Departamento de
Informática e Estatística
CTC • UFSC

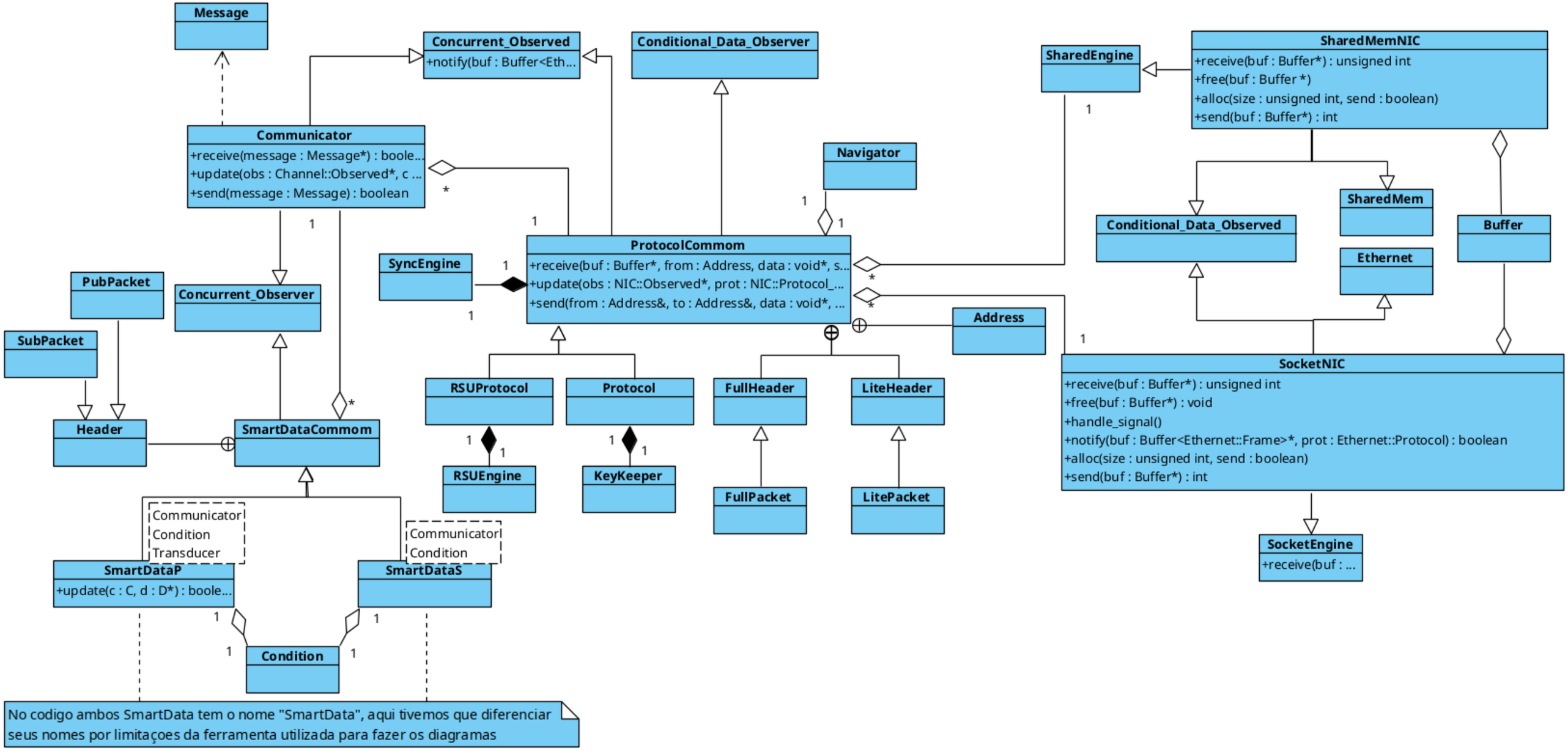
Comunicação local otimizada

INE5424 - Sistemas Operacionais II

Grupo A(Manhã): Vitor Calegari, Matheus Bigolin, Pedro Fountoura, Pedro Taglialenha

Objetivos da entrega 6

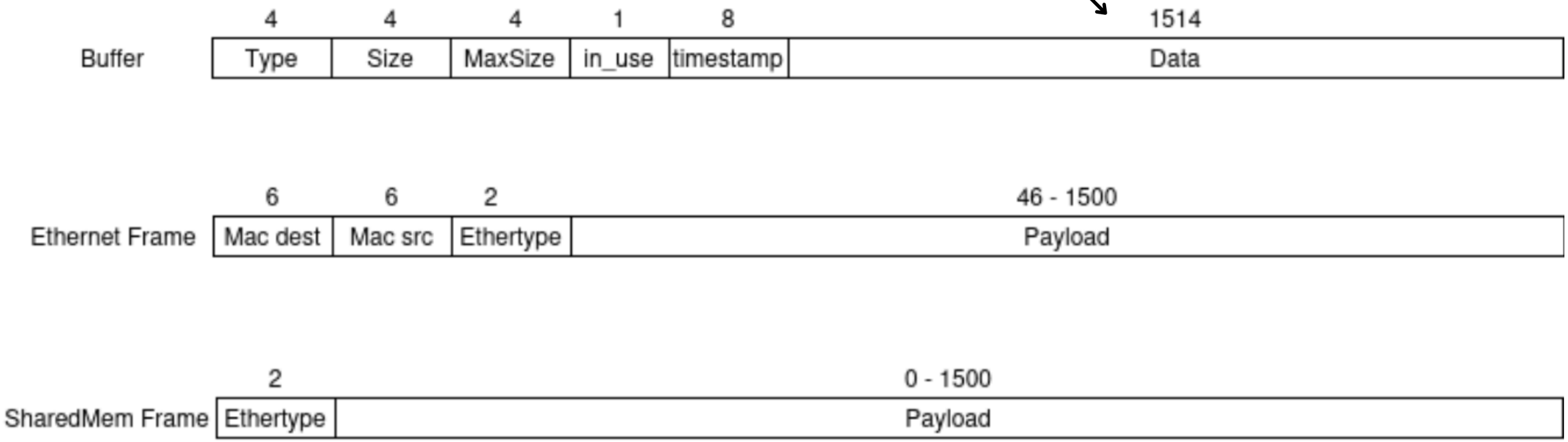
- Agentes internos (componentes/threads de um mesmo sistema autônomo/processo) não devem incorrer nos custos de transportar dados desnecessários (timestamp, mac, origem).
  - Otimizar mantendo a API transparente para o usuário.
  - Manter a semântica das mensagens trocadas entre sistemas, com menor custo.
- **Otimização de Dados:**
  - timestamp e MAC (Message Authentication Code) não são transmitidos em mensagens locais.
  - Se requisitados para uma mensagem local, devem ser fornecidos consistentemente (e.g., timestamp atual, origem, coordenadas).
- **Abordagem Implementada:**
  - Refatoração da SharedEngine e Buffer para não transportar Ethernet::Frame internamente.
  - Criação de formatos de pacote distintos para comunicação interna (LitePacket) e externa (FullPacket).


Message
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
Communicator
+receive(message : Message*) : boole...
+update(obs : Channel::Observed*, c ...
+send(message : Message) : boolean
Navigator
Conditional_Data_Observed
SharedMem
Buffer
Ethernet
PubPacket
Concurrent_Observer
SyncEngine
ProtocolCommom
+receive(buf : Buffer*, from : Address, data : void*, s...
+update(obs : NIC::Observed*, prot : NIC::Protocol_...
+send(from : Address&, to : Address&, data : void*, ...
Address
SubPacket
RSUProtocol
Protocol
FullHeader
LiteHeader
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
Header
SmartDataCommom
RSUEngine
KeyKeeper
FullPacket
LitePacket
SocketEngine
+receive(buf : ...
Communicator
Condition
Transducer
Communicator
Condition
SmartDataP
+update(c : C, d : D*) : boole...
SmartDataS
Condition
No codigo ambos SmartData tem o nome "SmartData", aqui tivemos que diferenciar seus nomes por limitaçoes da ferramenta utilizada para fazer os diagramas

Refatoração da Camada de Enlace e Buffer para Otimização

- SharedEngine e SharedMemNIC:
  - Estas classes, responsáveis pela comunicação intra-processo, agora operam utilizando SharedMem::Frame, definido em shared_mem.hh.
  - O SharedMem::Frame é mais simples, contendo apenas um SharedMem::Header (com o campo prot para o protocolo) e o payload de dados, eliminando os endereços MAC da camada Ethernet.
- Classe Buffer (buffer.hh):
  - Para acomodar diferentes tipos de frames, a classe Buffer agora inclui um membro BufferType _type, que pode ser EthernetFrame ou SharedMemFrame.
  - O tipo é definido no construtor Buffer(BufferType buf_type).
  - O método template <typename T> T* data() continua permitindo o acesso ao frame encapsulado.
- NIC<Engine> (nic.hh):
  - O template Engine agora define Engine::FrameClass, que pode ser Ethernet ou SharedMem, o que determina o tipo dos buffers da NIC.

Buffer.data armazena Ethernet::Frame ou SharedMem::Frame



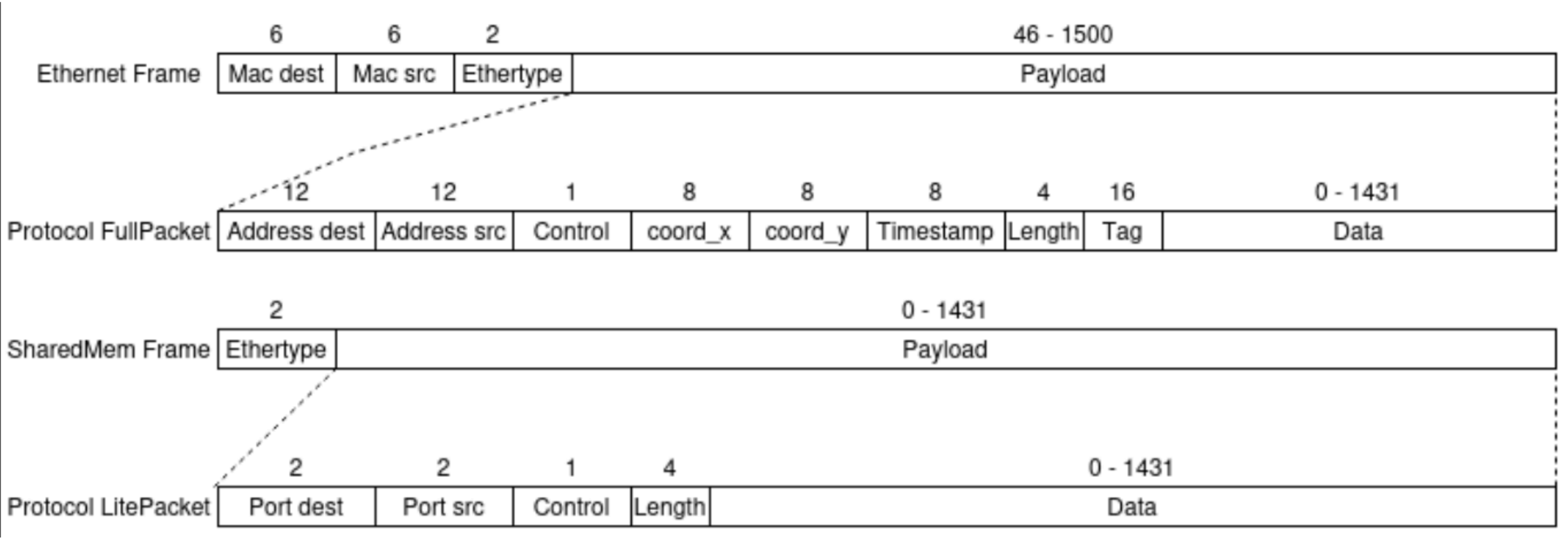
Novos Formatos de Pacote: LitePacket e FullPacket em ProtocolCommom

Distinguindo Pacotes para Eficiência Interna

- **ProtocolCommom::LiteHeader e ProtocolCommom::LitePacket:**
  - Função: São formatos otimizados, projetados especificamente para as mensagens que circulam internamente através da SharedMemNIC.
    - Conteúdo do LiteHeader:
      - Port origin;
      - Port dest;
      - Control ctrl;
      - std::size_t payloadSize;
  - Campos Omitidos para Eficiência: Endereços MAC (Physical_Address), SysID, coordenadas (_coord_x, _coord_y), o _timestamp de envio do protocolo e a MAC::Tag. Estes são desnecessários ou implícitos na comunicação interna.

Novos Formatos de Pacote: LitePacket e FullPacket em ProtocolCommom

- **ProtocolCommom::FullHeader e ProtocolCommom::FullPacket:**
  - Função: Mantêm o formato completo, utilizado para mensagens trocadas pela rede via SocketNIC.
  - Conteúdo do FullHeader (Consistente com a Entrega 5):
    - Address origin; e Address dest; (completos, com MAC, SysID e Porta).
    - Control ctrl;
    - std::size_t payloadSize;
    - double coord_x; e double coord_y;
    - uint64_t timestamp;
    - MAC::Tag tag;
- **Seleção Dinâmica em ProtocolCommom::send():**
  - O método send agora decide qual formato utilizar:
    - Se o SysID de destino (to.getSysID()) for igual ao _sysID local, a mensagem é interna. Então, chama sendSharedMem(), que utiliza fillLitePacket().
    - Caso contrário, a mensagem é externa. Chama sendSocket(), que utiliza fillFullPacket().
- **Tratamento na Recepção (Protocol::update / RSUProtocol::update):**
  - A lógica de update verifica o buf->type():
    - Se for Buffer::EthernetFrame, o conteúdo é tratado como um FullPacket.
    - Se for Buffer::SharedMemFrame, é tratado como um LitePacket.


6
6
2
46 - 1500
Ethernet Frame
Mac dest
Mac src
Ethertype
Payload
12
12
1
8
8
8
4
16
0 - 1431
Protocol FullPacket
Address dest
Address src
Control
coord_x
coord_y
Timestamp
Length
Tag
Data
2
0 - 1431
SharedMem Frame
Ethertype
Payload
2
2
1
4
0 - 1431
Protocol LitePacket
Port dest
Port src
Control
Length
Data

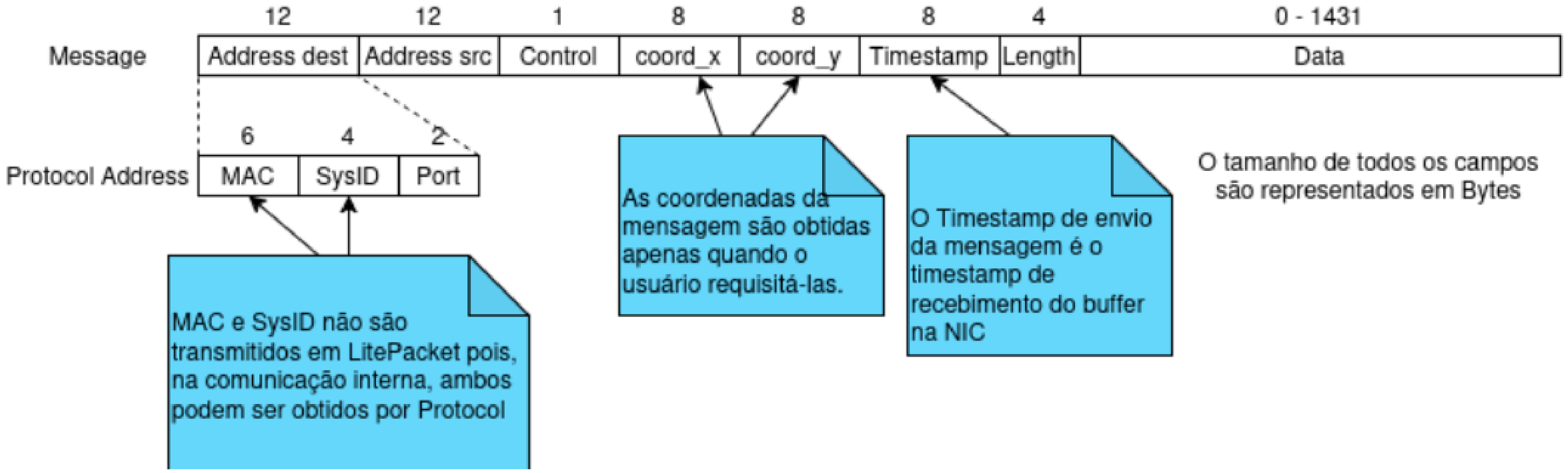
Detalhes da Otimização para Mensagens Locais com LitePacket

Reduzindo o Custo da Comunicação Dentro do Mesmo Sistema

- **Endereçamento Simplificado no LitePacket::Header:**
  - Para mensagens internas, usamos apenas Port origin e Port dest.
  - Campos como MAC físico e SysID completo são omitidos, pois a comunicação ocorre no mesmo processo, onde o _sysID e MAC são conhecidos.
- **Gerenciamento de _timestamp para Mensagens Locais:**
  - O LiteHeader não inclui o _timestamp de envio que o FullHeader possui.
  - Quando a aplicação acessa o timestamp de uma mensagem local (via Message::timestamp()), o valor fornecido é o de recebimento do buffer.
    - Isso significa que, para estas mensagens, o timestamp de envio é obtido no momento em que o buffer foi recebido pela SharedMemNIC, e não é um timestamp específico da camada Protocol no momento do envio interno.

Detalhes da Otimização para Mensagens Locais com LitePacket

- **Coordenadas (_coord_x, _coord_y) Acessadas Sob Demanda:**
  - Estes campos também não estão presentes no LiteHeader.
  - Na classe Message, os métodos getCoordX() e getCoordY() verificam se as coordenadas já foram obtidas. Se não, eles invocam _prot->getLocation() para buscar as coordenadas atuais.
    - Assim, a aplicação sempre obtém a localização mais recente no momento do acesso à mensagem, o que é consistente para dados internos.
- **Ausência de MAC::Tag em Mensagens Locais:**
  - O LiteHeader omite a MAC::Tag.
  - Consequentemente, não há cálculo nem verificação de MAC para as mensagens LitePacket que trafegam internamente, eliminando esse custo computacional.

