Data Link Protocol

Computer Networks
Bachelors in Informatics and Computing Engineering

3LEIC03_G3

Tiago Rodrigues up201907021@fe.up.pt
Mário Travassos up201905871@fe.up.pt

December 9, 2021

Summary

This report will cover the first work proposed for the Computer Networks Curricular Unit, with the objective of creating a small application that could transfer data through two computers asynchronously, through a serial port.

The application is capable of transferring files whilst mainting their integrity, and detect errors in transmission, resolving them if possible.

Introduction

This report is the result of an examination to the practical component, which was the development of a data transfer protocol. A serial port was used to transfer the files in an asynchronous fashion.

The report is organized as follows:

1. Architecture- Functional blocks and interfaces
2. Code Structure - APIs, main code structures and their relation with the architecture
3. Main use cases - Identification and Call Stack Sequence
4. Data link Protocol - Main functional aspects and implementation strategy
5. Application Protocol - Main functional aspects and implementation strategy
6. Validation - Description of the tests conducted
7. Efficiency - Statistical characterization of efficiency, against a Stop&Wait protocol
8. Conclusion - Summary of the above descriptions, reflection on the learning objectives

1 Architecture

The application consists of two main layers, one to interact with the file to be sent and another to interact with the hardware. They are the Application Layer, and the Data-Link Layer, and they are detailed below.

1.1 Application Layer

This layer can be found in the **rcom-ftp.c** file, and it encompasses everything related to interaction with the files, be it opening, closing, reading and writing to and from it. Besides that, this is the layer through which the user interacts with the application.

1.2 Data-Link Layer

This layer can be found in the **ll.c** file and it is responsible for ensuring a smooth data transmission over the hardware, including opening, closing, writing and reading from the serial port, with the help of the auxiliary functions present in **config.c**, **read.c**, **send.c**, **state.c** and **utils.c**.

2 Code Structure

The code is divided into seven source code files, separated by responsibility (reading from or writing to the serial port), and by layer. Also, each of them has a corresponding header file. Finally, there is a dedicated header file that hosts several common constants.

2.1 Application - rcom-ftp.c

This module contains the entire application layer developed.

Main Functions

- main - Interacts with the user and passes the arguments given to the rest of the program.
- sendFile - Sends the file requested by the user.
- readFile - Retrieves the file sent by the user.

Main Data Structures

- fileData - Responsible for holding some metadata of the file.

2.2 Config - config.c

This module contains the functions required for setting up the serial port for proper file transferring.

Main Functions

- set_config - Sets up the initial configuration for the serial port.
- reset_config - Restores the serial port to its initial state.

2.3 Link Layer - ll.c

This module contains the interface for the Link Layer of the protocol.

Main Functions

- llopen - Opens the serial port from frame transmission.
- llwrite - Writes a frame to the serial port.
- llread - Reads a frame from the serial port and checks its integrity.
- llclose - Closes the serial port after communication ceases.

2.4 Reading - read.c

This module contains the functions responsible for reading from the serial port.

Main Functions

- readSupervisionFrame - Reads a supervision frame and checks if the information is received correctly.
- readInformationMessage - Reads an information message and saves the data, which includes the BCC, in a buffer.

2.5 Writing - send.c

This module contains the functions responsible for writing to the serial port.

Main Functions

- writeSupervisionAndRetry - Attempts to write a supervision Frame in 3 attempts. If it succeeds, it stops.
- writeInformationAndRetry - Attempts to write an information Frame in 3 attempts. If it succeeds, it stops.

2.6 State Management - state.c

This module contains the functions responsible for managing the state of the application.

Main Functions

- handle_state - Function responsible for managing the state of the application, according to the data received.

Main Data Structures

- state_t - Enumeration containing the possible states of the application.
- state_machine - Besides the state of the machine, holds some of the main pieces of information from each frame.

2.7 Utilities - utils.c

This module contains some auxiliary functions that help the others with their operations.

Main Functions

- stuff_data - This functions stuffs the data given.
- unstuff_frame - This functions unstuffs the frame given.

2.8 Constants - defines.h

This module contains some of the more meaningful constants shared throughout the application.

3 Main use cases

The application should be first compiled with the provided Makefile, by running `make clean && make`. Then, if we are the receiver, we run `./rcom-ftp receiver <port>`, or, if we are the emitter, `./rcom-ftp emitter <port> <file>`, where port is the number of the port to be used, which will be translated to `/dev/ttyS<port>`, and file is the file to be sent over.

When starting up the application, the receiver should be called first, otherwise the emitter will be trying to connect for a total of 9 tries, making a stop of 1 second between each. If it can't succeed in connecting, it will halt. Otherwise, the application layer will begin dividing the file into packets, and sending to the receiver through the **llwrite** function, whilst the receiver is reading with the **llread** one. Finally, both of them should call **llclose** in order to cease transmission.

