



Trabajo Práctico - Cliente de Bittorrent Rustico

[75.42] Taller de Programacion I
Primer cuatrimestre de 2022

Alumno	Padrón	Email
Luciano Leon Trujillo Palomo	105664	ltrujillo@fi.uba.ar
Matias Gabriel Fusco	105683	mfusco@fi.uba.ar
Maria Vazquez Navarro	105576	mvazquezn@fi.uba.ar
Tomás Szwarcberg	103755	tszwarcberg@fi.uba.ar

Índice

1. Objetivos	3
2. Requerimientos	3
3. Herramientas y organización	4
3.1. GitHub	4
3.2. Editor: Virtual Studio Code	5
3.3. Miro	5
4. Conexión con tracker	6
4.1. Lectura del archivo .torrent	6
4.2. Obtención de Peers	6
5. Implementación del Cliente	8
5.1. Framework de Actores	8
5.2. El modelo Sender/Worker	8
5.2.1. Sender	8
5.2.2. Worker	9
5.2.3. Ejemplo de implementación	9
5.3. Actores del Sistema	11
5.3.1. Piece Manager	11
5.3.2. Peer Connection Manager	12
5.3.3. Open Peer Connection	13
5.3.4. Piece Saver	14
5.3.5. Logger	14
5.4. Estrategia de descarga de Piece Manager	15
5.4.1. Estrategia de pedido de piezas	15
5.4.2. Selección de peer	15
6. Flujo del programa	17
6.1. Inicializacion	17
6.2. Creacion de Conexiones	17
6.3. Primeras Descargas	19
6.4. Flujo de descargas: Caso Feliz	20
6.5. Flujo de descargas: Caso fallo en la descarga	21
6.6. Flujo de descargas: Caso fallo en la conexión	22
6.6.1. Volvemos a mandar Request al Tracker	23
6.7. Flujo de finalización	24
7. Implementación del servidor	25
7.1. Inicio y API	25
7.2. Acceptor	25
7.3. Server Connection	26
7.4. ThreadPool	27
8. Implementación de Interfaz Grafica	28
8.1. Comunicación con el cliente	29
8.2. Actualización de la Interfaz en base a los eventos	30
9. Manejo de errores	31
9.1. Errores que incluyen el error interno	31
9.2. Errores que incluyen texto con la razón de falla	32

10. Testing	34
10.1. Caso de Mock: HTTP Service	34
10.2. Caso de Mock: PeerMessage Service	34
10.3. Tests de Integración	35
11. Obstáculos encontrados	35
12. Potenciales mejoras	37
13. Conclusiones	38

1. Objetivos

El objetivo de nuestro equipo en este proyecto fue implementar un Cliente BitTorrent con funcionalidades acotadas en Rust, aplicando técnicas y metodologías de desarrollo aprendidas a lo largo de la carrera y de la materia. Con la intención de hacer un código robusto y completo, decidimos diseñarlo utilizando técnicas de programación concurrente, como el modelado en base a actores.

El desarrollo del proyecto fue acompañado por un constante trabajo de investigación en relación a las buenas prácticas en el desarrollo de software y en el manejo de Rust, intentando a cada paso respetar los estándares del lenguaje.

2. Requerimientos

El proyecto fue desarrollado teniendo en cuenta los requerimientos funcionales establecidos por la cátedra. Dentro de los principales, incluye: poder descargar más de un archivo por ejecución, tomar los archivos torrent deseados por parámetro al momento de ejecutar el programa, y que el cliente soporte torrents con tener múltiples archivos.

3. Herramientas y organización

3.1. GitHub

Desde el inicio del proyecto hicimos uso de la herramienta de github, junto con todas las features que ésta nos ofrece. Para comenzar a idear el inicio del proyecto, utilizamos la herramienta de Projects dentro de nuestro repositorio. Las tareas/issues fueron asignadas entre los distintos miembros del equipo para manterner una organización e historial de evolución a medida que avanzabamos con el desarrollo.

Utilizamos la funcionalidad de las cards, ofrecida por github, para diferenciar las issues dependiendo del estado en la cual estuvieran.

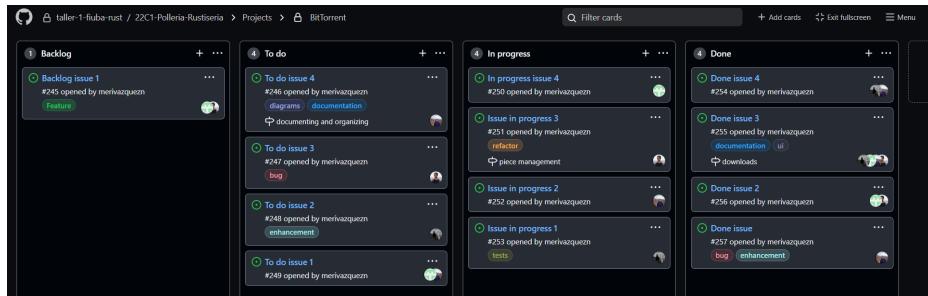


Figura 1: Cards manejadas a lo largo del proyecto.

Como podemos ver, creamos las cards "Backlog, To do, In progress, Done". En cada una fuimos agregando las issues que creamos, y dependiendo del estado de la misma, la movíamos a la card correspondiente.

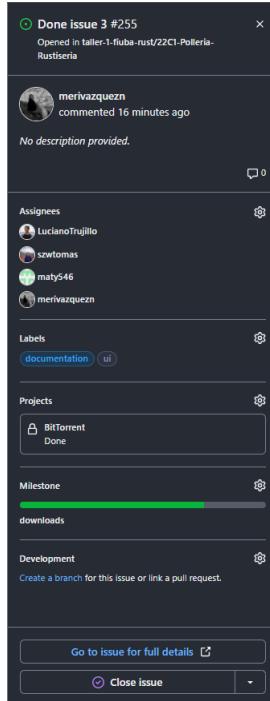


Figura 2: Configuracion de una issue.

Vemos cómo en la creación de una issue se pueden registrar las personas a las cuales se les

asigna la tarea. La issue debería tener un título que informe de forma clara el objetivo de la misma, además se le puede agregar un Body para información más puntual y detallada si es necesario. También nos sirvieron los labels que especifican la categoría de la tarea, ya sea que se trate de un bug, una feature, entre otras. Por último, se puede asignar la issue a un milestone para tener las cosas organizadas y abstraer la misma a su funcionalidad.

Al crear la issue pudimos usar la UI de GitHub para crear la branch que esté unida a la issue determinada. Una vez resuelta, pudimos crear un Pull Request desde dicha branch. Al aceptarse el Pull Request, entonces mergeamos la branch a main de una forma muy cómoda, lo cual cerraba la issue y la movía inmediatamente a Done (card del proyecto).

3.2. Editor: Virtual Studio Code

Para escribir el código utilizamos la herramienta de Visual Studio Code, utilizando las extensiones de Rust-Analyzer. Si bien Visual Studio Code es muy cómodo por la rapidez en su inicio y su interfaz de usuario user-friendly, notamos varios bugs que ralentizaron el desarrollo del proyecto. En un futuro optaríamos por usar otros editores, o incluso IDEs, que nos permitan utilizar las mismas herramientas sin que nos ponga un obstáculo en el día a día.

3.3. Miro

Herramienta útil para hacer diagramas. Le dimos mucha utilidad al plantear las ideas iniciales del flujo de programa que esperábamos tener. Una vez tenida la idea final, creamos un diagrama que nos sirvió como ayuda visual a la hora de desarrollar.

4. Conexión con tracker

En esta sección veremos cómo el cliente lee el archivo .torrent con metainformación sobre la red para conectarse con el tracker para obtener las direcciones ip y los puertos de los peers con los que más tarde estableceremos conexiones para pedirles piezas con partes del/los archivos a descargar.

4.1. Lectura del archivo .torrent

El archivo .torrent contiene la información en formato bencode, explicado en la documentación del protocolo. Para decodificar y codificar esta información se creó un módulo 'bencode', que recorre todos los bytes del archivo y crea una estructura de datos final para poder acceder fácilmente a la información del archivo en manera de diccionario. En caso de fallar, éste indica qué carácter no fue interpretado correctamente y la razón de por qué falló la decodificación. Esto permitió detectar fallas en .torrents mal formateados fácilmente. A continuación se muestra la estructura de datos:

```
pub enum BencodeDecodedValue {
    String(Vec<u8>),
    Integer(i64),
    List(Vec<BencodeDecodedValue>),
    Dictionary(HashMap<Vec<u8>, BencodeDecodedValue>)
}
```

4.2. Obtención de Peers

Una vez leído y parseado el .torrent, tenemos a nuestra disposición los *announce*, que son las urls de los Trackers de la red. Los Trackers son servidores **externos** y **centralizados** que tienen información sobre la red, particularmente acerca de cuáles son los peers que están (o lo estuvieron en el corto plazo) activos actuando tanto como clientes, es decir, descargando piezas, o bien como servidor teniendo en posesión una parte o la totalidad del archivo que se quiere compartir.

Teniendo ya forma de comunicarse con el Tracker, existe una serie de mensajes definidos que le podemos enviar para comunicarle que queremos participar de la red y pedirle que nos mande la lista de Peers. Esto se hace mediante el protocolo HTTP, el tracker acepta una request de tipo GET al endpoint */announce* donde le pasamos una serie de parámetros por querystring tales como la cantidad de bytes ya descargados, los compartidos y cuantos nos faltan. Además, le debemos pasar el *info hash*, que identifica inequívocamente a la red aplicandole el algoritmo de hashing SHA-1 al diccionario entero leído del metainfo.

Para mandar una request HTTP, creamos un módulo 'HTTP Service' en nuestro proyecto que contiene una interfaz (*trait* en Rust) para enviar una request de tipo GET a determinada url y puerto. Realizar esto mediante un trait nos permitirá usar la técnica de mocking e inversión de dependencias para poder testear mucho más fácilmente las interacciones reales con servidores externos (en este caso el Tracker), aunque profundizaremos esto más adelante en la sección de testing.

```
pub trait IHttpService {
    fn get(path: &str, query_params: &str) -> Result<Vec<u8>, HttpsServiceError>;
}
```

En la implementación del método GET, lo que hacemos es directamente abrir un socket en la url y puerto dado con los parámetros de querystring pedidos, y escribiendo los headers correspondientes a HTTP sacados del protocolo. Esto es un texto donde se indica el método y de ser necesario un cuerpo (aunque no lo necesitamos en este caso), y luego nos quedamos esperando con un timeout leyendo de ese mismo socket para escuchar lo que nos responda el tracker. En caso de que no nos llegue esa respuesta, es algo terminal para nuestro cliente porque no podemos continuar sin una lista de peers, por lo tanto el programa finaliza.

Una vez que el tracker tiene lista la lista de peers, escribe en el socket una response HTTP, que tiene también ciertos headers e información correspondiente al protocolo, pero en el cuerpo de la respuesta encontramos la información buscada. Lo que hay es una lista benencodeada, donde en cada índice hay un diccionario con la ip, puerto y id de cada peer. Lo que hacemos entonces es primero procesar este mensaje para quitar los bytes correspondientes al protocolo que no nos interesan, y finalmente usamos el módulo de bencoding de la sección anterior para procesar los bytes y obtener así una lista de peers en un formato que nos sirva (un vector de Peers, donde cada peer tiene su ip, id y puerto de contacto).

Esa lista está en un diccionario, donde también puede incluir el tiempo mínimo que tenemos que esperar para volver a contactarnos con el tracker.

Finalmente, el módulo responde una 'TrackerResponse', que es tanto la lista de Peers descripta anterioramente como un Option de este intervalo a esperar, que en caso de existir será usado más tarde para obtener más peers del tracker de ser necesario.

