Relatório - Trabalho Laboratorial Nº1

Licenciatura em Engenharia Electrotécnica e de Computadores

Redes de Computadores

Duarte Ribeiro Afonso Branco 201905327
Pedro Afonso da Silva Correia de Castro Lopes 201907097

Abril 2022

Índice

Sumário

Este trabalho, realizado no âmbito da unidade curricular de Redes de Computadores, teve como objetivo o desenvolvimento e testagem de um protocolo de ligação de dados através da porta série.

O resultado deste trabalho foi satisfatório, visto que foram alcançados os objetivos definidos, nomeadamente, foi garantida o estabelecimento de uma ligação robusta, entre emissor e recetor, capaz de detectar e reagir a erros do modo pretendido e de assegurar a transmissão da totalidade dos dados.

Introdução

Este trabalho teve como objetivo o desenvolvimento e implementação de um protocolo de ligação de dados resistente a erros de transmissão e consequente a testagem realizada no ambiente laboratorial através da manipulação da porta série de modo a introduzir erros.

O presente relatório encontra-se dividido em diversas secções. Em Arquitetura e Estrutura do código é descrita a organização do código de um modo geral. Em Casos de uso principais é descrito como executar o projecto e as sequências de chamadas de funções. Em Protocolo de ligação lógica é descrito em detalhe todas as funções do código implementado e em Protocolo de aplicação é descrito o funcionamento do código fornecido. Em Validação e Elementos de valorização são descritos os testes realizados e descritos os elementos de valorização implementados, respectivamente. Por fim, na Conclusão é feita uma síntese de todos os tópicos e reflexão do trabalho realizado.

Arquitetura

O projeto desenvolvido, correspondente ao ficheiro *linklayer.c*, constitui o bloco funcional correspondente à camada de ligação, nomeadamente, capaz de estabelecer e assegurar a correta transmissão da totalidade dos dados fornecidos pela camada de aplicação.

O ficheiro fornecido *main.c* constitui o outro bloco funcional, correspondente à camada de aplicação, responsável pela leitura do ficheiro e divisão em pacotes de dados a enviar, assim como, a consequente recessão dos pacotes de dados e escrita do ficheiro recebido. .

A interface entre os dois blocos é realizada através das funções principais do ficheiro *linklayer.c*, uma vez que este constitui uma biblioteca invocada pelo ficheiro *main.c*.

Estrutura do código

As principais funções implementadas no ficheiro *linklayer.c* correspondem às chamadas no ficheiro *main.c*, nomeadamente *llopen*, *llwrite*, *llread* e *llclose*. Estas funções constituem a interface entre os dois blocos principais e são utilizadas para, nomeadamente, estabelecer ligação, enviar pacotes, receber e guardar pacotes e, por fim, fechar ligação e mostrar estatísticas relevantes da ligação.

Os dados passados às funções principais correspondem à *struct linkLayer* com informação relevante ao estabelecimento de ligação, no caso de *llopen*, ao tamanho do vetor, em *llwrite*, a endereços para vetores, em *llwrite* e *llread* e a um inteiro que define a impressão

dos dados de estatística no *llclose*. Destaca-se ainda a definição do tamanho máximo dos pacotes e os valores de *default* para *baudrate*, número máximo de retransmissões e *timeouts*. São ainda utilizadas diversas variáveis globais de modo a agilizar a comunicação entre funções.

As funções principais utilizam diversas funções auxiliares, nomeadamente, várias máquinas de estado para detecção de recessão de *frames*, *handler* de alarme, cálculo e verificação de código de correção de dados, ou seja, o elemento *BCC*, *stuffing* e *destuffing*.

Casos de uso principais

Primeiramente, é necessário compilar o ficheiro *linklayer.c*, depois é necessário compilar *main.c* tendo o cuidado de invocar o ficheiro .o associado ao protocolo de ligação. Destaca-se, consequentemente, a necessidade de executar em primeiro lugar o programa recetor e apenas depois o transmissor, visto que, caso contrário existe o risco do transmissor exceder as tentativas de estabelecimento de ligação se o receptor não for executado atempadamente.

No caso de transmissor é executada a função *llopen*, após o estabelecimento de ligação passa para a função *llwrite* até enviados todos os dados e finalmente termina a ligação com *llclose*. Por sua vez, a sequência de chamada de funções no receptor corresponde a *llopen*, até establecimento de ligação, *llread* enquanto recebe pacotes de dados e término de ligação com *llclose*.

Protocolo de ligação

De destacar que todos os Fragmentos de código referidos encontram-se no Anexo 2 - Extratos de código e os diagramas de todas as funções correspondentes a máquinas de estado no Anexo 3. A totalidade do código desenvolvido encontra-se no Anexo 1 - Código fonte.

llopen

A função *llopen* inicia-se com o estabelecimento de uma ligação do tipo *non-canonical* semelhante à dos ficheiros fornecidos *writenoncanonical.c* e *noncanonical.c*, destacando-se a utilização da função *get_baud* que converte o *Baudrate* passado à função no formato correto e o facto do *read* ser satisfeito se um carácter lido ou o tempo exceder 100ms, como visualizável no Fragmento de código 1.

Posteriormente, através do parâmetro *connectionParameters.role* determina-se o modo de funcionamento do programa. Ao se tratar do transmissor (Fragmento de código 2) procede-se ao envio de *SET* e à inicialização do alarme, cuja função *heandler atende* que altera *alarm_stop* se o tempo definido para o alarme, correspondente ao tempo de *timeout*, excedido. Posteriormente, o transmissor lê os valores recebidos até verificar a recessão de *UA*, através da função *UA_state_machine*, ou até excedido o *timeout* e o valor de *alarm_stop* corresponder à unidade. Ao verificado *timeout* é incrementado o valor de *attempt* e repetido todo o processo, enquanto não atingido o número máximo de tentativas. Se recebido *UA* o alarme é desligado e retornado sucesso.

Ao se tratar do recetor (Fragmento de código 3) é iniciado um ciclo que lê os valores recebidos até verificar a receção de um *SET*, através da função *SET_state_machine*, e enviado *UA*.

Em caso de recebido um *role* não definido ou de erro em alguma função é indicado erro e terminada a execução.

llwrite

A função *llwrite* (Fragmento de código 6) é responsável por receber os dados a transmitir, no vetor *buf* e respectivo tamanho em *bufSize*, e enviar um *frame* do tipo I. Assim sendo, é alocada memória para o vetor a enviar *I_send* de tal modo a modo a que corresponda à necessária para o pior cenário, ou seja, para o cabeçalho, para o caso em que todos os dados e *BCC2* necessitam de *stuffing* e para a *FLAG* final.

Para tal, começou-se por definir o cabeçalho, tendo em conta o número de sequência. Depois foi calculado o elemento de verificação *BCC2* através da função *BCC2_calculate* (Fragmento de código 4). Esta função percorre o vetor *buf* procedendo ao cálculo do *byte BCC2* e consequentemente realiza o *stuffing* dos dados invocando a função *stuffing* (Fragmento de código 5). Por sua vez, a função *stuffing* assegura a transparência dos dados, implementando os mecanismos de escapatória definidos, nomeadamente, a alteração de um octeto correspondente a *FLAG* para 0x7D e 0x5E, e de um octeto 0x7D para 0x7D e 0x5D. Através do *stuffing* assegura-se a não terminação quando um *byte* corresponde a *FLAG*. É ainda importante destacar que se procede ao *stuffing* do byte *BCC2* após este ser calculado. No final é definida a *FLAG* final e realocada a memória associada a *I_send* para a estritamente necessária.

De um modo semelhante ao da função *llopen*, procede-se depois ao envio de *I_send*, e à leitura da resposta até verificada uma resposta através da função *RR_REJ_state_machine*. Se verificada uma resposta de não sucesso, *REJ*, ou excedido o *timeout*, reenvia-se o *frame* novamente, até esgotadas o número de tentativas definido. Note-se ainda que o número de sequência não é alterado se recebido um *REJ*.

Finalmente é desativado o alarme e retornado o tamanho do *frame* enviado. Ao longo da função se se verificar um erro em alguma função ou alguma variável exceder os limites definidos é indicado erro e terminada a execução.

llread

A função *llread* recebe um apontador para o vetor *packet* onde posteriormente se guardam os dados recebidos. Assim sendom começa por alocar memória ao vetor *I_recieve* correspondente ao pior cenário descrito anteriormente, onde todos os dados e *BCC2* necessitam de *stuffing*.

Seguidamente, entra num ciclo (Fragmento de código 7) onde lê para *I_recieve* o *frame* recebido até detectar a *FLAG* final através da função *I_state_machine*, tendo em conta os números de sequência. No caso de verificar um *SET* responde com um *UA* e continua à espera da recepção do *frame* I. Este cenário acontece se na função *llopen* for enviado um *UA,* por parte do recetor, não recebido pelo transmissor, que consequentemente envia um *SET* lido na função *llread*, visto que, logo após enviado o *UA* a execução do programa evolui para a função seguinte.

Após detectada a *FLAG* final é realocada a memória do vetor *I_recieve* para a estritamente necessária e alocada memória para o vetor *DataBCC2_DE* correspondente ao cenário onde todos os dados e *BCC2* necessitam de *stuffing*. Através da função *DE_stuffing* (Fragmento de código 8) é realizado o processo inverso ao *stuffing*, verificando-se os octetos originais de dados e *BCC2* no vetor *DataBCC2_DE*, cuja memória é depois realocada para a estritamente necessária..

A função *BCC2_verify* (Fragmento de código 9) utiliza o vetor *DataBCC2_DE* para calcular o *byte BCC2* com os dados recebidos que, no fim, compara com a última posição de *DataBCC2_DE* onde se encontra o valor de *BCC2* original calculado pelo transmissor. Se esta função não verificar a sua igualdade (Fragmento de código 10), é enviado imediatamente um *REJ*, não sendo alterado o número de sequência nem guardado os dados e a função *llread* retorna 0, visto que, o número de dados guardados é nulo.

Se for verificada a correta receção dos dados e o número de sequência indicar novos dados, estes são guardados no vetor *packet* e atualizado o número de sequência, sendo enviado *RR* e retornado o número de dados guardados. Se for verificada a correta receção de dados mas o número de sequência for repetido implica que se está a receber um pacote duplicado pelo que a informação não é guardada, logo é enviado *RR*,retornado 0 e o número de sequência é alterado ((Fragmento de código 11).

Novamente, ao se verificar um erro em alguma função é indicado erro e terminada a execução.

llclose

A função *llclose* (Fragmento de código 12), tem como objetivo o encerramento da ligação. Do lado do transmissor, começa, se defenido pelo utilizador, pela impressão dos elementos de estatística, descritos no tópico de Validação. Depois procede à declaração e envio do *frame* que vai sinalizar o encerramento, denominado *DISC_send*. De um modo semelhante a *llopen* inicia os mecanismos necessários à retransmissão e através da função *DISC_state_machine* espera a receção de *DISC*. Finalmente, após a recessão de *DISC* responde com um *UA* e termina a execução.

