Diseño de una arquitectura editor/subscriptor con un esquema *pull* (*EdSuPull*)

Se trata de un proyecto práctico de carácter **individual** cuyo plazo de entrega termina el **25 de mayo**.

Objetivo de la práctica

El objetivo principal es que el alumno pueda ver de una forma aplicada qué infraestructura básica requiere una arquitectura de comunicación basada en un modelo editor-subscriptor con un esquema de tipo *pull*.

Para ello, se plantea desarrollar un esquema de este tipo con los siguientes requisitos específicos que deben ser **obligatoriamente satisfechos**:

- 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 va a seguir un modelo basado en *temas*.
- Se utilizará un esquema con un único proceso que actúa como *broker* proporcionando el desacoplamiento espacial entre los editores y los subscriptores. Al usar un esquema de tipo *pull*, el *broker* se encargará de almacenar los eventos dirigidos a un subscriptor hasta que este los recoja.
- En el arranque del sistema, el *broker* será informado mediante un fichero de qué temas existen en el sistema y estos serán los únicos disponibles para editores y subscriptores. El nombre de un tema puede contener cualquier carácter exceptuando un espacio.
- 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 subscriptor (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. Nótese que, dado el diseño planteado para el *broker*, este podrá también servir adecuadamente a clientes que usan una conexión por cada petición.
- Cada cliente se identificará ante el *broker* con un UUID (`man dbus-uuidgen`). Se ofrece una función de apoyo que debe ser usada **obligatoriamente** para generar ese identificador único universal.
- En cuanto al contenido de un evento, se propone un esquema muy sencillo. Un evento se caracterizará por un nombre de tema, que es un *string* con un tamaño máximo de 2^{32}-1 bytes incluyendo el carácter nulo final, y un único valor asociado, que puede ser de tipo binario por lo que es necesario conocer explícitamente el tamaño del mismo (ese tamaño está limitado por el campo que se usa para especificar su longitud).
- El diseño del sistema **no debe establecer ninguna limitación en el número de temas y subscriptores existentes en el sistema ni en el número de eventos 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 valor de los eventos:
  - **No se pueden realizar copias ni en los clientes ni en el *broker* de estos dos campos.**
  - **Para evitar la fragmentación en la transmisión, tanto los clientes como el *broker* mandarán toda la información 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 (`map`), permitiendo asociar un valor con una clave, un tipo que gestiona un conjunto (`set`) y un tipo que gestiona una cola (`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 `libedsu/edsu.h`. A continuación, se describen esas operaciones, que devuelven un valor negativo en caso de error y 0 en caso contrario:

- Al iniciar su ejecución, el cliente invoca la primera función que se debe conectar de forma persistente con el *broker* identificándose con su UUID (en el fichero `comun.c` se proporciona una función que debe usarse obligatoriamente para generar este identificador) y dándose de alta. El uso de este UUID en las sucesivas peticiones de un cliente permite relacionarlas entre sí aunque se use una conexión por petición, que no es el caso de la práctica (hipotéticamente, aunque no se afronte en la práctica, permitiría seguir dando servicio a un cliente si este se reinicia después de una caída siempre que se dé de alta con el mismo UUID). Cuando se completa la ejecución, el cliente llama a la segunda función que debe dar de baja al cliente y proceder con la desconexión. Nótese que en el material de apoyo estas dos funciones ya son invocadas automáticamente cuando comienza y termina el programa, respectivamente.

```
1- int begin_clnt(void); // inicio de un cliente
2- int end_clnt(void); // fin del cliente
```

- Operaciones para subscribirse y darse de baja de un tema. Retornará un error si el tema no existe o el cliente identificado no está dado de alta.

```
3- int subscribe(const char *tema);
4- int unsubscribe(const char *tema);
```

- Operación para publicar un evento asociado a un determinado tema. El valor del evento puede ser binario (podría, por ejemplo, ser una imagen) y, por tanto, se especifica su tamaño en el tercer parámetro. Si ese parámetro vale 0, la operación no realizará ninguna labor, pero no se se considerará que se trata de un error. Retornará un error si el tema no existe.

```
5- int publish(const char *tema, const void *evento, uint32_t tam_evento);
```

