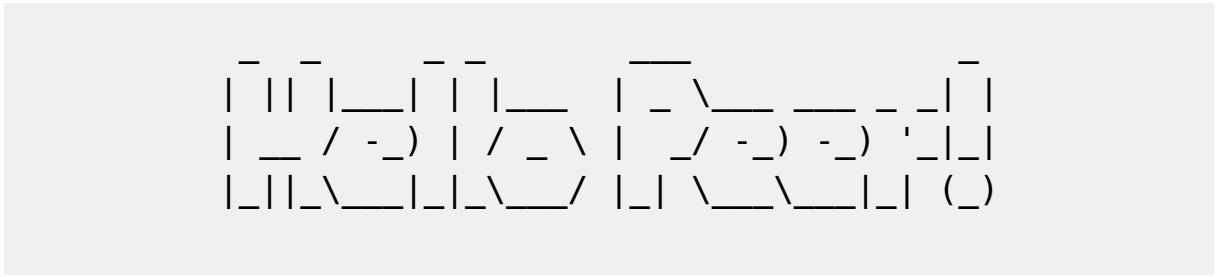


Peer-to-Peer Application
Project Documentation

Anton Dmitriev
Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava
xdmitriev@stuba.sk

Contents

1. Introduction

This project implements peer-to-peer application named “Hello Peer!” that allows message exchange between two nodes on a local Ethernet network. Application uses custom protocol based on UDP (User Datagram Protocol) to transfer data and maintain a connection between two devices acting simultaneously as senders and receivers.

2. Realization

Currently the following functionality is implemented:

Theoretical:
- Development of custom protocol

Practical:
- Connection establishment
- Connection termination
- Text transfer with/without fragmentation
- File transfer with/without fragmentation

Adjustments:
- Setting source node parameters (Port)
- Setting destination node parameters (IP and Port)
- Setting maximum fragment size
- Setting window size
- Setting file saving directory path

Controls:
- Control of message integrity (Checksum)
- Control of connection activity (Keep-Alive)
- Resend of corrupted messages (Automatic Repeat Request)

Testing:
- Setting the percentage of corrupted packets
- Setting the percentage of lost packets
- Setting the percentage of lost windows

Analysis:
- Lua script for Wireshark analysis

Additionally:
- Command Line Interface with command history
- Logging technical data

3. Application

3.1. Program Build

The program is implemented in C++ and utilizes CMake for the build process. The project includes a (`'CMakeLists.txt'`) file, which contains all the necessary configuration settings for compiling the program. If any dependencies are missing, CMake will generate an error message, just simply follow its instructions.

To build the program, follow these steps:

3.1.1. Requirements

Before building the program, ensure that the following requirements are installed on your system:

1. **CMake (version 3.22 or higher)**

- Required to configure and generate the build files. Install with:

```
sudo apt-get install cmake
```

2. **C/C++ compiler**

- The project requires a compiler that supports C/C++. Install with:

```
sudo apt-get install gcc
```

3. **pthread library**

- A standard POSIX thread library for multithreading. It is usually included with build-essential, but if not, make sure it is available:

```
sudo apt-get install build-essential
```

4. **readline library**

- Provides command-line editing and history capabilities. Install with:

```
sudo apt-get install libreadline-dev
```

5. **spdlog library**

- A fast logging library for C++. Install the development package with:

```
sudo apt-get install libspdlog-dev
```

3.1.2. Automatic Build

You can use an Integrated Development Environment (IDE) such as CLion, which provides built-in CMake support and automatically configures the build environment. This approach simplifies the process of program building (Recommended).

3.1.3. Manual Build

If you prefer to build the program manually, execute the following commands in the terminal:

```
mkdir build   # create build directory
cd build      # navigate to build directory
cmake ..      # generate build files using CMake
make          # compile the project
```

3.1.4. Handling Dependencies

Ensure all required dependencies are installed on your system. CMake will notify you of any missing dependencies and provide guidance on how to install them. For example you can install the necessary packages using:

```
sudo apt-get update
sudo apt-get install <package-name>
```

Replace (‘`<package-name>`’) with the required dependency’s name.

3.2. Application Launch

After successful build, simply launch application in two terminals with the following command:

```
./p2p
```

3.3. Help Command

After launching the application, user navigates to the main Command Line Interface (CLI), where immediately see hint, how to find all existing commands:

```
p2p application version 1.3.0 built on Nov 24 2024 01:51:12 by admtrv commit 35f9840
Copyright (c) 2024 Anton Dmitriev. Licensed under the MIT License.
source 'https://github.com/admtrv/HelloPeer'
type 'help' to see available commands

> help
commands:
  proc node port <port>           - set source node port will listen
  proc node dest <ip>:<port>      - set destination node ip and port
  proc node frag size <size>      - set maximum fragment size in bytes (0,1464)
  proc node window size <size>    - set manual window size (disable dynamic window
sizing)
  proc node window dynamic        - enable dynamic window sizing
  proc node file path <path>      - set file save path for received files (default /
home/admtrv/recv)

  proc node connect               - connect to destination node
  proc node disconnect            - disconnect with destination node

  send text <text>                - send text message to destination node
  send file <path>                - send file message to destination node

  set log level <level>           - set log level (trace, debug, info, warn, error,
critical)
  show log                        - display current logs

  set error rate <rate>           - set chance of corrupted packet (0,100)
  set packet loss rate <rate>     - set chance of lost packet (0,100)
  set window loss rate <rate>     - set chance of lost window (0,100)

  exit                            - exit application

>
```

3.4. Application Run

This sequence of commands will set the node to be ready to connect:

- Node 1:

```
> proc node port 5000             # process source node to bind with port 5000
> proc node dest 127.0.0.1:5001   # process destination node to bind with ip 127.0.0.1
and port 5001
>
```

- Node 2:

```
> proc node port 5001             # same logic
> proc node dest 127.0.0.1:5000   # i use loopback to test functionality
>
```

Then use this command on any one of the nodes to establish a connection:

- Node 1:

```
> proc node connect   # process source node to establish connection
connected
>
```

- Node 2:

```
connected             # at this time on destination node
>
```

Once the connection is established, you can exchange text or file messages and dynamically change the size of the fragment:

- Node 1:

```
> proc node frag size 2     # process node to change maximum fragment size to 2 bytes
> send text Hello Peer!
> send file /home/admtrv/file.txt
```

- Node 2:

```
> proc node file path /home/admtrv/recv   # process node to set file saving path
received text Hello Peer!                 # at this time on destination node
received file /home/admtrv/recv/file.txt
>
```

If you realize that you want to close the connection, similar to the command to connect, enter this command on one of the nodes:

- Node 1:

```
> proc node disconnect  # process source node to terminate connection
disconnected
>
```

- Node 2:

```
disconnected            # at this time on destination node
>
```

You can reconnect as many times as you want. When you want to quit the application, enter the command:

```
> exit                  # exit the application
```

3.5. Command History

To navigate through the command history and paste already entered commands simply use the up (`'^'`) and down (`'v'`) arrows.

3.6. Logging

The logs contain technical information that may be useful for debugging or for the more experienced user.

All technical information, such as the size of the transferred file or the time of file transfer, has been moved to the logs in order not to disturb the user during normal use with unnecessary details.

