Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2233 Programación Avanzada (2024-2)

Tarea 4

Entrega

- **Tarea y** `README.md`
  - **Fecha y hora oficial (sin atraso):** miércoles 20 de noviembre de 2024, 20:00.
  - **Fecha y hora máxima (2 días de atraso):** viernes 22 de noviembre de 2024, 20:00.
  - **Lugar:** Repositorio personal de GitHub — Carpeta: `Tareas/T4/`.
    El código debe estar en la rama (*branch*) por defecto del repositorio: `main`.
  - **Pauta de corrección**: en este enlace.
  - **Bases generales de tareas (descuentos)**: en este enlace.
  - **Formulario entrega atrasada:** en este enlace. Se cerrará el viernes 22 de noviembre, 23:59.
- **Ejecución de tarea**: La tarea será ejecutada **únicamente** desde la terminal del computador. Además, durante el proceso de corrección, se cambiará el nombre de la carpeta "`T4/`" por otro nombre y se ubicará la terminal dentro de dicha carpeta antes de ejecutar la tarea. **Los *paths* relativos utilizados en la tarea deben ser coherentes con esta instrucción.**

Objetivos

- Utilizar conceptos de interfaces y `PyQt5` para implementar una aplicación gráfica e interactiva.
- Traspasar decisiones de diseño y modelación en base a un documento de requisitos.
- Diseñar e implementar una arquitectura cliente-servidor. Además, en el cliente se debe entender y aplicar los conceptos de *back-end* y *front-end*.
- Aplicar conocimientos de *threading* en interfaces (`Thread` y/o `QThread`).
- Aplicar conocimientos de señales.
- Aplicar conocimientos de creación de redes para comunicación efectiva entre computadores (*networking*).
- Aplicar conocimientos de manejo de *bytes* para seguir un protocolo de comunicación preestablecido.

Índice

1. *DCComparte Archivos*

Los **Zombies**, luego de su exitoso negocio en el ámbito de la comida rápida, sentían un vacío que no podían llenar. A pesar del éxito de *Little DCCaesars*, algo dentro de ellos los llamaba de vuelta a sus raíces. Fue entonces cuando recordaron aquellos tiempos en los que se enfocaban en generar caos y destrucción, desatando el pánico y aterrorizando los jardines del mundo.

Mientras tanto, las plantas mutantes que habían estado disfrutando un tiempo de paz, sintieron una inquietud en sus cuerpos. Algo peligroso se avecinaba y era preocupante. Hernán, quien había estado recopilando información secreta acerca de como fortalecer a las plantas dejó todos sus conocimientos en su servidor antes de realizar un viaje[1].

Las plantas necesitan de tus magníficas capacidades de programación, y te imploran crear un programa capaz de extraer toda la información contenida en dicho servidor. Es por esto que decides crear *DCComparte Archivos*, programa que te permite a ti, y más usuarios, extraer la información de dicho servidor y compartirla entre ustedes. Pero cuidado, los **Zombies** intentarán a toda costa sabotear tus intentos de obtener esta valiosa información.

Figura 1: Logo de *DCComparte Archivos*

[1] Al país del sol naciente 🇯🇵 🥳 🎉

2. Flujo del programa

DCComparte Archivos es un programa que tiene como principal objetivo la descarga de archivos mediante *networking*. Para esto, al momento de iniciar el programa se abre la ventana de inicio. En esta ventana el usuario escribe su nombre de usuario. Este nombre debe pasar por un proceso de validación y verificar que no esté siendo utilizado por otro cliente. Si este ya está ocupado por otro cliente conectado en el servidor, se le notifica al usuario y se solicita otro nombre, en caso contrario el usuario entra exitosamente.

Luego de iniciar sesión , se abre la ventana principal, donde el usuario puede ver al resto de los usuarios conectados en el momento, ademas de poder elegir la sección que desea ver, siendo las opciones, **Mis Descargas** y **Mis Archivos**

En **Mis Descargas** el usuario puede consultar los **archivos disponibles** para descargar, donde se verán los nombres y extensiones de cada archivo disponible, para esto debe apretar el botón correspondiente.

El usuario también puede elegir apretar el botón de **descargar archivo**, donde se abrirá una nueva ventana en la cual debe ingresar el nombre del archivo junto a su extensión (si así lo desea). Luego de confirmar la selección, esta ventana se cerrará y comenzará la descarga. Mientras exista una descarga, se mostrará en pantalla una **barra de progreso** indicando el **porcentaje de descarga** en tiempo real, el cual hace referencia a la cantidad de información que ya ha sido descargada por el usuario. Junto con esto, se mostrará el estado de la descarga, siendo estos **'pausada', 'cancelada' o 'descargando'**. Ademas, el usuario tendrá 3 botones indicando las opciones de pausar, reanudar o cancelar la descarga. Estos botones deben estar relacionados con el estado actual de la descarga, es decir, en caso de que la descarga se este realizando correctamente, el botón de reanudar debe quedar desactivado, En caso de pausar la descarga, el botón de pausa debe quedar desactivado, el cual vuelve a activarse al reanudar la descarga. Luego de finalizar la descarga, todos estos botones deben quedar **desactivados**.

En **Mis Archivos** el usuario podrá ver los archivos disponibles ya descargados, en forma listada. En caso de apretar un archivo, el usuario podrá visualizarlo, es decir, si el archivo tiene extensión **.txt** el usuario podrá ver el contenido y en caso de ser una imagen el usuario podrá verla.

Además, existen varios Eventos aleatorios que pueden suceder en el Servidor. En caso de activarse, estos tendrán alguna consecuencia, ya sea en los archivos enviados, o en el mensaje codificado, que el cliente deberá solucionar para lograr obtener el archivo correctamente descargado.

Por ultimo, se aplicará un flujo de ***Peer2Peer*** para la descarga de archivos. En primera instancia, se mantiene un flujo normal de datos entre servidor y clientes, pero una vez al menos un cliente posea alguno de estos archivos, el servidor puede delegar el envío de archivo a dicho cliente. En otras palabras, durante una comunicación *Peer2Peer*, al cliente que desea el archivo el Servidor Principal le avisa que existe un segundo cliente que ya tiene dicho archivo. En ese caso ambos clientes tendrán que conectarse, para que uno de ellos actúe como un servidor y que gestione el envío del archivo al otro cliente. Mas detalles sobre dicho flujo pueden ser encontrados en la sección de *Networking*.

