Implementação da API de Comunicação com Memória Compartilhada

Grupo A: Vitor Calegari, Matheus Bigolin, Pedro Fountoura, Pedro Taglialenha

Comunicação Inter e Intra-Sistema

Problema a ser resolvida na segunda entrega:

- Suportar comunicação entre sistemas autônomos (processos POSIX, via rede) e entre componentes (threads POSIX, dentro de um processo).

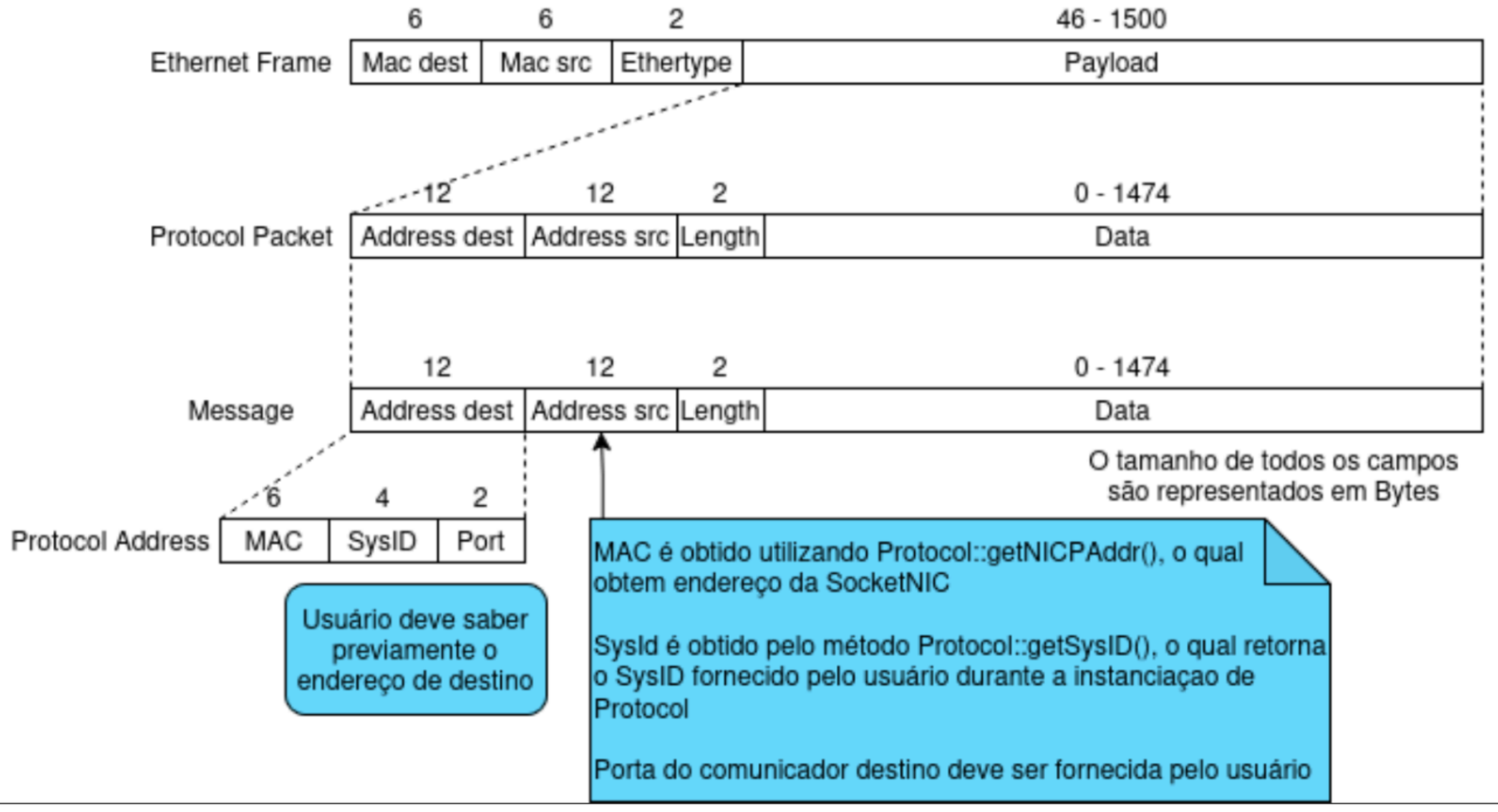
Adaptações do protocolo para nova Nic

- O Protocol agora é template em SocketNIC e SharedMemNIC, gerenciando instâncias de ambas para acessar rede e memória compartilhada.
- Implementa a lógica de decisões de:
  - Envio (send): Usa o SysID de destino para escolher entre SocketNIC ou SharedMemNIC.
  - Recepção (update / receive): Usa SysID para validar destino e direcionar o processamento/liberação do buffer para a NIC correta.
- Preenchimento de cabeçalho ETH: Assume a responsabilidade de preencher o cabeçalho Ethernet no send, necessário para abstrair a SharedMemNIC que não lida com camada 2.
- Suporte a Broadcast: Adiciona lógica específica para lidar com broadcast de porta e de sistema.

Concorrência

Mutexes para Estruturas compartilhadas

- Adicionados std::mutex às classes Conditionally_Data_Observed e Concurrent_Observed para proteger as listas de observers (_observers) durante attach, detach e notify. Evita condição de corrida quando múltiplas threads modificam/acessam a lista.
- A recepção assíncrona via SIGIO agora utiliza uma std::condition_variable para acordar a thread dedicada (recvThread).



Durante o envio, Protocol é encarregado de:
1) Alocar um Buffer por uma das NICs
2) Preencher o cabeçalho Ethernet
3) Copiar a mensagem para o Pacote do protocolo, o qual é o conteúdo do payload do quadro Ethernet

Durante o recebimento o conteúdo do payload é copiado do Buffer pela NIC para um Pacote de Protocol.

Protocol preenche a mensagem com os dados do Pacote recebido.

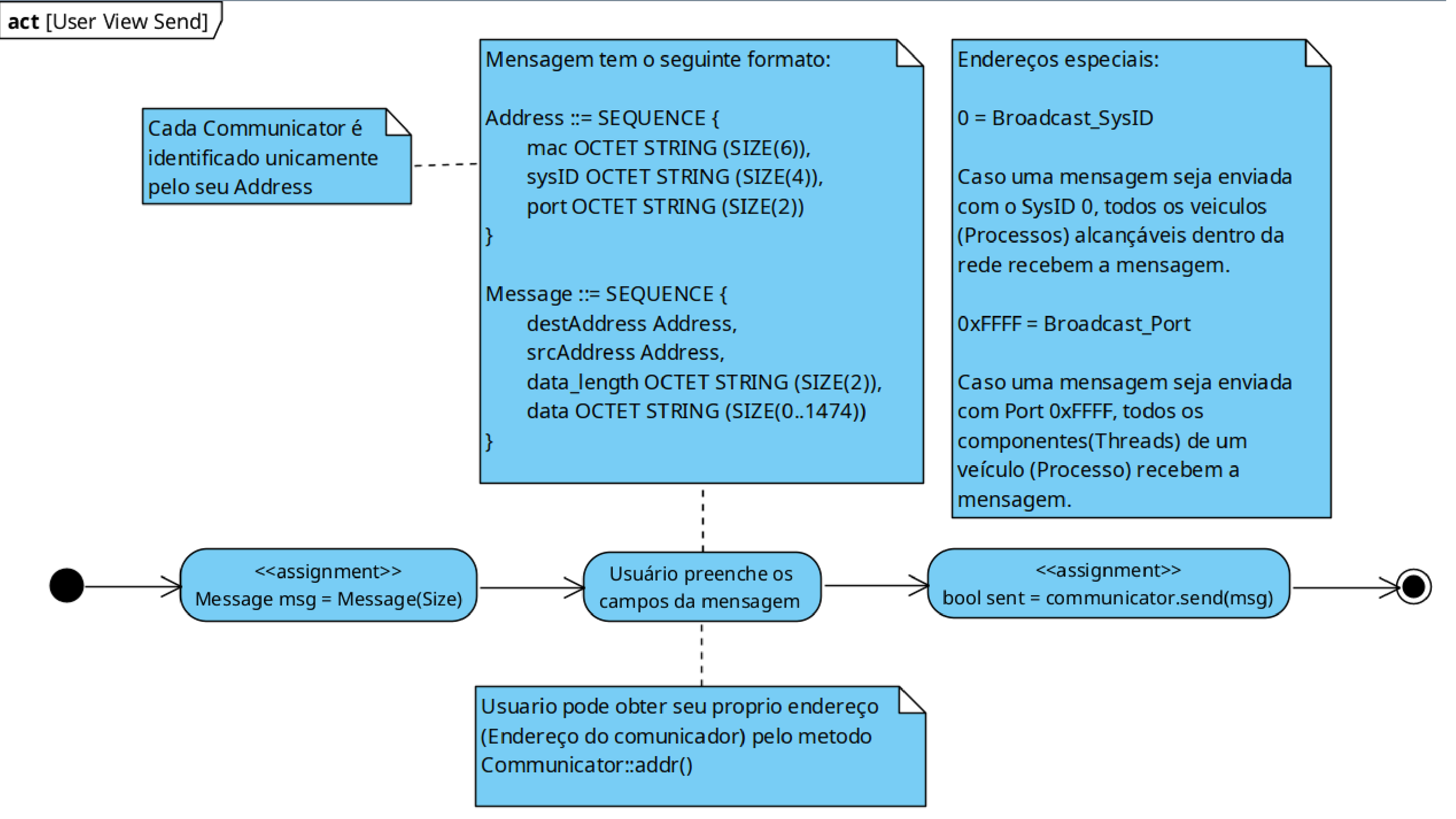
Endereços especiais:

0 = Broadcast_SysID

Caso uma mensagem seja enviada com o SysID 0, todos os veiculos (Processos) alcançáveis dentro da rede recebem a mensagem.

0xFFFF = Broadcast_Port

Caso uma mensagem seja enviada com Port 0xFFFF, todos os componentes(Threads) de um veículo (Processo) recebem a mensagem.


act [User View Send]
Cada Communicator é identificado unicamente pelo seu Address
Mensagem tem o seguinte formato:
Address ::= SEQUENCE {
mac OCTET STRING (SIZE(6)),
sysID OCTET STRING (SIZE(4)),
port OCTET STRING (SIZE(2))
}
Message ::= SEQUENCE {
destAddress Address,
srcAddress Address,
data_length OCTET STRING (SIZE(2)),
data OCTET STRING (SIZE(0..1474))
}
Endereços especiais:
0 = Broadcast_SysID
Caso uma mensagem seja enviada com o SysID 0, todos os veiculos (Processos) alcançáveis dentro da rede recebem a mensagem.
0xFFFF = Broadcast_Port
Caso uma mensagem seja enviada com Port 0xFFFF, todos os componentes(Threads) de um veículo (Processo) recebem a mensagem.
<<assignment>>
Message msg = Message(Size)
Usuário preenche os campos da mensagem
<<assignment>>
bool sent = communicator.send(msg)
Usuario pode obter seu proprio endereço (Endereço do comunicador) pelo metodo Communicator::addr()