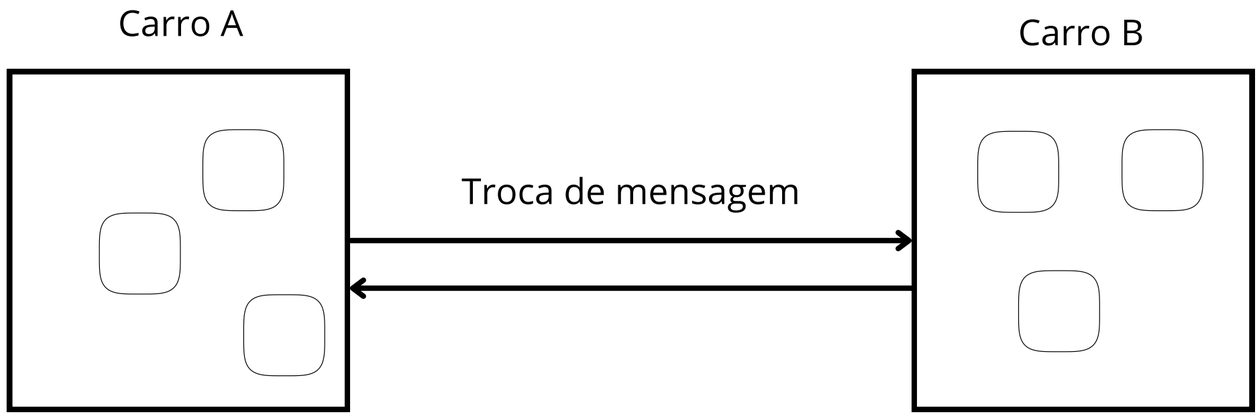

12
12
1
8
8
8
4
0 - 1431
Message
Address dest
Address src
Control
coord_x
coord_y
Timestamp
Length
Data
6
4
2
Protocol Address
MAC
SysID
Port
MAC e SysID não são transmitidos em LitePacket pois, na comunicação interna, ambos podem ser obtidos por Protocol
As coordenadas da mensagem são obtidas apenas quando o usuário requisitá-las.
O Timestamp de envio da mensagem é o timestamp de recebimento do buffer na NIC
O tamanho de todos os campos são representados em Bytes

Conclusões da Entrega 6

- **Comunicação Local Otimizada Implementada:**
  - Uso de SharedMem::Frame e LitePacket para mensagens intra-sistema, reduzindo o overhead de dados não essenciais (MAC físico, SysID, timestamp de envio do protocolo, MAC criptográfico, coordenadas fixas).
- **Manutenção da Semântica Externa:**
  - FullPacket continua sendo usado para comunicação inter-sistemas, preservando todos os campos necessários (incluindo MAC criptográfico, timestamp PTP, coordenadas).
- **Consistência de Dados Sob Demanda:**
  - _timestamp de envio para mensagens locais reflete o tempo de chegada na NIC interna.
  - Coordenadas (_coord_x, _coord_y) em Message são populadas no momento do acesso pela aplicação, garantindo valores atuais.
- **Flexibilidade da Camada Buffer:** Buffer agora suporta diferentes tipos de frames de enlace.
- **Impacto no Custo Computacional:** Redução do tamanho das mensagens internas e eliminação de cálculos de MAC para comunicação local.

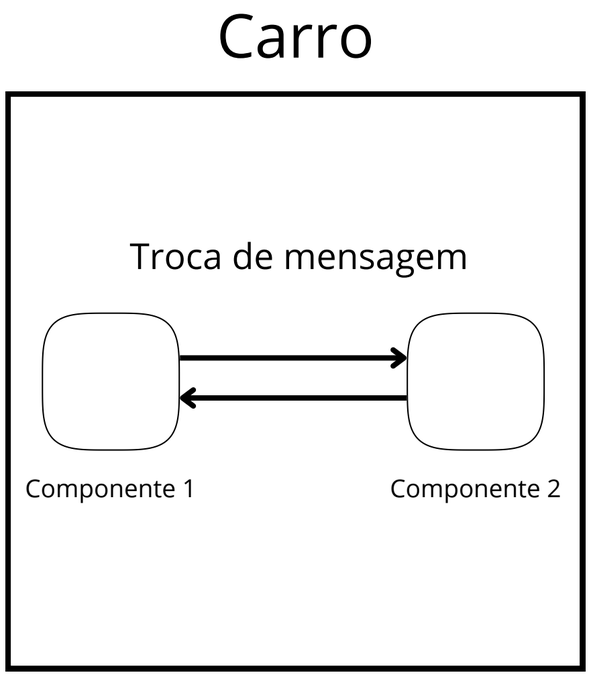
Testes Desenvolvidos e Validação

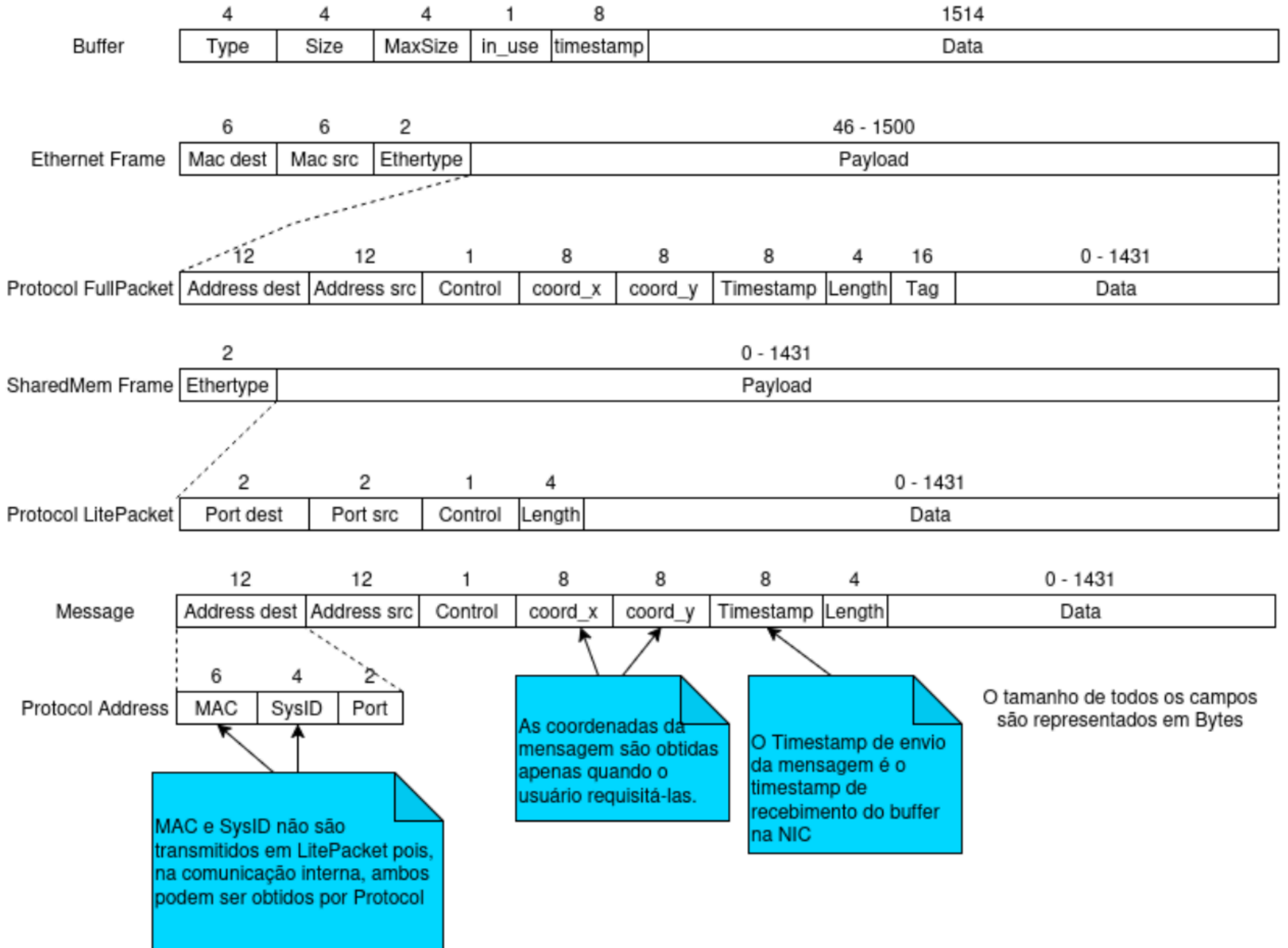
- **Teste de Comunicação via Socket**
  - O código testa a comunicação entre múltiplos veículos(processos) que trocam mensagens diretamente.
  - Getters dos campos da mensagem são utilizados para demonstrar API.



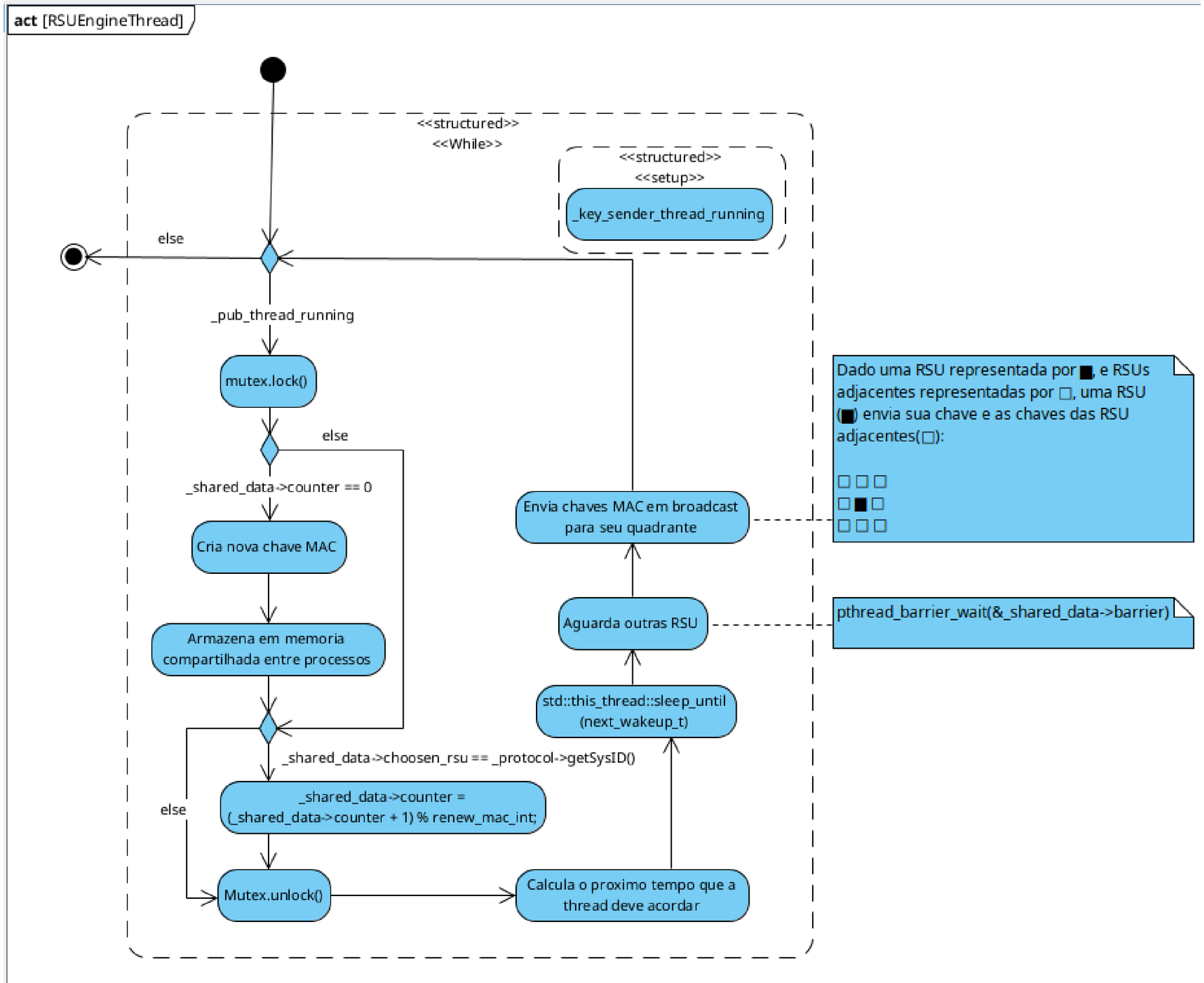
Testes Desenvolvidos e Validação

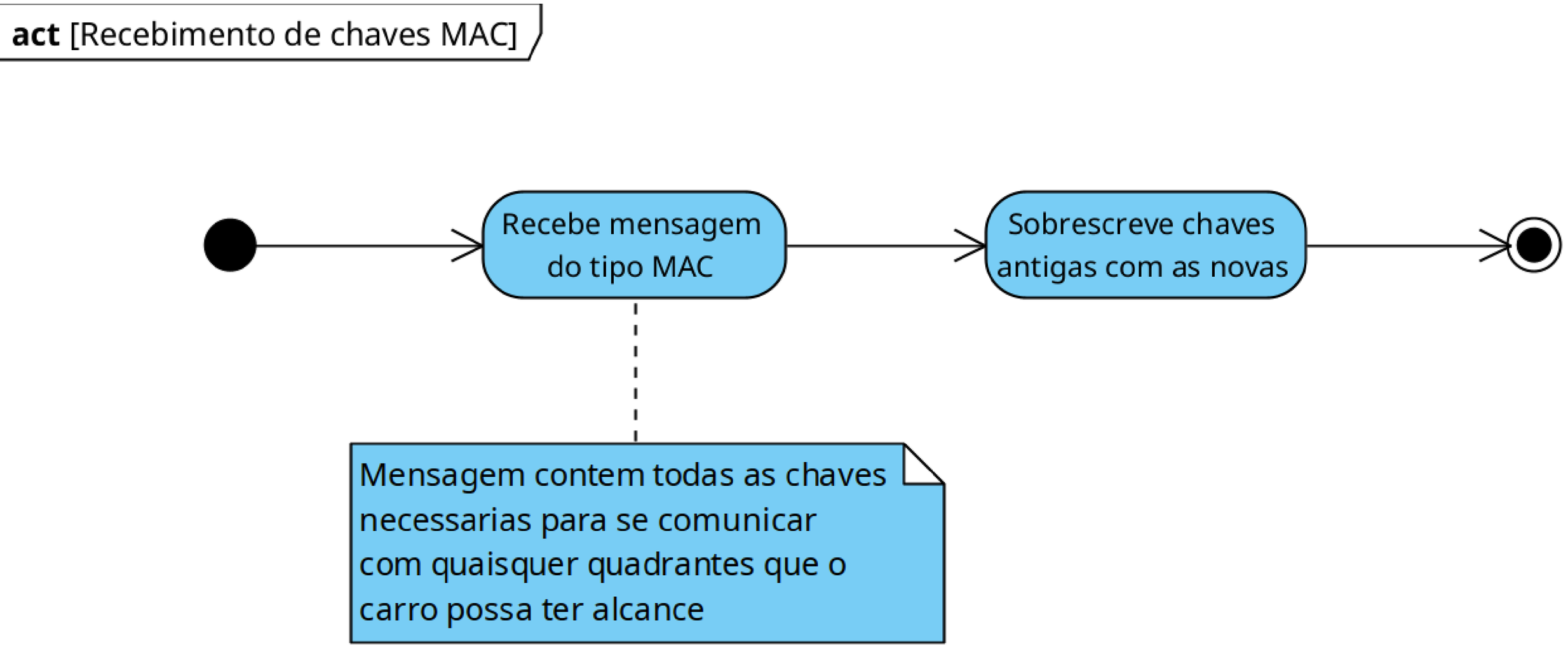
- **Teste de Comunicação via memória compartilhada**
  - O código testa a comunicação entre múltiplos componentes(threads) que trocam mensagens diretamente.
  - Getters dos campos da mensagem são utilizados para demonstrar API.

