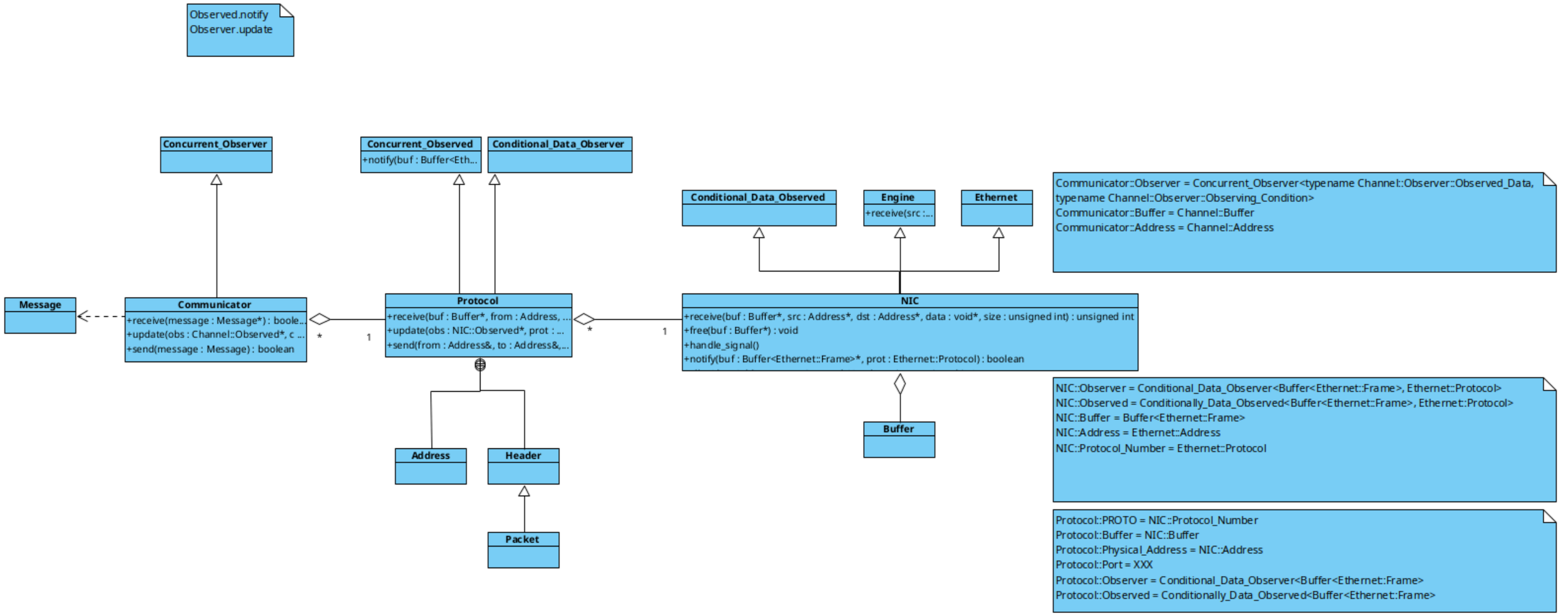
Entrega 1

Sistemas Operacionais 2

Matheus Fernandes Bigolin
Pedro Augusto da Fontoura
Pedro Henrique De Sena Trombini Taglialenha
Vitor Praxedes Calegari


