Práctica 1
Programación Distribuida y Tiempo Real

Lucas Di Cunzolo
Santiago Tettamanti

Abstract—Introducción a Sockets en C y Java.

Index Terms—Programación Distribuida y Tiempo Real, c, java, LaTeX, sockets

1 Punto 1

Identifique similitudes y diferencias entre los sockets en C y en Java.

Como primer pantallazo a los lenguajes, tenemos lo siguiente. En C, se utilizará la librería estándar **"socket.h"**, mientras que en Java, se va a utilizarán los paquetes **"java.net"**, aunque utilizando ciertos paquetes para la interacción con el socket, provenientes de la librería **"java.io"**, utilizando las clases **InputStream** y **OutputStream**

1.1 Similitudes y diferencias entre C y Java

Como primera y más grande diferencia entre ambos lenguajes, se puede remarcar la diferenciación entre "socket cliente" y "socket servidor", planteado por la librería de java. Lo que no sucede en C.

Para iniciar un socket en java, en modo servidor, se utiliza la clase **ServerSocket**, mientras que para iniciar uno en modo cliente, se utiliza la clase **Socket**.

En C, ambas creaciones quedan bajo la responsabilidad de la función **Socket()**, solo que para diferenciar el socket, el servidor debe "bindear" y quedarse a la espera de datos. Esto lo hace usando **Bind()** y **Listen()** respectivamente.

El cliente, se conectará a un socket utilizando la función **Connect()**

2 Punto 2

Tanto en C como en Java (directorios csock-javasock)

2.1 Inciso A

¿Por qué puede decirse que los ejemplos no son representativos del modelo c/s?

En cliente-servidor el servidor está continuamente esperando alguna conexion por parte de algún cliente, en este caso el servidor cierra la conexion al recibir el primer request y enviar la respuesta. El cliente no usa la respuesta recibida por el servidor, en cambio el servidor si usa lo enviado por el cliente y realiza acciones con esos datos que no tienen que ver con la respuesta a la petición (aunque se sobreentiende que es para mostrar el ejemplo). Tampoco se respetan los pasos de conexión: inicialización, envío/recepción de peticiones y finalización; en el ejemplo solo le hace el segundo paso, no existe handshake con el servidor o mensajes de finalización de conexión. Es más un ejemplo de conexión entre procesos usando sockets que de client/servidor puro.

2.2 Inciso B

Muestre que no necesariamente siempre se leen/escriben todos los datos involucrados en las comunicaciones con una llamada read/write con sockets.

2.2.1 csock

Se modificó el programa para aceptar varios tamaños de buffer (Ver apéndice A) Para probarlo rápidamente, se presenta la regla en make, lo que generará clientes y servidores que envian y reciben bytes

En este caso, la cantidad máxima de caracteres que se llegaron a leer fue de

```
# Genera todos los clientes
# y servidores
# 10^3 | 10^4 | 10^5 | 10^6 bytes
make all_buffers

# O generar uno especifico
# Buffer de x bytes
make BUFFER_SIZE=x

# O generar solo buffer de:
# 10^3 bytes
make

# 10^4 bytes
make 10000

# 10^5 bytes
make 100000

# 10^6 bytes
make 1000000
```

Para ver el código, se puede ír al Apéndice, subsección ”Punto 2-B”

2.3 Inciso C

Agregue a la modificación anterior una verificación de llegada correcta de los datos que se envían (cantidad y contenido del buffer), de forma tal que se asegure que todos los datos enviados llegan correctamente, independientemente de la cantidad de datos involucrados.

