1 Manual de uso

El software consta de dos carpetas, *Lab2Server* y *Lab2Client*, donde se encuentran los programas para el cliente y servidor respectivamente. En su interior, podrá encontrarse una carpeta llamada *bin*, dónde se encontrará el ejecutable del mismo.

El ejecutable contenido en *Lab2Server* debe ser ejecutado del lado del servidor, y el de *Lab2Client* en el cliente. Antes de ser ejecutado, el programa instalado en el cliente debe ser configurado con la dirección IP del servidor, por lo cual debe editarse la primera línea del archivo *src/ClientController.h* con la IP del servidor. Luego de ésto, se debe regresar a la carpeta raíz del software y ejecutar las siguientes dos instrucciones en orden: *make clean* y *make*

Al ejecutar el software del cliente, intentará conectarse directamente con el servidor, si la conexión es exitosa, recibirá un mensaje informándole de ello,

Connection successfully

de otro modo, se le informará que la conexión no fue exitosa y el programa quedará intentando conectarse al servidor. No se permitirá realizar ninguna acción hasta que haya una conexión exitosa.

Trying to connect to server... : Connection refused

Al ingresar al cliente, se mostrará el siguiente menú :

1. **Enter record** Al ingresar pide uno a uno, los campos de registro de una mascota que desee ser guardada.

2. **Show record** Al ingresar muestra el número de registros presentes y solicita el número del registro que se desee ver. A partir del número de registro ingresado, se abrirá en un editor de texto la historia clínica de la mascota para ser visualizada, y si se desea también, editada.

3. **Delete record** Al ingresar muestra el número de registros presentes y solicita el número del registro que se desee borrar.

4. **Search record** Solicita el nombre de alguna mascota que desee buscarse, a continuación, se mostrarán todos los registros que coincidan completamente con el nombre. No se distingue mayúsculas de minúsculas. En caso de no existir registro con el nombre deseado, le será informado.

5. **Exit** Cierra el programa.

Por otra parte, para el servidor, bastará con ejecutarlo y el mismo hará todas las operaciones necesarias para un correcto funcionamiento.

2 Especificaciones

Es necesario contar con al menos de 65 kbs en espacio en disco para el cliente, y 128 kbs para el servidor. Además, para el servidor hay que considerar que necesita usar más espacio en disco para guardar la información de las mascotas que desee guardarse, así como también, para guardar el registro de todas las operaciones que realiza, por tanto, pueden llegar a ocuparse hasta 1.6 gbs (al cargar diez millones de estructuras).

En cuanto a memoria RAM, su consumo es casi despreciable, no supera los 84 kbs para el servidor, y 64 kbs para el cliente.

3 Informe de elaboración

1. Cliente

   (a) main.c > main.h

   Está encargado de administrar las funcionalidades generales del software, todo lo relacionado con la interacción con el usuario (entradas y salidas) y de ejecutar las acciones correspondientes con base a estas interacciones.

```
#define HANDLE_ERROR(arg) perror(arg);

struct DogType {
    char name[32];
    char type[32];
    int age;
    char race[16];
    int height;
    double weight;
    bool sex;
};

struct DogType *p;

void wait();
void closeProgram();

bool getSex(char p);
void showMenu();
int openFile(char* name);
int deleteFile(char* name);
void printDogType(struct DogType *pet);

void enterRecord();
void showRecord();
void deleteRecord();
void searchRecord();
```

```
void closeprogram();
void printTable();
```

(b) ClientController.c > ClientController.h

Se encarga de controlar totalmente la conexión entre servidor y cliente, por sus funciones pasan cada uno de los mensajes. Fue necesario crear funciones especializadas para enviar y recibir el archivo del servidor puesto que tienen un protocolo especial para poder validar varias situaciones, como identificar si el archivo está vacío, tal como se muestra en el diagrama de comunicaciones en la figura 1.

La primera línea está dedicada a permitir cambiar la IP con la que se desea realizar la conexión de manera sencilla.