Do lado do receptor, são impressas as estatísticas associadas e verificada a recessão de *DISC* com *DISC_state_machine*. Finalmente, é verificada a correta receção de *UA* com *UA_state_machine* e terminado o programa.

Ao se verificar um erro em alguma função é indicado erro e terminada a execução.

Protocolo de aplicação

O protocolo de aplicação, fornecido no ficheiro *main.c*, começa por identificar o tipo de execução, como transmissor ou emissor, e define a *struct linklayer* que passa à função *llopen*. Como transmissor, abre o ficheiro a transmitir, lendo em segmentos com o tamanho máximo definido para o vetor *buffer*. Este vetor corresponde aos dados remetidos para a função *llwrite*. Este processo repete-se ciclicamente enquanto não forem lidos todos os *bytes* do ficheiro a transmitir. Por fim, utiliza *llclose* para fechar a ligação e termina a execução.

Como receptor, abre a ligação com *llopen* com os parâmetros definidos. Seguidamente, realiza a leitura dos dados transmitidos com *llread* para *buffer* que escreve no ficheiro destino, enquanto dados são lidos. Termina com o encerramento da ligação com *llclose* e acaba a execução.

Validação

Para validar a resistência a erros do código implementado, foram utilizadas diversas técnicas, manipulando o funcionamento da porta série, utilizado a ferramenta de simulação *cable.c*. Nomeadamente, a interrupção da ligação e a introdução de *noise* em diversos momentos do programa, tanto durante o estabelecimento ou encerramento de ligação como

durante a transmissão de informação ou alterando o código que define os cabeçalhos para valores errados.

Primeiramente verificou-se o funcionamento do programa nas condições ideais, ou seja, com o correto funcionamento da porta série ou *cable.c*.

Para verificar a detecção de erros tanto no estabelecimento de ligação, como no encerramento, foi utilizado o método de alteração do código, mais especificamente a modificação dos cabeçalhos dos frames: *SET, UA* e *DISC*. Assim sendo, foi possível verificar o funcionamento dos mecanismos de *timeout*, número de tentativas e da retransmissão de *SET* ou *DISC* e das máquinas de estado associadas. Verificou-se o não estabelecimento ou encerramento de ligação, como esperado, uma vez que, os cabeçalhos foram propositalmente colocados com valores errados, logo nunca reconhecidos como corretos. Num outro teste, introduziu-se *noise* até indicado um *timeout*, tanto no *cable.c como* na porta série, e verificou-se o sucesso no estabelecimento e encerramento de ligação, uma vez que, os *frames* são recebidos sem erros após a desativação do *noise*.

Seguidamente, para validar a transmissão de informação a ligação foi interrompida, tanto no *cable.c* como na porta série, verificando-se o correto funcionamento do *timeout,* a retransmissão do *frame* de informação, número de tentativas e envio do próximo pacote após excedido o número de tentativas. Num outro teste, foram introduzidos erros, tanto no *cable.c*.como na porta série, sendo possível comprovar o funcionamento correto dos processos de verificação de dados, associados ao byte *BCC2*, envio de *RR* ou *REJ* e atualização de números de sequência, necessários para a detecção de *frames* duplicados.
Num teste seguinte foi introduzido *noise* e interrompida a ligação em diversos momentos, verificando-se a receção de todos os dados.

Por fim, o último teste realizado consistiu no envio de diferentes ficheiros do tipo *.gif*. O envio de outros ficheiros permitiu verificar o funcionamento do código desenvolvido para ficheiros com detalhe e tamanhos superiores ao ficheiro de teste fornecido, incluindo em *.gif* animados.

Todos os testes mencionados anteriormente foram executados e repetidos várias vezes para garantir a certeza do desempenho correto do código implementado.

Elementos de valorização

Em primeiro lugar, os parâmetros de conexão utilizados foram obtidos através da utilização da s*truct linklayer*. Para a obtenção do parâmetro *baudRate*, foi implementada a função get baud, que converte um número inteiro correspondente à *Baudrate* no formato necessário à variável *newtio.c_flag*, de tipo *struct termios* .Quanto aos outros parâmetros, nomeadamente, o número máximo de retransmissões e o intervalo de tempo de *timeout* , obtidos a partir da *struct linklaye*r, foram implementadas variáveis globais de modo a guardar estes valores e utilizá-los noutras funções, uma vez que a *struct linklaye*r apenas estava disponível na função *llopen* (Fragmentos de código 2 e 3). O tamanho máximo do vetor de dados a transmitir encontrava-se definido no ficheiro *.o*, logo acessível em todo o programa. Assim sendo, foi implementada a seleção de parâmetros de conexão por parte do utilizador.

Em relação à geração de erros aleatória, entendeu-se não ser relevante a implementação de funções que gerassem erros, uma vez que, o ficheiro *cable.c* já tinha esta funcionalidade. No entanto, se tal fosse necessário, bastava criar uma função capaz de simplesmente alterar o valor de *BCC2* ou o valor dos dados lidos na função *llread*, associado a uma variável pseudo-aleatória ou probabilidade definida.

A implementação do frame de resposta *REJ* foi realizada através da utilização da máquina de estados *RR_REJ_state_machine*, capaz de distinguir se foi recebido um *RR* ou *REJ.* O envio de *REJ* acontecia quando a função *BCC_verify* (Fragmento de código 9) deteta a ocorrência e de erros na informação recebida pelo receptor. O envio e consequente recepção deste frame, implica a retransmissão imediata do *frame* enviado, por parte do transmissor, aumentando a eficiência, visto que não espera pelo fim do *timeout*.

Finalmente, na função *llclose* (Fragmento de código 12) foram imprimidas e registadas as estatísticas consideradas relevantes, tanto da parte do transmissor como do recetor: *statNum_retransmitted; statNum_recieved; statNum_timeout; statNum_RRsend; statNum_RRrecieve; statNum_REJsend; statNum_REJrecieve; error_read; error_write; statRead_0*. O método de seleção das estatísticas baseou-se na escolha daquelas ser úteis à visualização e demonstração do funcionamento do programa e possíveis erros.

É possível verificar os testes realizados no Anexo 4 - Testas

Destacam-se, então, a estatística *statRead_0*, que indica quantas vezes a função *read* não lê conteúdo, ou seja, quantos instantes não existia informação na porta-série. A estatística *statNum_REJsend* é também importante pois permitia determinar o número de vezes que a informação chegava ao receptor afetada por erros.

Conclusões

Através da realização deste trabalho foi possível o desenvolvimento de um protocolo de ligação entre dois sistemas via porta série, com protocolo *Stop-and-Wait*, resistente a erros de transmissão. Foi realizado o estabelecimento de ligação com parâmetros definidos, nomeadamente, o tempo de *timeout* e número de tentativas. Procedeu-se ao cálculo do *byte* de verificação de dados e *stuffing*, realizados em sequência, aumentando a eficiência, assim como utilizados números de sequência de modo a assegurar a receção da totalidade dos dados. Foi ainda implementado o *frame REJ* de modo a aumentar a eficiência e realizada uma testagem rigorosa, tanto em laboratório como através do ficheiro *cable.c*, de modo a assegurar o correto funcionamento do código.

O projeto foi realizado com sucesso, destacando-se o aumento nas competências associadas, nomeadamente, de programação, abordagem de problemas e *debugging* e implementação de protocolos complexos. Em suma, compreendemos agora seguramente melhor como é realizada a transferência de informação através da camada de ligação e quais os algoritmos associados aos protocolos de ligação.

