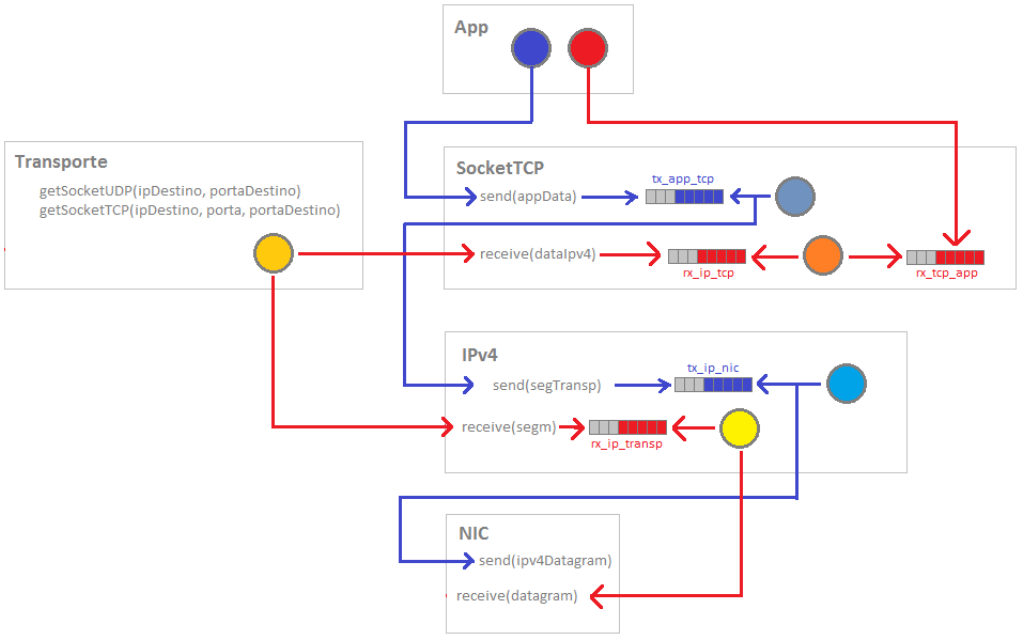
Universidade Federal de Santa Catarina
INE5424 - Sistemas Operacionais 2
Relatório do projeto final

Vicente Silveira Inácio
Fernando Agostinho
Taciane Martimiano

Index

Esse relatório apresenta as ideias e implementações de alguns protocolos utilizados na pilha de protocolos TCP/IP. Primeiramente faremos uma breve introdução da pilha, mostrando a interface de comunicação entre os diferentes protocolos e as threads envolvidas, deixando cada protocolo em específico como uma caixa preta, para depois explicá-los em detalhes.



A maneira que escolhemos para tratar o interfaceamento entre as camadas, e em alguns casos até mesmo na mesma camada, foi alocar buffers nas entradas e saídas das camadas, sendo que duas ou mais threads utilizam esse buffer no modelo produtor/consumidor. Conforme uma mensagem for descendo na pilha de protocolos para a inserção de cabeçalhos e os devidos processamentos de cada camada, a camada superior funciona como a thread produtora, enquanto a camada abaixo dela funciona como consumidora do buffer que os separam. No método receive o processo inverso ocorre. As camadas inferiores são produtoras das suas camadas acima.

Para o funcionamento correto de modo a evitar desperdício de recursos de hardware, foi seguido o modelo produtor/consumidor proposto por Dijkstra, utilizando-se dois semáforos para a realização da sincronização e exclusão mútua.

```
1  Semaforo sem_produtor = 1;
   Semaforo sem_consumidor = 0;
3  Buffer buffer;

5  void produzir( Elemento elemento ){
     sem_produtor.p( );
7    buffer.inserir( elemento );
     sem_consumidor.v( );
9  }

11 Elemento consumir( ){
     sem_consumidor.p( );
```

```
13    Elemento elemento = buffer.consumir( );
      sem_produtor.v( );
15    return elemento;
  }
```

/home/vicente/workspaceCDT/epos_p1/include/exemplo_prod_cons.h

1 P1 - IP (encapsulation, fragmentation and sending)

1.1 Encapsulation

Esta parte do trabalho objetiva a implementação no código epos da camada de acesso à rede da arquitetura TCP/IP. O epos por si só já nos provê as funcionalidades básicas para o envio de dados através da camada física (utilizando a epos-NIC). Por parte da implementação deste grupo, adicionamos as abstrações de IP e interfaceamento da camada de acesso à rede.

A partir do enunciado do trabalho foi possível extrair as informações necessárias e objetivos desta etapa, que seriam referentes a abstração dos endereços IP, encapsulamento IP e fragmentação destes pacotes.

Para isto criamos algumas classes responsáveis por estas funções (anexadas ao final do relatório) tais como "ip.h" e "ip_datagram.h". A primeira tem a função de fazer o interfaceamento e implementação do protocolo IP, lógica de envio e recebimento de dados, fazendo o uso da camada física NIC presente no epos. Já a segunda classe citada fica representa um datagrama IP propriamente dito, contendo cabeçalho e dados. Utilizamos o conceito de STRUCT de C, para fazer a estruturação adequada dos atributos desta classe. Para os endereços IP apenas tratamos ele como um unsigned char* (4 posições) e lidamos desta forma ao longo do projeto. O resultado é o que segue:

```
public:
2
  char datagram[1500];
4
  static const unsigned char VER = 4;
6 static const unsigned char IHL = 4;
  static const unsigned char TOS = 0;
8 static const unsigned char TTL = 64;

10 /* ********** DATAGRAM ********** */

12 struct DATAGRAM
  {
14   struct HEADER
     {
16     unsigned char _version :4;
       unsigned char _ihl :4;
18     unsigned char _tos:8;
       unsigned short _total_length:16;
20     unsigned short _identification:16;
       unsigned short _flags :3;
22     unsigned short _frag_offset :13;
       unsigned char _ttl:8;
24     unsigned char _protocol:8;
       unsigned short _header_checksum:16;
```

```
26          IP_Address _src_address;
            IP_Address _dst_address;
28        } *_header;
          char* _data;
30      } _datagram;

32  }__attribute__((packed, may_alias));
```

/home/vicente/workspaceCDT/epos_p1/include/exemplo_struct.h

Como pode ser observado pelo código acima, existe uma STRUCT que contém um char* representando os dados deste datagrama, e uma outra STRUCT representando o cabeçalho, com os atributos e seus campos seguindo a RFC.

