Departamento de
Informática e Estatística
CTC • UFSC

Implementação da API de Comunicação: Sincronização Temporal

INE5424 - Sistemas Operacionais II

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

Objetivos da entrega 4

Requisito da entrega:

- Mensagens devem ser etiquetadas com um timestamp na origem.
- Sistemas autônomos (processos) devem sincronizar seus relógios para permitir a identificação inequívoca de mensagens (origem + timestamp).
- Componentes (threads) dentro de um mesmo sistema compartilham a mesma percepção de tempo local.

Abordagem: Implementação simplificada do Precision Time Protocol (IEEE 1588).

- Eleição de um agente (líder) para prover a base temporal.

Novos Tipos de Mensagens para Sincronização (PTP):

- Necessidade de mensagens específicas para as fases do PTP (Sync, Delay_Req, Delay_Resp) e para o anúncio de presença (Announce).

Adaptações da Message para Sincronização e Controle

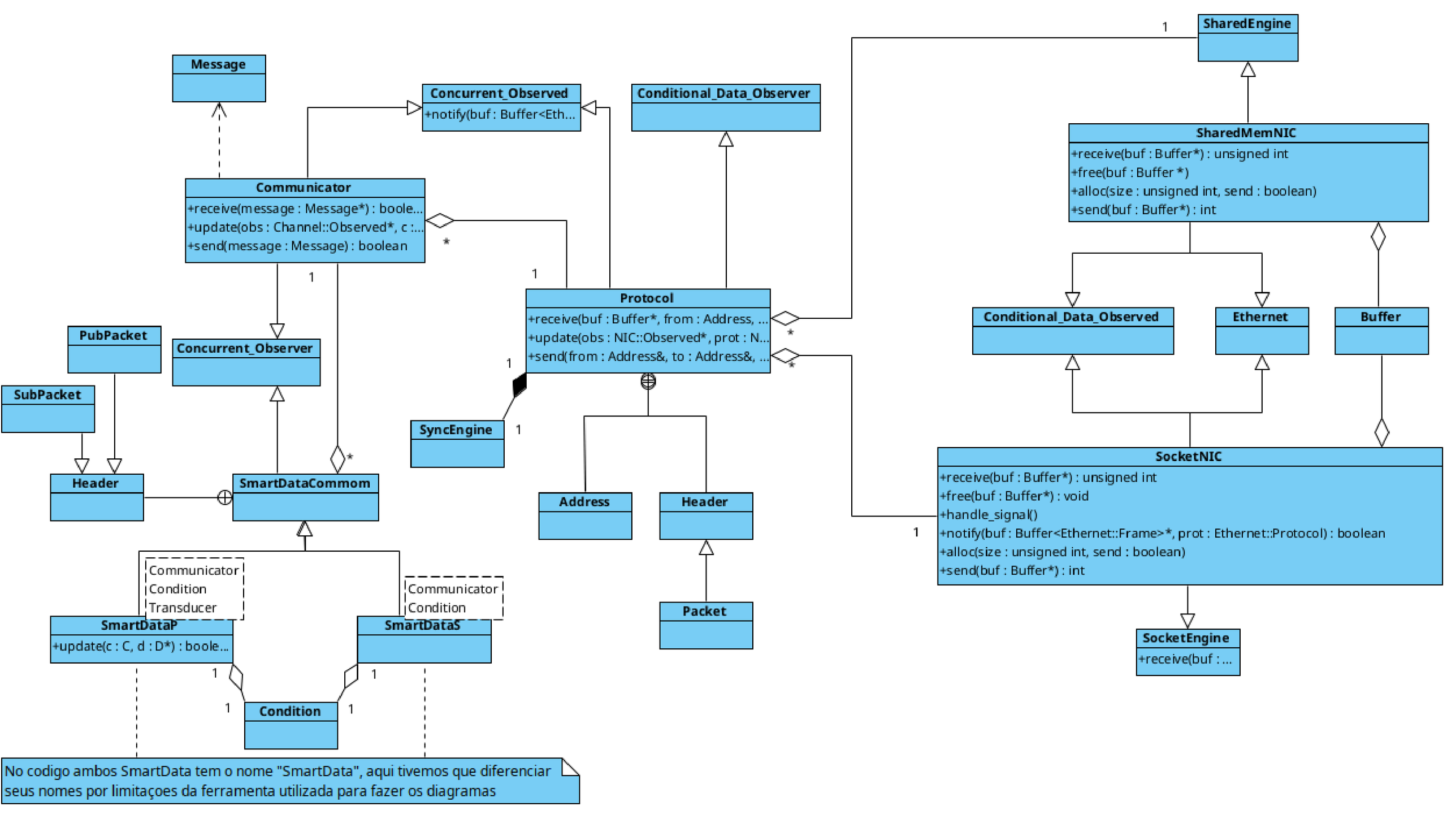
Novo Campo _timestamp em Message:

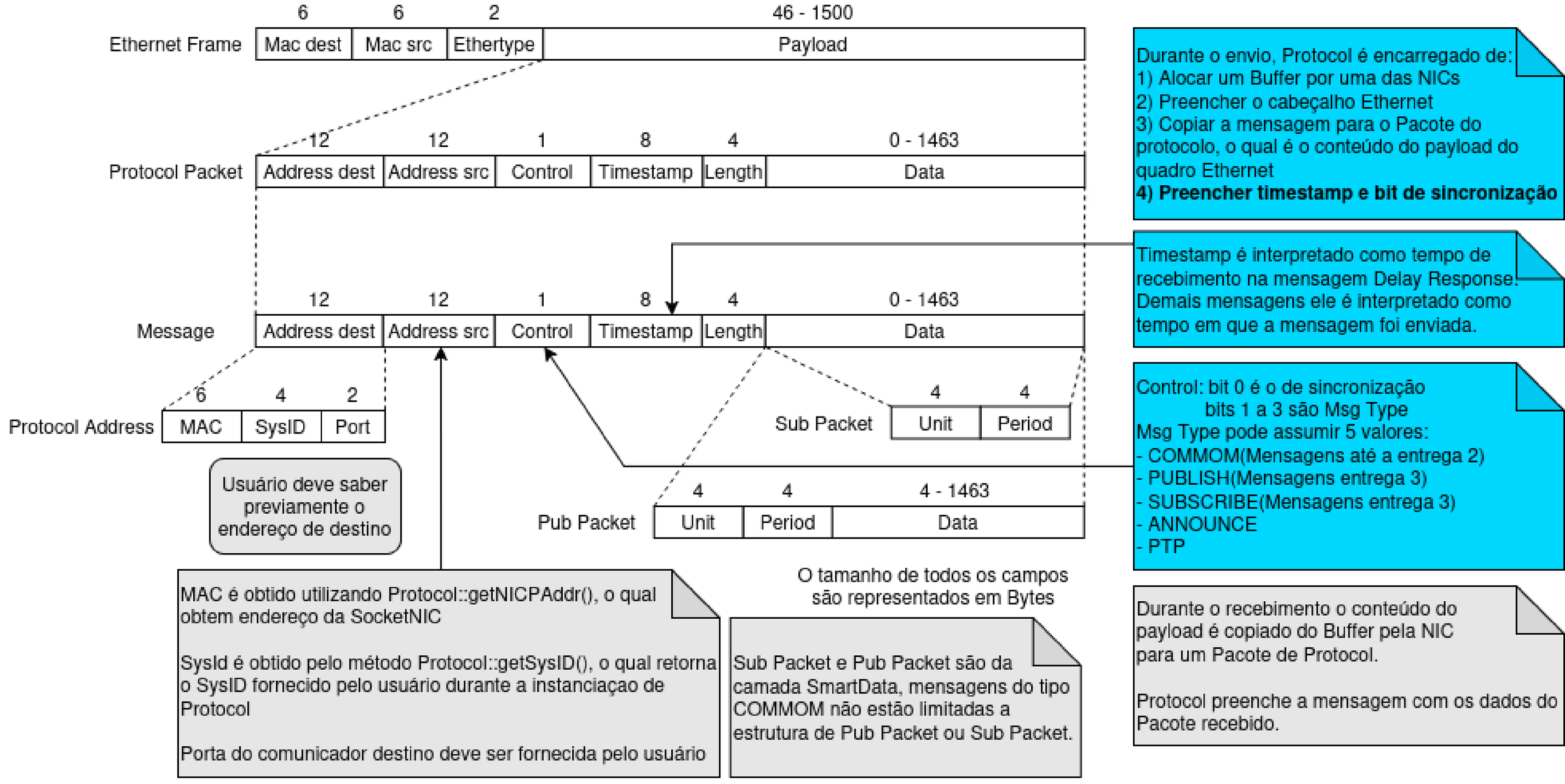
- uint64_t _timestamp;
- Armazena o tempo em que a mensagem foi enviada.
- Preenchido pela camada Protocol no momento do envio, utilizando _sync_engine.getTimestamp().

Estrutura Control (Substitui Message::Type):

- struct Control { uint8_t value; ... };
- O campo value contém:
  - Tipo da Mensagem (Bits TYPE):
    - COMMON, PUBLISH, SUBSCRIBE (funcionalidade E3 mantida).
    - ANNOUNCE: Para mensagens de anúncio no processo de eleição de líder.
    - PTP: Para mensagens do protocolo de sincronização.
  - Status de Sincronização (Bit SYNC):
  - Métodos isSynchronized() e setSynchronized(bool).
  - Indica se o relógio do emissor está sincronizado.
  - Preenchido por Protocol via _sync_engine.getSynced().

Resultado: Message transporta dados de tempo e estado de sincronização, além dos tipos.


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


6
6
2
46 - 1500
Ethernet Frame
Mac dest
Mac src
Ethertype
Payload
12
12
1
8
4
0 - 1463
Protocol Packet
Address dest
Address src
Control
Timestamp
Length
Data
12
12
1
8
4
0 - 1463
Message
Address dest
Address src
Control
Timestamp
Length
Data
6
4
2
Protocol Address
MAC
SysID
Port
4
4
Sub Packet
Unit
Period
4
4
4 - 1463
Pub Packet
Unit
Period
Data
Usuário deve saber previamente o endereço de destino
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 e bit de sincronização
Timestamp é interpretado como tempo de recebimento na mensagem Delay Response. Demais mensagens ele é interpretado como tempo em que a mensagem foi enviada.
Control: bit 0 é o de sincronização
bits 1 a 3 são Msg Type
Msg Type pode assumir 5 valores:
- COMMOM(Mensagens até a entrega 2)
- PUBLISH(Mensagens entrega 3)
- SUBSCRIBE(Mensagens entrega 3)
- ANNOUNCE
- PTP
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
O tamanho de todos os campos são representados em Bytes
Sub Packet e Pub Packet são da camada SmartData, mensagens do tipo COMMOM não estão limitadas a estrutura de Pub Packet ou Sub Packet.
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.

SyncEngine - Gerenciamento da Sincronização

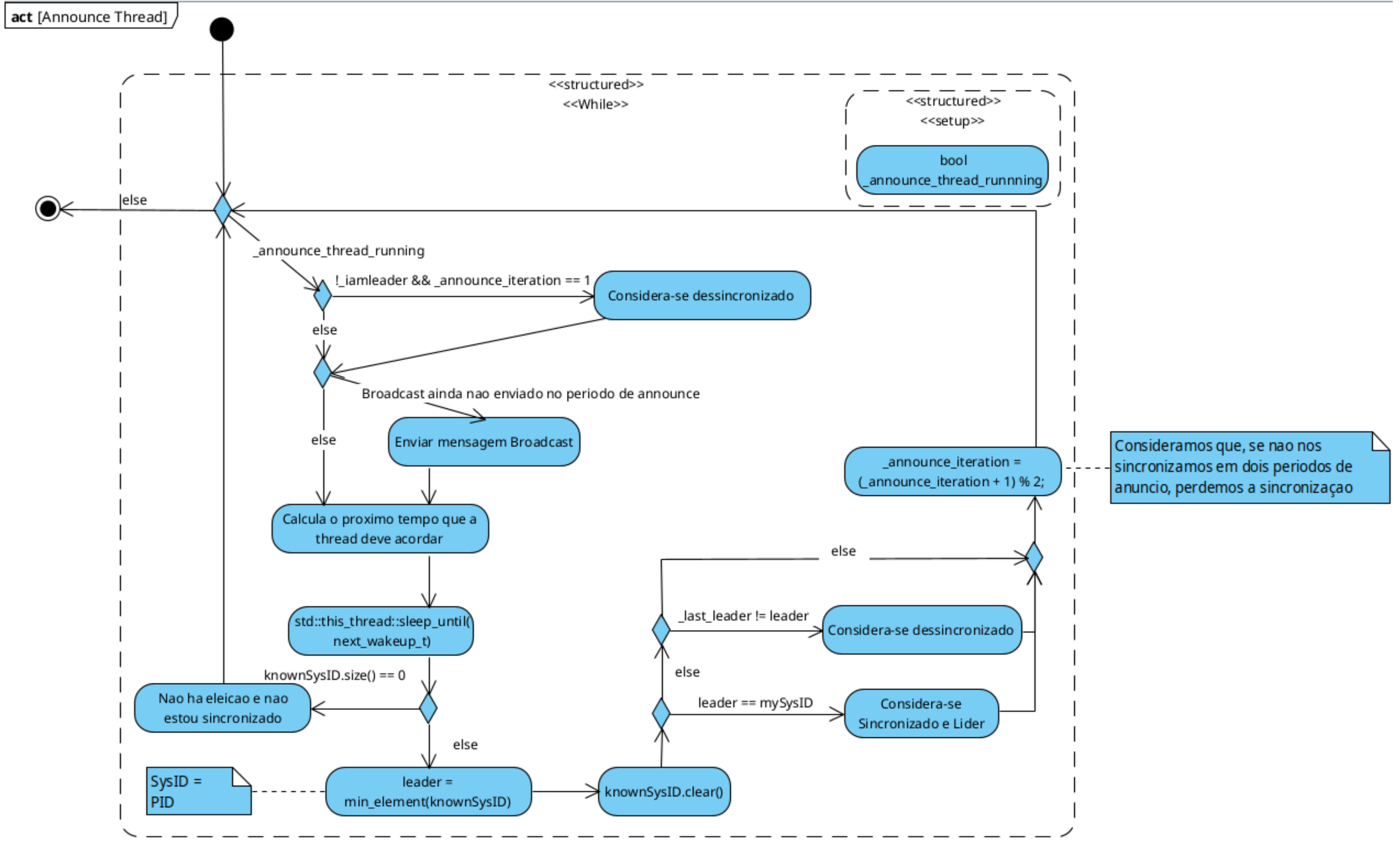
SyncEngine: Entidade Central de Sincronização

- Função: Implementa a lógica do protocolo PTP, eleição de líder e ajuste de relógio.
- Estado e Componentes Internos:
  - SimulatedClock _clock;: Relógio local com offset ajustável.
  - _iamleader;: Flag de liderança PTP.
  - _master_addr: Endereço do líder PTP.
  - _state: Guarda o estado da comunicação PTP (WAITING_SYNC, WAITING_DELAY) para slaves.
  - _synced: Flag de estado de sincronização do relógio.
  - Timestamps para PTP: _sync_t, _recvd_sync_t, _delay_req_t, _leader_recvd_delay_req_t.
  - Threads:
    - _announce_thread: Gerencia envio de ANNOUNCE e eleição.
    - _leader_thread: Se líder, envia Sync broadcast.
- Interação com Protocol:
  - Protocol cria e utiliza uma instância de SyncEngine.
  - No envio (Protocol::send(...)), _timestamp e bit isSynchronized de Control são definidos via SyncEngine.
  - Na recepção (Protocol::update(...)), todas as mensagens da engine de sockets passam por _sync_engine para serem reconhecidas, ou lógica do PTP ser tratada.

