Link Layer Emulator

In this assignment we will develop a link layer emulator for a fictitious network called DCCNET. The emulator will handle framing, sequencing, error detection and data retransmission.

Framing

The format of a DCCNET frame is shown in Diagram 1. The sequence used for synchronization between end points (`SYNC` , 32 bits) is always `0xDCC023C2` and shows up two times at the beginning of each frame. The error detection field (`chksum` , 16 bits) uses the Internet checksum algorithm for error detection. The size field (`length` , 16 bits) counts the number of bytes of data transmitted in the frame; the `length` field does *not* count frame header bytes. Although the `length` field is 16 bits long, the biggest payload a DCCNET frame can carry is 4096 bytes. The `ID` field (16 bits) identifies the transmitted or acknowledged frame. The `length` and `ID` fields must be transmitted using network byte order (big-endian).

```
DIAGRAM 1: DCCNET Frame Format

0        32       64        80        96        112   120
112+length
+---/----+---/----+---------+---------+---------+-----+------ ... ---+
|  SYNC  |  SYNC  | chksum  | length  | ID      |flags| DATA         |
+---/----+---/----+---------+---------+---------+-----+------ ... ---+
```

DCCNET Frame Types

A frame is classified into one of four categories, as defined by different combinations of `flags` set in the frame header:

- A *data* frame does not have the `ACK` flag set. Data frames usually have `length` larger than zero. A data frame can be marked as the end-of-transmission error by having the `END` flag set (see "Termination" below).
- An *acknowledgement* frame has the `ACK` flag set, and cannot carry any data (it must have a `length` of zero). An acknowledgement frame cannot have the `END` flag set.

- A *reset* frame has the `RST` flag set and a frame ID set to `0xFFFF`. A reset frame preceeds the termination of a connection and can be sent as a "last notification" to the remote endpoing of the error condition that triggered the connection reset.

Diagram 2 describes the meaning of each flag:

```
DIAGRAM 2: Flag control bit meanings

Bitmask (hex)    | Bit | Name | Function
1000 0000 (0x80) |  7  | ACK  | Data reception acknowledgement
0100 0000 (0x40) |  6  | END  | End-of-transmission bit
0010 0000 (0x20) |  5  | RST  | Reset connection due to unrecoverable error.
0001 1111 (0x3f) | 4-0 | ---  | Reserved. Should be set to zero.
```

Sequencing

We will emulate the DCCNET link layer using the stop-and-wait flow control algorithm, that is, both the transmission and reception windows will hold one frame only. Consequently, the values of the `ID` field are restricted to zero or one. Below we detail the functioning of a node to transmit and receive data. Note that, as DCCNET links are full-duplex (data can be transmitted in both directions), each node functions as a transmitter and receiver simultaneously.

Transmitter

To transmit data, the transmitting node creates a frame as specified in Diagram 1. Although the `length` field is 16-bits long, the maximum payload length in DCCNET is 4096 bytes.

The `ID` field contains the frame identifier. While the transmitted frame is not confirmed, it must be retransmitted periodically every second. Frame retransmissions must be retried at least 16 times. Upon receipt of the acknowledgement, the transmitter must change the value of the `ID` field (from zero to one, or from one to zero) before transmitting the next frame. The first transmitted frame must have `ID` zero.

Receiver

A frame can only be accepted if it is an `ack` nowledgement frame for the last transmitted frame; a data frame with an identifier (`ID`) different from that of the last received frame; a retransmission of the last received frame; or a reset frame.

Upon accepting a data frame (cases 2 and 3 above), the receiver *must* respond with an acknowledgement frame. Upon receiving a reset frame, the endpoing must shut down the connection.

The receiver must keep in memory the identifier (`ID`) and the checksum (`chksum`) of the last received data frame. If one received frame is a retransmission of the last received frame, the receiver must re-send the acknowledgement frame. A frame is considered a retransmission of the last received frame if both frames are identical.

Error Detection

Transmission errors are detected using the checksum present in the frame header. The checksum is the same as that used on the Internet and is calculated over the entire frame, including header and data. During checksum calculation, the header bits reserved for the checksum must be considered as zero. Frames with errors must not be accepted by the destination nor confirmed.

To detect the beginning of each frame, your emulator must wait for two occurrences of the synchronization pattern (`SYNC`) that mark the beginning of a frame. Detecting the beginning of each frame is important and more challenging after a transmission error, particularly if the transmission error corrupts the `length` field. When the `length` field is corrupted, your emulator will lose track of where the current frame ends and the next frame starts. Your emulator should continuously wait for two occurrences of the synchronization pattern (`SYNC`) and attempt to receive frames to recover from errors.

End of Transmission

When the transmitter sends the last data frame it must set the `END` bit of the `flags` field to indicate that there will be no more data frames after this frame, or send a new empty data frame with the `END` flag on. The receiver must acknowledge the frame that has the `END` flag set normally, and keep the information that the other side no longer has data to transmit. When a process receives and acknowledges the last data frame with the `END` flag set and has nothing else to transmit, it must close the connection and terminate execution.

Note that a DCCNET endpoint that has no more data to transmit and signals this by setting the `END` flag *can* still receive data normally. Note also that only the last frame of data must have the `END` flag set; in particular, acknowledgement frames (with `length` zero and the `ACK` flag set) should never have the `END` flag set.

