Programación de aplicaciones — Parte II

Sistemas Operativos 2023/2024

Vamos a desarrollar una herramienta para copiar archivos entre equipos conectados a Internet. En este caso, mejorar las características del programa `netcp` que desarrollamos en la parte anterior.

Contenidos

1. Introducción

1.1. Objetivo

Para no depender de `netcat`, la nueva versión de `netcp` soportará dos modos de funcionamiento. En el primer modo, lee el archivo indicado en la línea de comandos y envía su contenido al puerto UDP 8080 en la dirección IP 127.0.0.1.:

```
$ ./netcp testfile
```

El segundo modo se activa usando la opción `-l` y lo que hace es escuchar en el puerto 8080 y guardar los datos recibidos en un archivo con el nombre indicado en la línea de comandos:

```
$ ./netcp -l testfile2
```

En este segundo modo, si el archivo no existe, lo crea, y si existe lo sobrescribe.

En esta versión, no debe existir límite en el tamaño del archivo que se puede enviar o recibir, por lo que te recomendamos que uses `dd` para crear archivos relativamente grandes con datos aleatorios, para comprobar el funcionamiento del programa.

> **Nota**
>
> Recuerda que para comprobar que la copia se hace correctamente, puedes usar el comando `cmp` para comparar los archivos.

El puerto de escucha en el modo `-l` y el puerto de destino en el modo normal, se puede cambiar usando la variable de entorno `NETCP_PORT` –ver Sección 3.2.1–:

```
export NETCP_PORT=1234
$ ./netcp -l testfile
```

Mientras que la IP de destino en el modo normal, se puede cambiar usando la variable de entorno `NETCP_IP`:

```
export NETCP_IP=192.168.10.11
export NETCP_PORT=1234
$ ./netcp testfile
```

Finalmente, en ambos modos, si el programa recibe las señales `SIGINT`, `SIGTERM`, `SIGHUP` o `SIGQUIT`, el proceso terminar'a mostrando un mensaje con el código de la señal por la salida de error, liberando los recursos reservados y finalizando con un código de salida distinto de 0:

```
$ ./netcp testfile
netcp: terminando... (signal 15)
```

1.2. Comprobación

Ahora, para la comprobación, necesitamos un archivo de mayor tamaño. Por ejemplo, podemos usar el comando `dd` para crear un archivo de nombre `testfile` con 20 MiB de datos aleatorio:

```
$ dd if=/dev/urandom of=testfile bs=1M count=20 iflag=fullblock
```

Igual que en la parte anterior, podemos usar `cmp` para comprobar que el archivo se enviado correctamente.

2. Implementación

2.1. Modos de funcionamiento

Los dos modos que debe soportar `netcp` se pueden implementar en dos funciones distintas, que se llamarán desde `main()`, en función de los argumentos de la línea de comandos. Por ejemplo:

```
std::error_code netcp_send_file(const std::string& filename);
std::error_code netcp_receive_file(const std::string& filename);
```

La función `netcp_send_file()` se invoca en el modo normal –en el que se envía el contenido del archivo– y es similar al programa que desarrollamos en la parte anterior.

Mientras que `netcp_receive_file()` se utiliza en el modo de escucha –en el que se recibe el contenido del archivo–. En este caso, la función abre el archivo de destino en modo escritura, espera a recibir datos por el *socket* con `receive_from()` y los escribe en el archivo usando `write_file()`.

Las funciones `receive_from()` y `write_file()` deben implementarse de forma similar a `send_to()` y `read_file()`, tal y como se indica en las Secciones 5.7 y 4.3.

Además, al crear el *socket* en `netcp_receive_file()` con `make_sockets()`, no debemos olvidarnos de asignarle un puerto para la escucha. Y recuerda que en ambos modos, el puerto y la IP se pueden configurar usando las variables de entorno `NETCP_PORT` y `NETCP_IP`.

2.2. Copia completa del archivo

Como ahora el archivo puede tener cualquier tamaño, no podemos suponer que se va a leer de una sola vez.

Como se indica en la Sección 4.3, tendremos que llamar a `read_file()` en bucle, hasta que no se puede leer más. En cada iteración, se envía el contenido del buffer usando `send_to()`.

Al mismo tiempo, cada fragmento es recibido en el otro proceso usando `receive_from()` en bucle. En cada iteración, el contenido del buffer se escribe en el archivo de destino usando `write_file()`.

En condiciones normales, el bucle de lectura del archivo en `netcp_send_file()` debe terminar cuando ya no se puedan leer más datos del archivo.

Para que `netcp_receive_file()` sepa que el otro proceso ha terminado de enviar datos y que puede dar por terminada la copia y salir del bucle de escritura, vamos a usar un truco que consiste en que `netcp_send_file()` envíe un mensaje vacío después del último mensaje con datos del archivo.

De esta forma, cuando `netcp_receive_file()` reciban un mensaje de tamaño cero, sabe que la copia ha terminado.

2.3. Señales

El manejo de señales debe configurarse en las primeras líneas de tu programa, antes de que `main()` llame a `netcp_send_file()` o `netcp_receive_file()`.

En el tema 10 de los apuntes de la asignatura y en el repositorio de código, se pueden ver algunos ejemplos de cómo se usa `sigaction()` para instalar un manejador de señales.

2.3.1. Mensaje en el manejador de señales

En la función manejadora de señales se puede usar `write()` con el descriptor de archivo `STRDERR_FILENO` para mostrar el mensaje indicado, con el número de señal, por la salida de error. Recuerda que no puedes usar `std::iostream`, `std::fprtinf()` ni funciones similares, porque no son reentrantes, por lo que no es seguro usarlo en un manejador de señales.

2.3.2. Terminación del programa

Cuando llega una señal, la ejecución del programa salta a la función manejadora y luego vuelve a continuar por donde estaba. Por tanto, para que el programa termine, la función manejadora debe indicar de alguna forma al resto del código que abandone la tarea que esté haciendo y salga hacia `main()`, para terminar.

Esto se puede hacer usando una variable `quit_requested` de tipo `std::atomic<bool>`, que la función manejadora pone a `true` y el resto del código compruebe periódicamente. Por ejemplo, los bucles en `netcp_send_file()` y `netcp_receive_file()` pueden comprobar el valor de la variable `quit_requested` en cada iteración y salir del bucle si es `true`.

> **Nota**
>
> En el tema 13 de los apuntes de la asignatura, se puede ver un pequeño ejemplo del uso de `std::atomic`.

Sin embargo, esto no funciona correctamente si el proceso está *esperando* en alguna llamada al sistema, pues no tiene oportunidad de ejecutarse en la CPU comprobar el valor de la variable `quit_requested`.

Por ejemplo, cuando se ejecuta el programa en el modo de escucha, el proceso se bloquea en `recvfrom()`, esperando a recibir datos. Si nunca ejecutamos el otro programa para que envíe datos, el proceso nunca volverá de `recvfrom()`, por lo que el bucle de lectura de `netcp_receive_file()` nunca se enterará del cambio en la variable `quit_requested`, si le enviamos una señal.

Para solucionar esto, es importante instalar el manejador sin indicar `SA_RESTART` en el campo `sa_flags` de la estructura `sigaction`. Esto es así, porque si se indica `SA_RESTART`, las funciones de la librería del `read()`, `write()`, `sendto()` y `recvfrom()` no se interrumpirán después de que el proceso reciba la señal, por lo que el programa no tendrá oportunidad de comprobar la activación de la variable que señala la terminación del programa.

Por el contrario, si no se indica `SA_RESTART`, estas funciones terminarán inmediatamente, devolviendo el error `EINTR` en `errno`. Esto ofrece una oportunidad para que el programa comprueba la activación de la variable que señala la terminación del programa y retorne hacia `main()` para terminar.

