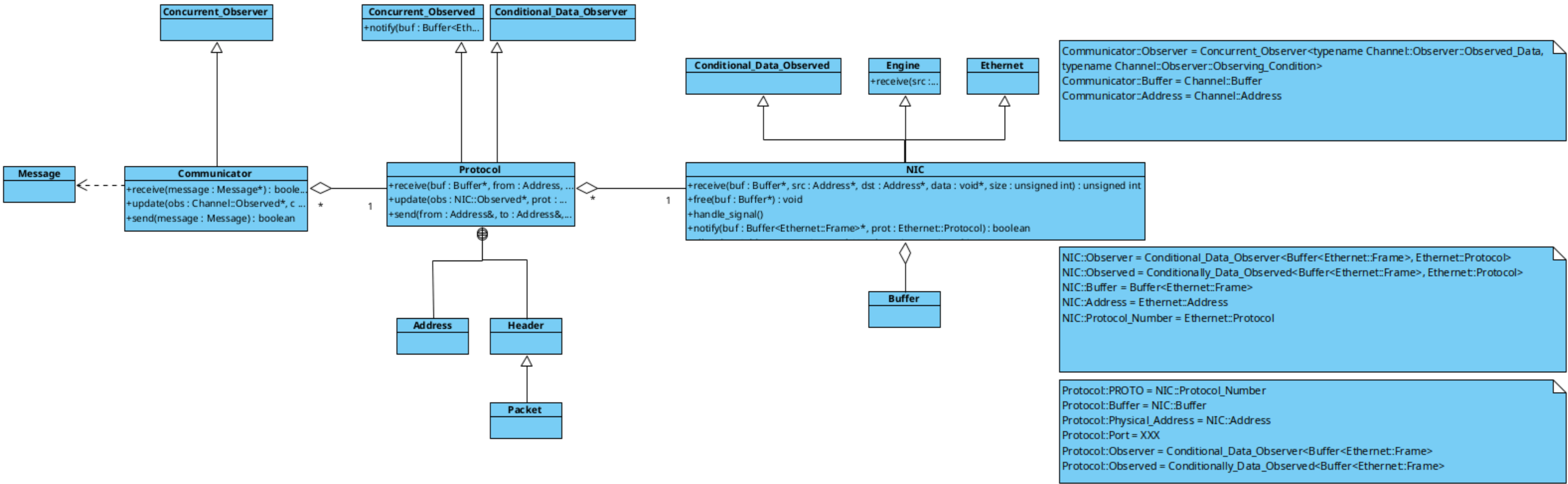


Observed.notify  
Observer.update



# 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 / Communicator**

- 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.

