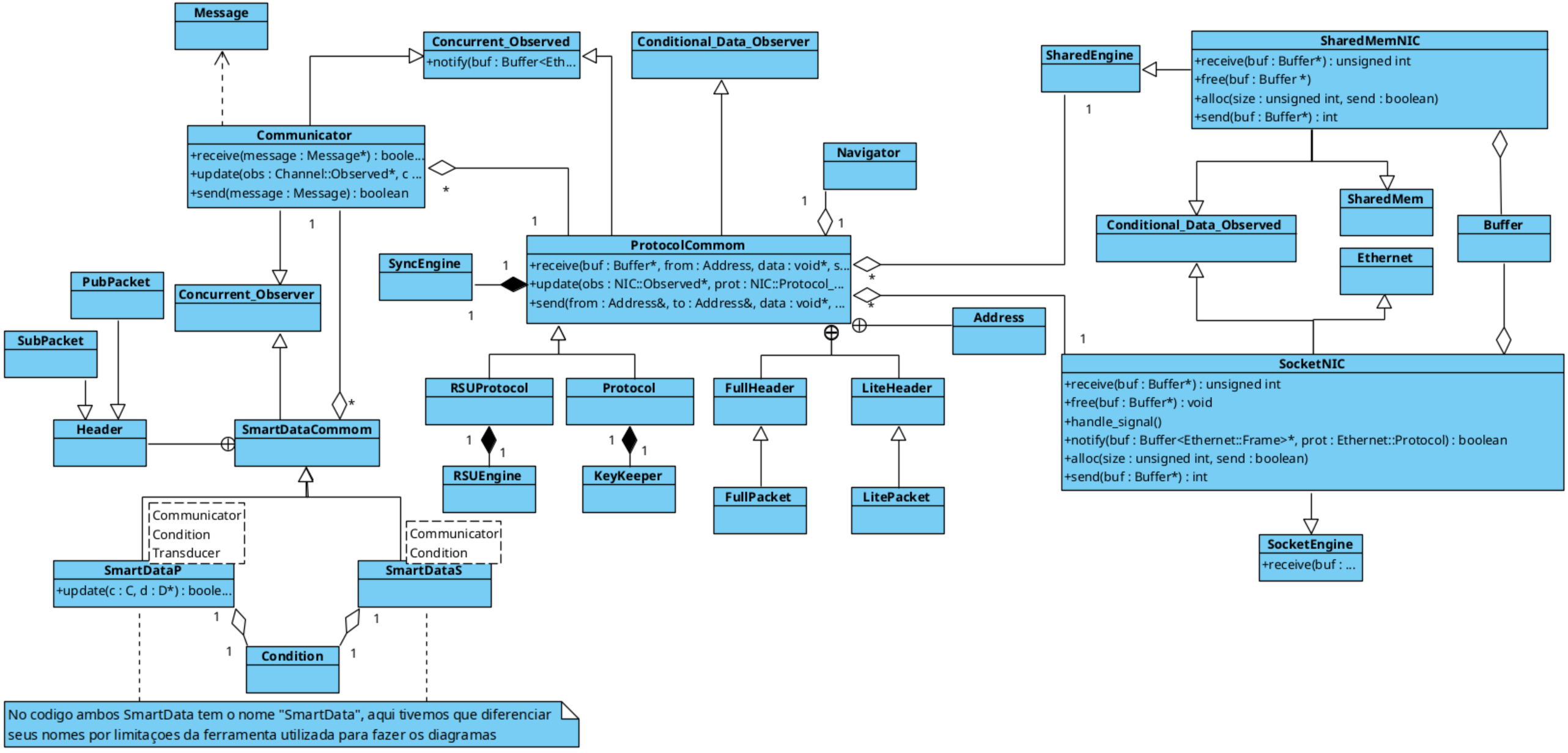
Departamento de
Informática e Estatística
CTC • UFSC

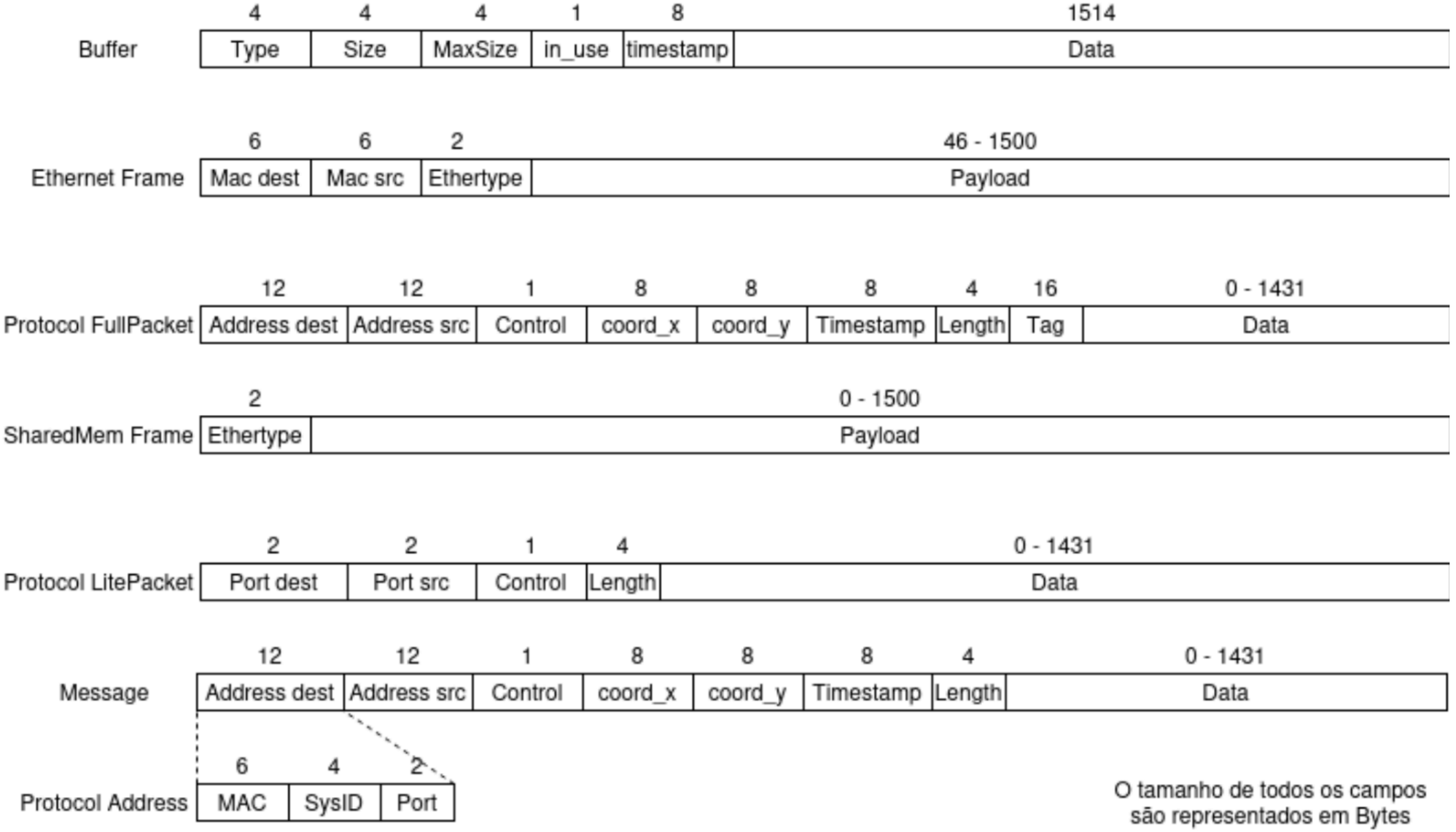
Visão geral do projeto

INE5424 - Sistemas Operacionais II

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


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

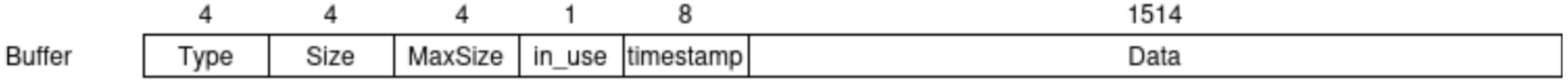
Principais Estruturas de Dados



Estrutura de dados: Buffer

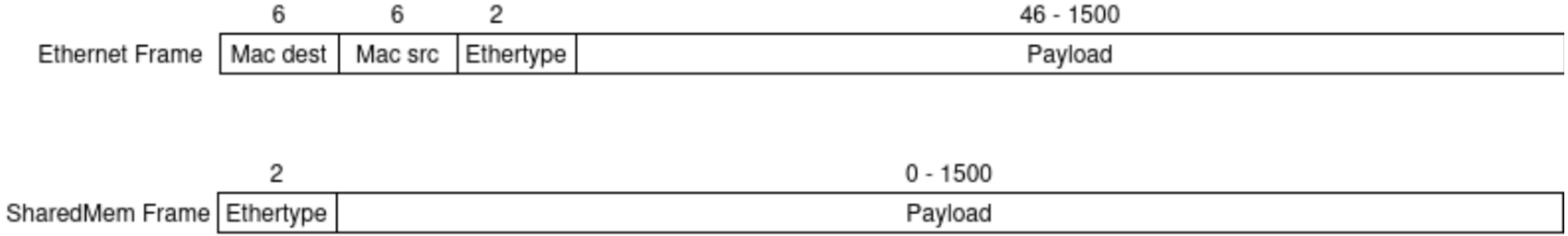
Nossa arquitetura é baseada em uma série de estruturas alinhadas, onde cada uma encapsula a próxima, permitindo que cada camada da pilha de comunicação opere com o nível de abstração apropriado.

- **Definição:** Contêiner de transporte de dados. É uma classe que encapsula um array de bytes de tamanho fixo e dados de controle.
- **Dados de controle:**
  - _type: Enumerador que define o conteúdo como EthernetFrame ou SharedMemFrame.
  - _in_use: Booleano para o gerenciamento do pool de memória. Controla se o buffer está alocado ou livre.
  - _receive_time: Timestamp (int64_t) capturado pela NIC no momento do recebimento de um quadro.



Estrutura de dados: Frames

- **Definição:** Representa o quadro da camada de enlace. Existem duas variantes:
  - **Ethernet::Frame:** Estrutura de 14 bytes de cabeçalho (MAC Destino, MAC Origem, EtherType) e payload, correspondendo a um quadro Ethernet padrão.
  - **SharedMem::Frame:** Estrutura com um cabeçalho de 2 bytes contendo apenas o EtherType e payload.
- Operação: O EtherType é presente em ambas as variantes para permitir que a NIC utilize um mecanismo de notificação uniforme para o Protocol, independentemente do Frame ter vindo da SocketEngine ou da SharedEngine.



Estrutura de Endereçamento Hierárquico

A identificação dos agentes de comunicação é feita por uma estrutura de endereço de 12 bytes, dividida em três níveis:

- **Nível Físico - MAC Address (6 bytes):** Identifica a interface de rede física. É utilizado pela SocketEngine para construir o cabeçalho Ethernet. É ignorado na comunicação interna.
- **Nível de Sistema - SysID (4 bytes, pid_t):** Identifica o sistema autônomo, modelado como um processo POSIX. Permite Protocol distinguir entre tráfego interno e externo.
- **Nível de Componente - Port (2 bytes, unsigned short):** Identifica um componente, que corresponde a uma instância de Communicator em uma thread específica.

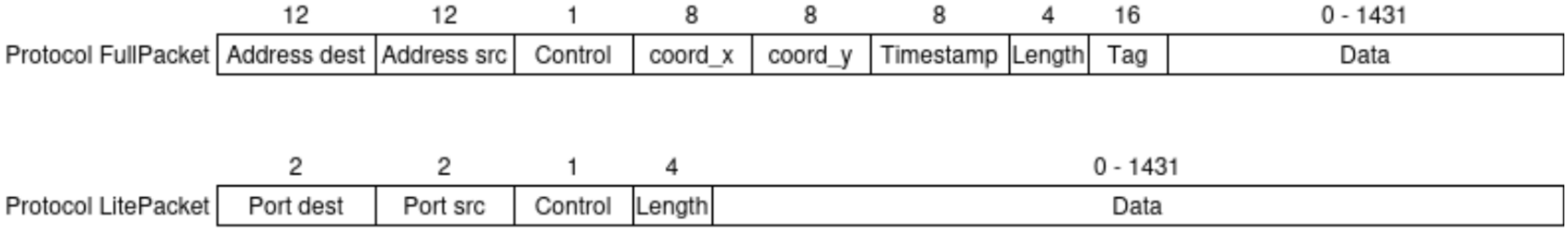
Endereços especiais:

SysID:
0 = Broadcast entre veículos e componentes
-2 = Broadcast entre veículos

Port:
0xFFFF = Broadcast entre componentes de um veículo

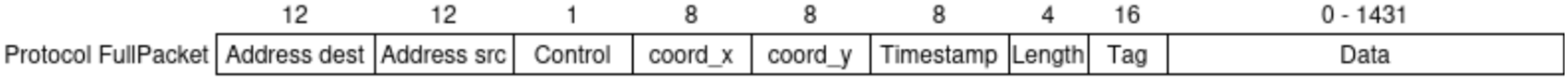
Estrutura de dados: Packet

- Os pacotes estão contidos no payload dos Frames e é onde os dados necessários para a lógica do nosso protocolo residem. Novamente, existem duas versões.
  - **FullPacket:** Utilizado para comunicação externa, seu cabeçalho contém endereços completos, coordenadas, timestamp e tag de segurança.
  - **LitePacket:** Utilizado para comunicação interna, contém um cabeçalho otimizado apenas com as informações essenciais.



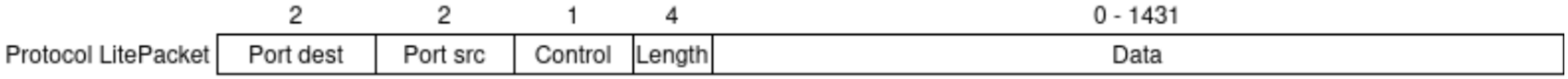
Formato do Pacote Externo: FullPacket

- **Campos do FullHeader:**
  - Address origin e Address dest: Endereços completos de 12 bytes.
  - Control ctrl: Byte de controle com flags de tipo e estado.
  - double coord_x, coord_y: Coordenadas do remetente, obtidas do Navigator.
  - uint64_t timestamp: Timestamp de envio, obtido do SyncEngine.
  - MAC::Tag tag: Tag de 16 bytes do Poly1305.



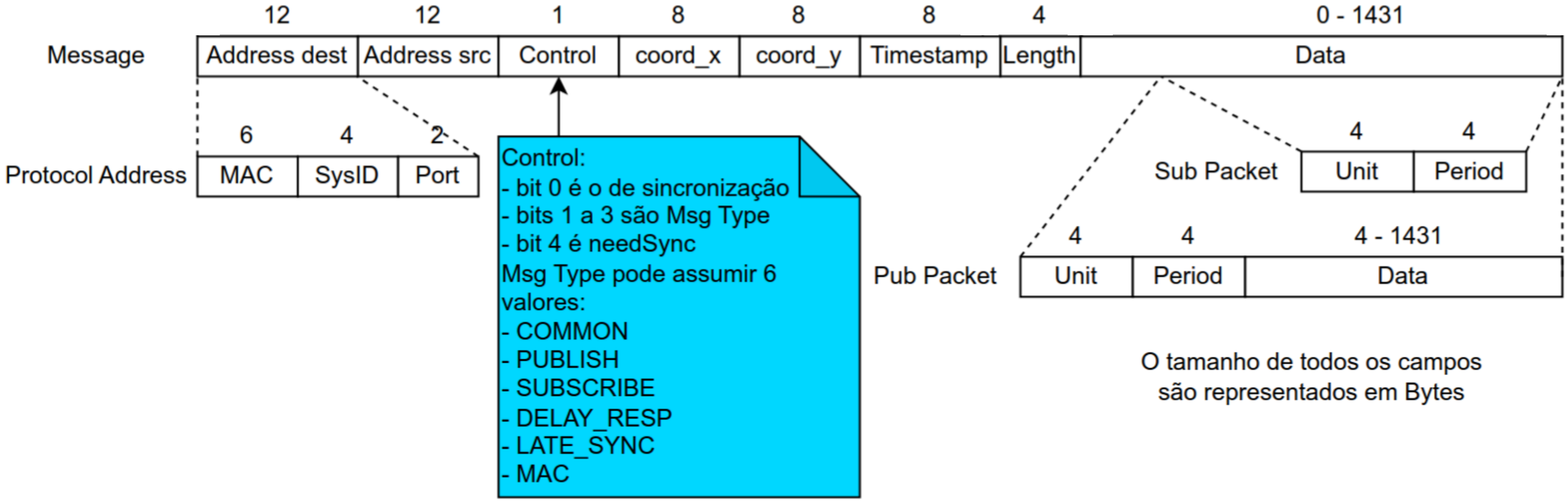
Formato do Pacote Interno: LitePacket e Otimização

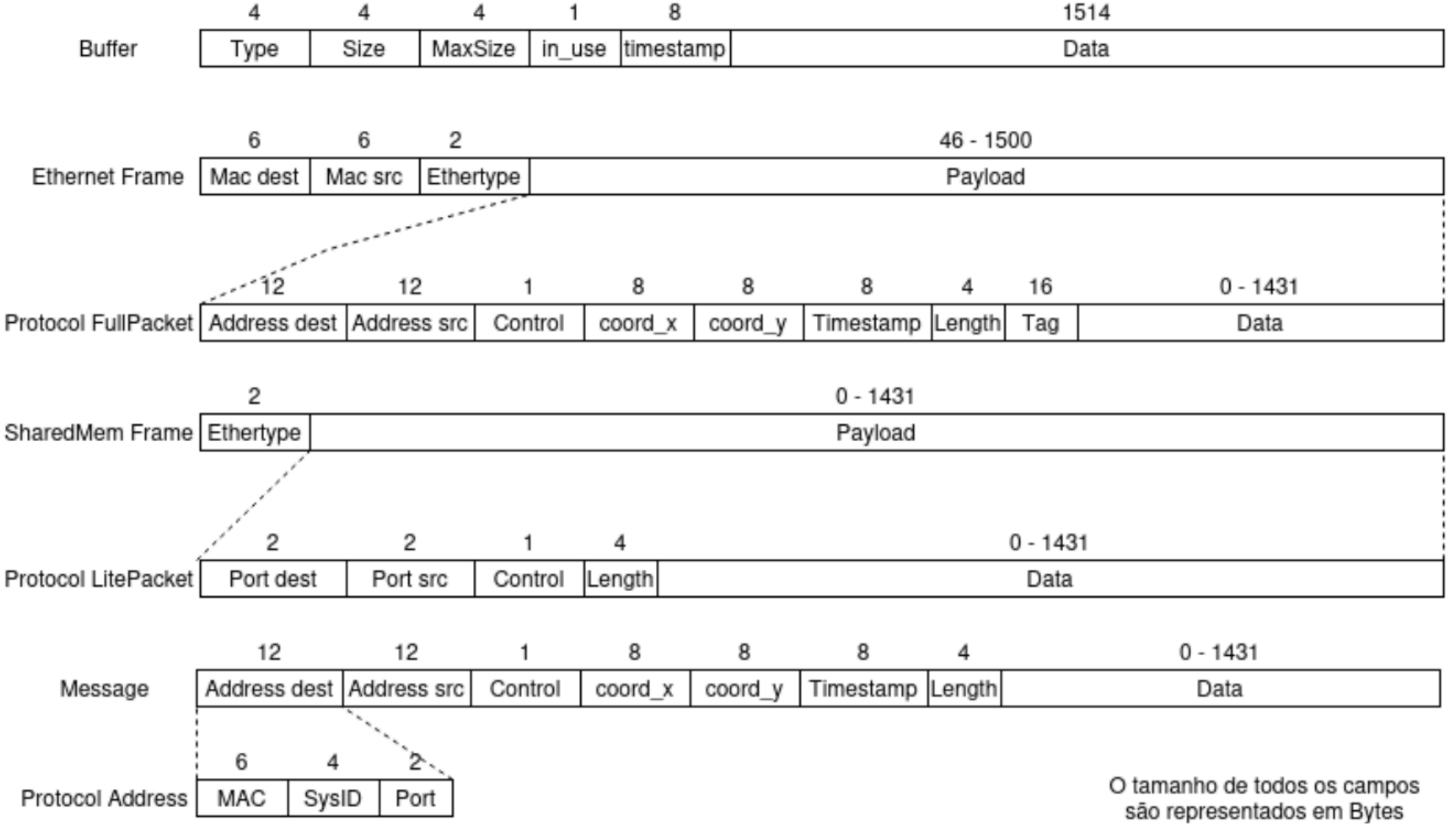
- **Campos Mantidos:** Apenas a Port de origem e destino, o campo de Control e o tamanho do payload.
- **Campos Omitidos:**
  - **Endereço Físico e SysID:** São implícitos, pois a comunicação ocorre dentro do mesmo processo.
  - **Coordenadas:** A classe Message foi projetada para obter esses valores sob demanda, consultando diretamente o Navigator do Protocol.
  - **Timestamp:** Timestamp é o obtido a partir do timestamp de recebimento do Buffer.
  - **Tag de Autenticação:** Removida, pois a verificação de segurança é desnecessária dentro do mesmo domínio.