---

3. Recomendaciones y consejos

Vamos a comentar algunas recomendaciones y consejos generales que pueden ayudar en el desarrollo de la práctica. Te recomendamos que los tengas presentes mientras lees y resuelves las distintas partes.

3.1. Evita el mal uso de los espacios de nombres

Aún hoy en día es frecuente encontrar en libros, blogs o en webs, como Stack Overflow, ejemplos similares al siguiente, en cuanto al uso de los espacios de nombres:

```
1  #include <cstdlib>
2  #include <iostream>
3
4  using namespace std;
5
6  int main(int argc, char* argv[])
7  {
8      cout << "¡Hola, mundo!\n";
9      return EXIT_SUCCESS;
10 }
```

El uso de `using namespace std` de forma global –tal y como se puede observar en la línea 4 del ejemplo anterior– es una mala práctica, según la comunidad de desarrolladores de C++.

Los espacios de nombre están para evitar la colisión de nombres entre clases y funciones. Cuanto más complejo es nuestro programa, más probable es que estas colisiones ocurran, de formar que lo mejor es usar simplemente `std::` –y otros espacios de nombre– donde sea necesario:

```
#include <cstdlib>
#include <iostream>

int main(int argc, char* argv[])
{
    std::cout << "¡Hola, mundo!\n";
    return EXIT_SUCCESS;
}
```

Un ejemplo de este problema lo ilustran perfectamente la función de la librería de sistema `bind()` –en sistemas que soportan *sockets*– y la función de la librería estándar de C++ `std::bind()`. Cuando se usa `using namespace std` y se invoca a `bind()` en alguna parte del código, el compilador puede acabar llamando a una función diferente a la que nos interesaba. Por eso es preferible ser explícitos con los espacios de nombres.

> **Nota**
>
> En todo caso, puede ser buena idea poner tu código en un espacio de nombres propio, para evitar conflictos con las funciones de algunas librerías.

3.2. Aprovecha las ventajas de C++ todo lo posible

Nuestro objetivo es aprender cómo funciona el sistema operativo. Así que usaremos, necesariamente, la interfaz de programación de aplicaciones del sistema operativo, que está escrita para C, pero no es necesario desarrollar toda la práctica en C. Al contrario, es recomendable crear clases y métodos que envuelvan esa interfaz de bajo nivel para acceder los servicios del sistema operativo de forma más sencilla desde C++.

C++ implementa una extensa librería estándar que es de gran ayuda para los programadores. Utilizarla todo lo posible para desarrollar la lógica del programa, con toda seguridad nos quitará mucho trabajo.

3.2.1. Cadenas de caracteres

Por ejemplo, la función de la librería del sistema que devuelve el valor de una variable de entorno en el contexto del proceso actual, tiene este prototipo:

```
char* getenv(const char* name);
```

Pero usar y manipular cadenas `char*` de C puede ser bastante tedioso. Por eso es mucho mejor envolver la función anterior con una versión que trabaje con `std::string`, para así aprovechar más fácilmente las comodidades de las cadenas de C++:

```
std::string getenv(const std::string& name)
{
    char* value = getenv(name.c_str());
    if (value) {
        return std::string(value);
    }
    else {
        return std::string();
    }
}
```

Al llamar a esta nueva versión, se obtiene una cadena con el valor de la variable, si la variable existe y tiene un valor. En caso contrario, devuelve una cadena vacía.

3.2.2. Argumentos de la línea de comandos

Como ya sabemos, los argumentos de la línea de comandos se reciben en `main()` mediante `argv`, un array de punteros a `char`:

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

Para facilitar su procesamiento, `argv` se puede convertir en un vector de `std::string_view`:

```
std::vector<std::string_view> args(argv + 1, argv + argc);
```

Usamos `std::string_view` porque nos ofrece una funcionalidad similar a la de `std::string` pero sin hacer una copia de cada cadena. Es decir, los argumentos de la línea de comandos están en `argv`, pero accedemos a ellos a través de los objetos `std::string_view` en `args`

Con este vector, es muy sencillo iterar para procesar los argumentos de la línea de comandos uno tras otro:

```
for (auto it = args.begin(), end = args.end(); it != end; ++it)
{
    if (*it == "-h" || *it == "--help")
    {
        show_help = true;
    }

    if (arg == "-o" || arg == "--output")
    {
        if (++it != end)
        {
            output_filename = *it;
        }
        else
        {
            // Error por falta de argumento...
        }
    }

    // ...
}
```

> **Nota**
>
> Obviamente, es mucho más sencillo usar un *for-range*, pero nos dará problemas si tenemos opciones que requieren un argumento, como es el caso de `-o` en el ejemplo anterior.

Para dividir la responsabilidad, se puede crear una estructura con las posibles opciones del programa y una función que procese los argumentos y retorne al terminar la estructura con los valores de las opciones:

```
struct program_options
{
    bool show_help = false;
    std::string output_filename;
    // ...
};

std::optional<program_options> parse_args(int argc, char* argv[])
{
    std::vector<std::string_view> args(argv + 1, argv + argc);
    program_options options;

    for (auto it = args.begin(), end = args.end(); it != end; ++it)
    {
        if (*it == "-h" || *it == "--help")
        {
            options.show_help = true;
        }

        if (*it  == "-o" || *it  == "--output")
        {
            if (++it != end)
            {
                options.output_filename = *it;
            }
            else
            {
                std::cerr << "Error...\n"                          (1)
                return std::nullopt;
            }
        }

        // Opciones adicionales...
    }
```

```
    return options;                                                    ②
}
```

① En caso de error, se puede mostrar un mensaje por la salida de error y retornar `std::nullopt`, para indicar que el procesado de las opciones de línea de comandos no pudo realizarse con éxito.
② Si todo ha ido bien, en el valor de retorno se devuelve la estructura con los argumentos y *flags* de las opciones encontradas.

La función `parse_args()` devuelve un `std::optional`[1] para indicar al mismo tiempo el éxito o error de procesar la línea de comandos y el resultado de dicho trabajo, en una estructura `program_options`.

Este función se puede invocar fácilmente desde `main()`:

```
int main(int argc, char* argv[])
{
    auto options = parse_args(argc, argv);                             ①
    if (! options)                                                     ②
    {
        return EXIT_FAILURE;                                           ③
    }

    // Usar options.value() para acceder a las opciones...
    if (options.value().show_help)                                     ④
    {
        print_usage();
    }

    // ...

    return EXIT_SUCCESS;
}
```

① Llamar a `parse_args()` para procesar los argumentos de línea de comandos en `argc` y `argv`.
② Comprobar si `parse_args()` terminó con éxito, comprobado el objeto `std::optional` devuelto.
③ En caso de error, terminar el programa.
④ La estructura `program_options` con el resultado de procesar la línea de comandos se puede obtener llamando al método `value()` del objeto `std::optional` devuelto. Como se ilustra en el ejemplo, así se puede acceder a `program_options::show_help` para comprobar si el usuario indicó que quería leer la ayuda del programa.

[1] `std::optional` está disponible desde gcc-11 (C++17) y se declara en `<optional>`.

> **Nota**
>
> Si se quiere que `parse_args()` devuelva diferentes errores en función del problema encontrado, se puede usar `std::expected` en lugar de `std::optional`, como veremos en la Sección 3.3.2. En ese caso, el `std::expected` devuelto por `parse_args()` puede contener un `program_options` en caso de éxito o un `enum class` con el motivo del error en caso de fallo.

