Redes de computadoras 2015
Obligatorio 2

Grupo 48
Alonzo Fulchi, Emiliano CI: 4.460.443-0
Damiano Izzi, Rodrigo CI: 4.260.487-0
Flores Saavedra, Martin CI: 4.656.307-0

Tutor: Eduardo Grampin

1 - Descripción del problema

Se desea implementar una aplicación (cliente y servidor) de mensajería en línea para grupos en una LAN. La misma utilizará el protocolo UDP y buscará ofrecer un servicio de mensajería confiable sobre éste, permitiendo el intercambio de mensajes entre todos los clientes conectados simultáneamente. El nivel de confiabilidad deseado es equivalente al RDT 3.0 visto en el curso sin considerar mensajes corruptos.

Se podrán enviar dos tipos de mensajes, mensajes privados, los cuales serán transmitidos por unicast al servidor y este último se encargará de re-enviarlo al cliente destinatario siendo transmitidos también por unicast. Por otro lado están los mensajes grupales, los cuales serán transmitidos por unicast desde un cliente al servidor y este último se encargará de re-enviarlo a todos los clientes que estén conectados. Para esto se utiliza la dirección de multicast 225.5.4.48.
Observación: solo el servidor es quien transmite mensajes a través de la dirección de multicast, todos lo mensajes enviados por los clientes se transmiten por unicast y son enviados al servidor.

2 - Protocolo de comunicación

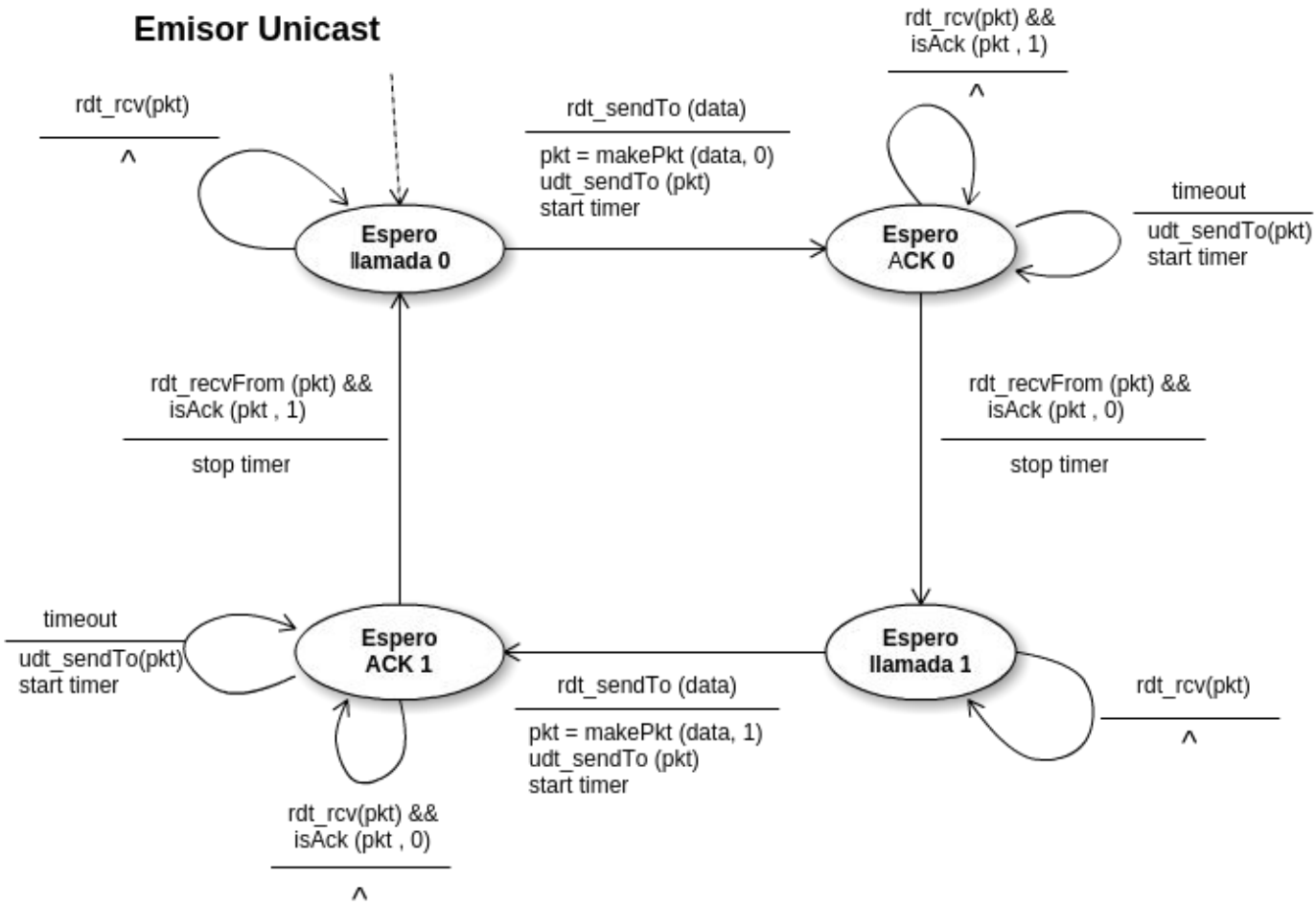
El protocolo de comunicaciones entre el cliente y el servidor deberá ser confiable, resolviendo pérdidas de paquetes y duplicados.
Dado que UDP no garantiza que los mensajes sean entregados, es responsabilidad de la aplicación controlar que se logre la confiabilidad deseada. Para esto implementamos un protocolo de comunicación que controla la pérdida de paquetes y la recepción de paquetes duplicados. Como se menciona en la sección anterior, el protocolo es equivalente al RDT3.0 visto en el curso con la diferencia que no se debe controlar los paquetes corruptos.
Entendemos que el protocolo es “stop & wait”, es decir que hasta que no se recibe la confirmación de que un paquete fue entregado correctamente no se permite enviar más paquetes. En caso que luego de enviar un paquete, pase cierto tiempo sin recibir la confirmación, se debe intentar enviar nuevamente el paquete. En caso de recibir un paquete duplicado se debe notificar al emisor que el paquete fue efectivamente recibido correctamente pero no se deben entregar datos a la capa superior.

