Relatório do Trabalho - Parte I
(Sistemas Multiprocessados em Chip)

Manuel Adahil Muniz Osterno Marco Vinícius Matos Oliveira

03/06/2024

Resumo

Relatório sobre o trabalho desenvolvido durante o primeiro semestre do ano de 2024, para a disciplina de Sistemas Multiprocessados em Chip, sob a supervisão dos professores: **Dr. César Augusto Missio Marcon e Dr. Fernando Gehm Moraes**

Conteúdo

1 Explicação da Solução Proposta (Pt.1) 2
- 1.1 Desenvolvimento . . . 2
- 1.2 ECC (*ecc.c e ecc.h*) . . . 2
- 1.3 Pacote (*packet.c e packet.h*) . . . 7
- 1.4 Processador Alfa (*1_proc_alfa.c*) . . . 8
- 1.5 Processador Beta (*2_proc_beta.c*) . . . 9
- 1.6 Geração de Erros (*error_gen.c e error_gen.h*) . . . 9
- 1.7 Injetor de Erro (*3_inject_error.c*) . . . 12

2 Resultados 12

1. Explicação da Solução Proposta (Pt.1)

Nesta seção serão explicados os aspectos que concernem à solução desenvolvida pelos alunos da disciplina.

1.1. Desenvolvimento

Os três processos são descritos em 3 diferentes arquivos de código, *1_proc_alfa.c*, *2_proc_beta.c* e *3_inject_error.c* e fazem uso de três bibliotecas que foram desenvolvidas para auxiliar o desenvolvimento dos mesmos, *ecc.h*, *packets.h* e *error_gen.h*.

1.2. ECC (*ecc.c e ecc.h*)

O ECC proposto é o hamming extendido que calcula um número de bits de redundância de modo que seja possível "cobrir"todos os bits da palavra original e, após o cálculo desses bits, um último bit de paridade é calculado executando a operação de ou-exclusivo sobre todos os bits da palavra com os de redundância.

Para que a redundância consiga cobrir toda a palvra, é necessário que a Equação 1.1 seja respeitada, onde r é o núméto de bits de redundância e n é o número de bits da palavra.

$$2^r - 1 \geq n \tag{1.1}$$

Desse modo, para uma palavra de 32 bits, seriam necessários 6 bits de redundância e mais um de paridade, resultando em 7 bits.

Originalmente, os bits de redundância são "mesclados"à palavra, ficando em posições de potência de dois, então o primeiro bit de redundância ficaria na posição 1 (2^0), o segundo na posição 2 (2^1) e assim por diante, como mostra a Figura 1.1.

Posição do bit		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
bits codificados		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11	p16	d12	d13	d14	d15	
bits de paridade	p1	x		x		x		x		x		x		x		x		x		x		
	p2		x	x			x	x			x	x			x	x			x	x		...
	p4				x	x	x	x					x	x	x	x					x	
	p8								x	x	x	x	x	x	x	x						
	p16																x	x	x	x	x	

Figura 1.1: Organização dos bits de redundância (p*) na palavra. (Fonte: [Wik20])

Ainda na Figura 1.1 é possível observar, marcados com um X, os bits de dados que são utilizados para calcular cada um dos bits de redundância. Esses bits são selecionados considerando a lógica de usar os bits cuja o número de posição tem o "enésimo"bit menos significativo igual a 1. Por exemplo, p1 (r1) ustiliza o próprio p1 (posição 1 - 1b001), d1 (posição 3 - 'b011), d2 (posição 5 - 'b101) e assim por diante.

Desse modo, desenvolveu-se dois métodos principais para codificar (*ham_encode*) e decodificar (*ham_decode*) a palavra.