3. Interfaz Gráfica e interacción

3.1. Modelación del programa

Se evaluarán, entre otros, los siguientes aspectos:

- Correcta **modularización** del programa, esto quiere decir que se debe respetar y seguir una adecuada estructuración entre ***front-end*** y ***back-end***, con un **diseño cohesivo** y de **bajo acoplamiento**.
- Correcto uso de **señales** entre ***back-end*** y ***front-end***, implementación de ***threading*** cuando corresponda.

- Un flujo prolijo a lo largo del programa. Esto quiere decir que el usuario puede navegar sin problemas entre las distintas partes que componen *DCComparte Archivos* solo ejecutando una vez el programa. En otras palabras, un usuario nunca debe quedarse atorado en alguna parte del programa que lo obliga a cerrar *DCComparte Archivos* y volver a ejecutarlo. A modo de ejemplo: si un usuario ingresa a una sección y no se puede mover a otra sección, esto es considerado como una mala implementación.
- Una interfaz interactiva integrada con las funcionalidades pedidas. Esto quiere decir, que se espera que la comunicación y el comportamiento del programa sea visible y controlable desde la interfaz. **No se asignará puntaje** si las funcionalidades solicitadas no son visibles en la interfaz ó si es que estas solo se pueden comprobar en la consola o en el código, a menos que se indique explícitamente lo contrario.
- Generación de las ventanas mediante código programado por el estudiante. Es decir, **no se permite** la creación de ventanas con el apoyo de herramientas como `QtDesigner`, `QML`, entre otros.

3.2. Ventanas

Para la correcta implementación de *DCComparte Archivos*, se espera la creación de al menos **dos ventanas**, cada una con **elementos mínimos** de interfaz que deben estar presentes (detallados a continuación).

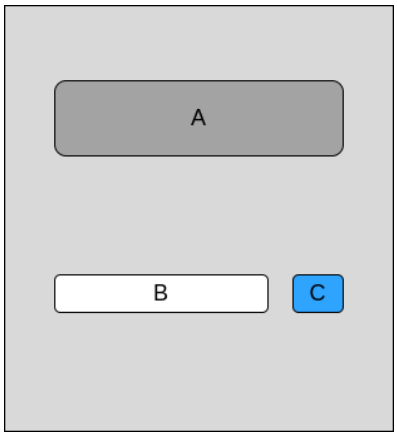
Aclaración importante: los esquemas y diseños que serán expuestos son únicamente referenciales. No es necesario que tu tarea sea una copia exacta de estos: puedes agregar creatividad inventando botones nuevos, interacciones nuevas, diseños y hasta animaciones. Sin embargo, lo único que será evaluado es que los elementos **mínimos** estén presentes y funcionen del modo que se exige (detallado más adelante). Estos elementos mínimos se explicitan en cada sección; pero en ningún caso se les exige cosas relacionadas a la "belleza" de la interfaz gráfica. Por lo tanto, una ventana que sólo tenga los elementos expuestos en los esquemas, tiene el mismo nivel de validez que otra llena de decoraciones y efectos interactivos, esto, suponiendo que ambas tengan el comportamiento deseado. Finalmente cabe destacar que si se desea desarrollar funcionalidades extras, estas serán evaluadas bajo los mismos criterios generales mencionados anteriormente. Esto quiere decir que, si el programa falla debido a una funcionalidad extra, se aplicara el descuento en el *item* correspondiente.

3.2.1. Ventana de inicio de sesión

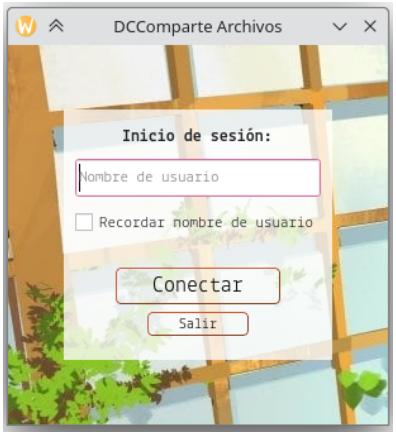
Esta ventana existe antes de llegar a la ventana principal, es la que se muestra cuando se inicia el programa. Se le pide al usuario ingresar un nombre de usuario con el que será identificado una vez conectado al servidor.

Tal como se describe en Flujo del programa, el usuario debe intentar conectarse con un nombre de usuario. Cuando el usuario presione el botón para conectarse, se deberá verificar que el nombre de usuario sea **alfanumérico**, contenga **al menos una mayúscula y un número, y posea un largo entre 3 y 16 caracteres**. Además, se debe verificar que el nombre de usuario no esté siendo utilizado por otro cliente conectado al servidor. En caso de no cumplir con alguna condición se deberá rechazar el ingreso y volver a mostrar la opción para intentar ingresar con otro nombre. El rechazo de inicio de sesión debe ser explícito para el usuario, esto se puede lograr indicando el problema a través de una ventana emergente, o como un aviso en la misma ventana de inicio de sesión.

En Esquema *log-in* se muestra un *mock-up* de cómo podrían distribuirse los elementos mínimos en la ventana de inicio, estos son **(A)** El nombre del programa, **(B)** El editor de línea para el nombre de usuario y **(C)** El botón de ingreso para intentar conectarse. En la figura Diseño *log-in* se muestra un posible diseño alternativo para la misma ventana de inicio de sesión.



(a) Esquema *log-in*



(b) Diseño *log-in*

3.2.2. Ventana principal

Debe tener dos secciones accesibles con funcionalidades distintas: sección **Mis Descargas**, y sección **Mis Archivos**. Estas secciones pueden estar presentes como paneles de una misma ventana, como dos ventanas distintas, como *tabs*, etc. La decisión respecto a cómo navegar entre estas secciones queda a su criterio, sin embargo, siempre debe ser posible acceder de la una a la otra sin consecuencias para el usuario, por ejemplo, no se puede cancelar alguna descarga por salir de la ventana de descargas.

Además de estas dos secciones, debe existir una vista con la información de los usuarios conectados al servidor en tiempo real. Al igual que las secciones anteriores, esta puede ser un panel en la ventana principal, o una ventana aparte. Esta lista de usuarios conectados siempre debe ser visible, independiente de la sección en que se encuentre el cliente.