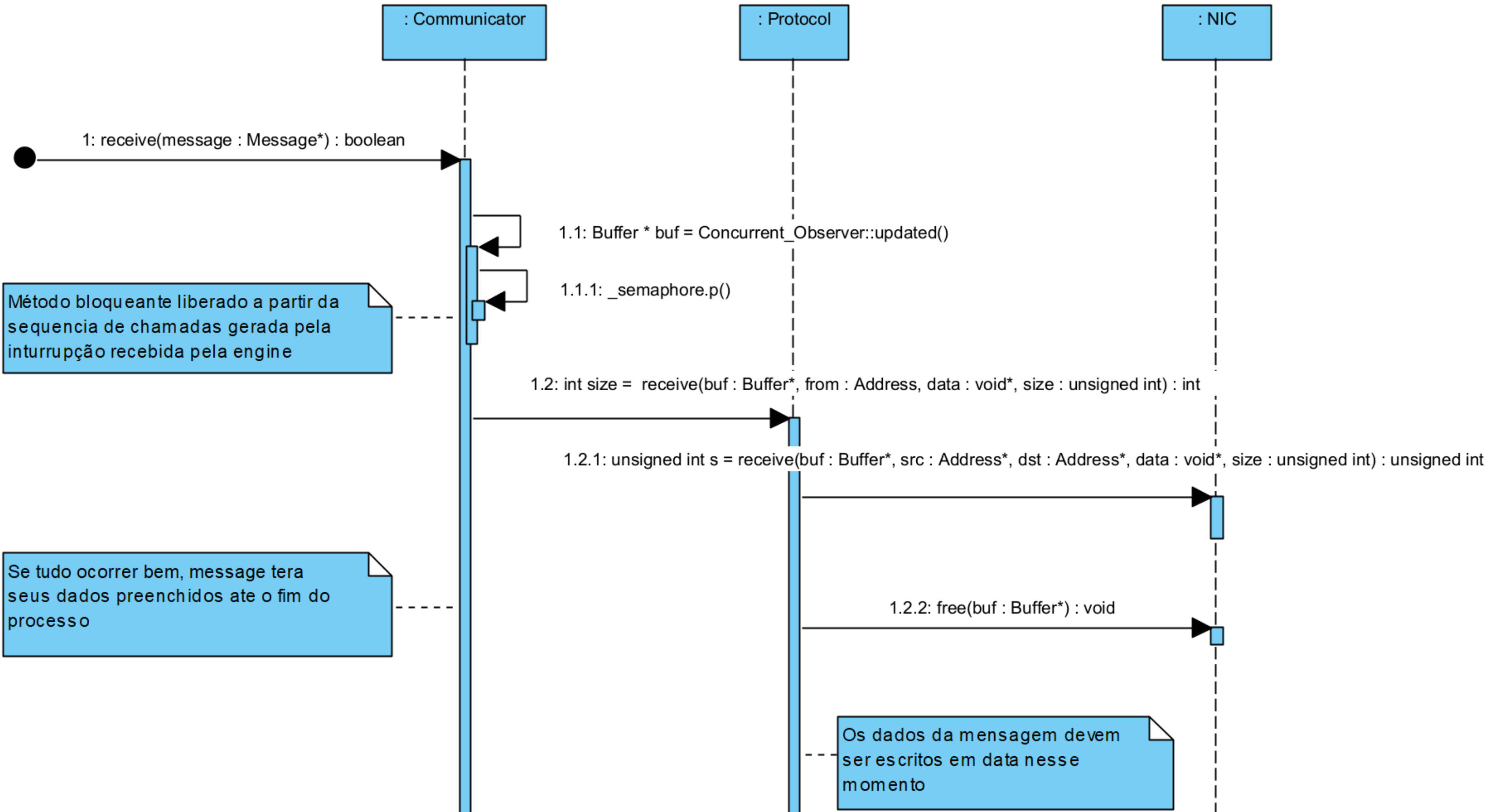
# Observers

- **NIC (Observed) -> Protocol (Observer)**

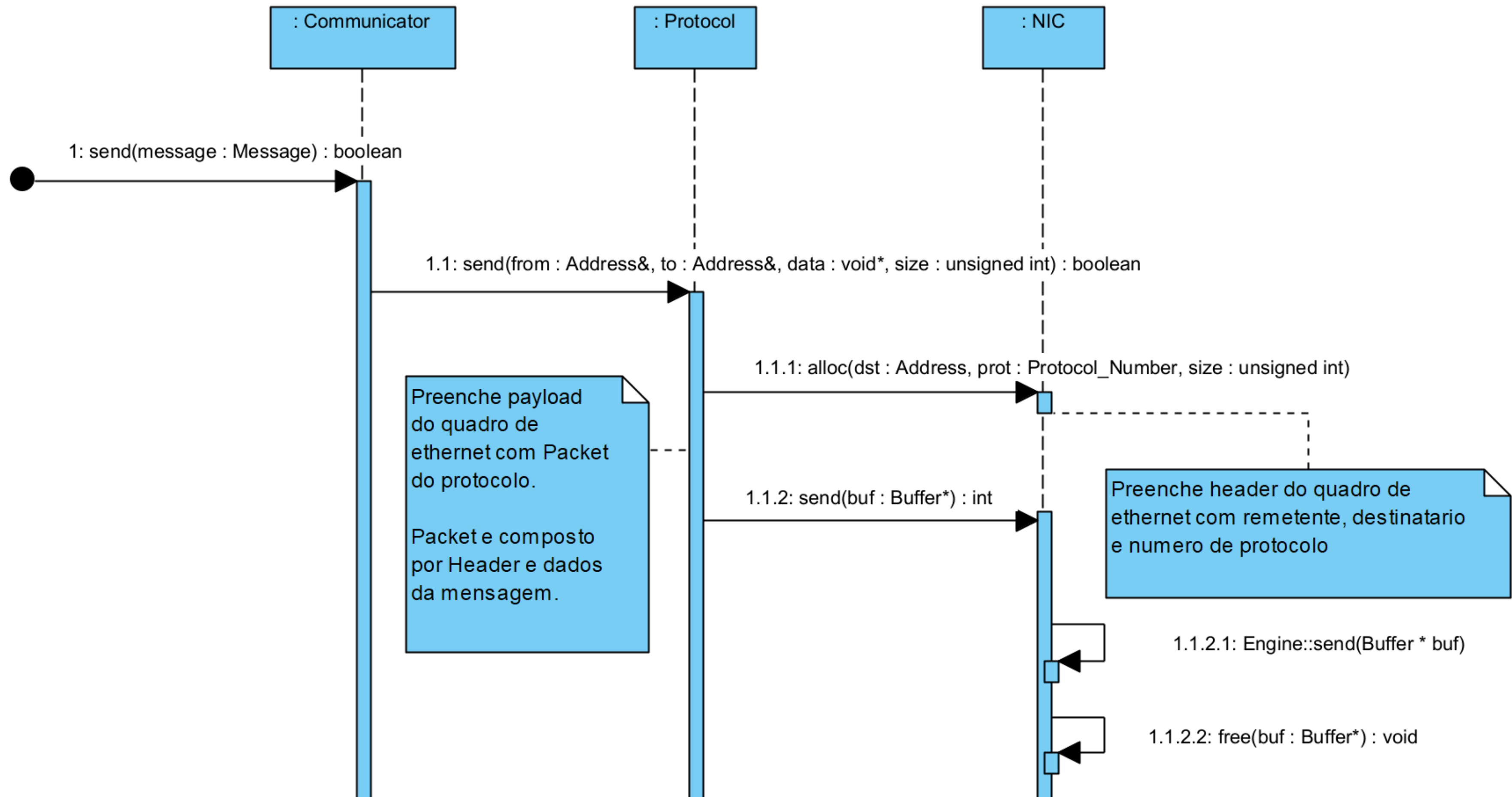
- NIC herda de `Conditionally_Data_Observed<BufferNIC, Ethernet::Protocol>`: Atua como a entidade observada.
- Protocol herda de `NIC::Observer (Conditional_Data_Observer<...