O primeiro método pode ser visto no Código 1. O mesmo retorna um *uint8_t* contendo os bits de redundância e a paridade ao final da variável como mostra a linha 7 do código. A função é genérica, desse modo, qualquer tamanho de palvra funcionaria o que facilita, pois usamos o mesmo método para codificar dois flits (64 bits) ou um flit (32 bits). O *loop* da linha 25 itera sobre todos os possíveis bits de redundância, desse modo, a cada interação deste loop, um bit de redundância será calculado, já o *loop* da linha 29 itera sobre todos os bits da palavra codificada (ou seja, o número de bits de dados e de redundância), internamente a este *loop*, se faz a checagem do bit do número da posição do bit de dados referente aquele bit redundância, ou seja, se o bit de redundância a ser codificado naquela interação é o 2, checa-se se o segunda bit da variável *i* (interador do segundo *loop*) é 1. Por fim, o *loop* mais interno da linha 31 juntamento com a checagem da linha 33 verificam por quanta redundâncias o bit atual da palavra já passou, pois a posição na palavra precisa ser calculada considerando este valor. Por exemplo, na Figura 1.1 o bit de dado D1 está na posição 3, se formos acessar apenas a plavra sem a redundância, temos que acessa a posição 1 (3-2), onde 2 é o número de redundância que já passaram quando selecionamos D1.

```
1  /**
2   * \brief Calculate the extended humming code considering the data and
       redudancy length.
3   *
4   * This function takes the data, the respective lengths and return the
       calculated ECC. All the ECCs will be
5   * stored starting from MSB of a uint8_t variable. For example, fo 3 redudancy
        bits:
6   * Position:  7  6  5  4  3  2  1  0
7   * Value    : e2 e1 e0  P  0  0  0  0
8   *
9   * \param data_in Data Input.
10  * \param nb_databits Data length in bits.
11  * \param nb_redbits Redundancy length in bits (only hamming, no parity).
12  * \return The ECC as an uint8_t type,where the first redundancy bit is in the
        MSB.
13  */
14
15 uint8_t ham_encode(uint32_t * data_in, uint8_t nb_databits, uint8_t nb_redbits
       ){
16      uint8_t ecc = 0;
17      uint8_t partial_ecc = 0;
18      uint8_t nb_totalbits = nb_databits + nb_redbits;
19      uint8_t nb_found_redbits = 0;
20      uint8_t flit = 0;
21      uint8_t p = 0;
22
23      // Calculating Hamming
24      // Iterate over the reundancy bits
25      for (uint8_t j = 0; j < nb_redbits; j++){
26          // Iterate over the entire data (redudancy + data)
27          for (uint8_t i = 1; i <= nb_totalbits; i++){
28              // Check if the referent redundancy bit is set
```

```
            if ( (i & (0b00000001 << j)) &&  (i & (~(0b00000001 << j))) ) {
                // Iterate over the power of two
                for (uint8_t k = 0; k < nb_redbits; k++) {
                    // Check it already "passed" by a redundancy
                    if (i & (0b00000001 << k)){
                        // Count the number of red bits already "passed"
                        nb_found_redbits = k +1;
                    }
                }
                flit = (uint8_t) ((i-nb_found_redbits)/32);
                partial_ecc ^= GET_D( ((i-nb_found_redbits)%32) , data_in[flit
    ]);
            }
        }
        //keep it begining from the MSB
        ecc |= partial_ecc << (8 - (nb_redbits-j));
        partial_ecc = 0;
    }

    p = parity(data_in, ecc, nb_databits, nb_redbits);

    ecc |= p << (7-nb_redbits);//keep it begining from the MSB

    return ecc;
}
```

Listing 1: Hamming Encode Method

No final do segundo *loop*, o acesso ao bit é feito e uma operação XOR é realizada e assim é feito para todos os bits de dados para cada bit de ECC, por fim, calcula-se a paridade final e esses bits de ECC junto com a paridade são armazenados em uma variável que é o retorno da função.

