```
 _  _     _ _       ___           _
| || |___| | |___  | _ \___ ___ _ _| |
| __ / -_) | / _ \ |  _/ -_) -_) '_|_|
|_||_\___|_|_\___/ |_| \___\___|_| (_)
```

Peer-to-Peer Application
Project Documentation

Anton Dmitriev
Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava

Contents

1. Protocol

1.1. 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.

1.2. 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.

```
+---------------------+
|        TCU          |
+---------------------+
|        UDP          |
+---------------------+
|  internet protocol  |
+---------------------+
|communication network|
+---------------------+
```

1.3. 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.

1.3.1. Header Format

```
                     1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            Length             |             Flags             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       Sequence Number         |           Checksum            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Length: 16 bits

- Length of the payload in bytes
- Since the UDP payload is limited to 65507 bytes, 16-bit field is sufficient
- This value includes only the payload size, not the header size

Flags: 16 bits (from left to right)

- Control flags that are used to indicate the packet's state
  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

Sequence Number: 16 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

Checksum: 16 bits
- Checksum used to verify the integrity of the packet, including the header and payload

1.3.2. 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:

```
   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
```

1.3.3. 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_crc()
{
  ...
}
```

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;
}
```

1.3.4. 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
```

1.3.5. 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
```