Departamento de
Informática e Estatística
CTC • UFSC

Implementação da API de Comunicação: Comunicação Segura em Grupos

INE5424 - Sistemas Operacionais II

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

Objetivos da entrega 5

- **Segurança de Mensagens:**
  - Implementar Message Authentication Code (MAC) para mensagens.
    - Algoritmo: Poly1305 (OpenSSL).
  - Descartar mensagens com falha na verificação do MAC.
- **Gerenciamento de Chaves MAC por RSUs (Roadside Units):**
  - RSUs geram e distribuem chaves MAC para veículos em sua vizinhança.
- **Sincronização Temporal PTP (Evolução da E4):**
  - Interação por demanda: Veículo inicia sincronização com a RSU de seu quadrante.
  - RSUs atuam como referência temporal para os veículos.
- **Coordenadas:**
  - Introdução de coordenadas (_coord_x, _coord_y) nas mensagens.
  - Filtragem de mensagens por proximidade (alcance de comunicação simulado).

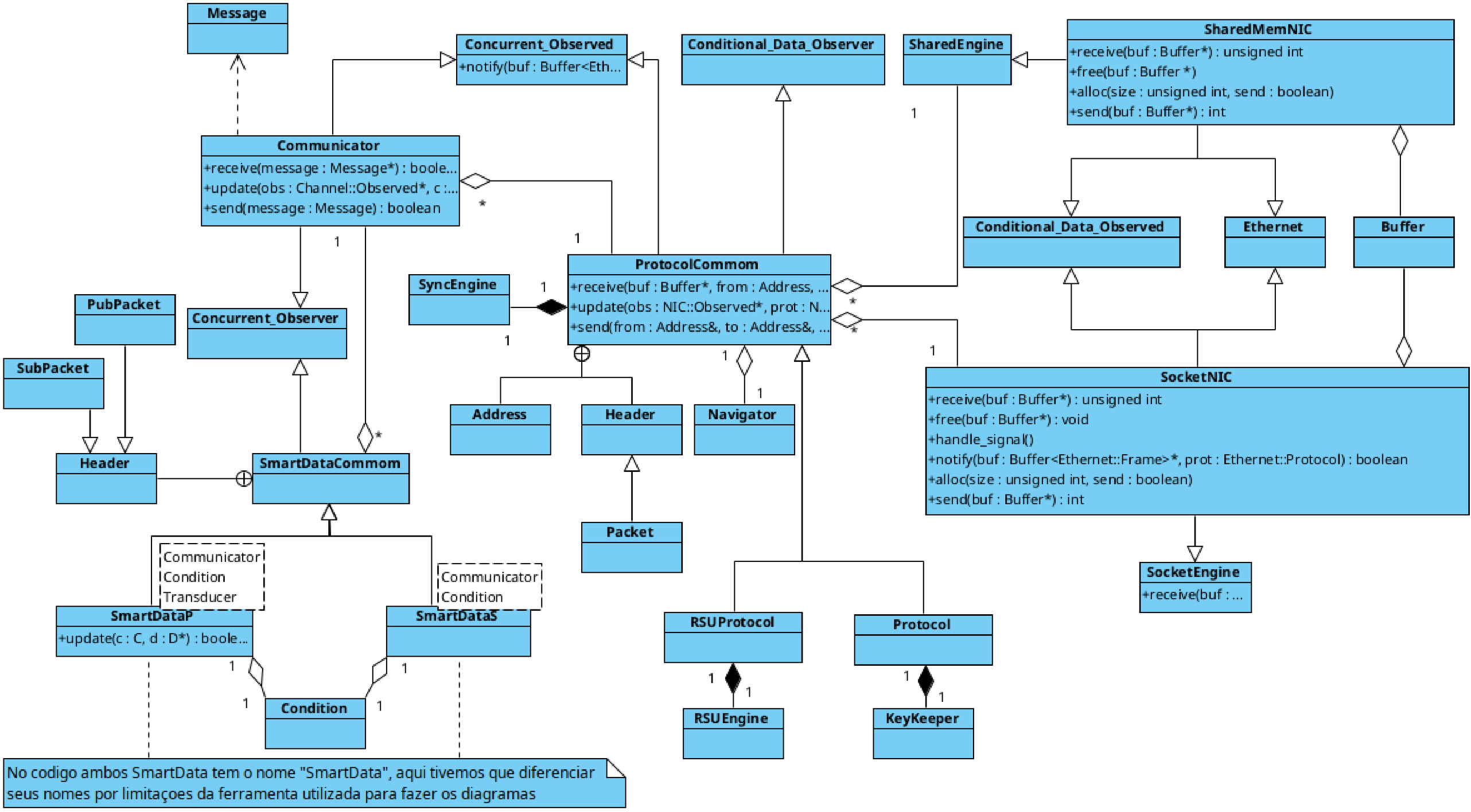
Introdução de Novas Classes e Entidades (1/2)

Componentes para Gerenciamento de Localização, Chaves e Papéis

- NavigatorCommon (e especializações NavigatorRandomWalk, NavigatorDirected):
  - Abstrai a lógica de movimentação e obtenção de coordenadas (_x, _y) para os veículos.
  - Utilizada por ProtocolCommom para preencher os campos _coord_x, _coord_y na Message.
  - Método is_in_range(Coordinate) usado em Protocol::update para filtrar mensagens de veículos fora do alcance de comunicação.
- Namespace MAC (mac.hh, mac.cc):
  - Encapsula a funcionalidade de criptografia MAC:
    - MAC::Key (std::array<std::byte, 32>)
    - MAC::Tag (std::array<std::byte, 16>)
    - MAC::compute(key, message): Gera a tag Poly1305.
    - MAC::verify(key, message, tag): Verifica a tag.
    - MAC::generate_random_key(): Gera uma chave aleatória.
- KeyKeeper (em key_keeper.hh, usado por Protocol - veículos):
  - Armazena um std::map<int, MAC::Key> (ID da RSU -> Chave MAC).
  - Método setKeys(const std::vector<MacKeyEntry>&): Populado por mensagens Control::Type::MAC recebidas das RSUs.
  - Método getKey(int rsu_id): Usado para obter a chave correta para calcular/verificar MACs.

Introdução de Novas Classes e Entidades (2/2)

- RSUs (Roadside Units):
  - Conceito introduzido, cada RSU responsável por um quadrante.
  - Implementadas como processos distintos com RSUProtocol.
  - RSUEngine<RSUProtocol> (em rsu_engine.hh):
    - Lógica específica das RSUs.
    - _key_sender_thread: Thread periódica para:
      - Gerar/Renovar sua própria MAC::Key.
      - A cada broadcast de mac, uma das RSUs incrementa um contador(em memória compartilhada entre processos) que determina quando todas devem gerar novas chaves MAC. (Sincronizações necessárias são implementadas utilizando pthread_barrier_t e pthread_mutex_t)
      - Obter chaves das RSUs adjacentes (vizinhança 3x3) da SharedData->entries.
      - Enviar estas chaves em broadcast (Control::Type::MAC).

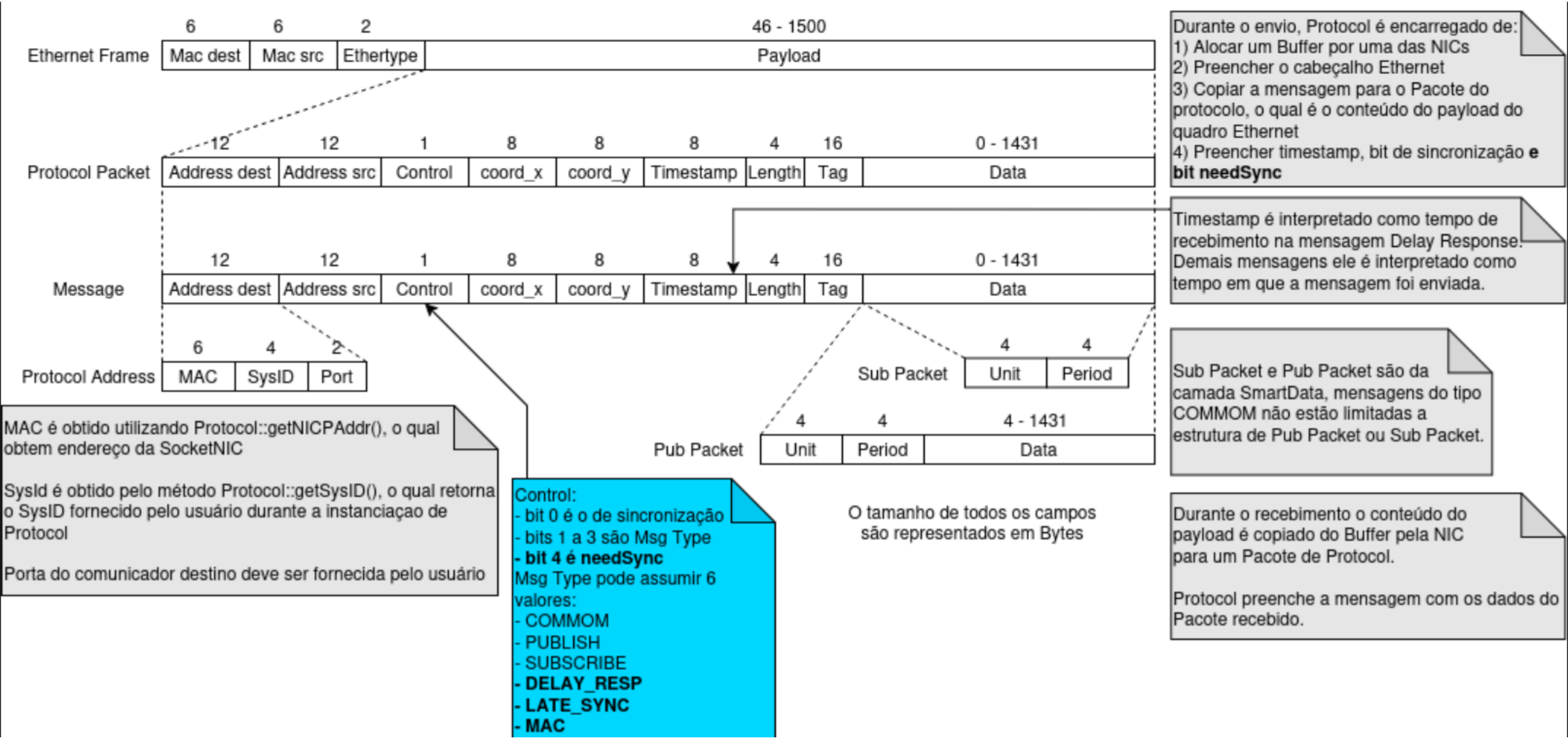

Message
Concurrent_Observed
+notify(buf : Buffer<Eth...
Conditional_Data_Observer
SharedEngine
SharedMemNIC
+receive(buf : Buffer*) : unsigned int
+free(buf : Buffer *)
+alloc(size : unsigned int, send : boolean)
+send(buf : Buffer*) : int
Communicator
+receive(message : Message*) : boole...
+update(obs : Channel::Observed*, c :...
+send(message : Message) : boolean
Conditional_Data_Observed
Ethernet
Buffer
ProtocolCommom
+receive(buf : Buffer*, from : Address, ...
+update(obs : NIC::Observed*, prot : N...
+send(from : Address&, to : Address&, ...
SyncEngine
PubPacket
Concurrent_Observer
SubPacket
Address
Header
Navigator
Header
SmartDataCommom
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 : ...
Communicator
Condition
Transducer
Communicator
Condition
SmartDataP
+update(c : C, d : D*) : boole...
SmartDataS
RSUProtocol
Protocol
Condition
RSUEngine
KeyKeeper
No codigo ambos SmartData tem o nome "SmartData", aqui tivemos que diferenciar seus nomes por limitaçoes da ferramenta utilizada para fazer os diagramas

Modificações na Estrutura Message e Protocol::Packet (1/2)

- **Novos Campos em Message<Addr> (e consequentemente em Protocol::Packet::Header):**
  - MAC::Tag _tag;:
    - Armazena a tag MAC (16 bytes para Poly1305) calculada sobre a mensagem.
    - Acessado via Message<Addr>::tag().
  - double _coord_x, double _coord_y:
    - Armazenam as coordenadas geográficas do emissor da mensagem.
    - Acessadas via Message<Addr>::getCoordX() e Message<Addr>::getCoordY().
    - Preenchidas em ProtocolCommom::fillBuffer() utilizando a instância de NavigatorCommon.
- **Preenchimento e Verificação do MAC:**
  - Envio (em Protocol::fillBuffer para veículos):
    - Após preencher todos os outros campos da Protocol::Packet (incluindo _timestamp, _ctrl, coordenadas).
    - Obtém a MAC::Key do quadrante que o veículo que vai enviar a mensagem se localiza.
    - Calcula a MAC::Tag usando MAC::compute(key, message_bytes).
    - A tag calculada é inserida em pkt->header()->tag.

Modificações na Estrutura Message e Protocol::Packet (2/2)

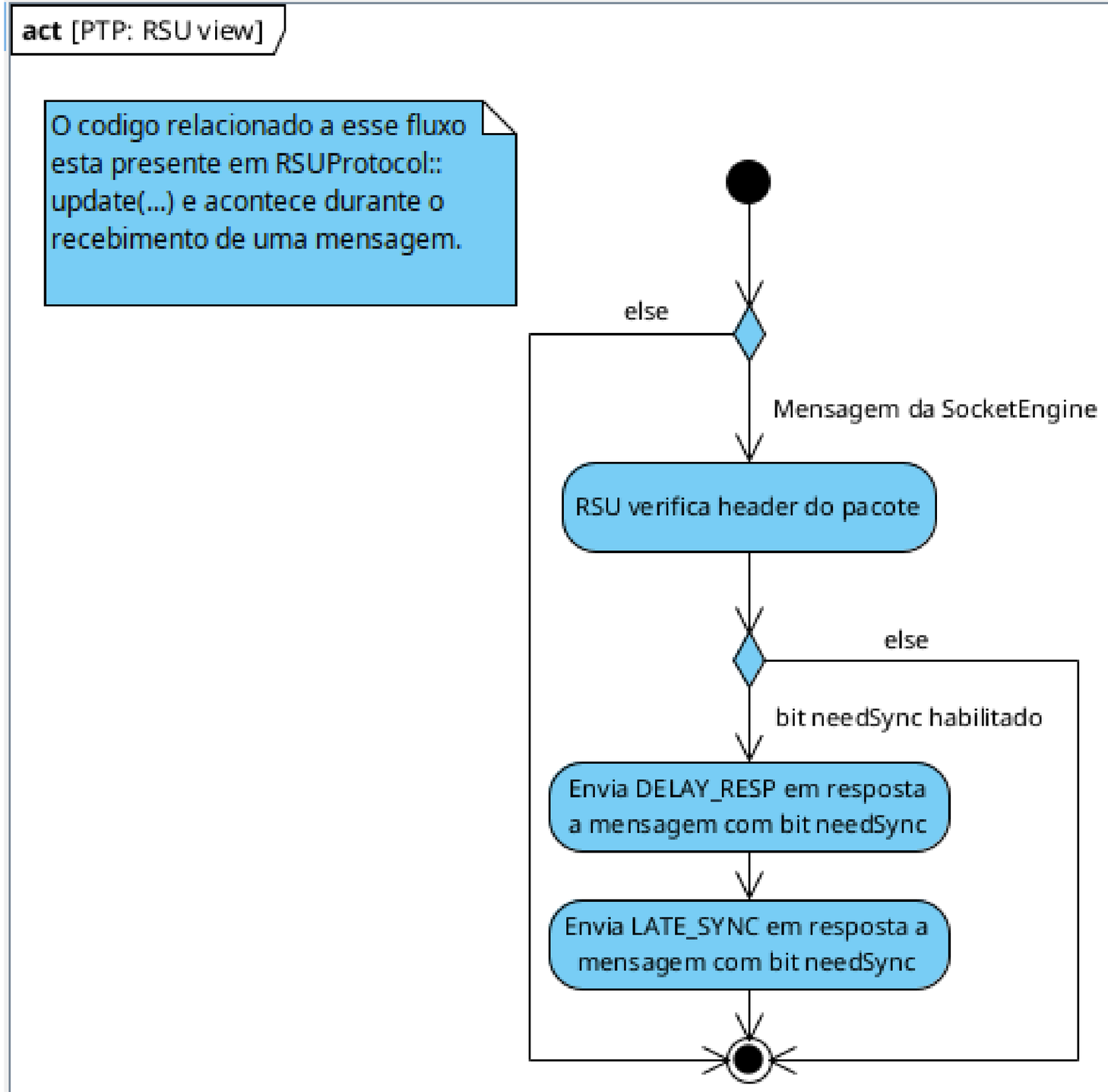
- **Preenchimento e Verificação do MAC:**
  - Recebimento (em Protocol::update para veículos):
    - Extrai a tag recebida de pkt->header()->tag.
    - Recalcula a tag localmente usando a chave apropriada e os bytes da mensagem recebida (excluindo a própria tag).
    - Verifica usando MAC::verify(key, message_bytes, received_tag).
    - Se a verificação falhar, a mensagem é descartada.
- **Campo Control em Message:**
  - Tipo PTP foi substituído por tipo DELAY_RESP e LATE_SYNC.
  - Adicionado bit **needSync**
  - Adicionado Control::Type::MAC para mensagens específicas de distribuição de chaves MAC pelas RSUs.