The program recognizes and uses several levels of logging:

- **trace:** Very detailed information (Highest level)
- **debug:** Debugging information
- **info:** General program events
- **warn:** Warnings about potential issues
- **error:** Error messages that need attention
- **critical:** Critical errors that may cause fail (Lowest level)

To customize the logging level, you can enter the command:

```
> set log level debug   # now program process logs from debug to critical level
>
```

To view logs while the application is running you can enter this command:

- Node 1:

```
> show log
2024-10-20 19:20:20 [info] [tcu_pcb::new_phase] new phase 2
2024-10-20 19:20:40 [info] [Node::send_tcu_conn_req] sending tcu connection request
2024-10-20 19:20:40 [info] [Node::send_packet] sent 8 bytes to 127.0.0.1:5001
2024-10-20 19:20:40 [info] [tcu_pcb::new_phase] new phase 3
2024-10-20 19:20:40 [info] [Node::wait_for_ack] waiting for tcu acknowledgment
2024-10-20 19:20:40 [info] [Node::receive_packet] received 8 bytes from 127.0.0.1:5001
2024-10-20 19:20:40 [info] [Node::process_tcu_conn_ack] received tcu connection
acknowledgment
2024-10-20 19:20:40 [info] [tcu_pcb::new_phase] new phase 4
2024-10-20 19:20:47 [info] [tcu_pcb::set_max_frag_size] set max fragment size 2
2024-10-20 19:20:57 [info] [Node::send_text] sent tcu fragment 1 size 2
2024-10-20 19:20:57 [info] [Node::send_packet] sent 10 bytes to 127.0.0.1:5001
2024-10-20 19:20:57 [info] [Node::send_text] sent tcu fragment 2 size 2
2024-10-20 19:20:57 [info] [Node::send_packet] sent 10 bytes to 127.0.0.1:5001
2024-10-20 19:20:57 [info] [Node::send_text] sent tcu fragment 3 size 2
2024-10-20 19:20:57 [info] [Node::send_packet] sent 10 bytes to 127.0.0.1:5001
...
```

- Node 2:

```
> show log
2024-10-20 19:19:30 [info] [tcu_pcb::new_phase] new phase 2
2024-10-20 19:20:40 [info] [Node::receive_packet] received 8 bytes from 127.0.0.1:5000
2024-10-20 19:20:40 [info] [Node::process_tcu_conn_req] received tcu connection request
2024-10-20 19:20:40 [info] [tcu_pcb::new_phase] new phase 3
2024-10-20 19:20:40 [info] [Node::send_tcu_conn_ack] sending tcu connection
acknowledgment
```

```
2024-10-20 19:20:40 [info] [Node::send_packet] sent 8 bytes to 127.0.0.1:5000
2024-10-20 19:20:40 [info] [tcu_pcb::new_phase] new phase 4
2024-10-20 19:20:57 [info] [Node::receive_packet] received 10 bytes from 127.0.0.1:5000
2024-10-20 19:20:57 [info] [Node::process_tcu_more_frag_text] received tcu fragment 1
2024-10-20 19:20:57 [info] [Node::receive_packet] received 10 bytes from 127.0.0.1:5000
2024-10-20 19:20:57 [info] [Node::process_tcu_more_frag_text] received tcu fragment 2
2024-10-20 19:20:57 [info] [Node::receive_packet] received 10 bytes from 127.0.0.1:5000
2024-10-20 19:20:57 [info] [Node::process_tcu_more_frag_text] received tcu fragment 3
2024-10-20 19:20:57 [info] [Node::receive_packet] received 10 bytes from 127.0.0.1:5000
2024-10-20 19:20:57 [info] [Node::process_tcu_more_frag_text] received tcu fragment 4
2024-10-20 19:20:57 [info] [Node::receive_packet] received 10 bytes from 127.0.0.1:5000
2024-10-20 19:20:57 [info] [Node::process_tcu_more_frag_text] received tcu fragment 5
2024-10-20 19:20:57 [info] [Node::receive_packet] received 9 bytes from 127.0.0.1:5000
2024-10-20 19:20:57 [info] [Node::process_tcu_last_frag_text] received tcu last
fragment 6
2024-10-20 19:20:57 [warning] [Node::process_tcu_last_frag_text] invalid checksum in
fragment 2
2024-10-20 19:20:57 [info] [Node::send_tcu_negative_ack] send tcu negative
acknowledgment for fragment 2
2024-10-20 19:20:57 [info] [Node::send_packet] sent 8 bytes to 127.0.0.1:5000
2024-10-20 19:20:57 [info] [Node::receive_packet] received 10 bytes from 127.0.0.1:5000
2024-10-20 19:20:57 [info] [Node::process_tcu_last_frag_text] received tcu last
fragment 2
2024-10-20 19:20:57 [info] [Node::send_tcu_positive_ack] send tcu positive
acknowledgment for fragment 0
2024-10-20 19:20:57 [info] [Node::send_packet] sent 8 bytes to 127.0.0.1:5000
...
```

Also logs can be viewed after the application termination in the file (`'.logs'`). This may help if the program has abruptly terminated and you need to check what caused it.

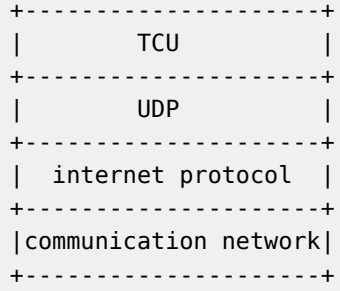
4. Protocol

4.1. Overview

The Transmission Control over UDP (TCU) Protocol is designed to provide a reliable data communication mechanism between two hosts using the UDP (User Datagram Protocol) at the transport layer of the TCP/IP network model.

4.2. Motivation

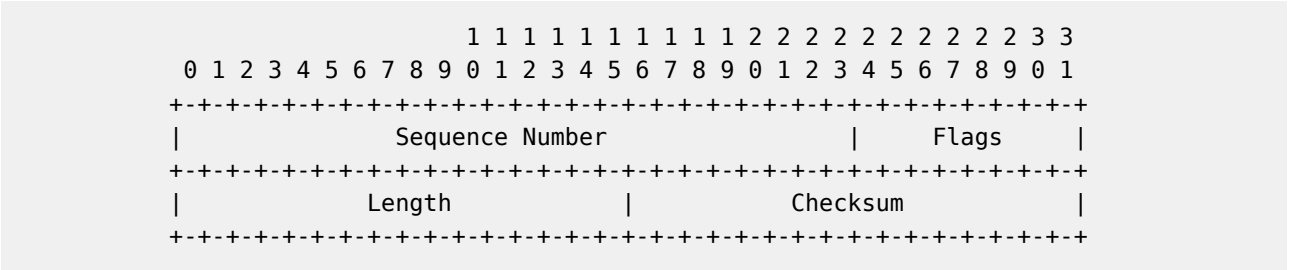
The TCU protocol is designed to provide a simpler and lightweight alternative to the Transmission Control Protocol (TCP) for application that do not require the full complexity of TCP. It is well-suited for tasks like exchanging text and files. By building on top of the User Datagram Protocol (UDP), TCU retains the low-latency and minimal overhead characteristics of UDP, while adding key reliability features to ensure data integrity and proper delivery.



4.3. Functional Specification

The TCU protocol is implemented over UDP, with each UDP datagram encapsulating a TCU packet. It consists of a simple header of constant length followed by the payload data.

4.3.1. Header Format



Sequence Number: 24 bits

- Sequence number of the packet
- When message is fragmented, each fragment has its own sequence number
- Ensures that receiver can reassemble data, even if fragments arrive out of order
- With 24 bits, it is enough to transfer numbers of 16 MB file with 1 byte fragments

Flags: 8 bits (from left to right)

- Control flags that are used to indicate the packet's state
- With 8 bits, it is perfectly sufficient to carry 8 flags and their combinations
  1. SYN (Synchronize): Initiates connection
  2. ACK (Acknowledgment): Acknowledges received packet
  3. FIN (Finish): Indicates termination of connection
  4. NACK (Negative Acknowledgment): Acknowledges corrupted packet
  5. DF (Don't Fragment): Packet is not fragmented
  6. MF (More Fragments): Packet is part of fragmented message and more fragments expected
  7. FL (File Message): Packet is file message
  8. KA (Keep-Alive Message): Packet is heart-beat message

Length: 16 bits

- Length of the payload in bytes
- With 16 bits, it is enough to accommodate maximum payload fragment length, which is 1464 bytes
- This value includes only the payload size, not the header size

Checksum: 16 bits

- Checksum used to verify the integrity of the packet, including the header and payload
- With 16 bits, it is sufficient to transmit checksum generated by chosen algorithm

4.3.2. Finite-State Machine

A connection progresses through a series of states during its lifetime. Briefly the meanings of the states are:

1. **DEAD** - represents dead phase when protocol control block (PCB) does not even exist yet
2. **HOLDOFF** - represents phase when node is in passive waiting state
3. **INITIALIZE** - represents phase when protocol control block (PCB) initialized with its content
4. **CONNECT** - represents phase when node has initiated a connection
5. **NETWORK** - represents phase when connection is fully established and data transmission can occur
6. **DISCONNECT** - represents process of closing connection
7. **CLOSED** - represents phase when connection is completely closed

A TCU connection progresses from one state to another in response to events. The events are the user calls, incoming segments that containing flags; and timeouts.

