Implementation of Data Link Protocol

Tiago A., Jakub J.

Imagine you downloaded and now need to read this report, but h&?ds cm!))sp mbs㊎ yaosp ^*& ps%#%. It is why reliable and efficient data transmission in today's digitalized world is crucial. This project aimed to implement basic Stop & Wait data link protocol, where the link layer ensures that data packets are transferred correctly over the physical layer and the application layer enables end-users to easily access network service.

This project demonstrated an implementation of a file transfer program using a protocol similar to HDLC to achieve reliable communication in noise prone environments.

I. Introduction

Main objective of the work was to familiarize the authors with the concepts and practical solutions appearing in the computer networks. The second objective was to implement a usable data link protocol for two machines which know that something is to be transferred from one to the other.

This report aims to present our work. It includes a presentation of the architecture of the work as well as a structure of the code. It identifies main use cases. Explains logic behind link layer and application layer. Validates our work through presentation of the tests. Shows our data link protocol efficiency and reflects on the achieved learning objectives.

II. Architecture

This data link protocol architecture is designed to ensure reliable transfer of data between two computers over a physical link (RS-232). It involves two layers, each responsible for distinct functions.

- Main component
  The main component serves as an entry point for initializing the protocol. It reads necessary parameters: port name, baud rate, role and path to the file that is to be transmitted/received. Checks the correctness of those parameters and starts the application layer.
- Application layer
  This component action depends on the role of the machine.
  a) Transmitter
  Application layer first establishes the connection using the link layer.
  Next it opens a file to be sent. Divides the file into smaller pieces of data and packs them. Obtained packets forwards to the link layer and continues with forwarding the next packet unless the link layer signalized an error. In this case the program stops. When the whole file is sent, the application layer closes the file and terminates the connection in agreement with the receiver using the link layer.
  b) Receiver
  In this case the application layer first opens/creates a file in which received data is to be stored. Then it reads received packets using the link layer unless the stop Control packet was received. Received packets are parsed. Parsing includes extracting data and saving it to the previously opened file.

After receiving all the data, the application layer closes the file and terminates the connection in agreement with the sender using the link layer.

- Link layer

  In the Link Layer, data transmission is managed to ensure reliable delivery between the transmitter and receiver. The process begins with llopen, where the link layer establishes a connection and defines the roles for each device. During transmission, the llwrite function is used to send data frames from the transmitter, while the receiver reads them using llread. The link layer handles error detection through checksums or CRC validation, and if any errors are detected, it requests retransmission by issuing negative acknowledgments (NAKs). If no errors are found, positive acknowledgments (ACKs) are sent to confirm successful receipt. This sequence repeats until all data frames are correctly transmitted. When data transfer is complete, the llclose function performs an orderly shutdown by exchanging final control frames to confirm the end of communication, ensuring both devices terminate the connection cleanly and reliably.

III. Code Structure

This section explains the organization of the code, which is designed to separate different layers and functionalities. Each component has its own APIs, function and data structures to ensure smooth functioning, interactions and layer independency.

- APIs

  Main APIs in this project are llfunctions. They allow application layer to use link layer functionalities.
  - llopen

    Establishes the connection, setting up the link layer protocol for communication based on the given parameters.
  - llwrite

    Sends frames. Handles acknowledgments, timeouts, retransmissions and errors
  - llread

    Receives and validates frames. Sends appropriate responses (RR, REJ).
  - llclose

    Responsible for closing the connection using the sequence of messages between transmitter and receiver.
- Data structures

  Both link and application layers have data structures which help them to work smoothly and find errors easily.
  - packet

    Technically it is just an array of chars. However, the way of populating this array is strictly defined according to the instructions given in the project guide.
    There are two types of packets, Control packets and Data packets.
    Both Control and Data packets have a control field at the beginning which implies if it is 1 - start Control packet, 2 - data packet, 3 - end control packet.
    After the control field the Control packet has the size and the name of the file coded as TLV.

    After the control field the Data packet has the sequence number which implies which Data packet (0-255) it is. Next there are two bits whose value (V) implies how many bytes of data is saved in this particular packet. Next there is a series of V data bytes.

    - Result
      Simple helper data structure that is returned by functions creating packets. It consists of a pointer to the created packet and a value which indicates this packet size.
    - frame
      The frame data structure stores the an organized version of what will be send or received from a transmission as to make it more intuitive to process. This has fields for the address, control, bcc1, data(packet) and bcc2.