Anexo 1 - Código fonte

```
//FEUP 2021/2022 - Licenciatura em Engenharia Electrotécnica e de
Computadores - Redes de Computadores
//Duarte Ribeiro Afonso Branco                    up201905327
//Pedro Afonso da Silva Correia de Castro Lopes up201907097

#include "linklayer.h"

#define FLAG 0x7E
#define A 0x03

#define SET_C 0x03

#define UA_C 0x07

#define I_C0 0x00
#define I_C1 0x02

#define RR_C0 0x01
#define RR_C1 0x21

#define REJ_C0 0x05
#define REJ_C1 0x25

#define C_DISC 0x0B

#define ESC1 0x7D
#define ESC2 0x5E
#define ESC3 0x5D

int fd, alarm_stop=0, conta=1, num_tries=0, time_out=0,
sequence_number_transmitter=0, sequence_number_reciever=0,
    sequence_number_old_reciever=1, PACKAGE_NUM_SEND=0,
PACKAGE_NUM_RECIEVE=1, role=0, statNum_retransmitted=0,
    statNum_recieved=0, statNum_timeOut=0, statNum_RRsend=0,
statNum_RRrecieve=0, statNum_REJsend=0, statNum_REJrecieve=0,
error_read=0, error_write=0, statRead_0=0;

int get_baud(int baud)
{
    switch (baud) {
    case 9600:
        return B9600;
```

```
    case 19200:
        return B19200;
    case 38400:
        return B38400;
    case 57600:
        return B57600;
    case 115200:
        return B115200;
    case 230400:
        return B230400;
    case 460800:
        return B460800;
    case 500000:
        return B500000;
    case 576000:
        return B576000;
    case 921600:
        return B921600;
    case 1000000:
        return B1000000;
    case 1152000:
        return B1152000;
    case 1500000:
        return B1500000;
    case 2000000:
        return B2000000;
    case 2500000:
        return B2500000;
    case 3000000:
        return B3000000;
    case 3500000:
        return B3500000;
    case 4000000:
        return B4000000;
    default:
        return -1;
    }
}

void atende()
{
  //printf("Alarme # %d\n", conta);
  alarm_stop=1;
```

```
  conta++;
  statNum_timeOut++;
  return;
}

int SET_state_machine(int state, unsigned char pos)
{
  switch (state)
  {
  case 0:
    if(pos==FLAG) state=1;
    break;

  case 1:
    if(pos==A) state=2;
    else if(pos==FLAG) state=1;
    else state=0;
    break;

  case 2:
    if(pos==SET_C) state=3;
    else if(pos==FLAG) state=1;
    else state=0;
    break;

  case 3:
    if(pos==A^SET_C) state=4;
    else if(pos==FLAG) state=1;
    else state=0;
    break;

  case 4:
    if(pos==FLAG) state=5;
    else state=0;
    break;

  case 5:
    break;
  }
  return state;
}

int UA_state_machine(int state, unsigned char pos)
```

```
{
  switch (state)
  {
  case 0:
    if(pos==FLAG) state=1;
    break;

  case 1:
    if(pos==A) state=2;
    else if(pos==FLAG) state=1;
    else state=0;
    break;

  case 2:
    if(pos==UA_C) state=3;
    else if(pos==FLAG) state=1;
    else state=0;
    break;

  case 3:
    if(pos==A^UA_C) state=4;
    else if(pos==FLAG) state=1;
    else state=0;
    break;

  case 4:
    if(pos==FLAG) state=5;
    else state=0;
    break;

  case 5:
    break;
  }
  return state;
}

int llopen(linkLayer connectionParameters)
{
  int res=0, res_old=0, state=0, attempt=0;
  struct termios oldtio, newtio;

  fd = open(connectionParameters.serialPort, O_RDWR | O_NOCTTY);
  if(fd<0) return -1;
```

```
  if(tcgetattr(fd,&oldtio)==-1) // save fd to oldtio
  {
      perror("tcgetattr");
      return -1;
  }

  bzero(&newtio, sizeof(newtio)); // delete sizeof(newtio) bytes in
pointer &newtio
  newtio.c_cflag = get_baud(connectionParameters.baudRate) | CS8 |
CLOCAL | CREAD;
  //newtio.c_cflag = CS8 | CLOCAL | CREAD;
  newtio.c_iflag = IGNPAR;
  newtio.c_oflag = 0;

  // set input mode (non-canonical, no echo,...)
  newtio.c_lflag = 0;

  newtio.c_cc[VTIME]    = 1;   // inter-character timer
  newtio.c_cc[VMIN]     = 0;   // blocking read until x chars received

  tcflush(fd, TCIOFLUSH);  //discards data in fd, flushes both data
received but not read and data written but not transmitted

  if (tcsetattr(fd,TCSANOW,&newtio)==-1) // save newtio to fd
  {
      perror("tcsetattr");
      return -1;
  }

  //printf("New termios structure set\n");

  num_tries=connectionParameters.numTries;
  time_out=connectionParameters.timeOut;
  role=connectionParameters.role;

  if(connectionParameters.role==TRANSMITTER)
  {
      unsigned char SET_send[5], UA_recieve;

      (void) signal(SIGALRM, atende);

      SET_send[0] = FLAG;
```

```
      SET_send[1] = A;
      SET_send[2] = SET_C;
      SET_send[3] = A^SET_C;
      SET_send[4] = FLAG;

      while(attempt<connectionParameters.numTries)
      {
          res = write(fd,SET_send,5);
          if(res<=0) {printf("\n\n----------------ERROR: write()
failed\n"); error_write++; return-1;}

          alarm(connectionParameters.timeOut);
          alarm_stop=0;

          state=0;
          res=0;
          res_old=0;

          while(alarm_stop==0 && state!=5)
          {
            res+=read(fd, &UA_recieve, 1);
            if(res<res_old) {printf("\n\n----------------ERROR: read()
failed\n"); error_read++; return-1;}
            res_old=res;

            state=UA_state_machine(state, UA_recieve);
          }

          if(state==5)
          {
            alarm(0);
            //printf("\n\n----------------GREAT SUCCESS: Connection
opened, recieved UA correctly\n");
            return 1;
          }
          attempt++;
      }
  }
  else if(connectionParameters.role==RECEIVER)
  {
      unsigned char SET_recieve, UA_send[5];
      res=0; res_old=0; state=0;
```

```
      while(state!=5)
      {
          res+=read(fd, &SET_recieve, 1);
          if(res<res_old) {printf("\n\n----------------ERROR: read()
failed\n"); error_read++; return-1;}
          res_old=res;

          state=SET_state_machine(state, SET_recieve);
      }

      if(state==5)
      {
          UA_send[0] = FLAG;
          UA_send[1] = A;
          UA_send[2] = UA_C;
          UA_send[3] = A^UA_C;
          UA_send[4] = FLAG;

          res = write(fd,UA_send,5);
          if(res<0) {printf("\n\n----------------ERROR: write()
failed\n"); error_write++; return-1;}
          return 1;
      }
    }
    else {printf("\n\n----------------ERROR: Wrong role\n"); return-1;}
}

int I_state_machine(int state, unsigned char pos)
{
  switch (state)
  {
  case 0:
    if(pos==FLAG) state=1;
    break;

  case 1:
    if(pos==A) state=2;
    break;

  case 2:
    if(pos==SET_C) state=30;
    else if((pos==I_C0 && sequence_number_reciever==0) || (pos==I_C1 &&
sequence_number_reciever==1)) state=3;
```

```
    else if(pos==FLAG) state=1;
    else state=0;
    break;

  case 3:
    if((pos==A^I_C0 && sequence_number_reciever==0) || (pos==A^I_C1 &&
sequence_number_reciever==1)) state=4;
    else if(pos==FLAG) state=1;
    else state=0;
    break;

  case 30:
    if(pos==A^SET_C) state=40;
    else if(pos==FLAG) state=1;
    else state=0;

  case 4:
    if(pos==FLAG) state=5;
    break;

  case 40:
    if(pos==FLAG) state=50;
    else if(pos==FLAG) state=1;
    else state=0;

  case 5:
    break;

  case 50:
    if(pos==FLAG) state=1;
    else state=0;
    break;
  }

  return state;
}

int RR_REJ_state_machine(int state, unsigned char pos)
{
  switch (state)
  {
  case 0:
    if(pos==FLAG) state=1;
```

```
    break;

  case 1:
    if(pos==A) state=2;
    else if(pos==FLAG) state=1;
    else state=0;
    break;

  case 2:
    if((pos==RR_C0 && sequence_number_transmitter==0) || (pos==RR_C1 &&
sequence_number_transmitter==1)) state=3;
    else if((pos==REJ_C0 && sequence_number_transmitter==1) ||
(pos==REJ_C1 && sequence_number_transmitter==0)) state=30;
    else if(pos==FLAG) state=1;
    else state=0;
    break;

  case 3:
    if((pos==A^RR_C0 && sequence_number_transmitter==0) ||
(pos==A^RR_C1 && sequence_number_transmitter==1)) state=4;
    else if(pos==FLAG) state=1;
    else state=0;
    break;

  case 30:
    if((pos==A^REJ_C0 && sequence_number_transmitter==1) ||
(pos==A^REJ_C1 && sequence_number_transmitter==0)) state=40;
    else if(pos==FLAG) state=1;
    else state=0;
    break;

  case 4:
    if(pos==FLAG) state=5;
    else state=0;
    break;

  case 40:
    if(pos==FLAG) state=50;
    else state=0;
    break;

  case 5:
    break;
```

```
  case 50:
    break;
  }

  return state;
}

void stuffing(unsigned char buf, unsigned char *I_send, int *j)
{
  if(buf==FLAG)
  {
    I_send[*j]=ESC1;
    I_send[*j+1]=ESC2;
    *j+=2;
  }
  else if(buf==ESC1)
  {
    I_send[*j]=buf;
    I_send[*j+1]=ESC3;
    *j+=2;
  }
  else
  {
    I_send[*j]=buf;
    *j+=1;
  }
}

int DE_stuffing(unsigned char* buf, int I_recieve_size, unsigned char*
buf_DE_stuffed)
{
  int j=0;

  for(int i=4; i<I_recieve_size-1; i++) //-1 so not to DE FLAG
  {
    if(buf[i]==ESC1 && buf[i+1]==ESC2)
    {
      buf_DE_stuffed[j]=FLAG;
      i++;
    }
    else if(buf[i]==ESC1 && buf[i+1]==ESC3)
    {
```

```
      buf_DE_stuffed[j]=ESC1;
      i++;
    }
    else
    {
      buf_DE_stuffed[j]=buf[i];
    }
    j++;
  }

  return j; //j=pos next=total
}

int BCC2_calculate(unsigned char* buf, int bufSize, unsigned char
*I_send)
{
  unsigned char BCC2=0x00;
  int j=4;

  for(int i=0; i<bufSize; i++)
  {
    BCC2=BCC2^buf[i];
    stuffing(buf[i], I_send, &j);
  }

  stuffing(BCC2, I_send, &j);

  return j; //j=pos next=size
}

int BCC2_verify(unsigned char* buf, int DataBCC2_DE_size)
{
  unsigned char BCC2_verify=0x00;

  for(int i=0; i<DataBCC2_DE_size-1; i++) //-1 so not check BCC2
  {
    BCC2_verify=BCC2_verify^buf[i];
  }

  if(BCC2_verify==buf[DataBCC2_DE_size-1]) {/*printf("\nGOOD\n");*/
return 1;}
  else {/*printf("\nNOT GOOD\n");*/ return -1;}
}
```

```
int llwrite(char* buf, int bufSize)
{
  if(bufSize>MAX_PAYLOAD_SIZE) return -1;

  unsigned char *I_send, RR_REJ_recieve;
  I_send=(unsigned char *)malloc(((bufSize+1)*2+5)*sizeof(unsigned
char)); //+1 for BCC2, *2 for stuffing, +5 for header+flag
  if(I_send==NULL) return -1;

  int I_send_stuffed_size=0, res=0, res_old=0, attempt=0, state=0, j=0;

  I_send[0]=FLAG;
  I_send[1]=A;

  //printf("\n----------------\nSN_reciever_send=%d\n",
sequence_number_transmitter);
  if(sequence_number_transmitter==0) {I_send[2]=I_C0;
sequence_number_transmitter=1;}
  else if(sequence_number_transmitter==1) {I_send[2]=I_C1;
sequence_number_transmitter=0;}
  I_send[3]=A^I_send[2];

  I_send_stuffed_size=BCC2_calculate(buf, bufSize, I_send);

  //printf("I_send_stuffed_size=%d\n", I_send_stuffed_size);
  I_send[I_send_stuffed_size]=FLAG;
  I_send_stuffed_size++;
  //printf("bufSize=%d\n", bufSize);
  //printf("I_send_tuffed_size=%d\n", I_send_stuffed_size);

  I_send=(unsigned char *)realloc(I_send, I_send_stuffed_size);
  if(I_send==NULL) return -1;

  PACKAGE_NUM_SEND++;

  (void) signal(SIGALRM, atende);
  conta=1;
  attempt=0;

  while(attempt<num_tries)
  {
```

```
    //printf("-----------------------------------------Attempt=%d\n",
attempt);
    res = write(fd, I_send, I_send_stuffed_size);
    if(res<0) {printf("\n\n----------------ERROR: write() failed\n");
error_write++; return-1;}
    //printf("PACKAGE_NUM_SEND=%d\n", PACKAGE_NUM_SEND);
    //printf("Attempt=%d\n", attempt);

    alarm(time_out);
    alarm_stop=0;

    state=0;
    j=0;
    res=0;
    res_old=0;

    while(alarm_stop==0 && state!=5 && state!=50)
    {
      res+=read(fd, &RR_REJ_recieve, 1);
      if(res<res_old) {printf("\n\n----------------ERROR: read()
failed\n"); error_read++; return-1;}
      res_old=res;

      //printf("\n\nState_initial: %d\n", state);
      //printf("RR_REJ_recieve[%d]=%x\n", j, RR_REJ_recieve);

      state=RR_REJ_state_machine(state, RR_REJ_recieve); //maquina de
estados que ve rr ou rej

      //printf("State_final: %d\n", state);

      j++;
    }
    if(state==5) {statNum_RRrecieve++; break;}
    else if(state==50) {/*printf("REJ\n");*/ statNum_REJrecieve++;}

    attempt++;
    statNum_retransmitted++;
  }

  if(state==50 && sequence_number_transmitter==0)
{/*printf("RESEND\n");*/ sequence_number_transmitter=1;}
```

```
  else if(state==50 && sequence_number_transmitter==1)
{/*printf("RESEND\n");*/ sequence_number_transmitter=0;}
  //printf("SN_reciever_recieve=%d\n", sequence_number_transmitter);

  alarm(0);
  return I_send_stuffed_size;
}

int llread(char* packet)
{
  int res=0, res_old=0, state=0, I_recieve_size=0, DataBCC2_DE_size=0;

  unsigned char *I_recieve;
  I_recieve=(unsigned char
*)malloc((MAX_PAYLOAD_SIZE+5)*2*sizeof(unsigned char));
  if(I_recieve==NULL) return -1;

  statNum_recieved++;

  //printf("\nState_incial: %d size=%d\n", state, I_recieve_size);
  //printf("\n----------------\n");
  //printf("PACKAGE_NUM_RECIEVE=%d\n", PACKAGE_NUM_RECIEVE);
  while(state!=5)
  {
      if(state==0) I_recieve_size=0;
      else if(state==1) I_recieve_size=1;

      res+=read(fd, &I_recieve[I_recieve_size], 1);
      if (res>res_old)
      {
        //printf("------------\n");
        //printf("\n\nState_initial: %d\n", state);
        //printf("I_recieve[%d]=%x\n", I_recieve_size,
I_recieve[I_recieve_size]);
        state=I_state_machine(state, I_recieve[I_recieve_size]/*,
&I_recieve_size,*/);
        //printf("State_final: %d\n", state);

        if(state==50)
        {
          //printf("\nSEND UA\n");
          unsigned char UA_send[5];
          int res_UA=0;
```

```
            UA_send[0] = FLAG;
            UA_send[1] = A;
            UA_send[2] = UA_C;
            UA_send[3] = A^UA_C;
            UA_send[4] = FLAG;

            res_UA = write(fd,UA_send,5);
            if(res_UA<0) {printf("\n\n----------------ERROR: write()
failed\n"); error_write++; return-1;}
          }

          if(I_recieve_size<((MAX_PAYLOAD_SIZE+5)*2)) I_recieve_size++;
//in the end=final flag
        }
        else if(res==res_old) {/*printf(".");*/ statRead_0++;}
        else if(res<res_old) {printf("\n\n----------------ERROR: read()
failed\n"); error_read++; return-1;}

        res_old=res;
    }
    //printf("\n");
    //printf("State_final: %d size=%d\n", state, I_recieve_size);
    //printf("SN_reciever_inicial=%d\n", sequence_number_reciever);
    //printf("I_recieve_size=%d\n", I_recieve_size);

    I_recieve=(unsigned char *)realloc(I_recieve, I_recieve_size);
    if(I_recieve==NULL) return -1;

    unsigned char *DataBCC2_DE, RR_REJ_send[5]={};
    DataBCC2_DE=(unsigned char *)malloc((I_recieve_size-
4)*sizeof(unsigned char));
    if(DataBCC2_DE==NULL) return -1;

    DataBCC2_DE_size=DE_stuffing(I_recieve, I_recieve_size, DataBCC2_DE);
// I_size_DE is total not pos

    DataBCC2_DE=(unsigned char *)realloc(DataBCC2_DE, DataBCC2_DE_size);
    if(DataBCC2_DE==NULL) return -1;

    //BCC2_verify(DataBCC2_DE, DataBCC2_DE_size);
```

```
  if(BCC2_verify(DataBCC2_DE, DataBCC2_DE_size)==-1) //dont save data,
flip sequence number, send REJ
  {
    //printf("SEND REJ\n");
    //printf("SN_reciever_recieve=%d\n", sequence_number_reciever);

    RR_REJ_send[0]=FLAG;
    RR_REJ_send[1]=A;

    if(sequence_number_reciever==0) RR_REJ_send[2]=REJ_C0;
    else RR_REJ_send[2]=REJ_C1;

    RR_REJ_send[3]=A^RR_REJ_send[2];
    RR_REJ_send[4]=FLAG;

    res = write(fd,RR_REJ_send,5);
    if(res<=0) {printf("\n\n----------------ERROR: write() failed\n");
error_write++; return-1;}
    statNum_REJsend++;

    return 0;
  }
  else //check sequence number, if new (save data, flip sequence
number), send RR
  {
    //printf("SEND RR\n");
    PACKAGE_NUM_RECIEVE++;
    //printf("SN_reciever_recieve=%d\n", sequence_number_reciever);

    if(sequence_number_reciever!=sequence_number_old_reciever) // if
new save data flip sequence number
    {
        for(int i=0; i<DataBCC2_DE_size-1; i++)
packet[i]=DataBCC2_DE[i];
        sequence_number_old_reciever=sequence_number_reciever;

        if(sequence_number_reciever==0) sequence_number_reciever=1;
        else sequence_number_reciever=0;
    }
    else {/*printf("DUPLICATE\n");*/ DataBCC2_DE_size=1;} // if not new
dont save flip sequence number return 0

    RR_REJ_send[0]=FLAG;
```

```
    RR_REJ_send[1]=A;

    if(sequence_number_reciever==0) RR_REJ_send[2]=RR_C0;
    else  RR_REJ_send[2]=RR_C1;

    RR_REJ_send[3]=A^RR_REJ_send[2];
    RR_REJ_send[4]=FLAG;

    res = write(fd,RR_REJ_send,5);
    if(res<0) {printf("\n\n----------------ERROR: write() failed\n");
error_write++; return-1;}

    statNum_RRsend++;

    return (DataBCC2_DE_size-1); //-1 cuz BCC2
  }

}

int DISC_state_machine(int state, unsigned char pos)
{
  switch (state)
  {
  case 0:
    if(pos==FLAG)state=1;
    break;

  case 1:
    if(pos==A) state=2;
    else state=1;
    break;

  case 2:
    if(pos==C_DISC) state=3;
    else state=0;
    break;

  case 3:
    if(pos==A^C_DISC) state=4;
    else state=0;
    break;

  case 4:
```

```
    if(pos==FLAG) state=5;
    else state=0;
    break;

  case 5:
    break;
  }

  return state;
}

int llclose(int showStatistics)
{
  int state=0, res=0, attempt=0;
  unsigned char DISC_send[5]={}, DISC_recieve, UA_send[5]={},
UA_recieve;

  DISC_send[0]=FLAG;
  DISC_send[1]=A;
  DISC_send[2]=C_DISC;
  DISC_send[3]=A^C_DISC;
  DISC_send[4]=FLAG;

  if(role==TRANSMITTER)
  {
    if(showStatistics==1)
    {
      printf("statNum_retransmitted=%d\n", statNum_retransmitted);
      printf("statNum_timeOut=%d\n", statNum_timeOut);
      printf("statNum_RRrecieve=%d\n", statNum_RRrecieve);
      printf("statNum_REJrecieve=%d\n", statNum_REJrecieve);
      printf("error_read=%d\n", error_read);
      printf("error_write=%d\n", error_write);
      printf("statRead_0=%d\n", statRead_0);
    }

    while(attempt<num_tries)
    {
      res=write(fd, DISC_send, 5);
      if(res<=0) {error_write++; return -1;}

      alarm(time_out);
      alarm_stop=0;
```

```
    state=0;
    res=0;

    while(alarm_stop==0 && state!=5)
    {
      res=read(fd, &DISC_recieve, 1);
      if(res>0) state=DISC_state_machine(state, DISC_recieve);
      else if(res<0) error_read++;
    }


    if(state==5)
    {
      alarm(0);
      UA_send[0] = FLAG;
      UA_send[1] = A;
      UA_send[2] = UA_C;
      UA_send[3] = A^UA_C;
      UA_send[4] = FLAG;

      res = write(fd,UA_send,5);
      if(res<=0) {error_write++; return -1;}
      return 1;
    }
    attempt++;
  }

  return -1;
}
else if(role==RECEIVER)
{
  if(showStatistics==1)
  {
    printf("statNum_recieved=%d\n", statNum_recieved);
    printf("statNum_RRsend=%d\n", statNum_RRsend);
    printf("statNum_REJsend=%d\n", statNum_REJsend);
    printf("error_read=%d\n", error_read);
    printf("error_write=%d\n", error_write);
    printf("statRead_0=%d\n", statRead_0);
  }

  while(state!=5)
```

```
    {
      res=read(fd, &DISC_recieve, 1);
      if(res>0) state=DISC_state_machine(state, DISC_recieve);
      else if(res==0) statRead_0++;
      else if(res<0) error_read++;
    }

    res=write(fd, DISC_send, 5);

    if(res<=0) {error_write++; return -1;}

    state=0;
    while(state!=5)
    {
      res=read(fd, &UA_recieve, 1);
      if(res>0) state=UA_state_machine(state, UA_recieve);
      if(res==0) statRead_0++;
      if(res==-1) error_read++;

    }

    return 1;
  }

  return -1;
}
```