This state diagrams illustrates state changes, together with the causing events and resulting actions:

Initialization:

```
   DEAD             INITIALIZE
    A --- (init pcb) --> A
```

Connection:

- TCU uses **Two-Way Handshake** to establish connection

```
INITIALIZE           CONNECT                    NETWORK
    A --- (snd SYN) --> A --- (rcv SYN + ACK) --> A
        user command

    B --- (rcv SYN) --> B --- (snd SYN + ACK) --> B
```

Termination:

- TCU uses **Two-Way Handshake** to terminate connection

```
 NETWORK            DISCONNECT                 HOLDOFF
    B --- (snd FIN) --> B --- (rcv FIN + ACK) --> B
        user command

    A --- (rcv FIN) --> A --- (snd FIN + ACK) --> A
```

Acknowledgment Handling:

During the connection and disconnection phases, the protocol uses the `'Node::wait_for_conf_ack()'` method to provide control. When an action request is sent, this method waits for an ACK (Acknowledgement) for a specified `('TIMEOUT_INTERVAL = 5')`. If no confirmation is received within this interval, the connection is automatically closed, and the node transitions to the HOLDOFF state:

```cpp
/*
 * node.cpp
 */

void Node::wait_for_conf_ack()
{
    auto start_time = std::now();
    while (std::now() - start_time < std::seconds(TCU_CONFIRM_TIMEOUT_INTERVAL))
    {
        if (_ack_received)
        {
            _ack_received = false;
            return;
        }
    }

    _pcb.new_phase(TCU_PHASE_HOLDOFF);
    std::cout << "destination node down, connection closed" << std::endl;
}
```

Main Function:

- The FSM in the Node class handles protocol logic based on packet flags:

```
/*
 * node.cpp
 */

void Node::fsm_process(unsigned char* buff)
{
    tcu_packet packet = tcu_packet::from_buff(buff);

    uint16_t flags = packet.header.flags;

    switch (flags)
    {
        case TCU_HDR_FLAG_SYN:
            process_tcu_conn_req(packet);
            break;
        case (TCU_HDR_FLAG_SYN | TCU_HDR_FLAG_ACK):
            process_tcu_conn_ack(packet);
            break;
            ...
    }
}
```

- It extracts flags from the packet header and calls specific methods based on the flag combination.
- Implements a clean separation of logic for different protocol phases.

4.3.3. Checksum

The TCU uses the **CRC16-CCITT** algorithm to calculate a 16-bit checksum value to ensure data integrity in both the header and payload.

Algorithm steps:

1. Initialization: Start with CRC = `0xFFFF`.
2. Byte processing: For each byte of data (header and payload):
   - Perform an XOR between the byte and the high byte of the CRC.
   - Shift the CRC 1 bit to the left.
   - If the high bit is set, perform an XOR with a polynomial of `0x1021`.
3. Repeat: Repeat steps for each bit of each byte.
4. Result: The resulting 16-bit value becomes the checksum, which we insert into the header.

Realization of CRC16-CCITT algorithm located in the method:

```
/*
 * tcu.cpp
 */

void tcu_packet::calculate_crc16(unsigned char* data, size_t length);
{
  ...
}
```

Important Detail:

Before calculating the checksum, the checksum field in the header is temporarily excluded to prevent the checksum from affecting its own value. Only after calculating the correct checksum it is inserted back into the packet's header.

```
/*
 * tcu.cpp
 */

void tcu_packet::calculate_crc()
{
    auto buffer = new unsigned char[total_size];

    std::memcpy(buffer, &header, sizeof(tcu_header) - sizeof(header.checksum));
    std::memcpy(buffer + sizeof(tcu_header) - sizeof(header.checksum), payload,
header.length);

    header.checksum = calculate_crc16(buffer, total_size);
}
```

```
/*
 * tcu.cpp
 */

bool tcu_packet::validate_crc()
{
    auto buffer = new unsigned char[total_size];

    std::memcpy(buffer, &header, sizeof(tcu_header) - sizeof(header.checksum));
    std::memcpy(buffer + sizeof(tcu_header) - sizeof(header.checksum), payload,
header.length);

    uint16_t computed_crc = calculate_crc16(buffer, total_size);

    return computed_crc == header.checksum;
}
```

4.3.4. Automatic Repeat Request

The transmitter ensures reliable data transmission through the **Selective Repeat (SR) with Dynamic Sliding Window** ARQ mechanism. Selective Repeat allows the receiver to request retransmission of only those fragments that were lost or corrupted, optimizing time by avoiding the need to retransmit successfully received fragments.

Transmission:

1. Sender splits a large message into multiple fragments.
2. Each fragment is assigned a sequence number for identification.
3. Fragments are transmitted with appropriate flags:

Text Message:

- Fragment: MF (More Fragments)
- Last Window Fragment: MF (More Fragments) + FIN (Final Fragment)
- Last Message Fragment: NONE

File Message:

- Fragment: MF (More Fragments) + FL (File Message)
- Last Window Fragment: MF (More Fragments) + FIN (Final Fragment) + FL (File Message)
- Last Message Fragment: FL (File Message)

Note: The FIN flag is interpreted as (Final Fragment) during message transmission, not (Finish).

Window Management:

1. Sender and receiver maintains buffers for managing fragments.
2. Sender slides with a window over buffer, sending fragments within the current window.
3. Receiver stores received fragments in buffer and acknowledges the receipt of window.

Dynamic Window Scaling:

- Protocol introduces a dynamic window size calculation using the method:

```
/*
 * node.cpp
 */

void Node::dynamic_window_size()
{
    _window_size = std::max(uint24_t(1), _total_num / uint24_t(5));
}
```

Before sending each message, this method ensures the window size is optimal for the current transmission, balancing efficiency and network conditions. Window size is set to handle 20% of the total fragments in one go. Even if `_total_num` is small, the window size will not be reduced to zero, which would halt data transmission.

- Users can also manually configure the window size using the command:

```
> proc node window size <size>
```

However, manual window configuration disables dynamic scaling.

- To re-enable dynamic window sizing, use the command:

```
> proc node window dynamic
```

Error Detection:

1. Each received fragment is validated by checking its checksum.
2. When receiver detects corrupted or lost fragment in received window, it sends message with NACK (Negative Acknowledgment) and sequence number of corrupted fragment to sender.

