

Taller de Programación I

Informe

Grupo	La Deymoneta
Integrantes	Felipe Marelli (106521) Joaquín Prada (105978) Lucas Sotelo (102730) Nicolás Continanza (97576)
Período	1er cuatrimestre - 2022

[Informe](#)

[Introducción](#)

[Arquitectura de la aplicación y flujo principal](#)

[Componentes y operaciones más relevantes](#)

[Bt Client](#)

[Config Manager](#)

[Logger](#)

[Torrent Parser](#)

[Tracker Handler](#)

[Torrent Handler](#)

[Torrent Status](#)

[Peer Session](#)

[Storage Manager](#)

[Server](#)

[UI](#)

[Comparaciones de performance](#)

[Conclusiones y aprendizajes](#)

Introducción

Se desarrolló **dTorrent**, un cliente BitTorrent utilizando el lenguaje de programación Rust. Se realizó en un período de 11 semanas. En el presente informe detallaremos el proceso de desarrollo del proyecto y la arquitectura de la aplicación con sus componentes más relevantes, describiéndolas y mencionando las decisiones tomadas para su implementación.

Arquitectura de la aplicación y flujo principal

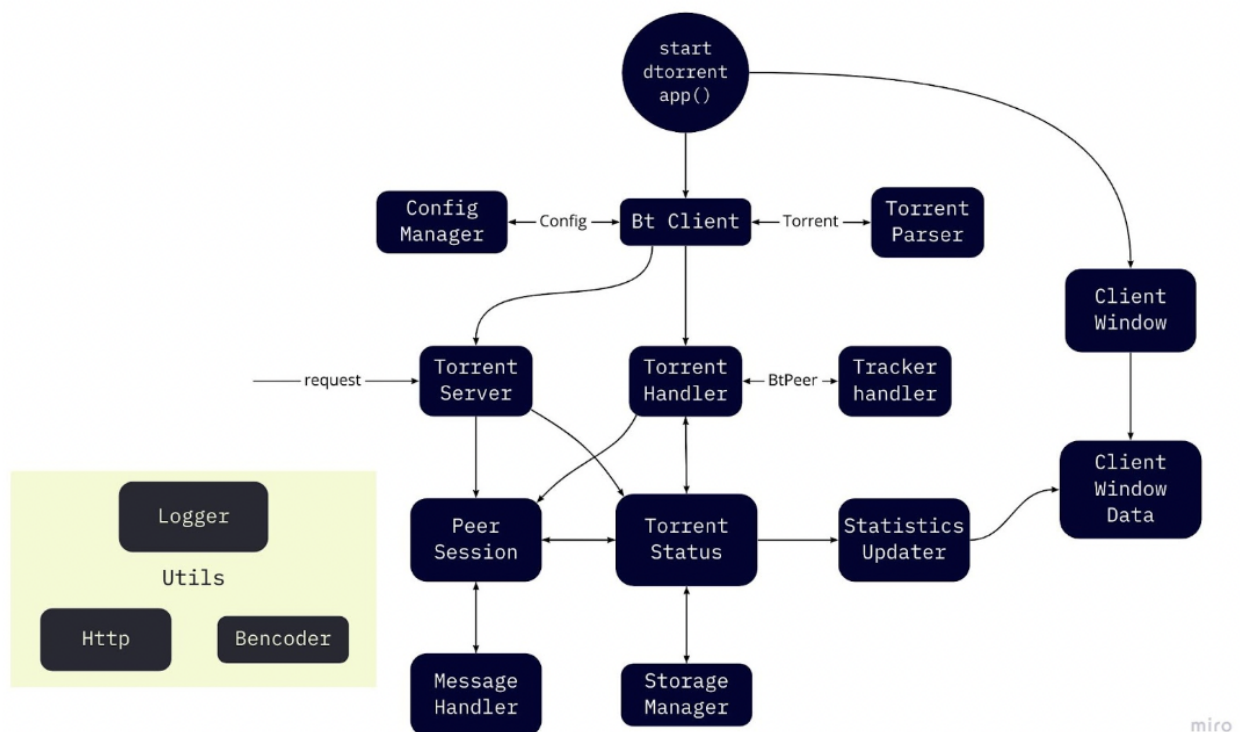


Diagrama de arquitectura

La aplicación inicia con la creación de la interfaz gráfica y del `BtClient`, que es el componente responsable de inicializar y administrar varios de los componentes que intervienen en el proceso de descarga de un torrent a partir del input del sistema, y de enviarle las estadísticas de descarga a la interfaz gráfica. Interactúa con cuatro entidades de la aplicación:

- `ConfigManager` : contiene la información obtenida del archivo de configuración, realizando algunas validaciones sobre la misma (validez de la ruta al archivo de configuración, formato válido, etc.).
- `TorrentParser` : lee el archivo torrent, lo decodifica utilizando el `Bencoder` y devuelve un struct `Torrent` con los datos necesarios para la comunicación con el tracker y los peers.
- `TorrentServer` : abstracción utilizada para manejar el comportamiento del cliente como *seeder* de piezas de un torrent.
- `TorrentHandler` : abstracción utilizada para manejar el comportamiento del cliente como *leecher* de piezas de un torrent. Es responsable de conectarse al *tracker* (a través del `TrackerHandler`), a cada *peer* y de mantener la entidad `TorrentStatus` que representa el estado de la descarga.