Cada una de estas secciones tiene un comportamiento mínimo esperado, junto con elementos que deben estar presentes:

3.2.3. Ventana principal: Mis Descargas

Los elementos mínimos son:

- **Botón** "Ver archivos disponibles" : este debe obtener el nombre y extensión de los archivos que pueden ser descargados desde el servidor.
- **Editor de línea**: Debe permitir al usuario escribir el nombre del archivo (con o sin extensión) que desea descargar.
- **Botón** "Descargar archivo" : Cuando se presiona este botón, se debe pedir confirmación del usuario para comenzar la descarga; esto implica el despliegue de otros dos botones, uno para confirmar la descarga (y continuar el proceso) y otro para cancelar (y no descargar el archivo). Si el usuario confirma, se debe comenzar la descarga del archivo escrito por el usuario en el **Editor de línea**.
- **Barra de progreso**: mientras exista una o más descargas, se debe mostrar una barra que haga referencia al porcentaje de información del archivo que ha sido descargada por el cliente. Esta barra debe estar asociada a cada archivo, y no una sola barra para todos los archivos, es decir, deben existir tantas barras de progreso como archivos descargándose.
- **Estado** de la descarga: debe ser un indicador (por ejemplo, un *label*) que muestre cuál es el estado de la descarga. Puede ser "Pausada", "Cancelada", "En progreso", "Completada".

- **Controladores** para la descarga: estos deben ser tres botones distintos que le ofrezcan al usuario la posibilidad de **pausar**, **reanudar** ó **cancelar** la descarga de un archivo que ya está en progreso. Luego de que un archivo se haya descargado por completo, estos controladores deben quedar deshabilitados para ese archivo.

En Esquema Mis Descargas se muestra una posible distribución de los elementos mínimos solicitados: **(A)** Listado de descargas, **(B)** Listado de archivos disponibles en servidor, **(C)** Listado de usuarios conectados, **(D)** Botones para ver archivos disponibles, solicitar descarga, pausar, reanudar, cancelar, etc. En Diseño Mis Descargas se muestra un posible diseño para la vista de descargas.

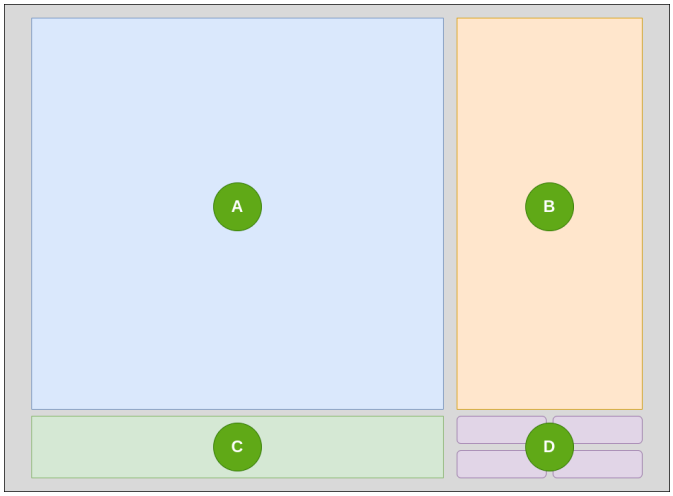


Figura 3: Esquema Mis Descargas

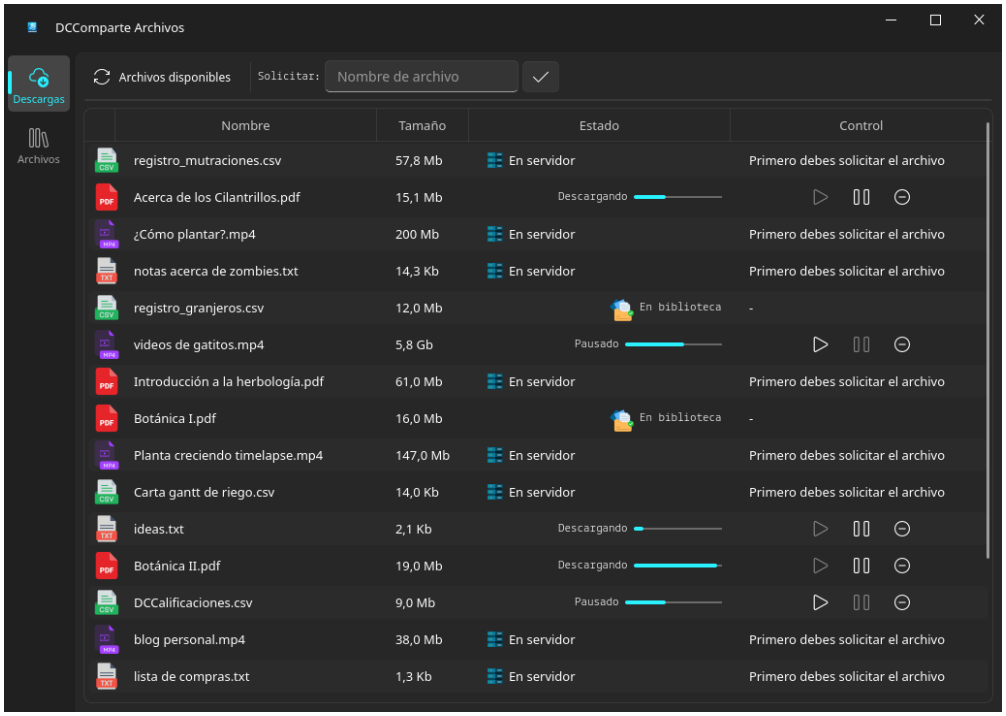


Figura 4: Diseño Mis Descargas

3.2.4. Ventana principal: sección Mis Archivos

En esta sección se deben mostrar, como mínimo, los archivos descargados. Pueden mostrarse en un formato de lista u otro de su preferencia, pero siempre debe ser posible ver el nombre del archivo y su tipo. Además, debe ser posible abrir un archivo descargado para mostrar todo su contenido en la interfaz: texto en caso de `.txt` o `.csv`, imagen en caso de `.png` ó `.jpeg`, sonido en caso de `.mp3`.

Para archivos cuya extensión sea distinta a alguna de las listadas anteriormente, se debe decidir y explicitar en el `README.md` si se podrán o no abrir desde *DCComparte Archivos*. Si decides que tu programa sea capaz de abrir y mostrar todo el contenido de estos archivos, se evaluará del mismo modo que los archivos mínimos, es decir, si en tu `README.md` mencionas que tu programa puede abrir por ejemplo archivos `.mp4` pero en tu programa esto no funciona correctamente, se descontará puntaje. Por otro lado, si específicas en tu `README.md` que tu programa no puede abrir otros tipos de archivos a los listados anteriormente, es necesario que indiques al usuario en la misma interfaz que ese tipo de archivo no se puede abrir, por ejemplo, por medio de una ventana de mensaje.

