**Instituto Politécnico Nacional**

Escuela Superior de Cómputo

Reporte Práctica Bucket Sort

**Integrantes del Equipo:**

Barrera Pérez Carlos Tonatihu

Castillo Reyes Juan Daniel

**Grupo:** 2CM11

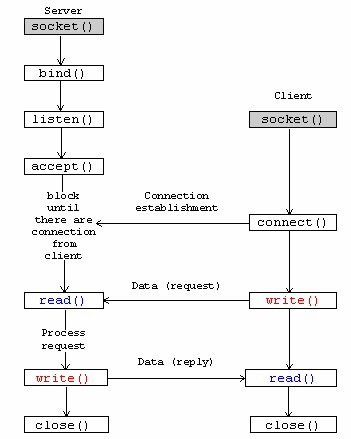
**Unidad de Aprendizaje:** Aplicaciones para Comunicaciones en Red

**Profesor:** Axel Ernesto Moreno Cervantes

Introducción

# Sockets de flujo en C

Un socket de flujo es aquel que está preparado para conexiones de tres pasos, en ingles se denomina **three way handshake**, ya que algún cliente se pueda conectar, primero le pide permiso al servidor, el servidor acepta o declina (supongamos que acepta), y a partir de ahí comienza el envío y la recepción de datos. Se denomina así porque es un "apretón de manos" en tres pasos.



Los sockets se organizan en estructuras para contener la información tanto del host como de cliente. Las estructuras que se utilizan para trabajar con sockets en C son las siguientes.

|  |
| --- |
| struct sockaddr\_in {  short int sin\_family; //Protocolo para la conexión  unsigned short int sin\_port; //Puerto para la conexión  struct in\_addr sin\_addr; //Estructura a la dirección IP  unsigned char sin\_zero[8]; //Relleno  };  struct in\_addr {  unsigned long s\_addr; //Contiene la dirección IP del host  };  struct sockaddr {  unsigned short sa\_family; //Protocolo  char sa\_data[14]; //Dirección IP del host  }; |

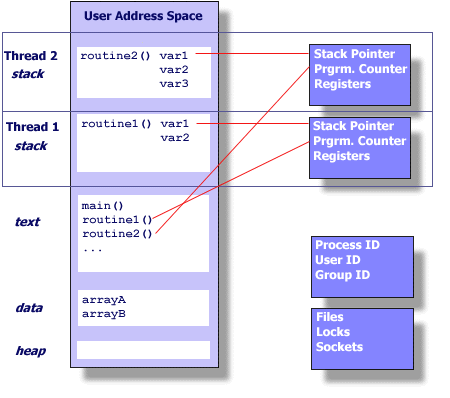
# Hilos en C

Los hilos dentro de un proceso comparten la misma memoria del proceso. Eso quiere decir que, si un hilo toca una variable, todos los demás hilos del mismo proceso verán el nuevo valor de la variable. Esto hace imprescindible el uso de [semáforos](http://www.chuidiang.org/clinux/ipcs/semaforo.php) o mutex para evitar que dos hilos accedan a la vez a la misma estructura de datos.

Es por esto por lo que un hilo es menos costoso que utilizar un proceso, esta es la razón por la cual a los hilos se les conoce como procesos ligeros.

**Cada hilo posee:**

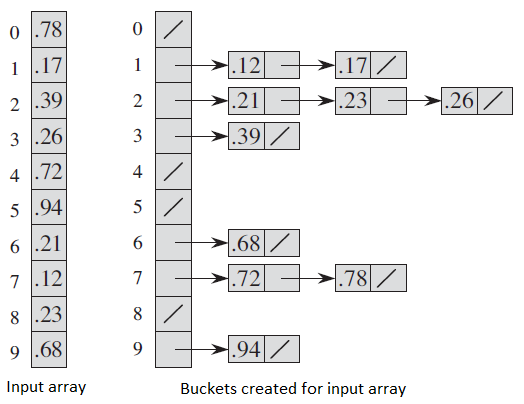
* Identificador
* Pila
* Conjunto de registros
* Contador de programa



Para escribir programas multihilo en C podemos utilizar la biblioteca pthread que implementa el standard POSIX (Portable Operating System Interface). Para ello en nuestro programa debemos incluir algunas cabeceras y a la hora de compilar es necesario enlazar el programa con la biblioteca.