Observed.notify
Observer.update
Concurrent_Observer
Concurrent_Observed
+notify(buf : Buffer<Eth...
Conditional_Data_Observer
Conditional_Data_Observed
Engine
+receive(src :...
Ethernet
Communicator::Observer = Concurrent_Observer<typename Channel::Observer::Observed_Data, typename Channel::Observer::Observing_Condition>
Communicator::Buffer = Channel::Buffer
Communicator::Address = Channel::Address
Message
Communicator
+receive(message : Message*) : boole...
+update(obs : Channel::Observed*, c ...
+send(message : Message) : boolean
*
1
Protocol
+receive(buf : Buffer*, from : Address, ...
+update(obs : NIC::Observed*, prot : ...
+send(from : Address&, to : Address&,...
*
1
NIC
+receive(buf : Buffer*, src : Address*, dst : Address*, data : void*, size : unsigned int) : unsigned int
+free(buf : Buffer*) : void
+handle_signal()
+notify(buf : Buffer<Ethernet::Frame>*, prot : Ethernet::Protocol) : boolean
Buffer
NIC::Observer = Conditional_Data_Observer<Buffer<Ethernet::Frame>, Ethernet::Protocol>
NIC::Observed = Conditionally_Data_Observed<Buffer<Ethernet::Frame>, Ethernet::Protocol>
NIC::Buffer = Buffer<Ethernet::Frame>
NIC::Address = Ethernet::Address
NIC::Protocol_Number = Ethernet::Protocol
Address
Header
Packet
Protocol::PROTO = NIC::Protocol_Number
Protocol::Buffer = NIC::Buffer
Protocol::Physical_Address = NIC::Address
Protocol::Port = XXX
Protocol::Observer = Conditional_Data_Observer<Buffer<Ethernet::Frame>
Protocol::Observed = Conditionally_Data_Observed<Buffer<Ethernet::Frame>

Responsabilidades dentro da arquitetura

Hierarquia dos componentes:

Engine -> NIC -> Protocol -> Communicator

- **NIC**

Fachada sobre a Engine. Responsável por:

  - Gerenciar um pool de objetos Buffer (_buffer_pool) que encapsulam a memória alocada pela Engine.
  - Orquestrar o fluxo de envio (send) e recepção (handle_signal).
  - Implementar o padrão *Observed* para despachar frames recebidos aos Protocols corretos baseado no EtherType (notify).
  - Manter estatísticas (_statistics).

- **Engine**
  - Interação direta com Socket.
  - Alocação/Liberação de memória bruta para os frames (allocate_frame_memory, free_frame_memory).
  - Configuração de interface via ioctl e filtragem inicial (BPF).
  - Configuração de notificação de eventos de I/O via Sinais (SIGIO).

- **Protocol / Comunicator**
  - Camadas superiores que lidam com a lógica específica do protocolo, demultiplexação por porta e API para a aplicação.

Responsabilidades dentro da arquitetura

- **Protocol**
  - Herda de NIC::Observer (Conditional_Data_Observer) para receber frames da NIC via update.
  - Atua como Observed (Concurrent_Observed) para notificar Communicators baseado na porta (lógica interna).
  - Define formato do pacote (Protocol::Packet) dentro do payload Ethernet.