Anexo 2 - Extratos de código

Fragmento de código 1

```
  int res=0, res_old=0, state=0, attempt=0;
  struct termios oldtio, newtio;

  fd = open(connectionParameters.serialPort, O_RDWR | O_NOCTTY);
  if(fd<0) return -1;

  if(tcgetattr(fd,&oldtio)==-1) // save fd to oldtio
  {
      perror("tcgetattr");
      return -1;
  }

  bzero(&newtio, sizeof(newtio)); // delete sizeof(newtio) bytes in
pointer &newtio
  newtio.c_cflag = get_baud(connectionParameters.baudRate) | CS8 |
CLOCAL | CREAD;
  //newtio.c_cflag = CS8 | CLOCAL | CREAD;
  newtio.c_iflag = IGNPAR;
  newtio.c_oflag = 0;

  // set input mode (non-canonical, no echo,...)
  newtio.c_lflag = 0;

  newtio.c_cc[VTIME]    = 1;   // inter-character timer
  newtio.c_cc[VMIN]     = 0;   // blocking read until x chars
received

  tcflush(fd, TCIOFLUSH);  //discards data in fd, flushes both data
received but not read and data written but not transmitted

  if (tcsetattr(fd,TCSANOW,&newtio)==-1) // save newtio to fd
  {
      perror("tcsetattr");
      return -1;
  }
```

