

Easycab

Aarón Blasco Tremiño (74379762Q)

7 de noviembre de 2024

Índice

1	Introducción	3
1.1	¿Qué es Easycab?	3
1.2	Enfoque y estructura de carpetas	3
1.3	Herramientas utilizadas	4
1.4	Componentes del sistema	4
2	Implementación	6
2.1	Cabeceras comunes	6
2.1.1	Common	6
2.1.2	Estructuras de datos	11
2.1.3	Funciones comunes de Ncurses	13
2.2	EC_Customer	14
2.2.1	Parámetros	15
2.2.2	Aspectos relevantes	15
2.2.3	Ejemplos de ejecución	16
2.3	EC_SE	18
2.3.1	Parámetros	18
2.3.2	Aspectos relevantes	18
2.3.3	Ejemplos de ejecución	18
2.4	EC_DE	21
2.4.1	Parámetros	21
2.4.2	Aspectos relevantes	22
2.4.3	Ejemplos de ejecución	22
2.5	EC_Central	25
2.5.1	Parámetros	25
2.5.2	Aspectos relevantes	25
2.5.3	Ejemplos de ejecución	26
3	despliegue	29
3.1	Componentes dockerizados	29
3.2	Interfaz gráfica	30
3.3	Kafka	30
3.4	Base de datos	31

1 Introducción

1.1 ¿Qué es Easycab?

Este proyecto implementa un sistema distribuido que simula una empresa de taxis autónomos. En el sistema, diferentes clientes pueden pedir servicios a diferentes puntos de la ciudad preestablecidos.

1.2 Enfoque y estructura de carpetas

Este proyecto se ha desarrollado principalmente en C y utiliza *cmake* como herramienta de construcción. Para facilitar su portabilidad y acceso, todo el proyecto está disponible en un [repositorio público en GitHub](#). A continuación, se presenta una breve descripción de la estructura de carpetas:

- La carpeta **src** contiene todo el código fuente del proyecto, incluyendo los archivos de cabecera (*.h*) y sus implementaciones (*.c*). También se encuentra el script [initialize.sql](#), que se utiliza para inicializar la base de datos. Además, esta carpeta incluye el symlink [compile_commands.json](#), que apunta a un archivo generado automáticamente al compilar con *cmake*. Este archivo es útil únicamente para el desarrollo, ya que facilita las sugerencias de código en *clangd*.
- La carpeta **res** contiene los archivos de texto necesarios para que *EC_Customer* y *EC_Central* lean los servicios a solicitar y las ubicaciones del mapa, respectivamente. Estos archivos se emplearon durante el desarrollo para realizar pruebas del sistema.
- En **docs** se encuentra la documentación del proyecto en formato LaTeX, tanto el código fuente como el archivo PDF generado.
- La carpeta **visual studio** alberga el subproyecto de Visual Studio que sirve como portabilidad para la interfaz gráfica en Windows. Esto se debe a que la dockerización de la interfaz gráfica resulta algo compleja, por lo que se optó por migrarla a un entorno nativo de Windows para facilitar su uso. Alternativamente, se puede compilar la versión con *cmake* descomentando el contenido correspondiente en el archivo [CMakeLists.txt](#) y asegurándose de contar con la librería *raylib*.

- La carpeta **.vscode** contiene únicamente el archivo de configuración [*settings.json*](#) para el editor *VSCo*de. Este archivo configura la extensión *Material Icon Theme*, de carácter estético.
- Finalmente, en la carpeta **raíz** se ubican los archivos de configuración del proyecto y del repositorio, como [*.gitignore*](#) y [*CMake-Lists.txt*](#).

1.3 Herramientas utilizadas

- **Visual Studio Code** y **Visual Studio 2019** como editores de código.
- **CMake** como sistema de compilación.
- **Git** como sistema de control de versiones.
- **Docker** para facilitar el despliegue.
- **Raylib** como librería para la interfaz gráfica.
- **Mysql Server** como sistema de gestión de base de datos.
- La librería **Ncurses** para implementar la interfaz gráfica en terminal.
- Las librerías **rdkafka**, **mysqlclient**, **uuid** y **glib** para el desarrollo general del proyecto

1.4 Componentes del sistema

- **EC_Central**: Componente angular del sistema, se encarga de la comunicación con los clientes y los taxis. Consiste en 3 módulos (sin incluir el entrypoint *EC_Central.c*):
 - **kafka_module**: Módulo que se encarga de la comunicación con los clientes y los taxis a través de Kafka.
 - **ncurses_gui**: Módulo que se encarga de la interfaz gráfica (en terminal) de la central.
 - **socket_module**: Módulo que se encarga de la autenticación de los taxis via sockets.
- **EC_DE**: Representa el software central de un taxi autónomo. Recibe instrucciones de la central y se mueve por el mapa independiente

e asíncronamente del resto de taxis. Este componente también incluye el módulo **EC_DE_ncurses_gui** que se encarga de la interfaz gráfica (en terminal) del taxi.

- **EC_SE**: Representa el sensor de un taxi autónomo. Este componente es dependiente de EC_DE, con el cual se comunica a través de sockets para informar sobre la situación del taxi.
- **EC_Customer**: Representa a un cliente. Su única función en el sistema es pedir servicios a la central continuamente.
- **Interfaz gráfica (gui)**: Lee las comunicaciones en kafka y muestra la información en forma de ventana. Como se ha comentado anteriormente, este componente tiene dos implementaciones en el proyecto, una en **src**, basada en un entorno de Linux; y otra en **visual studio**, basada en Windows. Puesto que su implementación en Linux se ha probado menos exhaustivamente, se recomienda el uso de la implementación en Windows.