A su vez, las siguientes entidades intervienen en el proceso de descarga y subida:

- `TrackerHandler` : para cada torrent, obtiene la lista de peers del tracker a través de un request HTTP (responsabilidad de la entidad `HttpHandler`) y se lo envía al `TorrentHandler`.
- `PeerSession` : representa la conexión con un *peer* (o un *leecher*) y es responsable de intercambiar mensajes con el mismo, utilizando el `MessageHandler`.
- `StorageManager` : se ocupa de guardar las piezas descargadas en el archivo de descarga.
- `StatisticsUpdater` : recibe y mantiene las estadísticas sobre la descarga que son enviadas a la interfaz gráfica.

Además del `Bencoder` y el módulo `Http` ya mencionados, se implementó un `Logger` que es utilizado por múltiples entidades de la aplicación para publicar los logs del sistema en los archivos creados para tal fin.

Finalmente, la interfaz gráfica cuenta con dos entidades principales:

- `ClientWindowData` : obtiene del modelo las estadísticas correspondiente a cada torrent y sus *peers*, y las dispone de manera tal que puedan ser manipuladas por `ClientWindow`.
- `ClientWindow` : representa al contenido de la interfaz y su disposición en la pantalla.

Componentes y operaciones más relevantes

Bt Client

El `BtClient` es la abstracción que representa al cliente en su totalidad. Se encarga de iniciar el cliente realizando las verificaciones necesarias, y de empezar la descarga y subida de todos los archivos Torrent que se encuentren en el directorio. Esto lo hace mediante los dos métodos detallados a continuación, delegando la responsabilidad en los diferentes componentes del proyecto:

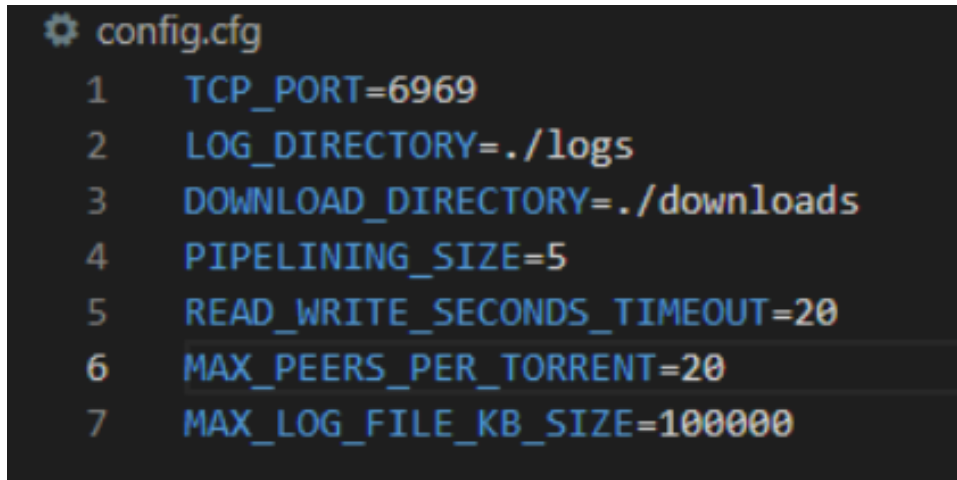
- `BtClient::init()` : Lo primero que sucede al iniciar el cliente es que se intenta leer el archivo de configuración mediante el **Config Manager**. Si se pudo procesar correctamente el archivo, se inicia el **Logger**. Luego, se abre el directorio recibido por consola, y se procesan todos los archivos con extensión `.torrent` haciendo uso del **Torrent Parser**, almacenando los correctamente leídos en una lista. Finalmente, se genera el *PeerID* concatenando el nombre del cliente con una serie de números aleatorios.
- `BtClient::run()` : Este método crea un **Torrent Handler** para cada archivo Torrent, cada uno en un *thread* separado. De esta manera se empiezan las descargas. Luego, inicia el **Statistics Updater**, que se encargará de actualizar la interfaz gráfica con las estadísticas de las descargas cada 300ms. Finalmente, crea el **BtServer** en un nuevo *thread* para que empiece a *seedear* las piezas descargadas.