- Operación para obtener el siguiente evento de ese cliente. Los tres parámetros son de salida y corresponden a la información del evento obtenido: a qué tema corresponde y qué valor tiene asociado (el valor y su tamaño). La propia función `get` se encarga de reservar en memoria dinámica espacio para el tema y el valor del evento. Es responsabilidad de la aplicación liberar ese espacio. Retornará un error si el cliente no se ha dado de alta. Si no hay ningún evento, la operación devolverá un 0 en el puntero recibido como último parámetro. Si en cualquiera de los parámetros se recibe un valor nulo, se procederá con la lectura normal pero, evidentemente, no se asignará un valor a ese parámetro.

```
6- int get(char **tema, void **evento, uint32_t *tam_evento);
```

- Operaciones que facilitan la depuración y evaluación de la práctica.

```
7 - int topics(); // cuántos temas existen en el sistema
8 - int clients(); // cuántos clientes existen en el sistema
9 - int subscribers(const char *tema); // cuántos subscriptores tiene este tema
10 - int events(); // nº eventos pendientes de recoger por este cliente
```

Arquitectura software del sistema

Hay dos partes claramente diferenciadas en el sistema:

- El `broker` (fichero fuente `broker/broker.c`) que incluye toda la funcionalidad del sistema, implementando las operaciones descritas en el apartado previo y gestionando el almacenamiento de eventos. Recibe como argumento el número del puerto por el que prestará servicio y el fichero que contiene los temas.
- La biblioteca (fichero fuente `libedsu/libedsu.c`) que usan los procesos que quieren interaccionar con el `broker`, ofreciéndoles las funciones explicadas en la sección anterior. Este módulo recibirá la dirección del *broker* como dos variables de entorno:
  - `BROKER_HOST`: **nombre** de la máquina donde ejecuta el *broker*.
  - `BROKER_PORT`: número de puerto TCP por el que está escuchando.

Para facilitar la reutilización de código entre ambos módulos, se incluyen los ficheros `comun.c` y `comun.h`, que están presentes en los directorios de ambos módulos (`broker` y `libedsu`, respectivamente) mediante el uso de enlaces simbólicos (**nota**: asegúrese 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. Estos 2 documentos básicamente son pa poner el código que vayas a usar tanto en el broker.c como en libedsu (edsu.c)

En el directorio `broker/util` se proporciona la implementación de tres tipos de datos (un mapa, un conjunto y una cola). **Se deben usar obligatoriamente estos tres tipos para implementar la práctica**. Asimismo, están disponibles ejemplos de uso (demos) de los mismos.

Ejecución de pruebas del sistema

CUANDO TOQUE PROBAR, YA LO MIRARÉ

Para probar la práctica, debería, en primer lugar, arrancar el `broker` especificando el puerto de servicio y el fichero de temas:

```
triqui3: cd broker ftemas.txt
triqui3: make
triqui3: ./broker 12345
```

A continuación, puede arrancar instancias del programa `test/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, excluyendo las que inician y terminan un cliente, que, como se explicó previamente, se ejecutan automáticamente al arrancar y completarse el programa. Para facilitar la introducción del valor del evento y la comprobación de que el evento notificado es igual al publicado, el programa `test` usa ficheros para almacenar el contenido de los eventos.

```
triqui4: cd test
triqui4: make
triqui4: export BROKER_PORT=12345
triqui4: export BROKER_HOST=triqui3.fi.upm.es
triqui4: ./test
```

Una instancia adicional:

```
triqui2: cd test
triqui2: make
triqui2: export BROKER_PORT=12345
triqui2: export BROKER_HOST=triqui3.fi.upm.es
triqui2: ./test
```

Etapas en el desarrollo de la práctica

Para facilitar el desarrollo progresivo de la práctica y su evaluación incremental, se plantea una serie de etapas que corresponden a la implementación de las diversas operaciones del sistema. Antes de describir cada fase, vamos a proponer algunas pautas para afrontar este proyecto.