Vale ressaltar também que estas estruturas possuem os atributos "packed" e "may_alias". O primeiro especifica que cada membro da estrutura é organizado com o objetivo de minimizar a memória requerida. Já o atributo "may_alias", define que o acesso através de ponteiros a tipos não estão sujeitos à análises de alias baseados em tipos, e ao invés disso é assumido que possam dar alias em qualquer outro tipo de objetos.

Nesta classe que representa a abstração do datagrama é necessário observar o endianess do epos afim de que a organização da informação não seja embaralhada, e que como visto em aula, o epos de 16 bits é little-endian e inverte os bytes de posição. Como solução para este caso nas atribuições e busca dos atributos foi necessário a utilização dos métodos já fornecido pelos epos chamados de CPU::htons() e CPU::ntons(), que tem esta semântica do endianess.

A criação dos datagramas IP se dá de forma simplificada. Quando da fragmentação e envio estes atributos vão sendo definidos através de métodos setters definidos na classe. Um exemplo da criação destes datagramas é o que segue:

```
/* Construtor do Datagrama */
2 IP_Datagram(const IP_Address & source, const IP_Address & dest,
      const Protocol & protocol, unsigned int size)
4 {
    DATAGRAM _datagram;
6
    _datagram._header->_version = VER;
8   _datagram._header->_ihl = IHL;
    _datagram._header->_tos = TOS;
10  _datagram._header->_total_length = CPU::htons(size);
    _datagram._header->_identification = CPU::htons(0);
12  _datagram._header->_flags = 0;
    _datagram._header->_frag_offset = 0;
14  _datagram._header->_ttl = TTL;
    _datagram._header->_protocol = protocol;
16  _datagram._header->_header_checksum = 0;
    _datagram._header->_src_address = source;
18  _datagram._header->_dst_address = dest;
  }
```

/home/vicente/workspaceCDT/epos_p1/include/exemplo_createDg.h

1.2 Fragmentation and Sending

A fragmentação de datagramas ocorre quando o tamanho dos dados passados supera a capacidade de transmissão de rede, no caso do epos o tamanho dos

dados passados através do método send ser maior que a nic.mtu() do epos. Para que seja enviado um frame ethernet deve ser menor que a MTU da rede (nic.mtu()). O frame se divide em um cabeçalho de 20 bytes e o tamanho dos dados equivalem a MTU menos o tamanho do cabeçalho. Existem portanto dois casos:

- **Não-Fragmentação**: Caso o tamanho dos dados não ultrapasse a capacidade de envio da rede, é apenas necessário o envio de um datagrama. Este datagrama será criado com um identificador único, com uma flag setada em zero significando que é um pacote único, com seus dados e sem ajuste de offset de fragmentação (offset = zero) e cálculo de checksum.

- **Fragmentação**: Assim que é detectado que o tamanho dos dados supera a capacidade de envio, é feito um cálculo com o objetivo de identificar a quantidade de fragmentos que serão enviados. Para cada fragmento a ser enviado as seguintes alterações são necessárias: O identificador do pacote deve corresponder ao mesmo em todos os fragmentos de um determinado pacote. O offset de fragmentação deverá ser incrementado para cada fragmento deste conjunto, sendo que o offset de fragmentação corresponde a posição deste em relação aos outros fragmentos do conjunto. Enquanto não for o último fragmento, estes deverão conter a flag MF (MORE FRAGMENTS = 1) e se este fragmento for o último este deverá conter a flag MF (MORE FRAGMENTS = 0), que resulta no fim da fragmentação. Por fim, os dados deverão ser definidos para cada fragmento individualmente e realizado o cálculo do checksum. Exemplo:

```
// FRAGMENTACAO E ENVIO
if (size > (size - HEADER_SIZE)) {
  // CALCULO DO NUMERO DE FRAGMENTOS A SER ENVIADOS
  int num_frag = (int) (size / MTU_SIZE);
  if (size % (size - HEADER_SIZE) > 0)
    num_frag += 1;

  // envio de fragmentos
  for (int i = 0; i < num_frag - 1; i++) {
    /* PREPARA DATAGRAMA */
    // CRIACAO DO DATAGRAMA
    IP_Datagram * header = new IP_Datagram(_my_IP, dst,
    PROTOCOL_IP,
        size);
    header->prepare_datagram(nic.mtu(), MF_FLAG, _packetID,
    _offset,
        data);
    _datagram_ip = header->get_datagram();

    /* ENVIA PACOTE (HEADER + DATA) */
    nic.send(dst_mac, PROTOCOL_IP, _datagram_ip, size);

    /* INCREMENTA OFFSET PARA PROXIMO FRAGMENTO
     * LIBERA MEMORIA DESTE FRAGMENTO*/
    _offset += 1500;
    free(header);
  }

  // envio do ultimo fragmento
  // CRIACAO DO DATAGRAMA
```

```
29      IP_Datagram * header = new IP_Datagram(_my_IP, dst,
          PROTOCOL_IP,
31          size);
        header->prepare_datagram(nic.mtu(), LF_FLAG, _packetID,
          _offset,
33          data);
        _datagram_ip = header->get_datagram();
35
        /* ENVIA PACOTE (HEADER + DATA) */
        nic.send(dst_mac, PROTOCOL_IP, _datagram_ip, size);
37
        /* LIBERA MEMORIA DESTE FRAGMENTO*/
39      free(header);

41    } else {
        /* PREPARA DATAGRAMA */
43      // CRIACAO DO DATAGRAMA
        IP_Datagram * header = new IP_Datagram(_my_IP, dst,
          PROTOCOL_IP,
45          size);
        header->prepare_datagram(nic.mtu(), LF_FLAG, _packetID,
          _offset,
47          data);
        _datagram_ip = header->get_datagram();
49
        /* ENVIA PACOTE (HEADER + DATA) */
51      nic.send(dst_mac, PROTOCOL_IP, _datagram_ip, size);

53      /* LIBERA MEMORIA DESTE FRAGMENTO*/
        free(header);
55    }
```

