Diseño de un esquema editor/subscriptor (EDSU)

Se trata de un proyecto práctico de carácter **individual** cuyo plazo de entrega termina el **22 de abril**.

NOTA IMPORTANTE: Sólo se considerará como correcta la práctica si funciona tanto en el caso de que todos los procesos involucrados ejecutan en la misma máquina como si cada uno está ejecutando en una máquina diferente. Además, debe asegurarse de que el diseño realizado permite que haya múltiples subscriptores ejecutando en la misma máquina.

AVISO IMPORTANTE: Aunque se puede desarrollar la práctica en cualquier máquina Linux, ésta debe funcionar en triqui, que es donde se llevará a cabo la corrección de la misma. Se debe tener en cuenta que triqui es una máquina de 64 bits, lo que puede afectar a operaciones que mezclan tipos de datos (por ejemplo, la asignación de un tipo puntero a un `int` no funcionará correctamente en este tipo de arquitecturas).

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.

Para ello, se plantea desarrollar un esquema de este tipo con las siguientes características específicas (Nota: puede ser conveniente que repase el material teórico de la asignatura para refrescar la terminología y los conceptos asociados a este esquema de interacción):

- Se va a seguir un modelo basado en *temas*.
- Se utilizará un esquema con un único proceso que actúa como intermediario proporcionando el desacoplamiento espacial entre los editores y los subscriptores. El intermediario se encarga de propagar cada evento generado por un editor a los subscriptores interesados en ese tipo de evento (tema).
- Va a usarse un esquema de tipo *push*. Como se irá viendo a lo largo del enunciado, y como se ha explicado en la parte teórica de la asignatura, la implementación de este tipo de comunicación *a contracorriente* (de tipo *callback*) y asíncrona es uno de los aspectos más complejos de construir en este esquema (la sección Manejo de eventos en el subscriptor estudia en detalle el problema).
- En cuanto al contenido de un evento, se propone un esquema muy sencillo. Un evento se caracterizará por un nombre de tema y un único valor asociado, correspondiéndose ambos elementos con *strings* que contienen cualquier carácter exceptuando un espacio.

En una primera versión básica de la práctica, no va a existir la posibilidad de creación dinámica de temas (tipos de eventos). En el arranque del sistema, el intermediario será informado mediante un fichero de qué temas existen en el sistema y éstos serán los únicos disponibles para editores y subscriptores.

En la versión avanzada, se permitirá la creación dinámica de temas, tal como se describirá en la sección dedicada a esa versión.

En cuanto a las tecnologías de comunicación usadas en la práctica, se utilizarán sockets de tipo *stream* y se supondrá un entorno de máquinas heterogéneas.

Descripción de la versión básica

Esta primera versión permite obtener una nota máxima de **siete** en la práctica. Como se explicó previamente, la principal diferencia con la versión avanzada está en cómo se gestionan los temas: en esta primera versión los temas están definidos a priori y no pueden crearse ni destruirse.

La aplicación a desarrollar se compone de tres módulos independientes:

- *Intermediario*. Se trata de un programa autónomo que incorpora la funcionalidad requerida para llevar a cabo ese rol de distribución de eventos.
- *Editor*. Es una biblioteca que se enlaza con las aplicaciones que requieren la funcionalidad de generar eventos.
- *Subscriptor*. Corresponde a una biblioteca que se enlaza con las aplicaciones que requieren la funcionalidad de ser notificadas de los eventos en los que están interesadas.

Nótese que, dado que los módulos editor y subscriptor son bibliotecas, nada impide que una aplicación se enlace con ambos si requiere ambos roles (por ejemplo, en un sistema de subastas, una aplicación que quiera tanto pujar por productos subastados como poder hacer un seguimiento de todas las pujas que se realizan sobre un producto). En cualquier caso, dentro del material de apoyo, se han incluido programas de prueba separados para cada módulo (ficheros `test_editor*.c` y `test_subscriptor*.c`, respectivamente).

En esta versión básica, entre estos tres módulos se producen las siguientes interacciones:

- El módulo subscriptor envía al intermediario peticiones de suscripción a un determinado tema, así como solicitudes para dar de baja una suscripción previa. El intermediario responde a esas peticiones informando de si se han realizado satisfactoriamente.
- El módulo editor envía eventos al intermediario; éste le confirma si los eventos son válidos.
- El módulo intermediario, cuando recibe un evento de un editor, lo envía a todos los subscriptores interesados en el mismo.

Queda a criterio del alumno definir qué formato se usará para los mensajes intercambiados entre los distintos módulos.

A continuación, se describe con más detalle la funcionalidad de cada uno de los tres módulos en esta versión básica.