2 Implementación

2.1 Cabeceras comunes

En el desarrollo de este proyecto he hecho uso de varias cabeceras comunes entre varios de los componentes. Estas cabeceras contienen funciones y estructuras de datos que son utilizadas en el resto del proyecto y que, en muchos casos, es importante que sean iguales en todos los componentes.

2.1.1 Common

Contiene el grueso de las definiciones y funciones comunes en el proyecto. Repasémoslas brevemente:

- Al comienzo de la cabecera podemos hayar algunas definiciones de constantes y enumeraciones que, puesto que son simples y están debidamente comentadas, ignoraré aquí. No obstante, recomiendo echarles un vistazo en el [código fuente](#).

El único que me parece merecedor de mención es el enum **SUBJECT**. La mayoría de las comunicaciones en este proyecto se definen por un *motivo* (el cómo se usa, se irá viendo a lo largo de la memoria). De esta manera, podemos categorizar las comunicaciones de manera clara y atenderlas ordenadamente. De nuevo, como es un enum largo (así lo es la cantidad de razones por las que los diferentes componentes pueden querer comunicarse entre ellos) y sus valores son razonablemente autoexplicativos, evitaré ensuciar el documento con el código.

- Continuamos con las estructuras de datos que usarán ampliamente durante el proyecto:

```
// Position in the map
typedef struct {
    int x, y;
} Coordinate;

// Represents a socket address
typedef struct {
    char ip[20];
    int port;
} Address;
```

Estas estructuras son muy simples y, como se puede ver, no requieren comentarios. Simplemente organizan los datos para una mayor legibilidad.

```
// Represents an entity relevant for the progress of the
// system
typedef struct {
    ENTITY_TYPE type;           // Customer, taxi or location
    USER_STATUS status;        // Unset if the entity is a
    location
    Coordinate coord;          // Position in the map
    int id;                     // Represents a char if the entity
    is not a taxi
    char obj;                   // Customer's destination or taxi's
    service
    bool carryingCustomer;      // Only for taxis
} Entity;
```

Este struct representa o bien una localización, un cliente o un taxi y contiene toda la información relevante para que las interfaces gráficas puedan representarlos adecuadamente. Nótese que no todos los valores del struct son usados para todos los tipos de entidad. Por ejemplo, *obj* no tiene sentido en caso de una localización y *carryingCustomer* sólo es relevante para los taxis.

La razón de que exista un único struct para representar todos los tipos de entidad es la organización y poder normalizar las comunicaciones. De esta manera, tanto los remitentes como los destinatarios de la información siempre manejan la misma cantidad de datos y podemos evitar los accesos fuera de rango tan característicos de C.

```
// Represents a message sent by a user to the central
typedef struct {
    SUBJECT subject;           // Purpose of the message
    Coordinate coord;          // Position of the author (may
    be unused)
    int id;                     // Identification of the
    author
    char data[UUID_LENGTH];     // Extra data, depending on
    the subject
    char session[UUID_LENGTH]; // Session id of the system,
    restarted each time the system restarts.
    // Its position as last in
    the struct is relevant, don't
    change it
} Request;
```

```

// Represents a message sent by the central to a user
typedef struct {
    SUBJECT subject;           // Purpose of the message
    int map[MAP_SIZE];         // 100 (taxis) + 30 (customers
                               // + 30 (locations)
                               // Represents the state of the
                               // map
    char id;                   // Identification of the
                               // addressee
    char data[UUID_LENGTH];    // Extra data, depending on
                               // the subject
    char session[UUID_LENGTH]; // Session id of the system,
                               // restarted each time the system restarts.
                               // Its position as last in
                               // the struct is relevant, don
                               // 't change it
} Response;

```

Por el mismo motivo que antes, todas las comunicaciones tienen la misma estructura y, por tanto, la misma cantidad de datos. Así pues, Request representará un mensaje enviado a través de kafka hacia la central y Response uno enviado desde la central.

Como se explicó anteriormente, tanto las requests como las responses se caracterizan por un *subject* o *motivo*. También contienen, como era de esperar, el id del remitente/destinatario y un campo para añadir datos adicionales si es necesario.

Es interesante mencionar que el mapa se envía como un array de enteros porque cada uno representa un **Entity** serializado. Aprovechamos el hecho de que la mayoría de sus campos no necesitan mucho del espacio que ocupan (como los valores **Coordinate**, ue a lo largo del programa solo adoptarán valores del 0 al 20) y logramos reducir el espacio del mapa y el peso de las comunicaciones considerablemente.

Por último, el campo **session** juega un papel importante. Cada vez que el sistema se reinicia, crearemos una nueva sesión con un id único. Así, todos los componentes podrán descartar mensajes que no sean relevantes y puedan generar líos entre ellos. Por ejemplo, si se nos ha olvidado cortar la ejecución de un taxi antes de reiniciar la bbdd, sin la ayuda de **session** la central podría estar tratando de mover un taxi que realmente no existe en la base de datos. Si bien no tiene por qué ser necesario si el sistema es robusto, no sobra tomar precauciones.

- Gestión de logs:

```
void log_handler(const gchar *log_domain, GLogLevelFlags
    log_level, const gchar *message,
    gpointer user_data);
```

Esta función redefine el formato de los mensajes de la librería **glib**. Así pues, podemos usar su API:

```
g_log_set_handler(NULL, G_LOG_LEVEL_DEBUG, log_handler,
    NULL);

g_message("This is a message");
g_warning("This is a warning");
g_error("This is an error");
g_critical("This is a critical error");
g_debug("This is a debug message");
```

Con un formato más acorde al estilo del proyecto (y al mío personal).

