# Práctica 1

## Ejercicio Evaluable 1:
## Colas de mensajes


Miguel Jimeno Casas - 100495932 - 100495932@alumnos.uc3m.es

Alberto Menchén Montero - 100495692 - 100495692@alumnos.uc3m.es

Grupo 82 - Ingeniería informática

# 1. Introducción

Este proyecto consiste en la implementación de un sistema distribuido basado en colas de mensajes POSIX para la gestión de tuplas de la forma `<key, value1, value2, value3>`. En su versión monolítica, el sistema maneja estas tuplas utilizando una API definida en `claves.h` y cuya implementación se encuentra en `claves.c`.

El objetivo de esta práctica es convertir la implementación en un servicio distribuido, separando la lógica del cliente y del servidor. Para ello, se introducen dos nuevos módulos:

- `proxy-mq.c`, que actúa como intermediario en el lado del cliente, enviando peticiones al servidor.
- `servidor-mq.c`, que gestiona las solicitudes recibidas y ejecuta las operaciones sobre las tuplas.

Adicionalmente, se desarrolla un `Makefile` para facilitar la compilación y generación de los ejecutables y la biblioteca compartida `libclaves.so`.

# 2. Resumen del Proyecto

El código desarrollado se estructura en los siguientes archivos:

- **`claves.h`**: Contiene la definición de la API que permite gestionar tuplas.
- **`reales.h`**: Contiene la definición de la API que permite gestionar tuplas de manera privada.
- **`claves.c`**: Implementa la lógica de almacenamiento y manipulación de tuplas.
- **`proxy-mq.c`**: Sustituye las funciones originales de la API, enviando las solicitudes al servidor mediante colas de mensajes.
- **`servidor-mq.c`**: Implementa el servidor, que recibe peticiones, ejecuta las funciones en `claves.c` y devuelve las respuestas al cliente.
- **`app-cliente-1/2.c`**: Programa cliente que prueba la funcionalidad del sistema utilizando las funciones de la API.
- **`clear_queu.c`**: Programa que limpia la cola del servidor, en caso de que el programa se quede pillado.
- **`Makefile`**: Automatiza la compilación de los diferentes módulos y la generación de los ejecutables.

# 3. Descripción del Código

## 3.1. Gestión de Tuplas (`claves.c`)

Este archivo implementa una estructura de lista enlazada para gestionar dinámicamente las tuplas sin límite en su cantidad. Cada nodo de la lista representa una tupla y contiene:

```C/C++
typedef struct Clave {
    int key;
    char value1[256];
    int N_value2;
    double V_value2[32];
    struct Coord value3;
    struct Clave *next;
} Clave;
```

Se implementan las siguientes funciones:

- **`_set_value`**: Inserta una nueva tupla si la clave no existe.
- **`_get_value`**: Recupera los valores asociados a una clave.
- **`_modify_value`**: Modifica los valores de una clave existente.
- **`_delete_key`**: Elimina una clave.
- **`_exist`**: Comprueba si una clave existe.
- **`_destroy`**: Borra todas las tuplas almacenadas.

Estas funciones tienen un '_' delante, ya que las hemos considerado reales o privadas. Tuvimos que realizar esta denominación, ya que en el proxy tenemos las funciones SIN el '_' que son las que se incluyen el **`claves.h`** proporcionado. Nosotros creamos el **`reales.h`** para que varios archivos diferentes puedan usar estas funciones privadas incluidas en **`claves.c`**

## 3.2. Comunicación Cliente-Servidor

### Proxy (proxy-mq.c)

El proxy es el componente que permite la comunicación entre el cliente y el servidor mediante colas de mensajes POSIX. En lugar de interactuar directamente con la estructura de datos, el cliente envía solicitudes al proxy, que las encapsula en un mensaje estructurado y las transmite al servidor a través de una cola denominada `SERVIDOR`.

Cada solicitud se empaqueta en una estructura `Message`, que contiene los datos de la operación, la clave a modificar y una cola específica para recibir la respuesta del servidor. Una vez enviada la petición, el proxy espera en su propia cola efímera hasta recibir la respuesta y se la devuelve al cliente.