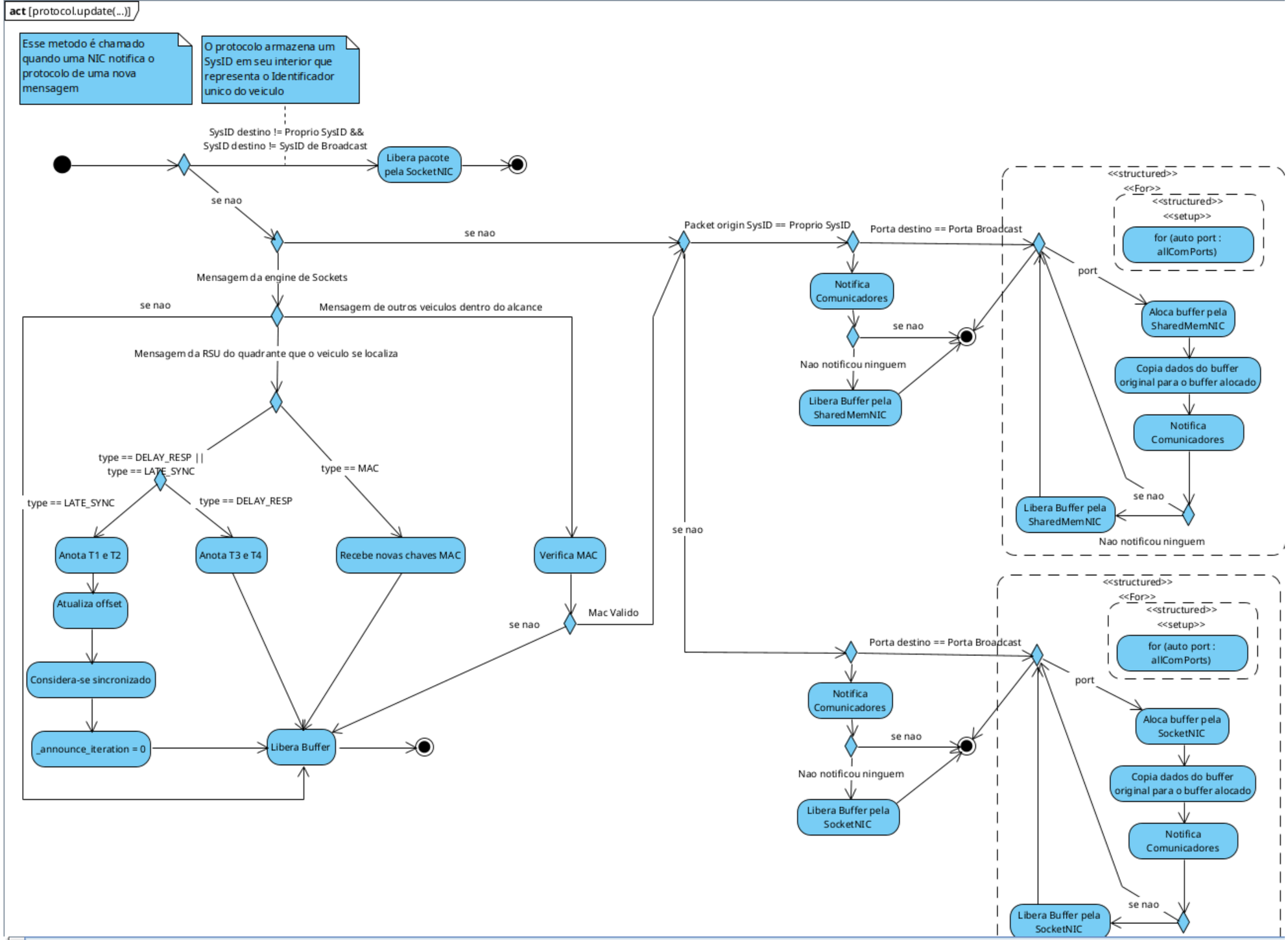


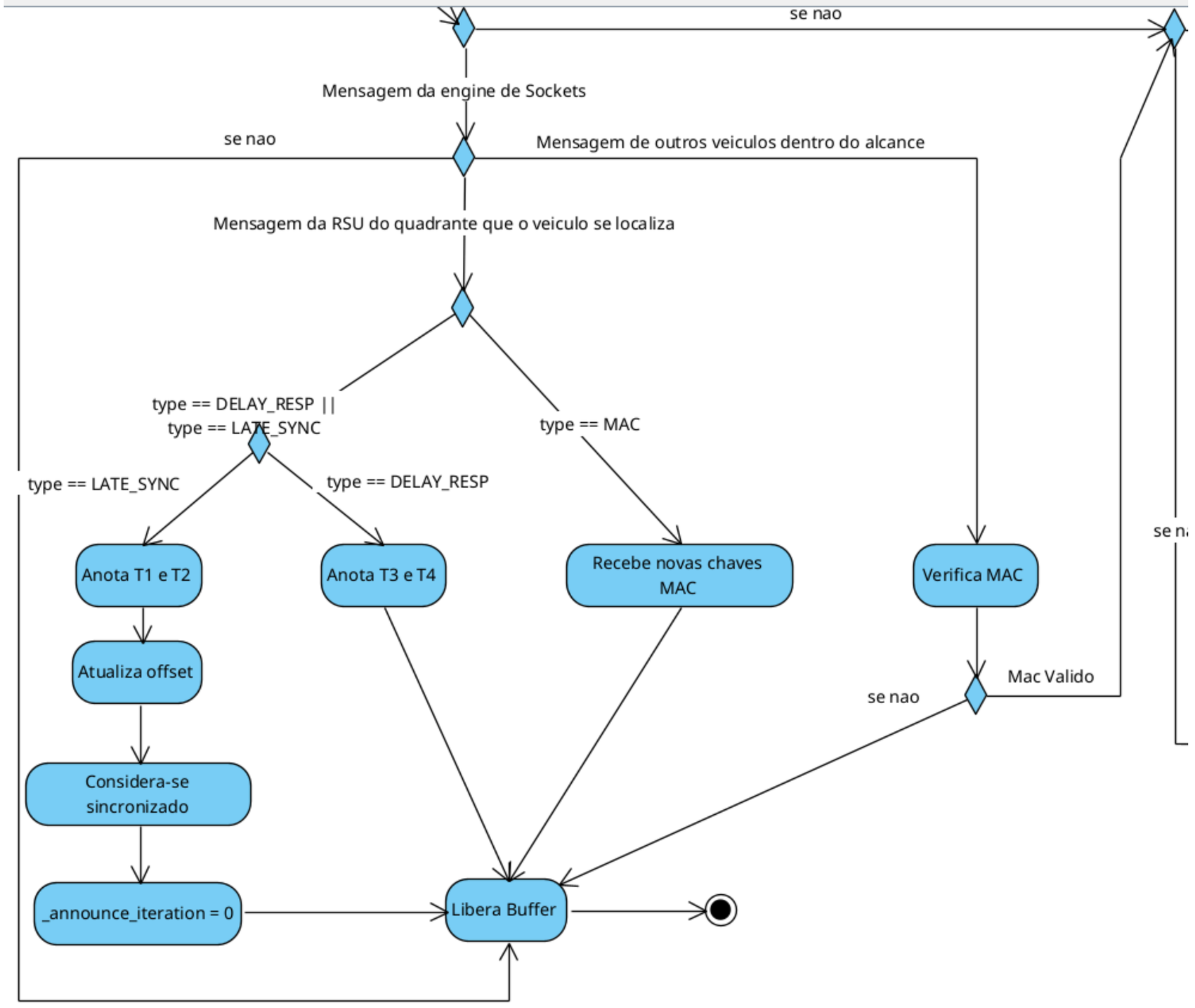

Buffer
4
4
4
1
8
1514
Type
Size
MaxSize
in_use
timestamp
Data
Ethernet Frame
6
6
2
46 - 1500
Mac dest
Mac src
Ethertype
Payload
Protocol FullPacket
12
12
1
8
8
8
4
16
0 - 1431
Address dest
Address src
Control
coord_x
coord_y
Timestamp
Length
Tag
Data
SharedMem Frame
2
0 - 1431
Ethertype
Payload
Protocol LitePacket
2
2
1
4
0 - 1431
Port dest
Port src
Control
Length
Data
Message
12
12
1
8
8
8
4
0 - 1431
Address dest
Address src
Control
coord_x
coord_y
Timestamp
Length
Data
Protocol Address
6
4
2
MAC
SysID
Port
MAC e SysID não são transmitidos em LitePacket pois, na comunicação interna, ambos podem ser obtidos por Protocol
As coordenadas da mensagem são obtidas apenas quando o usuário requisitá-las.
O Timestamp de envio da mensagem é o timestamp de recebimento do buffer na NIC
O tamanho de todos os campos são representados em Bytes

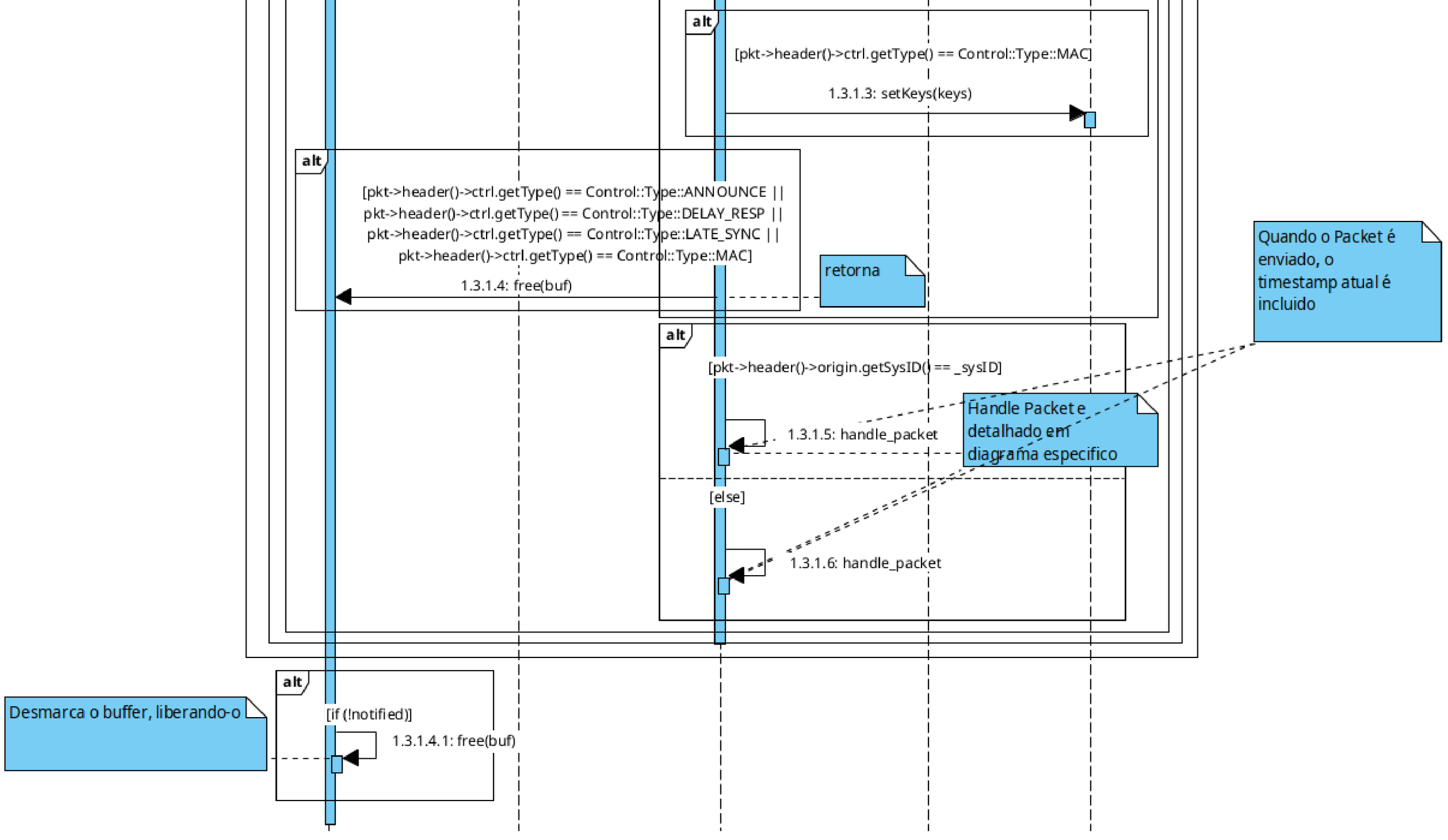
Outros Diagramas:

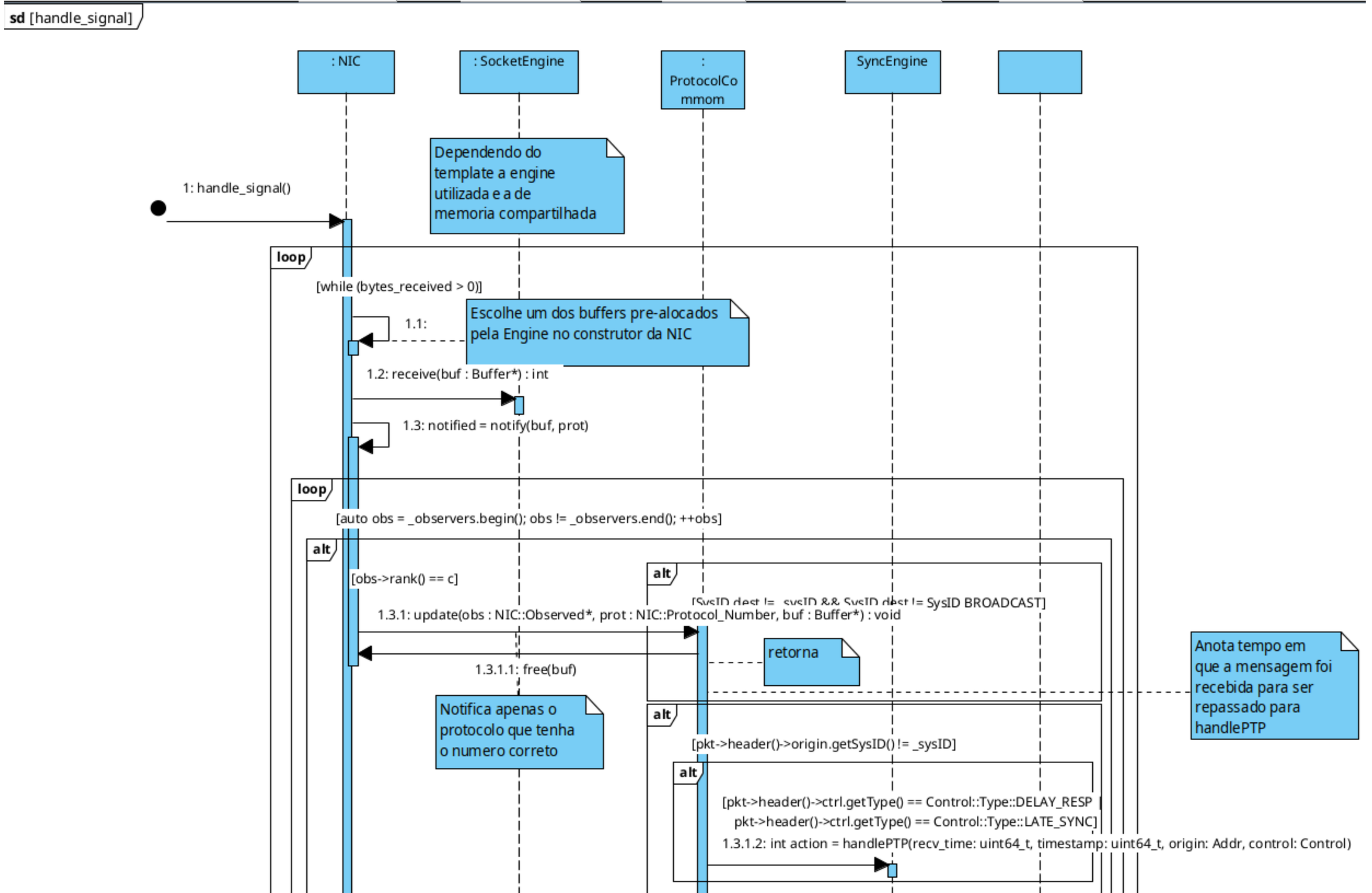

act [RSUEngineThread]
<<structured>>
<<While>>
<<structured>>
<<setup>>
_key_sender_thread_running
else
_pub_thread_running
mutex.lock()
else
_shared_data->counter == 0
Cria nova chave MAC
Armazena em memoria compartilhada entre processos
_shared_data->choosen_rsu == _protocol->getSysID()
else
_shared_data->counter = (_shared_data->counter + 1) % renew_mac_int;
Mutex.unlock()
Calcula o proximo tempo que a thread deve acordar
std::this_thread::sleep_until (next_wakeup_t)
Aguarda outras RSU
Envia chaves MAC em broadcast para seu quadrante
Dado uma RSU representada por ■, e RSUs adjacentes representadas por □, uma RSU (■) envia sua chave e as chaves das RSU adjacentes(□):
□ □ □
□ ■ □
□ □ □
pthread_barrier_wait(&_shared_data->barrier)

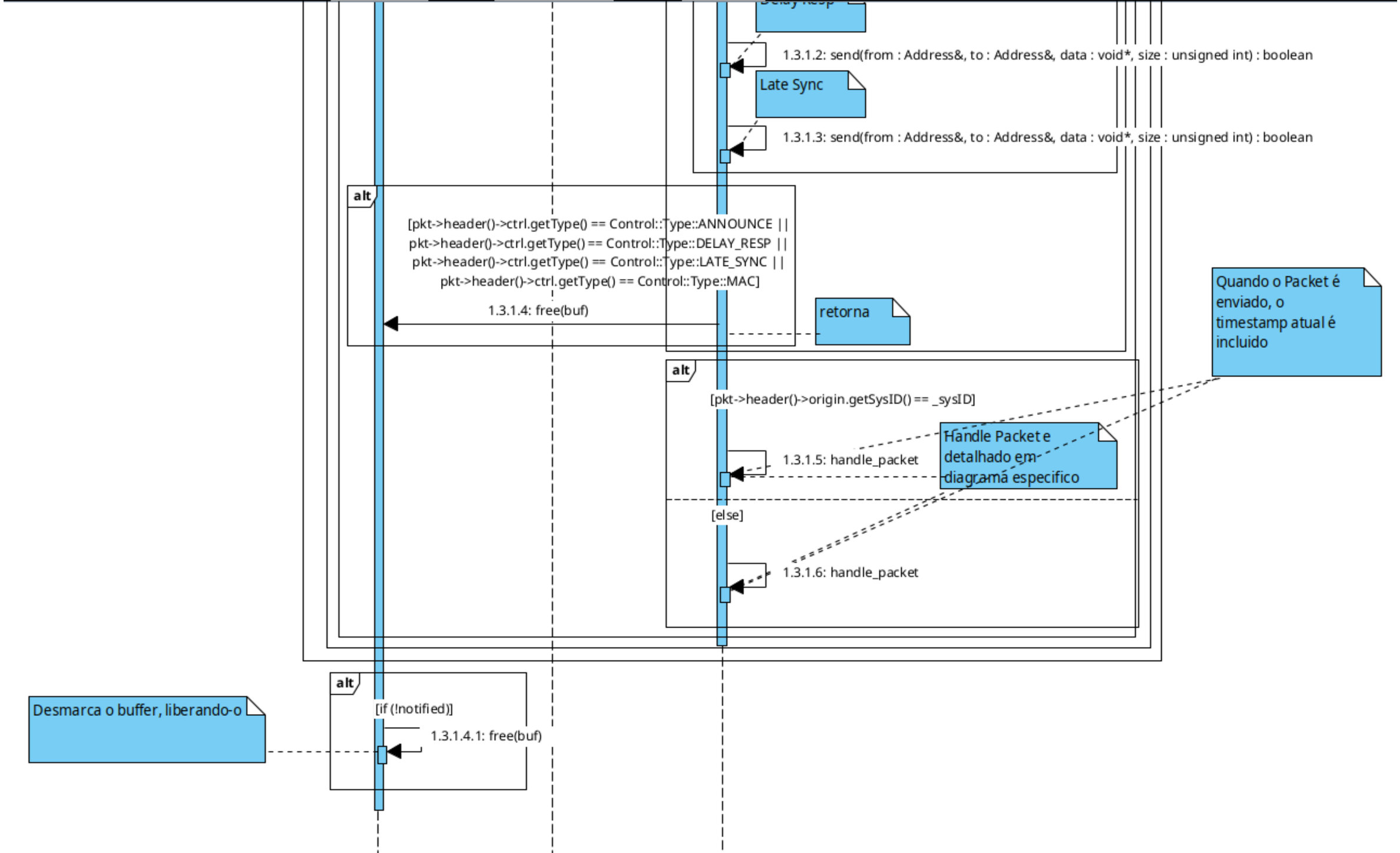

act [Recebimento de chaves MAC]
Recebe mensagem do tipo MAC
Sobrescreve chaves antigas com as novas
Mensagem contem todas as chaves necessarias para se comunicar com quaisquer quadrantes que o carro possa ter alcance