/home/vicente/workspaceCDT/epos_p1/include/exemplo_frag.h

Em ambos os casos esta interface de acesso à camada de rede faz a utilização de uma chamada da NIC do epos, através do comando nic.send() passando um char* correspondendo ao frame.

2 P2 - ARP and IP routing

O exercício proposto para esta fase do projeto destina-se a implementar o protocolo ARP, que é o protocolo de resolução de endereços, e a implantação de um sistema de roteamento de endereços IP.

Em nossa solução adicionamos comentátios ao máximo no código para que este ficasse de forma auto-explicativa. Nesta seção apenas contextualizamos O QUE foi feito (classe e métodos criados, etc) e na seção seguinte, COMO foi feito.

Nossa solução consiste na criação de duas classes, ARP (arp.h) e ROUTER (ip_router.h), e uma subclasse "arp_datagram.h" Ambas as classes possuem implementação semelhante, visto que lidam com mapas/coleções, de resolução de endereços e roteamento destes endereços. A classe "arp_datagram.h" representa um datagrama ARP propriamente dito, com os atributos e implementação seguindo a RFC.

2.1 ARP

O protocolo ARP consiste em um mapeamento de um endereço IP para um endereço físico MAC, e para isto utiliza uma tabela chamada tabela ARP. Assim como na implementação do datagrama ip, a classe que representa o datagrama ARP possui uma estrutura (STRUCT) do C, que define um elemento que pertencerá a tabela ARP, e possui também os atributos "packed" e "may_alias", comentados na primeira seção.

O protocolo define que se um endereço MAC é buscado fazendo utilização de um endereço IP, este deve ser buscado na tabela ARP. Caso este mapeamento exista na tabela ARP, apenas o endereço MAC é retornado. Caso contrário é feito um broadcast (ARP REQUEST) do datagrama arp, definindo o IP desejado como um de seus atributos (src_protocol_addr) e o campo referente ao MAC (dst_hardware_addr) buscado eh zerado afim de ser encontrado. Dado este broadcast a procura de um IP, o host que possuir este endereço responde com seu respectivo MAC através de um ARP REPLY, setando no pacote o campo (dst_hardware_addr) e realiza resposta direta (Unicast) para o host que requisitou. Por fim este mapeamento é adicionado na tabela a fim de acelerar possíveis buscas por este endereço.

O elemento da tabela ARP possui como atributos um MAC (NIC_Common::Address<6>do epos) e um IP (unsigned char* de nossa implementação).

```
typedef unsigned char* IP_Address;

typedef struct Table_Element {
  MAC_addr _mac;
  IP_Address _ip;
} element;

private:
  int _table_index;
  element _arp_table[LENGTH];
};
```

/home/vicente/workspaceCDT/epos_p1/include/exemplo_arp.h

A classe ARP implementa os seguintes métodos:

- Insert: Insere um novo elemento na tabela, o elemento como citado anteriormente, possui MAC e IP que definem uma entrada e se relacionam diretamente. O método realiza uma busca em sua tabela utilizando o método **search** da classe. parâmetros passados
- Get: Este método é o responsável por retornar o endereço MAC referente ao IP passado, consultando na tabela ARP.
- Search: Método utilizado internamente na classe para buscar o elemento na tabela ARP, de acordo com o parâmetro passado (endereço IP) e retorna o MAC relativo a este endereço.

```
// ARP REQUEST
// pacote protocolo ARP 0x0806
void arp_request(IP_Datagram* frame) {
  if (frame->_datagram._header->_dst_address == _my_IP) {
    ARP_Datagram* arp_req = reinterpret_cast<ARP_Datagram>(frame);
```

```
6
      MAC_Address mac_src = arp_req->_arp_packet._src_hw_address;
8     arp_req->_arp_packet._dst_hw_address = _my_mac;

10    // MANDA UNICAST PARA O REMETENTE COM SELF MAC INSERIDO NO
      PACOTE
      nic.send(mac_src, 0x0807, arp_req->frame, 28);
12  }
}
14
// ARP REPLY
16 // pacote protocolo ARP 0x0807
void arp_reply(IP_Datagram* frame) {
18  ARP_Datagram* arp_req = reinterpret_cast<ARP_Datagram>(frame);

20  // INSERE O REGISTRO NA TABELA ARP COM O MAC RECEBIDO NO REPLY
  _arp_table.insert_element(arp_req->_arp_packet._dst_hw_address,
22      arp_req->_arp_packet._dst_ip_address);

24 }
```

/home/vicente/workspaceCDT/epos_p1/include/exemplo_arpReqRep.h

2.2 Routing

O protocolo de roteamento define a identificação de uma rota para um determinado IP destino desejado. Como os hosts ligados a rede não se conhecem pelo endereço físico MAC e não tem uma conexão direta entre si, é necessário então calcular uma rota na qual seja possível estabelecer essa conexão com hosts intermediários estando, ou não, em uma mesma rede. Utilizando-se da camada de acesso à rede, implementada nas etapas anteriores deste projeto final, é possível enviar datagramas entre hosts utilizando-se da abstração do IP.