5. Implementación del Cliente

5.1. Framework de Actores

Como ya todos sabemos, hacer programación concurrente siempre es problemático si no se lo aborda con mucho detallismo y minuciosidad. Esto se debe a que las variables y factores que pueden alterar el resultado de nuestro programa aumenta exponencialmente, dado que perdemos control sobre el orden de ejecución de todas las instrucciones. Es por esto que surgieron muchos modelos de programación orientados a organizar un sistema donde hay diferentes responsabilidades bien definidas y pueden actuar de manera semi independiente.

Para este proyecto, decidimos implementar el cliente bittorrent con el modelo de concurrencia de actores. Nuestra decisión fue fruto de tratar de implementar por primera vez un sistema concurrente. Lo pensamos como diferentes personas con ciertas tareas específicas que se comunican entre sí para obtener un resultado final. Un sistema de actores está formado principalmente por tres componentes:

- **Los mensajes:** Siempre son asíncronos y contienen información de cualquier tipo.
- **Los buzones:** Los lugares en los que los actores reciben la información para ser procesada (channels normalmente)
- **Los actores:** Un proceso que se encarga de realizar una acción según la información recibida por el mensaje. En nuestro caso, son stateful (tienen datos internos para persistir información como piezas, peers, etc.).

A medida que pensabamos en la implementación, nos dimos cuenta que este modelo nos servía para manejar adecuadamente las tolerancia a fallas del sistema, lo cual es cotidiando en un bittorrent client, puesto que las conexiones por TCP son frágiles.

Los sistemas de actores actualmente permiten abordar la tolerancia a fallos usando el patrón reactivo “Let it Crash” para el manejo de fallos. Este “prefiere” que un actor se caiga y realizar posteriormente un reinicio (si llegara a tener sentido), en lugar de intentar controlar cualquier caso de error utilizando programación defensiva. Entonces, al fallar por ejemplo el pedido de una pieza, simplemente se reintenta n veces con timeout antes de dar por perdida la conexión.

El problema con el modelo de actores es que, al ser asíncrono, hay que tener cuidado con las actualizaciones que tienen que pasar en tiempo real. Por ejemplo, notificar la descarga de una pieza o la caída de una conexión.

5.2. El modelo Sender/Worker

Para las distintas partes del sistema que compone al cliente, utilizamos varias entidades, como se explico anteriormente. Cada entidad se comunica con las otras mediante mensajes para poder ejecutar sus tareas, y la característica fundamental de la forma que empleamos de organizar y dividir tareas es que puedan realizarse, dentro de lo posible, de manera independiente.

Entonces, necesitamos entidades que soporten un mecanismo mediante el cual podemos darles órdenes en cualquier momento, y que simultáneamente estén en escucha constante de las órdenes que puedan llegar.

5.2.1. Sender

Es la cara que la entidad ofrece para que podamos darle órdenes. Una estructura de datos muy simple que internamente contiene un extremo de escritura para un channel de la librería estandar de Rust, cuyo extremo de lectura esta guardado en el Worker.

Los métodos que ofrece el Sender equivalen a la funcionalidad esperada de la entidad. Lo único

que hacen estos metodos es enviar un mensajer al Worker mediante el channel que comparten. El Sender de una entidad es lo unico que el resto de entidades conocen de esta. Si una entidad A debe enviar un mensaje a una entidad B, se le da a A un Sender de B. Los Senders implementan el trait **Clone**, para que se puedan recibir mensajes desde otras entidades, aprovechando la estructura Multiple-Producer/Single-Consumer de los channels.

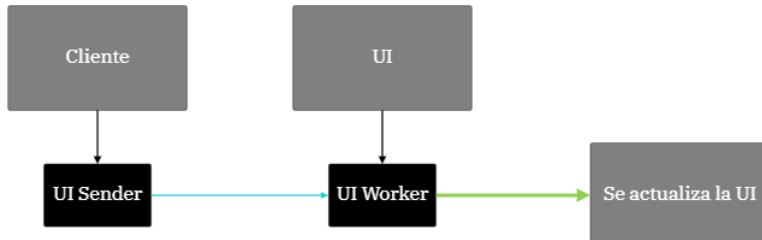
5.2.2. Worker

El worker esta pensado para inicializarse en su propia thread, estando continuamente a la escucha de mensajes que pueda llegar del Sender mediante el channel interno.

Su implementacion es simplemente un loop de escucha bloqueante sobre el channel. Cuando llega un mensaje, el Worker ejecuta las instrucciones que corresponden al tipo del mensaje, que eventualmente puede ser un mensaje que indique que el Worker ya no es necesario. En este caso, se rompe el loop para que la thread pueda terminar su ejecucion, y pueda ser posteriormente cerrada mediante un join.

5.2.3. Ejemplo de implementación

Para comprender mejor la idea del par Sender/Worker, veamos un ejemplo muy simple. Supongamos que tenemos un cliente de BitTorrent en medio de su ejecución que necesita comunicarse con la UI. De momento, es una interfaz simple que solo informa la cantidad de piezas que ya fueron descargadas. Entonces, queremos que cada vez que se descargue una pieza, la interfaz gráfica se actualice. Para esto podemos modelar la UI con un par Sender/Worker, donde el Worker es quien se encarga de actualizar la UI con los datos recibidos.



Entonces, como se ve en la imagen, vamos a pasarle a nuestro Cliente un Sender de la UI. El sender podría ofrecer un método llamado `new_piece_downloaded`, el cual informaría al UI Worker que tiene que actualizar la interfaz. Esto es necesario porque en esta implementacion, **el único que es capaz de modificar la interfaz es el UI Worker**.

Para entender el funcionamiento de este ejemplo, veamos un poco de código. Lo unico que tiene que hacer el Cliente es llamar a

`UISender.new_piece_downloaded()`.

La implementación del lado del UISender sería algo así:

```

pub struct UISender{
    pub sender: Sender<UIMessage>, //donde UIMessage es un enum
}
impl UISender{
    pub fn new_piece_downloaded(){
  
```

```

        self.sender.send(UIMessage::NewPieceDownloaded);
    }
}

```

Y la implementacion del UIWorker:

```

pub struct UIWorker{
    pub receiver: Receiver<UIMessage>, //donde UIMessage es un enum
}
impl UIWorker{
    fn new_piece_downloaded(){
        //codigo que actualiza la UI;
    }

    pub fn listen(&mut self) -> Result<(), RecvError>{
        loop {
            let message = self.receiver.recv()?;
            match message {
                UIMessage::NewPieceDownloaded => self.new_piece_downloaded();

                //...
            }
        }
    }
}

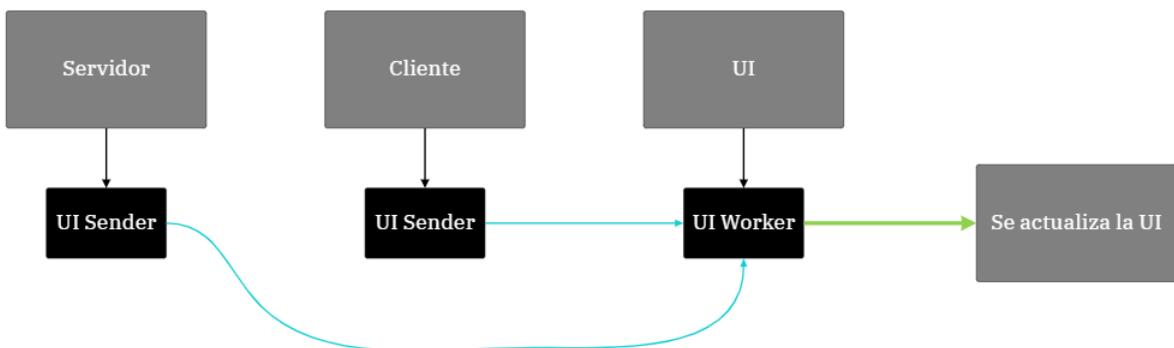
```

Como dijimos antes, el sender únicamente notifica al Worker, utilizando el channel que comparten, de que debe realizar una tarea. El Worker va a ejecutar la tarea cuando reciba ese mensaje, porque enviar al Worker a correr significa llamar a su método listen, para que quede a la espera de instrucciones.

Podemos ver que sería muy fácil añadir tareas que se le puedan dar al Worker. Veamos un breve ejemplo de como podríamos hacer esto.

Supongamos que ahora tenemos un Servidor corriendo, y queremos que nuestra UI nos indique cuantas conexiones abiertas tiene nuestro Servidor.

Para actualizar la interfaz correctamente, recordemos que quien tiene la capacidad de hacerlo es solo el UIWorker. Entonces, basta con pasarle un UISender al Servidor! Aprovechamos, como dijimos antes, la estructura Multiple-Producer/Single-Consumer de los channels para poder darle UISenders a toda entidad que los necesite. Veamos como queda el esquema con este agregado.



Con esto, solo basta crear un nuevo tipo de UIMessage, e implementar las funciones correspondientes en UISender y UIWorker. Luego, agregamos el tipo de mensaje al bloque de match en el

método listen() del UIWorker. Y finalmente, hacemos que el servidor llame al método nuevo de UISender cuando sea necesario.

Y eso es todo, con una implementación casi idéntica a la que necesitamos para el Cliente (difieren solo en la tarea que deba hacer el UIWorker), ya tenemos una forma de pasar instrucciones desde el Servidor a la UI.

5.3. Actores del Sistema

En esta sección se van a presentar los actores que intervienen en nuestro modelo. Vale la pena notar que por cada torrent que se está descargando, se tiene un set de estas entidades que colaboran para lograr la descarga. Cada entidad de las que se van a explicar a continuación está implementada con una estructura Sender/Worker.

5.3.1. Piece Manager

Actor encargado de la gestión de piezas y su descarga. En base a información sobre qué piezas existen, cuáles tenemos, cuáles faltan descargar, y qué peers activos tienen esas piezas, organiza tanto el orden en el que se piden y a qué peer en particular pedirle cada una. Como es de sospechar, hay infinitas maneras de optimizar la descarga a partir de esta información. Por eso mismo, queríamos dejar una implementación abstracta del mismo, en el cual toda la decisión sobre qué pieza descargar está centralizada en esta entidad. De esta manera, sería muy fácil implementar diferentes algoritmos hasta encontrar el que mejor rendimiento tenga en las pruebas, sin alterar el funcionamiento de las entidades que se comunican con esta.

Una vez que el Piece Manager recibe la información sobre qué peer tiene qué pieza, empieza a solicitarlas mediante sus mecanismos de selección y actualizando sus decisiones a medida que se reciben las piezas. Dado que esta entidad guarda información sobre qué piezas fueron descargadas por cada peer, optimizará su decisión de una manera que reduzca el tiempo total de descarga.

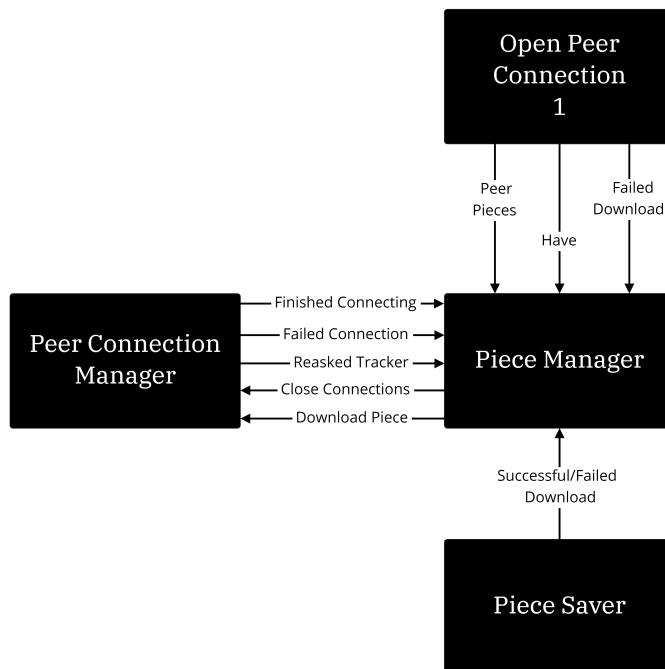


Figura 3: Grafico de los mensajes del Piece Manager.

