Kaska: editor/suscriptor con un esquema de tipo *streams*

Se trata de un proyecto práctico de carácter **individual** cuyo plazo de entrega se extiende hasta el final del **2 de junio** en la convocatoria ordinaria y hasta el final del **5 de julio** en la extraordinaria. La práctica se puede desarrollar en cualquier sistema Linux que tenga instalado el entorno de compilación de C. Puede usarse una máquina personal o cualquiera del conjunto de 4 máquinas asociadas al nombre `triqui.fi.upm.es.` gestionadas por el centro de cálculo de la escuela. En cualquier caso, como se explicará más adelante, hay que entregar la práctica en `triqui.fi.upm.es`.

AVISO IMPORTANTE
Como parte de la evaluación de la práctica, se realizará un control de plagio aplicando la normativa vigente para tal circunstancia tanto a los alumnos que han entregado una práctica que no han desarrollado como a los que han dejado su práctica a otros alumnos. Para evitar este tipo de incidencias desagradables para todos, querría que tuvierais en cuenta este par de consideraciones:

- Para aquellos que se sientan tentados de pedir prestada la práctica porque, por la razón que sea, no se ven capacitados para abordarla, el enunciado especifica con todo detalle qué pasos hay que realizar y cómo ir modificando el material de apoyo inicial para ir incorporando la funcionalidad pedida. Además, cuentan con mi ayuda para cualquier duda, por elemental que parezca. **Os garantizo que con un pequeño esfuerzo por vuestra parte y contactando conmigo por correo todas las veces que haga falta seréis capaz de superar esta práctica**.
- Para aquellos que se vean presionados para prestar su práctica, exponiéndose a ser penalizados, deberían tener en cuenta la consideración anterior.

Objetivo de la práctica

Como se ha estudiado en la parte teórica de la asignatura, los sistemas editor/suscriptor con un modo de operación *pull* y un almacenamiento persistente de eventos, es decir, con un esquema de *streaming*, proporcionan una arquitectura apropiada para numerosos escenarios distribuidos. Dentro de este tipo de arquitecturas, Apache Kafka es la principal plataforma y es en el sistema en el que se centra este proyecto práctico, cuyo nombre homenajea a dicha popular plataforma, además de servir como recordatorio de la falibilidad del software.

Evidentemente, se trata de una versión muy reducida de este complejo componente software dejando fuera muchas de sus funcionalidades (el uso de múltiples *brokers* con replicación y particiones, el *empaquetamiento* de peticiones y respuestas, el uso de claves, la persistencia de los mensajes, los grupos de consumidores, la recuperación de mensajes por *timestamp*, etc.), pero consideramos que permite apreciar qué tipo de funcionalidad tienen esta clase de sistemas y entender mejor cuál es su modo de operación interno.

A continuación, se incluyen algunos apartados que describen la funcionalidad y requisitos que debe satisfacer la práctica para que sean consultados cuando se necesite. En cualquier caso, puede ir directamente a la sección que explica paso a paso cómo desarrollar la práctica.

Repaso del modo de operación editor/suscriptor con *streaming*

Aunque ya lo hemos estudiado en la parte teórica de la asignatura, vamos a recordar los puntos más importantes de este tipo de sistemas suponiendo un modo de operación similar al de Kafka:

- El *broker* (*brokers* en el caso de Kafka) almacena los mensajes/eventos enviados por los productores/editores a los distintos temas.
- En Kafka un consumidor/suscriptor guarda a qué temas está suscrito y cuál fue el último mensaje/evento que leyó de cada uno de esos temas (el *offset*). Por tanto, esa parte del estado del sistema no se almacena en el servidor/*broker* sino en la biblioteca del cliente.
- Tiene un modo de operación *pull*: un consumidor/suscriptor pide al *broker* un nuevo mensaje indicándole a qué temas está suscrito y cuál es su *offset* para cada uno de esos temas.
- Cuando un consumidor/suscriptor se suscribe a un tema, su *offset* inicial será tal que solo podrá ver los mensajes que se envíen al mismo a partir de ese momento.
- Un consumidor/suscriptor puede modificar su *offset* en un tema para poder recibir mensajes anteriores a su suscripción o volver a recibir un mensaje nuevamente.
- Para permitir que un consumidor/suscriptor no tenga que estar siempre activo y pueda retomar el trabajo donde lo dejó al volver a ejecutarse, se pueden guardar de forma persistente sus *offsets* en el *broker* y recuperarlos al reiniciarse.