Pautas para el desarrollo de la práctica

Comencemos analizando qué tipos de objetos gestiona el *broker* y cuáles son las relaciones entre ellos:

- Básicamente, se gestionan tres tipos de objetos: temas, clientes y eventos. Se recomienda definir un descriptor para cada tipo. ¿Descriptor?
- Entre los temas y los clientes hay una relación N-M. Para implementarla, se propone que el descriptor de un cliente almacene el conjunto de descriptores de los temas a los que está subscrito y, de forma complementaria, un descriptor de tema guarde el conjunto de los descriptores de los clientes que lo han subscrito.
- Dada la necesidad de acceder al descriptor de un cliente a partir de su identificador (por ejemplo, para leer el siguiente evento), se debería crear un mapa de clientes para facilitar esa operación.
- Dado el requisito de acceder al descriptor de un tema a partir de su nombre (por ejemplo, para obtener cuántos subscriptores tiene un tema), se debería crear un mapa de temas para facilitar esa operación.

- Es necesario almacenar en el descriptor de un cliente una cola de eventos pendientes de recoger por parte del mismo.

Dado que para la implementación se requiere un uso intensivo de las estructuras de datos proporcionadas como apoyo, se recomienda la revisión de los programas de demostración de la utilización de las mismas, especialmente, la demostración *integral* que hace uso de los tres tipos de estructuras en un escenario con muchas similitudes con este proyecto práctico (también gestiona tres tipos de objetos: grupos, personas y mensajes). Nótese que estas estructuras pueden iniciarse para que usen internamente *mutex* en cada operación asegurando la exclusión mutua requerida por el uso de *threads*. Sin embargo, durante el desarrollo de la práctica, se recomienda activar las versiones no sincronizadas de estas estructuras para facilitar la depuración del programa.

Para facilitar el desarrollo del código que gestiona los sockets, se propone la reutilización del código incluido en los ejemplos de sockets estudiados en la asignatura:

- Dado que el cliente debe enviar toda la información con una sola operación, puede basarse en ejemplos como `cln_rev` o `envio_tam_variable_writev`, que usan `writev`.
- Dado que se plantea el uso de un servicio concurrente basado en *threads*, se puede usar como base inicial del *broker* el programa `srv_rev_thr`. Nótese que habría que modificar este programa en dos aspectos:
  - El *broker*, a diferencia del servidor nombrado, puede recibir distintos tipos de operaciones. Por tanto, es necesario definir algún tipo de mecanismo para identificar a qué operación corresponde una petición, quedando a criterio del lector (por ejemplo, puede usarse un valor entero, convertido a formato de red, con un valor por cada operación).
  - La respuesta del *broker* implica varios elementos por lo que requiere también el uso del `writev`.

NOTA: Se ~~planteará~~ un trabajo optativo que recogerá dos de las funcionalidades más avanzadas ~~que~~ finalmente no se han incluido ~~en la~~ práctica:

- Implementar una versión bloqueante (~~*long polling*~~) ~~del `get`~~.
- Cambiar el diseño del ~~*broker*~~ pasando de un esquema concurrente ~~a uno~~ basado en eventos (similar a ~~`srv_rev_evn`, que~~ usa *epoll*).

Etapa 1: Inicio del cliente (1,5 puntos)

Esta operación debe crear la conexión persistente usando los valores obtenidos de las variables de entorno y enviar el UUID del cliente para darlo de alta. En el *broker* hay que habilitar un descriptor para el cliente y darlo de alta en el mapa de clientes. No se considerará un error si ese cliente ya existe (como se comentó previamente, aunque no se afronte en la práctica, se podría plantear que, cuando se rearranca un cliente después de una caída, reutilice el mismo UUID para tolerar de esta forma ese problema).