# Bucket Sort

El ordenamiento por cubetas es principalmente útil cuando los datos de entrada se encuentran distribuidos uniformemente sobre un rango. Un ejemplo del uso de este ordenamiento es ordenar un conjunto de números decimales entre el 0 y el 1, pero con la condición de ordenarlos en tiempo lineal O(n).



La mayoría de algoritmos de ordenamiento solo pueden alcanzar un tiempo O(n log n) por lo que la opción a utilizar es el ordenamiento por cubetas. El algoritmo es el siguiente.

|  |
| --- |
| bucketSort(arreglo[], n)   1. Crear n cubetas vacias. 2. Hacer lo siguiente por cada elemento del arreglo    1. Insertar el elemento arreglo[i] en cubeta[n\*arreglo[i]] 3. Ordenar las cubetas de forma individual utilizando ordenamiento por inserción o algún otro algoritmo. 4. Concatenar todas las cubetas ordenadas |

Desarrollo

En esta practica se implemento el algoritmo de ordenamiento por cubetas utilizando sockets e hilos. Se crean N clientes y N servidores, la tarea de los clientes es seleccionar los elementos de su respectiva cubeta y mandar esta por medio de un socket de flujo al servidor que le corresponde y este se encargara de ordenar dicha cubeta para finalmente regresársela al cliente y que este a su vez la mande al hilo principal con el fin de que se ingrese en el arreglo ordenado final.

En el siguiente archivo se definió la cabecera del ordenamiento, esta cuenta con una estructura que se envía a través del socket la cual contiene el tamaño de la cubeta y los elementos que la componen.

Archivo bucket\_sort.h

|  |
| --- |
| #ifndef \_\_BUCKET\_SORT\_\_  #define \_\_BUCKET\_SORT\_\_  #include <pthread.h>  #include <time.h>  #include "comunicacion\_sockets.h"  #include "merge\_sort.h"  // Tamaño del arreglo de numeros aleatorios a ordenar  #define TAM 100  // Mensaje que se envia entre clientes y servidores  // Contiene el arreglo de numeros y su tamaño  struct Mensaje{  int tam;  int \*numeros;  };  // Definicion de las funciones que se utilizan  void \*cliente(void \*);  void \*servidor(void \*);  void llenar\_numeros();  void crear\_clientes(int);  void crear\_servidores(int);  int meter(int \*, int);  int numeros[TAM];  int arreglo\_final[TAM];  #endif |

También contiene las definiciones de las funciones que se utilizaron. El siguiente archivo tiene la descripción de las funciones que se usan para crear los N servidores y N clientes junto al código para que cada servidor y cliente corra en su respectivo hilo, por lo que en total se tendrán 2N hilos corriendo sin contar el hilo principal.

