Apéndice A: Código

Se muestra a continuación el código implementado, que fue organizado de la siguiente manera en archivos tcl:

- **cbtmain.tcl**: Contiene referencias a los demás archivos para facilitar la carga del código con una única sentencia: "source cbtmain.tcl".
- **cbt.tcl**: Contiene procedimientos de la clase CBT para interacción con las demás clases implementadas y el soporte provisto por el simulador.
- **timer.tcl**: Contiene la clase base *CBTTimer* y las derivadas que implementan los timers específicos.
- **agents.tcl**: Contiene las clases que definen los distintos agentes (derivados de *Agent/Message*): agente de emisión/recepción asociado a CBT, agentes de encapsulación y desencapsulación y algunos agentes para prueba del protocolo (*Prueba*, *Emisor_delay*, *Receptor_delay*).
- **simul.tcl**: Contiene métodos agregados a la clase *Simulator*, *map-tree*, *set-core*, etc.
- **debug.tcl**: Contiene métodos de varias de las clases relativos a las funciones de debugging implementadas.
- **recvr.tcl**: Contiene métodos CBT destinados a procesar las PDUs recibidas y al envío de PDUs a otros nodos.
- **interface.tcl**: Código asociado a las interfaces.
- **mfc.tcl**: Código correspondiente a las tablas transitoria y permanente mantenidas en cada nodo.

A.1 cbtmain.tcl

```
source cbt.tcl
source agents.tcl
source timer.tcl
source simul.tcl
source mfc.tcl
source recvr.tcl
source interface.tcl
source debug.tcl
```

A.2 cbt.tcl

```
Class CBT -superclass McastProtocol

# Definicion de constantes

# Tipos de PDU
CBT set HELLO        0
CBT set JOIN_REQUEST    1
CBT set JOIN_ACK     2
CBT set QUIT_NOTIFICATION   3
CBT set ECHO_REQUEST    4
CBT set ECHO_REPLY      5
CBT set FLUSH_TREE      6
```

```
# Timers and contadores (si son redefinidos tener en cuenta que
algunos
# estan expresados en funcion del valor de otros). Los tiempos en
segundos
CBT set HOLDTIME     3
CBT set RTX_INTERVAL     5
CBT set ECHO_INTERVAL    60
CBT set EXPECTED_REPLY_TIME 70
CBT set HELLO_INTERVAL      60
CBT set JOIN_TIMEOUT     [expr [CBT set RTX_INTERVAL] * 3.5]
CBT set TRANSIENT_TIMEOUT   [expr [CBT set RTX_INTERVAL] * 1.5]
CBT set CACHE_DEL_TIMER     [expr [CBT set HOLDTIME] * 1.5]
CBT set GROUP_EXPIRE_TIME   [expr [CBT set ECHO_INTERVAL] * 1.5]
CBT set HELLO_PREFERENCE 255
CBT set MAX_RTX          3


# Direccion multicast routers CBT
CBT set ALL_CBT_ROUTERS 0xFFCF


# Instanciacion de la entidad CBT en el nodo
CBT instproc init { sim node } {
   $self next
   $self instvar ns Node type enbl
   set ns $sim
   set Node $node
   set type "CBT"
   #Inicializa elementos que usara el agente (interfaces, tablas, etc)
   $self initialize
}


# Inicializacion de grupos en el nodo, agente de desencapsulacion,
# interfaces, tablas transitoria y permanente, agente de transmision y
# modulo para debugging
CBT instproc initialize { } {
   $self instvar ns Node messager  decapag joined_groups MFCTable
   $self instvar echorq TransientTable interfaces
   set joined_groups ""
   [$Node getArbiter] addproto $self
   set decapag [new decap]
   $Node attach $decapag
   #Inicializa Tabla Multicast Forwarding Cache
   set MFCTable [new MFC_Table $self]
   #Inicializa el objeto administrador de echo requests
   set echorq [new Echorq_adm $ns $Node $self]
   #Inicializa Tabla Multicast Transient Forwarding Cache
   set TransientTable [new Transient_Table $self]
   #Inicializa interfaces
   $self init_interfaces
   #Agentes messager(intercambio de PDUs) y decapag (decapsulacion)
   set messager [new Agent/Message/CBT]
   $Node attach $messager
   $messager set proto $self
   $messager set dst_ [CBT set ALL_CBT_ROUTERS]
   $Node join-group $messager [CBT set ALL_CBT_ROUTERS]
   #Inicializa modulo para debugging
   $self init_debug
}
```

```
# Arranque del protocolo, variable status en up (McastProtocol).
CBT instproc start {} {
   $self next
}


CBT instproc stop {} {
        $self instvar MRTArray groupArray cacheByGroup sourceArray
        if [info exists MRTArray] {
                unset MRTArray
        }
        if [info exists groupArray] {
                unset groupArray
        }
        if [info exists cacheByGroup] {
                unset cacheByGroup
        }
        if [info exists sourceArray] {
                unset sourceArray
        }
}



##########################################################################
###
# Metodos invocados por la parte de reenvio (upcall)

# Upcall invocado con code, srcID, group, interfaz de entrada
# Es invocado por cache_miss (no existe replicator para el par source
group o
# por wrong_iif (existe el par pero la interfaz de arribo es diferente
a la
# asociada con el replicator.
CBT instproc upcall { argslist } {
        set code [lindex $argslist 0]
        set argslist [lreplace $argslist 0 0]
        switch $code {
                "CACHE_MISS" { $self handle-cache-miss $argslist }
                "WRONG_IIF" { $self handle-wrong-iif $argslist }
                default { puts "codigo de upcall desconocido, $code" }
        }
}


# Para compatibilidad con las funciones del nodo (add-mfc, y del-mfc,
sobre
# todo). Se guarda aqui una lista propia de src, grp, iif para cada
entrada
# que se crea
CBT instproc handle-cache-miss { argslist } {
#  $self instvar sources groups iifs
   $self instvar ns interfaces
   $self instvar rep rep_obj rep_int
        set srcID [lindex $argslist 0]
        set group [lindex $argslist 1]
        set iface [lindex $argslist 2]
        $self instvar Node MFCTable
   #Para grupo CBT manda siempre
   if { $group == [CBT set ALL_CBT_ROUTERS] } {
      #Dada la interfaz de arribo, determina la totalidad de las
```

```
      #interfaces restantes (labels) en el nodo, (outifaces) para
crear los
      #elementos de forwarding para ellas.
      set outifaces ""
      foreach interf [$Node set ifaces_] {
              set la [$interf set id]
              if { $la != $iface } {
                 set tt [$interf set ifaceout]
                 lappend outifaces $tt
              }
           }
      if {![info exists sources]} {
         set sources ""
         set groups ""
         set iifs ""
      }
      #si alguna interfaz es un dynamic link, en lugar de objeto
      #networkinterface debe ponerse el objeto DynaLink
      set cnt 0
      foreach x $outifaces {
         set y [$Node set neighbor_]
         foreach yy $y {
            set xx [$ns set link_([$Node id]:[$yy id])]
            set in [$xx set ifacein_]
            if {$in == $x} {
               set he [$xx set head_]
                    set cl [$he info class]
                    if {$cl == "DynamicLink" } {
                       set outifaces [lreplace $outifaces $cnt $cnt
$he]
               }
            }
         }
         incr cnt 1
      }
      #Agregado del elemento de forwarding
      $Node add-mfc $srcID $group $iface $outifaces
      return
   }

   #Grupo diferente a CBT routers. En este caso, en funcion de la MFC
   #determina las interfaces para el grupo. Luego crea el replicators
   #para S,G y le asocia la interfaz de llegada del paquete; por
ultimo
   #inserta las demas ifaces del grupo como targets del replicator (lo
   #hace utilizando add-mfc).
   #Debe ademas insertarse el nuevo elemento (iface de arribo) en
   #TODOS los replicators para el grupo, para que se le envie info
   #Esto no se puede hacer data driven, entonces lo hace al agregar
   #la entrada en la tabla a traves del proc CBT actualizar-
replicators
   #Si la interfaz de arribo del paquete no corresponde a alguna de
las
   #del grupo, ignora y sale sin actualizar elementos de reenvio.
   set ent [$MFCTable get_entry $group]
   #No existe grupo en tabla, ignora
   if {$ent == -1 } {return}
   #Obtiene interfaces (iflist) y nexthops (iflist1) para el grupo
   set cc [$self get_ifaces $group]
        set iflist [lindex $cc 0]
   set iflist1 [lindex $cc 1]
```

```
   #Existe la iif de entrada para el grupo?
   set ex [lsearch -exact $iflist $iface]
   if {$ex == -1} {return}
   set iflist [lreplace $iflist $ex $ex]
   set iflist1 [lreplace $iflist1 $ex $ex]
   #Debe eliminarse todas las interfaces iguales, porque en un link
   #multiacceso se puede tener mas de una (en el caso de un  DR). Si
se
   #deja mas de una, el DR generara una copia adicional del paquete
   set lx [lsearch -exact $iflist $iface]
   while {$lx != -1} {
      set iflist [lreplace $iflist $lx $lx]
      set iflist1 [lreplace $iflist1 $ex $ex]
      set lx [lsearch -exact $iflist $iface]
   }
   #En olist, se colocan los objetos interface (el head_ del link)
   set olist ""
   foreach xx $iflist {
      set ifo [$self lbl2if $xx]
      lappend olist $ifo
   }
   #Para el caso de links unicast, debe reemplazarse el objeto
   #networkinterface por un agente encap. Para acceder a la info
   #respecto del tipo de link, se debe recurrir a la informacion
   #almacenada en cada objeto interfaz (CBT)
   #(Tener en cuenta que en en iflist estan los labels)
   set cnt 0
   foreach inf $iflist {
      #Pregunta por label -2, interfaz local
      if {$inf != -2} {
             set inf1 $interfaces($inf)
         set pp [$inf1 get_linktype]
         if {$pp == 3 || $pp == 4} {
            set nxhop [lindex $iflist1 $cnt]
            set kk [lindex $olist $cnt]
            #Si no existe el agente encap para S,G,iif,oif,nhop, se
crea
            if ![info exists ag($srcID:$group:$iface:$inf:$nxhop)] {
                   set ag($srcID:$group:$iface:$inf:$nxhop) [new
encap $srcID $group $iface $inf $nxhop $Node]
               #puts "Nodo [$Node id] crea
ag($srcID:$group:$iface:$inf:$nxhop)"
               $ns attach-agent $Node
$ag($srcID:$group:$iface:$inf:$nxhop)
               #Reemplaza en olist el objeto por el agente.
               set olist [lreplace $olist $cnt $cnt
$ag($srcID:$group:$iface:$inf:$nxhop)]
                }
         }
      }
      incr cnt 1
   }
   #Si alguna interfaz es un dynamic link, debe reemplazarse
   #el objeto networkinterface por el dynamiclink.
   set cnt 0
   foreach x $olist {
      set c ""
      if {$x != -1} {
        set c [$x info class]
      }
```

```
      #Si es un encap, la salida va unicast, no es necesario testear
el
      #target del replicator
      if {$c != "encap" } {
         set y [$Node set neighbor_]
         foreach yy $y {
            set xx [$ns set link_([$Node id]:[$yy id])]
            set in [$xx set ifacein_]
            if {$in == $x} {
               set he [$xx set head_]
               set cl [$he info class]
               if {$cl == "DynamicLink" } {
                  set olist [lreplace $olist $cnt $cnt $he]
               }
            }
         }
      }
      incr cnt 1
   }
   #Actualizacion de los elementos de reenvio, crea nuevo replicator
   #con los targets
   #Aca es necesario conocer el objeto replicator agregado. Como
   #add-mfc no lo devuelve, se compara una lista previa con la actual
   set previa [$Node array names replicator_]
   $Node add-mfc $srcID $group $iface $olist
   set actual [$Node array names replicator_]
   #Se supone que se agrega un replicator
   set ind [llength previa]
   foreach a $actual {
      set esta 0
      foreach p $previa {
         if {$a == $p} {set esta 1}
      }
      if {$esta == 0} break
   }
   set nrep [$Node set replicator_($a)]
   #Actualizacion de los elementos usados por CBT
   if {![info exists rep($srcID:$group:$iface)]} {
      set rep($srcID:$group:$iface) $nrep
      set rep_obj($srcID:$group:$iface) $olist
      set rep_int($srcID:$group:$iface) $iflist
   }
   #No else, si entro por miss no habia replicator
}


# En este caso se recibe un paquete por s,g por una interfaz diferente
a la
# asociada al replicator. Si bien en CBT no se tiene informacion de
source
# en las tablas, el replicator no puede procesar (reenviar) el paquete
ya
# que se produciria un loop, porque entre las interfaces de salida
podria
# estar la de arribo. Se decide descartar el paquete.
# Normalmente esta situacion no se podria producir en CBT.
CBT instproc handle-wrong-iif {argslist} {
        $self instvar Node MFCTable
   $self instvar sources groups iifs ns interfaces
   $self instvar rep rep_obj rep_int
        set srcID [lindex $argslist 0]
```

```
        set group [lindex $argslist 1]
        set iface [lindex $argslist 2]
   puts "Node [$Node id]  handle-wrong-iif. src:$srcID  gr: $group
iface: $iface"
   #Se debe cambiar la interfaz de arribo. Para ello se busca s,g y se
   #obtiene la interfaz anterior, luego se invoca al cambio con la
   #nueva (la que se informa al handle), si es que esta pertenece a la
   #entrada para el grupo
   #Si nueva interfaz (iface) no esta en el grupo, retorna
   #Obtiene interfaces (iflist) y nexthops (iflist1) para el grupo
   set cc [$self get_ifaces $group]
        set iflist [lindex $cc 0]
   set iflist1 [lindex $cc 1]
   set ex [lsearch -exact $iflist $iface]
   if {$ex == -1} {return}
   #Si nueva interfaz esta en el grupo
   #Busca el par src,grp y obtiene interfaz
   set repli array names rep
   foreach e $repli {
      set xx [split e :]
      set ss [lindex $xx 0]
      set gg [lindex $xx 1]
      set ii [lindex $xx 2]
      if { $ss == $srcID && $gg == $group } break
   }
   set oldif $ii
   set mc [$Node set multiclassifier_]
   $mc change-iface $srcID $group $oldif $iface
   #Cambia la info CBT
   set rep($srcID:$group:$iface) $rep($srcID:$group:$oldif)
   unset rep($srcID:$group:$oldif)
   set rep_obj($srcID:$group:$iface) $rep_obj($srcID:$group:$oldif)
   unset rep_obj($srcID:$group:$oldif)
   set rep_int($srcID:$group:$iface) $rep_int($srcID:$group:$oldif)
   unset rep_int($srcID:$group:$oldif)
   #Si la nueva interfaz de arribo estaba como salida en el rep, debe
   #anularse, de lo contrario ocurririan loops
   set cnt 0
   foreach oif $rep_int($srcID:$group:$iface) {
      if {$oif == $iface} {
      #Si iface == -2, reemplazo?
         set xoif [lindex $rep_obj($srcID:$group:$iface) $cnt]
         set rep_int($srcID:$group:$iface) [lreplace
$rep_int($srcID:$group:$iface) $cnt $cnt]        set rep_obj
         set rep_obj($srcID:$group:$iface) [lreplace
$rep_obj($srcID:$group:$iface) $cnt $cnt]
         $Node del-mfc $srcID $group $xoif
         break
      }
      incr cnt 1
   }
}



#Procedimiento invocado por mfc al agregar una entrada child para un
grupo.