Fragmento de código 2

```
 if(connectionParameters.role==TRANSMITTER)
  {
      unsigned char SET_send[5], UA_recieve;

      (void) signal(SIGALRM, atende);

      SET_send[0] = FLAG;
      SET_send[1] = A;
      SET_send[2] = SET_C;
      SET_send[3] = A^SET_C;
      SET_send[4] = FLAG;

      while(attempt<connectionParameters.numTries)
      {
          res = write(fd,SET_send,5);
          if(res<=0) {printf("\n\n----------------ERROR: write()
failed\n"); error_write++; return-1;}

          alarm(connectionParameters.timeOut);
          alarm_stop=0;

          state=0;
          res=0;
          res_old=0;

          while(alarm_stop==0 && state!=5)
          {
            res+=read(fd, &UA_recieve, 1);
            if(res<res_old) {printf("\n\n----------------ERROR:
read() failed\n"); error_read++; return-1;}
            res_old=res;

            state=UA_state_machine(state, UA_recieve);
          }

          if(state==5)
          {
            alarm(0);
```

```
            //printf("\n\n----------------GREAT SUCCESS: Connection
opened, recieved UA correctly\n");
            return 1;
          }
          attempt++;
      }
  }
```

Fragmento de código 3

```
else if(connectionParameters.role==RECEIVER)
  {
      unsigned char SET_recieve, UA_send[5];
      res=0; res_old=0; state=0;

      while(state!=5)
      {
          res+=read(fd, &SET_recieve, 1);
          if(res<res_old) {printf("\n\n----------------ERROR: read()
failed\n"); error_read++; return-1;}
          res_old=res;

          state=SET_state_machine(state, SET_recieve);
      }

      if(state==5)
      {
          UA_send[0] = FLAG;
          UA_send[1] = A;
          UA_send[2] = UA_C;
          UA_send[3] = A^UA_C;
          UA_send[4] = FLAG;

          res = write(fd,UA_send,5);
          if(res<0) {printf("\n\n----------------ERROR: write()
failed\n"); error_write++; return-1;}
          return 1;
      }
    }
```

Fragmento de código 4

```
int BCC2_calculate(unsigned char* buf, int bufSize, unsigned char
*I_send)
{
  unsigned char BCC2=0x00;
  int j=4;

  for(int i=0; i<bufSize; i++)
  {
    BCC2=BCC2^buf[i];
    stuffing(buf[i], I_send, &j);
  }

  stuffing(BCC2, I_send, &j);

  return j; //j=pos next=size
}
```

Fragmento de código 5

```
void stuffing(unsigned char buf, unsigned char *I_send, int *j)
{
  if(buf==FLAG)
  {
    I_send[*j]=ESC1;
    I_send[*j+1]=ESC2;
    *j+=2;
  }
  else if(buf==ESC1)
  {
    I_send[*j]=buf;
    I_send[*j+1]=ESC3;
    *j+=2;
  }
  else
  {
    I_send[*j]=buf;
    *j+=1;
  }
}
```

Fragmento de código 6

```
int llwrite(char* buf, int bufSize)
{
  if(bufSize>MAX_PAYLOAD_SIZE) return -1;

  unsigned char *I_send, RR_REJ_recieve;
  I_send=(unsigned char *)malloc(((bufSize+1)*2+5)*sizeof(unsigned
char)); //+1 for BCC2, *2 for stuffing, +5 for header+flag
  if(I_send==NULL) return -1;

  int I_send_stuffed_size=0, res=0, res_old=0, attempt=0, state=0,
j=0;

  I_send[0]=FLAG;
  I_send[1]=A;

  //printf("\n----------------\nSN_reciever_send=%d\n",
sequence_number_transmitter);
  if(sequence_number_transmitter==0) {I_send[2]=I_C0;
sequence_number_transmitter=1;}
  else if(sequence_number_transmitter==1) {I_send[2]=I_C1;
sequence_number_transmitter=0;}
  I_send[3]=A^I_send[2];

  I_send_stuffed_size=BCC2_calculate(buf, bufSize, I_send);

  //printf("I_send_stuffed_size=%d\n", I_send_stuffed_size);
  I_send[I_send_stuffed_size]=FLAG;
  I_send_stuffed_size++;
  //printf("bufSize=%d\n", bufSize);
  //printf("I_send_tuffed_size=%d\n", I_send_stuffed_size);

  I_send=(unsigned char *)realloc(I_send, I_send_stuffed_size);
  if(I_send==NULL) return -1;

  PACKAGE_NUM_SEND++;

  (void) signal(SIGALRM, atende);
  conta=1;
```

```
  attempt=0;

  while(attempt<num_tries)
  {
    //printf("-----------------------------------------
Attempt=%d\n", attempt);
    res = write(fd, I_send, I_send_stuffed_size);
    if(res<0) {printf("\n\n----------------ERROR: write()
failed\n"); error_write++; return-1;}
    //printf("PACKAGE_NUM_SEND=%d\n", PACKAGE_NUM_SEND);
    //printf("Attempt=%d\n", attempt);

    alarm(time_out);
    alarm_stop=0;

    state=0;
    j=0;
    res=0;
    res_old=0;

    while(alarm_stop==0 && state!=5 && state!=50)
    {
      res+=read(fd, &RR_REJ_recieve, 1);
      if(res<res_old) {printf("\n\n----------------ERROR: read()
failed\n"); error_read++; return-1;}
      res_old=res;

      //printf("\n\nState_initial: %d\n", state);
      //printf("RR_REJ_recieve[%d]=%x\n", j, RR_REJ_recieve);

      state=RR_REJ_state_machine(state, RR_REJ_recieve); //maquina
de estados que ve rr ou rej

      //printf("State_final: %d\n", state);

      j++;
    }
    if(state==5) {statNum_RRrecieve++; break;}
    else if(state==50) {/*printf("REJ\n");*/ statNum_REJrecieve++;}

    attempt++;
```

```
    statNum_retransmitted++;
  }

  if(state==50 && sequence_number_transmitter==0)
{/*printf("RESEND\n");*/ sequence_number_transmitter=1;}
  else if(state==50 && sequence_number_transmitter==1)
{/*printf("RESEND\n");*/ sequence_number_transmitter=0;}
  //printf("SN_reciever_recieve=%d\n", sequence_number_transmitter);

  alarm(0);
  return I_send_stuffed_size;
}
```

Fragmento de código 7

```
{
  int res=0, res_old=0, state=0, I_recieve_size=0,
DataBCC2_DE_size=0;

  unsigned char *I_recieve;
  I_recieve=(unsigned char
*)malloc((MAX_PAYLOAD_SIZE+5)*2*sizeof(unsigned char));
  if(I_recieve==NULL) return -1;

  statNum_recieved++;

  //printf("\nState_incial: %d size=%d\n", state, I_recieve_size);
  //printf("\n----------------\n");
  //printf("PACKAGE_NUM_RECIEVE=%d\n", PACKAGE_NUM_RECIEVE);
  while(state!=5)
  {
      if(state==0) I_recieve_size=0;
      else if(state==1) I_recieve_size=1;

      res+=read(fd, &I_recieve[I_recieve_size], 1);
      if (res>res_old)
      {
        //printf("------------\n");
        //printf("\n\nState_initial: %d\n", state);
        //printf("I_recieve[%d]=%x\n", I_recieve_size,
I_recieve[I_recieve_size]);
```

```
        state=I_state_machine(state, I_recieve[I_recieve_size]/*,
&I_recieve_size,*/);
        //printf("State_final: %d\n", state);

        if(state==50)
        {
          //printf("\nSEND UA\n");
          unsigned char UA_send[5];
          int res_UA=0;

          UA_send[0] = FLAG;
          UA_send[1] = A;
          UA_send[2] = UA_C;
          UA_send[3] = A^UA_C;
          UA_send[4] = FLAG;

          res_UA = write(fd,UA_send,5);
          if(res_UA<0) {printf("\n\n----------------ERROR: write()
failed\n"); error_write++; return-1;}
        }

        if(I_recieve_size<((MAX_PAYLOAD_SIZE+5)*2))
I_recieve_size++; //in the end=final flag
      }
      else if(res==res_old) {/*printf(".");*/ statRead_0++;}
      else if(res<res_old) {printf("\n\n----------------ERROR:
read() failed\n"); error_read++; return-1;}

      res_old=res;
  }
```

Fragmento de código 8

```
int DE_stuffing(unsigned char* buf, int I_recieve_size, unsigned
char* buf_DE_stuffed)
{
  int j=0;

  for(int i=4; i<I_recieve_size-1; i++) //-1 so not to DE FLAG
  {
    if(buf[i]==ESC1 && buf[i+1]==ESC2)
    {
```

```
      buf_DE_stuffed[j]=FLAG;
      i++;
    }
    else if(buf[i]==ESC1 && buf[i+1]==ESC3)
    {
      buf_DE_stuffed[j]=ESC1;
      i++;
    }
    else
    {
      buf_DE_stuffed[j]=buf[i];
    }
    j++;
  }

  return j; //j=pos next=total
}
```

Fragmento de código 9

```
int BCC2_verify(unsigned char* buf, int DataBCC2_DE_size)
{
  unsigned char BCC2_verify=0x00;

  for(int i=0; i<DataBCC2_DE_size-1; i++) //-1 so not check BCC2
  {
    BCC2_verify=BCC2_verify^buf[i];
  }

  if(BCC2_verify==buf[DataBCC2_DE_size-1]) {/*printf("\nGOOD\n");*/
return 1;}
  else {/*printf("\nNOT GOOD\n");*/ return -1;}
}
```

Fragmento de código 10

```
if(BCC2_verify(DataBCC2_DE, DataBCC2_DE_size)==-1) //dont save data,
flip sequence number, send REJ
  {
```

```
    //printf("SEND REJ\n");
    //printf("SN_reciever_recieve=%d\n", sequence_number_reciever);

    RR_REJ_send[0]=FLAG;
    RR_REJ_send[1]=A;

    if(sequence_number_reciever==0) RR_REJ_send[2]=REJ_C0;
    else RR_REJ_send[2]=REJ_C1;

    RR_REJ_send[3]=A^RR_REJ_send[2];
    RR_REJ_send[4]=FLAG;

    res = write(fd,RR_REJ_send,5);
    if(res<=0) {printf("\n\n----------------ERROR: write()
failed\n"); error_write++; return-1;}
    statNum_REJsend++;

    return 0;
  }
```

Fragmento de código 11

```
else //check sequence number, if new (save data, flip sequence
number), send RR
  {
    //printf("SEND RR\n");
    PACKAGE_NUM_RECIEVE++;
    //printf("SN_reciever_recieve=%d\n", sequence_number_reciever);

    if(sequence_number_reciever!=sequence_number_old_reciever) // if
new save data flip sequence number
    {
        for(int i=0; i<DataBCC2_DE_size-1; i++)
packet[i]=DataBCC2_DE[i];
        sequence_number_old_reciever=sequence_number_reciever;

        if(sequence_number_reciever==0) sequence_number_reciever=1;
        else sequence_number_reciever=0;
    }
    else {/*printf("DUPLICATE\n");*/ DataBCC2_DE_size=1;} // if not
new dont save flip sequence number return 0
```

```
    RR_REJ_send[0]=FLAG;
    RR_REJ_send[1]=A;

    if(sequence_number_reciever==0) RR_REJ_send[2]=RR_C0;
    else  RR_REJ_send[2]=RR_C1;

    RR_REJ_send[3]=A^RR_REJ_send[2];
    RR_REJ_send[4]=FLAG;

    res = write(fd,RR_REJ_send,5);
    if(res<0) {printf("\n\n----------------ERROR: write()
failed\n"); error_write++; return-1;}

    statNum_RRsend++;

    return (DataBCC2_DE_size-1); //-1 cuz BCC2
  }
```