Requisitos de la práctica

A continuación, se especifican una serie de requisitos que deben ser **obligatoriamente satisfechos** por la práctica desarrollada:

- La práctica debe funcionar tanto en local como en remoto.
- En cuanto a las tecnologías de comunicación usadas en la práctica, se programará en C, se utilizarán sockets de tipo *stream* y se supondrá un entorno de máquinas heterogéneas.
- Se utilizará un esquema con un único proceso que actúa como *broker* proporcionando el desacoplamiento espacial y temporal entre los editores (productores en terminología Kafka) y los suscriptores (consumidores en terminología Kafka). Al usar un esquema de tipo *streaming*, el *broker* se encargará de almacenar los eventos.
- El *broker* dará un servicio concurrente basado en la creación dinámica de *threads* encargándose cada *thread* de servir todas las peticiones que llegan por una conexión.
- Un proceso editor y/o suscriptor (recuerde que un proceso puede ejercer ambos roles) mantendrá una conexión persistente con el *broker* durante toda su interacción. Dado ese posible doble rol, en el resto del documento vamos a denominar clientes a este tipo de procesos.
- Como en el protocolo Kafka, el nombre de un tema es un *string* con un tamaño máximo de 2^{16}-1 bytes incluyendo el carácter nulo final.
- **Los mensajes/eventos enviados pueden tener un contenido binario** (por ejemplo, pueden corresponder a una imagen o a texto cifrado). Por tanto, no pueden tratarse como *strings* y es necesario conocer explícitamente su tamaño. Nótese que las aplicaciones que usan este sistema usarán el esquema de serialización que consideren oportuno para enviar la información que manejan,
- El diseño del sistema **no debe establecer ninguna limitación en el número de temas y clientes existentes en el sistema ni en el número de mensajes almacenados en el *broker*.**
- Recuerde que debe manejar adecuadamente las cadenas de caracteres recibidas asegurándose de que terminan en un carácter nulo.
- Se debe garantizar un comportamiento *zerocopy* tanto en la gestión de los nombres de tema como en el contenido de los mensajes:
    - **No se pueden realizar copias de estos campos en el *broker***.
    - **No se pueden realizar copias de estos campos en la biblioteca de cliente, excepto cuando el enunciado lo indique.**
    - **Para evitar la fragmentación en la transmisión, tanto los clientes como el *broker* mandarán toda la información de una petición o de una respuesta, respectivamente, con un único envío**.

- Debe optimizarse el uso de ancho de banda de manera que el tamaño de la información enviada sea solo ligeramente mayor que la suma del tamaño de los campos que deben enviarse.
- Para facilitar el desarrollo de la práctica, se proporciona una implementación de un tipo de datos que actúa como un mapa iterable (`map`), permitiendo asociar un valor con una clave, y un tipo que gestiona una cola *append-only* (`queue`). **Se deben usar obligatoriamente estos tipos de datos a la hora de implementar la práctica**.

API ofrecida a las aplicaciones

En esta sección, se describen las operaciones que se les proporcionan a las aplicaciones, que están declaradas en el fichero `libkaska/kaska.h` al formar parte de la biblioteca de servicio a los clientes. Lógicamente, esta API se basa en la del propio Kafka, pero simplificada y sin *empaquetamiento* de parámetros. Así, por ejemplo, la función create_topics del API de Kafka para Python permite crear simultáneamente varios temas, estando el protocolo también diseñado para enviarlos en una sola petición. Sin embargo, en nuestra API con esa función solo se puede crear un único tema (`create_topic`) y lo mismo sucede con el resto de funciones.

A continuación, se describen las operaciones del API, que devolverán un valor negativo en caso de error en la comunicación, cuya funcionalidad más detallada será explicada de forma incremental, según se vayan especificando los sucesivos pasos en el desarrollo de la práctica.