5.3.2. Peer Connection Manager

Actor encargado de administrar las conexiones con los peers. Abre las conexiones, mantiene control sobre ellas y las cierra cuando es necesario. Sirve también como gateway para hablar con un peer: El Piece Manager le manda un mensaje al Peer Connection Manager diciéndole que desea que cierto Peer descargue cierta pieza. Luego, será el Peer Connection Manager el encargado de que el pedido le llegue al peer correcto.

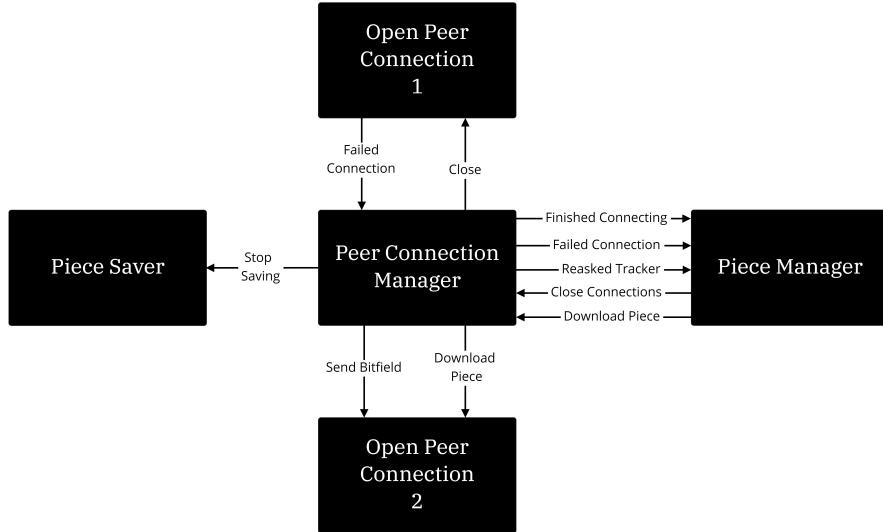


Figura 4: Grafico de los mensajes del Peer Connection Manager.

5.3.3. Open Peer Connection

Actor encargado de proporcionar una conexión abierta con un peer, ofrece un set de instrucciones para la comunicación con el mismo. Administra el protocolo de comunicación con peers y lo aisla del resto del programa.

Su funcionalidad se centra en el momento de recibir las piezas. En esta entidad se valida lo recibido, bloque a bloque. Si falla la validación de un bloque, se produce un error que desemboca en indicar una descarga fallida. Esto resulta en que la conexión con el peer sea cerrada por el Peer Connection Manager.

La conexión también será cerrada en el caso de no lograr una comunicación adecuada con el Peer a la hora de pedir piezas. Esto último se detecta mediante un mecanismo que incluye timeouts en los streams de TCP y una cantidad máxima de reintentos. Si en MAX_TRIES intentos no se recibe un mensaje sin que se active un timeout, cuenta como una descarga fallida.

Claro está que para pedir piezas, es necesario primero crear una conexión válida. Esto implica que implementa la lógica del intercambio de mensajes inicial para empezar a pedirlas. Además le manda el bitfield del peer al cual se conectó al Piece Manager, para que este pueda decidir si mandarle el pedido de una pieza en particular.

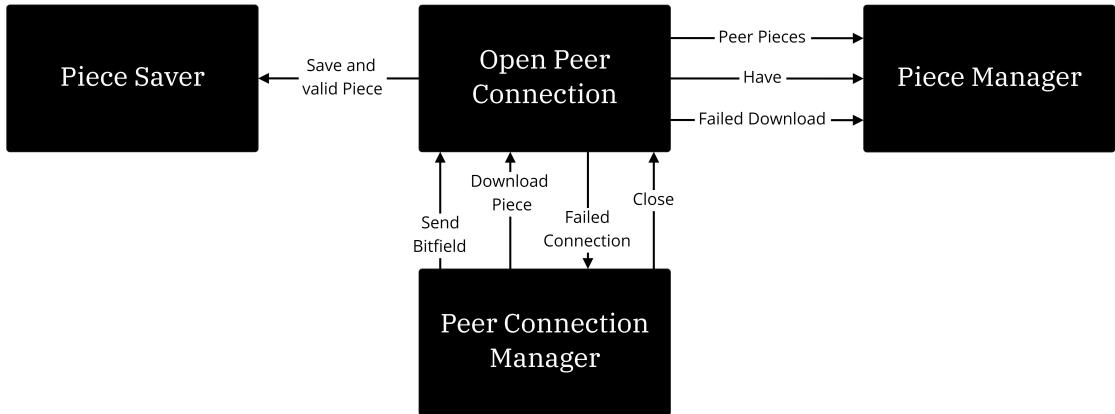


Figura 5: Grafico de los mensajes de la Open Peer Connection.

5.3.4. Piece Saver

Actor encargado de realizar todos los guardados en disco de las piezas y se comunica con el logger para registrar las descargas. Es importante tenerla implementada como entidad independiente para evitar que otras partes del sistema pierdan tiempo escribiendo en disco. Tiene también la responsabilidad de validar las piezas antes de guardarlas en disco, calculando su hash SHA-1 y comparándolo con el que está en el Metainfo para la pieza correspondiente. En caso de fallar la validación de la pieza, éste le informa al Piece Manager del fallo para que maneje la situación y vuelva a pedir la pieza.

5.3.5. Logger

Actor encargado de loguear la descarga de piezas en un archivo designado, indicando por cada una el número de pieza correspondiente. Recibe los mensajes del Piece Saver cuando una pieza fue correctamente descargada en disco, y luego escribe en el archivo de log. Lo bueno de tenerlo en un actor separado al Piece Saver - si bien su tarea es trivial - es que si el logging fuera más complejo, por ejemplo mandar una request HTTP a algún servidor, no demoraría la descarga de piezas en disco, ya que no es bloqueante para el actor Piece Saver.

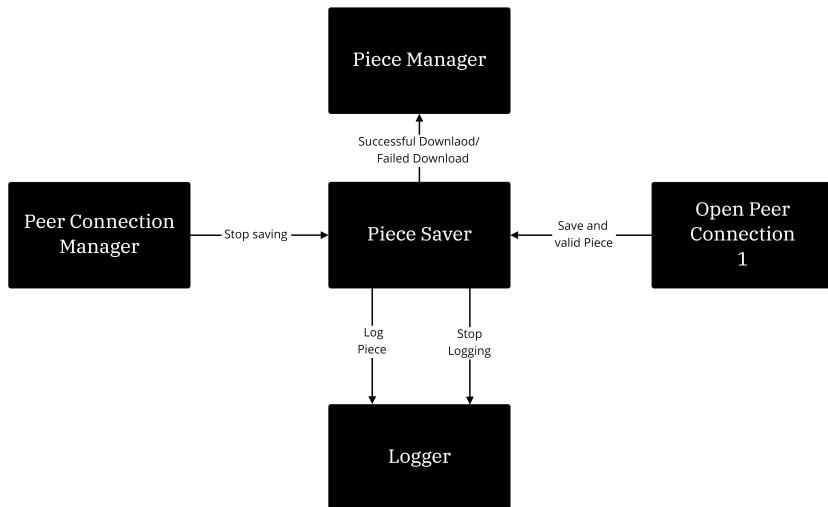


Figura 6: Grafico de los mensajes del Piece Saver y Logger.

5.4. Estrategia de descarga de Piece Manager

5.4.1. Estrategia de pedido de piezas

El algoritmo utilizado para la optimización del pedido de descarga de piezas es la estrategia de prioridades 'Rarest Piece First'. Una vez elegida la pieza más óptima para mandar a descargar, se selecciona al peer más óptimo al cual le pediremos la pieza mediante un algoritmo que elige al que tenga menor cantidad de pedidos de pieza encoladas.

5.4.1.1 Selección de pieza: Rarest Piece First

En nuestro sistema tenemos como prioridad descargar las piezas más raras (piezas que pocos peers tienen) antes. La decisión se basó en querer disminuir la probabilidad de perder la conexión con los pocos peers que pueden mandarlas.

Para lograr este objetivo, implementamos el siguiente algoritmo cada vez que pedimos una pieza:

1. recorremos todas las piezas del torrent que aún no fueron descargadas.
2. por cada pieza nos fijamos la cantidad de peers que la tienen disponible.
3. guardamos el índice de la pieza que tenga menor cantidad de peers con la pieza.

A primer mirada, parece poco óptimo recorrer la lista de las piezas del torrent cada vez que pedimos por una, pero como siempre, la implementación depende del contexto en donde se aplica.

En principio, se podría argumentar el uso de un cola de prioridad utilizando una estructura de datos como un Heap tree. De hecho, Rust tiene una implementación en la colección de librería estándar. Una vez recolectada la información sobre que peer tiene cada pieza, se podría crear rápidamente y luego ir sacando las piezas en tiempo constante $O(1)$. Lamentablemente, la cantidad de peers que tienen una pieza varía significantemente durante la vida del programa. Las conexiones se caen y a veces se crean nuevas. Esto genera que deberíamos mantener actualizada esta estructura de datos, que ya se vuelve no trivial. La implementación de la librería estándar de Rust no proporciona soporte para el caso de querer modificar la prioridad de un elemento existente en el Hash, y recrearlo terminaría siendo mas costoso que simplemente tener una lista y recorrerla cuando se pide una pieza.

Por otro lado, la cantidad de piezas que tienen los torrents no suelen superar las dos mil, para no tener archivos .torrent muy pesados y generar sobrecarga en los trackers. Esta cantidad es mínima para recorrer y no implica ningún tipo de desafío de procesamiento para la CPU.

En conclusión, utilizamos el algoritmo mas sencillo posible que convenientemente resulta ser el más útil para el caso de elección de piezas. Esto se debe a un conjunto de factores:

1. La prioridad de cada pieza va cambiando a lo largo de un programa
2. La cantidad de piezas no representa un desafío de procesamiento
3. Las piezas se abstraen como un simple número que representa el índice, y la lista de peers por cada pieza no suele superar cincuenta, con lo cual en general es una estructura de datos muy simple y liviana.

5.4.2. Selección de peer

Al iniciar el proyecto nuestra selección de peers se basaba en elegirlos de forma aleatoria entre los peers que puedan mandar la pieza. Si bien es cierto que la elección aleatoria de peers para descargar una pieza tiende a ser equitativa a la larga, encontramos ciertos problemas en particular:

1. Si el torrent cuenta con pocas piezas pero de gran tamaño, hay probabilidades de que se asigne un porcentaje significativo de piezas a descargar a un peer cuya velocidad de descarga es baja.
2. Si asignamos de manera aleatoria los peers a descargar, nos perdemos de utilizar la valiosa información sobre la velocidad de descarga de cada uno para asignar mas piezas al peer con mayor velocidad.

Como resultado, sucedía que al principio la descarga parecía ir muy rápida, pero una vez que los peers con mayor velocidad terminaban de mandar todas las piezas asignadas en un principio, quedaba en estado "idle" pues no se le asignaban más piezas a menos que haya fallado alguna descarga por parte de otro peer.