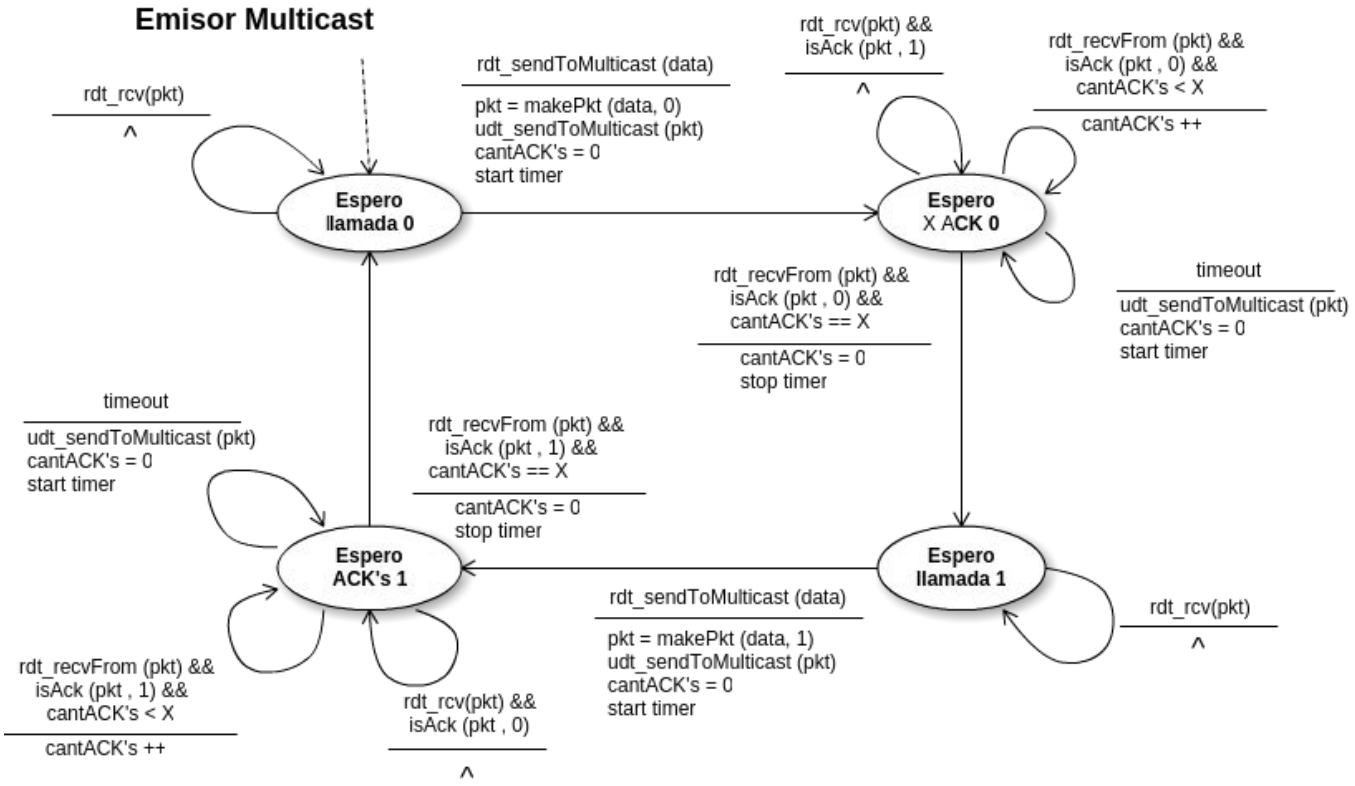
A continuación mostramos las máquinas de estados y luego comentamos más en detalle la implementación del protocolo.