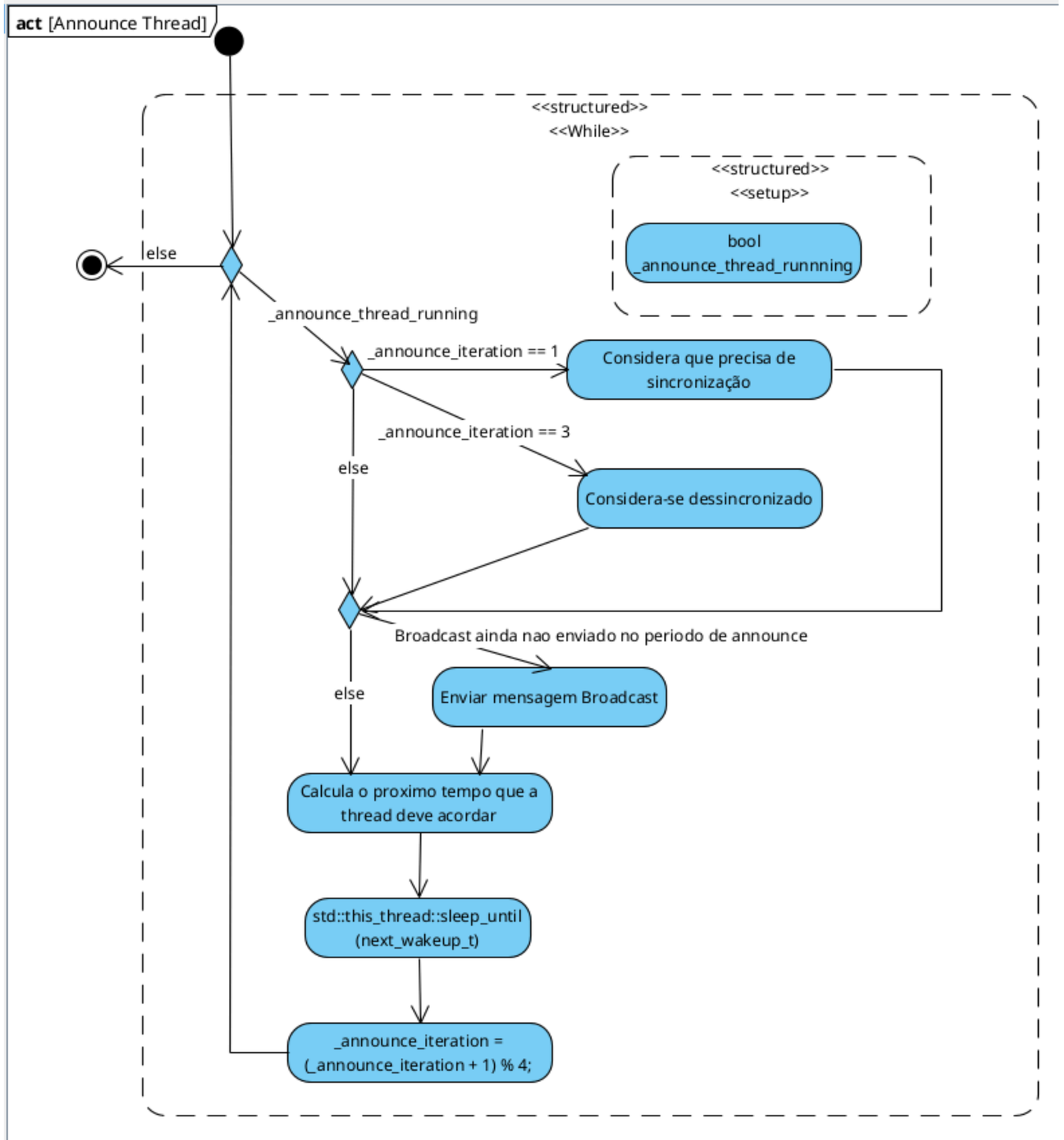

6
6
2
46 - 1500
Ethernet Frame
Mac dest
Mac src
Ethertype
Payload
12
12
1
8
8
8
4
16
0 - 1431
Protocol Packet
Address dest
Address src
Control
coord_x
coord_y
Timestamp
Length
Tag
Data
Message
6
4
2
Protocol Address
MAC
SysID
Port
4
4
Sub Packet
Unit
Period
4
4
4 - 1431
Pub Packet
Unit
Period
Data
Durante 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
4) Preencher timestamp, bit de sincronização e bit needSync
Timestamp é interpretado como tempo de recebimento na mensagem Delay Response. Demais mensagens ele é interpretado como tempo em que a mensagem foi enviada.
Sub Packet e Pub Packet são da camada SmartData, mensagens do tipo COMMOM não estão limitadas a estrutura de Pub Packet ou Sub Packet.
MAC é obtido utilizando Protocol::getNICPAddr(), o qual obtem endereço da SocketNIC
SysId é obtido pelo método Protocol::getSysID(), o qual retorna o SysID fornecido pelo usuário durante a instanciaçao de Protocol
Porta do comunicador destino deve ser fornecida pelo usuário
Control:
- bit 0 é o de sincronização
- bits 1 a 3 são Msg Type
- bit 4 é needSync
Msg Type pode assumir 6 valores:
- COMMOM
- PUBLISH
- SUBSCRIBE
- DELAY_RESP
- LATE_SYNC
- MAC
O tamanho de todos os campos são representados em Bytes
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.

Sincronização Temporal (PTP) por Demanda com RSUs

Evolução do Modelo PTP da Entrega 4

- **RSUs atuam como mestres PTP para veículos.**
- **Veículos (Slaves PTP):**
  - _announce_thread (SyncEngine) envia ANNOUNCE periodicamente.
  - Se ANNOUNCE do veículo indica !needSync() (verificado pela RSU):
    - RSU (RSUProtocol::update) responde com:
      - Control::Type::DELAY_RESP (Header: T4 da RSU).
      - Control::Type::LATE_SYNC (Header: T1 da RSU).
      - Mensagens contém em seu payload a qual ´needSync´ela está relacionada
    - Veículo (SyncEngine::handlePTP) processa DELAY_RESP e LATE_SYNC:
      - DELAY_RESP: Armazena T4 mapeado para T3 em unordered_map;.
      - LATE_SYNC: Usa T1 (da msg), T2 (local), T3 (payload) e T4 (do map) para calcular offset.
      - Ao fim do processo, atualiza offset e considera-se sincronizado;


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;

Hierarquia de Protocol

Especialização para Veículos e RSUs

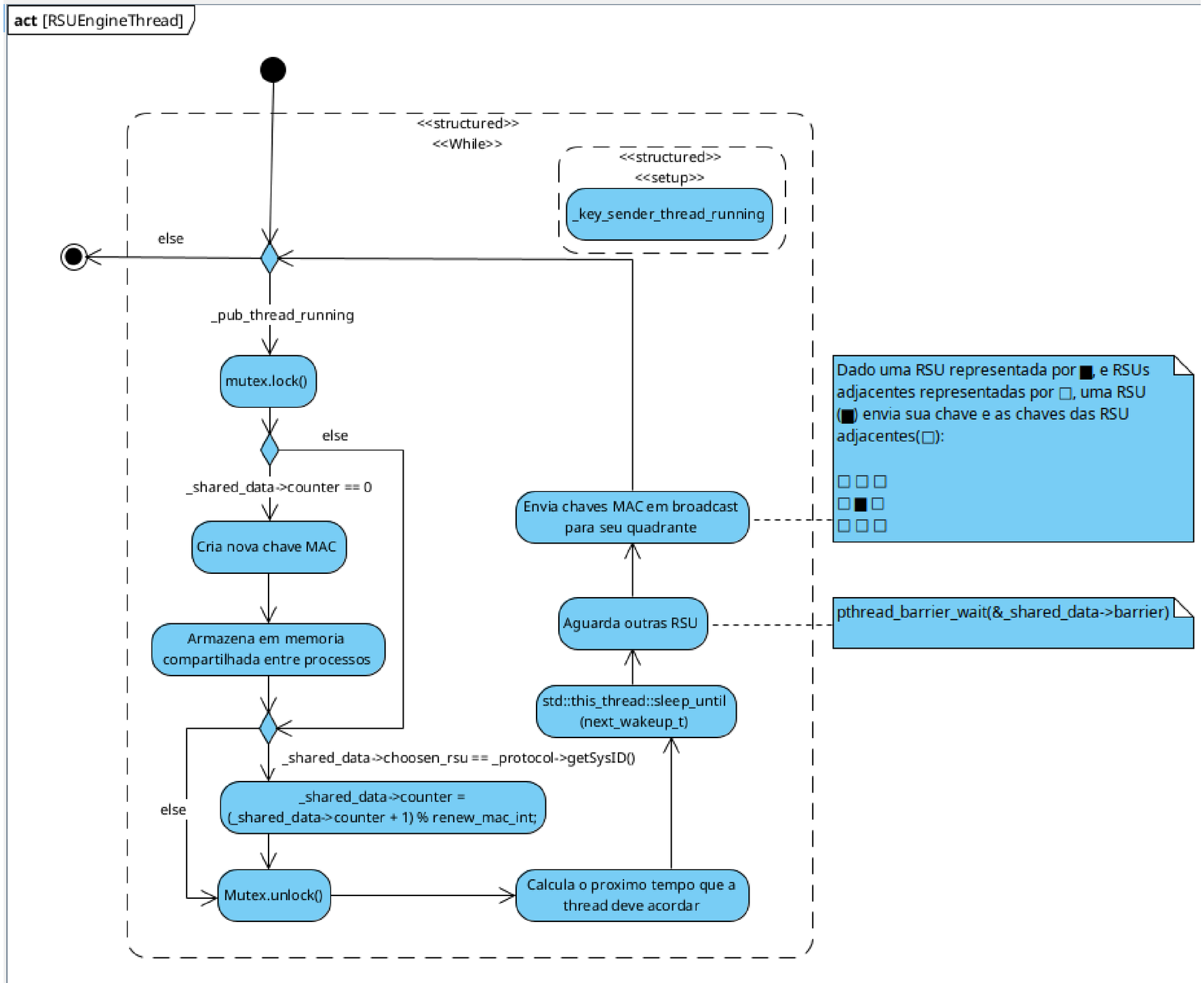
- **ProtocolCommom<SocketNIC, SharedMemNIC> (Base):**
  - Funcionalidades Comuns: Estruturas Address, Header, Packet (com _coord_x, _coord_y, _tag). Interação com SyncEngine (com flag isRSU) e NavigatorCommon. Métodos base send, receive.
  - Método update(...) é virtual, forçando especialização.
- **Protocol<SocketNIC, SharedMemNIC> (Veículos):**
  - Herda de ProtocolCommom.
  - update(...): Filtra por _nav->is_in_range(); verifica MAC com _key_keeper; processa Control::Type::MAC para _key_keeper.setKeys(); delega PTP à _sync_engine.
  - fillBuffer(): Adiciona cálculo de MAC (via MAC::compute).
  - Atributo KeyKeeper para armazenamento de chaves MAC.
- **RSUProtocol<SocketNIC, SharedMemNIC> (RSUs):**
  - Herda de ProtocolCommom.
  - Atributo RSUEngine para lógica de chaves MAC.
  - update(): Atua como mestre PTP por demanda. Se recebe mensagem de veículo não sincronizado, envia DELAY_RESP e LATE_SYNC. Ignora outras mensagens PTP e MAC.