Eleição de Líder em SyncEngine

Procedimento de Eleição do Líder PTP

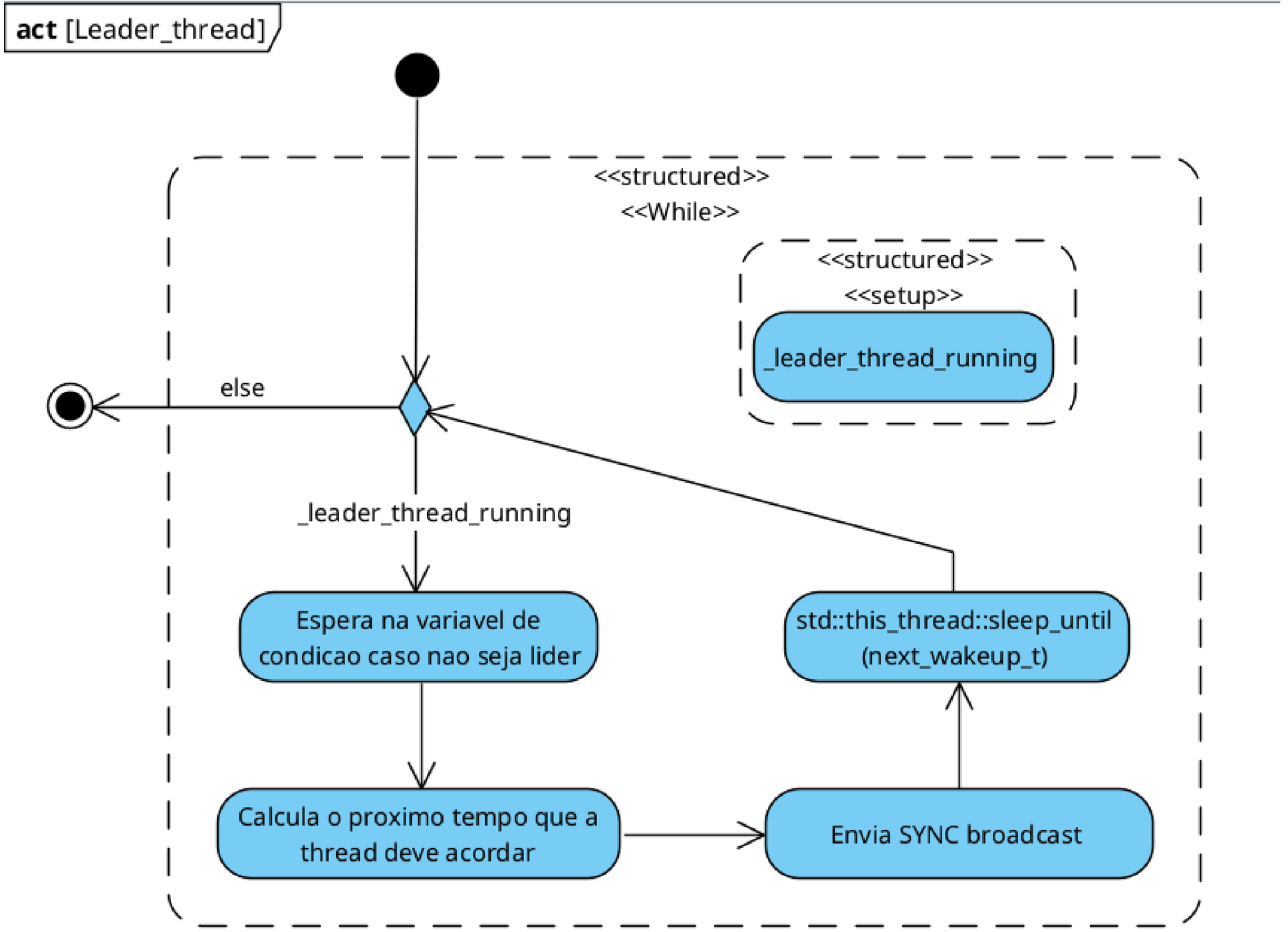
- Operação da _announce_thread:
  - Ciclo periódico (controlado por _announce_period).
  - Fase de Anúncio:
    - Se durante o período uma mensagem broadcast ainda não ter sido enviada, envia Message com Control::Type::ANNOUNCE via _protocol->send(...).
    - SysIDs de mensagens recebidas (via addSysID(), chamado em handlePTP) são adicionados known_sysid.
    - Espera _announce_period.
  - Fase de Eleição (ao final do _announce_period):
    - _leader = elect();
      - elect(): Retorna valor mínimo de _known_sysid.
    - _iamleader = (_leader == _protocol->getSysID());
    - clearKnownSysID() é chamado e limpa os SysIds conhecidos.
  - Ativação e desativação do Líder: _leader_cv.notify_one(); é chamado independentemente de o nó atual ser o líder. A _leader_thread reavaliará sua condição _iamleader ao acordar.
  - Status _synced é atualizado: _synced = true se _iamleader; _synced = false se last_leader != _leader e não é líder.
- Critério de Eleição: Menor SysID (pid_t) entre os SysIDs conhecidos e o próprio.

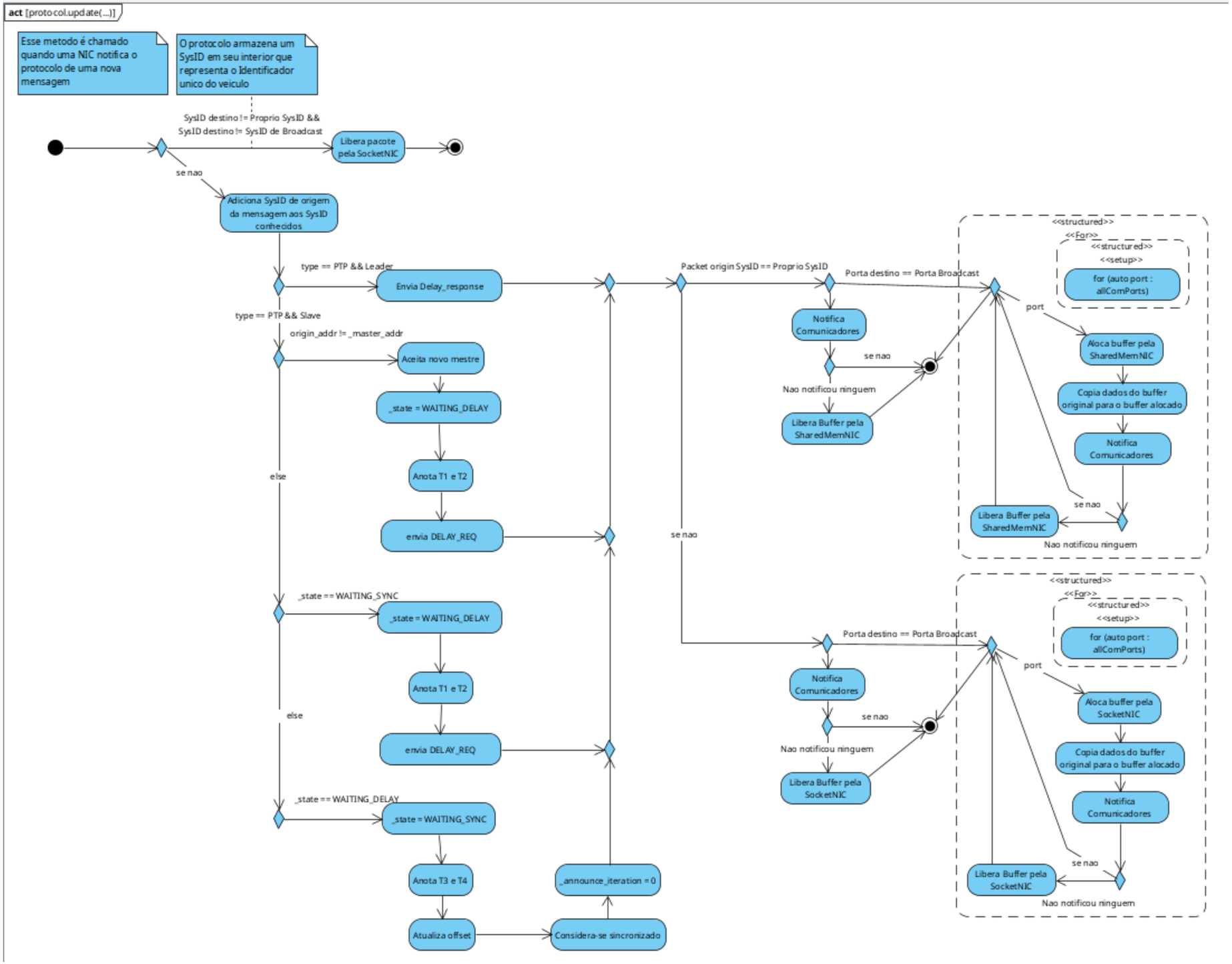

act [Announce Thread]
<<structured>>
<<While>>
<<structured>>
<<setup>>
bool
_announce_thread_runnning
else
_announce_thread_running
!_iamleader && _announce_iteration == 1
Considera-se dessincronizado
else
Broadcast ainda nao enviado no periodo de announce
else
Enviar mensagem Broadcast
Calcula o proximo tempo que a
thread deve acordar
std::this_thread::sleep_until(
next_wakeup_t)
knownSysID.size() == 0
Nao ha eleicao e nao
estou sincronizado
else
SysID =
PID
leader =
min_element(knownSysID)
knownSysID.clear()
else
_last_leader != leader
Considera-se dessincronizado
else
leader == mySysID
Considera-se
Sincronizado e Lider
_announce_iteration =
(_announce_iteration + 1) % 2;
Consideramos que, se nao nos
sincronizamos em dois periodos de
anuncio, perdemos a sincronizaçao

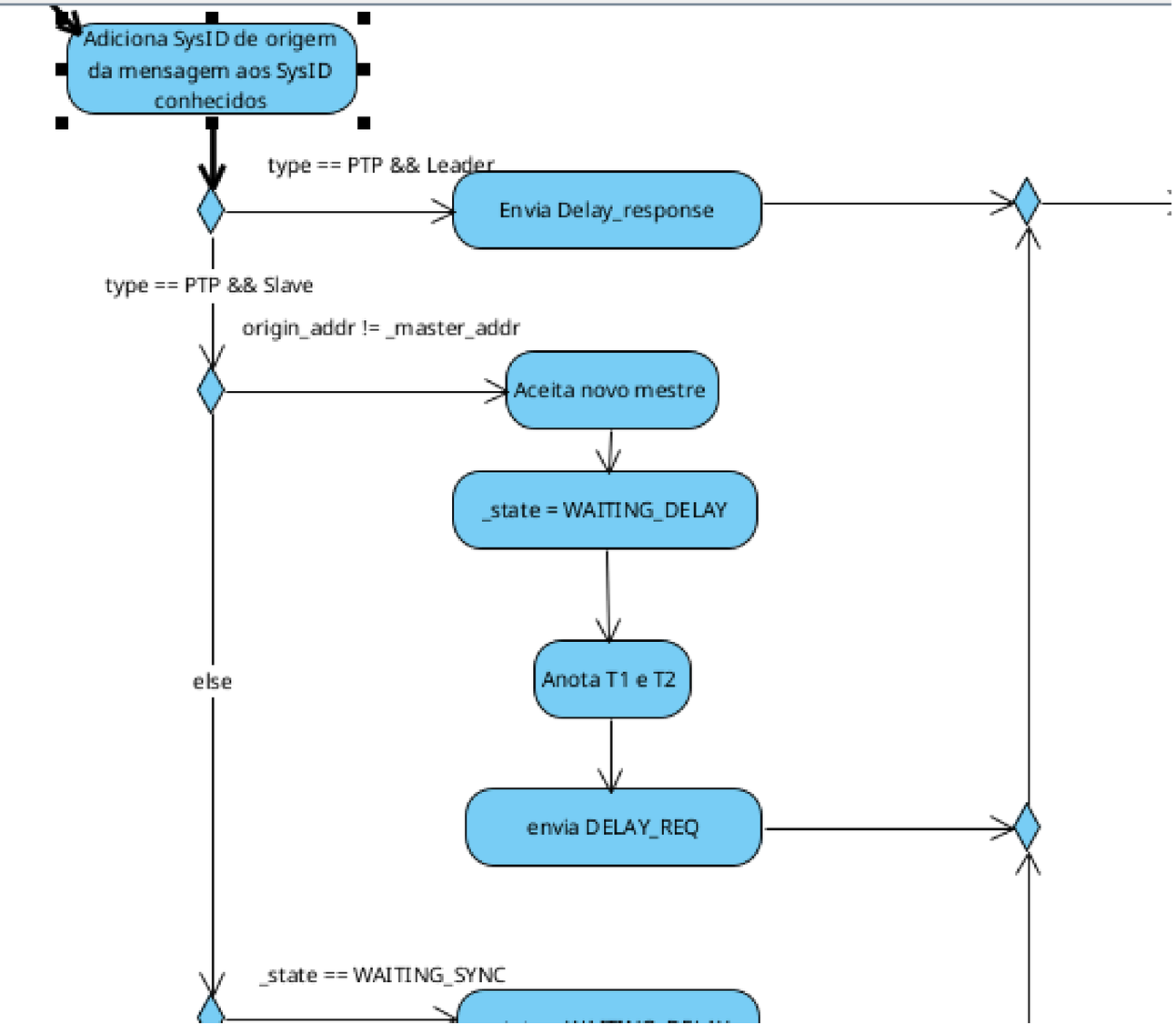
Processo de Sincronização de Relógio (PTP)