- Functions
  Each layer consists of a few functions which are responsible either for control flow or for doing some specific task.
    - int main(args)
      Starting point for the protocol. Also validates the arguments and calls applicationLayer.
    - void applicationLayer(args)
      Main function of the Application layer. It is called by main and is generally responsible for control flow in the protocol. It works as the application layer component described in Architecture paragraph
    - Result getControlPacket(args)
      Called by the applicationLayer. Given a value of the control field, file name and file size it creates a Control packet as described in the Data structures section above. It returns the Result structure.
    - Result getDataPacket(args)
      Called by the applicationLayer. Given a piece of file data obtained by the applicationLayer, size of this piece and sequence number it creates a Data packet as described in the Data structures section above. Returns the Result structure.
    - int parsePacket(args)
      Called by the applicationLayer. Given a pointer to the packet, packet size and file pointer it checks a type of the packet (control field) and acts adequately. If it is the Data packet it reads data from it and writes this data to the file. If it is the start Control packet it reads file name and file size from it. If it is the end Control packet it again reads file name and file size and compares it to the file name and file size read from the start Control packet.
    - int write_frame(args) and read_frame
      Called by the linkedLayer. these functions take a pointer of type frame and either write it to the serial port or read a valid frame from the serial port. they accommodate any type of frame and return negative values in case of errors

IV. Main Use Cases

The main use case of this project is to enable reliable file transfer between two computers over a direct physical link, specifically an RS-232 serial connection, by implementing a basic data link protocol using the Stop & Wait method. This setup allows the transmitter to send data packets to the receiver while ensuring each packet's integrity through acknowledgments, timeouts, and retransmissions when needed. With functions to start and close connections, send and receive data packets, and handle error detection, this protocol ensures that files are accurately and completely transmitted, even over potentially unreliable communication links, making it ideal for scenarios where data reliability is critical.

V. Logical Link Protocol

The Logical Link Protocol aims to insure reliability in the transmission of packets even when the connection is prone to errors. To this end it uses a checksum system in the form of BCC ensuring reliable and fast checks.

```c
void stuffing() {
    processedBufferSize = 0;

    for (int i = 0; i < bufferSize; i++) {
        uint8_t byte = buffer[i];

        if (byte == FLAG) {
            // Replace 0x7E with 0x7D 0x5E
            processedBuffer[processedBufferSize++] = ESCAPE;
            processedBuffer[processedBufferSize++] = byte ^ 0x20;
        } else if (byte == ESCAPE) {
            // Replace 0x7D with 0x7D 0x5D
            processedBuffer[processedBufferSize++] = ESCAPE;
            processedBuffer[processedBufferSize++] = byte ^ 0x20;
        } else {
            // Regular byte, add to output
            processedBuffer[processedBufferSize++] = byte;
        }
    }
}

void destuffing() {
    processedBufferSize = 0;
    for (int i = 0; i < bufferSize; i++) {
        uint8_t byte = buffer[i];

        if (byte == ESCAPE) {
            // Escape sequence detected, XOR the next byte with 0x20
            i++;  // Move to next byte
            if (i < bufferSize) {
                processedBuffer[processedBufferSize++] = buffer[i] ^
0x20;
            }
        } else {
            // Regular byte, add to output
            processedBuffer[processedBufferSize++] = byte;
        }
        printf("Processed buffer[%d]: %02x\n", processedBufferSize - 1,
processedBuffer[processedBufferSize - 1]);
    }

    for (int i = 0; i < processedBufferSize; i++) {
        printf("POST Processed buffer[%d]: %02x\n", i,
processedBuffer[i]);
    }
}
```

When reading a frame the read_frame function will keep reading between flags to find a possible frame. if just after a BCC failed it receives a valid address it will assume that it missed a flag and so will try to recover that frame

```
int populateFrame(Frame* frame){
    destuffing();

    frame->address = processedBuffer[0];
    frame->control = processedBuffer[1];
    frame->bcc1 = processedBuffer[2];

    if (frame->bcc1 != (frame->address^frame->control)){
        printf("BCC1 failed!\n");
        return -1; //BCC1 failed!
    }


    if (processedBufferSize == 3) return processedBufferSize; //Control
frame

    frame->infoFrame.dataSize = processedBufferSize - 4;
    uint8_t bcc = calculate_bcc(processedBuffer, 3,
frame->infoFrame.dataSize);

    if (bcc != (uint8_t)processedBuffer[processedBufferSize - 1]){
        printf("BCC2 failed!\n");
        printf("CALCULATED BCC2: %02x\n", bcc);
        printf("EXPECTED BCC2: %02x\n",
(uint8_t)processedBuffer[processedBufferSize - 1]);
        return -2; //BCC2 failed!
    }

    for (int i = 0; i < frame->infoFrame.dataSize; i++) {
        frame->infoFrame.data[i] = processedBuffer[i + 3];
    }

    return processedBufferSize;
}
```