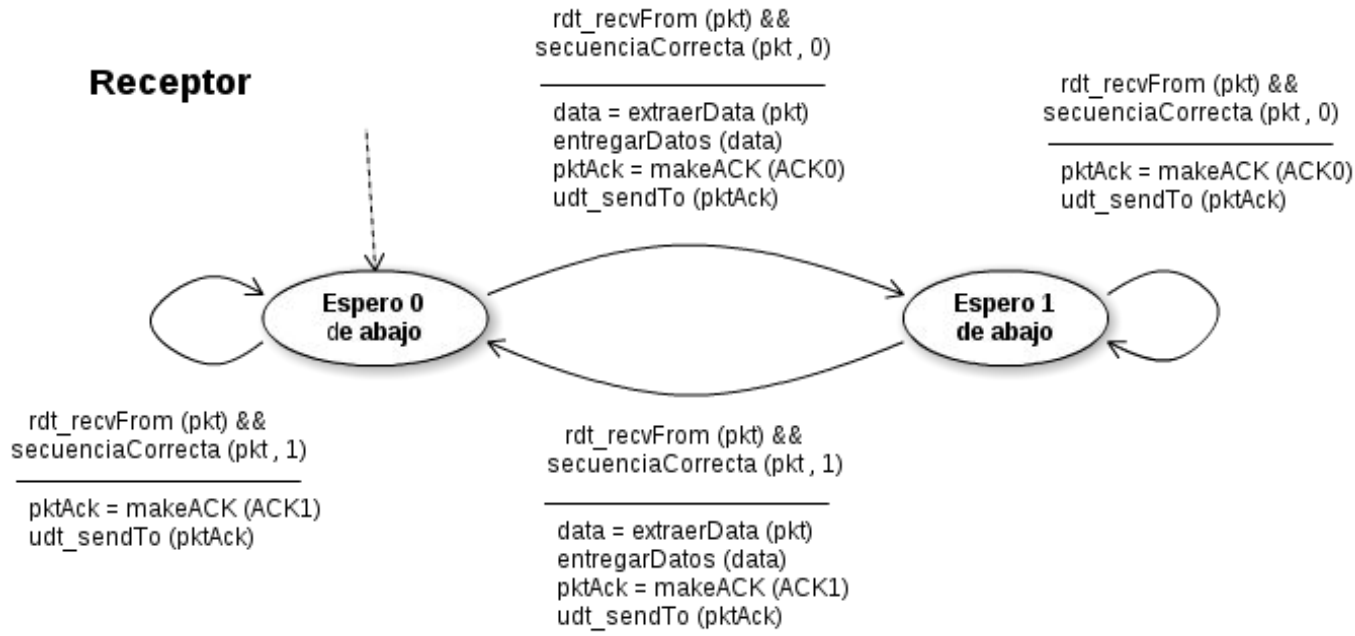
3 - Máquinas de estados del protocolo de comunicación

El protocolo de comunicación, al igual que el rdt3.0, es un protocolo “stop & wait” donde se va alternando el número de secuencia de los mensajes entre 0 y 1.
Para resolver el problema planteado en este obligatorio, decidimos diferenciar cuando se envía un mensaje de multicast de cuando se manda un mensaje privado debido a que los numeros de secuencia utilizados para cada tipo de mensaje son independientes, es decir, se manejara un número de secuencia para el flujo de mensajes unicast, y otro número de secuencia para el flujo de mensajes de multicast. Por esta razón, la maquina de estados del emisor la separamos en dos máquinas independientes.
Algo que creemos necesario detallar es qué rol interpretará el servidor y los clientes, lo cual realizaremos más adelante en la descripción de la solución.