O segundo método implmentado é o decodificador e pode ser visto no Código 2.

```
/**
 * \brief Check and (if possible) correct errors of a data input regarding the
    ECC.
 *
 * This function takes the data, the respective lengths, the ECC and return
   the type of the error found
 * sC sP  Error
 *  0  0  No Error
 *  0  1  Single Error (SE) (on redundancy area) -> No modification
 *  1  0  Double Error (DE)
 *  1  1  Single Error (SE) (on data area) -> Correct error on data_in
 *
 * \param data Data.
 * \param ecc The previous calculated ECC as a uint8_t type, where the first
   redundancy bit is in the MSB.
 * \param nb_databits Data length in bits.
 * \param nb_redbits Redundancy length in bits (only hamming, no parity).
 * \return A error_t type indicating the error kind
```

```
 */

error_t ham_decode(uint32_t * data, uint8_t ecc, uint8_t nb_databits, uint8_t
    nb_redbits){
    uint8_t recalc_ecc;
    uint8_t synd;
    uint8_t mask = 0;
    uint8_t p;
    error_t err;

    //Auxiliary to correct error
    uint8_t nb_found_redbits = 0;
    uint8_t flit = 0;
    uint8_t position = 0;

    recalc_ecc = ham_encode(data, nb_databits, nb_redbits);

    //OBS.: The parity calculated for recalc_ecc is not correct since
    //it takes into account the just calculated ECC not the previous
    //one, so we have to calculate the correct parity
    p = parity(data, ecc, nb_databits, nb_redbits);
    recalc_ecc &= (~(0b00000001 << (7-nb_redbits)));
    recalc_ecc |= p << (7-nb_redbits);

    //Calculate Syndrome
    synd = ecc ^ recalc_ecc;

    //Building hamming redundancy mask
    for (uint8_t i = 0; i < nb_redbits; i++){
        mask |= 0b00000001 << i;
    }
    mask = mask << (8-nb_redbits);

    //Check and correct
    if (!(synd & mask) && !(synd & ~mask)){
        err = NO_ERROR;
    } else if (!(synd & mask) && (synd & ~mask)){
        err = SE;
    } else if ((synd & mask) && !(synd & ~mask)){
        err = DE;
    } else {
        err = SE;

        synd = (synd & mask) >> (8-nb_redbits);
        // Iterate over the power of two, so we can covert the struct postion
    to data position
        for (uint8_t k = 0; k < nb_redbits; k++) {
            // Check it already "passed" by a redundancy
            if (synd & (0b00000001 << k)){
                // Count the number of redundancy bits already "passed"
                nb_found_redbits = k +1;
```

```
65              }
66          }
67
68          flit = (uint8_t) ((synd-nb_found_redbits)/32);
69          // Considering MSB
70          position = (uint8_t) 32 - ((synd-nb_found_redbits)%32);
71          //Flip the bit indicated by position
72          data[flit] ^= 0x00000001 << position;
73      }
74
75      return err;
76  }
```

Listing 2: Hamming Decode Method

Basicamente, o método de decodificação recebe a palavra e o ECC calculado, e recalcula o ECC e a paridade para esta palavra, Depois é calculada a síndrome que é, de forma simples, o XOR entre o novo ECC+paridade e o velho, isso é feito entre as linhas 30 e 40. Já entre as linhas 48 e 57 é checado que tipo de problema encontrado (ou não encontrado) cosiderando a Tabela 1.1.

Tabela 1.1: Resultado de síndrome.

sC	sP	Tipo de Erro
0	0	Sem Erro
0	1	Erro Simples (Na área de Redundância)
1	0	Erro Duplo (Não corrigível)
1	1	Erro Simples (Na área de dados)

Se o erro detectado é um erro simples na área de dados, então a correção deve ser feita. A posição do bit é dado pela interpretação do resultado de síndrome como um inteiro, mas esse valor considera que os bits de ECC estão mesclados com os bits de dados, logo temos que fazer a conversão como é feito para o decodificador e que foi explicado anteriormente, depois disso, o bit desta posição é invertido. No código isso é feito entre as linhas 58 e 73.

