addr}")
                continue
            session = self.sessions.get(addr, {"state": "CLOSED", "expected_seq": 0})
            if flags & SYN:
                print(f"[SERVER] SYN received from {addr}")
                self.send_packet(b"", addr, seq=0, flags=SYN | ACK)
                session["state"] = "SYN_RECEIVED"
                session["expected_seq"] = seq + 1
                self.sessions[addr] = session
                continue
            if session["state"] == "SYN_RECEIVED" and flags & ACK:
                print(f"[SERVER] Connection established with {addr}")
                session["state"] = "ESTABLISHED"
                self.sessions[addr] = session
                continue
            if session["state"] == "ESTABLISHED":
                if flags & FIN:
                    print(f"[SERVER] FIN received from {addr}")
                    self.send_packet(b"", addr, seq=seq + 1, flags=ACK)
                    print(f"[SERVER] Connection closed with {addr}")
                    del self.sessions[addr]
                    continue
                if seq != session["expected_seq"]:
                    print(f"[SERVER] Unexpected SEQ {seq} from {addr}, expected {session['expected_seq']}")
                    self.send_packet(b"", addr, seq=0, flags=ACK)
                    continue
                method, path, headers, body = self.parse_http_request(payload)
                if not method:
                    response = self.build_http_response(404, "Not Found")
                else:
                    print(f"[SERVER] {method} request for {path} from {addr}")
                    if method.upper() == "GET" and path == "/":
                        response_body = "<h1>Hello from RUDP HTTP Server!</h1>"
                        headers_resp = {
                            "Content-Type": "text/html",
                            "Content-Length": str(len(response_body))
                        }
                        response = self.build_http_response(200, response_body, headers_resp)
                    elif method.upper() == "POST":
                        response_body = f"Received POST data: {body}"
                        headers_resp = {
                            "Content-Type": "text/plain",
                            "Content-Length": str(len(response_body))
                        }
                        response = self.build_http_response(200, response_body, headers_resp)
                    else:
                        response = self.build_http_response(404, "Not Found")

                self.send_packet(response, addr, seq=seq+1, flags=ACK)
                continue
```

```
            print(f"[SERVER] Ignoring packet from {addr} in state {session['state']}")

if __name__ == "__main__":
    server = HTTPRUDPServer()
    server.serve_loop()
```

# rudp_simulations.py

```
import random
import zlib
import struct

# Define flag values for packet control operations
SYN = 0b0001
ACK = 0b0010
FIN = 0b0100

# Base class with common packing and unpacking logic for RUDP packets
class BaseRUDP:
    HEADER_FORMAT = "!IBI"  # SEQ (4 bytes), FLAGS (1 byte), CHECKSUM (4 bytes)

    # Combine header and payload with checksum into a full packet
    def pack(self, seq, flags, payload):
        checksum = zlib.crc32(payload)
        header = struct.pack(self.HEADER_FORMAT, seq, flags, checksum)
        return header + payload

    # Extract packet fields and verify checksum validity
    def unpack(self, packet):
        if len(packet) < 9:
            return None, None, None, None, False
        seq, flags, recv_checksum = struct.unpack(self.HEADER_FORMAT, packet[:9])
        payload = packet[9:]
        calc_checksum = zlib.crc32(payload)
        valid = (recv_checksum == calc_checksum)
        return seq, flags, recv_checksum, payload, valid

# Reliable socket with no simulation (used for baseline testing)
class CleanRUDPSocket(BaseRUDP):
    def send(self, sock, data, address, seq=0, flags=0):
        packet = self.pack(seq, flags, data)
        sock.sendto(packet, address)

    def recv(self, sock):
        packet, addr = sock.recvfrom(4096)
        seq, flags, checksum, payload, valid = self.unpack(packet)
        return seq, flags, payload, valid, addr

# Simulated socket that randomly drops packets based on drop_rate
class LossRUDPSocket(CleanRUDPSocket):
    def __init__(self, drop_rate=0.3):
        self.drop_rate = drop_rate
    def maybe_drop(self):
        return random.random() < self.drop_rate

    def send(self, sock, data, address, seq=0, flags=0):
        if self.maybe_drop():
            print("[LOSS] Packet dropped")
            return
        super().send(sock, data, address, seq, flags)