Endereçamento MAC:SYS_ID:PORT: Identificação e Roteamento

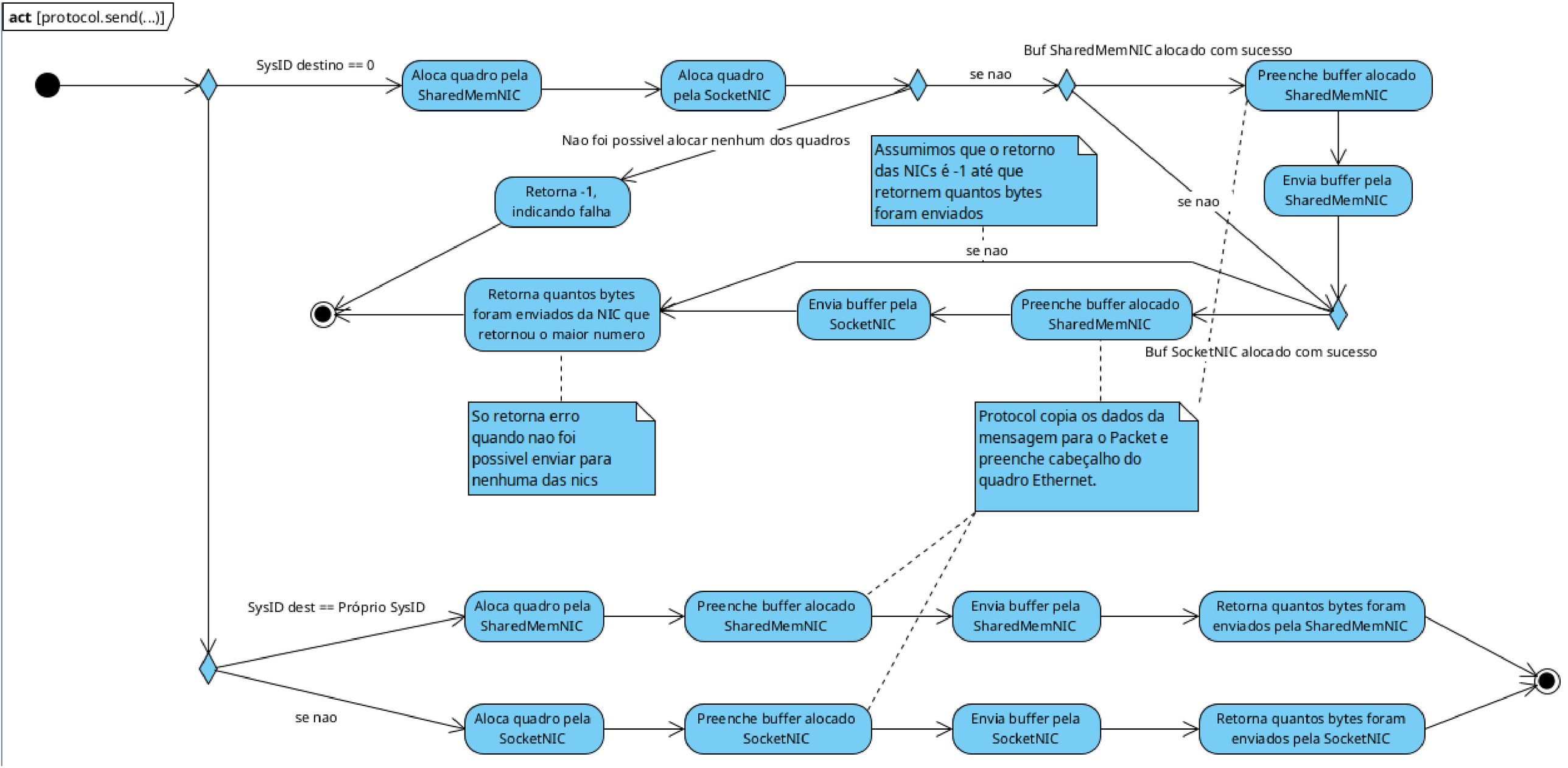
Estrutura Protocol::Address:

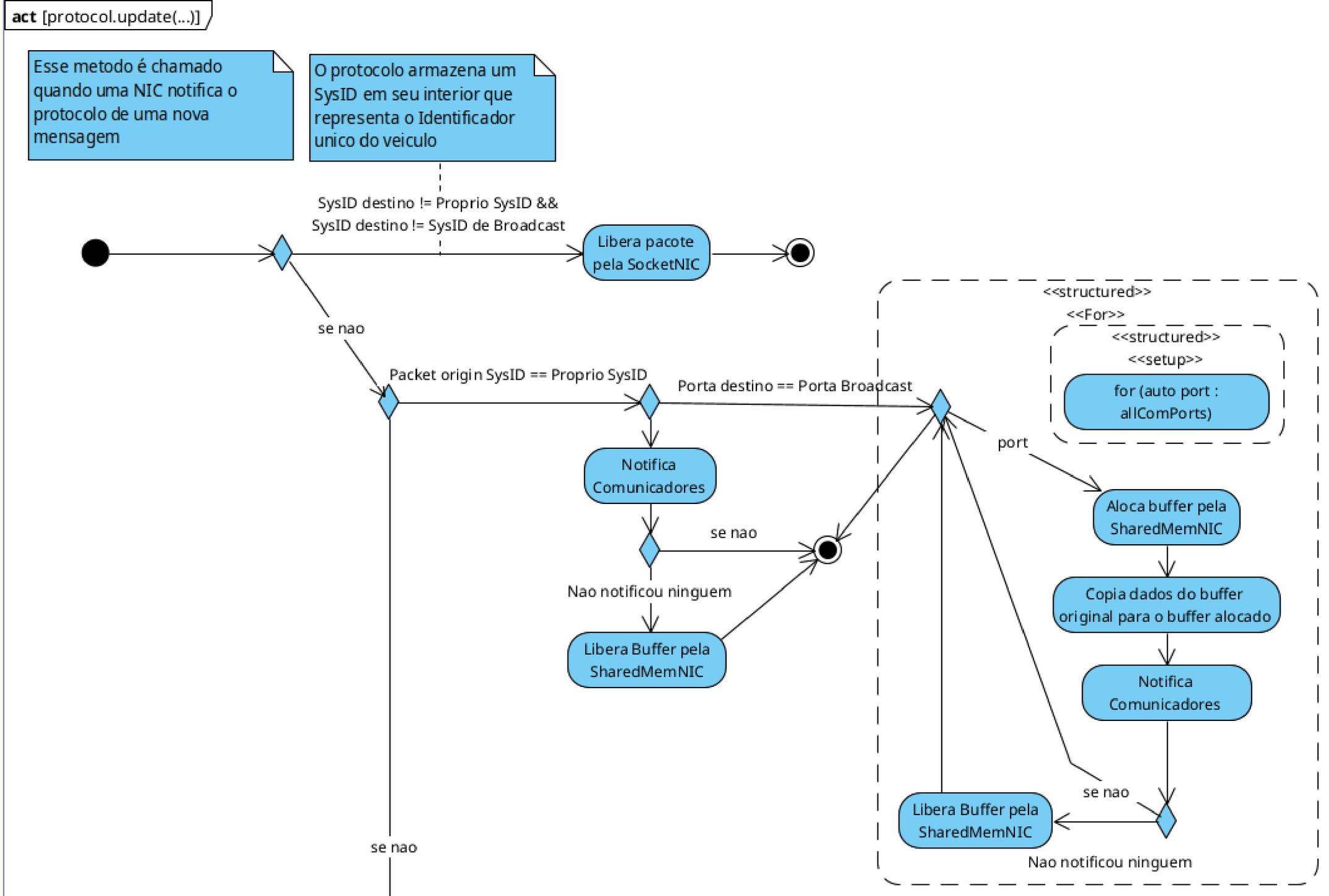
- Physical_Address _paddr (MAC): Identifica a interface física do sistema na rede. Relevante para SocketEngine.
- SysID _sysID (unsigned short): Identifica o Sistema Autônomo (Processo POSIX). É a chave de roteamento principal.
- Port _port (unsigned short): Identifica o Componente (Thread POSIX / Communicator) dentro de um sistema. Usado para demultiplexação final.

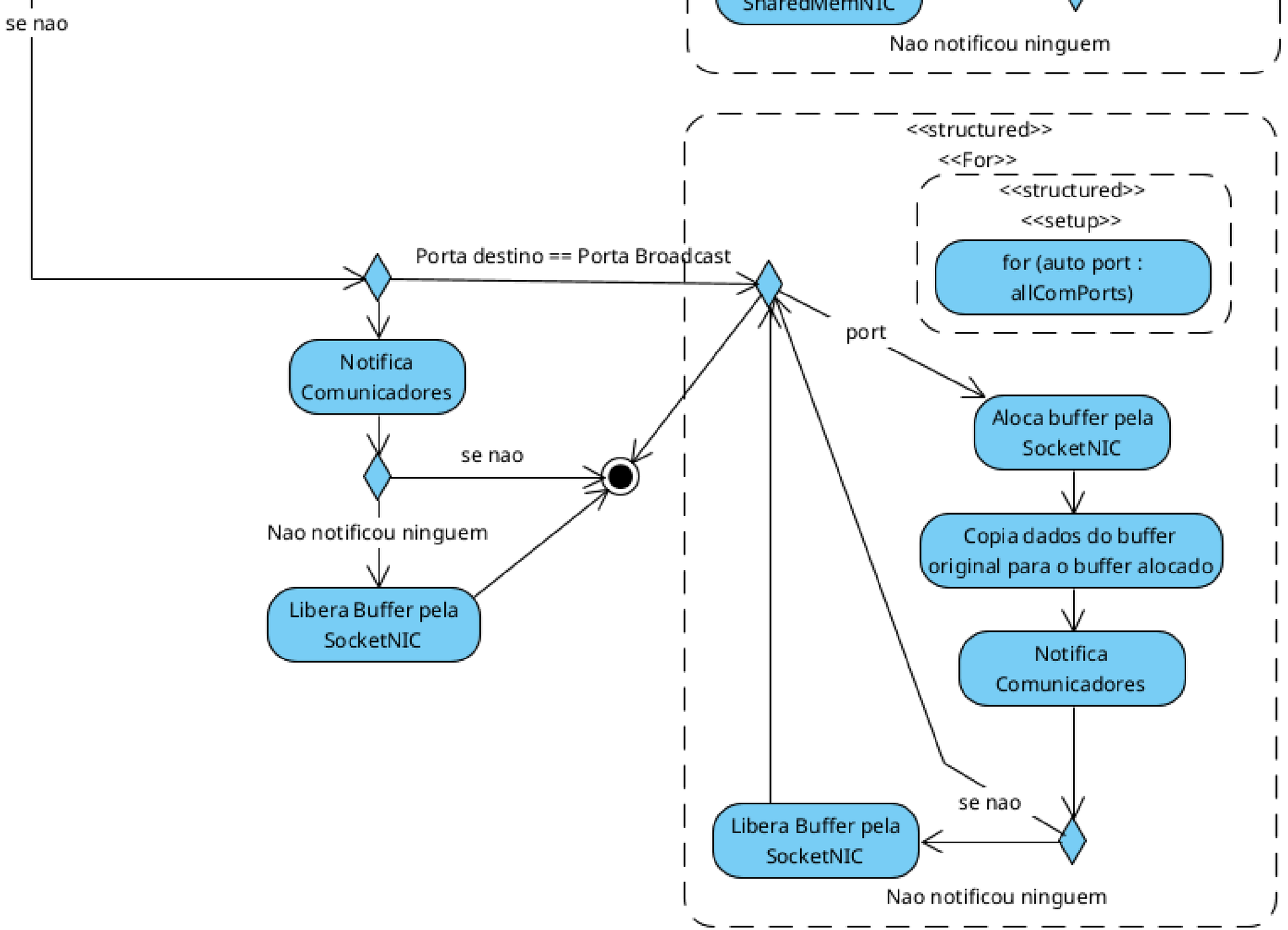
Roteamento no Protocol:

- send: Compara to.getSysID() com _sysID local -> Escolhe _smnic (local) ou _rsnic (rede) para alloc e send.
- update (Recepção): Chamado por _smnic ou _rsnic. Verifica dest.getSysID(). Se não for local, descarta. Se local, usa dest.getPort() para notificar Communicator. (Detalhado no diagrama a seguir)
- receive (Chamado pelo Communicator):
  - Recebe o Buffer* (buf, que contém o frame completo).
  - Cria um objeto Packet pkt localmente na pilha.
  - Determina a NIC de origem
  - Chama o receive da NIC correspondente (_smnic->receive ou _rsnic->receive), passando buf e &pkt.
  - NIC::receive então copia o payload Ethernet (buf->data()->data<char>()) para dentro do objeto pkt.
  - De volta ao Protocol::receive, extrai endereços from/to de pkt.header().
  - Realiza a cópia do pacote final dos dados da aplicação de pkt.template data<char>() para a Message do usuário (void* data).
  - Chama free da NIC correspondente para liberar o buf original.