Config Manager

El `Config Manager` se encarga de parsear el archivo `.cfg` que se encuentra en el directorio de ejecución. Además, se verifica que se cuente con todos los settings necesarios y que estén en buen formato. Se compone de 7 distintos settings, donde se encuentran:

- **TCP_PORT**: el puerto donde se va a escuchar desde el servidor.
- **LOG_DIRECTORY**: el directorio donde se guardan los logs.
- **DOWNLOAD_DIRECTORY**: el directorio donde se guardan las descargas.
- **PIPELINING_SIZE**: el tamaño del pipelining de descarga.
- **READ_WRITE_SECONDS_TIMEOUT**: el tiempo máximo, en segundos, para esperar una respuesta de un peer.

- **MAX_PEERS_PER_TORRENT:** máxima cantidad de peers conectados en un mismo momento a un torrent.
- **MAX_LOG_FILE_KB_SIZE:** el tamaño máximo, en KiloBytes, que puede alcanzar un archivo de log.



```

config.cfg
1  TCP_PORT=6969
2  LOG_DIRECTORY=./logs
3  DOWNLOAD_DIRECTORY=./downloads
4  PIPELINING_SIZE=5
5  READ_WRITE_SECONDS_TIMEOUT=20
6  MAX_PEERS_PER_TORRENT=20
7  MAX_LOG_FILE_KB_SIZE=100000

```

Ejemplo de archivo config

Logger

Componente encargado de manejar el loggeo en el programa. Su implementación consiste de usar `Channels` para comunicar un thread que quiere loggear con el que escribe en el archivo.

Usar channels nos permite que sea **Thread-Safe** y **no bloqueante**, que se pueda loggear y seguir la ejecución sin perder tiempo de ejecución.

Se divide en 2 componentes principales, el `LoggerReceiver` y el `LoggerSender`.

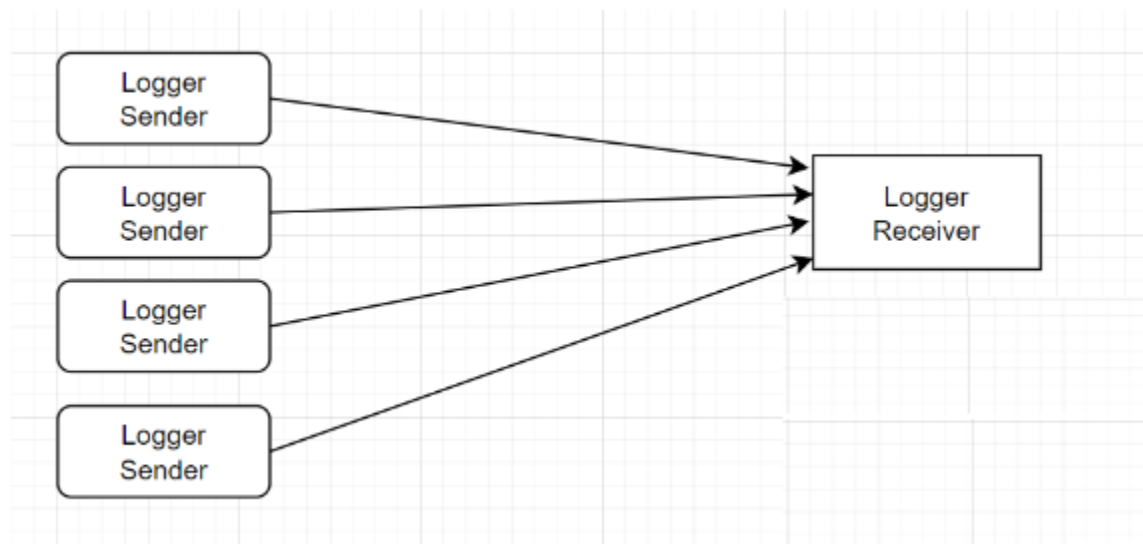
El **LoggerReceiver** funciona en un thread aparte y su función es estar escuchando constantemente nuevas entradas en el channel para loggearlas. Sólo puede haber un logger receiver por archivo.

El **LoggerSender** es el componente que envía la solicitud de loggeo al channel conectado con el receiver. Se puede tener tantos Loggers Senders como se necesiten clonándolos.

Hay 3 formas de loggear a través del **LoggerSender**:

- **Info():** para información general, por ejemplo que se descargó correctamente una pieza.

- **Warn():** para un error no crítico, que puede seguirse la ejecución normal del programa.
- **Error():** para un error crítico que puede terminar con la ejecución del programa.



Más de un Logger Sender apunta a un solo Logger Receiver

Torrent Parser

El archivo .torrent tiene que ser parseado pues contiene información sobre los archivos y el tracker. El mismo se encuentra encodeado en formato bencode, por lo cual fue necesario implementar un decoder y un encoder de dicho formato.

Para decodificar el torrent, se lo lee completo en memoria y se recorre recursivamente identificando cada tipo definido por el formato bencode (byte strings, integers, listas o diccionarios) que luego se guardan en un enum `Bencode`. Una vez que se tiene dicho enum, se puede usar para armar el struct `Torrent` que contiene toda la información del archivo .torrent.