Cualquier decisión que tomes respecto a estos archivos **no los excluye** de los requerimientos mínimos respecto a los demás archivos. Es decir, deben conservar todas las funcionalidades de *networking*; ser descargables, ser compartibles, etc.

En Esquema Mis Archivos se muestra una posible distribución de los elementos mínimos solicitados: **(A)** Listado de archivos descargados, **(B)** Listado de usuarios conectados, **(C)** Controles para navegar el programa o abrir archivos. En Diseño Mis Archivos se muestra un diseño alternativo para la misma sección.

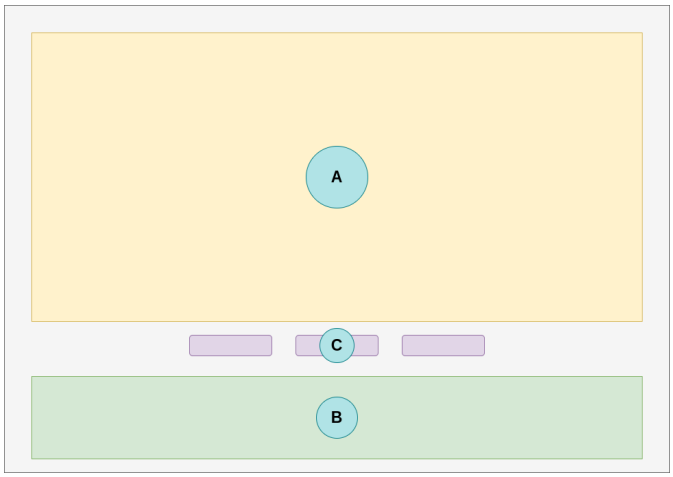


Figura 5: Esquema Mis Archivos

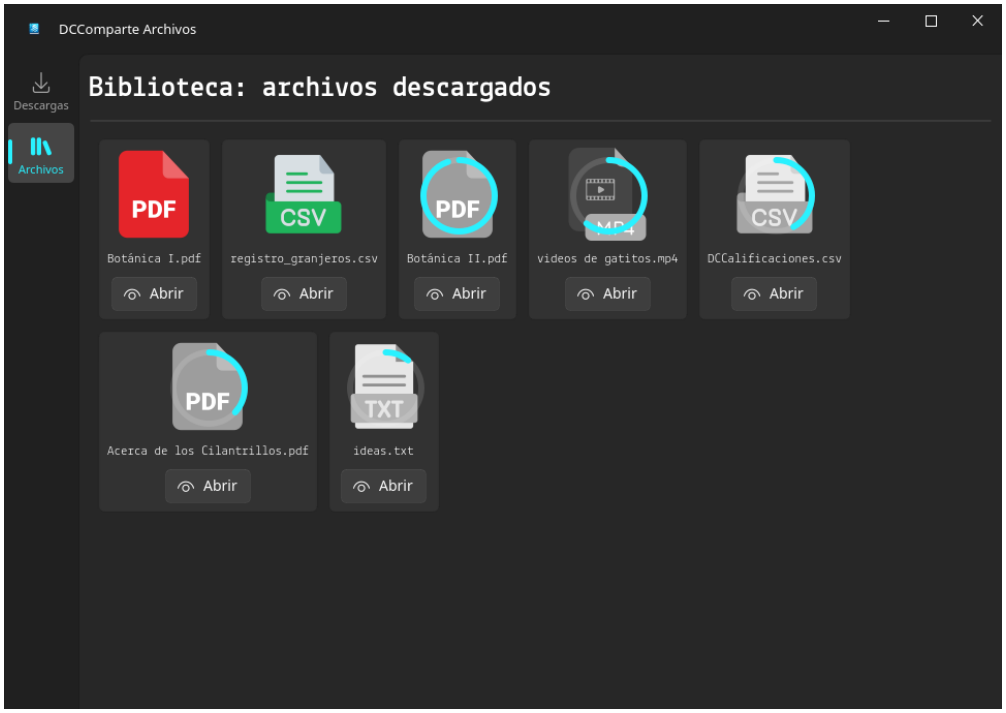


Figura 6: Diseño Mis Archivos

4. *Networking*

Cuando un usuario quiera descargar *archivos* de *DCComparte Archivos* tendrás que enviar su solicitud a un servidor utilizando todos tus conocimientos de *networking*. Deberás desarrollar una arquitectura `cliente` - `servidor` y `Peer2Peer` con el modelo **TCP/IP** haciendo uso del módulo **socket**.

Tu misión será implementar dos programas separados, uno para el **`servidor`** y otro para el **`cliente`**. De los mencionados, **siempre** se deberá ejecutar primero el servidor y este quedará escuchando para que se puedan conectar múltiples clientes de forma simultánea.

En una primera instancia, el Servidor Principal será quien envíe los archivos solicitados a los clientes. Sin embargo, este incluye una funcionalidad adicional que permite la delegación de este trabajo: si otro cliente conectado ya dispone del archivo solicitado, el Servidor Principal puede delegarle la transferencia de dicho archivo. Este delegará la transferencia de archivos a otro cliente conectado siempre que dicho cliente disponga del archivo solicitado y esté conectado. En caso de que el cliente con el archivo deseado se encuentre desconectado, el servidor tomará la responsabilidad de enviar el archivo al cliente solicitante.

La delegación seguirá el siguiente flujo:

- Al inicio de la ejecución de un cliente, este obtiene del Servidor Principal una serie de puertos los cuales podría disponibilizar para operar en el rol de proveedor de archivos. Dicha secuencia de puertos estará declarada y solamente podrá ser almacenada en el archivo `NOMBRE_ARCHIVO_NETWORKING_SETUP` de la parte del servidor.
- Cuando un cliente solicite descargar un archivo y pueda ser posible realizar dicha descarga mediante *Peer2Peer*, el Servidor Principal buscará algún cliente válido[2] y le indicará a dicho cliente que debe comenzar su rol como proveedor de archivos.

[2]Se entiendo como *válido* un cliente que **posee el archivo** y se encuentra conectado

- El cliente encargado de comenzar el rol como proveedor de archivos probará con la secuencia de puertos, enviada inicialmente, y le responderá al Servidor Principal con el puerto que logró utilizar para levantar su servidor. En caso que el cliente no lograra levantar un servidor con alguno de los puertos posibles, deberá indicar al Servidor Principal que no es posible cumplir dicho rol. Luego, en función de la respuesta enviada por el cliente, el Servidor Principal deberá realizar lo siguiente:
  - Si la respuesta es un puerto, entonces el Servidor Principal notificará al cliente que solicita el archivo que debe descargarlo de otra lugar y enviará el puerto de dicho proveedor.
  - Si la respuesta es que no logró ser un proveedor, entonces la responsabilidad de envío del archivo recaerá nuevamente sobre el Servidor Principal, siendo este el que finalmente envíe el archivo.
- Finalmente, ocurre un flujo de descarga normal, ya sea con el cliente asumiendo un rol de proveedor, o con el Servidor Principal.