O Protocol gerencia a complexidade do roteamento e do *unmarshalling* específico da NIC.


act [protocol.send(...)]
SysID destino == 0
Aloca quadro pela SharedMemNIC
Aloca quadro pela SocketNIC
se nao
Buf SharedMemNIC alocado com sucesso
Preenche buffer alocado SharedMemNIC
Envia buffer pela SharedMemNIC
se nao
Nao foi possivel alocar nenhum dos quadros
Retorna -1, indicando falha
Assumimos que o retorno das NICs é -1 até que retornem quantos bytes foram enviados
se nao
Retorna quantos bytes foram enviados da NIC que retornou o maior numero
Envia buffer pela SocketNIC
Preenche buffer alocado SharedMemNIC
Buf SocketNIC alocado com sucesso
So retorna erro quando nao foi possivel enviar para nenhuma das nics
Protocol copia os dados da mensagem para o Packet e preenche cabeçalho do quadro Ethernet.
SysID dest == Próprio SysID
Aloca quadro pela SharedMemNIC
Preenche buffer alocado SharedMemNIC
Envia buffer pela SharedMemNIC
Retorna quantos bytes foram enviados pela SharedMemNIC
se nao
Aloca quadro pela SocketNIC
Preenche buffer alocado SocketNIC
Envia buffer pela SocketNIC
Retorna quantos bytes foram enviados pela SocketNIC


act [protocol.update(...)]
Esse metodo é chamado quando uma NIC notifica o protocolo de uma nova mensagem
O protocolo armazena um SysID em seu interior que representa o Identificador unico do veiculo
SysID destino != Proprio SysID &&
SysID destino != SysID de Broadcast
Libera pacote pela SocketNIC
se nao
Packet origin SysID == Proprio SysID
Porta destino == Porta Broadcast
Notifica Comunicadores
se nao
Nao notificou ninguem
Libera Buffer pela SharedMemNIC
<<structured>>
<<For>>
<<structured>>
<<setup>>
for (auto port : allComPorts)
port
Aloca buffer pela SharedMemNIC
Copia dados do buffer original para o buffer alocado
Notifica Comunicadores
se nao
Libera Buffer pela SharedMemNIC
Nao notificou ninguem
se nao


se nao
SharedMemNIC
Nao notificou ninguem
<<structured>>
<<For>>
<<structured>>
<<setup>>
for (auto port : allComPorts)
Porta destino == Porta Broadcast
port
Notifica Comunicadores
se nao
Nao notificou ninguem
Libera Buffer pela SocketNIC
Aloca buffer pela SocketNIC
Copia dados do buffer original para o buffer alocado
Notifica Comunicadores
se nao
Libera Buffer pela SocketNIC
Nao notificou ninguem

Comunicação Inter e Intra-Sistema

Solução Proposta:

- Introdução de duas implementações da abstração Engine:
  - SocketEngine: Para comunicação inter-sistema (rede Ethernet, Raw Sockets). Usa SIGIO + CondVar para recepção assíncrona.
  - SharedEngine: Para comunicação intra-sistema (entre threads do mesmo processo). Usa memória compartilhada, modelo Produtor-Consumidor.
- A classe Protocol agora gerencia instâncias de ambas as NICs (SocketNIC, SharedMemNIC) e atua como o ponto de roteamento principal.
- O Communicator e a Message oferecem uma interface para a aplicação, abstraindo a complexidade da escolha da via de comunicação.

SocketEngine vs. SharedEngine

- A escolha entre as engines é feita pela camada Protocol

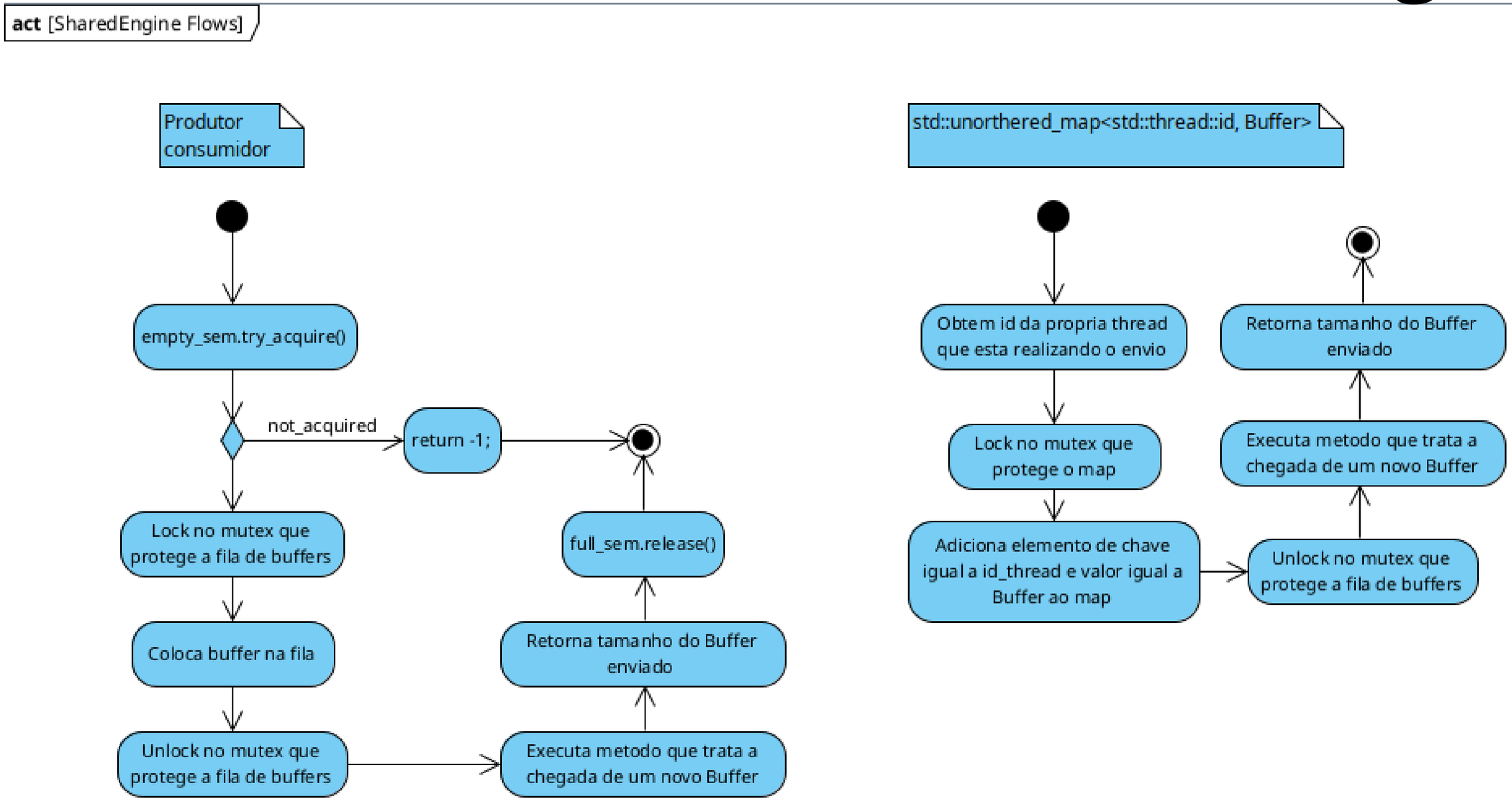
SocketEngine (Rede - Inter - Sistema):

- Usa Raw Sockets (AF_PACKET) para acesso à Camada 2.
- Recepção:SIGIO sinaliza evento -> signalHandler notifica std::condition_variable -> Thread dedicada (recvThread) acorda e chama handler (NIC::handle_signal). CondVar evita processamento direto no handler de sinal.
- Gerencia MAC Address e Index da interface física.

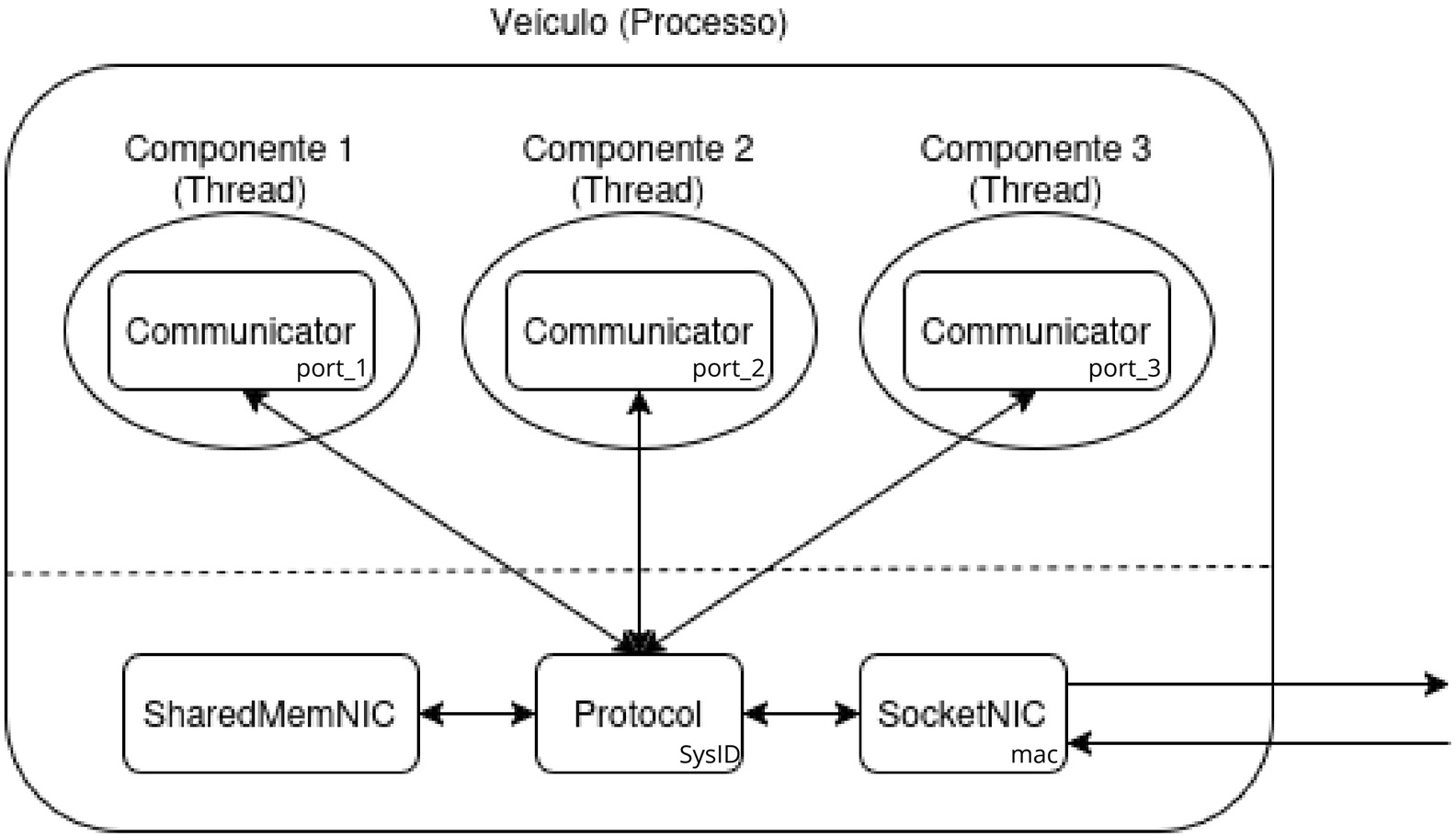
SharedEngine (Local - Intra-Sistema):

- Não usa Sockets. Opera puramente em memória.
- send enfileira, receive desenfileira.
- Send chama diretamente o handler (NIC::handle_signal) registrado via bind para imitar a chegada de mensagem.