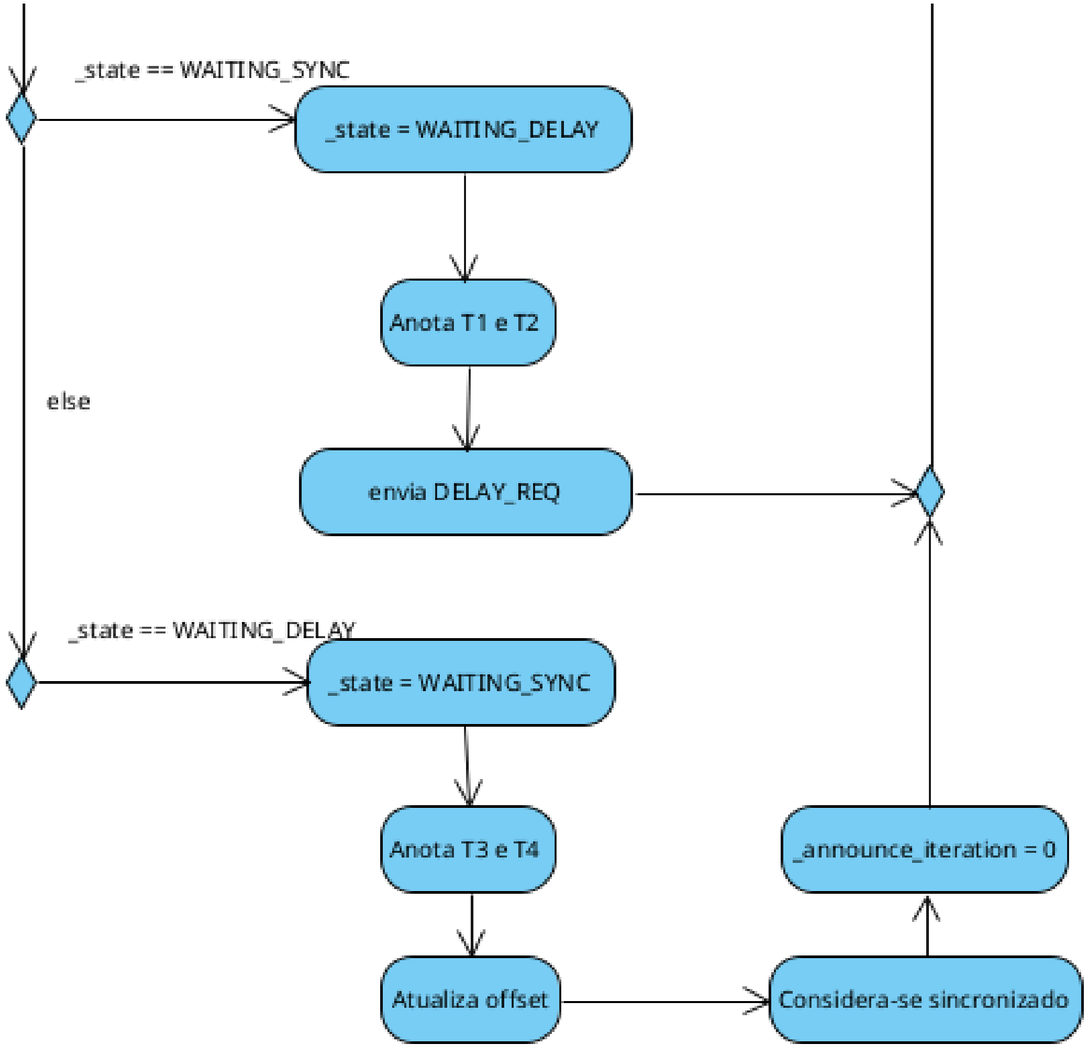
Mecanismo de Ajuste de _clock (Lógica em SyncEngine::handlePTP)

- Operação do Líder (Master - _iamleader == true):
  - _leader_thread envia Message de Sync periodicamente. O pkt->header()->timestamp (T1) é o tempo de envio do Master.
  - Ao receber PTP (Delay_Req) de um Slave (via handlePTP), responde com PTP (Delay_Resp). O pkt->header()->timestamp desta resposta (T4) é o tempo em que o Master recebeu o Delay_Req.
- **Operação do Escravo (_iamleader == false):**
  - A função handlePTP processa mensagens PTP recebidas, utilizando msg_timestamp (timestamp da mensagem PTP recebida) e recv_timestamp (tempo local de chegada da mensagem PTP).
  - Ao receber Sync do Líder:
    - Armazena timestamp (T1) e timestamp (T2).
    - Muda _state para WAITING_DELAY.
    - Solicita ao Protocol o envio de Delay_Req. O timestamp desta mensagem (T3) é salvo em _delay_req_t pelo _sync_engine.setDelayReqSendT(...).
  - Ao receber Delay_Resp do Líder (no estado WAITING_DELAY):
    - Armazena _leader_recvd_delay_req_t = msg_timestamp (T4).
    - Cálculos: **delay** = ((T4 - T3) + (T2 - T1)) / 2; **offset** = (T2 - T1) - delay;
    - Ajuste: _clock.setOffset(offset);
    - Muda _state para WAITING_SYNC; define _synced = true; define _announce_iteration = 0;.
- **Interação:** SyncEngine executa a lógica PTP. Protocol encaminha mensagens PTP e utiliza SyncEngine para obter/definir informações temporais.


act [Leader_thread]
<<structured>>
<<While>>
<<structured>>
<<setup>>
_leader_thread_running
else
_leader_thread_running
Espera na variavel de condicao caso nao seja lider
std::this_thread::sleep_until (next_wakeup_t)
Calcula o proximo tempo que a thread deve acordar
Envia SYNC broadcast


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
Adiciona SysID de origem da mensagem aos SysID conhecidos
type == PTP && Leader
Envia Delay_response
type == PTP && Slave
origin_addr != _master_addr
Aceita novo mestre
_state = WAITING_DELAY
Anota T1 e T2
envia DELAY_REQ
else
_state == WAITING_SYNC
_state = WAITING_DELAY
Anota T1 e T2
envia DELAY_REQ
else
_state == WAITING_DELAY
_state = WAITING_SYNC
Anota T3 e T4
Atualiza offset
Considera-se sincronizado
_announce_iteration = 0
Packet origin SysID == Proprio SysID
Porta destino == Porta Broadcast
Notifica Comunicadores
se nao
Nao notificou ninguem
Libera Buffer pela SharedMemNIC
<<structured>>
<<For>>
<<structured>>
<<setup>>
for (auto port : allComPorts)
port
Aloca buffer pela SharedMemNIC
Copia dados do buffer original para o buffer alocado
Notifica Comunicadores
se nao
Libera Buffer pela SharedMemNIC
Nao notificou ninguem
se nao
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

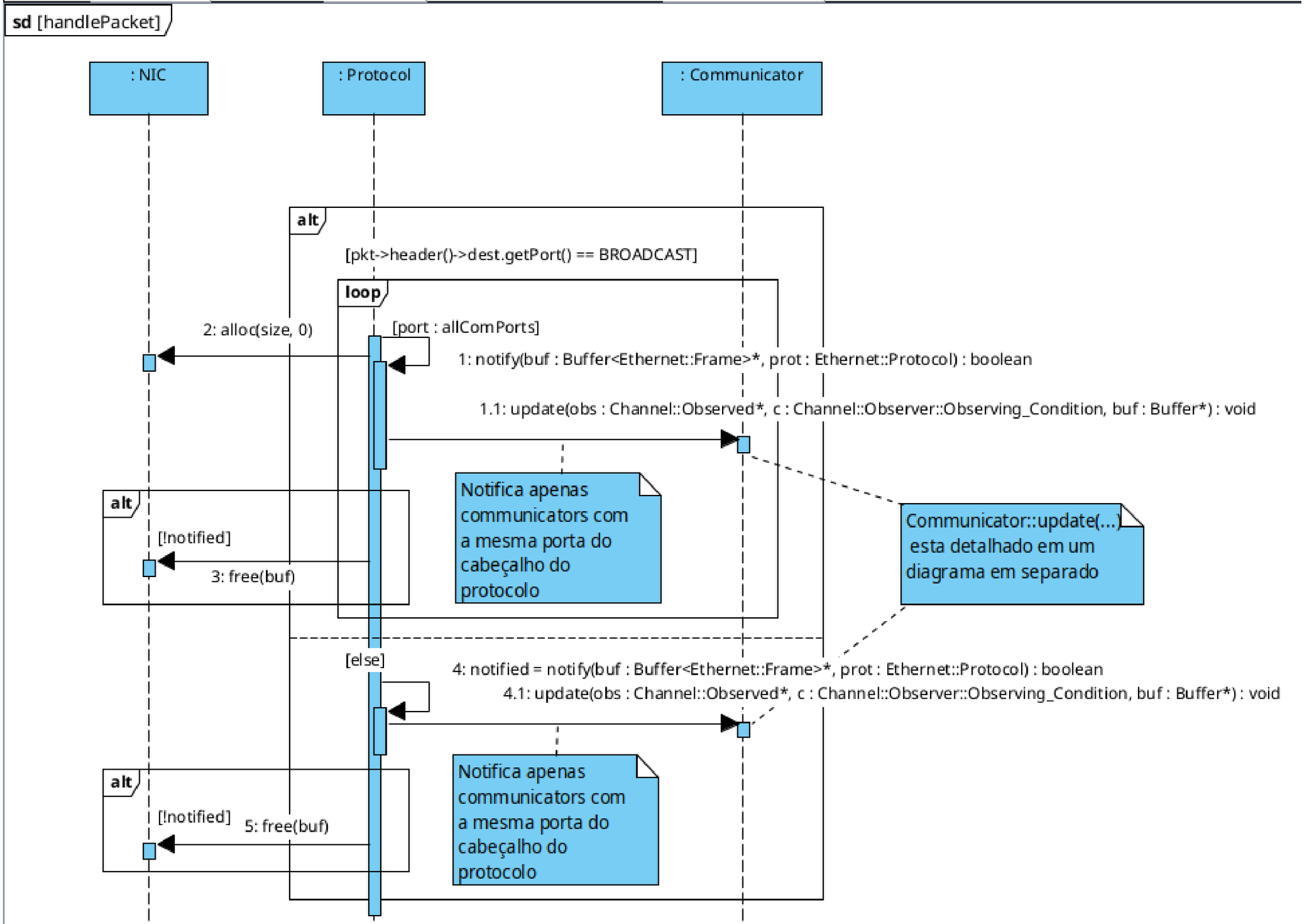

Adiciona SysID de origem da mensagem aos SysID conhecidos
type == PTP && Leader
Envia Delay_response
type == PTP && Slave
origin_addr != _master_addr
Aceita novo mestre
_state = WAITING_DELAY
Anota T1 e T2
envia DELAY_REQ
else
_state == WAITING_SYNC

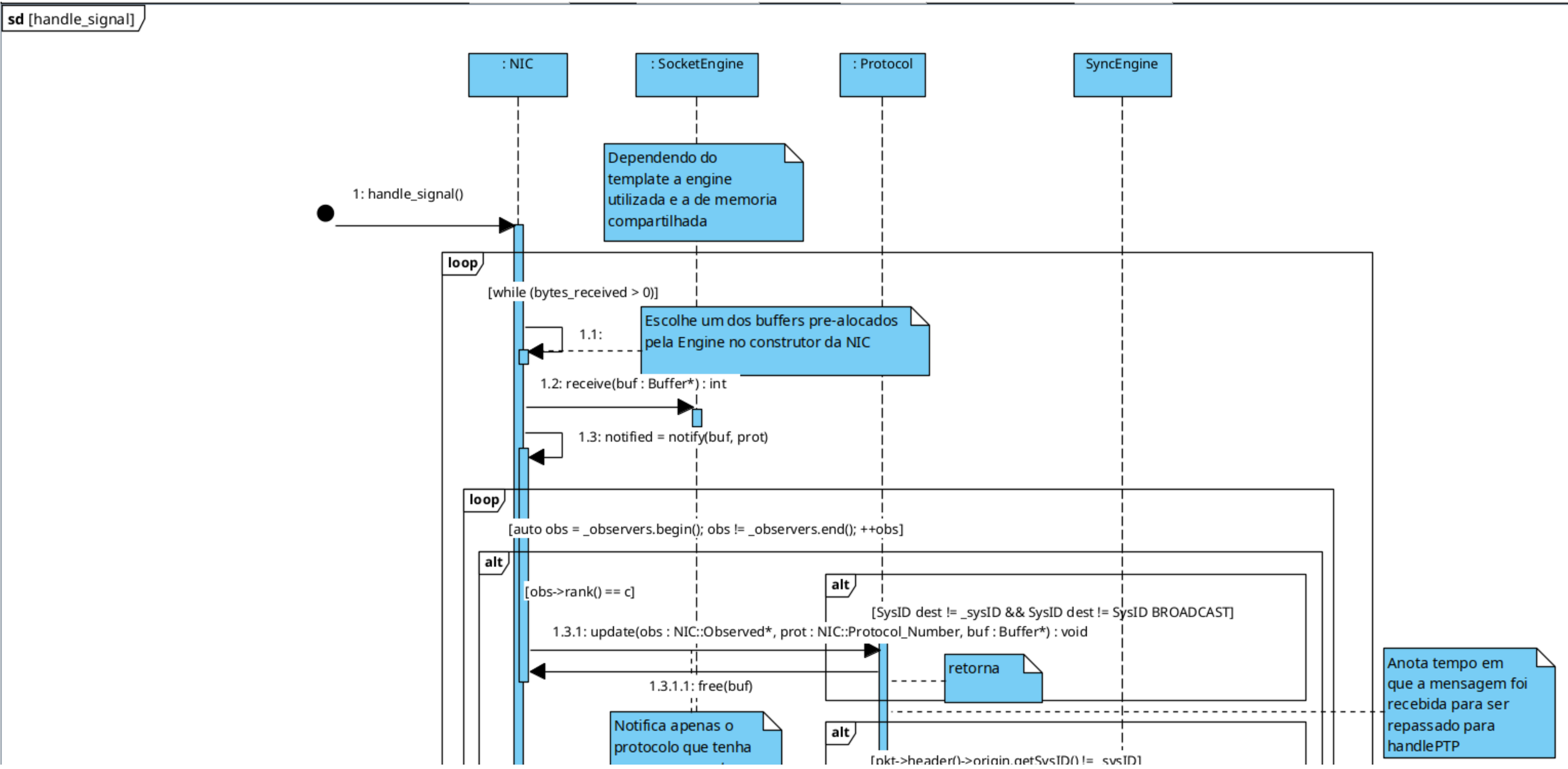

_state == WAITING_SYNC
_state = WAITING_DELAY
Anota T1 e T2
envia DELAY_REQ
else
_state == WAITING_DELAY
_state = WAITING_SYNC
Anota T3 e T4
Atualiza offset
Considera-se sincronizado
_announce_iteration = 0