4 The Data link Layer

This layer is responsible for interacting with the serial port, abstracting away the intricacies using it, producing a consistent layer of work for the application. It uses several auxiliary functions, as a way to better structure the code, assigning to each function a single responsibility. The main functions are divided as follows:

llopen

The **llopen** function sets up the communication between the transmitter and receiver. It start by configuring the serial for for proper reading and writing, setting the appropriate flags and setting **VTIME** to 30 and **VMIN** to 0, which ensures the read function won't have to wait for a character to return.

After that, and according to the provided role, it will either send a **SET** command and await for a **UA**, if it is the emitter, or wait for a **SET** command and then send a **UA**, in the case of the transmitter. To do this, they take advantage of the `writeSupervisionAndRetry` and the `readSupervisionFrame` functions, which, respectively, handle writing to the serial port and reading from it, setting the appropriate state.

llwrite

As the name implies, the **llwrite** function writes a given packet of data to the serial port. To do this, it calls the `writeInformationAndRetry` function, which, through the use of `writeInformationFrame`, appends the frame header and trailer, and writes it to the serial port. It retries if the writing is unsuccessful, a total of 3 times, point at which it returns with an error. After a successful writing, it waits to receive a confirmation message, either accepting or rejecting the frame. If it accepts, then the frame is written and **llwrite** halts. Otherwise, if it was rejected, then it resends the same frame again, not increasing the attempts made (as writing was successful, but the message got rejected). Finally, if some error occurred, it tries to resend the message until all tries all consumed.

llread

The workings of **llread** are similar to those of **llwrite**. First, it tries to read the incoming information message. When successfull, it then proceeds to unstuff the incoming frame and check if the data has been corrupted. If it hasn't, then it acknowledges the packet, sending back a **RR** frame, and if sent successfully, returns the size of the information read. When the data appears to be corrupted, it sends a **REJ** frame, telling the emitter to retry sending that information. Finally, if some error occurred, it returns with an error as well, ceasing transmission.

llclose

Finally, this function handles the cease of communication between the two computers. If the caller is the emitter, it will start of by sending a **DISC** frame, telling the receiver it wishes to stop communication. Then, it waits for a **DISC** response and sends a final **UA** frame before terminating. The receiver works the other way around, first waiting for a **DISC** frame to be read, then sending his **DISC**, and finally reading the **UA** before being able to shutdown. Both, in case there is a problem with writing, try again for a total of 9 attempts, before exiting with error.

5 The Application Layer

sendFile

This is the main function that drives the emitter. It starts by retrieving the metadata from the file, to make it easier to divide into equally sized packets. Afterwards, it generates a starter packet that contains the metadata from the file, and proceeds to send out the file contents. Each packet sent has a maximum size of 1024 bytes, which means that it will send 1024 byte packets for as long as possible, and in the last one, it sends the leftover bytes that didn't reach 1024. Finally, it resends the starter packet, but with a flag changed meaning that it will be end of the transmission. At each step, in the event of a malfunction, it returns with an error.

receiveFile

On the other end, the **receiveFile** function is in charge of driving the receiver. It's way of working is much like the one on **sendFile**. Firstly, it reads the starting packet containing the metadata on the file. It then creates the new file, appending received to the start of the original file name. Afterwards, it reads the packets containing information, one by one, and assembles the file after each successfull packet receive. Finally, it reads the end Packet, which will tell the receiver to stop trying to read any longer. Once again, if at any point the receiver detects an error reading any packet, it will stop with an error.

generateDataPacket

This function creates the packets that will be sent over to the receiver. It starts by appending the correct packet header, and then copies the information over to packet buffer, returning it.

readDataPacket

This function is analogous to the **generateDataPacket** one, and it's job is to read a packet and check for its integrity after transmission. To do this, it starts by checking if the packet is in fact a data packet, by verifying the first bit. Then it makes sure that the data packet received is not a duplicate, aborting if it is. Finally, and if all goes well, it retrieves the information contained and saves it in a buffer.

6 Validation

In order to test the protocol, files of different sizes were sent, namely **pinguim.gif** (10968 bytes) and testText.txt (2053 bytes). Also, the transmission was tested with timeouts on the lab, and with introduction of random errors, by placing a bit of metal that would send erroneous bits sometimes. All tests were verified by using the **md5sum** hashing function.

7 Efficiency

As of the writing of this report, efficiency was not tested.

8 Conclusion

The task assigned consisted in implementing a data transfer protocol to communicate over a serial port. In this assignment, it was key to create distinct layers, and, more importantly, make sure they didn't know each other's implementation. This lead to a structure in which the application layer, the "highest" section of the program, only knew what the data-link layer did, but not how it operated, and the data-link knew only that it would receive data and would have to transmit it. This effectively creates two distinct entities that in nothing influence each other, but work together for the objective of transmitting data.

The project was completed successfully, and it contributed a great deal for a deepened knowledge on how data is actually transferred, how errors in the data are handled, and how a file can almost magically appear on another computer.