3.2.3. Buffers

Muchas de las funciones de la librería del sistema necesitan que se les indiquen la dirección y el tamaño de un buffer donde almacenar el resultado de una operación u obtener los datos necesarios para realizarla. Por ejemplo, en las funciones `read()` y `write()`, que se utilizan para leer y escribir en archivos:

```
ssize_t read(int fd, void* buf, size_t count);
ssize_t write(int fd, const void* buf, size_t count);
```

El argumento `buf` debe usarse para indicar un puntero al buffer de memoria donde se guardarán los bytes leídos o se tomarán los bytes que deben ser escritos. Mientras que `count` es el tamaño del buffer, en bytes. Es decir, el máximo de bytes que se pueden leer del archivo y guardar en el buffer –cuando la operación es `read()`– o el número de bytes del buffer que se quieren escribir en el archivo –si la operación es `write()`–.

En C++ la forma recomendada de crear un buffer de tamaño fijo es con `std::array`:

```
std::array<uint8_t, 240> buffer;
ssize_t bytes_read = read(fd, buffer.data(), buffer.size());
if (bytes_read < 0)
{
    // Manejar error en read()...
}
```

Sin embargo, debemos tener presente que los buffers `std::array` se crean en la **pila** del hilo y que esta tiene un tamaño limitado –en muchos sistemas, 8 MiB–.

> **Nota**
>
> En sistemas Linux puedes conocer el tamaño por defecto de la pila, ejecutando el comando `ulimit -s`. El valor devuelto está en KiB, no en bytes.

Para reservar cantidades más grandes de memoria en el **montón** o cuando nos conviene que el buffer tenga un tamaño variable, es mejor usar `std::vector`:

```cpp
std::vector<uint8_t> buffer(16ul * 1024 * 1024);
ssize_t bytes_read = read(fd, buffer.data(), buffer.size());
if (bytes_read < 0)
{
    // Manejar error en read()...
}
buffer.resize(bytes_read);                                          (1)
```

(1) Reducir el tamaño del vector a la cantidad de bytes que `read()` ha leído en realidad, para que solo contenga los bytes válidos.

Como en el caso de `getenv()`, podemos envolver la función `read()` con una versión que trabaje con `std::vector`, para así aprovechar más fácilmente las comodidades de los vectores de C++ en el resto de nuestro código:

```cpp
int read_file(int fd, std::vector<uint8_t>& buffer)
{
    ssize_t bytes_read = read(fd, buffer.data(), buffer.size());
    if (bytes_read < 0)
    {
        return errno;
    }
    buffer.resize(bytes_read);
    return 0;
}
```

3.3. Manejo de errores

En los ejemplos que hemos visto con `getenv()` y `read()` siempre comprobamos el valor devuelto para detectar si se ha producido algún error.

> **!** Importante
>
> La mayor parte de las funciones que sirven servicios y recursos del sistema pueden fallar por diversos motivos, por lo que debemos comprobar esta condición antes de continuar y tratar de usar su resultado.

En los sistemas POSIX, la mayor parte de las funciones de la librería del sistema devuelven un valor negativo en caso de error, y un valor no negativo en caso de éxito. En el caso de `read()`, el valor devuelto es el número de bytes leídos, o -1 en caso de error. Por ejemplo, observa la comprobación sobre el valor devuelto por `read()` en la línea 4 de nuestra función:

```
1  int read_file(int fd, std::vector<uint8_t>& buffer)
2  {
3      ssize_t bytes_read = read(fd, buffer.data(), buffer.size());
4      if (bytes_read < 0)                                              (1)
5      {
6          return errno;
7      }
8      buffer.resize(bytes_read);
9      return 0;
10 }
```

(1) Comprobación de error de la operación `read()`.

El valor devuelto en caso de error no indica el tipo de error que ha ocurrido. Para obtener un código de error que nos indique el tipo, debemos consultar la macro `errno` (véase la línea 6 del ejemplo anterior). Este código nos ayudará a nosotros o al usuario que ha invocado el programa a identificar el origen del problema. Especialmente con ayuda de la función `strerror()` –o `std::strerror()` de C++– que nos permite obtener un mensaje de texto descriptivo para cada código de error.

> **Nota**
>
> Esta forma de gestionar los errores es muy común en API, librerías y programas en C. Por ejemplo, Windows API define una función `GetLastError()` para obtener el código de error de la última operación fallida. Las funciones de Windows API devuelven NULL o un valor negativo en caso de error al realizar una operación.

El valor de `errno` puede ser cambiado por cualquier función de la librería del sistema. Muchas de las funciones de la librería estándar del lenguaje o de otras librerías pueden hacer uso funciones de la librería del sistema sin que nosotros lo sepamos. Acciones tan sencillas como añadir un elemento a un vector o imprimir un mensaje por la salida estándar, pueden usar la librería del sistema. Por eso, **lo primero que hay que hacer tras detectar que una función de la librería del sistema ha fallado, es guardar el valor actual de `errno`** para preservar el código del error de cambios posteriores.

3.3.1. Propagación de códigos de error

Nuestra versión de `read_file()` devuelve 0 en caso de éxito o un valor distinto de 0 en caso de error, siendo ese valor el código de error de `errno`.

Podríamos hacer que `read_file()` terminase unilateralmente el programa en caso de error, pero eso, generalmente, **es una mala idea**. La función `read_file()` no sabe si en el contexto en el que está siendo llamada, el programa debe terminar o tiene que hacer otra cosa en caso de error.

Por ejemplo, si se usase `read_file()` para implementar una función `copy_file()` que copia un archivo a una ruta de destino, lo recomendable es que `read_file()` devuelva el error para que `copy_file()` pueda comprobar el valor devuelto, manejar adecuadamente la situación y, probablemente, propagar a su vez la condición de error a la función que la invocó a ella. Es decir, si `read_file()` falla, la operación `copy_file()` también fallará y esa condición, así como el motivo del error, debe ser comunicado a la función que invocó a `copy_file()`.

En general, no es buena idea que una función haga terminar unilateralmente el programa en caso de detectar un error. Es más flexible que comunique la situación a quien la invocó, para que así sea más fácil rehusar el código en distintas partes del mismo o en otros proyectos.

> **!** Importante
>
> En el caso de C++, **todo programa debería terminar con un `return` en `main()`** –con un 0 o un valor distinto de 0, en función de si el programa terminó con éxito o no, respectivamente– lo que nos obliga a propagar los errores hasta la función principal del programa.

La necesidad de gestionar y propagar códigos de error de distintas API y librerías es tan común, que C++ ofrece `std::error_code`[2] para facilitar el trabajo.

Por ejemplo, en la siguiente versión de nuestra función `read_file()`, se devuelve un objeto `std::error_code` de la categoría `std::system_category()`, con 0 en caso de éxito o el código de error de `errno` en caso de error:

```
std::error_code read_file(int fd, std::vector<uint8_t>& buffer)
{
    ssize_t bytes_read = read(fd, buffer.data(), buffer.size());
    if (bytes_read < 0)
    {
        return std::error_code(errno, std::system_category());        ①
    }
    buffer.resize(bytes_read);
    return std::error_code(0, std::system_category());                ②
}
```

① Retornar un objeto `std::error_code` con el código del error de `read()`.
② Retornar un objeto `std::error_code` con el valor 0, para indicar el éxito de la operación.

Cada objeto `std::error_code` almacena un código de error y una categoría de errores. El concepto de categoría es importante porque permite usar el mismo valor de error en tipos de error diferente si están en distintas categorías. Por ejemplo, una librería de comunicaciones en red puede usar los mismos valores de error que la librería del sistema, si define su propia categoría de errores.

[2] `std::error_code` se declara en `<system_error>`.

El valor devuelto por `std::system_category()` corresponde a la categoría de *errores del sistema*, donde podemos englobar los errores notificados por la librería del sistema.