#El nuevo replicator (asociado a la nueva interfaz) sera creado cuando
se
#produzca un cache-miss. Pero hay que actualizar todos los demas
replicators
```

```
#para el grupo, ya que el agregado de un nuevo target en el replicator
no
#es disparado por llegada de datos
CBT instproc actualizar-replicators {group ch_interface} {
        $self instvar Node MFCTable
    $self instvar sources groups iifs ns interfaces
    $self instvar rep rep_obj rep_int
    set oif [$self lbl2if $ch_interface]
    set todos [$self array names rep]
        set y [$Node set neighbor_]
    foreach yy $y {
       set xx [$ns set link_([$Node id]:[$yy id])]
       set in [$xx set ifacein_]
       if {$in == $oif} {
          set he [$xx set head_]
          set cl [$he info class]
          if {$cl == "DynamicLink" } {
             set oif  $he
          }
       }
         }
    if {[llength $todos] >= 1} {
       foreach elem $todos {
              set pp [split $elem :]
          set ss [lindex $pp 0]
          set gg [lindex $pp 1]
          set ii [lindex $pp 2]
               if {$gg == $group} {
             set rrp [$Node set replicator_($ss:$gg)]
                  if {$oif != -1} {
                $rrp insert $oif
                #actualiza CBT
                set ax rep_obj($elem)
                set ax [lappend $ax $oif]
                set rep_obj($elem) $ax
                set ax rep_int($elem)
                set ax [lappend $ax $ch_interface]
                set rep_int($elem) $ax
             }
             #Que pasa si provenia de un join-group, aca queda una
             #unica entrada pero el replicator puede tener mas de
             #una ya que es el join del nodo el que inserta el nuevo
             #target. Tampoco guardo aqui info del agente.Quiza no se
         #use
               if {$ch_interface == -2} {
            set ax rep_obj($elem)
                set ax [lappend $ax $oif]
                set rep_obj($elem) $ax
                set ax rep_int($elem)
                set ax [lappend $ax $ch_interface]
                set rep_int($elem) $ax
             }
               }
          }
    }
}



CBT instproc drop {rep src dst} {
    $self instvar Node
```

```
   puts "Node [$Node id]  drop.  replicator: $rep source: $src dest:
$dst"
}


#Dado un grupo, devuelve una lista con las interfaces involucradas
(sus
#labels) y otra con los nexthops. Como es a efectos de distribuir la
#informacion, no tiene en cuenta parent ni child.
CBT instproc get_ifaces { group } {
        $self instvar Node MFCTable
   #determina interfaz (cld) y nexthop (cld1) parent.
   set cld ""
   set pnt [$MFCTable get_parent $group]
   lappend cld [lindex $pnt 1]
   set cld1 ""
   lappend cld1 [lindex $pnt 2]
   #Si el nodo es core (interfaz -1) , no hay interfaz parent
   if {[lindex $cld 0] == -1 } {
           set cld ""
      set cld1 ""
       }
   #Determina la(s) interfaces y nexthop(s) child
   set iflist [$MFCTable get_child_if $group]
   set iflist1 [$MFCTable get_child_nh $group]
   if { [lindex $iflist 0] != -1} {
      if {[llength $cld] > 0}  {
         lappend iflist $cld
         lappend iflist1 $cld1
      }
   }
   set nlist [list $iflist $iflist1]
   return $nlist
}


##########################################################################
###
#Procedimientos para join y leave

#Este procedimiento es invocado cuando un agente realiza join a un
grupo.
#Es invocado luego del join del nodo. Si el grupo es CBT,no hace el
join ya
#no es necesario arbol de distribucion
#En el caso de IGMP, se debera hacer a traves de un agente local
CBT instproc join-group {group} {
        $self instvar joined_groups Node
   if { $group == [CBT set ALL_CBT_ROUTERS] } { return 0}
        if { [lsearch -exact $joined_groups [expr $group]] == -1 } {
      #Agregado de la entrada leaf
      set r [$self add-leaf $group -2]
      if {$r == 0} {
         lappend joined_groups [expr $group]
              puts "Grupo $group joined."
      }
   }
   puts "Grupo $group, ya joined."

}
```

```
CBT instproc add-leaf {group interface} {
   $self instvar ns interfaces MFCTable TransientTable qntimer
   set gr [expr $group]
   #Nodo es on-tree para el grupo, agrega child
   if { [$MFCTable exists_entry $gr] != -1 } {
      #Ver si hay que validar una relacion parent/leaf-child
      $MFCTable add_child $gr $interface -1
      return 0
   }
   #Nodo no es on-tree para el grupo
   #Existe entrada transitoria para el grupo
   if { [$TransientTable exists_entry $gr] != -1 } {
      #Ver si hay que validar relacion parent/leaf-child
      $TransientTable add_child $gr $interface -1
      return 0
   } else {
   #No existe entrada transitoria
      #Obtiene core para el grupo, por ahora del simulador, esto
      #dependera de si se implementa seteo estatico de los cores o
      #de si hay un procedimiento de  bootstrapping
      #Obtiene nexthop para el core (next_router_to_core)
      set core [$ns get-core $gr]
      if {$core == -1 } {
         puts "No es posible encontrar core para grupo $gr"
         return -1
      }
      set lst [$self next_router $core]
      #No hay ruta al core
      if { [lindex $lst 0] == 0 } {
         return -1
      }
      set iface [lindex $lst 2]
      set nxthop [lindex $lst 1]
      #Si link no mcast, el nexthop es el id del nodo.
      #Si link mcast y DR, el nexhop es el id del nodo
      #Si link mcast y no DR, nexthop es *
      set tpl [$interfaces($iface) get_linktype]
      set dr [$interfaces($iface) is_dr]
      if { $tpl <= 2 &&  $dr != 1} {
            set nxthop  *
      }
      #Ver si hay que validar la relacion parent/child
      #Nodo es leaf, crea asi la entrada transitoria
      set leaf 1
      #Si hay Quit_Notification_Timer para la entrada, lo elimina
      if [info exists qntimer($gr:$iface:$nxthop)] {
         $qntimer($gr:$iface:$nxthop) cancel
      }
      #Crea entrada transitoria
      $TransientTable add_entry $gr $core $iface $nxthop $leaf
      #Agrega child
      $TransientTable add_child $gr $interface -1
      return 0
   }
}


# Invocado por mcastproto cuando un agente deja un grupo. Si no quedan
mas
# agentes locales para el grupo, borra child.
CBT instproc leave-group {group} {
```

```
   $self instvar Node joined_groups
   set ag [$Node get_Agents $group]
   if { [llength $ag] == 0 } {
      set idx [lsearch -exact $joined_groups [expr $group]]
      if {$idx > -1} {
         set joined_groups [lreplace $joined_groups $idx $idx]
         $self delete-leaf $group -2
         puts "Leaving group $group"
      }
   }
}

CBT instproc delete-leaf {group interface} {
   $self instvar MFCTable
   set gr [expr $group]
   if { [$MFCTable exists_entry $gr] != -1 } {
      $MFCTable delete_child $gr $interface -1
      return
   }
}


##############################################################################
####
# Interaccion con el agente para envio de PDUs
# Resuelve la menera de invocar al agente de mensajes CBT, en base al
tipo
# de envio.
CBT instproc send_msg { msg lbloif type dest} {
   $self instvar messager ns Node
   #En primer termino determina el target del messager
(DynamicLink,etc)
        set oif [$self lbl2if $lbloif]
   set x $oif
   set y [$Node set neighbor_]
   foreach yy $y {
      set xx [$ns set link_([$Node id]:[$yy id])]
      set in [$xx set ifacein_]
      if {$in == $x} {
         set he [$xx set head_]
              set cl [$he info class]
              if {$cl == "DynamicLink" } {
                 #set olist [lreplace $olist $cnt $cnt $he]
            set oif $he
              }
          }
   }
   $messager target $oif
   #En segundo termino, determina el destino del messager y su ttl
   if {$type == "M"} {
          $messager set ttl_ 2
      $messager set dst_ [CBT set ALL_CBT_ROUTERS]
   } else {
          $messager set ttl_ 32
          set n [$ns set Node_($dest)]
          set ar [$n set mcastproto_]
          set cb [$ar getType CBT]
          set agen [$cb set messager]
          $ns connect $messager $agen
   }
   #Por ultimo envia PDU
```

```
    $messager transmit $msg
}



##########################################################################
####
#Metodos para configuracion

#Configuracion de Designated Router en vinculos multiacceso
#Se pasa, un nodo de la red, para individualizar la interfaz, y el
#designated router. Esto debe hacerse p/c nodo en un maccess,
manualmente
#Lo deberia hacer a traves de Hello

#Configura el DR en la interfaz del nodo hacia dstnode
CBT instproc set_des_router { dstnode des_router } {
    $self instvar interfaces
    set lab [$self label_to_neighbor $dstnode]
    if {$lab == -1} { return }
    $interfaces($lab) set_des_router $des_router
}

#Devuelve el DR de una interfaz identificada por su label
CBT instproc get_des_router {lbl} {
    $self instvar interfaces
    if { ![info existe interfaces($lbl)] } {
        return -1
    }
    return [$interfaces($lbl) get_des_router]
}

#Configuracion de vinculos solo con capacidad unicast
#Configura el vinculo desde el nodo hacia dstnode como unicast (debe
#configurarse independientemente el otro nodo -dstnode-)
CBT instproc set_unicast { dstnode }  {
    $self instvar interfaces
    set lbl [$self label_to_neighbor $dstnode]
    set lkt [$interfaces($lbl) get_linktype]
    if {$lkt == 1 || $lkt == 2} {
        incr lkt 2
        $interfaces($lbl) set_linktype $lkt
    }
}


#Configuracion de cores
#Convierte al CBT del nodo en core para el grupo dado.
CBT instproc set-core { group } {
    $self instvar  Node ns MFCTable
    #Lista global de cores
    $ns set-core $group [$Node id]
    set gr [expr $group]
    $MFCTable add_core_entry $gr [$Node id]
}



#Protocolo Hello no implementado
CBT instproc recv-hello { msg } {
```

```
   puts "PROTOCOLO CBT $self : RECIBE  HELLO, no implementado"

}



#########################################################################
######
# Otros metodos



CBT instproc get_simulator {} {
   $self instvar ns
   return $ns
}

#Devuelve la multicast forwarding cache del agente CBT
CBT instproc get_MFCTable {} {
   $self instvar MFCTable
   return $MFCTable
}


#Devuelve la multicast transient forwarding cache del agente CBT
CBT instproc get_TMFCTable {} {
   $self instvar TMFCTable
   return $TMFCTable
}

#Devuelve el objeto que administra los timers para echo request
CBT instproc get_Echorq {} {
   $self instvar echorq
   return $echorq
}


#Devuelve el id del nodo
CBT instproc get_nodeid {} {
   $self instvar Node
   return [$Node id]
}


#Dado un rotulo de interfaz, retorna el objeto networkinterface de
salida
CBT instproc lbl2if {lbl} {
   $self instvar Node
   foreach ifx [$Node set ifaces_] {
     set ifc [$ifx set iface]
     set lb [$ifc set intf_label_]
     if {$lb == $lbl} { return [$ifx set ifaceout]}
   }
        return -1
}


#Dado un neighbor, devuelve el label de la interfaz que va hacia el
CBT instproc label_to_neighbor {neighbor} {
   $self instvar Node ns
   set ne [$Node set neighbor_]
```

```
   set cnt [llength $ne]
   if {$cnt > 0} {
      set nids ""
      set idx 0
      while {$idx < $cnt} {
         set nd [lindex $ne $idx]
         lappend nids [$nd id]
         incr idx 1
      }
   }
   set x [lsearch -exact $nids $neighbor ]
   if {$x < 0} {
      puts "Node $neighbor no es neighbor"
      return -1
   }
   set lnk [$ns set link_([$Node id]:$neighbor)]
   set ifn [$lnk set ifacein_]
   set lbl [$ifn set intf_label_]
   return $lbl
}


#Inicializa las interfaces del nodo
CBT instproc init_interfaces {}  {
   $self instvar Node interfaces ns
   #En primer termino determina las interfaces (labels) y por c/u
   #crea un objeto CBTinterface que almacena en interfaces($lbl)
   #por ahora para interfaz local, -2 creada a mano
   set interfaces(-2) [new CBTinterface $self -2]
   $interfaces(-2) set_linktype 1
   set li [$Node set ifaces_]
   foreach l $li {
      set ou [$l set ifaceout]
      set lbl [$ou set intf_label_]
      set interfaces($lbl) [new CBTinterface $self $lbl]
   }
   #Determina para cada iff, el tipo de link 1:pap-mcast;
   #2:macc-mcast; 3:pap-nmcast; 4:macc-nmcast.
   #Hasta ahora, ns solo permite multicast (todo es multicast), si
   #luego se quiere poner alguna como unicast, debe hacerselo
   #explicitamente
   #Existe una manera mejor de hacerlo?
   set ne [$Node set neighbor_]
   foreach nn $ne {
      set lnk [$ns set link_([$Node id]:[$nn id])]
      set ifn [$lnk set ifacein_]
      set lbl [$ifn set intf_label_]
      #Aca podria seleccionarse no hacer dos veces el poner el tipo
      #en las ifaces maccess, a mejorar posteriormente
      set typ 1
      if {[$lnk info class] == "DummyLink"} {
         set typ 2
      }
      $interfaces($lbl) set_linktype $typ
   }
}



#Se borra un child para el grupo. Pasos:
```

```
#1-Buscar todos los replicators <cualquier src,grupo>
#    si el iif del replicator es la if child, inhabilitar todos los
oifs
#    si el iif del replicator NO es la if child, inhabilitar el oif =
child
#Para tener en cuenta: al handle cache miss entra una sola vez, cuando
crea
#el replicator. Por eso, al inhabilitar replicators, no se debe
sacarlos de
#las listas <sources, ....).