>)`: Atua como o observador.
- **Registro:** Protocol chama `_nic->attach(this, PROTO)` no construtor, registrando interesse no 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.

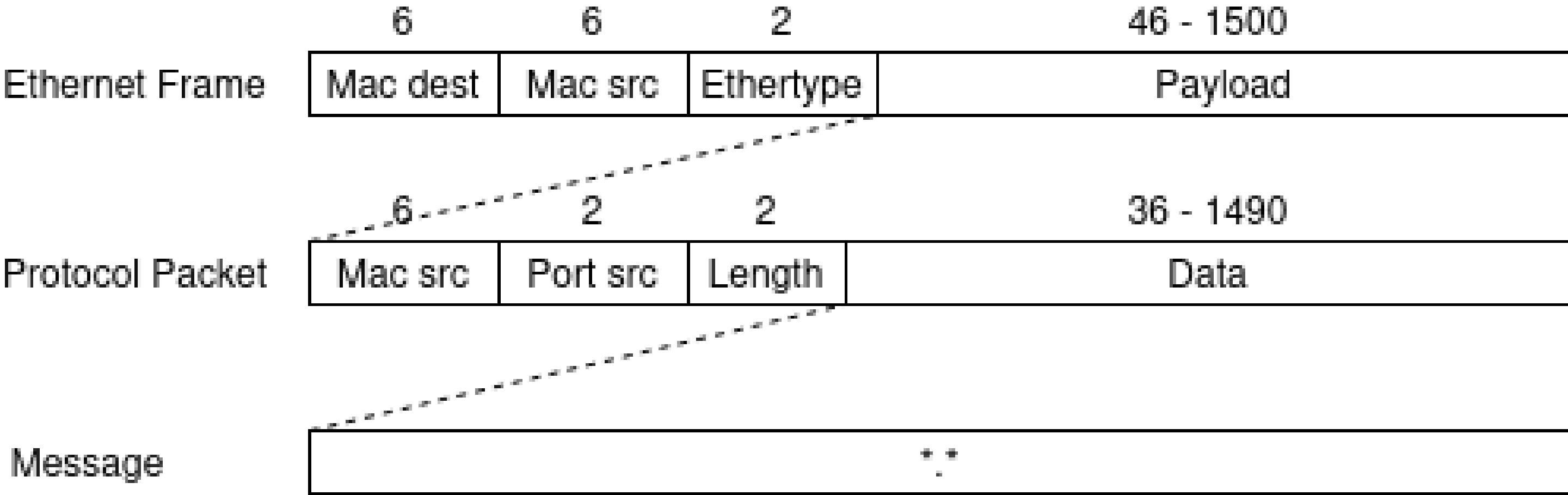
- **Protocol (Observed) -> Communicator (Observer)**