A continuación se muestran las máquinas de estados.




Emisor Multicast
rdt_rcv(pkt)
^
rdt_sendToMulticast (data)
pkt = makePkt (data, 0)
udt_sendToMulticast (pkt)
cantACK's = 0
start timer
rdt_rcv(pkt) &&
isAck (pkt , 1)
^
rdt_recvFrom (pkt) &&
isAck (pkt , 0) &&
cantACK's < X
cantACK's ++
Espero
llamada 0
Espero
X ACK 0
rdt_recvFrom (pkt) &&
isAck (pkt , 0) &&
cantACK's == X
cantACK's = 0
stop timer
timeout
udt_sendToMulticast (pkt)
cantACK's = 0
start timer
timeout
udt_sendToMulticast (pkt)
cantACK's = 0
start timer
rdt_recvFrom (pkt) &&
isAck (pkt , 1) &&
cantACK's == X
cantACK's = 0
stop timer
Espero
ACK's 1
Espero
llamada 1
rdt_sendToMulticast (data)
pkt = makePkt (data, 1)
udt_sendToMulticast (pkt)
cantACK's = 0
start timer
rdt_rcv(pkt)
^
rdt_recvFrom (pkt) &&
isAck (pkt , 1) &&
cantACK's < X
cantACK's ++
rdt_rcv(pkt) &&
isAck (pkt , 0)
^


Receptor
rdt_recvFrom (pkt) &&
secuenciaCorrecta (pkt , 0)
data = extraerData (pkt)
entregarDatos (data)
pktAck = makeACK (ACK0)
udt_sendTo (pktAck)
rdt_recvFrom (pkt) &&
secuenciaCorrecta (pkt , 0)
pktAck = makeACK (ACK0)
udt_sendTo (pktAck)
Espero 0
de abajo
Espero 1
de abajo
rdt_recvFrom (pkt) &&
secuenciaCorrecta (pkt , 1)
pktAck = makeACK (ACK1)
udt_sendTo (pktAck)
rdt_recvFrom (pkt) &&
secuenciaCorrecta (pkt , 1)
data = extraerData (pkt)
entregarDatos (data)
pktAck = makeACK (ACK1)
udt_sendTo (pktAck)

Las primitivas utilizadas se describen a continuación:

- **`rdt_sendTo(data)`**: utilizada para recibir datos de la capa de aplicación que tienen que ser enviados por unicast.
- **`rdt_sendToMulticast(data)`**: utilizada para recibir datos de la capa de aplicación que tienen que ser enviados por multicast.
- **`rdt_recvFrom(pkt)`**: utilizada para recibir un paquete entregado por el canal no confiable.
  `udt_sendTo(pkt)`: utilizado para enviar un paquete sobre el canal no confiable por unicast.
- **`udt_sendToMulticast(pkt)`**: utilizado para enviar un paquete sobre el canal no confiable por multicast.
- **`makePkt(data,X)`**: utilizada para crear un paquete con número de secuencia X que contiene los datos de capa de aplicación data (data también puede ser ACK0 o ACK1 cuando se quiere enviar un paquete de respuesta ACK).
- **`makeACK(ACKn)`**: utilizado para crear un paquete de respuesta ACK con número de secuencia n.
- **`isAck(pkt,X)`**: se utiliza para chequear que el paquete pkt sea un ACK con número de secuencia X.
- **`secuenciaCorrecta(pkt, X)`**: verifica si el paquete pkt tiene el número de secuencia X.
- **`extraerData(pkt)`**: utilizada para extraer los datos del paquete pkt.
- **`entregarDatos(data)`**: utilizada para entregar datos a la capa de aplicación

(Obs. En los diagramas aparece una primitiva con el nombre rdt_rcv(pkt), esta primitiva es la misa que rdt_recvFrom(pkt) pero por un error quedó con distinto nombre)
Hay que destacar que las primitivas utilizadas en las máquinas de estados son de forma ilustrativa y pretenden más que nada mostrar el flujo de mensajes. Si bien hay un mapeo con la implementación, no necesariamente existe la implementación de todas estas primitivas.

