Redes de Computadoras

Curso 2023

Grupo 24

Informe - Obligatorio 2

Autores:
Braian Mendizábal
Ian Arazny
Favio Cardoso

Supervisor:
Prof. Leonardo Alberro

October 15, 2023

Contents

1 Descripción de la solución propuesta. 2

1.1 Funcionamiento Servidor 2

1.2 Funcionamiento Cliente 6

2 Decisiones tomadas al hacer la solución. 8

3 Problemas encontrados al hacer la solución. 9

4 Posibles mejoras de nuestra solución. 10

5 Pruebas realizadas. 11

6 ANEXO 1: STREAM-Server 13

7 ANEXO 2: STREAM-Client 19

1 Descripción de la solución propuesta.

Se procedió a la implementación de un servidor de transmisión de vídeo y un cliente encargado administrar el consumo de dicho vídeo. En primer término, es pertinente exponer la estrategia concebida para la configuración del servidor.

El concepto fundamental subyacente se basa en la recepción y posterior redistribución de datagramas de vídeo procedentes de una fuente, en este caso, VLC. Estos datagramas son recibidos por el servidor y transmitidos selectivamente a aquellos clientes que han sido admitidos y han manifestado su intención de visualizar el flujo de vídeo en tiempo real.

Para llevar a cabo esta tarea, se implementaron dos estructuras de datos; un arreglo destinado a retener a los clientes que se encuentran en estado de espera para recibir los datagramas de la transmisión, y otro destinado a aquellos clientes que han experimentado una interrupción en su conexión. La razón detrás de esta diferenciación radica en la necesidad de gestionar eficazmente los recursos y garantizar una transmisión fluida de vídeo.

Para asegurar una manipulación efectiva de los arreglos mencionados previamente, se incorporaron dos semáforos, cuya finalidad principal es facilitar el acceso y la eliminación de elementos sin incurrir en errores de sincronización. Dado que en el contexto de esta aplicación pueden ejecutarse simultáneamente múltiples hilos, resulta imperativo asegurar una coordinación adecuada y exenta de conflictos en las operaciones realizadas sobre dichos arreglos.

1.1 Funcionamiento Servidor

El servidor es encargado de las siguientes tareas:

- Creación de un socket maestro TCP (master).
- Bind del socket maestro a una dirección IP y un puerto del servidor, estos datos son seleccionados por el usuario al momento de iniciar el servidor y deben ser dentro de la red LAN en la que trabajará el mismo, de manera que pueda aceptar conexiones de los clientes en la red local. El puerto es seleccionado también al inicio y debe ser mayor que 1024 para no interferir con otros servicios que el sistema se encuentre corriendo.

- Puesta en escucha del socket maestro (server), se especificaron una cantidad prudente de 20 conexiones al servidor de manera de no saturar los recursos del servidor. Esto no sucede en el código cartilla ya que es un pseudocódigo y no se pretende tener tanta especificidad.

- Creación de un nuevo hilo que ejecutara la función udp-eater-puker encargada de la retransmisión de los datagramas consecuentemente para lograrlo es necesario que reciba por parámetro la dirección IP del servidor.

- En cada iteración del bucle de start_server se realizan las siguientes tareas:

    1. Se acepta una nueva conexión de cliente y se almacena en el socket "cliente". También se verifica si ocurrió algún error durante la aceptación de la conexión.
    2. Si el socket "cliente" es nulo (indicando que no se pudo aceptar la conexión), se sale del bucle, en Python se genera una excepción del tipo connection closed.
    3. Se crea un nuevo hilo de ejecución para atender al cliente recién aceptado. La función "CONTROLSTREAM" (que recibe por parámetro el socket, la IP y el puerto del cliente donde para obtenerla recurrimos a la operación getPeer en código cartilla y en Python es la propia operación de conexión la que nos sirve de la dirección del cliente) se utiliza para gestionar la comunicación con el cliente. Cada cliente se maneja en su propio hilo para permitir conexiones concurrentes.

- Cierre del socket del servidor (server).