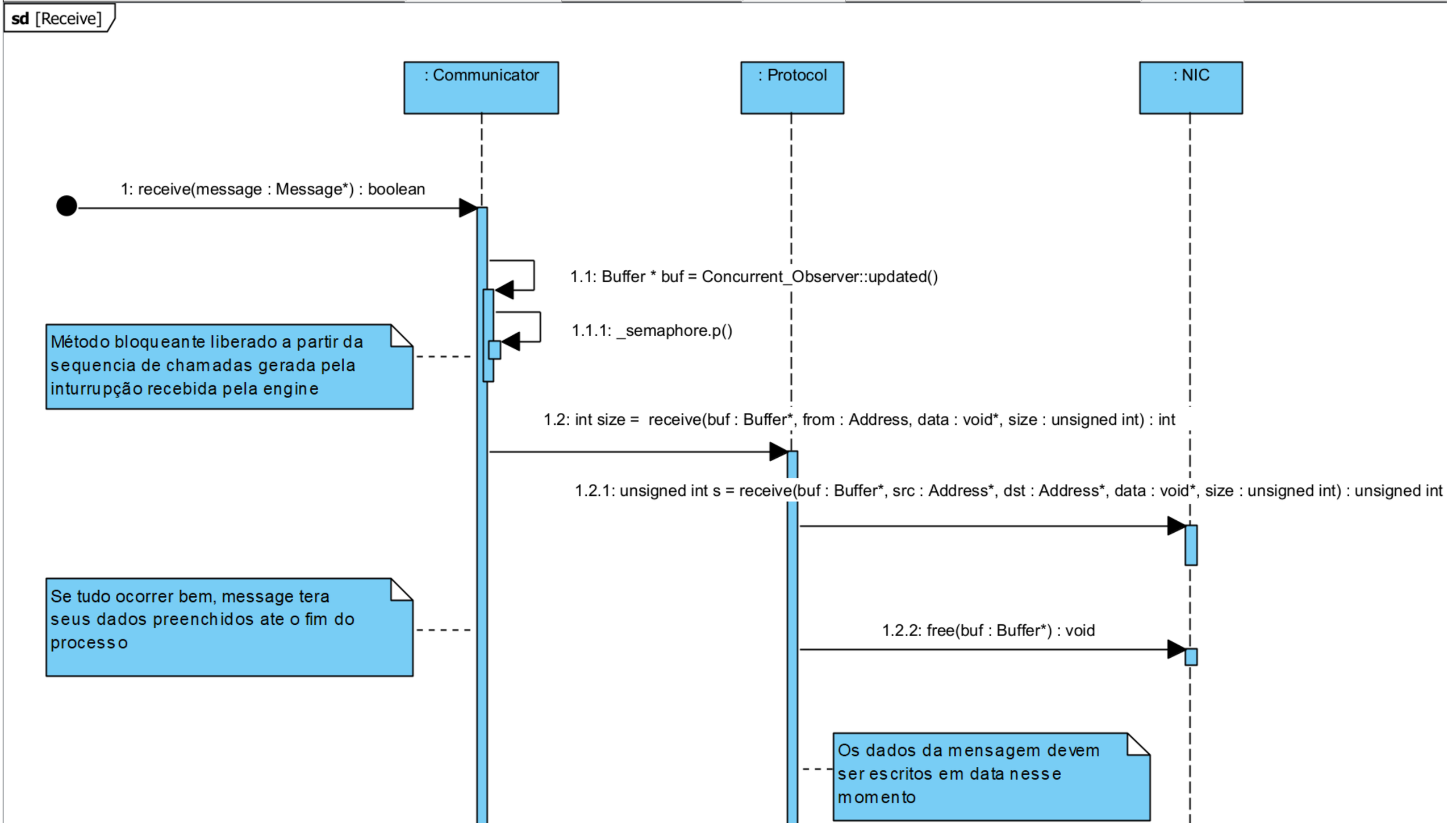
- **Communicator**
  - Fornece API send/receive síncrona (bloqueante no receive) para a aplicação.

