Introducción a las Redes de Computador{ae}s y Comunicación de Datos

Solución Examen – 17 de diciembre de 2010

(ref: eirc1012.odt)

Instrucciones

- **Indique su nombre completo y número de cédula en cada hoja.**
- **Numere todas las hojas e indique la cantidad total de hojas que entrega en la primera.**
- **Escriba las hojas de un solo lado y utilice una caligrafía claramente legible.**
- **Comience cada pregunta teórica y cada ejercicio en una hoja nueva.**
- **Sólo se contestarán dudas de letra. No se aceptarán dudas de ningún tipo los últimos 30 minutos del examen.**
- **El examen es individual y sin material. Apague su celular mientras este en el salón del examen.**
- **Es obligatorio responder correctamente al menos 15 puntos en las preguntas teóricas.**
- **El puntaje mínimo de aprobación es de 60 puntos.**
- **Para todos los ejercicios, si es necesario, puede suponer que dispone de los tipos de datos básicos (p.ej. lista, cola, archivo, string, etc.) y sus funciones asociadas (ej: tail(lista), crear(archivo), concatenar(string, string).**
- **Duración: 3 horas. Culminadas las 3 horas el alumno no podrá modificar las hojas a entregar de ninguna forma.**

Preguntas Teóricas

Pregunta 1 (8 puntos)

El DNS es una base de datos distribuida y jerárquica, compuesta por varios tipos de servidores.

a) Cuando un servidor de nombres aprende un mapeo, lo *cachea.* ¿Cuál es el campo del registro de recurso (RR) que decide si se puede cachear o no? ¿Cuál es el servidor que define ese tiempo? ¿El valor que toma ese campo, a qué refiere y en que unidades está expresado?

b) Suponga que desde un PC de una empresa, se hace una consulta tipo **A** al servidor de nombres local por el nombre `pruebaexmen.edu`, ¿cuales son los pasos siguientes que debe realizar el servidor antes de dar la respuesta al usuario?

Respuesta 1

a) El campo es el TTL, que refiere al tiempo restante por el cual ese RR es válido y por lo tanto se puede mantener en caché. El TTL está definido por el servidor autoritativo de ese RR, y es la cantidad de segundos para los que ese registro es válido. El servidor que lo mantiene en cache, deberá ir decrementando ese valor, hasta que cuando llega a cero, deja de ser válido y se quita del caché.

b)
PC => Consulta SERVIDOR LOCAL por TIPO A, nombre pruebaexmen.edu

SERVIDOR LOCAL => Consulta ROOT SERVER (no lo tiene en caché) por el servidor TLD del .edu

ROOT SERVER responde con la IP del servidor TLD del edu

SERVIDOR LOCAL => Consulta TLD del .edu por el RR tipo A, nombre pruebaexmen.edu

Servidor TDL responde con la dirección IP del nombre pruebaexmen.edu al SERVIDOR LOCAL

SERVIDOR LOCAL cachea la respuesta y responde al PC con la dirección IP del nombre pruebaexmen.edu

Pregunta 2 (8 puntos)

a) Describa el mecanismo de multiplexación/demultiplexación para sockets UDP, discutiendo los siguientes puntos:
   1. Cuando un *host* recibe un segmento UDP, ¿de qué manera decide a qué proceso derivárselo?
   2. Cuando el proceso recibe el segmento, ¿cómo sabe a dónde enviar la respuesta?
   3. ¿Con qué información queda completamente identificado un *socket* UDP en la red?

b) Indique la diferencia entre el mecanismo de la parte a) y el utilizado para *sockets* TCP.
¿Con qué información queda completamente identificado un *socket* TCP en la red?

Respuesta 2

a) Cuando un host recibe un segmento UDP, utiliza el **puerto destino** indicado en el segmento para determinar a qué proceso debe enviárselo.
El proceso que lee el segmento utiliza los campos **IP origen** y **puerto origen** para saber cuál es el socket UDP donde debe enviar la respuesta.
Un socket UDP queda identificado completamente utilizando la **IP** del host y el número de **puerto**.

b) El caso de TCP es un poco diferente:
- Un proceso estará esperando conexiones en un puerto.
- Cuando se establece una conexión, el sistema operativo creará un socket que enlaza **<IP origen, Puerto origen, IP destino, Puerto destino>**, y se lo devolverá al proceso para que lo atienda.
- Cuando llega un segmento TCP al host, se utilizará los cuatro valores para determinar a cuál de los sockets hay que derivar el segmento (en realidad alcanzaría con utilizar **IP origen**, **Puerto origen**, **Puerto destino**).