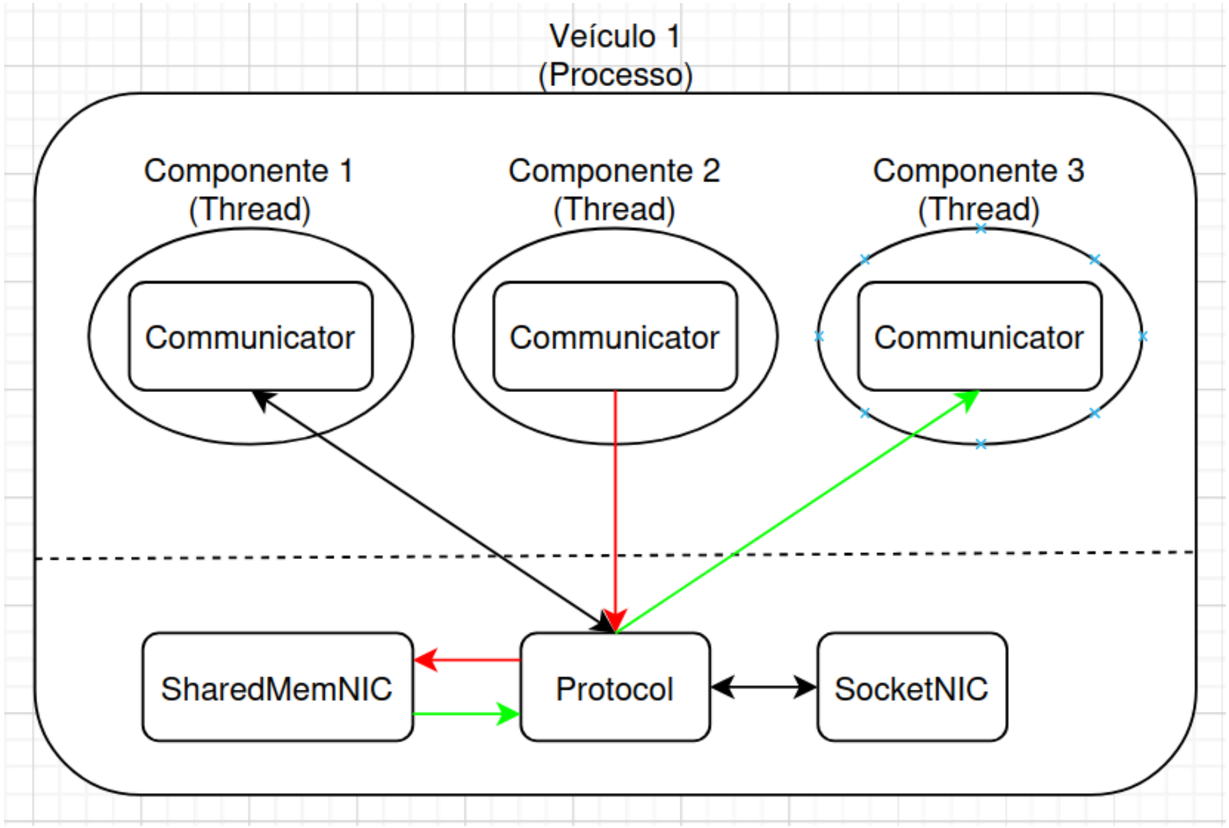
Funcionamento da SharedMemEngine

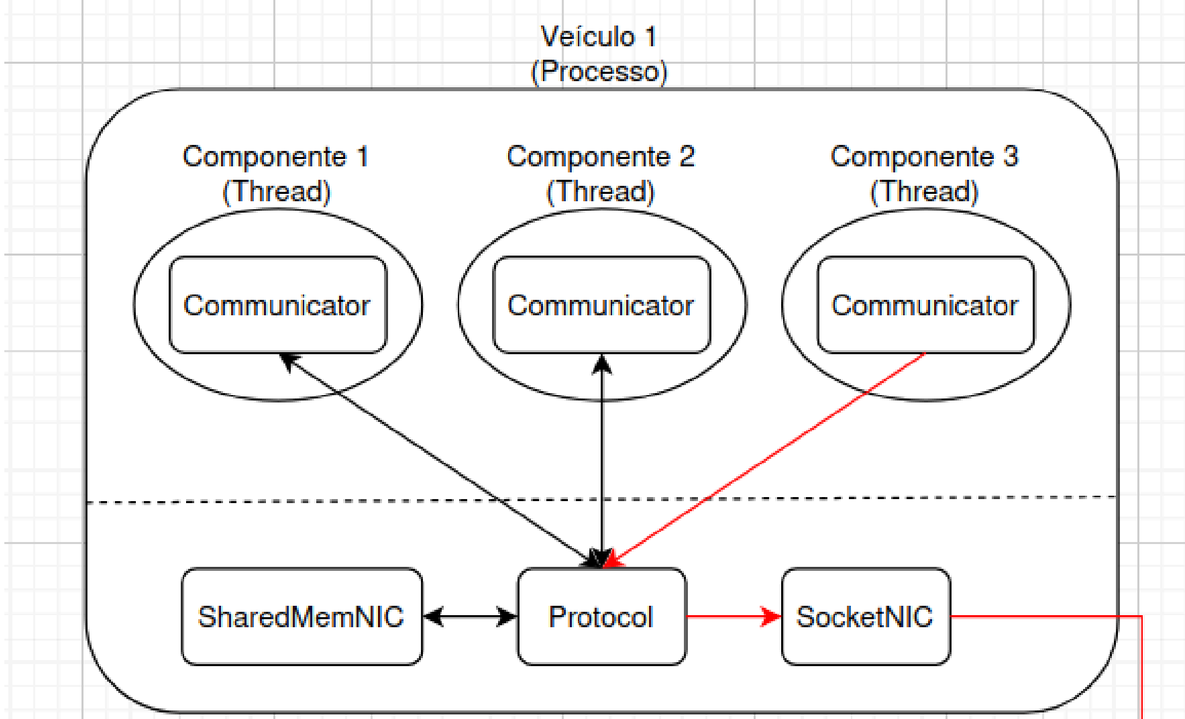


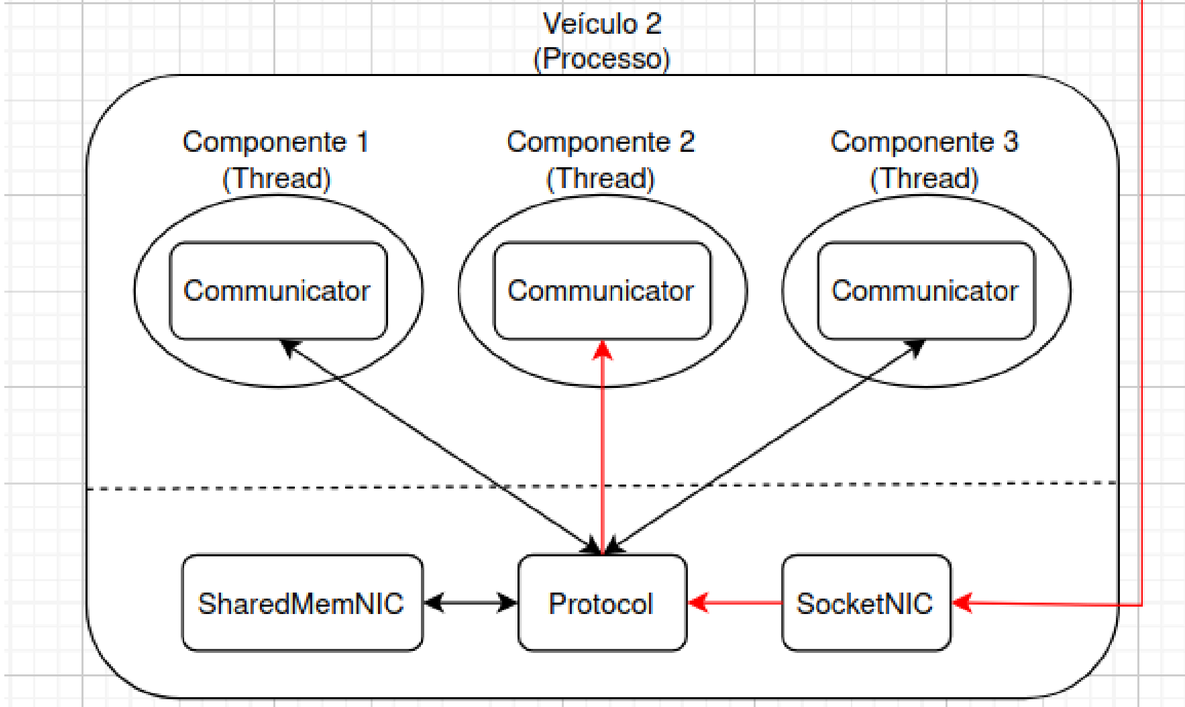
Representação do veículo



Exemplos de comunicação

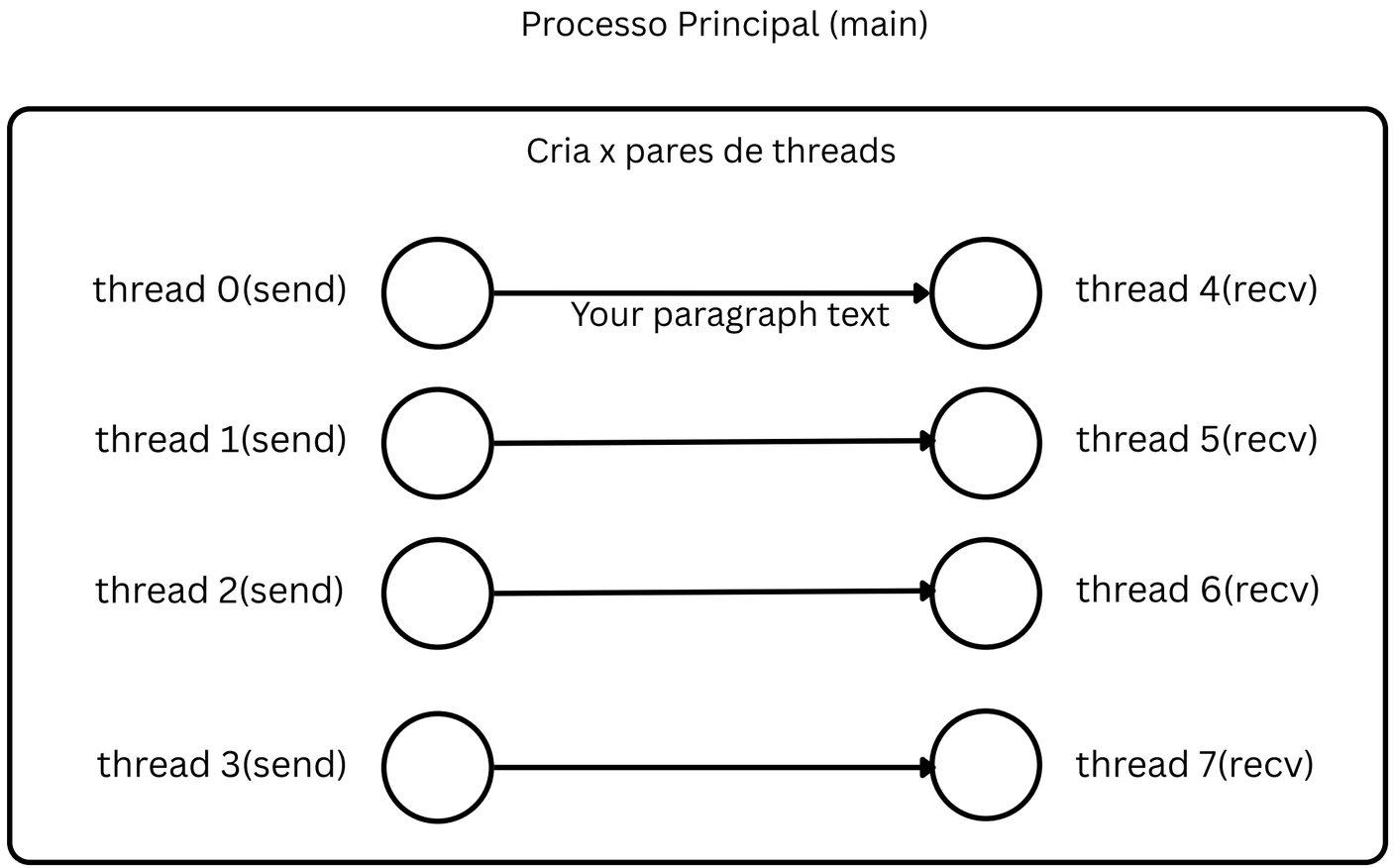






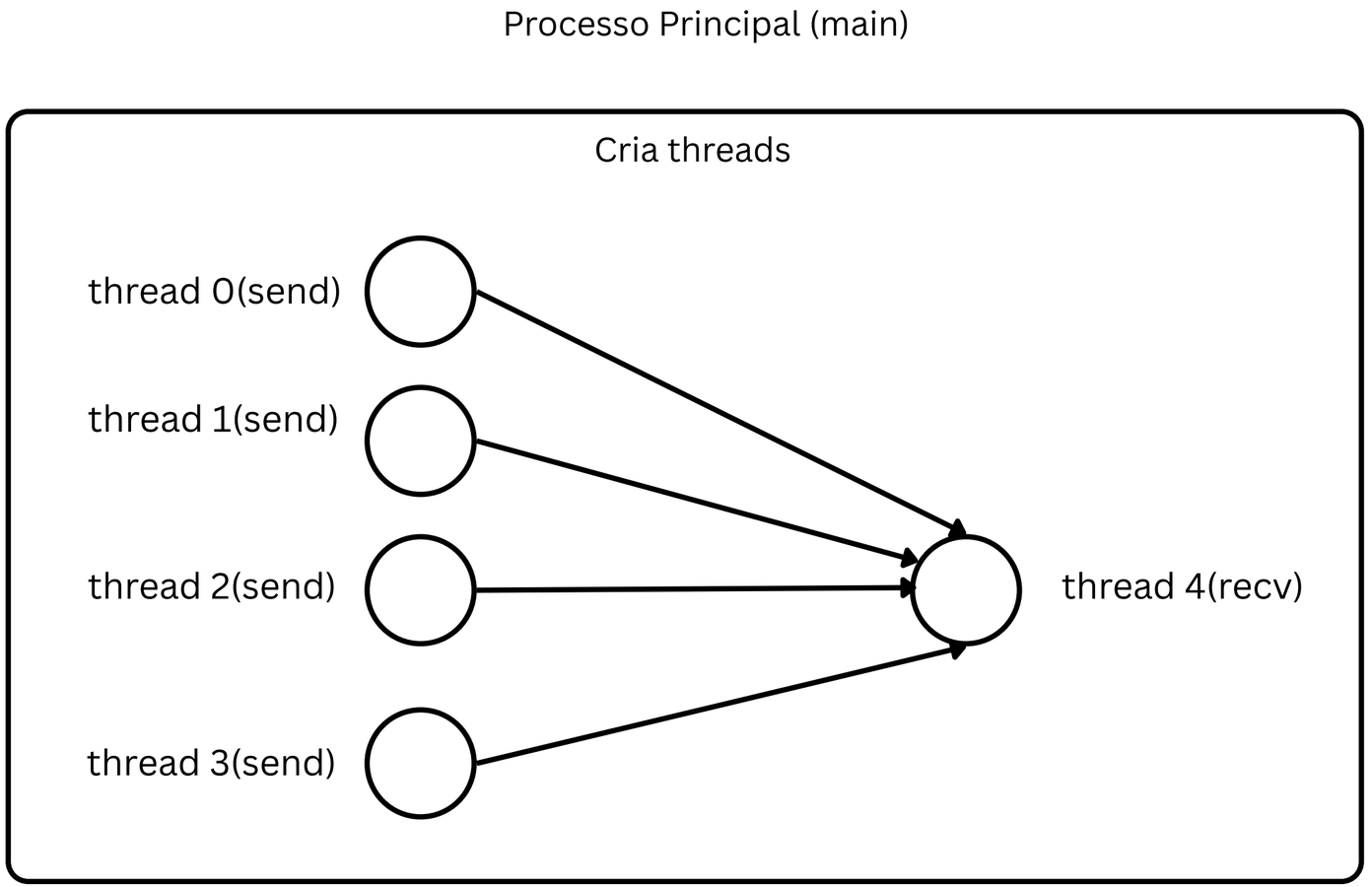
Testes Desenvolvidos

Muitas threads para muitas threads



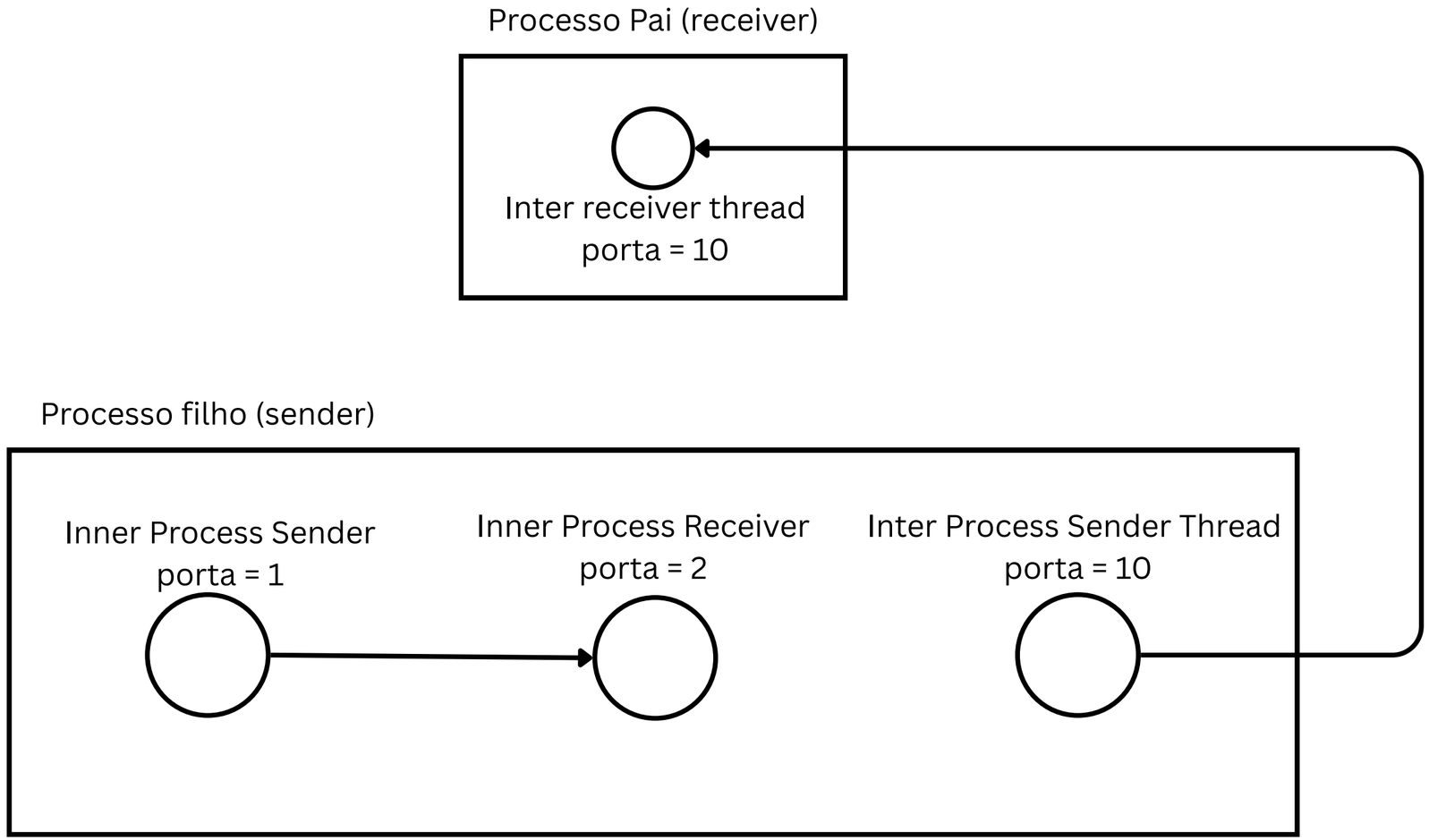
Testes Desenvolvidos

Muitas threads para uma thread



Testes Desenvolvidos

Uma thread para uma thread



Testes Desenvolvidos

Teste de broadcast

```
Thread (13): Received (0): d8 4e 3b bf a8
Thread (14): Received (0): d8 4e 3b bf a8