- Serialización de entidades:

```
int serializeEntity(Entity *entity);
void deserializeEntity(Entity *dest, int entity);
```

Como se comentó anteriormente, ambas funciones aprovechan el hecho de que los campos de **Entity** no necesitan todo el espacio que ocupan para almacenar sus datos en el tamaño de 4 bytes.

```

[22:29:38.594] Message: This is a message
[22:29:38.594] ** WARNING **: This is a warning
[22:29:38.594] Debug: This is a debug message
[22:29:38.594] ** CRITICAL **: This is a critical error
[22:29:38.594] ** ERROR **: This is an error

```

Figura 1: Ejemplo de ejecución de mensajes a través de la API de glib.

- Funciones comunes sobre sockets:

```

int openSocket(int port);
int connectToServer(Address *server);

```

Estas funciones hacen lo que prometen y nada más: abrir un socket servidor o conectarse a uno, devolviendo en ambos casos el descriptor de socket correspondiente. Además, se encargan de manejar errores y asignar una rutina de cierre de socket al cierre del programa.

- Funciones comunes sobre kafka:

```

rd_kafka_t *createKafkaUser(Address *server,
    rd_kafka_type_t type, char *id);
void subscribeToTopics(rd_kafka_t **consumer, const char **
    topics, int topicsCount);
void sendEvent(rd_kafka_t *producer, const char *topic,
    void *value, size_t valueSize);
rd_kafka_message_t *poll_wrapper(rd_kafka_t *rk, int
    timeout_ms);

```

Las dos primeras funciones son similares a las del apartado anterior, en el sentido de que inicia la comunicación pero en este caso, con kafka. Es importante mencionar que es posible usar **createKafkaAgent** para crear un producer o un consumer, únicamente cambiando el segundo parámetro. La segunda función, sin embargo, solo tiene sentido para un consumer.

Las últimas dos funciones son simplemente envoltorios de las funciones para enviar o recibir mensajes con la finalidad de mejorar la legibilidad del código. La única funcionalidad un tanto especial es la de **poll_wrapper** que solo devuelve los mensajes enviados después de la marca de tiempo determinada (relativa al momento actual). Esto se puede hacer porque la key de todos los mensajes enviados es la fecha de envío en formato *UNIX*

Algo que quizá sí que es merecedor de mención es la configuración con que se inician los consumers/producers:

```

SET_CONFIG(conf, "log_level", "0", errstr);
SET_CONFIG(conf, "bootstrap.servers", serverAddressStr,
    errstr);
SET_CONFIG(conf, "client.id", id, errstr);
SET_CONFIG(conf, "acks", "all", errstr);
SET_CONFIG(conf, "auto.offset.reset", "earliest", errstr);
if (type == RD_KAFKA_CONSUMER) {
    SET_CONFIG(conf, "group.id", id, errstr);
    SET_CONFIG(conf, "session.timeout.ms", "6000", errstr);
    SET_CONFIG(conf, "max.poll.interval.ms", "6000", errstr);
}

```

Nótese que las variables `id` y `serverAddressStr` son las pasadas por parámetro (y una traducción de `Address` a `string` en caso de la segunda). mientras que la primera configuración puede no ser muy relevante para la ejecución del sistema, el resto sí lo son. Su intención es asegurar que las comunicaciones se realizan de un modo seguro (*acks=all*) y que no se pierdan mensajes (*auto.offset.reset=earliest*). No obstante, esta última no significa que cada vez que se cree un consumer empiece realmente a leer desde el principio del topic. Puesto que en la mayoría de casos se crean siempre con la misma `id`, esta configuración solo significa que retomarán la lectura desde donde lo dejaron. Con respecto a las dos últimas configuraciones, tienen los valores mínimos para que los consumers/producers expiren tan pronto como sea posible si no se les ha cerrado explícitamente mediante código.

2.1.2 Estructuras de datos

Este archivo contiene un par de estructuras de datos simples que yo mismo he desarrollado:

- **TABLE:** Representa literalmente una tabla con la intención de ser impresa en una interfaz gráfica. Puesto que no es relevante para el funcionamiento del sistema, no profundizaremos en ella. Esta es su API:

```

/// @brief Initializes a table
///
/// @param table Table to be initialized
/// @param start Coordinate where the table will start (top
    left corner)
/// @param title Title of the table
/// @param headers Array of strings that will be used as
    headers

```

```

/// @param headers_num Number of headers
/// @param err_color Color in which the rows will be
    printed if set as erroneous
void initTable(Table *table, Coordinate start, char *title,
    char **headers, int headers_num,
    int err_color);

/// @brief Prints a table in a ncurses window
///
/// @param table Table to be printed
/// @param win Window in which the table will be printed
void printTable(Table *table, WINDOW *win);

/// @brief Adds a row to a table
///
/// @param table Table to which the row will be added
/// @param row Array of strings that will be used as row's
    content
/// @param status Whether the row is erroneous or not (if
    erroneous, the row will be printed in
/// a different color)
void addRow(Table *table, const char **row, bool status);

/// @brief Deletes all the rows from a table
///
/// @param table Table to be emptied
void emptyTable(Table *table);

/// @brief Disposes the memory allocated for a table
///
/// @param table Table to be destroyed
void destroyTable(Table *table);

```

- **QUEUE:** Representa una cola FIFO, otra vez usada únicamente en las interfaces gráficas de terminal. No hay mucho que explicar sobre ella, esta es su API:

```

/// @brief Returns a new queue
///
/// @return Queue* Empty queue
Queue *newQueue();

/// @brief Initializes a queue iterator
///
/// @param queue Queue to be iterated
/// @return QueueIterator Iterator to be used
QueueIterator initQueueIterator(Queue *queue);

```

```

/// @brief Gets the next element in the queue
///
/// @param queue Queue to be iterated
/// @param it Iterator to be used
/// @param element Pointer to the element to be filled
/// @return true There is an element to be read
/// @return false There aren't anymore elements to be read
bool getNext(Queue *queue, QueueIterator *it, void *element
);

/// @brief Pops the first element in the queue (if queue is
///         not empty)
///
/// @param queue Queue to be iterated
/// @param element Pointer to the element to be filled
/// @return true There is an element to be popped
/// @return false The queue is empty
bool dequeue(Queue *queue, void *element);

/// @brief Pushes an element in the queue. If the queue is
///         full, the oldest element will be
///         popped
///
/// @param queue Queue to be iterated
/// @param element Pointer to the element to be pushed
void enqueue(Queue *queue, void *element);

```

Nótese que pueden faltar algunas funciones características de colas. Esto es porque solo se han definido las funciones que se necesitan para el funcionamiento del sistema.