Para mais detalhes sobre as implementações, acessar o código fonte no .zip anexo ao trabalho

1.3. Pacote (*packet.c e packet.h*)

Além da biblioteca de ECC uma biblioteca chamada *packet.h* foi criada para lidar com operações relacionadas aos pacotes, desde criar até a parte de envio e recepção.

```
typedef struct {
    uint32_t target;
    uint32_t size;
    union{
        struct {
            uint32_t service_header[SH_SIZE];
            uint32_t payload[PAYLOAD_SIZE];
        };
        uint32_t pkt_payload[SH_SIZE+PAYLOAD_SIZE];
    };
    uint8_t ecc [ECC_SIZE*4];
} packet_t;
```

Listing 3: Packet Struct

O Código 3 mostra como o acote é representado dentro do nosso projeto, é criada um *struct* C onde cada atributo é um campo do pacote, os campos referentes ao *payload* e ao *service header* são adicionados a uma *union* junto com um vetor chamado *pkt_payload* para possamos acessar esses dois campos como um só na hora de codificar/decodificar o ECC.

Além disso, dois métodos foram desenvolvidos para manipular os pacotes, um deles é o *encode_packet* responsável por receber o identificador do *target*, o número de sequência, o *payload* e, então, montar o pacote. O *payload* e *service header* são separado em M blocos de N flits (M e N depende da quantidade de flits reservados para os ECCs), onde cada bloco é separado em sub-blocos de 2 flits cadas e o ECC é calculado, quando apenas um 1 flit de dados sobra, o seu ECC é gerado individualmente. Ese método foi feito de forma a ser genérico, aceitando qualquer tamanho de *payload* e ECC.

O Segundo deles é o *decode_packet* que faz o papel inverso do anterior, este método utiliza o método *hum_decode* mostrado anteriormente e retorna um tipo chamado *pkt_error_t* que discrimina qual erro houve com aquele pacote como mostrado no Código 4.

```
typedef enum {
    NONE,
    WRONG_TARGET,
    WRONG_SEQ_NB,
    ECC_SE,
    ECC_DE,
} pkt_error_t;
```

Listing 4: Packet Error Type

Por fim, nesta biblioteca, também são implementados os métodos para envio e recepção de pacotes. Este processo de comunicação é emulado através de acessos de leitura e escrita sobre arquivos onde canais são criados e usados para a troca de mensagens. Como cada canal é um arquivo, optou-se pela a ideia de canais unidirecionais, deste modo evitando condições de corrida entre os processos.

Para mais detalhes sobre as implementações, acessar o código fonte no .zip anexo ao trabalho

1.4. Processador Alfa (*1__proc__alfa.c*)

O processador Alfa, conforme especificado na descrição da parte I do trabalho, deve realizar a transmissão de pacotes para o outro processador e a partir da resposta recebida pelo Protocolo de Controle de Transmissão. Na solução desenvolvida, são declaradas as variáveis utilizadas para geração e transmissão do pacote, além de dos dois ponteiros utilizados no código para manipulação de arquivos:

- input_fptr - Ponteiro usado para o arquivo com o texto que será codificado e enviado pelo processador;
- log_fptr - Ponteiro usado para o arquivo que contém os dados dos contadores de respostas recebidas (*ACK, ACK_ERROR, NACK*), contadores os quais são declarados logo após.

Logo após, é checado se o arquivo de entrada existe, caso o mesma não exista, é exibida uma mensagem de erro. Caso o mesmo exista, são iniciados o processo de transmissão, mais especificamente o processo de conexão do canal 0 (entre o Alfa e o Injetor de Erros) e do canal 2 (entre Beta e Alfa, onde circula o controle de transmissão).
Na próxima parte do código, é apresentado o *loop* principal para transmissão de pacotes, onde acontece (de forma sequencial):

- Codificação de cada pacote (à partir do arquivo recebido);
- Envio do pacote para o injetor de erros (por meio do canal 0);
- Novo *loop* onde o envio é feito conforme o retorno do protocolo de controle de transmissão, pois quando recebido (de Beta para Alfa):
  - NACK - Pacote deve ser enviado pelo o canal 1 e é acrescido em 1, no contador de pacotes cuja mensagem foi "NACK"
  - ACK_ERR - Pacote é dado como enviado (mesmo que com erros) e é acrescido em 1, no contador de pacotes cuja mensagem foi "ACK_ERR"
  - ACK - Pacote é dado como enviado e é acrescido em 1, no contador de pacotes cuja mensagem foi "ACK"

Observação: Apenas em casos de "NACK", o contador de pacotes que circularam o canal não é acrescido em 1. Isto ocorre quando há o "ACK", com ou sem erro.