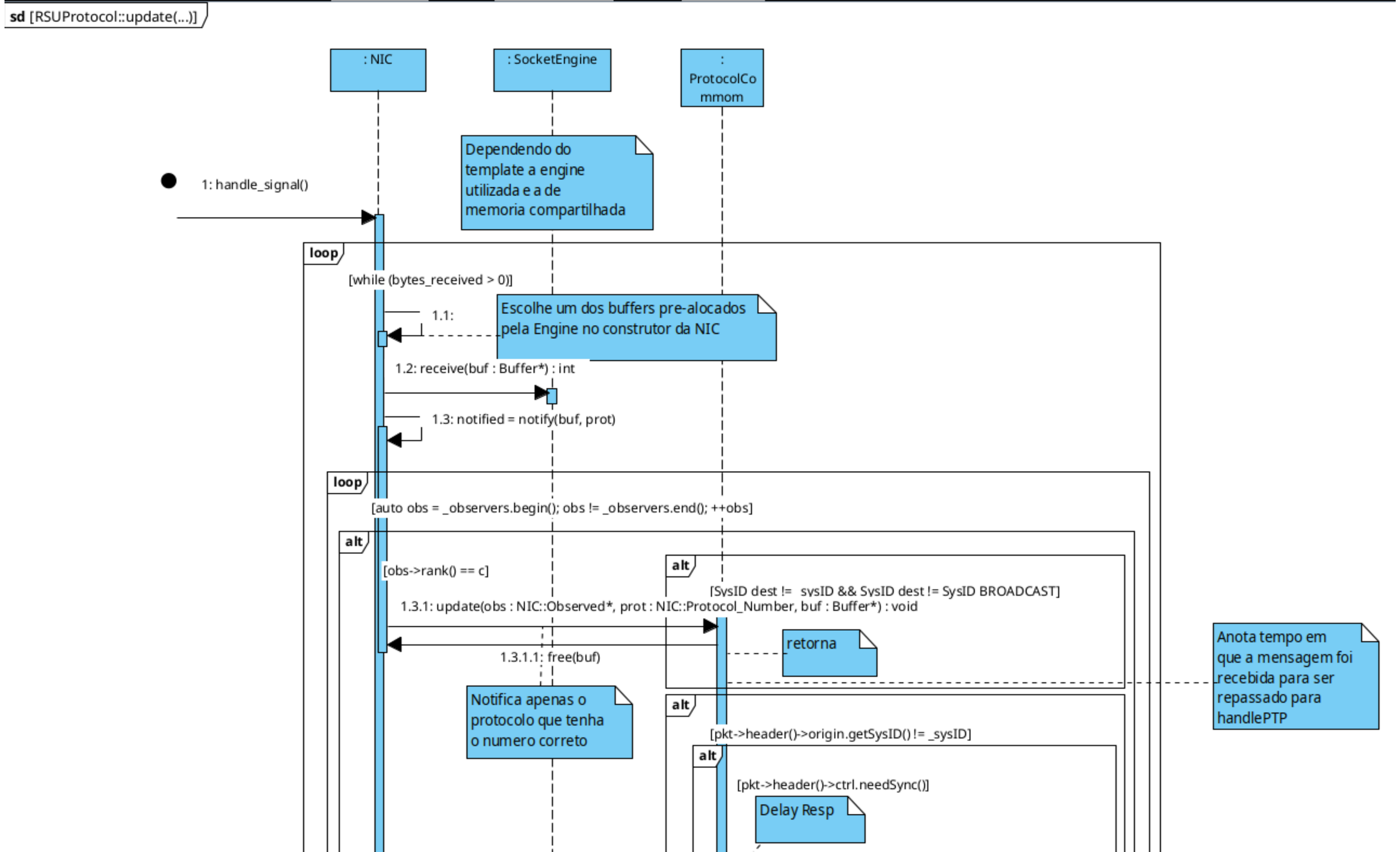

act [protocol.update(...)]
Esse metodo é chamado quando uma NIC notifica o protocolo de uma nova mensagem
O protocolo armazena um SysID em seu interior que representa o Identificador unico do veiculo
SysID destino != Proprio SysID && SysID destino != SysID de Broadcast
Libera pacote pela SocketNIC
se nao
se nao
Mensagem da engine de Sockets
se nao
Mensagem de outros veiculos dentro do alcance
Mensagem da RSU do quadrante que o veiculo se localiza
type == DELAY_RESP || type == LATE_SYNC
type == MAC
type == LATE_SYNC
type == DELAY_RESP
Anota T1 e T2
Anota T3 e T4
Recebe novas chaves MAC
Verifica MAC
Atualiza offset
Considera-se sincronizado
_announce_iteration = 0
Libera Buffer
se nao
Mac Valido
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
Porta destino == Porta Broadcast
Notifica Comunicadores
se nao
Nao notificou ninguem
Libera Buffer pela SocketNIC
<<structured>>
<<For>>
<<structured>>
<<setup>>
for (auto port : allComPorts)
port
Aloca buffer pela SocketNIC
Copia dados do buffer original para o buffer alocado
Notifica Comunicadores
se nao
Libera Buffer pela SocketNIC

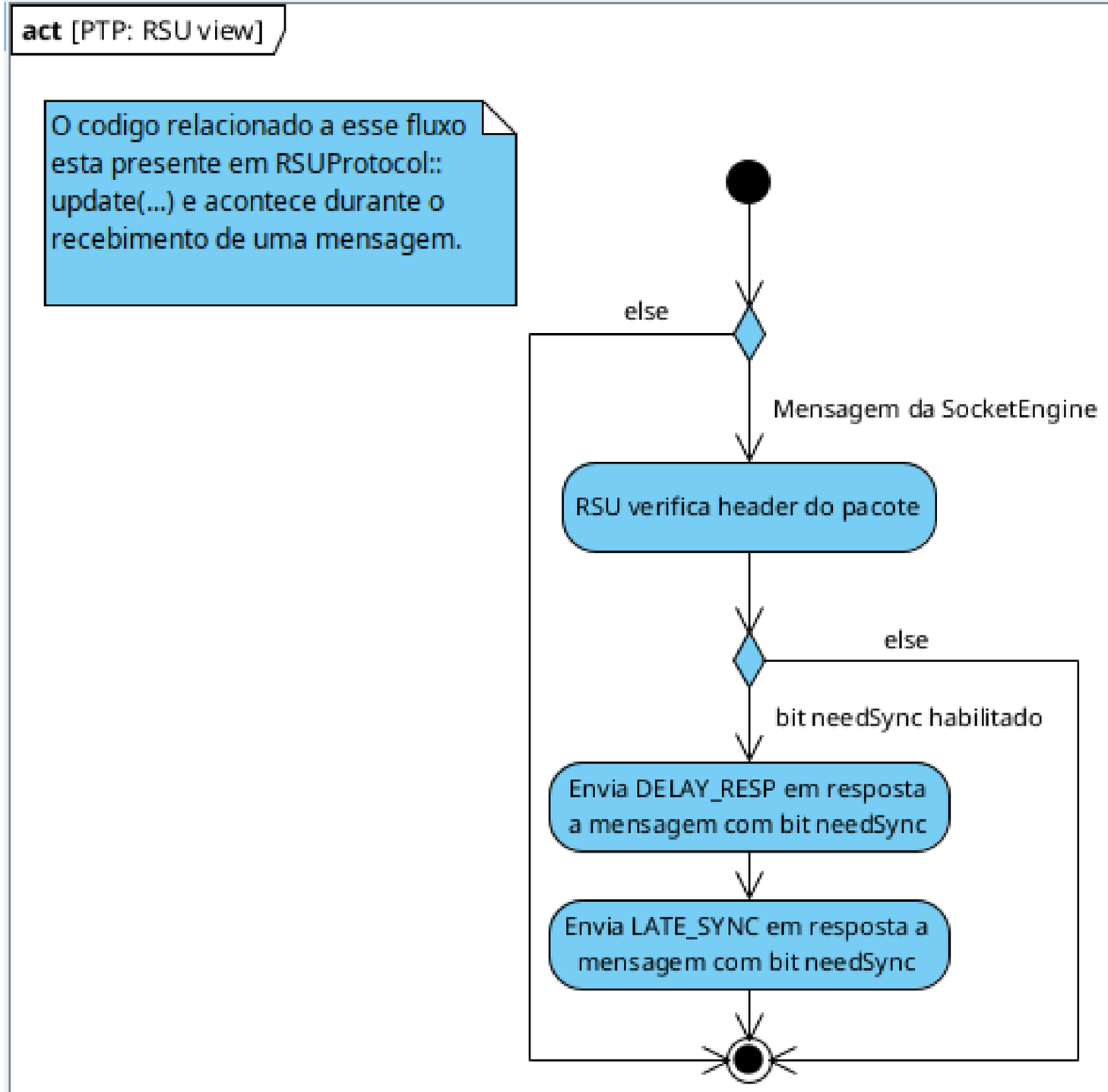

se nao
Mensagem da engine de Sockets
se nao
Mensagem de outros veiculos dentro do alcance
Mensagem da RSU do quadrante que o veiculo se localiza
type == DELAY_RESP ||
type == LATE_SYNC
type == MAC
type == LATE_SYNC
type == DELAY_RESP
Anota T1 e T2
Anota T3 e T4
Recebe novas chaves MAC
Verifica MAC
Atualiza offset
Mac Valido
se nao
Considera-se sincronizado
_announce_iteration = 0
Libera Buffer

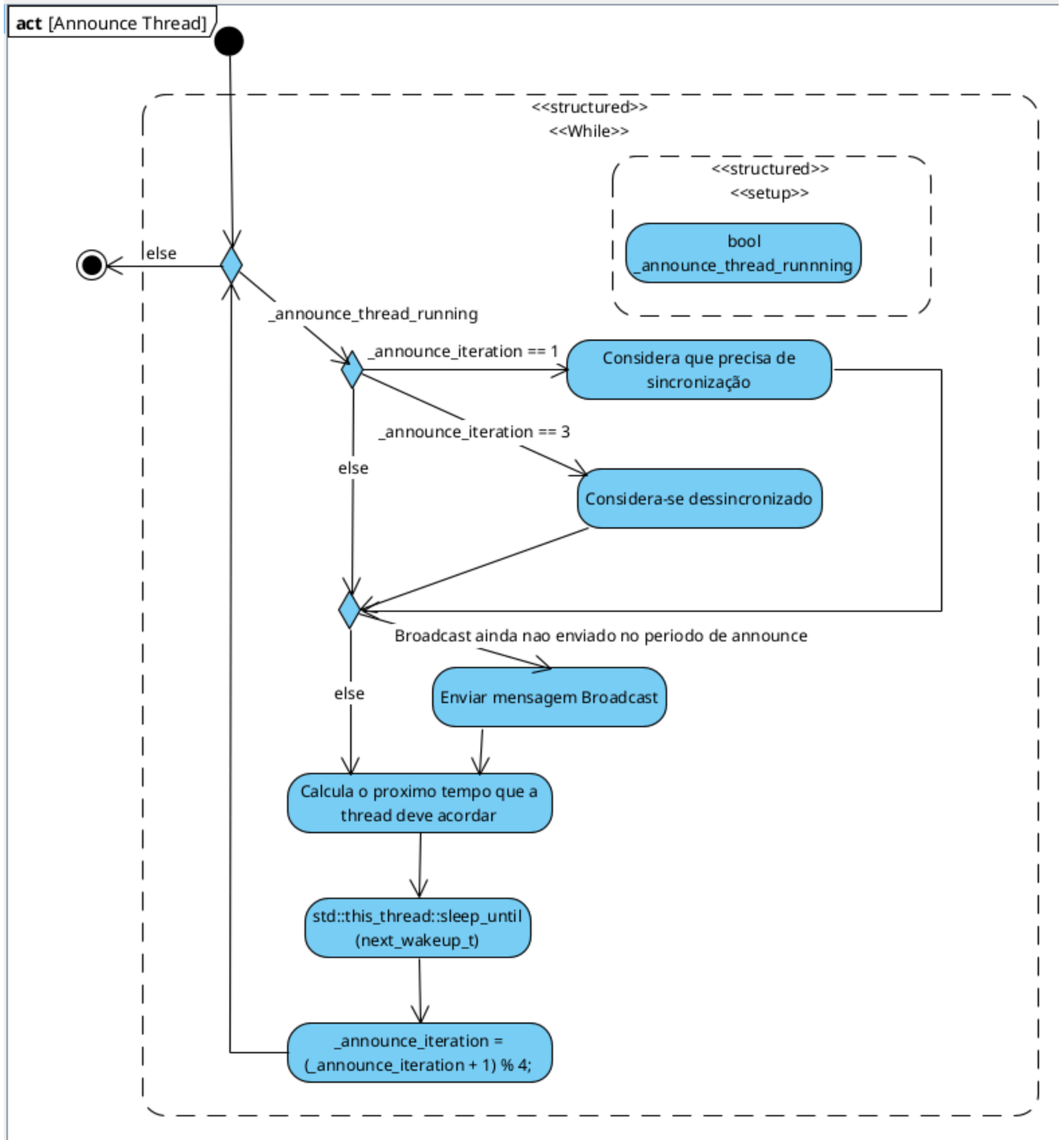

alt
[pkt->header()->ctrl.getType() == Control::Type::MAC]
1.3.1.3: setKeys(keys)
alt
[pkt->header()->ctrl.getType() == Control::Type::ANNOUNCE ||
pkt->header()->ctrl.getType() == Control::Type::DELAY_RESP ||
pkt->header()->ctrl.getType() == Control::Type::LATE_SYNC ||
pkt->header()->ctrl.getType() == Control::Type::MAC]
1.3.1.4: free(buf)
retorna
Quando o Packet é enviado, o timestamp atual é incluido
alt
[pkt->header()->origin.getSysID() == _sysID]
1.3.1.5: handle_packet
Handle Packet e detalhado em diagrama especifico
[else]
1.3.1.6: handle_packet
alt
Desmarca o buffer, liberando-o
[if (!notified)]
1.3.1.4.1: free(buf)