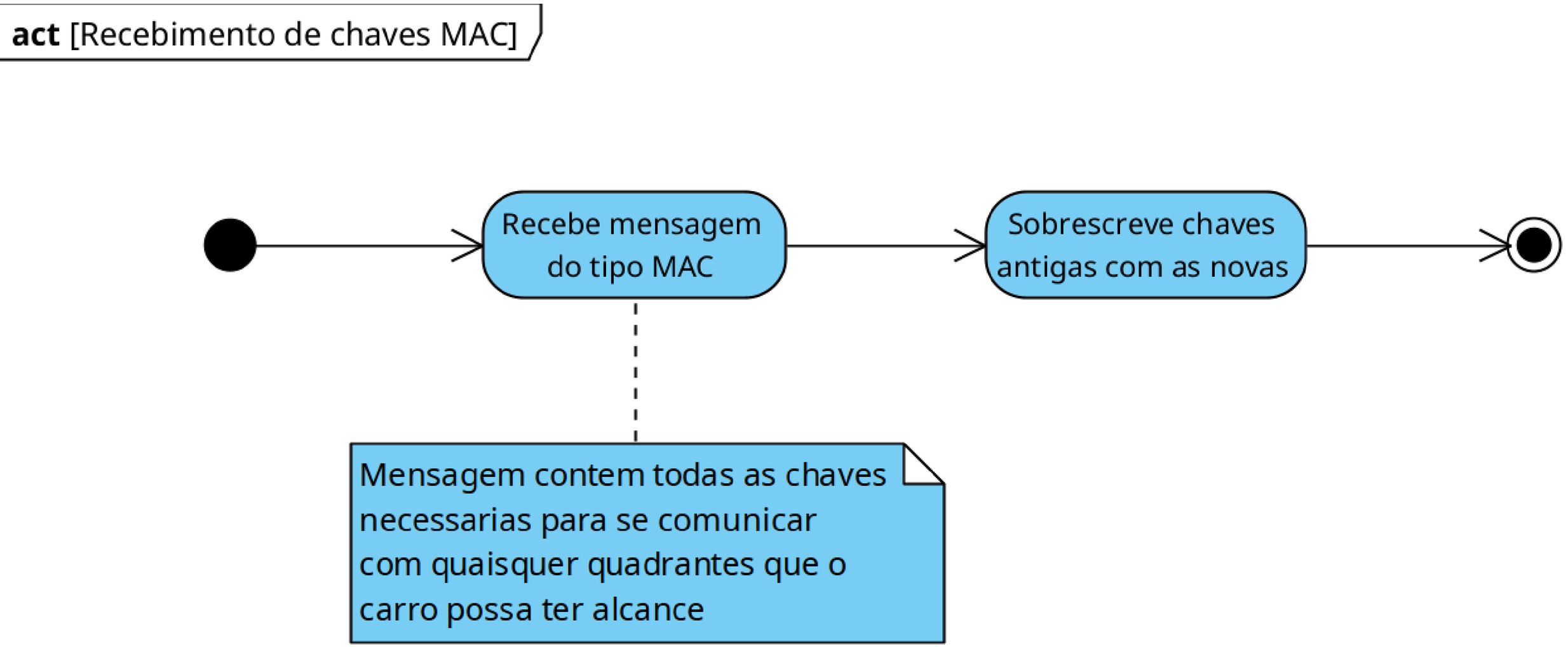
Distribuição de Chaves MAC

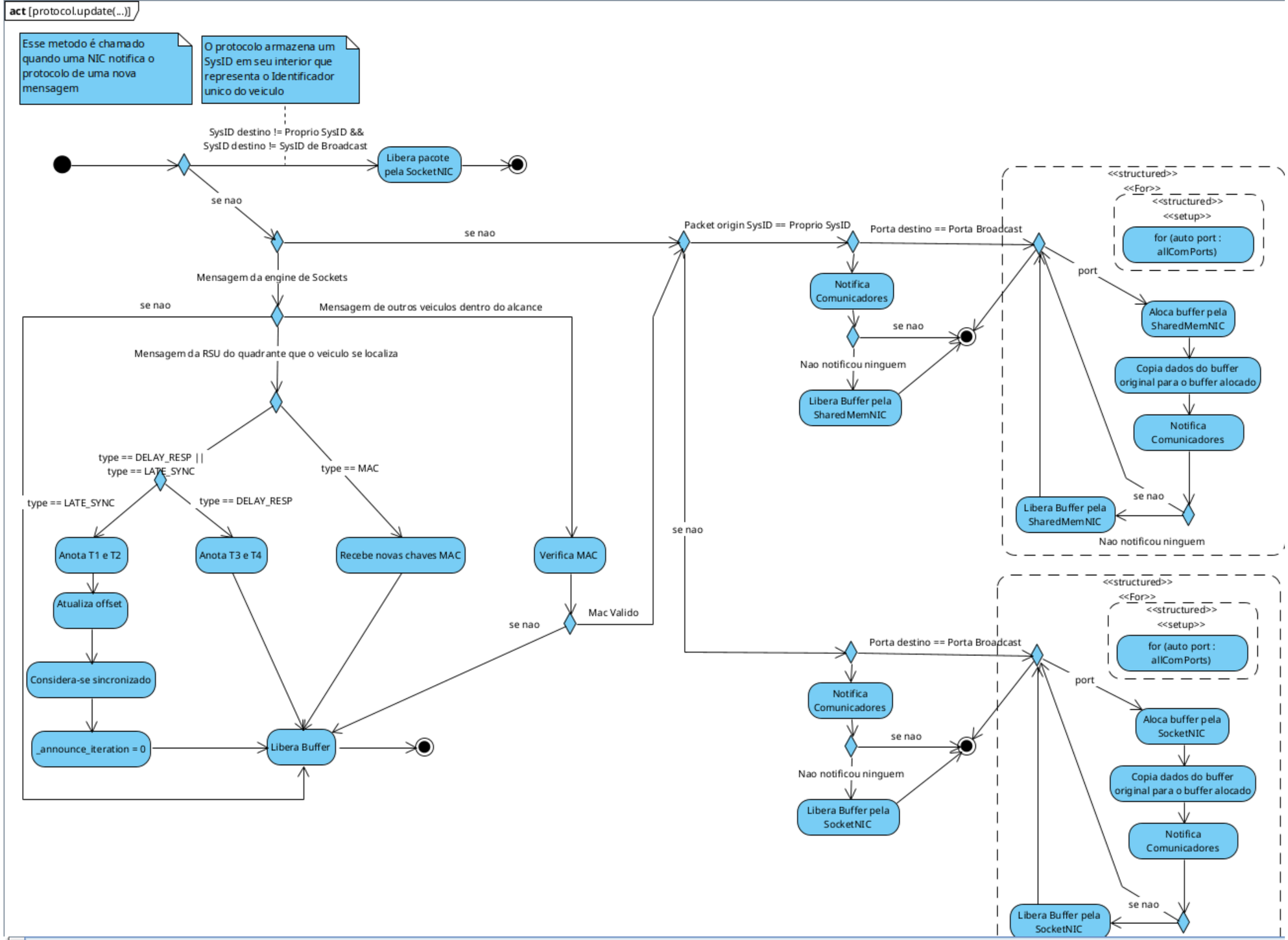
- **RSUs (RSUEngine): Geração e Distribuição de Chaves**
  - _key_sender_thread (periódica):
    - Coordena com outras RSUs (via SharedData, pthread_barrier_t, pthread_mutex_t) para que uma RSU designada (_shared_data->choosen_rsu) atualize o contador _shared_data->counter.
    - Se _shared_data->counter == 0, a RSU atualiza sua MacKeyEntry em _shared_data->entries com MAC::generate_random_key().
    - Lê chaves da vizinhança (3x3) de _shared_data->entries.
    - Envia neighborhood_keys em Message com Control::Type::MAC (broadcast).
- **Veículos (Protocol): Recebimento e Armazenamento**
  - Protocol::update(): Ao receber Control::Type::MAC, usa _key_keeper.setKeys(...) com as chaves recebidas.

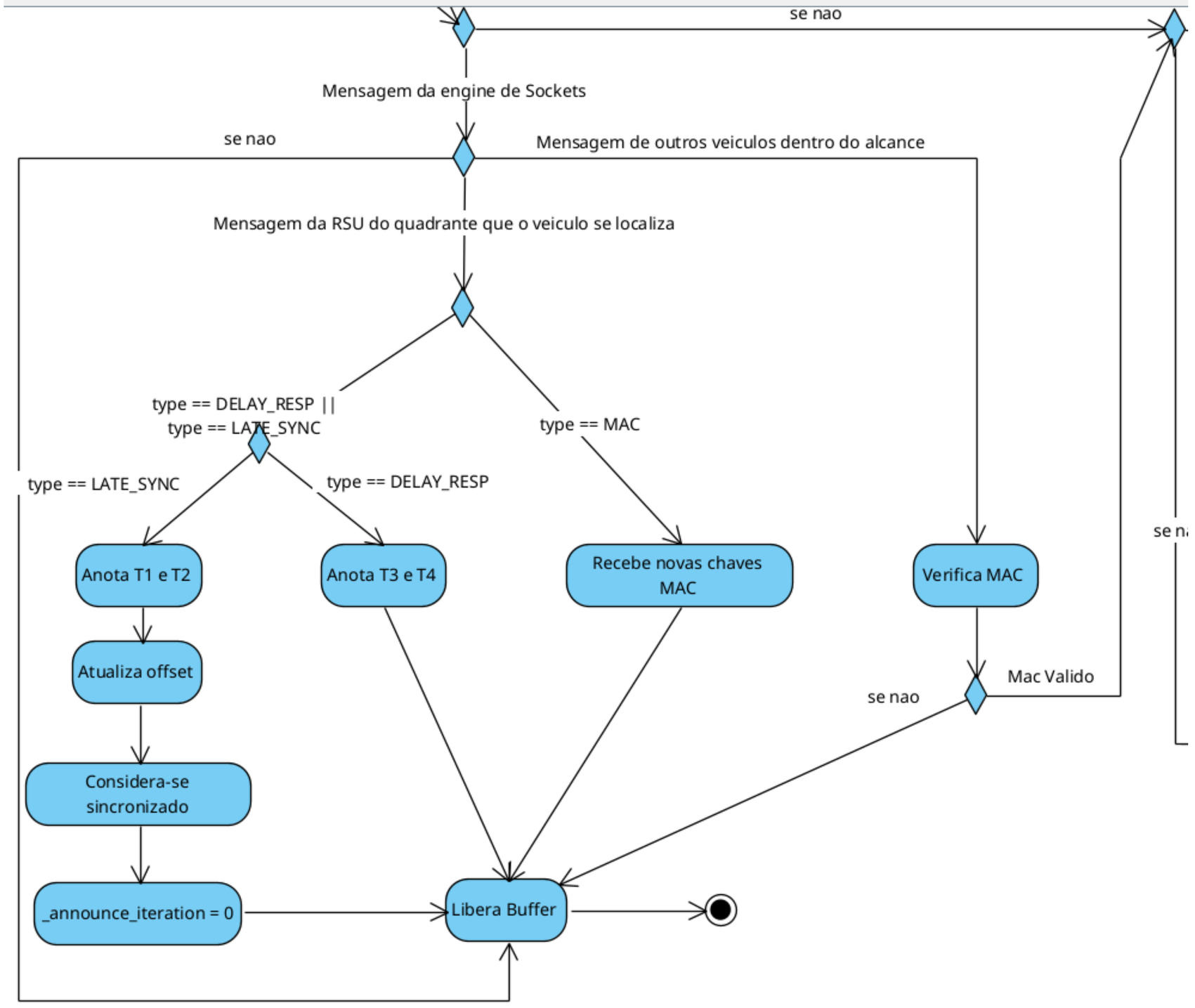
Uso de Chaves MAC

- **Veículos (Protocol::fillBuffer): Cálculo de MAC no Envio**
  - Obtém a MAC::Key do quadrante que o veículo que vai enviar a mensagem se localiza.
  - Calcula tag = MAC::compute(key, serialized_packet_bytes).
  - Preenche tag no cabeçalho da mensagem.
- **Veículos (Protocol::update): Verificação de MAC no Recebimento**
  - A partir da localização do veículo que enviou a mensagem, escolhe uma de suas chaves MAC para verificar a mensagem.
  - Verifica com MAC::verify(key, serialized_packet_bytes, pkt->header()->tag).
  - Descarta mensagem se MAC não for válido.


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

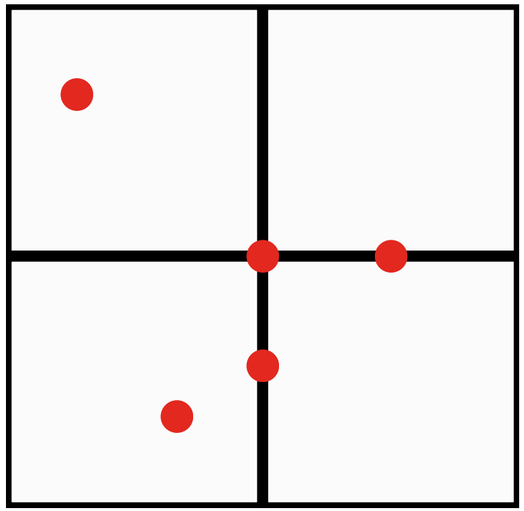

se nao
Mensagem da engine de Sockets
se nao
Mensagem de outros veiculos dentro do alcance
Mensagem da RSU do quadrante que o veiculo se localiza
type == DELAY_RESP ||
type == LATE_SYNC
type == MAC
type == LATE_SYNC
type == DELAY_RESP
Anota T1 e T2
Anota T3 e T4
Recebe novas chaves MAC
Verifica MAC
Atualiza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

Testes Desenvolvidos e Validação

- Teste de quadrante

Carros

Range da RSU



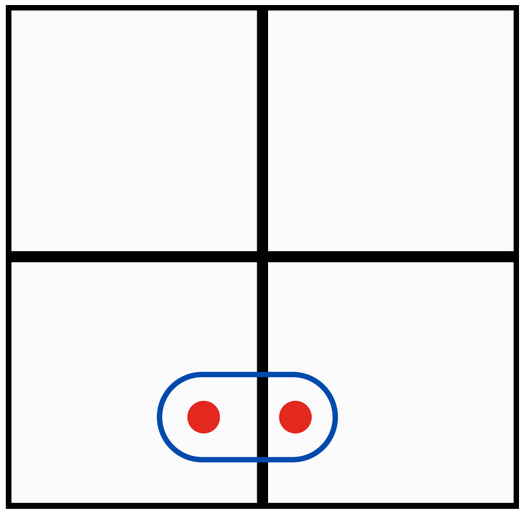
Verifica em que quadrante está o ponto e verifica o que acontece se estiver em intersecções

Testes Desenvolvidos e Validação

- Teste de comunicação entre carros em diferentes quadrantes

Carros

Range da RSU



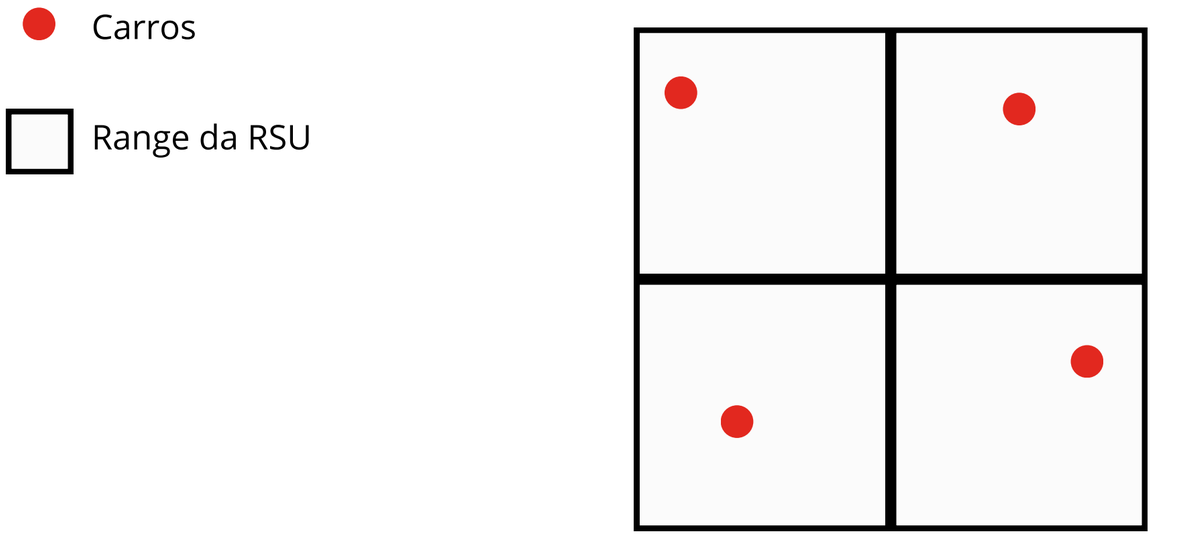
O que está sendo testado:

- Comunicação inter-quadrante
- Funcionamento de múltiplas RSUs
- Correto recebimento das mensagens na topologia

Testes Desenvolvidos e Validação

- Teste out of range

Este teste valida o comportamento do sistema quando veículos estão fora do alcance de comunicação direta, testando as limitações de conectividade. Resultado esperado é timeout por falha de comunicação.