Estrutura de dados: Message

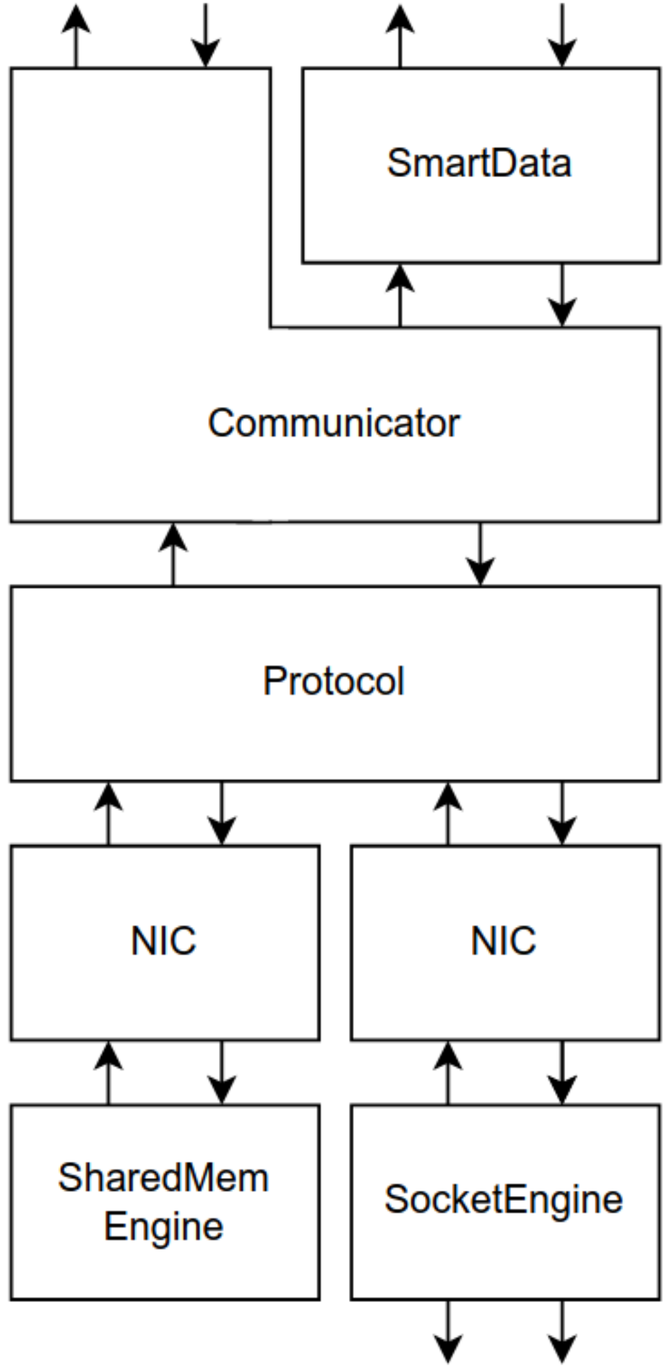
- Message - A Interface para a Aplicação
  - **Função:** É a estrutura de dados que a camada de aplicação manipula diretamente. Ela abstrai os formatos de pacote subjacentes.
  - **Mecanismo de Marshalling/Unmarshalling:** No envio, o Protocol serializa (marshal) os dados de um objeto Message para dentro de um FullPacket ou LitePacket. Na recepção, ele desserializa (unmarshal) os dados de um Packet para preencher um objeto Message.




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
O tamanho de todos os campos
são representados em Bytes

Visão Geral da Arquitetura

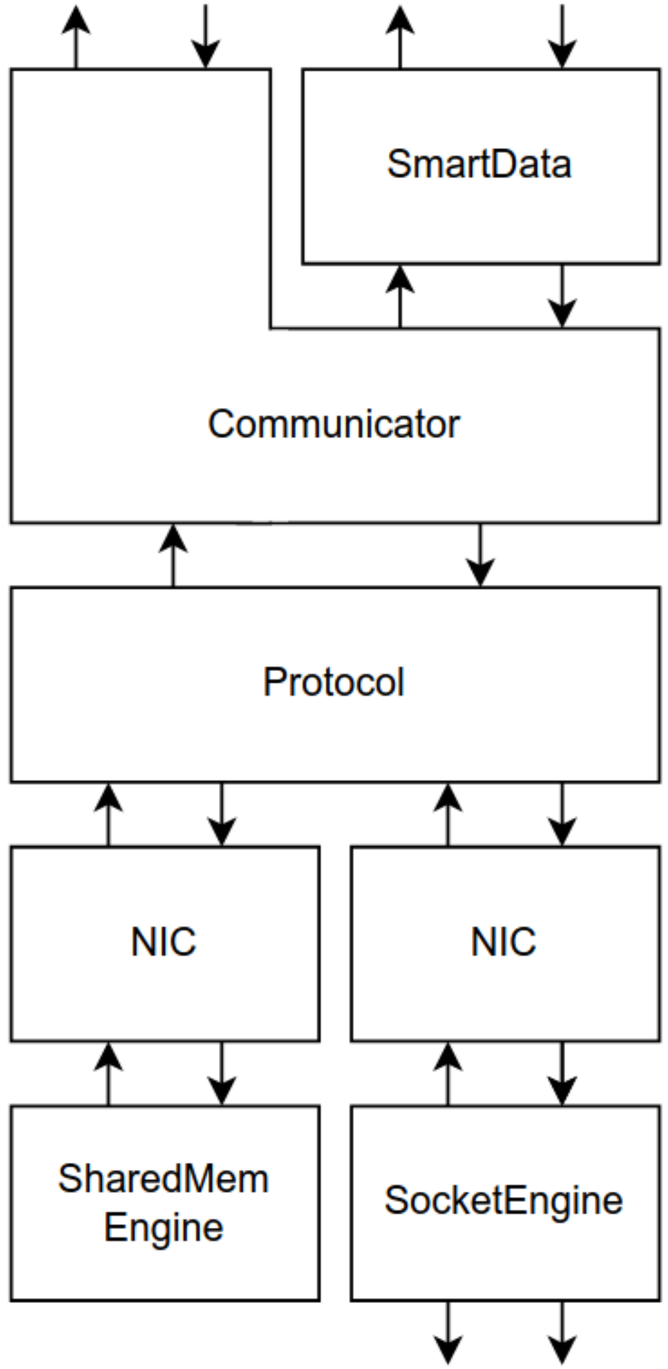
- **Communicator e SmartData:** Camada de interface com a aplicação. SmartData constrói sobre o Communicator para implementar a lógica de Publish-Subscribe.
- **Protocol:** Roteamento de mensagens (interno vs. externo) com base no SysID, demultiplexação por porta para os Communicators, implementação da lógica de segurança (verificação de MAC) e de sincronização temporal (PTP).



Visão Geral da Arquitetura

- **NIC:** Atua como uma fachada sobre a Engine. É responsável pelo gerenciamento do ciclo de vida dos buffers e notifica a camada Protocol com base no EtherType dos quadros recebidos.

- **Engines:** Encapsulam comunicação por rawsockets ou memória compartilhada.



SocketEngine - comunicação entre veículos

- **Mecanismo de Comunicação:** Utiliza um Raw Socket para envio e recebimento de quadros Ethernet.
- **Recepção Assíncrona:** A chegada de pacotes é notificada pelo sinal SIGIO. A Engine se registra para receber este sinal usando fcntl com a flag F_SETOWN.
- **Tratamento de Sinal:** O *signal handler* registrado para SIGIO executa uma única função: notificar uma std::condition_variable. A thread de recebimento, que está bloqueada nesta variável, é então acordada e invoca a lógica de processamento (NIC::handle_signal).
- **Justificativa:** Esta abordagem foi escolhida para contornar as restrições dos *signal handlers* em POSIX. Funções não Assync-Signal-Safe, como funções não reentrantes, alocação de memória ou aquisição de mutexes, não podem ser chamadas de forma segura dentro de um handler. Ao delegar o trabalho a uma thread separada, garantimos que o processamento dos pacotes ocorra em um contexto controlado.

SharedMemEngine - comunicação interna

A SharedMemEngine é a implementação da Engine para comunicação intra-processo.

- **Mecanismo de Comunicação:** A transferência de dados ocorre inteiramente em memória compartilhada. A estrutura de dados central é um mapa que associa o identificador de uma thread a um **ponteiro para um Buffer**. O acesso a este mapa é protegido através de um semáforo binário.
  - Para manter a API consistente com a SocketEngine, o método send(...) da SharedMemEngine, após inserir o buffer no mapa, invoca diretamente o método NIC::handle_signal() da SharedMemNIC. Isso permite que a camada NIC trate ambos os tipos de Engine de forma idêntica.

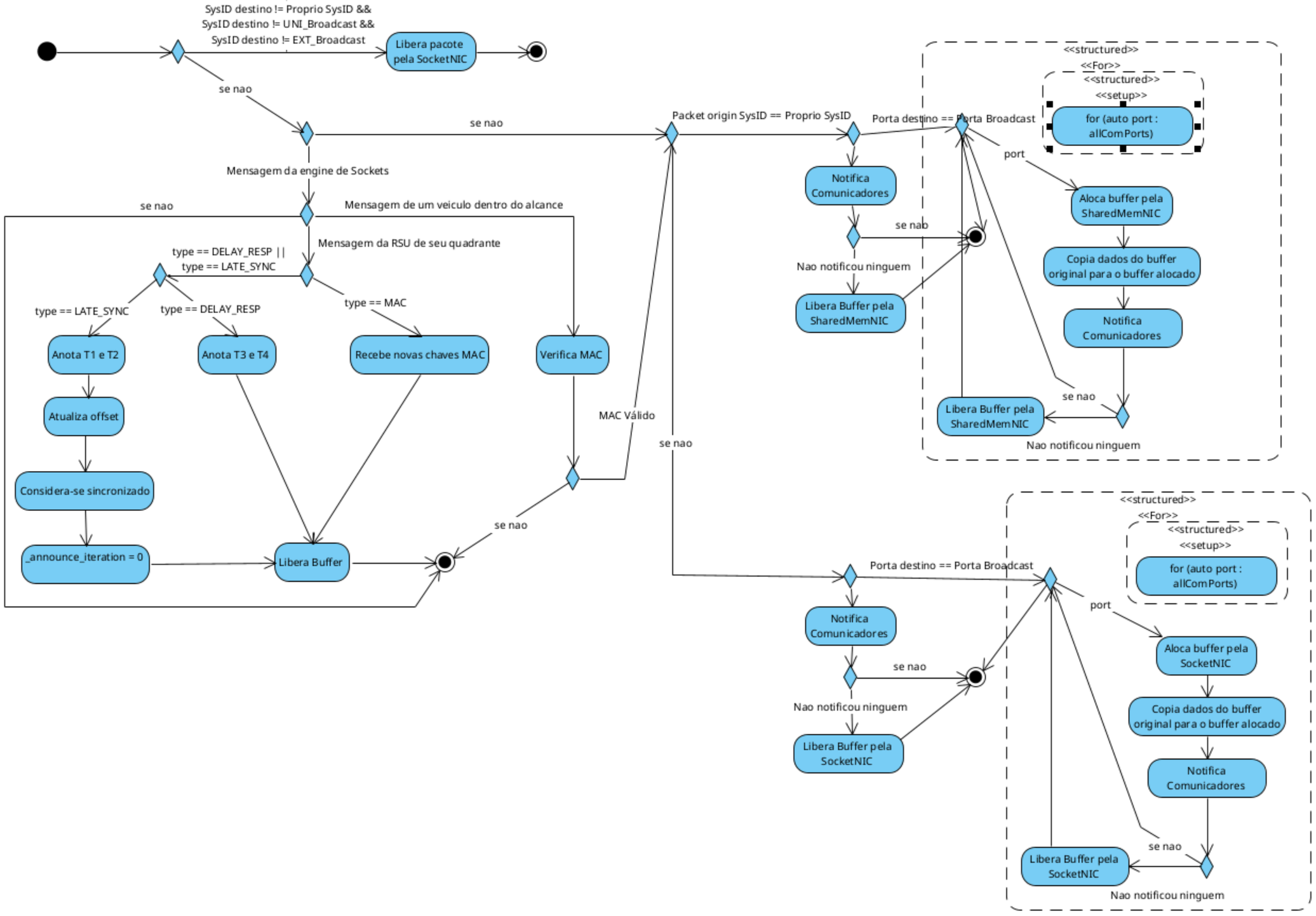
NIC e o Gerenciamento de Buffers

A NIC abstrai a Engine e implementa uma política de gerenciamento de memória para os buffers de comunicação.

- **Pools de buffers:** A NIC pré-aloca, em sua inicialização, dois arrays de objetos Buffer: um para envio e outro para recepção. O tamanho desses pools é fixo e definido em tempo de compilação.
- **Ciclo de Vida do Buffer:**
  - alloc(...): A camada Protocol solicita um buffer. O método alloc(...) percorre o pool de forma circular, protegido por um std::mutex, para encontrar o primeiro buffer que não está em uso (!is_in_use()).
  - O buffer é marcado como "em uso" e retornado.
  - free(...): Após o uso, o buffer é devolvido ao pool, e sua flag in_use é resetada.
- **Justificativa:** Evitar a alocação dinâmica de memória (new/delete) durante o fluxo crítico de comunicação. Alocações dinâmicas introduzem latência, com o pooling, a obtenção de um buffer se torna uma operação mais performática.

Protocol

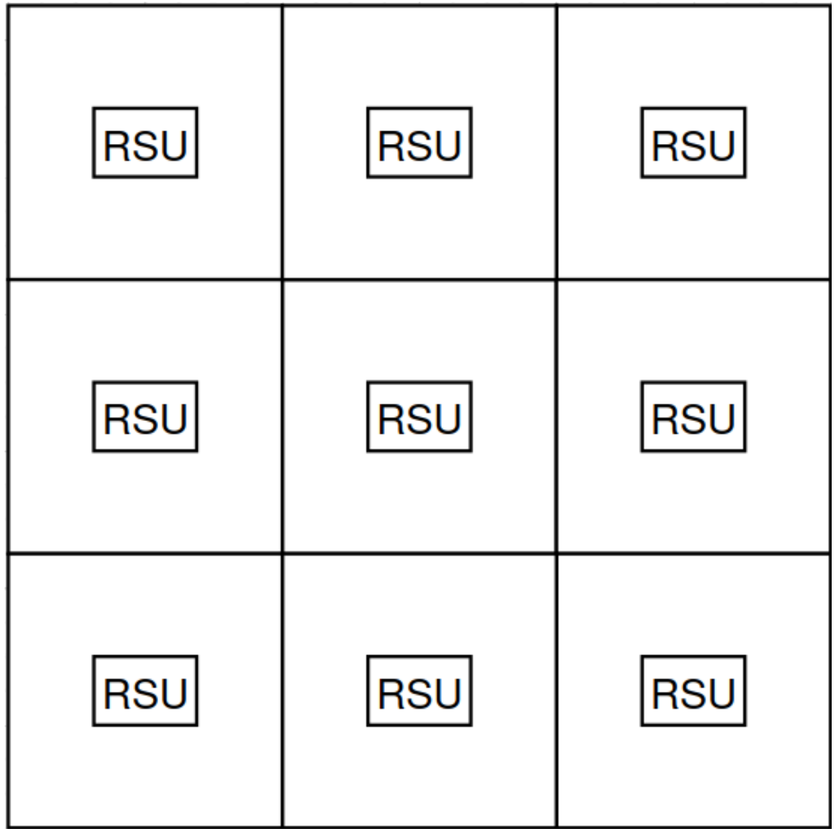
- **Envio de pacotes:**
  - Se o SysID de destino é igual ao SysID local, a mensagem é interna e encaminhada para a instância da SharedMemNIC.
  - Caso contrário, a mensagem é externa e encaminhada para a instância da SocketNIC.
  - Implementa broadcast interno ao veículo, externo e ambos simultaneamente.
- **Recebimento de pacotes:**
  - Protocol atua como observador das NICs. Quando seu método update(...) é invocado por uma das NICs, ele extrai a Port de destino do cabeçalho do pacote recebido. Em seguida, ele notifica apenas o Communicator que se registrou para aquela Port específica.
- **Sincronização Temporal**
- **Comunicação segura**


SysID destino != Proprio SysID &&
SysID destino != UNI_Broadcast &&
SysID destino != EXT_Broadcast
Libera pacote pela SocketNIC
se nao
se nao
Mensagem da engine de Sockets
se nao
Mensagem de um veiculo dentro do alcance
Mensagem da RSU de seu quadrante
type == DELAY_RESP ||
type == LATE_SYNC
type == LATE_SYNC
type == DELAY_RESP
type == MAC
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 Válido
se nao
Packet origin SysID == Proprio SysID
Notifica Comunicadores
se nao
Nao notificou ninguem
Libera Buffer pela SharedMemNIC
<<structured>>
<<For>>
<<structured>>
<<setup>>
for (auto port : allComPorts)
Porta destino == Porta Broadcast
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
Nao notificou ninguem

RSUs

- RSUs fazem o papel de líder do PTP para os veículos dentro de seu quadrante.
- RSUs distribuem periodicamente as chaves MAC necessárias para os veículos se comunicarem, ou seja, periodicamente os veículos recebem a chave de sua RSU e das RSUs vizinhas.

Mapa Exemplo



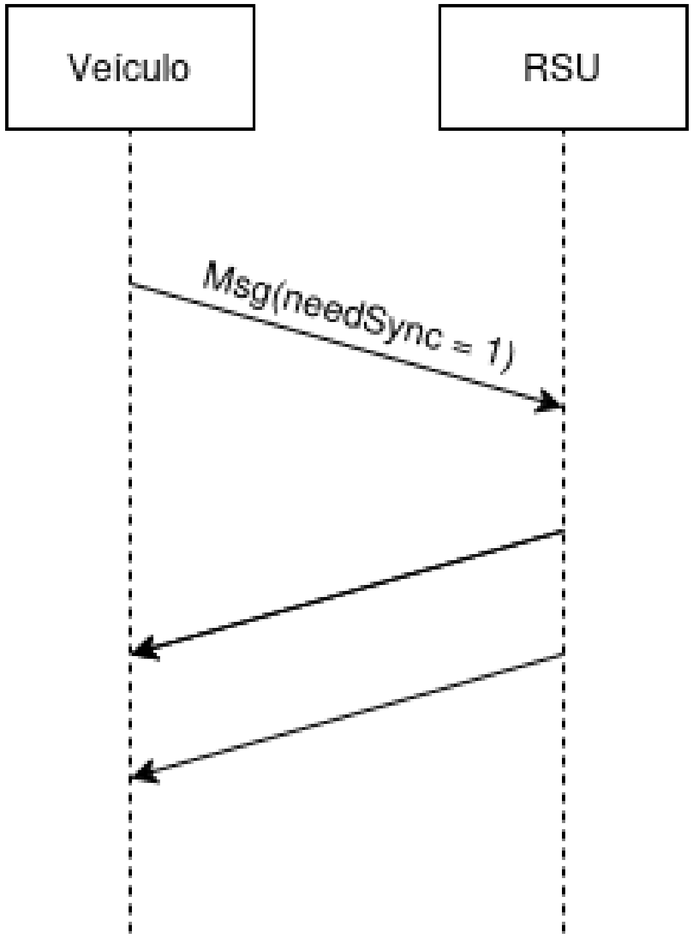
Uso das Chaves no Veículo

- **KeyKeeper:** Classe auxiliar dentro de Protocol que armazena um mapa que associa o ID de um quadrante à sua chave MAC.
- **Navigator:** Fornece a localização geográfica do veículo e a topologia da rede.
- **Seleção da Chave Correta:**
  - **No envio:** O Protocol utiliza o Navigator para determinar seu quadrante atual. O ID deste quadrante é usado para consultar a chave correta no KeyKeeper e calcular a tag da mensagem.
  - **No recebimento:** O Protocol extrai as coordenadas do remetente do cabeçalho do pacote, determina o quadrante de origem da mensagem e utiliza a chave daquele quadrante para verificar a tag. Mensagens com tags inconsistentes são descartadas.