Esta arquitectura permite una transferencia de archivos tipo **Peer2Peer** entre clientes conectados, disminuyendo así la demanda del Servidor Principal y mejorando la experiencia de descarga para los usuarios de *DCComparte Archivos*.

4.1. Arquitectura cliente-servidor

Las siguientes consideraciones **deben ser cumplidas al pie de la letra**. Lo que no esté especificado aquí puedes implementarlo según tu criterio, siempre y cuando se cumpla con lo solicitado y no contradiga nada de lo indicado. Si algo no está especificado o si no queda completamente claro, puedes **preguntar aquí**.

4.1.1. Separación funcional

El cliente y el servidor deben estar separados. Esto implica que deben estar en directorios diferentes e independientes, uno llamado `cliente` y otro llamado `servidor`. Cada directorio debe contar con los archivos y módulos necesarios para su correcta ejecución, asociados a los recursos que les correspondan, además de un archivo principal `main.py`, el cual inicializa cada una de estas entidades funcionales. La estructura que debes seguir se indica en el siguiente diagrama:

```
T4
├── cliente
│   ├── main.py
│   ├── backend
│   │   └── ...
│   └── frontend
│       └── ...
├── servidor
│   ├── main.py
│   └── ...
├── .gitignore
└── README.md
```

Si bien las carpetas asociadas al `cliente` y al `servidor` se ubican en el mismo directorio (`T4/`), la ejecución del `cliente` **no debe depender de archivos en la carpeta del `servidor`**, y la ejecución del `servidor` **no debe depender de archivos y/o recursos en la carpeta del `cliente`**. Esto significa que debes tratarlos como si estuvieran ejecutándose en computadores diferentes. La Figura 7 muestra una representación esperada para los distintos componentes del programa:

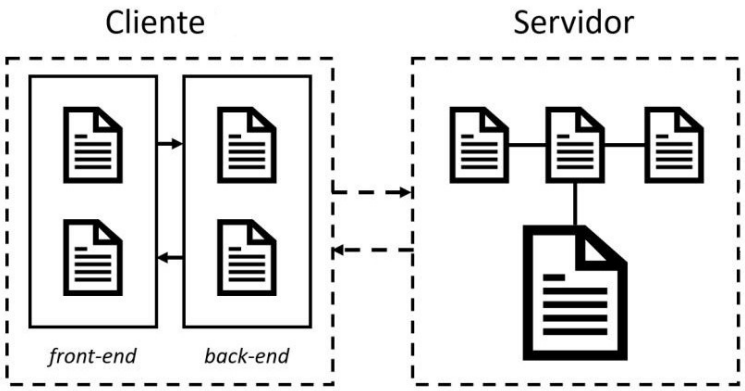


Figura 7: Separación cliente-servidor y *front-end*/*back-end*.

Cabe destacar que **solo el cliente tendrá una interfaz gráfica**. Por lo tanto, todo cliente debe contar con una separación entre *back-end* y *front-end*, mientras que la comunicación entre el `cliente` y el `servidor` debe realizarse mediante *sockets*. Ten en cuenta que la separación deberá ser de carácter **funcional** según lo indicado en Roles.

4.1.2. Conexión

Tanto el `servidor` como el **cliente** contaran con un archivo de formato `JSON` (mas detalles en la sección de Archivos). Este archivo debe contener todos los parámetros necesarios para realizar una comunicación exitosa utilizando *sockets* y el protocolo *TCP*. El archivo podría tener el siguiente formato:

```
{
    "host": "direccion ip",
    ...
}
```

Como funcionalidad extra, el programa debe ser capaz de manejar una selección del puerto del *socket*, tanto para el `cliente` como para el `servidor`, por medio de **argumento de consola**, pasando el puerto como argumento al ejecutar el archivo `.py`. Por ejemplo, si se quiere elegir el puerto 8000, se deberá ejecutar el archivo en consola de la siguiente forma: `python main.py 8000`.

Es importante recalcar que el `cliente` y el `servidor` **no deben usar el mismo archivo** `JSON` para obtener los parámetros.

4.1.3. Método de codificación

Cuando se establece la conexión entre **cliente** y **servidor**, deberán encargarse de intercambiar información constantemente entre sí. Por ejemplo, el **cliente** le comunica al **servidor** el archivo que desea descargar y este le responderá si el resultado corresponde al esperado, para luego mandar el archivo al cliente en forma de descarga.

Como queremos evitar que los zombies obtengan esta información única para su derrota, y evitar que intenten *hackear* el mensaje para alterarlo, debemos asegurarnos de codificar el contenido de todo mensaje antes de enviarlo, de tal forma que si alguien lo intercepta no pueda descifrarlo. Esta codificación se implementa bajo una estrategia de seccionar el contenido del mensaje en bloques de 32 *bytes* y enviarlo en paquetes de datos. Es por esto que deberás codificar cada mensajes enviado en *DCComparte Archivos* según la siguiente estructura:

- El emisor del mensaje define un identificador único para el mensaje a codificar. Este identificador debe ser guardado en 64 *bytes* y queda a criterio del desarrollador qué utilizar como identificador único.
- Los primeros 68 *bytes* indican el contexto de dicho mensaje. Este contexto se define como 64 *bytes* que corresponden al identificador único y los siguientes 4 *bytes* corresponden al **largo total del mensaje** a enviar antes de ser seccionado en bloques de contenidos de un largo de 32 *bytes*. El largo debe ser enviado en el formato *big endian*. [3]
- A continuación, se procederá a mandar el mensaje. Por cada bloque de contenido de 32 *bytes*, deberás enviar un *chunk* de 131 *bytes* en donde:
  - En los primeros 3 *bytes* se encuentra un número que representa la posición del bloque en el mensaje original. Este número inicia en cero, aumenta en 1 unidad por bloque y está codificados en *big endian*.
  - En los siguientes 64 *bytes* se encuentra codificado el identificador único del mensaje definido en el primer punto.
  - Luego, vienen 32 *bytes* que corresponden al `hash` del bloque de contenido. Para obtener este *hash*, deberás utilizar la función `generar_hash_bloque` sobre el bloque de contenido a enviar (más información en `funciones.pyc`).
  - Finalmente, están los 32 *bytes* que corresponde a dicho bloque de contenido.
- En caso que el último bloque de contenidos tenga un largo menor a 32 *bytes*, deberás rellenar los espacios restantes del bloque con *bytes* ceros (`b'\x00'`) hasta lograr que el bloque sea de 32 *bytes*. Luego, deberás enviarlo como un *chunk* de 131 *bytes* respetando las mismas instrucciones explicadas en el punto anterior.