Não deve haver comunicação entre esses carros

Testes Desenvolvidos e Validação

- Teste de validação do MAC

Teste que passa por todos os métodos de MAC para verificação de seu funcionamento

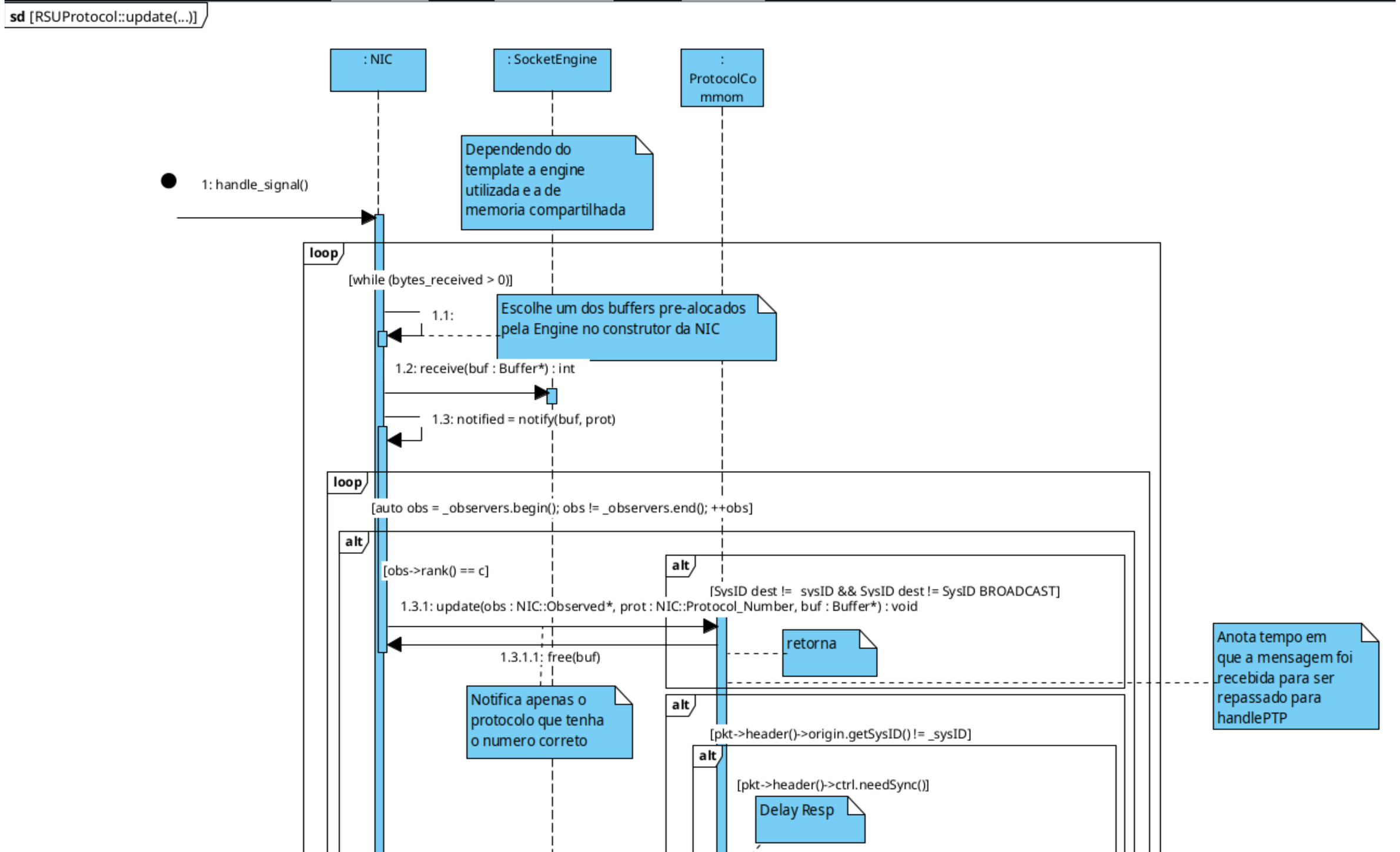
- Testa todos os métodos públicos da classe MAC
- Cobre cenários positivos e negativos
- Valida integridade e autenticidade
- Verifica detecção de ataques/corrupção

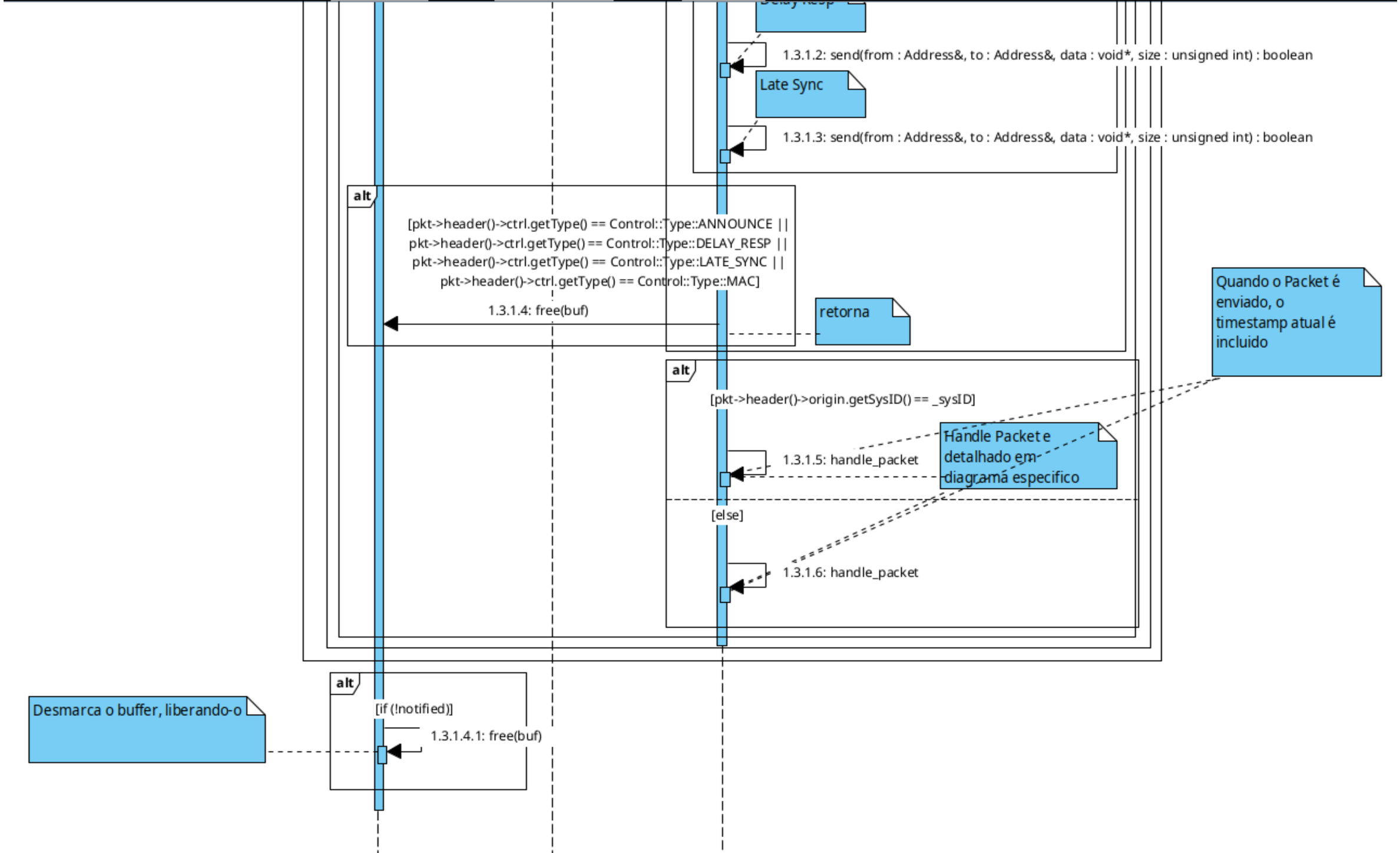
- Teste de Descarte de Mensagens

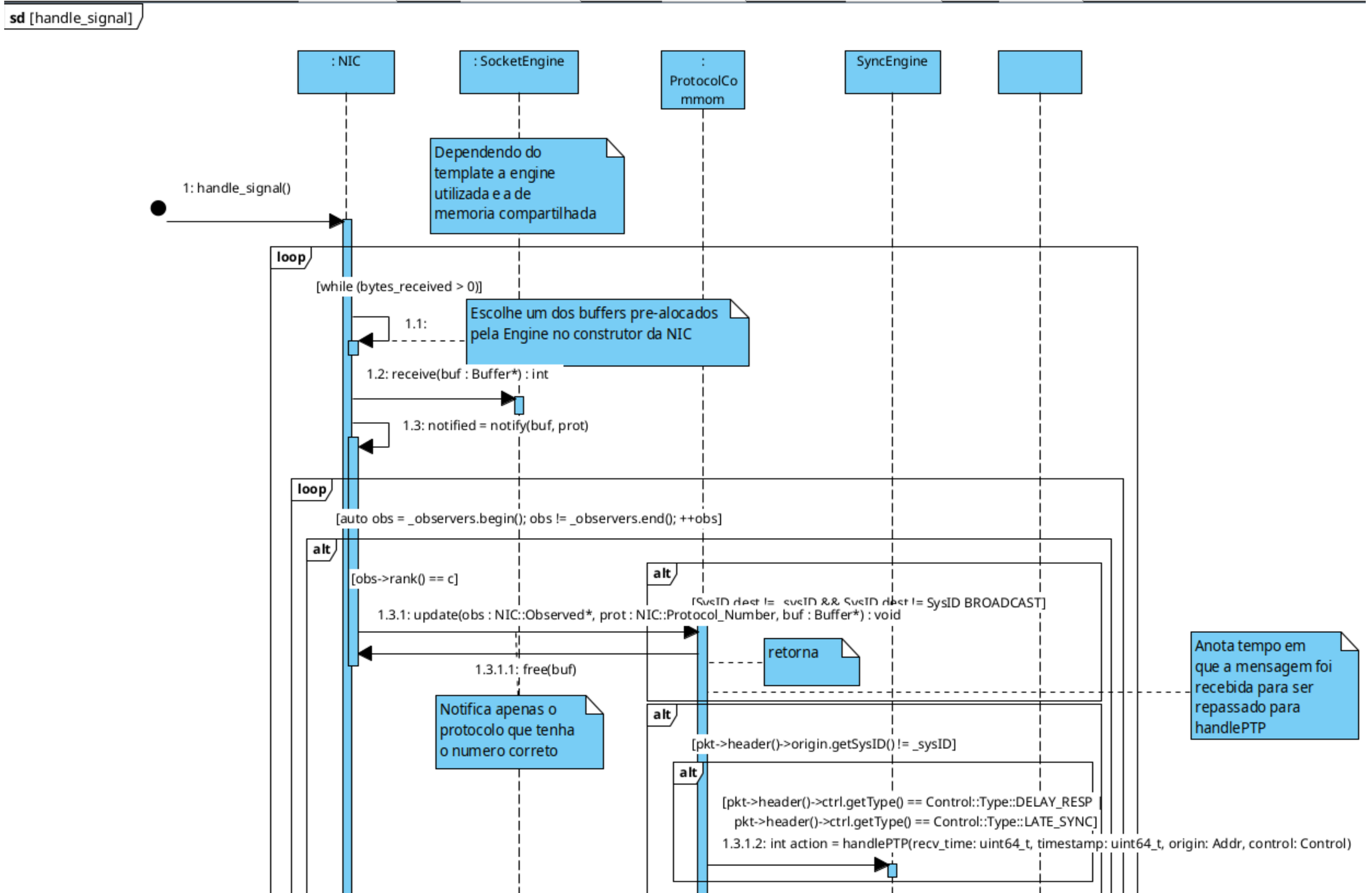
Teste que valida o descarte automático de mensagens com MAC inválido em comunicação V2V unicast, garantindo que apenas mensagens autenticadas sejam processadas pelo receptor.
Objetivo: testar se o sistema automaticamente filtra/descarta mensagens que:

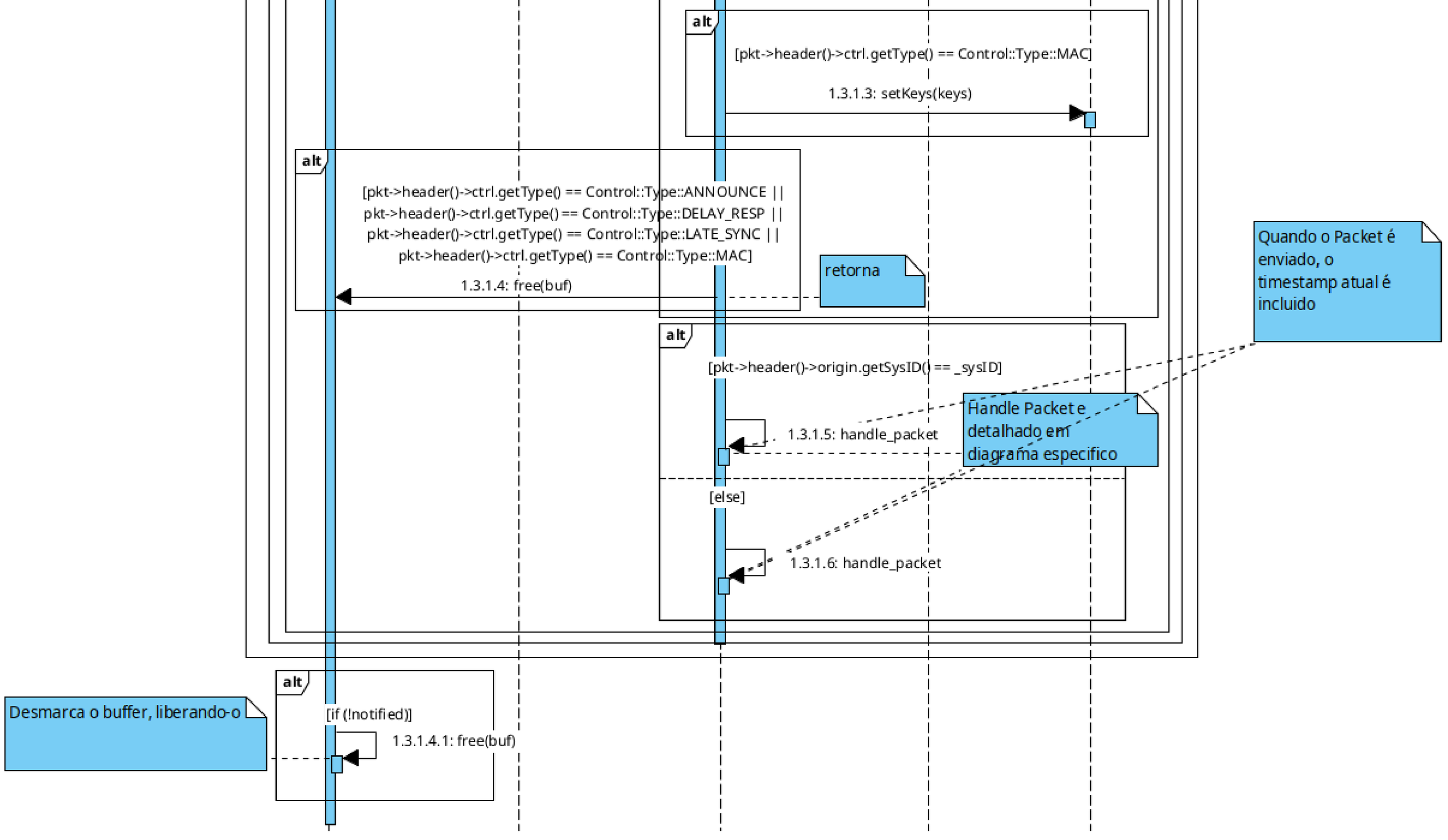
- Não passaram na validação MAC
- Foram corrompidas durante transmissão
- Vieram de fontes não autenticadas

Diagramas Novos ou Atualizados:

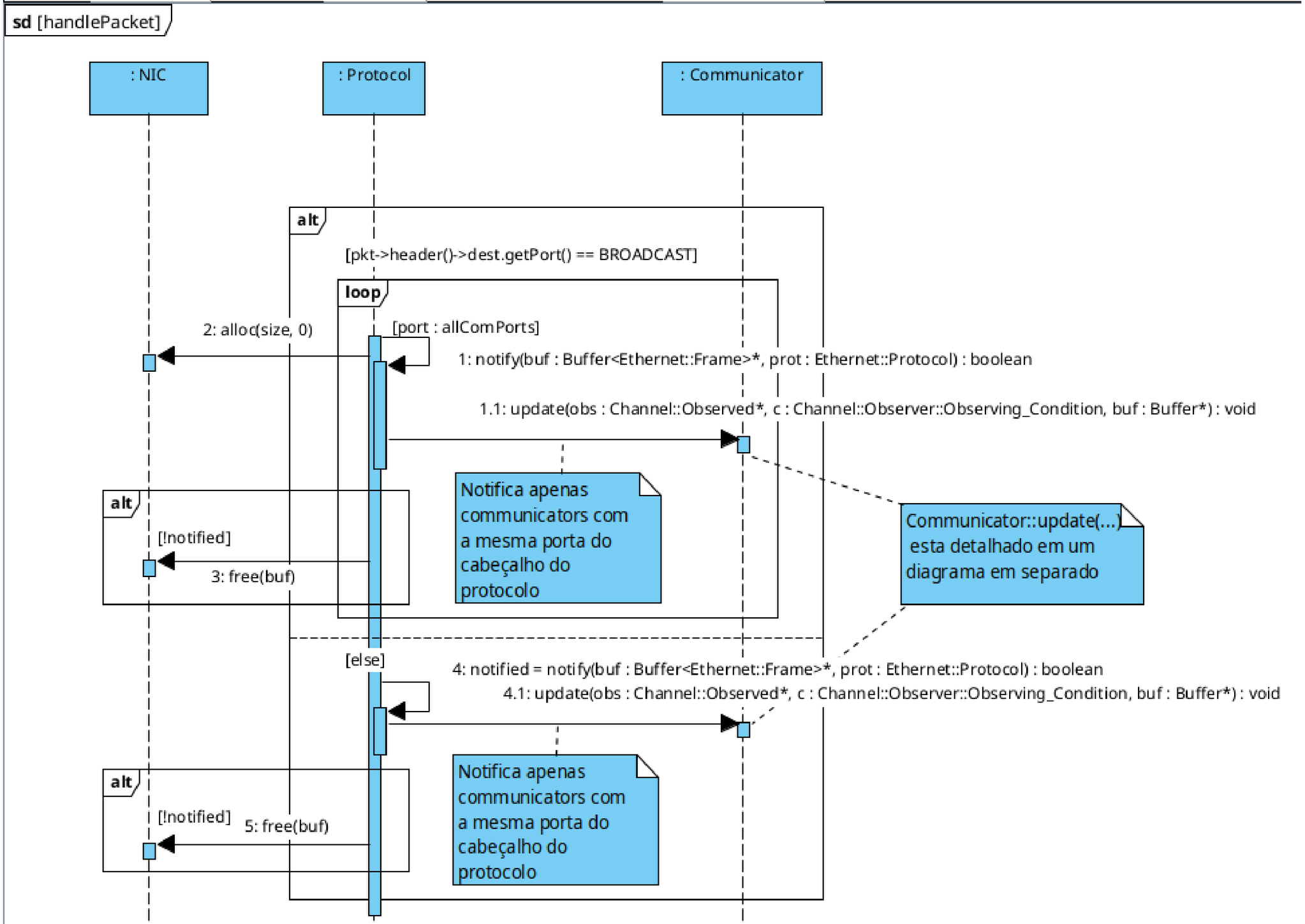

sd [RSUProtocol::update(...)]
: NIC
: SocketEngine
: ProtocolCommom
Dependendo do template a engine utilizada e a de memoria compartilhada
1: handle_signal()
loop
[while (bytes_received > 0)]
1.1:
Escolhe um dos buffers pre-alocados pela Engine no construtor da NIC
1.2: receive(buf : Buffer*) : int
1.3: 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


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

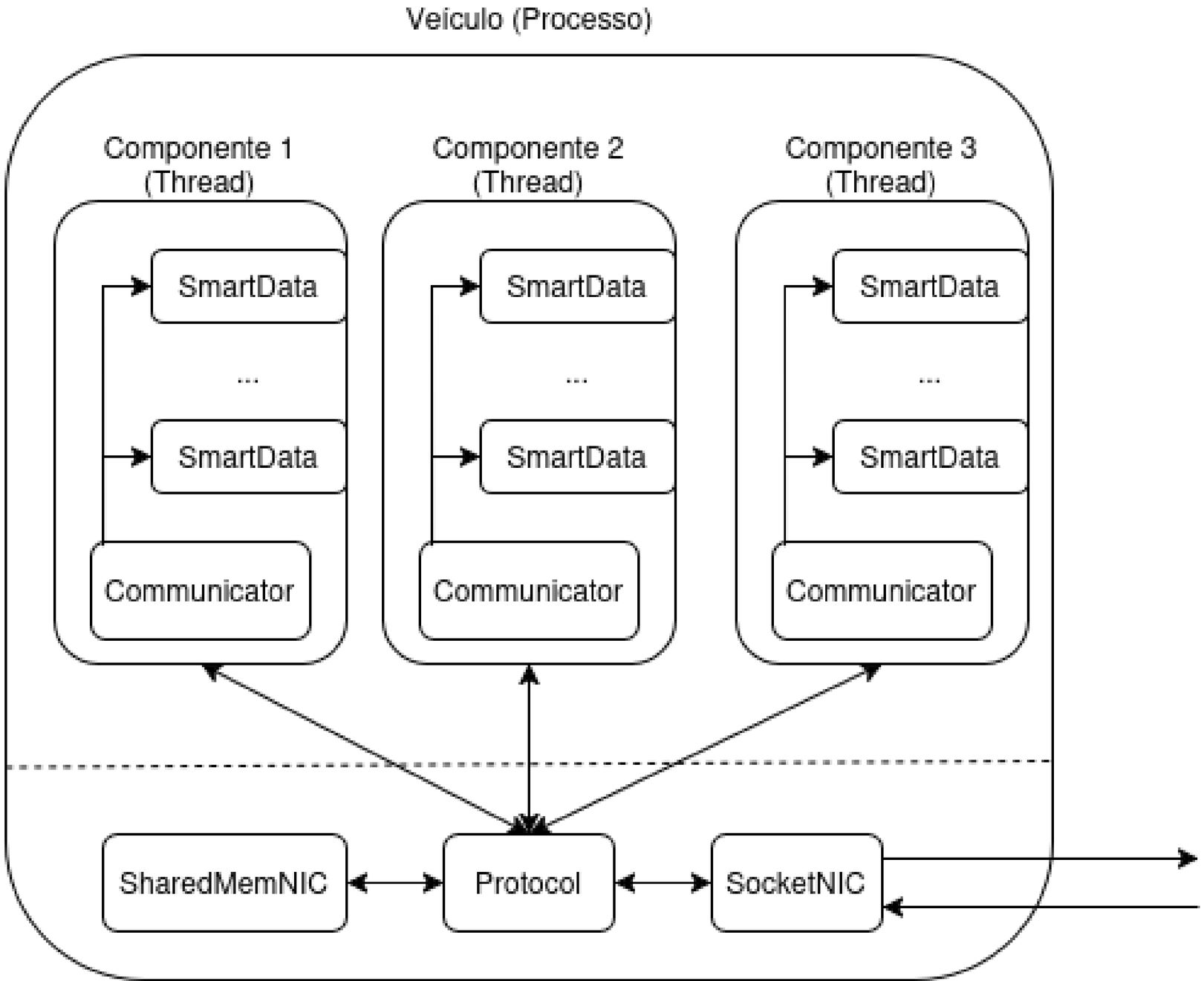

sd [handle_signal]
: NIC
: SocketEngine
: ProtocolCommom
SyncEngine
Dependendo do template a engine utilizada e a de memoria compartilhada
1: handle_signal()
loop
[while (bytes_received > 0)]
1.1:
Escolhe um dos buffers pre-alocados pela Engine no construtor da NIC
1.2: receive(buf : Buffer*) : int
1.3: 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)


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)

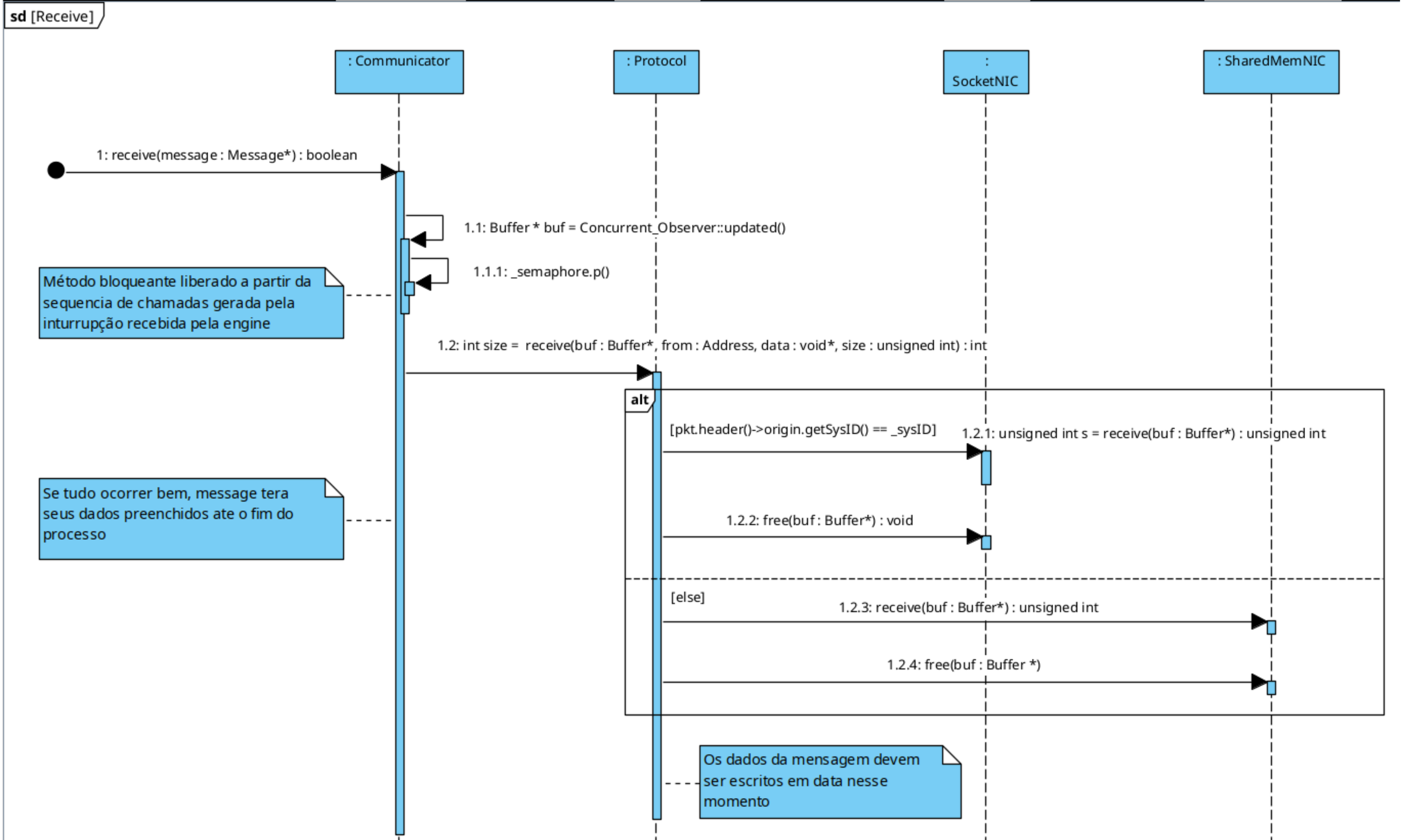
Outros Diagramas:

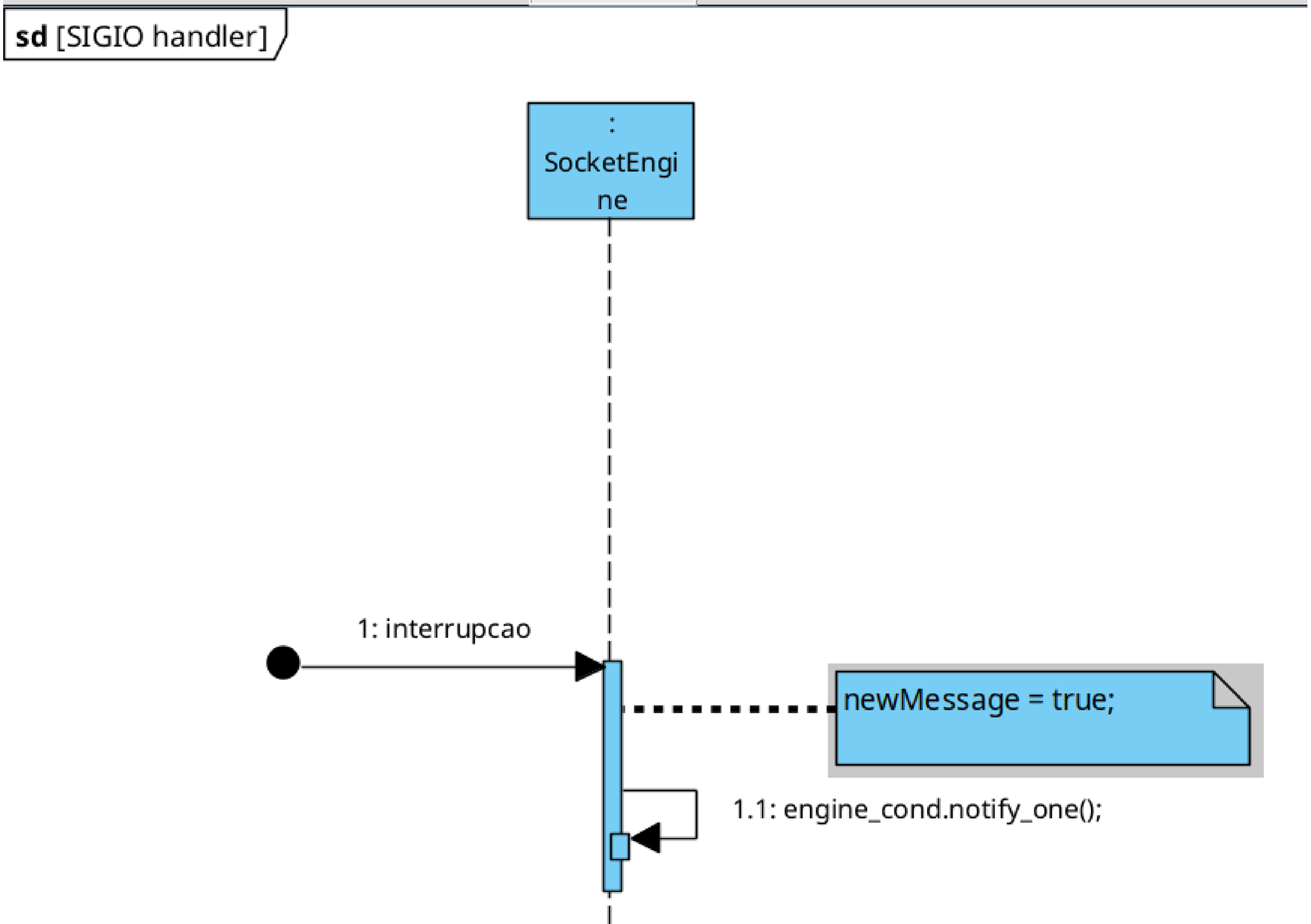

sd [handlePacket]
: NIC
: Protocol
: Communicator
alt
[pkt->header()->dest.getPort() == BROADCAST]
loop
2: alloc(size, 0)
[port : allComPorts]
1: notify(buf : Buffer<Ethernet::Frame>*, prot : Ethernet::Protocol) : boolean
1.1: update(obs : Channel::Observed*, c : Channel::Observer::Observing_Condition, 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)