- Crea el tema especificado, devolviendo 0 si la operación es correcta y un valor negativo en caso de error porque el tema ya exista.
- `int create_topic(char *topic);`
- Devuelve cuántos temas existen en el sistema.
- `int ntopics(void);`
- Envía el mensaje al tema especificado. Nótese la necesidad de indicar el tamaño del mensaje ya que puede tener un contenido de tipo binario. Devuelve el *offset* donde queda almacenado el mensaje en la cola asociada al tema y un valor negativo en caso de error debido a que el tema no exista. Si el segundo parámetro vale 0, la operación no realizará ninguna labor, pero no se se considerará que se trata de un error.
- `int send_msg(char *topic, int msg_size, void *msg);`
- Devuelve la longitud del mensaje almacenado en ese *offset* del tema indicado y un valor negativo en caso de error debido a que el tema no existe. Si no hay mensajes en ese *offset*, se debe devolver un 0.
- `int msg_length(char *topic, int offset);`

- Obtiene el último *offset* asociado a un tema en el *broker*, que corresponde al del último mensaje enviado más uno y, dado que los mensajes se numeran desde 0, coincide con el número de mensajes asociados a ese tema. Devuelve un valor negativo en caso de error debido a que el tema no existe.
- `int end_offset(char *topic);`
- Se suscribe al conjunto de temas recibidos. No permite suscripción incremental: hay que especificar todos los temas de una vez. Si un tema no existe o está repetido en la lista simplemente se ignora. Devuelve el número de temas a los que realmente se ha suscrito y un valor negativo solo si ya estaba suscrito a algún tema.
- `int subscribe(int ntopics, char **topics);`
- Se da de baja de todos los temas suscritos. Devuelve un valor negativo si no había suscripciones activas. Nótese que se trata de una operación local.
- `int unsubscribe(void);`
- Devuelve el *offset* del cliente para ese tema y un número negativo en caso de error porque no esté suscrito a ese tema. Nótese que se trata de una operación local.
- `int position(char *topic);`
- Modifica el *offset* del cliente para ese tema devolviendo un error si no está suscrito a ese tema. Nótese que se trata de una operación local.
- `int seek(char *topic, int offset);`
- Obtiene el siguiente mensaje destinado a este cliente. Los dos parámetros son de salida: a qué tema corresponde y el mensaje recibido. La propia función `poll` se encarga de reservar en memoria dinámica espacio para el tema y el mensaje. Es responsabilidad de la aplicación liberar ese espacio. Si no hay ningún mensaje, la operación retornará un 0. Devolverá un valor negativo si no está suscrito a ningún tema. Si en cualquiera de los parámetros se recibe un valor nulo, se procederá con la operación normal pero, evidentemente, no se asignará un valor a ese parámetro.
- `int poll(char **topic, void **msg);`
- Almacena de forma persistente en el *broker* el *offset* especificado para el tema indicado asociándolo con ese cliente. Devuelve un error si el tema no existe. No se requiere estar suscrito al tema.
- `int commit(char *client, char *topic, int offset);`
- Recupera del *broker* el *offset* almacenado para ese tema correspondiente a ese cliente. Devuelve un error si el tema o el cliente no existen. No se requiere estar suscrito al tema.
- `int commited(char *client, char *topic);`

Desarrollo de la práctica paso a paso

En esta sección iremos detallando los pasos que hay realizar para desarrollar la práctica identificando 5 fases, con una calificación de 2 puntos por cada fase:

1. Crear temas, así como todos los pasos iniciales.
2. Envío de mensajes.
3. Subscripción.
4. Lectura de mensajes.
5. *Commit* de *offsets*.

Fase 1 (2 puntos): Creación de temas y primeros pasos

El primer paso es descargarse el material de apoyo de la práctica, que está disponible en el Moodle de la asignatura. También se puede descargar directamente desde una URL:

```
wget https://laurel.datsi.fi.upm.es/_media/docencia/asignaturas/sd/kaska-2023.tgz
```

El material de apoyo está empaquetado en un fichero TGZ:

```
tar xvf kaska-2023.tgz
```

Es importante resaltar que en **el material de apoyo existen varios enlaces simbólicos que se requieren para poder compartir ficheros entre los distintos directorios. Si copia el material de apoyo directamente, puede perderlos y no funcionará correctamente**. Puede volver a crearlos o descargarse nuevamente el material de apoyo en otro directorio y copiar los ficheros que ha modificado.