Ao final, são fechadas as conexões dos canais e são escritos os registros que irão no arquivo onde estão contidas as contagens de cada uma das mensagens recebidas no protocolo de controle de transmissão, fechando então as funcionalidades do processador Alfa.

1.5. Processador Beta (*2_proc_beta.c*)

O processador Beta, conforme especificado na descrição da parte I do trabalho, deve realizar o recebimento dos pacotes do outro processador e enviar uma resposta do protocolo de controle de transmissão.

Na solução desenvolvida, são declaradas as variáveis utilizadas para recebimento do pacote e transmissão da resposta do protocolo de controle de transmissão, além de um ponteiro utilizado no código para manipulação de arquivos:

- output_fptr - Ponteiro usado para o arquivo onde será gravado o texto com as mensagens recebidas e decodificadas pelo processador;

Na próxima parte do código, é apresentado o *loop* principal para recepção de pacotes, onde acontece (de forma sequencial):

- É estabelecida a conexão do canal do protocolo de controle de transmissão (por meio do canal 2);
- É estabelecida a conexão do canal de recebimento de pacotes (por meio do canal 1);
- Decodificação de cada pacote (à partir do arquivo recebido);
- São abertas as condições sobre o parecer que será dado pelo protocolo de controle de transmissão, pois será enviado (de Beta para Alfa):
  - NACK - Pacote recebido tinha erro duplo ou número de sequência do pacote incorreto;
  - ACK_ERR - Pacote recebido tinha erro simples;
  - ACK - Pacote recebido não tinha nenhum erro.

Observação: Apenas em casos de "NACK", o pacote não é armazenado. Isto ocorre quando há o "ACK", com ou sem erro.

Ao final, são fechadas as conexões dos canais e são escritos os registros que irão no arquivo com os dados gravados pelo Beta, fechando então as funcionalidades deste processador.

1.6. Geração de Erros (*error_gen.c e error_gen.h*)

O processo de injeção de erros faz uso de uma biblioteca desenvolvida chamada *error_gen.h* que auxilia na geração de erros aleatórios controlados, de modo que possamos gerar cenários diferentes e aleatorizá-los.

```
#define RAND_NONE  0b00000000
#define RAND_FIELD 0b00000001
#define RAND_FLIT  0b00000010
#define RAND_POS   0b00000100

#define SCENARIO_NB 7

typedef enum {
    SH_FIELD,
    PAYLOAD_FIELD,
    ECC_FIELD,
} field_t;

typedef struct {
    uint8_t error_rand;
    field_t field;
    uint8_t flit;
    uint32_t position;
    uint8_t nb_error;
} error_config_t;
```

Listing 5: Error Configuration Struct

Como é mostrado no Código 5, uma estrutura de configuração de erros é criada de modo que possamos indicar em quais parâmetros do pacote os erro será gerado (*field*, *flit*, *position*) ou se esses parâmetros serão gerados aleatoriamente (*error_rand*). Por fim, existe um atributo para indicar o número de erros (que será gerado no mesmo flit selecionado, *nb_error*).

Variáveis do tipo *error_config_t* são usadas como entradas no método *configure_eror* que, utilizando os dados da estrutura, atualizam uma "máscara"que nada mais é um pacote com todos os canpos configurados para zero e os bits que devem ser alterados para gerar erro são atribuídos o valor de 1. Um segundo método chamado *inject_error* recebe a máscara e o pacote recebido e, de fato, modifica-o para a geração de erro.