Es por esto que decidimos hacer dos cambios fundamentales:

1. La selección de peer a descargar cierta pieza se hace con el siguiente objetivo: asignar mayor cantidad de piezas a los peers con velocidades altas, pero al mismo tiempo mantener pedidos de piezas activamente a los peers de velocidades bajas. De esta manera mantenemos presión en todo tipo de conexiones, sin dejar a ninguna inutilizada.
2. No asignar la descarga de todas las piezas al principio, dado que para cumplir con 1. debemos ir analizando el comportamiento de cada peer y en base a eso tomar la decisión. Con lo cual, las piezas se mandan a descargar a medida que la descarga de otras terminan (o fallan).

En concreto, la implementación se aborda de la siguiente manera:

Primero, se manda un diez porciento de la cantidad de piezas a descargar de manera aleatoria entre los peers. Es un porcentaje arbitrario elegido con el objetivo de empezar rápidamente a descargar varias piezas y obtener información sobre la velocidad de descarga de cada peer lo antes posible.

Siendo que comenzamos el programa pidiendo el diez porciento de las piezas, a medida que se vayan descargando de forma exitosa, volveremos a pedir la siguiente pieza utilizando el mecanismo explicado anteriormente (Rarest First).

Al elegir al peer, elegiremos al peer que tenga menor cantidad de piezas encoladas a descargar. Como actualizamos este registro recién al recibir una descarga exitosa, el peer con menor cantidad de piezas encoladas es equivalente a decir el que más rápido hizo su trabajo de mandarnos todas las piezas que le pedimos, mientras que un peer con velocidad baja tiene varias piezas encoladas pues tarda más en darnos cada una.

De esta manera muy sencilla, logramos aprovechar a los peers con conexión rápida sin necesidad de estar calculando directamente la velocidad de descarga de cada uno, si no más bien analizando un efecto colateral de la diferencia de velocidades: La cantidad de piezas que le faltan enviar a cada uno.

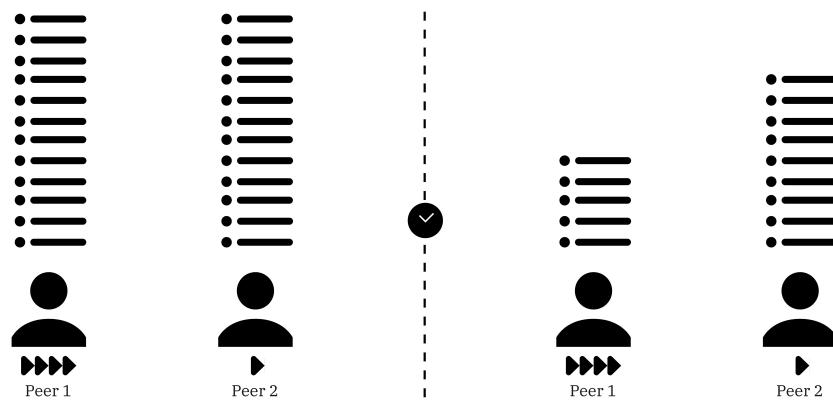


Figura 7: Fastest peer downloads faster.

Como podemos ver, el peer con mayor velocidad irá teniendo cada vez menos piezas a descargar, liberando así más rápido la cola de piezas pedidas.

Utilizando las estrategias mencionadas, el piece manager será el encargado de, habiendo decidido que pieza y a quién pedirle descargarla, comunicarle la decisión al peer connection manager.

6. Flujo del programa

En el esquema de actores que planteamos, las acciones de cada entidad son, casi en su totalidad, dependientes de los mensajes que les lleguen de los otros. Es por eso que las comunicaciones entre entidades estan modeladas de manera que queden separadas claramente las responsabilidades, buscando presentar a cada entidad como una caja negra que recibe y envia mensajes, e internamente ejecuta las instrucciones necesarias.

En esta sección se va a explicar la interacción entre las entidades de nuestro sistema, y al mismo tiempo el flujo del programa, para poder así dejar en claro exactamente en que situación se da cada interacción.

6.1. Inicializacion

Se crean las entidades con su respectivo par Sender/Worker desde el thread del programa principal. Los Workers se mandan a correr en un thread exclusivo llamando a una función común ‘listen()’, la cual como mencionamos antes, los atrapa en un loop de escucha constante sobre su extremo del channel. La función listen toma por parámetro clones de los Senders que la entidad necesite.

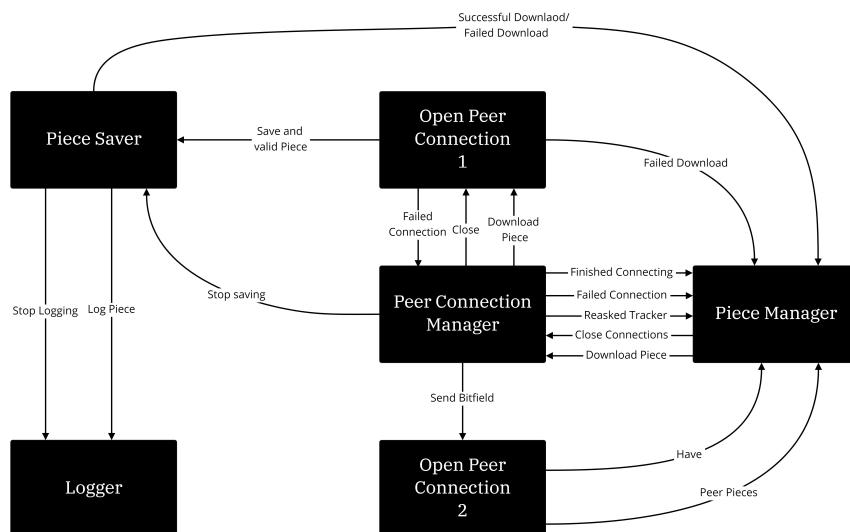


Figura 8: Grafico de los mensajes del proyecto.

6.2. Creacion de Conexiones

El inicio del programa es una vez inicializada la entidad de Peer Connection Manager. Desde ella tomamos los peers mandados por el tracker y comenzamos los intentos de conexión con cada uno de ellos. No todas las conexiones se inicializaran de forma exitosa, se pueden dar los escenarios en los que el peer decide no abrir la conexión, o que la respuesta del mismo no cumpla con el timeout estipulado en nuestro cliente.

Cada vez que se logra iniciar una conexión de forma exitosa, el Peer Connection Manager le pide a la misma que le mande su bitfield al Piece Manager. Así, cada vez que se inicia una conexión, el Piece Manager recibe un nuevo bitfield, y sabe entonces que piezas podrá aportar el peer.

Una vez iniciadas todas las conexiones, el Peer Connection Manager avisará al Piece Manager que terminó de establecer un X número de conexiones. Este mensaje es crucial para avisar al Piece

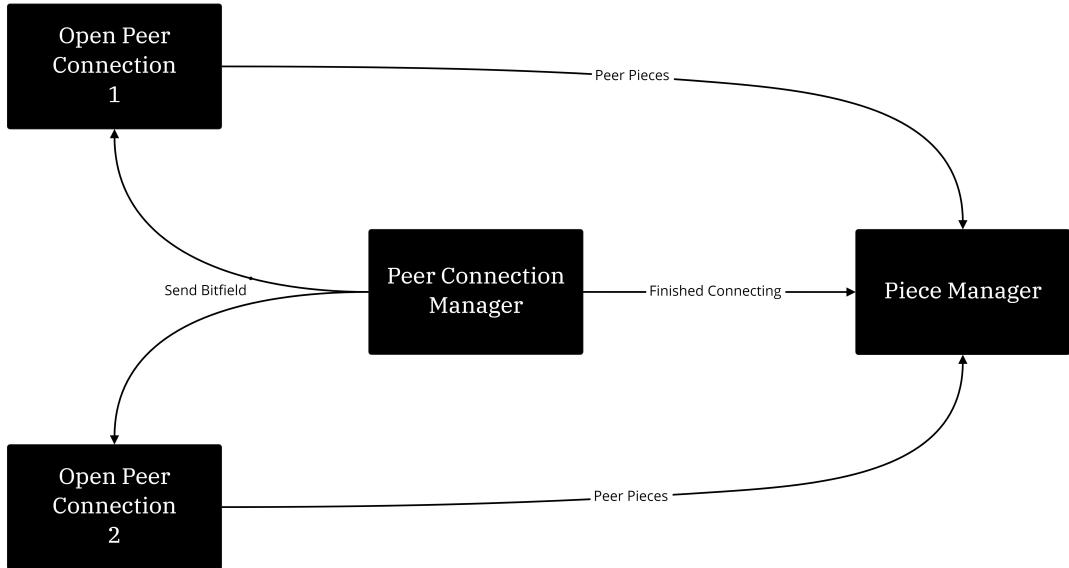


Figura 9: Fase 1: Inicialización y recibimiento de bitfields.

Manager cuantos bitfields debe esperar antes de comenzar la descarga. Si bien esperar a que todos los peers se hayan conectado, y enviado los bitfields, puede demorar unos segundos el inicio de la descarga, llegamos a la conclusión de que preferíamos no perder la oportunidad de optimizar la descarga conociendo la información de todos los peers, por lo que esperamos a que lleguen todos los bitfields antes de comenzar la ejecución de la descarga.

A medida que se vayan abriendo las conexiones, el Piece Manager, en simultáneo, irá recibiendo los bitfields de las mismas, pero como dijimos no iniciará la descarga hasta tener la cantidad de bitfields. Esperará a saber cuántos debe esperar, y una vez llegada la cantidad de bitfields que se esperan comenzará el proceso de descarga.

6.3. Primeras Descargas

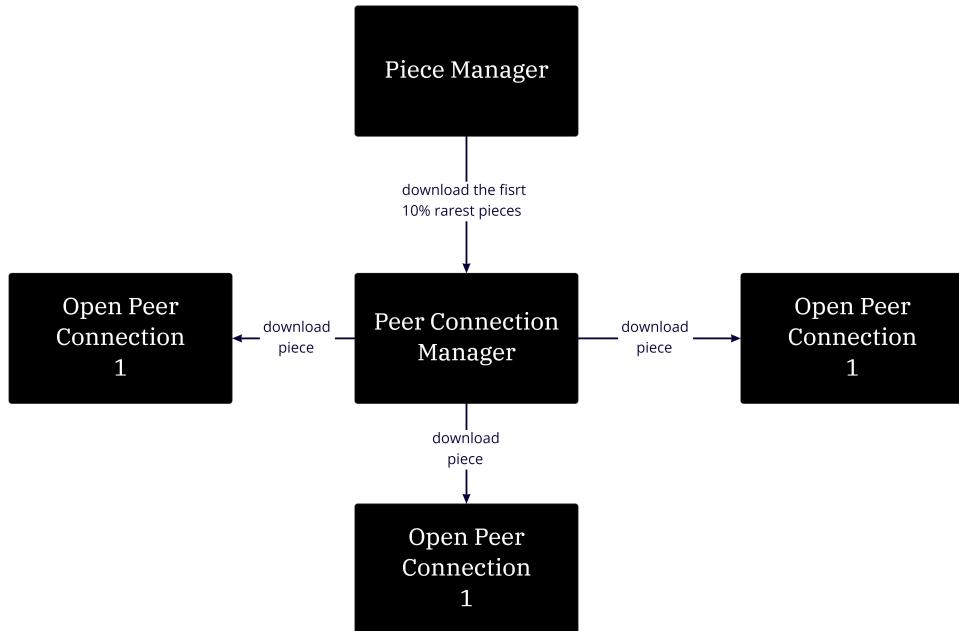


Figura 10: Fase 2: Primeras descargas.