Lo interesante a la hora de implementar el decodificador fue que se debía mantener el orden del diccionario *info* que está en el torrent (porque para obtener el infohash hay que aplicarle un hash, y si el orden cambia, el hash también lo hará), entonces se recurrió al uso de un `BTreeMap` en lugar de un `HashMap` cuando se parsean los diccionarios, puesto que mantiene el orden de inserción de las keys.

El encoder de bencode fue necesario para obtener el infohash, porque este se obtiene hasheando con SHA-1 el bencode del diccionario info que se encuentra en el torrent.

Recibe un enum `Bencode` y lo recorre para armar un string siguiendo el formato bencode.

Tracker Handler

Para la comunicación con el tracker primeramente fue necesario poder interpretar la URL announce que se encuentra dentro del archivo .torrent. Para esto se implementó un parser que separa el host, el puerto y el endpoint para el announce, y también identifica si el protocolo es HTTP o HTTPS.

Una vez obtenida la URL se tienen que realizar requests a la misma, por lo tanto se cuenta con un módulo `http` que implementa el protocolo HTTP de manera básica. Dicho módulo abre un stream de TCP con el tracker, arma los headers y query parameters correspondientes, y luego envía el request. Finalmente, se ocupa de parsear la respuesta recibida en un struct `TrackerResponse` y de esta manera ya se tiene una lista de peers a los cuales conectarse.

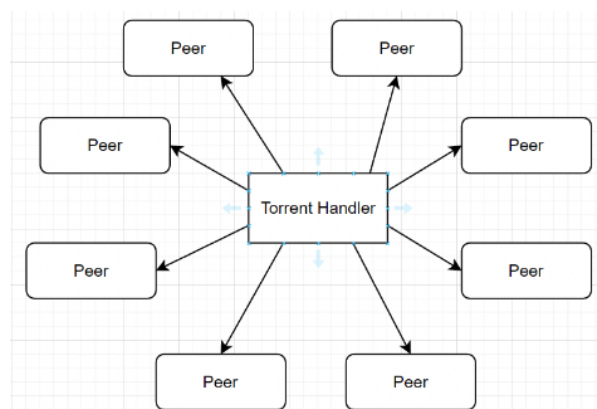
Torrent Handler

La entidad encargada de empezar la descarga del torrent es el `TorrentHandler`. Empieza creando un `TorrentStatus` en un nuevo thread que mantendrá toda la información actual sobre el torrent y luego inicializa la conexión con el tracker a través del `TrackerHandler`.

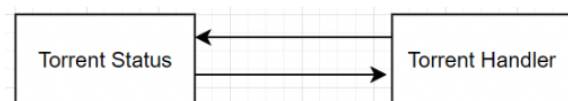
La función principal que va a realizar esta entidad y repetir hasta que se termine la descarga es:

1. Pedirle una nueva lista de peers al tracker.
2. Recorrer la lista e intentar conectarse a ellos.
3. Si se alcanzó el límite de peers conectados configurado en los settings, espera hasta que el status le avise que se desconectó algún peer.

Una vez que se termina la descarga, se lo avisa al logger y termina su ejecución.



Un Torrent Handler crea múltiples Peer Sessions



El Torrent Handler consulta con el Torrent Status

Torrent Status

El `TorrentStatus` es el encargado de manejar toda la concurrencia del programa, manteniendo la información actual del torrent para que pueda ser accedida desde cualquier componente y thread que lo necesite. Está implementado utilizando `Mutex` y `Atomic` dependiendo del tipo de dato, de esta manera se consigue que sea *Thread-Safe* pero bloqueante. Para mitigar los bloqueos se optó por crear un *Mutex* o *Atomic* por cada atributo en vez de un solo mutex por fuera, de esta manera se consigue que solo sea bloqueante si se accede al mismo atributo.

Entre sus responsabilidades se encuentran:

Mantener el estado de las piezas

Se utiliza un *HashMap* con el índice como key y un `PieceStatus` como valor. La pieza puede tener como estado `Finished` si ya se terminó de descargar, `Downloading` si está descargándose por algún peer o `Free` si está libre para ser descargada.

Mantener la cantidad de peers conectados

Hay dos tipos de peers conectados que es necesario mantener: los peers que ya efectuaron un `Handshake` correctamente y los que se están intentando conectar. Se necesita la cantidad de peers conectándose para no superar nunca el número máximo de peers a los que es posible conectarse, pero tampoco queremos mostrar, por ejemplo en la `UI`, peers que aún no se conectaron. Es por eso que necesitamos mantener los 2 tipos.

Mantener el estado de todas las sesiones de peers

Se necesita saber el estado de cada peer para saber cosas como su velocidad de descarga o si está `Unchoked`.