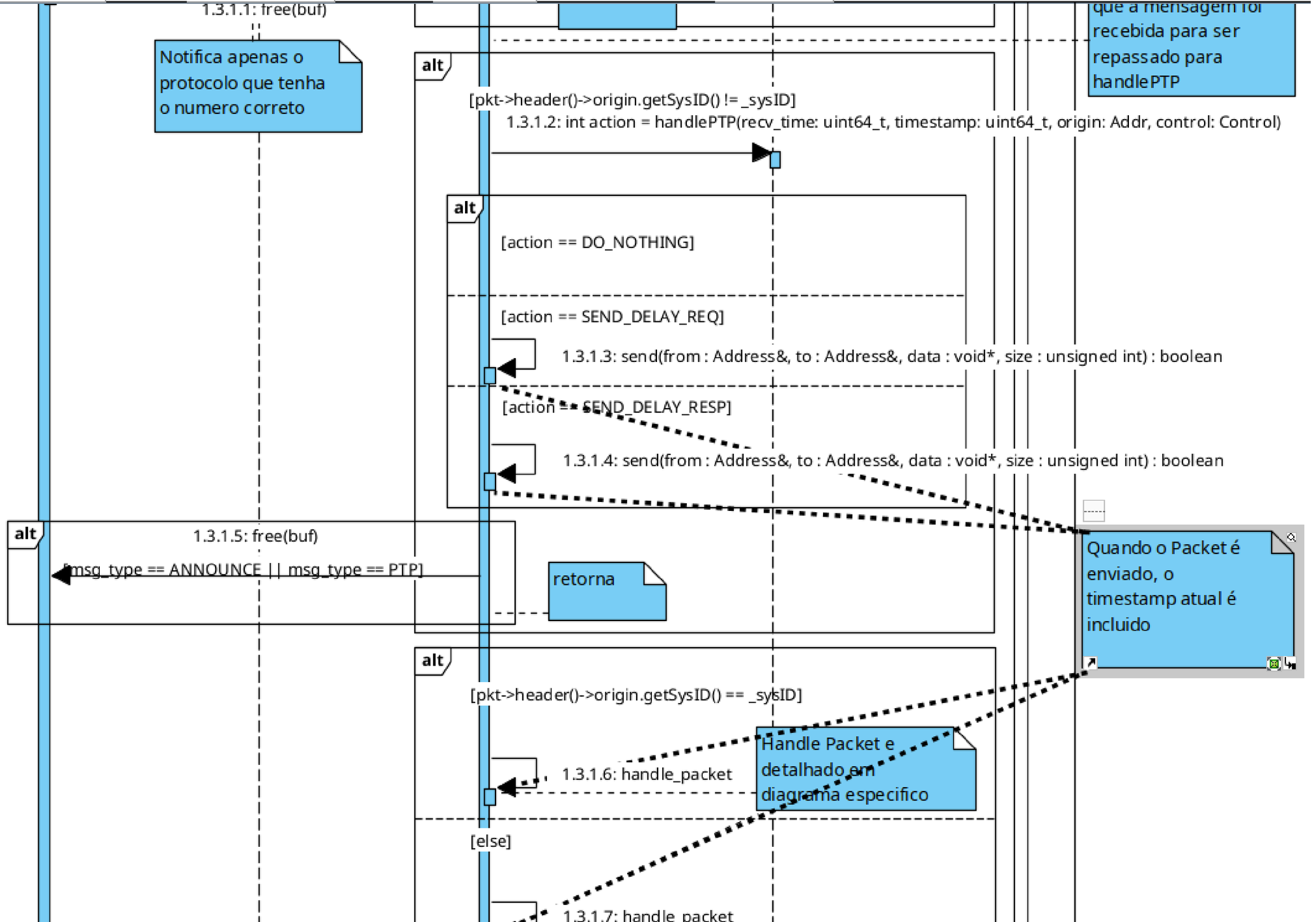
Testes Desenvolvidos e Validação

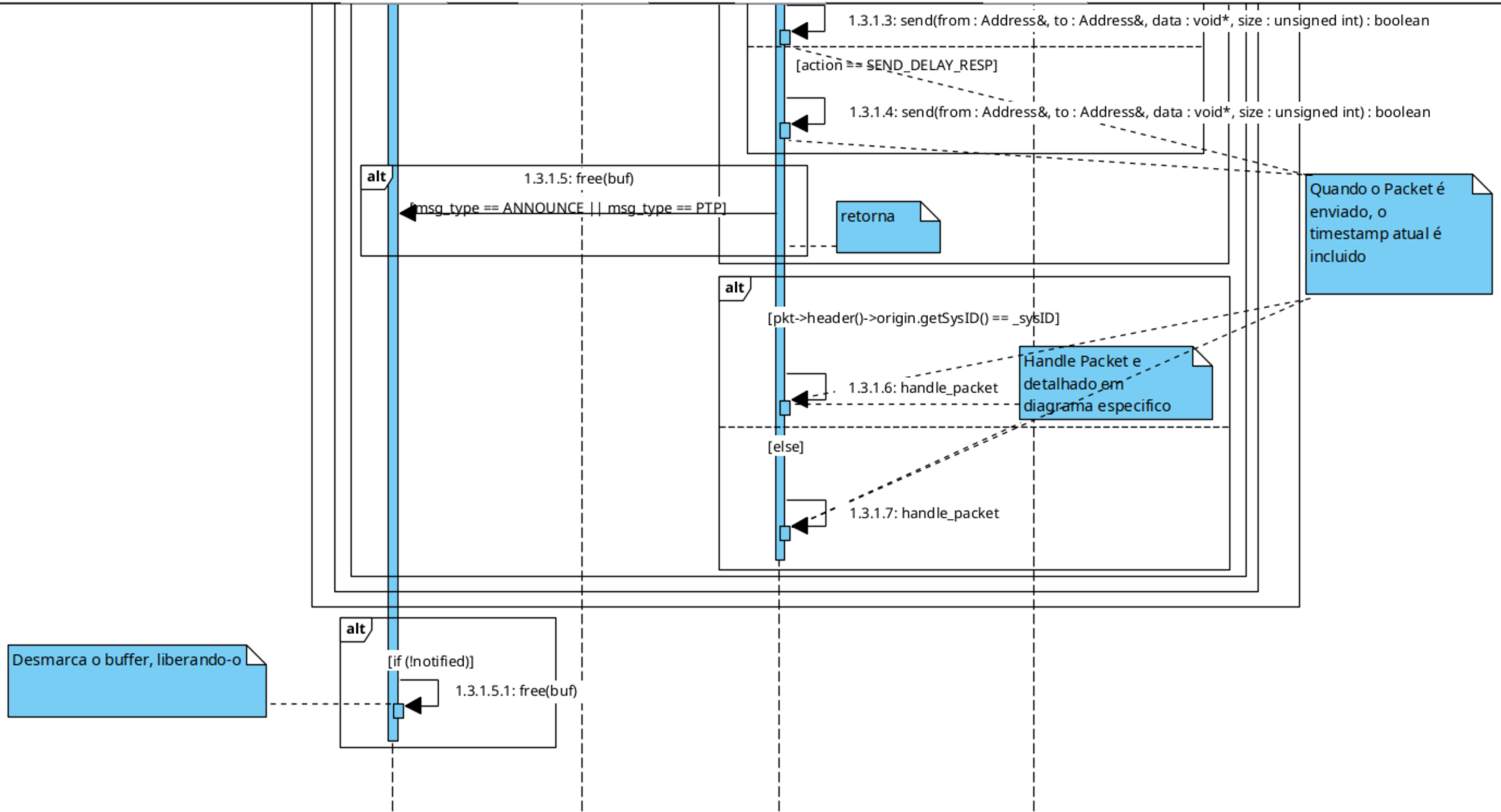
- Teste de Desaparecimento do Líder PTP:
  - Configuração: Uma rede com múltiplos processos, um _leader PTP estabelecido e slaves _synced. O processo líder é encerrado abruptamente.
- Teste entre Componentes do Mesmo Processo (Intra-Veículo):
  - Configuração: Dois ou mais Communicators (ou SmartDatas) instanciados em threads diferentes dentro do mesmo processo.
- Teste entre dois veículos com objetivo de verificar offset calculado.
- Teste com Entrada Dinâmica de Processos na Simulação:
  - Configuração: Processos (veículos) são iniciados em momentos distintos.
- Teste com Múltiplos Publicadores (SmartData):
  - Configuração: Múltiplos SmartDatas Publicadores, potencialmente para a mesma SmartUnit ou diferentes, coexistindo com um ou mais Subscritores.
- Teste de Não Eleição (Processo Único):
  - Configuração: Apenas um processo (veículo) é iniciado.
- Teste Unitário da classe SyncEngine com objetivo de verificar handlePTP isoladamente.
- Teste de Unsubscribe

Diagramas Novos ou Atualizados:


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)

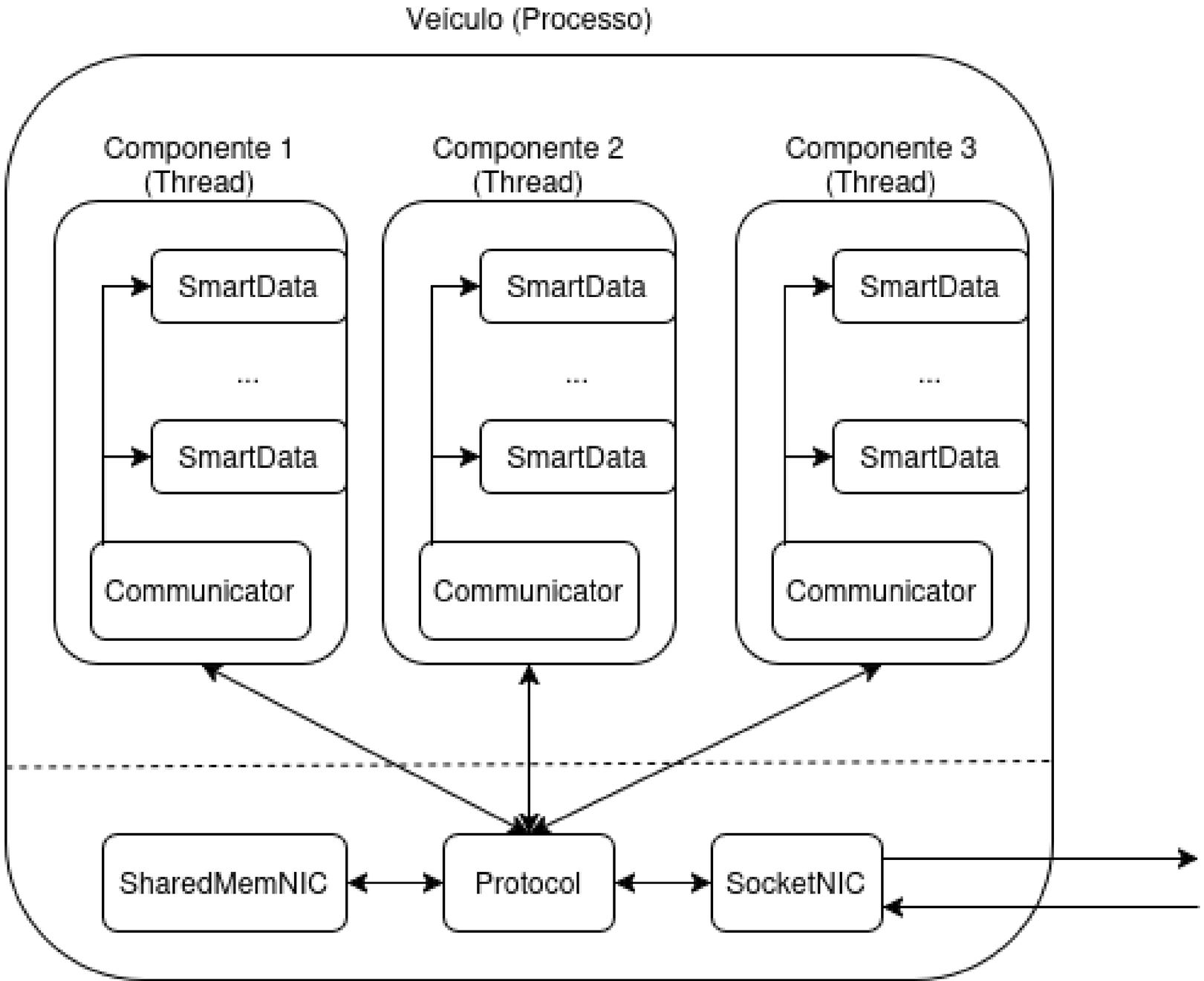

sd [handle_signal]
: NIC
: SocketEngine
: Protocol
SyncEngine
Dependendo do template a engine utilizada e a de memoria compartilhada
1: handle_signal()
loop
[while (bytes_received > 0)]
1.1:
Escolhe um dos buffers pre-alocados pela Engine no construtor da NIC
1.2: receive(buf : Buffer*) : int
1.3: notified = 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)
Anota tempo em que a mensagem foi recebida para ser repassado para handlePTP
Notifica apenas o protocolo que tenha
alt
[pkt->header()->origin.getSysID() != _sysID]