Los arreglos socket_senders y sockets_interrupted están destinadas a almacenar la información de los objetos de la clase clienteSender, que son los clientes que se encuentran recibiendo activamente el stream y para los que solicitaron la interrupción respectivamente. Esta decisión esta basada en que es poco eficiente almacenar en el mismo arreglo todos los clientes conectados sin distinguir los que se encuentran interrumpidos. De forma que al momento de enviar los datagramas solo sean tomados los datos de aquellos clientes que desean recibir el stream, sin consumir ciclos de CPU en aquellos que solicitaron la interrupción de la emisión.

Se decidió implementar un acceso mutuo-excluido a estos arreglos para poder asegurar que las operaciones realizadas en los arreglos se cumplan en forma exclusiva, evitando conflictos en situaciones de concurrencia.

La función "udp-eater-puker" tiene como propósito recibir datagramas UDP y reenviarlos a todos los clientes que están registrados en el arreglo "sockets senders". Si los arreglos "sockets senders" y "socket interrupted" se encuentran vacíos, no se acumulará ningún datagrama recibido en el servidor.

Dentro de esta función, se crean dos sockets UDP esenciales: uno llamado "receptor", responsable de recibir los datagramas desde la fuente VLC, y otro denominado "emisor", utilizado exclusivamente para reenviar estos datagramas. El socket "receptor" se configura con una dirección local en la que se especifica "localhost" como la dirección IP y "65534" como el puerto donde se esperan los datagramas, tal como es explicitado en la propuesta del laboratorio. El socket "emisor" no necesita configuración adicional pues no es importante para los clientes conocer desde donde se están emitiendo los datagramas pues en ningún caso es viable una comunicación.

A continuación, se inicia un bucle encargado de cumplir con el control del acceso a las estructuras de datos compartidas (i.e.: los arreglos "socket senders" y "sockets interrupted").

Se procede a recibir datagramas UDP desde la fuente VLC utilizando el socket "receptor". La función "receive" se utiliza para realizar esta operación de manera bloqueante, lo que significa que el hilo se detendrá hasta que se reciba un datagrama. Además en Python la función receive cuenta con un parámetro que es la cantidad de bytes que puede leer del paquete que reciba, en este caso usamos 2048 debido a que experimentalmente llegamos a que el tamaño de paquete enviado por UDP era cercano a 1024 y decidimos duplicar ese tamaño de bytes para prevenir un posible error (como la llegada de dos datagramas muy cercanos). Luego de recibir un datagrama, se recorre un bucle "for" que itera sobre el arreglo de clientes activos. Para cada cliente en la lista, se reenvía el datagrama, utilizando el socket "emisor", hacia la dirección IP y el puerto asociados a ese cliente específico. Una vez que se ha completado el reenvío del datagrama a todos los clientes activos, se libera el semáforo "s1". Esto permite que otros hilos puedan acceder y modificar la estructura de datos compartida.

La función "CONTROLSTREAM" se encarga de gestionar la interacción con

un cliente que se conecta al servidor.

A continuación, se explica su funcionamiento: La función inicia con un bucle "do-while" que tiene como condición de salida la detección de la palabra "CONECTAR" con los datos recibidos del cliente. Durante este bucle, los datos enviados por el cliente se acumulan en un buffer. Si se produce un error en la recepción de datos o el cliente cierra la conexión, el socket del cliente se cierra, y la función finaliza. Una vez que se detecta la palabra "CONECTAR" con los datos del cliente (cosa que en la cartilla se hace con una función contains, en Python tiene su análogo con expresiones regulares), se extrae el número de puerto del buffer y se actualiza el buffer para contener el resto de los datos.

A continuación, se crea un objeto (struct con IP y puerto) "clienteSender" para representar al cliente, almacenando su dirección IP y el puerto extraído de la conexión. La idea de utilizar un objeto de esta clase fue con motivo de mantener en un solo lugar la información de cada cliente conectado de manera independiente. Este objeto se agrega al arreglo de "clientes activos" ("sockets senders") utilizando los semáforos creados anteriormente para evitar conflictos en escenarios multihilo. Luego, se envía un mensaje de confirmación al cliente mediante la función "enviarDatos", aquí nos encontramos con una diferencia importante con Python es que esta función no se encuentra implementada ya que no es necesaria, solamente se implementó con un send que si llegase a fallar un try catch muestra en pantalla la excepción ocurrida y a continuación sale del thread.