CBT instproc borrar_child {group child} {
   $self instvar sources iifs groups MFCTable Node
   $self instvar rep rep_int rep_obj
   set ent [$MFCTable get_entry $group]
   #No group
   if {$ent == -1 } {return}
   #lista de interfaces para el grupo, parent y child
   set cld ""
   set pnt [$MFCTable get_parent $group]
   lappend cld [lindex $pnt 1]
   if {[lindex $cld 0] == -1 } { set cld ""}
   set iflist [$MFCTable get_child_if $group]
   if { [lindex $iflist 0] != -1} {
      if {[llength $cld] > 0}  {
         lappend iflist $cld
      }
   }
   #Primero elimina el replicator asociado al grupo y al rotulo de
interfaz
   #(puede ser uno o mas, en el caso de dos sources enviando por la
misma iif)
   #Si detecta igual grupo y alguna iface de salida igual a la
recibida,
   #elimina el target
   set todos [$self array names rep]
   if {[llength $todos] >= 1} {
      foreach elem $todos {
              set pp [split $elem :]
         set ss [lindex $pp 0]
         set gg [lindex $pp 1]
         set ii [lindex $pp 2]
         #Elimina todos los targets
              if {$gg == $group && $ii == $child} {
            #Actualiza replicators
            set rrp [$Node set replicator_($ss:$gg)]
            set targets [$rrp array names elements_]
            $Node del-mfc $ss $gg $targets
            #Actualiza CBT
            unset rep($ss:$gg:$ii)
            unset rep_obj($ss:$gg:$ii)
            unset rep_int($ss:$gg:$ii)
         } else {
            #Para todo el grupo (la entrada a eliminar) ve si la iif
es
            #de salida
            if {$gg == $group} {
               set e [lsearch -exact $rep_int($ss:$gg:$ii) $child]
               if { $e != -1 } {
                  #Elimina la interfaz de salida del replicator
                  set rp $rep($ss:$gg:$ii)
                  set obj [lindex $rep_obj($ss:$gg:$ii) $e]
```

```
                  if { $obj != -1 } {
                     $Node del-mfc $ss $gg $obj
                  }
                  #Actualiza CBT
                  set $rep_int($ss:$gg:$ii) [lreplace
$rep_int($ss:$gg:$ii) $e $e]
                  set $rep_obj($ss:$gg:$ii) [lreplace
$rep_obj($ss:$gg:$ii) $e $e]
                  if { [llength $rep_int($ss:$gg:$ii)] == 0} {
                     unset $rep($ss:$gg:$ii)
                     unset $rep_int($ss:$gg:$ii)
                     unset $rep_obj($ss:$gg:$ii)
                  }
               }
            }
         }
      }
   }

   #Recorre la lista propia que guarda CBT con S,G,iif
   #Esto no es necesario ya que con el esquema de replicators de Ns
   #no es posible un par S,G asociado a dos interfaces de entrada, es
   #decir, bastara con encontrar el replicator_(S,G)
   #Primer paso: para el replicator de entrada inhabilitar todos los
   #targets
}

#Determina proximo router camino al core y la interfaz asociada
#target: id del nodo core
#Devuelve una lista con los elementos:
#code: codigo de retorno (0 OK, 1 error)
#nexthop: id del nodo next-hop camino al core
#if_lab: label de la interfaz
#if_obj: objeto networkinterface
CBT instproc next_router {target} {
        $self instvar ns Node
        set lr ""
        set code 0
   #Determinar proximo router camino al core
   set nexthop [ [ $ns set routingTable_] lookup [$Node set id_]
$target ]
        #No hay ruta al core del grupo
   if {$nexthop == -1} {
      puts "Nodo [$Node id]: JR .No ruta al core $target"
           lappend lr $code
      return $lr
   }
        #Es este nodo el next hop al core
   if {[$Node id] == $nexthop} {
      puts "Nodo $nexthop: JR para CORE $target "
      puts "Descartado porque no hay ruta al core"
           lappend lr $code
      return $lr
   }
   #En base al next hop para el core, obtiene la interfaz
   set ifaz [$Node get-oif [$ns set link_([$Node id]:$nexthop)]]
        set if_lab [lindex $ifaz 0]
        set if_obj [lindex $ifaz 1]
        set code 1
        lappend lr $code
        lappend lr $nexthop
```

```
        lappend lr $if_lab
        lappend lr $if_obj
        return $lr
}

#Dada una interfaz de arribo y un origen o tipo de envio devuelve una
lista
#con los grupos que son parents para el arribo.
#Los pares pueden ser (iif, nxthop) si el envio fue unicast
#o (iif, *) si el envio fue multicast
CBT instproc get_parents {ifce nhop} {
   $self instvar MFCTable
   set lst ""
   set gr [$MFCTable get_groups]
   foreach g $gr {
      set ex [$MFCTable lookup_child $g $ifce $nhop]
      if {$ex != -1} {lappend lst $g}
   }
   return $lst
}


#Dada una interfaz y una lista de grupos (-1 significa todos) devuelve
#los grupos de la lista para los cuales la interfaz es parent
CBT instproc if_parent {interface listgroups} {
   $self instvar MFCTable
   if { [lindex $listgroups 0] == -1 } {
      set listgroups [$MFCTable get_groups]
   }
   set xlist ""
   foreach g $listgroups {
      set x [$MFCTable get_parent $g]
      if {[lindex $x 1] == $interface} {
         lappend xlist $g
      }
   }
   return $xlist
}



#Dado un conjunto de grupos, y una interfaz, devuelve la interseccion
#entre el conjunto de grupos parametro y el conjunto de grupos child
de la
#interfaz.
#Si conjunto vacio devuelve -2
#Devuelve  -1 conjunto grupos parametro igual a conjunto child
#Devuelve ademas los nexthops para cada interfaz
CBT instproc childs {grps} {
   $self instvar MFCTable ifs nhp
   set gr [$MFCTable get_groups]
   foreach g $gr {
      set lch [$MFCTable get_child_if $g]
      set lnh [$MFCTable get_child_nh $g]
      set cnt -1
      foreach ii $lch {
      incr cnt 1
         if {![info exists ifs($ii)]} {
            set l ""
            set ifs($ii) $l
            set nhp($ii) [lindex $lnh $cnt]
```

```
         }
         set l $ifs($ii)
         lappend l $g
         set ifs($ii) $l
      }
   }
   set aa [array names ifs]
   foreach a $aa {
      set l1 $ifs($a)
      set l2 ""
      foreach g $grps {
         set s [lsearch -exact $l1 $g]
         if {$s > -1} {
            lappend l2 $g
         }
      }
      if { [llength $l1] == [llength $l2] } {
         set l2 ""
         lappend l2 -1
         set ifs($a) $l2
      } else {
         if { [llength $l2] == 0} {
            lappend l2 -2
            set ifs($a) $l2
         } else {
            set ifs($a) $l2
         }
      }
   }
}


#Determina si la relacion parent child es correcta. Devuelve 0 si lo
es, 1
#si no
# Link PaP con o sin  mcast:if child no puede coincidir con if parent
# Link maccess con mcast: si nodo NO DR, if child no puede coincidir
con parent
# Link maccess sin mcast: si nodo es DR, pares (if,nxthop) no pueden
coincidir
# Link maccess sin mcast: si nodo NO DR, if child no puede coincidir
con parent
CBT instproc child_is_valid { int1 nhop1 int2 nhop2 } {
   $self instvar interfaces
   if {$int1 == $int2 && $nhop1 == $nhop2} {return 1}
   if {$int1 == $int2} {
      #set itype [$self ifce_type $int1]
      set itype [$interfaces($int1) get_linktype]
      set dr [$interfaces($int1) is_dr]
      switch itype {
        case 1 {return 1}
        case 3 {return 1}
        case 2 {if { $dr != 1} {return 1}}
        case 4 {if { $dr != 1} {return 1}}
        }
      }
   return 0
}
```

A.3 timer.tcl

```
#########################################################################
######
#Timer handlers

#Los objetos timer, de clases propias CBT, se ejecutan un cierto
numero de
#veces (puede ser una) ejecutando una determinada accion (por ejemplo,
#enviando una PDU). Cuando el timer finaliza (luego de una o mas
ejecuciones
#del proc timeout definido para la nueva clase), la clase CBTtimer se
#encarga de hacer un delete del objeto timer y llamar a una funcion
del
#objeto cuya clase se deriva de timer. Los metodos a crear son tout y
tfin



Class CBTtimer

CBTtimer instproc init {tid simulator cbt tpars fpars} {
   $self instvar id ns fprocc object fparams tprocc tparams cbtagent
   set id $tid
   set fparams $fpars
   set tparams $tpars
   set ns $simulator
   set cbtagent $cbt
   $cbtagent tmr_puts "Node $id (Creando en tiempo: [$ns now])"
   return $self
}

#El sched contiene el tiempo del timer y se agrega las veces q se
ejecutara
CBTtimer instproc sched { delay times} {
   $self instvar cnt del ns id_
   if [info exists id_] {
      puts "Timer ya activo"
      return
   }
   set cnt $times
   set del $delay
   set id_ [$ns at [expr [$ns now] + $del] "$self timeout"]
}


#El resched larga de nuevo un timer ya existente
CBTtimer instproc resched { delay times} {
   $self instvar cnt del id ns id_ cbtagent
   $cbtagent tmr_puts "Node $id (Rescheduling en tiempo: [$ns now])"
   if [info exists id_] {
           $ns cancel $id_
           unset id_
       }
   set cnt $times
   set del $delay
   set id_ [$ns at [expr [$ns now] + $del] "$self timeout"]
}


CBTtimer instproc sched_ { delay times} {
   $self instvar cnt del ns id_
```

```
   set id_ [$ns at [expr [$ns now] + $del] "$self timeout"]
}


CBTtimer instproc cancel {} {
   $self instvar ns id_ object fprocc fparams tproc tparams id
cbtagent
   if [info exists id_] {
      $ns cancel $id_
      unset id_
   }
   $cbtagent tmr_puts "Node $id (Cancelando en tiempo: [$ns now])"
   #delete $self, despues del return !!! (si no se cuelga)
   $ns at [$ns now] "delete $self"
       $self tfin CANCEL $fparams
}


CBTtimer instproc timeout {} {
   $self instvar id cnt del ns object fprocc fparams tprocc tparams
   $self instvar cbtagent
       if {$cnt == "*"} {
   $cbtagent tmr_puts "Node $id (Timeout en tiempo: [$ns now])"
          $self sched_ $del 0
      $self tout $tparams
          return
       }
   incr cnt -1
   if {$cnt <= 0} {
      #delete $self, despues del return !!!
      $ns at [$ns now] "delete $self"
      #Aca hay que invocar al proceso timeout que qquiero que se
ejecute c/vez q
      #vence el timer.
          $self tout $tparams
      #Aca invoca al proc final
      $self tfin NORMAL $fparams
      return
   }
   if {$cnt != 0} {
      $self sched_ $del 0
      $cbtagent tmr_puts "Node $id (Timeout en tiempo: [$ns now])"
      #Aca hay que invocar al proceso timeout que qquiero que se
ejecute c/vez q
      #vence el timer.
      $self tout $tparams
      return
   }
}



#############################################################################
#######

#Clase Delete_Entry_Timer: representa objetos timer que invocan al
delete
#entry al finalizar.

Class Delete_Entry_Timer  -superclass CBTtimer
```

```
#Timeout no hace nada (se invoca con 0 reintentos), solo tiene efecto
el cancel
Delete_Entry_Timer instproc tout {tparams} {
}

#Procedimiento que se ejecuta al vencer el tiempo
#Invoca al borrado de la entrada correspondiente
#Recibe como parametros la tabla mfc y el grupo a borrar
Delete_Entry_Timer instproc tfin {code fparams} {
   $self instvar ns
   #puts "Tiempo: [$ns now]: Eliminacion de entrada on tree, grupo
[lindex $fparams 1]"
   [lindex $fparams 0] delete_entry [lindex $fparams 1]
   #Debe reengancharse los miembros locales del grupo
   $self instvar cbtagent
   $cbtagent reeng [lindex $fparams 1]
}


##########################################################################
#######

#Clase Delete_T_Entry_Timer: representa objetos timer que invocan al
delete
#entry al finalizar.Pero al transient, que requiere code de
terminacion

Class Delete_T_Entry_Timer  -superclass CBTtimer

#Timeout no hace nada (se invoca con 0 reintentos), solo tiene efecto
el cancel
Delete_T_Entry_Timer instproc tout {tparams} {
}

#Procedimiento que se ejecuta al vencer el tiempo
#Invoca al borrado de la entrada correspondiente
#Recibe como parametros la tabla mfc y el grupo a borrar
Delete_T_Entry_Timer instproc tfin {code fparams} {
   $self instvar ns
   #puts "Tiempo: [$ns now]: Eliminacion de entrada transitoria, grupo
[lindex $fparams 1]"
   [lindex $fparams 0] delete_entry [lindex $fparams 1]
}


##########################################################################
########
#Clase Echo_Request_Timer: Timer encargado de enviar echo request

Class Echo_Request_Timer -superclass CBTtimer

#Cada vez que se cumple un timeout, se envia un echo request
Echo_Request_Timer instproc tout {params} {
   set msg   [lindex $params 0]
   set iflbl [lindex $params 1]
        set envio [lindex $params 2]
        set dest  [lindex $params 3]
   set cbt   [lindex $params 4]
   #puts " $self Enviando echo request ...($msg) a Node $dest"
   $cbt send_msg $msg $iflbl $envio $dest
}
```

```
Echo_Request_Timer instproc tfin {code params} {
   $self instvar echorq_timer
   set interface [lindex $params 0]
        set nexthop [lindex $params 1]
   set adm_echorq [lindex $params 2]
        set exists [$adm_echorq exists $interface $nexthop]
   if {$exists == 1} {
            $adm_echorq elim $interface $nexthop
      #puts " $self unsetting echo_request($interface:$nexthop)"
   }
}

#########################################################################
######
#Clase Quit_Not_Timer: cada vez que se cumple timeout, envia qn

Class Quit_Not_Timer -superclass CBTtimer

#tfin no hace nada
Quit_Not_Timer instproc tfin {code params} {
   set gr [lindex $params 0]
   set pif [lindex $params 1]
   set dest [lindex $params 2]
   set cbt [lindex $params 3]
   $cbt unset qntimer($gr:$pif:$dest)
}


Quit_Not_Timer instproc tout {params} {
   set msg   [lindex $params 0]
   set iflbl [lindex $params 1]
        set envio [lindex $params 2]
        set dest  [lindex $params 3]
   set cbt   [lindex $params 4]
   $cbt send_msg $msg $iflbl $envio $dest
}

#########################################################################
######
#Clase Resend_Join_Timer: Se utiliza para eliminar una entrada
temporaria
#por timeout no hace nada, por final invoca al delete entry

Class Resend_Join_Timer -superclass CBTtimer

#tfin no hace nada
Resend_Join_Timer instproc tfin {code params} {
}

Resend_Join_Timer instproc tout {params} {
   set msg   [lindex $params 0]
   set iflbl [lindex $params 1]
        set envio [lindex $params 2]
        set dest  [lindex $params 3]
   set cbt   [lindex $params 4]
   $cbt send_msg $msg $iflbl $envio $dest
}
```

```
#########################################################################
######
#Clase Send_Join_Timer: cada vez que se cumple timeout, envia join
request

Class Send_Join_Timer -superclass CBTtimer

#tfin no hace nada
Send_Join_Timer instproc tfin {code params} {
}


Send_Join_Timer instproc tout {params} {
   set msg   [lindex $params 0]
   set iflbl [lindex $params 1]
        set envio [lindex $params 2]
        set dest  [lindex $params 3]
   set cbt   [lindex $params 4]
   $cbt send_msg $msg $iflbl $envio $dest
}



#########################################################################
######
#Clase Cache_Del_Timer: cada vez que se cumple timeout, elimina la
entrada
#child para el grupo

Class Cache_Del_Timer -superclass CBTtimer

#tfin unsetting de cache_del_timer de cbt
Cache_Del_Timer instproc tfin {code params} {
   set gr [lindex $params 0]
   set iif [lindex $params 1]
   set cbt [lindex $params 2]
   $cbt unset cache_del_timer($gr:$iif)
}


Cache_Del_Timer instproc tout {tparams} {
   [lindex $tparams 0] delete_child [lindex $tparams 1] [lindex
$tparams 2] [lindex $tparams 3]
}
```

A.4 agents.tcl

```
#########################################################################
######
# El agente de transmision y recepcion es el encargado de enviar y
recibir
# las PDUs CBT invocando en este ultimo caso a la funcion
correspondiente del
# protocolo

Class Agent/Message/CBT -superclass Agent/Message


# transmit tiene en cuenta directivas de debugging y luego invoca al
metodo
```

```
# send definido en la clase Agent/Message
Agent/Message/CBT instproc transmit msg {
   $self instvar proto node_
   $proto debug_send  $msg
   $self send $msg
}


# handle es invocado cada vez que llega un mensaje direccionado
(unicast o
# multicast al agente. Realiza una decodificacion parcial que le
permite
# determinar el rotulo de la interfaz local de arribo y el tipo de
PDU. En
# base a el invoca al metodo CBT correspondiente
Agent/Message/CBT instproc handle msg  {
   $self instvar proto  node_ iif_
   #Obtencion de la interfaz de arribo. En base al ultimo de los
campos
   #de la PDU, router origen, se obtiene el nodo adyacente (se supone
   #que no hay politicas de ruteo implementadas, ya que Ns no lo
soporta)
   set simul [$proto set ns]
   set rr [$simul set routingTable_]
   set list [split $msg /]
   set ll [llength $list]
   incr ll -1
   set nnod [lindex $list $ll]
   set origen [$rr lookup [$node_ id] $nnod]
   set oi [$node_ get-oif [$simul set link_([$node_ id]:$origen)]]
   set oi1 [lindex $oi 1]
   set iif_ [$oi1 set intf_label_]
   set pdu_type [lindex $list 0]
   set list [lreplace $list 0 0]
   $proto debug_recv $msg
   switch $pdu_type {
      0       { $proto recv-hello $list }
      1       { $proto recv-join_request $list }
      2       { $proto recv-join_ack $list }
      3       { $proto recv-quit_notification $list }
      4       { $proto recv-echo_request $list }
      5       { $proto recv-echo_reply $list }
      6       { $proto recv-flush_tree $list }
      default { puts "$self Tipo de CBT recibida desconocido:
$pdu_type" }
   }
}



#########################################################################
######
#Agentes para prueba del protocolo

Class Agent/Message/Prueba -superclass Agent/Message

Agent/Message/Prueba instproc handle { msg } {
   $self instvar node_
   set ns [$node_ set ns_]
   puts "Node:[$node_ id] Agent/Message/Prueba recibe: $msg at [$ns
now]"
```

```
}


Agent/Message/Prueba instproc transmit {msg} {
   $self instvar ttl_ node_
   set ns [$node_ set ns_]
#       puts "Node:[$node_ id] Agent(self: $self  ttl: $ttl_) send:
$msg at [$ns now]"
   #OJO con ttl, cuando era 2 propagaba JR a otros agentes cbt, ahora
   #,1, solo lo envia al local
   #$self set ttl_ 2
   $self send $msg
}