Al invocar a `read_file()` es muy sencillo comprobar si ha habido algún error, ya que el objeto `std::error_code` tiene un operador de conversión a `bool` que devuelve `true` si el código de error es 0 y `false` en caso contrario:

```
std::error_code error copy_file(const std::string& src_path,
    const std::string& dst_path)
{
    int src_fd = open(src_path, /* ... */);

    // ...

    std::error_code error = read_file(src_fd, buffer);              ①
    if (error)                                                      ②
    {
        close(src_fd);                                              ③
        return error;                                               ④
    }

    // ...

    close(src_fd);
    return std::error_code(0, std::system_category());              ⑤
}
```

① Llamar a `read_file()` para leer del archivo.
② Comprobar si `read_file()` terminó con éxito comprobado el objeto `std::error_code` devuelto.
③ En caso de error, es necesario manejarlo. Por ejemplo, cerrando y liberando recursos reservados dentro de `copy_file()`, que ya no van a ser necesarios debido a que estamos a punto de salir de la función.
④ Propagar el error al invocador de `copy_file()` usando el valor de retorno. El invocador tendrá que hacer su propio manejo del error y continuar con la propagación del mismo por la pila de llamadas.
⑤ Si todo ha ido bien, cerrar y librear recursos y terminar indicado que se ha tenido éxito.

Además, el objeto `std::error_code` tiene un método `message()` que devuelve un mensaje de texto descriptivo del error y un método `value()` que devuelve el código de error.

```
std::error_code error = read_file(fd, buffer);
if (error)
{
    std::cerr << std::format("Error ({}): {}\n", error.value(),     ①
        error.message());
```

```
}
```

① Usar los métodos `value()` y `message` del objeto `std::error_code` para componer un mensaje de error.

En C es muy común gestionar los errores mediante el retorno de códigos de error y su propagación a través del retorno de las funciones hacia `main()`.

Algunos lenguajes modernos –como Rust o Go– también han optado por esta solución y en C++ hay cierto interés en mejorar su soporte introduciendo algunas ayudas adicionales. El motivo es que obliga a los programadores a tratar los errores de forma explícita, en el punto donde se producen.

En los lenguajes más modernos, incluso se puede impedir que el programador ignore el código de error devuelto, obligando a añadir el código necesario para gestionarlo. En C++ se puede indicar al compilador que muestre un *warning* si el programador olvida leer el código de error devuelto por una función, especificando el atributo `nodiscard` al declararla:

```
[[nodiscard]]
std::error_code read_file(int fd, std::vector<uint8_t>& buffer)
```

> **Nota**
>
> Por claridad, en los guiones de prácticas no usaremos el atributo `nodiscard`. Sin embargo, **recomendamos usarlo** para resolver la práctica, con el objeto de que nos recuerde que **siempre tenemos que comprobar los errores devueltos por las funciones** y manejarlos adecuadamente.

3.3.2. Propagación de errores con valor de retorno

En nuestra función `read_file()` los datos leídos del archivo se devuelve mediante un argumento `buffer` pasado por referencia. Por tanto, no tenemos problema en usar el valor de retorno para el código de error. Sin embargo, ¿qué podemos hacer cuando queremos retornar algún valor en caso de éxito y un código de error en caso de error?.

Por ejemplo, la función `open()` de la librería del sistema se declara así:

```
int open(const char *path, int flags, mode_t mode);
```

Esta función devuelve un descriptor de archivo en caso de éxito y un -1 en caso de error, con el código del error correspondiente guardado en `errno`. Al envolverla en una función en C++, nos gustaría hacerlo un poco mejor. En lugar de usar el *hack* de devolver un entero que sirva tanto para guardar el descriptor de archivo como para indicar que ha ocurrido un error, queremos devolver el descriptor de archivo –un `int`– en caso de éxito o un `std::error_code` con el código de error, en caso de error.

Para poder devolver uno de los dos valores, según el caso, C++ ofrece `std::expected`[3]. Esta clase se utiliza como se muestra en la siguiente función `open_file()`:

```
std::expected<int, std::error_code> open_file(const std::string& path,  ①
    int flags, mode_t mode);
{
    int fd = open(path.c_str(), flags, mode);
    if (fd == -1)
    {
        std::error_code error(errno, std::system_category());
        return std::unexpected(error);                                   ②
    }

    return fd;                                                           ③
}
```

① La función retorna `std::expected<int, std::error_code>`. El primer parámetro de `std::expected` debe ser el tipo del objeto a retornar en caso de éxito, mientras que el segundo es el tipo del objeto para devolver el error.

② En caso de error, devolvemos un objeto `std::expected`. Para que sea un objeto que señala un error, se crea con `std::unexpected()`, pasándole el objeto con la información sobre el error.

③ En caso de éxito, también se devuelve un objeto `std::expected` creado con el valor que queremos que retorne la función, que en este caso es el descriptor de archivos `fd`. Esta línea es equivalente a `return std::expected(fd)`, solo que al intentar retornar `fd` la creación del objeto `std::expected` ocurre de forma implícita, por comodidad.

Al volver de `open_file()`, es muy sencillo comprobar si ha habido algún error, ya que el objeto `std::expected` tiene un operador de conversión a `bool` que devuelve `true` si no hubo error y `false` en caso contrario:

```
std::expected<int, std::error_code> result = open_file("test.txt",   ①
    flags, mode);
if (!result)                                                          ②
{
    return result.error();                                            ③
}

int fd = *result;                                                     ④

std::vector<uint8_t> buffer(1024);
std::error_code error = read_file(fd, buffer);                        ⑤
```

① Llamar a `open_file()` para abrir el archivo.

[3] `std::expected` está disponible desde gcc-12 (C++23) y se declara en `<expected>`.

② Comprobar si `open_file()` terminó con éxito comprobado el objeto `std::expected` devuelto.
③ En caso de error, es necesario manejarlo, por ejemplo, cerrando y liberando recursos reservados. Después se propaga el código de error –el objeto `std::error_code` en `std::expected`– al invocador de la función. Este objeto se puede obtener llamado al método `error()` del objeto `std::expected`.
④ Si todo ha ido bien, se puede usar el operador `*` para acceder al descriptor de archivo almacenado en el objeto `std::expected`.
⑤ El descriptor de archivos se puede usar para leer o escribir en el archivo, entre otras operaciones.

Como los tipos de retorno de las funciones con `std::expected` pueden tener nombres muy largos que dificultan la legibilidad, puede ser conveniente crear alias:

```
using open_file_result = std::expected<int, std::error_code>;          ①

open_file_result open_file(const std::string& path, int flags,         ②
    mode_t mode);
{
    // ...
}

// ...

open_file_result result = open_file("test.txt", flags, mode);
if (result)
{
    int fd = *result;

    // ...
}
```

① Definir un alias para el tipo retornado por `open_file()`.
② Usar el alias para definir la función `open_file()` y para crear un variable en la que guardar el objeto retornado.

O, mejor, usando el atributo `nodiscard` para que no nos olvidemos de guardar el resultado de la función:

```
using open_file_result = std::expected<int, std::error_code>;

[[nodiscard]]
open_file_result open_file(const std::string& path, int flags, mode_t mode);
```

3.3.3. Excepciones de C++

En C++ y otros lenguajes con orientación a objetos, la forma más común de propagar errores es mediante excepciones. Si estás familiarizado con este concepto y lo prefieres, **puedes utilizar excepciones para gestionar los errores en la práctica**. En caso contrario, puedes ignorar el resto de este apartado.