2.1.3 Funciones comunes de Ncurses

Al igual que el `data_structures.h`, este archivo contiene funciones exclusivamente relevantes para las interfaces gráficas de terminal (implementadas con la librería **ncurses**). Continuando con la tónica de no profundizar en lo relativo con las interfaces gráficas, he aquí la API de su cabecera:

```

/// @brief Initializes a 256-based rgb color in the ncurses
///         palette
///
/// @param color Color to be initialized
/// @param r Red component
/// @param g Green component
/// @param b Blue component

```

```

void init_color_rgb(int color, int r, int g, int b);

/// @brief Prepares the ncurses GUI for the use of colors (e.g.
        initializing the palette)
void start_color_wrapper();

/// @brief Handles the logging of the ncurses GUI through the
        glib API
void ncurses_log_handler(const gchar *log_domain, GLogLevelFlags
        log_level, const gchar *message,
                        gpointer user_data);

/// @brief Prints a pop-up message informing the user that the
        program is exiting
///
/// @param extra Extra message to be printed, if NULL, no more
        than the default message will be
/// printed
void printFinishPopUp(char *extra);

/// @brief Finds a character in a string
///
/// @param text String to be searched
/// @param character Character to be found
/// @return int Position of the character in the string or -1 if
        not found
int find(const char text[], char character);

/// @brief Splits a string into lines of a given length without
        breaking words
///
/// @param dest Array of strings where the lines will be stored
/// @param text String to be split
/// @param lineLength Maximum length of each line
/// @return int Number of lines
int split(void *dest, const char text[], int lineLength);

```

2.2 EC_Customer

Comencemos con “la chicha” del proyecto. El componente EC_Customer es probablemente el más pequeño y sencillo de los componentes. Su flujo de ejecución es el siguiente:

1. Se comprueba que los parámetros de entrada sean correctos y se almacenan sus valores en variables en caso de ser así.
2. Se lee el archivo indicado a través de los parámetros de entrada.

3. Se lleva a cabo un intercambio de mensajes inicial con el servidor central. Esto sirve tanto para la central como para el cliente, para poder asegurar que no estamos mandando mensajes a nadie y no hacer perder tiempo innecesario al usuario.
4. Uno a uno, se piden los servicios a la central (esperando 4s entre cada uno) y comunicando por pantalla las respuestas de la central como sea pertinente.

2.2.1 Parámetros

Este programa recibirá 5 parámetros de entrada:

1. **Dirección del broker de Kafka:** Dirección del broker de Kafka a utilizar para la autenticación de los taxis. ¹
2. **Id:** Identificador único del cliente, los valores admitidos son de la 'a' a la 'z'.
3. **Nombre del archivo que contiene los servicios a pedir:** Nombre del archivo que contiene los servicios a pedir. No se comprobará que el archivo exista. Sin embargo, dado que la lectura del mismo se realiza justo después de comprobar los argumentos de entrada, prácticamente es como si se hiciese (si no existe, se imprimirá un error y se finalizará el programa controladamente).
4. **Coordenadas de inicio:** Tanto el 4º como el 5º parámetro están dedicados a las coordenadas iniciales del cliente en el mapa. Solo se admitirán valores entre el 1 y el 20. ²

2.2.2 Aspectos relevantes

Justo después de conectarse con el servidor central, el programa asignará un hilo secundario a mandar constantemente pings a la central para indicar que el cliente está activo.

Este es el único componente en el que se hace uso de dos consumidores en el mismo hilo de ejecución. Esto se debe a que se necesitan consumidores con dos ids diferentes dependiendo del punto del programa:

¹Las direcciones se esperarán en formato *ip:puerto*.

²Es interesante mencionar que, mientras que de caras afuera el mapa tiene coordenadas 1-20, entre bambalinas, el código fuente siempre usará coordenadas 0-19.

- **Conexión con el servidor central:** Se usa un consumidor con una id única, para asegurarnos de que la conversación entre cliente y central es de uno a uno y no se mezclan mensajes de otros EC_Customer con la misma id.
- **Petición de servicios:** Una vez se ha conectado con el servidor central (cosa que solo será posible si no existe activo ningún otro cliente con la misma id que pueda interferir en las comunicaciones), se crea un consumidor con la id igual a la del cliente (pasada por parámetro). Esto es útil porque comenzará a leer por donde lo dejó el último cliente con la misma id e ignorará, por tanto, los mensajes dirigidos a esta id que ya hayan sido leídos, evitando ofrecer al usuario mensajes irrelevantes.

Por otro lado, cabe recalcar que el formato del documento a leer es un archivo de texto con un único caracter por línea, indicando el servicio a pedir. Realmente, no pasa nada si no se cumplen las condiciones de este formato, después de todo se leerá el primer caracter aunque sea el salto de línea y si no existe la localización correspondiente, la central sencillamente denegará el servicio y el cliente terminará controladamente el programa.

Por último, me gustaría mencionar que meramente por capricho y por añadir un poco de diversión, el cliente mostrará un mensaje aleatorio cada vez que se complete su servicio como excusa mientras espera a pedir el siguiente.