Quando estes hosts comunicantes não estão conectados diretamente entre si, entra em ação o papel de um roteador, que é uma classe (ip_router.h) implementada pelo grupo, que fica responsável por resolver a rota na qual o host 'A' deverá encaminhar seus dados para o host 'B', e é através desta rota (gateway) na qual será feito um tunelamento para a comunicação entre estes hosts.

```
// PROCURA NA TABELA DE ROTAS PELO ENDERECO PASSADO CASO
2 // EXISTA UMA ROTA PARA ESTE DESTINO RETORNA GATEWAY.
IP_Address resolve(IP_Address dest_ip, IP_Address src_ip,
4     IP_Address src_mask, IP_Address src_gtway) {

6   /* Aproveita e insere na tabela de roteamento uma entrada com as
     * informacoes do requisitante (source)*/
8   insert_route(src_ip, src_mask, src_gtway);

10  // Aplica mascara para verificar se estao na mesma rede
    IP_Address _my_netw = apply_mask(src_ip, src_mask);
12  IP_Address _dst_netw = apply_mask(dest_ip, src_mask);

14  /* Se estao na mesma rede, adiciona uma entrada na tabela com o
      gateway
     * do source para o dest e retorna diretatemente o gateway do
      source
16   * */
    if (_my_netw == _dst_netw) {
```

```
18      insert_route(dest_ip, src_mask, src_gtway);
        return src_gtway;
20    }

22    int index = 0;
      for (int i = 0; i < LENGTH; i++) {
24      if ((_routing_table[i]->_destination[0] == dest_ip[0])
            && (_routing_table[i]->_destination[1] == dest_ip[1])
26          && (_routing_table[i]->_destination[2] == dest_ip[2])
            && (_routing_table[i]->_destination[3] == dest_ip[3])) {
28        index = i;
        }
30    }

32    return _routing_table[index]->_gateway;
  }
```

/home/vicente/workspaceCDT/epos_p1/include/exemplo_routerResolve.h

A classe "ip_router.h" possui uma tabela de roteamento que contem registro de rotas. Estes registros, assim como na tabela ARP, estão na forma de uma STRUCT de C, e possuem como atributos quatro elementos: IP destination, que representa o endereço IP do host de destino buscado, IP netmask, representando a máscara de subrede na qual o host se encontra, IP gateway, correspondendo ao caminho, ou endereço na qual o sender deverá submeter seus dados afim de alcançar seu destino final e o último elemtento é uma STRUC que representa um outra entrada com estes atributos (de forma recursiva) para representar redes externas e next hop.

```
  typedef unsigned char* IP_Address;
2
  typedef struct Route_Element {
4   IP_Address _destination;
    IP_Address _netmask;
6   IP_Address _gateway;
    Route_Element next_hop;
8 } element;

10 private:
    int _table_index;
12  element* _routing_table[LENGTH];
  };
```

/home/vicente/workspaceCDT/epos_p1/include/exemplo_router.h

Esta classe possui os seguintes métodos:

- Insert: Insere uma rota na tabela de rotas formada a partir dos parâmetros passados na função destination, netmask, gateway.
- Resolve: Este método recebe como parâmetros do sender: ip, máscara, gateway e mac, e do receiver apenas o ip de destino. Este método realiza um cálculo da máscara com o ip do destino (um AND lógico) afim de verificar se os hosts estão na mesma rede e realiza uma busca interna em sua tabela pelo ip ser retornado (gateway).

```
1 /* APLICA A MASCARA SOBRE O IP DE DESTINO PASSADO
```

```
 * PARA VERIFICAR SE PERTENCEM A MESMA REDE
 * RETORNA A REDE EM QUESTAO */
IP_Address apply_mask(IP_Address dest, IP_Address mask) {
  IP_Address subnet;
  int i = 0;
  while (i < 4) {
    subnet[i] = dest[i] & mask[i];
    i += 1;
  }

  return subnet;
}
```

/home/vicente/workspaceCDT/epos_p1/include/exemplo_maskrouter.h

3 P3 - IP (receiving and buffer management)

3.1 Receiving

Esta etapa corresponde à entrada de dados a partir da camada de enlace, implementada pela NIC (epos), na camada de rede. O método escolhido ainda não está definido. A princípio utilizaremos o padrão de projeto Observado/Observador, de modo que quando a NIC (observada) estiver com o datagramaIP pronto para ser entregue para a camada superior, a classe Ipv4 (observadora) será notificada.

3.2 Considerações sobre o IPv4

O protocolo IP fragmenta os datagramas na hora do envio quando o datagrama ultrapassa o tamanho da MTU da rede. Dessa forma, após a fragmentação, diferentes frames do datagrama podem seguir por caminhos diferentes (rotas) pela rede até atingir o host destino, já que a remontagem dos frames em um datagrama único é feita apenas no host destino. Dessa forma os diferentes frames podem chegar no destino fora de ordem ou mesmo se perderem no meio do caminho. Um envio de um datagrama IP não é confiavel. Caso aconteça de algum frame se perder no meio do caminho, o remetente não será notificado dessa perda. Dado um datagrama que seja fragmentado em dois ou mais frames, caso algum desses frames não chegue completamente em um roteador final ou intermediário, o datagrama inteiro é descartado.

3.3 Classe IPv4 (ip.h)

Para o processo de remontagem, a classe IPv4 conta com duas estruturas de dados. A primeira delas é o **bufferDatagramasReceive**, um Hash Map que mapeia o identificador de um datagrama a um objeto da classe **IpDatagramaReceive**. Esse objeto representa o conjunto de todos os frames que chegam na camada de rede e que possuam um identificador em comum.