Jerarquía de ficheros de la práctica

Revisemos los distintos directorios:

- `ejemplo_sockets`: Contiene un ejemplo de comunicación con sockets donde el cliente usa la función `writev` para enviar con una sola operación todos los datos requeridos. Concretamente, envía un entero, un *string* y un *array* de bytes, que son justo los tres tipos de objetos que hay que enviar en la práctica. Por tanto, como se explica más adelante, usaremos estos ficheros como base de la práctica, pero no realizaremos ningún trabajo en este directorio.
- `util`: Contiene la implementación de un mapa iterable (`map`) y de una cola de solo *append* (`queue`). Incluye un ejemplo del uso de estas estructuras (`demo`) que usaremos como base de la práctica, como se explica más adelante; no realizaremos ningún trabajo en este directorio. Recuerde que **se deben usar obligatoriamente estos dos tipos para implementar la práctica**.
- `clients`: Se proporciona un cliente (`test.c`), que ofrece una interfaz de texto, para probar la práctica. Este programa no requiere ningún argumento excepto en la quinta fase, donde se especifica el identificador del cliente. No hay que realizar ningún trabajo en este directorio.

- `broker`: Contiene la funcionalidad de este componente del sistema, que se incluirá en el fichero `broker.c`, que inicialmente está vacío. Como se explicará en breve, se recomienda usar como versión inicial de este componente parte del ejemplo de servidor con sockets e ir añadiendo ciertos fragmentos de la demo del directorio `util`. Este programa recibe como argumento el puerto de servicio y, para la quinta fase, el directorio donde se almacenarán los *offsets* salvados.
- `libkaska`: Corresponde a la biblioteca de cliente, cuyo código se incluirá en el fichero `kaska_client_lib.c`, que inicialmente contiene las funciones de servicio previamente identificadas, pero vacías. Como se explicará en breve, se recomienda usar como versión inicial de este componente parte del ejemplo de cliente con sockets e ir añadiendo ciertos fragmentos de la demo del directorio `util`. Este módulo recibirá la dirección del *broker* como dos variables de entorno:
    - `BROKER_HOST`: **nombre** de la máquina donde ejecuta el *broker* (debe funcionar tanto si se especifica un nombre como una IP, lo que está resuelto directamente al usar `getaddrinfo` tal como se hace en el ejemplo de cliente con sockets).
    - `BROKER_PORT`: número de puerto TCP por el que está escuchando.

  Para facilitar la reutilización de código entre los dos últimos módulos, se incluyen los ficheros `comun.c` y `comun.h`, que están presentes en los directorios de ambos módulos (`broker` y `libkaska`, respectivamente) mediante el uso de enlaces simbólicos (recuerde asegurarse de que durante la manipulación de los ficheros de la práctica no pierde por error estos enlaces), donde puede incluir funcionalidad común a ambos módulos si lo considera oportuno.

Recapitulando, el desarrollo de la práctica se centra en los ficheros:

- `broker/broker.c`
- `libkaska/kaska_client_lib.c`
- `broker/comun.[ch]`, si lo considera conveniente.

Ejecución de pruebas del sistema

Aunque todavía no hemos empezado con la funcionalidad, parece conveniente explicar desde el principio cómo se realizan las pruebas en esta práctica.