Se implementó un chequeo por la correcta llegada de los datos al proceso server, tanto en cantidad como el contenido. Para el chequeo de cantidad se hizo que el cliente, primero que nada, envíe la cantidad total de datos que va a enviar y luego envíe los datos aguardando una respuesta por parte del server; el server por su parte, primero recibirá la cantidad de datos que se van a enviar y luego, cuando realmente se reciban los datos comparará esa cantidad recibida al inicio con la cantidad de datos recibidos. Si la cantidad es igual quiere decir que los datos se recibieron correctamente (al menos en cantidad), y enviará un mensaje al cliente indicando que los datos llegaron bien y no debe enviar más; en cambio si la cantidad no es igual quiere decir que no se recibieron todos los datos, por lo que enviará al cliente un mensaje indicando que debe enviar más y la cantidad de datos recibidos hasta el momento. Así el cliente dependiendo de la respueta del servidor o terminará la comunicación o enviará los datos faltantes comenzando de nuevo el ciclo. Para el chequeo de contenido el cliente enviará un checksum de los datos a enviar. El servidor, luego de recibir todos los datos, comparará el checksum recibido por el cliente con el generado por el mismo (usando la misma función que el cliente). Si ambos checksums son iguales quiere decir que los datos no fueron corrompidos en la comunicación, en cambio si son distintos los datos fueron corrompidos (sufrieron alguna modificación en la comunicación).

Para ver el código, se puede ír al Apéndice, subsección ”Punto 2-B”

Para ver capturas de la ejecución de los programas, puede ir al Apéndice de capturas

2.4 Inciso D

Grafique el promedio y la desviación estándar de los tiempos de comunicaciones de cada comunicación. Explique el experimento realizado para calcular el tiempo de comunicaciones.

Para este punto, se plantea calcular los tiempos de la comunicaciónes, esto quiere decir, el tiempo que tarda el cliente en establecer la conexión con el servidor, una vez establecida, se contempla el envío de los datos por parte del cliente, y la recepción por parte del servidor. Una vez terminado todo este flujo, se puede dar por concluída la comunicación.

El contéo del tiempo se realizó del lado del **cliente**.

Para tomar los datos para realizar el promedio y la desviación estándar de cada tamaño

de buffer, se cuenta con un script en el mismo directorio de csock llamado *get_data* Dicho script ejecutara repetídas veces la conexión cliente-servidor y volcará los tiempos en un archivo en el mismo directorio, llamado **buffer_[BUFFER_SIZE]_data**, donde **BUFFER_SIZE** es el tamaño del buffer con el que se corrió dicho script.

El script acepta 2 paŕametros:

1) Cantidad de iteraciones a realizar - Por defecto en 10
2) Tamaño del buffer a testear - Por defecto en 10^3

En el apéndice C, se pueden encontrar archivos de datos de buffers de tamaños 10^3, 10^4, 10^5 y 10^6

3 PUNTO 3

¿Por qué en C se puede usar la misma variable tanto para leer de teclado como para enviar por un socket? ¿Esto sería relevante para las aplicaciones c/s?

3.1 Comunicación mediante sockets Tipos

En un programa C, para el envío de datos por un socket, se puede utilizar cualquier variable de tipo puntero. Con esto, se entiende que se puede utilizar hasta la misma variable que se utilizó para leer desde teclado (tipi char*).
Todo puntero en C, puede utilizarse de dos maneras.

- Manera estática:

```
int main()
{
  char buffer[256];
  // Uso de la variable buffer
  return 0;
}
```

  Manejando memoria de manera estática, nos ahorramos el alocado de la memoria previo al uso, simplificando el código. Las principales desventajas son que si se necesita más de 256 caracteres, va a ser necesario cambiar de variable, y que de utilizar solamente 1 caracter, se tiene **siempre** alocado para 256 caracteres. La alternatica es utilizar memoria dinámica.

- Manera dinámica:

```
#include <stdlib.h>
int main()
{
  char *buffer = NULL;
  /* No se puede usar
     la variable buffer */
  buffer = (char*)malloc(256);
  /* Ya se puede usar
     la variable buffer */
  return 0;
}
```

  En este caso va a ser necesario llamar a alguna de las funcionas para alocado de memoria antes de utilizar la variable. De no hacer, generará errores.
  La principal ventaja de utilizar memoria de esta manera, es la posibilidad de realocar su espacio en cualquier momento. Como principal desventaja es que se le complejidad al código.

Luego, el socket acepta como buffer una variable de tipo **void***, lo que significa que se le puede enviar un puntero a cualquier tipo. El mayor problema que puede taer esto es que, como son bytes, el cliente no sepa como interpretarlos.