A outra estrutura é o **conjuntoDatagramasPerdidos**, que é apenas um Hash Set contendo todos os identificadores de datagramas que levaram mais tempo que o permitido para enviar todos os seus fragmentos. Logo se algum frame chegar de outro host na camada de rede, tal que seu identificador esteja contido nesse conjunto, o frame será descartado.

Caso o frame não seja descartado, um objeto **IpDatagramaReceive** é criado (ou recuperado através do **bufferDatagramasReceive**) para que possamos inserir um outro frame que possua o mesmo identificador dos outros. O objeto **IpDatagramaReceive** recebe frames de um mesmo datagrama, para que no momento do recebimento completo de todos os frames do datagrama, os dados de cada frame recebido possam ser reestruturados em um único char*. Esse char* é o que será passado para a camada de transporte.

```
1  void ipv4_receive(IP_Datagram* frame )
   {
3    if(estaContidoNoConjuntoPerdidos(frame->_datagram._header->
       _identification))
       return;
5
     if(!estaContidoNoBufferDatagramReceive(frame->_datagram._header->
       _identification)){
7      IP_Datagram_Receiver aux = new IP_Datagram_Receiver();
       Simple_List<IP_Datagram_Receiver>::Element * e = new
       Simple_List<IP_Datagram_Receiver>::Element(aux);
9      bufferDatagramasReceive.insert_tail(e);
     }
11
     IP_Datagram_Receiver datagrama_receive = getIpDatagramReceive(
       frame->_datagram._header->_identification);
13   IP_Datagram_Receiver::StatusFrameReceiver status =
       datagrama_receive.addFrame(frame);

15   if(status == IP_Datagram_Receiver::TEMPO_ESGOTADO){
       Simple_List<unsigned short>::Element * e = new Simple_List<
       unsigned short>::Element(frame->_datagram._header->
       _identification);
17     conjuntoDatagramasPerdidos.insert_tail(e);
     }else if(status == IP_Datagram_Receiver::COMPLETO){
19     char* datagramaCompleto = datagrama_receive.
       getDatagramaDataCompleto();
       _rx_ip_transp->produz(datagramaCompleto);
21   }
   }
```

/home/vicente/workspaceCDT/epos_p1/include/exemplo.ipv4rec.h

3.4 IpDatagramaReceive

A cada frame recebido do datagrama, é executado o método **addFrame(IpDatagrama frame)** no **IpDatagramaReceive** específico. Primeiramente é verificado se o frame demorou mais do que foi especificado. Caso o frame passe nesse teste, ele é colocado na lista de frames do datagrama e o valor do tamanho de seus dados é diminuido de uma **somaTotal**, que na criação do objeto é iniciada em zero.

Quando o último frame do datagrama chegar, não necessariamente em último na escala de tempo, saberemos o tamanho total dos dados do datagrama não fragmentado. Esse tamanho é somado da **somaTotal**. Quando esse último frame chega, para cada novo frame que chegar, incluindo esse último, é realizado um teste para verificar se a **somaTotal** é igual a zero. Caso for igual a zero, significa que todos os frames desse datagrama chegaram, e eles podem ser remontados novamente.

O processo de remontagem primeiramente cria um char*, que possui o tamanho da quantidade de dados do datagrama completo. Em seguida, os dados de cada frame são passados para o char*, na posição correta, conforme o offset de cada frame.

```
StatusFrameReceiver addFrame(IP_Datagram * frame)
{
  _tempoFinal = _chronometer.read();

  if (_tempoFinal - _tempoInicio > WAIT_TIME)
    return TEMPO_ESGOTADO;
  else
    _tempoInicio = WAIT_TIME;

  _somaTotal = _somaTotal - frame->_datagram._header->_total_length
     + 20; // tira 20 do cabecalho

  Simple_List<IP_Datagram>::Element* element =
      new Simple_List<IP_Datagram>::Element(frame);
  _frames.insert(element);

  // LAST FRAGMENT
  if (frame->_datagram._header->_flags == false)
    chegouUltimoFrame(frame);

  if (_chegouUltimoFrame) {
    if (_somaTotal == 0) {
      _datagramaCompleto = true;
      return COMPLETO;
    }
  }
  return INCOMPLETO;
}
```

/home/vicente/workspaceCDT/epos_p1/include/exemplo_ipdatagramrec.h

4 UDP - TCP

4.1 Camada de Transporte

A camada de transporte é formada pelos protocolos UDP e TCP. O protocolo UDP não dá garantias de entrega dos seus segmentos em ordem, ou mesmo não corrompidos. Além de não possuir conexão e seu controle de fluxo ficar a cargo da camada de aplicação que o utiliza. Já o protocolo TCP garante a uma aplicação cliente a entrega da mensagem de forma não corrompida e em ordem. No TCP, os dados só são entregues ao seu destino após a realização de uma conexão entre os hosts.

A camada de transporte cria a abstração de uma porta de comunicação. Quando um socket TCP ou UDP é criado, ele faz a ponte de comunicação entre dois processos distintos em máquinas distintas. As máquinas são mapeadas para IP, enquanto os processos são mapeados para uma destas portas. Dessa forma a comunicação de dois hosts separados pela rede é mapeada para dois IPs (remetente e destino) e suas portas de comunicação usadas.