Fragment Retransmission:

1. Upon receiving resend request, sender retransmits only corrupted fragment.
2. Only when all received fragments are successful, the receiver sends ACK (Acknowledgment) confirming receipt of entire window.

```
1.      FILE      A                        B
    [1,2,3,4,5,6] A                        B [ , , , , , ]

2.        [1,2,3] A ----- [MF+FL,1] -----> B [1, , , , , ]
          [1,2,3] A ----- [MF+FL,2] -----> B [1,2, , , , ]
          [1,2,3] A --- [MF+FIN+FL,3] ---> B [1,2,3, , , ]

3.        [1,2,3] A                        B [1,2,3, , , ] ERROR 2
          [1,2,3] A <----- [NACK,2] ------ B [1, ,3, , , ]

4.        [1,2,3] A --- [MF+FIN+FL,2] ---> B [1,2,3, , , ] NO ERROR
          [1,2,3] A <----- [ACK,3] ------- B [1,2,3, , , ]
```

```
5.        [4,5,6] A ------- LOST --------> B [1,2,3, , , ]
          [4,5,6] A ----- [MF+FL,5] -----> B [1,2,3, ,5, ]
          [4,5,6] A ------ [FL,6] -------> B [1,2,3, ,5,6]

6.        [4,5,6] A                        B [1,2,3, ,5,6] LOST 4
          [4,5,6] A <---- [NACK,4] ------- B [1,2,3, ,5,6]

7.        [4,5,6] A ------ [FL,4] -------> B [1,2,3,4,5,6] NO LOST
          [4,5,6] A <----- [ACK,6] ------- B [1,2,3, , , ]

8.  [1,2,3,4,5,6] A                        B [1,2,3,4,5,6]
                  A                        B     FILE
```

Window Resending:

If the sender does not receive a positive acknowledgment for the transmitted window after `('TIMEOUT_INTERVAL = 60')`, it resends the entire window using the method:

```
/*
 * node.cpp
 */

void Node::wait_for_recv_ack()
{
    int retry_count = 0;
    int max_retries = TCU_ACTIVITY_ATTEMPT_COUNT;

    while (retry_count < max_retries)
    {
        auto start_time = std::now();
        while (std::now() - start_time < std::seconds(TCU_RECEIVE_TIMEOUT_INTERVAL))
        {
            if (_ack_received)
            {
                _ack_received = false;
                return;
            }
        }
        retry_count++;
        send_window();
    }

    _pcb.new_phase(TCU_PHASE_HOLDOFF);
    std::cout << "destination node down, connection closed" << std::endl;
}
```

If acknowledgment is still not received after resending window for `('ATTEMPT_COUNT = 3')`, the connection is closed.

Simulating Network Conditions:

To test and simulate different network conditions, the following commands are available:

```
> set error rate <rate>         # adjust probability of corrupted packets
> set packet loss rate <rate>   # adjust probability of lost packets
> set window loss rate <rate>   # adjust probability of lost windows
```

Testing Functionality:

To validate ARQ mechanism functionality and robustness, the following test scenarios were conducted with a **2 MB** file:

1. Baseline Test (No Errors):

- Node 1:

```
> send file /home/admtrv/2mb.txt
sending file...
complete
> show log
2024-11-24 04:51:15 [392359] [info] [Node::set_window_size] set dynamic window size 273
2024-11-24 04:51:15 [392359] [info] [Node::send_file] sent tcu fragmented file name
2mb.txt size 2000012 fragments 1367 fragment size 1464
2024-11-24 04:51:15 [392359] [info] [Node::send_window] sending window range [1,273]
2024-11-24 04:51:15 [392359] [info] [Node::send_window] sending tcp fragment 1
2024-11-24 04:51:15 [392359] [info] [Node::send_packet] sent 1472 bytes to
127.0.0.1:5000
2024-11-24 04:51:15 [392359] [info] [Node::send_window] sending tcp fragment 2
2024-11-24 04:51:15 [392359] [info] [Node::send_packet] sent 1472 bytes to
127.0.0.1:5000

...

2024-11-24 04:51:17 [392359] [info] [Node::send_window] sending tcp fragment 1367
2024-11-24 04:51:17 [392359] [info] [Node::send_packet] sent 196 bytes to
127.0.0.1:5000
2024-11-24 04:51:17 [392714] [info] [Node::receive_packet] received 8 bytes from
127.0.0.1:5000
2024-11-24 04:51:17 [392714] [info] [Node::process_tcu_positive_ack] received tcu
positive acknowledgment packet 1367
2024-11-24 04:51:17 [392714] [info] [Node::process_tcu_positive_ack] all packets
successfully sent
2024-11-24 04:51:17 [392359] [info] [Node::wait_for_recv_ack] waiting for tcu receive
acknowledgment
2024-11-24 04:51:17 [392359] [info] [Node::send_file] file transmission completed
```

- Node 2:

```
receiving file...
received file /home/admtrv/recv/2mb.txt
> show log
2024-11-24 04:51:15 [392488] [info] [Node::receive_packet] received 1472 bytes from
127.0.0.1:5001
2024-11-24 04:51:15 [392488] [info] [Node::process_tcu_more_frag_file] received tcu
file packet 1
2024-11-24 04:51:15 [392488] [info] [Node::receive_packet] received 1472 bytes from
127.0.0.1:5001
2024-11-24 04:51:15 [392488] [info] [Node::process_tcu_more_frag_file] received tcu
file packet 2

...

2024-11-24 04:51:17 [392488] [info] [Node::receive_packet] received 196 bytes from
127.0.0.1:5001
```

```
2024-11-24 04:51:17 [392488] [info] [Node::process_tcu_last_frag_file] received tcu
last file packet 1367
2024-11-24 04:51:17 [392488] [info] [Node::send_tcu_positive_ack] send tcu positive
acknowledgment for fragment 1367
2024-11-24 04:51:17 [392488] [info] [Node::send_packet] sent 8 bytes to 127.0.0.1:5001
2024-11-24 04:51:17 [392488] [info] [Node::assemble_file] received file message size
2000000 time 1509
```

- Time: 1509 ms

2. Test with Error Rate (10%):

- Node 1:

```
> set error rate 10
> send file /home/admtrv/2mb.txt
sending file...
complete
> show log
2024-11-24 05:43:14 [420910] [info] [Node::set_packet_loss_rate] set error rate 10
2024-11-24 05:43:14 [420910] [info] [Node::set_window_size] set dynamic window size 273
2024-11-24 05:43:14 [420910] [info] [Node::send_file] sent tcu fragmented file name
2mb.txt size 2000012 fragments 1367 fragment size 1464
2024-11-24 05:43:14 [420910] [info] [Node::send_window] sending window range [1,273]
2024-11-24 05:43:14 [420910] [info] [Node::send_window] sending tcp fragment 1
2024-11-24 05:43:14 [420910] [info] [Node::send_packet] sent 1472 bytes to
127.0.0.1:5000
2024-11-24 05:43:14 [420910] [info] [Node::send_window] sending tcp fragment 2
2024-11-24 05:43:14 [420910] [info] [Node::send_packet] simulated packet corruption
2024-11-24 05:43:14 [420910] [info] [Node::send_packet] sent 1472 bytes to
127.0.0.1:5000

...

2024-11-24 05:43:16 [420910] [info] [Node::send_window] sending tcp fragment 1367
2024-11-24 05:43:16 [420910] [info] [Node::send_packet] sent 196 bytes to
127.0.0.1:5000
2024-11-24 05:43:16 [420910] [info] [Node::wait_for_recv_ack] waiting for tcu receive
acknowledgment
2024-11-24 05:43:16 [421093] [info] [Node::receive_packet] received 8 bytes from
127.0.0.1:5000
2024-11-24 05:43:16 [421093] [info] [Node::process_tcu_positive_ack] received tcu
positive acknowledgment packet 1367
2024-11-24 05:43:16 [421093] [info] [Node::process_tcu_positive_ack] all packets
successfully sent
2024-11-24 05:43:16 [420910] [info] [Node::send_file] file transmission completed
```

- Node 2:

```
receiving file...
received file /home/admtrv/recv/2mb.txt
> show log
2024-11-24 05:43:14 [420991] [info] [Node::receive_packet] received 1472 bytes from
127.0.0.1:5001
2024-11-24 05:43:14 [420991] [info] [Node::process_tcu_more_frag_file] received tcu
file packet 1
```

```
2024-11-24 05:43:14 [420991] [info] [Node::receive_packet] received 1472 bytes from
127.0.0.1:5001
2024-11-24 05:43:14 [420991] [info] [Node::process_tcu_more_frag_file] received tcu
file packet 2
2024-11-24 05:43:14 [420991] [warning] [Node::process_tcu_more_frag_file] invalid
checksum for packet 2

...

2024-11-24 05:43:16 [420991] [info] [Node::receive_packet] received 196 bytes from
127.0.0.1:5001
2024-11-24 05:43:16 [420991] [info] [Node::process_tcu_last_frag_file] received tcu
last file packet 1367
2024-11-24 05:43:16 [420991] [info] [Node::send_tcu_positive_ack] send tcu positive
acknowledgment for fragment 1367
2024-11-24 05:43:16 [420991] [info] [Node::send_packet] sent 8 bytes to 127.0.0.1:5001
2024-11-24 05:43:16 [420991] [info] [Node::assemble_file] received file message size
2000000 time 1766
```