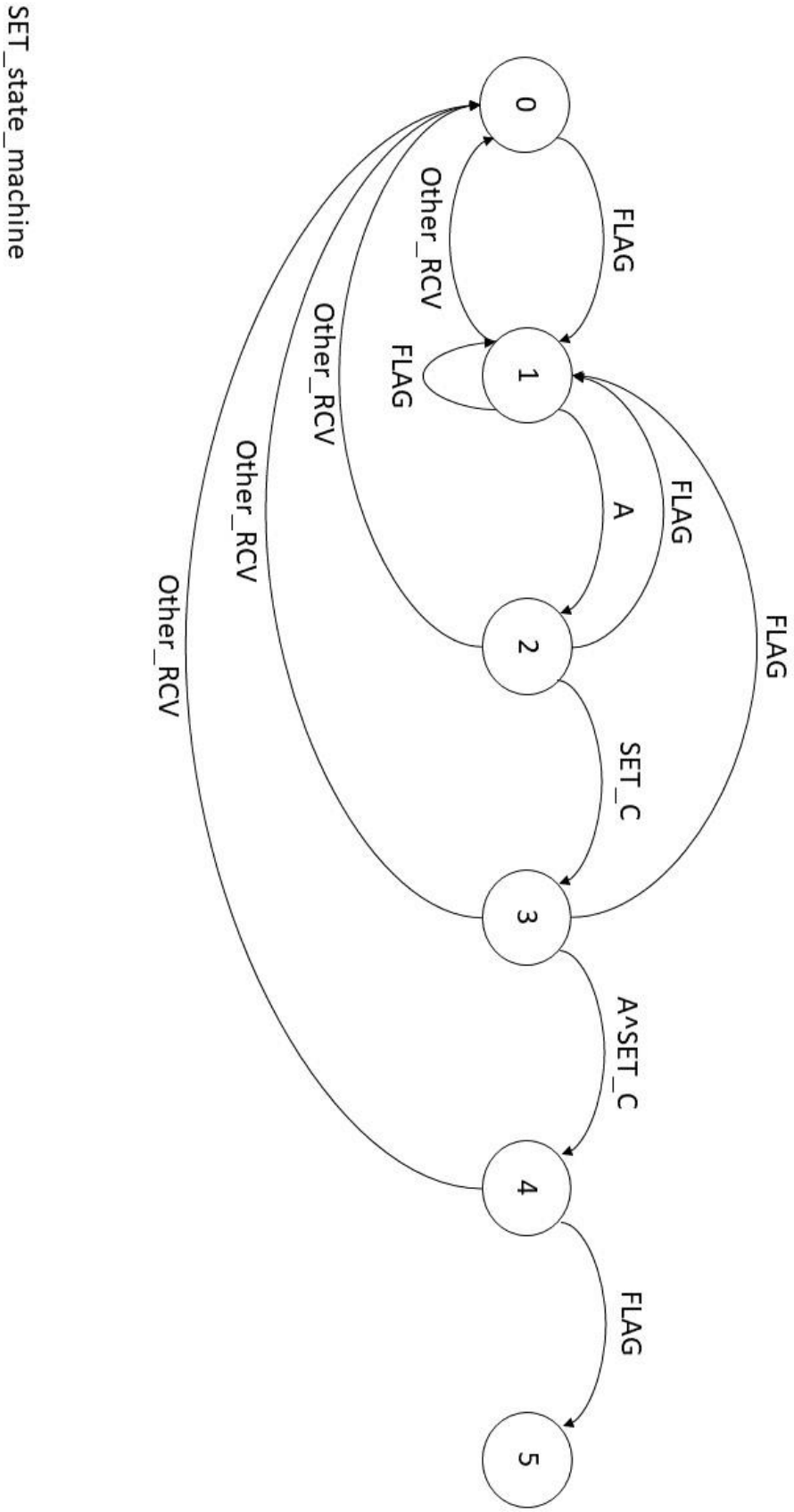
Fragmento de código 12

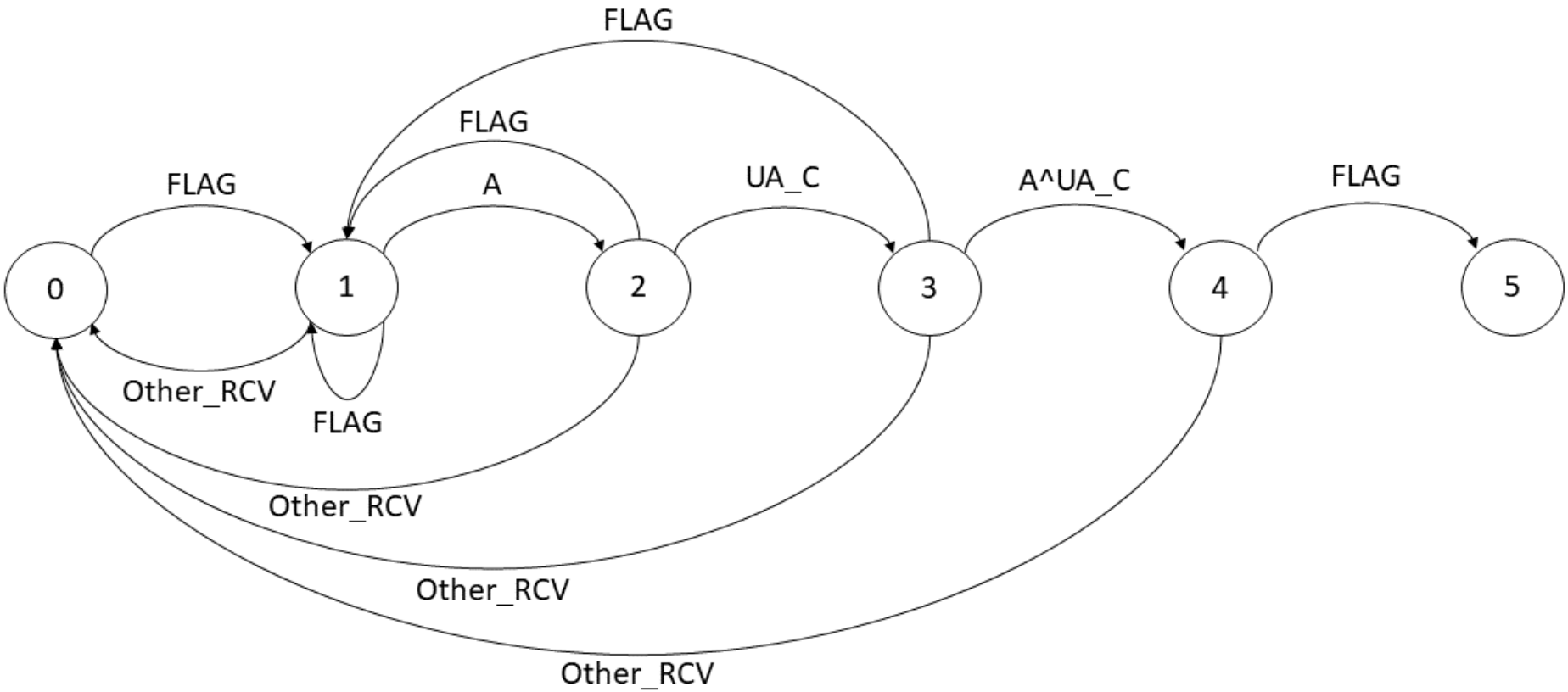
```
int llclose(int showStatistics)
{
  int state=0, res=0, attempt=0;
  unsigned char DISC_send[5]={}, DISC_recieve, UA_send[5]={},
UA_recieve;

  DISC_send[0]=FLAG;
  DISC_send[1]=A;
  DISC_send[2]=C_DISC;
  DISC_send[3]=A^C_DISC;
  DISC_send[4]=FLAG;

  if(role==TRANSMITTER)
  {
    if(showStatistics==1)
    {
      printf("statNum_retransmitted=%d\n", statNum_retransmitted);
      printf("statNum_timeOut=%d\n", statNum_timeOut);
      printf("statNum_RRrecieve=%d\n", statNum_RRrecieve);
      printf("statNum_REJrecieve=%d\n", statNum_REJrecieve);
```

```
  printf("error_read=%d\n", error_read);
  printf("error_write=%d\n", error_write);
  printf("statRead_0=%d\n", statRead_0);
}

while(attempt<num_tries)
{
  res=write(fd, DISC_send, 5);
  if(res<=0) {error_write++; return -1;}

  alarm(time_out);
  alarm_stop=0;

  state=0;
  res=0;

  while(alarm_stop==0 && state!=5)
  {
    res=read(fd, &DISC_recieve, 1);
    if(res>0) state=DISC_state_machine(state, DISC_recieve);
    else if(res<0) error_read++;
  }


  if(state==5)
  {
    alarm(0);
    UA_send[0] = FLAG;
    UA_send[1] = A;
    UA_send[2] = UA_C;
    UA_send[3] = A^UA_C;
    UA_send[4] = FLAG;

    res = write(fd,UA_send,5);
    if(res<=0) {error_write++; return -1;}
    return 1;
  }
  attempt++;
}

return -1;
```