Finalmente, el mensaje codificado que se envía se verá de la siguiente forma:

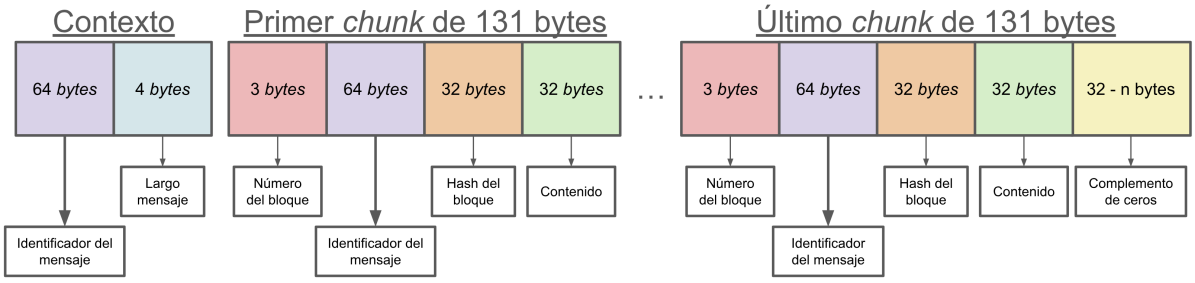


Figura 8: Ejemplo de un mensaje codificado.

4.1.4. Desconexión repentina

Dado que los zombies podrían interrumpir la conexión, el sistema debe estar preparado para manejar desconexiones de clientes o servidores. En caso de una desconexión durante la transferencia de uno o más archivos, queda a criterio del estudiante decidir cómo enfrentar este caso. El único requisito es que la desconexión no genere un error que impida ejecutar el programa como tampoco impida descargar el archivo en una siguiente ejecución.

[3] El *endianness* es el orden en el que se guardan los *bytes* en un respectivo espacio de memoria. Esto es relevante porque cuando intentes usar los métodos `int.from_bytes` e `int.to_bytes` deberás proporcionar el *endianness* que quieras usar, además de la cantidad de *bytes* que quieres usar para representarlo. Para más información puedes revisar este **enlace**.

- Si es el `servidor` quien se desconecta, cada cliente conectado debe mostrar un mensaje en la ventana (ya sea como texto plano o *pop-up*) explicando la situación, antes de cerrar el programa.
- Si es un `cliente` quien se desconecta, el servidor descarta su conexión y muestra en su consola un mensaje que indica lo anterior. Además, se debe visualizar su desconexión en la interfaz de los demás clientes conectados, en la cual el cliente desaparece del Listado de usuarios conectados.
- Además, en caso que el cliente desconectado esté cumpliendo un **rol de proveedor de archivo** y exista uno o más clientes descargando algún archivo de dicho proveedor, se debe cancelar la descarga automáticamente para cada uno de los clientes afectados. Junto a lo anterior, cualquier cliente afectado debe continuar su flujo normal sin cerrar su programa.

4.2. Roles

A continuación, se detallan las funcionalidades que deben ser manejadas por el `servidor` y las que deben ser manejadas por el `cliente`:

4.2.1. Servidor

El servidor está encargado de administrar las descargas de archivos. Para lograr este objetivo, debe cumplir con las siguientes funcionalidades:

- **Parte 1**

  Para que los usuarios puedan conectarse al servidor y recibir archivos, el servidor debe cumplir los siguientes requisitos:

  - **Permitir** el inicio de sesión de algún Cliente.
  - **Llevar registro** de que usuarios existen.
  - **Enviar archivos** a Clientes.
  - **Manejar todas las solicitudes del cliente para que este logre una descarga apropiada**. Por ejemplo, si algún Cliente solicita un archivo mientras se está descargando otro, el servidor debe poder realizar esto sin problemas.

- **Parte 2**

  Es necesario que el servidor pueda delegar el envío de archivos a otros clientes conectados en el caso que ya los tengan descargado. Para lograr esto, el servidor debe también cumplir las siguientes funcionalidades:

  - **Llevar registro** de que archivos descargados tiene cada cliente.
  - **Verificar** que sea un cliente válido para el archivo que se desea delegar su descarga.
  - **Actualizar** el registro de un cliente si es que no tiene un archivo que debería tener según la información registrada.
  - **Delegar la transferencia** de archivos a clientes para la lógica *Peer2Peer* enviando la información necesaria a los clientes involucrados.

4.2.2. Cliente

El Cliente en la arquitectura *Peer2Peer* se divide en dos roles con distintas características:

- **Parte 1 (RolCliente)**: en este rol el cliente actúa como receptor de archivos, descargándolos desde un servidor o desde otro cliente. Sus funcionalidades incluyen:
  - **Enviar su nombre de usuario** al Servidor Principal para poder iniciar sesión.
  - **Solicitar el listado de archivos** que están disponibles para descargar al Servidor Principal .
  - **Solicitar la descarga de un archivo específico** al Servidor Principal o a otro cliente que esté actuando temporalmente como servidor.
  - **Manipular el estado de una descarga activa**, permitiendo pausar, reanudar o cancelar una descarga, tanto con el Servidor Principal como con otro cliente que actúe como servidor.
  - **Notificar descarga exitosa** de un archivo al Servidor Principal para mantener actualizado el registro de descargas.
  - **Visualizar y gestionar** archivos descargados localmente.
  - **Ver clientes conectados** al Servidor Principal durante todo momento y en tiempo real.
- **Parte 2 (RolServidor)**: cuando un cliente asume este rol, se vuelve responsable de enviar archivos, cumpliendo con las siguientes funcionalidades:
  - **Transferir archivos** desde su propio registro hacia otro Cliente que lo solicite. Esto genera la descarga de un archivo desde el ClienteServidor para algún Cliente.
  - **Verificar la existencia del archivo solicitado** antes de iniciar una transferencia y notificar al servidor.
  - **Manejar las solicitudes** que el cliente puede hacerle al servidor, como pausas, reanudaciones o cancelaciones de descargas.