Thread (16): Received (0): d8 4e 3b bf a8
Thread (18): Received (0): d8 4e 3b bf a8
Thread (15): Received (0): d8 4e 3b bf a8
Thread (19): Received (0): d8 4e 3b bf a8
Thread (0): Sending (0): d8 4e 3b bf a8
Thread (17): Received (0): d8 4e 3b bf a8
Thread (11): Received (0): d8 4e 3b bf a8
Broadcast test finished!
```

Teste de broadcast vizinho

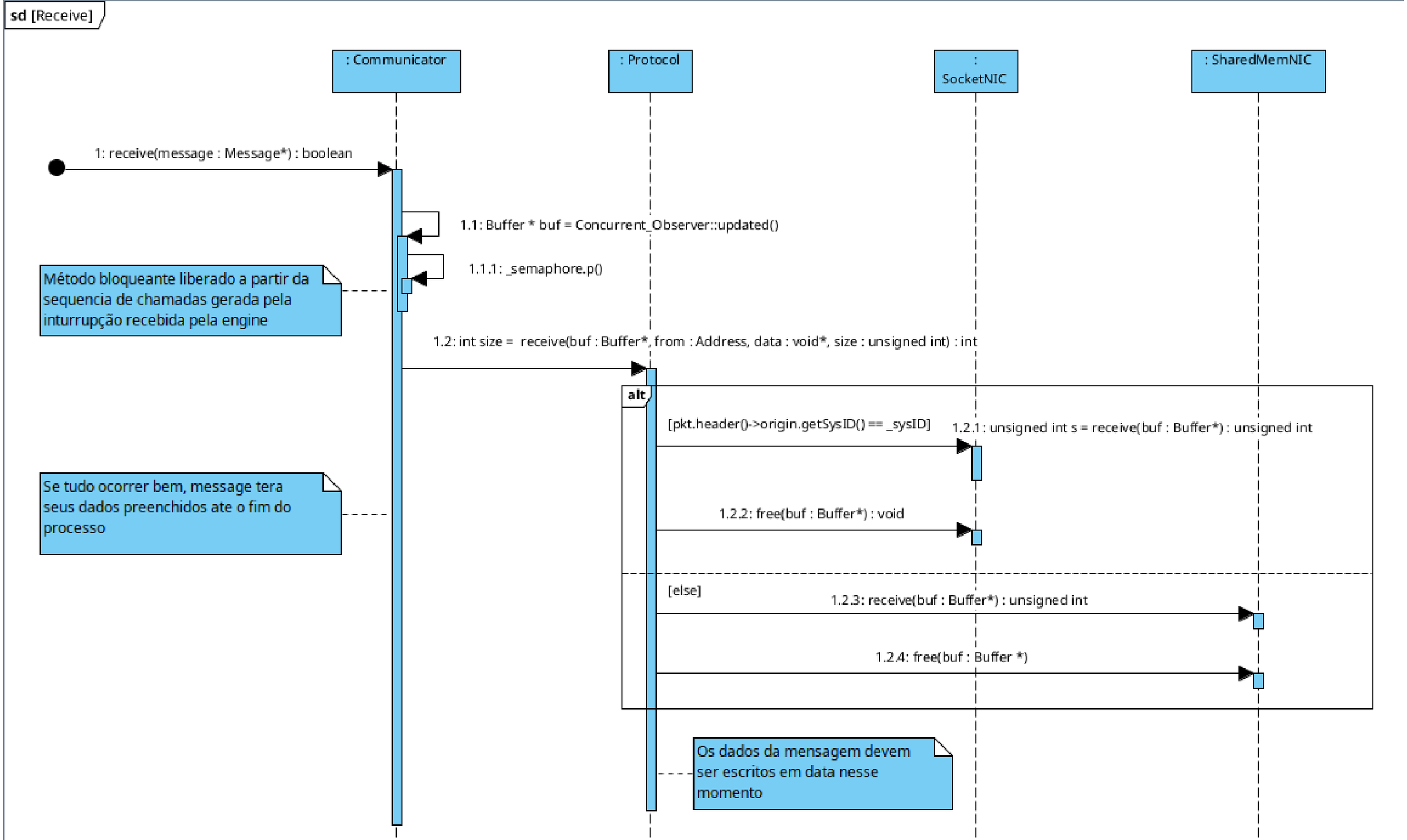
```
Processo (30565): Received (23): ab 18 1 a2 54
Processo (30564): Received (22): ab 18 1 a2 54
Processo (30565): Received (24): 17 8 e5 32 ea
Processo (30564): Received (23): ab 18 1 a2 54
Processo (30564): Received (24): 17 8 e5 32 ea
Broadcast test finished!
```