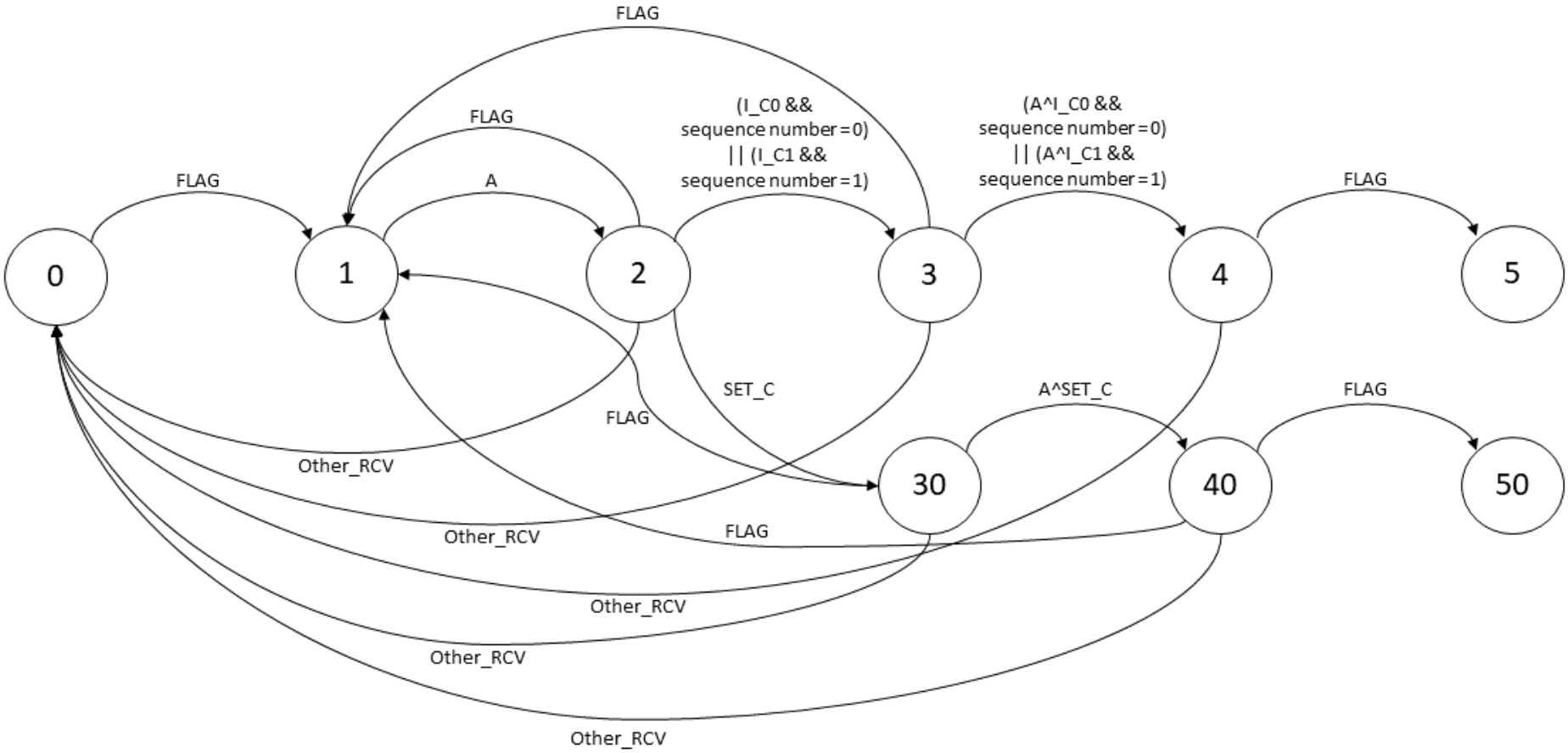
```
  }
  else if(role==RECEIVER)
  {
    if(showStatistics==1)
    {
      printf("statNum_recieved=%d\n", statNum_recieved);
      printf("statNum_RRsend=%d\n", statNum_RRsend);
      printf("statNum_REJsend=%d\n", statNum_REJsend);
      printf("error_read=%d\n", error_read);
      printf("error_write=%d\n", error_write);
      printf("statRead_0=%d\n", statRead_0);
    }

    while(state!=5)
    {
      res=read(fd, &DISC_recieve, 1);
      if(res>0) state=DISC_state_machine(state, DISC_recieve);
      else if(res==0) statRead_0++;
      else if(res<0) error_read++;
    }

    res=write(fd, DISC_send, 5);

    if(res<=0) {error_write++; return -1;}

    state=0;
    while(state!=5)
    {
      res=read(fd, &UA_recieve, 1);
      if(res>0) state=UA_state_machine(state, UA_recieve);
      if(res==0) statRead_0++;
      if(res==-1) error_read++;

    }

    return 1;
  }

  return -1;
}
```