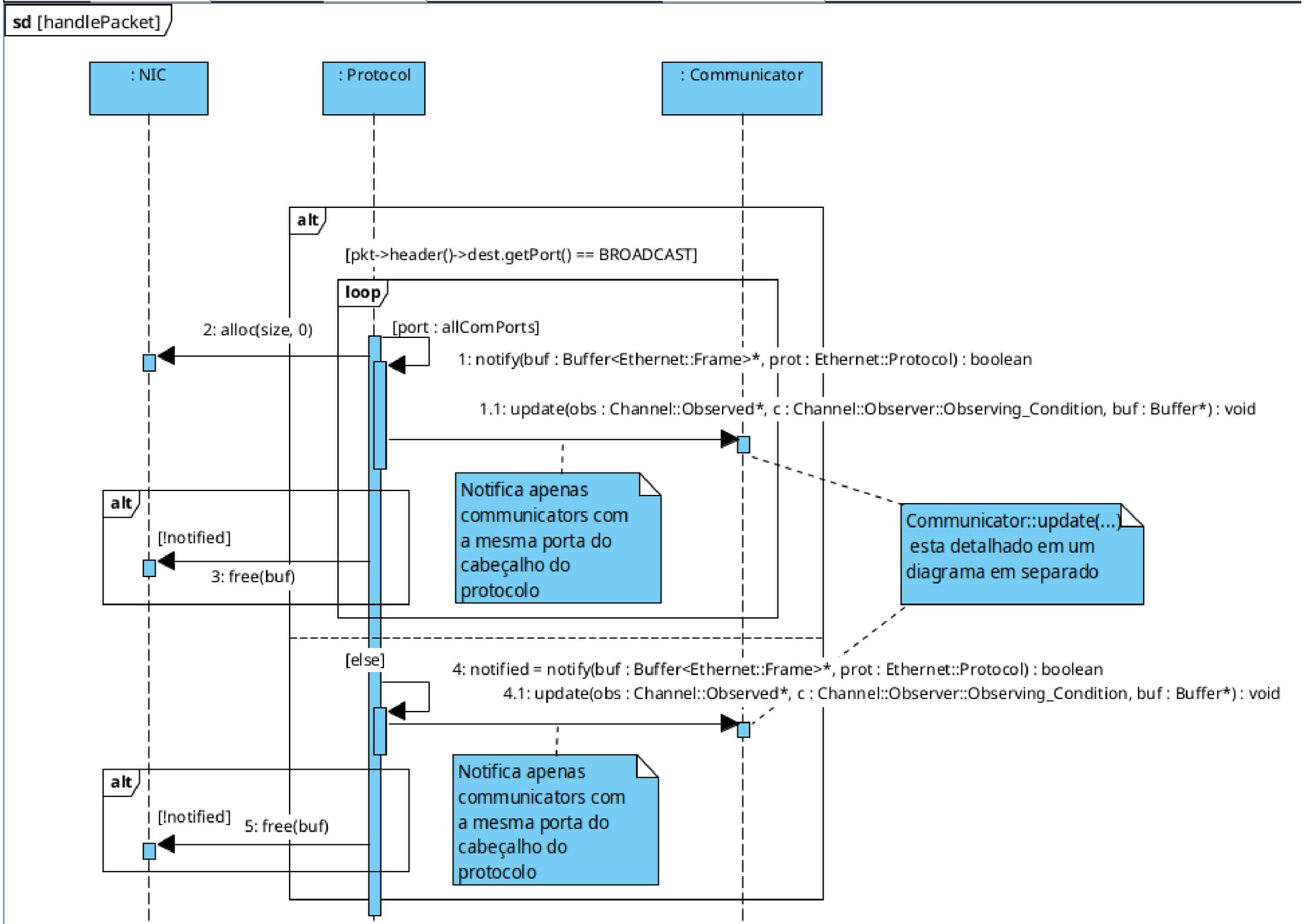

sd [handle_signal]
: NIC
: SocketEngine
: ProtocolCommom
SyncEngine
Dependendo do template a engine utilizada e a de memoria compartilhada
1: handle_signal()
loop
[while (bytes_received > 0)]
1.1:
Escolhe um dos buffers pre-alocados pela Engine no construtor da NIC
1.2: receive(buf : Buffer*) : int
1.3: notified = notify(buf, prot)
loop
[auto obs = _observers.begin(); obs != _observers.end(); ++obs]
alt
[obs->rank() == c]
alt
[SysID dest != _sysID && SysID dest != SysID BROADCAST]
1.3.1: update(obs : NIC::Observed*, prot : NIC::Protocol_Number, buf : Buffer*) : void
retorna
1.3.1.1: free(buf)
Notifica apenas o protocolo que tenha o numero correto
Anota tempo em que a mensagem foi recebida para ser repassado para handlePTP
alt
[pkt->header()->origin.getSysID() != _sysID]
alt
[pkt->header()->ctrl.getType() == Control::Type::DELAY_RESP
pkt->header()->ctrl.getType() == Control::Type::LATE_SYNC]
1.3.1.2: int action = handlePTP(recv_time: uint64_t, timestamp: uint64_t, origin: Addr, control: Control)


1.3.1.2: send(from : Address&, to : Address&, data : void*, size : unsigned int) : boolean
Late Sync
1.3.1.3: send(from : Address&, to : Address&, data : void*, size : unsigned int) : boolean
alt
[pkt->header()->ctrl.getType() == Control::Type::ANNOUNCE ||
pkt->header()->ctrl.getType() == Control::Type::DELAY_RESP ||
pkt->header()->ctrl.getType() == Control::Type::LATE_SYNC ||
pkt->header()->ctrl.getType() == Control::Type::MAC]
1.3.1.4: free(buf)
retorna
Quando o Packet é enviado, o timestamp atual é incluido
alt
[pkt->header()->origin.getSysID() == _sysID]
1.3.1.5: handle_packet
Handle Packet e detalhado em diagrama especifico
[else]
1.3.1.6: handle_packet
alt
[if (!notified)]
1.3.1.4.1: free(buf)
Desmarca o buffer, liberando-o

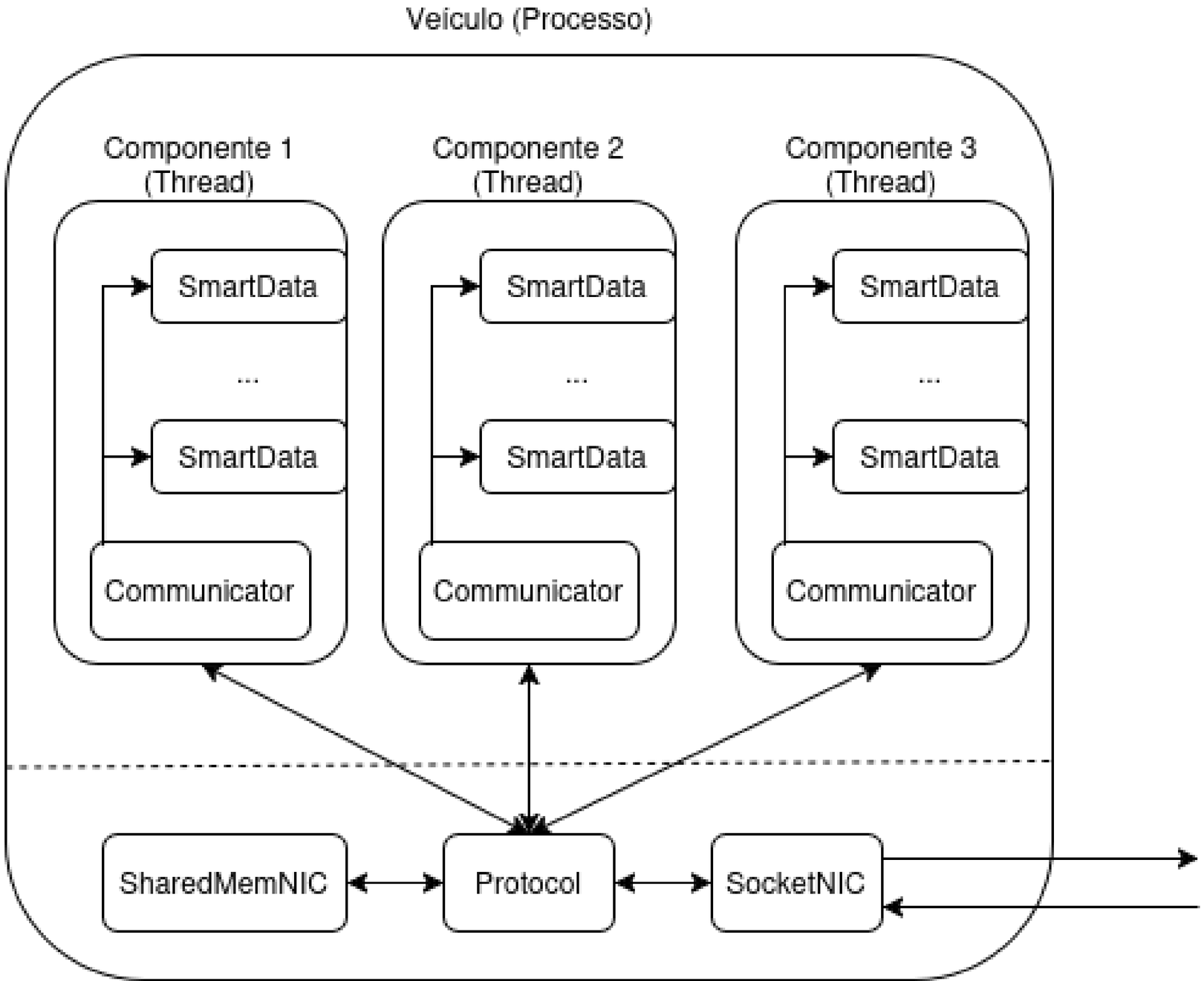

sd [RSUProtocol::update(...)]
: NIC
: SocketEngine
: ProtocolCommom
Dependendo do template a engine utilizada e a de memoria compartilhada
1: handle_signal()
loop
[while (bytes_received > 0)]
1.1:
Escolhe um dos buffers pre-alocados pela Engine no construtor da NIC
1.2: receive(buf : Buffer*) : int
1.3: notified = notify(buf, prot)
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
1.3.1.1: free(buf)
Notifica apenas o protocolo que tenha o numero correto
alt
[SysID dest != sysID && SysID dest != SysID BROADCAST]
retorna
Anota tempo em que a mensagem foi recebida para ser repassado para handlePTP
alt
[pkt->header()->origin.getSysID() != _sysID]
alt
[pkt->header()->ctrl.needSync()]
Delay Resp


act [PTP: RSU view]
O codigo relacionado a esse fluxo esta presente em RSUProtocol:: update(...) e acontece durante o recebimento de uma mensagem.
else
Mensagem da SocketEngine
RSU verifica header do pacote
else
bit needSync habilitado
Envia DELAY_RESP em resposta a mensagem com bit needSync
Envia LATE_SYNC em resposta a mensagem com bit needSync

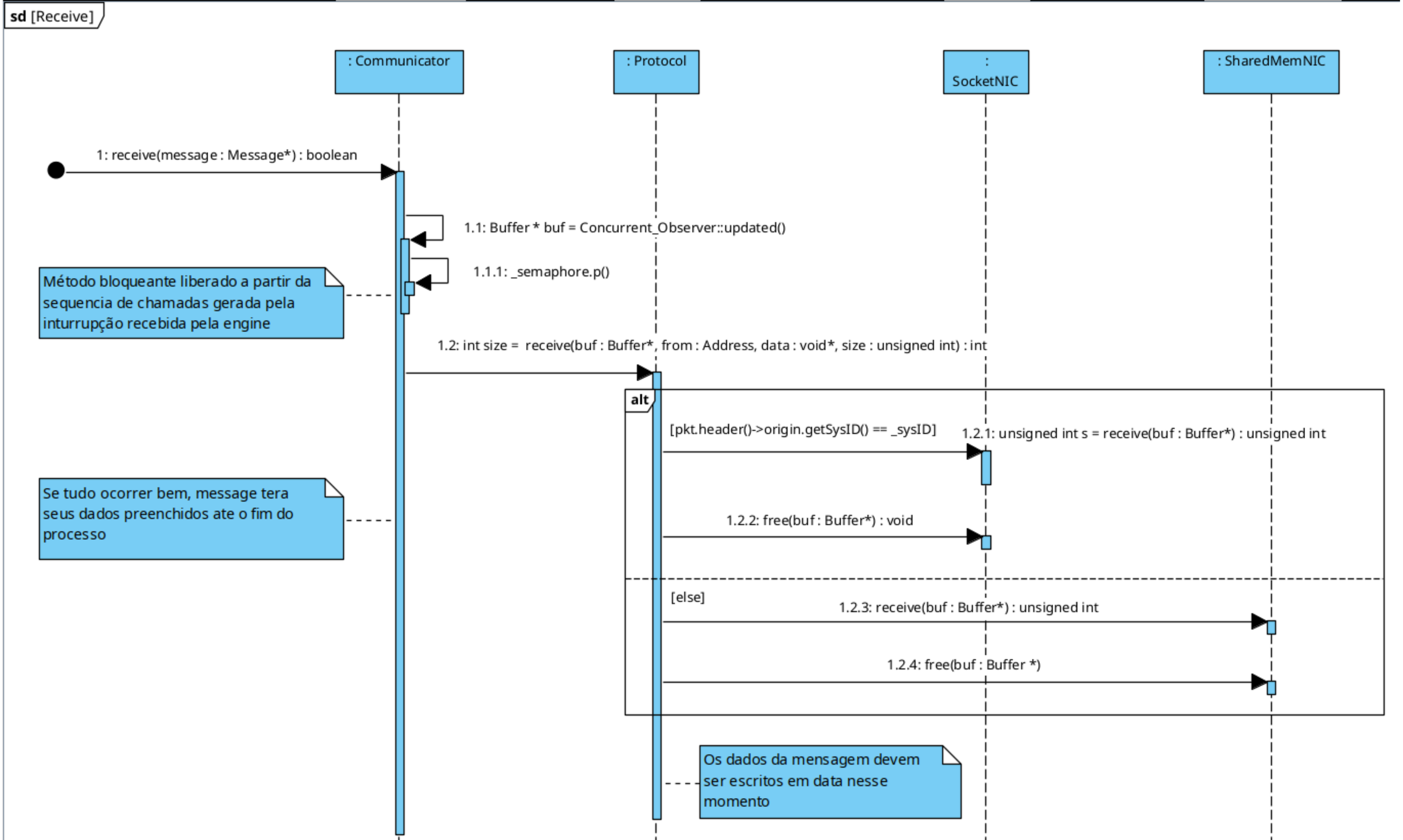

act [Announce Thread]
<<structured>>
<<While>>
<<structured>>
<<setup>>
bool
_announce_thread_runnning
else
_announce_thread_running
_announce_iteration == 1
Considera que precisa de
sincronização
_announce_iteration == 3
else
Considera-se dessincronizado
Broadcast ainda nao enviado no periodo de announce
else
Enviar mensagem Broadcast
Calcula o proximo tempo que a
thread deve acordar
std::this_thread::sleep_until
(next_wakeup_t)
_announce_iteration =
(_announce_iteration + 1) % 4;

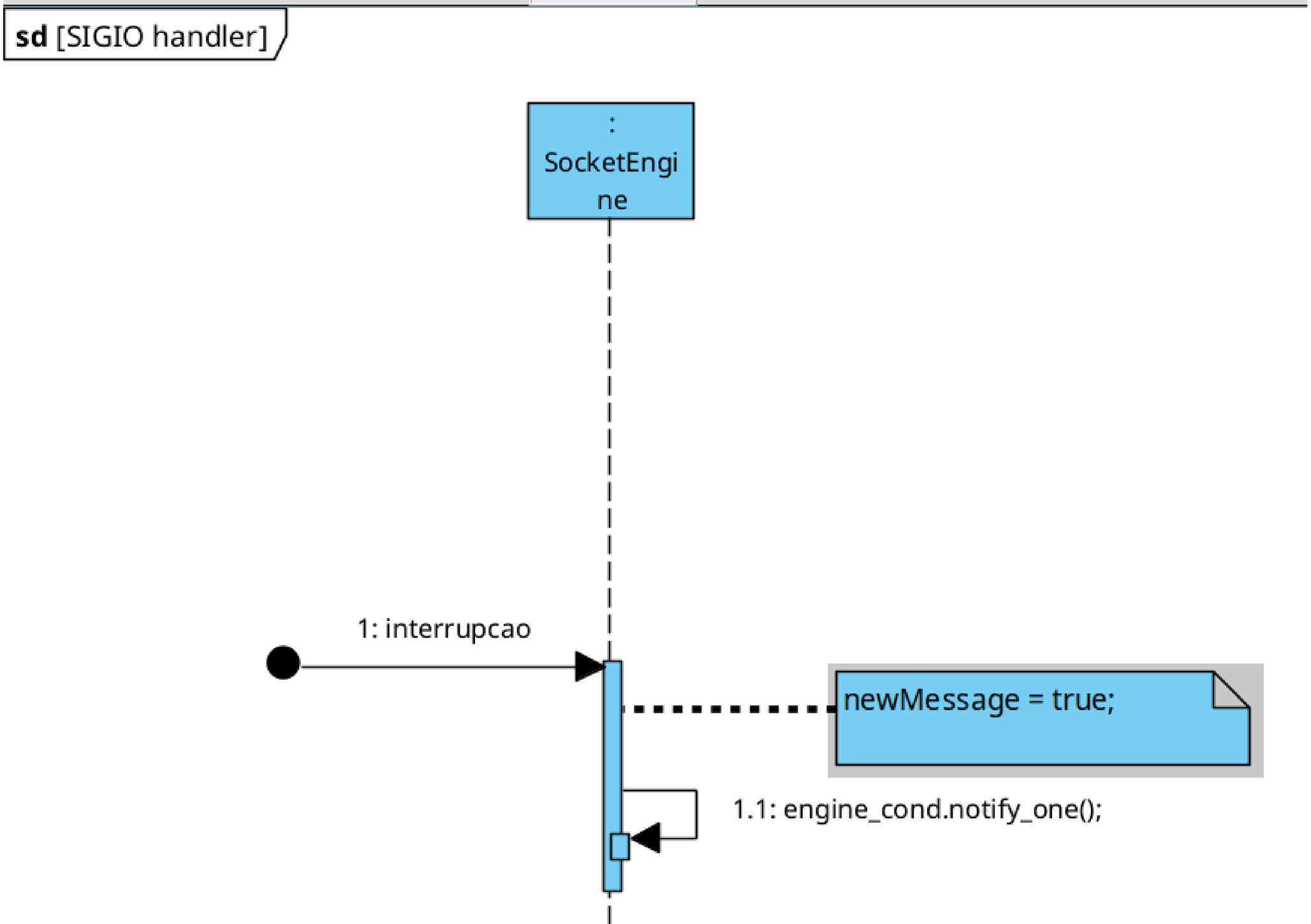

sd [handlePacket]
: NIC
: Protocol
: Communicator
alt
[pkt->header()->dest.getPort() == BROADCAST]
loop
2: alloc(size, 0)
[port : allComPorts]
1: notify(buf : Buffer<Ethernet::Frame>*, prot : Ethernet::Protocol) : boolean
1.1: update(obs : Channel::Observed*, c : Channel::Observer::Observing_Condition, buf : Buffer*) : void
Notifica apenas communicators com a mesma porta do cabeçalho do protocolo
Communicator::update(...) esta detalhado em um diagrama em separado
alt
[!notified]
3: free(buf)
[else]
4: notified = notify(buf : Buffer<Ethernet::Frame>*, prot : Ethernet::Protocol) : boolean
4.1: update(obs : Channel::Observed*, c : Channel::Observer::Observing_Condition, buf : Buffer*) : void
Notifica apenas communicators com a mesma porta do cabeçalho do protocolo
alt
[!notified]
5: free(buf)