La fase del inicio de las descargas se basa en utilizar el algoritmo de selección de piezas y peers explicados en la sección: Estrategia de pedido de piezas. Pediremos una primer cantidad de piezas, la cual será de tamaño variable dependiendo de la cantidad total de pieza. Habiendo probado con distintos porcentajes decidimos comenzar la descarga pidiendo todas las piezas elegidas por el algoritmo dentro del 10 por ciento de las piezas totales.

Siempre antes de enviar a descargar se chequea que la misma no haya sido mandada anteriormente, y que efectivamente tenga peers que puedan otorgarla.

Una vez mandados los primeros pedidos, el Piece Manager queda a la espera de avisos sobre descargas fallidas o exitosas, o cierres de ciertas conexiones.

Comenzaremos la explicación acerca del flujo de descarga con un caso feliz en el cual todas las descargas que se manden a descargar lo hagan de forma exitosa, y las conexiones no se caigan en ningún momento.

6.4. Flujo de descargas: Caso Feliz

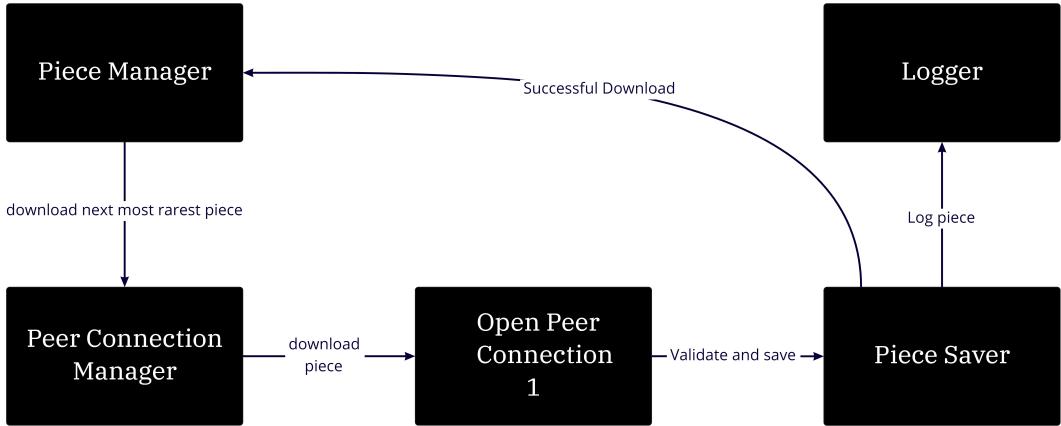


Figura 11: Fase 3: Flujo de descarga: Caso Feliz.

La figura 11 indica un loop que puede ser seguido por el orden de las flechas. Dado que en la sección anterior describimos como el Piece Manager queda a la espera de nuevos mensajes, comenzaremos la explicación desde que reciba un mensaje que informe una descarga exitosa. Como podemos ver, este mensaje será mandado por el Piece Saver una vez que pueda guardar la pieza recibida en disco, quien también tendrá que avisar al Logger que debe loggear la nueva descarga.

La responsabilidad del Piece Manager al recibir un aviso de descarga exitosa es mandar a descargar la siguiente pieza. Una vez más utiliza la estrategia de prioridades, y no manda a descargar una pieza que esté siendo descargada, o no tenga peers para ser mandada.

El flujo de pedido de descarga es el mismo explicado anteriormente, especificando el peer a quien pedir la pieza al Peer Connection Manager, y siendo este el responsable de mandarle la instrucción de descargar la pieza al respectivo peer. Este entonces se encargará de pedir la pieza, y una vez recibida la respuesta de la misma, pide al Piece Saver la validación y guardado en disco de la misma.

Recordemos que en esta sección estamos mostrando un caso feliz, por lo que asumimos que las piezas siempre son válidas, por lo que después de validarla ilustramos que se debe mandar un mensaje de Successful download al Piece Manager, y un aviso al logger, reiniciando así el loop.

6.5. Flujo de descargas: Caso fallo en la descarga

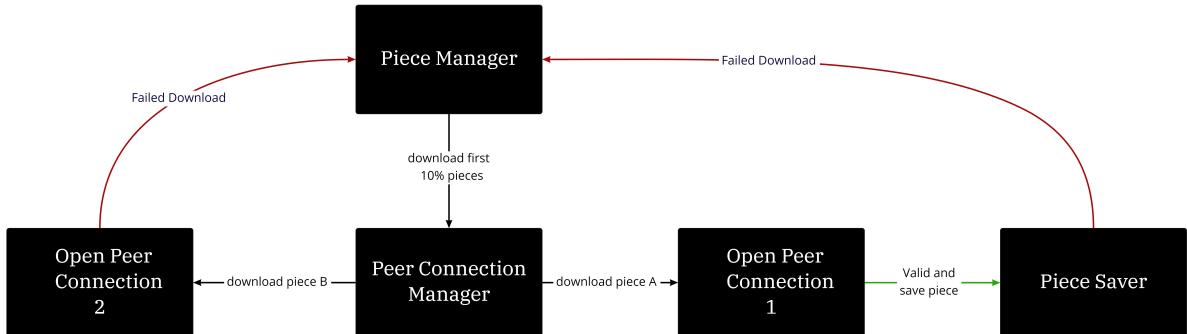


Figura 12: Fase 3: Flujo de descarga: Caso fallo en la descarga.

Como dijimos en la sección: Primeras Descargas el Piece Manager queda a la espera de mensajes que informen el estado de las descargas. Detallaremos las posibles causas del recibimiento de un mensaje de descarga fallida.

Una descarga puede fallar por tres razones. Si al pedir bloques de una pieza vemos que la conexión no cumple con mandarnos los que estemos pidiendo, verificando el índice y offset de lo recibido, paramos la descarga y mandamos el mensaje de Failed Download desde . También se puede dar el caso en el cual la verificación por SHA1 no nos de lo esperado, o que por más de que la pieza haya sido válida, haya habido un error al guardarla en disco, en cuyo caso el mensaje será enviado desde Piece Saver.

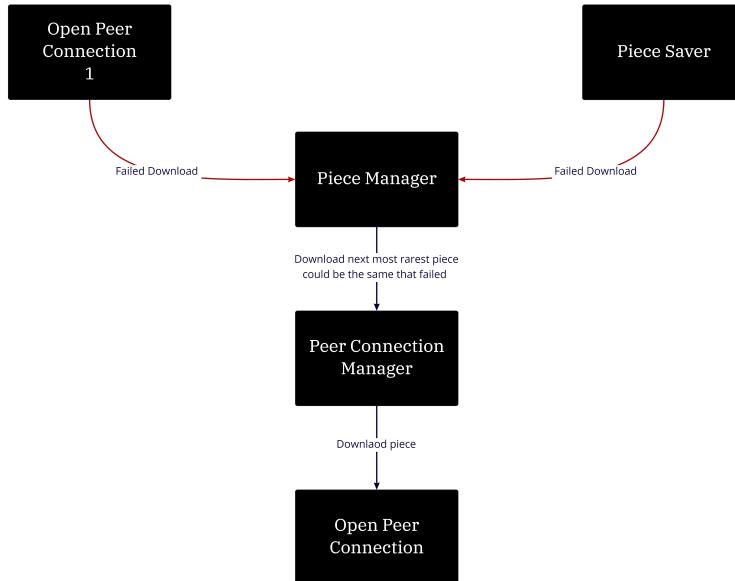


Figura 13: Fase 3: Flujo de descarga: Caso fallo en la descarga: Pedimos devuelta una pieza.

En caso de recibir un mensaje de descarga fallida, independientemente de quien haya enviado el mensaje, el Piece manager discierne cuál es la mejor pieza para mandar a descargar, y le manda al Peer Connection Manager la decisión. La pieza elegida puede o no ser la misma pieza que tuvo la falla, depende de qué mensajes se hayan recibido desde que se mandó a descargar por primera vez, y la recibida del fallo de su descarga.

6.6. Flujo de descargas: Caso fallo en la conexión

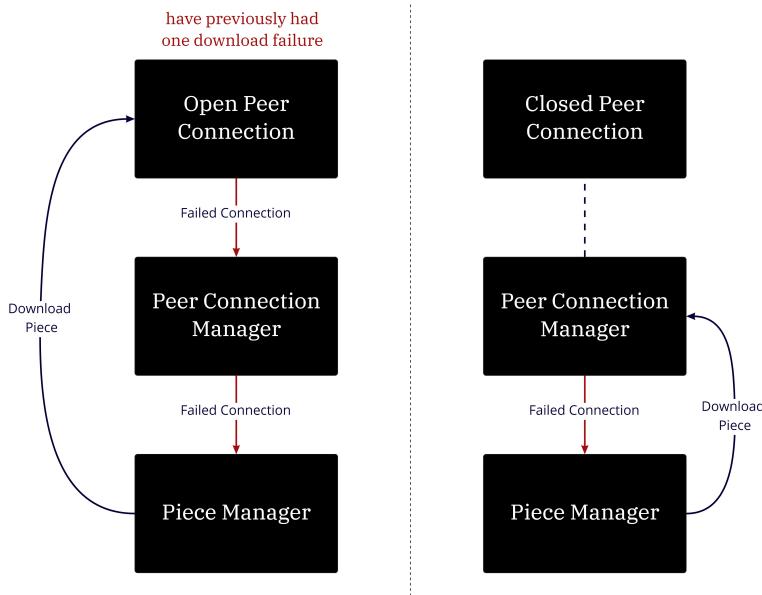


Figura 14: Fase 3: Flujo de descarga: Caso fallo en la conexión.

Al estar en medio de la descarga de un archivo, las conexiones con peers pueden caerse, ya sea por errores de la red, o por decisión nuestra. Decidimos que era conveniente desconectarnos de los peers apenas tardará más de un determinado tiempo en responder, o nos mandaran una pieza errónea, pues no queremos que nos manden piezas invalidas.

Vemos en la figura 14 las posibles fallas de conexión. En el caso de la izquierda nos encontramos con el caso de un pedido de descarga de una pieza desde el Piece Manager. Siendo que ya se había tenido un error en la descarga con ese peer, decidimos cortar la conexión.

La Open Peer Connection entonces antes de cerrarse enviará al Peer Connection Manager que hubo un fallo en su conexión. El Peer Connection Manager se encargará de avisar al Piece Manager sobre la desconexión del peer.

El segundo caso ilustra la secuencia en la cual al Peer Connection Manager le llega un mensaje de pedido de descarga hacia un peer ya desconectado. Esta situación es consecuencia de la concurrencia, siendo que el mensaje pudo haber sido mandado porque el Piece Manager aún no recibió el aviso de conexión fallida, o bien porque el mensaje simplemente es viejo y el Piece Manager lo mando antes de enterarse de la desconexión.

Cual sea el caso, el Peer Connection Manager se encargará de enviar un mensaje de Failed Connection al Piece Manager, para asegurarse de que el mismo se entere del fallo de la conexión.

Habiendo recibido el mensaje de fallo de conexión, el piece manager se encargará de actualizar sus datos internos, para luego volver a pedir piezas. Esto lo hará con el manejo de prioridades mencionado, y lo hará una vez por cada pieza que se le haya mandado a descargar al peer. De esta forma nos aseguramos de intentar mantener constante la cantidad de piezas que estemos pidiendo.

6.6.1. Volvemos a mandar Request al Tracker

El Peer Connection Manager tiene la responsabilidad de volver a pedir una Tracker Response si hubo algún fallo en una conexión. Esto no podrá hacerlo de forma indefinida, pues estará limitado por un máximo de 3 requests, un máximo de peers que se pueden tener para pedir al tracker (18 en nuestro caso) y el timer intervals del tracker. Si el tracker no envió un intervalo, entonces verificamos el tiempo entre requests.