2.2.3 Ejemplos de ejecución

```

~/universidad/easycab P main 123 ?5 cmake --build build && ./build/EC_Customer localhost:9092 b res/cli.txt 5 5
[ 38%] Built target EC_Central
[ 66%] Built target EC_DE
[ 85%] Built target EC_SE
[100%] Built target EC_Customer
[23:34:30.398] ** ERROR **: Couldn't open file res/cli.txt

~/universidad/easycab P main 123 ?5 cmake --build build && ./build/EC_Customer localhost:9092 b res/cli.txt
[ 38%] Built target EC_Central
[ 66%] Built target EC_DE
[ 85%] Built target EC_SE
[100%] Built target EC_Customer
[23:34:33.337] ** ERROR **: Usage: ./build/EC_Customer <Kafka IP:Port> <ID> <File name> <x> <y>
~/universidad/easycab P main 123 ?5

```

Figura 2: Ejemplos de ejecución erróneos.


```
~/universidad/easycab  P main 123 75  cmake --build build && ./build/EC_Customer localhost:9092 b res/client1.txt 5 5
[ 38%] Built target EC_Central
[ 66%] Built target EC_DE
[ 85%] Built target EC_SE
[100%] Built target EC_Customer
[23:32:41.776] Message: Loading...
[23:32:42.776] Message: Sending request for connection
[23:32:43.783] Message: Waiting for confirmation
[23:32:45.889] Message: Central accepted the connection
[23:32:45.889] Message: Asking for service to go to Z...
[23:32:45.989] Message: Service accepted. Taxi 2 is coming for you
[23:32:50.357] Message: Taxi 2 has picked you up
[23:32:57.354] Message: We've arrived to Z
[23:32:57.354] Message: Shopping at the mall...
[23:33:01.354] Message: Asking for service to go to B...
[23:33:01.455] Message: Service accepted. Taxi 2 is coming for you
[23:33:02.354] Message: Taxi 2 has picked you up
[23:33:09.354] Message: We've arrived to B
[23:33:09.354] Message: Visiting my grandma...
```

Figura 3: Ejemplo de ejecución satisfactorio.

2.3 EC_SE

Este es otro componente relativamente sencillo, si no tenemos en cuenta su interfaz gráfica. Consta de dos hilos de ejecución:

1. **Principal:** Se encargará de comprobar los argumentos de entrada y de inicializar el sistema. Una vez todo esté listo (incluida la conexión con el taxi), creará el segundo hilo y se quedará a cargo de la interfaz gráfica.
2. **Comunicación con el taxi:** Se encargará de la comunicación con el taxi: A cada segundo enviará un mensaje a través del socket informando de si se puede o no mover y por qué (se entenderá más adelante), acto seguido leerá un mensaje del taxi indicando si este se encuentra parado o no (por razones ajenas al sensor, como si no tiene ningún servicio que atender).

Ambos hilos se comunican a través de variables globales y herramientas de control de hilos (mutexes en este caso).

2.3.1 Parámetros

Este programa recibirá solo un parámetro de entrada: la dirección del socket de escucha del taxi.

2.3.2 Aspectos relevantes

Este componente es responsable de que el sistema sea relativamente síncrono. Aunque no afecta al progreso del sistema y este podría funcionar completamente asíncrono, las comunicaciones del sensor con el taxi están sincronizadas con los segundos reales. Esto se ha añadido meramente como curiosidad.

En cada ocasión que el usuario indique al sensor que el taxi debe pararse, el hilo principal subirá un contador a la cantidad de segundos que debe esperar el taxi. Por otro lado, aprovechando que el segundo hilo realiza una iteración cada segundo, será este el encargado de ir reduciendo este contado hasta llegar a 0 de nuevo.

2.3.3 Ejemplos de ejecución

Las imágenes siguientes conforman un único proceso de ejecución del programa:

Taxi is not moving.
Please wait...

Options
[>] Minor inconvenience
[] Inconvenience
[] Major inconvenience
[] Fatal error
[] Exit

Stuck behind a slow-moving truck on a narrow road...

14 seconds

Accidentally bumped into a confused bird. The poor taxi
is traumatized and won't move!

[RECOVER] [TERMINATE] [EXIT]

Goodbye!
(Press any key to exit...)

2.4 EC_DE

Siguiendo en orden de complejidad, llega el turno de la digital engine. Este componente separa su ejecución en 4 hilos justo después de comprobar que los parámetros de entrada son correctos:

- **Pings:** Este hilo es prácticamente idéntico a su homónimo en el componente EC_Customer. Se encargará de mandar pings a la central cada segundo para indicar que el cliente está activo.
- **Hilo de movimiento:** Este hilo entrará en un bucle infinito en el cual espera una señal de otro hilo para avanzar la posición del taxi hacia el destino y volver a esperar una señal.
- **Conexión con el sensor:** Se encargará de esperar la comunicación del sensor del taxi e indicar al hilo de movimiento cuando se puede mover. Se define que el taxi no se puede mover si la señal recibida por el sensor lo indica o si el hilo responsable la conexión con la central lo dictamina de esa manera (puede ser porque no haya servicio que atender o porque se haya ordenado al taxi que se detenga).
- **Conexión con la central:** Estará a la espera de mensajes de la central. Comunicará lo relevante al resto de hilos a través de variables globales. Así pues, este hilo es el que modifica las variables como objetivo (al que se moverá el hilo de movimiento) o *canMove*, que indica al hilo de conexión con el sensor si puede enviar la señal de movimiento o no.

