# Indice

# Configuración de Netcp

Hasta el momento, la IP y el puerto del proceso que va a recibir nuestros archivos se fijan en el código. Eso no resulta muy práctico, porque si queremos conectar con otro Netcp, tenemos que cambiarlo en el código y compilarlo. Así que vamos a preparar nuestro programa para que se pueda configurar mediante variables de entorno.

En la sección cómo pasar parámetros de inicialización a los procesos hijo de los apuntes, se muestra como leer desde un proceso las variables de entorno con getenv(). La idea es que el programa lee las variables de entorno NETCP_DEST_IP, NETCP_DEST_PORT y NETCP_PORT para conocer la IP y el puerto donde escucha el proceso al que queremos mandar los archivos y el número de nuestro puerto local, para recibir archivos, respectivamente.

En BASH estas variables de entorno se pueden asignar mediante *export*, antes de ejecutar el programa:

```
# export NETCP_DEST_IP=127.0.0.1
# export NETCP_DEST_PORT=8000
# export NETCP_PORT=8001
# ./netcp
```

o se puede hacer en una sola línea:

```
# NETCP_DEST_IP=127.0.0.1 NETCP_DEST_PORT=8000 NETCP_PORT=8001 ./netcp
```

## [TAREA] Soportar la configuración de Netcp mediante variables de entorno

Implementa la forma comentada de configurar Netcp usando variables de entorno. Ten en cuenta que el usuario puede olvidarse de configurar las variables de entorno que necesitas, en cuyo caso getenv() devuelve NULL. Obviamente, en ese caso, si intentas usar el puntero devuelto por getenv(), la aplicación se detendrá con una violación de segmento.

**Debes detectar ese caso y usar un valor por defecto o detener el programa indicado al usuario que debe configurar las variables de entorno**. Nunca es una opción dejar que el programa falle de esta manera.

# Netcp con envío múltiple

Vamos a modificar algunos de los comandos para permitir hacer varios envíos al mismo tiempo:

---

- **Send [nombre_archivo]** iniciará el envío del archivo indicado. Pero ahora **mostrará por la salida estándar el identificador de envío de fichero**, que se podrá usar en comandos posteriores.
- **Abort [id_envío]**, abortará el envío con el identificador indicado. Si el comando es **Abort receive**, se desactiva el modo de recepción, como antes.
- **Pause [id_envío]**, pausará el envío con el identificador indicado.
- **Resume [id_envío]**, continuará el envío con el identificador indicado.

# [TAREA] Envío múltiple

Modifica tu prototipo de Netcp para implementar los comandos como hemos comentado al principio de esta sección:

- **Necesitarás una estructura de datos a modo de tabla de tareas de envío**. Cada envío activo deben tener una entrada en la tabla con un identificador único —un entero que se incrementará en cada ocasión—. Así el hilo de la tarea 1 puede buscar la tarea para gestionarla —pausar, aborta, etc.— y el hilo principal puede cancelar las tareas pendientes antes de terminar la aplicación.
- Cada vez que el usuario usa **el comando Send, el hilo de la tarea 1 crea un hilo que se hará cargo de acceder al archivo y enviarlo**. Así es como se soportan varios envíos al mismo tiempo. El objeto del hilo se incorpora a la tabla de tareas de envío para que esté accesible al resto de la aplicación.
- **Cuando un hilo de envío termina**, tras mostrar que el envío se ha completado, **el hilo se extrae a sí mismo de la tabla de envíos y termina**.
- El hilo de la **tarea 1 puede cancelar o pausar un envío cuando el usuario lo pide**, usando los mecanismos de cancelación y coordinación que desarrollamos para la parte anterior de la práctica.

En el hilo de la tarea 3 —el de recepción— ahora recibirá a través de su socket paquetes de diferentes envíos, para guardar en distintos archivos. ¿Cómo sabe, cuando llega un paquete, si es un nuevo envío —recordemos que el primer paquete tiene nombre de archivo y tamaño— o si es contenido para un envío ya iniciado? Sencillo:

- **Cada hilo de envío usará un socket diferente para hacer el envío**. Así se hará desde la IP del equipo y un puerto diferente para cada envío.
- **El hilo de recepción, al leer un paquete, recibe también la dirección y puerto del remitente**. Esa información sirve para distinguir unos envíos de otros. Simplemente se consulta en una tabla de envíos.
    - Si no hay entrada, es el primer paquete de un nuevo envío. Así que se crea el archivo, se mapea, etc. y se inserta la información necesaria en la tabla
    - Si hay entrada, es que se trata de un paquete para un envío ya iniciado. Así que se recupera la información del envío desde la tabla y se usa el contenido del paquete como corresponda.