Sincronização Temporal:

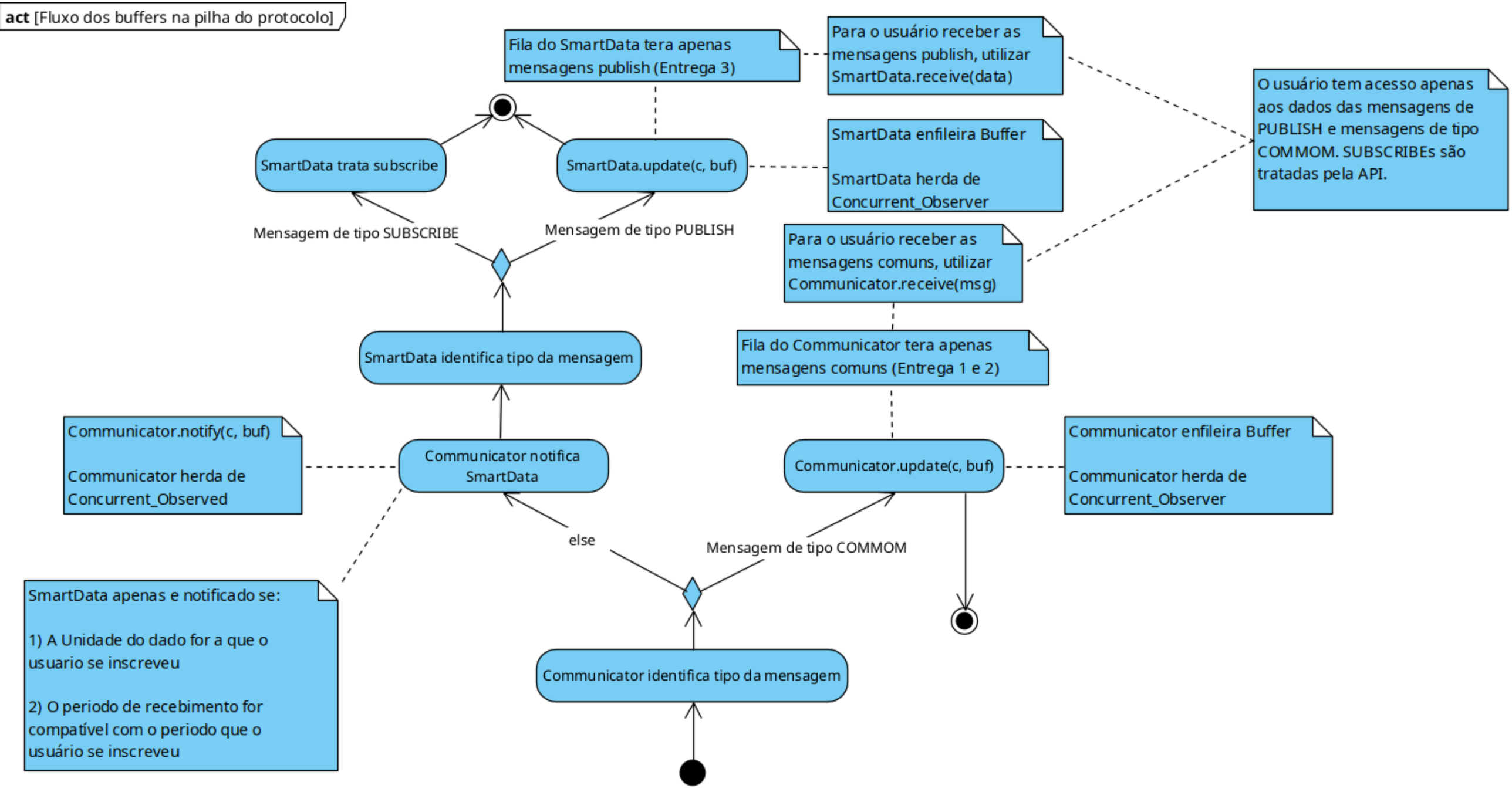
A sincronização temporal é implementada através de um modelo sob demanda.

- **Papéis:** RSUs atuam como Mestres; Veículos atuam como Escravos.
- **Modelo de Interação:**
  - Um veículo que precisa de sincronização passa a enviar mensagens com a flag needSync ativada.
  - Se não está transmitindo Broadcasts, envia uma mensagem de Announce periodicamente.
  - A RSU, ao receber este pedido, responde com um par de mensagens DELAY_RESP e LATE_SYNC, que contêm os timestamps necessários para o cálculo do offset.



Communicator

- Interface que o usuário pode utilizar para se comunicar por meio de mensagens do tipo **COMMON**
- Notifica camada superior (SmartData) quando necessário.

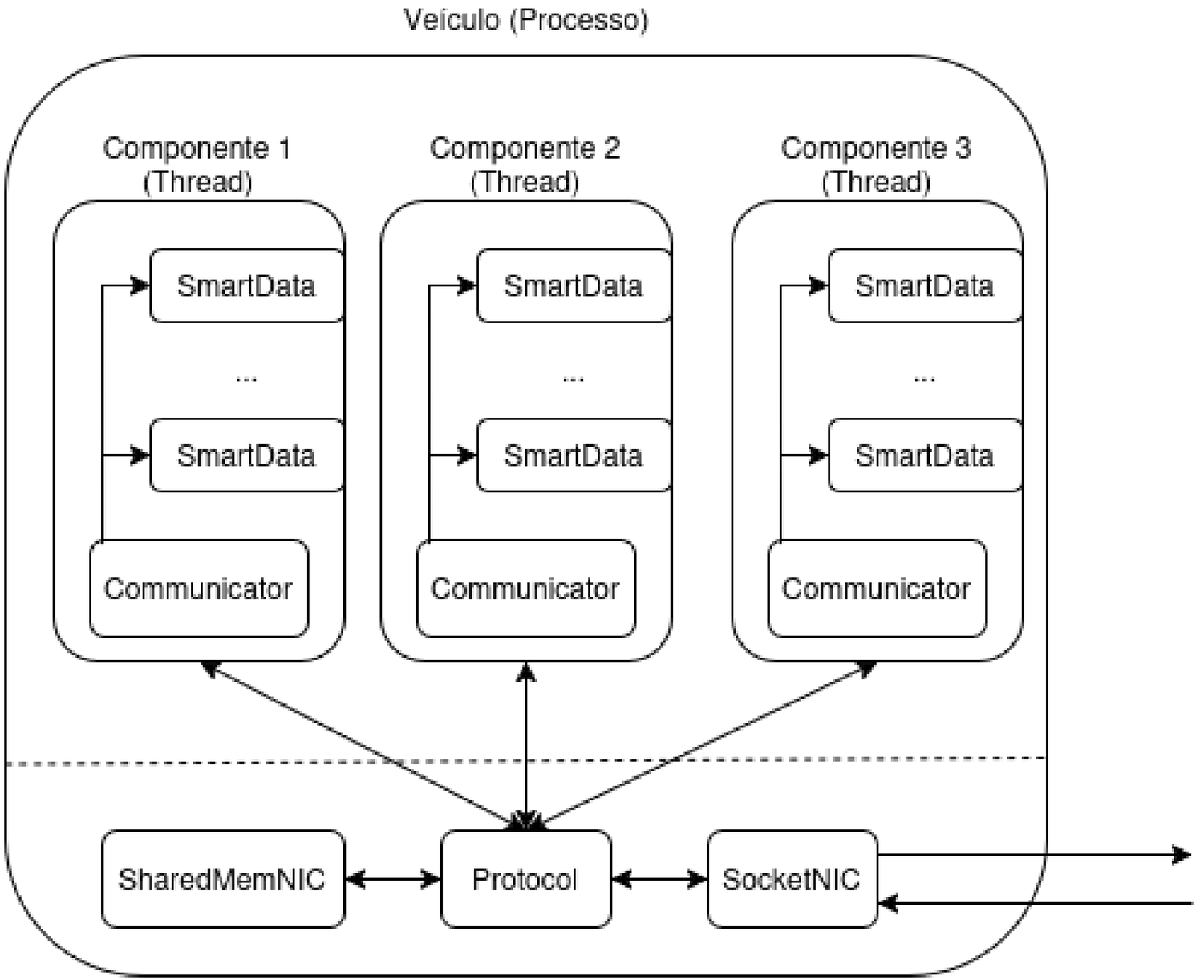

act [Fluxo dos buffers na pilha do protocolo]
Fila do SmartData tera apenas mensagens publish (Entrega 3)
Para o usuário receber as mensagens publish, utilizar SmartData.receive(data)
O usuário tem acesso apenas aos dados das mensagens de PUBLISH e mensagens de tipo COMMOM. SUBSCRIBEs são tratadas pela 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

SmartData: Lógica do Publisher

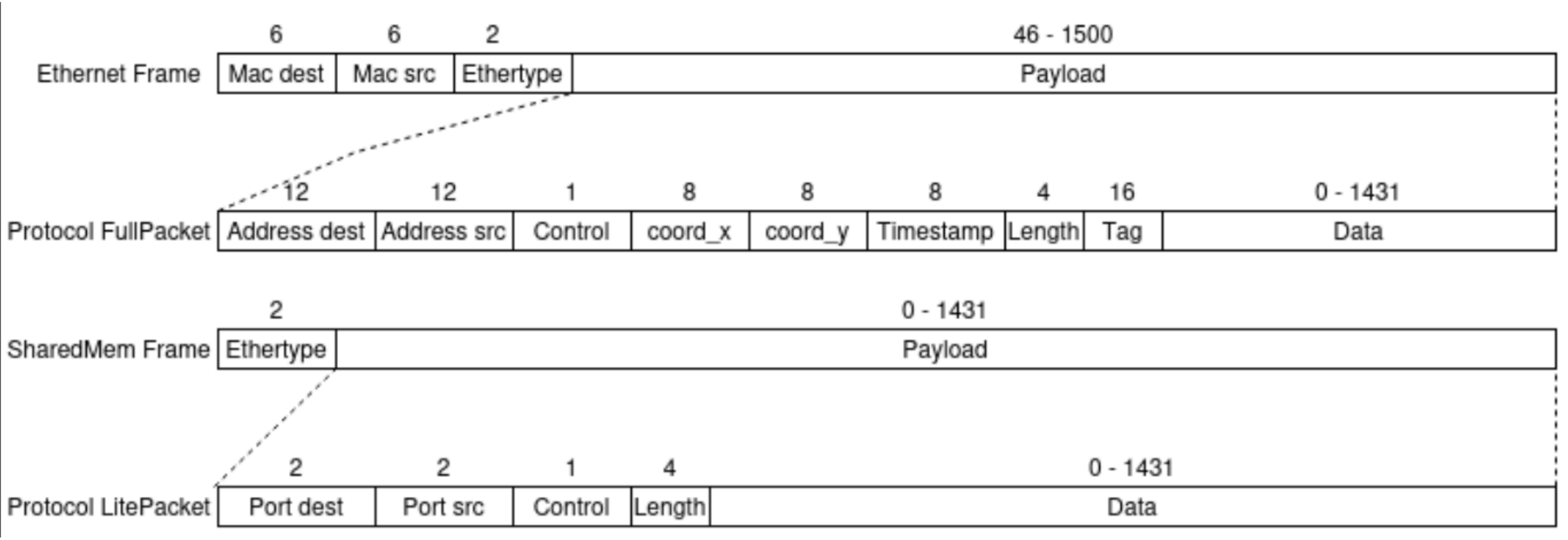
- **Entidade:** SmartData<..., ..., Transducer>.
  - Um SmartData Publisher tem um ponteiro para o transdutor que produz o seu dado.
- **Gerenciamento de Período:** O período é recalculado dinamicamente como o Máximo Divisor Comum (MDC) dos períodos de todos os assinantes ativos.
- **Contador de Ciclo:** Cada mensagem PUBLISH carrega um contador de ciclo. Este contador é incrementado pelo valor do MDC a cada publicação.
- **Justificativa:** O uso do MDC é uma otimização que garante que a frequência de publicação seja a mais baixa possível que ainda satisfaz o assinante mais exigente. O contador de ciclo permite que cada assinante saiba quais publicações lhe dizem respeito.

SmartData: Lógica do Subscriber

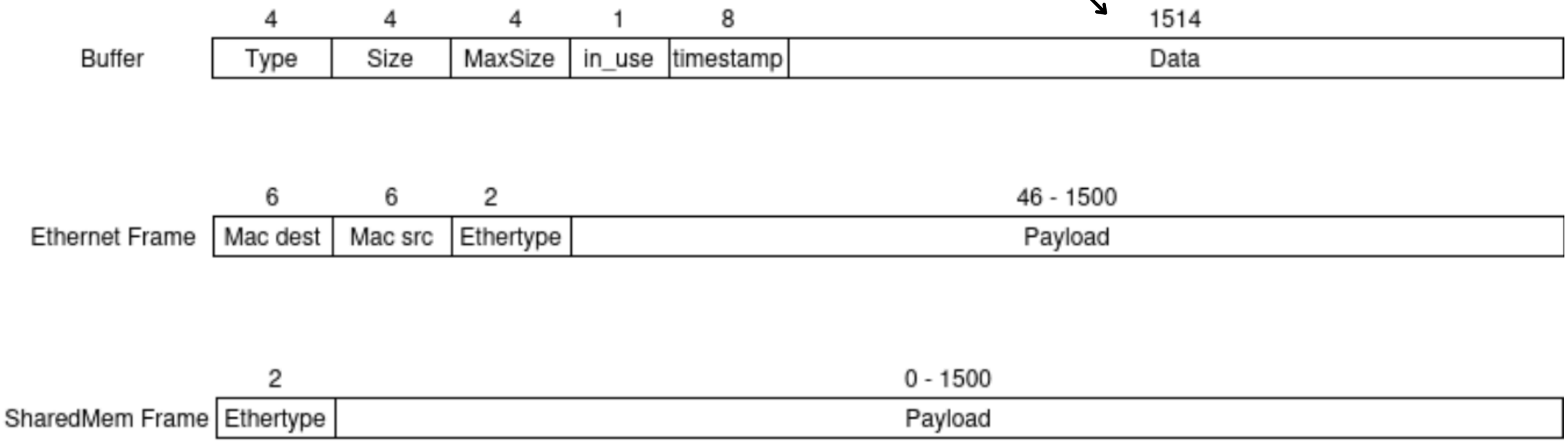
- **Entidade:** SmartData<..., ..., void>.
- **Filtragem na Recepção:** Um Subscriber só processa uma mensagem PUBLISH se duas condições forem atendidas:
  - Correspondência de Tipo: A "unidade" de dados na mensagem é a mesma que a de sua subscrição.
  - Correspondência de Momento: O contador de ciclo da publicação é um múltiplo do período que ele solicitou.
- **Justificativa:** A verificação do contador de ciclo permite que o subscriber descarte publicações que não são para ele, mesmo que sejam do tipo de dado correto. Isso evita processamento de dados redundantes e permitindo que cada componente opere em sua própria taxa.

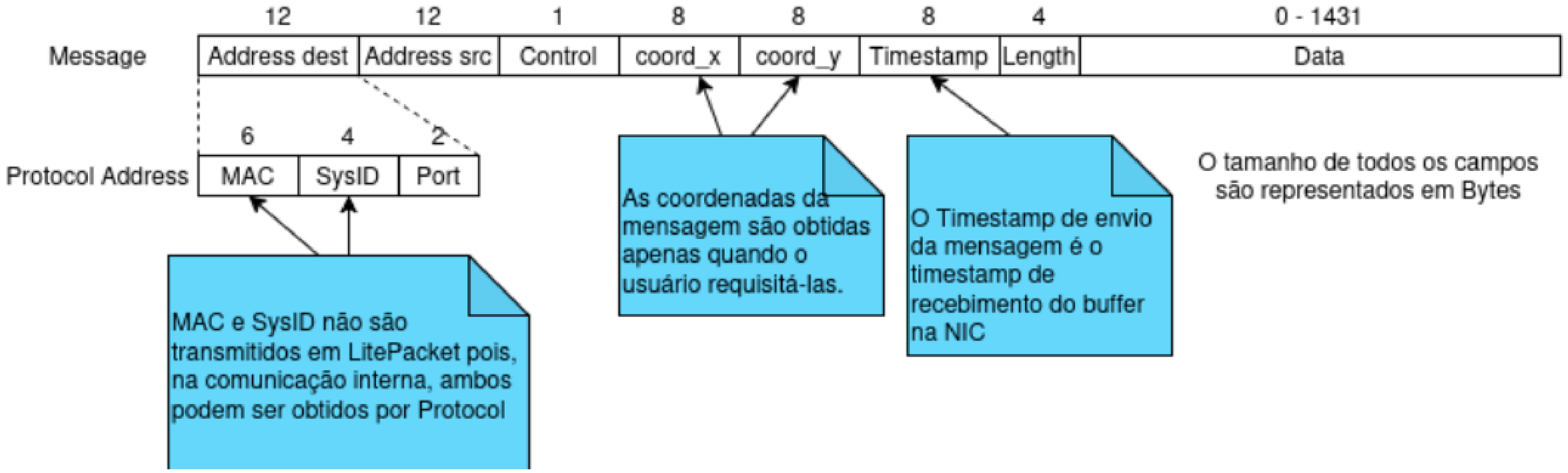

Veículo (Processo)
Componente 1
(Thread)
Componente 2
(Thread)
Componente 3
(Thread)
SmartData
...
SmartData
Communicator
SmartData
...
SmartData
Communicator
SmartData
...
SmartData
Communicator
SharedMemNIC
Protocol
SocketNIC