Por otra parte, la variable TIME_TO_RECONNECT controla el tiempo de espera entre intentos de conexión, cuando éste no es exitoso y SIZE_CHUNK la cantidad de bytes que se enviarán por pasada en el envío de archivos.

```
#define IP_SERVER "127.0.0.1"

// ------ Don't touch any line below that ----- //

#include <stdio.h>
#include <netinet/in.h>

#define TIME_TO_RECONNECT 1
#define SIZE_CHUNK 1024

extern int socketClient;

char buffer[SIZE_CHUNK];
char fileName[100];

FILE* file;

struct sockaddr_in dummyAddr;
ssize_t bytesSent, bytesRead;

void requestConnection();
void closeConnection();

void sendMessage(void* message, size_t len);
void receiveMessage(void* message, size_t len);

char* receiveFile();
void sendFileUpdated(char *name, size_t len);
```

2. Servidor

(a) main.c > main.h

Está encargado de administrar las funcionalidades generales del software, todo lo relacionado con la interacción con el usuario (entradas y salidas) y de ejecutar las acciones correspondientes con base a estas interacciones. Además posee cinco utilidades usadas globalmente:

i. notifyRecordFound y notifySearchFinished son funciones utilizadas para comunicarle la existencia o ausencia de registros.
ii. buildName es una función que construye el nombre de un archivo, concatenando el número de registro con el nombre de la mascota.
iii. substring es una función que extrae una porción de una cadena específica.
iv. calcRecordNumber calcula la posición en que se encuentra un registro en el archivo que los contiene.

La variable FOLDER_NAME define el nombre de la carpeta en la cual se guardarán las historias clínicas.

```
#define HANDLE_ERROR(arg) perror(arg)
#define FOLDER_NAME "medicalrecords/"

extern long firstReg;
extern long lastReg;

struct DogType *dummyPet;

void initProcess();

void enterRecord();
void showRecord();
void deleteRecord();
void searchRecord();
void closeProgram();
void printTable();

void setFirstRecord(long first);
void setLastRecord(long last);

// Utilities

void notifyRecordFound(struct DogType *pet);
void notifySearchFinished();
void buildName(char* nm, int res, char* ans);
char* subString (const char* input, int offset, int len, char* dest);
long calcRecordNumber();
```

(b) ServerController.c > ServerController.h

Controla todas las acciones necesarias para realizar una comunicación exitosa con los clientes que deseen conectarse al software.

La variable MAX_CLIENTS define la cantidad máxima de clientes que pueden conectarse a la vez al servidor. La variable SIZE_CHUNK tiene el mismo objetivo descrito en el archivo análogo en el cliente.

Así mismo, las funciones para enviar y recibir archivos están especializadas para realizar una comunicación con base en el protocolo descrito en la figura 1 para el envío de archivos.

```
#define MAX_CLIENTS 32
#define SIZE_CHUNK 1024

extern int socketServer, socketClient;
extern int bufferInt;

char buffer[SIZE_CHUNK];
char fileName[100];
char ipDirection[INET_ADDRSTRLEN];

FILE *tempFile;

ssize_t bytesRead, bytesSent;
socklen_t addrlen;

struct sockaddr_in dummyAddr;
struct sockaddr_in* ipv4Addr;
struct in_addr ipAddr;

void initServer();
bool acceptConnection();
void closeConnection();
void closeServer();

void sendMessage(void* message, size_t len);
void receiveMessage(void *pointer, size_t len);

void sendFile(char* name, size_t len);
void receiveFile();

char *getClientIp();
```

(c) Tests.c > Tests.h

Precarga el software con la cantidad de animales definida por la variable ANIMALS con datos aleatorios, los nombres de cada uno de ellos son extraídos aleatoriamente desde el archivo con el nombre especificado en NAME_TEST_FILE.

```
#define NAME_TEST_FILE "names.txt"
#define ANIMALS 10000000

FILE *namesFile;
```

```
void fillWithRandomAnimals();
```

(d) Logger.c > Logger.h

Hace control de cada una de las operaciones realizadas por los clientes sobre el servidor y la guarda en LOGGER_FILE_NAME. Guarda la fecha de realización, la dirección del cliente que realiza la operación, el tipo de operación (definida por las variables INSERTION, READING, WRITING, DELETION) y los datos que se modificaron con esta operación.

```
#define LOGGER_FILE_NAME "serverDogs.dat"

#define INSERTION 0
#define READING 1
#define WRITING 2
#define DELETION 3

struct Log {
    char date[14];
    char ip[16];
    short operation;
    char input[50];
};

FILE *logger;
struct Log *dummyLog;

char tmp[50];

time_t t;
struct tm* tim;

size_t result;

void initLogger();
void registerOperation(char *ip, short operation, char *input,
                       size_t sizeInput);
void registerIntOperation(char *ip, short operation, int input);
void printLog(struct Log *log);
```

(e) HashTable.c > HashTable.h

Gestiona todas las operaciones que se realizan sobre la tabla hash implementada para guardar los registros en el software, ésto con el fin de realizar búsquedas óptimas. La tabla hash fue implementada con una estrategia de resolución de colisiones por listas encadenadas. Cada uno de los nodos, definidos por la estructura Node, tienen el valor del hash que les corresponde, la estructura del animal a guardar en esa posición, la dirección en el archivo del siguiente item en la hash y su posición actual.