Archivo bucket\_sort.c

|  |
| --- |
| #include "bucket\_sort.h"  // Creamos N servidores que ordenaran una cubeta  void crear\_servidores(int N) {  pthread\_t hilos[N];  int error;  for (int i = 0; i < N; i++) {  int \*j = malloc(sizeof(int));  \*j = i;  error = pthread\_create(&hilos[i], NULL, &servidor, (void \*)j);  if(error)  manejador\_errores(error, "CREAR SERVIDORES");  }  sleep(1);  }  // Se crean los clientes y se seleccionan los numeros  // que perteneceran a su cubeta  void crear\_clientes(int N) {  int rango = TAM/N;  pthread\_t hilos[N];  int \*indices;  int inicio = 0;  int error;  // Creacion de hilos y seleccion de numeros  for (int i = 0; i < N; ++i) {  indices = (int \*) malloc(sizeof(int)\*3);  indices[0] = i;  indices[1] = inicio;  inicio += rango;  indices[2] = inicio;  if (i == N-1 && inicio < TAM)  indices[2] = TAM;  error = pthread\_create(&hilos[i], NULL, cliente, (void \*)indices);  if(error)  manejador\_errores(error, "CREAR CLIENTE");  }  // Se espera a que termine cada hilo en orden para introducir  // Los elementos en el arrreglo final  int iniciar = 0;  for (int i = 0; i < N; i++){  int \*ordenados;  error = pthread\_join(hilos[i], (void \*\*) &ordenados);  if (error)  manejador\_errores(error, "JOIN CLIENTE");  iniciar = meter(ordenados, iniciar);  }  }  // Funcion que mete los cubetas ordenadas en el arreglo final  int meter(int \*ordenados, int inicio) {  int i = 0;  while (ordenados[i] != -1){  arreglo\_final[inicio++] = ordenados[i++];  }  return inicio;  }  // Generacion de numeros aleatorios  void llenar\_numeros() {  srand(time(NULL));  for (int i=0; i < TAM; i++) {  numeros[i] = rand() % TAM;  printf("%d ", numeros[i]);  }  printf("\n");  }  // Servidor que se encarga de recibir una cubeta del cliente  // la ordena por merge-sort y la devuelve al cliente  void \*servidor(void \*indice) {  int \*dato = (int \*)indice;  int puerto = 8000 + dato[0];  char pto[10];  snprintf(pto, 10, "%d", puerto);  int serverSocket = iniciar\_servidor(pto);    printf("Servidor #%d listo, esperando..\n", dato[0]);  int clienteSocket;  socklen\_t ctam;  char hbuf[NI\_MAXHOST], sbuf[NI\_MAXSERV];  struct sockaddr\_storage c\_addr; // connector's address  for (;;) {  ctam = sizeof(c\_addr);  clienteSocket = accept(serverSocket, (struct sockaddr \*)&c\_addr, &ctam);  if (clienteSocket == -1) {  perror("servidor: accept");  continue;  }  if (getnameinfo((struct sockaddr \*)&c\_addr, sizeof(c\_addr), hbuf, sizeof(hbuf), sbuf,  sizeof(sbuf), NI\_NUMERICHOST | NI\_NUMERICSERV) == 0)  printf("Cliente conectado desde %s:%s\n", hbuf, sbuf);  char b[200];  memset(&b, 0, sizeof(b));  read(clienteSocket, b, sizeof(b));  struct Mensaje \*msj\_desordenados = (struct Mensaje \*) b;  merge\_sort(msj\_desordenados->numeros, 0, msj\_desordenados->tam-1);  write(clienteSocket, (const char\*)msj\_desordenados, sizeof(struct Mensaje));  close(clienteSocket);  break;  }  close(serverSocket);  return NULL;  }  // El cliente simplemente se conecta al servidor y envia  // su cubeta, espera y la recibe para proceder a mandarla  // al hilo principal  void \*cliente(void \*indices) {  int \*datos = (int \*)indices;  int desordenados[TAM];  int j = 0;  for (int i = 0; i < TAM; i++ ) {  if (numeros[i] >= datos[1] && numeros[i] < datos[2]){  desordenados[j++] = numeros[i];  }  }  int puerto = 8000 + datos[0];  char pto[10];  snprintf(pto, 10, "%d", puerto);  int clienteSocket = iniciar\_cliente(pto);  struct Mensaje \*msj\_desordenados = (struct Mensaje \*) malloc(sizeof(struct Mensaje));  msj\_desordenados->tam = j;  msj\_desordenados->numeros = desordenados;  write(clienteSocket, (const char\*)msj\_desordenados, sizeof(struct Mensaje));  char b[200];  memset(&b, 0, sizeof(b));    read(clienteSocket, b, sizeof(b));  struct Mensaje \*msj\_ordenados = (struct Mensaje \*) b;  int \*ordenados = malloc((j+1)\*sizeof(int));  for (int i = 0; i < j; i++)  ordenados[i] = msj\_ordenados->numeros[i];  ordenados[j] = -1;  close(clienteSocket);  return (void \*) ordenados;  } |

El siguiente archivo es la cabecera del archivo que tiene las funciones necesarias para crear el socket servidor y socket cliente además de funciones que se utilizan para manejar errores.

Archivo comunicacion\_sockets.h

|  |
| --- |
| #ifndef \_\_COMUNICACION\_SOCKETS\_\_  #define \_\_COMUNICACION\_SOCKETS\_\_  #include <unistd.h> // close read write  #include <sys/socket.h> // Aqui esta todo lo que dice socket  #include <netdb.h> //getaddrinfo() getnameinfo() freeaddrinfo()  #include <errno.h>  #include <stdlib.h>  #include <stdio.h>  #include <string.h>  // Definicion de las funciones que se utilizan  int iniciar\_servidor(const char \*);  int iniciar\_cliente(const char \*);  void manejador\_errores(int, char \*);  void desplegar\_error(char \*msj);  #endif |