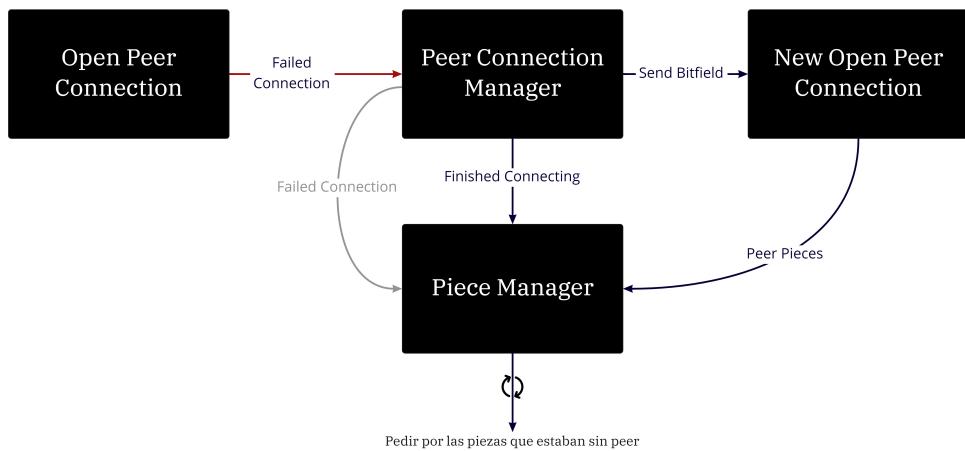


Figura 15: Fase 3: Flujo de descarga: Caso fallo en la conexión: Pedido al Tracker.

Vemos ilustrado en la figura Fase 3: Flujo de descarga: Caso fallo en la conexión: Pedido al Tracker. que la funcionalidad del Peer Connection Manager al recibir un mensaje de Failed Connection no es solo avisar al Piece Manager que la conexión fue fallida, sino que también, si se cumplen los requisitos, reconectarse con el tracker para conseguir una nueva lista de peers. Una vez conseguida la lista, pediremos a las nuevas conexiones, es decir las que no tengamos en nuestro sistema, que manden el bitfield al Piece Manager. El flujo en esta sección es muy similar al utilizado al iniciar el programa.

El piece manager irá recibiendo bitfields, y no comenzará la ejecución del programa hasta no tener la cantidad que se hayan mandado desde el Peer Connection Manager una vez terminado de hacer las nuevas conexiones.

Una vez recibidos todos los nuevos bitfields, se mandan a descargar las piezas que hayamos registrado como piezas sin peers, pues ahora tenemos nuevos bitfields. El signo de flechas utilizado en el gráfico sirve para ilustrar el sentido de loop el cual espera todos los bitfields antes de empezar, y para mostrar que se manda a descargar una cantidad dinámica de piezas, pues son las piezas que se hayan registrado como piezas sin peer.

6.7. Flujo de finalización

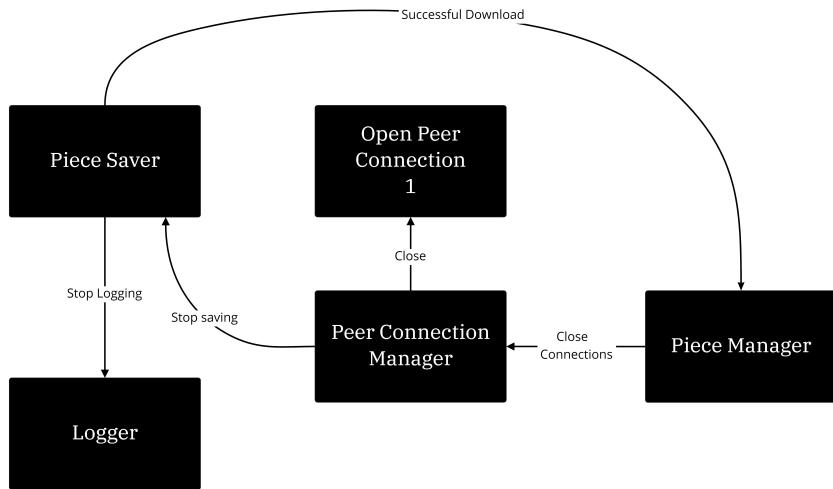


Figura 16: Fase 3: Flujo de finalización.

Luego de manejar cada mensaje que le llegue al piece manager, el mismo se fijara si es necesario cortar el programa. Las razones por las cuales tendría que cortar el programa son el haber terminado de descargar todas las piezas de forma exitosa, o habiendo llegado al punto en el que no tengamos peers a los cuales pedirles las piezas restantes. Si nos encontramos con alguna de las dos situaciones debemos cortar el programa. Para eso se ejecuta al siguiente secuencia:

Suponemos que nos llega un mensaje de Successful download, luego de manejarlo y actualizar nuestros datos nos fijamos si se cumple alguna de las situaciones anteriormente mencionadas. En caso de cumplirse, se seguirá el flujo de la imagen.

Primero se manda un mensaje de Close Connection al Peer Connection Manager, desde el Piece Manager. Recibiendo ese mensaje, el Peer Connection Manager cerrará todas sus conexiones abiertas, mandando un mensaje de Close a cada Open Peer Connection. Luego enviará un Stop Saving al Piece Saver, quien al recibir el último mensaje, enviará un Stop al Logger.

Por último desde el cliente se hace un join a cada handle de las threads de las entidades mencionadas. En caso de que el programa se haya terminado habiendo descargado todas las piezas de forma exitosa, se juntan todas las piezas y se arma el archivo completo, guardando el resultado en disco.

7. Implementación del servidor

En esta sección, veremos cual fue la arquitectura y algunos detalles de implementación del servidor, que es el módulo del programa que se ocupa de poder compartir a otros peers piezas que ya tenemos descargadas para formar parte y enriquecer la red del torrent.

7.1. Inicio y API

Primero, vemos como se inicia y pone a correr el servidor desde el flujo de la aplicación. Todas las funciones y estructuras expuestas se encuentran en el módulo *server*, por lo que se pueden importar simplemente como:

```
use bittorrent_rustico::server::nombre_estructura
```

o como:

```
use crate::server::nombre_estructura
```

cuando es código dentro de la misma crate.

Para iniciar el servidor, se llama el método estático *run* de la estructura *server*, que se encarga de mandar a correr el servidor, y devuelve un 'Server' mediante el cual le podemos mandar el mensaje de detenerse cuando queramos. El uso sería entonces:

```
use bittorrent_rustico::server::Server;

let server: Server = Server::run(
    client_peer_id,
    metainfo,
    port,
    sleep_duration,
    pieces_directory
);

// Mandamos a correr el cliente, esperamos a que termine

server.stop();
```

Los parámetros *client peer id*, *metainfo*, *port* y *pieces directory* son el id que generamos desde el cliente, la estructura generada por el metainfo leído del torrent, el puerto en el que debemos abrir el socket para escuchar (el mismo que le pasamos al tracker) y el directorio en el que almacenaremos las piezas respectivamente. El parámetro 'sleep duration' lo veremos más adelante cuando veamos como se manejan las conexiones entrantes, pero de momento alcanza con saber que es de tipo *std::time::Duration*.

Una vez que este corriendo el server, hasta que reciba un stop estará 'encendida' la funcionalidad que permite compartir piezas.

7.2. Acceptor

Una vez llamada run, se crea un thread que comenzará a escuchar conexiones, en un submódulo dentro del server llamado 'Acceptor'.

Una versión resumida de lo que hace es:

```

let listener: TcpListener = TcpListener::bind(&server_address)?;
listener.set_nonblocking(true)?;
let pool: ThreadPool = ThreadPool::new(workers_count)?;
for stream in listener.incoming() {
    match stream {
        Ok(stream) => {
            pool.execute(|| {
                let message_service = PeerMessageService::from_peer_connection(stream);
                ServerConnection::new(client_id, metainfo, Box::new(message_service)).run();
            });
        }
        Err(ref error) if error.kind() == ErrorKind::WouldBlock => {
            std::thread::sleep(sleep_duration);
        }
    }
}

```

Pasamos a continuación a explicar este ciclo. Primero, notamos que se usa un threadpool para manejar la cantidad máxima de threads con conexiones corriendo simultáneamente, hablaremos de esto en la próxima sección.

El método 'incoming' del listener nos devuelve una conexión de las encoladas que nos llegan al socket, pero lo hace de forma bloqueante. Es decir, si nunca nos llega una conexión, el flujo del thread se quedará infinitamente en ese método esperando que llegue algo. Por lo tanto, decidimos hacerlo no bloqueante manualmente, de tal forma que inmediatamente luego de ser llamada nos devuelve un Result donde, en caso de existir una conexión encolada, se devuelve un Ok() con el stream de la conexión, y caso contrario un error de tipo `std::io::ErrorKind::WouldBlock`, que no es exactamente un error, sino que significa que no existe una conexión nueva en el socket.

En caso de que no haya que parar verificamos entonces el contenido del stream. Primero veamos el caso en el que no hay ninguna conexión entrante. Cuando esto sucede, la idea es empezar el loop de 0, pero primero debemos mandar a dormir por un intervalo al thread. Esto es así porque al ser todas las lecturas no bloqueantes, si no entrara nunca en una conexión estaría todo el tiempo corriendo en un ciclo costoso de leer el channel y el socket, devolviendo inmediatamente que no se encontró conexión ni mensaje. Para evitar esto, que es costoso, usamos una duración de espera que recibimos por parámetro, que de momento es arbitraria de tal forma que funcione bien pero en un futuro se podría usar un algoritmo para que vaya variando según la frecuencia de nuevas conexiones.

Cuando la conexión si existe, encolamos en la threadpool una conexión nueva para ser manejada por un thread distinto, que veremos detalles de implementación en la próxima sección.

7.3. Server Connection

Ahora, veremos cual es el flujo dentro de una conexión con un peer una vez que la threadpool la efectivamente la ejecuta. El método 'run' de la conexión recibe por parámetro un servicio de mensajes entre peers, que es el trait donde están las verdaderas implementaciones de recibir y escribir mensajes del protocolo en un stream, que son las mismas que se usan para el cliente. Al recibir esto como una interfaz, podemos usar mocking para testear distintos flujos de las conexiones simulando que mensajes nos mandan peers externos a nosotros.

Una vez comenzada la conexión, esperamos recibir un handshake tal como indica el protocolo. Realizamos primero entonces las validaciones pertinentes, y le mandamos al otro peer tanto la

respuesta del handshake como un mensaje de Unchoke (para que sepa que nos puede pedir piezas) como un mensaje de tipo Bitfield con un bitmap de las piezas que ya tenemos, para que sepa cuales nos puede pedir. Para armar el bitfield, simplemente recorremos de disco el directorio en el que el cliente guarda las piezas, dado que tenemos un archivo distinto para cada pieza, y de esta forma evitamos perder tiempo mandandole mensajes al cliente.

Luego, directamente podemos ponernos a escuchar mensajes del socket, y responder acordemente dependiendo el tipo de mensaje. Para cualquier mensaje que sea distinto a 'Request', o bien lo ignoramos o terminamos la conexión. Esto es así porque al servidor no le importa otro tipo de mensajes, de esos se ocupan las conexiones que manejamos en el cliente. En un futuro, se podría agregar manejo a funciones como Have, Bitfield, Cancel u Interested, ya que podríamos dejar de compartir una pieza en caso de que nos lo pidan o usar informacion que nos manden para darsela al cliente o pedir piezas que nos sirvan, pero por simplicidad decidimos que el servidor solo se ocupe de compartir piezas.