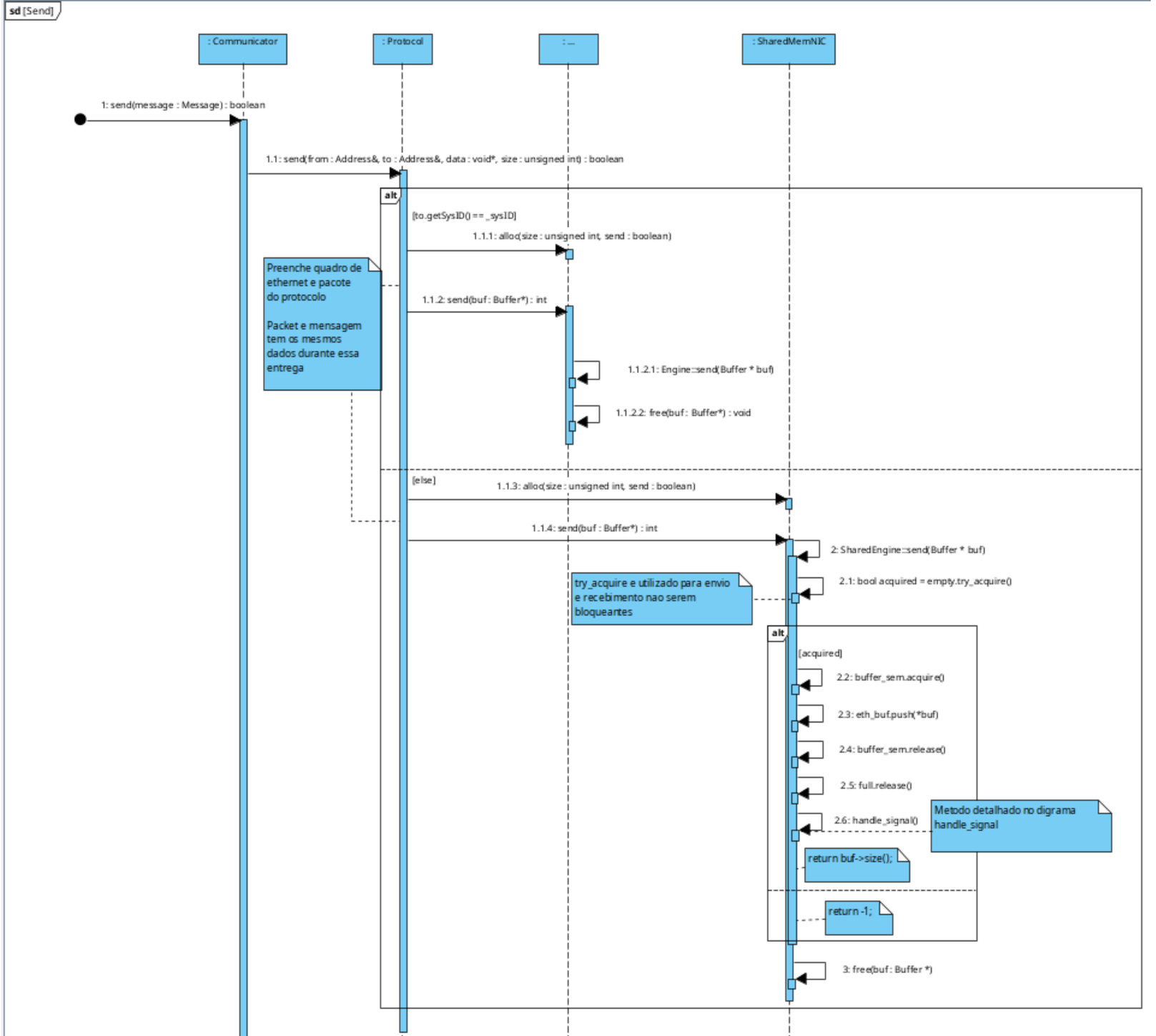
Estrutura do Veículo

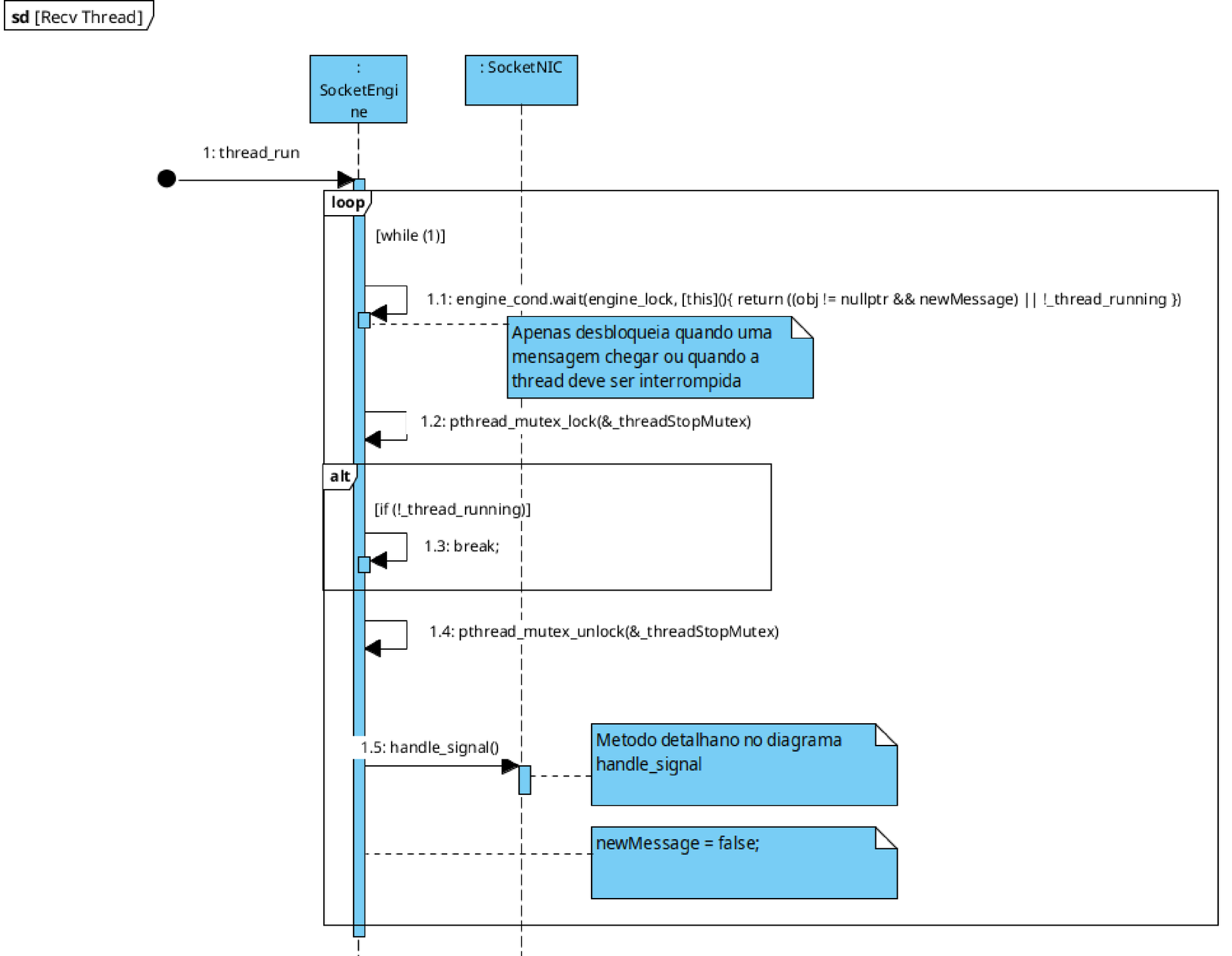


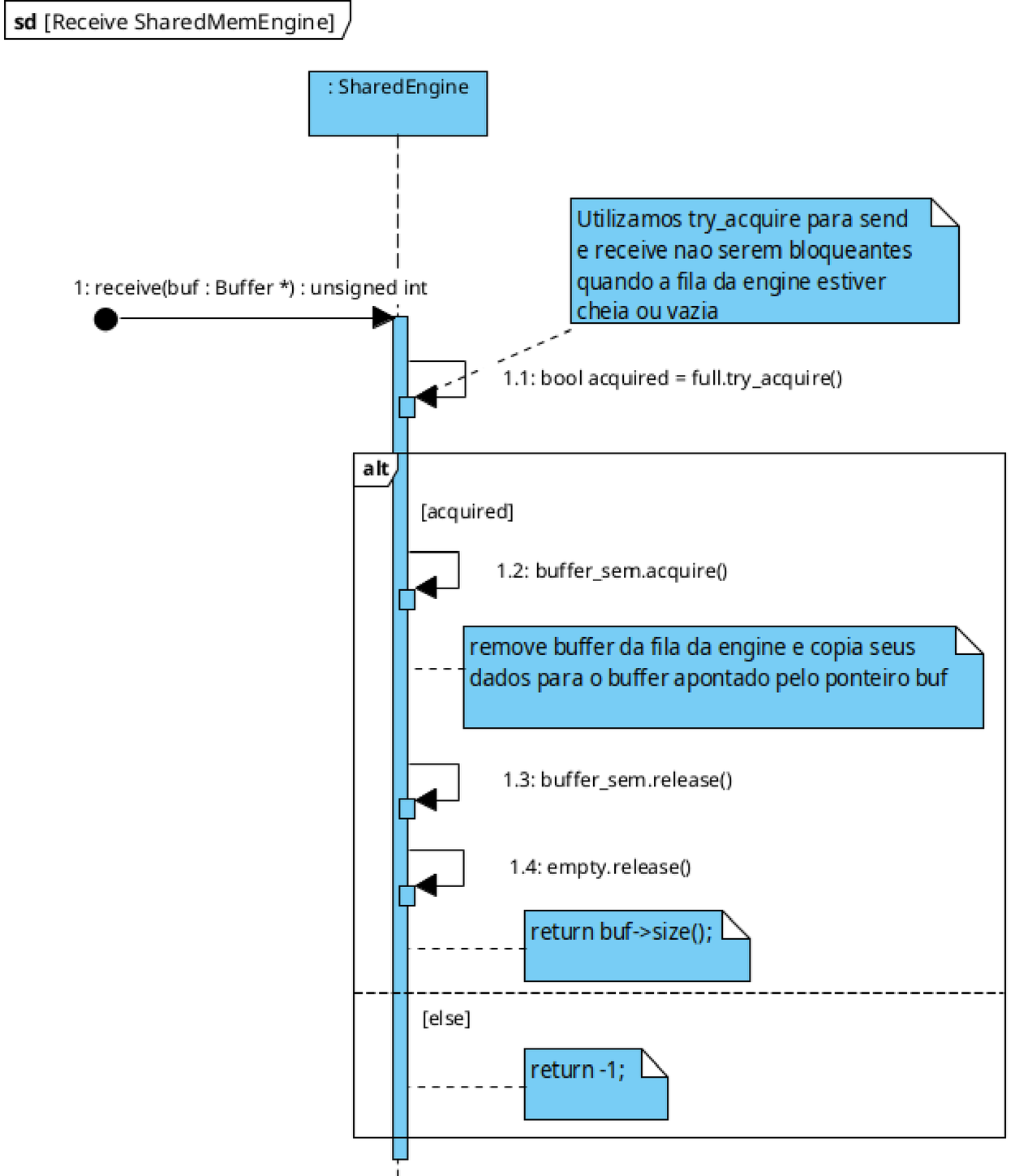
Diagramas de Sequência

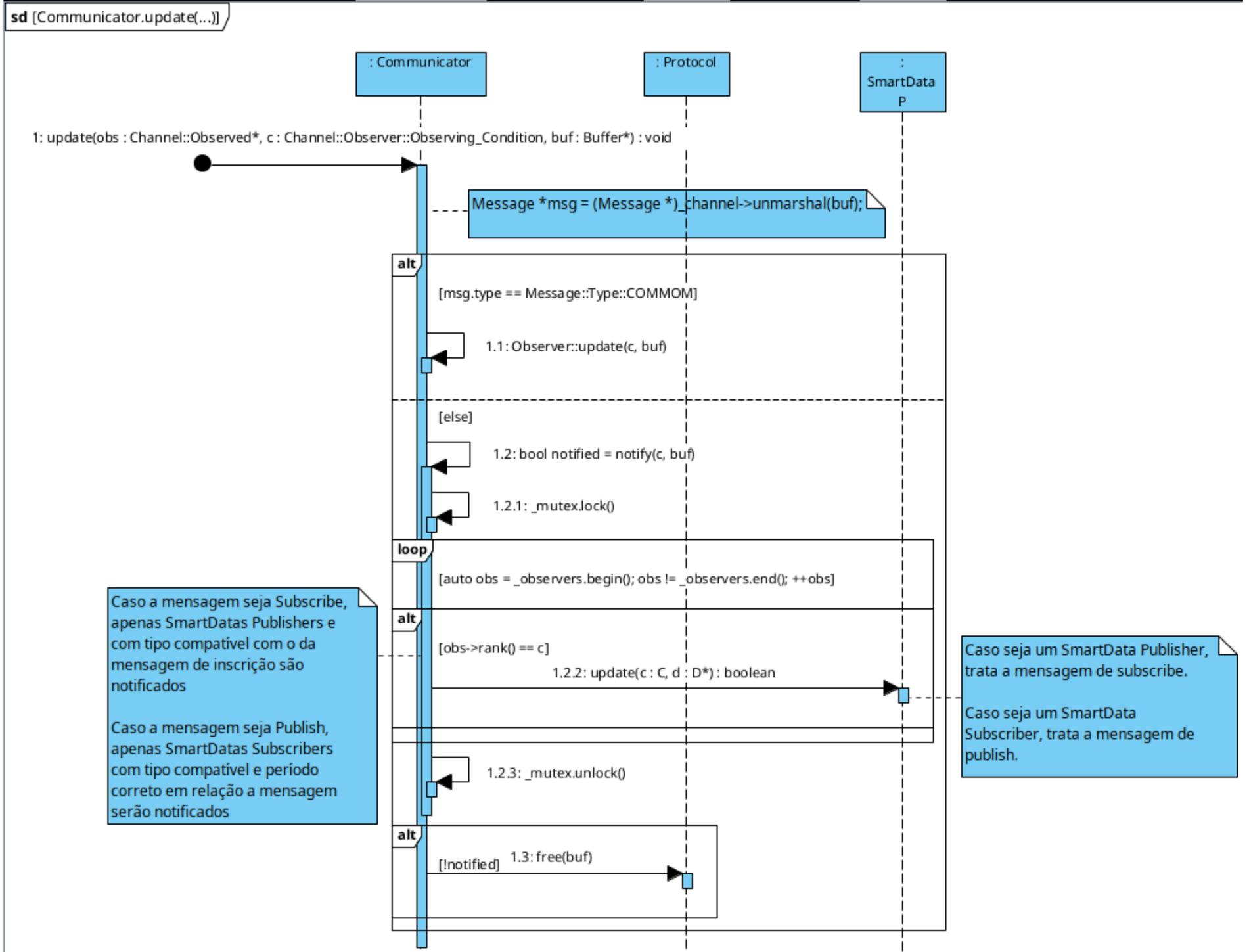

sd [Receive]
: Communicator
: Protocol
: SocketNIC
: SharedMemNIC
1: receive(message : Message*) : boolean
1.1: Buffer * buf = Concurrent_Observer::updated()
1.1.1: _semaphore.p()
Método bloqueante liberado a partir da sequencia de chamadas gerada pela inturrupção recebida pela engine
1.2: int size = 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
Se tudo ocorrer bem, message tera seus dados preenchidos ate o fim do processo
1.2.2: free(buf : Buffer*) : void
[else]
1.2.3: receive(buf : Buffer*) : unsigned int
1.2.4: free(buf : Buffer *)
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();