Archivo comunicacion\_sockets.c

|  |
| --- |
| #include "comunicacion\_sockets.h"  // Crea un servidor socket  int iniciar\_servidor(const char \*PUERTO) {  struct addrinfo \*lista;  struct addrinfo hints;  memset(&hints, 0, sizeof(hints));  hints.ai\_family = AF\_INET; /\* IPv4 o IPv6 \*/  hints.ai\_socktype = SOCK\_STREAM;  hints.ai\_flags = AI\_PASSIVE; // use my IP  hints.ai\_protocol = 0; /\* Any protocol \*/  hints.ai\_canonname = NULL;  hints.ai\_addr = NULL;  hints.ai\_next = NULL;  int rv;  int v = 1;  if ((rv = getaddrinfo("localhost", PUERTO, &hints, &lista)) != 0) {  fprintf(stderr, "getaddrinfo: %s\n", gai\_strerror(rv));  exit(1);  }  struct addrinfo \*p;  int serverSocket;  for (p = lista; p != NULL; p = p->ai\_next) {  // Creamos un socket  if ((serverSocket = socket(p->ai\_family, p->ai\_socktype, p->ai\_protocol)) == -1) {  perror("servidor: socket");  continue;  }  if (setsockopt(serverSocket, SOL\_SOCKET, SO\_REUSEADDR, &v, sizeof(int)) == -1)  desplegar\_error("servidor: setsockopt");    // Asignamos el socket a un puerto  if (bind(serverSocket, p->ai\_addr, p->ai\_addrlen) == -1) {  close(serverSocket);  perror("servidor: bind");  continue;  }  break;  }  freeaddrinfo(lista);  if (p == NULL) {  fprintf(stderr, "servidor: bind\n");  exit(EXIT\_FAILURE);  }  // Le decimos al sistema que permita conecciones en un puerto  if (listen(serverSocket, 5) == -1)  desplegar\_error("servidor: listen");  return serverSocket;  }  // Crea un cliente socket  int iniciar\_cliente(const char \*PUERTO) {  struct addrinfo hints, \*servinfo, \*p;  // descriptor, mandar datos, bits enviados  int clienteSocket, rv;  // Limpiamos llenando de ceros  memset(&hints, 0, sizeof hints);  hints.ai\_family = AF\_INET; /\* Allow IPv4 or IPv6 familia de dir\*/  hints.ai\_socktype = SOCK\_STREAM; // Usaremos socket de flujo  hints.ai\_protocol = 0; // Puedo especificar TCP pero mejor que lo haga el sistema    // Pila de protocolos para IP  if ((rv = getaddrinfo("localhost", PUERTO, &hints, &servinfo)) != 0) {  fprintf(stderr, "getaddrinfo: %s\n", gai\_strerror(rv));  exit(EXIT\_FAILURE);  }  // Intentamos crear el socket con algun protocolo devuelto en la parte anterior  // Como IPv4 e IPv6  for(p = servinfo; p != NULL; p = p->ai\_next) {  if ((clienteSocket = socket(p->ai\_family, p->ai\_socktype,p->ai\_protocol)) == -1) {  perror("cliente: socket");  continue;  }  if (connect(clienteSocket, p->ai\_addr, p->ai\_addrlen) == -1) {  close(clienteSocket);  perror("cliente: connect");  continue;  }  break;  }//for  // Si no me pude conectar  if (p == NULL) {  fprintf(stderr, "cliente: error al conectar con el servidor\n");  exit(EXIT\_FAILURE);  }  freeaddrinfo(servinfo); // all done with this structure  return clienteSocket;  }  // Para desplegar errores  void manejador\_errores(int error, char \*msj) {  errno = error;  perror(msj);  exit(EXIT\_FAILURE);  }  // Para desplegar errores  void desplegar\_error(char \*msj) {  perror(msj);  exit(EXIT\_FAILURE);  } |