Después de esta fase inicial, se inicia un bucle "while" que continúa hasta que se reciba el comando "DESCONECTAR" del cliente. Dentro de este bucle, se recibe y procesa la información enviada por consola desde el cliente, y se almacenan los comandos enviados explicitados en la letra de la propuesta (que pueden ser "INTERRUMPIR", "CONTINUAR" o "DESCONECTAR"), esto es otra diferencia con Python donde cada comando viene en un mensaje por separado y no hallamos la forma de concatenar mensajes en un único stream de manera de ejecutar todos uno a continuación del otro. Si se detecta el comando "INTERRUMPIR" y el cliente no estaba previamente interrumpido, se procede a eliminarlo del arreglo de "clientes activos" y se agrega al arreglo de "clientes interrumpidos" ("sockets interrupted"), utilizando semáforos para garantizar la integridad de los arreglos compartidos. Si el comando es "CONTINUAR" y el cliente estaba previamente interrumpido, se elimina del arreglo de "clientes interrumpidos" y se agrega nueva-

mente al arreglo de "clientes activos". Se envía un mensaje de confirmación "OK" al cliente después de procesar cada comando. Si se produce un error durante el envío, se cierra la conexión del cliente. Si el cliente estaba interrumpido, se elimina del arreglo de "clientes interrumpidos"; de lo contrario, se elimina del arreglo de "clientes activos". Finalmente, cuando se recibe el comando "DESCONECTAR", se determina si el cliente estaba interrumpido o no, y se procede a eliminarlo del arreglo correspondiente. Luego se cierra la conexión del cliente.

1.2 Funcionamiento Cliente

El cliente es encargado de las siguientes tareas:

- Creación de un socket maestro TCP (master), para permitir la conexión con el servidor.
- Se conecta al servidor mediante serverIP y serverPort ingresados al momento de inicializar el cliente. En caso de error cierra el cliente.
- Se crea una función llamada conectar, donde se verifica que en el buffer al ingresar por comando CONECTAR contenga a continuación el puerto al que el cliente espera recibir el stream, retornándolo. En Python fue implementado usando expresiones regulares para hallar la sintaxis de comando CONECTAR.
- Se procede con la creación de un bucle sin condición donde se manejará el contacto del cliente con el servidor, en el cual se realizarán las siguientes tareas:
  - Se almacena en el buffer lo escrito por el cliente en la terminal.
  - A continuación dependiendo del comando escrito se realizan diferentes tareas, los comandos pueden ser: CONECTAR, INTERRUMPIR, CONTINUAR, DESCONECTAR.
    1. Si es INTERRUMPIR o CONTINUAR, se envía al servidor dicho comando y se recibe el mensaje de confirmación desde el.
    2. Si no esta conectado y la función conectar devuelve un puerto correcto, se envía al servidor lo escrito en el buffer y se recibe el mensaje de confirmación del servidor.

3. Si esta conectado y la función conectar devuelve un puerto correcto, se procede a mostrar el mensaje de error "Finalice la sesión actual para recibir en un nuevo puerto", esto es para que dada una sesión en un cliente no se pueda intercambiar de puertos de recepción del stream.
4. Si esta conectado y el comando ingresado es DESCONECTAR, se envía el comando al servidor, recibe el mensaje de confirmación y se procede a cerrar el socket.
5. En el caso de que lo escrito en la terminal no sea correcto, se devuelve "NO OK".

La implementación del cliente se llevó a cabo empleando dos parámetros esenciales: el serverIP, que se refiere a la dirección IP del servidor al cual se pretende establecer una conexión, y el serverPort, que se refiere al número de puerto al que se busca conectar. En este proceso, se genera un socket maestro de tipo TCP con el propósito de establecer la conexión con el servidor especificado. A su vez, se inicializa una variable denominada conectadoUDP que se fija inicialmente en "false", haciendo referencia a que el cliente aún no ha ejecutado el comando "CONECTAR" exitosamente.

Se inicia un bucle infinito (while true) que permite la recepción de información proporcionada a través de la línea de comandos, deteniendo la captura cuando se detecta el carácter de nueva línea "\n". El contenido recibido se almacena en un buffer para su posterior procesamiento.