- Time: 1766 ms

3. Test with Packet Loss Rate (10%):

- Node 1:

```
> set packet loss rate 10
> send file /home/admtrv/2mb.txt
sending file...
complete
> show log
2024-11-24 05:43:14 [420910] [info] [Node::set_packet_loss_rate] set error rate 10
2024-11-24 05:52:28 [425506] [info] [Node::set_window_size] set dynamic window size 273
2024-11-24 05:52:28 [425506] [info] [Node::send_file] sent tcu fragmented file name
2mb.txt size 2000012 fragments 1367 fragment size 1464
2024-11-24 05:52:28 [425506] [info] [Node::send_window] sending window range [1,273]
2024-11-24 05:52:28 [425506] [info] [Node::send_window] sending tcp fragment 1
2024-11-24 05:52:28 [425506] [info] [Node::send_packet] simulated packet loss
2024-11-24 05:52:28 [425506] [info] [Node::send_window] sending tcp fragment 2
2024-11-24 05:52:28 [425506] [info] [Node::send_packet] sent 1472 bytes to
127.0.0.1:5000

...

2024-11-24 05:52:30 [425506] [info] [Node::send_window] sending tcp fragment 1367
2024-11-24 05:52:30 [425506] [info] [Node::send_packet] sent 196 bytes to
127.0.0.1:5000
2024-11-24 05:52:30 [425646] [info] [Node::receive_packet] received 8 bytes from
127.0.0.1:5000
2024-11-24 05:52:30 [425646] [info] [Node::process_tcu_positive_ack] received tcu
positive acknowledgment packet 1367
2024-11-24 05:52:30 [425646] [info] [Node::process_tcu_positive_ack] all packets
successfully sent
2024-11-24 05:52:30 [425506] [info] [Node::wait_for_recv_ack] waiting for tcu receive
acknowledgment
2024-11-24 05:52:30 [425506] [info] [Node::send_file] file transmission completed
```

- Node 2:

```
receiving file...
received file /home/admtrv/recv/2mb.txt
> show log
2024-11-24 05:52:28 [425563] [info] [Node::receive_packet] received 1472 bytes from
127.0.0.1:5001
2024-11-24 05:52:28 [425563] [info] [Node::process_tcu_more_frag_file] received tcu
file packet 2

...

2024-11-24 05:52:30 [425563] [info] [Node::receive_packet] received 196 bytes from
127.0.0.1:5001
2024-11-24 05:52:30 [425563] [info] [Node::process_tcu_last_frag_file] received tcu
last file packet 1367
2024-11-24 05:52:30 [425563] [info] [Node::send_tcu_positive_ack] send tcu positive
acknowledgment for fragment 1367
2024-11-24 05:52:30 [425563] [info] [Node::send_packet] sent 8 bytes to 127.0.0.1:5001
2024-11-24 05:52:30 [425563] [info] [Node::assemble_file] received file message size
2000000 time 1806
```

- Time: 1806 ms

4. Test with Window Loss Rate (10%):

- Node 1:

```
> set window loss rate 10
> send file /home/admtrv/2mb.txt
sending file...
complete
> show log
2024-11-24 05:58:38 [425506] [info] [Node::set_window_loss_rate] set window rate 10
2024-11-24 05:58:43 [425506] [info] [Node::set_window_size] set dynamic window size 273
2024-11-24 05:58:43 [425506] [info] [Node::send_file] sent tcu fragmented file name
2mb.txt size 2000012 fragments 1367 fragment size 1464
2024-11-24 05:58:43 [425506] [info] [Node::send_window] sending window range [1,273]
2024-11-24 05:58:43 [425506] [info] [Node::send_window] sending tcp fragment 1
2024-11-24 05:58:43 [425506] [info] [Node::send_packet] sent 1472 bytes to
127.0.0.1:5000
2024-11-24 05:58:43 [425506] [info] [Node::send_window] sending tcp fragment 2
2024-11-24 05:58:43 [425506] [info] [Node::send_packet] sent 1472 bytes to
127.0.0.1:5000

...

2024-11-24 05:58:44 [425646] [info] [Node::process_tcu_positive_ack] received tcu
positive acknowledgment packet 1092
2024-11-24 05:58:44 [425646] [info] [Node::process_tcu_positive_ack] move to next
window starting 1093
2024-11-24 05:58:44 [425506] [info] [Node::send_window] simulated window loss
2024-11-24 05:58:44 [425506] [info] [Node::wait_for_recv_ack] waiting for tcu receive
acknowledgment
2024-11-24 05:59:44 [425506] [info] [Node::wait_for_recv_ack] no tcu receive
acknowledgment, resending window 1/3
2024-11-24 05:59:44 [425506] [info] [Node::send_window] sending window range
[1093,1365]
2024-11-24 05:59:44 [425506] [info] [Node::send_window] sending tcp fragment 1093
```

```
2024-11-24 05:59:44 [425506] [info] [Node::send_packet] sent 1472 bytes to
127.0.0.1:5000

...

2024-11-24 05:59:44 [425506] [info] [Node::send_window] sending tcp fragment 1367
2024-11-24 05:59:44 [425506] [info] [Node::send_packet] sent 196 bytes to
127.0.0.1:5000
2024-11-24 05:59:44 [425646] [info] [Node::receive_packet] received 8 bytes from
127.0.0.1:5000
2024-11-24 05:59:44 [425646] [info] [Node::process_tcu_positive_ack] received tcu
positive acknowledgment packet 1367
2024-11-24 05:59:44 [425646] [info] [Node::process_tcu_positive_ack] all packets
successfully sent
2024-11-24 05:59:44 [425506] [info] [Node::wait_for_recv_ack] waiting for tcu receive
acknowledgment
2024-11-24 05:59:45 [425506] [info] [Node::send_file] file transmission completed
```

- Node 2:

```
receiving file...
received file /home/admtrv/recv/2mb.txt
> show log
2024-11-24 05:58:43 [425563] [info] [Node::receive_packet] received 1472 bytes from
127.0.0.1:5001
2024-11-24 05:58:43 [425563] [info] [Node::process_tcu_more_frag_file] received tcu
file packet 1
2024-11-24 05:58:43 [425563] [info] [Node::receive_packet] received 1472 bytes from
127.0.0.1:5001
2024-11-24 05:58:43 [425563] [info] [Node::process_tcu_more_frag_file] received tcu
file packet 2

...

2024-11-24 05:59:44 [425563] [info] [Node::receive_packet] received 196 bytes from
127.0.0.1:5001
2024-11-24 05:59:44 [425563] [info] [Node::process_tcu_last_frag_file] received tcu
last file packet 1367
2024-11-24 05:59:44 [425563] [info] [Node::send_tcu_positive_ack] send tcu positive
acknowledgment for fragment 1367
2024-11-24 05:59:44 [425563] [info] [Node::send_packet] sent 8 bytes to 127.0.0.1:5001
2024-11-24 05:59:44 [425563] [info] [Node::assemble_file] received file message size
2000000 time 61781
```

- Time: 61781 ms

After each transmission, the integrity of the transmitted and received files was verified. All tests confirmed that the files were identical in size and content. The following commands were used for verification:

```
cmp ~/2mb.txt ~/recv/2mb.txt    # no output indicates identical files
diff ~/2mb.txt ~/recv/2mb.txt   # no output indicates no differences
md5sum ~/2mb.txt ~/recv/2mb.txt # identical hash values
074e55b1e57365b7333f94f0d17a9994  /home/admtrv/2mb.txt
074e55b1e57365b7333f94f0d17a9994  /home/admtrv/recv/2mb.txt
ls -l ~/2mb.txt ~/recv/2mb.txt  # identical file size of 2 MB
```

```
-rw-r--r-- 1 admtrv admtrv 2000000 Nov 24 04:49 /home/admtrv/2mb.txt
-rw-r--r-- 1 admtrv admtrv 2000000 Nov 24 05:59 /home/admtrv/recv/2mb.txt
```

4.3.5. Keep-Alive

The Keep-Alive mechanism in the communication of nodes ensures that a connection remains active by periodically checking whether both nodes are still connected.