Seleccionar la próxima pieza a descargar

Cuando un `PeerSession` quiere descargar una pieza le pasa su bitfield al status para que elija una pieza `Free` aleatoria que coincida con el Bitfield. Si no hay ninguna pieza libre le devuelve un *None*. Si solo quedan piezas en estado `downloading`, estamos finalizando la descarga, hacemos uso del `End Game`. Consiste en darle la misma pieza a múltiples peers y guardar la que descarga más rápido.

Guardar / Obtener piezas

Si se finaliza la descarga o se quiere obtener una pieza ya descargada, el status llama al `StorageManager` para guardar u obtener la pieza.

Peer Session

Este componente es el encargado de coordinar los mensajes enviados y recibidos por cada peer a los que estamos conectados. Dentro de los mensajes posibles se encuentra el pedido de bloques a otros peers y la recepción de los mismos, por lo tanto este componente se encarga también de la descarga de las piezas.

Posee un loop para escuchar mensajes entrantes, luego parsearlos con ayuda del `MessageHandler` y finalmente realizar las acciones correspondientes al mensaje recibido.

Al momento de enviar mensajes lo hace de manera similar: le avisa al `MessageHandler` para que arme el mensaje con el payload correspondiente que luego es enviado por el stream.

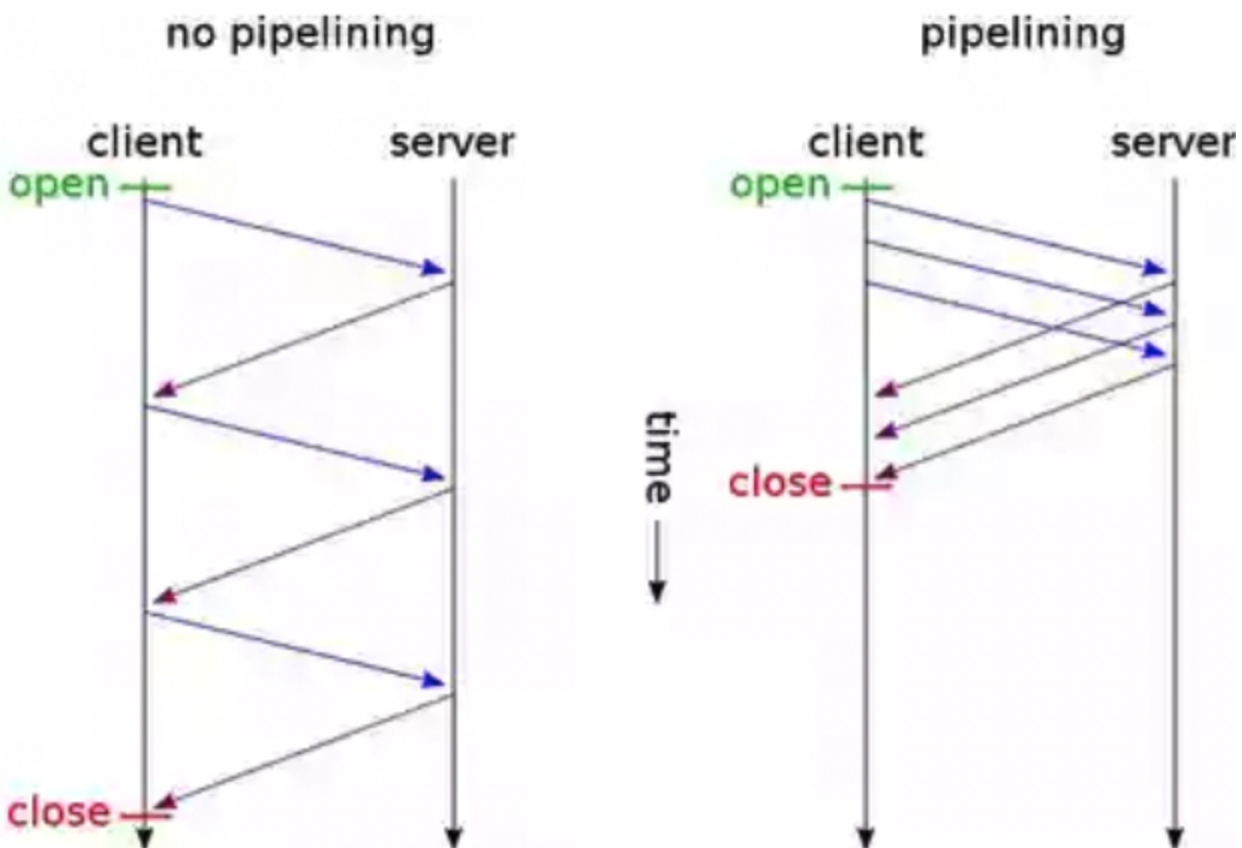


Imagen ilustrativa de como se realizan los requests haciendo un pipelining de los mismos

Para descargar una pieza, por lo general, es necesario realizar varios *requests* de bloques de la pieza a un peer, que tienen un cierto tamaño en bytes y una vez recibido el bloque pedido se lo guarda en disco. Un primer enfoque posible es enviar un request y luego esperar hasta recibir el bloque pedido, repitiendo hasta tener toda la pieza. La desventaja de este método es que se pierde tiempo esperando, por lo que se decidió optar por una implementación superadora llamada **pipelining** de requests, donde primero se envía una cierta cantidad de requests de bloques y, en vez de esperar entre request y request, se espera luego de haber mandado todos esos requests. De esta forma se hace más ágil la descarga de piezas al pasar menos tiempo esperando las respuestas.