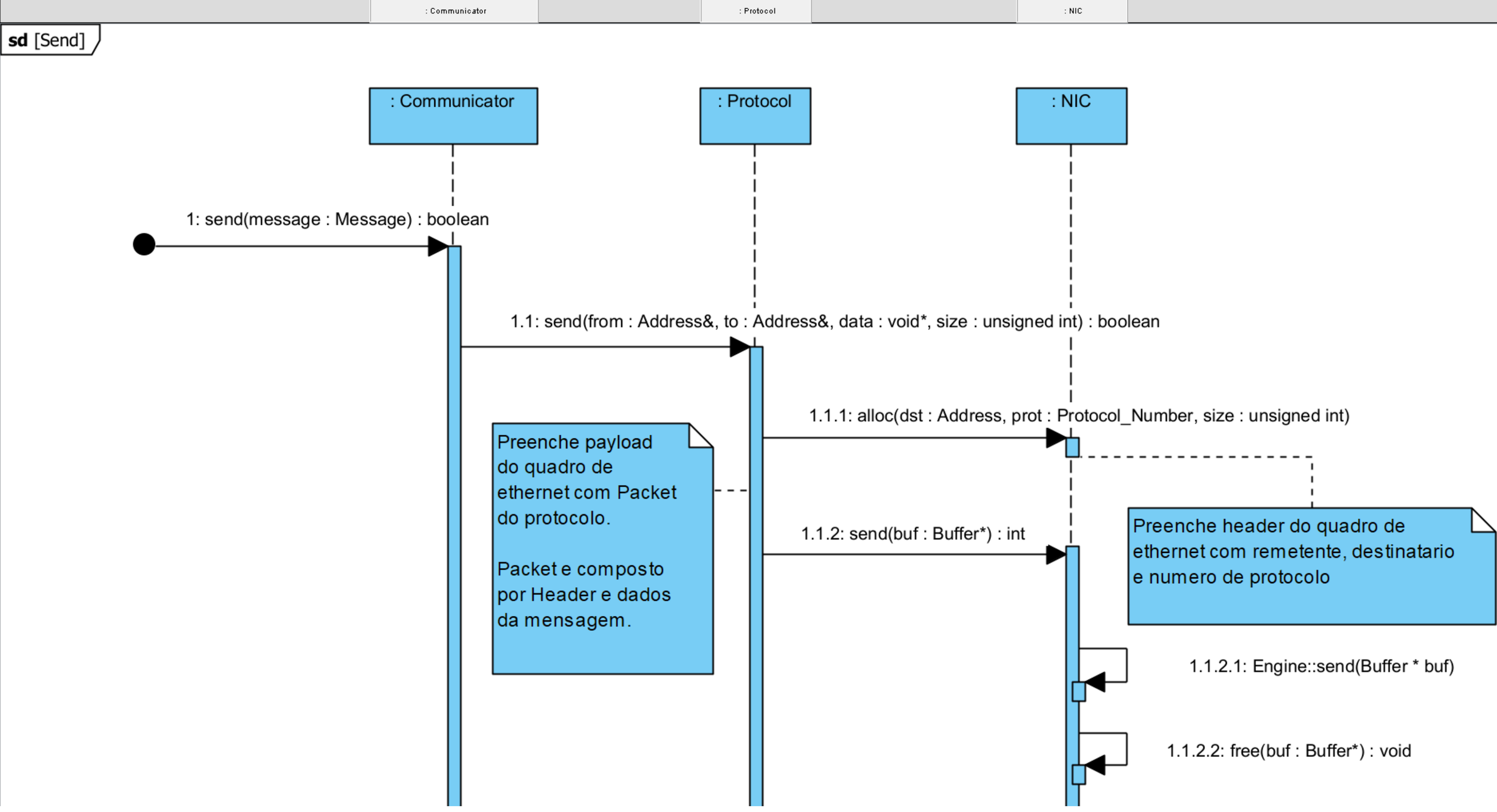
- **Buffers**
  - Buffer encapsula ponteiro (data()) e metadados (size, in_use), não gerencia memória diretamente.