Para utilizar excepciones solo necesitas saber que podemos lanzar una excepción para un código de error de `errno` concreto:

```
throw std::system_error(errno, std::system_category(),
    std::string("__FILE__") + "#" + std::to_string(__LINE__));
```

El tercer argumento del constructor de `std::system_error`[4] es un mensaje de texto que precede al mensaje descriptivo del código de error. En el ejemplo se construye con el nombre del fichero de código fuente –usando el valor de la macro `__FILE__`– y el número de línea –usando el valor de `__LINE__`– donde se ha producido el error. Esto puede ser muy útil para localizar rápidamente los errores al depurar el programa.

Para obtener el mensaje de error completo, podemos usar el método `what()` del objeto de la excepción al capturarla:

```
try
{
    read_file(int fd, buffer);
}
catch (std::system_error& e)
{
    std::cerr << "Error: " << e.what() << '\n';
}
```

3.3.4. Gestión de recursos

Buena parte de las peticiones al sistema operativo son para solicitar recursos que, posteriormente, debemos liberar cuando ya no los necesitemos. Por ejemplo, al abrir un archivo se obtiene un descriptor de archivo, que no es sino un índice en la tabla de archivos abiertos del proceso. Con este descriptor indicamos al resto de llamadas al sistema el archivo sobre el que queremos operar y, cuando ya no lo necesitamos, debemos cerrarlo para liberar el descriptor y los recursos reservados que puedan ser reutilizados por otro proceso.

En el siguiente ejemplo, vemos un caso donde se abren dos archivos, quizás con la idea de copiar el contenido de uno en el otro:

[4] `std::system_error` se declara en `<system_error>`.

```
1  std::error_code error copy_file(const std::string& src_path,
2      const std::string& dst_path)
3  {
4      int src_fd = open(src_path.c_str(), O_RDONLY);                    ①
5      if (src_fd < 0)
6      {
7          return std::error_code(errno, std::system_category());
8      }
9
10     int dst_path = open(dst_path.c_str(), O_WRONLY);                  ②
11     if (dst_fd < 0)
12     {
13         return std::error_code(errno, std::system_category());
14     }
15
16     // Copiar el contenido del archivo src_fd en dst_fd...
17
18     close(src_fd);
19     close(dst_fd);
20 }
```

① Si este `open()` falla, estamos a salvo porque el primer archivo no llegó a abrirse.
② Si falla este `open()`, el archivo origen en `src_path` se quedaría abierto al retornar de la función en la línea 13 sin cerrar antes `src_fd`.

Para evitar estos problemas en C++, debemos utilizar un objeto que se encargue de gestionar el recurso y que se asegure de que se libera cuando ya no lo necesitemos. En el caso de la memoria no es problema porque vamos a usar `std::array`, `std::vector` , `std::string` y otros contenedores, todo lo posible. Estos siempre se encargan de liberar su memoria de forma automática cuando se destruyen.

Para otros tipos de recursos, podemos crear nuestras propias clases o utilizar *scope guards*.

> **Nota**
>
> Si se usa un compilador antiguo, se puede descargar el archivo `scope.hpp` en el proyecto, incluirlo en el programa como `#include <scope.hpp>` y utilizarlo como `scope::scope_exit()`.

Existen varias implementaciones de *scope guards* desde hace tiempo, aunque no se ha incluido una en el estándar hasta C++20 y solo de forma experimental –disponible en algunos compiladores[5] como `std::experimental::scope_exit`–.

[5] `std::experimental::scope_exit` está disponible desde gcc-13 (C++23) y se declara en `<experimental/scope>`.

```
1  using std::experimental::scope_exit;
2
3  std::error_code error copy_file(const std::string& src_path,
4      const std::string& dst_path)
5  {
6      int src_fd = open(src_path, O_RDONLY);                          ①
7      if (src_fd < 0)
8      {
9          return std::error_code(errno, std::system_category());
10     }
11
12     auto src_guard = scope_exit(
13         [src_fd]{ close(src_fd); }                                  ②
14     );
15
16     int dst_path = open(dst_path, O_WRONLY);                        ①
17     if (dst_fd < 0)
18     {
19         return std::error_code(errno, std::system_category());
20     }
21
22     auto dst_guard = scope_exit(
23         [dst_fd]{ close(dst_fd); }                                  ②
24     );
25
26     // Copiar el contenido del archivo src_fd en dst_fd...
27 }
```

① Se crean los objetos `src_guard` y `dst_guard`, que son de tipo `std::experimental::scope_exit`[6]. Como ambas son variables locales, estos objetos se destruyen automáticamente al salir de la función. Antes de hacerlo, llaman a la función que se le pasó como argumento –en las líneas 13 y 23– durante la creación del objeto.

② La función invocada durante la destrucción de los objetos `scope_exit`, llama a `close()` para cerrar los descriptores de archivo, cuando ya no son necesarios.

La sintaxis de las funciones de las líneas 13 y 24 es un poco extraña porque son funciones *lambda*. El cuerpo de la función es lo que se pone entre llaves, mientras que entre corchetes se indican las variables locales de `copy_file()` que deben ser capturadas para que estén disponibles dentro del cuerpo de la función lambda. En el ejemplo, se capturan los descriptores de archivo, para poder usarlos como argumentos de `close()`.

[6] La palabra clave *auto* indica al compilador que desconocemos el tipo de las variables, por lo que queremos que sea él quien lo infiera.

4. Operaciones con archivos

4.1. Abrir archivos

Muchas de las funciones de la librería del sistema para manipular archivo requieren un descriptor de archivo. Este descriptor es un número entero que identifica un archivo abierto en el sistema de archivos y que se puede obtener fácilmente mediante llamando a la función `open()`.

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

La función `open()` abre el archivo cuya ruta se pasa como primer argumento (`pathname`) y devuelve, en caso de éxito, un descriptor de archivo.

4.1.1. Opciones de apertura

El segundo argumento (`flags`) es un conjunto de opciones que indican cómo se va a abrir el archivo. Algunas de estas opciones son:

- `O_RDONLY` abre el archivo para lectura.
- `O_WRONLY` abre el archivo para escritura.
- `O_RDWR`: abre el archivo para lectura y escritura.
- `O_TRUNC` trunca el archivo a 0 bytes al abrirlo.
- `O_APPEND` abre el archivo para escritura y coloca el puntero de escritura al final del archivo. Esto es útil cuando sabemos que queremos añadir datos al final del archivo.
- `O_CREAT` crea el archivo si no existe. Si no se indica esta opción, y el archivo no existe, la función `open()` fallará con `ENOENT`.
- `O_EXCL` en combinación con `O_CREAT`, si el archivo ya existe, la llamada a `open()` falla con `EEXIST`

Se pueden combinar varias de estas opciones usando el operador `'|'` –*or* bit a bit–:

```
int fd = open(filename, O_RDONLY | O_CREAT);
if (fd == -1)
{
    // Error al abrir el archivo
}
```

4.1.2. Permisos de los nuevos archivos

El tercer argumento (`mode`) indica los permisos del nuevo archivo, si es que va a ser creado porque no existía previamente. Obviamente, esta opción solo tiene sentido si se pasa la opción `O_CREAT` en `flags` porque, de lo contrario, el archivo abierto ya existe y conserva sus permisos originales.

Si creamos un archivo nuevo y no indicamos `mode`, el archivo se creará con permisos `0000`, es decir, sin permisos para nadie. Así que lo adecuada es siempre indicar los permisos que queremos que tenga el archivo, para lo que podemos usar la misma notación octal que utilizamos con el comando `chmod` en la shell:

```
int fd = open(filename, O_RDONLY | O_CREAT, 0666);
if (fd == -1)
{
    // Error al abrir el archivo
}
```

El nuevo archivo hereda del proceso que llama a `open()` el propietario y el grupo. Sin embargo, los permisos indicados en `mode` son modificados por la máscara **umask** del proceso, de tal forma que los permisos reales del archivo serán `mode & ~umask`.