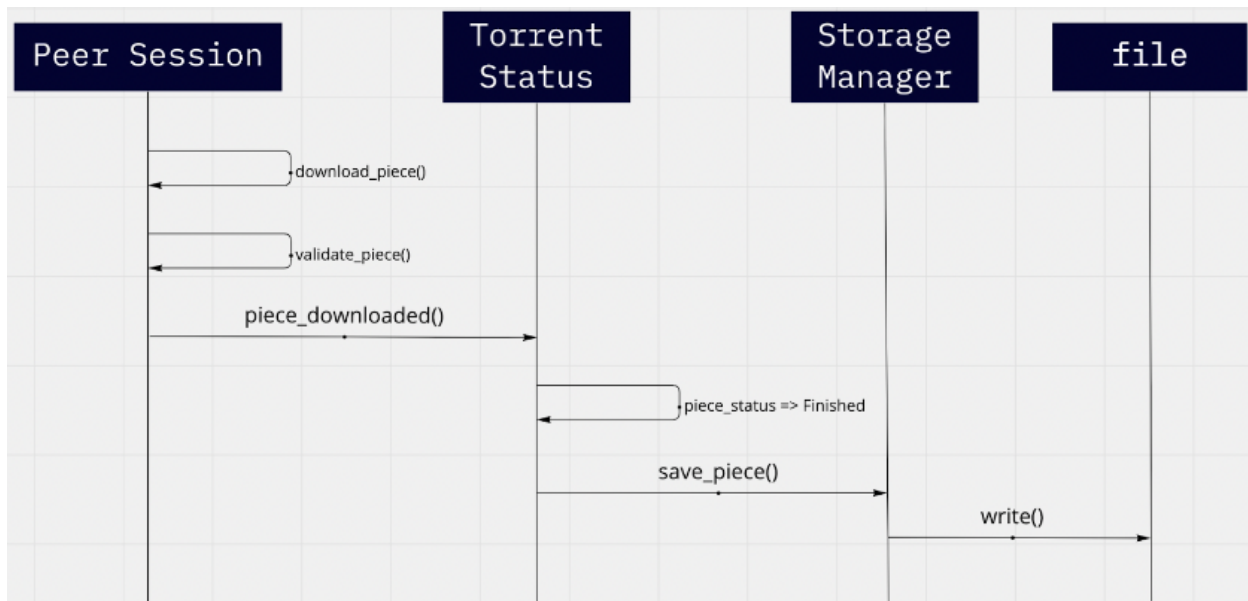


Diagrama de secuencia de la descarga de una pieza

Una vez descargada una pieza, antes de ser almacenada, se le aplica SHA-1 y se compara el hash obtenido con el hash que le corresponde (especificado en el archivo .torrent). Si el hash coincide, la pieza es válida y se la marca como descargada, para luego ser guardada en el archivo de descarga.

Storage Manager

Tanto para el almacenamiento de una pieza en un archivo (cuando las descargamos) como para la obtención de un bloque (cuando estamos sirviendo piezas a otros peers), contamos con este componente para realizarlo.

Guardar una pieza

Las piezas están divididas en bloques y dichos bloques son los que se envían y reciben entre peers. En nuestra implementación el archivo se va almacenando de a piezas, es decir, se tiene un buffer que representa una pieza y se va llenando con cada bloque descargado que pertenece a la misma. Recién al momento de tener la pieza completa en memoria, se la almacena en disco.

Para guardar una pieza necesitamos saber a qué parte del archivo pertenece. Esto lo obtenemos multiplicando su índice por su tamaño en bytes (tamaño que se encuentra en el archivo .torrent) y con esto tenemos el offset en bytes del inicio de la pieza dentro del archivo. Por lo tanto, se realiza un seek en el archivo al offset obtenido y se guarda la pieza en ese lugar.

Obtener un bloque

Obtener un bloque de un archivo que ya tenemos descargado es similar a guardar una pieza: se hace un seek al inicio del bloque y se lee la cantidad de bytes pedidos. En este caso, se habla de bloques y no de piezas, porque lo que vamos a enviar son justamente bloques.

Server

El `BtServer` se encarga de manejar las conexiones entrantes a través de un `TcpListener`.