A classe **transporte** tem duas funções. A primeira é a criação de sockets para serem utilizados pelas aplicações. Conforme a escolha da aplicação, pode

ser criado um **socketTCP** ou um **socketUDP**. Para a sua criação é necessário o IP destino, porta que se quer utilizar, e a porta com que se quer comunicar do destino. Uma porta só pode estar associada a um socket. O valor da porta remetente é o valor da chave do scoket.

```
Socket getSocket(IP_Address ipDestino, int portaRemetente, int
    portaDestino, TipoSocket tipo)
{
  Socket* socket;
  if(!portaEstaSendoUsada(portaRemetente)){

    if(tipo == TipoSocket::TCP){
      socket = new Socket_TCP(ipDestino,portaRemetente,
    portaDestino,_ipV4);
    }else if(tipo == TipoSocket::UDP){
      socket = new UDP_Socket(ipDestino,portaRemetente,
    portaDestino,_ipV4);
    }

    Simple_List<Socket>::Element * e = new Simple_List<Socket>::
    Element(socket);
    sockets_ativos.insert_tail(e);

    return socket;
  }
}
```

/home/vicente/workspaceCDT/epos_p1/include/exemplo_getsocket.h

Quando uma aplicação, já em posse de seu socket, quer enviar um dado para seu destino, ela utiliza o método **send( )** do socket. O socket utilizado tem uma referência direta para o objeto Ipv4, responsável pela camada de rede. Porém, no destino, quando a camada de rede termina de processar um datagrama completo que chega do host remetente, ela o coloca no buffer **rx_ip_transp** (produtor). A segunda função da classe **Transporte** é mapear o segmento recebido do ipv4 para o socket correto. Existe uma thread especializada para isso. Ela consome os dados do buffer **rx_ip_transp**, avalia qual é o socket correto para receber o segmento e o coloca no buffer de entrada dele.

```
void receiver(){
  while(true){
    char* ipData = ipv4.receive();
    int porta = getPorta(ipData);
    Socket socket = getSocketAtivo(porta);
    socket.receiveIpData(ipData);
  }
}
```

/home/vicente/workspaceCDT/epos_p1/include/exemplo_transp_receiver.h

4.2 UDP

Para implementação do protocolo de transporte UDP foram criadas duas classes: udp_datagrama e udp_socket. A classe udp_datagrama é constituída da estrutura do datagrama em si, contendo a parte dos dados e a parte do cabeçalho

(tamanho do pacote, porta fonte, porta destino e checksum). Essa classe também conta com informações do chamado pseudo cabeçalho, as quais são utilizadas apenas para o cálculo mais preciso do checksum (não são enviadas no datagrama), como endereço IP fonte, endereço IP destino, protocolo, etc.

Além disso, possui métodos para alocação e desalocação de memória para o datagrama, recebimento dos dados que serão enviados no datagrama, cálculo do checksum e montagem do datagrama completo.

```
public: //datagrama com cabe alho e dados
  char* datagram;

  struct DATAGRAM
  {
    struct HEADER
    {
      unsigned short _source_port :16;
      unsigned short _destination_port :16;
      unsigned short _checksum:16;
      unsigned short _length :16; //Comprimento do datagrama (
    Cabe alho UDP + Dados)
    } *_header;
    char* _data;
  } _datagram;
}__attribute__((packed, may_alias));
```

/home/vicente/workspaceCDT/epos_p1/include/exemplo_UDP.h

Já a classe udp_socket serve como interface entre a camada de aplicação e a camada de rede, sendo contactada pela aplicação para o envio dos dados e contactando a aplicação para passar dados recebidos. Logo, nessa classe existem dois métodos: send e receive. O send cria o datagrama (através da classe udp_datagrama) e chama a camada de rede para encaminhar o envio dos dados.

No método receive, são recebidos os dados da classe transporte, checa-se o checksum e transmitem-se apenas os dados para a aplicação (retira-se o cabeçalho).

A classe udp_socket abstrai a ligação entre o socket e a porta de dados propriamente dita.

```
//CRIAR DATAGRAMA UDP e enviar atrav s da camada IP
void send(char* data)
{
  _udp_datagrama.prepare_datagram(_size, data);
  _ipV4->send(_ip_dest, _udp_datagrama.get_datagram(), _size);
  _udp_datagrama.free_datagram();
}

//Checar checksum e passar apenas os dados para aplica   o
void receive_ipdata(char* data) {

  UDP_Datagram datagram = getUDPdatagram(data);

  // Se os checksums forem iguais, o pacote foi transmitido
    corretamente
  // e pode ser entregue    aplica   o
  if (checksum(datagram.get_length(), datagram.get_src_address(),
      datagram.get_dest_address(), datagram.get_padding(),
      datagram.get_buffer()) == datagram.get_checksum())
```

```
19    {
        // Buffer recebe os dados
21      _buffer->produz(datagram.get_data());
      }
23  }

25  // Aplicacao acessa os dados (sem header)
    char* receive(){
27    return _buffer->consome();
    }
```

/home/vicente/workspaceCDT/epos_p1/include/exemplo_UDPsendrec.h

4.3 SocketTCP

Um **socketTCP** está associado a um IP destino e a um par de portas (remetente e destino). Quando uma aplicação quer enviar um dado ao processo destino, ela utiliza o método **send(mensagem)**. Esse método fragmenta a mensagem em diversos segmentos, de acordo com o tamanho máximo definido para cada segmento, e os colocam em ordem no buffer **tx_app_tcp**.

```
void send(char* mensagem){
2   char[][] fragmentosDaMensagem = getFragmentosMensagem(mensagem);
    for(char* frag: fragmentosDaMensagem)
4     tx_app_tcp.produz(frag);
}
```

/home/vicente/workspaceCDT/epos_p1/include/exemplo_socketTCPsend.h

Quando uma aplicação quer receber um dado, ela utiliza o método **receive( )**. O método **receive( )**, como todo consumidor dos buffers de interfaceamento entre camadas, dorme caso o buffer não tenha nenhum elemento no momento, vindo a acordar quando algum elemento chegar.

```
1   char* receive(){
      return rx_tcp_app.consome();
3   }
```