1.3.1.1: free(buf)
Notifica apenas o protocolo que tenha o numero correto
alt
[pkt->header()->origin.getSysID() != _sysID]
1.3.1.2: int action = handlePTP(recv_time: uint64_t, timestamp: uint64_t, origin: Addr, control: Control)
alt
[action == DO_NOTHING]
[action == SEND_DELAY_REQ]
1.3.1.3: send(from : Address&, to : Address&, data : void*, size : unsigned int) : boolean
[action == SEND_DELAY_RESP]
1.3.1.4: send(from : Address&, to : Address&, data : void*, size : unsigned int) : boolean
alt
1.3.1.5: free(buf)
[msg_type == ANNOUNCE || msg_type == PTP]
retorna
alt
[pkt->header()->origin.getSysID() == _sysID]
1.3.1.6: handle_packet
Handle Packet e detalhado em diagrama especifico
[else]
1.3.1.7: handle_packet
que a mensagem foi recebida para ser repassado para handlePTP
Quando o Packet é enviado, o timestamp atual é incluido

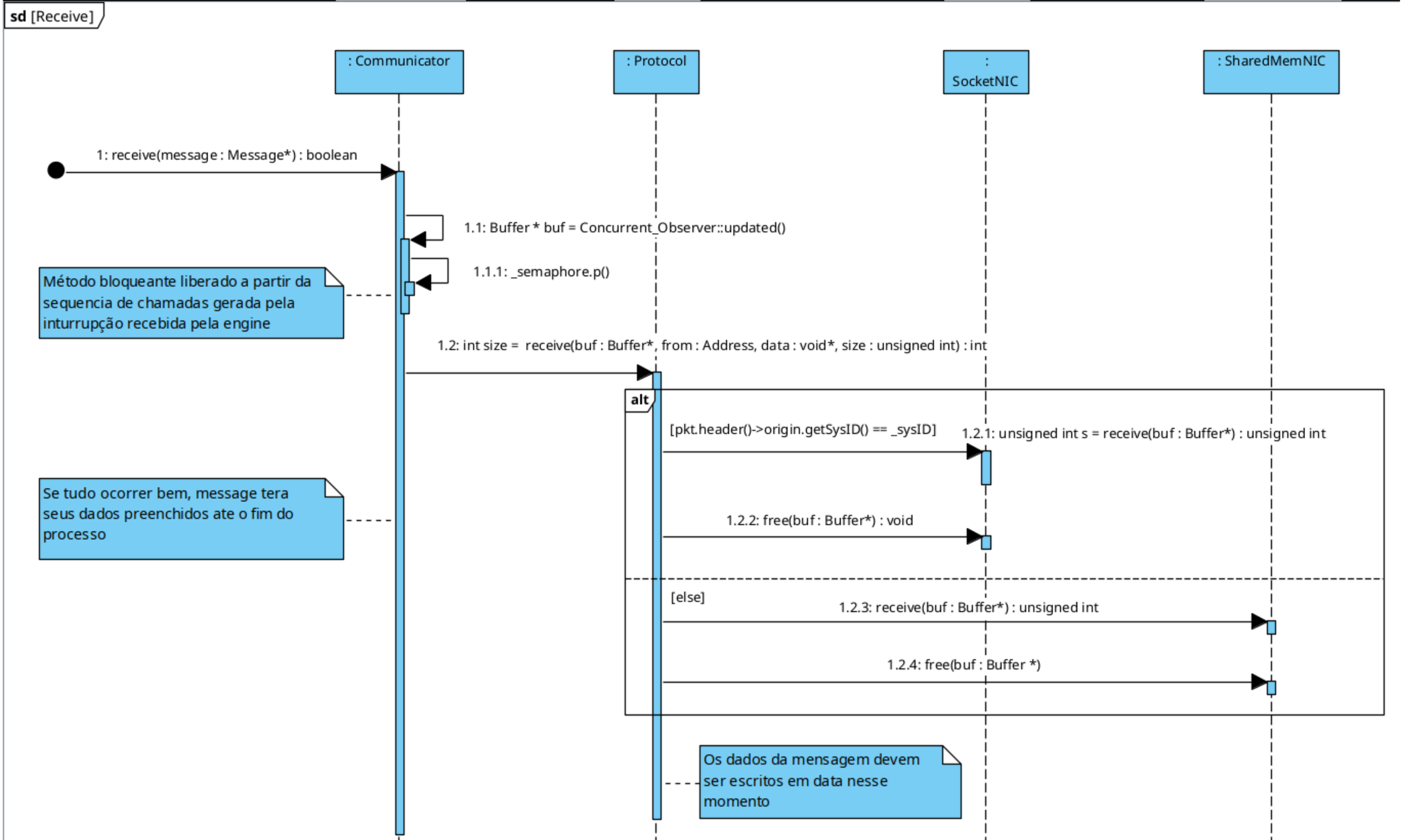

1.3.1.3: send(from : Address&, to : Address&, data : void*, size : unsigned int) : boolean
[action == SEND_DELAY_RESP]
1.3.1.4: send(from : Address&, to : Address&, data : void*, size : unsigned int) : boolean
alt
1.3.1.5: free(buf)
[msg_type == ANNOUNCE || msg_type == PTP]
retorna
Quando o Packet é enviado, o timestamp atual é incluido
alt
[pkt->header()->origin.getSysID() == _sysID]
Handle Packet e detalhado em diagrama especifico
1.3.1.6: handle_packet
[else]
1.3.1.7: handle_packet
alt
[if (!notified)]
1.3.1.5.1: free(buf)
Desmarca o buffer, liberando-o

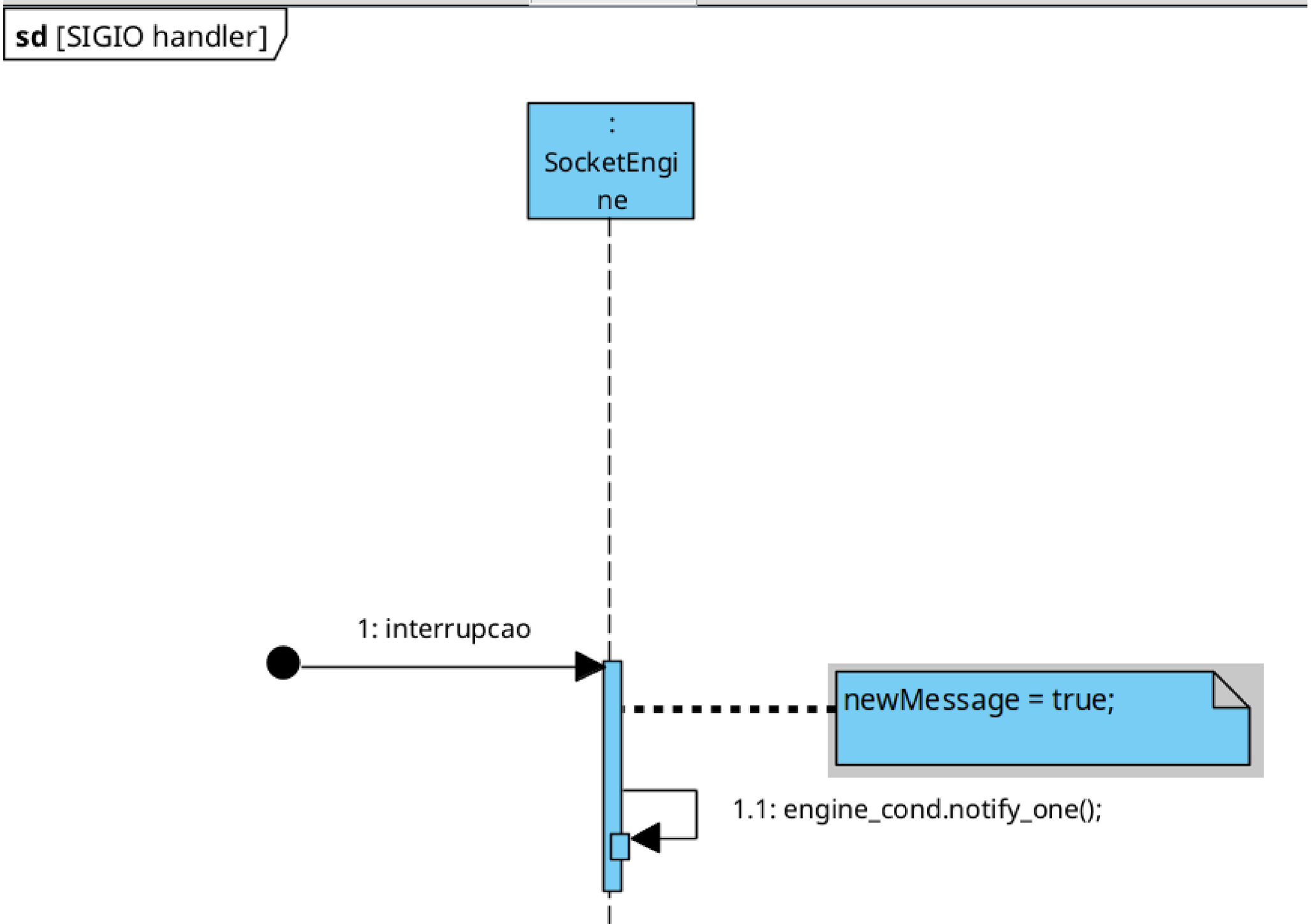
Outros Diagramas:

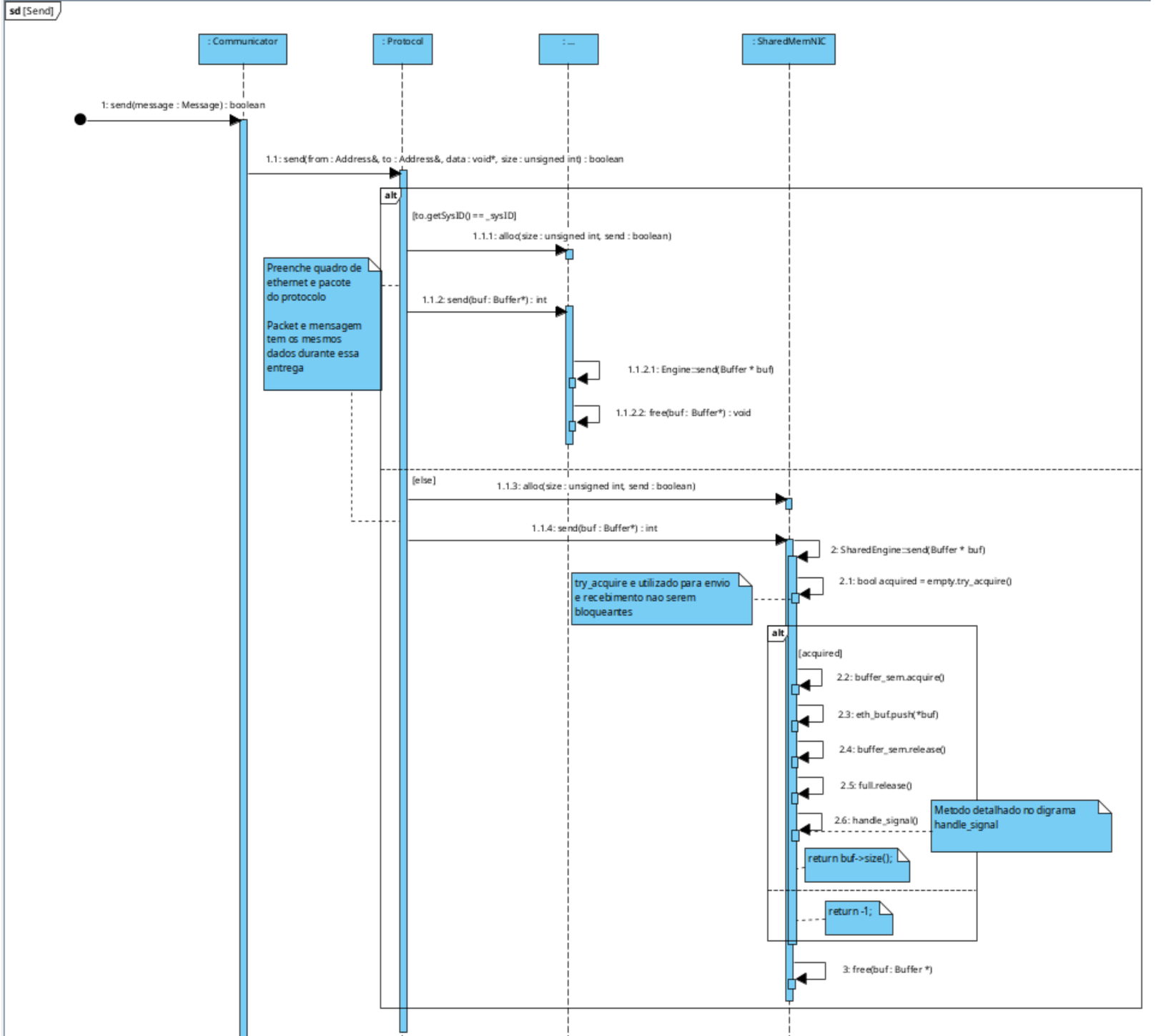
Estrutura do Veículo

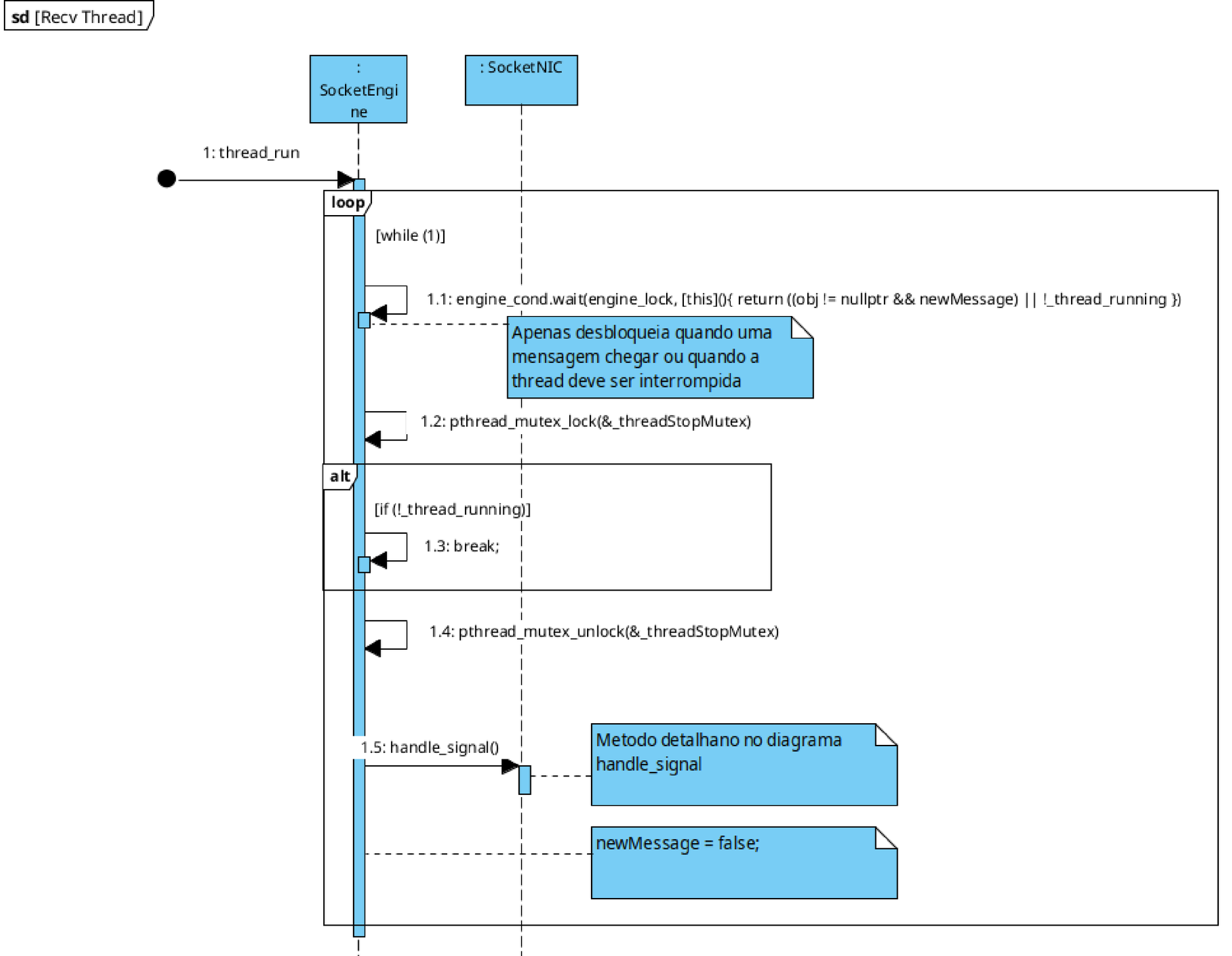


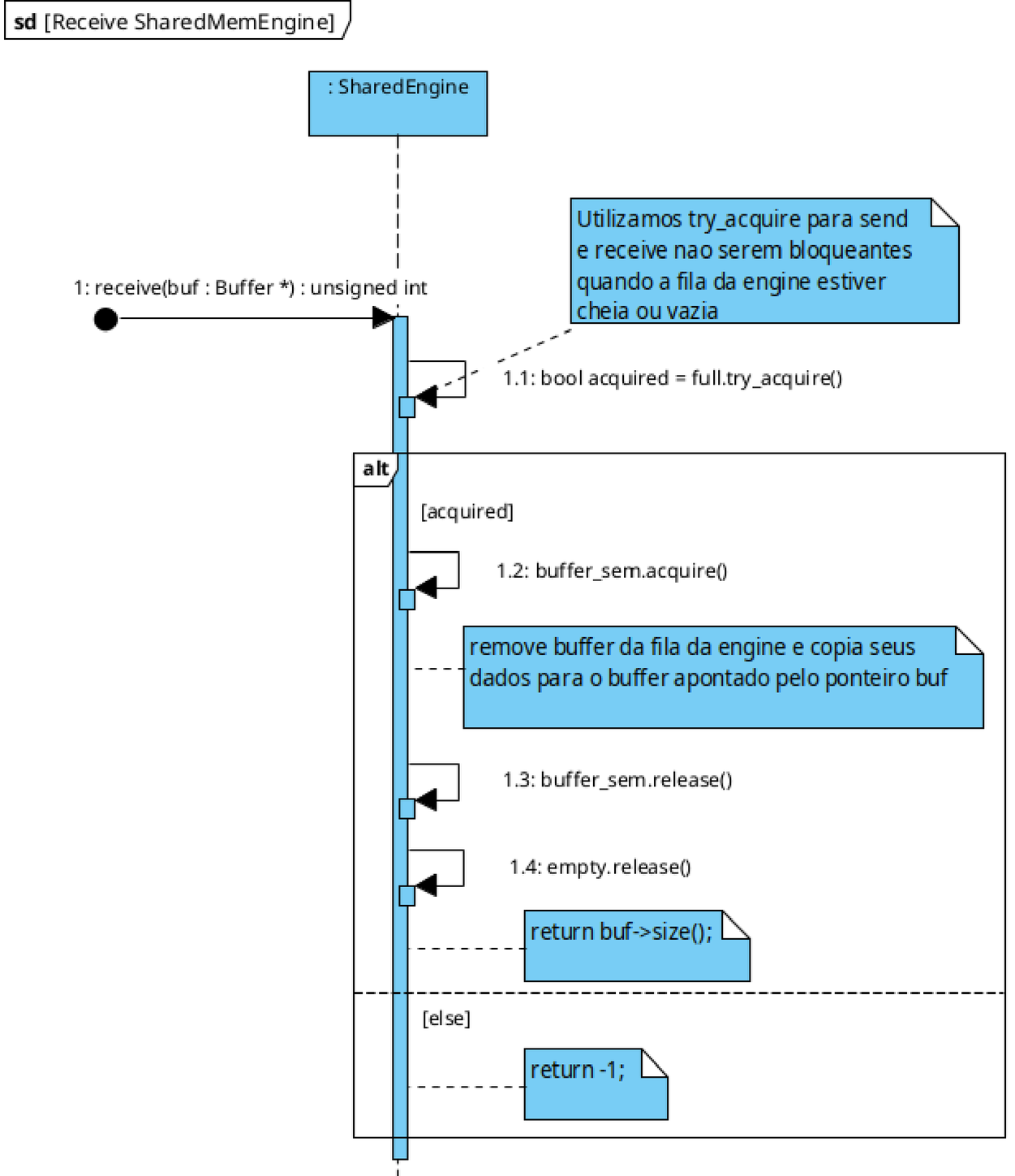
Diagramas de Sequência

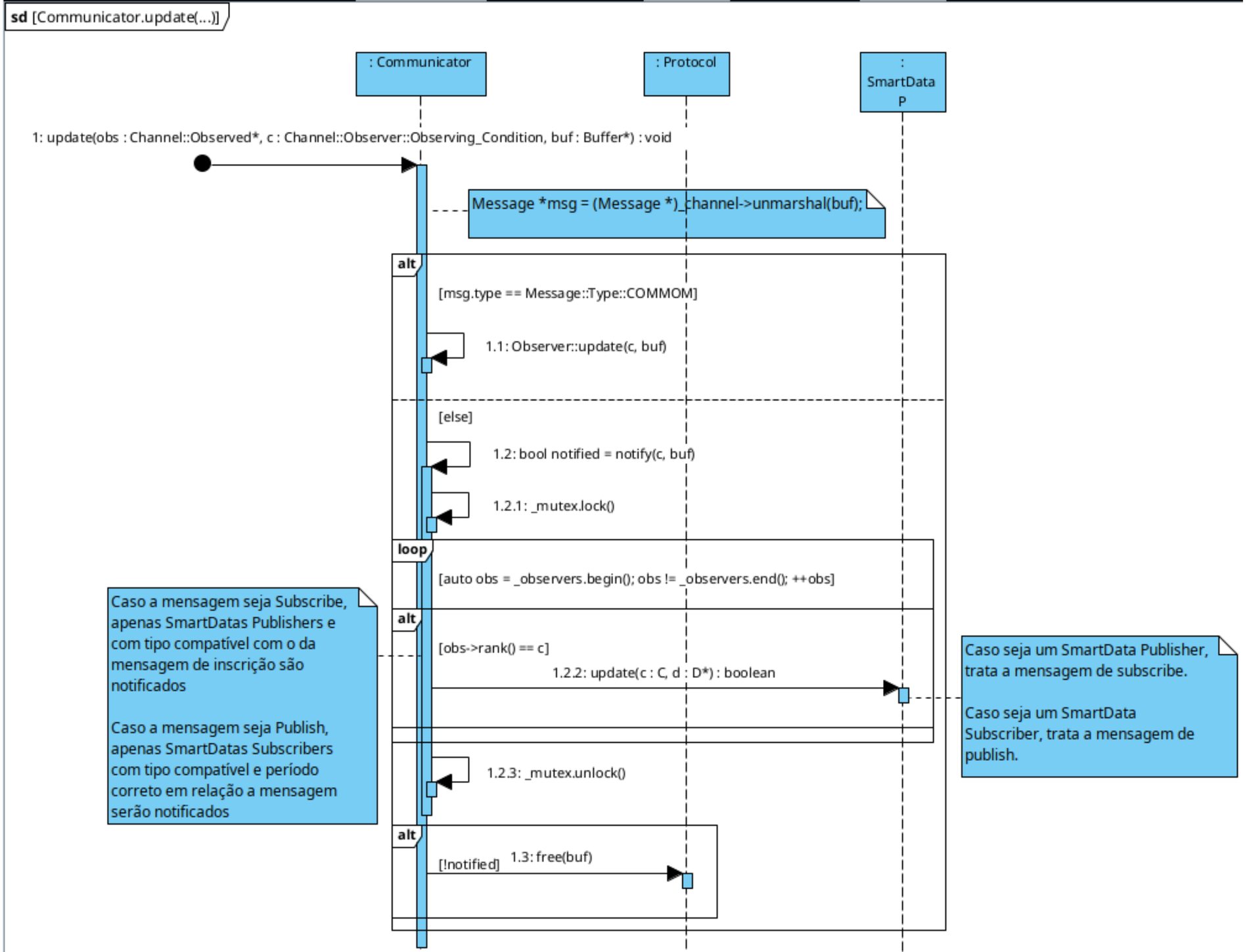

sd [Receive]
: Communicator
: Protocol
: SocketNIC
: SharedMemNIC
1: receive(message : Message*) : boolean
1.1: Buffer * buf = Concurrent_Observer::updated()
1.1.1: _semaphore.p()
Método bloqueante liberado a partir da sequencia de chamadas gerada pela inturrupção recebida pela engine
1.2: int size =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

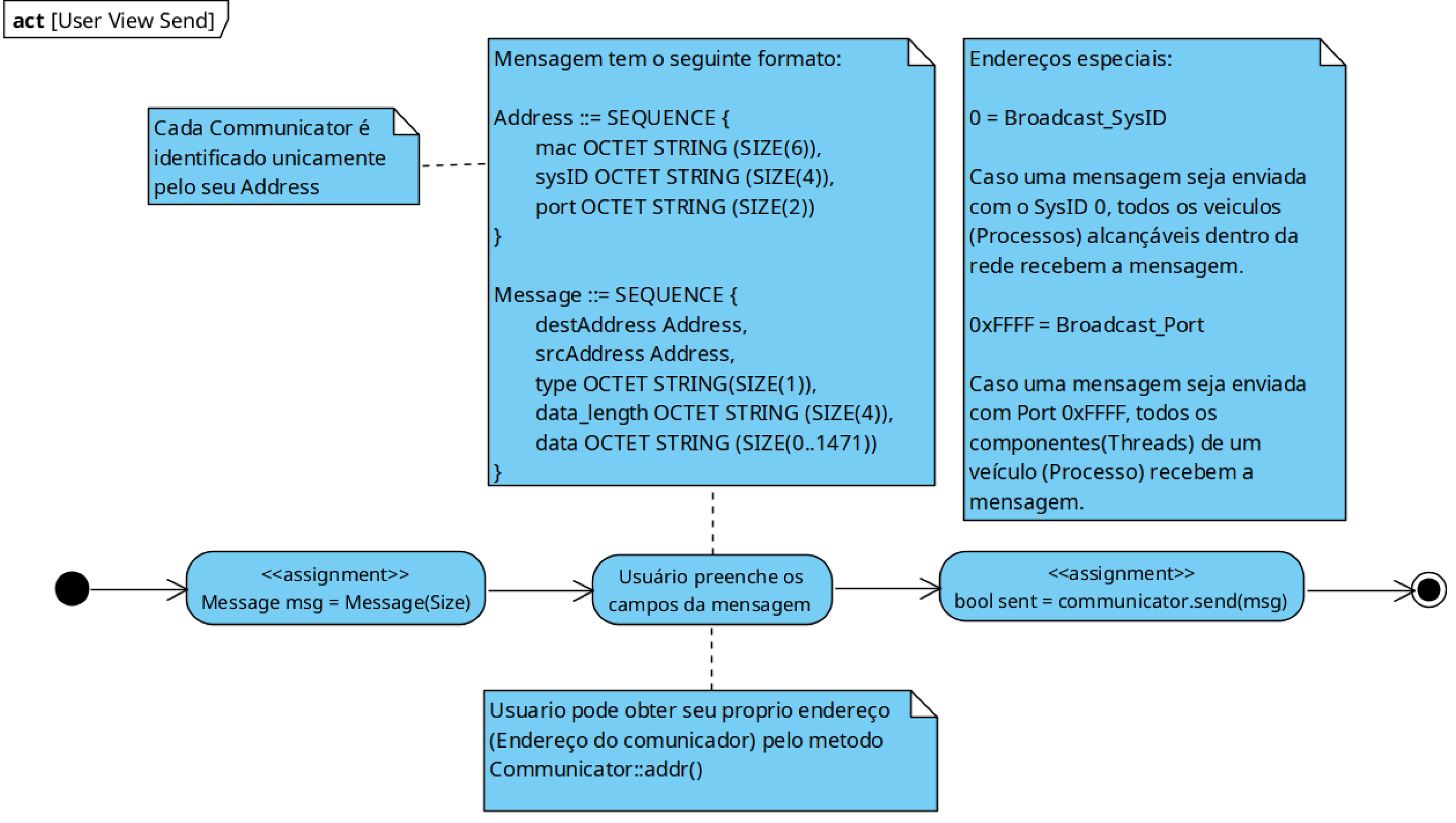

sd [Send]
: Communicator
: Protocol
: ...
: SharedMemNIC
1: send(message : Message) : boolean
1.1: send(from : Address&, to : Address&, data : void*, size : unsigned int) : boolean
alt
[to.getSysID() == _sysID]
1.1.1: alloc(size : unsigned int, send : boolean)
Preenche quadro de ethernet e pacote do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