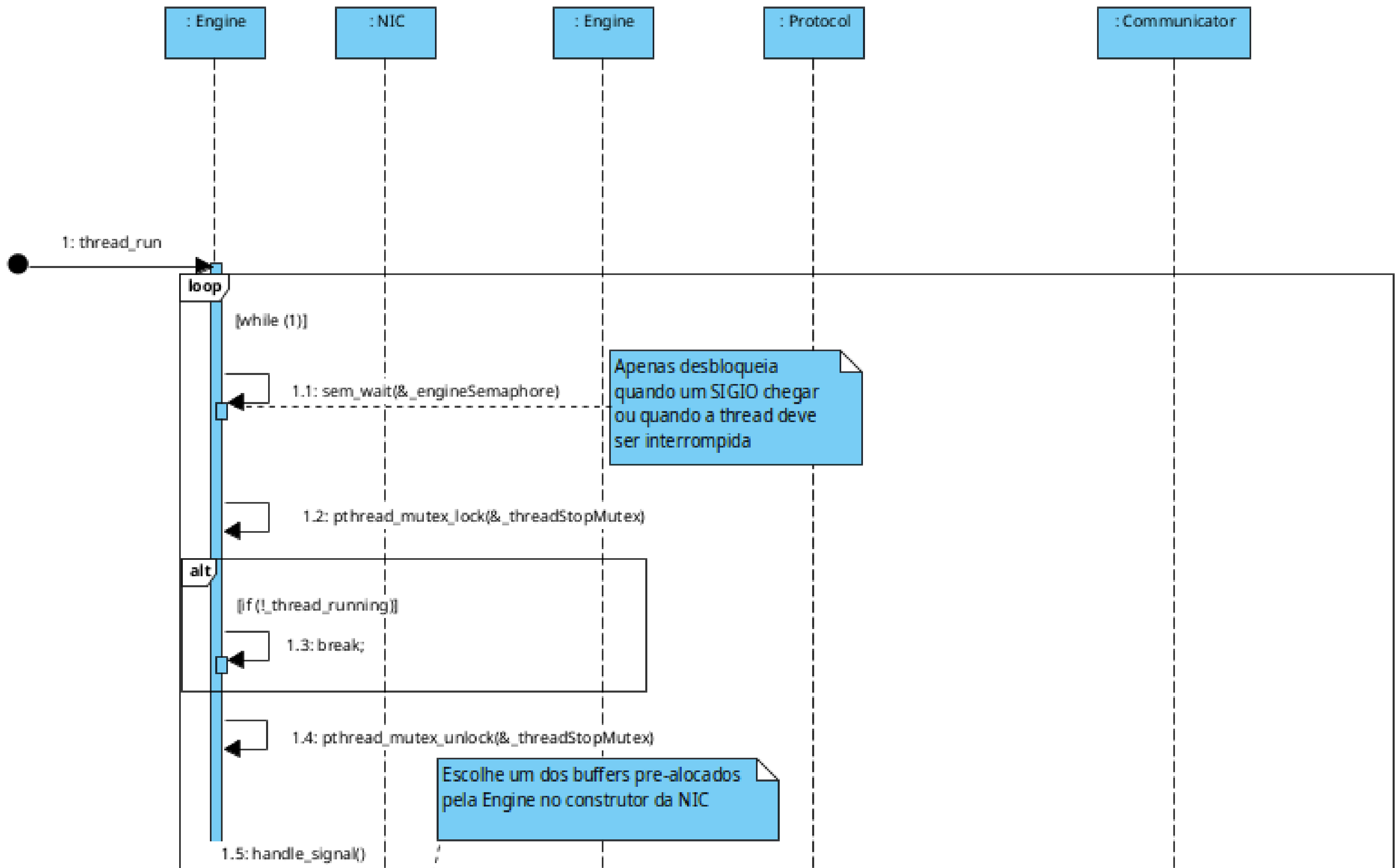
- Protocol herda de `Concurrent_Observed<BufferNIC, Port>`: Atua como observado para seus clientes (Communicators).
- Communicator atua como observador final.
- **Registro:** Communicator chama `_channel->attach(this, port)` no construtor, registrando interesse em uma Port específica.
- **Notificação:** `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.



