**Interface:**

**Requests:**

**subscribe (**topic**)**

**unsubscribe (**topic**)**

**publish** **(**topic, m**)**

**Indications:**

**psDelivery** **(**topic, m**)**

**State:**

diameter // diameter of the overlay

mySubscriptions // a map of the subscriptions of the node Map[topic] = {ttl}

radiusSubsByTopic // includes own subscriptions Map[<topic>] = {(<process>, <TTL>)}

subHops // max hops of gossip when renewing subscriptions

pubHops // max hops of gossip when publishing a message of a given topic

neighbors // partial view of the overlay

delivered // set of Ids of messages

pendingSub // subscribe messages to be forwarded

pendingUnsub // unsubscribe messages to be forwarded

pendingPub // publish messages to be forwarded

**Upon Init () do:**

diameter ← ln (#π \* 10)

radiusSubsByTopic ← {}

mySubscriptions ← {}

subHops ← (diameter + 1) / 2

pubHops ← (diameter + 1) / 2

neighbors ← ⊥

delivered ← {}

pendingSub ← {}

pendingUnsub ← {}

pendingPub ← {}

**Setup Periodic Timer RenewSub (T)** // T < ttl

**Setup Periodic Timer CleanExperiedSubs(CLEAN\_FREQUENCY)**

**Upon subscribe (**topic**) do:**

addToRadiusSubs(myself, topic, ttl)

mid ← generateUID({myself, topic, getTimeOfSystem() })

delivered ← delivered U {mid}

pending ← pendingSub U {(**SUB**, myself, topic, TTL, subHops - 1, mid)}

**Trigger** **GetNeighbors ()**

**Upon unsubscribe (**topic**) do:**

removeFromRadiusSubs(myself, topic, ttl)

mid ← generateUID({**UNSUB**, myself, topic, getTimeOfSystem()})

delivered ← delivered U {mid}

pending ← pendingUnsub U {(**UNSUB**, myself, topic, subHops - 1, mid)}

**Trigger GetNeighbors ()**

**Upon publish (**topic, m**) do:**

mid ← generateUID({**PUB**, myself, topic, m})

delivered ← delivered U {mid}

pending ← pendingPub U {(**PUB**, topic, pubHops, m, mid)}

**Trigger GetNeighbors()**

**Upon RenewSub () do:**

**For each** (topic, ttl) ∈ mySubscriptions ∧ ttl < ttl \* 0.2 **do:**

**Trigger subscribe (**topic**)**

**Upon Neighbors (**N**) do:**

neighbors ← N

**For each** (**SUB**, subscriber, topic, ttl, subHops, mid) ∈ pendingSub **do**:

**Trigger Gossip** **(SUB**, subscriber, topic, ttl, subHops, mid**)**

**For each (UNSUB**, unsubscriber, topic, subHops, mid**)** ∈ pendingUnsub **do**:

**Trigger Gossip(UNSUB**, unsubscriber, topic, subHops, mid**)**

**For each (PUB**, topic, m**)** ∈ pendingPub **do**:

**Trigger Gossip(PUB**, topic, pubHops, m, mid**)**

pendingSub ← {}

pendingUnsub ← {}

pendingPub ← {}

**Upon Receive (SUB**, subscriber, topic, ttl, subHops, mid**) do:**

**if** mid ∉ delivered **then**

addToRadiusSubs(subscriber, topic, ttl)

delivered ← delivered U {mid}

**if** subHops > 0 **then**

pending ← pending U {(**SUB**, subscriber, topic, ttl, subHops - 1, mid)}

**Trigger GetNeighbors()**

**Upon Receive (UNSUB**, unsubscriber, topic, subHops, mid**) do:**

**if** mid ∉ delivered **then**

removeFromRadiusSubs(s, topic)

delivered ← delivered U {mid}

**if** subHops > 0 **then**

pending ← pending U {(**UNSUB**, unsubscriber, topic, TTL, subHops - 1, mid)}

**Trigger GetNeighbors()**

**Upon Receive (PUB**, topic, pubHops, m, mid**) do:**

**if** mid ∉ delivered **then**

delivered ← delivered U {mid}

**if** {myself, ttl} ∈ radiusSubsByTopic [topic] ∧ validTll(ttl, p, topic) **then**

**Trigger psDelivery(**topic, m**)**

**For Each** {p, ttl} ∈ radiusSubsByTopic [topic] ∧ validTll(ttl, p, topic) **do:**

**Trigger Send (**topic, m, mid**)**

**if** pubHops > 0 **then**

pending ← pending U {(**PUB**, topic, pubHops - 1, m)}

**Trigger GetNeighbors()**

**Upon Receive (DIRECTMSG**, topic, m, mid**) do:**

if mid ∉ delivered **then**

delivered ← delivered U {mid}

**if** topic ∈ mySubscriptions ∧ validTll(ttl, p, topic) **then**

**Trigger psDelivery(**topic, m**)**

**Upon CleanOldSubs() do:**

**For each** topic,{(process,ttl)} ∈ radiusSubsByTopic ∧ ¬validTll(ttl, process, topic) **do:**

removeFromRadiusSubs(process, topic)