Si lo ingresado es INTERRUMPIR o CONTINUAR entramos a un nuevo while donde enviamos al server dicho comando y cerramos el socket del servidor mientras no ocurra un error al enviar ya que en ese caso mostraremos en la terminal "Error de conexión", en cualquier caso saldremos del while de envío. Luego, se ingresa a un while true donde vamos a recibir respuesta del servidor, si no ocurre nada inesperado. En Python la operación receive de los sockets debe indicar un valor de bytes que espera recibir como máximo del stream, a diferencia del código en cartilla. Al recibir se verifica que el mensaje contenga "\n", y se hace un break. En cuyo caso que ocurra un error se muestra en la terminal "Conexión cerrada de forma inesperada".

Entre tanto no sea ninguno de los dos comandos anteriores los escritos en la

terminal, se verifica si no esta conectado aun, por medio de conectadoUDP(variable booleana) y la verificación mediante la función llamada conectar donde mandamos el buffer, verifica que en él se escribió el comando CONECTAR seguido del puerto del cliente que se retorna si es así, 0 si no lo es, y si es mayor al valor 1024 se actualiza la variable conectadoUDP a true. La rutina conectar fue implementada en Python utilizando nuevamente las expresiones regulares. Luego se ingresa a un while donde se le va a mandar al server lo escrito en la terminal, al igual que anteriormente si ocurre algo inesperado al enviar se muestra el mensaje "Error de conexión" y se procede a cerrar el server. Se recibe lo mandado por el servidor almacenándose en el buffer. Si ocurre un error se muestra "Conexión cerrada de forma inesperada" cerrándose el server y si no ocurre nada se verifica que el buffer contenga la finalización del mensaje con "\n" mostrándose en la terminal.

En el caso de que el cliente esté conectado y la función conectar devuelva el valor del puerto del cliente con su condición de ser mayor a 1024, se muestra en pantalla "Finalice la sesión actual para recibir en un nuevo puerto" y continúa el while inicial leyendo nuevamente lo escrito en la terminal. Si no estaba conectado o la función conectar devolvió el valor "0" o un valor menor a 1024 , se consulta si se escribió en la terminal DESCONECTAR, se envía al servidor el comando, si ocurre un error se muestra "Error de conexión", se cierra el servidor. Luego se recibe lo enviado por el servidor, si ocurre un error se muestra "Conexión cerrada de forma inesperada" cerrándose el servidor, se verifica que el buffer contenga "\n", se muestra en pantalla cerrándose el servidor.

Mientras no ocurre nada de lo anteriormente mencionado, se decide mostrar en pantalla el mensaje "NO OK".

2 Decisiones tomadas al hacer la solución.

- Fueron utilizados dos arreglos socket_senders y socket_interrupted.
- Fueron utilizados dos semáforos para controlar la concurrencia entre hilos que compiten por el uso de los recursos compartidos socket_senders y socket_interrupted.
- En el servidor se almacenan dos listas diferentes para los clientes conectados que están recibiendo la emisión y para los conectados que han solicitado la

interrupción.

- El servidor informa cada vez que un nuevo cliente se registra con su dirección IP y puerto de recepción. Informa también cuando un cliente se desconecta.
- La utilización de un objeto clienteSender por cada cliente conectado fue con motivo de mantener en un solo lugar la información de cada cliente (IP y puerto) de manera independiente.
- La implementación en el Cliente de la función conectar que al recibir por parámetro el buffer verifica que se cumpla con la sintaxis del comando "CONECTAR" en caso afirmativo retorna el puerto, en caso negativo retorna 0.
- No es posible intercambiar dinámicamente el puerto de VLC donde recibe el stream el cliente.
- El cliente cuando inicia lo hace estableciendo la dirección y el puerto del servidor al que se quiere conectar.
- Cuando el cliente envía un comando que no pertenece la protocolo le es respondido desde el propio Cliente "NO OK".

3 Problemas encontrados al hacer la solución.

- Encontrar la forma de que Python subdivida una misma entrada en función del carácter "\n".
- Encontrar la forma de identificar el puerto en el comando CONECTAR, cosa que se pudo solventar utilizando expresiones regulares.
- Inicialmente el rendimiento de la aplicación no era buena con vídeos de alta calidad, sin embargo se noto una mejora al aumentar el tamaño del buffer de recepción del servidor que decidimos que se quedara en 2048 principalmente porque el tamaño supera el tamaño del paquete enviado por UDP por VLC, de manera que pueda viajar por la red sin problemas.