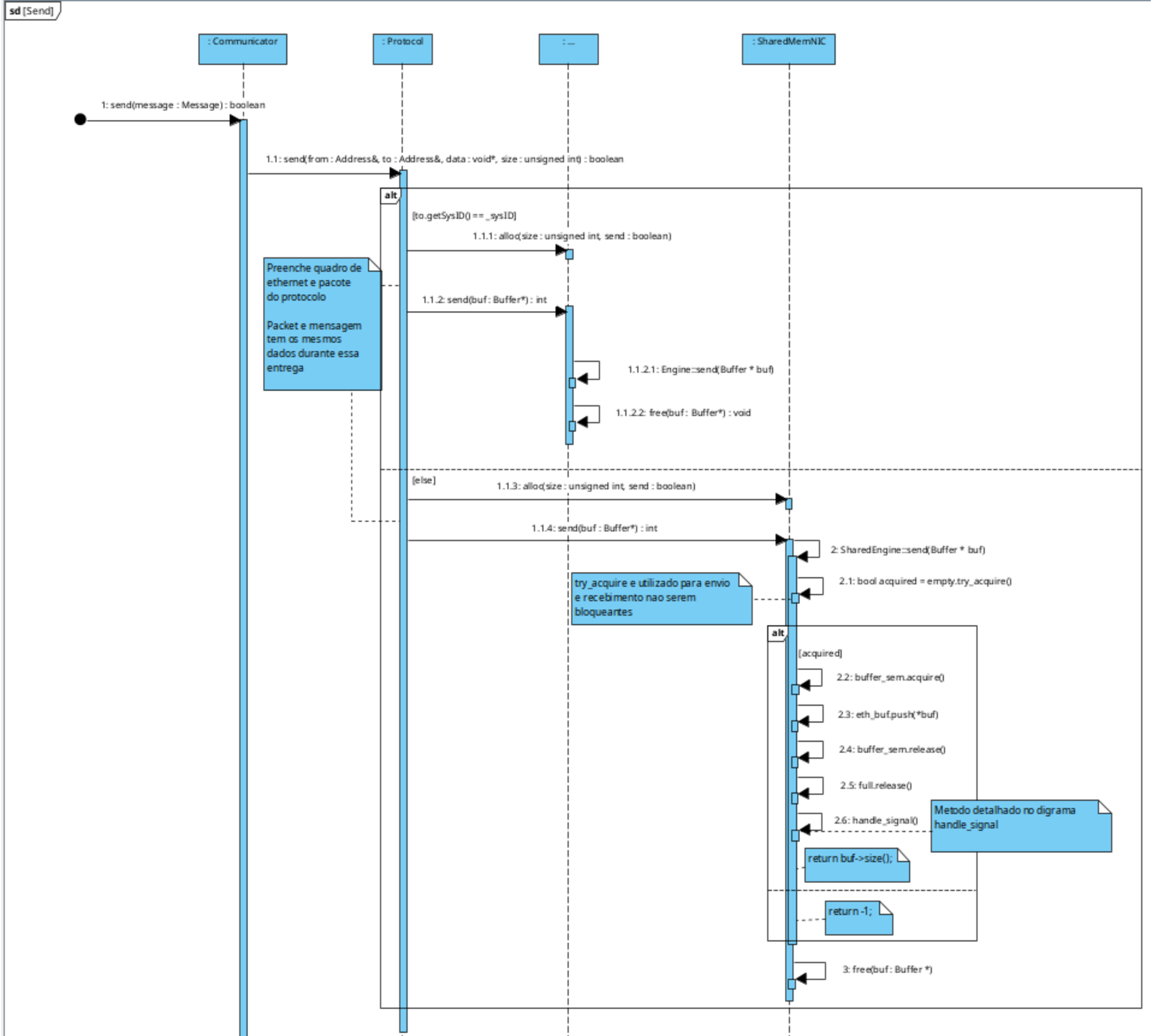
Estrutura do Veículo

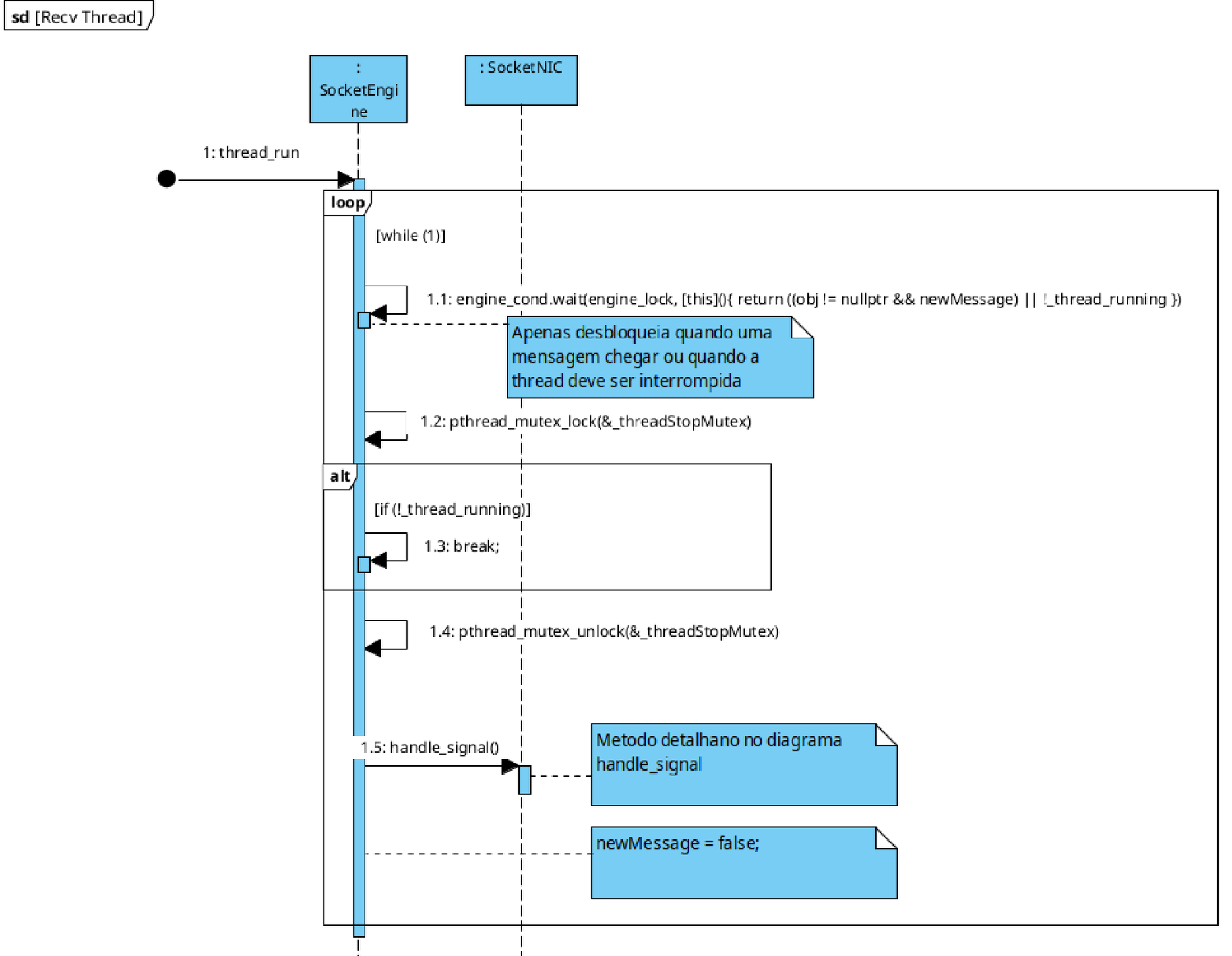


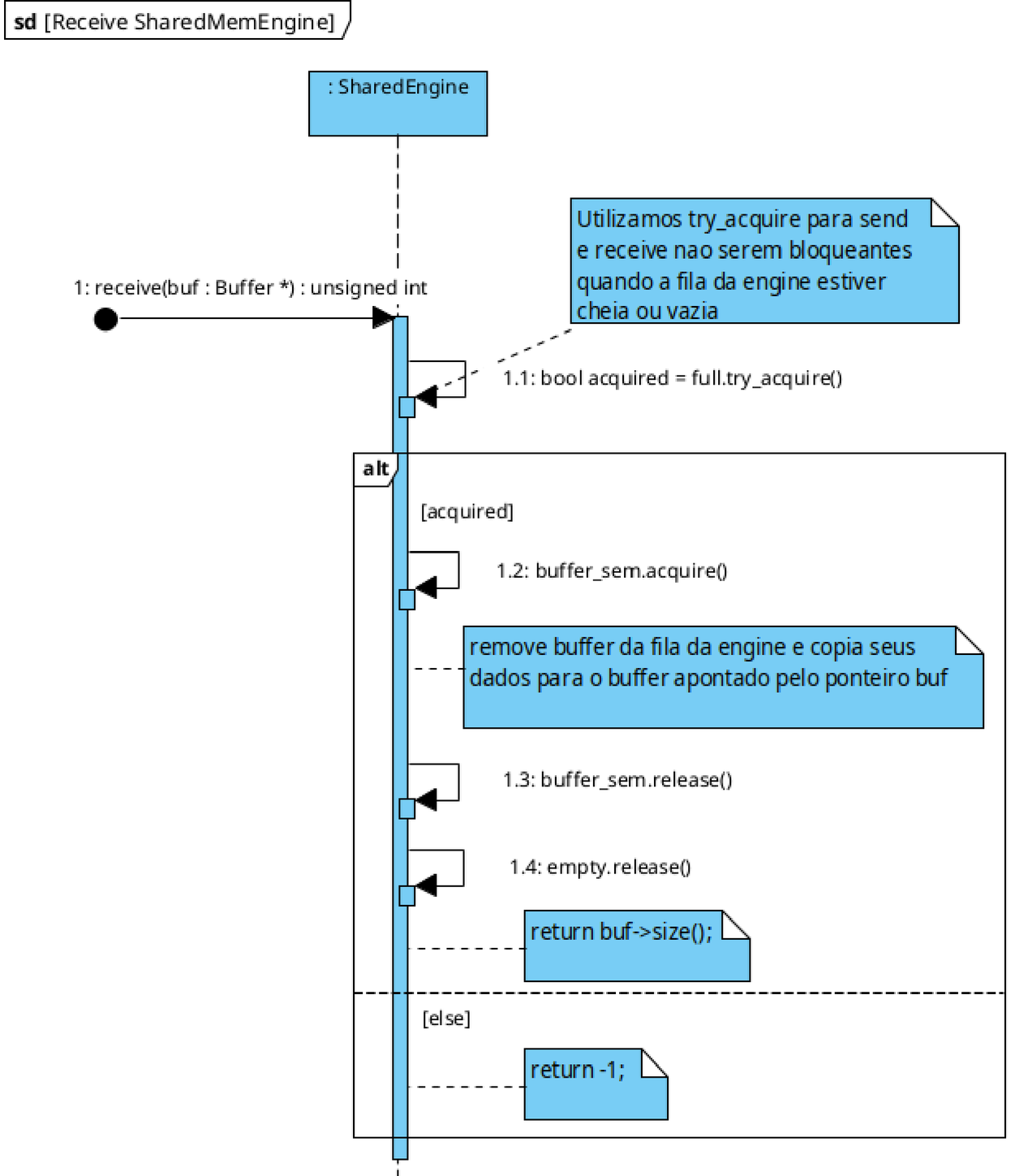
Diagramas de Sequência

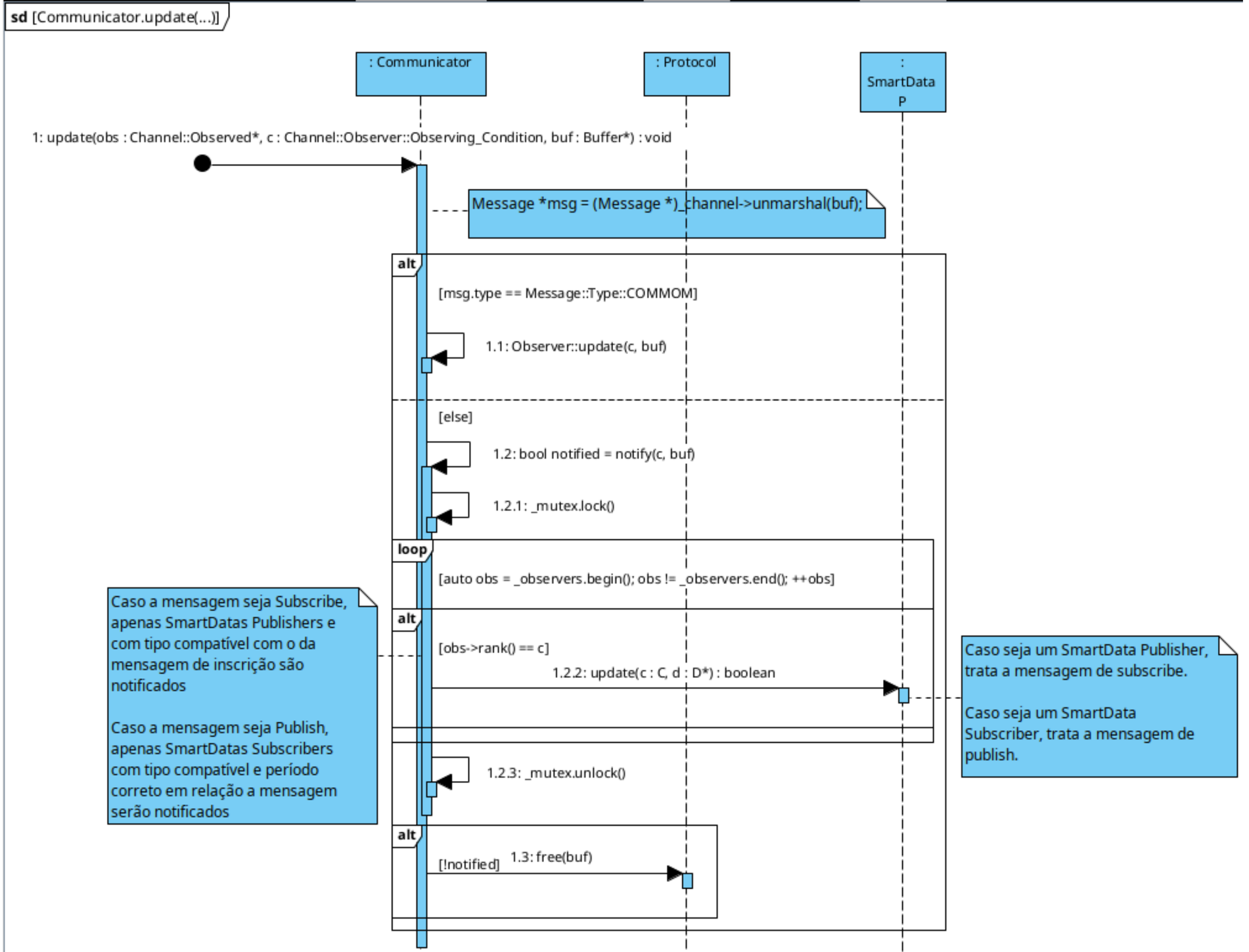

sd [Receive]
: Communicator
: Protocol
: SocketNIC
: SharedMemNIC
1: receive(message : Message*) : boolean
1.1: Buffer * buf = Concurrent_Observer::updated()
1.1.1: _semaphore.p()
Método bloqueante liberado a partir da sequencia de chamadas gerada pela inturrupção recebida pela engine
1.2: int size = 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();