**procedure** removeFromRadiusSubs(process, topic) **do**:

**if** process == myself **then**

mySubscriptions[topic] ← mySubscriptions[topic] \ {-}

radiusSubsByTopic[topic] ← radiusSubsByTopic[topic] \ {(process, -)}

**procedure** addToRadiusSubs(process, topic, ttl) **do**:

**if** process == myself **then**

mySubscriptions[topic] ← mySubscriptions[topic] U {ttl}

radiusSubsByTopic[topic] ← radiusSubsByTopic[topic] U {(process, ttl)}

**procedure** validTll(ttl, process, topic) **do**:

**if** ttl <= 0 **then**

removeFromRadiusSubs(process, topic)

**return** false

**else**

**return** true

//timeout para apagar delivered

//tirar os métodos para apagar/add os dois mapas

**Gestão da Active View**

Estratégia reactiva

**Upon TCP\_DISCONNECT ou BLOCKING do:** //achar um nome para isto

q <- pickRandom(passiveView, 1) // Selecionar 1 nó random q da passive view

status <- TCPConnect(q) // Estabelecer conexão TCP com q

**If** status == INSUCESS **then**

removeNodeFromPassiveView(q)

**Trigger** TCP\_DISCONECT ou BLOCKING

**Else** //conexão tcp teve sucesso

**If** #activeView == 0 **do**

priority <- HIGH

**Else**

priority <- LOW

**TCP\_SEND**(q, NEIGHBOR, myself, priority)//Enviar para q um NEIGHBOR request (com identificador de p e o nível de prioridade)

**Upon NEIGHBOR(**sender, priority**) do**

**If** priority == HIGH **then** //se high aceita sempre

addNodeActiveView(sender)

**If** priority level == LOW **then**

**If** #activeView != activeViewMaxSize //possui um espaço livre na ActiveView

addNodeActiveView(sender)

SendPelaConexãoTCPJaAberta( NEIGHBOR\_ACCEPT)

**Else**

SendPelaConexãoTCPJaAberta NEIGHBOR\_REJECT

Upon SendPelaConexãoTCPJaAberta NEIGHBOR\_ACCEPT do:

Remover o identificador de q da vista passiva

Adicionar o identificador de q à vista activa

Upon SendPelaConexãoTCPJaAberta NEIGHBOR\_ REJECT do

Selecionar outro nó da vista passiva e repetir o procedimento anterior (sem remover q da vista passiva porque ele está bom só nem é espaço para o p na sua active view)

**Gestão da Passive View**

Estratégia cíclica

Upon PVCyclicCheck do:

q <-Escolher um gajo ao random

exchangeList <- id do p, Ka nós da vista activa, Kp nós da vista passiva

//shuffle

Trigger Send(SHUFFLE, q, exchangeList, timeToLive)

Upon Receive(SHUFFLE, exchangeList, timeToLive) do:

timeToLive –

if timeToLive > 0 and #activeView > 1 then

x <- Escolher um gajo ao random da activa (não pode ser o emissor do shuffle aka p)

Trigger Send(SHUFFLE, x, exchangeList, timeToLive)

Else

passiveViewSample <- selectAtRandom(passiveView, #exchangeList)

SendPorTCPTemporário(sender aka p, SHUFFLE\_REPLY, passiveViewSample)

//adicionar a exchange list a passive view (filtrar o myself e elementos que já estejam na pv)

// se pv estiver cheia tirar nós para por estes novos, primeiro tenta-se tirar os nós que foram enviados para o outro peer se mesmo assim não der tira-se aleatoriamente

Upon Receive (SHUFFLE\_REPLY, passiveViewSample) do:

//adicionar a exchange list a passive view (filtrar o myself e elementos que já estejam na pv)

// se pv estiver cheia tirar nós para por estes novos, primeiro tenta-se tirar os nós que foram enviados para o outro peer se mesmo assim não der tira-se aleatoriamente