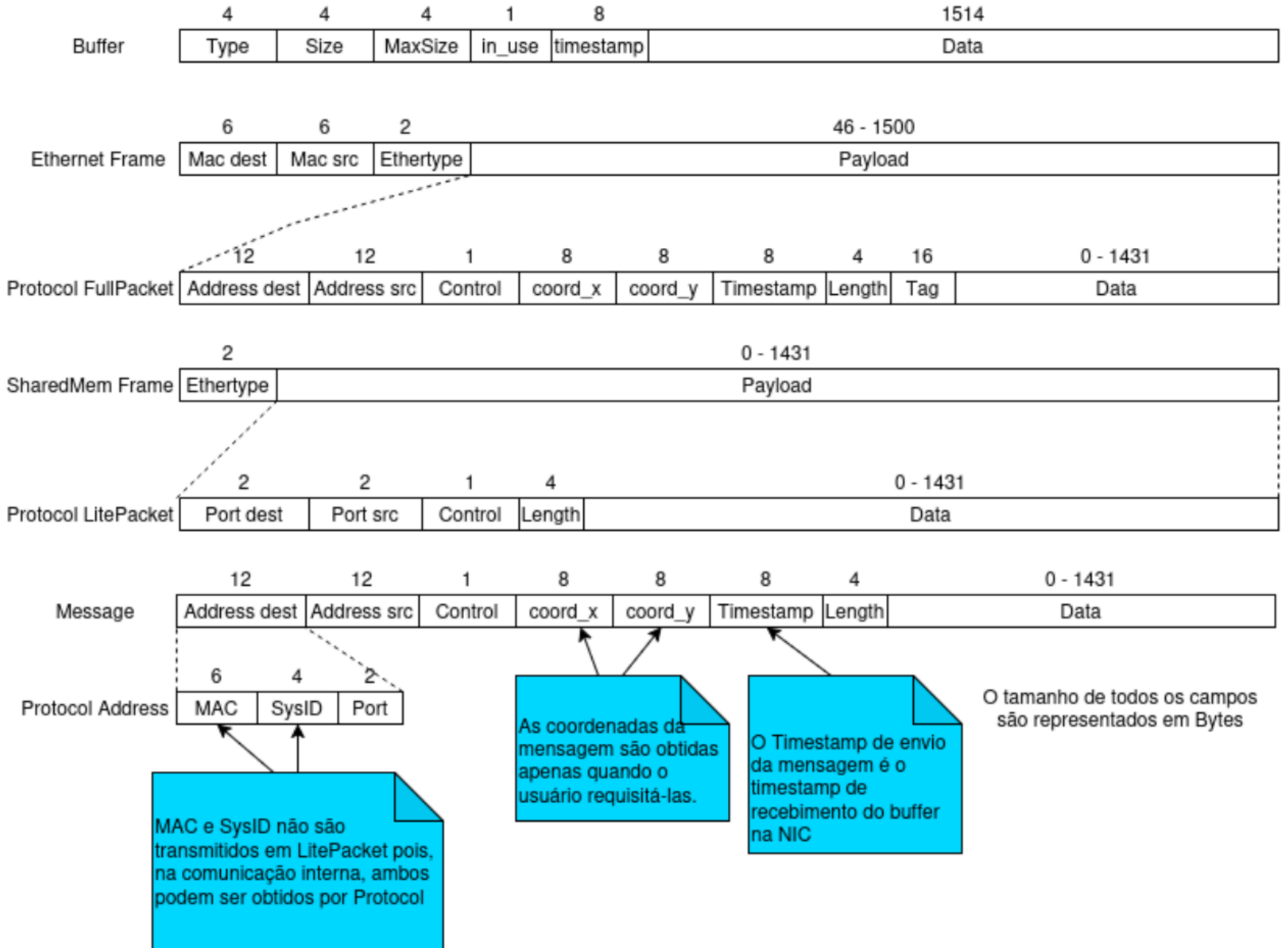
Outros Diagramas:

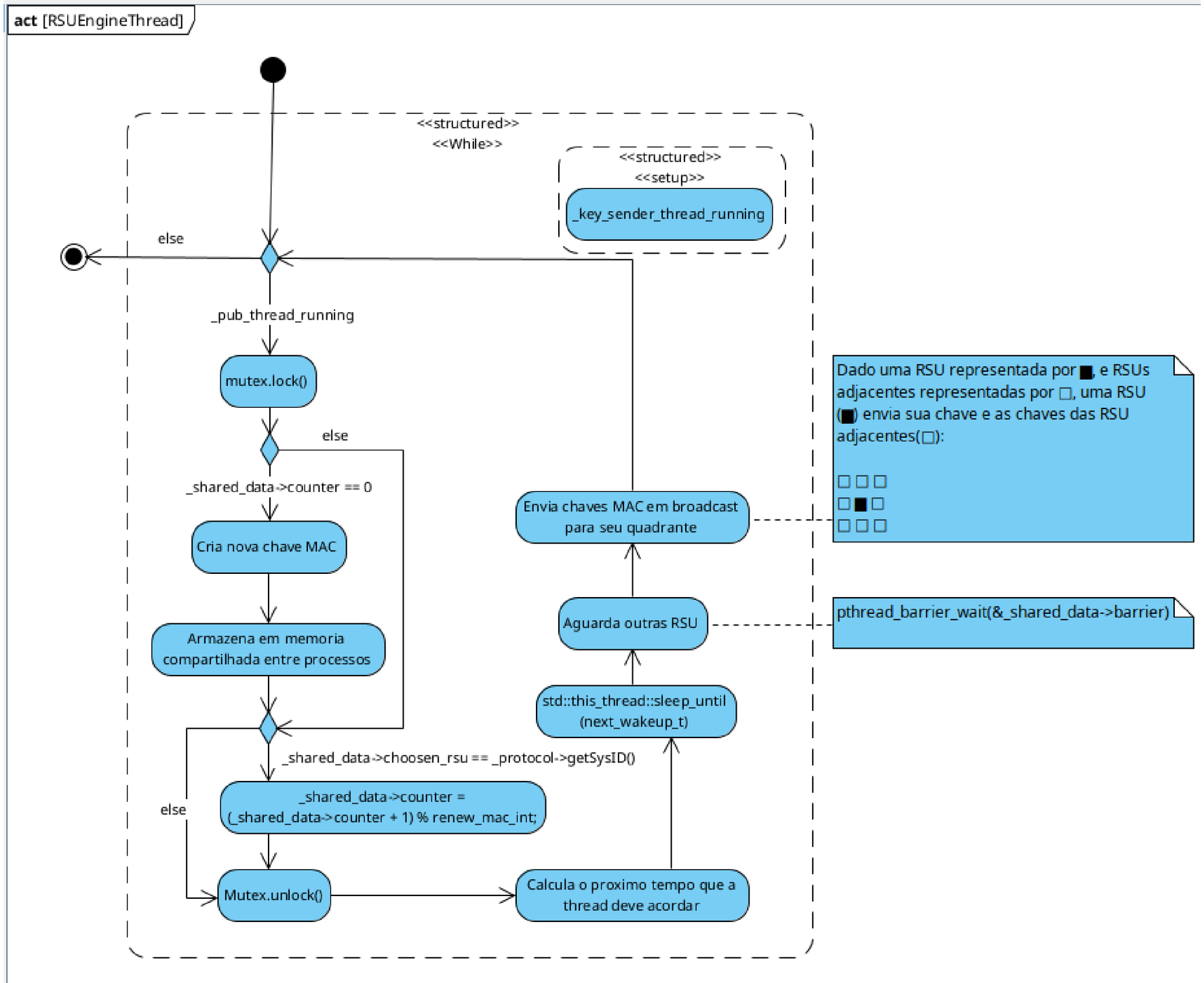

6
6
2
46 - 1500
Ethernet Frame
Mac dest
Mac src
Ethertype
Payload
12
12
1
8
8
8
4
16
0 - 1431
Protocol FullPacket
Address dest
Address src
Control
coord_x
coord_y
Timestamp
Length
Tag
Data
2
0 - 1431
SharedMem Frame
Ethertype
Payload
2
2
1
4
0 - 1431
Protocol LitePacket
Port dest
Port src
Control
Length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

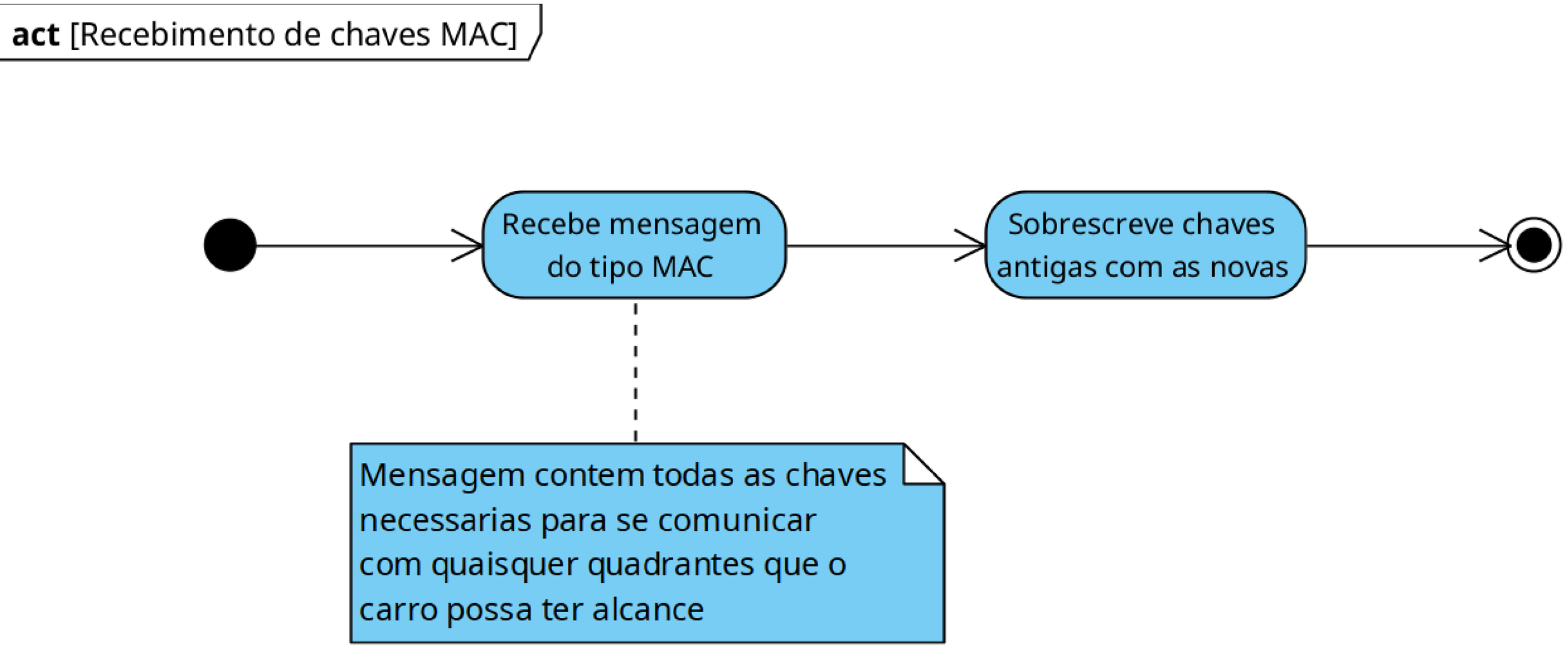
Buffer.data armazena Ethernet::Frame ou SharedMem::Frame

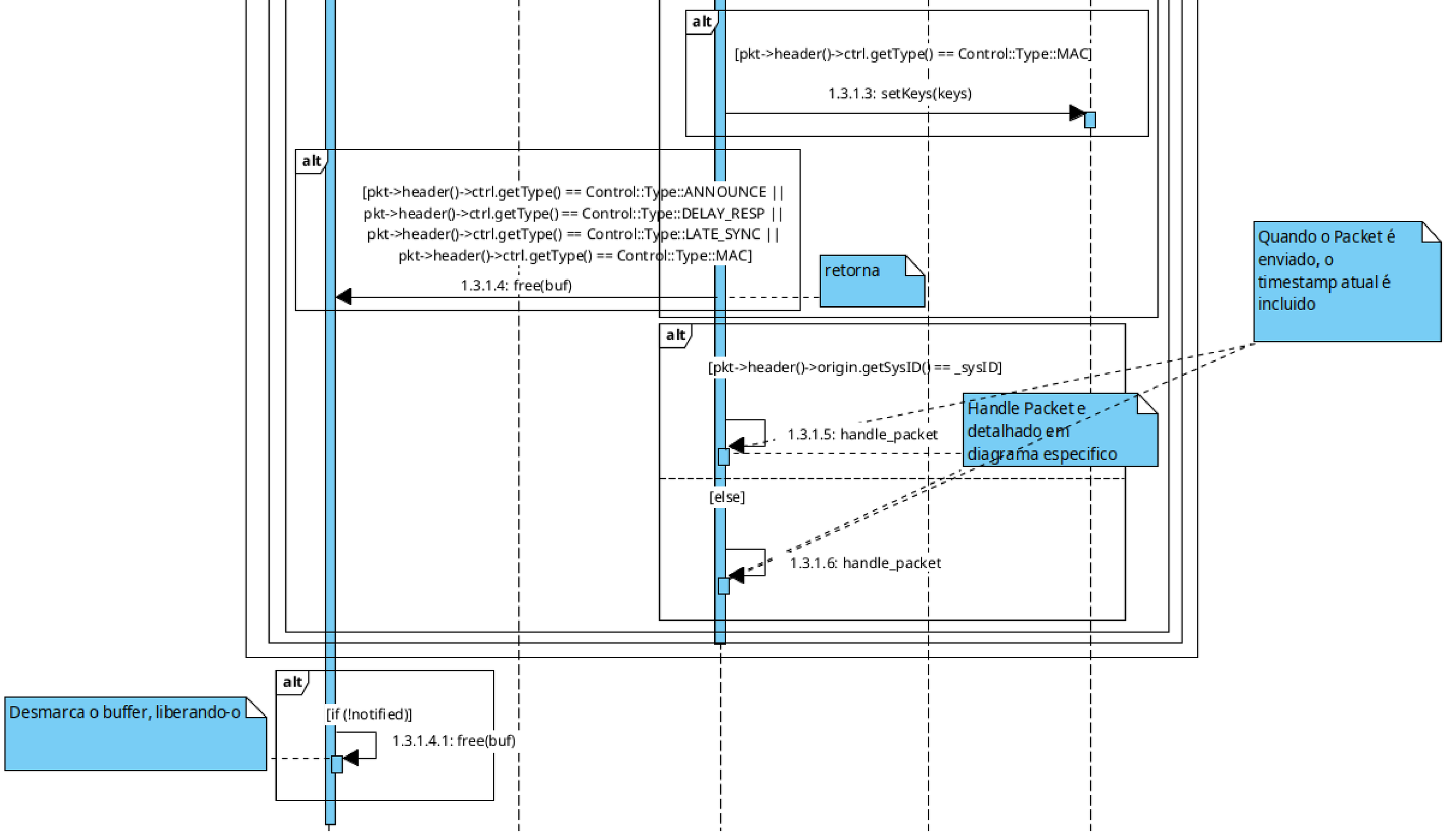


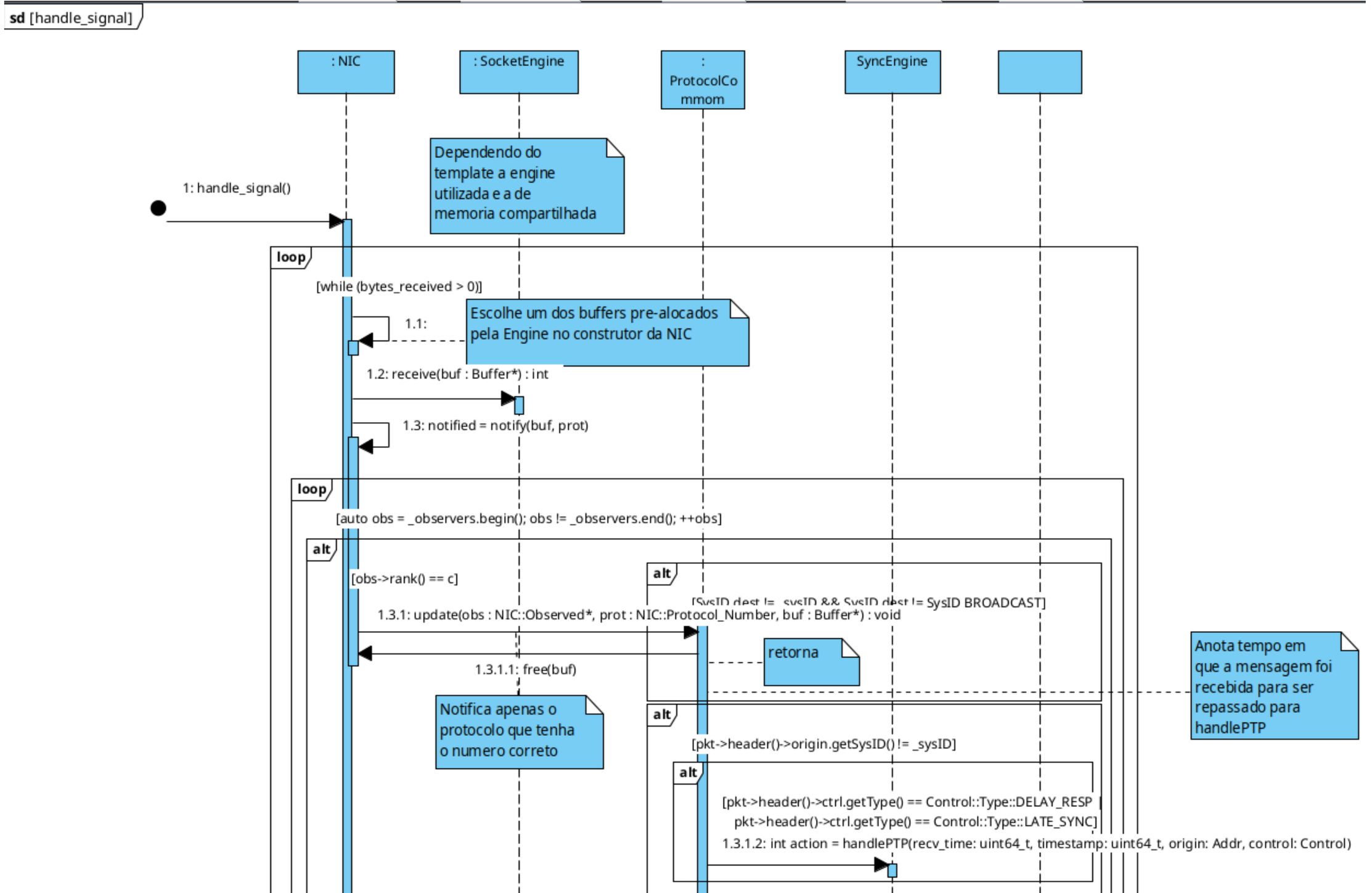

12
12
1
8
8
8
4
0 - 1431
Message
Address dest
Address src
Control
coord_x
coord_y
Timestamp
Length
Data
6
4
2
Protocol Address
MAC
SysID
Port
MAC e SysID não são transmitidos em LitePacket pois, na comunicação interna, ambos podem ser obtidos por Protocol
As coordenadas da mensagem são obtidas apenas 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

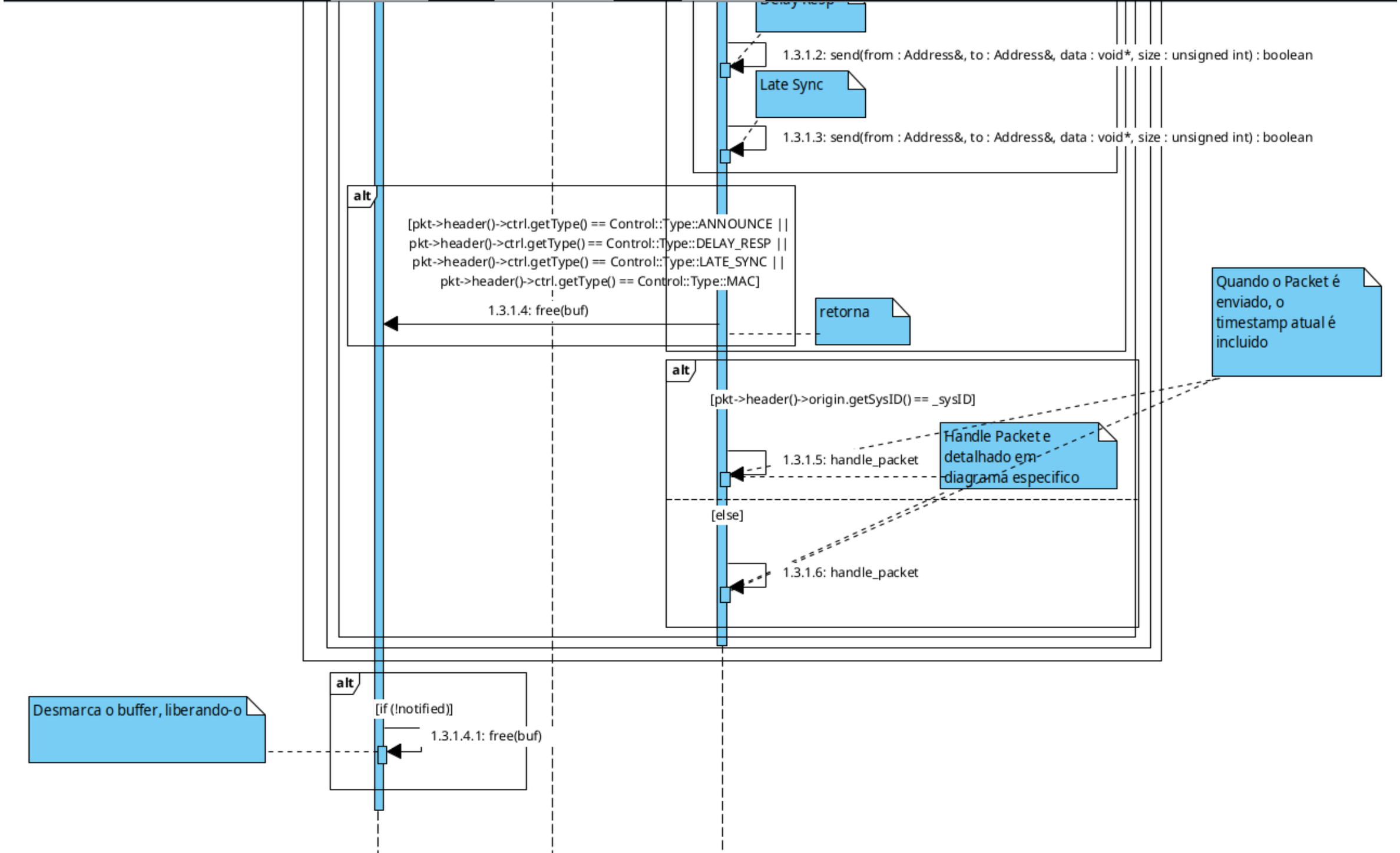

Buffer
4
4
4
1
8
1514
Type
Size
MaxSize
in_use
timestamp
Data
Ethernet Frame
6
6
2
46 - 1500
Mac dest
Mac src
Ethertype
Payload
Protocol FullPacket
12
12
1
8
8
8
4
16
0 - 1431
Address dest
Address src
Control
coord_x
coord_y
Timestamp
Length
Tag
Data
SharedMem Frame
2
0 - 1431
Ethertype
Payload
Protocol 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