Para probar la práctica, debería, en primer lugar, arrancar el `broker` especificando el puerto de servicio que considere oportuno (para la quinta fase habrá que especificar también el directorio donde se almacenarán los *offsets*):

```
cd broker
make
./broker 12345
```
A continuación, puede arrancar instancias del programa `test` en la misma máquina o en otras. Este programa ofrece una interfaz de texto para que el usuario pueda solicitar la ejecución de cualquiera de las operaciones del sistema. Recuerde que para la quinta fase hay que especificar como argumento del programa el identificador del cliente:
```
cd clients
make
export BROKER_PORT=12345
export BROKER_HOST=nombre_del_host_del_broker
./test
```
Una instancia adicional pero pasándole las variables de entorno en la propia línea del mandato:
```
cd clients
make
BROKER_HOST=nombre_del_host_del_broker BROKER_PORT=12345 ./test
```

Versión inicial de la práctica

En primer lugar, se recomienda empezar creando una versión inicial, basada en los ejemplos de sockets proporcionados, de los dos ficheros donde se incluye el código de la práctica (recuerde que también puede hacerlo en `comun.c`):

- `broker.c`: Se recomienda usar directamente el código de `servidor.c`. Se trata de un servidor que crea dinámicamente *threads* tal que cada uno se encarga de servir todas las peticiones que llegan por una conexión y que, lógicamente, pertenecen al mismo cliente. En ese código inicial, ya está organizada la estructura de servicio, pero siempre espera recibir la misma información (un entero, un *string* y un *array* de bytes). Si opta por utilizar ese código de partida, mantenga estas líneas del *broker* original que incluye un control de errores adecuado para la quinta fase de la práctica:
```
    if (argc!=2 && argc!=3) {
        fprintf(stderr, "Uso: %s puerto [dir_commited]\n", argv[0]);
        return 1;
    }
```
- `kaska_client_lib.c`: Se recomienda incluir en el código inicial la función `init_socket_client` de `cliente.c`, que debe invocarse solo una vez cuando se llama por primera vez a una función de la biblioteca de manera que el socket conectado quede disponible para todas las funciones. Por el momento, el código de la función `petición` puede incorporarlo en la primera función (`create_topic`), que trataremos en el siguiente apartado. Nótese que debe cambiar los parámetros usados al llamar a `init_socket_client` ya que

la dirección y el puerto de servidor llegan como variables de entorno en vez de como argumentos.

Nótese que se ha optado por no incluir ese código de comunicación directamente sino mantenerlo como una recomendación para dejar a cada uno la libertad de usarlo o desarrollar el suyo propio.

Antes de empezar con la primera función, hay que tomar una decisión de diseño inicial: ¿cómo se van a distinguir las distintas operaciones en el protocolo que vamos a definir para la comunicación entre los clientes y el *broker*? Una opción es usar un entero que codifique cada posible operación lo que nos permite aprovechar mejor el código de ejemplo de sockets que ya envía un entero como primer valor (para quien tenga curiosidad, el protocolo de Kafka usa un entero de 16 bits). Con este esquema, la biblioteca enviaría por cada petición su código y el resto de información relevante, mientras que el *broker* leería en primer lugar ese código ejecutando a continuación el código específico de esa operación. Asimismo, parece razonable usar un *int* para enviar el resultado de cada operación, como ya está hecho en los ejemplos de sockets.

Otro aspecto a tener en cuenta en este punto es qué estructuras de datos usará el *broker* para gestionar el almacén de mensajes por tema. Una posible opción es crear un mapa que asocie el nombre de un tema, que actúa de clave, con el descriptor del tema, que sería el valor. Ese descriptor podría ser un `struct` con el nombre del tema y una cola asociada. En la demo del directorio `util` puede ver un ejemplo similar con cuentas, que equivaldrían a los temas, y la lista de operaciones realizadas sobre una cuenta, que corresponderían a los mensajes. Dado que el mapa se a va acceder desde múltiples *threads*, se debe crear, dentro del código de inicialización del *broker* con la opción de sincronización interna activa.

En cualquier caso, el alumno tiene total libertad en el diseño siempre que se cumplan los requisitos enunciados.

Función `create_topic`

En la parte de la biblioteca, simplemente hay que realizar el envío del código de operación y del nombre del tema y la recepción del resultado. Se puede, por tanto, reutilizar el código de apoyo eliminando el envío del *array*.

En cuanto al procesado de esa operación en el *broker*, habría que crear el descriptor del tema y la cola, con la opción de sincronización interna activa, e insertar ese descriptor en el mapa, tal como se hace en el programa demo, enviando un mensaje

de respuesta que refleje si la operación se ha realizado correctamente o ha habido un error porque el tema ya existía.