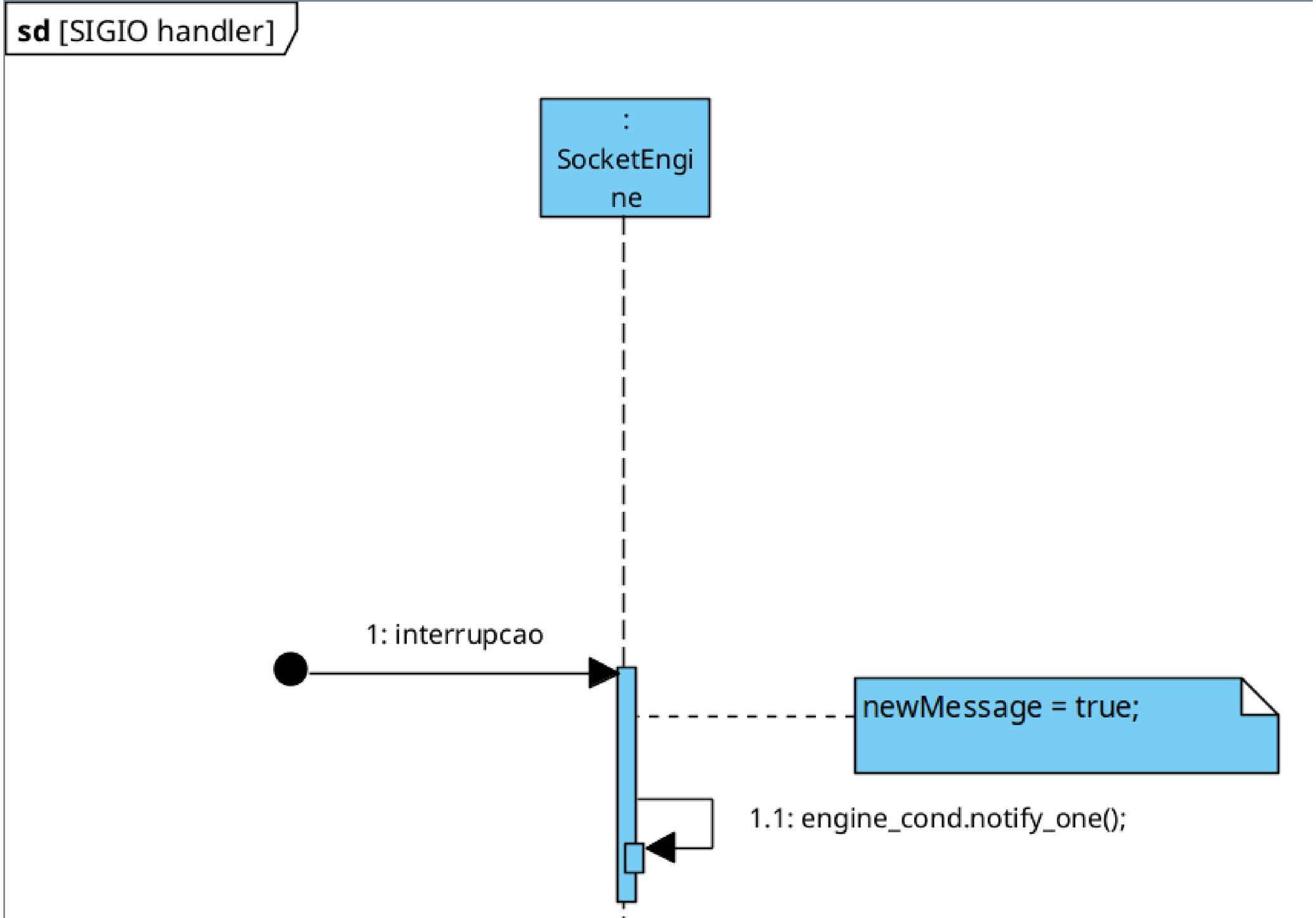
Teste de latência

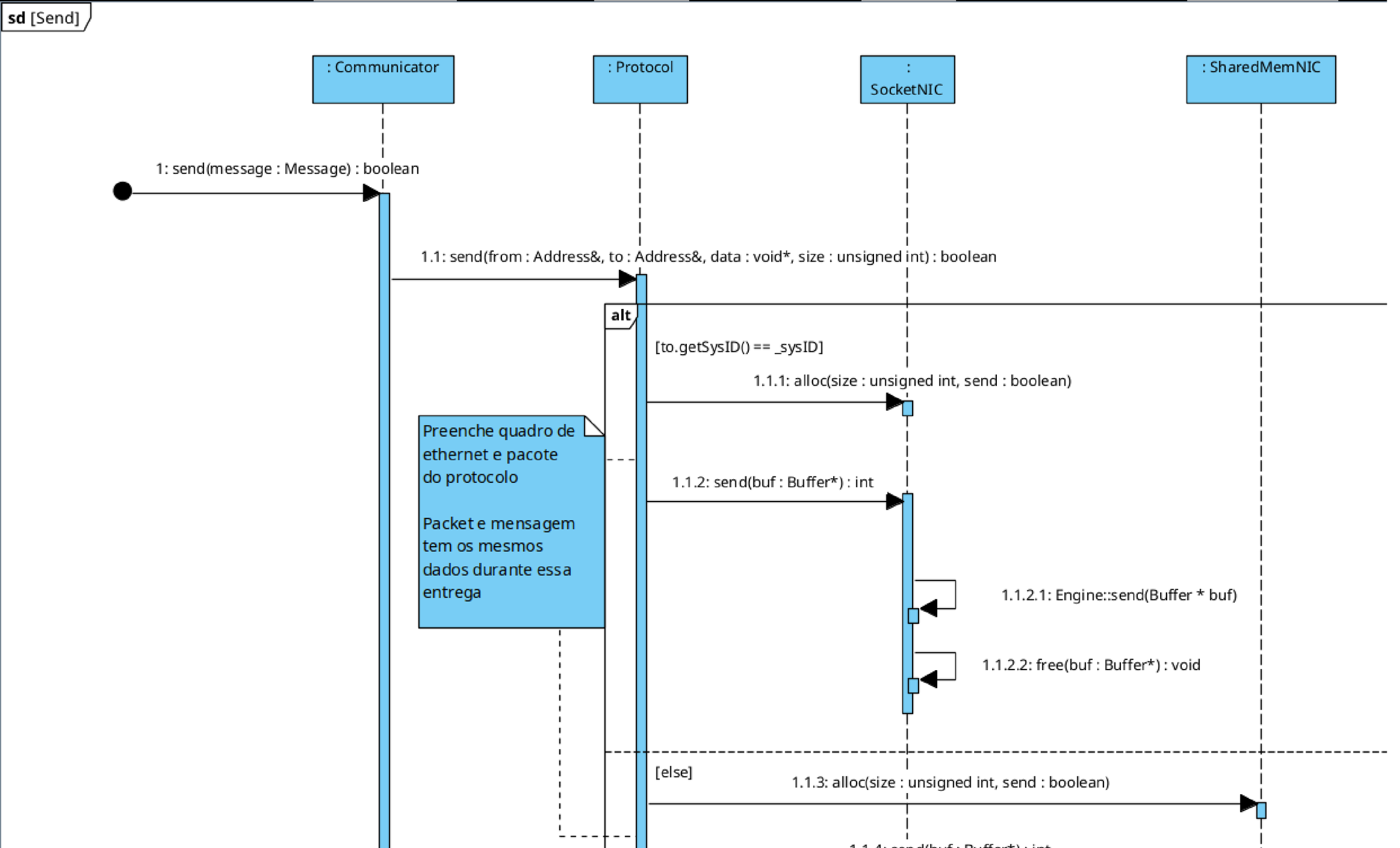
```
Comunicação entre threads de um mesmo processo:
Latência média observada: 12.173 µs

Comunicação entre threads de processos diferentes:
Latência média observada: 27.508 µs
```

