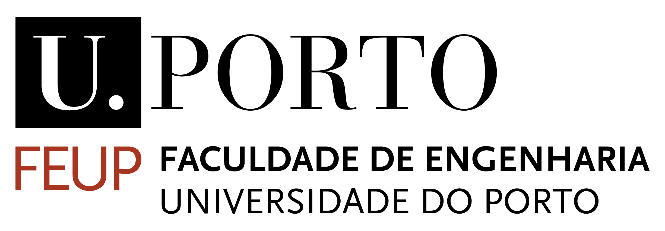
Faculty of Engineering of the University of Porto

Computer Networks



**Serial Port Project Report**

Developed by:

Daniel Marques - up201503822

João Carvalho - up201504875

Renato Campos - up201504942

Porto, 4th November 2017

**Summary**

We are students of a Master’s Degree in Informatics and Computing Engineering at FEUP. In Computer Networks class, we have been challenged to develop a program that uses the serial port to send data between two connected Linux machines.

We’ve concluded that a serial port transfer is, by today’ standards, very slow compared to other forms of data transfer, at least with the protocol that was adopted. On the other hand, with our implementation of the protocol it resists errors caused by interference or interrupted connection.

**Introduction**

The objective was to create software that sends information between two Linux machines using the serial port, using a Stop-and-Wait ARQ mechanism and a specific protocol. This report will explain the architecture and its modules; the code structure by showing the code flow as well as listing the most important data structures and functions; the protocols used in both the link layer and application layer.

**Architecture**

The program consists of two decupled primary modules: application layer and link layer. On the other hand, the link layer module depends on a set of secondary modules: alarm, serial port and utilities.

The link layer is a set of functions that create an API. It’s responsible for developing a protocol that allows it to open/close the serial port connection, read/write data from/to the serial port buffer and recover from errors and interruptions.

The application layer is a generic program that uses the link layer API to send data through a serial port. Its purpose is to read a file’s data and call the link layer functions to open/close the serial port and read/write the data. It’s responsible for adopting an application-specific protocol, so that both sender and receiver work in sync.