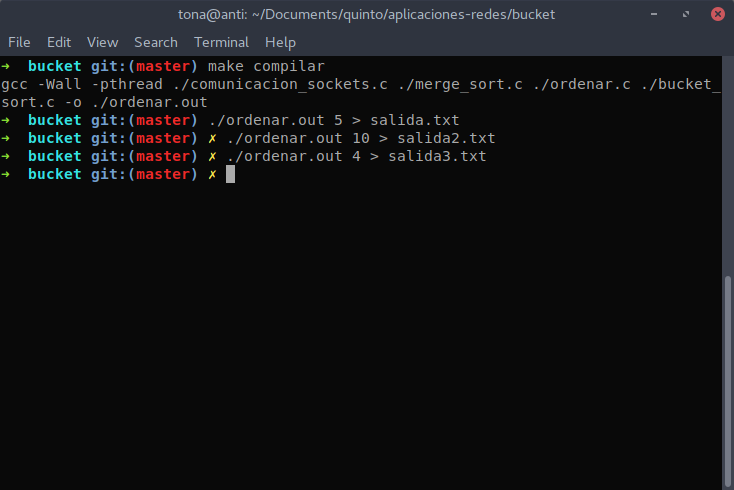
Esta es la función principal que manda a llamar las funciones para la generación de números, creación de servidores y clientes y finalmente desplegar los números ordenados.

Archivo main.c

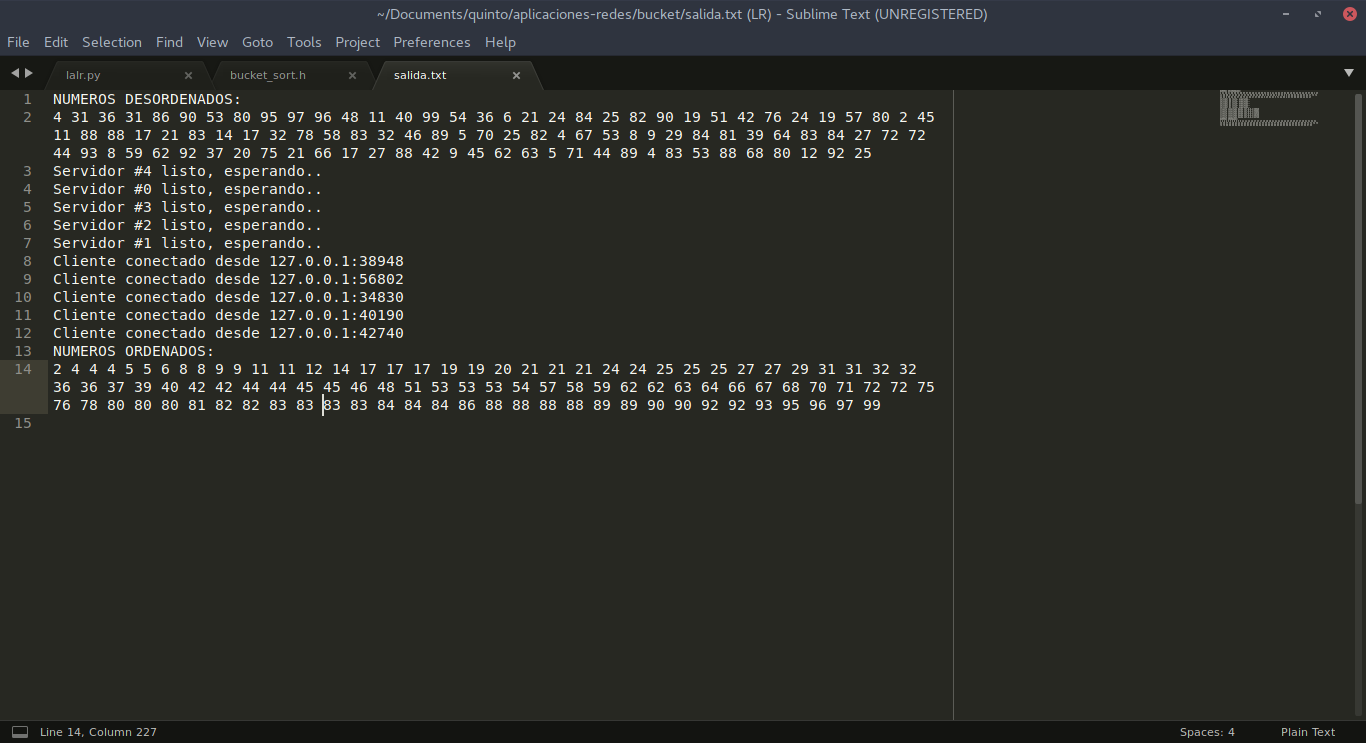
|  |
| --- |
| #include "bucket\_sort.h"  // Funcion que manda a ordenar los numeros  int main(int arc, char \*argv[]) {  // Se debe de ingresar el numero de hilos  if (arc < 2){  printf("%s\n", "Ingresa el numero de hilos plis");  return -1;  }  printf("%s\n", "NUMEROS DESORDENADOS:");  // Generacion de numeros  llenar\_numeros();  // Creamos N servidores que recibiran las cubetas  crear\_servidores(atoi(argv[1]));  // Creamos N clientes  crear\_clientes(atoi(argv[1]));  printf("%s\n", "NUMEROS ORDENADOS:");  // Imprimimos los numeros ordenados  for (int i = 0; i < TAM; i++)  printf("%d ", arreglo\_final[i]);  printf("\n");  return 0;  } |