Un socket TCP queda identificado completamente utilizando los cuatro campos: **IP de origen**, **Puerto de origen**, **IP de destino**, **Puerto de destino**.

Pregunta 3 (8 puntos)

a) Considere una red de varios nodos interconectados por un *hub*, ¿cuál es el dominio de colisión y cuál es el de *broadcast*? Justifique.
b) Si en la red anterior cambiamos el *hub* por un *switch*. ¿Se modifican los dominios de colisión y *broadcast*? ¿De qué manera?
c) Describa el mecanismo de *self-learning* que utilizan los *switches*.

Respuesta 3

a) El dominio de colisión y el dominio de broadcast son iguales y abarcan a todos los nodos de la red. Esto se debe a que el hub no implementa ningún protocolo de acceso al medio y simplemente reenvía todas las tramas por todas las interfaces.

b) En el caso de un switch existe un dominio de colisión por cada interfaz del switch, por lo que en este caso cada enlace con los nodos tendrá su propio dominio de colisión. El switch impediría que ocurran colisiones. El dominio de broadcast sigue abarcando a todos los nodos de la red.

c) El mecanismo de self-learning funciona de la siguiente manera:
- Inicialmente la tabla del switch está vacía.
- Cuando llega una trama del nodo A con destino B a través de la interfaz X:
  - Anotamos en la tabla que se puede llegar a A a través de la interfaz X.
  - Si hay un registro (B, Y) en la tabla, y X distinto de Y, reenviamos la trama por Y.
  - Si no existe el registro, reenviamos la trama por todas las interfaces excepto X.

Pregunta 4 (8 puntos)

En el protocolo de capa de transporte TCP:
a) ¿Por qué se necesitan números de secuencia?
b) ¿Por qué son necesarios timers?

Respuesta 4

a) TCP es un protocolo de transporte que asegura que no se perderán datos durante la transmisión. Para ello, el emisor debe recibir confirmación de que el receptor recibió correctamente cada uno de los paquetes enviados. Para aprovechar mejor el enlace, TCP es también un protocolo *pipelined* por lo que el emisor envía varios paquetes antes de recibir confirmación por alguno de ellos. Por esta razon, el emisor debe poder distinguir a que paquete o paquetes corresponde una determinada confirmación recibida por lo que numera los paquetes enviados

secuencialmente. Las confirmaciones enviadas por el receptor utilizan esos números para distinguir que paquete o conjunto de paquetes están confirmando.

b) TCP es un protocolo de transporte que asegura que no se perderán datos durante la transmisión. Para ello, el emisor debe recibir confirmación de que el receptor recibió correctamente cada uno de los paquetes enviados. De no recibir esta confirmación la, el emisor no sabe si esto se debe a que se perdió el paquete enviado, la confirmación o simplemente alguno de los dos esta demorado. La única solución es retransmitir el paquete no confirmado. Para ésta retransmisión debe esperar un tiempo prudencial luego de enviado cada paquete por lo que necesita un *timer* que le indique cuando hacerlo.

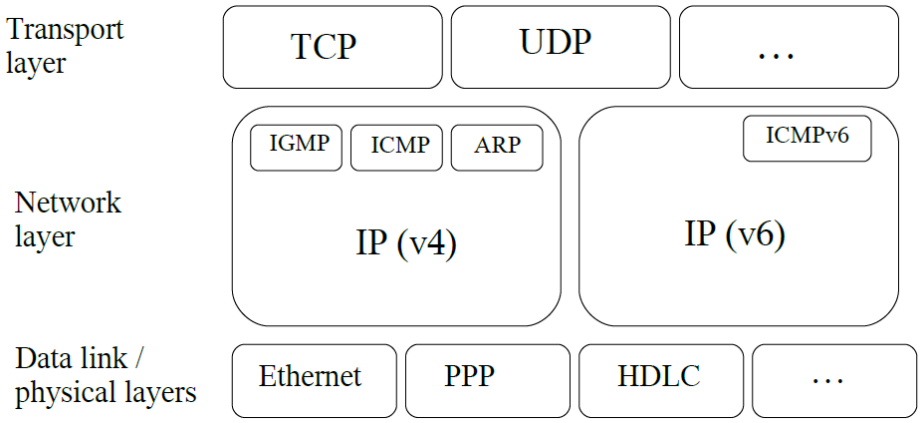
Pregunta 5 (8 puntos)