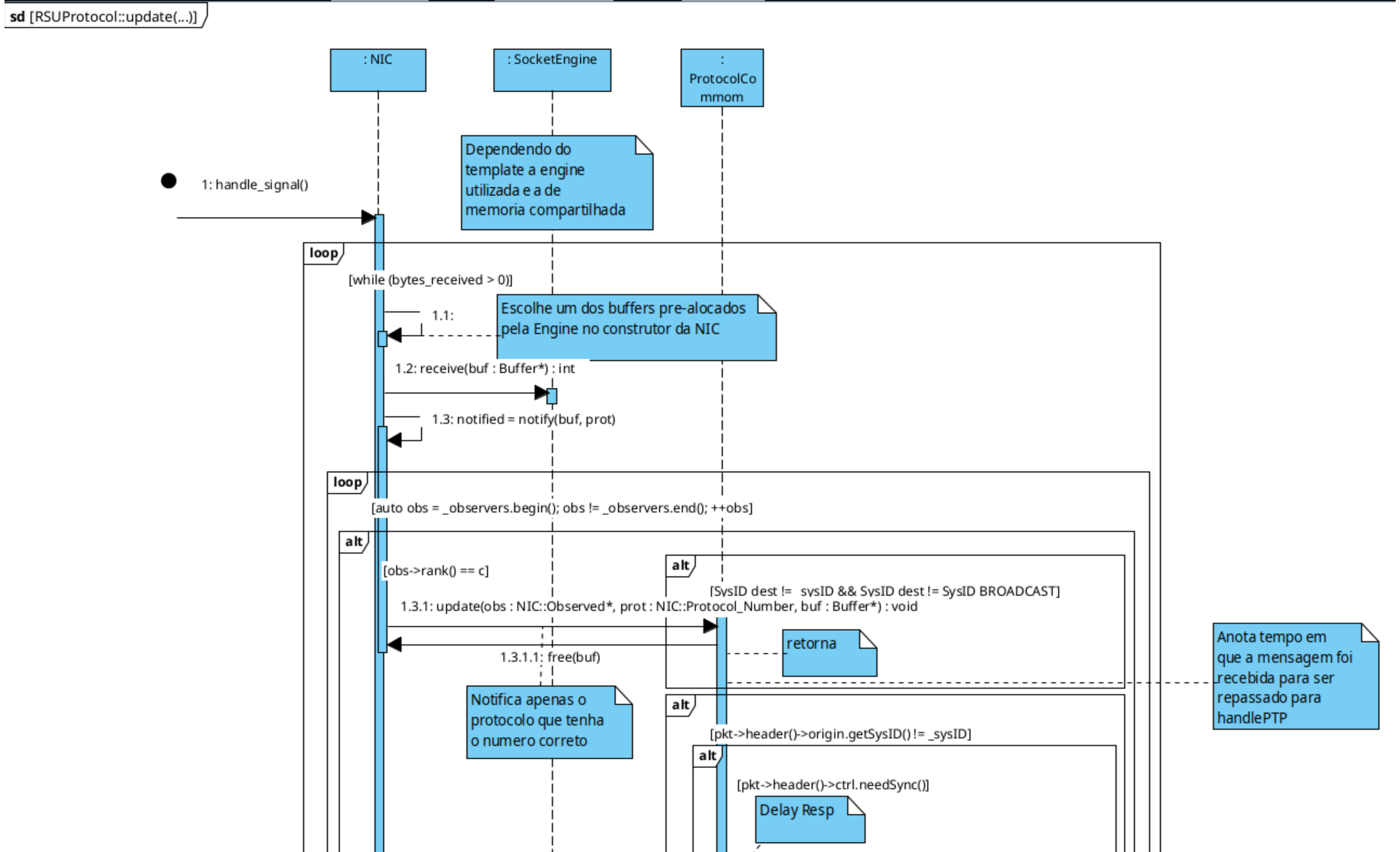

act [RSUEngineThread]
<<structured>>
<<While>>
<<structured>>
<<setup>>
_key_sender_thread_running
else
_pub_thread_running
mutex.lock()
else
_shared_data->counter == 0
Cria nova chave MAC
Armazena em memoria compartilhada entre processos
_shared_data->choosen_rsu == _protocol->getSysID()
else
_shared_data->counter = (_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)

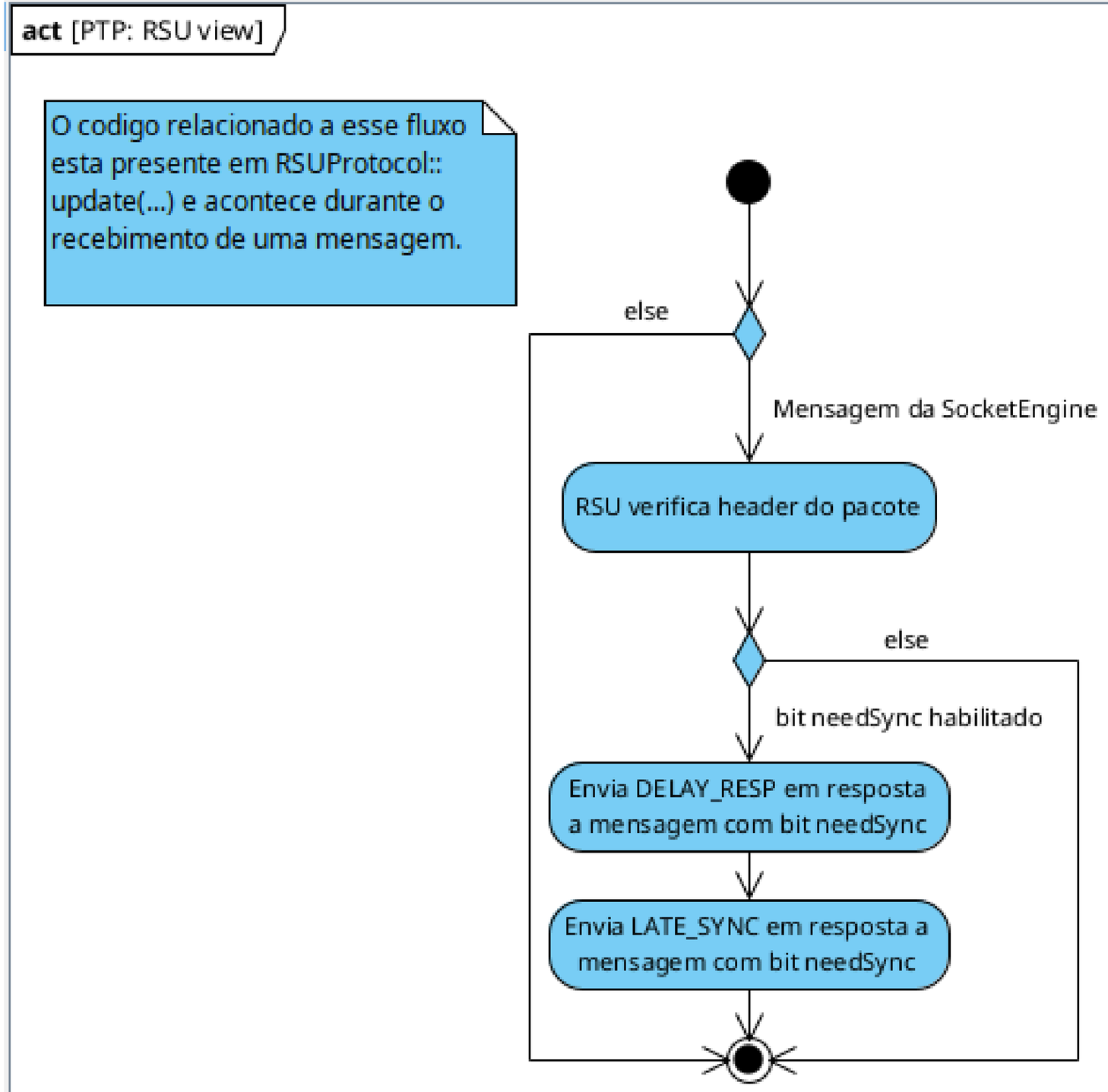

act [Recebimento de chaves MAC]
Recebe mensagem do tipo MAC
Sobrescreve chaves antigas com as novas
Mensagem contem todas as chaves necessarias para se comunicar com quaisquer quadrantes que o carro possa ter alcance

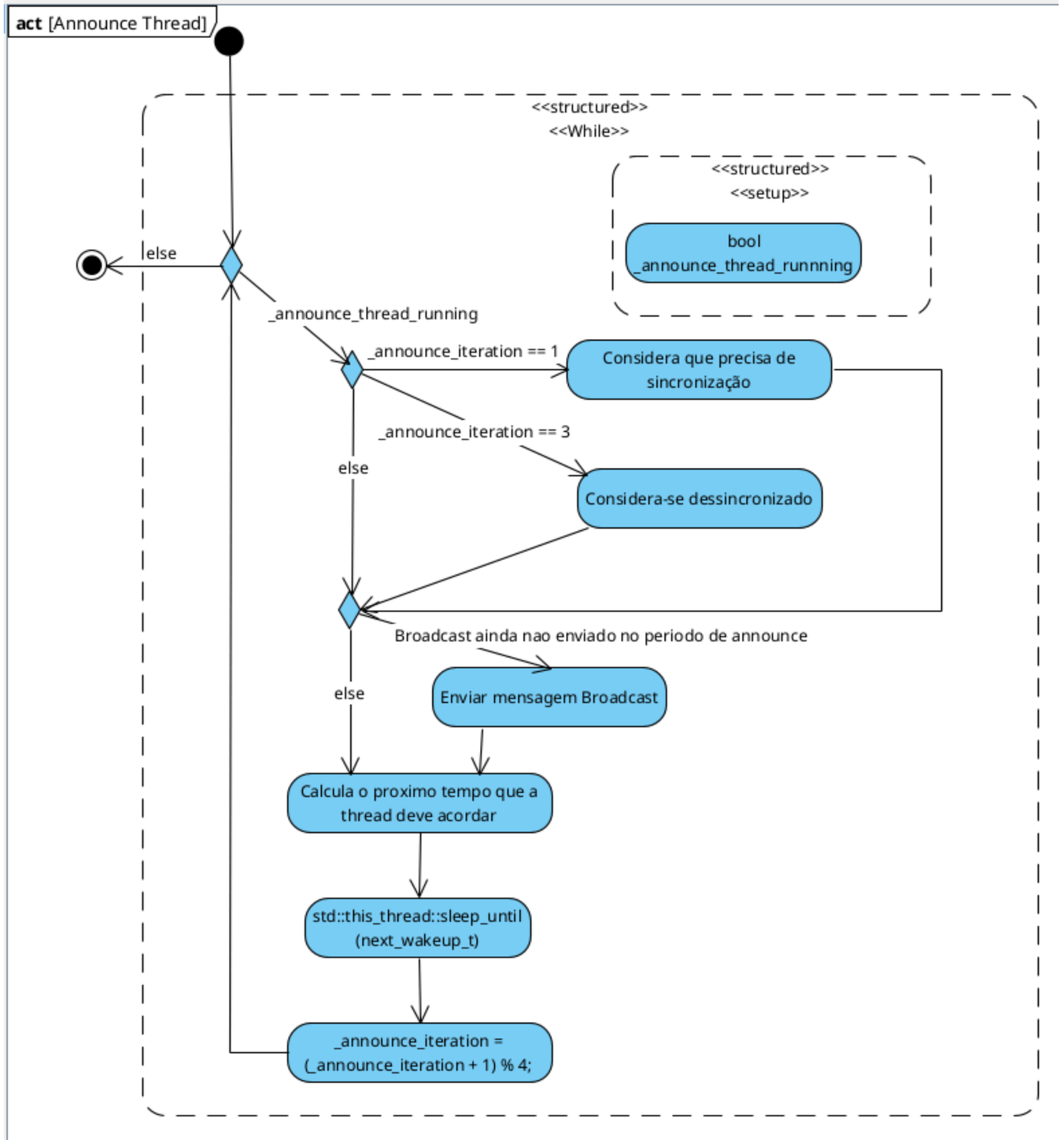

alt
[pkt->header()->ctrl.getType() == Control::Type::MAC]
1.3.1.3: setKeys(keys)
alt
[pkt->header()->ctrl.getType() == Control::Type::ANNOUNCE ||
pkt->header()->ctrl.getType() == Control::Type::DELAY_RESP ||
pkt->header()->ctrl.getType() == Control::Type::LATE_SYNC ||
pkt->header()->ctrl.getType() == 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)

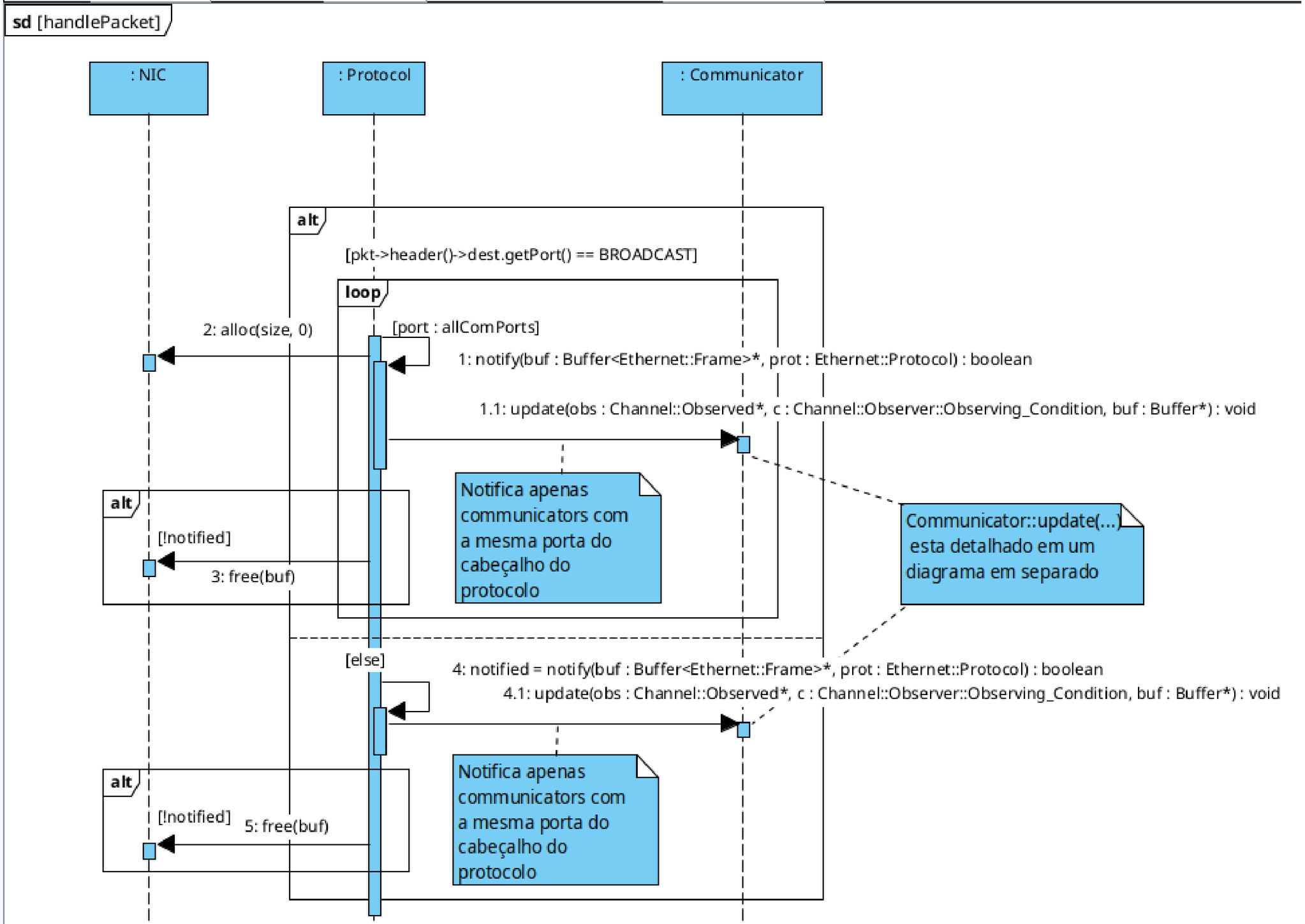

sd [handle_signal]
: NIC
: SocketEngine
: ProtocolCommom
SyncEngine
Dependendo do template a engine utilizada e a de memoria compartilhada
1: handle_signal()
loop
[while (bytes_received > 0)]
1.1:
Escolhe um dos buffers pre-alocados pela Engine no construtor da NIC
1.2: receive(buf : Buffer*) : int
1.3: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