Como parte de esta etapa, hay que implementar la operación `clients` que permite verificar si se han dado de alta los clientes. Pruebe a ejecutar múltiples instancias del programa `test` y comprueba que el número de clientes es correcto.

Etapa 2: Lectura de temas (1 puntos)

Esta operación debe realizarse en el *broker* antes de entrar en el bucle de servicio. Se debe leer el fichero de temas y darlos de alta en el mapa de temas, evitando, evidentemente, dar de alta temas duplicados.

Como parte de esta etapa, hay que implementar la operación `topics` que permite comprobar si se han dado de alta los temas del fichero.

Etapa 3: Subscripción (1,5 puntos)

Esta operación requiere incluir al cliente en el conjunto de subscriptores del tema correspondiente y, de forma complementaria, añadir el tema en el conjunto de temas subscritos, implementando de esta forma la relación N-M entre ambos tipos de objetos. Se debe comprobar que el cliente está dado de alta, el tema existe y el cliente no está previamente subscrito a ese tema.

Como parte de esta etapa, hay que implementar la operación `subscriptors` que permite comprobar cuántos clientes están subscritos a un determinado tema.

Etapa 4: Publicación (1,5 puntos)

En esta operación el *broker* tiene que crear el evento e insertarlo al final de la cola de cada cliente subscriptor. Se debe comprobar que el tema existe. Nótese que, como es habitual en un esquema editor-subscriptor, la no existencia de subscriptores en el momento de la publicación no se considera un error.

Como parte de esta etapa, hay que implementar la operación `events` que permite comprobar cuántos eventos tiene encolados el cliente. Nótese que si no está subscrito a ningún tema se devolverá que tiene 0 eventos, pero no se considerará un error (de hecho, como se verá a continuación, un cliente puede tener eventos encolados aunque no esté subscrito en este momento a ningún tema).

Etapa 5: Operación `get` (1,5 puntos)

En esta operación el *broker* lee el siguiente evento de la cola asociada al cliente (el primero de la cola) y se lo envía al cliente. Nótese que el descriptor del evento y el propio evento no se deben liberar hasta que el evento no haya sido enviado a todos los subscriptores a los que va destinado.

Nótese que en la parte cliente, como se explicó previamente, la propia función `get` se encarga de reservar memoria dinámica para el tema y el valor del evento.

Etapa 6: Baja de subscripción (1,5 puntos)

Esta operación elimina el descriptor del cliente del conjunto de subscritos al tema y, recíprocamente, elimina el descriptor del tema del conjunto de temas a los que está subscrito el cliente. Observe que los eventos pendientes de recibir se mantienen en la cola aunque estén vinculados con ese tema ya que se generaron cuando el cliente todavía era subscriptor. Se debe comprobar que el tema existe y que el cliente estaba subscrito al mismo.

Etapa 7: Fin del cliente (1,5 puntos)

Esta última operación debe, en primer lugar, dar de baja al cliente de todos los temas a los que estaba subscrito. Además, debe eliminar la entrada en el mapa correspondiente a ese cliente y liberar el propio descriptor del cliente que implica, a su vez, liberar el conjunto de temas subscritos y la cola de eventos descartando los eventos encolados.

Material de apoyo de la práctica

El material de apoyo de la práctica se encuentra en este enlace.

Al descomprimir el material de apoyo se crea el entorno de desarrollo de la práctica, que reside en el directorio: `$HOME/DATSI/SD/edsupull.2022/`.

Entrega de la práctica

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

```
entrega.sd edsupull.2022
```

Este mandato recogerá los siguientes ficheros:

- `autores` Fichero con los datos de los autores:

  `DNI APELLIDOS NOMBRE MATRÍCULA`

  ¿hay que crearlo de 0?

- `memoria.txt` Memoria de la práctica. En ella se deben comentar los aspectos del desarrollo de su práctica que considere más relevantes. Asimismo, puede exponer los comentarios personales que considere oportuno.

  ¿necesaria 100%?
- `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.
- `libedsu/edsu.c` Código de la biblioteca.

¿hay que hacerlos obligatoriamente?