4 PUNTO 4

¿Podría implementar un servidor de archivos remotos utilizando sockets?
Describa brevemente la interfaz y los detalles que considere más importantes del diseño.

No solo sería posible, sino que ya existen 2 protocolos para envio de archivos por la red.

Estos protocolos se diferencian en, orientado a la conexion **FTP** o no orientado a la conexion **TFTP**.

4.1 FTP

Un servidor FTP se podría implementar utilizando 2 sockets para el servidor, uno utilizado

para los datos, y otro utilizado para el control. (Puertos 20 y 21 en el estándar, respectivamente). Ambos socket deben ser del tipo TCP.
Se cuenta con instrucciónes específicas (muy similares a las de linux) para el uso de los archivos, como por ejemplo:

- **ls:** Lista los archivos.
- **cd:** Cambia de directorio.
- **close:** Cierra la conexión con el servidor remoto (TCP FIN).
- **delete:** Elimina un archivo.
- **get:** Baja un archivo.
- **mkdir:** Crea un directorio.

Entre otras instrucciones.
*Todas las acciones se ejecutan **sobre** el filesystem del servidor remoto FTP.*

4.2 TFTP

Un servidor TFTP utiliza solamente un socket, para realizar la conexión, del tipo UDP. Cuenta con un conjunto de instrucciones mucho más reducido que el servidor FTP.

4.3 Implementación propia

Para la implementación de un servidor básico, pensaría en un grupo de comandos similar al FTP, pero acotado tanto en cantidad, como en seguridad de cada una.

Para seguir completamente un cliente FTP, como se dijo anteriormente, es necesario realizar una conexión TCP completa, eso quiere decir, con un saludo de 3 vías, y un cierre también de 3 vías.

Como instrucciones iniciales, pensaría las siguientes:

- **ls:** Lista los archivos.
- **add:** Agregar un archivo.
- **get:** Baja un archivo.
- **delete:** Elimina un archivo.

Para el manejo de los comandos el servidor tiene que manejar 2 instancias. La primer instancia es la de comandos.
El servidor, en principio, se encuentra a la espera de alguno de los comandos dichos anteriormente. Cualquier palabra diferente de los comandos va a dar un mensaje de 'comando no reconocido'.

Luego, cada comando deberá reconocer las acciones a realizar luego.

*Nota: para simplificar el funcionamiento del servidor, se asume un **único** directorio.*

4.3.1 LS - Listar archivos

En principio, este comando no representa mayores problemas. Desde el cliente se le manda la palabra `ls` o `list`.

Luego, el servidor contesta con la lista de archivos que tiene. El listado de archivos lo puede obtener mediante la función `readdir` Previamente abriento el directorio con la función `opendir` Se debe verificar el tamaño de los datos enviados desde el servidor para asgurarse que lleguen todos.

4.3.2 ADD - Agregar un archivo

Este puede ser el comando más complejo de implementar de la lista. Luegamente, en primer instancia se manda el nombre del comando, junto con el path relativo al archivo a subir.

Desde el cliente, se verifica la existencia del archivo y, de existir, lo abre. Luego comprueba el tamaño del archivo y lo envía al servidor. Para comprobar la integridad del archivo, se envía por último un *checksum*, el cual el servidor debe contrastar contra un checksum del archivo que le llegó.

4.3.3 GET - Baja un archivo

Este comando presenta una complejidad similar al anterior, pero ahora la carga se presenta del lado del servidor.

En primer instancia, el cliente manda la palabra `get`, seguida del nombre del archivo a bajar.

El servidor comprueba la existencia del archivo, de no existir lo informa, de existir, primero abre el archivo y comprueba su tamaño. Luego genera un checksum del archivo. Por último, envía el archivo y el checksum por el socket.

4.3.4 DELETE - Elimina un archivo

En primera instancia, el cliente envía la palabra `delete`, seguído del nombre del archivo a eliminar.

El servidor comprueba que exista el archivo, y lo elimina.