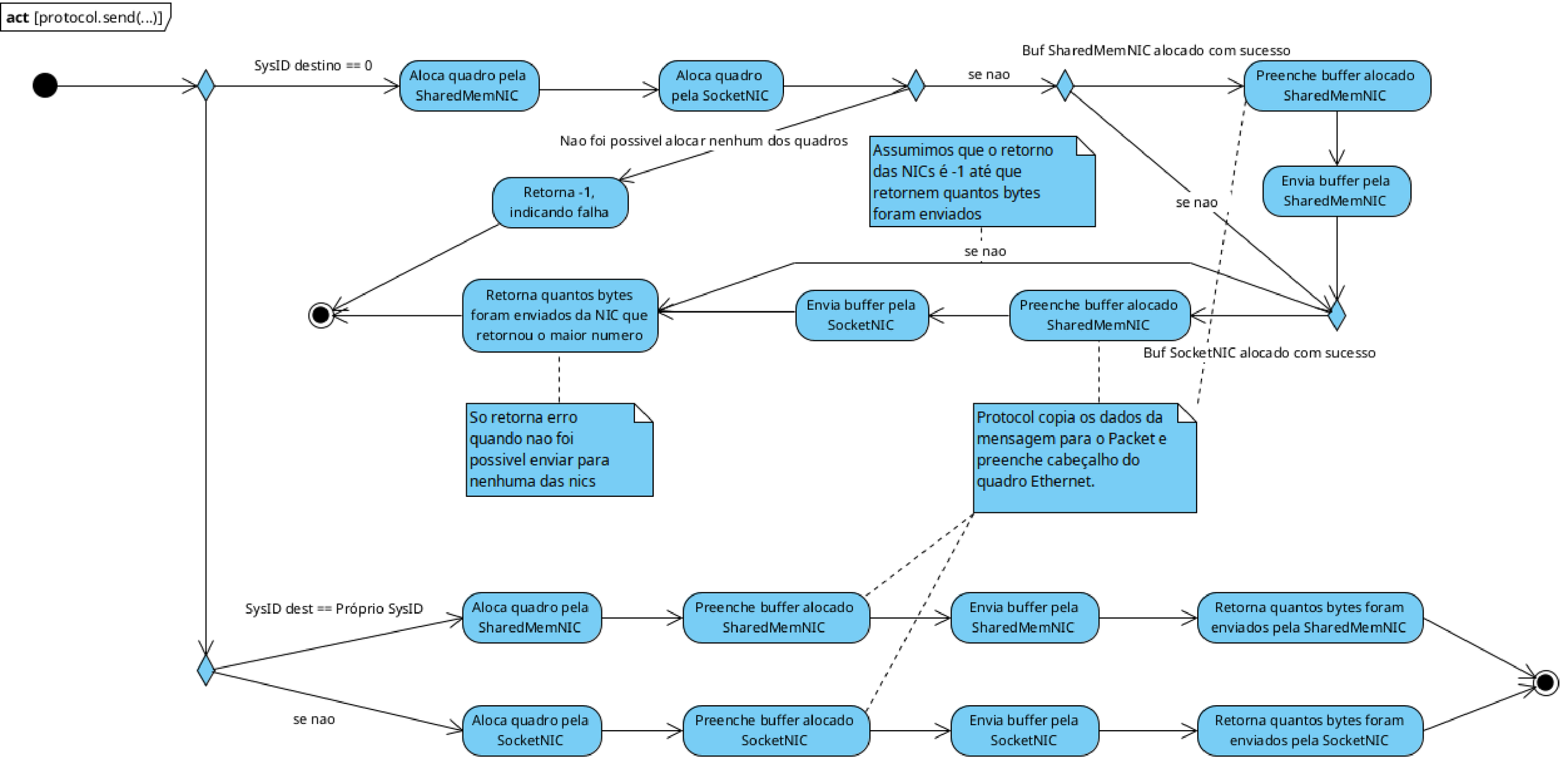

sd [Recv Thread]
: SocketEngine
: SocketNIC
1: thread_run
loop
[while (1)]
1.1: engine_cond.wait(engine_lock, [this](){ return ((obj != nullptr && newMessage) || !_thread_running })
Apenas desbloqueia quando uma mensagem chegar ou quando a thread deve ser interrompida
1.2: pthread_mutex_lock(&_threadStopMutex)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

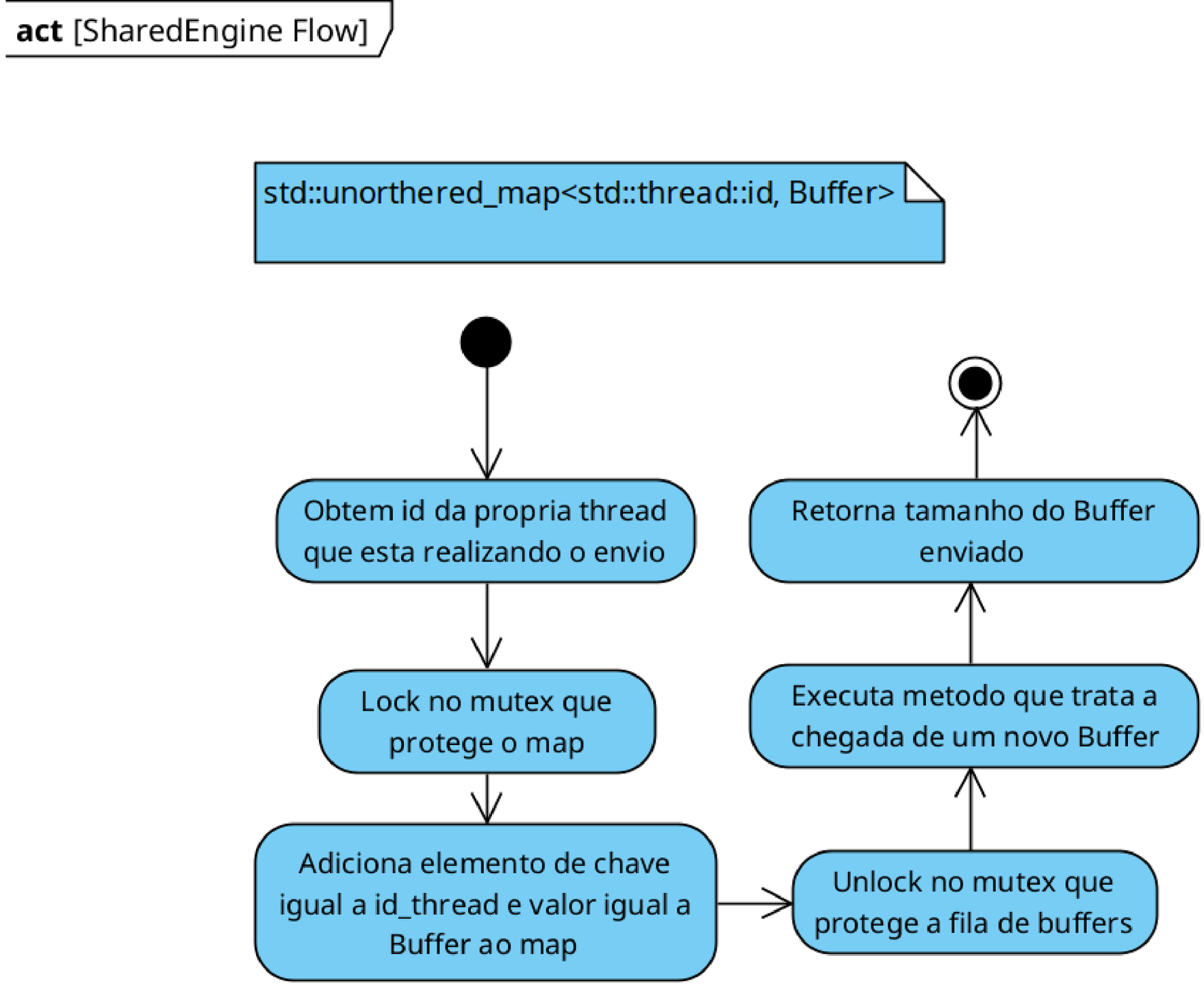

sd [Receive SharedMemEngine]
: SharedEngine
1: receive(buf : Buffer *) : unsigned int
Utilizamos try_acquire para send e receive nao serem bloqueantes quando a fila da engine estiver cheia ou vazia
1.1: bool acquired = full.try_acquire()
alt
[acquired]
1.2: buffer_sem.acquire()
remove buffer da fila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

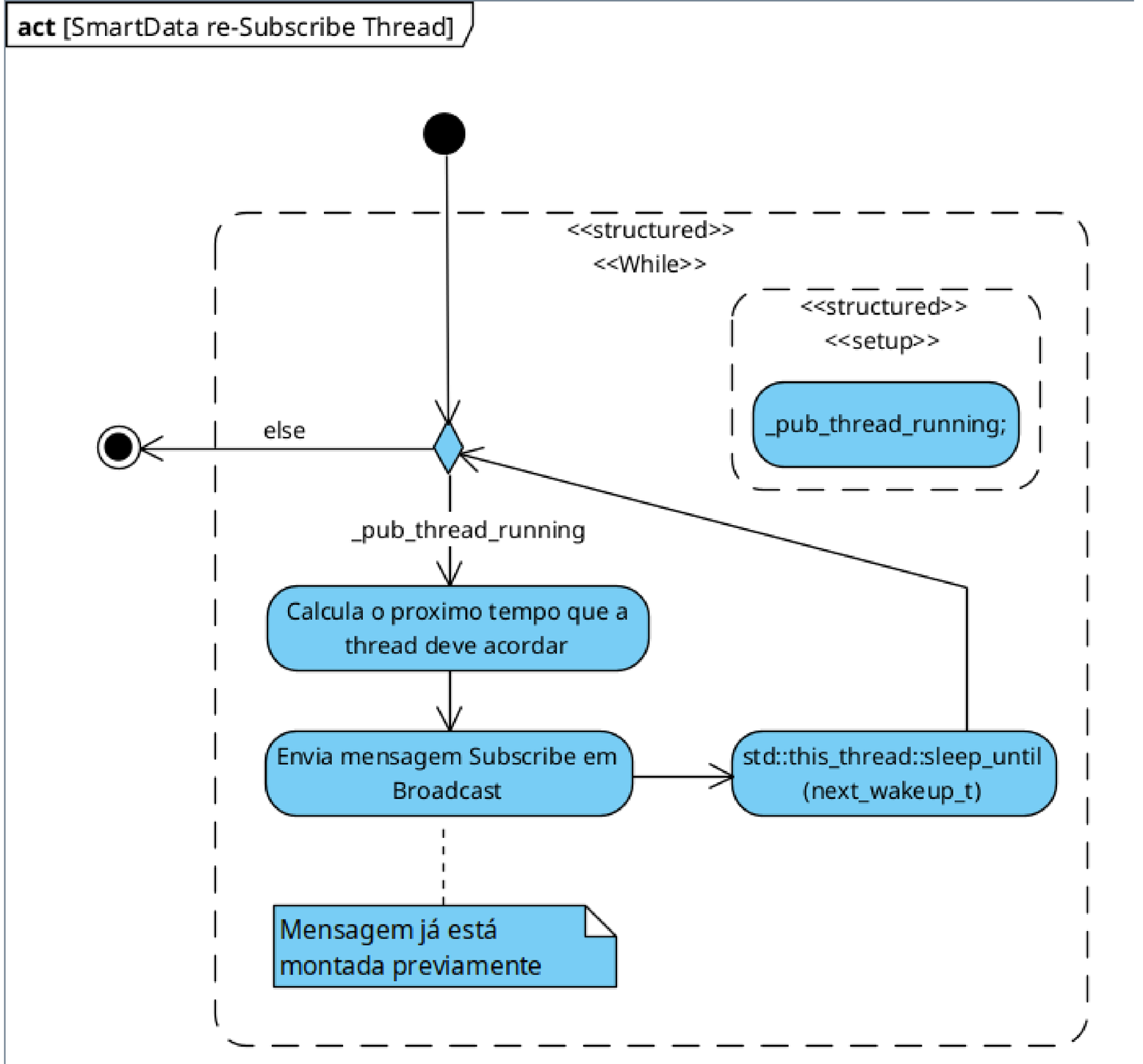

sd [Communicator.update(...)]
: Communicator
: Protocol
: SmartData P
1: update(obs : Channel::Observed*, c : Channel::Observer::Observing_Condition, buf : Buffer*) : void
Message *msg = (Message *)_channel->unmarshal(buf);
alt
[msg.type == Message::Type::COMMOM]
1.1 : Observer::update(c, buf)
[else]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