4 - Descripción de la solución

En esta sección se mencionan los aspectos relacionados a las decisiones que fueron tomadas a la hora de diseñar la solución.

Formato de mensajes

Dado que parte de lo que se está implementando es un protocolo de comunicación, consideramos conveniente definir una estructura que cumple el rol de header de nuestro protocolo, para esto definimos la estructura Mensaje rdt la cual se encuentra en el archivo mensaje.h, a continuación se presentan los campos que contiene un mensaje:

- **esMulticast:** un booleano para indicar si es un mensaje que fue transmitido por multicast.
- **esAck:** un booleano para indicar si es un mensaje de ACK o no.
- **seq:** un entero que indica el número de secuencia del mensaje.
- **mensaje:** una cadena de caracteres donde se envían los datos de capa de aplicación.

Los primero tres campos serían el header de nuestro protocolo.

Estructuras del protocolo

Para el mantenimiento de los números de secuencia tanto del proceso emisor como receptor se creó el mapa **TablaSecuencias**. Este contiene como clave la ip y como valor un entero indicando el número de secuencia. Dado que esta estructura es única por cada proceso que se corra (es compartida por todos los hilos) decidimos utilizar dos variables, una para recibir y otra para enviar mensajes.

```
typedef map<string, int> TablaSecuencias;
TablaSecuencias* emisor = new TablaSecuencias;
TablaSecuencias* receptor = new TablaSecuencias;
```

Cada tabla indica en qué número de secuencia se se encuentra cada emisor/receptor.

Al enviar un mensaje multicast se optó por pasar por parámetro a la función **rdt_sendToMulticast** las IPs y puertos de los destinatarios. Esto se hace para que el proceso emisor sepa de donde tienen que venir los diferentes ACK. Para esto se utiliza el mapa **TablaClienteId**. Este mapa contiene como clave una dirección ip y como valor un booleano (que es utilizado para saber si ya se recibió el ACK o no). El nombre de esta variable quizás no resulte mnemotécnico porque en el protocolo rdt no manejamos comunicación entre clientes, sino que identificamos procesos de un host (ip y puerto). La capa de aplicación cuando desea realizar un **rdt_send_multicast** debe pasar esta estructura con la ip puerto de los cliente conectados al momento de enviar el mensaje (son quienes deberían recibir el mensaje y por lo tanto responder con un ACK), todos con el valor inicializado en false.

Estructuras del servidor

Dentro del servidor se manejan dos estructuras, los clientes y los mensajes.
Se define la estructura **Cliente** con los siguientes campos:

- **nick:** Cadena de caracteres para el apodo
- **puerto:** Número de puerto donde recibe el cliente
- **ip:** Cadena de caracteres de la IP donde se conectó el cliente
- **ult_actividad:** Timestamp de la última vez que el cliente mandó un mensaje.

Los clientes se guardan en un map utilizando como clave su IP

A su vez se utiliza una cola (queue) para guardar los mensajes a enviar.
La estructura de cada **Mensaje** es la siguiente

- **destino:** IP de destino del mensaje.
- **dest_puerto:** puerto de destino del mensaje
- **origen:** IP que originó el mensaje
- **orig_puerto:** puerto de origen del mensaje
- **msg:** contenido del mensaje
- **multicast:** Booleano que indica si el mensaje es multicast.

La cola de mensajes es compartida por dos hilos, un hilo que procesa los comandos enviados por el cliente y “pushea” el **Mensaje** generado de parsear el comando. Mientras que otro hilo consumidor de esta cola, realizará pop sobre la misma para luego enviar comandos al cliente.

5 - Implementación del protocolo de comunicación

En esta sección pretendemos entrar en detalle de cómo implementamos el protocolo de comunicación, y de alguna manera explicar cómo se mapean las primitivas utilizadas en las máquinas de estados con la implementación del mismo.

Primero que nada hay que destacar que nuestra implementación del protocolo de comunicación está desacoplada de lo que es la implementación del cliente y el servidor, lo cual creemos es una solución elegante ya que intenta respetar la separación de la capa de aplicación de la capa de transporte.