Unrecoverable Errors

When a side of the transmission enters an unrecoverable error state where transmission cannot continue, it should send frame with the `RST` flag set. As the connection is in an unrecoverable state, a `RST` frame should set all bits of the `ID` field to 1 (i.e., it should be 65535). The `chksum` field should be computed as for any other frame, but no retransmissions are performed. The side sending a `RST` should close the underlying TCP connection (see below) after sending the `RST` frame. Finally, a `RST` frame may optionally contain a payload to inform the other side what error condition was encountered.

Implementation Details

You must implement DCCNET over a TCP connection. Your DCCNET emulator must be able to function as transmitter and receiver simultaneously. Your emulator must interoperate with other emulators (test with colleagues' emulators), including the reference emulator implemented by the instructor.

Grading

This assignment can be implemented in Python, C, C++, Rust, or Java but must interoperate with emulators written in other languages.

Any inconsistency or ambiguity in the specification must be reported to the instructor; depending on the inconsistency or ambiguity, the student who first report it will receive extra credit.

Submit your documentation in PDF format with up to 4 pages (two pages), using font size 10. Documentation should discuss project challenges, difficulties and unforeseen events, as well as the solutions adopted for problems. In particular, your documentation should:

- Discuss the mechanism implemented to recover framing after errors;
- Discuss how the implementation handles transmission and reception in parallel;
- Discuss how applications interface with the DCCNET implementation.

Grading Application 1

To test your implementation of DCCNET, you should also create an application that will exchange data with the grading server.

0. All data transmitted back-and-fourth between the client and the server will be ASCII-encoded strings. Each message will be terminated by a newline `\n` character. (Note that DCCNET frames may carry part of a message, one message, or multiple messages.)

1. Your app should establish a DCCNET connection with the grading server. The server's name and port will be posted on the course website.

2. Once establishing a connection, your program should start by sending a message containing the your group's authentication sequence (GAS), as defined in assignment 1. (Note that your authentication message should be terminated by a `\n` character.)

3. After authentication, the server will start transmitting lines of text (terminated with `\n`) to your program. Your program should then compute MD5 checksums for each line of ASCII text received from the server. More precisely, the server will continuously send `\n`-terminated lines of ASCII text to your program. After each line of text is received, your program should compute the MD5 checksum of the line of text (*not* including the `\n` byte). After computing the MD5 checksum, your program should transmit it to the server as a hexadecimal string (containing only the characters `a-f` and `0-9`, as printed by Python's hexdigest function); transmit one newline character (`\n`) after each MD5 checksum.

4. The program should terminate after all lines are received (use the `END` flag for identifying when the server's transmission completes) and the MD5 checksums acknowledged.

> Note that messages (lines) are always terminated by the newline characters. Messages (lines) can be split across multiple frames, and a single frame can contain multiple messages (lines).

User Interface

Your MD5 application using DCCNET must be executed as follows:

```
./dccnet-md5 <IP>:<PORT> <GAS>
```

Where the `IP:PORT` are the IP and port number provided in the course website for the automated grading server, and `<GAS>` is the group authentication sequence to complete authentication.

Grading Application 2

As another test, you should also create a bidirectional file transfer application that will send data from a file to the remote endpoint, and save all data received from the remote endpoint into a file. When the files finish transmission in both directions, the program should terminate.

User Interface

The file transfer application should be executable as a passive server (wait for connections) or active client (establish a connection). The passive server should be executed as follows:

```
./dccnet-xfer -s <PORT> <INPUT> <OUTPUT>
```

Where `PORT` specified the port where the server will listen for a connection. All data exchanged through this port must follow the DCCNET protocol. The program must wait for connections on all IPs of the machine it is running on (including IPv6 IPs). `INPUT` is the name of a file with the data that will be sent to the remote end of the link, and `OUTPUT` is the name of a file where the data received from the remote end of the link will be stored.

The active client should be executed as follows:

```
./dccnet-xfer -c <IP>:<PORT> <INPUT> <OUTPUT>
```

Where the `IP` parameter is the IP address of the machine where the passive server is running. The other parameters have the same meaning as above. (Note that `IP` can be `127.0.0.1` if the programs are running on the same machine, but it can also be the address of another machine accessible over the network, possibly an IPv6 address.)

Tips

Below is an example of a DCCNET frame transmitting four bytes with values 1, 2, 3 and 4 (in this order). The following bytes, in hexadecimal notation, will have to be transmitted at the link layer. This example assumes that the frame identifier (`ID`) is zero and that more data needs to be transmitted (so the `END` flag is not set):

```
dc  c0  23  c2  dc  c0  23  c2  f8  f1  00  04  00  00  00  01  02  03  04
SYNC------------------------  chksum  length  id----  flg payload-------
```

A Note on the Tests with Transmission Errors

The version of the program to be submitted must work correctly even if there is an error in any of the received frames. As the programs will run over the TCP protocol, there will never be wrong bytes upon arrival, as the TCP connection already performs error checking. Bytes received with `recv` will match exactly to the bytes sent with `send` . However, in testing, we can use programs that deliberately, purposefully generate and `send` corrupted frames, causing the remote end's `recv` to return corrupted frames and triggering the code to detect synchronization errors, compute checksums, etc.

To test your program in these cases, you can create modified versions of the program that intentionally generate and send corrupted frames to see how the remote program behaves. It is not necessary to hand over these programs used for testing, only the programs that works according to this specification. The instructor has a test program that can inject these errors in the connection during testing. The grading server will also generate corrupted frames to exercise during the MD5 checksum tests.