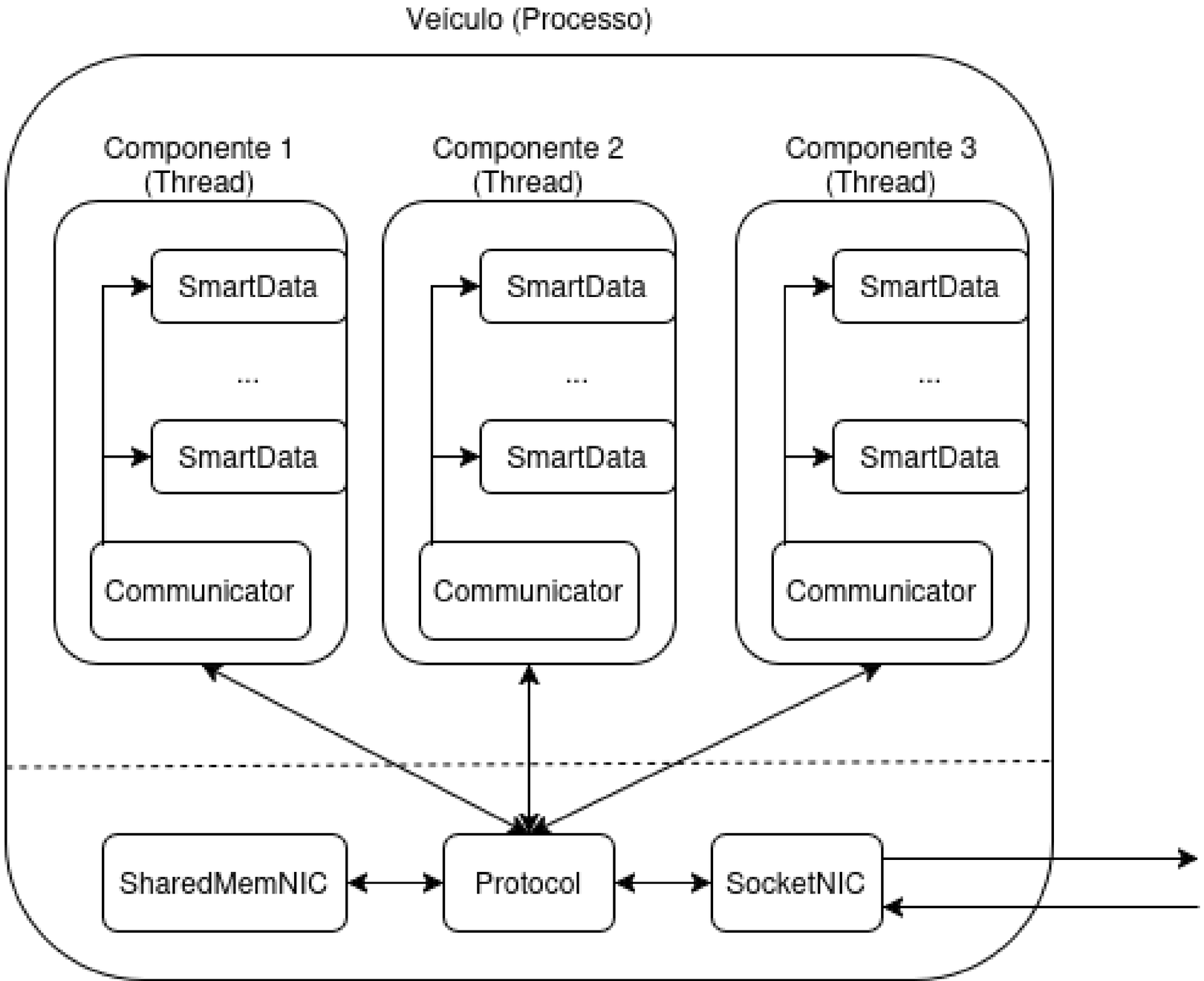

sd [RSUProtocol::update(...)]
: NIC
: SocketEngine
: ProtocolCommom
Dependendo do template a engine utilizada e a de memoria compartilhada
1: handle_signal()
loop
[while (bytes_received > 0)]
1.1:
Escolhe um dos buffers pre-alocados pela Engine no construtor da NIC
1.2: receive(buf : Buffer*) : int
1.3: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

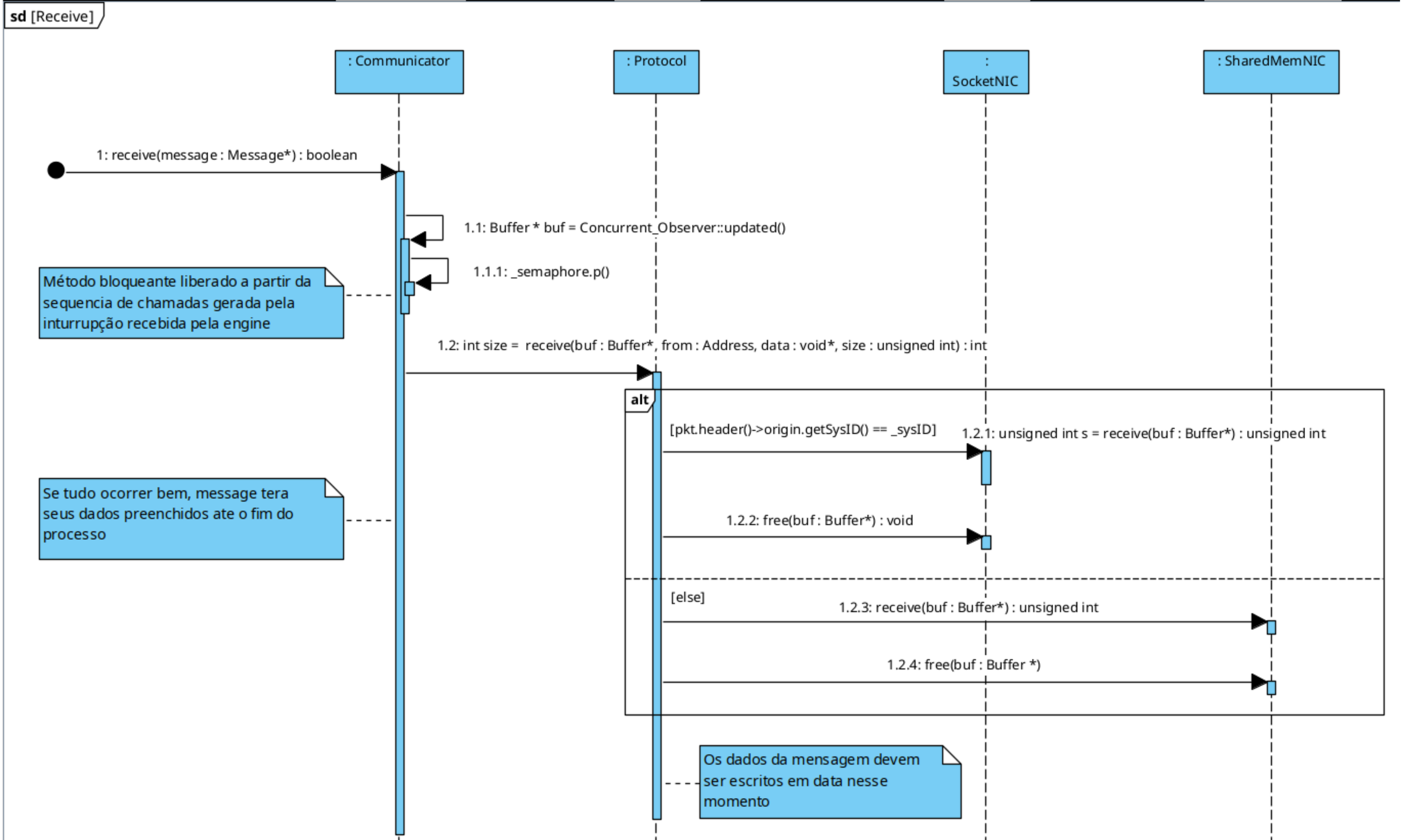

act [Announce Thread]
<<structured>>
<<While>>
<<structured>>
<<setup>>
bool
_announce_thread_runnning
else
_announce_thread_running
_announce_iteration == 1
Considera que precisa de
sincronização
_announce_iteration == 3
else
Considera-se dessincronizado
Broadcast ainda nao enviado no periodo de announce
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;

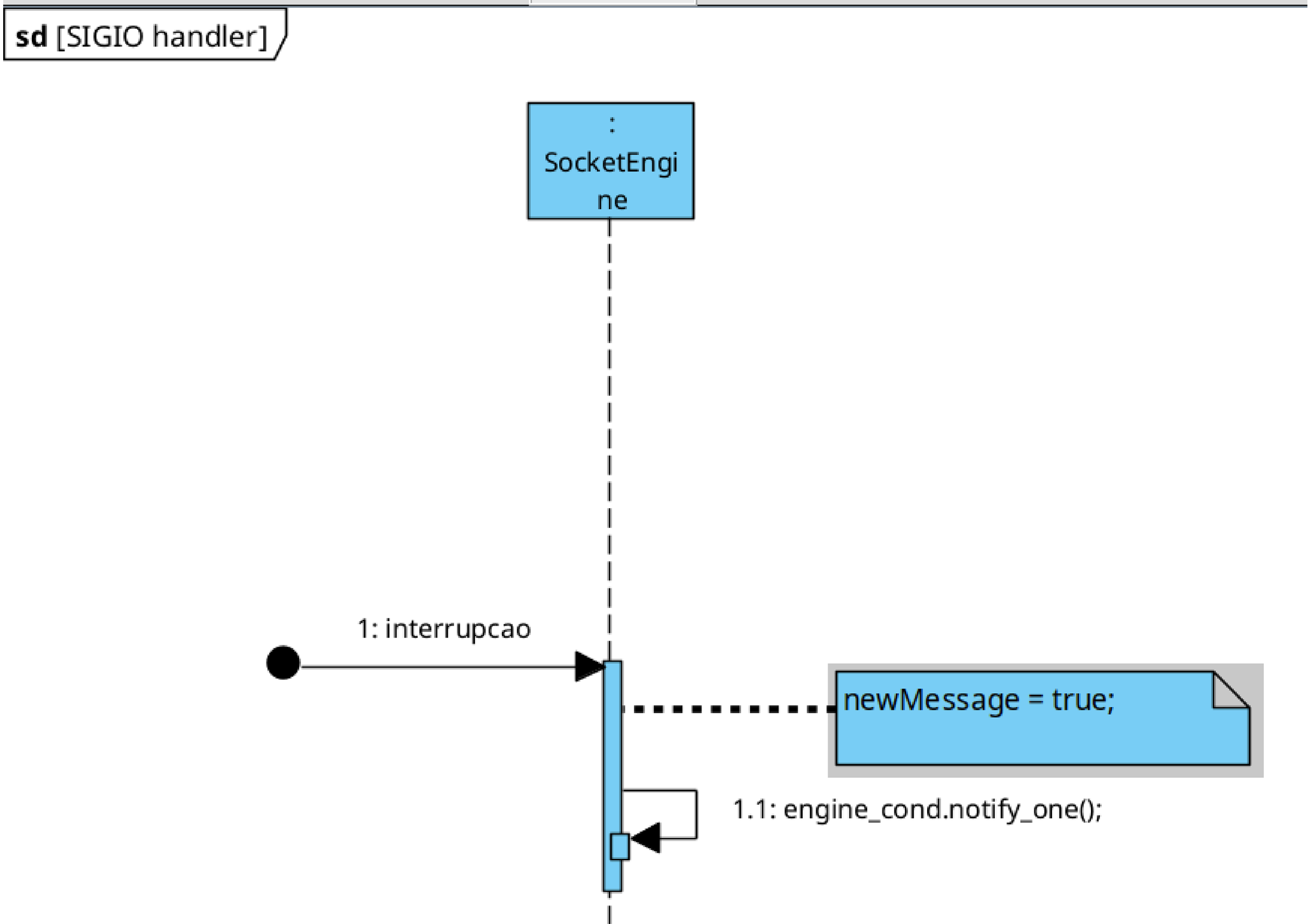

sd [handlePacket]
: NIC
: Protocol
: Communicator
alt
[pkt->header()->dest.getPort() == BROADCAST]
loop
2: alloc(size, 0)
[port : allComPorts]
1: notify(buf : Buffer<Ethernet::Frame>*, prot : Ethernet::Protocol) : boolean
1.1: update(obs : Channel::Observed*, c : Channel::Observer::Observing_Condition,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