El protocolo IPv6 se presenta como un reemplazo del protocolo Ipv4.

a) Presente los stacks de referencia de ambos protocolos. Ejemplifique al menos dos protocolos con los que ambos coexisten de capa superior y al menos dos protocolos de capa inferior.
b) Algunos protocolos definidos dentro de ICMP debieron ser re-definidos en ICMPv6. Mencione el reemplazo del protocolo ARP y describa no menos de 4 cambios significativos en su implementación en Ipv6.

Respuesta 5

a)



b)

- ND es parte de ICMPv6, en su totalidad, se transporta sobre IPv6
- Se utiliza Multicast para Neighbor Solicitation en vez de Broadcast
- Los mapeos son de direcciones *link-layer* a IPv6 en vez de *link-layer* a IPv4
- ND combina la resolución de direcciones con autoconfiguración y detección de alcance.
- Los nodos pueden usar mensajes NS para propagar cambios en su configuración

1.Problemas Prácticos

Problema 1 (30 puntos)

Se desea implementar un sistema para brindar un servicio de conexión inalámbrica móvil a Internet, donde el usuario puede elegir el proveedor que le ofrezca un precio más bajo.

Este sistema se implementará como una API en el equipo del usuario con la función `connectCheaper()` y un servicio que se ejecuta en los puntos de acceso (AP).

Para conectarse a Internet, la API del usuario deberá informarse del precio de cada punto de acceso a su alcance y luego conectarse al más barato. Si el punto de acceso más barato rechaza la conexión, se intentará con el siguiente, y así hasta lograr conectarse o no tener mas puntos de acceso disponibles.

En tanto, el servicio del punto de acceso deberá comunicar su precio actual a los usuarios.
Los puntos de acceso pueden aceptar un máximo de `maxUsers` conectados.

En los equipos de los usuarios se cuenta con las siguientes primitivas:

- `list()` retorna la lista de APs dentro del alcance como una lista de SSID.
- `associate(ssid)`, se asocia al AP con el ssid dado. Los APs cuentan con un servicio DHCP por el cual en el momento de asociarse también se le asigna al usuario una dirección IP y direcciones de DNS.
- `connected()`, informa al sistema que el usuario se encuentra conectado a un AP y se pueden enviar datos.

En los APs se cuenta con las siguientes primitivas:

- `countUsers()`, retorna la cantidad de usuarios conectados a este punto de acceso.
- `getPrice()`, devuelve el precio actual del punto de acceso.

Se pide:

Diseñar el protocolo de comunicación necesario para el pedido de precios y de conexión.

Para esto deberá:

1. Especificar todos los mensajes necesarios para la comunicación.
2. Dibujar las maquinas de estado que especifiquen los procesos usuario y punto de acceso.
3. Implementar en un lenguaje de alto nivel las maquinas de estado definidas en el punto anterior. La comunicación entre usuarios y puntos de acceso se deberá realizar mediante *sockets* UDP. El usuario recibe las ofertas de precio en el puerto 12345.

Solución Problema 1

1. Los mensajes necesarios para la comunicación son:

   - OFFER:"precio", el AP envia su oferta de precio.

   - CONNECT, solicitud de conexión de un usuario a determinado AP.

   - COMMIT, el AP acepta el pedido de conexión de un usuario.

   - REFUSE, el AP rechaza el pedido de conexión de un usuario.

2. La idea de la maquina de estados es:
   El AP en un loop infinito envia mensajes broadcast ofreciendo su precio. Además en otro proceso recibe pedidos de conexión y dependiendo de la cantidad de usuarios conectados acepta o rechaza los pedidos.
   La función de los usuarios se asocia de a uno a todos los ap y escucha el mensaje broadcast de cada ap. A partir de este mensaje obtiene la dir ip de cada ap. Luego obtenidos los precios, se intenta conectar al mas barato.