Ademais, ainda nesta mesma biblioteca foram criados métodos de cenários (*scenario n*, onde n é o número do cenário) que configuram os erros de forma a gerar padrões conhecidos e desejados e retornam uma máscara de geração de erro. Estes cenários são então escolhidos de forma aleatória quando chamamos o método *rand_selec_scenario* motrado no Código 6.

```
uint8_t rand_select_scenario(packet_t * packet_mask){
    uint8_t scne = rand() % SCENARIO_NB;

    switch (scne){
        case 0 : scenario0(packet_mask); break;
        case 1 : scenario1(packet_mask); break;
        case 2 : scenario2(packet_mask); break;
        case 3 : scenario3(packet_mask); break;
        case 4 : scenario4(packet_mask); break;
        case 5 : scenario5(packet_mask); break;
        case 6 : scenario6(packet_mask); break;
        default: scenario0(packet_mask); break;
    }

    return scne;
}
```

Listing 6: Random Scneario Selector

Desse modo, para criação de um cenário novo, basta criar o método e adicioná-lo no case do método *rand_selec_scenario*. A decisão de fazer deste modo fpoi tomada, pois tornar a injeção de erro completamente aleatória para todos os pacotes poderia gerar uma alta taxa de NACKs e a simulação não gerar um resultado interessante para o nosso propósito. Desse moso 7 cenários foram criados de modo a exercitar as características listadas na Tabela 1.2.

Tabela 1.2: Descrição dos Cenários.

Cenário	Descrição
scenario0	Sem Erro
scenario1	Erro Simples aleatoriamente gerado em relação a "flit", "field"e "position".
scenario2	Erro Duplo aleatoriamente gerado em relação a "flit", "field"e "position".
scenario3	Dois Erros Simples em diferentes e aleatoriamente gerados em relação a "flit", "field"e "position".
scenario4	N Erros Simples em diferentes e aleatoriamente gerados em relação a "flit", "field"e "position". Onde N també é aleatório.
scenario5	Erro Simples no campo SH aleatoriamente gerado em relação a "flit", e "position".
scenario6	Erro Simples no campo ECC aleatoriamente gerado em relação a "flit", e "position".

Para mais detalhes sobre as implementações, acessar o código fonte no .zip anexo ao

trabalho

1.7. Injetor de Erro (*3_inject_error.c*)

O processo de injeção de erros, começa inicializando a máscara e configuração de erros e então estabelecendo conexão com os processos alfa e beta. Ao entrar no *loop* principal, ele abre um arquivo de *log* onde os cenários de erros escolhidos para cada são armazenados. Então, ele escolhe um cenário de erro aleatoriamente utilizando o método *rand_select_scenario*, escreve o cenário selecionado no arquivo de *log* e espera pela recepção de um pacote através da conexão com o processo alfa.

Uma vez que o pacote chega, o erro é aplicado a este através do método *inject_error* e o mesmo é enviado para o processo beta. Após isto, a máscara de erros é resetada novamente e, então, o procedimento se repete dentro do *loop* infinito.

2. Resultados

Nesta seção descreveremos os resultados obtidos em relação ao número de erros percebidos pelo processo alfa, aos cenários gerados e ao arquivo final que o processo beta gera, considerando um arquivo de entrada de 100 pacotes.

É importante ressaltar que, apesar de o arquivo ter o total de 100 pacotes, mais pacotes podem ser transitados/transmitidos já que existe situações de retransmissão. Desse modo, vemos que, ao final da simulação, tivemos o total de 132 pacotes transitados, como é possível ver na Tabela 2.1 que também mostra o total de ACK, ACK ERROR e NACK recebidos por Alfa.

Tabela 2.1: Número de ACK, ACK ERROR e NACKS

Estatística	
ACK	40
ACK ERROR	60
NACK	32

Os cenários e número de vezes que eles foram gerados durante esta simulação pode ser visto na Tabela 2.2.

Por fim, o arquivo de saída gerado por beta, para esta simulação, ficou praticamente igual ao arquivo de entrada, exceto por dois pacotes/linhas, como pode ser visto na Figura 2.1.

Isso se deve ao fato de que os ECCs pode "se confundir"quando uma quantidade de erros grande ocorre e o cenários 4 que utilizamos é muito agressivo e pode, aleatoriamente, gerar este tipo de situação. Quando removemos o cenário 4 da lista, este problema não ocorre mais.