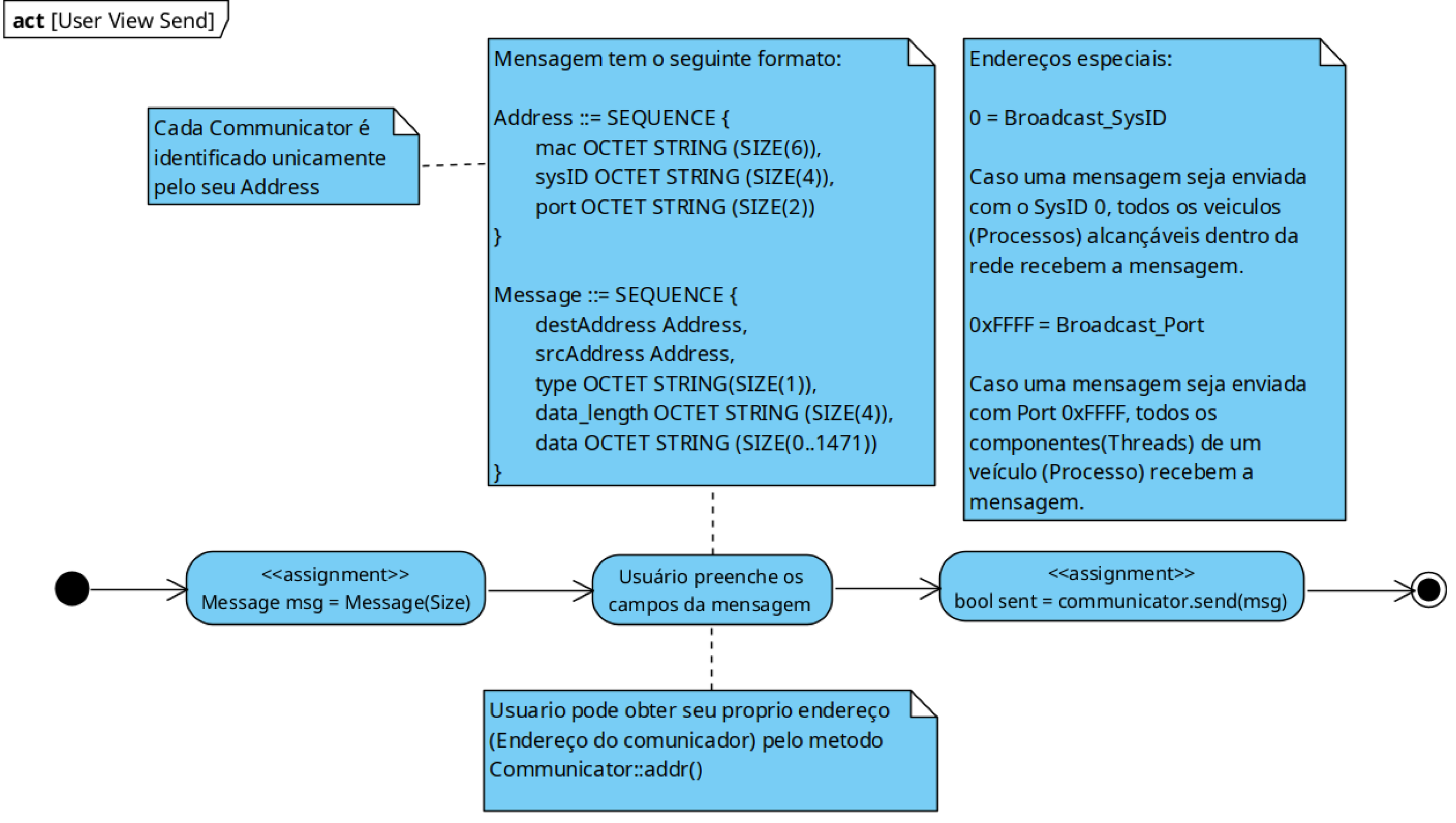

sd [Send]
: Communicator
: Protocol
: ...
: SharedMemNIC
1: send(message : Message) : boolean
1.1: send(from : Address&, to : Address&, data : void*, size : unsigned int) : boolean
alt
[to.getSysID() == _sysID]
1.1.1: alloc(size : unsigned int, send : boolean)
Preenche quadro de ethernet e pacote do 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)
2.1: bool acquired = empty.try_acquire()
try_acquire e utilizado para envio e recebimento nao serem bloqueantes
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 *)

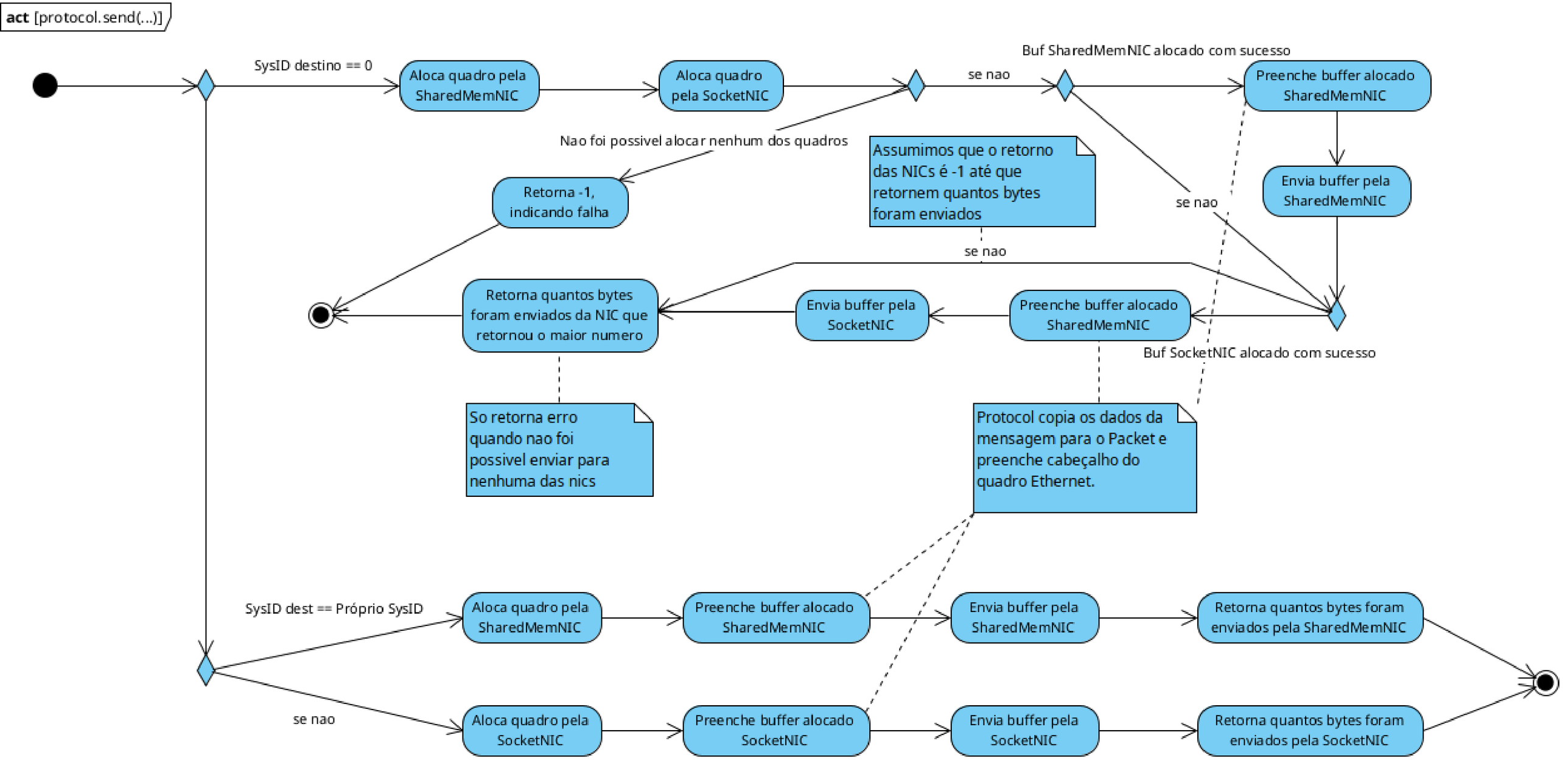

sd [Recv Thread]
: SocketEngine
: SocketNIC
1: thread_run
loop
[while (1)]
1.1: engine_cond.wait(engine_lock, [this](){ return ((obj != nullptr && newMessage) || !_thread_running })
Apenas desbloqueia quando uma mensagem chegar ou quando a thread deve ser interrompida
1.2: pthread_mutex_lock(&_threadStopMutex)
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;

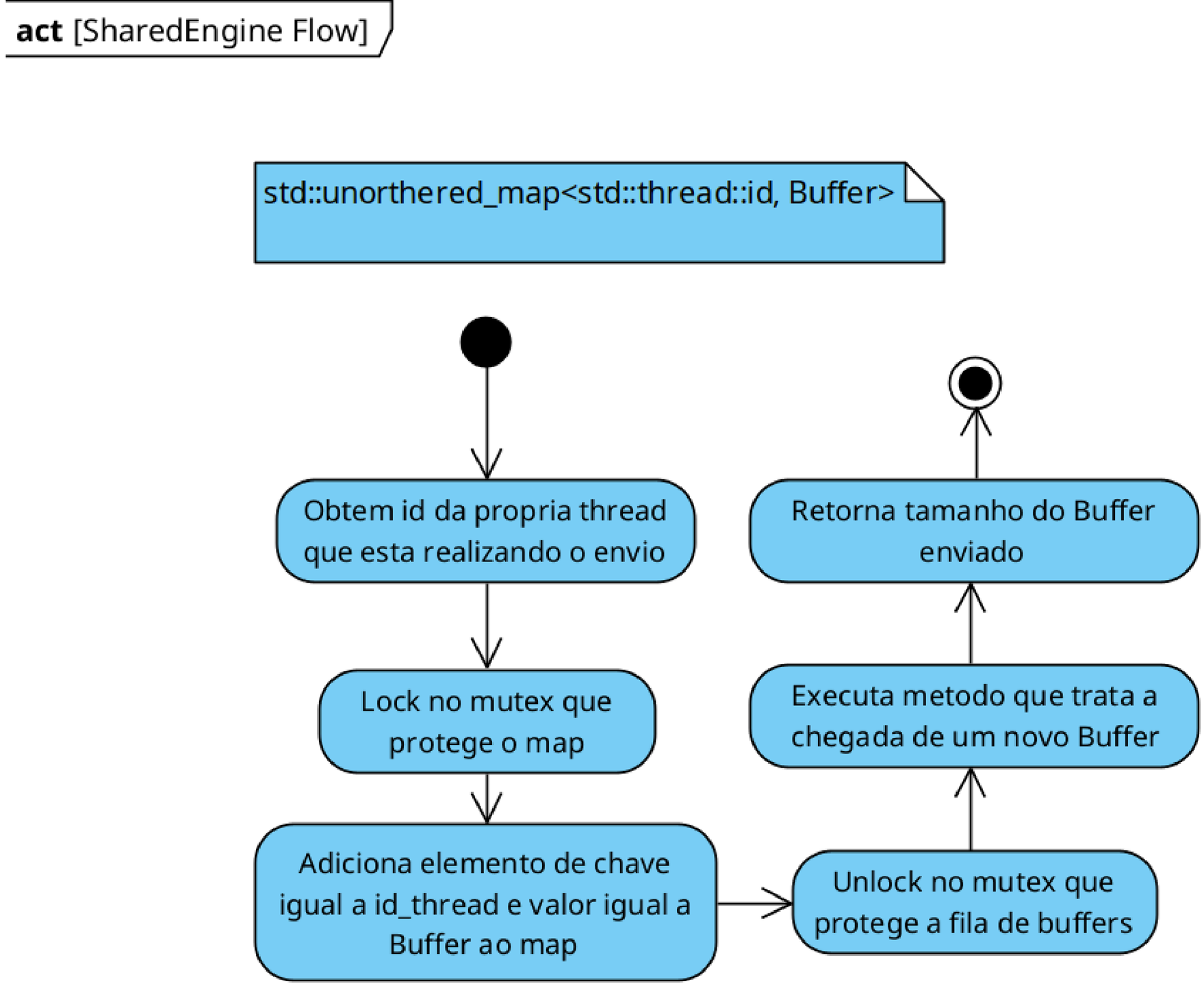

sd [Receive SharedMemEngine]
: SharedEngine
1: receive(buf : Buffer *) : unsigned int
Utilizamos try_acquire para send e receive nao serem bloqueantes quando a fila da engine estiver cheia ou vazia
1.1: bool acquired = full.try_acquire()
alt
[acquired]
1.2: buffer_sem.acquire()
remove buffer da fila da engine e copia seus dados para o buffer apontado pelo ponteiro buf
1.3: buffer_sem.release()
1.4: empty.release()
return buf->size();
[else]
return -1;

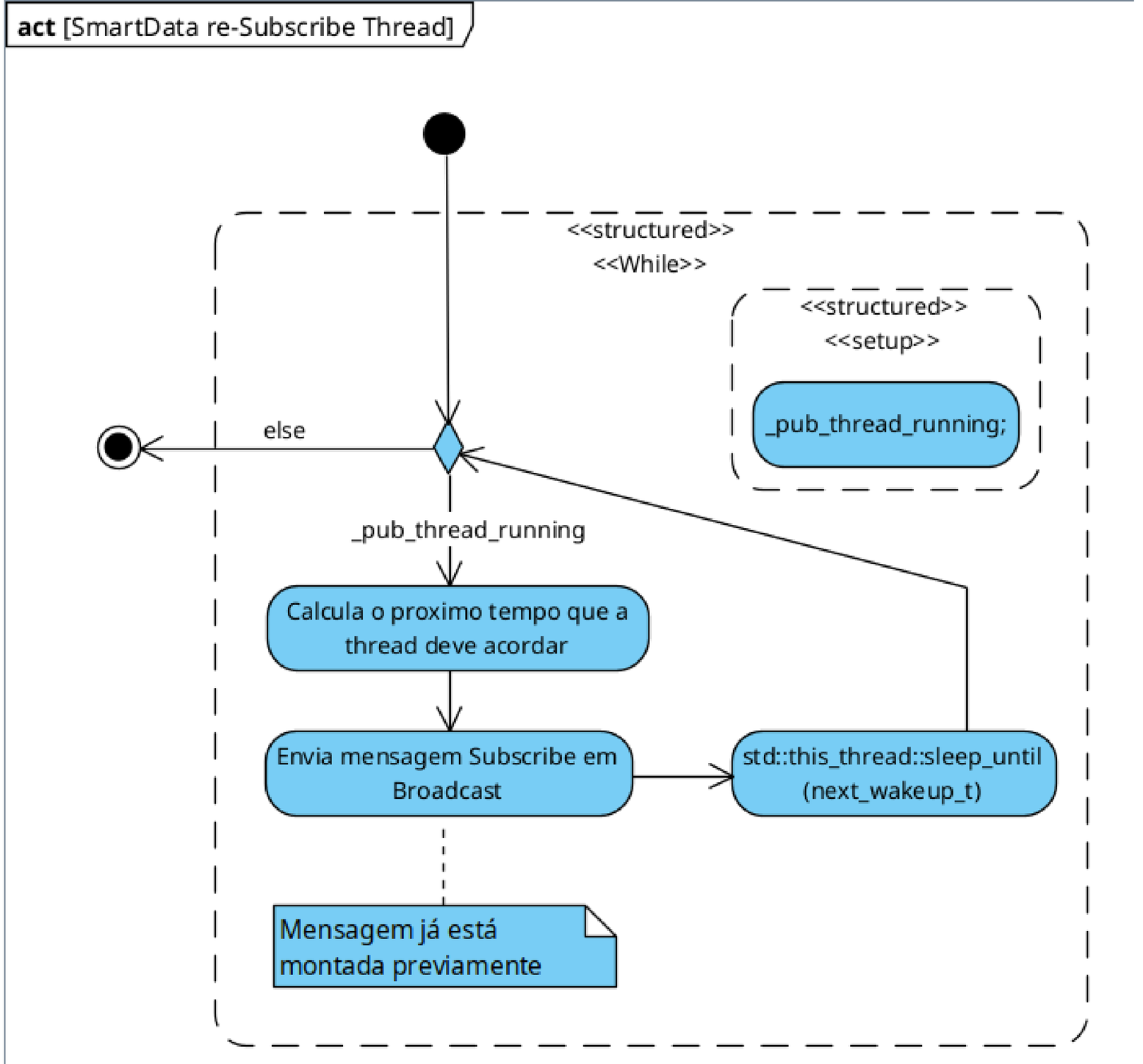

sd [Communicator.update(...)]
: Communicator
: Protocol
: SmartData P
1: update(obs : Channel::Observed*, c : Channel::Observer::Observing_Condition, buf : Buffer*) : void
Message *msg = (Message *)_channel->unmarshal(buf);
alt
[msg.type == Message::Type::COMMOM]
1.1 : Observer::update(c, buf)
[else]
1.2: bool notified = notify(c, buf)
1.2.1: _mutex.lock()
loop
[auto obs = _observers.begin(); obs != _observers.end(); ++obs]
alt
[obs->rank() == c]
1.2.2: update(c : C, d : D*) : boolean
1.2.3: _mutex.unlock()
alt
[!notified]
1.3: free(buf)
Caso a mensagem seja Subscribe, apenas SmartDatas Publishers e com tipo compatível com o da mensagem de inscrição são notificados
Caso a mensagem seja Publish, apenas SmartDatas Subscribers com tipo compatível e período correto em relação a mensagem serão notificados
Caso seja um SmartData Publisher, trata a mensagem de subscribe.
Caso seja um SmartData Subscriber, trata a mensagem de publish.