Pruebas

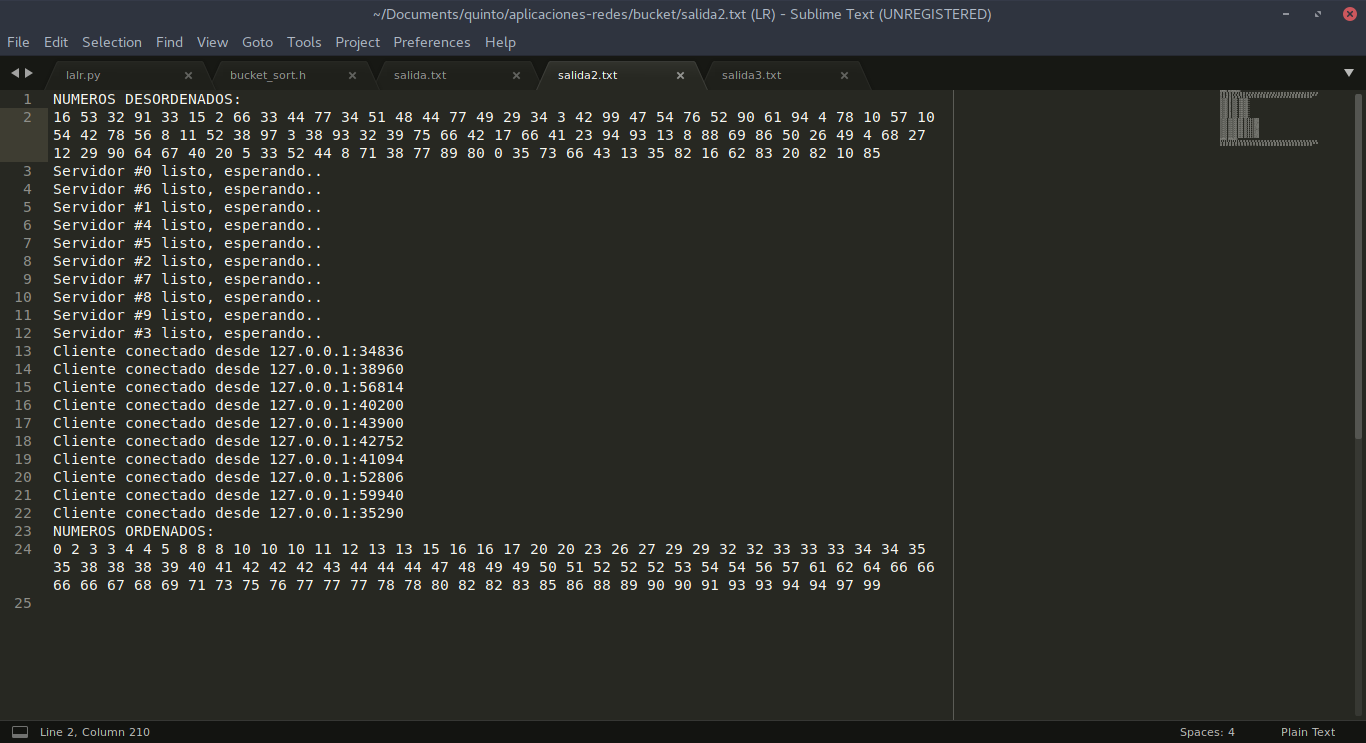
En la siguiente imagen se puede observar la compilación y tres diferentes pruebas en las cuales se ingresa un numero diferente de cubetas a utilizar en el ordenamiento a su vez esto genera que se creen diferente número de servidores y clientes.