- Lograr encontrar un bug en el programa que hacia que el rendimiento del servidor al iniciar se empobreciera de manera exagerada. El responsable era el thread encargado de realizar la recepción los datos enviados desde VLC en el servidor que se encontraba en un bucle infinito preguntando si existían clientes conectados, esto se solvento utilizando la operación bloqueante del receive de UDP, por lo que solo se transmitiría algo si algún paquete llega al servidor. Antes de eso el thread espera bloqueado.

4 Posibles mejoras de nuestra solución.

- Implementar una interfaz gráfica para el servidor.

- Una mejora para la implementación del servidor en Python es sacar los clientes del arreglo de clientes activos o interrumpidos según corresponda a los que sufren una excepción.

- Considerar una mejor solución para la distribución de datagramas a los clientes de forma mas paralelizada, logrando demorar lo menos posible el envío de la información incluso si la demora esta dada por el envío a los otros clientes. Esto puede tener una implementación si se generasen k hilos que envíen los datagramas a $\lfloor n/k \rfloor$ clientes dentro del arreglo "client senders". Se debe tener en cuenta que cuanto mas grande sea k, mas hilos hay y por tanto mas recursos se utilizaran, pero cuanto mas chico sea k mas demora experimentaran los clientes que se encuentren al final de la k-ésima sección de client_senders.

- El consumo del servidor es importante tenerlo en cuenta en la solución, pues notamos que siempre que se estén recibiendo datos desde VLC en el servidor el bucle de envío estaría tratando de reenviar los paquetes UDP a los clientes, pero si todos estuvieran interrumpidos esto solo serviría para gastar ciclos de procesador. Y además el consumo de CPU se despega rápidamente. Notamos que esto no es tan así cuando hay clientes recibiendo porque el tiempo consumido en las operaciones de envío de datagramas a los clientes alivian un poco el recurso CPU.

- Es una buena idea considerar la encriptación de la solución sobre la red pues tal como está un ataque de MITM es posible, solo bastaría hacerse pasar por la dirección IP de destino de los paquetes UDP de la transmisión para recibirla.

- Ya que la transmisión es la misma para todos los usuarios se podría aprovechar para que los mismos clientes se conviertan en retransmisores de la emisión como forma de alivianar la carga del servidor, una especie de símil P2P.

- No es mala idea pensar un sistema de autenticación en el propio servidor para cada conexión TCP, de manera que para diferentes usuarios con diferentes rangos se les proporcione con distintas transiciones.

- Se podría pensar en migrar el sistema hacia la utilización de HTTP como protocolo de cabecera y sobre el trabajar con tecnologías mas desarrolladas como webRTC.

- Una mejora viable es permitir que en una misma sesión entre el cliente y el servidor sea posible intercambiar el puerto de recepción de VLC del cliente sin tener que cerrar el programa cliente.

- Lograr que Python subdivida una misma entrada en función del carácter "\n".

- Implementar una interfaz de gráfica para el cliente.

5 Pruebas realizadas.

Las pruebas que logramos realizar fueron 2 principalmente: entre un host Windows y un guest Linux y puramente en un entorno Linux.

- Las pruebas realizadas teniendo como host a Windows y como guest a Linux fueron un tanto difíciles por la configuración que conlleva la instalación de estas maquinas virtuales. Por otro lado es importante que la aplicación servidor sea capaz de comunicarse con otros dispositivos en la red LAN, cosa que no esta activada por defecto en las maquinas virtuales, por lo que establecer una comunicación con el resto de la red LAN y con el propio host

también fue una inversión de tiempo. La prueba consistió en encender la emisión de VLC, encender el servidor (tanto el servidor como la emisión de CVLC se encontraron en Linux y enviando y recibiendo en localhost entre ellos) en una dirección de la red LAN, y luego comenzar a conectar hosts locales. En total llegamos a conectar 8 clientes y en todos la recepción fue buena.