El siguiente fragmento de código muestra la lógica central del envío de mensajes al servidor y la recepción de respuestas:

C/C++
```
// Creación de una cola efímera para recibir la respuesta
type_mq client_mq = mq_open(client_queue_name, O_CREAT |
O_RDONLY, permisos, &attr);

// Envío del mensaje al servidor
mq_send(server_mq, (char *)msg, sizeof(struct Message), 0);

// Recepción de la respuesta
mq_receive(client_mq, (char *)msg, sizeof(struct Message),
NULL);

// Cierre y eliminación de la cola del cliente
mq_close(client_mq);
mq_unlink(client_queue_name);
```

### Servidor (servidor-mq.c)

El servidor actúa como el núcleo del sistema, escuchando continuamente en la cola `SERVIDOR` las peticiones de los clientes. Cuando recibe un mensaje, crea un hilo para procesarlo, lo que permite manejar múltiples clientes de manera concurrente.

Cada mensaje se desempaqueta y, dependiendo del tipo de operación, se llama a la función correspondiente de `claves.c` para gestionar la tupla solicitada. La respuesta se empaqueta nuevamente y se envía a la cola específica del cliente.

El siguiente fragmento muestra cómo se procesa una petición recibida:

C/C++
```
switch (msg_local.operation) {
    case 1: msg_local.result = _set_value(msg_local.key,
msg_local.value1, msg_local.N_value2, msg_local.V_value2,
msg_local.value3); break;
    case 2: msg_local.result = _get_value(msg_local.key,
msg_local.value1, &msg_local.N_value2, msg_local.V_value2,
&msg_local.value3); break;
    case 3: msg_local.result = _modify_value(msg_local.key,
msg_local.value1, msg_local.N_value2, msg_local.V_value2,
msg_local.value3); break;
    case 4: msg_local.result = _delete_key(msg_local.key);
break;
```

```
    case 5: msg_local.result = _exist(msg_local.key); break;
    case 6: msg_local.result = _destroy(); break;
    default: msg_local.result = -1;
}
```

### 3.3. Cliente de Pruebas (`app-cliente.c y app-cliente-limite.c`)

Estos archivos ejecutan pruebas sobre la API evaluando pruebas con valores límite para comprobar que se lleva a cabo una buena gestión de los parámetros pasados por las funciones.
Si se ejecutan por separado cada cliente podemos visualizar por la terminal del cliente los valores que retorna:

- **`app-cliente-1.c`**: todas las funciones retornan 0 al terminar sus procesos de set , get y modify y hace comprobaciones de exit, delete y destroy demostrando el resultado correcto por terminal.
- **`app-cliente-2`**: se prueban todas las pruebas limite que acepta nuestra implementación, reflejando entre paréntesis la respuesta esperada si el sistema funciona correctamente, para una rápida depuración de los métodos.

## 4. Forma de Compilar y Ejecutar

Para compilar el proyecto, se usa `Makefile`. Los pasos son:

1. Compilar el servidor: ‘make’
2. Iniciar el servidor: ./servidor
3. Compilar y ejecutar el cliente: ./app-cliente-1 o ./app-cliente-2 (tener en cuenta que se puede hacer de forma concurrente)
4. Para limpiar archivos generados: ‘make clean’

Como vimos en clase, el programa se queda pillado cada vez en un sitio de la implementación y no encontramos la justificación. Llegamos a la conclusión contigo de que era un bug en el pthread_cond_wait  o algún fallo del hardware que no entendíamos por qué no funcionaba. Sin embargo, ejecutando un debug, vimos que si eramos capaces de llegar a la solución esperada.

## 5. Conclusiones

Durante el desarrollo del proyecto hemos aprendido:

- La gestión de colas de mensajes POSIX para la comunicación entre procesos.

- La creación de una biblioteca compartida (`libclaves.so`).
- La importancia de la sincronización y gestión de errores en sistemas distribuidos.
- La implementación de estructuras dinámicas para gestionar datos sin límites predefinidos.

El resultado final es un sistema funcional que separa el cliente del servidor, permitiendo la manipulación distribuida de tuplas mediante colas de mensajes POSIX.