Diagrama AP:

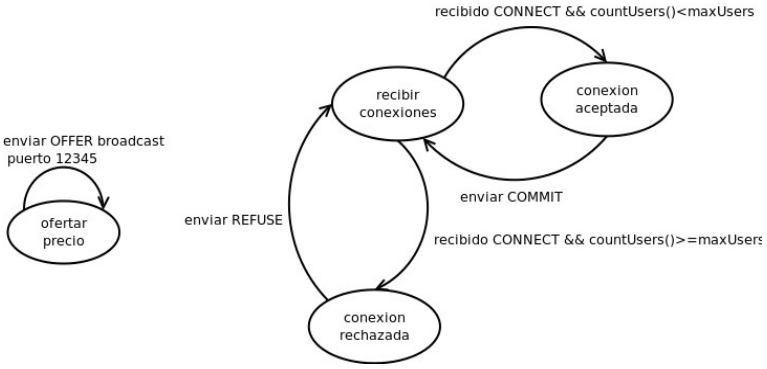
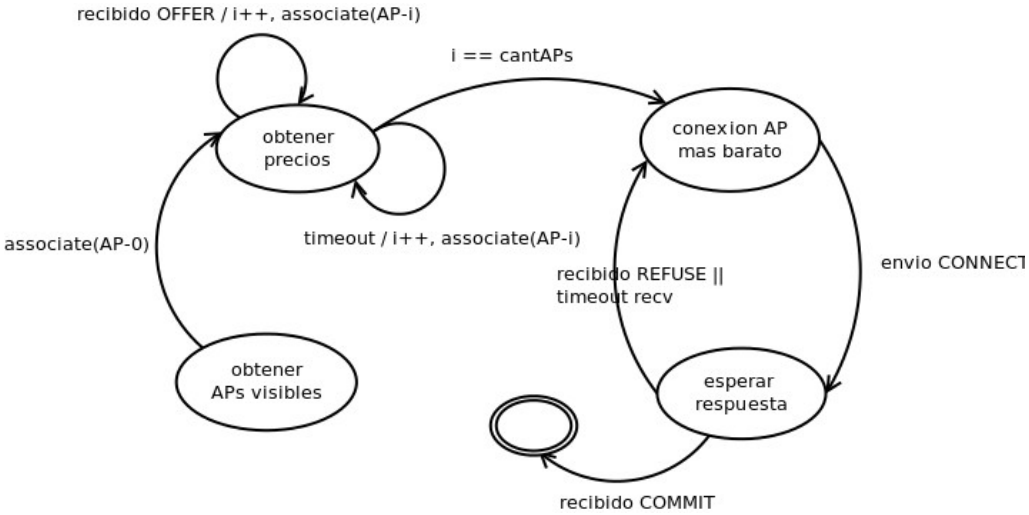


Diagrama usuario:



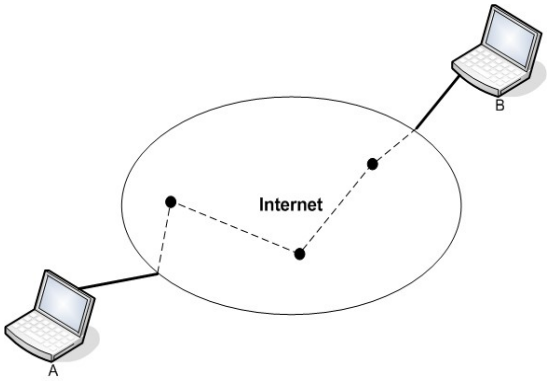
3. Servicio del punto de acceso:

```
void priceService(){
   //creo socket udp para la comunicacion con los usuarios
   int socket = socket(AF_INET, SOCK_DGRAM, 0);

   while (true){

      sleep(1); //espero 1 segundo para volver a enviar precio

      //se debe habiliar broadcast en el socket
      int broadcast = 1;
      setsockopt(socket, SOL_SOCKET, SO_BROADCAST, &broadcast, sizeof broadcast)

      //armo la direccion de broadcast
      struct sockaddr_in addr;
      socklen_t addr_size = sizeof addr;
      addr.sin_family = AF_INET;
      addr.sin_port = htons(12345);
      addr.sin_addr.s_addr = inet_addr(INADDR_BROADCAST);

      //envio el precio de manera broadcast
      char msg[5];
      sprintf(msg, %d, getPrice());
      int msg_size = strlen(msg);
      int sent_msg_size = sendto(socket, msg, msg_size, 0, (struct sockaddr*)&addr, addr_size);
   }
}

void connectionService(){
   //creo socket udp para la comunicacion con los usuarios
   int socket = socket(AF_INET, SOCK_DGRAM, 0);

   //bind
   struct sockaddr_in addr;
   socklen_t addr_size = sizeof addr;
   addr.sin_family = AF_INET;
   addr.sin_port = htons(6789);
   addr.sin_addr.s_addr = inet_addr(INADDR_ANY);
   bind(socket, (struct sockaddr*)&addr, addr_size);

   while (true){
      //estructuras para el recv
      struct sockaddr_in sta_addr;
      socklen_t sta_addr_size = sizeof sta_addr;
      char* conection = malloc(MAX_MSG_SIZE);

      //escuchar pedidos de conexion, obtengo la ip del usuario
       int  data_size  =  recvfrom(socket,  conection,  data_size,  0,  (struct  sockaddr*)&sta_addr,
&sta_addr_size);

      if (m == "CONNECT"){
         if (countUsers() < maxUsers){
            userConnected(sta_addr); //aviso al sistema que se acepto la conexion del usuario
            char *msg = "COMMIT";
            int msg_size = strlen(msg);
            sendto(sock, msg, 0, (struct sockaddr*)&sta_addr, sta_addr_size)
         } else{
            char *msg = "REFUSE";
            int msg_size = strlen(msg);
            sendto(sock, msg, 0, (struct sockaddr*)&sta_addr, sta_addr_size)
         }
      }
   }
}
```