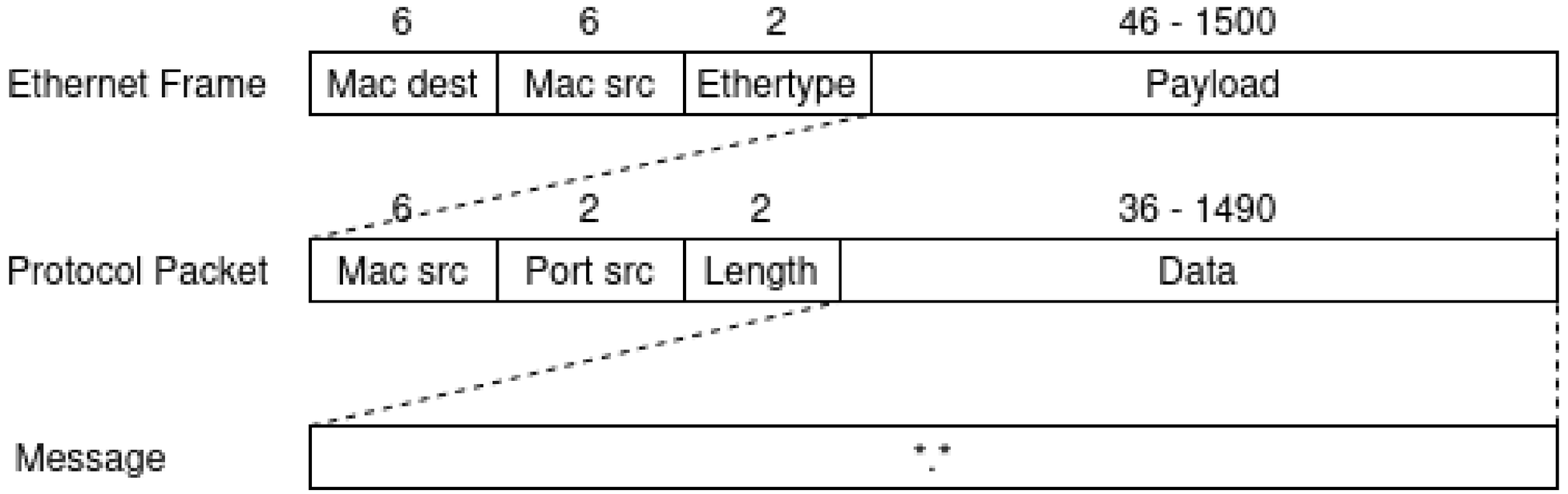
Observers

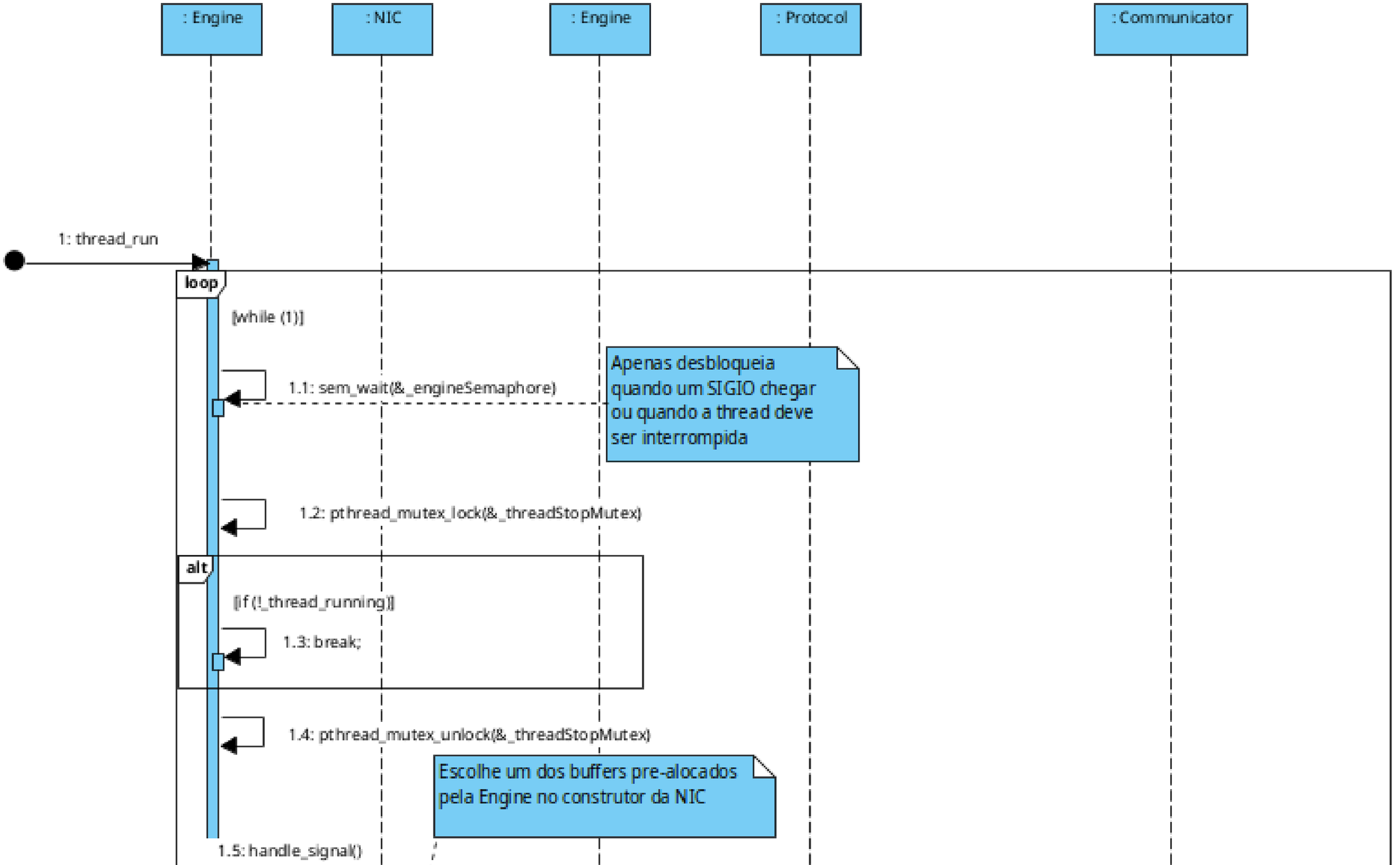
- **NIC (Observed) -> Protocol (Observer)**
  - NIC herda de Conditionally_Data_Observed<BufferNIC, Ethernet::Protocol>: Atua como a entidade observada.
  - Protocol herda de NIC::Observer (Conditional_Data_Observer<...>): Atua como o observador.
  - **Registro:** Protocol chama _nic->attach(this, PROTO) no construtor, registrando interesse no seu EtherType específico (PROTO).
  - **Notificação:** NIC::handle_signal chama this->notify(ether_type, buf). notify encontra o Protocol cujo rank() (EtherType) corresponde e chama seu método update(..., ether_type, buf).
  - Permite que múltiplos protocolos coexistam na mesma interface física, com a NIC direcionando os frames corretamente.