Además, la variable HASH_SIZE define el tamaño de la tabla hash, se eligió un número primo de seis cifras. Primo puesto que disminuye la posibilidad de colisiones, y de seis cifras para intentar que a lo mucho, hayan 6 elementos por cada nodo en promedio con el archivo de prueba de diez millones de estructuras, asumiendo que las mismas se distribuirán uniformemente.

La función hash fue implementada de acuerdo al algoritmo *DJB2*, especial para hacer hash de strings http://www.cse.yorku.ca/~oz/hash.html.

También posee cuatro utilidades, dos de ellas para realizar debug de la tabla hash (printTable y printFirstNodes), una función para convertir cadenas a minúsculas y otra función para imprimir el registro de un animal.

Para la operación de eliminado en la tabla se usó un **algoritmo *perezoso (lazy)***, donde al borrar un elemento se marca como borrado, pero la tabla es reacomodada solo en el momento de realizar una búsqueda, con el fin de que el software tenga un borrado óptimo y en el caso de que se borren muchos registros seguidos, el servidor no se sature haciendo operaciones y deba hacer mucho procesamiento, lo hará sólo hasta que los datos sean solicitados.

```
#define HASH_SIZE 6000011

struct DogType {
    char name[32];
    char type[32];
    int age;
    char race[16];
    int height;
    double weight;
    bool sex;
};

struct Node {
    int key;
    struct DogType data;
    long nextHashItem;
    long position;
};

struct HashItem {
    long head;
};

struct HashItem* hashArray;
struct HashItem* dummyItem;

extern long previousItem;

struct Node* prevDummyNode;
struct Node* dummyNode;
```

```
struct Node* lastNode;

extern bool booleanBuffer;

void initHashTable(bool first);
unsigned long hash(char *str);
void insert(struct DogType *dogType);
void search(char* key);
void removeItem(struct Node* node);

// Utilities

void toLower(char* str);
void printDogType(struct DogType* dogType);
void printTable();
void printFirstNodes(long quantity);
```

(f) FileManager.c > FileManager.h

Se encarga de gestionar todo lo relacionado con la escritura del archivo donde se guardarán los datos ingresados por los clientes.

La variable FILE_NAME indica el nombre del archivo donde se guardará en binario toda la información registrada por el software.

```
#define FILE_NAME "dataDogs.dat"

FILE *file;

bool initFile();
void closeFile();
int deleteFile(char *name);

long calcNodePosition(long nodePos);
long calcHashItemPosition(long hashItemPos);

void updateKeyInNode(int newKey, long position);

void recordNode(struct Node* node, long position);
void recordNodeInCurrentPosition(struct Node* node);
bool getNodeByEntryOrder(struct Node* node, long position);
bool getNode(struct Node* node, long position);

void recordHashItem(struct HashItem* item, long position);
void recordHashItemInCurrentPosition(void* item);

void recordTable(struct HashItem* hashArray, int hashSize);
void getHashItem(struct HashItem* item, long position);

void resetFileIndicator();
void setFileIndicator(long position);
void setFileIndicatorAtEnd();
```

```
long getCurrentPosition();

void loadFirstRecord();
void loadLastRecord();
void recordFirstRecord();
void recordLastRecord();
```

Diagrama de comunicación entre el cliente y el servidor.