Función `ntopics`

La implementación es directa ya que solo es necesario enviar el código de operación y responder con el tamaño del mapa.

Fase 2 (2 puntos): Envío de mensajes

En esta fase entra en juego la cola de mensajes y es necesario, por tanto, definir un descriptor del mensaje que podría ser un `struct` con el tamaño del mensaje y la referencia al mismo. Cada entrada de la cola será un descriptor de mensaje. En la demo del directorio *util* puede ver un ejemplo similar.

Función `send_msg`

En la biblioteca de cliente hay que enviar, además del código de operación y el nombre del tema, también el *array* con el mensaje, es decir, lo mismo que en el ejemplo de sockets original.

En cuanto al *broker*, debe crear el descriptor del mensaje y añadirlo al final de la cola del tema indicado retornando su *offset* (la función `queue_append` lo devuelve).

Función `msg_length`

En la biblioteca de cliente hay que enviar un segundo entero que representa el *offset* solicitado, mientras que en el *broker* hay que usar `queue_get` para acceder a ese mensaje.

Función `end_offset`

La biblioteca de cliente tiene que enviar la misma información que en `create_topic`, mientras que el *broker* debe devolver simplemente el tamaño de la cola correspondiente.

Fase 3 (2 puntos): suscripción

Todo el código desarrollado en esta clase corresponde a la biblioteca de cliente. Hay que incluir en la misma un mapa que permita asociar los temas suscritos con sus *offsets* locales. Al no ser una biblioteca *multithread* no es necesario activar la sincronización interna del mapa.

Función `subscribe`

Debe crear el mapa de temas suscritos. Recuerde que no tiene un modo de operación incremental: da error si ya existe un mapa. Por cada tema a suscribir, usa directamente la función `end_offset` para comprobar que existe y conocer su *offset* actual, que será el que se almacene en la nueva entrada que se añade al mapa. Tenga en cuenta que el cliente no tiene acceso inicialmente a los mensajes previos. Nótese que en esta función tenemos que romper la estrategia *zerocopy* ya que no podemos añadir en el mapa una referencia al nombre de tema recibido como parámetro puesto que la aplicación puede reutilizarlo. Puede usarse la función `strdup` para crear un duplicado e insertar en el mapa una referencia al mismo.

Función `unsubscribe`

Libera el mapa de temas, dando error si no existía previamente.

Función `position`

Obtiene el *offset* local asociado al tema especificado usando `map_get` para acceder al mismo.

Función `seek`

Actualiza el *offset* local asociado al tema especificado usando `map_get` para acceder al mismo.

Fase 4 (2 puntos): lectura de mensajes (`poll`)

En este punto, ya tenemos un 6 en la práctica por lo que el enunciado deja de explicar con tanto detalle la implementación de cada función pasando a dar una descripción más somera de manera que el alumno tenga que hacer un mayor esfuerzo para programarla.

Con respecto a la parte de la biblioteca, la función `poll` itera por todos los temas suscritos (véase la demo de la biblioteca *util*) y, por cada tema, envía un mensaje al *broker* especificando, además del código de operación, el nombre del tema y el *offset* local (igual que en `msg_length`), obteniendo como respuesta un error, una indicación de que no hay un mensaje de ese tema con ese *offset* o el mensaje leído (nótese que es la única operación que devuelve un dato no escalar: un *array* de bytes que contiene el mensaje).

En ese último caso, se detiene la iteración y se devuelve en los parámetros de tipo puntero recibidos la referencia al mensaje y, volviendo a romper el *zerocopy*, un duplicado del nombre del tema (no podemos devolver la referencia al nombre de tema en el mapa ya que en ese caso la aplicación podría liberar la memoria asociada al mismo corrompiendo el mapa), retornando asimismo el tamaño del mensaje. En caso de error o falta de mensaje, se continúa la iteración, llegándose al final de la misma si no hay mensajes pendientes para ese cliente en ninguno de los temas.