5. Eventos

El usuario trata de investigar más a fondo las plantas mutantes, con el objetivo de mejorarlas aun más. Debido a esto, los **Zombies** tratan de sabotear la descarga de archivos por parte de los usuarios, generando la aparición de ciertos eventos aleatorios. En algunos de estos eventos se debe hacer uso de la función `generar_hash_bloque` del archivo `funciones.pyc`. Con esto, los eventos aleatorios a implementar son:

Corrupción de archivo: Cada vez que el Servidor Principal manda un archivo, existe una probabilidad `PROB_CORRUPCION_ARCHIVO` de corromper dicho archivo. La corrupción consiste en que se perderá uno o más bloques del archivo. Por lo tanto, si el servidor decide perder el bloque j del archivo completo, entonces no se enviaría el *chunk* correspondiente a dicho bloque. Puedes asumir que el contexto de un mensaje y el último *chunk*, asociado al último bloque del mensaje, siempre serán enviados correctamente.

Esto quiere decir que finalmente el servidor no estaría enviando el archivo completo, sino partes de él. El programa debe poder enfrentarse a este evento y solucionarlo sin problemas, es decir, lograr obtener el archivo completo. Cómo solucionarlo dependerá de la implementación de cada uno, pero si o si debe ser solucionado dentro de una misma descarga. Solicitar el archivo completo nuevamente para obtener los *chunks* faltantes no será considerada una solución válida.

Corrupción de *chunk*: Cada vez que un Servidor Principal manda un archivo, puede ocurrir este evento. En particular, después de que se calcule el *hash* del bloque de contenido a enviar, el Servidor Principal tiene una probabilidad de corromper dicho bloque de contenido, la cual se define por `PROB_CORRUPCION_BLOQUE`. Esto se realiza con la función `corromper_bloque` del archivo **funciones.pyc** (mas información sobre ella

en la sub sección `funciones.pyc`). Esto genera que el bloque a enviar se altere a nivel de `bits`, por lo que la información recibida por el cliente esta corrupta.

Para solucionar esto, el cliente debe obtener el `hash` del bloque original, localizado dentro de los `bytes` obtenidos. Este `hash` debe ser comparado con el obtenido aplicándole `generar_hash_bloque` al bloque almacenado en los 32 `bytes` recibidos. En caso de ser iguales el bloque no fue corrupto, en caso contrario se debe pedir nuevamente el *chunk* hasta obtener un `chunk` sin errores. Cómo solucionarlo dependerá de la implementación de cada uno, pero si o si debe ser solucionado dentro de una misma descarga. Solicitar el archivo completo nuevamente para obtener los *chunks* corruptos no será considerada una solución válida.

Eliminación de archivo: Para cada cliente que ha descargado archivos previamente, si se desconecta y vuelve a ingresar, hay una probabilidad `PROB_ELIMINACION_ARCHIVO` de que se eliminen los archivos pedidos al servidor previamente. Por lo tanto, la información que almacena el servidor sobre el registro de qué archivo posee cada cliente no es completamente confiable, es por esto que cada vez que un cliente pida un archivo que, en teoría tiene otro cliente, el servidor debe verificarlo.

6. Archivos

Esta tarea se centra, dentro de otras cosas, en el trabajo con archivos. Existen dos grupos de archivos con lo que se tendrá que trabajar, los archivos de **configuración** y los archivos de **datos**.

Archivos de configuración: deberán estar previamente definidos en el cliente y/o servidor. Son archivos tipo **JSON**, los cuales presentan información sobre el estado o configuraciones para los clientes o el servidor. Estos archivos son **requeridos** y sus nombres deben ser especificados en el archivo `parametros.py`. Es **obligatorio** tener al menos los siguientes 3:

- `NOMBRE_ARCHIVO_CLIENTES_DESCARGAS`: Archivo ubicado en el servidor, el cual lleva un registro de qué archivos exitosamente descargados posee cada cliente.
- `NOMBRE_ARCHIVO_CLIENTES_PUERTOS`: Archivo ubicado en el servidor, el cual lleva un registro sobre qué puertos puede utilizar un cliente en modo servidor durante una comunicación *Peer2Peer*.
- `NOMBRE_ARCHIVO_NETWORKING_SETUP`: Archivo ubicado en el servidor y el cliente, el cual debe contener todos los parámetros necesarios para realizar una comunicación exitosa utilizando *sockets* y el protocolo *TCP*.

Cualquier otro archivo que considere necesario crear puede hacerlo, siempre y cuando anote en qué consiste y realice los supuestos necesarios en el `README`.

Archivos de datos: Estos son los archivos que inicialmente contendrá el servidor que podrán ser descargados por el cliente. Sus nombres, sin considerar extensión, siempre serán y deben ser únicos e irrepetibles. Estos son archivos con diferentes extensiones, de las cuales dependerá su contenido. Además, los archivos que serán trabajados nunca superarán los `400 Mb` de uso de almacenamiento por archivo individual.

Los archivos podrán ser descargados haciendo *click* en el siguiente enlace.

Algunos ejemplos son:

- **.txt**: Poseen texto plano en su contenido.
- **.csv**: Poseen información separada por comas.
- **.jpg**: Archivos de tipo imagen.
- **.png**: Archivos de tipo imagen.
- **.mp4**: Archivos de tipo video.

Finalmente, respecto a dichos archivos, estos deben ser **almacenados** de manera **correcta** y **consistente**. Esto quiere decir de que inicialmente solo deben existir en algún directorio , cuyo nombre queda a criterio del lector y debe ser especificado en el archivo `parametros.py`, que se ubique dentro del directorio del Servidor Principal. Luego, a medida que algún cliente descargue exitosamente los archivos, estos deben quedar almacenados dentro de algún directorio, cuyo nombre queda a criterio del lector y debe ser especificado en el archivo `parametros.py`, que se ubique dentro del directorio del cliente. Además, debes especificar, en tu `README.md`, la ruta en donde deben estar creadas estas carpetas, tanto en el directorio de cliente y servidor, de modo que la tarea funcione correctamente.

6.1. `funciones.pyc`

Ademas, se entregara un archivo **.pyc**, el cual contendrá funciones que deben utilizar para el desarrollo de la tarea. Estas funciones solo deben ser importadas y utilizadas, sin la posibilidad de ser modificadas o ver su funcionamiento. Estas funciones son:

- No modificar `def corromper_bloque(bloque: bytearray) -> bytearray:`

  Recibe un `bytearray` y modifica sus *bits* con el objetivo de corromper dicho `bytearray`. Retorna un `bytearray` corrupto.