Intermediario

Este proceso recibe dos argumentos:

- El puerto TCP por el que proporcionará servicio.
- El nombre de un fichero que contendrá la lista de temas existentes. Cada línea del fichero corresponderá a un tema, pudiendo incluir el nombre de un tema cualquier carácter excepto un espacio. Se supone que en el fichero no hay líneas (nombres de tema) repetidas (para asegurarse de ello, bastaría con que el programa intermediario aplicara el mandato `sort -u` al fichero, pero no lo contemplamos en este trabajo práctico).

El proceso intermediario debe gestionar una estructura de datos que le permita conocer quiénes están subscritos a cada tema, es decir, que incluya información suficiente para poder contactar con todos los interesados en un determinado tema.

En su arranque, el programa incluirá en esta estructura todos los temas contenidos en el fichero recibido como argumento, que, como se ha explicado previamente, en esta versión básica serán los mismos (ni se crean, ni se destruyen) durante toda la ejecución del intermediario. Evidentemente, no habrá en ese momento ningún subscriptor para ningún tema.

Para facilitar el desarrollo de la práctica, se recomienda que el proceso intermediario sea de **tipo secuencial** para eliminar los problemas de sincronización que se producirían en una versión concurrente. Asimismo, en aras de mantener esta simplicidad, se aconseja usar un esquema con una conexión por cada intercambio de mensajes.

Por lo demás, se trataría del clásico servidor de tipo *stream* que puede recibir tres tipos de mensajes ya previamente comentados:

- Alta de una suscripción a un determinado tema, que incorporaría al proceso remitente del mensaje en la lista de interesados en ese tema y que enviaría un mensaje especificando si la operación se ha realizado satisfactoriamente o se ha producido un error (por ejemplo, debido a que el tema solicitado no existía).
- Baja de una suscripción a un determinado tema, que eliminaría al proceso remitente del mensaje de la lista de interesados en ese tema y que enviaría un mensaje especificando si la operación se ha realizado satisfactoriamente o se ha producido un error (por ejemplo, debido a que el tema solicitado no existía o a que el proceso no estaba dado de alta en el mismo).
- Recepción de evento generado por un editor, que provocaría el envío de dicho evento a todos los subscriptores de ese tipo de eventos. El intermediario debe confirmar al editor si el evento era válido o erróneo (por ejemplo, debido a que estaba asociado a un tema no existente).

El código de este programa se incluirá en el fichero `intermediario.c`. En cualquier caso, para evitar duplicidades de código, si en el código de su práctica existen definiciones o implementaciones que comparten los tres módulos, puede incluirlas en los ficheros `comun.h` y `comun.c`, respectivamente.

Editor

Este módulo recibirá la dirección del intermediario como dos variables de entorno:

- `SERVIDOR`: nombre de la máquina donde ejecuta el intermediario.
- `PUERTO`: número de puerto TCP por el que está escuchando.

Este módulo incluye la funcionalidad para generar eventos y se implementará en el fichero `editor.c`, ofreciendo una única operación en esta versión básica (definida en `editor.h`):

```
int generar_evento(const char *tema, const char *valor);
```

Los argumentos de esa operación corresponden a la pareja que define cada evento: el tema al que está vinculado y el valor que tiene asociado dicho evento. En cuanto al valor de retorno, será 0 si la operación ha sido correcta y -1 en caso de error.

Subscriptor

Este módulo recibirá la información sobre la dirección del intermediario de la misma manera que lo hacía el módulo editor. De hecho, si así lo desea, puede incluir ese código o cualquier otro código común a los módulos editor y subscriptor, pero no al intermediario, en los ficheros `edsu_comun.c`, si se trata de implementaciones, y `edsu_comun.h`, en caso de ser definiciones.

Este módulo incluye la funcionalidad para subscribirse, y darse de baja posteriormente, a temas de interés, y se implementará en el fichero `subscriptor.c`, ofreciendo las siguientes operaciones para la versión básica (definidas en `subscriptor.h`):

```
int inicio_subscriptor(void (*notif_evento)(const char *, const char *),
        void (*alta_tema)(const char *),
        void (*baja_tema)(const char *));

int alta_suscripcion_tema(const char *tema);

int baja_suscripcion_tema(const char *tema);
```

En cuanto a la primera operación, en esta versión básica, sólo se usará el primer parámetro, dejando los otros con un valor `NULL`. Este primer parámetro permite que la aplicación que usa este módulo pueda especificar qué función ha definido para ser notificada de los eventos en los que está interesada (función de *callback*). Esta operación debe ser invocada por la aplicación antes de poder usar ninguna de las otras dos operaciones.