En este comando no es necesario comprobar el tamaño del nombre del archivo dado que el **máximo** path admitido en linux es de 4096 (definido por la variable `PATH_MAX`, en `linux/limits.h`)

5 PUNTO 5

*Defina qué es un servidor con estado (**stateful server**) y qué es un servidor sin estado (**stateless server**).*

5.1 Servidor con estado

Un servidor con estado, es aquel que recuerda el estado en el que habia quedado el sistema en la anterior conexión de un cliente, sin la necesidad de mantener la conexión. Esto quiere decir, por ejemplo, un servidor con estado es aquel que puede mantener una página web por ejemplo.

5.2 Servidor sin estado

Un servidor sin estado, es aquel que no almacena el estado entre conexiones de un cliente. Esto quiere decir, que el servidor **siempre** arranca en el mismo estado para todas las conexiones.

APPENDIX A
CÓDIGO C

Se presentarán secciones del código C de principal relevancia.
Cambios planteados al cliente de csock - `csock/client.c`

A.1 Punto 2-B

A.1.1 Cliente

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <unistd.h>
#include <sys/time.h>


#ifndef BUFFER_SIZE
#define BUFFER_SIZE 1000
#endif

double dwalltime()
{
  double sec;
  struct timeval tv;

  gettimeofday(&tv, NULL);
  sec = tv.tv_sec + tv.tv_usec/1000000.0;
  return sec;
}

void error(char *msg)
{
  perror(msg);
  exit(0);
}

int main(int argc, char *argv[])
{
  int sockfd, portno, n;
  struct sockaddr_in serv_addr;
  struct hostent *server;

  char buffer[BUFFER_SIZE];
  if (argc < 3) {
```

```
    fprintf(stderr,"usage %s hostname port\n", argv[0]);
    exit(0);
  }
  portno = atoi(argv[2]);
  double time = dwalltime();
  sockfd = socket(AF_INET, SOCK_STREAM, 0);
  if (sockfd < 0)
    error("ERROR opening socket");
  server = gethostbyname(argv[1]);
  if (server == NULL) {
    fprintf(stderr,"ERROR, no such host\n");
    exit(0);
  }
  bzero((char *) &serv_addr, sizeof(serv_addr));
  serv_addr.sin_family = AF_INET;
  bcopy((char *)server->h_addr,
       (char *)&serv_addr.sin_addr.s_addr,
       server->h_length);
  serv_addr.sin_port = htons(portno);
  if (connect(sockfd,(const struct sockaddr*)&serv_addr,sizeof(serv_addr)) < 0
    error("ERROR connecting");
  bzero(buffer,BUFFER_SIZE);
  int i = 0;
  buffer[i++] = 'r';
  for (i; i < BUFFER_SIZE-4; i++) {
    buffer[i] = 'a';
  }
  buffer[i++] = 'w';
  buffer[i++] = 'r';
  buffer[i] = '!';
  n = write(sockfd,buffer,BUFFER_SIZE);
  if (n < 0)
    error("ERROR writing to socket");
  bzero(buffer,BUFFER_SIZE);
  n = read(sockfd,buffer,BUFFER_SIZE-1);
  if (n < 0)
    error("ERROR reading from socket");
  //printf("%s\n",buffer);
  printf("%g\n", dwalltime()-time);
  return 0;
}
```

A.1.2 Servidor

```
/* A simple server in the internet domain using TCP
   The port number is passed as an argument */
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>
```

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>

#ifndef BUFFER_SIZE
#define BUFFER_SIZE 1000
#endif

double dwalltime()
{
  double sec;
  struct timeval tv;

  gettimeofday(&tv, NULL);
  sec = tv.tv_sec + tv.tv_usec/1000000.0;
  return sec;
}

void error(char *msg)
{
  perror(msg);
  exit(1);
}

int main(int argc, char *argv[])
{
  int sockfd, newsockfd, portno, clilen;
  char buffer[BUFFER_SIZE];
  struct sockaddr_in serv_addr, cli_addr;
  int n;
  if (argc < 2) {
    fprintf(stderr,"ERROR, no port provided\n");
    exit(1);
  }
  sockfd = socket(AF_INET, SOCK_STREAM, 0);
  if (sockfd < 0)
    error("ERROR opening socket");
  bzero((char *) &serv_addr, sizeof(serv_addr));
  portno = atoi(argv[1]);
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_addr.s_addr = INADDR_ANY;
  serv_addr.sin_port = htons(portno);
  if (bind(sockfd, (struct sockaddr *) &serv_addr,
        sizeof(serv_addr)) < 0)
    error("ERROR on binding");
  listen(sockfd,5);
  clilen = sizeof(cli_addr);
  newsockfd = accept(sockfd,
```

```
      (struct sockaddr *) &cli_addr,
      &clilen);
  if (newsockfd < 0)
    error("ERROR on accept");
  double time = dwalltime();
  bzero(buffer,BUFFER_SIZE);
  n = read(newsockfd,buffer,BUFFER_SIZE-1);
  if (n < 0) error("ERROR reading from socket");
  printf("Here is the message!\nRead return value: %d\n", n);
  n = write(newsockfd,"I got your message",18);
  if (n < 0) error("ERROR writing to socket");
  return 0;
}
```