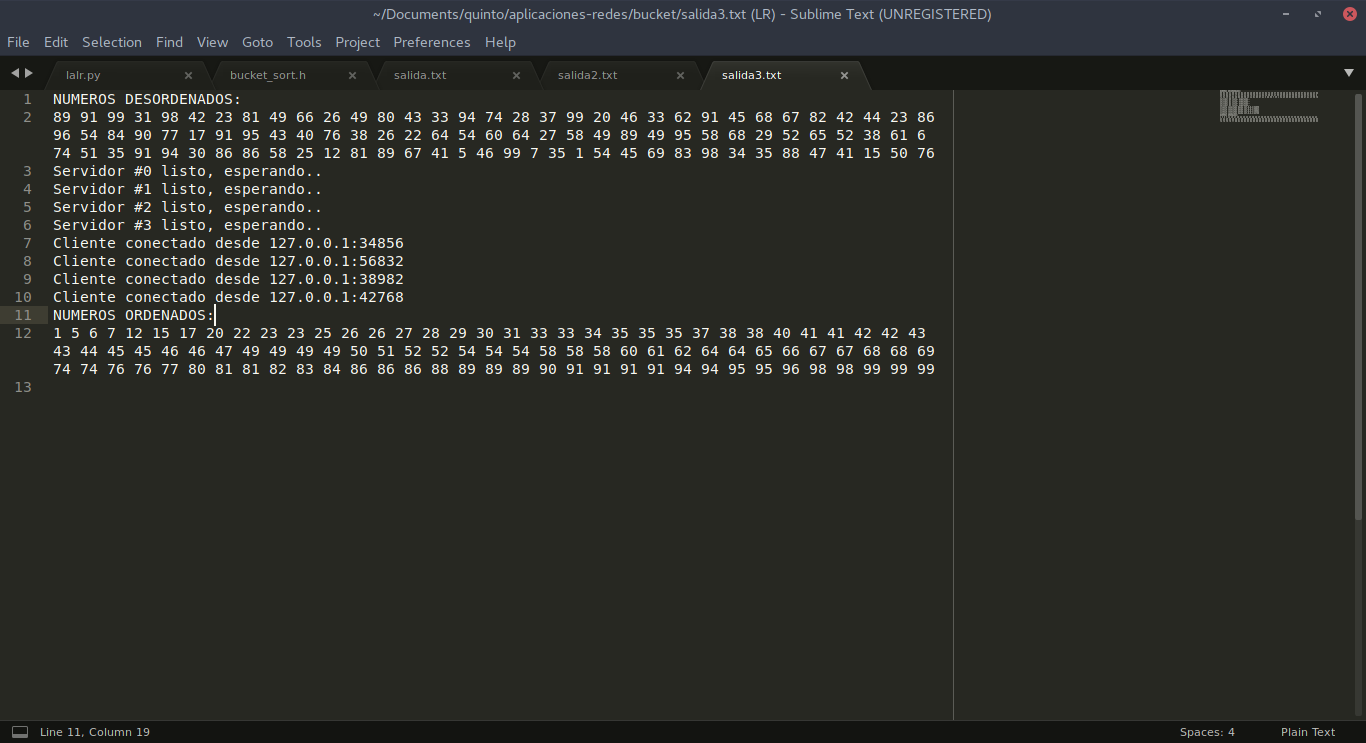
# Prueba 1



# Prueba 2



# Prueba 3



Conclusiones

En esta práctica se demostró que el combinar herramientas como lo son los hilos y los sockets nos permite realizar tareas en poco tiempo y de forma eficiente y que de no buscar una forma óptima de realizarlas pueden llegar a ser complicadas y que pueden llegar a necesitar muchos recursos (memoria y tiempo).

El problema de ordenar números es una muestra de esto, debido a que es una tarea de suma importancia en la computación y que se busca obtener el mejor resultado posible. Y ya que se utiliza el ordenamiento por cubetas junto al uso de hilos se reduce la complejidad de esta tarea.

Sin embargo, es importante tener en cuenta que para crear un hilo hay que considerar los recursos que se utilizaran y que estos no sobrepasen los recursos de los que dispone la computadora.

Referencias

* [1] M. Donahoo and K. Calvert, *TCP/IP sockets in C*, 1st ed. San Francisco [etc]: Morgan Kaufmann Publishers.
* [2] T. Cormen, C. Leiserson, R. Rivest and C. Stein, *Introduction to algorithms*, 3rd ed. Cambridge, Massachusetts: The MIT Press, 2009.