- Las pruebas en entornos puramente de Linux fueron hechas en las PC de la FING donde pudimos replicar el experimento hecho con maquinas virtuales y notar un uso adecuado de los recursos del CPU (como se dijo antes estos se disparan cuando se esta emitiendo pero los clientes solicitan no recibir la emisión) allí pudimos experimentar con varias PC dentro de la misma subred que eran capaces de conectarse a la emisión sin problema alguno. Incluso probamos cambiar la emisión de VLC por una de un vídeo de mucha mas calidad y tampoco hubieron problemas en la transmisión (tras investigar un poco pudimos ver que esto es debido al ancho de banda de la red, pues la diferencia entre la emisión de un vídeo de baja o alta calidad son la cantidad de paquetes que deben llegar a destino para reproducir un frame, a mayor calidad mas paquetes necesarios y esto se ve directamente comprometido cuando contamos con un ancho de banda pequeño pues en tal caso los paquetes no son capaces de llegar a tiempo y la emisión se experimentaría trancada o entrecortada).

6 ANEXO 1: STREAM-Server

```
1  // Rutina de establecimiento de conexion
2
3  clienteSender {
4      string ip;
5      int port;
6  };
7
8  dynamic_array sockets_senders[clienteSender];
9  dynamic_array sockets_interrupted[clienteSender];
10
11 //semaphores
12 s1 = threading.Semaphore(1);
13 s2 = threading.Semaphore(1);
14
15 start_server() {
16     master = socket.tcp();
17     master.bind(ipaddress, ServerPort);
18     server = master.listen();
19
20     thread.new(udp-eater-puker, serverIP);
21
22     do {
23         cliente, error = server.accept();
24         if(cliente == nil)
25             return;
26         thread.new(CONTROLSTREAM, cliente,
               cliente.getPeer()[0]);
27     }while(true);
28
29     server.close();
30 }
31
32 //-------------------------------------------------------------
33
```

```
/*
 La idea que por cada datagrama que llega al server, tengo
    que reenviarlo a todos
los clientes, para hacerlo a todos los que esten conectados
   les paso el datagrama.
Si no hay nadie conectado entonces no acumulo nada.
*/

udp-eater-puker(serverIP) {
    receptor = socket.udp();
    receptor.bind(localhost, 65534);
    emisor = socket.udp();
    while(true) {
        datagrama, ip, puerto = receptor.receive();
        s1.acquire();
            foreach(sockets_senders : i) {
                i.emisor.send(datagrama, i.ip, i.port);
            }
        s1.release();
    }
}


enviarDatos(socket cliente, string mensaje) {
    do {
        mensaje, error =  cliente.send(mensaje);
        if(error <> '')
            return error;
    } while(mensaje <> '');
    return '';
}


//-------------------------------------------------------

// control del Stream:
```

```
68
69 CONTROLSTREAM(socket cliente, cliente_address) {
70     buff = {};
71     command = '';
72     interrupted = False;
73
74     //Inicialmente debe de solicitar la conexion
75     do {
76         data, error = cliente.receive();
77         if(error == 'closed') {
78             cliente.close();
79             print("Cliente Desconectado");
80             return;
81         }
82         buff = buff + data;
83     }while( not buff.contains('CONECTAR (%d)+ \n') );
84
85     // Me quedo con el puerto en la variable port
86     buff.extractWhere('CONECTAR ', port, '\n');
87     // resto del buffer
88     buff = buff.substring( buff.positionAt('CONECTAR (%d)+
           \n')+1, buff.length() );
89
90     // Con esta data pretendemos inicializar el sender UDP
           del stream al cliente
91     clienteSender cs;
92     cs.ip = cliente_address;
93     cs.port = port;
94
95     s1.acquire();
96     sockets_senders.add(cs);
97     s1.release();
98     // Con esto queda almacenado el socket de esta conexion
           particular
99
100    //Operacion para enviarle el ok
```

```
101     if(enviarDatos(cliente, 'OK\n') <> '') {
102         cliente.close();
103         s1.acquire();
104         socket_senders.remove(cs);
105         s1.release();
106         print("Cliente Desconectado");
107         return;
108     }
109 
110     while(command <> 'DESCONECTAR') {
111 
112         if(buff.positionAt('\n') == -1) {
113             do {
114                 data, error = cliente.receive();
115                 if(error == 'closed') {
116                     cliente.close();
117                     s1.acquire();
118                     sockets_senders.remove(cs);
119                     s1.release();
120                     return;
121                 }
122                 buff = buff + data;
123             }while(buff.positionAt('\n') == -1);
124         }
125 
126         command = buff.substring(0, buff.positionAt('\n'));
127         buff = buff.substring( buff.positionAt('\n')+1,
               buff.length() );
128 
129         if(command == 'INTERRUMPIR') {
130             if(not interrupted) {
131                 s1.acquire();
132                 sockets_senders.remove(cs);
133                 s1.release();
134 
135                 s2.acquire();
```

```
136                 sockets_interrupted.add(cs);
137                 s2.release();
138             }
139         }else if(command == 'CONTINUAR') {
140             if(interrupted) {
141                 s2.acquire();
142                 sockets_interrupted.remove(cs);
143                 s2.release();
144 
145                 s1.acquire();
146                 sockets_senders.add(cs);
147                 s1.release();
148             }
149         }
150         if(enviarDatos(cliente, 'OK\n') <> '') {
               //Operacion para enviarle el ok
151             cliente.close();
152             if(interrupted):
153                 s2.acquire();
154                 sockets_interrupted.remove(cs);
155                 s2.release();
156             else:
157                 s1.acquire();
158                 sockets_senders.remove(cs);
159                 s1.release();
160             print("Cliente Desconectado");
161             return;
162         }
163     }
164 
165     // Caso de DESCONECTAR
166     if(interrupted) {
167         s2.acquire();
168         sockets_interrupted.remove(cs);
169         s2.release();
170     }else{
```

```
171             s1.acquire();
172             sockets_senders.remove(cs);
173             s1.release();
174         }
175         cliente.close();
176         print("Cliente Desconectado");
177         return;
178     }
```