A.2 Punto 2-C

A.2.1 Cliente

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <unistd.h>
#include <sys/time.h>


#ifndef BUFFER_SIZE
#define BUFFER_SIZE 1000
#endif

unsigned long hash(unsigned char *str)
{
  unsigned long hash = 5381;
  int c;

  while (c = *str++)
    hash = ((hash << 5) + hash) + c; /* hash * 33 + c */

  return hash;
}

double dwalltime()
{
  double sec;
  struct timeval tv;

  gettimeofday(&tv, NULL);
  sec = tv.tv_sec + tv.tv_usec/1000000.0;
```

```
  return sec;
}

void error(char *msg)
{
  perror(msg);
  exit(1);
}

int main(int argc, char *argv[])
{
  int sockfd, portno, silent, n;
  struct sockaddr_in serv_addr;
  struct hostent *server;

  char buffer[BUFFER_SIZE];
  if (argc < 3) {
    fprintf(stderr,"usage %s hostname port\n", argv[0]);
    exit(0);
  }

  portno = atoi(argv[2]);

  if (argc == 4) {
    silent = atoi(argv[3]);
  } else {
    silent = 0;
  }
  double time = dwalltime();
  sockfd = socket(AF_INET, SOCK_STREAM, 0);
  if (sockfd < 0)
    error("ERROR opening socket");
  server = gethostbyname(argv[1]);
  if (server == NULL) {
    fprintf(stderr,"ERROR, no such host\n");
    exit(0);
  }
  bzero((char *) &serv_addr, sizeof(serv_addr));
  serv_addr.sin_family = AF_INET;
  bcopy((char *)server->h_addr,
      (char *)&serv_addr.sin_addr.s_addr,
      server->h_length);
  serv_addr.sin_port = htons(portno);
  if (connect(sockfd,(const struct sockaddr*)&serv_addr,sizeof(serv_addr)) < 0
    error("ERROR connecting");
  bzero(buffer,BUFFER_SIZE);
  int i = 0;
  buffer[i++] = 'r';
  for (i; i < BUFFER_SIZE-4; i++) {
```

```
    buffer[i] = 'a';
  }
  buffer[i++] = 'w';
  buffer[i++] = 'r';
  buffer[i] = '!';

  long int bufferi[1];
  bufferi[0]=BUFFER_SIZE-1;
  char response[2];
  response[0]='M';
  i=0;

  unsigned long buffer_hash = hash(buffer);

  n = write(sockfd,bufferi,4);

  while(response[0] == 'M'){

    n = write(sockfd,&buffer[i],BUFFER_SIZE-1);

    n = read(sockfd,response,2 );

    if (response[0] == 'M'){
      n = read(sockfd,bufferi,BUFFER_SIZE-1);
      i=bufferi[0];
      if (n < 0)
        error("ERROR reading from socket");
    }
  }
  if (!silent) {
    printf("%s\n",response);

    printf("Check: %lu\n", buffer_hash);
  }

  n = write(sockfd, &buffer_hash,sizeof(buffer_hash));

  printf("%g\n", dwalltime()-time);
  return 0;
}
```

A.2.2 Servidor

```
/* A simple server in the internet domain using TCP
   The port number is passed as an argument */
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>
#include <sys/types.h>
```