2.4.1 Parámetros

Este programa recibirá 4 parámetros de entrada:

1. **Dirección de la central:** Ip del equipo donde se ha desplegado la central y el puerto de escucha que se ha configurado.
2. **Dirección del broker de Kafka:** Dirección del broker de Kafka a utilizar para la autenticación de los taxis.
3. **Puerto de escucha del sensor:** Puerto por el que se escuchará el sensor del taxi.
4. **Id:** Identificador único del taxi, los valores admitidos son del 0 al 99.

2.4.2 Aspectos relevantes

Este componente es el que más hilos contiene y más uso hace de herramientas de control de hilos:

- **mutexes:** Se usan para sincronizar el acceso a las variables globales entre los distintos hilos.
- **condiciones:** Son las que usa el hilo de conexión con el sensor para indicar al hilo de movimiento cuando puede o no moverse. Además, se utiliza un juego de condiciones para sincronizar los hilos al comienzo de la ejecución: mientras que el hilo principal lleva a cabo el proceso de autenticación con el servidor central, el hilo de conexión con el sensor comienza a abrir el socket y preparar la escucha del sensor. Ambos hilos se informan mutuamente para indicar cuando pueden avanzar en sus tareas (por ejemplo, no se comenzará la autenticación hasta que se haya comprobado que se puede abrir el socket sin errores).

La autenticación con el servidor central (a través de sockets) utiliza el formato de tramas sugerido por los profesores: cada comunicación que no es atómica (ACK, NACK, ENQ, etc) contará con un primer carácter STX al comienzo del buffer, un carácter ETX en la penúltima posición del mismo y un carácter LRC al final del mensaje. Esto se hace para asegurar que los mensajes sean transmitidos correctamente.

2.4.3 Ejemplos de ejecución

[X] Logs
[] Table
[] Exit

```
[23:55:12.235] Message: Moving to [2, 3]
[23:55:13.233] Message: Moving to [3, 4]
[23:55:14.236] Message: Moving to [4, 5]
[23:55:15.334] ** WARNING **: Cannot move: Accidentally bumped into a confused bird. The poor taxi is traumatized and won't move!
[23:56:14.724] ** WARNING **: Error reading from socket
[23:56:14.724] ** WARNING **: Sensor disconnected
[23:56:14.724] Message: Waiting connection from sensor...
[24:18:12.353] Message: Sensor connected
[24:18:12.453] Message: Moving to [5, 6]
[24:18:12.553] Message: Destination reached. Stopping...
[24:18:16.791] Message: Central ordered to move to [5, 6]
[24:18:17.237] Message: Moving to [5, 6]
[24:18:17.337] Message: Destination reached. Stopping...
[24:18:17.791] Message: Central ordered to move to [3, 16]
[24:18:18.235] Message: Moving to [4, 5]
[24:18:19.233] Message: Moving to [3, 4]
[24:18:20.233] Message: Moving to [3, 3]
[24:18:21.237] Message: Moving to [3, 2]
[24:18:22.235] Message: Moving to [3, 1]
[24:18:23.234] Message: Moving to [3, 20]
[24:18:24.234] Message: Moving to [3, 19]
[24:18:25.232] Message: Moving to [3, 18]
[24:18:26.237] Message: Moving to [3, 17]
[24:18:27.236] Message: Moving to [3, 16]
[24:18:27.336] Message: Destination reached. Stopping...
[24:18:31.478] Message: Central ordered to move to [3, 16]
[24:18:32.237] Message: Moving to [3, 16]
[24:18:32.337] Message: Destination reached. Stopping...
[24:18:32.478] Message: Central ordered to move to [3, 4]
[24:18:33.237] Message: Moving to [3, 17]
[24:18:34.235] Message: Moving to [3, 18]
[24:18:35.234] Message: Moving to [3, 19]
[24:18:36.237] Message: Moving to [3, 20]
[24:18:37.235] Message: Moving to [3, 1]
[24:18:38.234] Message: Moving to [3, 2]
[24:18:39.237] Message: Moving to [3, 3]
[24:18:40.236] Message: Moving to [3, 4]
[24:18:40.336] Message: Destination reached. Stopping...
```

You can paste the image from the clipboard.

[] Logs
[X] Table
[] Exit

Position	[03, 04]
Going towards	-
Service	-
Taxi status	Stopped
Reason	-
Sensor status	Connected
Last order	Go to [03, 04]
Last order status	Completed

2.5 EC_Central

Este último componente es el encargado del control de todo el sistema. Nada más comenzar, se separará en tres procesos:

- **Interfaz gráfica:** Se encargará de la comunicación con el usuario y de la presentación de los mensajes del sistema.
- **Socket:** Se encargará de escuchar peticiones de autenticación de los taxis a través de los sockets y de atenderlas.
- **Kafka:** Se encargará de escuchar peticiones a través de kafka y de atenderlas. Todo lo necesario será guardado en la base de datos.

2.5.1 Parámetros

Este programa recibirá tres parámetros de entrada:

1. **Puerto de escucha:** Puerto a través del cual se escucharán las conexiones de los taxis para su autenticación.
2. **Dirección del broker de Kafka:** Dirección del broker de Kafka a utilizar para la comunicación con el resto de componentes.
3. **Dirección de la base de datos:** Dirección de servidor de gestión de la base de datos a utilizar durante la ejecución.

2.5.2 Aspectos relevantes

Dentro del módulo de kafka, se creará un hilo separado que irá comprobando continuamente si hay clientes o taxis que han parado de enviar pings. Si los hay, se notificará al hilo principal a través de un mensaje de kafka, como si de otro componente separado se tratase.

Las localizaciones se leerán de un archivo "locations.csv" en la carpeta "res" del proyecto. Si no existe este archivo y se desea usar otra ruta, se debe modificar la variable de entorno *LOCATIONS_PATH*. El formato esperado es el siguiente:

```
"A", "1", "1"  
"B", "2", "2"  
...
```

La central atenderá a la variable de entorno *RESET_DB* que indicará si se debe reiniciar la sesión y la bbdd (true) o no (false). En realidad, solo

comprobará si está definido y su valor es *true*, en caso contrario dará por hecho que su valor es *false*.