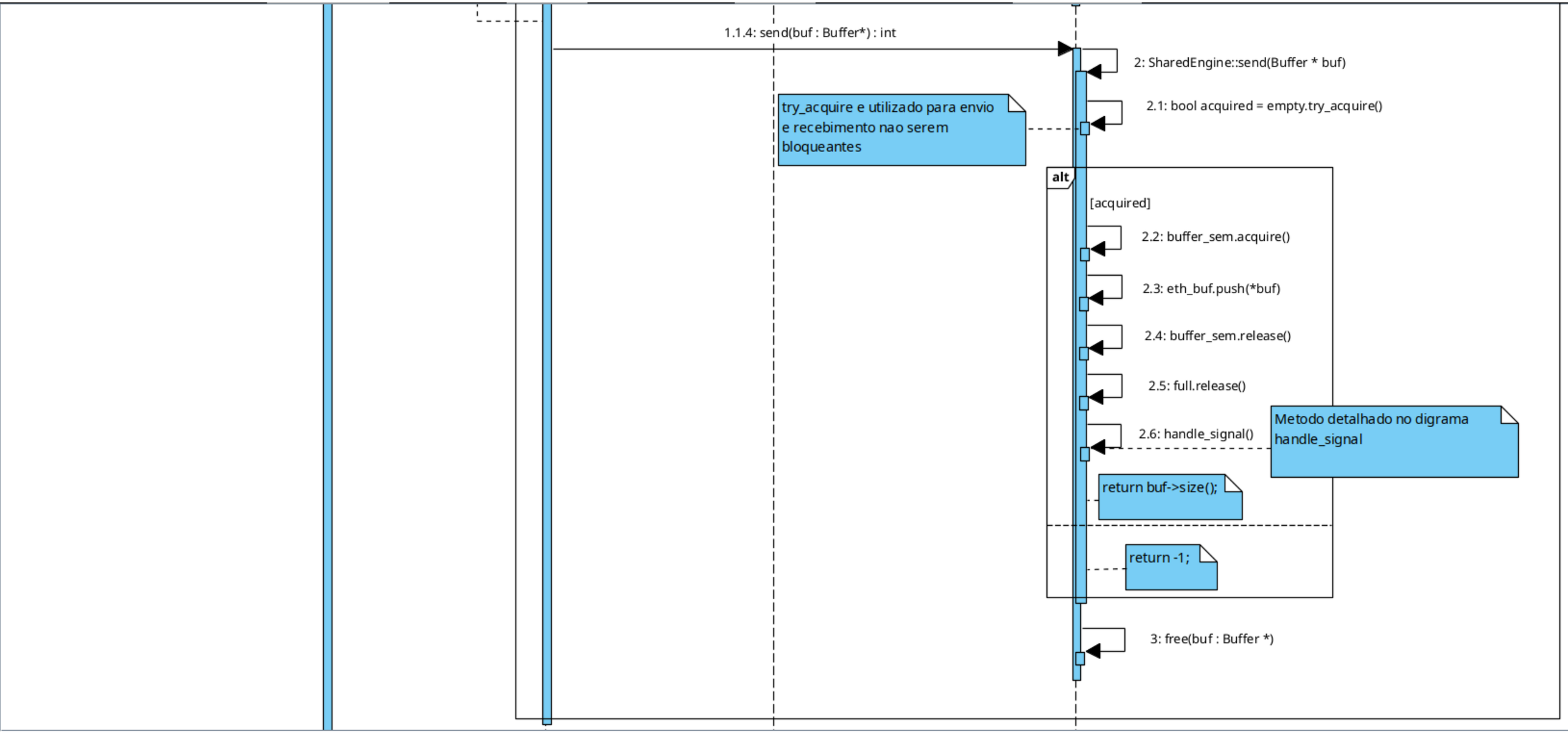
Teste de vazão

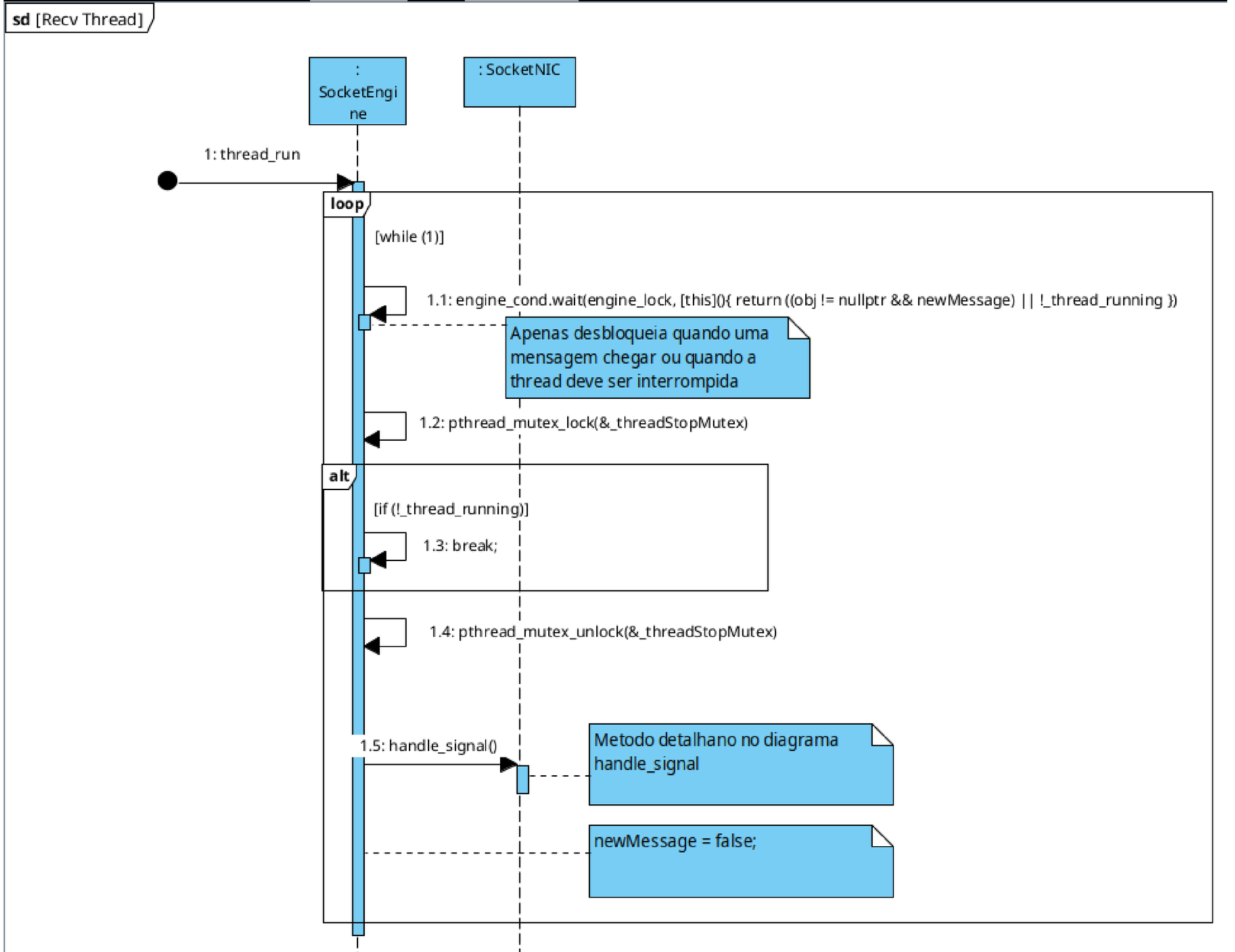
```
=====================================
INE5424 Testing: running e2_throughput_test
-------------------------------------
Enviando leva de 1000 mensagens:
Vazão média: 70736.37 mensagens/s
Vazão máxima: 80627.71 mensagens/s

Enviando leva de 5000 mensagens:
Vazão média: 83845.72 mensagens/s
Vazão máxima: 84012.27 mensagens/s

Enviando leva de 25000 mensagens:
Vazão média: 87773.69 mensagens/s
Vazão máxima: 87783.18 mensagens/s

Enviando leva de 125000 mensagens:
Timeout na recepção de mensagens.
```

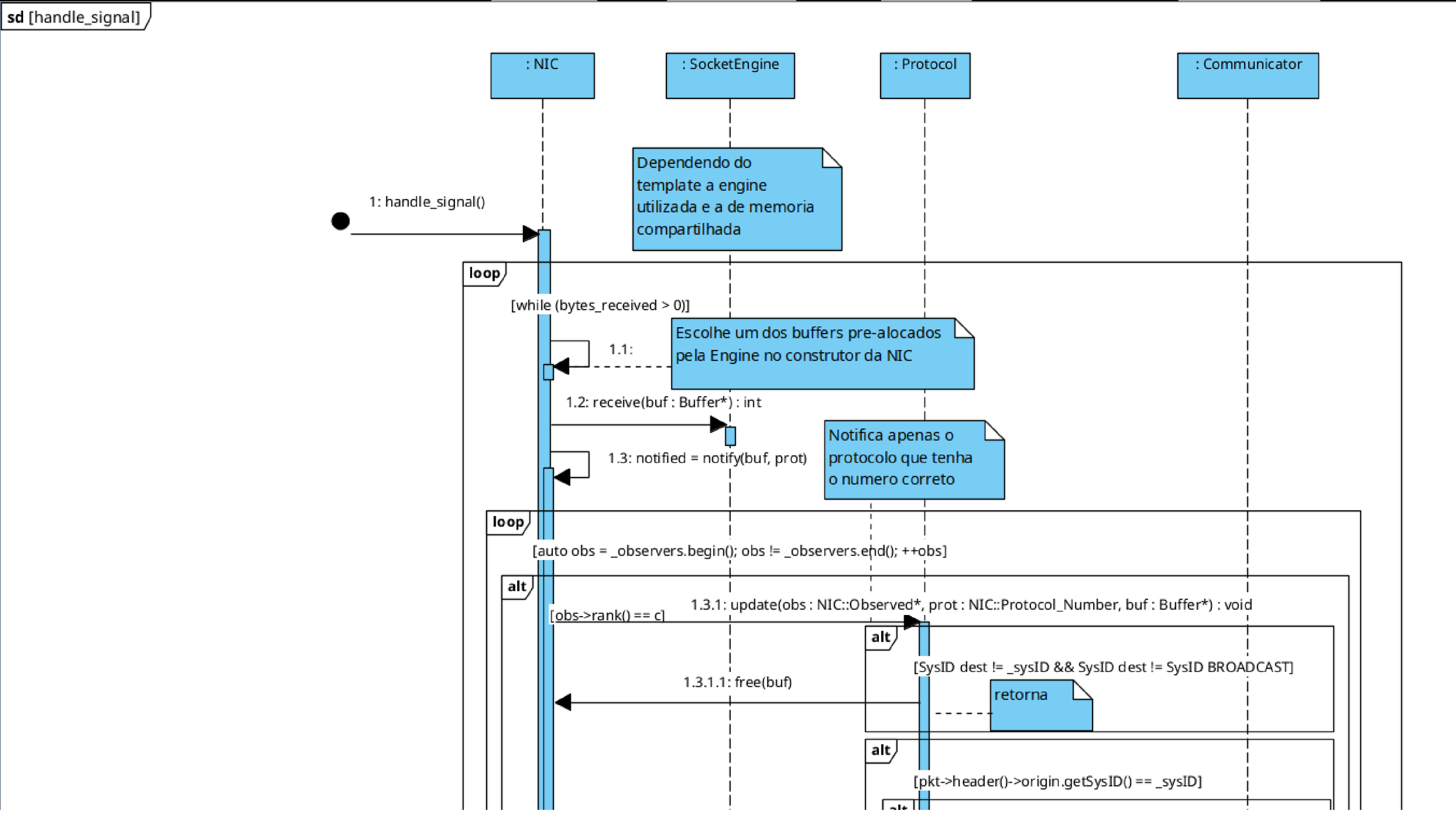

sd [Receive]
: Communicator
: Protocol
: SocketNIC
: SharedMemNIC
1: receive(message : Message*) : boolean
1.1: Buffer * buf = Concurrent_Observer::updated()
1.1.1: _semaphore.p()
Método bloqueante liberado a partir da sequencia de chamadas gerada pela inturrupção recebida pela engine
1.2: int size =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

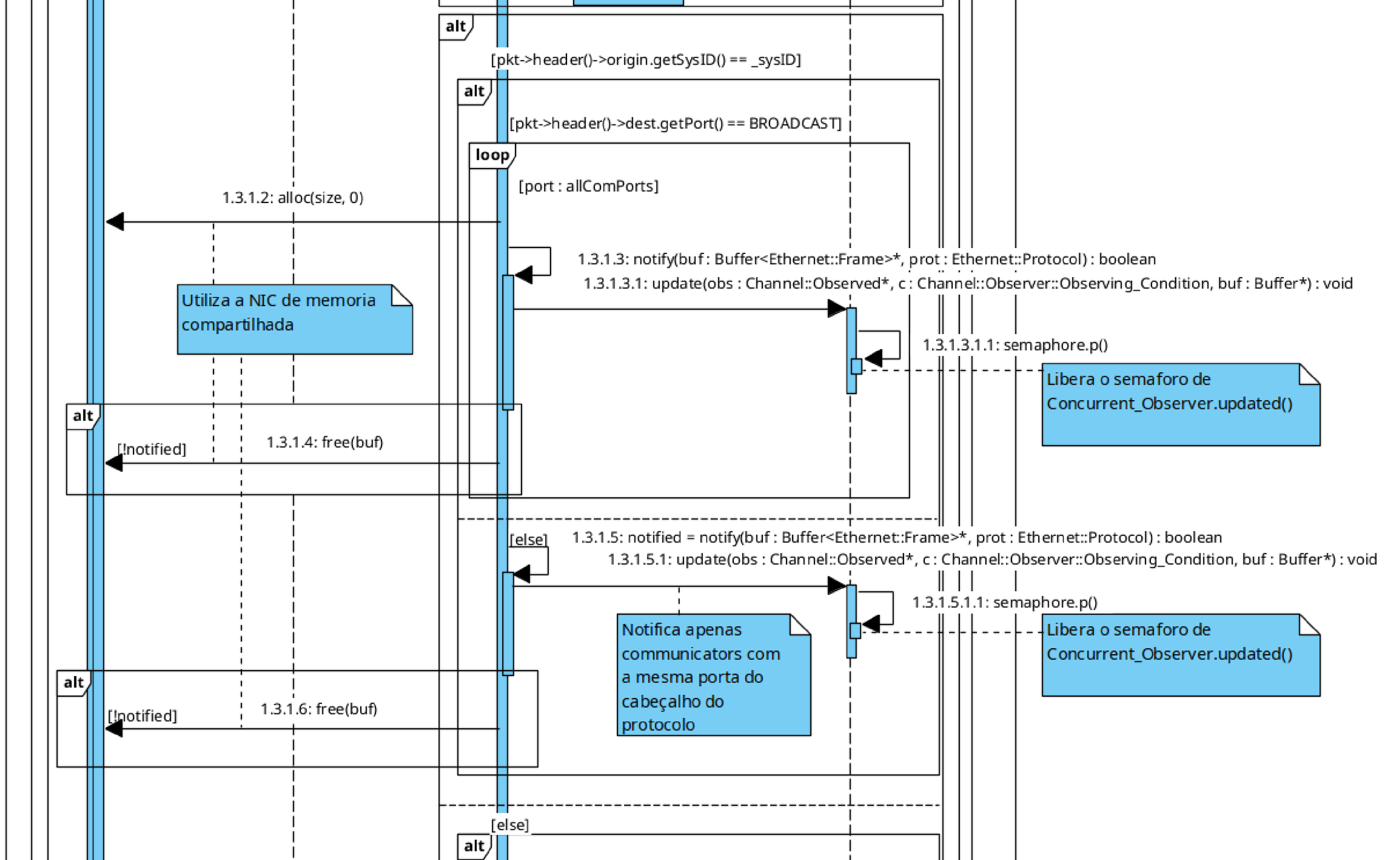

sd [SIGIO handler]
: SocketEngine
1: interrupcao
newMessage = true;
1.1: engine_cond.notify_one();