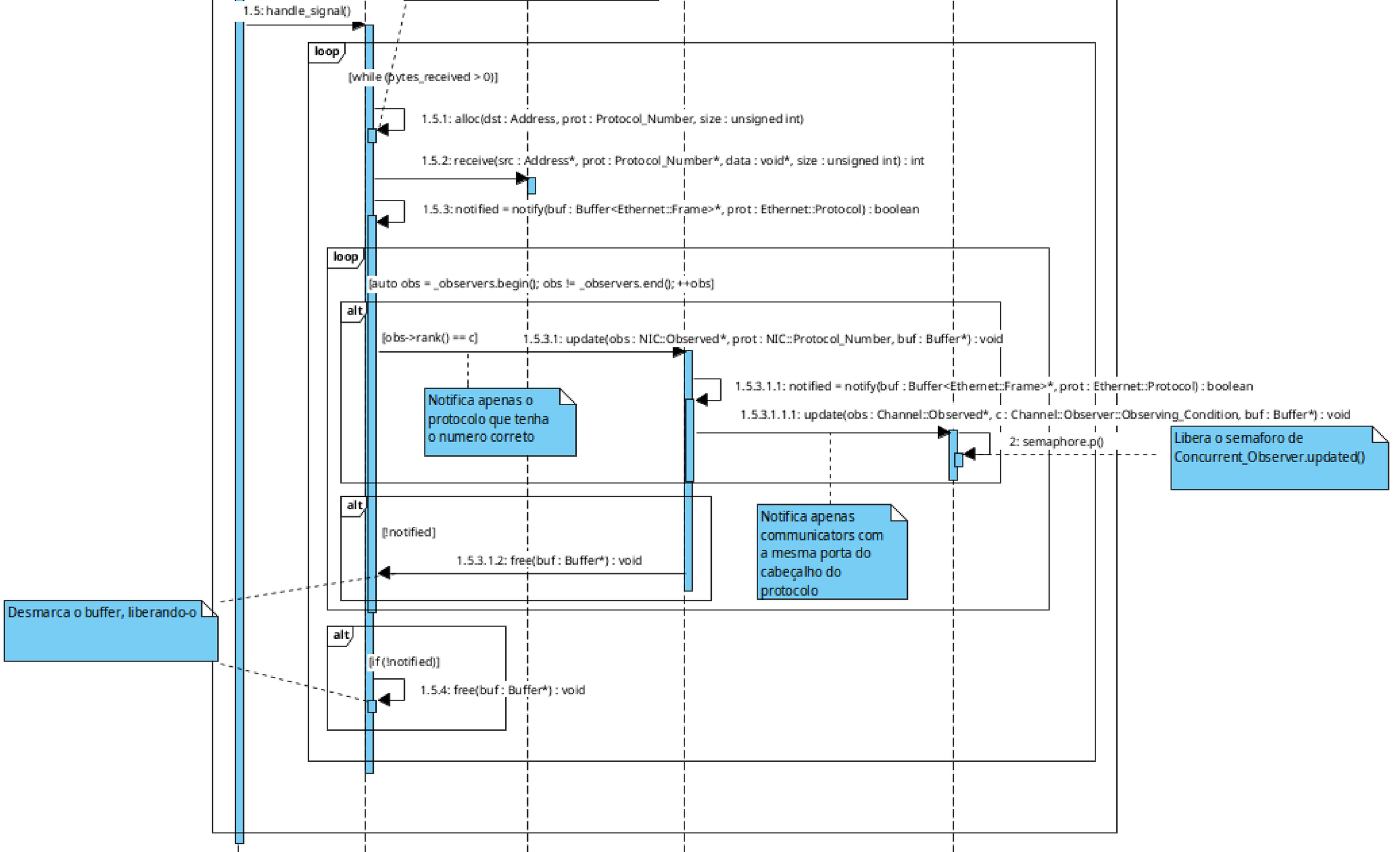
- **Protocol (Observed) -> Communicator (Observer)**
  - Protocol herda de Concurrent_Observed<BufferNIC, Port>: Atua como observado para seus clientes (Communicators).
  - Communicator atua como observador final.
  - **Registro:** Communicator chama _channel->attach(this, port) no construtor, registrando interesse em uma Port específica.
  - **Notificação:** Protocol::update (chamado pela NIC) extrai a porta do cabeçalho do protocolo e chama this->notify(port, buf). notify encontra o Communicator cujo rank() (Porta) corresponde e chama seu update(..., port, buf).
  - Usa Concurrent_Observer com std::counting_semaphore para permitir que Communicator::receive bloqueie até a chegada de um pacote.