El módulo de kafka consta de un loop infinito en el que escucha una petición de kafka y la atiende. Casi cada posible motivo tiene su propia función que sigue la siguiente estructura:

- **Llamar al PROCEDURE de la base de datos correspondiente** (definidos en **initialize.sql**)
- **Comprobar si la llamada se ha efectuado correctamente**
- **Informar de los resultados al usuario**
- **Comunicar los resultados al cliente o taxi que sea pertinente**

2.5.3 Ejemplos de ejecución

```
VIEWS
[X] Logs
[ ] Table
(Press tab to change view)

ACTIONS
> Go to
  Return to base
  Stop
  Continue

[24:34:04.150] Message: 6 locations read and stored successfully
[24:34:05.159] Message: Sent initial map to responses topic
[24:34:09.649] Message: New taxi registered. Updating the map...
[24:34:09.751] Message: There aren't any customers in queue
[24:34:12.641] Message: Taxi 2 can move (again)
[24:34:17.161] Message: Inserted customer 'b'
[24:34:19.277] Message: Service accepted. Taxi 2 assigned to customer 'b'
[24:34:19.377] Message: Ordering taxi 2 to go to [5, 5]
[24:34:20.748] Message: Taxi 2 moved to [2, 2]
[24:34:21.245] Message: Taxi 2 moved to [3, 3]
[24:34:22.241] Message: Taxi 2 moved to [4, 4]
[24:34:23.242] Message: Taxi 2 moved to [5, 5]
[24:34:23.345] Message: Customer 'b' picked up by taxi 2. They are now going towards location 2
[24:34:24.243] Message: Taxi 2 moved to [4, 4]

[24:34:04.138] Message: Socket bound
[24:34:04.138] Message: Listening on port 2401
[24:34:09.638] Message: Processing authentication request 0
[24:34:09.641] Message: [request 0] Assigned ID 2
[24:34:09.743] Message: Updated map
[24:34:09.743] Message: [request 0] Closing connection

(Press 'q' to exit)
```

VIEWS
[] Logs
[X] Table

ACTIONS
> Go to
Return to base
Stop
Continue

(Press 'q' to exit)

Locations	
ID	Coordinate
A	[03, 04]
B	[05, 06]
F	[01, 02]
H	[03, 16]
X	[13, 02]
Z	[18, 19]

Customers			
ID	Coordinate	Destination	Status
b	[18, 19]	^?	Doing errands

Taxis			
ID	Coordinate	Service	Status
02	[18, 19]	^?	Stopped

```

VIEWS
[X] Logs
[ ] Table

ACTIONS
  Go to
  >> Return to base
  Stop
  Continue

OPTIONS
  > Taxi: 12
  Execute
(Press 'b' to cancel)

[24:34:23.345] Message: Customer 'b' picked up by taxi 2. They are now going towards location Z
[24:34:24.243] Message: Taxi 2 moved to [4, 4]
[24:34:25.244] Message: Taxi 2 moved to [3, 3]
[24:34:26.240] Message: Taxi 2 moved to [2, 2]
[24:34:27.241] Message: Taxi 2 moved to [1, 1]
[24:34:28.244] Message: Taxi 2 moved to [20, 20]
[24:34:29.241] Message: Taxi 2 moved to [19, 19]
[24:34:30.242] Message: Taxi 2 moved to [18, 19]
[24:34:30.348] Message: Customer 'b' service has been completed. Taxi 2 left the customer on the location Z [18, 19] and is now available
[24:34:30.550] Message: There aren't any customers in queue
[24:34:34.363] Message: Service accepted. Taxi 2 assigned to customer 'b'
[24:34:34.464] Message: Ordering taxi 2 to go to [18, 19]
[24:34:35.239] Message: Taxi 2 moved to [18, 19]
[24:34:35.342] Message: Customer 'b' picked up by taxi 2. They are now going towards location B
[24:34:36.242] Message: Taxi 2 moved to [19, 20]
[24:34:37.244] Message: Taxi 2 moved to [20, 1]
[24:34:38.241] Message: Taxi 2 moved to [1, 2]
[24:34:39.242] Message: Taxi 2 moved to [2, 3]
[24:34:40.243] Message: Taxi 2 moved to [3, 4]
[24:34:41.245] Message: Taxi 2 moved to [4, 5]
[24:34:42.241] Message: Taxi 2 moved to [5, 6]
[24:34:42.345] Message: Customer 'b' service has been completed. Taxi 2 left the customer on the location B [5, 6] and is now available
[24:34:42.546] Message: There aren't any customers in queue

[24:34:04.138] Message: Socket bound
[24:34:04.138] Message: Listening on port 2401
[24:34:09.638] Message: Processing authentication request 0
[24:34:09.641] Message: [request 0] Assigned ID 2
[24:34:09.743] Message: Updated map
[24:34:09.743] Message: [request 0] Closing connection

(Press 'q' to exit)
```

```

age: Taxi 2 moved to [4, 5]
age: Taxi 2 moved to [5, 6]
age: Customer
age: There ar
age: Service
age: Ordering
age: Taxi 2 m
age: Customer 'b' picked up by taxi 2. They are now goin
age: Taxi 2 moved to [4, 5]

Exiting...
(Press any key to continue)
```

3 despliegue

Como se comentó en la introducción, el despliegue está basado en Docker y, en menor medida, en Visual Studio.

3.1 Componentes dockerizados

Incluye todos los componentes a excepción de la interfaz gráfica. Para desplegar cualquiera de ellos, el primer paso es crear y ejecutar un contenedor basado en la imagen [easycab_image](#).