Función del usuario:

```
define MAX_MSG_SIZE 128

void connectCheaper() {
   //creo socket udp para la comunicacion
   int socket = socket(AF_INET, SOCK_DGRAM, 0);

   //bind
   struct sockaddr_in addr;
   socklen_t addr_size = sizeof addr;
   addr.sin_family = AF_INET;
   addr.sin_port = htons(12345);
   addr.sin_addr.s_addr = inet_addr(INADDR_ANY);
   bind(socket, (struct sockaddr*)&addr, addr_size);

   //pedir lista de aps
   string* lista = list();

   string matriz [length(lista)][3];

   //estructuras para el recv
   struct sockaddr_in ap_addr;
   socklen_t ap_addr_size = sizeof ap_addr;
   char* precio = malloc(MAX_MSG_SIZE);

   //timeout de un minuto para esperar las respuestas del ap
   struct timeval tv;
   tv.tv_sec = 60;
   setsockopt(socket, SOL_SOCKET, SO_RCVTIMEO, (char *)&tv,  sizeof tv);

   //informarse del precio de cada ap
   for (int i=0; i<length(lista); i++){
      associate(lista[i]);
      //escuchar precio y almacenar dir del AP
      int d_size=recvfrom(socket, precio, data_size, 0, (struct sockaddr*)&ap_addr, &ap_addr_size);
      if (d_size != 0){
         matriz[i][0] = lista[i];
         matriz[i][1] = atoi(precio);
         matriz[i][2] = ap_addr;
      }else{
         matriz[i][0] = lista[i];
         matriz[i][1] = NULL;
         matriz[i][2] = ap_addr;
      }
   }

   //ordenar lista de aps de acuerdo a los precios obtenidos
   sort(matriz);

   bool connected = false;
   int i = 0;
   while (!connected && i<length(lista)){
      //asociarse y conectarse con el mejor ap
      if (matriz[i][2] != NULL){
         associate(matriz[i][0]);
         sendto(socket, "CONNECT", 7, 0, matriz[i][2], addr_size);
         char* response = malloc(MAX_MSG_SIZE);
         int data_size = recv(socket, response, data_size, 0);
         if (response == "COMMIT"){
            connected();
            connected = true;
         }else{
            i++;
         }
      }
   }
}
```

Problema 2 (30 puntos)



El MTU (*maximum transmission unit*) es el tamaño máximo de paquete que puede pasar por determinado enlace. En base a esta defínición y a lo visto en clase se pide:

a) En una red de varios routers y enlaces, defina los conceptos de MTU, fragmentación y “Path MTU” (la MTU de un camino)

b) Describa el comportamiento de un router al reenviar paquetes por un enlace con una MTU de “M” bytes. Describa los diferentes casos de acuerdo al valor del bit DF del encabezado IP, haciendo particular hincapié en los eventuales mensajes de control que se generen.

c) Se desea implementar un algoritmo que permita determinar el “path MTU” entre dos nodos, A y B, conectados a la misma red IP. El algoritmo deberá estar basado en los mensajes ICMP que se pueden escuchar en la red y podrá enviar paquetes de prueba. El mismo deberá depender unicamente de uno de los extremos (es decir “A” no debe requerir colaboración activa de “B”, salvo el comportamiento por defecto del stack de protocolos).