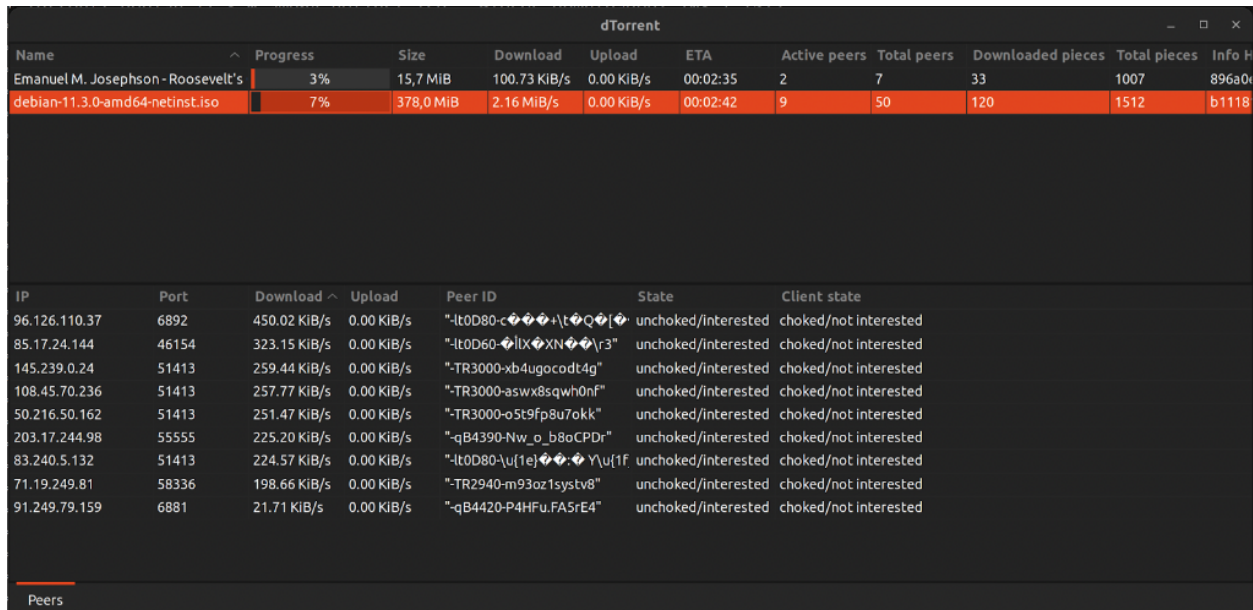
Al recibir una nueva conexión lo primero que hace es recibir el `Handshake` del peer que se quiera conectar, y si no se puede recibir se cierra la conexión.

Luego, a través del `Info Hash` obtenido en el handshake, se puede identificar a qué torrent quiere descargar. Esto es importante ya que se pueden estar 'seedeando' múltiples torrents al mismo tiempo, y ésta es la única manera de reconocerlos.

Por último, luego de ya tener el torrent identificado junto con su torrent status, el server pasa a crear un nuevo `Peer Session` desde el lado del leecher y vuelve a escuchar nuevas conexiones.

El servidor también respeta el límite máximo de peers conectados, por lo que antes de crear el *peer session* se fija si hay espacio libre para crear uno nuevo.

UI



The screenshot shows the dTorrent application window. The top section displays a list of torrents with columns for Name, Progress, Size, Download, Upload, ETA, Active peers, Total peers, Downloaded pieces, Total pieces, and Info Hash. Two torrents are listed: 'Emanuel M. Josephson - Roosevelt's' and 'debian-11.3.0-amd64-netinst.iso'. The bottom section shows a list of peers with columns for IP, Port, Download, Upload, Peer ID, State, and Client state. The 'Peers' tab is selected at the bottom.

Name	Progress	Size	Download	Upload	ETA	Active peers	Total peers	Downloaded pieces	Total pieces	Info Hash
Emanuel M. Josephson - Roosevelt's	3%	15,7 MiB	100.73 KiB/s	0.00 KiB/s	00:02:35	2	7	33	1007	896a0c...
debian-11.3.0-amd64-netinst.iso	7%	378,0 MiB	2.16 MiB/s	0.00 KiB/s	00:02:42	9	50	120	1512	b1118...