Tabela 2.2: Número de execução de cada cenário.

Estatística	
Cenário 0	16
Cenário 1	18
Cenário 2	24
Cenário 3	19
Cenário 4	18
Cenário 5	15
Cenário 6	22

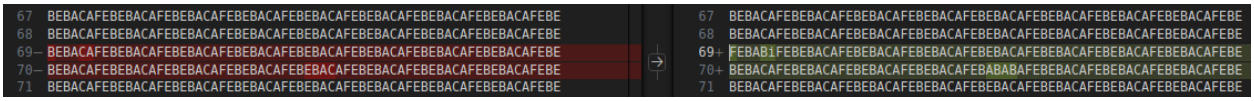

Figura 2.1: Diferença entre os pacotes de entrada e saída.

Uma captura de tela da execução dos 3 processos pode ser vista na Figura 2.2.

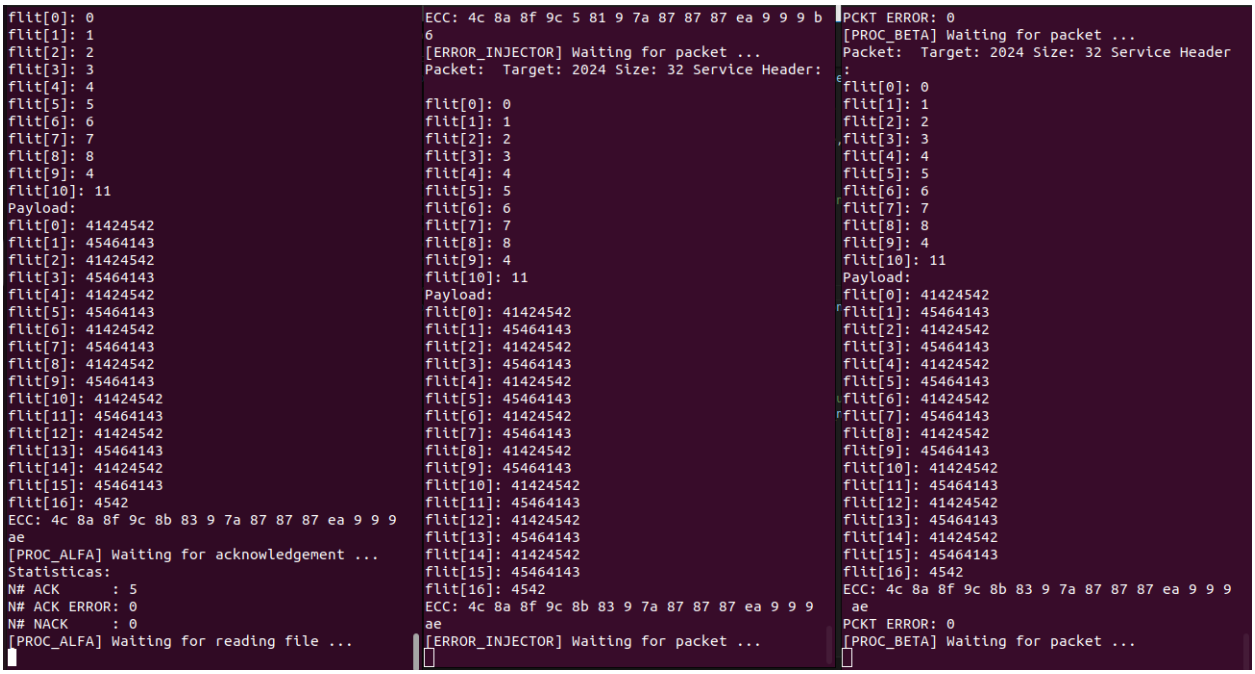

Figura 2.2: Execução dos Processos

Referências

[Wik20] Wikipédia. *Código de Hamming — Wikipédia, a enciclopédia livre*. Acesso em 31 de Maio de 2024. 2020.