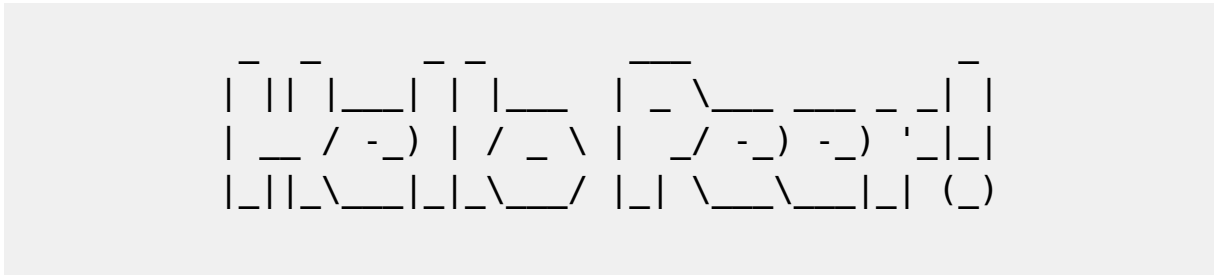


Peer-to-Peer Application
Project Documentation

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 file saving directory path

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

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 1.0.0 built on Oct 20 2024 18:18:48 by admtrv commit 322b8f0
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 (0,65499)
  proc node connect           - connect to destination node
  proc node disconnect        - disconnect with destination node
  send text <text>            - send text message to destination node
  set log level <level>       - set log level (trace, debug, info, warn, error,
critical)
  show log                    - display current logs
  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.

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.

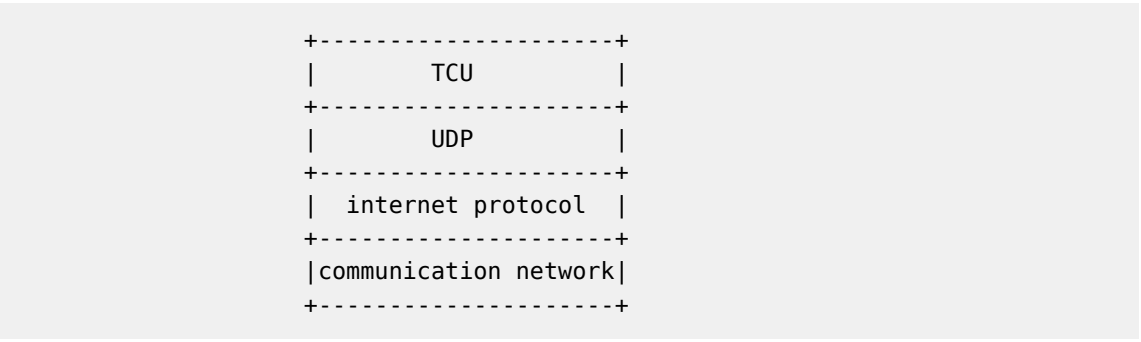
4. Protocol

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

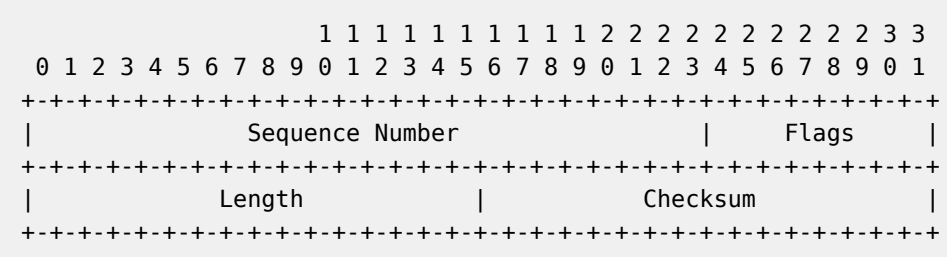
The TCU protocol is designed to provide a simpler and lightweight alternative to the Transmission Control Protocol (TCP) for application that do not require the full complexity of TCP. It is well-suited for tasks like exchanging text and files. By building on top of the User Datagram Protocol (UDP), 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 a 16 MB file in 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

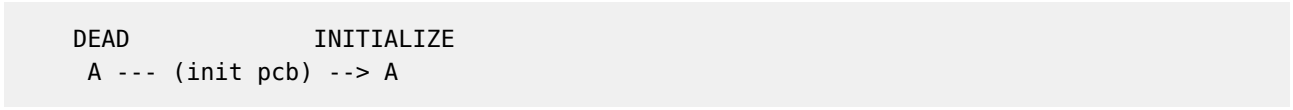
A connection progresses through a series of states during its lifetime. Briefly the meanings of the states are:

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

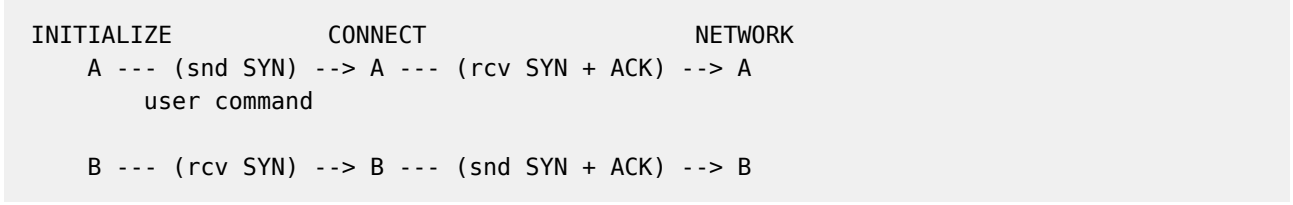
A TCU connection progresses from one state to another in response to events. The events are the user calls, incoming segments that containing flags; and timeouts.

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

Inicialization:



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



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
```

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:

```
void tcu_packet::calculate_crc16(buffer)
{
  ...
}
```

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 is it inserted back into the packet's header.

```
void tcu_packet::calculate_crc()
{
    buffer[];

    memcpy(buffer, header - checksum);
    memcpy(buffer, payload);

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

```
bool tcu_packet::validate_crc()
{
    buffer[];

    memcpy(buffer, header - checksum);
    memcpy(buffer, payload);

    computed_crc = calculate_crc16(buffer);

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

4.3.4. Automatic Repeat Request

The TCU protocol ensures reliable data transmission through the **Selective Repeat (SR)** 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:
   - All fragments: MF (More Fragments)
   - Last fragment: MF (More Fragments) flag removed

Window Management:

1. Receiver maintains window to store received fragments.
2. Receiver processes each fragment as it arrives, verifying sequence and storing it in the correct place.
3. Fragments can arrive out of order, so they are placed in the window based on their sequence number.

Error Detection:

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

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 message.

```
1. TEXT    A                B
   [1,2,3] A                B [ , , ]

2. [1,2,3] A --- [MF,1] ---> B [1, , ]
   [1,2,3] A --- [MF,2] ---> B [1,2, ]
   [1,2,3] A --- [  ,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 --- [  ,2] ---> B [1,2,3] NO ERROR
   [1,2,3] A <--- [ACK,0] -- B [1,2,3]
           A                B    TEXT
```

4.3.5. Keep-Alive

The Keep-Alive mechanism in the TCU protocol 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. If sender does not receive acknowledgement, it will force connection to close.

```
NETWORK A              B DEAD

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

HOLDOFF A              B DEAD
```