# Los agentes Receptor_delay y Emisor_delay se utilizan para medicion
de
# demoras punta a punta, desde una fuente a un destino en particular.
#
# El agente de recepcion recibe un descriptor de archivo donde generar
# informacion correspondiente a cada paquete recibido. Esta
informacion incluye
# numero de secuencia del paquete, tiempo de envio y tiempo de demora.
#
# El agente de emision recibe un intervalo de tiempo para enviar sus
paquetes
# regularmente al grupo multicast, desde que es invocado su arranque
hasta
# su terminacion. Esta caracteristica puede cambiarse para que genere
trafico
# de acuerdo a laguna distribucion.
#
# Ambos agentes deben configurarse con la direccion adecuada

Class Agent/Message/Receptor_delay -superclass Agent/Message

Agent/Message/Receptor_delay instproc init {file} {
   $self instvar file_
   set file_ $file
   $self next
}

Agent/Message/Receptor_delay instproc handle msg {
   $self instvar node_ file_
   set ns [$node_ set ns_]
   set list [split $msg /]
   set nod [lindex $list 0]
   set time [lindex $list 1]
   set seq [lindex $list 2]
   set n [$ns now]
   set dif [expr $n - $time]
# set dif 5
#       puts "Node:[$node_ id] Agente recibe: $msg at [$ns now]
(delay: $dif)"
       puts $file_ "Node:[$node_ id] Recibe:$msg at [$ns now] Delay:
$dif"
#  puts $file_ "$nod [$node_ id] $seq $dif"
}
```

```
Class Agent/Message/Emisor_delay -superclass Agent/Message

Agent/Message/Emisor_delay instproc init {interval file} {
   $self instvar interval_ fid_ stop file_
   set interval_ $interval
   set file_ $file
   set fid_ 1
   set stop 1
   $self next
}

Agent/Message/Emisor_delay instproc start {} {
   $self instvar seq stop
   set seq 1
   $self transmit
}

Agent/Message/Emisor_delay instproc transmit {} {
   $self instvar ttl_ node_ interval_ fid_ file_ seq stop
   set ns [$node_ set ns_]
   set msg "[$node_ id]/[$ns now]/$seq"
   incr seq 1
        puts $file_ "Node:[$node_ id] Envia: $msg at [$ns now] "
#       puts "Node:[$node_ id] Agente(self: $self  ttl: $ttl_) send:
$msg at [$ns now]"
   if {$stop == 1} {
      $ns at [expr [$ns now] + $interval_] "$self transmit"
   }
   $self send $msg
}


Agent/Message/Emisor_delay instproc stop {} {
   $self instvar stop
   set stop 0
}




# Un objeto encap es el encargado de tomar paquetes multicast que son
para un
# destino unicast o para un tunel, encapsularlos y mandarlos (unicast)
a
# destino.
# Los paquetes llegan a el porque siempre que el proc cache_miss
detecta que
# hay un paquete multicast para una interfaz unicast, en lugar de
poner en
# el replicator un objeto networkinterface o un dynalink, pone un
agente
# encap creado especialmente. Esto quiere decir que para cada
src,dst,iif se
# podran crear varios agentes encap, uno por destino (igual que si
fueran
# objetos networkinterface o dynamiclink).
# La encapsulacion consistira en poner como dato el grupo de destino y
el
# origen del paquete multicast
```

```
# El destino (el decap correspondiente) lo averiguara via ruteo o por
# configuracion manual en caso de tunneling

Class encap -superclass Agent/Message

encap instproc init { src grp iif oif nxhop nod} {
   $self instvar source group iintf ointf nexthop node
   set source $src
   set group $grp
   set iintf $iif
   set ointf $oif
   set nexthop $nxhop
   set node $nod
   #puts "Encap creado, src:$source grp:$group iif:$iintf oif:$ointf"
   $self next
}

encap instproc transmit {msg} {
   $self instvar dst_ source group iintf ointf nexthop node
   #puts "ENCAP: $self (nodo [$node id]: transmite"
   #Determina la direccion del agente decap del nexthop (MFCTable)
   set sim [$node set ns_]
   set nn [$sim set Node_($nexthop)]
   #puts "ES [$nn id]"
   set mc [$nn set mcastproto_]
   set lp [$mc set protocols]
   #Se asume que existe CBT
   set prot -1
   foreach pr $lp {
      set t [$pr set type]
      if { $t == "CBT" } {
         set prot $pr
      }
   }
   set cc [$prot set decapag]
   set dst_ [$cc set addr_]
   #Encapsulacion: source y group se conocen por construccion del
objeto
   set msg "$source/$group/$msg"
   $self send $msg
}

# Handle recibe desde el propio nodo (multiclassifier y replicators
...). Lo
# que hace es ubicar su decap, encapsular y enviar
encap instproc handle {msg} {
   $self transmit $msg
}


# Existe un agente decap por nodo.
# Su funcion es recibir paquetes multicast que han sido encapsulados,
debido
# a que el vinculo de arribo es unicast o a que se ha configurado
# (manualmente y en forma estatica -no admite dynamics-) un tunel. (La
# diferencia entre el tunel y el vinculo unicast es que en este ultimo
caso
# el next hop se obtiene interactuando con la funcion de ruteo (en
este caso
# el ruteo unicast provisto por ns) y en el primero debe ser
configurado
```

```
# manualmente.
# La manera de operar del agente es la siguiente: al recibir un
paquete
# unicast direccionado a el (proveniente de un agente encap remoto)
por una
# interfaz (ns) determinada, toma el origen del paquete (src) y el
grupo de
# destino (group) y genera un paquete multicast con estas
caracteristicas.
# El paquete multicast se inyecta en el nodo, de manera tal que a
efectos
# del manejo multicast (forwarding en base a replicators existentes o
# creacion de nuevos elementos de forwarding en base a las tables de
ruteo
# multicast (MFCTable), se comporte como un paquete normal. Para ello
hay
# que tener en cuenta la interfaz de llegada e inyectarlo (target) en
el
# objeto networkinterface (iface) correspondiente (recordar que el
# forwarding se realiza teniendo en cuenta src, dest e iif)

Class decap -superclass Agent/Message

decap instproc handle {msg} {
   $self instvar node_ dst_ addr_ iif_
#      puts "Decap de node [$node_ id] recibe mensaje: $msg"
   set list [split $msg /]
   set source [lindex $list 0]
   set group [lindex $list 1]
   set msg [lindex $list 2]
   set int -1
   #Se asume que la iif existe
   set xx [$node_ set ifaces_]
   foreach x $xx {
      set l [$x set id]
      if {$l == $iif_} {
         set int $x
      }
   }
   set ifac [$x set iface]
   $self target $ifac
   #addr_ se cambia temporariamente para el send, luego debe
   #restaurarse para que el agente pueda ser encontrado y recibir
   set tmp $addr_
   ####set dst_ 65490
   set dst_ $group
   ####set addr_ 2818
   set addr_ [expr $source << 8]
   $self send $msg
   set addr_ $tmp
}
```

A.5 simul.tcl

```
# Metodos agregados a clases Ns


# Devuelve los agentes locales joined a un grupo
Node instproc get_Agents {group} {
   $self instvar Agents_
```

```
   if { ![info exists Agents_($group)] } {
      return -1
   } else {
      return $Agents_($group)
   }
}


#Proc agregado al simulator para setear cores. Esto se utiliza solo si
los
#CBTs son estaticos. Consiste en tener una lista en el simulador con
#los grupos (grplist) y otra con los nodos core asociados (corelist)
#Aca solo se setean las listas; cada agente cbt es responsable se
colocar su
#nodo (id) y su grupo
Simulator instproc set-core { group node } {
   $self instvar grplist corelist
   if {![info exists grplist]} { set grplist ""}
        if {![info exists corelist]} {set corelist ""}
   lappend grplist $group
   lappend corelist $node
}

# Para obtener el core de unn grupo
Simulator instproc get-core {group} {
   $self instvar grplist corelist
   set idx [lsearch -exact $grplist $group]
   if {$idx < 0 } { return -1 }
   set core [lindex $corelist $idx]
   return $core
}

# Para obtener lista de cores y grupos
Simulator instproc get-cores {} {
   $self instvar grplist corelist
   puts "Cores: $corelist"
   puts "Groups: $grplist"
}


# Muestra por salida standard el arbol de distribucion para  un grupo
Simulator instproc map-tree {group} {
   set lista ""
   set co [$self get-core $group]
   if {$co == -1} {
      puts "No core para grupo $group"
      return
   }
   lappend lista $co
   set longi [llength $lista]
   while {$longi > 0} {
      set co [lindex $lista 0]
      set nodo [$self set Node_($co)]
      set prot [$nodo set mcastproto_]
           set cb [$prot getType CBT]
      set tab [$cb get_MFCTable]
      set lis [$tab get_child_if $group]
      set lis1 ""
      set cnt 0
      while { [llength $lis] > $cnt} {
         set ll [lindex $lis $cnt]
```

```
         if {$ll > -2} {
            set nei [$nodo set neighbor_]
            foreach x $nei {
             set lnk [$self set link_([$nodo id]:[$x id])]
                 set ifn [$lnk set ifacein_]
                 set lbl [$ifn set intf_label_]
                 if {$lbl == $ll} {
                    lappend lis1 [$x id]
                    lappend lista [$x id]
                 }
              }

         }
         incr cnt 1
      }


#SACAR REPETIDOS de lis1 (para vinculos multiacceso)
      set list2 ""
           foreach l1 $lis1 {
         set esta 0
         foreach l2 $list2 {
            if { $l1 == $l2 } {set esta 1}
              }
             if { $esta == 0 } { lappend list2 $l1 }
           }
      set lis1 $list2

# y de lista
      set list2 ""
           foreach l1 $lista {
         set esta 0
         foreach l2 $list2 {
            if { $l1 == $l2 } {set esta 1}
              }
              if { $esta == 0 } { lappend list2 $l1 }
           }
      set lista $list2


      puts "Nodo: $co, child: $lis1"
      set lista [lreplace $lista 0 0]
      set longi [llength $lista]
   }
}
```

A.6 debug.tcl

```
#Procedimientos para debugging
#Son metodos de varias clases agrupados en este archivo

#Lista interfaces
CBT instproc list_interfaces {} {
   $self instvar interfaces Node ns
   puts "Agente CBT en nodo [$Node id].Tiempo: [$ns now]. Interfaces:"
   set inf [array names interfaces]
   if { [llength $inf] == 0 } {
      puts "No existen interfaces"
      return
   } else {
```

```
      foreach g $inf {
         $interfaces($g) list
      }
   }
}


#Lista MFCTable
CBT instproc list_MFCTable {} {
   $self instvar MFCTable Node ns
   puts "Agente CBT en nodo [$Node id]. Tiempo: [$ns now]. MFCTable:"
   $MFCTable list
}



#Lista TransientTable
CBT instproc list_TransientTable {} {
   $self instvar TransientTable Node ns
   puts "Agente CBT en nodo [$Node id]. Tiempo: [$ns now].