# Simulated socket that randomly corrupts payload based on corrupt_rate
class CorruptRUDPSocket(CleanRUDPSocket):
    def __init__(self, corrupt_rate=0.3):
        self.corrupt_rate = corrupt_rate

    def maybe_corrupt(self, data):
        if random.random() < self.corrupt_rate:
            return bytes([b ^ 0xFF for b in data])  # invert bytes
        return data

    def send(self, sock, data, address, seq=0, flags=0):
        packet = self.pack(seq, flags, data)
        header, payload = packet[:9], packet[9:]
        corrupted_payload = self.maybe_corrupt(payload)
        corrupted_packet = header + corrupted_payload
        sock.sendto(corrupted_packet, address)

# Simulated socket that can both drop and corrupt packets
class LossCorruptRUDPSocket(BaseRUDP):
    def __init__(self, drop_rate=0.3, corrupt_rate=0.3):
        self.drop_rate = drop_rate
        self.corrupt_rate = corrupt_rate

    def maybe_drop(self):
        return random.random() < self.drop_rate

    def maybe_corrupt(self, data):
        if random.random() < self.corrupt_rate:
            return bytes([b ^ 0xFF for b in data])
        return data

    def send(self, sock, data, address, seq=0, flags=0):
        if self.maybe_drop():
            print("[LOSS+CORRUPT] Packet dropped")
            return
        packet = self.pack(seq, flags, data)
        header, payload = packet[:9], packet[9:]
        corrupted_payload = self.maybe_corrupt(payload)
        corrupted_packet = header + corrupted_payload
        sock.sendto(corrupted_packet, address)
    def recv(self, sock):
        packet, addr = sock.recvfrom(4096)
        seq, flags, checksum, payload, valid = self.unpack(packet)
        return seq, flags, payload, valid, addr
```

# rudp_socket.py

```
from rudp_simulations import (
    CleanRUDPSocket,
    LossRUDPSocket,
    CorruptRUDPSocket,
    LossCorruptRUDPSocket,
    SYN,
    ACK,
    FIN
)
import os

CONFIG_FILE = "rudp_config.py"

# Write the simulation configuration dictionary to rudp_config.py as Python code
def write_python_config(config_dict):
    with open(CONFIG_FILE, "w") as f:
        f.write(f"config = {repr(config_dict)}\n")

# Prompt user to enter a rate value between 0 and 1
def get_rate(prompt):
    while True:
        try:
            rate = float(input(prompt))
            if 0.0 <= rate <= 1.0:
                return rate
            else:
                print("Please enter a number between 0 and 1.")
        except ValueError:
            print("Invalid input. Please enter a valid number.")

# Prompt user to choose one of the four simulation types
def get_simulation_choice():
    while True:
        print("Choose Simulation Type:")
        print("1 - Clean (No loss or corruption)")
        print("2 - Loss only")
        print("3 - Corruption only")
        print("4 - Loss + Corruption")
        choice = input("Enter choice [1-4]: ").strip()
        if choice in {"1", "2", "3", "4"}:
            return choice
        else:
            print("Invalid choice. Please select a valid option (1-4).")
```

```
# Ask user for parameters depending on choice and store the config to file
# Then create and return the corresponding simulation socket
def choose_simulation():
    choice = get_simulation_choice()
    config = {"type": choice}

    if choice == "2":
        config["drop_rate"] = get_rate("Enter drop rate (0 to 1): ")
    elif choice == "3":
        config["corrupt_rate"] = get_rate("Enter corruption rate (0 to 1): ")
    elif choice == "4":
        config["drop_rate"] = get_rate("Enter drop rate (0 to 1): ")
        config["corrupt_rate"] = get_rate("Enter corruption rate (0 to 1): ")

    write_python_config(config)
    return create_socket_from_config(config)

# Based on loaded config, return the proper RUDP socket class instance
def create_socket_from_config(config):
    if config["type"] == "2":
        return LossRUDPSocket(drop_rate=config["drop_rate"])
    elif config["type"] == "3":
        return CorruptRUDPSocket(corrupt_rate=config["corrupt_rate"])
    elif config["type"] == "4":
        return LossCorruptRUDPSocket(
            drop_rate=config["drop_rate"],
            corrupt_rate=config["corrupt_rate"]
        )
    else:
        return CleanRUDPSocket()

# Load existing simulation config or fall back to user prompt
try:
    from rudp_config import config
    rudp_socket = create_socket_from_config(config)
except (ImportError, FileNotFoundError, AttributeError):
    rudp_socket = choose_simulation()
```

# test_rudp.py