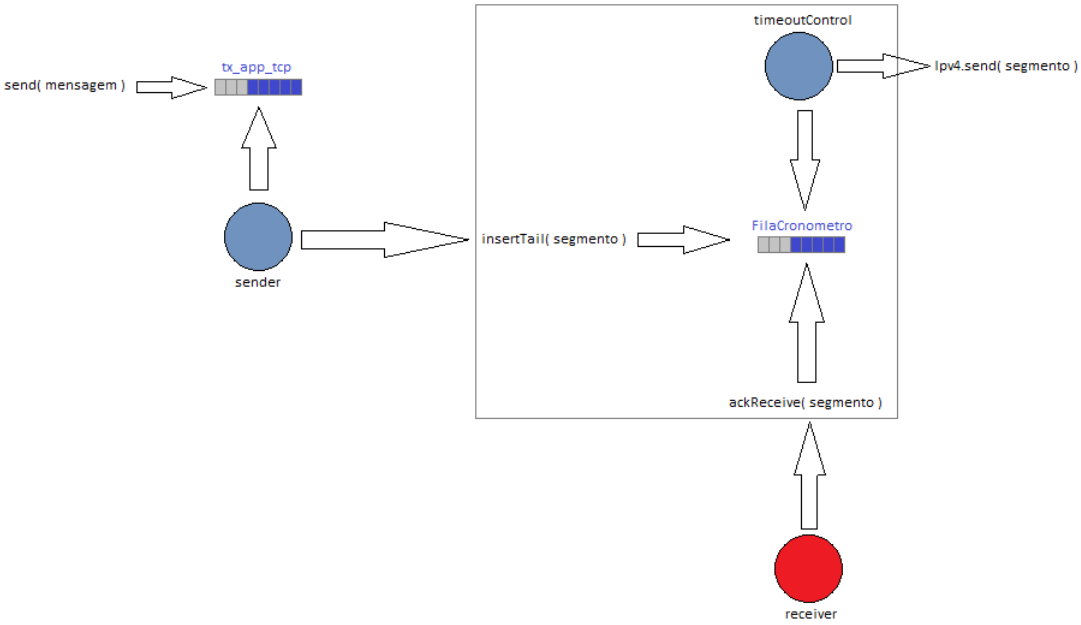
/home/vicente/workspaceCDT/epos_p1/include/exemplo_socketTCPreceive.h

Três threads atuam em cada **socketTCP**. A primeira delas executa o método **sender( )** durante o tempo todo. Esse método é responsável por pegar um segmento do buffer **tx_app_tcp**, preparar o cabeçalho apropriado para o envio e enviá-los para o objeto **FilaSending**.

```
1   void sender(){
      while(true){
3
        if(status == StatusTCP.NAO_CONECTADO){
5         SegmentoTCP syn = SegmentoTCP.getSegmentoSYN();
          status = StatusTCP.CONECTANDO;
7         filaSending.insertTail(syn);
          esperarConexao.p();
9         esperarConexao.v();
        }else if(status == StatusTCP.CONECTADO){
```

```
11          SegmentoTCP segSending = new SegmentoTCP(tx_app_tcp.getHeader
        ());
            tx_app_tcp.consome();
13          segSending.setChecksum();
            filaSending.insertTail(segSending);
15      }else{
            esperarConexao.p();
17          esperarConexao.v();
        }
19    }
  }
```

/home/vicente/workspaceCDT/epos_p1/include/exemplo_socketTCPfilasending.h



O método **insertTail(segmento)** da **FilaSending** contém um mutex chamado **mutexInserirNaFila**. A thread que pega o mutex, libera-o apenas se a janela do destinatário tiver tamanho suficiente para receber novos segmentos. Outra possibilidade de se liberar esse mutex é através do método **ackReceive**.

```
void insertTail(SegmentoTCP segmento){
2   mutexInserirNaFila.p();
    mutexFilaSending.p();
4
    segmento.valorCronometro = getValorCronometroAtual() +
      tempoMaximoDeEspera;
6
    Elemento e = new Elemento(segmento);
8   filaOrdenadaPorCronometro.insertTail(e);
    valorJanelaDeRecepcaoDoDestino--;
10
    if(valorJanelaDeRecepcaoDoDestino > 0)
12    mutexInserirNaFila.v();

14  mutexFilaSending.v();
```

```
}
```

/home/vicente/workspaceCDT/epos_p1/include/exemplo_socketTCPinsertTail.h

A segunda thread executa o método **timeoutControl( )** dentro do objeto **FilaSending**. Esse método é responsável por reenviar os segmento de dados tcp caso um ack referente a esse segmento não tenha chego em um tempo máximo pré-estabelecido. A **FilaSending** contém uma SimpleList que mantém os segmentos que estão sendo enviados em órdem de envio para o ipv4. Assim que é enviado, é associado ao segmento um valor de cronômetro atual somado ao tempo máximo de envio, e esse segmento é jogado para o final da lista. O método **timeoutControl( )** pega os segmentos do início da lista, verifica se o valor do cronômetro deles é menor que o cronômetro atual. Caso seja, ele reenvia esse segmento associando outro valor de cronômetro pra ele e é jogado novamente para o final da lista. Caso não for, a thread dorme durante o valor do cronômetro do segmento subtraído do valor do cronômetro atual. Conforme os acks que chegarem, é requisitado o método **ackReceive( )** da **FilaSending**. Esse método faz uma busca na lista e retira o segmento de lá.

```
1  void timeoutControl(){
2    while(true){
3      mutexTimeoutControl.p();
4      mutexFilaSending.p();
5
6      long tempoEspera = -1;
7      long timeoutAtual = getValorCronometroAtual();
8
9      SegmentoTCP seg = (SegmentoTCP)filaOrdenadaPorCronometro.
       getHeader().getObject();
       tempoEspera = (int) (timeoutAtual - seg.valorCronometro);
11
12     while(timeoutAtual > seg.valorCronometro){
13       filaOrdenadaPorCronometro.removeHeader();
14       seg.valorCronometro = timeoutAtual + tempoMaximoDeEspera;
15       filaOrdenadaPorCronometro.insertTail(new Elemento(seg));
16       ipv4.send(seg.getSegmentoTCPemFormatoArrayChar());
17       seg = (SegmentoTCP)filaOrdenadaPorCronometro.getHeader().
       getObject();
18       tempoEspera = timeoutAtual - seg.valorCronometro;
19     }
20
21     mutexFilaSending.v();
22     mutexTimeoutControl.v();
23
24     if(tempoEspera > 0)
25       sleep(tempoEspera);
26   }
27 }
```