El protocolo de comunicación implementado intenta cubrir algunas de las carencias que tiene UDP si lo comparamos con TCP. Con esto nos referimos a que UDP no garantiza que los mensajes sean entregados, que estos lleguen en orden, etc. Para lograr cubrir estas carencias, implementamos las siguientes primitivas primitivas.

char* rdt_recibe(int soc, char*& ipEmisor, int& puertoEmisor)
Esta primitiva, al igual que en el rdt3.0, la utilizamos para obtener un paquete de la capa subyacente. Como se ve en la maquina de estados del receptor, cuando se recibe un paquete, se chequea si el número de secuencia es el correcto. En caso de serlo, se obtienen los datos del paquete y son enviados a la capa de aplicación, se crea el ACK correspondiente y es enviado al emisor del paquete recibido. En caso que el número de secuencia no sea el correcto, se crea el ACK correspondiente y es enviado al emisor del paquete recibido pero no se entregan datos a la capa de aplicación ya que al recibir un paquete con número de secuencia incorrecto significa que es un paquete duplicado, esto puede suceder cuando se pierde el ACK o cuando se termina el timer del lado del emisor.

Al utilizar esta primitiva, el proceso que lo haga, queda bloqueado esperando que llegue un paquete de la capa subyacente. Además escribe las variables ip y puerto con los valores del proceso que envió el mensaje.

-- pseudocódigo de rdt_recibe(...) --

```
loop:
      recvfrom(socket, mensaje, infoConexion)
      if mensaje.esMulticast
            seqEsperado=obtenerSecuenciaMulticast()
      else
            seqEsperado=obtenerSecuencia(tablaReceptor, ip)
      respuesta.esAck = true
      respuesta.seq = mensaje.seq
      sendTo(socket, respuesta, infoConexion)
      if esValido(mensaje.seq, seqEsperado)
            [ipEmisor,puertoEmisor] = getInfo(infoConexion)
            updateSeq(tablaRe, ceptormensaje.esMulticast)
            return mensaje.mensaje, ipEmisor, puertoEmisor
end loop.
```

int rdt_sendto(int soc, char* mensaje, char* ipDestino, int puertoDestino) Esta primitiva es utilizada por la capa de aplicación para enviar un mensaje para que sea transmitido por unicast. En esta función se envía a través del socket soc un mensaje al proceso que se está ejecutando en el host con dirección IP “ipDestino” y escucha en el puerto “puertoDestino”. Como el protocolo es “stop & wait” el proceso que la utilice queda bloqueado hasta que recibe de parte del destinatario el ack correspondiente. En caso que pase un determinado tiempo sin recibir una respuesta, se asume que el mensaje no llego al receptor y se vuelve a enviar, volviendo a esperar por una respuesta del destinatario. Si bien el protocolo debería garantizar el envío del paquete, a modo de evitar que el proceso quede en un loop infinito, el paquete se intenta re-enviar una cantidad finita de veces y en caso de no poder conseguir enviarlo se informa al emisor que no pudo ser posible enviar su mensaje.

-- pseudocódigo de rdt_sendto(...) --

```
while(cantIntentos < CANT_INTENTOS_REENVIO):
      seq = obtenerSecuencuencia(tablaEmisor, ip)
      mensaje = armarMensaje(strMensaje, ip, puerto, seq)
      conexionInfo = armarInfo(ip, puerto)
      valido = sendTo(socket, mensaje, conexionInfo)
      if valido
            setTimerSocket(socket)
            recvfrom(socket, respuesta, conexionInfo)
            if not timeOut
            AND conexionInfo.ip == ip
            AND respuesta.esAck
            AND respuesta.seq == seqEsperado
                  updateSecuencia(tablaEmisor, ip)
                  return ok
end while.
return not ok
```