La **umask** puede ser configurada por cada usuario mediante el comando `umask` de la shell:

```
$ umask 0022
```

Que también se puede usar para comprobar el valor actual de la máscara:

```
$ umask
0022
```

Todos los procesos lanzados por el usuario heredan la **umask** de la shell, lo que permite que los procesos controlen los permisos que tendrán los archivos que creen. Por ejemplo, si en nuestro programa abrimos un nuevo archivo con `mode = 0666` y `umask` es `0022`, el archivo tendrá permisos `0666 & ~0022 = 0644`. Es decir, por defecto, los nuevos archivos no tendrán permisos de escritura ni para el grupo y ni para otros.

> **Nota**
>
> No tiene sentido que indiquemos permisos de ejecución en `mode` –por ejemplo, `0777`– a menos que estemos desarrollando un compilador y sepamos que el archivo que vamos a crear va a ser un ejecutable.

4.2. Cerrar descriptores de archivo

Los descriptores de archivo se cierran usando la función `close()`.

El motivo de error más común es intentar cerrar un descriptor inválido o que ya ha sido cerrado.

4.3. Leer y escribir datos

Las funciones `read()` y `write()` permiten leer y escribir datos de un archivo abierto. Solo hacer falta indicarle el descriptor del archivo abierto, el buffer donde se almacenarán los datos y el tamaño máximo de los datos a leer o escribir. Lo que no debemos perder de vista es el valor de retorno de estas funciones, que indica el número de bytes leídos o escritos por `read()` o `write()`, respectivamente, cuando no se produce ningún error.

Hay varios motivos por los que la función puede escribir menos bytes de los que se le han indicado, cuando se usa `write()` para escribir en un *socket* o en una tubería. Sin embargo, en el caso de un archivo, lo más común es que se deba a que no queda espacio suficiente en el dispositivo. Lo correcto es comprobar el valor de retorno de `write()` y si es menor que el tamaño del buffer indicado, volver a llamar a `write()` con el resto de bytes que quedan por escribir. Si el problema persiste, entonces sí retornara con un -1 y el valor de `errno` indicará el error –por ejemplo, `ENOSPC`–.

Como comentamos en la Sección 3.2.3, es recomendable envolver la llamada a `write()` en nuestra propia función `write_file()` que implemente la lógica comentada de repetir la llamada hasta que se hayan escrito todos los bytes de `buffer` o se produzca un error:

```
std::error_code write_file(int fd, const std::vector<uint8_t>& buffer);
```

Por otro lado, la función `read()` retornará el número de bytes realmente leídos y también recomendamos crear nuestra propia función `read_file()`:

```
std::error_code read_file(int fd, const std::vector<uint8_t>& buffer);
```

El valor de retorno de `read()` es útil para redimensionar el vector al tamaño adecuado, tal y como se muestra en Sección 3.2.3.

Tendremos que leer los datos en un bucle, ya que no podemos asegurar que la primera llamada a `read()` devuelva todos los bytes que queremos leer, ni que el archivo al completo quepa en el buffer que hemos reservado. Cuando `read()` retorne 0 –por lo que el `buffer` devuelto por `read_file()` estará vacío– habremos alcanzado el final del archivo, por lo que ya no hay más datos que leer. Entonces podremos cerrar los descriptores de ambos archivos abiertos.

4.4. Acceder a los atributos de un archivo

Las funciones `stat()` y `fstat()` ofrecen una manera de comprobar la existencia de un archivo y de obtener acceso a sus atributos para obtener información como:

- Tipo de archivo.

- Tamaño y número de bloques ocupados.
- Número de inodo.
- Propietario y grupo.
- Permisos.
- Fechas de acceso y modificación.

```
int stat(const char* file_name, struct stat* buf);
int fstat(int filedes, struct stat* buf);
```

Si `stat()` o `fstat()` fallan con el error `ENOENT`, es que el archivo por el que se pregunta no existe.

Ambas versiones de `stat()` reciben un puntero a una estructura `stat`, que se rellena con la información del archivo al volver de la llamada. La única diferencia entre `stat()` y `fstat()` es que `stat()` recibe la ruta del archivo y `fstat()` recibe el descriptor de archivo que debemos abrir previamente.

La estructura `stat` contiene información sobre el archivo, como el número de inodo, los permisos, el tamaño, el número de enlaces, el propietario, el grupo o las fechas de acceso y modificación:

```
struct stat {
    dev_t         st_dev;      /* dispositivo */
    ino_t         st_ino;      /* inodo */
    mode_t        st_mode;     /* protección y tipo */
    nlink_t       st_nlink;    /* número de enlaces físicos */
    uid_t         st_uid;      /* ID del usuario propietario */
    gid_t         st_gid;      /* ID del grupo propietario */
    dev_t         st_rdev;     /* tipo dispositivo (si es
                                  dispositivo inodo) */
    off_t         st_size;     /* tamaño total, en bytes */
    blksize_t     st_blksize;  /* tamaño de bloque para el
                                  sistema de ficheros de E/S */
    blkcnt_t      st_blocks;   /* número de bloques asignados */
    time_t        st_atime;    /* hora último acceso */
    time_t        st_mtime;    /* hora última modificación */
    time_t        st_ctime;    /* hora último cambio */
};
```

4.4.1. Comprobar si dos archivos son el mismo

Cada archivo tiene un número de inodo único que lo identifica en el sistema de archivos. Un mismo archivo puede ser accesible por medio de varias rutas, por lo que podemos comparar los números de inodos de archivos en rutas diferentes para determinar si son el mismo archivo.

El inconveniente es que el número de inodo solo es único dentro del mismo sistema de archivos, es decir, que dos archivos pueden tener el mismo número de inodo en sistemas de archivos diferentes. Por eso necesitamos comparar tanto `st_dev` –que identifica el dispositivo donde se encuentra el archivo– como `st_ino` –que identifica el archivo dentro del dispositivo– de la estructura `stat`, para saber si dos rutas conducen al mismo archivo.

4.4.2. Acceso a los permisos y al tipo de archivo

Tanto los permisos como el tipo del archivo se almacenan en el campo `st_mode` de la estructura `stat`. Para comprobar si un archivo es de un tipo en particular, se puede utilizar alguna de las siguientes macros:

- `S_ISLNK(m)`: ¿Es un enlace simbólico?
- `S_ISREG(m)`: ¿En un fichero regular?
- `S_ISDIR(m)`: ¿Es un directorio?
- `S_ISCHR(m)`: ¿Es un dispositivo de caracteres?
- `S_ISBLK(m)`: ¿Es un dispositivo de bloques?
- `S_ISFIFO(m)`: ¿Es una tubería con nombre (*FIFO*)?
- `S_ISSOCK(m)`: ¿Es un *socket* de dominio UNIX con nombre?

Por ejemplo:

```
stat st;
if (stat(filepath, &st) == -1)
{
    // Error al obtener los atributos de 'filepath'
}
else {
    if (S_ISLNK(st.st_mode))
    {
        // 'filepath' es un enlace simbólico
    }
    else {
        // 'filepath' no es un enlace simbólico
    }
}
```

La macro `S_IFMT` contiene una máscara con todos bits que sirven para indican el tipo de archivo a 1. Es decir, que podemos usar `S_IFMT` para extraer por separado los bits de tipo y los permisos del archivo:

```
mode_t filetype = st.st_mode & S_IFMT;
mode_t file_permissions = st.st_mode & ~S_IFMT;
```

5. Operaciones con sockets

5.1. Sockets

Un *socket* es una abstracción que representa un extremo de un canal de comunicación entre dos procesos en una red de ordenadores. Cada proceso puede utilizar su *socket* para recibir mensajes de otros procesos o para enviarlos a través de dicho canal de comunicación, sin tener que entrar en los detalles de cómo se usa el protocolo de red escogido, ni cómo se gestionan los dispositivos de red involucrados en la comunicación. Estos aspectos son responsabilidad del sistema operativo y del hardware de la red, pero no del programador de las aplicaciones.