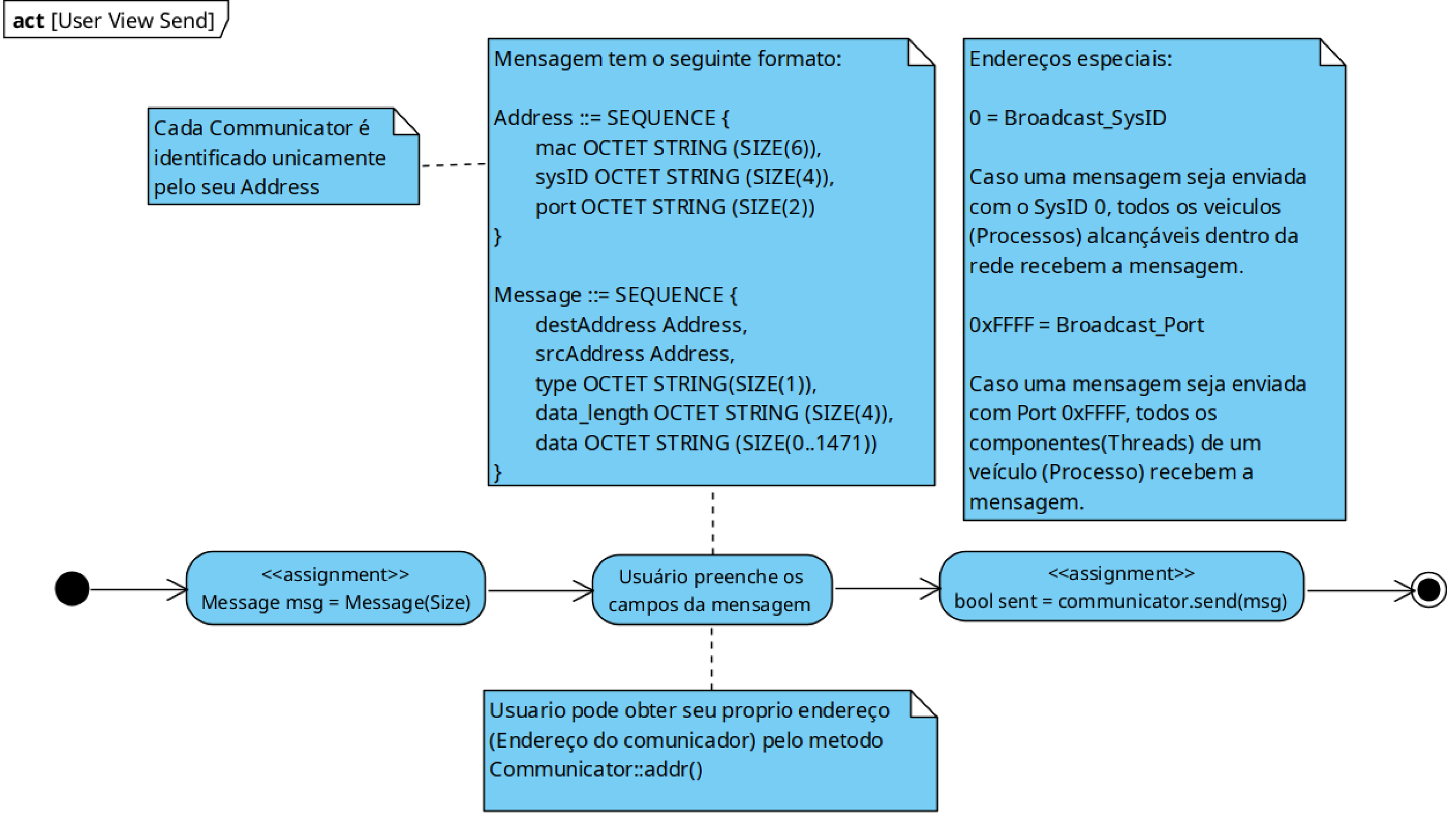

sd [Send]
: Communicator
: Protocol
: ...
: SharedMemNIC
1: send(message : Message) : boolean
1.1: send(from : Address&, to : Address&, data : void*, size : unsigned int) : boolean
alt
[to.getSysID() == _sysID]
1.1.1: alloc(size : unsigned int, send : boolean)
Preenche quadro de ethernet e pacote do 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 *)

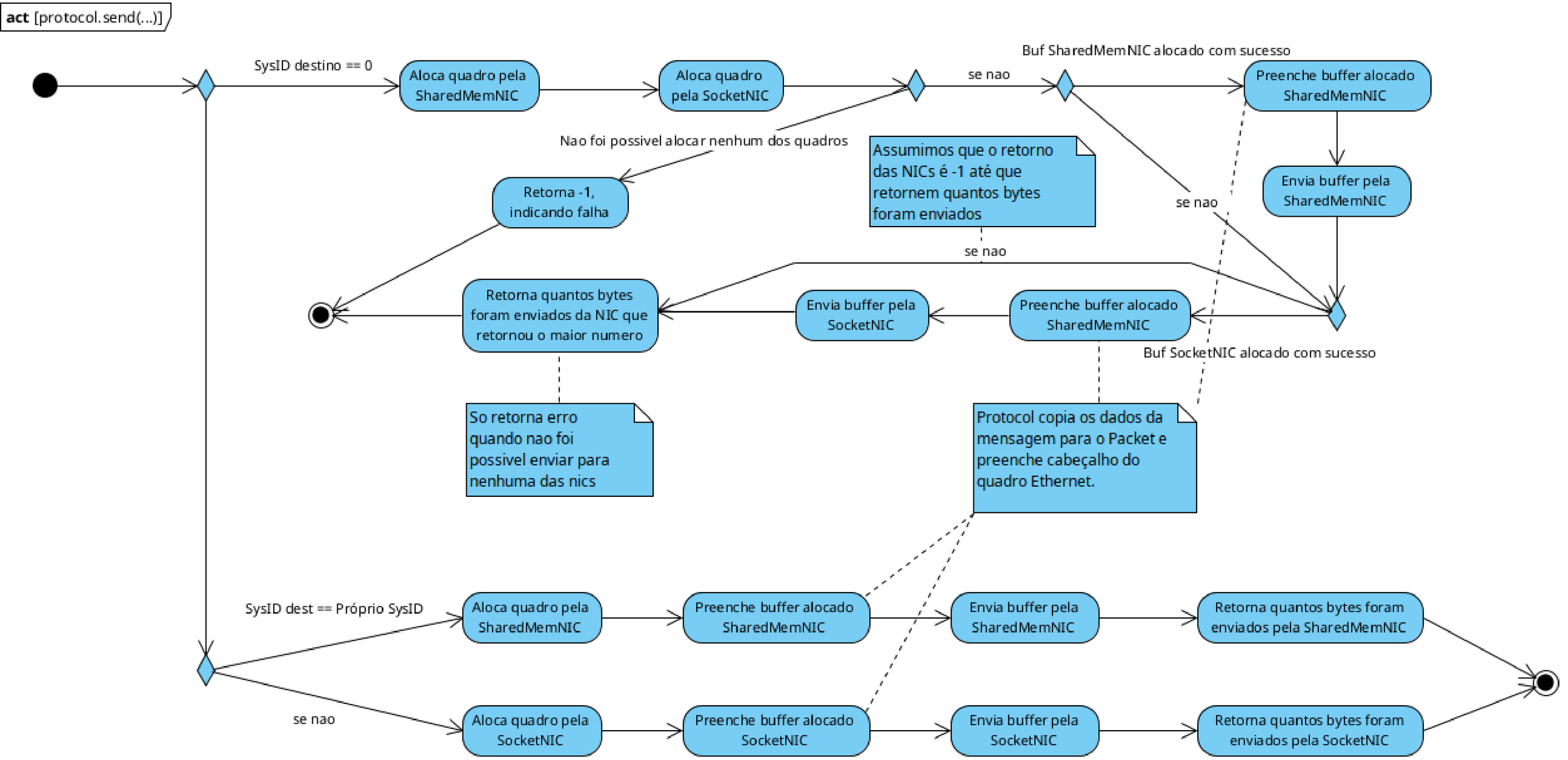

sd [Recv Thread]
: SocketEngine
: SocketNIC
1: thread_run
loop
[while (1)]
1.1: engine_cond.wait(engine_lock, [this](){ return ((obj != nullptr && newMessage) || !_thread_running })
Apenas desbloqueia quando uma mensagem chegar ou quando a thread deve ser interrompida
1.2: pthread_mutex_lock(&_threadStopMutex)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

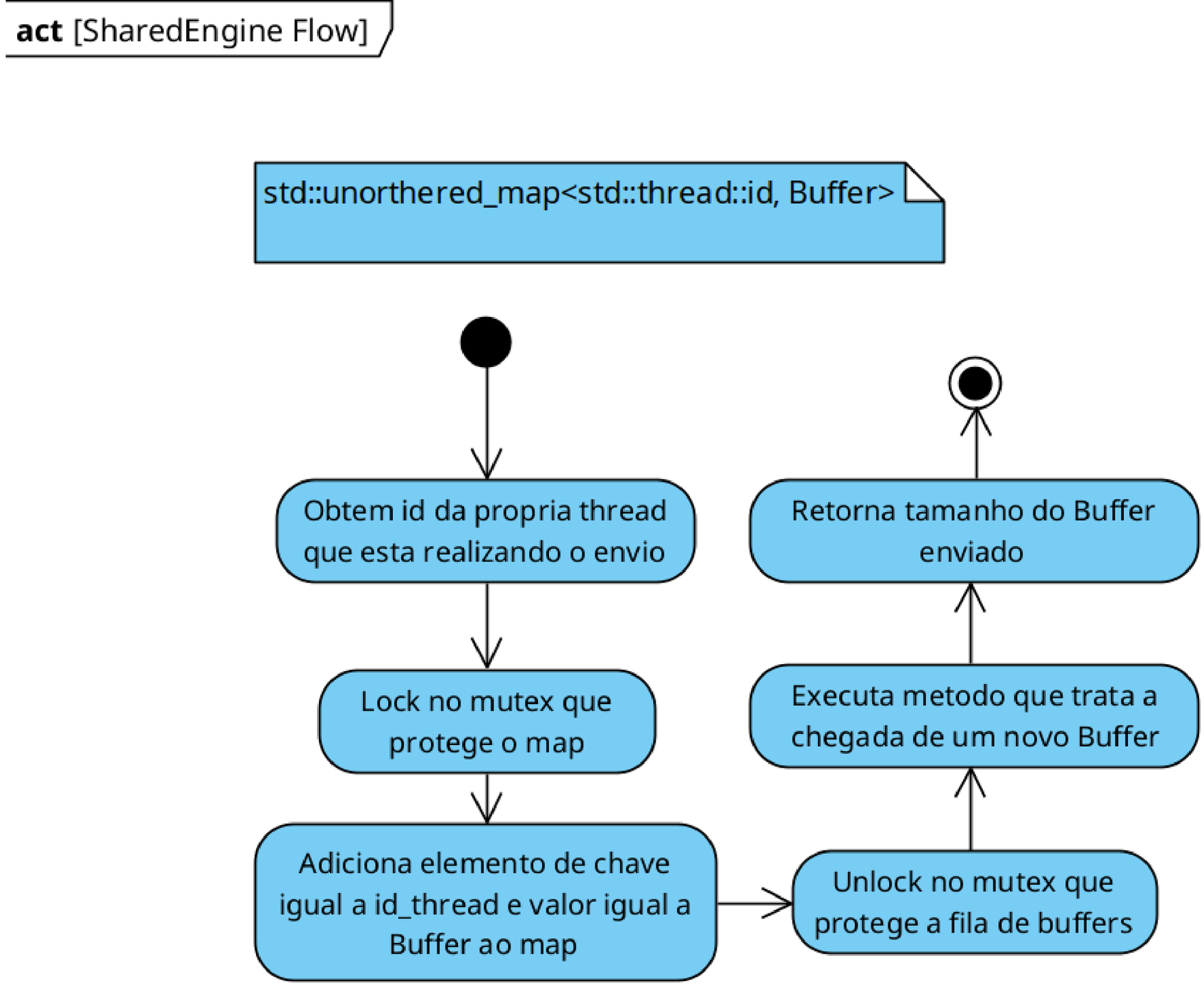

sd [Receive SharedMemEngine]
: SharedEngine
1: receive(buf : Buffer *) : unsigned int
Utilizamos try_acquire para send e receive nao serem bloqueantes quando a fila da engine estiver cheia ou vazia
1.1: bool acquired = full.try_acquire()
alt
[acquired]
1.2: buffer_sem.acquire()
remove buffer da fila 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;

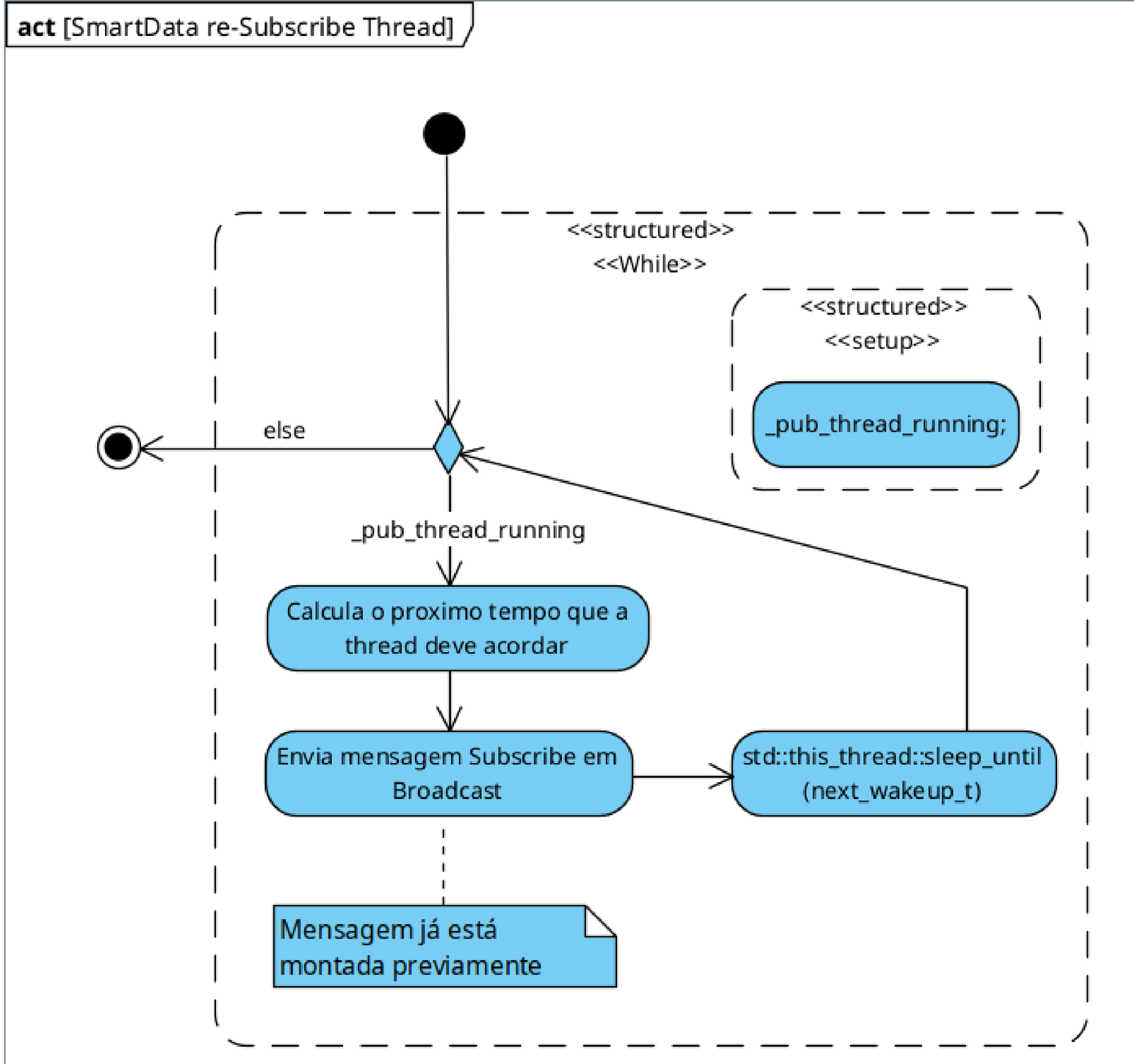

sd [Communicator.update(...)]
: Communicator
: Protocol
: SmartData P
1: update(obs : Channel::Observed*, c : Channel::Observer::Observing_Condition, buf : Buffer*) : void
Message *msg = (Message *)_channel->unmarshal(buf);
alt
[msg.type == Message::Type::COMMOM]
1.1 : Observer::update(c, buf)
[else]
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.