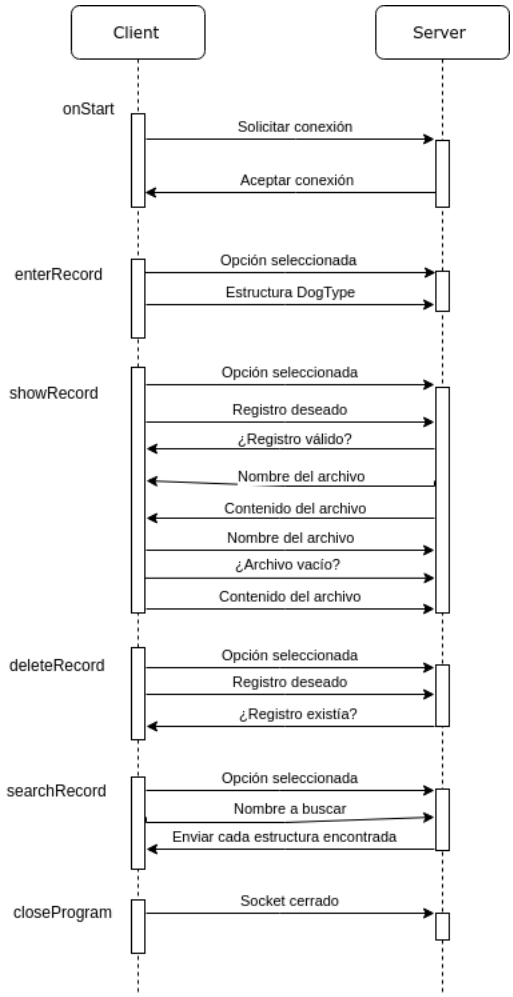

Figure 1: Diagrama de comunicación

Diagrama de bloques.

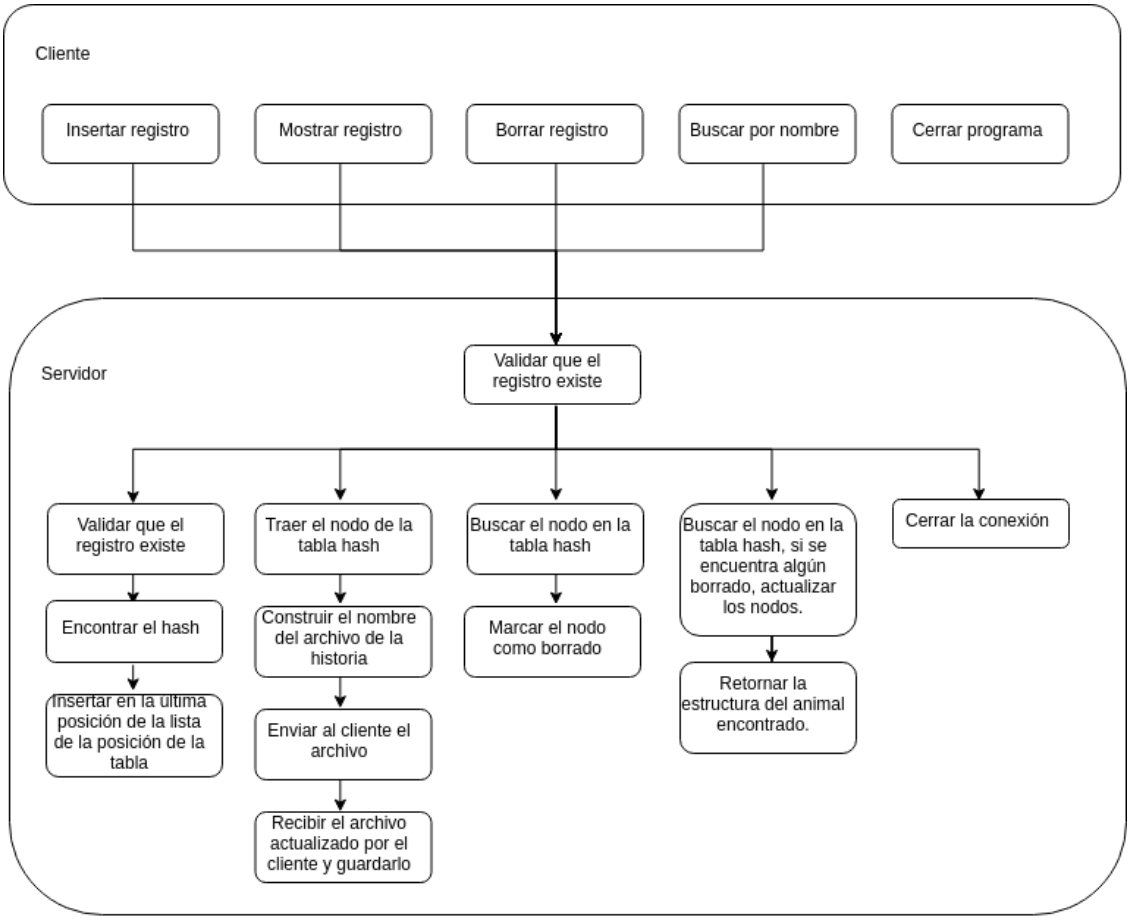

Figure 2: Diagrama de bloques.