El siguiente fragmento extraído del programa de prueba del subscriptor (`test_subscriptor.c`) muestra el uso de esta operación:

```
static void notificacion_evento(const char *t, const char *e){
        printf("\n-> Recibido evento de tema %s con valor %s\n", t, e);
        ......................
}

int main(int argc, char *argv[]) {
        ......................

        inicio_subscriptor(notificacion_evento, NULL, NULL);
        ......................
```

Nótese que esta operación no requiere en esta primera versión contactar con el intermediario, consistiendo básicamente en que el módulo subscriptor *anote* en su estado interno cuál es la función especificada por la aplicación para, posteriormente, invocarla cuando se reciba un evento del intermediario. Como se explicará en breve, en la sección Manejo de eventos en el subscriptor, la ejecución de la función especificada por la aplicación se realizará en un contexto concurrente al de la propia aplicación (de manera relativamente similar a lo que puede ocurrir con la rutina de tratamiento de una señal en UNIX, un *callback* de Java RMI o el *Event Dispatch Thread* de Java).

Con respecto a la segunda operación, la de suscripción al tema recibido como argumento, requerirá enviar el mensaje correspondiente al intermediario, que, como se comentó previamente, responderá especificando si se ha realizado satisfactoriamente o no. La operación devolverá un 0 en caso de no producirse error y -1 si lo ha habido. También devolverá un -1 en caso de que no se haya invocado previamente la operación que establece la función de notificación.

En cuanto a la operación de baja de la suscripción del tema recibido como argumento, su tratamiento en este módulo será similar al realizado en el caso de la operación de alta, tal como se acaba de describir.

Aunque se ha completado la descripción de las tres operaciones proporcionadas por este módulo, falta por explicar la parte más compleja de implementar requerida por el esquema de tipo *push*: el manejo de los eventos en el subscriptor.

Manejo de eventos en el subscriptor

Como ya se ha comentado previamente, el uso de un esquema editor-subscriptor con un modo de operación de tipo *push* presenta el problema de que los mensajes que envía el intermediario a un subscriptor notificándole sobre los eventos generados por los editores llegan de forma *asíncrona* y *a contracorriente*:

- *Asíncrona*: El subscriptor está englobado en una aplicación que está llevando a cabo la funcionalidad para la que ha sido diseñada y no puede, por tanto, bloquearse a la espera de los mensajes de notificación de eventos que le lleguen del intermediario.
- *A contracorriente*: Supongamos que se ha optado por una solución que mantiene conexiones persistentes con el intermediario. Con ese esquema, cuando un subscriptor *X* invoca, por ejemplo, una operación de alta en el tema *Y*, se queda bloqueado esperando el mensaje del intermediario que le confirme si ésta se ha realizado satisfactoriamente. Sin embargo, puede ocurrir que, justo antes de que el intermediario reciba el mensaje de *X* con la petición de alta, le haya llegado de un editor un evento del tema *Z* en el que también estaba interesado *X*, con lo que el próximo mensaje que recibe *X* no es la confirmacíón del alta sino la notificación del evento. En el caso de utilizar una solución que usa una conexión por petición, nos encontramos con que en las peticiones del subscriptor, es éste el que debería ser el elemento activo (realizar el `connect`), mientras que en las notificaciones es el intermediario quien debería ejercer ese rol. Obviamente, un socket no permite ese cambio de papeles.

Aunque hay distintas soluciones a este problema, en este enunciado se sugiere la siguiente:

- Para tratar el problema de la asincronía, el subscriptor puede utilizar un *thread* independiente para recibir y procesar los mensajes de notificación del intermediario, invocando la rutina de notificación especificada por la aplicación por cada evento recibido. Nótese como esta rutina se ejecuta en un contexto concurrente con el de la aplicación.
- Para afrontar el problema de los mensajes cruzados *a contracorriente*, se sugiere que el subscriptor use dos sockets: uno para enviar las peticiones de alta/baja y recibir las confirmaciones de esas operaciones, y otro, que sólo utilizaría el *thread*, por el que se recibirían los mensajes de notificación de eventos.

Obsérvese que el *thread* oyente proporciona un contexto independiente de ejecución para la función de *callback* especificada por el programa. Sin embargo, si el tiempo de ejecución de dicha función es muy largo o se bloquea, se quedará congelada la recepción de notificaciones mientras tanto (lo que podría también bloquear al intermediario e incluso a los editores). Asimismo, puede ser problemático si desde la función de *callback* se invoca a su vez una función de la biblioteca, pudiendo producirse, dependiendo de cómo sea la implementación, interbloqueos en el sistema.

De cara a la realización de este proyecto, no se considerarán estas situaciones problemáticas, presentes en cualquier esquema con llamadas de tipo *callback* , siendo suficiente con la solución basada en un único *thread* oyente.

En un sistema real, se podría optar, por ejemplo, por un esquema donde el *thread* oyente crea *threads* adicionales para proporcionar un contexto de ejecución propio a las funciones *callback*.