- No modificar `def generar_hash_bloque(bloque: bytearray) -> bytearray:`

  Recibe un `bytearray` y retorna el *hash* de dicho `bytearray`.

6.2. `parametros.py`

Para esta tarea, se requiere la creación de un archivo `parametros.py` donde deberás **completar todos los parámetros mencionados a lo largo del enunciado**. Dichos parámetros se presentarán en `ESTE_FORMATO` y en ese color. Además, es fundamental incluir cualquier valor constante necesario en tu tarea, así como cualquier tipo de *path* utilizado.

Un parámetro siempre tiene que estar nombrado de acuerdo a la función que cumplen, por lo tanto, asegúrate de que sean descriptivos y reconocibles. Un ejemplo de parametrizaciones es la siguiente:

```
1 CINCO = 5 # mal parámetro
2 PROBABILIDAD_CORRUPCION = 0.2 # buen parámetro
```

Si necesitas agregar algún parámetro que varíe de acuerdo a otros parámetros, una correcta parametrización sería la siguiente:

```
1 PI = 3.14
2 RADIO_CIRCUNFERENCIA = 3
3 AREA_CIRCUNFERENCIA = PI * (RADIO_CIRCUNFERENCIA ** 2)
```

Dentro del archivo `parametros.py`, es obligatorio que hagas uso de todos los parámetros almacenados y los importes correctamente. Cualquier información no relacionada con parámetros almacenada en este archivo resultará en una penalización en tu nota. Recuerda que no se permite el ***hard-coding***[4], ya que esta práctica se considera incorrecta y su uso conllevará una reducción en tu calificación.

[4] *Hard-coding* es la práctica de ingresar valores directamente en el código fuente del programa en lugar de parametrizar desde fuentes externas.

7. `.gitignore`

Para esta tarea **deberás utilizar un** `.gitignore` para ignorar los archivos indicados, este deberá estar dentro de tu carpeta `Tareas/T4/`.

Los elementos que no debes subir y **debes ignorar mediante el archivo** `.gitignore` para esta tarea son:

- El enunciado.
- El archivo `README_inicial.md` (no confundir con el archivo **obligatorio** `README.md`)
- Cualquier archivo *bytecode*, es decir, cualquier archivo con extensión `.pyc`
- Cualquier carpeta que almacene archivos de tipo dato (los especificados en esta parte Archivos)

Recuerda **no ignorar archivos vitales de tu tarea como los que tú creas o modificas, o tu tarea no podrá ser revisada**.

Es importante que hagan un correcto uso del archivo `.gitignore`, es decir, los archivos **deben** no subirse al repositorio debido al uso correcto del archivo `.gitignore` y no debido a otros medios.

8. Importante: Corrección de la tarea

En el **siguiente enlace** se encuentra la distribución de puntajes. En esta señalará con color **amarillo** cada ítem que será evaluado a nivel funcional y de código, es decir, aparte de que funcione, se revisará que el código esté bien confeccionado y que la funcionalidad esté correctamente integrada en el programa. En color **azul** se señalará cada ítem a evaluar el correcto uso de señales para la comunicación *front-end*/*back-end*. Todo aquel que no esté pintado de amarillo o azul, significa que será evaluado si y sólo si se puede probar con la ejecución de su tarea.

Importante: Todo ítem corregido por el cuerpo docente será evaluado únicamente de forma ternaria: cumple totalmente el ítem, cumple parcialmente o no cumple con lo mínimo esperado. Finalmente, todos los descuentos serán asignados manualmente por el cuerpo docente respetando lo expuesto en **el documento de bases generales**.

La corrección se realizará en función del último *commit* realizado antes de la fecha oficial de entrega (miércoles 20 de noviembre a las 20:00). Si se desea continuar con la evaluación en el periodo de entrega atrasado, es decir, realizar un nuevo *commit* después de la fecha de entrega, **es imperante responder el formulario de entrega atrasada** sin importar si se utilizará o no cupones. Responder este formulario es el mecanismo que el curso dispone para identificar las entregas atrasadas. El enlace al formulario está en la primera hoja de este enunciado y estará disponible para responder hasta el viernes 22 de noviembre a las 23:59.

Para terminar, si durante la realización de tu tarea se te presenta algún problema o situación que pueda afectar tu rendimiento, no dudes en contactar al ayudante de Bienestar de tu sección. El correo está en el **siguiente enlace**.

9. Restricciones y alcances

- Esta tarea es **estrictamente individual**, y está regida por el **Código de honor de Ingeniería**.
- Tu programa debe ser desarrollado en Python 3.11.X con X mayor o igual a 7.
- Tu programa debe estar compuesto por uno o más archivos de extensión `.py` que estén correctamente ordenados por carpeta. **No se revisará archivos en otra extensión como `.ipynb`**.
- Toda el código entregado debe estar contenido en la carpeta y rama (*branch*) indicadas al inicio del enunciado. Ante cualquier problema relacionado a esto, es decir, una carpte distinta a `T2` o una rama distinta a `main`, se recomienda preguntar en las ***issues* del foro**.
- Si no se encuentra especificado en el enunciado, supón que el uso de cualquier librería Python está prohibido. Pregunta en la *issue* especial del **foro** si es que es posible utilizar alguna librería en particular.
- Debes adjuntar un **único archivo markdown**, llamado `README.md`, **conciso y claro**, donde describas los alcances de tu programa, cómo correrlo, las librerías usadas, los supuestos hechos, y las referencias a código externo. El no incluir este archivo, incluir un readme vacío o el subir más de un archivo `.md`, conllevará un **descuento** en tu nota.
- Esta tarea se debe desarrollar **exclusivamente** con los contenidos liberados al momento de publicar el enunciado. No se permitirá utilizar contenidos que se vean posterior a la publicación de esta evaluación.
- Se encuentra estrictamente prohibido citar código que haya sido publicado **después de la liberación del enunciado**. En otras palabras, solo se permite citar contenido que ya exista previo a

la publicación del enunciado. Además, se encuentra estrictamente prohibido el uso de herramientas generadoras de código para el apoyo de la evaluación.

- Cualquier aspecto no especificado queda a tu criterio, siempre que no pase por sobre otro que sí sea especificado por enunciado.

Las tareas que no cumplan con las restricciones del enunciado obtendrán la calificación mínima (1,0)**.**