Transient_Table:"
   $TransientTable list
}


#Lista la interfaz
CBTinterface instproc list {} {
   $self instvar CBTagent label linktype pref_value des_router address
   $self instvar hello_int
   set lt UNKNOWN
   switch $linktype {
     1 {set lt "PaP/M"}
     2 {set lt "Mac/M"}
     3 {set lt "PaP/U"}
     4 {set lt "Mac/U"}
   }
   puts "IF($self): lbl($label) -link($lt) -DR($des_router) -
Pr.val($pref_value) -Hello($hello_int) -Addr($address)"
}

#Lista la MFCTable
MFC_Table instproc list {} {
   $self instvar entry
   set x [array names entry]
   if { [llength $x] == 0 } {
      puts "No existen entradas en MFCTable"
   } else {
      foreach g $x {
         $entry($g) list
      }
   }
}



#Lista la TransientTable
Transient_Table instproc list {} {
   $self instvar entry
   set x [array names entry]
   if { [llength $x] == 0 } {
      puts "No existen entradas en Transient_Table"
```

```
   } else {
      foreach g $x {
         $entry($g) list
      }
   }
}



MFC_entry instproc list {} {
   $self instvar group core_id parent_label parent_nexthop del_entry
   $self instvar Mfctable CBTagent child_if child_nh
   if {[info exists del_entry]} {
      puts "Group:$group - Core:$core_id - if to parent:$parent_label
- nexthop: $parent_nexthop - delete timer: $del_entry"
   } else {
      puts "Group:$group - Core:$core_id - if to parent:$parent_label
- nexthop: $parent_nexthop - delete timer: ---"
   }
   puts "   Child iifs    : $child_if"
   puts "   Child nexthops: $child_nh"
}




Transient_entry instproc list {} {
   $self instvar group core_id parent_label parent_nexthop del_entry
   $self instvar TMfctable CBTagent child_if child_nh leaf retransmit
   set l N
   set ret N
   if {$leaf == 1} {
      set l Y
      set ret $retransmit
   }
   puts "Group:$group - Core:$core_id - if to parent:$parent_label -
nexthop: $parent_nexthop - leaf: $l"
   puts "delete timer: $del_entry - retransmit timer: $ret"
   puts "   Child iifs    : $child_if"
   puts "   Child nexthops: $child_nh"
}


CBT instproc debug_send {msg} {
   $self instvar enbl Node ns
   if {![info exists enbl]} {return}
   set list [split $msg /]
   set pdu_type [lindex $list 0]
   set list [lreplace $list 0 0]
   set p "@"
   switch $pdu_type {
      0  { if {$enbl(0) == 1} { set p "HELLO" } }
      1  { if {$enbl(1) == 1} { set p "JOIN_REQ" } }
      2  { if {$enbl(2) == 1} { set p "JOIN_ACK" } }
      3  { if {$enbl(3) == 1} { set p "QUIT_NOT" } }
      4  { if {$enbl(4) == 1} { set p "ECHO_REQ"} }
      5  { if {$enbl(5) == 1} { set p "ECHO_RPL" } }
      6  { if {$enbl(6) == 1} { set p "FLSH_TRE" } }
      default { set p "UNKNOUN PDU" }
```

```
   }
   if {$p == "@"} {return}
        puts "Node:[$Node set id_] ==>$p: $list  Tiempo:[$ns now]"
}


CBT instproc debug_recv {msg} {
   $self instvar Node ns enbl
   if {![info exists enbl]} {return}
   set list [split $msg /]
   set pdu_type [lindex $list 0]
   set list [lreplace $list 0 0]
   set p @
   switch $pdu_type {
      0  { if {$enbl(0) == 1} { set p "HELLO" } }
      1  { if {$enbl(1) == 1} { set p "JOIN_REQ" } }
      2  { if {$enbl(2) == 1} { set p "JOIN_ACK" } }
      3  { if {$enbl(3) == 1} { set p "QUIT_NOT" } }
      4  { if {$enbl(4) == 1} { set p "ECHO_REQ"} }
      5  { if {$enbl(5) == 1} { set p "ECHO_RPL" } }
      6  { if {$enbl(6) == 1} { set p "FLSH_TRE" } }
      default { set p "UNKNOUN PDU" }
   }
   if {$p == "@"} {return}
        puts "Node:[$Node set id_] <== $p: $list  Tiempo:[$ns now]"
}



#Inicializa las variables de debugging. Debe ser corrido en el init
del
#agente CBT. Pone todas las variables en cero
CBT instproc init_debug { } {
   $self instvar enbl tmr dtables disc
   #enbl: determinan si se producen mensajes por PDUs recibidas y
enviadas
   set enbl(0) 0
   set enbl(1) 0
   set enbl(2) 0
   set enbl(3) 0
   set enbl(4) 0
   set enbl(5) 0
   set enbl(6) 0
   #tmr: Determina si se produce mensaje por acciones sobre los timers
   set tmr 0
   #dtables: Determina si se producen mensajes por modificaciones en
   #         las tablas (agregado/borrado child o agregado/borrado
entry)
   set dtables 0
   #disc: Determina si se producen mensajes por PDU descartadas (por
   #      ejemplo un ack para el que no hay gupo)
   set disc 0
}

CBT instproc enable_timers {} {
   $self instvar tmr
   set tmr 1
}

CBT instproc disable_timers {} {
   $self instvar tmr
```

```
   set tmr 0
}

CBT instproc tmr_puts {msg} {
   $self instvar tmr
   if {$tmr == 1} {puts "$msg"}
}

CBT instproc enable_tables {} {
   $self instvar dtables
   set dtables 1
}

CBT instproc disable_tables {} {
   $self instvar dtables
   set dtables 0
}

CBT instproc tab_puts {msg} {
   $self instvar dtables
   if {$dtables == 1} {puts "$msg"}
}


CBT instproc enable_discard {} {
   $self instvar disc
   set disc 1
}

CBT instproc disable_discard {} {
   $self instvar disc
   set disc 0
}

CBT instproc disc_puts {msg} {
   $self instvar disc
   if {$disc == 1} {puts "$msg"}
}


CBT instproc enable_ALL {} {
   $self enable_pdus ALL
   $self enable_timers
   $self enable_discard
   $self enable_tables
}


CBT instproc disable_ALL {} {
   $self disable_pdus ALL
   $self disable_timers
   $self disable_discard
   $self disable_tables
}



CBT instproc enable_pdus {pdu} {
   $self instvar enbl
   switch $pdu {
     "HELLO" {set enbl(0) 1}
```

```
     "JOIN_REQ" {set enbl(1) 1}
     "JOIN_ACK" {set enbl(2) 1}
     "QUIT_NOT" {set enbl(3) 1}
     "ECHO_REQ" {set enbl(4) 1}
     "ECHO_RPL" {set enbl(5) 1}
     "FLSH_TRE" {set enbl(6) 1}
     "ALL"      {set cnt 0
                    while {$cnt <= 6} {
                       set enbl($cnt) 1
             incr cnt 1
                    }
              }
     default    {puts "CBT:enable UNKNOUN PDU, optios are:"
           puts " HELLO JOIN_REQ JOIN_ACK QUIT_NOT ECHO_REQ ECHO_RPL
FLSH_TRE ALL"
          }
   }
}


CBT instproc disable_pdus {pdu} {
   $self instvar enbl
   switch $pdu {
     "HELLO" {set enbl(0) 0}
     "JOIN_REQ" {set enbl(1) 0}
     "JOIN_ACK" {set enbl(2) 0}
     "QUIT_NOT" {set enbl(3) 0}
     "ECHO_REQ" {set enbl(4) 0}
     "ECHO_RPL" {set enbl(5) 0}
     "FLSH_TRE" {set enbl(6) 0}
     "ALL"      {set cnt 0
                    while {$cnt < 6} {
                       set enbl($cnt) 0
             incr cnt 1
                    }
              }
     default    { puts "CBT:disable UNKNOUN PDU, opciones:"
           puts "HELLO JOIN_REQ JOIN_ACK QUIT_NOT ECHO_REQ ECHO_RPL
FLSH_TRE ALL"
          }
   }
}
```

A.7 recvr.tcl

```
####################JOIN ACK

######### SEND
#Este proceso se encarga de recibir requerimientos para envio de join
acks
CBT instproc send_join_ack {group interface nexthop} {
   $self instvar MFCTable ns Node node_
   set orig_router [$Node id]
   set type [CBT set JOIN_ACK]
   set ls1 [$MFCTable get_parent $group]
   set target [lindex $ls1 0]
   set dest $nexthop
   set ifce $interface
   if {$dest == "*"} {
      set envio M
```

```
   } else {
      set envio U
   }
   set msg "$type/$group/$target/$envio/$orig_router"
   $self send_msg $msg $ifce $envio $dest
}


############## RECEIVE
#Recibe join_ack, reenvia y actualiza tablas
CBT instproc recv-join_ack { ja_pdu } {
   $self instvar ns Node node_ TransientTable qntimer
   #Separacion de las componentes del join ack
   set group [lindex $ja_pdu 0]
   set target [lindex $ja_pdu 1]
   set type [lindex $ja_pdu 2]
   set nexthop [lindex $ja_pdu 3]
   set interface [[$self set messager] set iif_]
   #Si el ack no coincide con una entrada transitoria, lo descarta
   set ent [$TransientTable get_parent $group]
   set int [lindex $ent 1]
   set nhp [lindex $ent 2]
   #Acepta J-ack si interfaz coincide, y si nexthop coincide
   set acc [$self jac_accept $group $interface $type $nexthop]
   if { $acc == -1} {
     set m "Nodo:[$Node id] descarta join_ack (no grupo) tiempo [$ns
now]"
     $self disc_puts $m
     return
   }
   #Convierte entrada transitoria a entrada on-tree (elimina entrada
   #transitoria)
   $self transient2ontree $group
}


#Convierte una entrada transitoria en entrada on-tree. Es invocado al
#recibir un join-ack.
#   - 1 Crea una entrada (group) en tabla on-tree, si ya existe,
aborta
#       el proceso
#   - 2 Copia las child transient en la nueva entrada on-tree (add
entry,
#       que implica el envio de join ack)
#   - 3 Elimina las entradas transient
CBT instproc transient2ontree {group} {
   $self instvar TransientTable MFCTable
   #Si existe la entrada, retorna
   set ex [$MFCTable get_entry $group]
   if {$ex != -1 } {return}
   set l1 [$TransientTable get_parent $group]
   $MFCTable add_entry $group [lindex $l1 0] [lindex $l1 1] [lindex
$l1 2]
   set ifaces [$TransientTable get_child_if $group]
   set nhops  [$TransientTable get_child_nh $group]
   set cnt 0
   set lng [llength $ifaces]
   while {$cnt < $lng} {
      $MFCTable add_child $group [lindex $ifaces $cnt] [lindex $nhops
$cnt]
```

```
      $TransientTable delete_child $group [lindex $ifaces $cnt]
[lindex $nhops $cnt]
      incr cnt 1
   }
}


#Dado un grupo para el que llega un join ack, y la interface, tipo de
envio y
#nexthop por las que arribo, determina si el agente lo acepta o no.
#Si las interfaces no coinciden, no se acepta
#Si el nodo no es DR, se testea que coincida solo la iif si envio M
#Si el nodo no es DR, se testea que coincida iif y nexthop si envio U
#Si el nodo es DR, solo acepto envios U, y debe coincidir iif nexthop
CBT instproc jac_accept { group interface envio nexthop} {
   $self instvar TransientTable interfaces
   set ent [$TransientTable get_parent $group]
   set int [lindex $ent 1]
   set nhp [lindex $ent 2]
   #Coinciden interfaces
   if { $int != $interface } { return -1}
   #Node es DR
   if { [$interfaces($int) is_dr] == 1} {
      #Nodo DR, envio no U
      if { $envio != "U" } {
         return -1
      } else {
      #Nodo DR, envio U
         #iif y nexthop no coinciden
         if { $int != $interface || $nexthop != $nhp } {
       return -1
         } else {
       return 0
         }
      }
   } else {
   #Nodo no es DR
      if {$envio == "U" } {
         if {$int == $interface && $nexthop == $nhp } {
       return 0
         } else {
       return -1
         }
      } else {
      #Envio M
         return 0
      }
   }
}



############### JOIN REQUEST

######## SEND
#Envia join request; si la entrada del grupo es leaf, setea el timer
para
#reintentos. en este caso devuelve el timer creado
CBT instproc send_jr { group } {
   $self instvar TransientTable ns Node node_ interfaces
   set orig_router [$Node id]
```

```
   #Obtiene core, interface, nexthop para el grupo
   set list1 [$TransientTable get_parent $group]
   set target [lindex $list1 0]
   set pif  [lindex $list1 1]
   set dest [lindex $list1 2]
   set envio [$interfaces($pif) sendmode]
   if {$envio == "M"} {set dest *}
   #Nodo es DR en parent, JR se envia unicast
   #Envio del primer join request
   set type [CBT set JOIN_REQUEST]
   set msg "$type/$group/$target/$orig_router/$envio/$orig_router"
   $self send_msg $msg $pif $envio $dest
   set leaf [$TransientTable get_leaf $group]
   #Si nodo es leaf, setea timer para reenvio
   if {$leaf == 1} {
      set tout ""
      lappend tout $msg
      lappend tout $pif
      lappend tout $envio
      lappend tout $dest
      lappend tout $self
      set ptimeout $tout
      set fin ""
      set pfin $fin
      set id "[$Node id] Resend_Join_Timer($pif,$dest)"
      set timer [new Resend_Join_Timer $id $ns $self $ptimeout $pfin]
      $timer sched [CBT set RTX_INTERVAL] *
      $TransientTable set_retransmit_timer $group $timer
   }
}


########## RECEIVE

CBT instproc recv-join_request { jr_pdu } {
   $self instvar  ns Node MFCTable
   $self instvar TransientTable jr_pending interfaces
   $self instvar qntimer
   #Separacion de las componentes del join request (target=id del
core)
   set group [lindex $jr_pdu 0]
   set target [lindex $jr_pdu 1]
   set orig_router [lindex $jr_pdu 2]
   set type [lindex $jr_pdu 3]
   set sender [lindex $jr_pdu 4]
   #sender es la node id del nodo que envia (o reenvia) la PDU -no el
origen
   set gr [expr $group]
   #Determinar interfaz de arribo, si es no valida descartar
   #Casos especiales de iif:
   # -1 es interfaz no valida
   #La iif -2, indica que recibo el request a traves de un agente
local
   #(puede ser un agente IGMP o un agente para introducir la pdu desde
   #el simulador)
   set iif [[$self set messager] set iif_]
   if {$iif == -1} {
      set m "Nodo:[$Node id] Descarta JREQ (recibido por interfaz
incorrecta, $iif) tiempo [$ns now]"
      $self disc_puts $m
      return
```

```
    }
    #Determinar si se acepta o no el JR. Un JR es rechazado si:
    #vinculo multiacceso, join multicast, nodo NO es DR
    #vinculo multiacceso, join unicast, nodo DR (==> Error, dos DRs?)
    set linktype [$interfaces($iif) get_linktype]
    set dr [$interfaces($iif) is_dr ]
    if { $linktype == 2 && $type == "M" && $dr != 1 } {
       return
    }
    if { $linktype == 2 && $type == "U" && $dr == 1 } {
       puts "Error dos DRs?"
       return
    }
    #Determina nexthop en base al tipo de join ??
    if {$type == "U"} {
       set nhop $sender
    } else {
       set nhop *
    }
    #Nodo es on-tree para el grupo.
    if { [$MFCTable exists_entry $gr] != -1 } {
       set l1 [$MFCTable get_entry $gr]
       #PDU recibido por parent
       set ip [lindex $l1 1]
       set tp [lindex $l1 2]
       #Si recibe por parent, ignora excepto que sea link mcast maccess
       #el join recibido Mcast y el nodo NO DR
       if { $ip == $iif && $tp == $nhop} {
          if { $type == "M" && $linktype == 2 } {
           #si hay jreq scheduled, lo elimina
        if [info exists jr_pending($gr)] {
            $jr_pending($gr) cancel
         }
          } else {
         set m "Nodo:[$Node id] Descarta JREQ (recibido por interfaz
parent, $iif) at [$ns now]"
         $self disc_puts $m
          }
          return
       } else {
       #Si if no parent, se valida la relacion parent/child, si OK,
       #se hace directamente un add_child, que agrega
       #la entrada si no esta y envia (siempre) un JACK
          if {[$self child_is_valid $iif $nhop $ip $tp] == 1} {return}
          $MFCTable add_child $gr $iif $nhop
          return
       }
    }
    #Nodo NO es on tree para el grupo
    #Existe entrada transitoria para el grupo: Debe agregarse al nodo
    #como child. Se testea que no sea invalida la relacion parent-child
    if { [$TransientTable exists_entry $gr] != -1 } {
       set l1 [$TransientTable get_parent $gr]
       if {[$self child_is_valid $iif $nhop [lindex $l1 1] [lindex $l1
2]] == 1} {
          return
       }
       $TransientTable add_child $gr $iif $nhop
    } else {
    #No existe entrada transitoria
       #Obtiene core para el grupo, es target que vino en PDU
```

```
      #Obtiene nexthop para el core (next_router_to_core)
      set lst [$self next_router $target]
      #No hay ruta al core
      if { [lindex $lst 0] == 0 } { return }
      set iface [lindex $lst 2]
      set nxthop [lindex $lst 1]
      #Si link no mcast, el nexthop es el id del nodo.