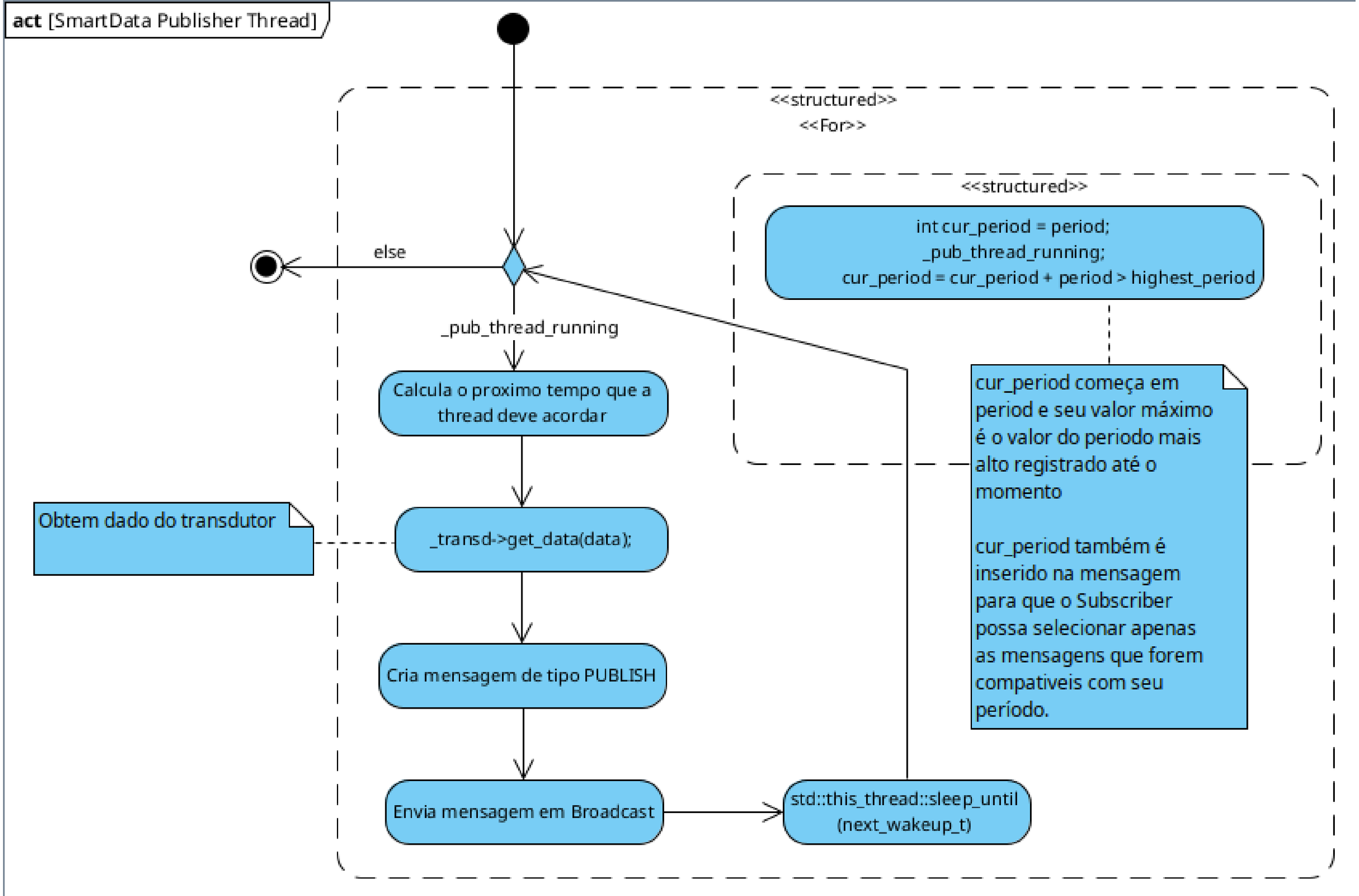
Diagramas de Atividade

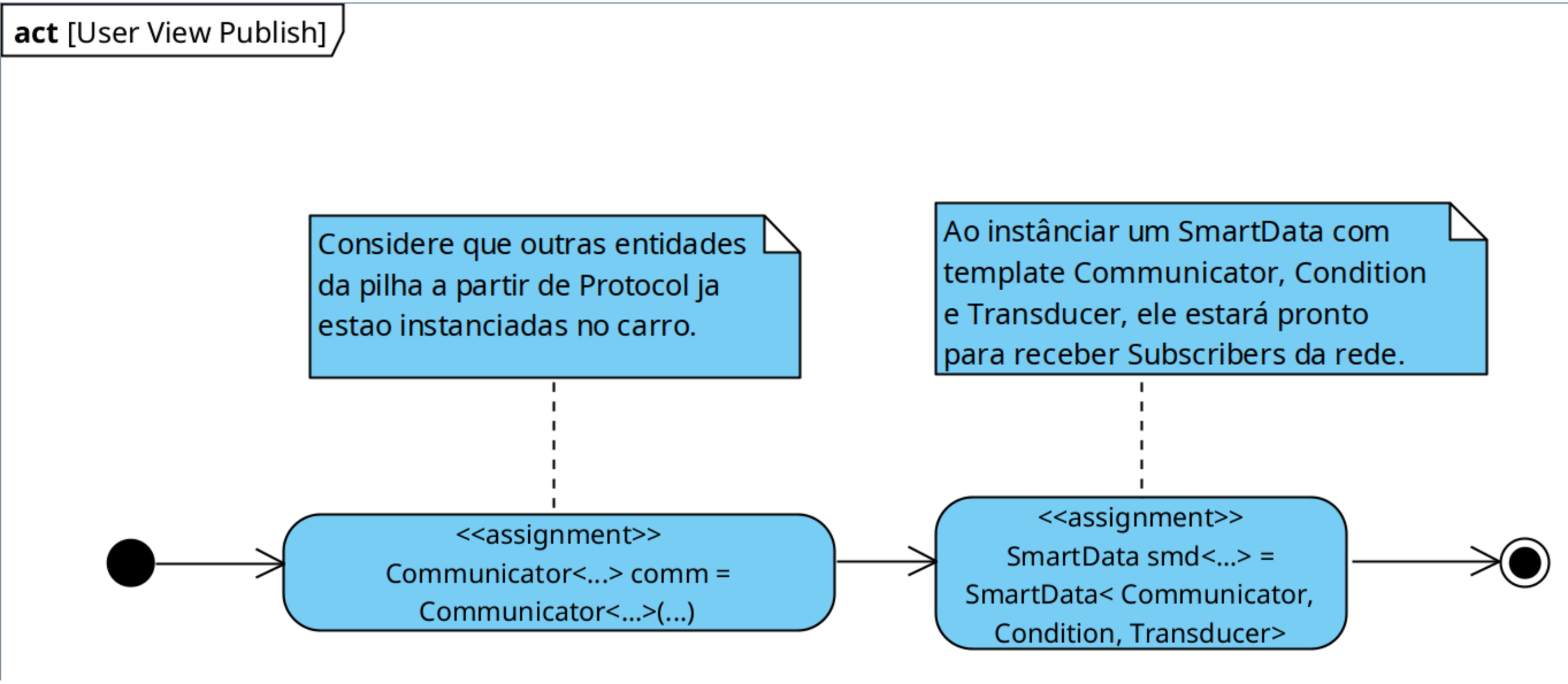

act [User View Send]
Cada Communicator é identificado unicamente pelo seu Address
Mensagem tem o seguinte formato:
Address ::= SEQUENCE {
mac OCTET STRING (SIZE(6)),
sysID OCTET STRING (SIZE(4)),
port OCTET STRING (SIZE(2))
}
Message ::= SEQUENCE {
destAddress Address,
srcAddress Address,
type OCTET STRING(SIZE(1)),
data_length OCTET STRING (SIZE(4)),
data OCTET STRING (SIZE(0..1471))
}
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()

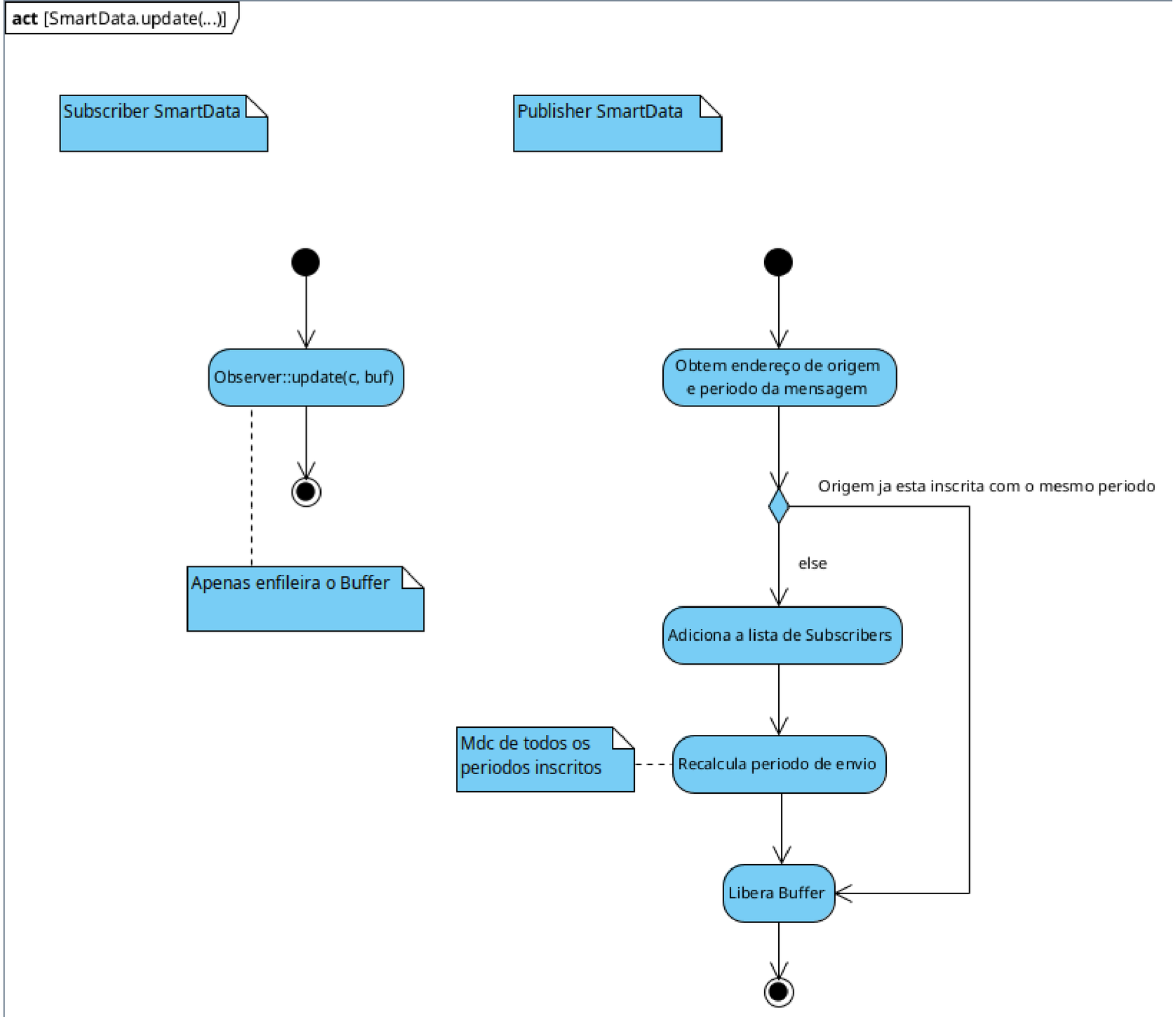

act [protocol.send(...)]
SysID destino == 0
Aloca quadro pela SharedMemNIC
Aloca quadro pela SocketNIC
Nao foi possivel alocar nenhum dos quadros
Retorna -1, indicando falha
se nao
Buf SharedMemNIC alocado com sucesso
Preenche buffer alocado SharedMemNIC
Envia buffer pela SharedMemNIC
se nao
Assumimos 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

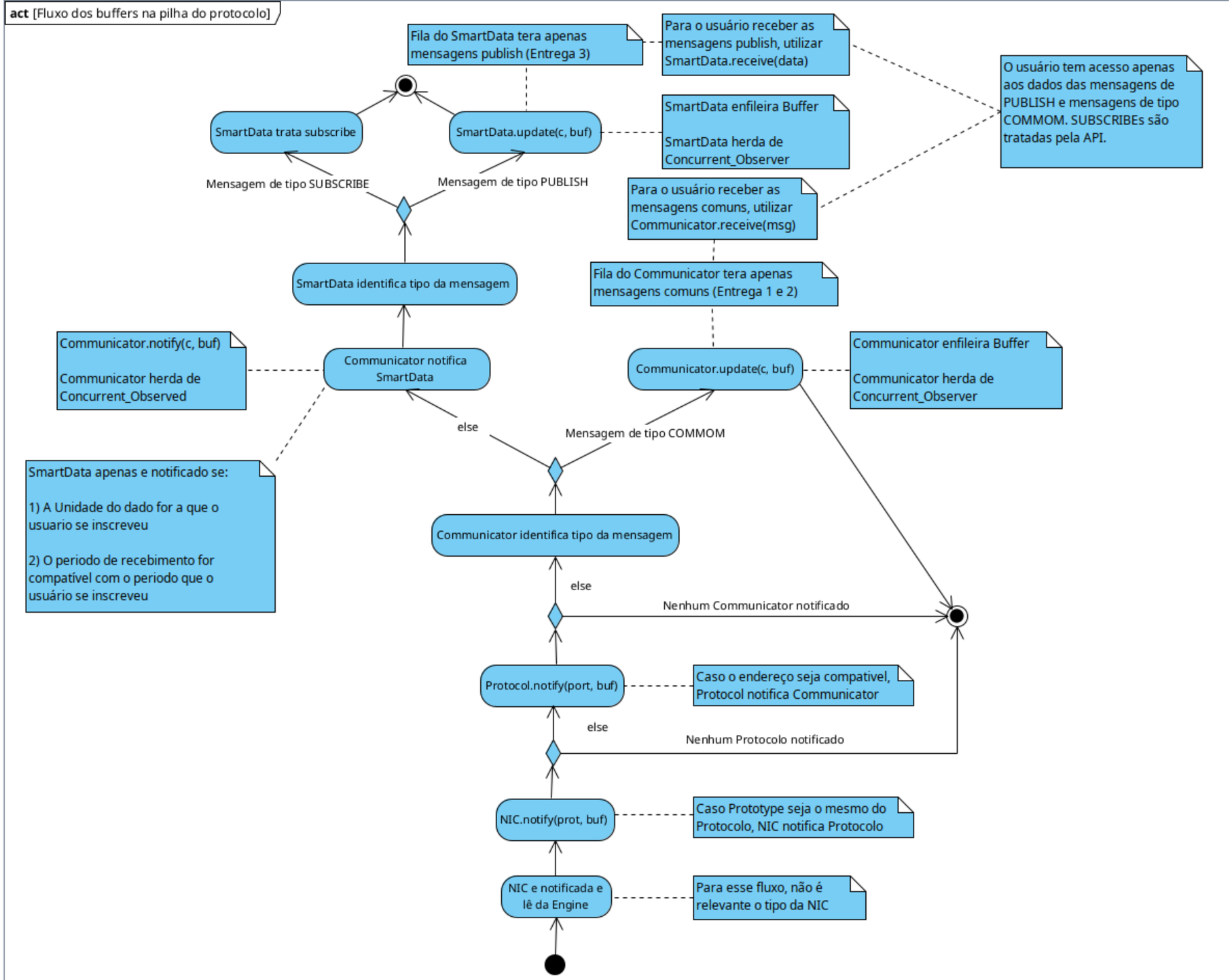

act [SharedEngine Flow]
std::unorthered_map<std::thread::id, Buffer>
Obtem id da propria thread que esta realizando o envio
Lock no mutex que protege o map
Adiciona elemento de chave igual a id_thread e valor igual a Buffer ao map
Unlock no mutex que protege a fila de buffers
Executa metodo que trata a chegada de um novo Buffer
Retorna tamanho do Buffer enviado


act [SmartData re-Subscribe Thread]
<<structured>>
<<While>>
<<structured>>
<<setup>>
_pub_thread_running;
else
_pub_thread_running
Calcula o proximo tempo que a thread deve acordar
Envia mensagem Subscribe em Broadcast
std::this_thread::sleep_until (next_wakeup_t)
Mensagem já está montada previamente


act [SmartData Publisher Thread]
<<structured>>
<<For>>
<<structured>>
int cur_period = period;
_pub_thread_running;
cur_period = cur_period + period > highest_period
else
_pub_thread_running
Calcula o proximo tempo que a thread deve acordar
cur_period começa em period e seu valor máximo é o valor do periodo mais alto registrado até o momento
cur_period também é inserido na mensagem para que o Subscriber possa selecionar apenas as mensagens que forem compativeis com seu período.
Obtem dado do transdutor
_transd->get_data(data);
Cria mensagem de tipo PUBLISH
Envia mensagem em Broadcast
std::this_thread::sleep_until (next_wakeup_t)


act [User View Publish]
Considere que outras entidades da pilha a partir de Protocol ja estao instanciadas no carro.
Ao instânciar um SmartData com template Communicator, Condition e Transducer, ele estará pronto para receber Subscribers da rede.
<<assignment>>
Communicator<...> comm =
Communicator<...>(...)
<<assignment>>
SmartData smd<...> =
SmartData< Communicator,
Condition, Transducer>


act [SmartData.update(...)]
Subscriber SmartData
Publisher SmartData
Observer::update(c, buf)
Apenas enfileira o Buffer
Obtem endereço de origem
e periodo da mensagem
Origem ja esta inscrita com o mesmo periodo
else
Adiciona a lista de Subscribers
Mdc de todos os
periodos inscritos
Recalcula periodo de envio
Libera Buffer


act [Fluxo dos buffers na pilha do protocolo]
Fila do SmartData tera apenas mensagens publish (Entrega 3)
Para o usuário receber as mensagens publish, utilizar SmartData.receive(data)
O usuário tem acesso apenas aos dados das mensagens de PUBLISH e mensagens de tipo COMMOM. SUBSCRIBEs são tratadas pela API.
SmartData trata subscribe
SmartData.update(c, buf)
SmartData enfileira Buffer
SmartData herda de Concurrent_Observer
Mensagem de tipo SUBSCRIBE
Mensagem de tipo PUBLISH
Para o usuário receber as mensagens comuns, utilizar Communicator.receive(msg)
SmartData identifica tipo da mensagem
Fila do Communicator tera apenas mensagens comuns (Entrega 1 e 2)
Communicator.notify(c, buf)
Communicator herda de Concurrent_Observed
Communicator notifica SmartData
Communicator.update(c, buf)
Communicator enfileira Buffer
Communicator herda de Concurrent_Observer
else
Mensagem de tipo COMMOM
SmartData apenas e notificado se:
1) A Unidade do dado for a que o usuario se inscreveu
2) O periodo de recebimento for compatível com o periodo que o usuário se inscreveu
Communicator identifica tipo da mensagem
else
Nenhum Communicator notificado
Protocol.notify(port, buf)
Caso o endereço seja compativel, Protocol notifica Communicator
else
Nenhum Protocolo notificado
NIC.notify(prot, buf)
Caso Prototype seja o mesmo do Protocolo, NIC notifica Protocolo
NIC e notificada e lê da Engine
Para esse fluxo, não é relevante o tipo da NIC