Nótese que la próxima llamada `poll`, por equidad, debería comenzar a iterar el mapa justo por la entrada que está después de donde se quedó la llamada anterior. Para implementar esta funcionalidad, como se puede apreciar en la demo, hay que guardar una variable global de tipo `map_position`, que se inicializaría en `subscribe` y se liberaría en `unsubscribe`, para almacenar en qué posición termina una iteración (`map_iter_exit`) y usarla como parámetro en la siguiente (`map_iter_init`). Tenga en cuenta que **hay que asegurarse de que, cuando se interrumpe una iteración sin completarla, debido a que se ha encontrado un mensaje, el iterador queda apuntando al siguiente elemento**.

Con respecto al *broker*, debe acceder al mensaje pedido de manera similar a la operación `msg_length`, pero retornando esta vez su contenido.

Fase 5 (2 puntos): *Commit* de *offsets*

Hacer persistente un *offset* proporciona a los clientes la posibilidad de poder reanudar el tratamiento de mensajes justo en el punto donde se quedó después de un reinicio voluntario o involuntario. Dándole el control a la aplicación de cuándo se hace persistente un *offset*, permite que esta gestione el modelo de tratamiento de los errores que más le convenga. Así, por ejemplo, puede decidir hacer persistente un *offset* nada más leer el mensaje. Sin embargo, con esta estrategia, si se cae la aplicación justo en ese momento, el mensaje quedaría sin procesar. Otra alternativa es hacer persistente el *offset* después de procesar el mensaje, en cuyo caso, si se cae la aplicación después de procesar el mensaje pero antes de hacer persistente su *offset*, se volvería a procesar el mismo mensaje cuando reanude su ejecución.

Si se almacena el *offset* en la memoria del *broker*, este puede mantenerse aunque se caiga el cliente, pero no sobreviviría a la caída del *broker*. Kafka resuelve este problema de dos formas complementarias: replicando y almacenando en disco toda la información, tanto los *offsets* como los propios mensajes. Nótese que no serviría de mucho guardar en el disco los *offsets* sino se almacenan también los mensajes.

De cara a la práctica, sin embargo, dado que se complica el hacer persistentes en disco los mensajes (si alguien está interesado se podría plantear una práctica al respecto), vamos a conformarnos en esta última fase con guardar en el disco de la máquina donde ejecuta el *broker* los *offsets* salvados por los clientes.

Para esta fase, el programa `test` recibirá como argumento un idenficador de cliente que será el que permita vincular las sucesivas ejecuciones de una misma aplicación. Asimismo, el `broker` recibirá como segundo argumento el nombre de un directorio, que debe existir previamente, donde se almacenarán los *offsets*. Dentro de ese directorio, se creará un subdirectorio con el nombre del cliente para cada cliente que haya salvado alguna vez un *offset*. El *offset*, concretamente, se almacenará en un fichero de ese subdirectorio con el nombre del tema.

Función `commit`

En la biblioteca habría que enviar 2 *strings*: el cliente y el tema, junto con el código de operación y el *offset*. En cuanto al *broker*, habría que crear el subdirectorio correspondiente al cliente y el fichero asociado al tema, siempre que no existan, y escribir en el fichero el *offset*.

Función `commited`

En la biblioteca habría que enviar 2 *strings*: el cliente y el tema, junto con el código de operación. En cuanto al *broker*, habría que leer el *offset* almacenado en el fichero correspondiente.

Entrega de la práctica

Se realizará en la máquina `triqui`, usando el mandato:

```
entrega.sd kaska.2023
```

Este mandato recogerá los siguientes ficheros:

- `autores` Fichero con los datos de los autores:
- `DNI APELLIDOS NOMBRE MATRÍCULA`
- `memoria.txt` El fichero debe existir pero no es obligatorio rellenarlo. Ese fichero puede usarlo el alumno, si lo considera oportuno, para realizar los comentarios sobre la práctica que desee.
- `broker/broker.c` Código del *broker*.
- `broker/comun.h` Fichero de cabecera donde puede incluir, si lo precisa, definiciones comunes a los dos módulos, es decir, al *broker* y a la biblioteca.

- `broker/comun.c` Fichero donde puede incluir, si lo precisa, implementaciones comunes a los dos módulos, es decir, al *broker* y a la biblioteca.
- `libkaska/kaska_client_lib.c` Código de la biblioteca.