Cuando el mensaje es de tipo request, significa que nos están pidiendo un bloque. En ese caso, el manejo es sencillo: Primero verificamos que la pieza efectivamente la tengamos buscando en disco el número de pieza, en ese caso la traemos a memoria. Traemos el slice correspondiente al bloque pedido como:

$$\text{Bloque} = [\text{Inicio}; \text{Inicio} + \text{largo bloque})$$

donde *inicio* y *largo* son parámetros del mensaje de tipo request. Con esto, armamos el mensaje de *Piece* con el payload correspondiente, insertando el bloque obtenido, y lo escribimos en el socket. Una versión resumida de esto es:

```
fn handle_request(message_service, request) {
    if !client_has_piece(request.index) return;

    let piece_data: Vec<u8> = read_piece_from_disk(request.index);
    let block = piece_data[request.begin..(request.begin + request.length)].to_vec();
    let message = PeerMessage::piece(block, request.index, request.begin, request.length);
    message_service.send_message(message);
}
```

7.4. ThreadPool

Por último, veremos detalles sobre por qué y como implementamos un threadpool para manejar la cantidad de conexiones simultáneas que tiene el server. Primero, repasemos que es un threadpool. Si nosotros no pusieramos un límite sobre la cantidad máxima de conexiones al mismo tiempo que podemos tener, entonces podría altentarse mucho y/o colapsar en caso de que muchas conexiones entren en un corto periodo de tiempo, ya que cada una de estas corre en un thread distinto. El threadpool es una forma de no solo limitar este número, sino de encolar las conexiones restantes para no descartarlas si superan el límite de tal forma que eventualmente sean atendidas.

La idea es tener como bien dijimos un número fijo de threads simuláneos, o *workers* que se ocupan de agarrar 'trabajos' o 'jobs' de una estructura de tipo FIFO, ejecutarlos e ir a buscar otro una vez terminado, repitiendo el ciclo. Le decimos 'job' en vez de conexión porque este concepto no es sólo útil para un servidor, sino para cualquier aplicación donde se quiera limitar la cantidad de cierto cómputo que se quiera ejecutar de forma concurrente o paralela.

El threadpool lo implementamos en un submódulo dentro del módulo dentro del servidor. La API es bastante sencilla, simplemente se crea y primero indicando la cantidad de workers que se quiere, y luego, cada vez que se quiera implementar un trabajo, se llama al método 'execute' pasándole un closure con el código a ejecutar.

Para implementarlo, usamos un channel para encolar los jobs, que son de tipo:

```
type Job = Box<dyn FnOnce() + Send + 'static>;
```

El Receiver del channel lógicamente debe estar protegido por un Lock, ya que todos los workers deben leer del channel, entonces necesitamos que al leer intenten primero tomar el lock para evitar problemas de concurrencia, y soltarlo una vez leído. Al crear el threadpool, se genera entonces un thread por cada worker, y a cada uno se le pasa una copia del lock del receiver. Los workers ejecutan indeterminadamente un loop que se puede resumir como:

```
loop {
    match receiver.lock() {
        Ok(receiver) {
            if let Ok(job) = receiver.recv() => {
                job();
            }
        }
        Err(_) => error!("A thread holding the lock panicked"),
    }
}
```

Al terminar el threadpool, implementamos un mensaje de tal forma que se esperen que todos los workers terminen su job actual, de tal forma que podría no terminar inmediatamente ya que también puede haber otros jobs encolados, porque luego de mandar el mensaje se hace un join del thread de cada uno de los workers. Una potencial mejora sería que implementemos el método 'Drop' cortando los jobs ejecutando, de tal forma de también evitar mensajes de stop para terminar la ejecución, ya que no es ideal por esta espera.

8. Implementación de Interfaz Grafica

La interfaz gráfica fue implementada con la librería GTK, para desarrollar interfaces con el modelo de GNOME. Para comunicar el cliente con la UI utilizamos un channel cuyo transmitter es clonado y utilizado en diferentes partes del cliente, todos mandando eventos referidos al módulo donde suceden. El receiver es utilizado para recibir todos estos eventos y ante ellos, modificar la UI para que esos cambios se vean reflejados.

A continuación se muestra la tab de información general donde se encuentra datos de cada torrent corriendo en simultáneo. Cosas como la cantidad de peers conectados, progreso, tiempo de espera, y mas datos relevantes que se encuentran en un dialog box si se clickea en details:



Figura 17: Tab de información general.

Además, cuenta con una tab de estadísticas de descarga donde se puede ver en forma de fila información sobre cada conexión con peer en particular: a que torrent pertenece, IP, piezas descargadas, velocidad, estado de conexión:

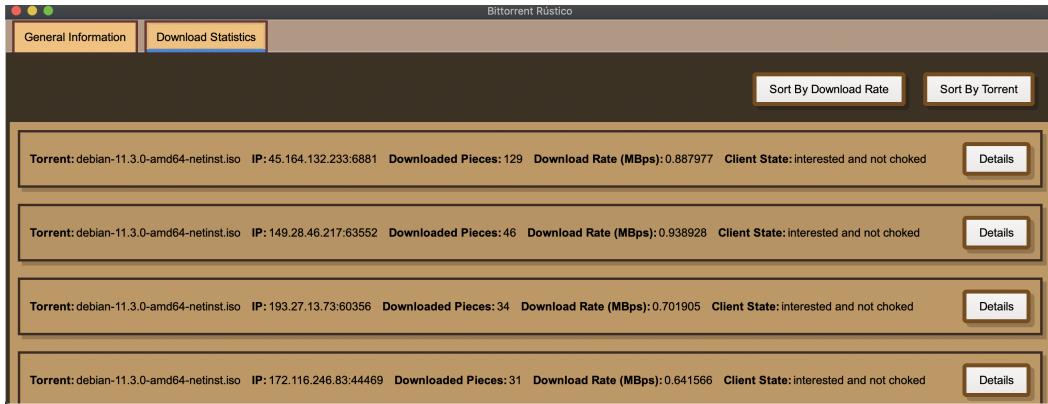


Figura 18: Tab de estadísticas de descarga.

8.1. Comunicación con el cliente

Tenemos muchos mensajes para definir diferente comportamiento, a continuación se muestran algunos:

```
pub enum UIMessage {
    AddTorrent(MetaInfo),
    PieceDownloaded(TorrentName, PeerId),
    NewConnection(TorrentName, PeerId),
    ClosedConnection(TorrentName, PeerId),
    UpdatePeerUploadRate(f32, PeerId),
    UpdatePeerDownloadRate(f32, PeerId),
    UpdateDownloadedPiece(Piece, PeerId),
    UpdatePeerConnectionState(PeerConnectionState, PeerId)
}
```

Cada uno indica la información necesaria para que la UI pueda buscar el elemento que corresponde y actualizarlo. A su vez, cada vez que se recibe un mensaje, la UI redirige el mensaje a cada una de las tabs y ellas lo handlean como deseen. De esta manera, muchas tabs pueden reaccionar de distinta manera ante el mismo mensaje. Un ejemplo es la actualización de la tab de estadísticas de descarga de los peers. Tenemos handle update que recibe un mensaje y en base a eso llama a alguna función para actualizarse (veremos a continuación como) o simplemente deja el mensaje pasar porque no lo necesita:

```
pub fn update(&mut self, message: &UIMessage) -> Result<(), DownloadStatisticsTabError> {
    match message {
        UIMessage::PieceDownloaded(_, peer_id) => {
            self.update_downloaded_pieces(peer_id)?;
        }
        UIMessage::UpdatePeerUploadRate(rate, peer_id) => {
            self.update_upload_rate(*rate, peer_id)?;
        }
        UIMessage::UpdatePeerDownloadRate(rate, peer_id) => {
            self.update_download_rate(*rate, peer_id)?;
        }
        _ => {}
    }
    Ok(())
}
```

8.2. Actualización de la Interfaz en base a los eventos

Para poder mantener la información de la UI actualizada, separamos el modelo de datos de los componentes de interfaz.

Al crear las tabs, bindeamos el modelo a los componentes, para que cada vez que se actualice el mismo, se vea instantáneamente reflejado en la interfaz. El modelo consta en listas que tienen información de torrents y peers, podemos ver un ejemplo:

```
pub struct DownloadStatistics {  
    torrentname: RefCell<Option<String>>,  
    id: RefCell<Option<String>>,  
    ipport: RefCell<Option<String>>,  
    clientstate: RefCell<Option<String>>,  
    peerstate: RefCell<Option<String>>,  
    downloadrate: RefCell<f32>,  
    uploadrate: RefCell<f32>,  
    downloadedpieces: RefCell<u32>,  
}
```

y para bindearlo, utilizamos algo del estilo (muy simplificado):

```
download_statistics_list.bind( |item| {  
    let label = Label::new();  
    item.bind_property(label, "ip");  
})
```

De esta manera, bindeamos la propiedad IP de un elemento de la lista a un label.

9. Manejo de errores

Rust provee distintas opciones para manejar los errores, que se pueden adaptar según se necesite para cada caso dentro del proyecto eligiendo entre varias opciones de manejo. Nuestro enfoque fue principalmente crear tipos de errores personalizados para todas las situaciones que puedan generar errores dentro de nuestro programa, agrupados por módulos o funcionalidad. Con la finalidad de atrapar los errores sin dejar que el programa aborte (usar panic o unwrap) y de dejar el código lo más limpio posible visualmente, intentamos usar donde sea posible el operador de burbujeo `?`, que devuelve un error o el valor dentro del Ok en caso de que no lo sea, pero terminando la función si es un error. Para poder usarlo fue necesario que las funciones devuelvan un 'Result' con un tipo de error que coincida o sea convertible con el trait 'from' a todos los posibles errores que puedan surgir dentro de la misma.

A continuación, vemos ejemplos de tipos de errores que definimos.

9.1. Errores que incluyen el error interno

Dado que los errores e definen como Enums, y los enums pueden incluir valores de cualquier tipo dentro, una posible solución para poder burbujejar errores es definir un 'tipo' del enum del error tal que tenga guardado el error que efectivamente sucede, y al burbujejar que se castee al de retorno de la función. Por ejemplo, vemos algunos de los posibles errores del servidor:

```
pub enum ServerError {
    TcpStreamError(std::io::Error),
    ThreadPoolError(ThreadPoolError),
    LoggerCreationError(LoggerError),
}
```

La idea es que las funciones creadas por nosotros que puedan fallar devuelvan errores de tipo 'ServerError', aunque internamente se usen funciones que, por ejemplo, puedan fallar por un *std :: io :: Error*, un *ThreadPoolError* (que también lo creamos nosotros), o *LoggerError*. Esta implementación cuando es posible es la ideal, por qué no perdemos información de por qué falló, ya que al imprimir el error podemos ver que falló internamente en cada uno de esos casos pero a un error de más alto nivel, implementando el display de la siguiente forma:

```
impl fmt::Display for ServerError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match self {
            ServerError::TcpStreamError(error) => write!(f, "TcpStream error: {}", error),
            ServerError::ThreadPoolError(error) => write!(f, "ThreadPool error: {}", error),
            ServerError::LoggerCreationError(error) => {
                write!(f, "Logger creation error: {}", error)
            }
        }
    }
}
```

Para poder burbujejar estos errores automáticamente y que estén incluidos a un más alto nivel, es necesario especificar como debe ser convertido cada uno en el error 'padre', en este caso, en un *ServerError*. Esto se hace implementando el trait from, y que mostremos que se crea el error poniéndolo dentro del *ServerError*. Vemos un ejemplo de esto para el caso el *TcpStreamError*, que se usa cuando un *TcpStream* falla devolviendo un error de tipo *std :: io :: Error*:

```
impl From<std::io::Error> for ServerError {
    fn from(error: std::io::Error) -> Self {
        ServerError::TcpStreamError(error)
```

```

    }
}

```

Un ejemplo de como se burbujearía el error luego es:

```

use crate::server::ServerError;

fn server_listen(ip: &str, port: u16) -> Result<(), ServerError> {
    let address: String = format!("{}:{}", address, port);
    // TcpListener::bind Result<TcpListener, std::io::Error>
    let listener: TcpListener = TcpListener::bind(address)?;
    // set_nonblocking devuelve Result<(), std::io::Error>
    listener.set_nonblocking(true)?;

    // ... se corre el servidor

    Ok(())
}

```

De esta forma, no es necesario mapear errores manualmente ni leer unwraps o manejos voluminosos con un match, sino que simplemente se agrega un '?' al final y el compilador de rust se ocupa de las conversiones de tipos en caso de error.