Connection Initialization:

1. When connection established, node starts keep-alive loop that monitors connection's activity.
2. After the last activity from destination node, loop sleeps for predefined timeout interval (`'TIMEOUT_INTERVAL = 300'`).

Activity Check:

1. After the timeout without activity expires, node will send KA (Keep-Alive) Request messages in the number of (`'ATTEMPT_COUNT = 3'`) and with the interval (`'ATTEMPT_INTERVAL = 5'`) between.
2. In reply to this message, receiver must send KA (Keep-Alive) + ACK (Acknowledgment) message to confirm his activity.

```
   TIMEOUT...

   NETWORK A                    B NETWORK
         1 A ------ (KA) -----> B
           A <-- (KA + ACK) --- B
   NETWORK A                    B NETWORK

   TIMEOUT...
```

3. After receiving acknowledgement, sender sleeps again until the next similar check.
4. If sender does not receive acknowledgement, it will force connection to close.

```
   TIMEOUT...

   NETWORK A              B DEAD

         1 A --- (KA) --> B
         2 A --- (KA) --> B
         3 A --- (KA) --> B

   HOLDOFF A              B DEAD
```

Implementation:

- The Keep-Alive functionality is managed by a dedicated thread:

```
/*
 *  node.h
 */

class Node {
public:
    /* Thread methods */
    void start_keep_alive();
    void stop_keep_alive();
```

```
private:
    /* Keep-Alive thread params */
    void keep_alive_loop();
    std::atomic<bool> _keep_alive_running{false};
    std::thread _keep_alive_thread;
};
```

- This thread is separated from the Application + CLI flow in order not to cause freezes and delays in program responsiveness and runs the keep-alive loop with handling activity monitoring:

```
/*
 *  node.cpp
 */

void Node::keep_alive_loop()
{
    while (_keep_alive_running)
    {
        bool ack_received = false;

        std::sleep_for(std::seconds(TCU_ACTIVITY_TIMEOUT_INTERVAL));

        for (int i = 0; i < TCU_ACTIVITY_ATTEMPT_COUNT && _keep_alive_running; i++)
        {
            send_keep_alive_req();

            std::sleep_for(std::seconds(TCU_ACTIVITY_ATTEMPT_INTERVAL));

            if (_pcb.is_activity_recent())
            {
                ack_received = true;
                break;
            }
        }

        if (!ack_received)
        {
            _pcb.new_phase(TCU_PHASE_HOLDOFF);
            std::cout << "destination node down, connection closed" << std::endl;
        }
    }
}
```

4.4. Code Realization

The TCU (Transmission Control over UDP) protocol is implemented in the `'protocols/tcu.h'` file and consists of the following main components:

```
/*
 * tcu.h
 */

#define TCU_PHASE_DEAD          0
#define TCU_PHASE_HOLDOFF       1
#define TCU_PHASE_INITIALIZE    2
```

```
#define TCU_PHASE_CONNECT       3
#define TCU_PHASE_NETWORK       4
#define TCU_PHASE_DISCONNECT    5
#define TCU_PHASE_CLOSED        6

#define TCU_HDR_NO_FLAG         0x00
#define TCU_HDR_FLAG_SYN        0x01
#define TCU_HDR_FLAG_ACK        0x02
#define TCU_HDR_FLAG_FIN        0x04
#define TCU_HDR_FLAG_NACK       0x08
#define TCU_HDR_FLAG_DF         0x10
#define TCU_HDR_FLAG_MF         0x20
#define TCU_HDR_FLAG_FL         0x40
#define TCU_HDR_FLAG_KA         0x80

#define ETH2_MAX_PAYLOAD_LEN    1500
#define IPV4_HDR_LEN            20
#define UDP_HDR_LEN             8
#define TCU_HDR_LEN             8
#define TCU_MAX_PAYLOAD_LEN     (ETH2_MAX_PAYLOAD_LEN - IPV4_HDR_LEN - UDP_HDR_LEN -
TCU_HDR_LEN)

#define TCU_ACTIVITY_TIMEOUT_INTERVAL   300
#define TCU_ACTIVITY_ATTEMPT_COUNT      3
#define TCU_ACTIVITY_ATTEMPT_INTERVAL   5

#define TCU_CONFIRM_TIMEOUT_INTERVAL    5
#define TCU_RECEIVE_TIMEOUT_INTERVAL    60

struct tcu_header {
    uint24_t seq_number;        // Sequence packet number
    uint8_t flags;              // Flags
    uint16_t length;            // Payload length
    uint16_t checksum;          // CRC sum
};

struct tcu_packet {
    tcu_header header{};        // Header
    unsigned char* payload;     // Payload

    unsigned char* to_buff();
    tcu_packet from_buff(unsigned char* buff);

    void calculate_crc();
    bool validate_crc() ;
};

uint16_t calculate_crc16(const unsigned char* data, size_t length);

struct tcu_pcb {
    /* Connection params */
    void new_phase(int phase);
    uint8_t phase = TCU_PHASE_DEAD;

    /* Source node params */
    uint16_t src_port;
```

```
    /* Destination node params */
    uint16_t dest_port;
    in_addr dest_ip;
    struct sockaddr_in dest_addr;
};
```

4.4.1. Payload Size

The TCU protocol calculates the maximum payload size based on the constraints imposed by the Ethernet frame size and other protocol headers. Here's how it is determined:

Ethernet Payload Size:

- The maximum Ethernet payload is 1500 bytes (`ETH2_MAX_PAYLOAD_LEN`).

Protocol Header Sizes:

- IPv4 header size: 20 bytes (`IPV4_HDR_LEN`).
- UDP header size: 8 bytes (`UDP_HDR_LEN`).
- TCU header size: 8 bytes (`TCU_HDR_LEN`).

Resulting Payload Size:

- The maximum payload size for a TCU packet is:

```
/*
 * tcu.h
 */

#define TCU_MAX_PAYLOAD_LEN (ETH2_MAX_PAYLOAD_LEN - IPV4_HDR_LEN - UDP_HDR_LEN -
TCU_HDR_LEN)
```

- As a result, the maximum payload size of a TCU single packet is 1464 bytes.

Note: This calculation is necessary to avoid additional fragmentation at the link layer, ensuring that each TCU packet fits within the Ethernet frame size.

4.4.2. Sequence Number

The `uint24_t` type was introduced in the `'types/uint24_t.h'` file to efficiently handle sequence numbers in the TCU protocol:

```
/*
 * uint24_t.h
 */

class uint24_t {
private:
    uint32_t value : 24;

public:
    /* Overriding arithmetics */
    uint24_t& operator+=(uint32_t v);
    uint24_t& operator-=(uint32_t v);
    ...
};

static_assert(sizeof(uint24_t) == 3, "size of uint24_t not 3 bytes");

/* Convert to network byte order */
```

```
uint24_t hton24(uint24_t host24);
uint24_t ntoh24(uint24_t net24);
```

Sequence Number Requirements:
- A sequence number must uniquely identify every fragment.
- For a 2 MB file split into 1-byte fragments, 2 million unique values are needed.

Standard Data Types:
- `uint16_t` (2 bytes): Can represent up to 65,535 unique values — not enough for 2 MB.
- `uint32_t` (4 bytes): Can represent over 4 billion values — sufficient but introduces unnecessary overhead.

Optimization:
- A custom `uint24_t` type uses 3 bytes, which is sufficient for up to ~16 million unique values, reducing header size while meeting protocol requirements.