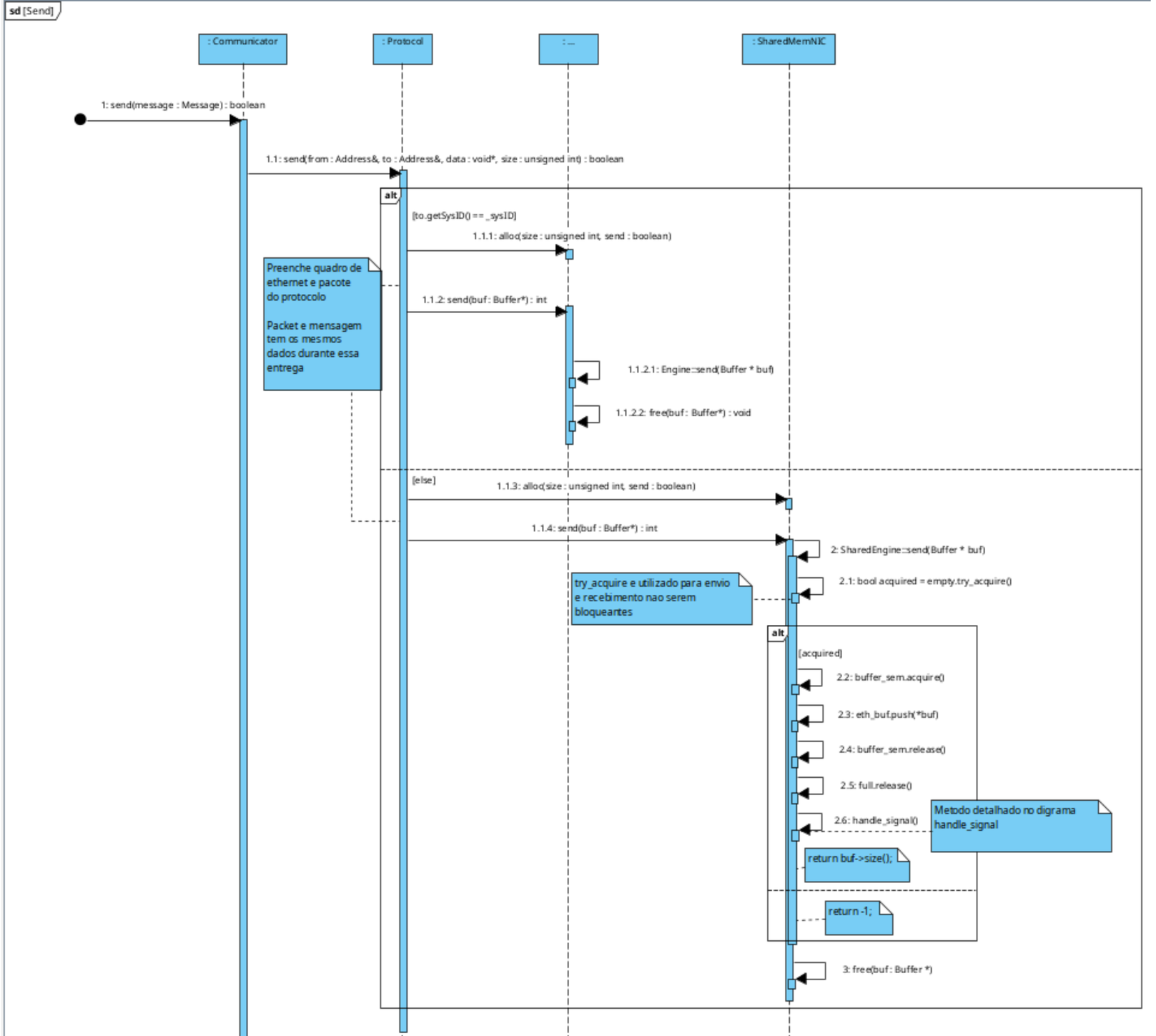
Estrutura do Veículo

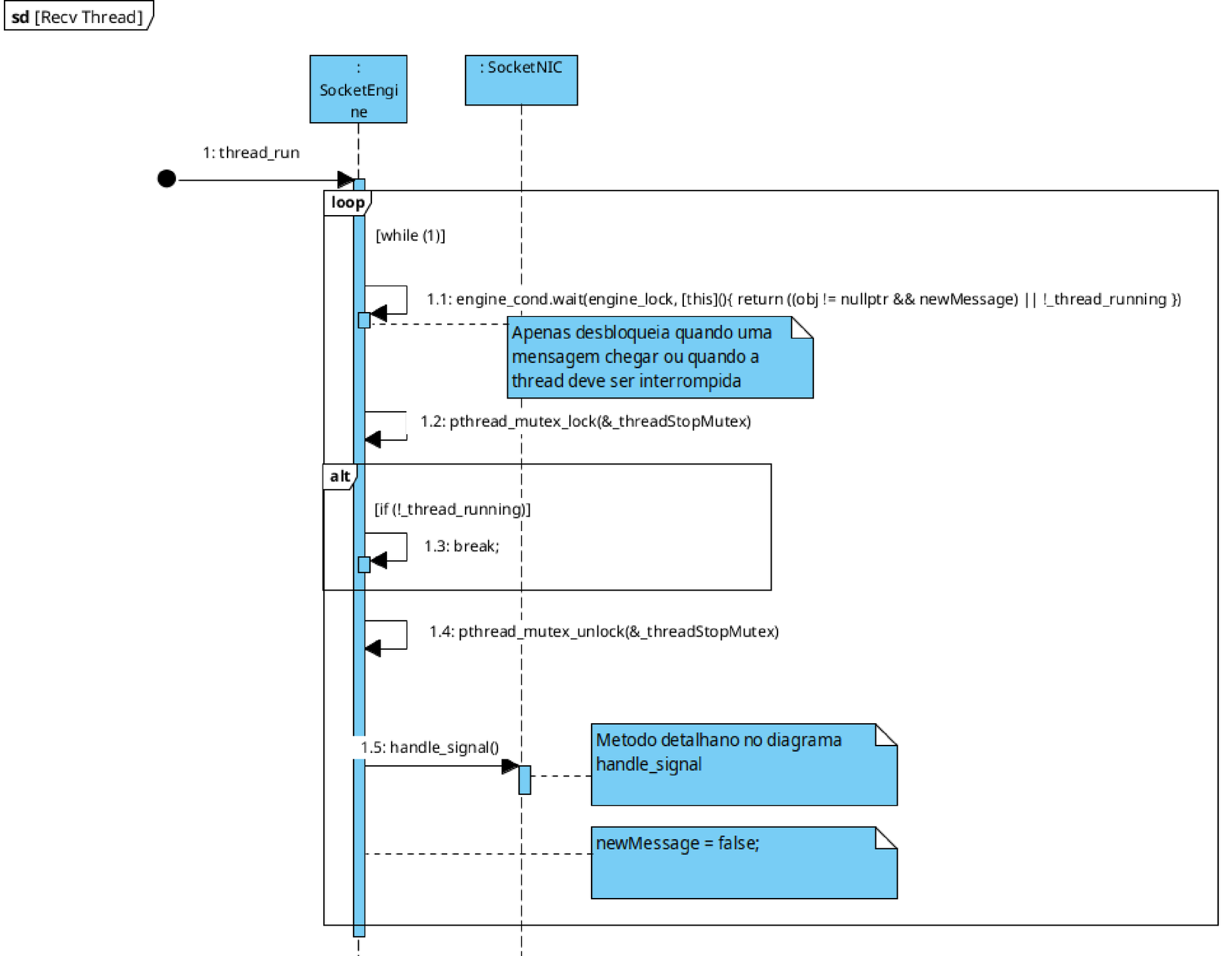


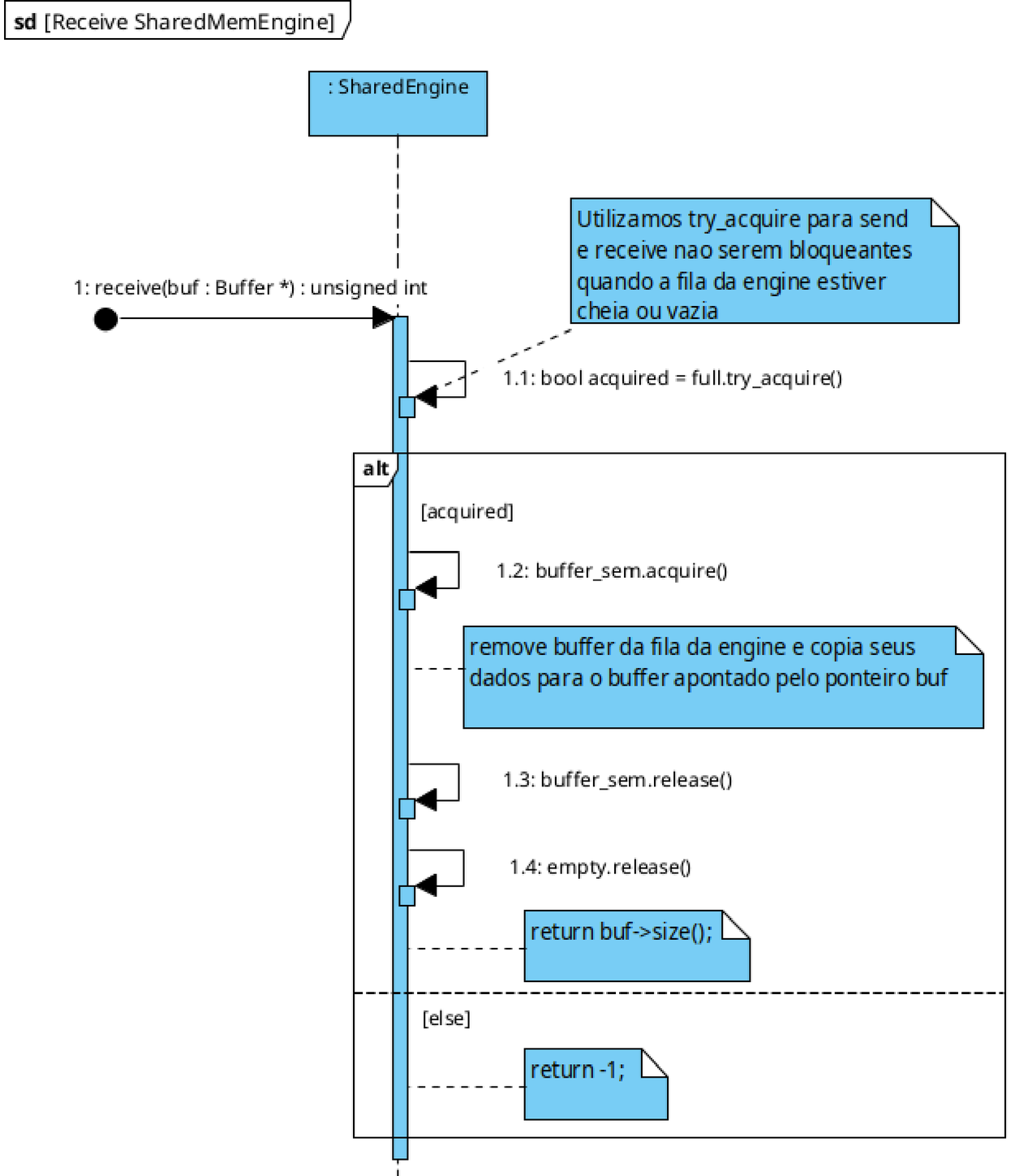
Diagramas de Sequência

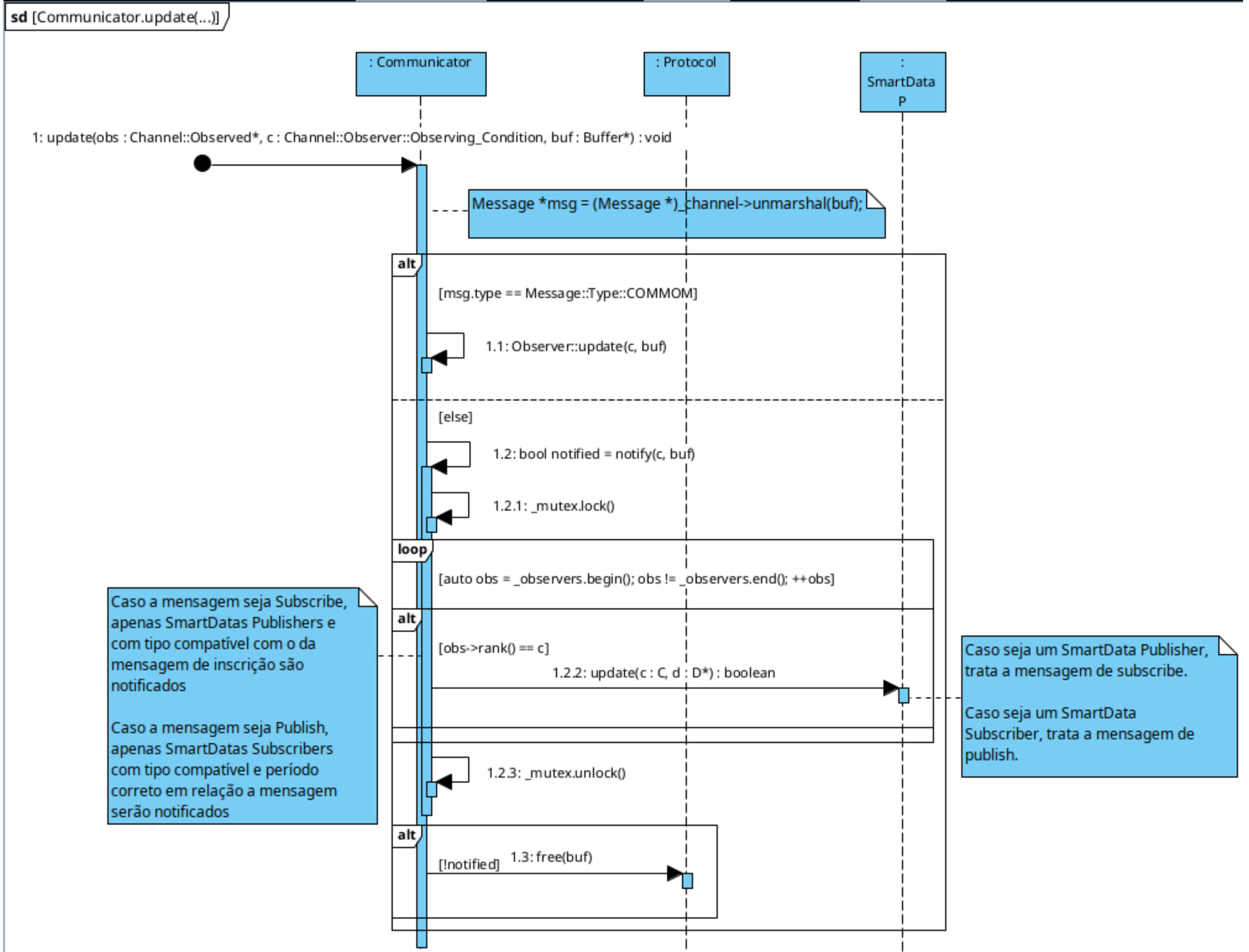

sd [Receive]
: Communicator
: Protocol
: SocketNIC
: SharedMemNIC
1: receive(message : Message*) : boolean
1.1: Buffer * buf = Concurrent_Observer::updated()
1.1.1: _semaphore.p()
Método bloqueante liberado a partir da sequencia de chamadas gerada pela inturrupção recebida pela engine
1.2: int size =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

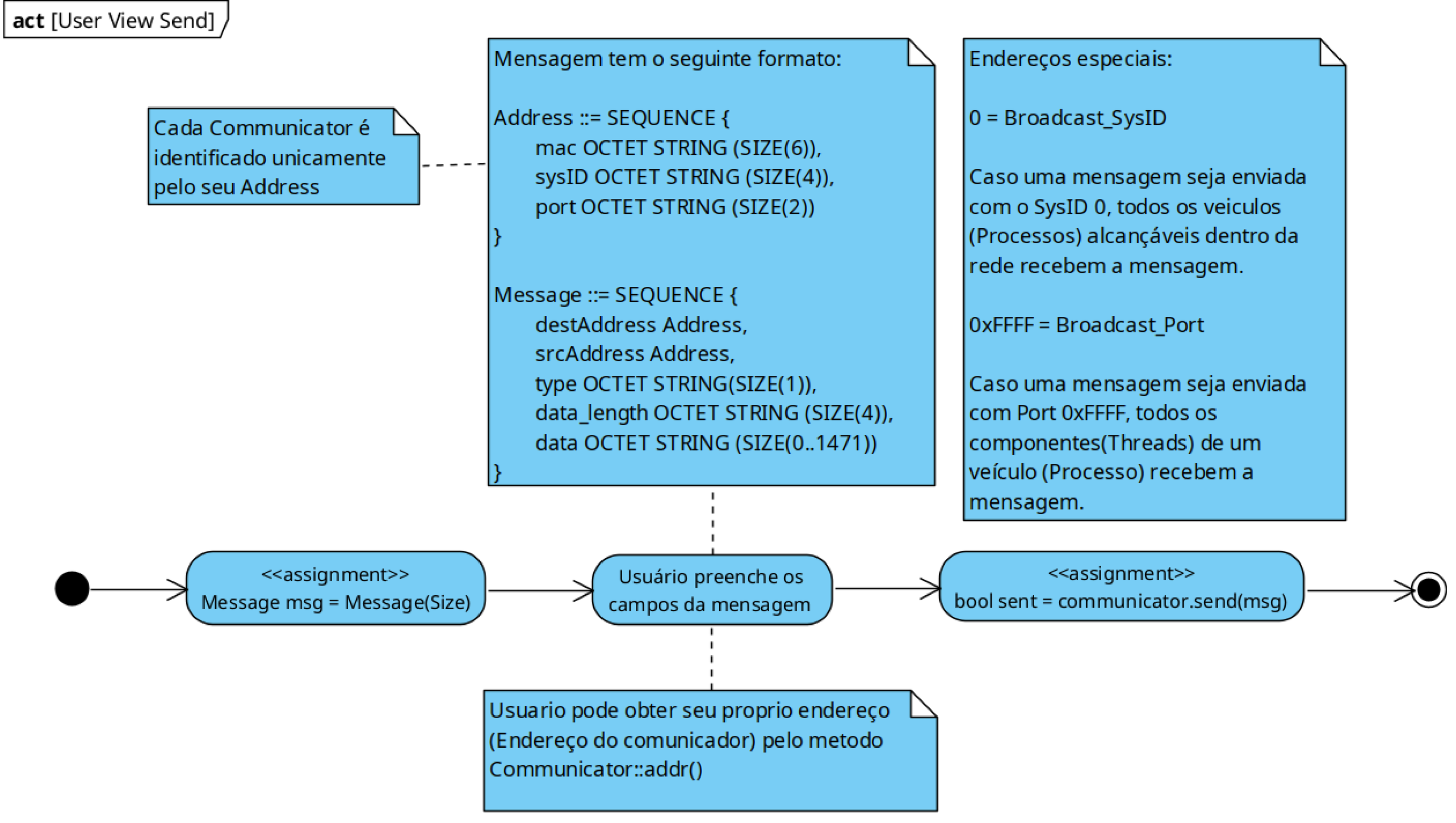

sd [Send]
: Communicator
: Protocol
: ...
: SharedMemNIC
1: send(message : Message) : boolean
1.1: send(from : Address&, to : Address&, data : void*, size : unsigned int) : boolean
alt
[to.getSysID() == _sysID]
1.1.1: alloc(size : unsigned int, send : boolean)
Preenche quadro de ethernet e pacote do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

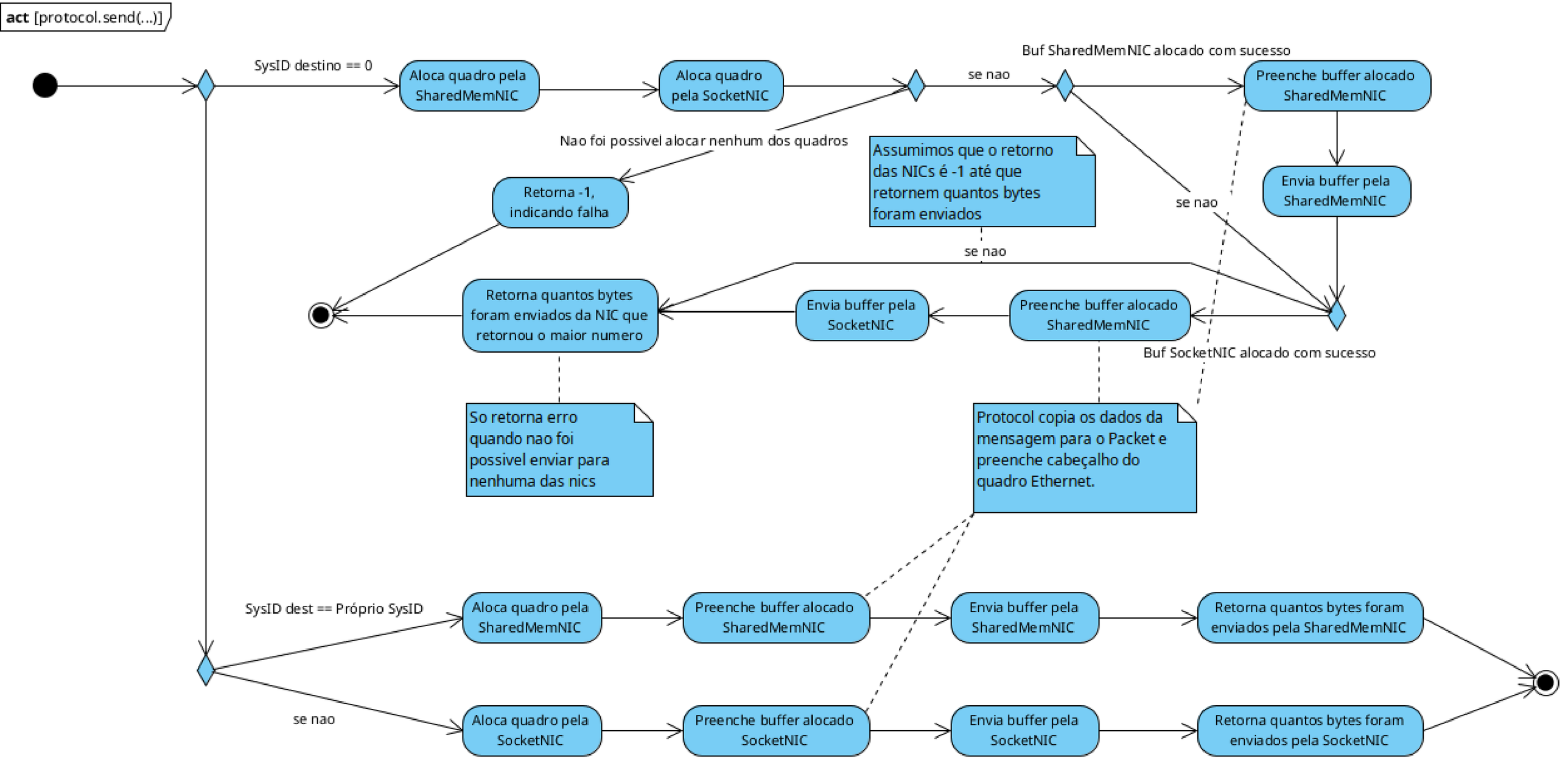

sd [Recv Thread]
: SocketEngine
: SocketNIC
1: thread_run
loop
[while (1)]
1.1: engine_cond.wait(engine_lock, [this](){ return ((obj != nullptr && newMessage) || !_thread_running })
Apenas desbloqueia quando uma mensagem chegar ou quando a thread deve ser interrompida
1.2: pthread_mutex_lock(&_threadStopMutex)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

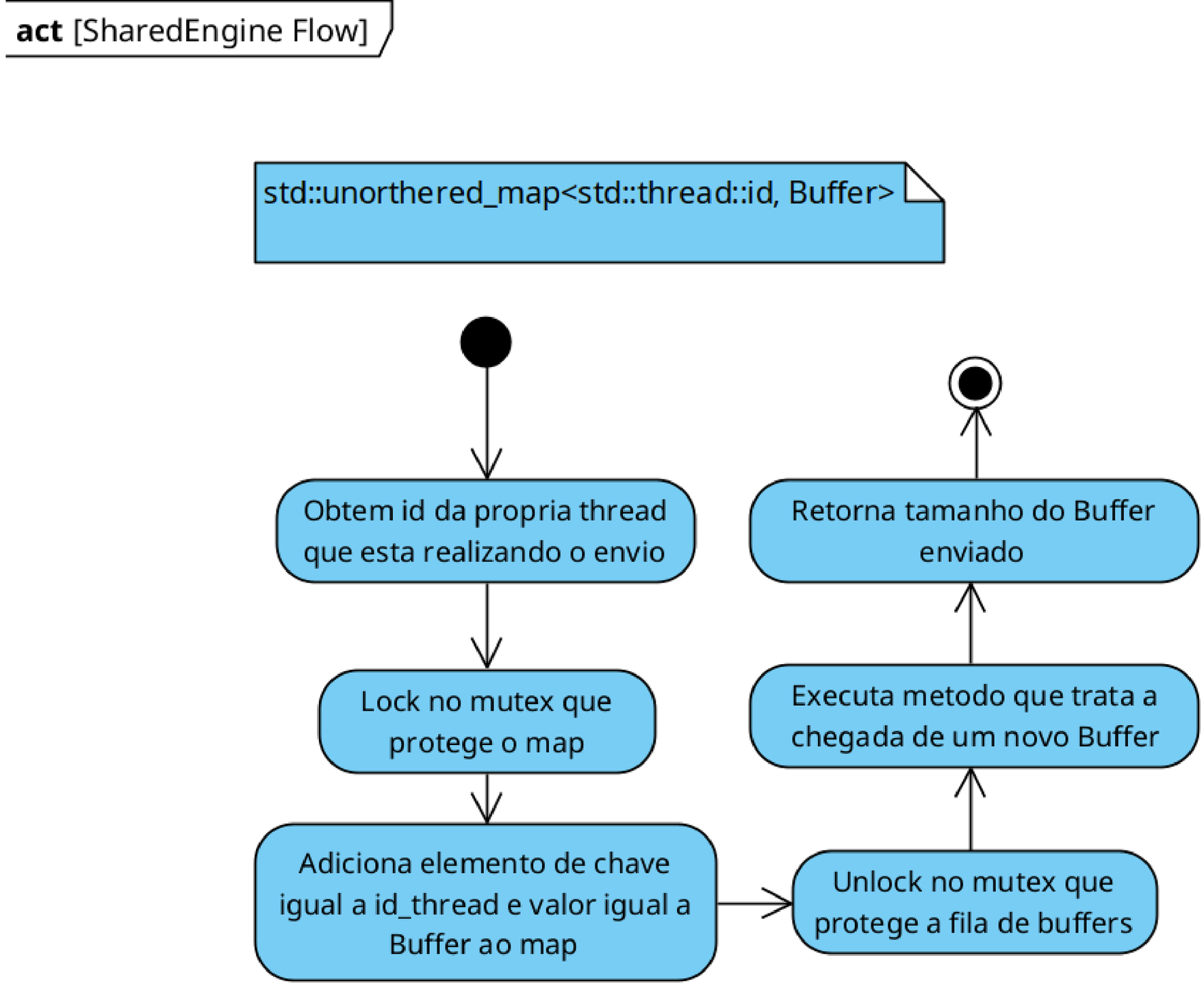

sd [Receive SharedMemEngine]
: SharedEngine
1: receive(buf : Buffer *) : unsigned int
Utilizamos try_acquire para send e receive nao serem bloqueantes quando a fila da engine estiver cheia ou vazia
1.1: bool acquired = full.try_acquire()
alt
[acquired]
1.2: buffer_sem.acquire()
remove buffer da fila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