IP	Port	Download	Upload	Peer ID	State	Client state
96.126.110.37	6892	450.02 KiB/s	0.00 KiB/s	"lt0D80-c...+tQ..."	unchoked/interested	choked/not interested
85.17.24.144	46154	323.15 KiB/s	0.00 KiB/s	"lt0D60-lX...XN...r3"	unchoked/interested	choked/not interested
145.239.0.24	51413	259.44 KiB/s	0.00 KiB/s	"-TR3000-xb4ugocodt4g"	unchoked/interested	choked/not interested
108.45.70.236	51413	257.77 KiB/s	0.00 KiB/s	"-TR3000-aswx8sqwh0nf"	unchoked/interested	choked/not interested
50.216.50.162	51413	251.47 KiB/s	0.00 KiB/s	"-TR3000-oSt9fp8u7okk"	unchoked/interested	choked/not interested
203.17.244.98	55555	225.20 KiB/s	0.00 KiB/s	"-qB4390-Nw_o_b8oCPDr"	unchoked/interested	choked/not interested
83.240.5.132	51413	224.57 KiB/s	0.00 KiB/s	"lt0D80-uf1e)...Y\uf1f"	unchoked/interested	choked/not interested
71.19.249.81	58336	198.66 KiB/s	0.00 KiB/s	"-TR2940-m93oz1systv8"	unchoked/interested	choked/not interested
91.249.79.159	6881	21.71 KiB/s	0.00 KiB/s	"-qB4420-P4HFu.FA5rE4"	unchoked/interested	choked/not interested

Para el diseño de la interfaz gráfica se cuenta con un archivo XML exportado de [Glade](#), donde se modelaron a las dos pestañas de estadísticas del torrent y de sus peers respectivos utilizando un `GtkListStore` contenido en un `GtkTreeView` en cada caso. De esta manera, se envían los structs `TorrentStats` y `PeerStats` desde la entidad `StatisticsUpdater` hacia `ClientWindow` utilizando un channel de glib, para de esta forma mostrar la información correspondiente a cada celda en las columnas de cada `ListStore`.

Para cambiar las estadísticas sobre peers mostradas en la pestaña inferior según qué fila está seleccionada en la pestaña de estadísticas de torrents, se utilizó la signal de `gtk` correspondiente a un único click sobre una fila en el `GtkTreeView`, que permite identificar rápidamente cuál fue la fila sobre la que se hizo click y acceder a los datos que contienen sus celdas. A partir de esto es sencillo encontrar el torrent asociado a esa fila y mostrar debajo las estadísticas sobre sus peers asociados.

Comparaciones de performance

Se realizaron pruebas de velocidad para comparar con otros clientes comerciales.

Los clientes utilizados fueron **uTorrent** (Stable 3.5.5 Build 45952) y **qBitTorrent** (v4.4.3.1).

La prueba consistió en descargar 2 torrents en simultaneo, **debian-11.3.0-amd64-netinst.iso** y **Emanuel_M.Josephson._1955_PDF_EN_rutracker-5450718**.

Se limitaron a 100 peers máximos en nuestro cliente.

Los resultados fueron los siguientes:

- **dTorrent**
 - **Debian** descargado en 1 minuto y 30 segundos.
 - **Emanuel_M.Josephson** descargado en 2 minutos y 59 segundos.
- **uTorrent**
 - **Debian** descargado en 38 segundos.
 - **Emanuel_M.Josephson** descargado en 37 segundos.
- **qBittorrent**
 - **Debian** descargado en 1 minuto 25 segundos.
 - **Emanuel_M.Josephson** descargado en 1 Minuto 27 segundos.

Viendo los resultados, el claro ganador es uTorrent con los mejores tiempos. Contra qBittorrent la descarga de debian estuvo con tiempos similares.

Algo a destacar es que, aunque hayamos limitado los peers a 100, solo se alcanzó a conectar a 20/30 peers simultaneos, mientras que los otros dos clientes alcanzaban los ~80 en debian. Esto puede estar pasando ya que hay peers que deben pasar la IPv6 para conectarse y no disponemos de la configuración necesaria del modem para poder aceptarlos. También pueden estar rechazandonos la conexión por no estar utilizando un cliente conocido.

Conclusiones y aprendizajes

- Rust era un lenguaje desconocido para los cuatro integrantes del equipo al momento de comenzar la cursada de la materia y resultó ser una herramienta rápida, eficiente, robusta y con una documentación de gran calidad, a cambio de una curva de aprendizaje pronunciada en el inicio.
- A nivel herramientas, el uso de `gtk-rs` para el desarrollo de la interfaz gráfica fue la más difícil de incorporar al proyecto debido a los pocos ejemplos que pudimos encontrar a modo de referencia y la calidad de su documentación.
- El desarrollo de este proyecto fue una muy buena forma de aprender un lenguaje de programación nuevo y un protocolo complejo que conocíamos solo superficialmente en un período de tiempo relativamente corto.
- La metodología de *pair programming* adoptada para casi la totalidad de las tareas que compusieron al desarrollo integral del proyecto fue muy beneficiosa para avanzar rápido y con una primer revisión de código inmediata.
- Trabajar organizando *sprints* semanales fue muy útil para ordenar el trabajo, establecer objetivos intermedios y tener la instancia de la reunión semanal con los docentes en el rol de clientes ayudó notablemente para validar las decisiones tomadas y el progreso del proyecto.
- Realizar más tests y pruebas de integración hubiera contribuido a entender problemas que surgieron en la etapa final, en particular dificultades asociadas al manejo de la concurrencia.