Como hemos comentado, necesitamos controlar el número de tareas de envío activas, con lo que se sugiere utilizar una estructura tipo map, que asocie el identificador del envío con el hilo que lo

gestiona. Al crear un hilo debemos añadir una entrada en el map, y cuando la tarea termina, debe eliminar su entrada:

```
std::unordered_map<int, SendingTask>> sending_tasks;
```

La estructura `SendingTask` debe contener la información que necesitamos preservar de cada tarea de envío que debe ser accesible para el resto de la aplicación: objeto std::thread, variables atómicas para coordinar la cancelación o la pausa, etc.

Esta estructura nos servirá también para terminar de manera controlada todos los hilos activos cuando deba terminar la aplicación.

La tarea 3, para recibir archivos, necesita un map parecido:

```
std::unordered_map<std::pair<uint32_t, in_port_t>, ReceptionTask>>
reception_tasks;
```

Este map se indexa por la pareja (dirección IP, puerto) para obtener la información sobre la tarea de recepción guardada en `ReceptionTask`: descriptor de archivo, tamaño, puntero a la zona mapeada de la memoria, etc.

Por último, **no te olvides de manejar los posibles errores adecuadamente.** Es decir, comprobar condiciones de error y lanzado e interceptando las excepciones correspondientes.

## Ojo con los recursos compartidos

La estructura de datos `sending_task` es accedida desde varios hilos, por lo que debe ser protegida con un mecanismo de sincronización como `std::mutex`, para evitar condiciones de carrera.

En la sección mutex de los apuntes, se enlazan varios ejemplos de su uso en C++.

## [TAREA] Evita condiciones de carrera

Evitar las condiciones de carrera cuando varios hilos acceden `sending_task` al mismo tiempo para manipularlo.

Lo recomendable es crear una clase con `sending_task` y el std::mutex como miembros y métodos adecuados para acceder a la estructura de datos evitando condiciones de carrera.

# Verificar el archivo transferido

El protocolo UDP no es fiable, así que si utilizamos Netcp en Internet el archivo puede que no se transfiera correctamente. Para detectar esa situación, el hilo de envío puede ejecutar el comando:

---

```
# md5sum <archivo>
```

para obtener la suma de comprobación MD5 del archivo enviarla en el primer mensaje, junto con el nombre y el tamaño. El hilo que recibe hace lo mismo tras recibir el archivo y ambos sumas de comprobación se muestran al usuario al informar de que la transferencia ha terminado.

Para ejecutar el comando md5sum tenemos que utilizar fork() y exec(). Y para capturar la salida estándar del comando, por la que md5sum muestra la suma de comprobación, tenemos que usar tuberías. Al ejecutar md5sum con la salida redirigida a la tubería, podemos leer el otro extremo de la tubería para obtener la salida del comando desde el hilo encargado del envío.

# fork() y exec()

En la sección operaciones con procesos vimos de los apuntes, vimos cómo crear procesos con fork() y cómo ejecutar un programa en sistemas POSIX con las funciones de la familia exec(). Aparte de la explicación paso a paso, se enlazan varios ejemplos completos que pueden servir de ayuda.

# Comunicando procesos mediante tuberías

En la sección tuberías de los apuntes, se explican las características de las tuberías y se enlazan dos ejemplos interesantes. El ejemplo **fork-pipe.cpp** muestra cómo se pueden comunicar un proceso padre con su hijo mediante tuberías. Mientras que el ejemplo **fork-redir.c** muestra cómo un proceso hijo puede usarse para ejecutar otro programa, redirigiendo lo que escriba en la salida estándar, al proceso padre, mediante una tubería.

## [TAREA] Implementa el cálculo de la suma de comprobación MD5.

Modifica tu prototipo de Netcp para que ejecute el comando md5sum para obtener la suma de comprobación antes de comenzar a enviar y al terminar de recibir. El hilo encargado del envío debe mandar la suma al receptor, junto con el nombre del archivo y el tamaño. Y el receptor debe mostrar ambas al terminar la recepción del archivo.