Descripción de la versión avanzada

Esta versión, que puede otorgar hasta **tres** puntos adicionales, incluye la posibilidad de gestión dinámica de temas: los subscritores van a poder crear nuevos temas y destruirlos cuando lo consideren oportuno. Los subscriptores serán notificados de estos dos nuevos tipos de eventos: el anuncio de un nuevo tema y el

de su eliminación. Nótese que se podría interpretar que hay una especie de "meta-tema" en el que todos los subscriptores están interesado y cuyos eventos corresponden a la creación y destrucción de temas.

Para implementar esta nueva funcionalidad, el intermediario deberá conocer qué subscriptores hay en el sistema para enviarles eventos de creación y destrucción de temas.

Para explicar cómo se debe comportar esta nueva versión, se va a detallar cómo se llevan a cabo las nuevas operaciones que aparecen en esta versión, explicando primero las que atañen al editor y, a continuación, las que afectan al subscriptor.

Editor

Las nuevas operaciones permiten crear y destruir temas:

```
int crear_tema(const char *tema);
int eliminar_tema(const char *tema);
```

La operación `crear_tema` requiere que el intermediario actualice sus estructuras de datos para incorporar el nuevo tema y anuncie a todos los subscriptores la aparición de un nuevo tema. Los subscriptores al recibir el anuncio ejecutarán la función especificada por la aplicación para tal circunstancia (segundo parámetro de `inicio_subscriptor`). La función `crear_tema` devolverá 0 si todo va bien y -1 en caso de error (por ejemplo, debido a que el tema ya existe).

La operación `eliminar_tema` debe causar que el intermediario actualice sus estructuras de datos para eliminar ese tema y anuncie a todos los subscriptores esta circunstancia. Los subscriptores al recibir el anuncio ejecutarán la función especificada por la aplicación a tal efecto (tercer parámetro de `inicio_subscriptor`). La función `eliminar_tema` devolverá 0 si todo va bien y -1 en caso de error (por ejemplo, debido a que el tema no exista previamente).

Subscriptor

Con respecto a este módulo, sólo hay una nueva operación, que permite que el subscriptor se dé de baja, aunque también hay que extender la funcionalidad de la operación que inicia al subscriptor:

```
int inicio_subscriptor(void (*notif_evento)(const char *, const char *),
        void (*alta_tema)(const char *),
        void (*baja_tema)(const char *));

int fin_subscriptor();
```

Con respecto a la operación `inicio_subscriptor`, en esta versión la aplicación debe especificar las tres funciones de notificación, usándose las dos últimas para anunciarle altas y bajas de temas, como se explicó previamente. Además, en caso de que se invoque a esta función con un valor distinto de nulo en el segundo parámetro, el subscriptor debe contactar con el intermediario para que le incluya en sus estructuras de datos como un nuevo subscriptor al que se le notificarán la aparición y desaparición de temas. La función `inicio_subscriptor` devolverá un 0 si todo va bien y -1 si hay un error (por ejemplo, debido a que ese proceso ya está previamente subscrito).

Como parte del proceso de darse de alta como subscriptor, al proceso se le comunicará cuáles son los temas que existen actualmente. Por homogeneidad, esta información la recibirá como notificaciones, de la misma manera que conocerá la aparición de nuevos temas en el futuro.

La operación `fin_subscriptor` requiere que el intermediario elimine a ese subscriptor de todas sus

estructuras de datos, devolviendo un -1 si hay un error (por ejemplo, debido a que no esté dado de alta previamente) y 0 en caso contrario.

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/EDSU.2015/`.

Como material de apoyo, sólo se proporciona programas de prueba (`test_subscriptor*.c` y `test_editor*.c`), así como las interfaces de servicio (`editor.h` y `editor.h`, respectivamente).

Entrega de la práctica

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

```
entrega.sd edsu.2015
```

Este mandato recogerá los siguientes ficheros:

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

  ```
  DNI APELLIDOS NOMBRE MATRÍCULA
  ```

- `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.
- `intermediario/intermediario.c` Código del intermediario.
- `intermediario/comun.h` Fichero de cabecera donde puede incluir, si lo precisa, definiciones comunes a los tres módulos, es decir, intermediario, editor y subscriptor.
- `intermediario/comun.c` Fichero donde puede incluir, si lo precisa, implementaciones comunes a los tres módulos, es decir, intermediario, editor y subscriptor.
- `subscriptor/subscriptor.c` Código del módulo subscriptor.
- `subscriptor/edsu_comun.h` Fichero de cabecera donde puede incluir, si lo precisa, definiciones comunes a los módulos editor y subscriptor.
- `subscriptor/edsu_comun.c` Fichero donde puede incluir, si lo precisa, implementaciones comunes a los módulos editor y subscriptor.
- `editor/editor.c` Código del módulo editor.