Desde donde se llama a la función 'padre' de 'server listen' (supongamos desde el cliente), también tenemos que manejar el error. En este caso, que falle el error al crearse es algo terminal para el servidor (el cliente sigue corriendo igualmente), pero es importante mostrar de alguna manera que el error sucedió. Entonces, desde el cliente se puede hacer algo como:

```

match server.listen(ip, port) {
    Ok(_) => logger.info("Server listening at {}:{}", ip, port),
    Err(e) => logger.error("Server failed to start listening: {:?}", e),
}

```

De esta forma, podemos obtener la causa del error subyacente.

9.2. Errores que incluyen texto con la razón de falla

En algunos casos, podemos llegar a detectar errores que no surgen por fallas de funciones ya existentes, sino que son de la propia lógica del programa, o por alguna razón no queremos usar el error ya existente. Cuando esto sucede, podemos hacer que los posibles valores del enum guarden no otro tipo de error, sino un texto con la razón del fallo generado por nosotros. Al hacer el display, alcanzaría con imprimir ese String, y las conversiones de tipos se harían de la misma manera que antes para los casos donde nosotros decidimos mapear otros errores a estos pero con mensajes personalizados.

Mostramos otro ejemplo, ahora del ThreadPool error que usamos antes, que internamente guarda los errores con Strings:

```

pub enum ThreadPoolError {
    CreationError(String),
    JoinError(String),
}

```

A veces, es mas claro modificar el tipo de error explícitamente dado el contexto donde queremos loggearlo. Por ejemplo:

```
fn stop_threadpool(self) -> Result<(), ThreadPoolError> {
    for worker in self.workers {
        worker.handle.join().map_err(|_| {
            ThreadPoolError::JoinError(
                format!("Join for worker with id {} failed", worker.id)
            )
        })?;
    }
}
```

De esta forma, descartamos el error del join que no nos interesa, y en vez de eso le damos un mensaje que sea significativo para nosotros.

10. Testing

Para hacer pruebas automatizadas, utilizamos tanto tests unitarios como de integración.

Dado que la cátedra no permitía usar librería de Mocks, fuimos por el camino de utilizar interfaz e implementación para cada una de las entidades de nuestra aplicación, así poder crear implementación real y la implementación de mock cuando queremos testear interacción compartida.

10.1. Caso de Mock: HTTP Service

Para testear la interacción con un tracker sin necesidad de probar con uno real, por todo lo que lleva, creamos un Mock de servicio HTTP que devuelve cierto contenido arbitrario cada vez que se le hace un get request:

```
pub struct MockHttpsService {
    pub read_bytes: Vec<u8>,
}

impl IHttpService for MockHttpsService {
    fn get(&mut self, _path: &str, _query_params: &str) -> Result<Vec<u8>, HttpsServiceError> {
        Ok(self.read_bytes.clone())
    }
}
```

Al construirlo, lo inicializamos con los bytes que va a devolver cuando desde algún lugar se llame a la función get del servicio:

```
let connection = MockHttpsService {
    read_bytes: bencode::encode(peers),
};

let mut tracker_service =
    TrackerService::from_metalinfo(&metainfo, config.listen_port, &peer_id, connection);
assert_eq!(tracker_service.get_peers(), peers);
```

10.2. Caso de Mock: PeerMessage Service

No todos los mocks son tan sencillos. Para mockear interacción con Peers que devuelven diferentes mensajes a través del tiempo (unchoke, piezas, bitfield, etc), tenemos que implementar algo más complejo. El mock de Peer Connection tiene un estado interno inicializado en la creación, que le dice qué bloques ir devolviendo en cada pedido.

```
pub struct PeerMessageServiceMock {
    pub counter: u32,
    pub file: Vec<u8>,
    pub block_size: u32,
}
```

Y para utilizarlo, al igual que en el caso de HTTP, lo inyectamos en la creación de los peers para que estos se comuniquen con el servicio de mensajes mock, completamente controlado por el test:

```
impl PeerConnection {
    pub fn new(
        peer: Peer,
        message_service: IMessageservice,
    ) -> PeerConnection {
        Peer Connection {
            peer: peer,
            message_service: message_service
        }
    }
}
```

10.3. Tests de Integración

Para los tests de integración utilizamos el PeerMessageServiceMock y hicimos que al correr la aplicación, todos los peers se conecten a estos servicios sin necesidad de interactuar con el mundo exterior, pero pudiendo probar casos borde que podrían aparecer en el mundo real. Lo bueno de utilizar mocks para los tests de integración es que permiten ser más confiables a la hora de correrlos y abarcan la lógica del programa entero. Simplemente declarando que parte del archivo compartiría cada Peer, pudimos probar la concurrencia del programa y como interactúan absolutamente todas las entidades entre sí. Esto nos permitió detectar muchos bugs que teníamos y que nos pasaban por alto al probar con torrents externos.

11. Obstáculos encontrados

A continuación, mencionamos algunos de los problemas que nos surgieron realizando el proyecto que valen la pena mencionar:

- **Problemas de concurrencia:** A lo largo del desarrollo del trabajo, especialmente cuando tuvimos que convertir el cliente de ser secuencial a concurrente al usar las piezas, nos encontramos con muchísimos errores causados por la concurrencia, especialmente de sincronización al tener que mandar mensajes entre entidades. En muchas situaciones considerábamos que siempre un mensaje iba a llegar primero que otro, pero esto no siempre es así, por lo cual tuvimos algunos bugs muy difícil de resolver ya que había que considerar todos los ordenes posibles. Nos ayudó mucho tener muchos tests, especialmente los de integración, para darnos cuenta de errores de este índole que de no tenerlos no nos hubieramos dado cuenta.
- **Dificultad de testing:** Muchos módulos encontramos que eran muy difíciles de testear, particularmente los que interactúan con peers o conexiones reales, tales como el tracker o módulos de conexiones. Para poder hacer los tests, decidimos usar traits y el concepto de servicios para modelar lo que deberían hacer estos módulos, de tal forma de poder simular correctamente los mensajes de estas entidades externas. Consideramos que hacer esto en rust es más engorroso y difícil que en otros lenguajes como Java de más alto nivel, aunque vale la pena considerar usar algún crate externo para hacer mocks que en este caso no teníamos permitido. Dicho esto, consideramos que tener un set robusto de pruebas nos ayudó muchísimo a dejar funcionando correctamente el proyecto, especialmente en las fases finales donde un pequeño cambio podía hacer dejar de andar alguna parte, y caso contrario sería muy difícil darse cuenta.
- **interfaz gráfica:** La librería GTK si bien es muy reconocida dada la larga trayectoria, no tiene una gran cantidad de contenido de ejemplo en rust. Esto significa tener que navegar mediante páginas deprecated y ejemplo desactualizados de versiones previas. Además, el código que termina quedando es mucho boilerplate para hacer simples cosas. Sin embargo, una vez que se encuentra una estructura para comunicarse con entidades externas y recibir eventos, ya no es tan difícil como a primera vista.

- **Obtención de torrents reales:** Un problema con el que nos enfrentamos fue encontrar torrents que nos sirvan para hacer pruebas, ya que excepto los torrents oficiales de las distribuciones de Ubuntu y Debian, el resto de los torrents tenían o bien redes extremadamente inactivas donde no había peers suficientes para realizar la descarga, o bien los metainfos tenían formatos extraños no compatibles con el estándar. Por esto tuvimos que trabajar casi únicamente con el de Debian, que al pesar más de 300MB, tarda bastante en descargarse, haciendo muy lentas algunas pruebas.
- **Soporte para IPV6:** En el transcurso del desarrollo del proyecto, el tracker de la red de Ubuntu comenzó a mandar únicamente peers con IPV6 en vez de IPV4, lo cual nos trajo grandes problemas y retrasos dado que a ninguno de los integrantes del grupo nos soporta estas IPs nuestros proveedores de internet. Luego de días de intentar resolverlo sin éxito, terminamos migrando nuestras pruebas al torrent de Debian, que al mandar IPV4 lo pudimos usar sin mayores problemas.

12. Potenciales mejoras

Si bien consideramos que el resultado final es muy satisfactorio, se nos fueron ocurriendo muchas mejoras que mejorarían la velocidad de descarga, pero por cuestiones de tiempo no las pudimos implementar. Sin embargo, creemos que vale la pena mencionar algunas de ellas para potencialmente hacerlas en un futuro:

- Distribuir en varios threads el guardado de piezas en disco, ya que al tener un único thread en este momento que se encarga de validar y encolar las piezas que llegan en un channel este se satura, y hay siempre varias piezas encoladas por lo que ese thread tiene mucho trabajo.
- Terminar conexiones con peers que no tienen piezas que nos sirvan, dado que no nos aportan nada
- Agregar 'endgame mode': Existen algoritmos muy específicos para descargar las piezas finales y más raras, que podrían acelerar mucho la parte final de la descarga. Consiste principalmente en pedir a muchos peers distintos estas piezas, y una vez que nos llega cancelar el resto de los pedidos.
- Usar caching en el server al leer piezas de disco, ya que actualmente por cada pedido de un bloque, la ServerConneciton se trae la pieza entera y calcula el bloque que le sirve por cada request. Una mejor opción sería detectar el primer pedido de un bloque de una pieza específica y traerla entera a memoria y perdurarla por un tiempo, para no leer de disco en cada pedido.
- Mandar keep alive a los peers para que no nos corten conexiones

13. Conclusiones

Para cerrar el informe nos gustaría hablar brevemente de nuestra experiencia trabajando en este proyecto.

En primer lugar, queremos mencionar que el lenguaje Rust nos pareció muy útil. En un principio, al ser fuertemente tipado y bastante estricto, nos costó usarlo. Pero a medida que pasaba el tiempo fuimos observando el lado bueno de estas características, y le sacamos ventaja a todas las indicaciones que ofrecía cargo. Notamos que fueron especialmente útiles para el manejo de concurrencia, indicando los traits que son necesarios en distintos objetos para que un programa sea thread-safe.

También es importante mencionar la dinámica de trabajo con la que nos manejamos a lo largo del cuatrimestre. El uso de GitHub Jobs, las reviews cruzadas entre Pull Requests y el uso de issues nos fueron de especial ayuda a la hora de asegurar que el estado del proyecto cumple con todas las características requeridas y para la creación y división de tareas, porque permitieron hacerlo de una manera muy fácil y conveniente.

Cabe destacar que para varios de nosotros, es la primera vez que trabajamos en un proyecto de este calibre, por lo que fue una experiencia nueva e interesante, que puso a prueba las habilidades y conocimientos que adquirimos durante la carrera. Definitivamente es nuestra intención guardarnos todo lo que podamos de esta experiencia, y aplicarlo en proyectos futuros.