Usability:
- The `uint24_t` implementation includes overloaded operators for arithmetic, comparison, and conversion.
- This allows it to be used in the same way as a standard integer type.
- Appears as include header library, so it can easily be ported to other projects.

```
#include "../types/uint24_t.h"

for (uint24_t i = _seq_num; i <= _last_num; i++)
{
  ...
}
```

4.4.3. Flags

The TCU protocol uses 8 flags stored in a `uint8_t` field, making the header compact and efficient.

Structure:
1. The `uipnt8_t` field provides exactly 8 bits, allowing for the perfect representation of all flags without wasted space.
2. The flags have clear, descriptive meanings, making them easy to understand and all their use is tied to their meaning. Examples:
   - SYN + ACK for Connection Acknowledgment
   - KA + ACK for responding to Keep-Alive Request
   - MF + FL for Fragment of File

```
case (TCU_HDR_FLAG_MF | TCU_HDR_FLAG_FL):
    process_tcu_more_frag_file(packet);
    break;
```

4.4.4. Protocol Control Block

The Protocol Control Block (TCU PCB) structure manages the state and configuration of a TCU connection:

Connection Management:
- Tracks the current phase of the connection using phase, such as CONNECT or DISCONNECT.
- The `'tcu_pcb::new_phase(int new_phase)'` method updates the phase of the connection and ensures clear phase transition.

Node Parameters:

- Source node (`src_port`) and destination node (`dest_ip`, `dest_port`) details are stored for communication.
- The PCB centralizes all protocol-related information, simplifying the implementation of connection state management.

```
/*
 * node.cpp
 */

void Node::process_tcu_conn_req(tcu_packet packet)
{
    if (_pcb.phase <= TCU_PHASE_INITIALIZE)
    {
        _pcb.new_phase(TCU_PHASE_CONNECT);
        send_tcu_conn_ack();
    }
    else
    {
        spdlog::error("[Node::process_tcu_conn_req] unexpected phase {}", _pcb.phase);
        exit(EXIT_FAILURE);
    }
}
```

5. Node

5.1. Overview

The Node serves as the central unit in the application, acting both as the sender and receiver in the peer-to-peer connection. It encapsulates the protocol logic and implements the seamless connectivity. Through interaction with the user via the Command-Line Interface (CLI), the Node provides configurability, allowing manual adjustments of key parameters.

5.2. Code Realization

The Node class is implemented in the `'entities/node.h'` file and consists of the following components:

```
/*
 *  node.h
 */

class Node {
public:
    /* Abstract methods */
    void send_packet(unsigned char* buff, size_t length, bool service);
    void receive_packet();

    /* Concrete methods */
    void send_text(const std::string& message);
    void send_file(const std::string& path);
    void send_window();

    /* Process information methods */
    void assemble_text();
    void assemble_file();
    void save_file(const File& file);
```

```
    /* FSM methods */
    void fsm_process(unsigned char* buff, size_t length);

    /* Processing */
    void process_tcu_conn_req(tcu_packet packet);
    void process_tcu_conn_ack(tcu_packet packet);
    ...

    /* Sending */
    void send_tcu_conn_req();
    void send_tcu_conn_ack();
    ...

private:
    /* Socket control block */
    Socket _socket;

    /* TCU protocol control block */
    tcu_pcb _pcb;

    /* Sending params */
    std::map<uint24_t, tcu_packet> _send_packets;
    uint24_t _seq_num;
    uint24_t _last_num;
    uint24_t _total_num;
    uint24_t _window_size;

    /* Receiving params */
    std::map<uint24_t, tcu_packet> _received_packets;

    /* File saving params*/
    std::string _file_path;
};
```

5.2.1. Receiving

- The receiving thread manages incoming packets in a separate execution context, ensuring that packet reception does not block other operations:

```
/*
 *  node.h
 */

class Node {
public:
    /* Thread methods */
    void start_receiving();
    void stop_receiving();

private:
    /* Receiving thread params */
    void receive_loop();
    std::atomic<bool> _receive_running{false};
    std::thread _receive_thread;
};
```

- The `'Node::receive_packet()'` abstract method processes incoming data packets. It uses a non-blocking approach with the select system call to monitor the socket and reads incoming data only when it's available:

```
/*
 *  node.cpp
 */

void Node::receive_packet()
{
    temp_buff[2048];

    fd_set read_fds;
    FD_ZERO(&read_fds);
    FD_SET(_socket.get_socket(), &read_fds);

    struct timeval timeout{0, 50000};

    if (select(_socket.get_socket() + 1, &read_fds, nullptr, nullptr, &timeout) > 0 &&
FD_ISSET(_socket.get_socket(), &read_fds))
    {
        struct sockaddr_in src_addr{};
        socklen_t src_addr_len = sizeof(src_addr);

        ssize_t num_bytes = recvfrom(_socket.get_socket(), temp_buff,
sizeof(temp_buff), 0, (struct sockaddr*)&src_addr, &src_addr_len);
        if (num_bytes > 0)
        {
            _pcb.update_last_activity();
            fsm_process((temp_buff, num_bytes);
        }
    }
}
```

5.2.2. Sending

1. The abstract method, which is primary in the transfer logic, is responsible for sending the packet through the socket:

```
/*
 *  node.cpp
 */

void Node::send_packet(unsigned char* buff, size_t length)
{
    ssize_t num_bytes = sendto(_socket.get_socket(), buff, length, 0, &_pcb.dest_addr,
sizeof(_pcb.dest_addr));

    spdlog::info("[Node::send_packet] sent {} bytes to {}:{}", num_bytes,
inet_ntoa(_pcb.dest_addr.sin_addr), ntohs(_pcb.dest_addr.sin_port));
}
```

2. Other methods, implement it by using its functionality for concrete tasks:
   - `'Node::send_text(const std::string& message)'`
   - `'Node::send_file(const std::string& path)'`
   - `'Node::send_window();'`

5.2.3. Process Information

The Node class includes utility methods for assembling and saving data during file or message transmission.

Text Assembling

- `'Node::assemble_text()'`
- Combines fragments of received text messages in the correct order into a complete message.
- Displays received message to the user.

```
> receiving text...
> received text Hello Peer!
```

File Assembling

- `'Node::assemble_file()'`
- Same as the previous method, combines fragments of received file messages in the correct order into a complete message.
- Passes the received file as input to the `'Node::save_file(File& file)'` method to properly save the file to the correct location.

File Saving

- `'Node::save_file(File& file)'`
- Handles file storage after successful reconstruction.
- Extracts file metadata (name, size) and saves the data to a user selected directory.

```
> receiving file...
> received file /home/admtrv/recv/2mb.txt
```

File Entity

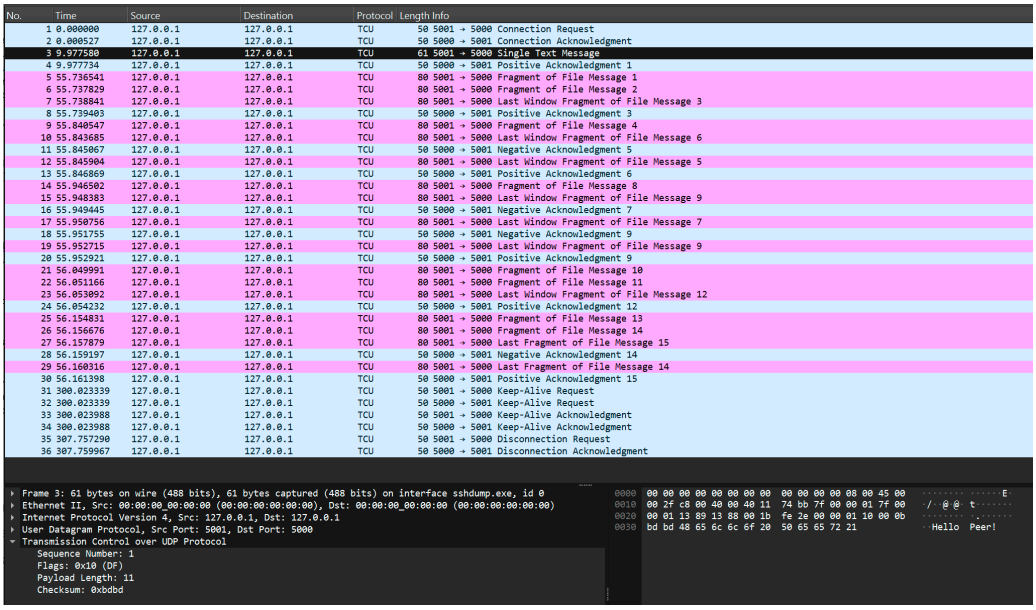
The File class, implemented in `'entities/file.h'`, is designed for efficient file transmission and reconstruction. It combines file metadata with content, enabling optimized payload usage.

```
/*
 * file.h
 */

#define FILE_NAME_MAX_LEN 255

struct FileHeader {
    uint8_t name_length;                // File name length
    char file_name[FILE_NAME_MAX_LEN];  // File name
    uint32_t file_size;                 // File size
};

class File {
public:
    unsigned char* to_buff();
    File from_buff(unsigned char* buff);

private:
    FileHeader header;      // File header
    unsigned char* data;    // File data
};
```