VI. Application Protocol

Application protocol aim is to manage the protocol's control flow and to care for the high-level transmission of the file using a low-level link layer. First step for both the transmitters and the receivers application protocol is to initialize the connection using llopen.

```
if(llopen(connectionParameters) < 0){
    printf("Couldnt open the connection\n");
    exit(-1);
}
```

After this step, the functioning of the application protocol differs.

- TRANSMITTER
  Opens the file to be transmitted, checks its size and creates a buffer of appropriate size.

```
#define SEQUANCE_MAX 255 //because there is 1 byte for seq num in packet
//...
int buffer_size = ceil(file_size / SEQUANCE_MAX+1);
        printf("buffer size: %d bytes\n", buffer_size);
        unsigned char* buffer = (unsigned char*)malloc(buffer_size *
sizeof(unsigned char));
```

Next, the opening control packet is sent using llwirte.

```
Result r = getControlPacket(1, file_name, file_size);
```

```
if(llwrite(r.pointer,r.value) < 0){
    printf("llwrite couldnt send opening packet\n");
    exit(-1);
}
```

Next the application protocol enters the loop which ends after the whole file was read and successfully sent. Data from the application layer is passed in packets using llwrite to the link layer.

```
while((actual_size = fread(buffer,1,  buffer_size, fptr)) > 0){
    //creation of the packet
    r = getDataPacket(buffer, actual_size, sequence);
    //...
    int send = TRUE;
    while(send){
        switch(llwrite(r.pointer, r.value)){ //actual sending
        case -1: // big error
            printf("llwrite couldnt send packet %d\n", sequence);
            exit(-1);
            break;
        //other error cases...
        default: // >0 so okay
            send = FALSE;
            break;
        }
    }
}
```

Next the closing control packet is sent the same way as opening one. Final step is terminating the connection in agreement with the receiver using llclose and also closing the file.

```
if(llclose(1)<0){
    printf("smh wrong with tx llclose\n");
    exit(-1);
}
fclose(fptr);
```

- RECEIVER

  Receiver creates a buffer and next it is constantly reading packets using llread and parsing them. It stops only after receiving the closing control packet or in the case of an error. After successful receipt of the whole file it calls llclose and closes the file.

```
unsigned char* buffer = (unsigned char*)malloc(MAX_PACKET_SIZE *
sizeof(unsigned char));
int END = FALSE;
while(!END){
    result = llread(buffer);
    if(result>=0){
        if(parsePacket(buffer, result, fptr)==3) END = TRUE;
    }
        if(result<0){
            printf("llread returned error\n");
            exit(-1);
        }
    }
    if(llclose(1)<0){
        printf("smh wrong with rx llclose\n");
        exit(-1);
    }
    fclose(fptr);
```

VII. Data Link Protocol Efficiency

Below table presents average execution time of our protocol in different scenarios. Average execution time for each scenario was calculated on a sample of only 3 tests.

		bit error ratio		
		0	0.0001	0.001
baud rate	1200	2m03s	2m14s	4m49s
	9600	0m15s	0m24s	1m48
	115200	0m1 .2s	0m3s	1m37

It is worth noting that the biggest time cost is the cost of timeouts. In above tests the timeout was set to 3 seconds, but in the scenario of 9600 baud rate and 0.001 bit error ratio with timeout set for 1 second the average execution time was only 46 seconds! Another important fact is that the received file is not always identical to the sent one because it uses an easily failing method of detecting errors in the data field.

VIII. Conclusion

In conclusion, this project successfully demonstrated a simplified Stop & Wait data link protocol designed to ensure reliable file transfer over a direct serial link, even in environments prone to transmission errors. The layered architecture—with the application layer for high-level control and file management, and the link layer for low-level data handling—allowed for efficient, modular, and adaptable communication. This setup showcased the core principles of packet-based error detection, control frame validation, and data integrity checks, highlighting the importance of reliable data transmission protocols in network communications. Through testing, we observed the impact of various transmission speeds and error rates, which provided insight into the trade-offs between reliability and efficiency. This implementation serves as a foundational model of data link protocols, illustrating the practical application of network theory in real-world digital communication.

Appendix

For the entire project see this repository
For the project guide see this file in the above repository