**Code Structure**

**Code Flow**

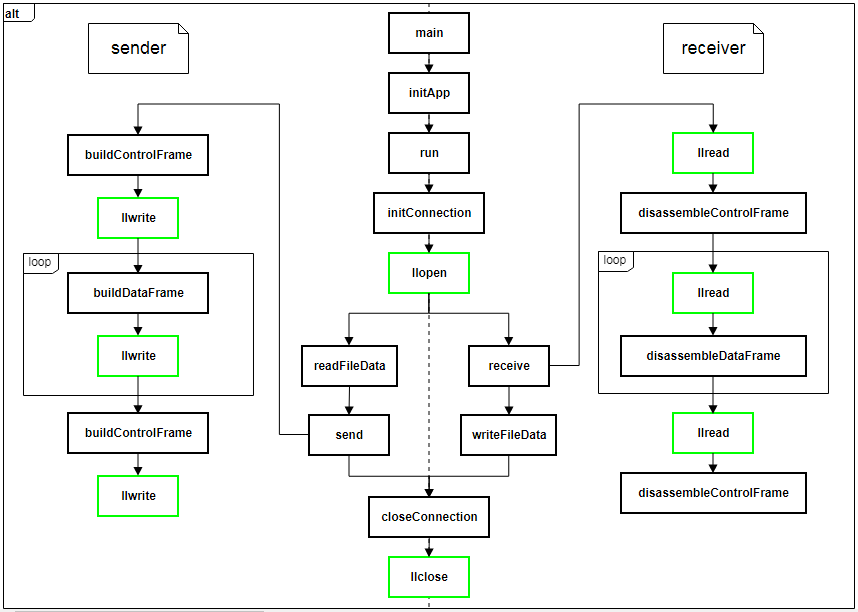
****

Figure - Application layer code flow

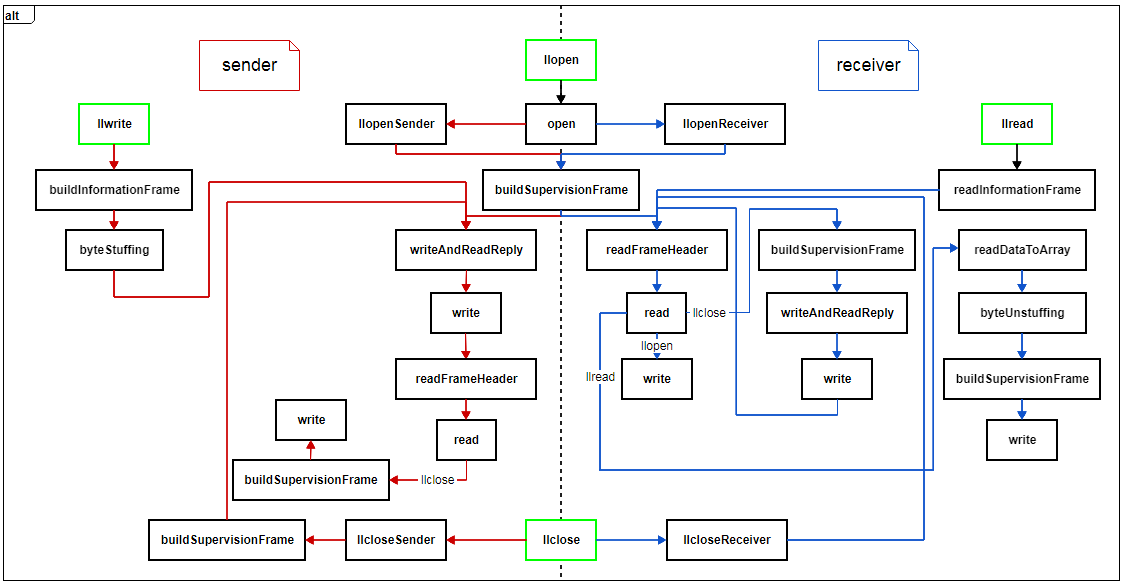
****

Figure - Link layer code flow

**Primary data structures**

****

The application layer has three important data structures. ‘ApplicationLayer’ holds data regarding the serial port connection, mode of operation and information about the file. ‘ControlFrame’ and ‘DataFrame’ holds important fields from a control/data frame as well as the complete frame itself.

The link layer has two important data structures. ‘LinkLayer’ holds some serial port settings, protocol helping variables and the current frame. ‘FrameHeader’ holds the fields from a frame’s header that are needed for post processing in other functions.

****

**Primary functions**



This function is capable of reading any frame’s header using the state machine defined in the link layer protocol. It reads byte by byte using the ‘read’ function and while applying the state machine logic, determines if the header is what’s expected (OK), duplicate message (DUPLICATED), rejection message (REJECTED) or timeout (ERROR), returning the respective enum. To avoid blocking, in the while loop that covers almost the entire function there’s a condition that breaks the loop if the alarm flag is set by the alarm handler.



This function writes any frame to the serial port buffer and waits for a reply. After writing ‘frame\_to\_write’ with the ‘write’ function the alarm is set for the alarm timeout and the program calls ‘readFrameHeader’. If it succeeds to read it before the alarm timeout, the alarm is cancelled and the return value of it is checked. If it’s OK or DUPLICATED, the function ends with success (returns 0). If it’s REJECTED or ERROR, the frame is retransmitted, with the last one incrementing the amount of tries it took to send it. If this amount surpasses the predefined maximum of tries, the loop breaks and the function ends with error (returns -1).

**Protocols**

The implementation in our software of the following protocols allows for total layer independency, meaning they have no knowledge of each other’s protocol.

**Link Layer Protocol**

The program can send and read Information, Supervision and Unnumbered acknowledge frames. We’ll define Supervision and Unnumbered Acknowledge as Headers, as they are very similar to Information frame headers.

A Header consists of ADDRESS FIELD, CONTROL FIELD and BCC1.

**ADDRESS FIELD** is used to identify who is sending the frame and the type of the frame.

**CONTROL FIELD** is used to identify the sequence number of the frame on Information frames. On Control frames it is used to identify the type of control byte being sent which can be SET, DISC, UA, RR or REJ.

SET - establish the connection.

DISC - close the connection.

UA - unnumbered acknowledge.

RR - receiver ready: positive acknowledge.