```c
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>

#ifndef BUFFER_SIZE
#define BUFFER_SIZE 1000
#endif

double dwalltime()
{
  double sec;
  struct timeval tv;

  gettimeofday(&tv, NULL);
  sec = tv.tv_sec + tv.tv_usec/1000000.0;
  return sec;
}

void error(char *msg)
{
  perror(msg);
  exit(1);
}

int main(int argc, char *argv[])
{
  int sockfd, newsockfd, portno, clilen;
  char buffer[BUFFER_SIZE];
  struct sockaddr_in serv_addr, cli_addr;
  int n;
  if (argc < 2) {
    fprintf(stderr,"ERROR, no port provided\n");
    exit(1);
  }
  sockfd = socket(AF_INET, SOCK_STREAM, 0);
  if (sockfd < 0)
    error("ERROR opening socket");
  bzero((char *) &serv_addr, sizeof(serv_addr));
  portno = atoi(argv[1]);
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_addr.s_addr = INADDR_ANY;
  serv_addr.sin_port = htons(portno);
  if (bind(sockfd, (struct sockaddr *) &serv_addr,
        sizeof(serv_addr)) < 0)
    error("ERROR on binding");
  listen(sockfd,5);
  clilen = sizeof(cli_addr);
  newsockfd = accept(sockfd,
      (struct sockaddr *) &cli_addr,
```

```
      &clilen);
  if (newsockfd < 0)
    error("ERROR on accept");
  double time = dwalltime();
  bzero(buffer,BUFFER_SIZE);
  n = read(newsockfd,buffer,BUFFER_SIZE-1);
  if (n < 0) error("ERROR reading from socket");
  printf("Here is the message!\nRead return value: %d\n", n);
  n = write(newsockfd,"I got your message",18);
  if (n < 0) error("ERROR writing to socket");
  return 0;
}
```

Cambios planteados al servidor de csock - `csock/server.c`

```
/* A simple server in the internet domain using TCP
   The port number is passed as an argument */
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>

#ifndef BUFFER_SIZE
#define BUFFER_SIZE 1000
#endif

unsigned long hash(char *str)
{
  unsigned long hash = 5381;
  int c;

  while (c = *str++)
    hash = ((hash << 5) + hash) + c; /* hash * 33 + c */

  return hash;
}

double dwalltime()
{
  double sec;
  struct timeval tv;

  gettimeofday(&tv, NULL);
  sec = tv.tv_sec + tv.tv_usec/1000000.0;
```

```
  return sec;
}

void error(char *msg)
{
  perror(msg);
  exit(1);
}

int main(int argc, char *argv[])
{
  int sockfd, newsockfd, portno, clilen;
  char buffer[BUFFER_SIZE];
  struct sockaddr_in serv_addr, cli_addr;
  int n;
  if (argc < 2) {
    fprintf(stderr,"ERROR, no port provided\n");
    exit(1);
  }
  sockfd = socket(AF_INET, SOCK_STREAM, 0);
  if (sockfd < 0)
    error("ERROR opening socket");
  bzero((char *) &serv_addr, sizeof(serv_addr));
  portno = atoi(argv[1]);
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_addr.s_addr = INADDR_ANY;
  serv_addr.sin_port = htons(portno);
  if (bind(sockfd, (struct sockaddr *) &serv_addr,
        sizeof(serv_addr)) < 0)
    error("ERROR on binding");
  listen(sockfd,5);
  clilen = sizeof(cli_addr);
  newsockfd = accept(sockfd,
      (struct sockaddr *) &cli_addr,
      &clilen);
  if (newsockfd < 0)
    error("ERROR on accept");
  double time = dwalltime();
  bzero(buffer,BUFFER_SIZE);
  long int bufferi[1];
  int i=0;

  n = read(newsockfd,bufferi,4);
 printf("%d\n",bufferi[0] );
  while(bufferi[0] != i){
    i+= read(newsockfd,&buffer[i],BUFFER_SIZE-1);
    if (i != bufferi[0]){
      write(newsockfd,"M",2);
      write(newsockfd,&i,2);
```

```
    }
  }
   n = write(newsockfd,"I",2);
  if (n < 0) error("ERROR writing to socket");

  //printf("Here is the message! %s\nRead return value: %d\n %d\n",buffer,i,bu

  unsigned long buffer_checksum;
  n = read(newsockfd, &buffer_checksum,sizeof(buffer_checksum));
  printf("Checksum de client: %lu\nMi checsum %lu\n", buffer_checksum, hash(bu


  return 0;
}
```

APPENDIX B
CÓDIGO JAVA

Al código C
Se presentarán secciones del código Java de principal relevancia.