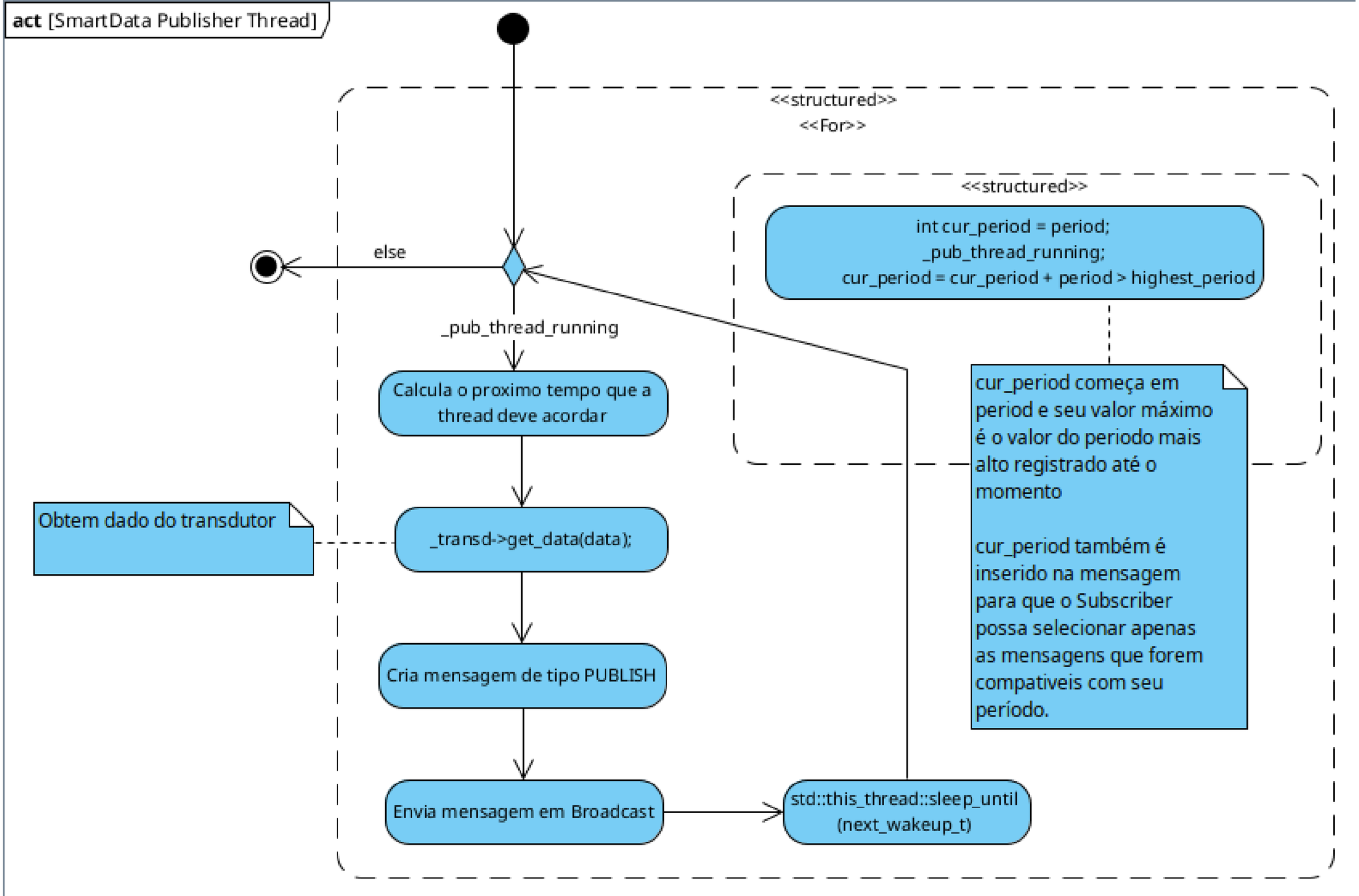
Diagramas de Atividade

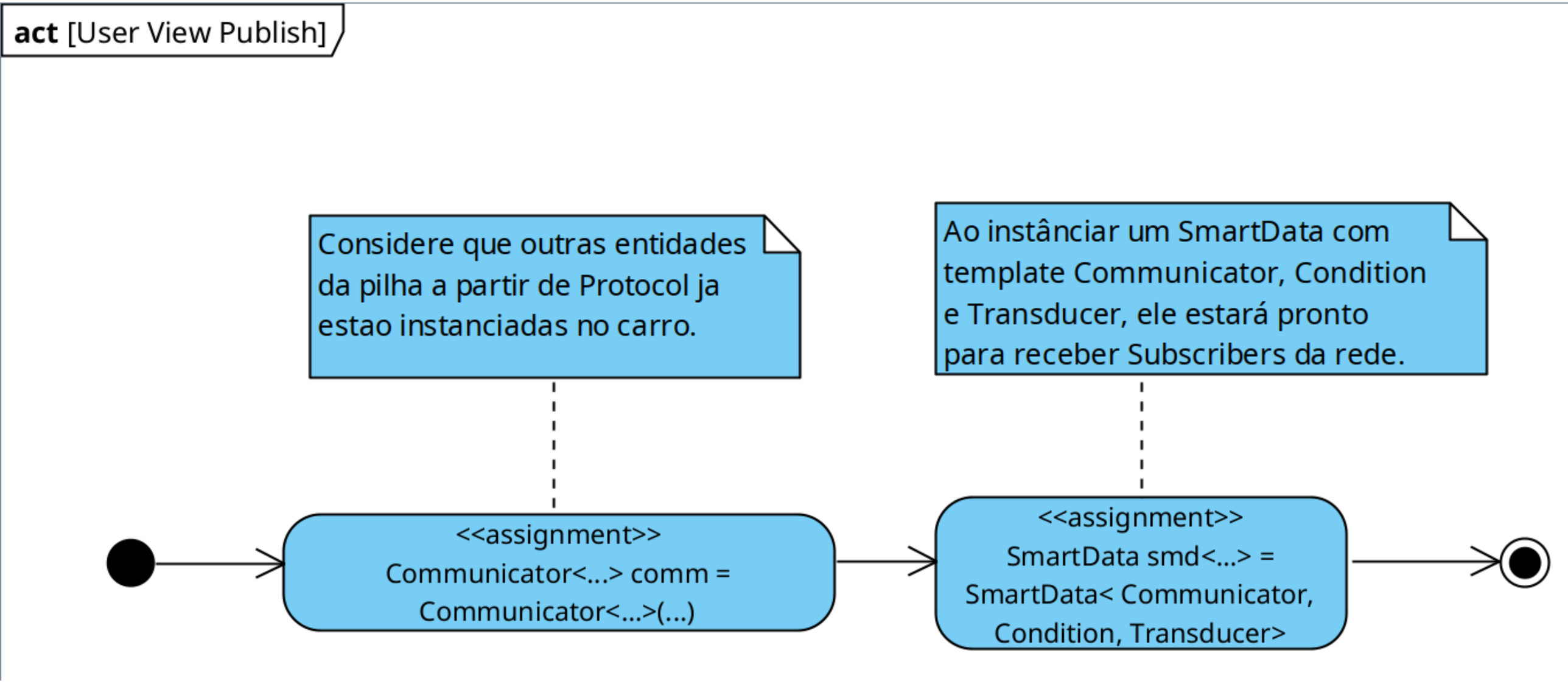

act [User View Send]
Cada Communicator é identificado unicamente pelo seu Address
Mensagem tem o seguinte formato:
Address ::= SEQUENCE {
mac OCTET STRING (SIZE(6)),
sysID OCTET STRING (SIZE(4)),
port OCTET STRING (SIZE(2))
}
Message ::= SEQUENCE {
destAddress Address,
srcAddress Address,
type OCTET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

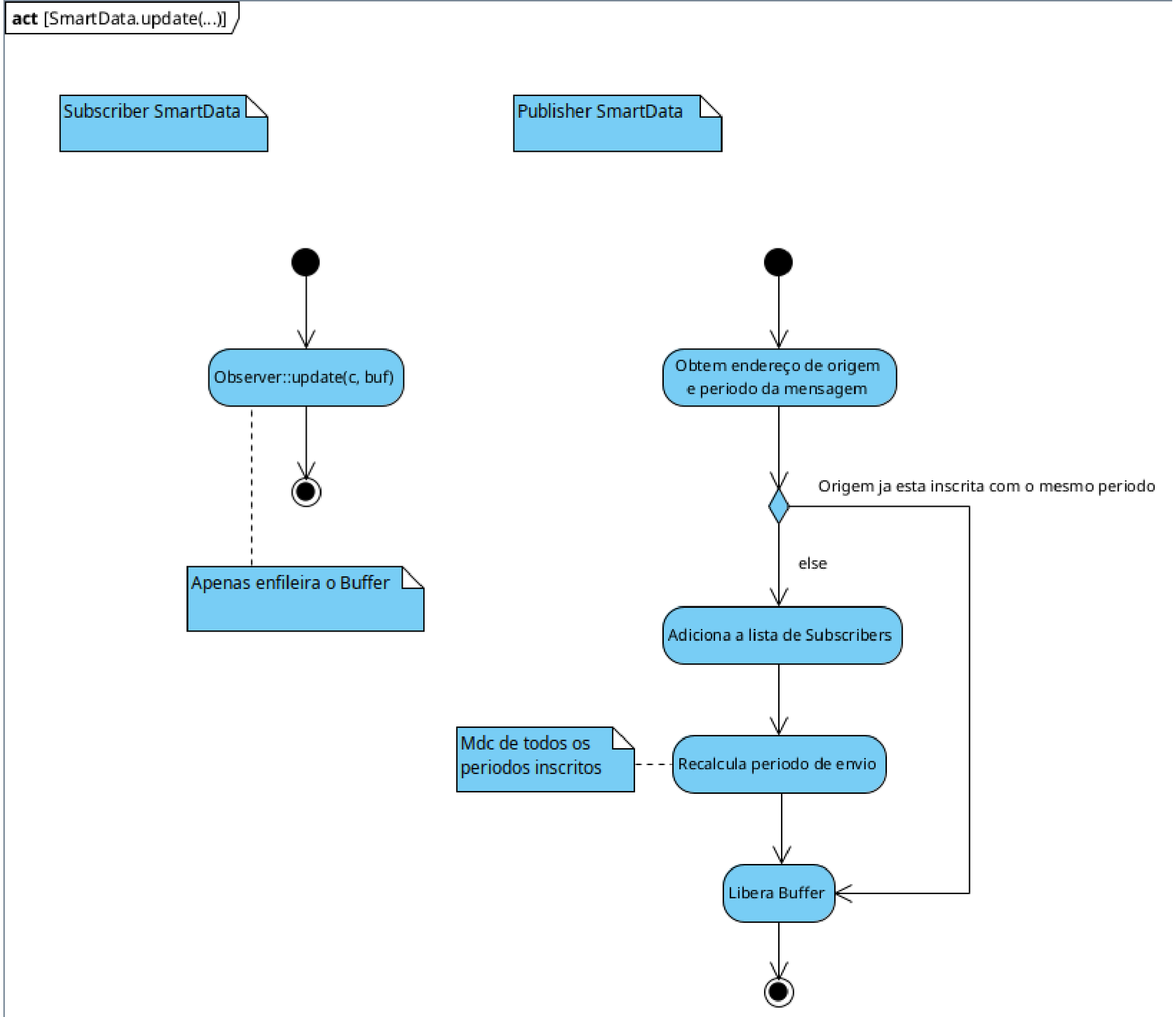

act [protocol.send(...)]
SysID destino == 0
Aloca quadro pela SharedMemNIC
Aloca quadro pela SocketNIC
Nao foi possivel alocar nenhum dos quadros
Retorna -1, indicando falha
se nao
Buf SharedMemNIC alocado com sucesso
Preenche buffer alocado SharedMemNIC
Envia buffer pela SharedMemNIC
se nao
Assumimos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

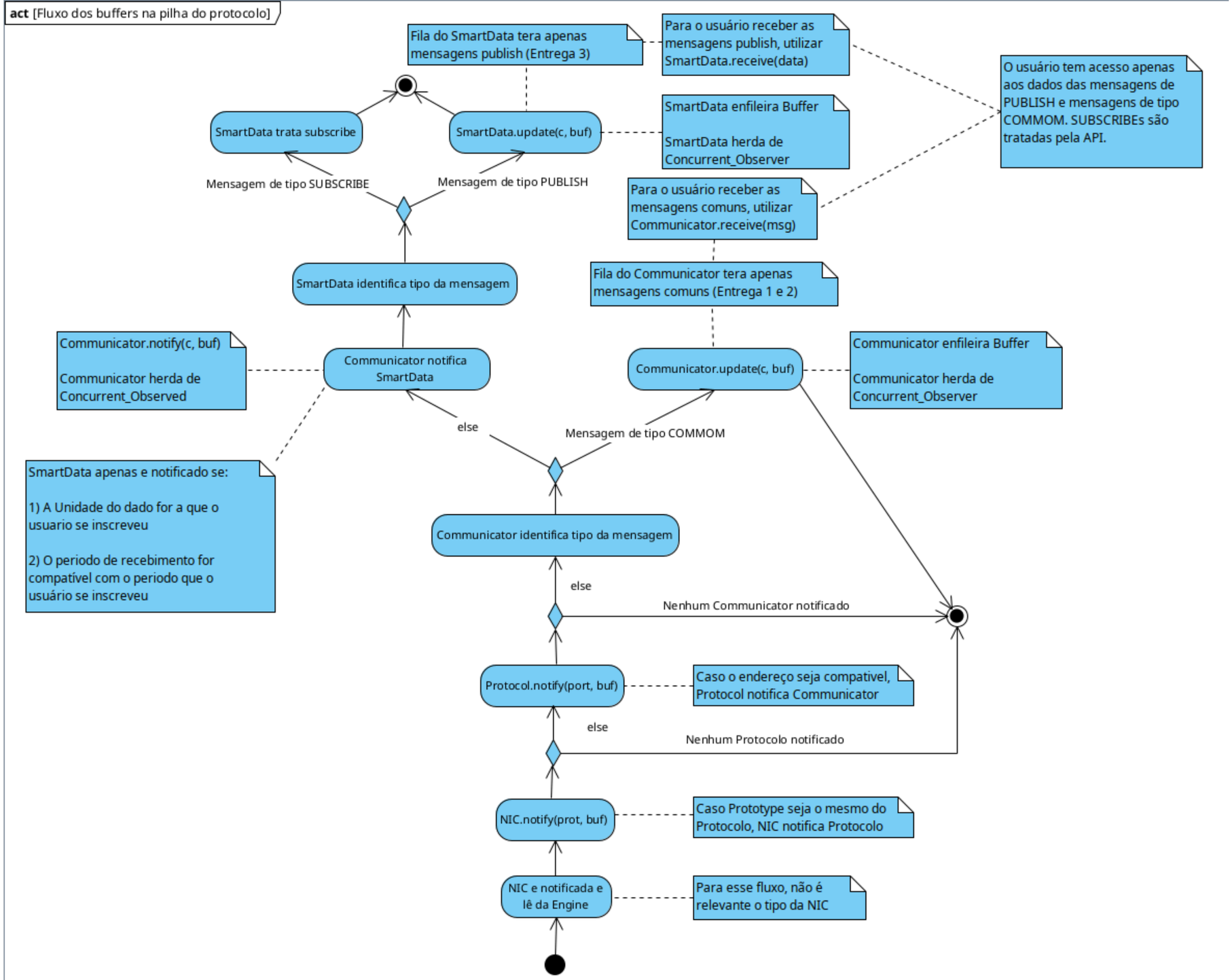

act [SharedEngine Flow]
std::unorthered_map<std::thread::id, Buffer>
Obtem id da propria thread que esta realizando o envio
Lock no mutex que protege o map
Adiciona elemento de chave igual a id_thread e valor igual a Buffer ao map
Unlock no mutex que protege a fila de buffers
Executa metodo que trata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