REJ - reject: negative acknowledge.

**BCC1** is the parity byte, calculated as the XOR between ADDRESS and CONTROL BYTE.

An Information frame consists of a Header, DATA and BCC2.

**DATA** is the data to be transmitted.

**BCC2** is the parity byte, calculated as the XOR between all DATA bytes.

All the frames are delimited on both ends by a **FLAG** byte.

To ensure transparency when sending Information frames, byte stuffing is done on all DATA and BCC2 bytes thus avoiding a byte being interpreted as a FLAG or as the ESCAPE character used on the process. For the same reason, when reading Information frames, DATA and BCC2 bytes are unstuffed before checking BCC2 correctness.

To read **Headers** a state machine is used to make sure that the **Sender** sent a valid frame.

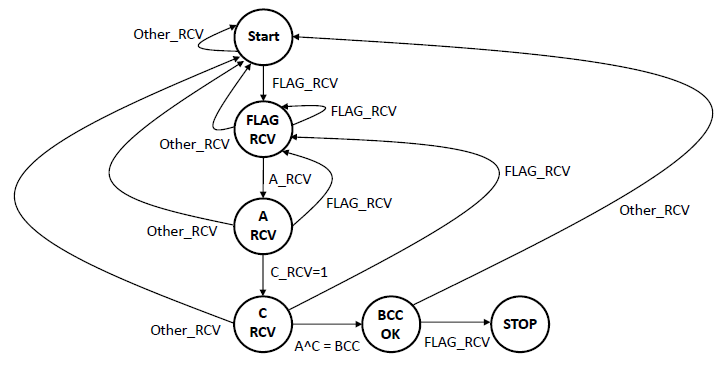


Figure - State machine used to read a frame header

**FALTA FALAR DE ERROS, REJECTS, ALARM TIME-OUT …**

The implementation of this protocol can be best seen in following functions:

* buildSupervisionFrame
* buildInformationFrame
* byteStuffing / byteUnstuffing
* writeAndReadReply
* readFrameHeader

**Application Layer Protocol**

The protocol defines two types of packets: control and data. Control packets mark the start and end of transmitting data from a file as well as relevant information about said file, such as its size and name. Data packets, on the other hand, contain fragments of data from the file.

They both have a Control(**C**) field. Its possible values are: 1 – data, 2 – start, 3 – end.

Additionally, the data packets have the following byte structure:

**N** – ‘Sequence number’ mod 255.

**L2** and **L1** – Size of the data. Size = 256 \* L2 + L1.

**Data** – A fragment of data from the file.

Likewise, the control packets follow a TLV structure with 2 parameters:

**T** – Parameter index (0 – file size, 1 – file name).

**L** – Length of the V field

**V** – Parameter value

The implementation of this protocol can be best seen in following functions:

* buildControlFrame / disassembleControlFrame
* buildDataFrame / disassembleDataFrame

**Validation**

**Link Layer Protocol’s Efficiency**

**Conclusions**

After all the objectives being met, we can safely conclude that using the serial port with the adopted protocol is inefficient and, by today’ standards, quite slow. However, it does recover from any type of errors, whether they are cause by interference or interruption of connection. Another advantage is the encapsulation of the protocol. Each layer is independent and “blind” to the other protocols.

On a learning note, we believe this project taught us a lot about networks and programming in general. It gave us the ability to go through any protocol’s documentation and implement it, test it and reflect on it’s efficiency.

**Attachments**

**Attachment I - Source Code**

The complete source code can be seen at: LINK HERE.

**Attachment II - Usage**

Firstly, execute ‘make’ inside the sources directory to compile the program. Both sender and receiver are the same application, though they receive different command line arguments. To launch the application, type the name of the executable (i.e. ‘./app’). Then, you should type the port (e.g. /dev/ttyS0) and the mode on which the program should operate (i.e. ‘receiver’ or ‘sender’).

        The sender program will also need to receive as a command line argument the path of the file to send (e.g. ‘banana.gif’). Optionally, you can specify the amount of file data in bytes to send per packet (e.g. ‘1024’).

        Example of usage:

-       ‘./app /dev/ttyS0 receiver’

-       ‘./app /dev/ttyS0 sender banana.gif 1024’