Anexo 3 - Diagramas das máquinas de estado

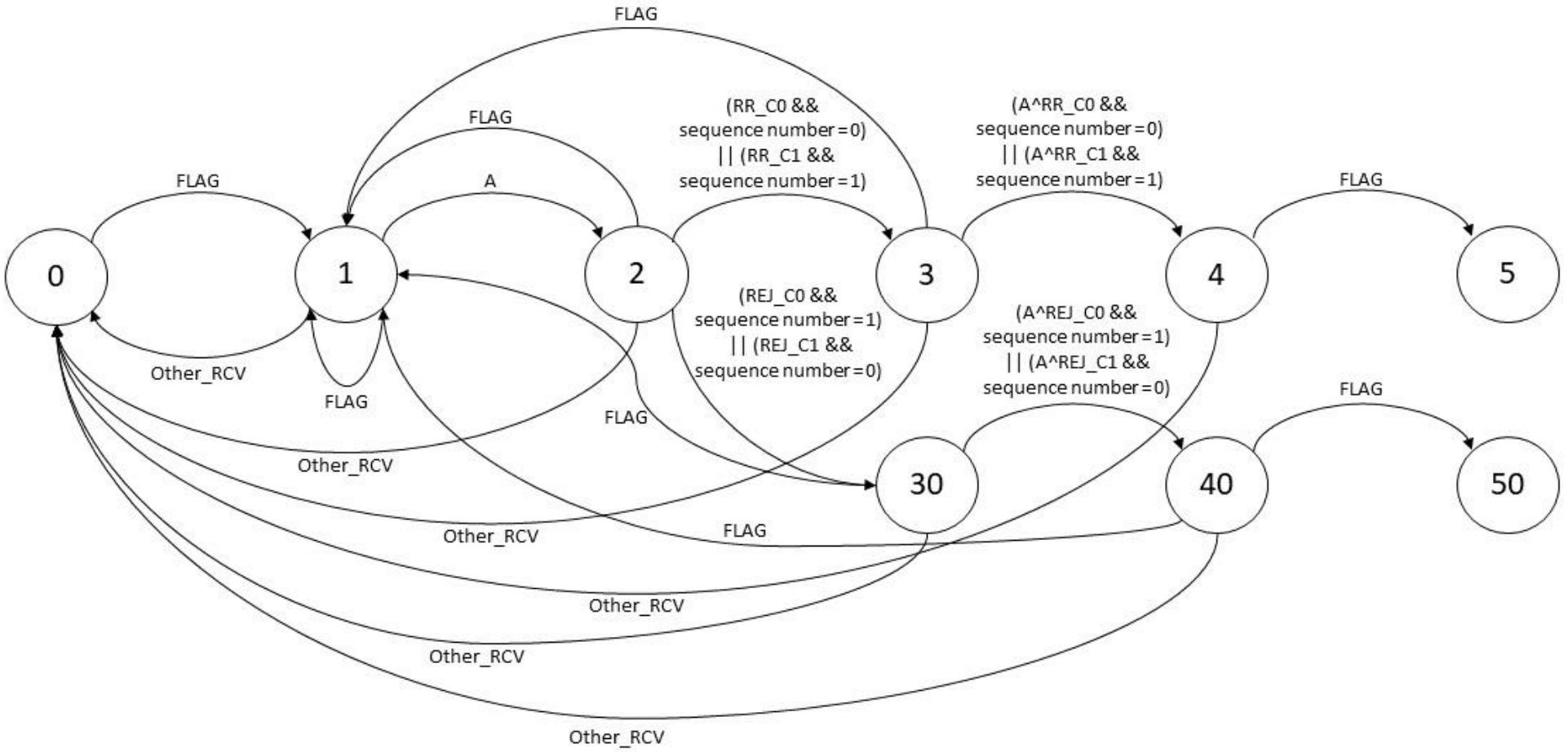




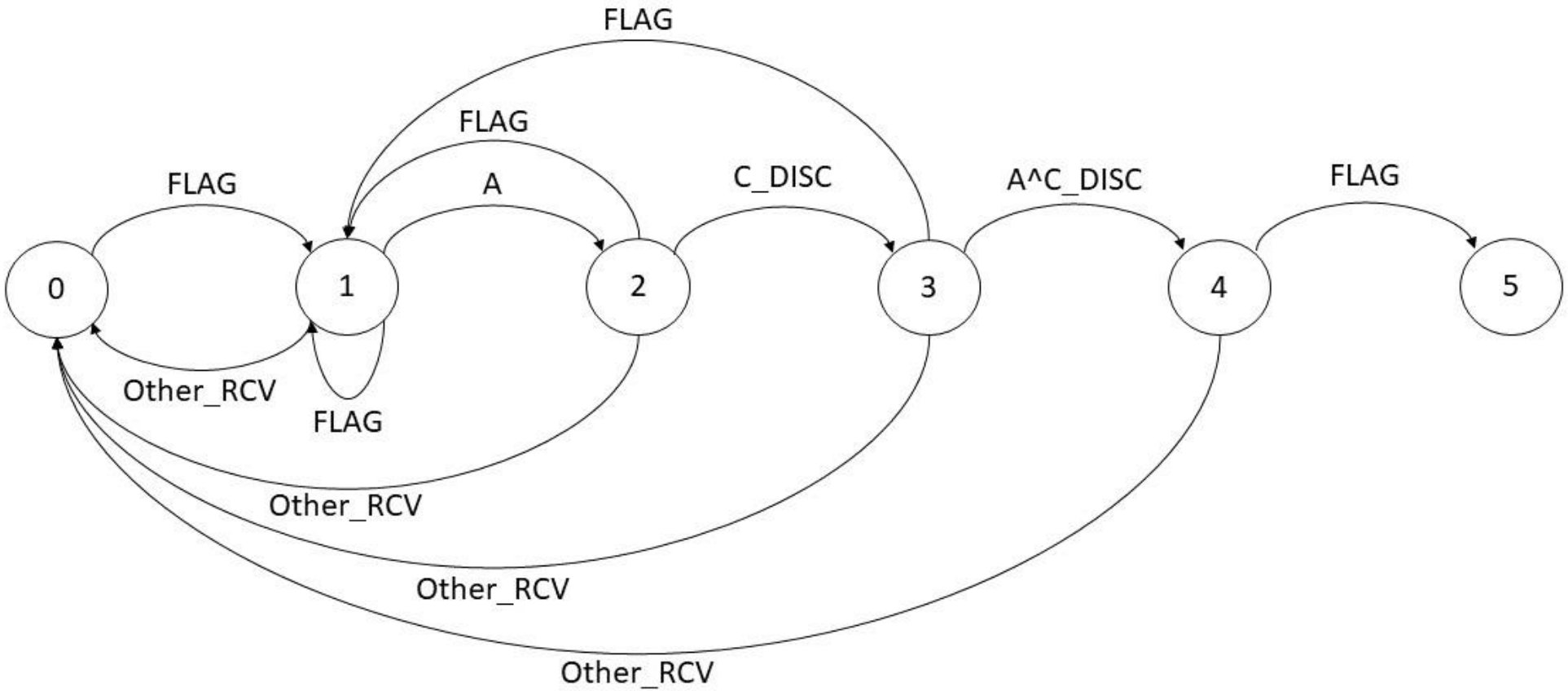
UA_state_machine



I_state_machine



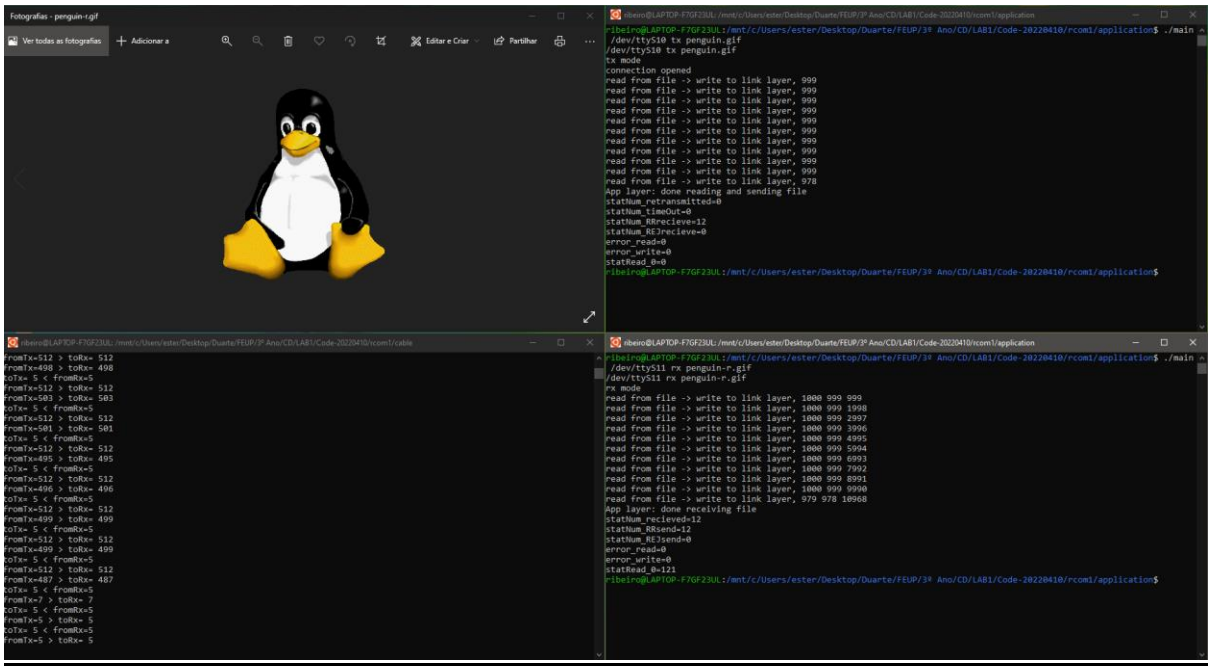
RR_REJ_state_machine



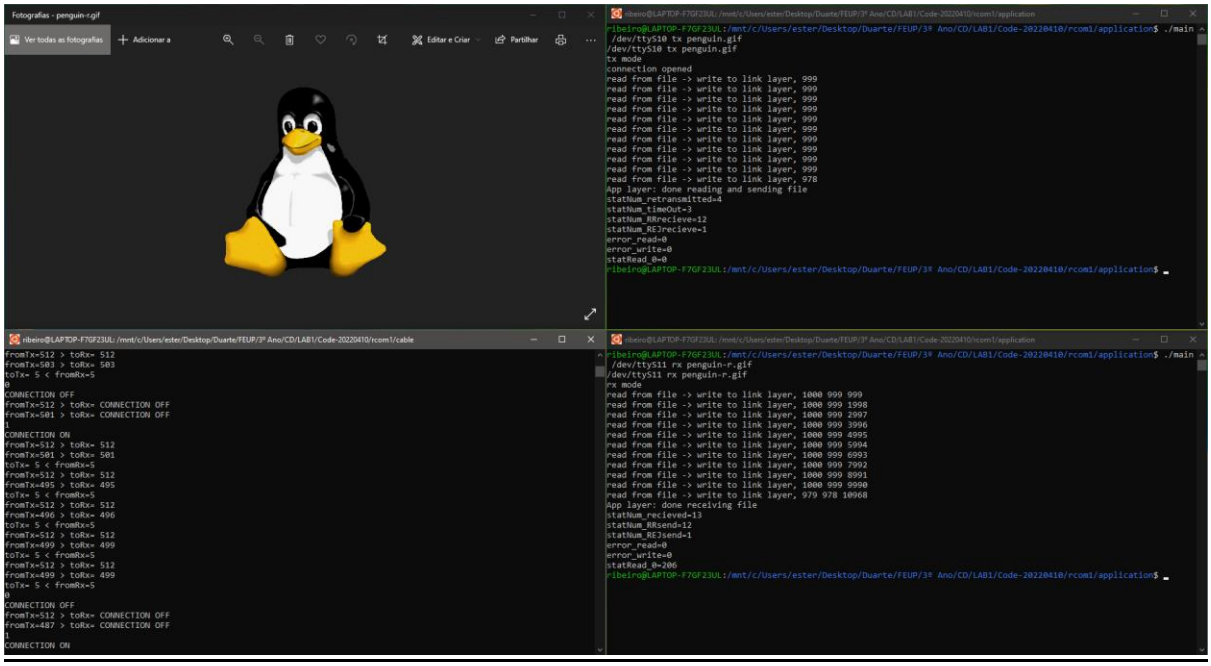
DISC_state_machine

Anexo 4 - Testes

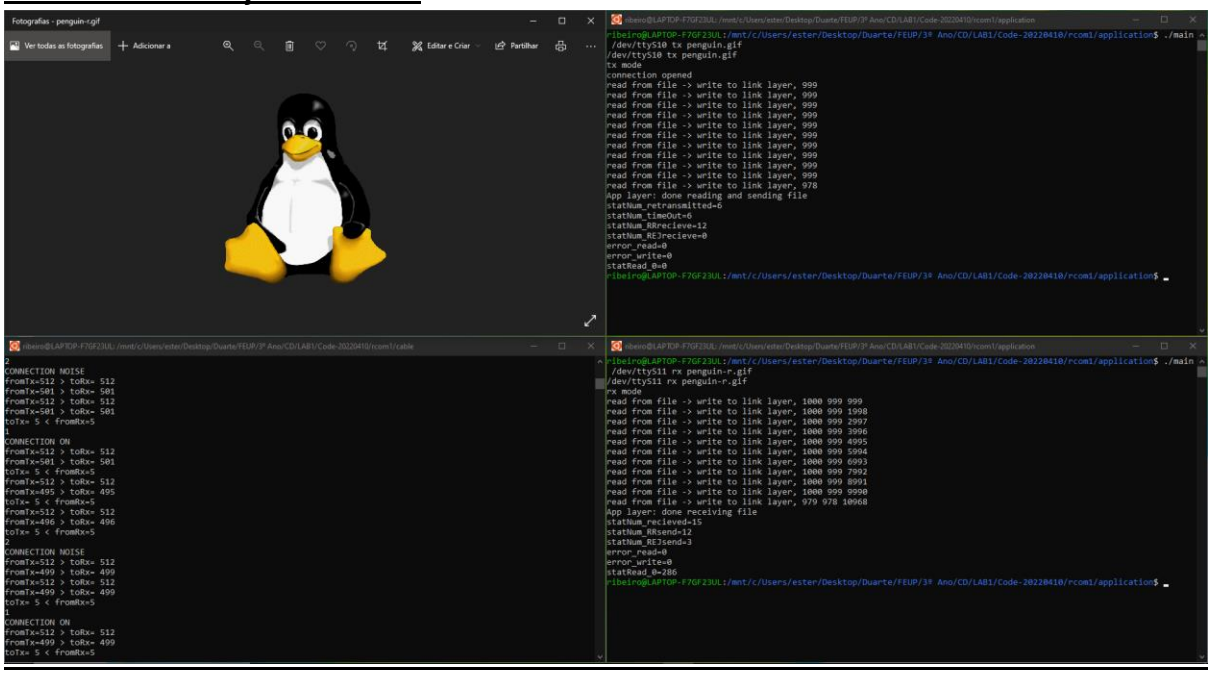
Teste 1 - Funcionamento ideal



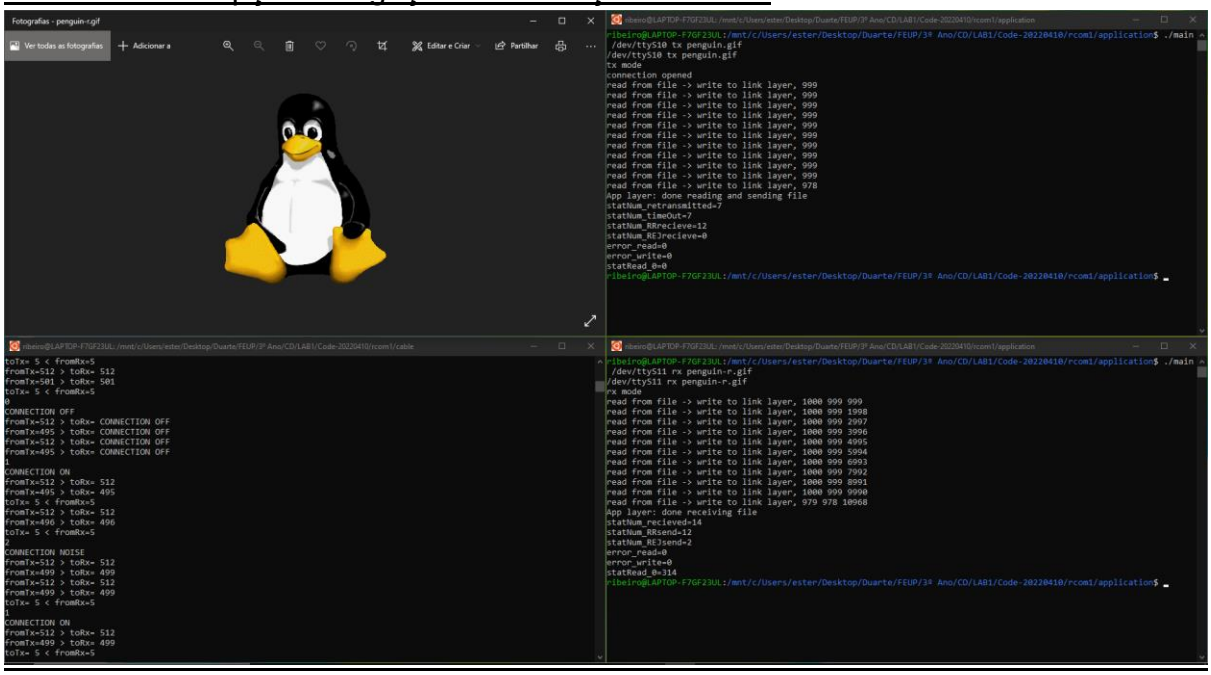
Teste 2 - Interrupção da ligação



Teste 3 - Introdução de *noise*



Teste 4 - Interrupção da ligação e introdução de *noise*



Teste 5 - Envio de ficheiro de teste de tamanho superior