```
import java.io.*;
import java.net.*;

public class Client
{
  public static void main(String[] args) throws IOException
  {
    /* Check the number of command line parameters */
    if ((args.length != 2) || (Integer.valueOf(args[1]) <= 0) )
    {
      System.out.println("2 arguments needed: serverhostname port");
      System.exit(1);
    }

    /* The socket to connect to the echo server */
    Socket socketwithserver = null;

    try /* Connection with the server */
    {
      socketwithserver = new Socket(args[0], Integer.valueOf(args[1]));
    }
    catch (Exception e)
    {
      System.out.println("ERROR connecting");
      System.exit(1);
    }

    /* Streams from/to server */
    DataInputStream  fromserver;
```

```
    DataOutputStream toserver;

    /* Streams for I/O through the connected socket */
    fromserver = new DataInputStream(socketwithserver.getInputStream());
    toserver   = new DataOutputStream(socketwithserver.getOutputStream());

    /* Buffer to use with communications (and its length) */
    byte[] buffer;

    /* Get some input from user */
    Console console  = System.console();
    String inputline = console.readLine("Please enter the message: ");

    /* Get the bytes... */
    buffer = inputline.getBytes();

    /* Send read data to server */
    toserver.write(buffer, 0, buffer.length);

    /* Recv data back from server (get space) */
    buffer = new byte[256];
    fromserver.read(buffer);

    /* Show data received from server */
    String resp = new String(buffer);
    System.out.println(resp);

    fromserver.close();
    toserver.close();
    socketwithserver.close();
  }
}
```

APPENDIX C
DATOS LEÍDOS

Datos del buffer 10^3:

0.00169897
0.000709057
0.00171304
0.000823021
0.00102401
0.001755
0.000813961
0.00102806
0.0013299
0.000697136
==========

Media: 0.0115922
Desviacion estandar: 0.000408169

Datos del buffer 10^4:

0.000947952
0.00128698
0.0014441
0.00104308
0.000712872
0.00134587
0.00132895
0.000955105
0.000814915
0.00142288
==========
Media: 0.0113027
Desviacion estandar: 0.000252947

Datos del buffer 10^5:

0.00239491
0.00434995
0.00417805
0.00427604
0.00441694
0.00422311
0.00419116
0.0408249
0.00423694
0.00434208
==========
Media: 0.0774341
Desviacion estandar: 0.0110417

Datos del buffer 10^6:

0.173446
0.0515101
0.055784
0.052412
0.0521331
0.052444
0.0523751
0.0516021
0.0573621
==========
Media: 0.599069
Desviacion estandar: 0.0378364

APPENDIX D
CAPTURAS

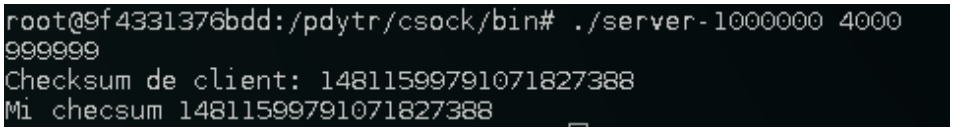

Fig. 1. Servidor

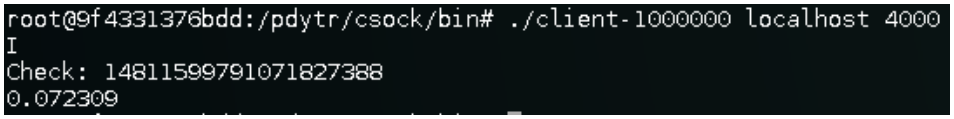

Fig. 2. Cliente