      #Si link mcast y DR, el nexhop es el id del nodo
      #Si link mcast y no DR, nexthop es *
      set tpl [$interfaces($iface) get_linktype]
      set dr [$interfaces($iface) is_dr]
      if { $tpl <= 2 &&  $dr != 1} {
       set nxthop  *
      }
      #Si la relacion parent/child es no valida, retorna
      if { [$self child_is_valid $iif $nhop $iface $nxthop] == 1 }
{return}
      #Si nodo es leaf, (iif = -2, por ahora) crea asi la entrada
      set leaf 0
      if { $iif == -2 } { set leaf 1}
      #Elimina Quit_Notificatio_Timer, si existe
      if [info exists qntimer($gr:$iface:$nxthop)] {
         $qntimer($gr:$iface:$nxthop) cancel
      }
      #Crea entrada transitoria
      $TransientTable add_entry $gr $target $iface $nxthop $leaf
      #Agrega child
      $TransientTable add_child $gr $iif $nhop
   }
}


######### ECHO REQUEST


######## SEND ADM


##########################################################################
########

#Clase Echorq_adm
#Existe un objeto de este tipo por cada agente CBT
#Debido a que los echo requests son manejados por timers y que estos
pueden
#involucrar a varios grupos, aca se administra esto. Cuando una
entrada en
#MFC pide agregarse a echo request, se crea un timer si no existe, o
se
#agrega el grupo al ya existente. Cuando una entrada se elimina, se
testea
#si este es el ultimo usuario del timer, y si lo es, se borra
Class Echorq_adm

Echorq_adm instproc init {simul node cbtagent} {
   $self instvar ns Node CBTagent
   set CBTagent $cbtagent
   set Node $node
   set ns $simul
}
```

```
Echorq_adm instproc exists {interface nexthop}  {
   $self instvar echorq_timer
   if [info exists echorq_timer($interface:$nexthop)] {
      return 1
   } else {
      return 0
   }
}

Echorq_adm instproc elim {interface  nexthop} {
   $self instvar ns Node echorq_timer
   unset echorq_timer($interface:$nexthop)
}

Echorq_adm instproc add_group {group interface nexthop} {
   $self instvar ns Node echorq_timer echorq_list CBTagent
   if [info exists echorq_timer($interface:$nexthop)] {
      set lis $echorq_list($interface:$nexthop)
      set idx [lsearch -exact $lis $group]
      if {$idx < 0} {
         lappend lis $group
         set echorq_list($interface:$nexthop) $lis
      }
   } else {
      set lis ""
      lappend lis $group
      set echorq_list($interface:$nexthop) $lis
      #Instancia el objeto timer
      set type [CBT set ECHO_REQUEST]
      set origen [$CBTagent get_nodeid]
      if {$nexthop == "*"} {
         set envio M
      } else {
         set envio U
      }
      set msg "$type/$origen/$envio/$origen"
      #Parametros para el proc timeout
      set tout ""
      lappend tout $msg
      lappend tout $interface
      lappend tout $envio
      lappend tout $nexthop
      lappend tout $CBTagent
      set ptimeout $tout
      #Parametros para el proc final
      set fin ""
      lappend fin $interface
      lappend fin $nexthop
      lappend fin $self
      set pfin $fin
      set simul [$CBTagent get_simulator]
      set id "[$Node id] Echo_Request_Timer($interface,$nexthop)"
      set echorq_timer($interface:$nexthop) [new Echo_Request_Timer
$id $simul $CBTagent $ptimeout $pfin]
      $echorq_timer($interface:$nexthop) sched [CBT set ECHO_INTERVAL]
*
   }
}

#La busqueda se hace por grupo, ya que esto identifica univocamente al
#timer involucrado
```

```
Echorq_adm instproc delete_group {group} {
   $self instvar ns Node echorq_timer echorq_list CBTagent
   set nam [array names echorq_list]
   foreach timer $nam {
      set lis $echorq_list($timer)
      set idx [lsearch -exact $lis $group]
      if {$idx >= 0} {
         set lis [lreplace $lis $idx $idx]
         if {[llength $lis] == 0} {
          set table [$CBTagent get_MFCTable]
          set li [$table get_parent $group]
          set interface [lindex $li 1]
          set nexthop [lindex $li 2]
          $echorq_timer($interface:$nexthop) cancel
          unset echorq_list($timer)
         } else {
       set echorq_list($timer) $lis
         }
      }
   }
}


########### RECEIVE
CBT instproc recv-echo_request { echorq_pdu } {
   $self instvar ns Node node_
   $self instvar cache_del_timer join_request_pending
   set iif [[$self set messager] set iif_]
   #Separacion de las componentes del echo request
   set orig_router [lindex $echorq_pdu 0]
   set type_join [lindex $echorq_pdu 1]
   set dest_join [lindex $echorq_pdu 2]
   #Obtiene los grupos que son parents para (iif, hop) en pgrps
   if {$type_join == "M" } {set nh *}
   if {$type_join == "U" } {set nh $dest_join}
   set pgrps [$self get_parents $iif $nh]
   #Envio de echo reply para los grupos
   #hay grupos parents en la interfaz
   if {[llength $pgrps] > 0} {
      set type [CBT set ECHO_REPLY]
      set org [$Node id]
      set dd 0
      if {$type_join == "U"} {set dd $nh}
      #Hay que ver si mando todos los grupos o uno solo en el reply???
      set msg "$type/$org/$pgrps/$type_join/$org"
      $self send_msg $msg $iif $type_join $dd
   }
   #Interfaz de arribo es parent para algun grupo y echo_rq multicast
   #Es posible preguntar directamente por los timers para echo request
   set nam [array names echorq_timers]
   set idx [lsearch -exact $nam $iif:*]
   if {$idx >= 0 && $type_join == "M" } {
      #Rescheduling del timer para envio del echo request
      $send_echorq($iif:*) resched [CBT set ECHO_INTERVAL] *
   }
}


############### ECHO REPLY

##########  SEND
```

```
########### RECEIVE

CBT instproc recv-echo_reply { echoreply_pdu } {
   $self instvar ns Node MFCTable
   $self instvar cache_del_timer join_request_pending delete_entry
   set iif [[$self set messager] set iif_]
   #Separacion de las componentes del echo reply
   #Puede venir una unica direccion o varias
   set long [llength $echoreply_pdu]
   set orig_router [lindex $echoreply_pdu 0]
   set type [lindex $echoreply_pdu [expr $long -2]]
   set sender [lindex $echoreply_pdu [expr $long -1]]
   set listgroups [lreplace $echoreply_pdu [expr $long -2] [expr $long
-1] ]
   set listgroups [lreplace $listgroups 0 0]
   foreach grp $listgroups {
      set x [$MFCTable get_parent $grp]
      if {[lindex $x 0] > -1} {
         set l1 [lindex $x 1]
         set l2 [lindex $x 2]
         #si la interfaz, nexthop es parent para el grupo, refresh
         if {$l1 == $iif && ($l2 == $sender || $l2 == "*") } {
       $MFCTable resched_delete $grp
         }
      }
   }

}



##################### FLUSH-TREE

############## SEND
#Este proceso se encarga de recibir requerimientos para envio de flush
#tree. Por ahora solo envia el flush tree para cada requerimiento por
#separado. En el futuro podra estar controlado por un timer para
acumular
#posibles flush-tree. Esto en realidad arma el flush tree. Habra q
modif
CBT instproc send_flush_tree {group} {
   $self instvar listg MFCTable ns Node node_
   set gr [$MFCTable get_groups]
   foreach g $gr {
      set lch [$MFCTable get_child_if $g]
      set lnh [$MFCTable get_child_nh $g]
      set cnt -1
      foreach ii $lch {
      incr cnt 1
         if {![info exists ifs($ii)]} {
       set l ""
       set ifs($ii) $l
       set nhp($ii) [lindex $lnh $cnt]
         }
         set l $ifs($ii)
         lappend l $g
         set ifs($ii) $l
      }
```

```
   }
   set aa [array names ifs]
   foreach a $aa {
      set l1 $ifs($a)
      set l2 ""
      set grps ""
      lappend grps $group
      foreach g $grps {
         set s [lsearch -exact $l1 $g]
         if {$s > -1} {
       lappend l2 $g
         }
      }
      if { [llength $l1] == [llength $l2] } {
         set l2 ""
         lappend l2 -1
         set ifs($a) $l2
      } else {
         if { [llength $l2] == 0} {
       lappend l2 -2
       set ifs($a) $l2
         } else {
       set ifs($a) $l2
         }
      }
   }
   #Genera y envia los flush tree para cada interfaz (interfaz,
nexthop)
   set type [CBT set FLUSH_TREE]
   set sender [$Node id]
   foreach yy [array names ifs] {
      set lis $ifs($yy)
      set nhop $nhp($yy)
      set tm U
      if {$nhop == "*"} {
         set tm M
      }
      if {[lindex $lis 0] != -2} {
         set msg "$type/$lis/$tm/$sender"
         #Como se pudo generar la interfaz local, no le envia el flush
tree
         if {$yy != -2 } {
        $self send_msg $msg $yy $tm $nhop
         }
      }
   }
}


########### RECEIVE

#El flush tree se envia U o M segun
#la capacidad del link, y todo el que lo recibe por una interfaz
parent lo
#toma como valido
#Esto debe desencadenar el reenganche ..............

CBT instproc recv-flush_tree {ft_pdu} {
   $self instvar ns Node node_ TransientTable MFCTable
   $self instvar cache_del_timer join_request_pending delete_entry
   set iif [[$self set messager] set iif_]
```

```
   #Separacion de las componentes del flush tree
   #Se adopta que viene una unica  direccion, individual o todos los
grp
   #Todos los grupos, se asume grp = -1
   set long [llength $ft_pdu]
   set type [lindex $ft_pdu [expr $long -2]]
   set sender [lindex $ft_pdu [expr $long -1]]
   set listgroups [lreplace $ft_pdu [expr $long -2] [expr $long -1] ]
   #Se determina la totalidad de grupos para los cuales la int. es
parent
   set flushlist [$self if_parent $iif $listgroups]
   #Eliminacion de la informacion de los grupos involucrados
   foreach g $flushlist {
     #Aca hay que borrar tambien el timer (del_entry o invocarlo)
     #Si se invoca el proc de borrado de la entrada directamente,no se
     #cancelara el timer de borrado asociado. Esto puede ocasionar
     #problemas cuando se crea una nueva entrada.
     #En su lugar, se invoca el timer y se lo cancela.
     ###$MFCTable delete_entry $g
     set ent [$MFCTable get_entry $g]
     set tim [$ent set del_entry]
     $tim cancel
   }
   #Reenganche de los grupos locales
   $self instvar joined_groups
   foreach g $flushlist {
      set lx [lsearch -exact  $joined_groups $g]
      if { $lx != -1 } {
         $self add-leaf $g -2
      }
   }
}


CBT instproc reeng {g} {
   $self instvar joined_groups
   set lx [lsearch -exact  $joined_groups $g]
   if { $lx != -1 } {
      $self add-leaf $g -2
   }
}


###################QUIT NOTIFICATION

####### SEND
CBT instproc send_qn { group } {
   $self instvar MFCTable ns Node node_ interfaces
   $self instvar qntimer
   set orig_router [$Node id]
   #Obtiene core, interface, nexthop para el grupo
   set list1 [$MFCTable get_parent $group]
   set pif  [lindex $list1 1]
   set dest [lindex $list1 2]
   set envio [$interfaces($pif) sendmode]
   if {$envio == "M"} { set dest * }
   #Envio del primer quit notification
   set type [CBT set QUIT_NOTIFICATION]
   set msg "$type/$group/$orig_router/$envio/$orig_router"
   $self send_msg $msg $pif $envio $dest
   #setea timer y cuenta de reenvio
```

```
   set tout ""
   lappend tout $msg
   lappend tout $pif
   lappend tout $envio
   lappend tout $dest
   lappend tout $self
   set ptimeout $tout
   set fin ""
   lappend fin $group
   lappend fin $pif
   lappend fin $dest
   lappend fin $self
   set pfin $fin
   set pfin $fin
   set id "[$Node id] Quit_Notification_Timer($group:$pif,$dest)"
   set qntimer($group:$pif:$dest) [new Quit_Not_Timer $id $ns $self
$ptimeout $pfin]
   $qntimer($group:$pif:$dest) sched [CBT set HOLDTIME] [CBT set
MAX_RTX]
}


### RECEIVE

CBT instproc recv-quit_notification { qn_pdu } {
   $self instvar  ns Node MFCTable TransientTable cache_del_timer
   $self instvar jr_pending interfaces
   #Separacion de las componentes del quit notification
   set group [lindex $qn_pdu 0]
   set orig_router [lindex $qn_pdu 1]
   set type [lindex $qn_pdu 2]
   set sender [lindex $qn_pdu 3]
   set gr [expr $group]
   #Determinar interfaz de arribo, si es no valida descartar
   #Casos especiales de iif:
   # -1 es interfaz no valida
   #La iif -2, indica que recibo el request a traves de un agente
local
   #(puede ser un agente IGMP o un agente para introducir la pdu desde
   #el simulador)
   set iif [[$self set messager] set iif_]
   if {$iif == -1} {
      set m "Nodo:[$Node id]  Descarta QUIT-NOT (recibido por interfaz
incorrecta, $iif) tiempo [$ns now]"
      $self disc_puts $m
      return
   }
   if {$type == "U"} {
      set nhop $sender
   } else {
      set nhop *
   }
   #Nodo no es on-tree para el grupo.