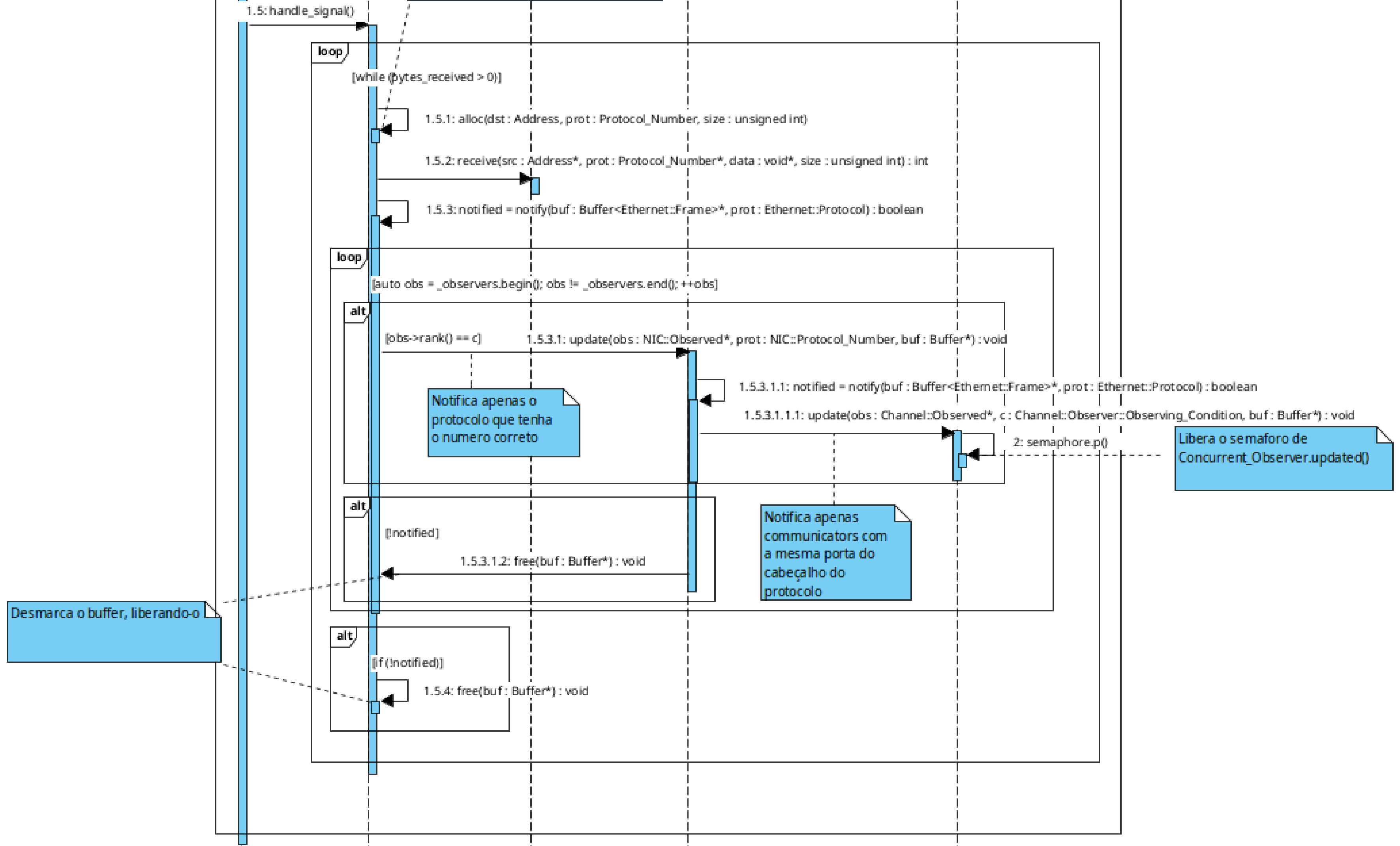
sd [Send]











## 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](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

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

    // Limpa possíveis 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);
    }
}
```

```
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
```