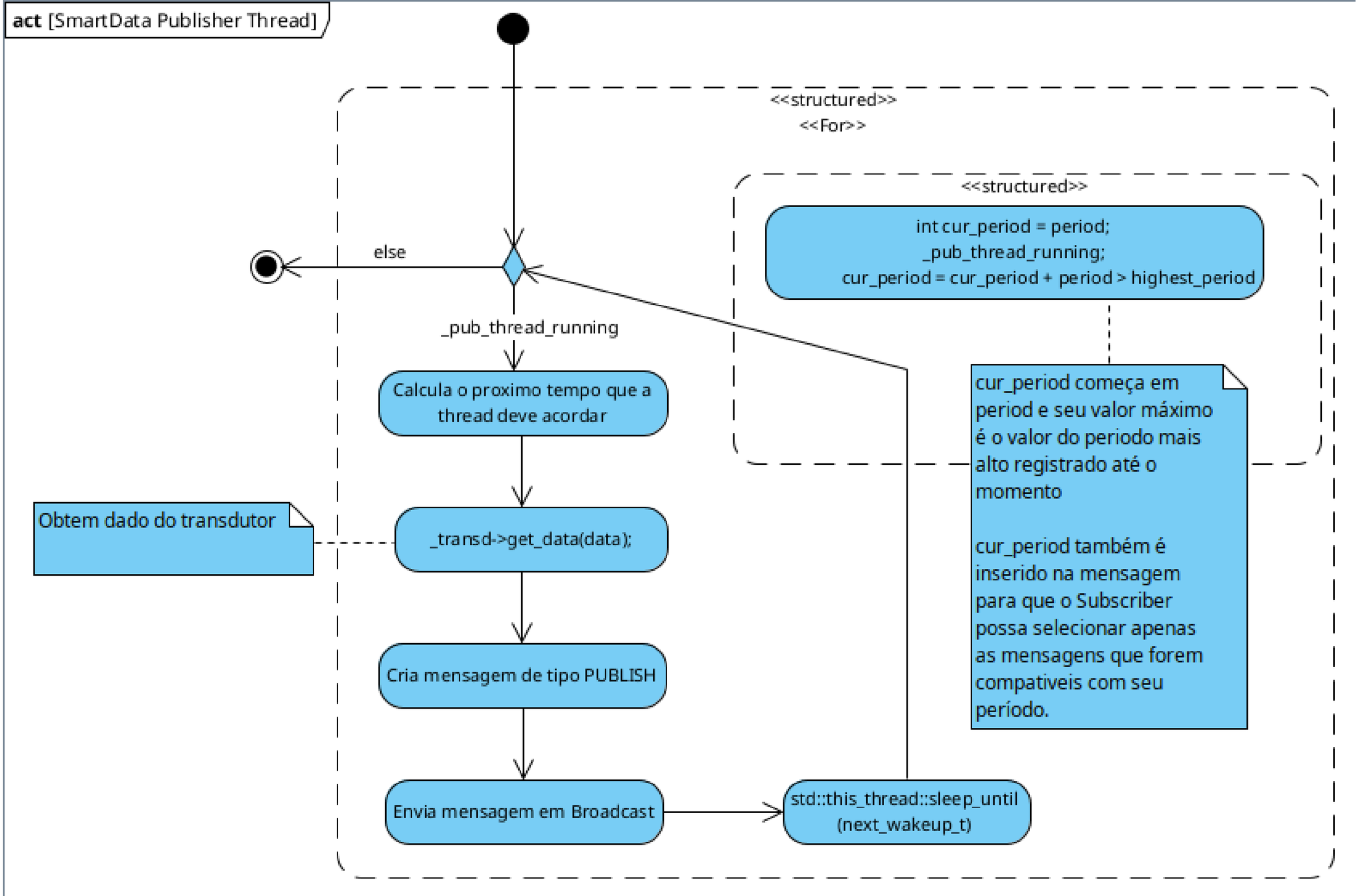
Diagramas de Atividade

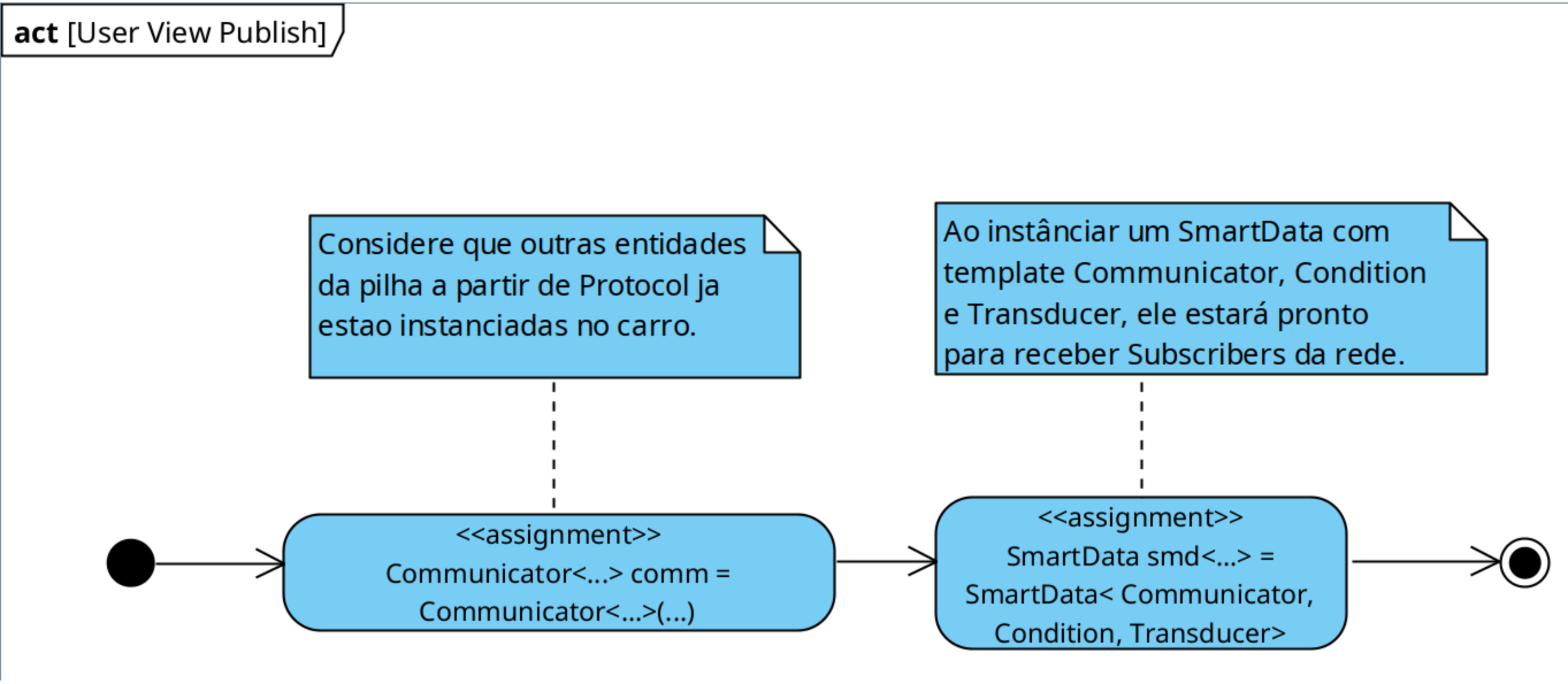

act [User View Send]
Cada Communicator é identificado unicamente pelo seu Address
Mensagem tem o seguinte formato:
Address ::= SEQUENCE {
mac OCTET STRING (SIZE(6)),
sysID OCTET STRING (SIZE(4)),
port OCTET STRING (SIZE(2))
}
Message ::= SEQUENCE {
destAddress Address,
srcAddress Address,
type OCTET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

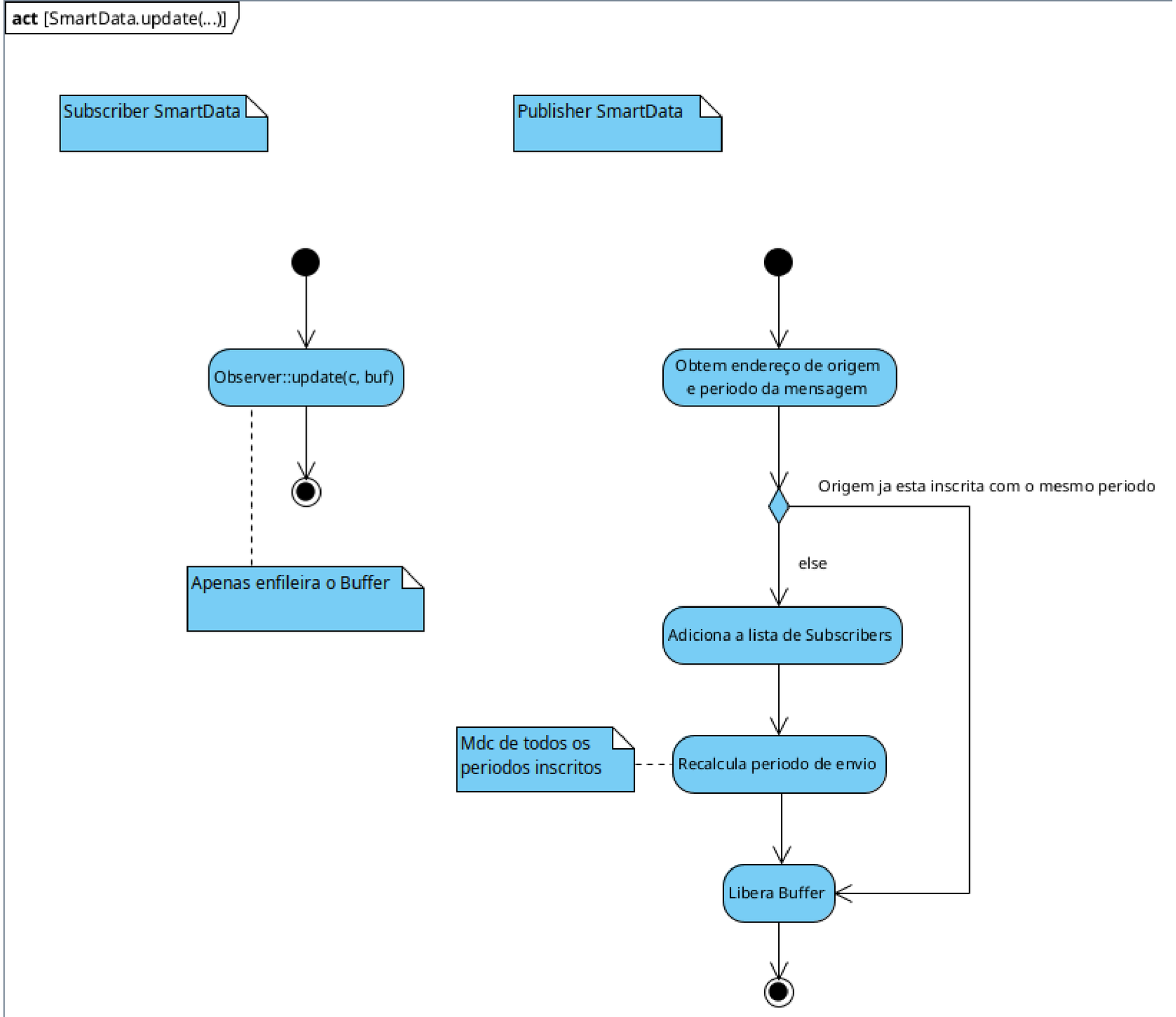

act [protocol.send(...)]
SysID destino == 0
Aloca quadro pela SharedMemNIC
Aloca quadro pela SocketNIC
Nao foi possivel alocar nenhum dos quadros
Retorna -1, indicando falha
se nao
Buf SharedMemNIC alocado com sucesso
Preenche buffer alocado SharedMemNIC
Envia buffer pela SharedMemNIC
se nao
Assumimos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

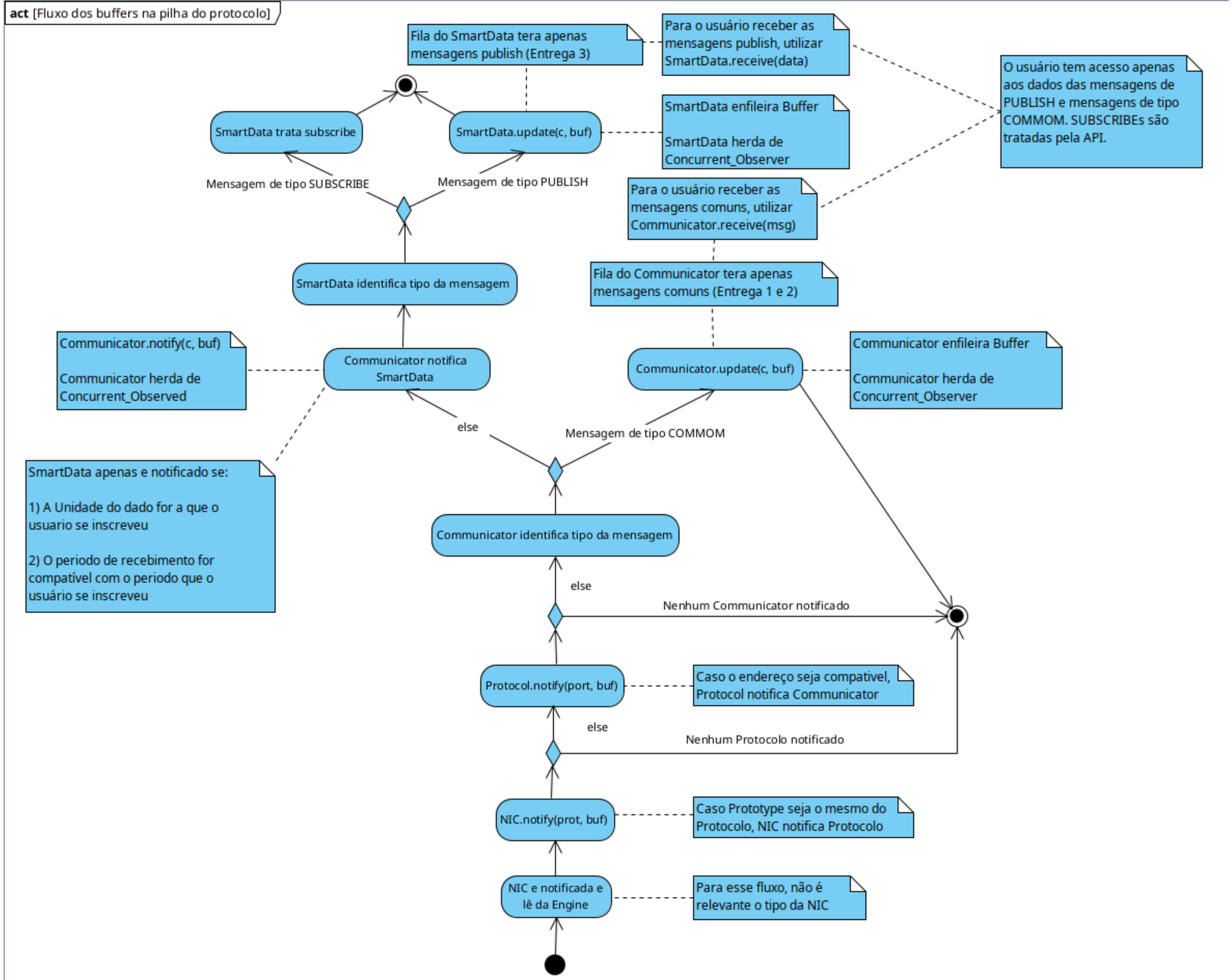

act [SharedEngine Flow]
std::unorthered_map<std::thread::id, Buffer>
Obtem id da propria thread que esta realizando o envio
Lock no mutex que protege o map
Adiciona elemento de chave igual a id_thread e valor igual a Buffer ao map
Unlock no mutex que protege a fila de buffers
Executa metodo que trata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