7 ANEXO 2: STREAM-Client

```
1  // Rutina de conexion y uso al stream server
2
3  stream-client(string serverIP, integer serverPort) {
4      //apertura del socket de conexion
5      master = socket.tcp();
6      server, error = master.connect(serverIP, serverPort);
7      if(error == 'failure')
8          return;
9      conectadoUDP = False;
10
11     while(true) {
12       // devuelve una linea finalizada en \n
13       buff = readIOLine();
14
15       if(buff == 'INTERRUMPIR\n' or buff == 'CONTINUAR\n') {
16         while(buff <> '') {
17             buff, error = server.send(buff);
18             if(error == 'closed') {
19                 print('ERROR DE CONEXION');
20                 server.close();
21                 return;
22             }
23         }
24         while(true) {
25             buff, error = server.receive();
26             if(error == 'closed') {
27                 print('Conexion cerrada de forma
                       inesperada');
28                 server.close();
29                 return;
30             }
31             if(buff.contains('\n')) {
32                 print(buff);
33                 break;
```

```
34              }
35          }
36      }else if(not conectadoUDP and conectar(buff) > 1024) {
37          conectadoUDP = True;
38          while(buff <> '') {
39              buff, error = server.send(buff);
40              if(error == 'closed') {
41                  print('ERROR DE CONEXION');
42                  server.close();
43                  return;
44              }
45          }
46          while(true) {
47              buff, error = server.receive();
48              if(error == 'closed') {
49                  print('Conexion cerrada de forma
                        inesperada');
50                  server.close();
51                  return;
52              }
53              if(buff.contains('\n')) {
54                  print(buff);
55                  break;
56              }
57          }
58      }else if (conectadoUDP and conectar(buff) > 1024){
59          print("Finalice la sesion actual para recibir en un
                nuevo puerto");
60          continue;
61      }else if (buff == 'DESCONECTAR\n'){
62          while(buff <> '') {
63              buff, error = server.send(buff);
64              if(error == 'closed') {
65                  print('ERROR DE CONEXION');
66                  server.close();
67                  return;
```

```
68              }
69          }
70          while(true) {
71              buff, error = server.receive();
72              if(error == 'closed') {
73                  print('Conexion cerrada de forma
                       inesperada');
74                  server.close();
75                  return;
76              }
77              if(buff.contains('\n')) {
78                  print(buff);
79                  server.close();
80                  return;
81              }
82          }
83        }else{
84            writeIOLine("NO OK\n");
85        }
86      }
87  }
88
89
90  //Rutina para averiguar si la variable cumple con la
       sintaxis de conectar
91  // si lo hace entonces retorna el valor del puerto al que
       el cliente espera recibir el stream.
92
93  conectar(buff) {
94      if (buff.contains('CONECTAR (%d)+ \n')) {
95          return buff.extractWhere('CONECTAR ', port, '\n');
96      }
97      return 0;
98  }
```