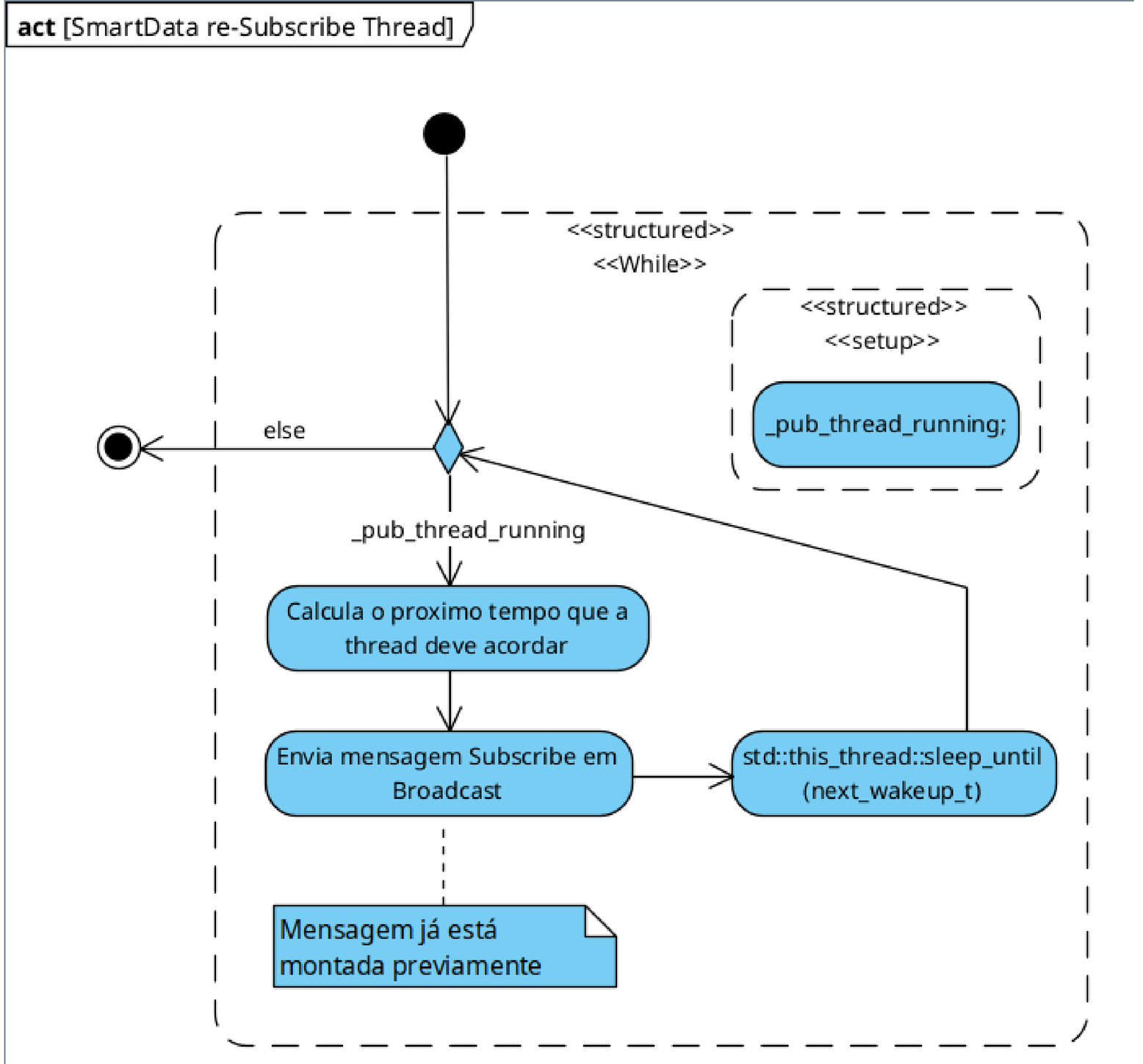

sd [Communicator.update(...)]
: Communicator
: Protocol
: SmartData P
1: update(obs : Channel::Observed*, c : Channel::Observer::Observing_Condition, buf : Buffer*) : void
Message *msg = (Message *)_channel->unmarshal(buf);
alt
[msg.type == Message::Type::COMMOM]
1.1 : Observer::update(c, buf)
[else]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

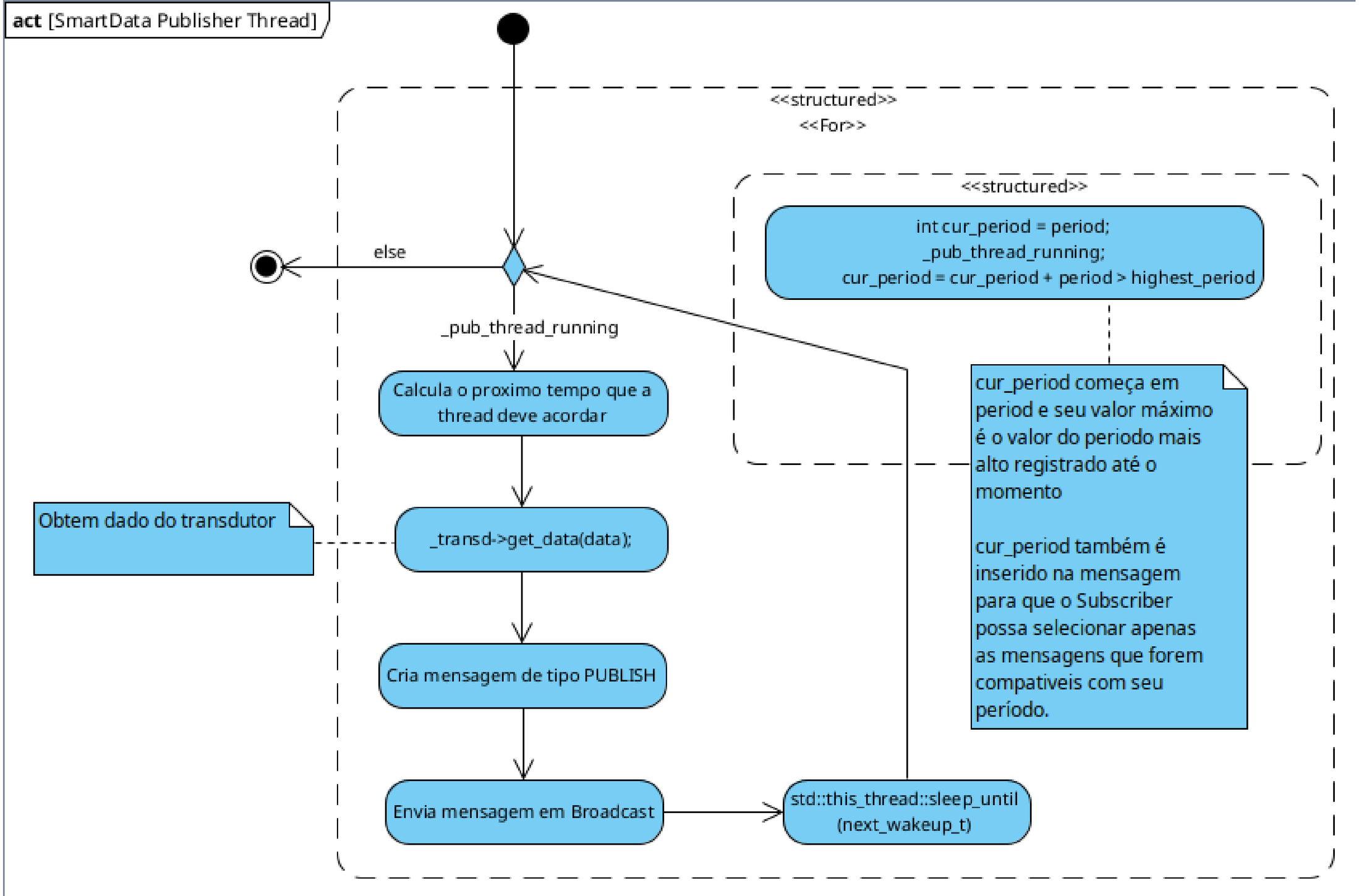
Diagramas de Atividade

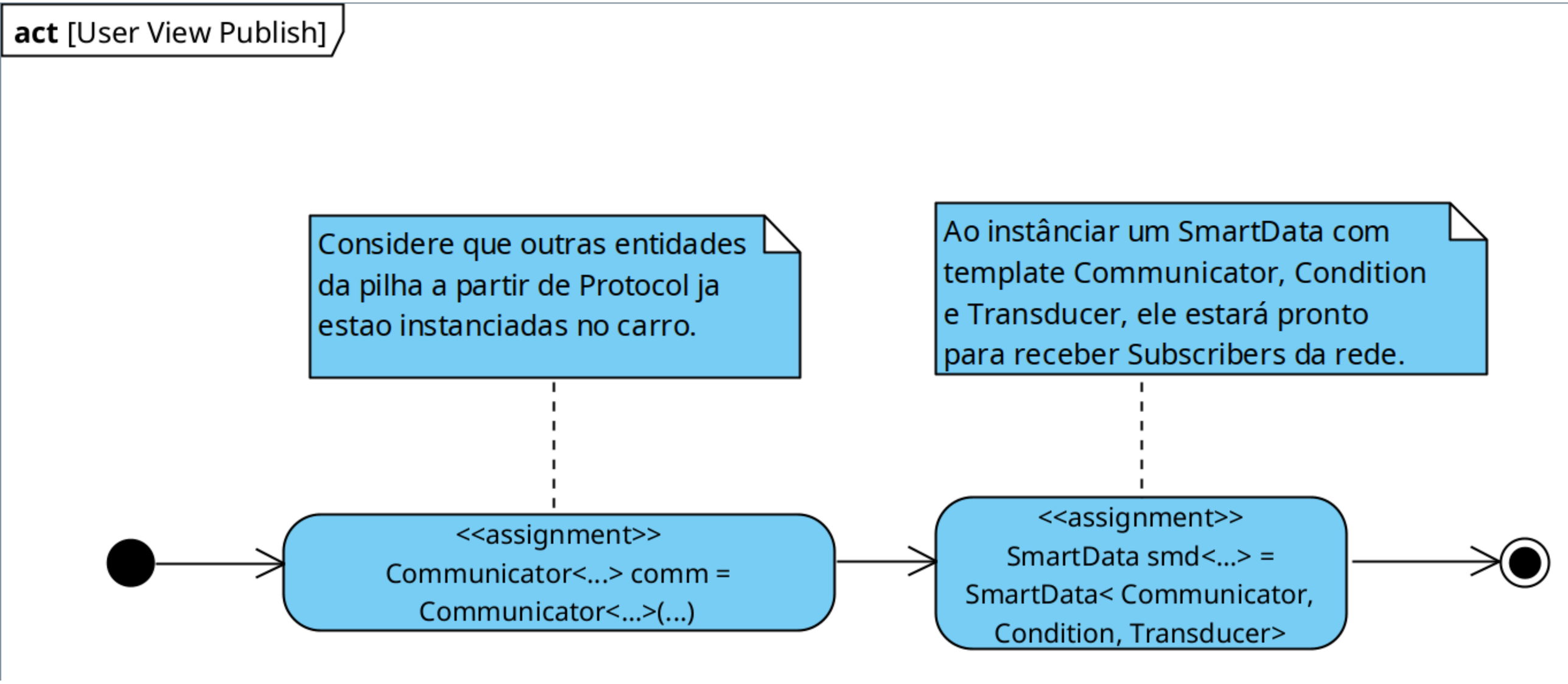

act [User View Send]
Cada Communicator é identificado unicamente pelo seu Address
Mensagem tem o seguinte formato:
Address ::= SEQUENCE {
mac OCTET STRING (SIZE(6)),
sysID OCTET STRING (SIZE(4)),
port OCTET STRING (SIZE(2))
}
Message ::= SEQUENCE {
destAddress Address,
srcAddress Address,
type OCTET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

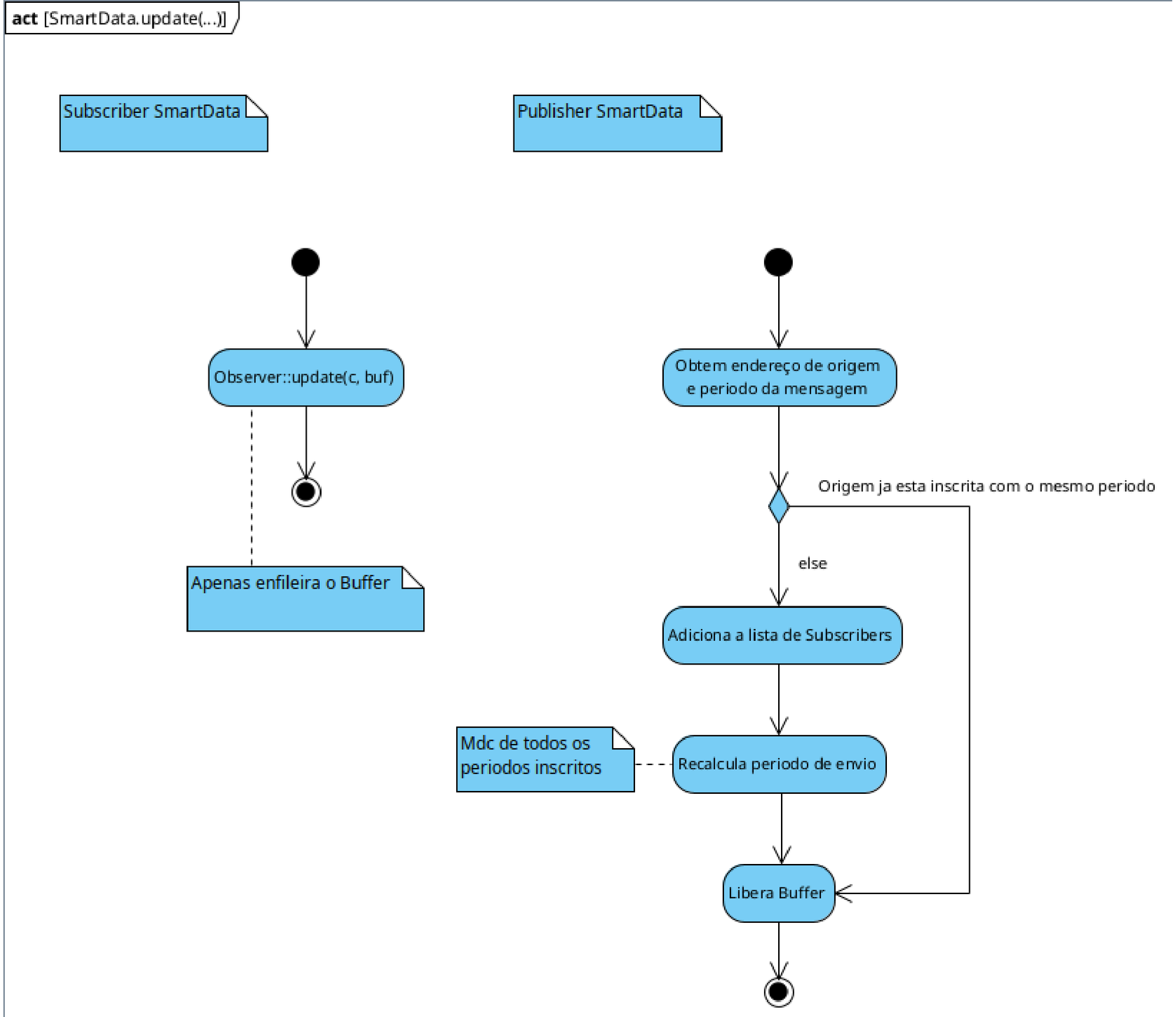

act [protocol.send(...)]
SysID destino == 0
Aloca quadro pela SharedMemNIC
Aloca quadro pela SocketNIC
Nao foi possivel alocar nenhum dos quadros
Retorna -1, indicando falha
se nao
Buf SharedMemNIC alocado com sucesso
Preenche buffer alocado SharedMemNIC
Envia buffer pela SharedMemNIC
se nao
Assumimos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

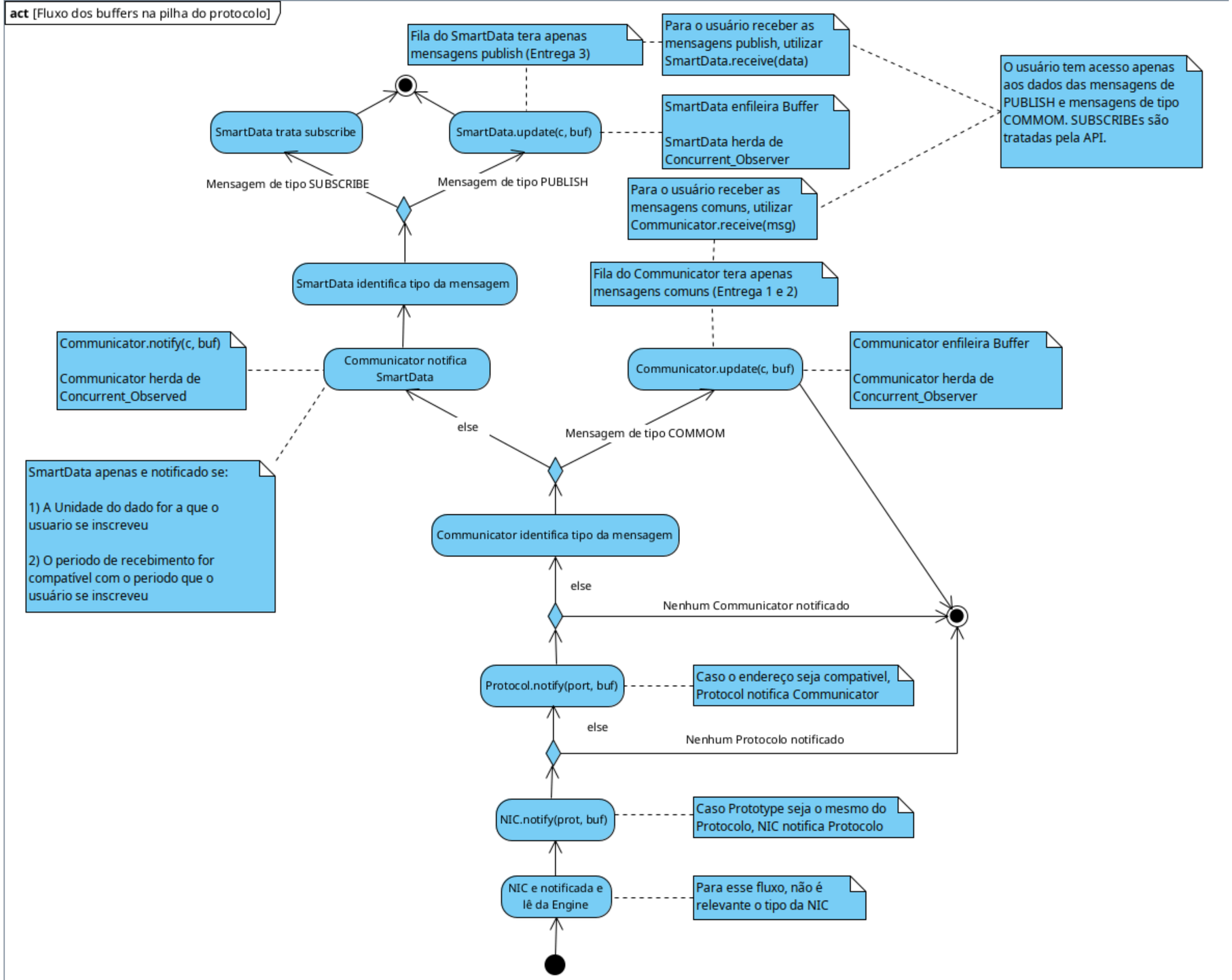

act [SharedEngine Flow]
std::unorthered_map<std::thread::id, Buffer>
Obtem id da propria thread que esta realizando o envio
Lock no mutex que protege o map
Adiciona elemento de chave igual a id_thread e valor igual a Buffer ao map
Unlock no mutex que protege a fila de buffers
Executa metodo que trata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

Sincronização Temporal: Detalhes do Cálculo

O cálculo do offset no veículo depende da correlação de quatro timestamps.

- **Timestamps:**
  - **T1:** Tempo de envio da LATE_SYNC pela RSU.
  - **T2:** Tempo de recebimento da LATE_SYNC pelo veículo.
  - **T3:** Tempo de envio da ANNOUNCE original pelo veículo.
  - **T4:** Tempo de recebimento da ANNOUNCE pela RSU.
- **Mecanismo de Correlação:** O veículo, ao enviar ANNOUNCE, armazena T3. A RSU inclui T3 no payload de suas respostas. Ao receber LATE_SYNC e DELAY_RESP, o veículo usa o T3 do payload para consultar T4 em um mapa interno.
- **Cálculo:** Com os quatro valores, as fórmulas padrão do PTP são aplicadas para calcular o atraso de propagação e o offset. O offset é então aplicado ao relógio local.

Comunicação Segura: Coordenação de RSUs

A geração e distribuição sincronizada de chaves MAC entre as RSUs (processos distintos) é realizada com memória compartilhada.

- **Estrutura SharedData:** Um segmento de memória compartilhada contém:
  - **pthread_mutex_t:** Para serializar o acesso ao contador e à matriz de chaves.
  - **pthread_barrier_t:** Para sincronizar a execução das threads de todas as RSUs.
  - Um contador de ciclo para a renovação das chaves.
  - Uma matriz que armazena a chave MAC de cada RSU.
- **Fluxo de Sincronização:**
  - Cada RSU gera sua chave e a escreve em sua posição na matriz compartilhada.
  - Todas as RSUs bloqueiam na barreira.
  - Somente após todas as RSUs passarem pela barreira, elas prosseguem para ler as chaves de sua vizinhança da memória compartilhada e distribuí-las em broadcast.
- **Justificativa:** O uso de uma barreira POSIX garante que nenhuma RSU distribua chaves antigas enquanto outras já geraram as novas, mantendo a consistência do sistema de chaves em toda a rede.