```
import subprocess
import threading
import time

# Start server as a blocking subprocess in a separate thread
def start_server():
    subprocess.run(["python", "rudp_http_server.py"])

# Launch GET request scenario by running the client
def run_client_get():
    subprocess.run(["python", "rudp_http_client.py"])
```

```
# Launch POST request scenario by creating the client and calling send_http_request()
def run_client_post():
    import rudp_http_client
    client = rudp_http_client.HTTPRUDPClient()
    client.send_http_request(method="POST", path="/submit", headers={"Content-Type": "text/plain"},
body="Testing POST")

# Launch server and test both GET and POST requests
if __name__ == "__main__":
    server_thread = threading.Thread(target=start_server, daemon=True)
    server_thread.start()
    time.sleep(3)  # Allow server to initialize before sending requests

    print("Running GET request test")
    run_client_get()

    print("Running POST request test")
    run_client_post()

    print("All tests completed.")
```

Bonus Requirement Report - CC451 Final Project

Bonus Requirement:

The bonus requirement was to enable valid HTTP communication from any web browser to the newly implemented HTTP server over the simulated Reliable UDP (RUDP) layer, and to visualize the traffic using Wireshark.

Implemented Bridge Functionality:

A TCP-to-RUDP bridge was created in the `TCP_UDP_Bridge.py` script. This bridge listens for incoming HTTP requests from web browsers on port 8888 and forwards them as simulated HTTP requests over UDP using the RUDP client to the custom HTTP RUDP server running on port 8080.

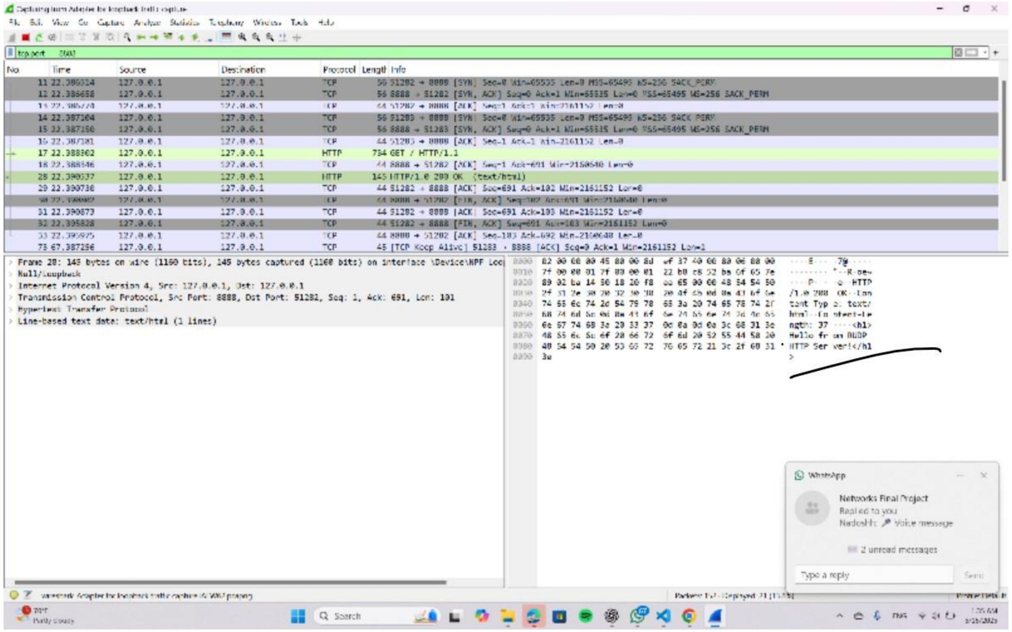
The RUDP client (`HTTPRUDPClient`) handles the 3-way handshake (SYN, SYN+ACK, ACK), HTTP request transmission, and teardown (FIN/ACK sequence) using stop-and-wait logic with timeouts and retries.

Edits Made to Enable Bridge:

1. The `HTTPRUDPClient` class in `rudp_http_client.py` was structured to support a standalone method (`send_http_request`) for reuse inside the bridge.
2. The `TCP_UDP_Bridge.py` file wraps socket handling logic to receive raw HTTP requests from browsers and uses the RUDP client to re-issue them over UDP.
3. Threads are used to allow multiple browser connections concurrently using Python's `threading.Thread`.
4. Return values from the client are validated (`valid=True`) before sending responses back to the browser.
5. The bridge sends appropriate HTTP status codes (502 or 500) in case of RUDP failure.
6. The entire flow supports both GET and POST methods with complete header parsing.

Traffic Validation:

Traffic between the browser and the RUDP HTTP server was captured using Wireshark. Since the bridge uses a local socket on port 8888 and sends to port 8080, the UDP packets and custom flags (SYN, ACK, FIN) along with HTTP payloads were observable in the `.pcapng` file generated.


Capturing from Adapter for loopback traffic capture
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
tcp.port == 8888
WhatsApp
Networks Final Project
Type a reply
Search