Puesto que las imágenes de Docker no son precisamente ligeras, para no depender de la conexión a internet lo ideal será guardar la imagen previamente en formato *.tar*:

```
docker save -o ./easycab_image.tar easycab_image
```

Para posteriormente instalarla localmente mediante el comando:

```
docker load -i ./easycab_image.tar
```

Si, por el contrario, no se dispone de una imagen previamente guardada, se puede descargar mediante el comando:

```
docker pull abtb2/easycab_image:latest
```

Una vez disponemos de la imagen, iniciaremos el contenedor con el comando:

```
sudo docker run --rm -e TERM=xterm-256color -ti easycab_image
```

3

Es importante mencionar que, si el componente que se pretende ejecutar dentro del contenedor va a abrir un puerto de escucha, es necesario añadir el argumento *-p p1:p2*, donde la dirección *ip_host:p1* se conectará con el puerto *p2* dentro del contenedor. Es decir, si iniciamos el contenedor con el argumento *-p 5000:8000* y, desde fuera del contenedor, tratamos de conectarnos a *localhost:5000*, dentro del contenedor se recibirá una petición de conexión en el puerto **8000**. En cualquier caso, para evitar confusiones, siempre es recomendable que ambos puertos sean el mismo.

³La variable de entorno *-e TERM=xterm-256color* es meramente estética. La mayoría de componentes cuentan con una interfaz gráfica de terminal basada en *ncurses* y con colores. Si no se establece esta variable, simplemente toda la interfaz será en el mismo color, blanco y negro. A partir de aquí, incluirla o no ya es decisión del usuario.

Una vez iniciado el contenedor, nos encontraremos con un directorio de trabajo similar al del [proyecto](#), con la diferencia de que en este caso, podremos encontrar la carpeta *build* con los archivos generados por cmake y los binarios compilados. Estos últimos son los que nos interesan.

Así pues, procederemos a ejecutar el componente deseado según las incidencias en la sección de cada componente.

3.2 Interfaz gráfica

Para desplegar la interfaz gráfica, bastará con descargar la carpeta *visual studio* del repositorio público y ejecutar el archivo *gui.sln* en un entorno de Windows. Esto abrirá una ventana de Visual Studio con el proyecto de la interfaz gráfica. Una vez abierto, nos dirigiremos al administrador de paquetes de NuGet de la solución y instalaremos los paquetes requeridos para el proyecto. A continuación, nos dirigiremos a propiedades del proyecto -> debugging -> argumentos adicionales y añadiremos la dirección al broker de Kafka. Por último, ya podemos ejecutar el proyecto en modo release y ver el programa en acción.

3.3 Kafka

Requisitos: tener java instalado en el sistema. **Pasos:**

1. **Descargar el archivo de Kafka:** <https://kafka.apache.org/downloads>
2. **Extraer el archivo de Kafka:** `tar -xzvf kafka_2.8.1.tgz`
3. **Iniciar el servidor de Zookeeper:** `bin/zookeeper-server-start.sh config/zookeeper.properties`
4. **Iniciar el servidor de Kafka:** `bin/kafka-server-start.sh config/server.properties`
5. **Desactivar windows defender o añadir una regla de firewall para permitir el acceso al puerto 9092** (desde el panel de control de seguridad de windows)
6. **Inicializar los topics** (desde el directorio de Kafka):

```
bin/kafka-topics.sh --bootstrap-server localhost:9092
--create --topic customer_responses &&
bin/kafka-topics.sh --bootstrap-server localhost:9092
--create --topic taxi_responses &&
```

```
bin/kafka-topics.sh --bootstrap-server localhost:9092
--create --topic map_responses &&
bin/kafka-topics.sh --bootstrap-server localhost:9092
--create --topic requests
```

3.4 Base de datos

Terminamos explicando el despliegue de MySQL Server, que no tiene mucho misterio en realidad. Para desplegar la base de datos, lo primero que tenemos que comprobar es que tengamos instalada la imagen de Docker `mysql:latest`. Esto se puede hacer descargándola de Docker Hub o instalándola localmente, [como se comentó anteriormente](#). Una vez instalada, bastará con ejecutar el siguiente comando:

```
docker run --rm --name mysql-server -e MYSQL_ROOT_PASSWORD
=1234 -v /ruta/initialize.sql:/docker-entrypoint-initdb.d/
script.sql -p 3306:3306 mysql:latest
```

¿Qué está haciendo este comando?

Las flags **`-rm`** y **`-name`** no son imprescindibles, sirven para eliminar el contenedor cuando acabe su ejecución y para etiquetarlo respectivamente.

Por el contrario, el siguiente argumento sí que es importante. **`-e MYSQL_ROOT_PASSWORD=1234`** define la contraseña para acceder a la bbdd y, dado que actualmente el código de la central no es flexible respecto a su uso (está definida como una constante en el código fuente), se recomienda no cambiarla.

El siguiente parámetro **`-v /ruta/initialize.sql:...`** indica que la base de datos cargará el archivo *initialize.sql* al inicializarse. Es importante tener acceso a este archivo (se encuentra en el [repositorio público](#) del proyecto) para desplegar la bbdd correctamente.

Por último, el parámetro **`-p 3306:3306`** publica el puerto 3306 del contenedor en el host para poder recibir conexiones, como se explicó en la sección de [componentes dockerizados](#).

Nota final: debe notarse que el documento empieza siendo bastante trabajado y va desinflandose a medida que avanza. A esto no tengo nada que hacer más que pedir disculpas. Recomiendo echarle un vistazo a las cabeceras (.h) del repositorio público en github, donde están explicdas más a detalle cada una de las funciones que componen el proyecto.

Gracias por leer hasta el final.