   if { [$MFCTable exists_entry $gr] == -1 } {
      set m "Nodo:[$Node id]  Descarta QUIT-NOT (grupo no on tree)
tiempo [$ns now]"
      $self disc_puts $m
      return
   }
   #Nodo es on tree para el grupo
   if { [$MFCTable exists_entry $gr] != -1 } {
```

```
      set l1 [$MFCTable get_parent $gr]
      set ip [lindex $l1 1]
      set tp [lindex $l1 2]
      #Si soy el core del grupo lo ignoro
      if {$ip == -1 && $tp == -1} {
         set m "Nodo:[$Node id]  Descarta QUIT-NOT (core para grupo)
tiempo [$ns now]"
         $self disc_puts $m
         return
      }
      #QN recibido por parent
      if { $ip == $iif && $tp == $nhop} {
         #QN enviado unicast, es descartado
         if { $type != "M" } {
        set m "Nodo:[$Node id]  Descarta QUIT-NOT (unicast recibido por
interfaz parent) tiempo [$ns now]"
        $self disc_puts $m
        return
         } else {
         #QN enviado multicast, otro nodo desea prune, debe enviarse
JR
        #Pone un timer al azar entre 0 y Holdtime para enviar join
request
        #JR sera enviado multicast
        #Puede haber un problema, los nodos mcast cambiaron de rol
        #debido al rearmado del arbol, ahora lo recibo por parent
        #mcast pero no es maccess; debo ignorarlo en este ultimo
        #caso. Es decir, si link es pap mcast (1) descarto
        set lt [$interfaces($iif) get_linktype]
        if {$lt == 1} { return}
        set fin ""
        set pfin $fin
        #Crea params para el proc timeout
        set type [CBT set JOIN_REQUEST]
        #set core [lindex $parent 0]
        set core [lindex $l1 0]
        set orig [$Node id]
        set msg "$type/$group/$core/$orig/M/$orig"
        set tout ""
        lappend tout $msg
        lappend tout $iif
        lappend tout M
        lappend tout $nhop
        lappend tout $self
        set ptimeout $tout
        set id "[$Node id] Send_Join_Timer($iif, $nhop)"
        set jr_pending($gr) [new Send_Join_Timer $id $ns $self
$ptimeout $pfin]
        #Genera numero al azar entre 0 y HOLDTIME
        set rand [expr [ns-random]/double(0x7fffffff)]
        set rand [expr $rand * [CBT set HOLDTIME]]
        $jr_pending($gr) sched $rand 0
        return
         }
      } else {
      #QN recibido por una interfaz no parent
         set ischild [$MFCTable lookup_child $gr $iif $nhop]
         #Interfaz es child
         if {$ischild >= 0} {
        #QN multicast, timer para eliminar entrada child
        if {$type == "M"} {
```

```
           #Si aun no hay timer
           if {![info exists cache_del_timer($gr:$iif)]} {
              set fin ""
              lappend fin $gr
              lappend fin $iif
              lappend fin $self
              set pfin $fin
              #Crea params para el proc timeout
              set tout ""
              lappend tout $MFCTable
              lappend tout $gr
              lappend tout $iif
              lappend tout $nhop
              set ptimeout $tout
              set id "[$Node id] Cache_Del_Timer($gr,$iif,$nhop)"
              set cache_del_timer($gr:$iif) [new Cache_Del_Timer $id
$ns $self $ptimeout $pfin]
              $cache_del_timer($gr:$iif) sched [CBT set
CACHE_DEL_TIMER]  0
              return
           }
        } else {
        #QN no es multicast, se elimina la interfaz
           $MFCTable delete_child $gr $iif $nhop
        }
          }
       }
   }
}
```

A.8 interface.tcl

```
#Clase CBTinterface
#Cada objeto interface representa una interfaz del nodo

Class CBTinterface

#Inicializa las variables, pone valores por defecto para los
configurables
#Los valores para cada interfaz, son tomados de CBTv2 MIB
CBTinterface instproc init {agent lbl} {
   $self instvar CBTagent label linktype pref_value des_router address
   $self instvar hello_int
   set CBTagent $agent
   #Label: cbtInterfaceIfIndex
   set label $lbl
   #Address: cbtInterfaceAddress
   set address [[$CBTagent set Node] id]
   #Designated router
   set des_router -1
   #Preference value for hello protocol: cbtInterfaceHelloPreference
   set pref_value [CBT set HELLO_PREFERENCE]
   #Intervalo para el hello
   set hello_int [CBT set HELLO_INTERVAL]
   #Status: si esta habilitado o no el CBT en esta interface. A
agregar
   #Tipo de link, al inicializar, -1
   set linktype -1
}

#Setea tipo de link
```

```
CBTinterface instproc set_linktype {type} {
   $self instvar linktype
   set linktype $type
}


#Devuelve tipo de link
CBTinterface instproc get_linktype {} {
   $self instvar linktype
   return $linktype
}


#Setea preference value
CBTinterface instproc set_preference_value {value} {
   $self instvar pref_value
   set pref_value $value
}


#Devuelve preference value
CBTinterface instproc get_preference_value {} {
   $self instvar pref_value
   return $pref_value
}


#Setea valor de designated router en la interfaz
CBTinterface instproc set_des_router {value} {
   $self instvar des_router linktype
   if { ($linktype == 1 || $linktype == 3) && $value != -1 } {
      puts "No es posible setear designated router en link no maccess"
      return
   }
   set des_router $value
}


#Devuelve valor de designated router en la interfaz
CBTinterface instproc get_des_router {} {
   $self instvar des_router
   return $des_router
}

#Devuelve 1 si el nodo es el dr en la interfaz
CBTinterface instproc is_dr { } {
   $self instvar des_router CBTagent
   set n [[$CBTagent set Node] id ]
   if {$n == $des_router} {
      return 1
   } else {
      return 0
   }
}


#Devuelve el modo en que se debe enviar una PDU: U si vinculo no
soporta
#mcast o si vinculo es macces y es DR
CBTinterface instproc sendmode { } {
   set dr [$self is_dr]
```

```
   set lt [$self get_linktype]
   set mode U
   if {$lt == 1 } {set mode M}
   if {$dr == 0 && $lt ==2 } {set mode M}
   return $mode
}
```

A.9 mfc.tcl

```
#########################################################################
#######
#Clase MFC_entry : Entradas en la multicast forwarding cache
#Estos procedimientos contienen acciones especificas del protocolo

Class MFC_entry

# 1 Crea una nueva entrada en la tabla. Recibe los siguientes
parametros:
#   - Table: la tabla a la cual pertenece la entrada
#   - CBTag: el agente CBT correspondiente
#   - core: id del nodo core para el grupo. Si es core este nodo el id
es el de el
#   - pnt_interface: Label de la interfaz local por donde se accede al
core
#   - pnt_nexthop: Proximo nodo camino al core (* si es multicast)
# 2 - Si no existe un timer para generar ECHO_REQUEST para el par
interfaz,
#     nexthop, lo crea.
# 3 Crea e inicializa un timer para el borrado de la entrada en caso
de no
#   recibirse en echo reply en el tiempo EXPIRE_GROUP_TIME
MFC_entry instproc init {Table CBTag grp core pnt_interface
pnt_nexthop} {
   $self instvar group core_id parent_label parent_nexthop del_entry
   $self instvar Mfctable CBTagent child_if child_nh
   set group $grp
   set child_if ""
   set child_nh ""
   set CBTagent $CBTag
   set Mfctable $Table
   set core_id $core
        set parent_label $pnt_interface
        set parent_nexthop $pnt_nexthop
   set n [$CBTagent set Node]
   set s [$CBTagent set ns]
   set m "Node:[$n id] Agregando entrada MFC, group:$group core:$core
if:$parent_label nxp:$parent_nexthop at [$s now]"
   $CBTagent tab_puts $m
   #Si la entrada no es core . . .
   if {$parent_nexthop != -1 && $parent_label != -1} {
      #Creacion e inicializacion del timer para borrado de la entrada
      #Parametros para el proc timeout
      set tout ""
      lappend tout $Mfctable
      lappend tout $group
      set ptimeout $tout
      #Parametros para el proc final
           set fin ""
      lappend fin $Mfctable
      lappend fin $group
```

```
      set pfin $fin
      set simul [$CBTagent get_simulator]
      set Node [$CBTagent set Node]
      set id "[$Node id] Delete_Entry_Timer($group)"
      set del_entry [new Delete_Entry_Timer $id $simul $CBTagent
$ptimeout $pfin]
      $del_entry sched [CBT set GROUP_EXPIRE_TIME] 0
      #Crea o se agrega al envio de echo requests por la interfaz
parent
      set adm [$CBTagent get_Echorq]
      $adm add_group $group $parent_label $parent_nexthop
   }
        return $self
}

# Rescheduling del timer asociaado a la entrada, ejecutado al recibir
# ECHO_REPLY
MFC_entry instproc resched_delete {} {
   $self instvar del_entry
   $del_entry resched [CBT set GROUP_EXPIRE_TIME] 0
}


#Procedimientos a llevar a cabo cuando se elimina una entrada en la
tabla
# 1 Enviar quit notfication al parent
# 2 Enviar flush tree a cada child
# 3 Eliminarse de la lista de grupos involucrados en el echo request
MFC_entry instproc delete  {} {
   $self instvar group core_id parent_label parent_nexthop
delete_entry
   $self instvar Mfctable CBTagent del_entry
   #Envio de flush tree a cada child. Esto debe ser un notificar
pedido
   #de envio
   set n [$CBTagent set Node]
   set s [$CBTagent set ns]
   set m "Node:[$n id] Borrando entrada MFC, group:$group
core:$core_id if:$parent_label nxp:$parent_nexthop at [$s now]"
   $CBTagent tab_puts $m
   $CBTagent send_flush_tree $group
   #Envio de quit notification al parent
   $CBTagent send_qn $group
   #Eliminacion del timer para echorequest
   set adm [$CBTagent get_Echorq]
   $adm delete_group $group
   #Cada child, se elimina usando delete_child
   set ifz [$self get_child_if]
   set nhp [$self get_child_nh]
   set cnt 0
   foreach iz $ifz {
      incr cnt 1
      set nh [lindex $nhp $cnt]
      $self delete_child $iz $nh
   }
   return 0
}


#Devuelve una lista con info del parent: core, interface, nexthop
MFC_entry instproc get_parent {} {
```

```
   $self instvar core_id parent_label parent_nexthop
   set list1 ""
   lappend list1 $core_id
   lappend list1 $parent_label
   lappend list1 $parent_nexthop
   return $list1
}


#Devuelve una lista con info de child: interface
MFC_entry instproc get_child_if {} {
   $self instvar child_if
   return $child_if
}


#Devuelve una lista con info de child: nexthop
MFC_entry instproc get_child_nh {} {
   $self instvar child_nh
   return $child_nh
}


#Agrega un child en la entrada. Devuelve el indice de la entrada
agregada,
#si ya estaba, devuelve el indice sin agregarla
#La agrege o no, debe generar un join ack.
#La generacion del ack depende de si la interfaz de origen del JR es
local
#(-2) o es causada por un agente CBT.