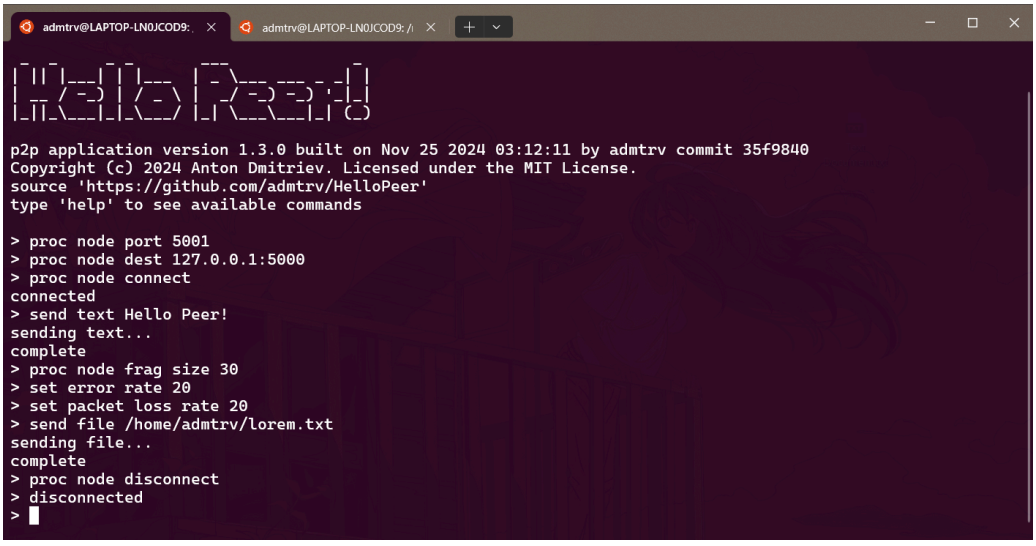
- Has implemented methods for buffering data so that this data can then be put into the payload buffer of the TCU packet.
- This structure avoids the inefficient transfer of a filename in one separate packet.
- Optimizes packet space for both metadata and content.

6. Script

- These figures show the **Lua Script** and **Coloring Rule** work in **Wireshark** to recognize TCU protocol packets in complex conditions:



- From this example, we can see:
  - Packages 1-2: Connection Establishment
  - Packets 3-4: Send Single Text Message; also detailed overview of Packet 3
  - Packets 5-30: Sending Fragmented File Message; where:
    1. Packets 5-8: Successfully Sent and Confirmed whole Window [1,3]
    2. Packets 9-13: Successfully Resent of Lost Fragment 5 for Window [4,6]
    3. Packets 25-30: Successfully Resent of Corrupted Fragment 14 for Last Window [13,15]
  - Packets 31-34: Keep-Alive Mechanism
  - Packets 35-36: Connection Termination
- This example, but on behalf of the User:



- Lua Script and Coloring Rule are located in the `'wireshark'` folder under the names `'tcu.lua'` and `'tcu_coloring_rule'`.
- More examples can be found in the `'wireshark/tcu_example.pcapng'`file with the appropriate files `'pc1.png'` and `'pc2.png'`.

7. Changes

During the implementation phase, several significant adjustments were made to the original protocol design:

Payload Size Calculation:

Initially, the payload size was calculated based on the assumption that the maximum size of a UDP datagram is 65,535 bytes. However, this approach led to potential additional fragmentation at the link layer, as the Ethernet frame size typically supports a maximum payload of 1,500 bytes. To resolve this issue, the payload size calculation was revised to consider the constraints imposed by the Ethernet frame and protocol headers. As a result, the maximum payload size was set to 1,464 bytes, avoiding link-layer fragmentation.

Sequence Number Representation:

The initial design used a 2-byte field (16 bits) for the sequence number, allowing for a maximum of 65,535 unique values. This was insufficient for scenarios involving large files, such as a 2 MB file split into 1-byte fragments, which requires over 2 million unique sequence numbers. Using 4 bytes (32 bits) for the sequence number would have been excessive for the protocol's requirements, as it provides over 4 billion unique values, far more than necessary. To address this limitation while avoiding unnecessary overhead, a custom 3-byte (24-bit) data type, `uint24_t`, was introduced. This change increased the sequence number capacity to over 16 million, ensuring compatibility with larger files and maintaining header compactness.

Flags Representation:

Initially, 2 bytes (16 bits) were used by me for flags in the protocol header. This was excessive for the protocol's 8 flags, which could be effectively represented using just 1 byte (8 bits). The header design was revised to use a single byte for flags, reducing overhead and creating more space for payload data.

8. Conclusion

The purpose of my realization of this project was to create a really compact but complete and meaningful protocol that meets all the requirements of reliable data transmission. To minimize redundancy in the header, I even had to implement my own `uint24_t` variable type, allowing me to efficiently use 3 bytes for the sequence number instead of the standard 4 bytes, so that more data can now potentially fit into the payload. The protocol is also designed with a focus not only on compactness but also on simplicity and understandability, including a clear system of flags and phases.

The transmitter implements key data transmission techniques such as fragmentation, integrity checking, resending corrupted data, and connection maintenance. Additionally, I focused on designing an intuitive and user-friendly CLI interface, making it easy to interact with the transmitter and utilize its features effectively. The final solution demonstrates stability and high performance under various transmission conditions, including simulated errors.

Bibliography

[1] Defense Advanced Research Projects Agency and Information Processing Techniques Office, "Transmission Control Protocol: DARPA Internet Program Protocol Specification," RFC 793, 1400 Wilson Boulevard, Arlington, Virginia 22209, Sep. 1981. [Online]. Available: https://www.ietf.org/rfc/rfc793.txt

[2] J. Postel, “User Datagram Protocol,” RFC 768, Aug. 1980. [Online]. Available: https://www.ietf.org/rfc/rfc768.txt

[3] J. Geluso, “CRC16-CCITT.” [Online]. Available: https://srecord.sourceforge.net/crc16-ccitt.html

[4] Andrew S. Tanenbaum and David J. Wetherall, *Computer Networks*, 5th ed. Boston: Pearson Prentice Hall, 2011.

[5] GeeksforGeeks, “Sliding Window Protocol: Set (3 Selective Repeat).” [Online]. Available: https://www.geeksforgeeks.org/sliding-window-protocol-set-3-selective-repeat/

[6] The Open Group, “POSIX Threads (pthread.h) Documentation.” The Single UNIX Specification, Version 2. [Online]. Available: https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html

[7] C. Ramey, “The GNU Readline Library Documentation.” [Online]. Available: https://tiswww.case.edu/php/chet/readline/rltop.html

[8] G. Melman, “spdlog: Fast C++ logging library.” GitHub Repository. [Online]. Available: https://github.com/gabime/spdlog

[9] “Wireshark Lua Script Tutorial.” Wireshark, 2024. [Online]. Available: https://wiki.wireshark.org/Lua/Examples