int rdt_send_multicast(int soc, char* mensajeToSend, TablaClienteId* tablaClientes)
Esta primitiva, al igual que rdt_sendTo(...), es utilizada por la capa de aplicación para enviar un mensaje pero con la diferencia que este mensaje debe ser transmitido por multicast a todos los clientes que estén conectados. Esta función envía a través del socket “soc” un mensaje a la dirección de multicast. Luego de enviar el mensaje espera por la respuesta de todos los clientes que recibieron el mensaje. De la misma forma, el proceso que utilice esta primitiva queda bloqueado, esperando hasta que reciba todos los ACKs correspondientes de los clientes indicados en el mapa **TablaClienteId.** Los clientes son identifican en nuestra implementación del protocolo de capa de transporte como el string “ip:puerto”. Luego de pasado un determinado tiempo, si no se recibieron todas las respuestas de confirmación de recepción, se asume que el mensaje no le llegó a alguno de los receptores por lo que es enviado nuevamente. Por las misma razón que se menciona en el rdt_sendTo(...) el mensaje se intenta re-enviar una cantidad finita de veces.
(Obs. Es tarea del proceso receptor identificar cuando un mensaje es un duplicado y tomar las medidas necesarias para afrontar este problema. Esto se menciona en la primitiva rdt_recibe(...))

```
-- pseudocódigo de rdt_send_multicast(...) --

while(cantIntentos < CANT_INTENTOS_REENVIO):
      seqEsperado = obtenerSecuencuenciaMulticast()
      mensaje = armarMensajeMult(strMensaje, ip, puerto, seq)
      multicastInfo = armarInfoMulticast(ipMult, puertoMult)
      valido = sendTo(socket, mensaje, multicastInfo)
      if valido
            setTimerSocket(socket)
            while(cantClientes>ackRecibidos)
                  recvfrom(socket, respuesta, conInfo)
                  ipExiste = tablaIp.existe(conInfo.ip)
                  if not timeOut
                  AND ipExiste
                  AND respuesta.esAck
                  AND respuesta.esMulticast
                  AND tablaIp[conInfo.ip]==false
                  AND respuesta.seq == seqEsperado
                        ackRecibidos++
                        tablaIp[conInfo.ip] = true
            end while.
            if cantClientes == ackRecibidos
                  return ok
end while.
return not ok
```

int CrearSocket(int puerto, bool multicast) A la hora de crear el socket decidimos que además de indicar el puerto debíamos indicar si este va a recibir los mensajes que son enviados a la dirección de multicast. La dirección de multicast fue parametrizada en una constante por lo tanto un cambio en la misma implica volver a compilar el código.

6 - Implementación del Cliente

Como habíamos dicho antes, en esta sección hablariamos de los roles que tomaría el cliente. De la forma que fue implementado, este toma los dos roles, tanto el del emisor como el de receptor. El rol de emisor lo toma a la hora de enviar los comandos al servidor. Esto es porque cuando el cliente envía un comando al servidor (MESSAGE <msg> <CR>), se queda esperando por la respuesta del servidor indicando que ese mensaje le llego bien (ACK).

Por otro lado, el rol de receptor lo toma a la hora de recibir los mensajes de otros clientes (que en realidad son enviados por el servidor). Esto es porque cuando al cliente le llega un mensaje, ya sea privado o del grupo de multicast, el cliente responde al servidor con un ACK para informarle que el mensaje le llego correctamente.

Para implementar esta solución, decidimos utilizar dos hilos de ejecución, de forma que un hilo tome el rol de emisor, utilizando un socket específico para esto (`socketComandos`) y otro hilo que funcione como rol de receptor utilizando otro socket, distinto del mencionado antes (`socketMensajes`) que además se debe inicializar con la bandera multicast en true, ya que a través de este es por donde se reciben los mensajes transmitidos por multicast.

El hilo con rol emisor, que se encarga de producir mensajes, es quien envía comandos al servidor. Para ello se queda escuchando comandos desde la entrada estándar. Luego de parseada la entrada (entre las que están: LOGOUT, GET_CONNECTED, MESSAGE, PRIVATE_MESSAGE) se envía el mensaje al servidor a través del `socketComandos` al hilo del servidor.
El hilo con rol receptor es quién se encarga de recibir los comandos enviados por el servidor. Estos pueden venir por unicast o multicast. El hilo se queda bloqueado escuchando con rdt_recibe a la espera de un nuevo mensaje.
Pseudocódigo de ambos hilos:

-- hilo emisor --	**-- hilo receptor --**
loop: comando = stdin mensaje = parsearComando() rdt_send(mensaje) **end loop.**	**loop:** mensaje = rdt_recibe resultado = parsearMensaje() print(resultado) **end loop.**

Como se comentó antes, la razón por la cual se separó en dos hilos de ejecución fue para evitar la situación en la que al estar enviando un comando al servidor el cliente quede bloqueado sin poder recibir mensajes de otros clientes.