sd [Receive]
: Communicator
: Protocol
: NIC
1: receive(message : Message*) : boolean
1.1: Buffer * buf = Concurrent_Observer::updated()
1.1.1: _semaphore.p()
Método bloqueante liberado a partir da sequencia de chamadas gerada pela inturrupção recebida pela engine
1.2: int size = receive(buf : Buffer*, from : Address, data : void*, size : unsigned int) : int
1.2.1: unsigned int s = receive(buf : Buffer*, src : Address*, dst : Address*, data : void*, size : unsigned int) : unsigned int
Se tudo ocorrer bem, message tera seus dados preenchidos ate o fim do processo
1.2.2: free(buf : Buffer*) : void
Os dados da mensagem devem ser escritos em data nesse momento


: Communicator
: Protocol
: NIC
sd [Send]
: Communicator
: Protocol
: NIC
1: send(message : Message) : boolean
1.1: send(from : Address&, to : Address&, data : void*, size : unsigned int) : boolean
1.1.1: alloc(dst : Address, prot : Protocol_Number, size : unsigned int)
Preenche payload do quadro de ethernet com Packet do protocolo.
Packet e composto por Header e dados da mensagem.
1.1.2: send(buf : Buffer*) : int
Preenche header do quadro de ethernet com remetente, destinatario e numero de protocolo
1.1.2.1: Engine::send(Buffer * buf)
1.1.2.2: free(buf : Buffer*) : void


6
6
2
46 - 1500
Ethernet Frame
Mac dest
Mac src
Ethertype
Payload
6
2
2
36 - 1490
Protocol Packet
Mac src
Port src
Length
Data
Message