Se cuenta con las primitivas: *enviar_msg_ICMP(destino, tipo, código, largo_payload)* y *recibir_msg_ICMP(fuente).* Esta última devuelve un mensaje ICMP recibido de la fuente o *null* en caso de que pase un cierto tiempo sin recibir nada (implementa un timeout).

Se ofrece la siguiente tabla como referencia:

Tipo	**Nombre**	**Códigos para el mensaje tipo 3 (destino inalcanzable)**
0	Echo Reply	0 Net Unreachable
1	Unassigned	1 Host Unreachable
2	Unassigned	2 Protocol Unreachable
3	Destination Unreachabe	3 Port Unreachable
4	Source Quench	4 Fragmentation Needed and Don't Fragment was Set
5	Redirect	5 Source Route Failed
6	Alternate Host Address	6 Destination Network Unknown
7	Unassigned	7 Destination Host Unknown
8	Echo Request	8 Source Host Isolated
9	Router Advertisement	9 Communication with Destination Network is Administratively Prohibited
10	Router Selection	
11	Time Exceeded	10 Communication with Destination Host is Administratively Prohibited
12	Parameter Problem	

Solución:

a) **Definiciones:**

- *MTU*: es el tamaño máximo de trama que una cierta tecnología de capa 2 es capaz de transmitir.
- *Fragmentación*: cuando un router se ve en la situación de tener que enviar un paquete mas largo que la MTU del enlace de salida puede eventualmente **fragmentar** dicho paquete y reenviarlo en partes. Esta es una tarea pesada para un router, ya que le implica mantener estado de ciertos paquetes y sus fragmentos.
- *Path MTU*: en el caso de dos hosts que estan separados por mas de un hop los MTUs de los diferentes enlaces pueden ser diferentes. Se define entonces el Path MTU como el **mínimo** de todos los MTUs involucrados en el camino de un nodo a otro.

- Este Path MTU puede a veces variar con el tiempo (si los caminos cambian) y puede ser diferente dependiendo del sentido del trafico (un minimo para la ida y otro para la vuelta) en el caso de que el ruteo sea asimétrico.

b) DF: “Do not Fragment”

Paquete entrante *Pi* de largo *L*
MTU del enlace saliente *MTUo*

```
if L <= MTUo then
        transmitir normalmente Pi
else if L > MTUo and Pi.DF == 1 then
        descartar paquete Pi
        enviar msg ICMP (destino=Pi.source, tipo=3, codigo=4)
else if L> MTUo and Pi.DF == 0 then
        fragmentar paquete Pi
        transmitir fragmentos de Pi
```

Mensaje tipo 3: Destination unreachable
Codigo 4: “fragmentación necesaria pero DF encendido”

c) La estrategia básica es transmitir paquetes de prueba y escuchar los mensajes ICMP de error que regresan. El algoritmo similar a una bipartición pero simplificada.

Partimos de una resultado tentativo: sabemos que el path MTU no va a ser mas grande que el MTU del enlace por donde los paquetes destinados a B son encaminados.

```
Pmtu = MTU(enlace hacia B)

tries = 0
while true
        // envio un ping hacia el destino de largo el maximo tentativo
        enviar_msg_ICMP (destino=B, tipo = 8, largo = Pmtu)
        pkt = recibir_msg_ICMP(fuente=B)
        if pkt != Null then
                if pkt.code == 0 then
                        // es un echo reply, llegue al destino bien
                        // el MTU es el que hemos adivinado
                        return Pmtu
                else
                        // calculo nueva aproximación
                        // esta aproximación es bastante agresiva, se podría hacer mejor
                        // tiende a sub-estimar el pmtu
                        Pmtu = Pmtu * 0.85
                end
        else
                // posiblemente timeout u otro error de red
                // reintento con un valor menor
                // cuento los reintentos y fallo si son mas de 5 retorno un error
                Pmtu = Pmtu * 0.85
                tries++
                if tries >= 5 then
                        return False
                end
end
```