Como vimos en el tema 10 de los apuntes de la asignatura, un *socket* se crea con la función de la librería del sistema `socket()` y devuelve un descriptor de archivo, con el que identificar al *socket* en cuestión en futuras operaciones que queramos hacer con él.

```
#include <sys/types.h>
#include <sys/socket.h>

int fd = socket(domain, type, protocol)
```

Tal y como indica el manual de socket, el significado de los parámetros de la función es el siguiente:

- `domain` es el dominio o familia de la dirección. Indica el tipo de tecnología de comunicación que queremos que utilice nuestro socket. Valores posibles son:
  - `AF_INET`, si queremos usar TCP/IP
  - `AF_INET6`, si queremos usar TCP/IPv6
  - `AF_UNIX`, si queremos usar un tipo de socket local a la máquina, similar a las tuberías. **Nosotros usaremos AF_INET porque estamos interesados en la tecnología TCP/IP utilizada en Internet**.
- `type` indica el tipo de socket que nos interesa según los requisitos de la aplicación. Valores posibles son:
  - `SOCK_STREAM`, si domain es AF_INET, indica que el socket es para comunicaciones orientadas a conexión con TCP
  - `SOCK_DGRAM`, si domain es AF_INET, indica que queremos UDP.
  - `SOCK_RAW`, indica acceso directo a los paquetes IP. **Tal y como hemos comentado, nosotros usaremos SOCK_DGRAM.**
- `protocol` indica el protocolo específico que queremos que sea utilizado internamente por el socket para el envío de los datos. Esto es útil con algunas familias o dominios que soportan varios protocolos para cada tipo de socket. En nuestro caso la familia IP (`AF_INET`) con sockets tipo datagram (`SOCK_DGRAM`) solo admite UDP como protocolo, por lo que este argumento debe ser 0.

Con todo esto, crear nuestro primer socket sería algo así

```
#include <sys/socket.h>

int fd = socket(AF_INET, SOCK_DGRAM, 0);
if (fd < 0)
{
    // Error al crear el socket
}
```

Observa que siempre comprobamos si hemos tenido éxito antes de continuar. Existen muchos motivos por los que puede fallar la función `socket()`.

5.2. Cerrar descriptores de sockets

Los descriptores de archivo de *sockets* se cierran usando la función `close()`.

El motivo de error más común es intentar cerrar un descriptor inválido o que ya ha sido cerrado.

5.3. Asignar una dirección al socket

Para poder mandar un mensaje al *socket* de otro proceso, el *socket* destinatario debe tener una dirección única en la red. En la tecnología TCP/IP (`AF_INET`) la dirección de un *socket* se indica por la dirección IP de la máquina y un número de puerto entre 1 y 65535.

> **Nota**
>
> Si no se asigna un puerto y una dirección IP a un *socket* y se usa para enviar mensajes, el sistema operativo escogerá un puerto libre y una de las direcciones IP de la máquina local. Sin embargo, para que un *socket* pueda recibir mensajes, es necesario que primero se le asigne un puerto y una dirección IP, que también debe ser conocida por los procesos que quieren enviar mensajes a este destinatario.

Una dirección se asigna a un *socket* que acabamos de crear mediante la función `bind()` de la librería del sistema.

```
int bind(int fd, const sockaddr* addr, socklen_t addrlen);
```

El primer argumento `fd` de la función es el descriptor de archivo del socket previamente creado con la función `socket()`, mientras que `addr` y `addrlen` indican la dirección que queremos asignar al *socket*.

La interfaz de socket fue diseñada con una interfaz genérica que debíA dar cabida a todas las tecnologías de comunicación existentes y futuras. Por eso `bind()` no acepta directamente una dirección IP y un número de puerto como parámetros. En su lugar, recibe un puntero

a una estructura [`sockaddr`] genérica que pueda dar cabida a cualquier tipo de dirección de cualquier tecnología existente.

Según la familia de protocolos con la que queramos trabajar, se usa una versión de `sockaddr` adecuada para las direcciones de red de esa familia. Por ejemplo, si queremos trabajar con la familia `AF_INET` tendremos que utilizar la estructura `sockaddr_in`, declarada en `netinet/ip.h`, para crear las direcciones:

```
struct sockaddr_in {
    sa_family_t     sin_family;                                    ①
    in_port_t       sin_port;                                      ②
    struct in_addr  sin_addr;   // Estructura con la dirección IP
};
```

① Tiene que valer `AF_INET` siempre.
② Indica el número de puerto, entre 1 y 65535:

- En arquitecturas x86, el número de puerto debe pasar por la función `htons()` y asignar a `sin_port` el valor devuelto.
- Si se indica 0, el sistema operativo asignará un puerto libre cualquiera.

La estructura `in_addr` del campo `sin_addr` se declara como:

```
struct in_addr {
    uint32_t        s_addr;                                        ①
};
```

① Dirección IP en formato entero de 32 bits:

- Se puede utilizar la función inet_aton() para convertir cadenas con direcciones de la forma '192.168.200.5' en estos enteros de 32 bits que necesita la estructura. Esta función retorna 0 si la cadena no contiene una dirección IP válida.
- Si se indica INADDR_LOOPBACK, es para asignar la dirección 127.0.0.1 de la interfaz de red loopback de la máquina local.
- Si se indica INADDR_ANY –o dirección 0.0.0.0– es porque queremos que el *socket* esté abierto en todas las direcciones IP de la máquina local.

Veamos un ejemplo para asignar dirección a nuestro socket de forma que escuche en todas las direcciones IP de nuestra máquina en un puerto escogido por el sistema operativo:

```
sockaddr_in local_address{};                                       ①
local_address.sin_family = AF_INET;                                ②
local_address.sin_addr.s_addr = htonl(INADDR_ANY);                 ③
local_address.sin_port = htons(0);                                 ④

int result = bind(
```

```
    sock_fd,                                                    ⑤
    reinterpret_cast<const sockaddr*>(&local_address),          ⑥
    sizeof(local_address)                                       ⑦
)
if (result < 0)
{
    // Error al asignar una dirección
}
```

① *Value-initializacion* para inicializar la zero la estructura antes de asignarle valores, tal y como se recomienda. Si programamos en C, se puede utilizar `memset()` para poner todos los bytes de la estructura a 0.
② Siempre se asigna `AF_INET` porque el *socket* es de ese dominio, así como la estructura `sockaddr_in` que se está utilizando.
③ Asignar todas las direcciones IP locales del equipo.
④ Asignar un puerto libre cualquiera. En este caso, como el número es 0, es indiferente usar `htons()` o simplemente asignar un 0.
⑤ Descriptor del *socket* al que se pide asignar la dirección `local_address`.
⑥ La conversión de tipos `reinterpret_cast<const sockaddr*>()` es necesaria porque `bind()` espera un puntero al formato genérico de direcciones `sockaddr`, pero `local_address` es `sockaddr_in`.
⑦ Tamaño de la estructura que contiene la dirección.