: Engine
: NIC
: Engine
: Protocol
: Communicator
1: thread_run
loop
[while (1)]
1.1: sem_wait(&_engineSemaphore)
Apenas desbloqueia quando um SIGIO chegar ou quando a thread deve ser interrompida
1.2: pthread_mutex_lock(&_threadStopMutex)
alt
[if (!_thread_running)]
1.3: break;
1.4: pthread_mutex_unlock(&_threadStopMutex)
Escolhe um dos buffers pre-alocados pela Engine no construtor da NIC
1.5: handle_signal()


1.5: handle_signal()
loop
[while (bytes_received > 0)]
1.5.1: alloc(dst : Address, prot : Protocol_Number, size : unsigned int)
1.5.2: receive(src : Address*, prot : Protocol_Number*, data : void*, size : unsigned int) : int
1.5.3: notified = notify(buf : Buffer<Ethernet::Frame>*, prot : Ethernet::Protocol) : boolean
loop
[auto obs = _observers.begin(); obs != _observers.end(); ++obs]
alt
[obs->rank() == c]
1.5.3.1: update(obs : NIC::Observed*, prot : NIC::Protocol_Number, buf : Buffer*) : void
Notifica apenas o protocolo que tenha o numero correto
1.5.3.1.1: notified = notify(buf : Buffer<Ethernet::Frame>*, prot : Ethernet::Protocol) : boolean
1.5.3.1.1.1: update(obs : Channel::Observed*, c : Channel::Observer::Observing_Condition, buf : Buffer*) : void
2: semaphore.p()
Libera o semaforo de Concurrent_Observer.updated()
alt
[!notified]
1.5.3.1.2: free(buf : Buffer*) : void
Notifica apenas communicators com a mesma porta do cabeçalho do protocolo
Desmarca o buffer, liberando-o
alt
[if (!notified)]
1.5.4: free(buf : Buffer*) : void

ASYNC-SIGNAL SAFETY top

The mutex functions are not async-signal safe. What this means is that they should not be called from a signal handler. In particular, calling **pthread_mutex_lock** or **pthread_mutex_unlock** from a signal handler may deadlock the calling thread.

https://man7.org/linux/man-pages/man3/pthread_mutex_lock.3.html

SA_NODEFER

Do not add the signal to the thread's signal mask while the handler is executing, unless the signal is specified in *act.sa_mask*. Consequently, a further instance of the signal may be delivered to the thread while it is executing the handler. This flag is meaningful only when establishing a signal handler.

https://man7.org/linux/man-pages/man2/sigaction.2.html

Libera semáforo da thread de recebimento

```cpp
void Engine::setupSignalHandler() {
  // Armazena a função de callback
  struct sigaction sigAction;
  sigAction.sa_handler = Engine::signalHandler;
  sigAction.sa_flags = SA_RESTART;

  // Limpa possiveis sinais existentes antes da configuracao
  sigemptyset(&sigAction.sa_mask);

  // Configura sigaction
  // nullptr indica que nao queremos salvar a sigaction anterior
  if (sigaction(SIGIO, &sigAction, nullptr) < 0) {
    perror("sigaction");
    exit(EXIT_FAILURE);
  }
}
```

```cpp
recvThread = std::thread([this]() {
    while (1) {
        sem_wait(&_engineSemaphore);
        pthread_mutex_lock(&_threadStopMutex);
        if (!_thread_running) { break; }
        pthread_mutex_unlock(&_threadStopMutex);
        _self->handler(_self->obj);
    }
});
```

Teste de carga

- Instanciamos 15 processos de envio
- Instanciamos 1 processo de recebimento
- Cada processo de envio manda 100 mensagens ao de recebimento

Teste Ping Pong

- Instanciamos um processo para verificação e um para contagem
- O processo de verificação envia uma mensagem com o valor atual do contador e espera receber o próximo valor
- O processo de contagem recebe a mensagem, adiciona a contagem e reenvia o novo valor do contador

Teste de Latência

- Instanciamos um processo de envio e um de recebimento
- As mensagens do processo de envio são enviadas com tempo de seu envio
- No processo de recebimento, a latência é calculada ao obter o tempo contido na mensagem recebida e diminuir do tempo em que ela foi obtida

```
Sent: 99999
Received: 99999
Latência média observada: 44.4451 µs
```