```
No.  Time       Source     Destination  Protocol Length Info
  11 22.386514  127.0.0.1  127.0.0.1    TCP        56 51282 → 8888 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
  12 22.386658  127.0.0.1  127.0.0.1    TCP        56 8888 → 51282 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
  13 22.386774  127.0.0.1  127.0.0.1    TCP        44 51282 → 8888 [ACK] Seq=1 Ack=1 Win=2161152 Len=0
  14 22.387104  127.0.0.1  127.0.0.1    TCP        56 51283 → 8888 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
  15 22.387156  127.0.0.1  127.0.0.1    TCP        56 8888 → 51283 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
  16 22.387181  127.0.0.1  127.0.0.1    TCP        44 51283 → 8888 [ACK] Seq=1 Ack=1 Win=2161152 Len=0
  17 22.388302  127.0.0.1  127.0.0.1    HTTP      734 GET / HTTP/1.1
  18 22.388346  127.0.0.1  127.0.0.1    TCP        44 8888 → 51282 [ACK] Seq=1 Ack=691 Win=2150640 Len=0
  28 22.390537  127.0.0.1  127.0.0.1    HTTP      145 HTTP/1.0 200 OK  (text/html)
  29 22.390730  127.0.0.1  127.0.0.1    TCP        44 51282 → 8888 [ACK] Seq=691 Ack=102 Win=2161152 Len=0
  30 22.390807  127.0.0.1  127.0.0.1    TCP        44 8888 → 51282 [FIN, ACK] Seq=102 Ack=691 Win=2150640 Len=0
  31 22.390873  127.0.0.1  127.0.0.1    TCP        44 51282 → 8888 [ACK] Seq=691 Ack=103 Win=2161152 Len=0
  32 22.395828  127.0.0.1  127.0.0.1    TCP        44 51282 → 8888 [FIN, ACK] Seq=691 Ack=103 Win=2161152 Len=0
  33 22.395975  127.0.0.1  127.0.0.1    TCP        44 8888 → 51282 [ACK] Seq=103 Ack=692 Win=2150640 Len=0
  73 67.387256  127.0.0.1  127.0.0.1    TCP        45 [TCP Keep Alive] 51283 → 8888 [ACK] Seq=0 Ack=1 Win=2161152 Len=1
```

```
> Frame 28: 145 bytes on wire (1160 bits), 145 bytes captured (1160 bits) on interface \Device\NPF_Loo
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> Transmission Control Protocol, Src Port: 8888, Dst Port: 51282, Seq: 1, Ack: 691, Len: 101
> Hypertext Transfer Protocol
> Line-based text data: text/html (1 lines)
```

```
0000  02 00 00 00 45 00 00 8d  af 37 40 00 80 06 00 00   ....E... .7@.....
0010  7f 00 00 01 7f 00 00 01  22 b8 c8 52 ba 0f 65 7e   ........ "..R..e~
0020  89 02 ba 14 50 18 20 f8  aa 65 00 00 48 54 54 50   ....P. . .e..HTTP
0030  2f 31 2e 30 20 32 30 30  20 4f 4b 0d 0a 43 6f 6e   /1.0 200  OK..Con
0040  74 65 6e 74 2d 54 79 70  65 3a 20 74 65 78 74 2f   tent-Typ e: text/
0050  68 74 6d 6c 0d 0a 43 6f  6e 74 65 6e 74 2d 4c 65   html..Co ntent-Le
0060  6e 67 74 68 3a 20 33 37  0d 0a 0d 0a 3c 68 31 3e   ngth: 37 ....<h1>
0070  48 65 6c 6c 6f 20 66 72  6f 6d 20 52 55 44 50 20   Hello fr om RUDP 
0080  48 54 54 50 20 53 65 72  76 65 72 21 3c 2f 68 31   HTTP Ser ver!</h1
0090  3e                                                 >
```