7 - Implementación del Servidor

De la misma forma que el cliente toma tanto el rol del emisor como el de receptor, con el servidor sucede lo mismo. Este toma el rol de emisor a la hora de reenviar los mensajes, privados o de multicast, ya que luego de enviarlos se queda esperando por una respuesta de los clientes, donde se informa que recibieron bien los mensajes. Por otro lado, toma el rol de receptor cuando espera por los comandos enviados por los clientes, a los cuales responde con un mensaje de ACK indicando que fueron recibidos correctamente. Para cumplir con el requerimiento de quitar clientes inactivos se utilizó un hilo MonitorClientes que se encarga de

quitar los clientes que no hayan realizado actividad en un intervalo de tiempo establecido. Finalmente un último hilo se encarga de manejar la consola de comandos de la entrada estándar.

Como los hilos trabajan de manera concurrente y acceden a las mismas estructuras, como lo son, la lista de clientes y la cola de mensajes, fue necesario sincronizar mediante las primitivas *pthread_mutex_lock*, *pthread_mutex_unlock*, *pthread_cond_signal* y *pthread_cond_wait* el acceso a los recursos del servidor. *pthread_mutex_lock* y *pthread_mutex_unlock* se utilizan para controlar el acceso a las estructuras, mientras que *pthread_cond_signal* y *pthread_cond_wait* se utilizan para sincronizar la ejecución de los hilos.

```
                    -- hilo emisor del servidor --
loop:
        if colaMensajes.Vacia()
                wait(condicion)
        mensaje = colaMensajes.pop()
        if mensaje.isMulticast()
                rdt_sendTo(SocketEmisor, mensaje)
        else
                rdt_sendMulticast(SocketEmisor, mensaje, listaClientes)
end loop.
```

```
                    -- hilo receptor del servidor --
loop
        [strMsj, ip, puerto] = rdt_recibe(SocketReceptor)
        comando =  parse(strMsj, ip, puerto)
        mensaje = procesarComando(comando)
        colaMensajes.push(mensaje)
        signal(condicion)
end loop.
```

```
                    -- hilo monitor del servidor --
loop
        sleep(MONITOR_TIME segundos)
        for each c in listaClientes
                if tiempoActual - c.ultimaActividad > TTL_CLIENTES
                        encolar(goodbye,)
                        borrarCliente(c)
end loop.
```

```
                    -- hilo consola del servidor --
loop:
      comando = stdin
      resultado = parsearComando(comando)
      print(resultado)
end loop.
```

8 - Mejoras y problemas encontrados

El sistema como está planteado no soporta el pasaje de comando de error del servidor al cliente. Esto sería útil cuando el cliente quiere mandar un mensaje privado a otro cliente que no se encuentra registrado, es decir un nick que no existe en el servidor, indicarle a través del comando de error que el cliente con el nick indicado no se encuentra conectado. La implementación actual descarta dicho mensaje en el servidor sin realizar aviso al cliente.

Como ya se mencionó el sistema al hacer un envío realiza un cantidad máxima de intentos. Si se trata de un mensaje multicast superar esta cantidad puede descoordinar los números de secuencia. Esto se da porque algunos clientes pueden haber recibido el mensaje y otros no. Por elección el emisor no mueve el número de secuencia cuando se supera la cantidad máxima de intentos. Esto hará que el siguiente mensaje enviado a los clientes que hayan recibido el primer mensaje lo descarten. Luego de esto los números de secuencia quedan coordinados.

9 - Anexo

Junto con el código fuente se entregarán los siguientes scripts utilizados durante el desarrollo:
Makefile: Archivo make con reglas para compilar tanto el cliente como el servidor, ejecutar ambas aplicaciones y limpiar los binarios generados.
servidor: script bash para ejecutar el servidor.
cliente: script bash para ejecutar el cliente
clienteLocal: script bash para ejecutar el cliente que le pasa como ip del servidor, la dirección ip del host donde se ejecuta el script.
El Makefile crea 2 archivos ocultos .cliente y .servidor. El servidor se ejecuta simplemente ejecutando ./.servidor. El cliente se debe ejecutar de la siguiente manera ./.cliente <apodo> <ip_servidor> <puerto_cliente>