5.4. make_ip_address() y ip_address_to_string()

Recomendamos implementar `make_ip_address()`, una función encargada de crear direcciones `sockaddr_in`, a partir de una cadena con la dirección IP y un número de puerto, y utilizarla en todos los lugares de la práctica dónde sea necesario:

```
std::optional<sockaddr_in> make_ip_address(
    const std::optional<std::string> ip_address, uint16_t port=0);
```

Las posibles formas de usarla serían:

```
auto address = make_ip_address(nullopt);                        ①
auto address = make_ip_address(nullopt, 8080);                  ②
auto address = make_ip_address("192.168.10.2");                 ③
auto address = make_ip_address("192.168.10.2:8080");            ④
auto address = make_ip_address("192.168.10.2", 8080);           ⑤
auto address = make_ip_address("192.168.10.2:8080", 1234);      ⑥
```

① Devuelve una dirección con `sin_port` a 0 y `sin_addr.s_addr` a `INADDR_ANY`. Por tanto, se trata de un *socket* en cualquier dirección IP de la máquina local y con un puerto escogido por el sistema operativo.

② Devuelve una dirección con `sin_port` a 8080 y `sin_addr.s_addr` a `INADDR_ANY`.
③ Devuelve una dirección con `sin_port` a 0 y `sin_addr.s_addr` a con la dirección IP indicada.
④ Devuelve una dirección con `sin_port` a 8080 y `sin_addr.s_addr` con la dirección IP indicada. Es bastante típico que el número de puerto se pueda indicar en la misma cadena que la dirección IP, usando el caracter `':'` como separador.
⑤ También devuelve una dirección con `sin_port` a 8080 y `sin_addr.s_addr` con la dirección IP indicada.
⑥ Debería devolver un error, porque se ha indicado el número de puerto de dos maneras incompatibles.

Se recomienda que la función devuelva un `std::optional` porque así, si la cadena indicada en `ip_address` no se puede convertir con `inet_aton()` a una dirección IP válida, el objeto se devuelve vacío.

Igualmente, recomendamos crear una función para convertir una dirección `sockaddr_in` en una cadena, puesto que será útil para depurar el programa:

```
std::string ip_address_to_string(const sockaddr_in& address);
```

La idea es que la cadena devuelta contenga la dirección IP y el número de puerto separados por el caracter `':'` –por ejemplo `"192.168.10.2:8080"`–. Para convertir `sin_addr.s_addr` en una cadena con la dirección IP se puede utilizar la función inet_ntoa().

5.5. make_socket()

También recomendamos implementar `make_socket()` para crear un *socket* –usando `socket()`– y, opcionalmente, asignarle una dirección –usando `bind()`–.

```
using make_socket_result = std::expected<int, std::error_code>;
make_socket_result make_socket(
    std::optional<sockaddr_in> address = std::nullopt);
```

Las posibles formas de usarla serían:

```
auto result = make_socket();                                    ①
if (result)
{
    sock_fd = *result
}

auto address = make_ip_address("192.168.10.2", 8080);
auto result = make_socket(address.value());                     ②
if (result)
{
```

```
    sock_fd = *result
}
```

① Crea un *socket* `AF_INET` usando `socket()` pero no le asigna ninguna dirección.
② Crea un *socket* `AF_INET` usando `socket()` y le asigna la dirección indicada en `address` usando `bind()`.

Esta función devuelve un `std::expected` porque puede retornar un `int` con el descriptor del *socket* o un código de error causado por `socket()` o por `bind()`.

En la Sección 3.3.2 se explica en detalle esta forma de gestionar y propagar los errores.

5.6. Enviar un mensaje

Enviar un mensaje al *socket* de otro proceso desde el nuestro se puede hacer con la función `sendto()`:

```
int sendto(
    int sock_fd,                                    ①
    const void* buffer,
    size_t length,
    int flags,                                      ②
    const sockaddr* dest_addr,                      ③
    socklen_t dest_len                              ④
)
```

① Los primeros tres argumentos son similares a los de la función `write()`: el descriptor, el buffer con los datos que se quieren enviar y el número de bytes en el buffer.
② No usaremos los `flags`, por lo que podemos indicar siempre un 0.
③ Argumento para indicar la dirección del destinatario de nuestro mensaje.
④ Argumento para indicar el tamaño de la estructura a la que apunta el argumento `dest_addr`. Es decir, siempre deberíamos indicar `sizeof(sockaddr_in)`.

Por lo tanto, así es como podríamos mandar un mensaje a otro proceso:

```
sockaddr_in remote_address = make_ip_address("127.0.0.1").value()       ①
std::string message_text("¡Hola, mundo!");

int bytes_sent = sendto(sock_fd,
    message_text.data(), message_text.size(), 0,
    reinterpret_cast<const sockaddr*>(&remote_address),                 ②
    sizeof(remote_address));
if (bytes_sent < 0)
{
    // Error al enviar el mensaje.
```

```
}
```

① Usamos `value()` para recuperar la dirección del objeto `std::optional`. Este método lanza una excepción si el objeto `std::optional` devuelto por `make_ip_address()` está vació.
② Recuerda que necesitamos hacer una conversión de tipos porque nuestras direcciones son de tipo `sockaddr_in` pero `sendto()` espera un puntero a `sockaddr`.

Como comentamos en las Secciones 4.3 y 3.2.3, recomendamos envolver el código que llama a `sendto` y gestionar sus errores en una función. Esta debe recibir como argumentos: el descriptor del *socket*, el mensaje –según los formatos que necesitemos en nuestro programa– y la dirección del destinatario:

```
std::error_code send_to(int fd, const string& message,
    const sockaddr_in& address);
std::error_code send_to(int fd, const std::vector<uint8_t>& message,
    const sockaddr_in& address);
// ...
```

5.7. Recibir un mensaje

Recibir un mensaje en nuestro *socket* enviado desde otro proceso `recvfrom()`:

```
int recvfrom(
    int sock_fd,                                                    ①
    void* buffer,
    size_t length,
    int flags,                                                      ②
    sockaddr* src_addr,                                             ③
    socklen_t* src_len                                              ④
)
```

① Los primeros tres argumentos son similares a los de la función `read()`: el descriptor, el buffer donde copiar los datos recibidos y el número máximo de bytes que caben en el buffer.
② No usaremos los `flags`, por lo que podemos indicar siempre un 0.
③ Argumento donde se guardará la dirección del emisor del mensaje, al retornar de la función.
④ Argumento donde se guardará el tamaño de la estructura copiada en `src_addr`. Es decir, en nuestro caso debería valer siempre `sizeof(sockaddr_in)` a la vuelta de la función.

Por lo tanto, así es como podríamos recibir un mensaje de otro proceso:

```
sockaddr_in remote_address{};                                          ①
socklen_t src_len = sizeof(remote_address);

std::string message_text();                                            ②
message_text.resize(100);

int bytes_read = recvfrom(sock_fd,
    message_text.data(), message_text.size(), 0,
    reinterpret_cast<sockaddr*>(&remote_address),                      ①
    &src_len);
if (bytes_read < 0)
{
    // Error al recibir el mensaje.
}

message_text.resize(bytes_read);                                       ③

// Imprimir el mensaje y la dirección del remitente
std::cout << std::format("El sistema '{}'' envió el mensaje '{}'\n",
    ip_address_to_string(remote_address),
    message.text.c_str()
)
```

① Aquí estará guardada la dirección del emisor del mensaje al volver de `recvfrom()`.
② En este buffer se guardarán los datos leídos.
③ Ajustar el tamaño del `std::string` al número real de bytes leídos.

Nuevamente, recomendamos envolver el código que llama a `recvfrom` y gestionar sus errores en una función, tal y como comentamos en las Secciones 4.3 y 3.2.3. Esta debe recibir como argumentos: el descriptor del *socket*, un buffer para guardar el mensaje –según los formatos que necesitemos en nuestro programa– y una referencia para guardar la dirección del emisor:

```
std::error_code receive_from(int fd, string& message,
    sockaddr_in& address);
std::error_code receive_from(int fd std::vector<uint8_t>& message,
    sockaddr_in& address);
// ...
```