/home/vicente/workspaceCDT/epos_p1/include/exemplo_socketTCPtimeoutctrl.h

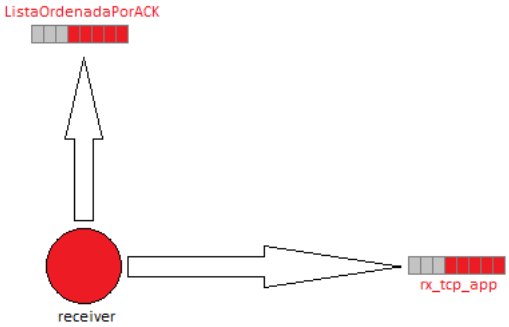
A terceira thread executa o método **receiver( )**, que é responsável por pegar um segmento do buffer **rx_ip_tcp**, avaliar que tipo de segmento TCP se trata (dados, ack, syn, synack, fin) e, dependendo do tipo, ela pode enviar um segmento resposta de volta para o Ipv4, completar uma conexão, finalizá-la ou validar um segmento previamente enviado.

```
void receiver(){
  while(true){
    SegmentoTCP segmento = new SegmentoTCP(rx_ip_tcp.consome());

    if(segmento.isSYN() && status == StatusTCP.NAO_CONECTADO){
      SegmentoTCP seg = SegmentoTCP.getSegmentoSYNACK();
      status = StatusTCP.CONECTANDO;
      numSequenciaEsperado = 0;
      ipv4.send(seg.getSegmentoTCPemFormatoArrayChar());
    }else if(segmento.isSYNACK() && status == StatusTCP.CONECTANDO)
    {
      SegmentoTCP synack = SegmentoTCP.getSegmentoACK();
      status = StatusTCP.CONECTADO;
      rx_tcp_app = new Buffer(5000);
      numSequenciaEsperado = 0;
      ipv4.send(synack.getSegmentoTCPemFormatoArrayChar());
      esperarConexao.v();
    }else if(segmento.isACK() && status == StatusTCP.CONECTANDO){
      status = StatusTCP.CONECTADO;
      rx_tcp_app = new Buffer(5000);
      esperarConexao.v();
    }else if(segmento.isData() && status == StatusTCP.CONECTADO){
      validarReceive(segmento);
    }else if(segmento.isACK() && status == StatusTCP.CONECTADO){
      filaSending.ackReceive(segmento);
      if(tx_app_tcp.isEmpty() && filaSending.isEmpty()){
        status = StatusTCP.FINALIZANDO;
        SegmentoTCP seg = SegmentoTCP.getSegmentoFIN();
        filaSending.insertTail(seg);
        ipv4.send(seg.getSegmentoTCPemFormatoArrayChar());
      }
    }else if(segmento.isFIN() && status == StatusTCP.CONECTADO){
      status = StatusTCP.FINALIZANDO;
      SegmentoTCP ack = SegmentoTCP.getSegmentoACK();
      ipv4.send(ack.getSegmentoTCPemFormatoArrayChar());
      SegmentoTCP fin = SegmentoTCP.getSegmentoFIN();
      ipv4.send(fin.getSegmentoTCPemFormatoArrayChar());
    }else if(segmento.isACK() && status == StatusTCP.FINALIZANDO){
      status = StatusTCP.NAO_CONECTADO;
      esperarConexao.p();
    }
  }
}
```

/home/vicente/workspaceCDT/epos_p1/include/exemplo_socketTCPterceirathread.h

Caso um segmento que chegue ao destino seja um segmento com o número de sequência maior que o valor esperado pelo receptor, ou seja, seja um segmento adiantado na ordem de chegada, então ele é colocado na **filaReceiveSemAck** na ordem de acordo com o número de sequência. Quando o segmento que chega ao destino é um segmento com o número de sequência igual ao esperado, então esse segmento e todos os segmentos da **filaReceiveSemAck** adjacentes a partir do segmento recém chegado são colocados no buffer de saída, o **rx_tcp_app**. Depois disso é enviado um ack para o remetente com o número de reconhecimento esperado para o próximo segmento.

```
void validarReceive(SegmentoTCP segmento) {

  if(!segmento.comparaChecksum())
```




```
4       return;

6     if(segmento.numSequencia == numSequenciaEsperado){
        rx_tcp_app.produz(segmento.data);
8       numSequenciaEsperado = segmento.numSequencia + segmento.data.
        length;

10      boolean parar = false;
        while(!parar){
12        SegmentoTCP seg = (SegmentoTCP)filaReceiveSemAck.getHeader().
        getObject();
          if(seg.numSequencia == numSequenciaEsperado){
14          filaReceiveSemAck.removeHeader();

16          rx_tcp_app.produz(seg.data);
            numSequenciaEsperado = seg.numSequencia + seg.data.length;
18        }else{
            parar = true;
20        }
        }
22
        SegmentoTCP ack = SegmentoTCP.getSegmentoACK();
24      ack.numReconhecimento = numSequenciaEsperado;
        ack.janelaRecepcao = rx_tcp_app.size();
26      ipv4.send(ack.getSegmentoTCPemFormatoArrayChar());
      }else{
28      colocarReceiveNaFilaSemAck(segmento);
      }
30  }
```

/home/vicente/workspaceCDT/epos_p1/include/exemplo_socketTCPvalidarreceive.h