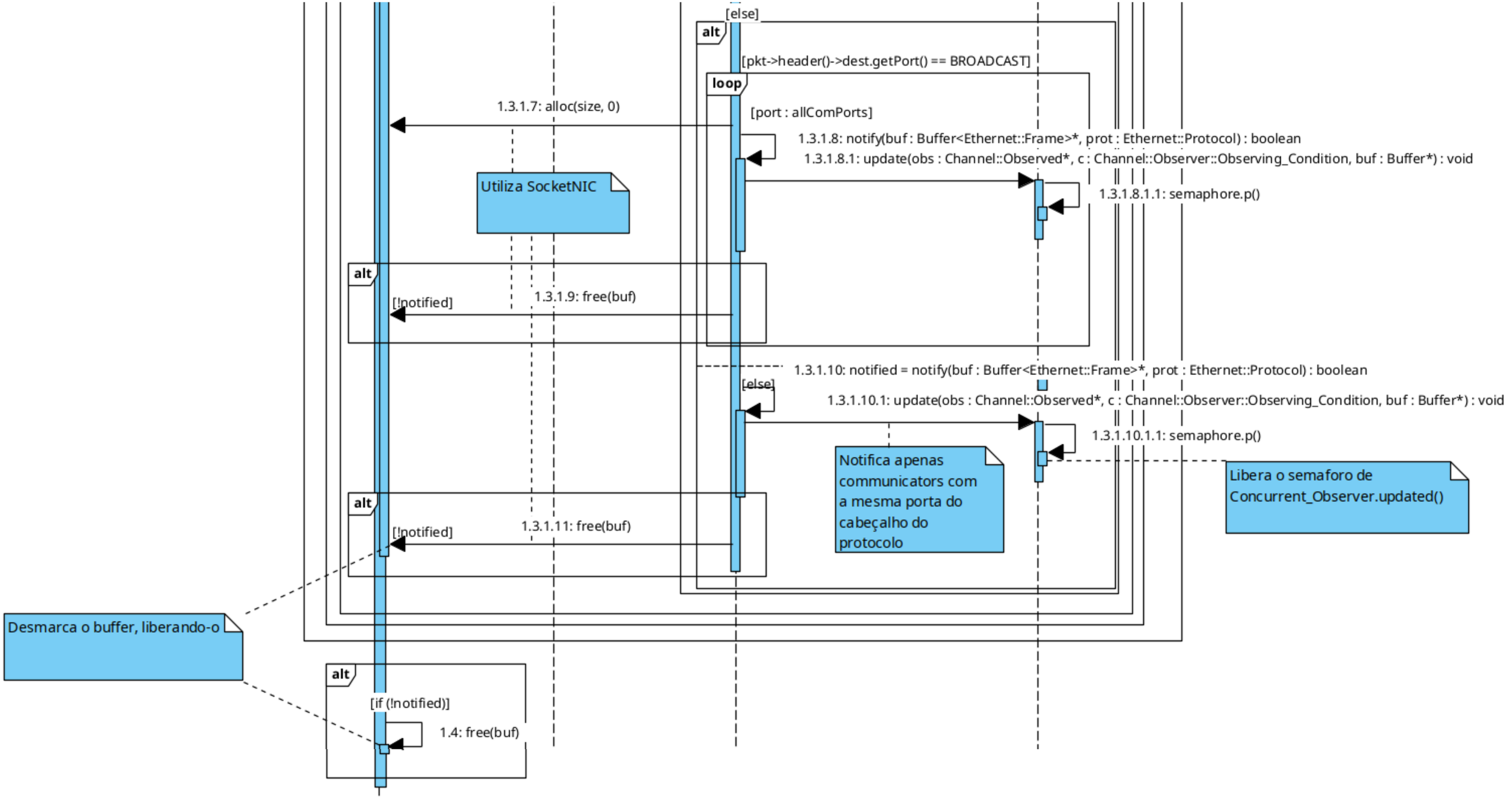

sd [Send]
: Communicator
: Protocol
: SocketNIC
: SharedMemNIC
1: send(message : Message) : boolean
1.1: send(from : Address&, to : Address&, data : void*, size : unsigned int) : boolean
alt
[to.getSysID() == _sysID]
1.1.1: alloc(size : unsigned int, send : boolean)
Preenche quadro de ethernet e pacote do protocolo
Packet e mensagem tem os mesmos dados durante essa entrega
1.1.2: send(buf : Buffer*) : int
1.1.2.1: Engine::send(Buffer * buf)
1.1.2.2: free(buf : Buffer*) : void
[else]
1.1.3: alloc(size : unsigned int, send : boolean)

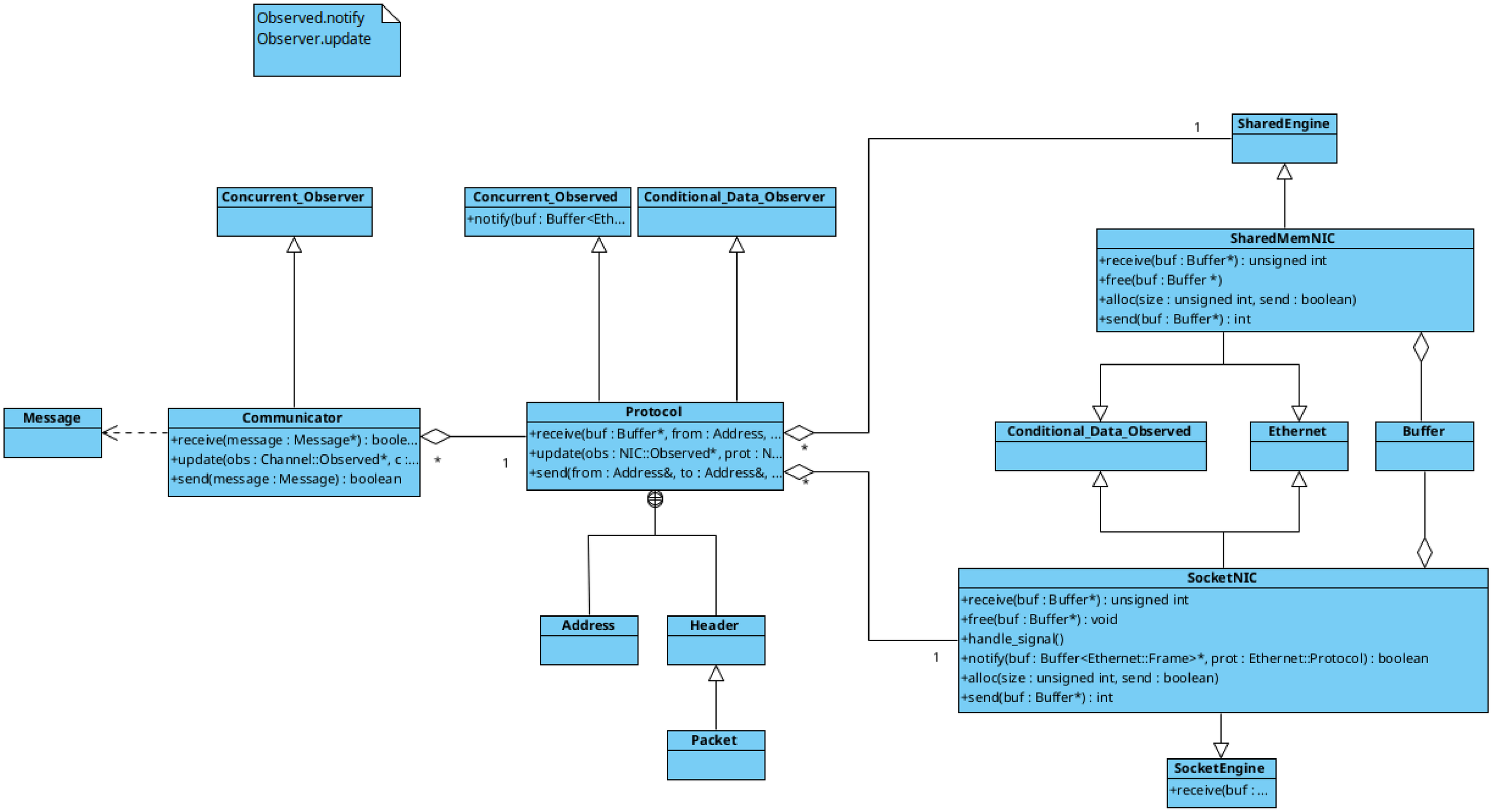

1.1.4: send(buf : Buffer*) : int
2: SharedEngine::send(Buffer * buf)
try_acquire e utilizado para envio e recebimento nao serem bloqueantes
2.1: bool acquired = empty.try_acquire()
alt
[acquired]
2.2: buffer_sem.acquire()
2.3: eth_buf.push(*buf)
2.4: buffer_sem.release()
2.5: full.release()
2.6: handle_signal()
Metodo detalhado no digrama handle_signal
return buf->size();
return -1;
3: free(buf : Buffer *)


sd [Recv Thread]
: SocketEngine
: SocketNIC
1: thread_run
loop
[while (1)]
1.1: engine_cond.wait(engine_lock, [this](){ return ((obj != nullptr && newMessage) || !_thread_running })
Apenas desbloqueia quando uma mensagem 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)
1.5: handle_signal()
Metodo detalhano no diagrama handle_signal
newMessage = false;


sd [handle_signal]
: NIC
: SocketEngine
: Protocol
: Communicator
Dependendo do template a engine utilizada e a de memoria compartilhada
1: handle_signal()
loop
[while (bytes_received > 0)]
1.1:
Escolhe um dos buffers pre-alocados pela Engine no construtor da NIC
1.2: receive(buf : Buffer*) : int
1.3: notified = notify(buf, prot)
Notifica apenas o protocolo que tenha o numero correto
loop
[auto obs = _observers.begin(); obs != _observers.end(); ++obs]
alt
[obs->rank() == c]
1.3.1: update(obs : NIC::Observed*, prot : NIC::Protocol_Number, buf : Buffer*) : void
alt
[SysID dest != _sysID && SysID dest != SysID BROADCAST]
1.3.1.1: free(buf)
retorna
alt
[pkt->header()->origin.getSysID() == _sysID]


alt
[pkt->header()->origin.getSysID() == _sysID]
alt
[pkt->header()->dest.getPort() == BROADCAST]
loop
[port : allComPorts]
1.3.1.2: alloc(size, 0)
1.3.1.3: notify(buf : Buffer<Ethernet::Frame>*, prot : Ethernet::Protocol) : boolean
1.3.1.3.1: update(obs : Channel::Observed*, c : Channel::Observer::Observing_Condition, buf : Buffer*) : void
Utiliza a NIC de memoria compartilhada
1.3.1.3.1.1: semaphore.p()
Libera o semaforo de Concurrent_Observer.updated()
alt
[!notified]
1.3.1.4: free(buf)
[else]
1.3.1.5: notified = notify(buf : Buffer<Ethernet::Frame>*, prot : Ethernet::Protocol) : boolean
1.3.1.5.1: update(obs : Channel::Observed*, c : Channel::Observer::Observing_Condition, buf : Buffer*) : void
1.3.1.5.1.1: semaphore.p()
Notifica apenas communicators com a mesma porta do cabeçalho do protocolo
Libera o semaforo de Concurrent_Observer.updated()
alt
[!notified]
1.3.1.6: free(buf)
[else]
alt


[else]
alt
[pkt->header()->dest.getPort() == BROADCAST]
loop
1.3.1.7: alloc(size, 0)
[port : allComPorts]
1.3.1.8: notify(buf : Buffer<Ethernet::Frame>*, prot : Ethernet::Protocol) : boolean
1.3.1.8.1: update(obs : Channel::Observed*, c : Channel::Observer::Observing_Condition, buf : Buffer*) : void
Utiliza SocketNIC
1.3.1.8.1.1: semaphore.p()
alt
[!notified]
1.3.1.9: free(buf)
1.3.1.10: notified = notify(buf : Buffer<Ethernet::Frame>*, prot : Ethernet::Protocol) : boolean
[else]
1.3.1.10.1: update(obs : Channel::Observed*, c : Channel::Observer::Observing_Condition, buf : Buffer*) : void
1.3.1.10.1.1: semaphore.p()
Notifica apenas communicators com a mesma porta do cabeçalho do protocolo
Libera o semaforo de Concurrent_Observer.updated()
alt
[!notified]
1.3.1.11: free(buf)
Desmarca o buffer, liberando-o
alt
[if (!notified)]
1.4: free(buf)


Observed.notify
Observer.update
Concurrent_Observer
Concurrent_Observed
+notify(buf : Buffer<Eth...
Conditional_Data_Observer
SharedEngine
SharedMemNIC
+receive(buf : Buffer*) : unsigned int
+free(buf : Buffer *)
+alloc(size : unsigned int, send : boolean)
+send(buf : Buffer*) : int
Message
Communicator
+receive(message : Message*) : boole...
+update(obs : Channel::Observed*, c :...
+send(message : Message) : boolean
Protocol
+receive(buf : Buffer*, from : Address, ...
+update(obs : NIC::Observed*, prot : N...
+send(from : Address&, to : Address&, ...
Conditional_Data_Observed
Ethernet
Buffer
Address
Header
Packet
SocketNIC
+receive(buf : Buffer*) : unsigned int
+free(buf : Buffer*) : void
+handle_signal()
+notify(buf : Buffer<Ethernet::Frame>*, prot : Ethernet::Protocol) : boolean
+alloc(size : unsigned int, send : boolean)
+send(buf : Buffer*) : int
SocketEngine
+receive(buf : ...
1
*
1
*
*
1