#En ambos casos, el nexthop es -1, y no se genera un join ack
MFC_entry instproc add_child {ch_interface ch_nexthop} {
   $self instvar child_if child_nh CBTagent group core_id
   set resul [$self lookup_child $ch_interface $ch_nexthop]
   #Encontro la entrada, envia ack y devuelve indice
   if { $resul != -1 } {
      if {$ch_nexthop != -1 && $ch_nexthop != -2 } {
         $CBTagent send_join_ack $group $ch_interface $ch_nexthop
      }
      return $resul
   } else {
   #No encontro la entrada, la agrega, envia ack y devuelve indice
      set n [$CBTagent set Node]
      set s [$CBTagent set ns]
      set m "Node:[$n id] MFC:Agregando child, group:$group
core:$core_id if_ch:$ch_interface nxp_ch:$ch_nexthop at [$s now]"
      $CBTagent tab_puts $m
      lappend child_if $ch_interface
      lappend child_nh $ch_nexthop
      if {$ch_nexthop != -1  && $ch_interface != -2 } {
         $CBTagent send_join_ack $group $ch_interface $ch_nexthop
      }
      set resul [$self lookup_child $ch_interface $ch_nexthop]
      #Al agregar una entrada se debe hacer algo para que la proxima
vez
      #que se reciba un paquete src, gr, iif, se produzca un cache
miss
      #y esto provoque el agregado del replicator correspondiente
      #esto se produce automaticamente ya que el nuevo S lo producira
      #si manda algo; Lo que debe hacerse aparte es agregar la nueva
      #interfaz en los replicators ya existentes
```

```
      #Para todos los replicators del group, se hace el insert oif
      #Para esto se invoca a actualizar-replicators de CBT
      $CBTagent actualizar-replicators $group $ch_interface

   return $resul
   }
}


#Elimina un child en la entrada. Devuelve -1 si no existe
MFC_entry instproc delete_child {ch_interface ch_nexthop} {
   $self instvar child_if child_nh group CBTagent core_id
   set resul [$self lookup_child $ch_interface $ch_nexthop]
   if {$resul == -1} {
      return $resul
   } else {
      set n [$CBTagent set Node]
      set s [$CBTagent set ns]
      set m "Node:[$n id] MFC:Eliminando child, group:$group
core:$core_id if_ch:$ch_interface nxp_ch:$ch_nexthop at [$s now]"
      $CBTagent tab_puts $m
      #Eliminacion de los elementos de reenvio (inhabilita targets en
      #los replicators)
      $CBTagent borrar_child $group $ch_interface
      set child_if [lreplace $child_if $resul $resul]
      set child_nh [lreplace $child_nh $resul $resul]
      return 0
   }
}


#Dada una interfaz y un nexthop, busca la entrada child
correspondiente
#Devuelve el indice si lo encuentra y si no -1
MFC_entry instproc lookup_child {ch_interface ch_nexthop} {
   $self instvar child_if child_nh
   set cnt [llength $child_if]
   if {$cnt == 0} {
      return -1
   } else {
      set idx 0
      while {$idx < $cnt} {
         set inf [lindex $child_if $idx]
         set nhp [lindex $child_nh $idx]
         if {$inf == $ch_interface && $nhp == $ch_nexthop} {
            return $idx
         }
         incr idx 1
      }
      return -1
   }
}

#Devuelve la cantidad de entradas child
MFC_entry instproc num_child {} {
   $self instvar child_if
   return [llength $child_if]
}
```

```
##########################################################################
########
#Clase Transient_entry : Entradas en la transient table
#Estos procedimientos contienen acciones especificas del protocolo

Class Transient_entry

# 1 Crea una nueva entrada en la tabla. Recibe los siguientes
parametros:
#   -Table: objeto Table al que pertenece
#   -CBTag: Objeto agente CBT
#   -core: id del nodo core para el grupo. Si es core este nodo el id
es el de el
#   -pnt_interface: Label de la interfaz local por donde se accede al
core
#   -pnt_nexthop: Proximo nodo camino al core (* si es multicast)
#   -leaf: Indicacion si el nodo es leaf para la entrada (1) o no (0)
# 2 - Inicializa los datos child (vacio)
# 3 - Crea un timer para eliminar la entrada si no se recibe ack
Transient_entry instproc init {Table CBTag grp core pnt_interface
pnt_nexthop lf} {
   $self instvar group core_id parent_label parent_nexthop del_entry
   $self instvar TMfctable CBTagent child_if child_nh leaf retransmit
   set leaf $lf
   set retransmit 0
   set group $grp
   set child_if ""
   set child_nh ""
   set CBTagent $CBTag
   set TMfctable $Table
   set core_id $core
        set parent_label $pnt_interface
        set parent_nexthop $pnt_nexthop
   set n [$CBTagent set Node]
   set s [$CBTagent set ns]
   set y Y
   if {$leaf == 0} {set y N}
   set m "Node:[$n id] Agregando entrada Transitoria, group:$group
core:$core leaf:$y if:$parent_label nxp:$parent_nexthop at [$s now]"
   $CBTagent tab_puts $m
   #Creacion e inicializacion del timer para borrado de la entrada
   #Parametros para el proc timeout
   set tout ""
   set ptimeout $tout
   #Parametros para el proc final
        set fin ""
   lappend fin $TMfctable
   lappend fin $group
   #agrega code =1 , terminacion anormal
   lappend fin 1
   set pfin $fin
   set simul [$CBTagent get_simulator]
   set Node [$CBTagent set Node]
   set id "[$Node id] Delete_T_Entry_Timer($group)"
   set del_entry [new Delete_T_Entry_Timer $id $simul $CBTagent
$ptimeout $pfin]
   if {$leaf == 0} {
      $del_entry sched [CBT set TRANSIENT_TIMEOUT] 0
   } else {
      $del_entry sched [CBT set JOIN_TIMEOUT] 0
   }
```

```
   #Envia primer join request al parent (si leaf seteara timer y
   #devuelve objeto timer)
   #Esto lo debe hacer la tabla, ya q el jreq pide a la tabla el entry
   #y hasta q este proc no retorne, no existe
   ##$CBTagent send_jr $group
        return $self
}


#Procedimientos a llevar a cabo cuando se elimina una entrada en la
tabla
#La eliminacion puede ocurrir en forma normal (se recibio un ack) o en
#forma anormal (expiro el tiempo). Se recibe esta indicacion en code
(0 si
#terminacion normal, 1 si anormal)
#Terminacion anormal:
#   1-Cancelar el timer de reenvio si leaf
#Terminacion normal
#   1-Cancelar timer de reenvio si leaf
#   * El timer de expiracion no se puede cancelar, porque invoca a
este
#     procedimiento. Lo que ocurrira es que cuando cancele por tiempo
#     no tendra entrada para eliminar
Transient_entry instproc delete {} {
   $self instvar group core_id parent_label parent_nexthop
delete_entry
   $self instvar TMfctable CBTagent retransmit leaf
   set n [$CBTagent set Node]
   set s [$CBTagent set ns]
   set m "Node:[$n id] Eliminando entrada Transitoria, group:$group
core:$core_id if:$parent_label nxp:$parent_nexthop at [$s now]"
   $CBTagent tab_puts $m
   #Eliminacion del timer de reenvio
   if {$leaf == 1} {
      $retransmit cancel
   }
   return 0
}

#Devuelve informacion acerca de si el nodo es leaf en la entrada
transitoria
Transient_entry instproc get_leaf { } {
   $self instvar leaf
   return $leaf
}

Transient_entry instproc set_retransmit_timer {timer} {
   $self instvar retransmit
   set retransmit $timer
}

#Devuelve una lista con info del parent: core, interface, nexthop
Transient_entry instproc get_parent {} {
   $self instvar core_id parent_label parent_nexthop
   set lis ""
   lappend lis $core_id
   lappend lis $parent_label
   lappend lis $parent_nexthop
   return $lis
}
```

```
#Devuelve una lista con info de child: interface
Transient_entry instproc get_child_if {} {
   $self instvar child_if
   return $child_if
}


#Devuelve una lista con info de child: nexthop
Transient_entry instproc get_child_nh {} {
   $self instvar child_nh
   return $child_nh
}


#Agrega un child en la entrada. Devuelve el indice de la entrada
agregada,
#si ya estaba, devuelve el indice sin agregarla
Transient_entry instproc add_child {ch_interface ch_nexthop} {
   $self instvar child_if child_nh CBTagent group core_id
   set resul [$self lookup_child $ch_interface $ch_nexthop]
   if {$resul != -1} {
      return $resul
   } else {
      set n [$CBTagent set Node]
      set s [$CBTagent set ns]
      set m "Node:[$n id] Agregando child (transitorio), group:$group
core:$core_id if_ch:$ch_interface nxp_ch:$ch_nexthop at [$s now]"
      $CBTagent tab_puts $m
      lappend child_if $ch_interface
      lappend child_nh $ch_nexthop
      return 0
   }
}


#Elimina un child en la entrada. Devuelve -1 si no existe
Transient_entry instproc delete_child {ch_interface ch_nexthop} {
   $self instvar child_if child_nh core_id group CBTagent
   set resul [$self lookup_child $ch_interface $ch_nexthop]
   if {$resul == -1} {
      return $resul
   } else {
      set n [$CBTagent set Node]
      set s [$CBTagent set ns]
      set m "Node:[$n id] Eliminando child (transitorio), group:$group
core:$core_id if_ch:$ch_interface nxp_ch:$ch_nexthop at [$s now]"
      $CBTagent tab_puts $m
      set child_if [lreplace $child_if $resul $resul]
      set child_nh [lreplace $child_nh $resul $resul]
      return 0
   }
}


#Dada una interfaz y un nexthop, busca la entrada child
correspondiente
#Devuelve el indice si lo encuentra y si no -1
Transient_entry instproc lookup_child {ch_interface ch_nexthop} {
   $self instvar child_if child_nh
   set cnt [llength $child_if]
```

```
   if {$cnt == 0} {
      return -1
   } else {
      set idx 0
      while {$idx < $cnt} {
         set inf [lindex $child_if $idx]
         set nhp [lindex $child_nh $idx]
         if {$inf == $ch_interface && $nhp == $ch_nexthop} {
            return $idx
         }
         incr idx 1
      }
      return -1
   }
}

#Devuelve la cantidad de entradas child
Transient_entry instproc num_child {} {
   $self instvar child_if
   return [llength $child_if]
}




##########################################################################
######

#Clase MFC_Table: multicast forwarding cache, contiene MFC-entries

Class MFC_Table

#Inicializa un objeto MFC_Table: pone su tabla en 0
MFC_Table instproc init {cbt_ag} {
   $self instvar entry cbt_agent
        set cbt_agent $cbt_ag
   return $self
}

#Devuelve el objeto Entry asociado al grupo
#-1 si no existe entrada para el grupo
MFC_Table instproc get_entry {group} {
   $self instvar entry
   if [info exists entry($group)] {
           return $entry($group)
        } else {
           return -1
        }
}


#Borra la  entrada asociada al grupo. Invoca al procedimiento delete
de la
#entrada
#Si el grupo no existe o si la entrada da error, devuelve -1
MFC_Table instproc delete_entry {group} {
   $self instvar entry
   if [info exists entry($group)] {
           set ent $entry($group)
           set result [$ent delete]
           if {$result != -1} {unset entry($group)}
```

```
            return $result
   } else {
            return -1
        }
}


#Crea una nueva entrada en la tabla. Devuelve -1 si la entrada ya
existe o si
#el procedimiento de creacion del objeto entry da error
MFC_Table instproc add_entry {group core interface nexthop} {
   $self instvar entry cbt_agent
   if [info exists entry($group)] {
            return -1
        } else {
      set ent [new MFC_entry $self $cbt_agent $group $core $interface
$nexthop]
            if {$ent == -1} {
               return -1
            } else {
               set entry($group) $ent
               return 0
            }
   }
}



#Crea una entrada core en la tabla. Falta: Devuelve -1 si la entrada
ya existe
# o si ya hay otro core, etc ....
MFC_Table instproc add_core_entry {group core} {
   $self instvar entry cbt_agent
   if [info exists entry($group)] {
            return -1
        } else {
      set ent [new MFC_entry $self $cbt_agent $group $core -1 -1]
            if {$ent == -1} {
               return -1
            } else {
               set entry($group) $ent
               return 0
            }
   }
}




#Dado un grupo, devuelve informacion del parent: core, interface,
nexthop
MFC_Table instproc get_parent {group} {
   $self instvar entry
   if {![info exists entry($group)]} {
            return -1
        } else {
      return [$entry($group) get_parent]
   }
}
```

```
MFC_Table instproc resched_delete {group} {
   $self instvar entry
   if {![info exists entry($group)]} {
           return -1
        } else {
      return [$entry($group) resched_delete]
   }
}



#Dado un grupo, una interface y un nexthop, devuelve -1 si no existe
child
MFC_Table instproc lookup_child {group interface nexthop} {
   $self instvar entry
   if {![info exists entry($group)]} {
           return -1
        } else {
      return [$entry($group) lookup_child $interface $nexthop]
   }
}



#Dado un grupo, devuelve informacion de child: interface
MFC_Table instproc get_child_if {group} {
   $self instvar entry
   if {![info exists entry($group)]} {
           return -1
        } else {
      return [$entry($group) get_child_if]
   }
}


#Dado un grupo, devuelve informacion de child: nexthop
MFC_Table instproc get_child_nh {group} {
   $self instvar entry
   if {![info exists entry($group)]} {
           return -1
        } else {
      return [$entry($group) get_child_nh]
   }
}


#Dado un grupo, crea una entrada child en el entry correspondiente
#Esto implica el envio del join-ack por la child interface (a cargo
del add
#que realiza el entry)
MFC_Table instproc add_child {group interface nexthop} {
   $self instvar entry cbt_agent
   if {![info exists entry($group)]} {
           return -1
        } else {
           set resul [$entry($group) add_child $interface $nexthop]
      return $resul
   }
}
```

```
#Dado un grupo, elimina una entrada child en el entry correspondiente
#Si es la ultima child para el grupo debe eliminar la entrada
MFC_Table instproc delete_child {group interface nexthop} {
   $self instvar entry cbt_agent
   if {![info exists entry($group)]} {
           return -1
        } else {
           set resul [$entry($group) delete_child $interface $nexthop]
      if { [$entry($group) num_child] == 0 } {
         $self delete_entry $group
      }
      return $resul
   }
}


#Dado un grupo una interface de arribo y un nexthop (previous)
devuelve
#-1 si la entrada no existe, 0 si existe
MFC_Table instproc exists_entry {group } {
#interface nexthop
   $self instvar entry cbt_agent
   if {![info exists entry($group)]} {
           return -1
   }
   return 0
}


#Devuelve los grupos on tree en el nodo
MFC_Table instproc get_groups {} {
   $self instvar entry
   set xl [array names entry]
   return $xl
}


##############################################################################
#######

#Clase Transient_Table: transient table, contiene Transient_entries

Class Transient_Table

#Inicializa un objeto Transient_Table: pone su tabla en 0
Transient_Table instproc init {cbt_ag} {
   $self instvar entry cbt_agent
        set cbt_agent $cbt_ag
   return $self
}

#Devuelve el objeto Entry asociado al grupo
#-1 si no existe entrada para el grupo
Transient_Table instproc get_entry {group} {
   $self instvar entry
   if [info exists entry($group)] {
           return $entry($group)
        } else {
           return -1
        }
}
```

```
#Borra la  entrada asociada al grupo. Invoca al procedimiento delete
de la
#entrada
#Recibe un codigo de terminacion normal(0) o anormal (1)
#Si el grupo no existe o si la entrada da error, devuelve -1
#Este caso puede ocurrir,  ya que el timer que cancela la entrada
(timeout)
#al no haber recibido un ack, puede querer borrar la entrada que ya ha
#sido borrada normalmente
Transient_Table instproc delete_entry {group} {
   $self instvar entry
   if [info exists entry($group)] {
           set ent $entry($group)
           set result [$ent delete ]
           if {$result != -1} {unset entry($group)}
           return $result
   } else {
           return -1
        }
}


#Crea una nueva entrada en la tabla. Devuelve -1 si la entrada ya
existe o si
#el procedimiento de creacion del objeto entry da error
Transient_Table instproc add_entry {group core interface nexthop leaf}
{
   $self instvar entry cbt_agent
   if [info exists entry($group)] {
           return -1
       } else {
      set ent [new Transient_entry $self $cbt_agent $group $core
$interface $nexthop $leaf]
           if {$ent == -1} {
              return -1
           } else {
              set entry($group) $ent
         $cbt_agent send_jr $group
              return 0
           }
   }
}

#Dado un grupo, devuelve informacion acerca de si la entrada es leaf o
no
Transient_Table instproc get_leaf {group} {
   $self instvar entry
   if {![info exists entry($group)]} {
           return -1
        } else {
      return [$entry($group) get_leaf]
   }
}


#Dado un grupo, devuelve informacion del parent: core, interface,
nexthop
Transient_Table instproc get_parent {group} {
   $self instvar entry
```

```
   if {![info exists entry($group)]} {
           return -1
        } else {
      return [$entry($group) get_parent]
   }
}


#Dado un grupo, devuelve informacion de child: interface
Transient_Table instproc get_child_if {group} {
   $self instvar entry
   if {![info exists entry($group)]} {
           return -1
        } else {
      return [$entry($group) get_child_if]
   }
}


#Dado un grupo, devuelve informacion de child: nexthop
Transient_Table instproc get_child_nh {group} {
   $self instvar entry
   if {![info exists entry($group)]} {
           return -1
        } else {
      return [$entry($group) get_child_nh]
   }
}

#Dado un grupo una interface de arribo y un nexthop (previous)
devuelve
#-1 si la entrada no existe, 0 si existe
Transient_Table instproc exists_entry {group} {
   $self instvar entry cbt_agent
   if {![info exists entry($group)]} {
           return -1
   }
   return 0
}


#Dado un grupo, crea una entrada child en el entry correspondiente
Transient_Table instproc add_child {group interface nexthop} {
   $self instvar entry cbt_agent
   if {![info exists entry($group)]} {
           return -1
        } else {
           set resul [$entry($group) add_child $interface $nexthop]
      return $resul
   }
}


#Dado un grupo, elimina una entrada child en el entry correspondiente
#Si es la ultima child para el grupo debe eliminar la entrada
#Hay que ver si es necesario enviar code de terminacion normal o
anormal
Transient_Table instproc delete_child {group interface nexthop} {
   $self instvar entry cbt_agent
   if {![info exists entry($group)]} {
           return -1
```

```
        } else {
           set resul [$entry($group) delete_child $interface $nexthop]
       if { [$entry($group) num_child] == 0 } {
          $self delete_entry $group
       }
       return $resul
   }
}

Transient_Table instproc set_retransmit_timer {group timer} {
   $self instvar entry
   if {![info exists entry($group)]} {
           return -1
        } else {
           $entry($group) set_retransmit_timer $timer
   }
}
```