Taller de Programación I

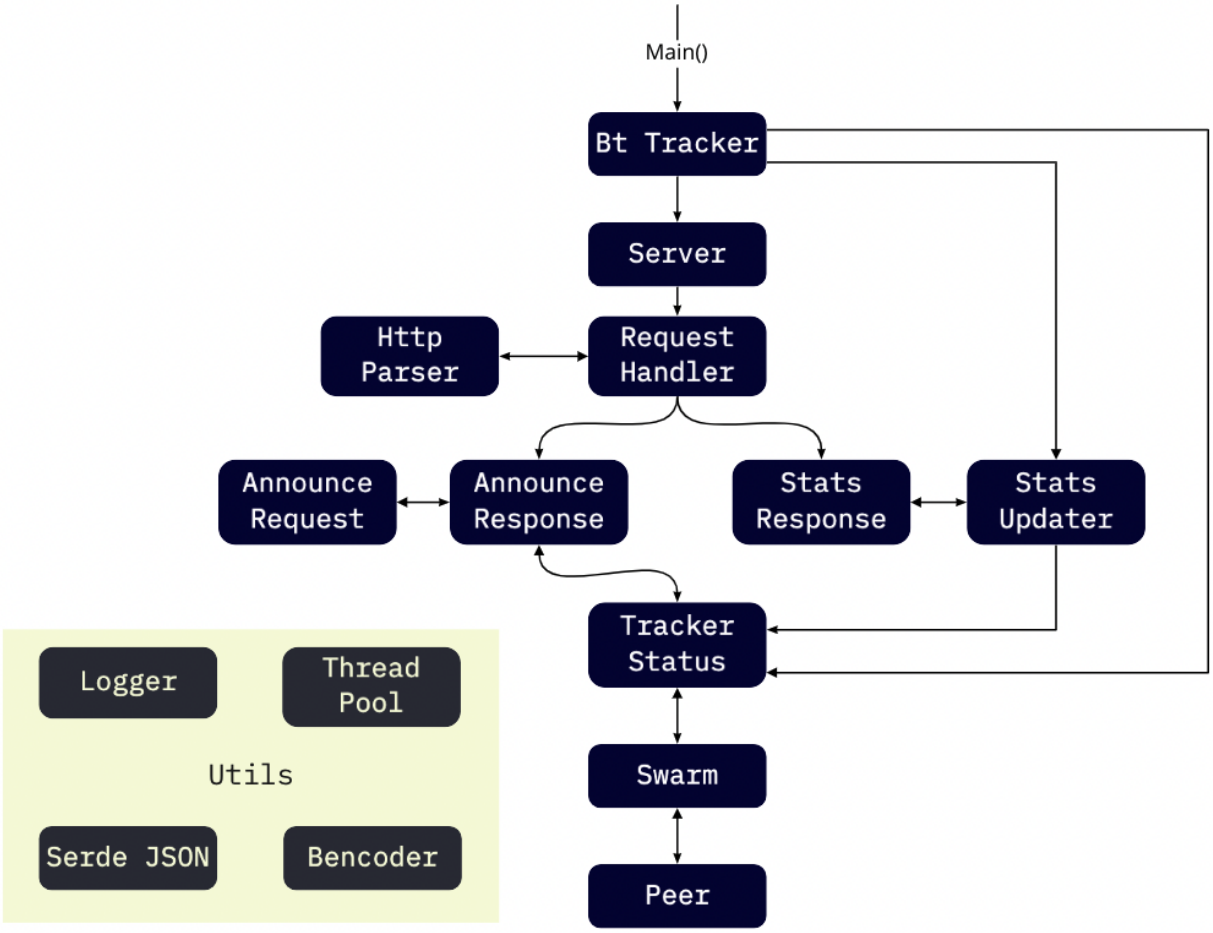
Informe

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

Introducción

Se desarrolló dTracker, un tracker de BitTorrent, en el lenguaje Rust como parte final del proyecto de la materia. 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



La aplicación comienza con la inicialización y ejecución del `BtTracker`, que tiene un rol equivalente al de `BtClient` de la primera parte del proyecto. Este componente será el responsable de levantar el servidor HTTP, que se quedará escuchando la llegada de

requests HTTP a través de un TcpListener de la biblioteca estándar de Rust. El servidor HTTP gestiona la llegada de cada request al tracker, manejando cada una de ellas en un thread aparte haciendo uso de la entidad `Request Handler`. Allí se leerá y parseará el contenido de la request colaborando con el `Http Parser` y, si lo recibido es válido, se procederá a crear la respuesta según el endpoint al que se haya enviado la request (`Announce Response` y `Stats Response`). Una vez obtenida la respuesta correspondiente según el caso, se actualiza el estado de las estadísticas si corresponde, se construye la respuesta a la request enviada y el `RequestHandler` envía la respuesta a través de su `TcpStream`, obtenido a partir del `TcpListener` del `Server`.

Componentes y operaciones más relevantes

BtTracker

El `BtTracker` es la abstracción que representa al *tracker* en su totalidad, y funciona como punto de entrada del programa. Se encarga de crear las estructuras necesarias para su funcionamiento, y finalmente de iniciar el servidor que manejará los pedidos entrantes.

También tiene la responsabilidad de iniciar el `Stats Updater` en un nuevo hilo de ejecución, para que pueda actualizar el historial de estadísticas cada 1 minuto.

Server

El `Server` es la estructura que se ocupa de escuchar las requests entrantes, distribuir su ejecución en threads y colaborar con el `RequestHandler` para la construcción de la respuesta.

Cuenta con una instancia de `ThreadPool`, inicialmente configurada para manejar 1000 threads, que se utilizará para ejecutar cada request en un hilo aparte.

Para escuchar requests cuenta con un `TcpListener` de la biblioteca estándar del lenguaje, que se instancia al momento de la inicialización del `Server`. Con el `TcpStream` obtenido del `TcpListener` para iterar sobre las requests entrantes se instancia un `RequestHandler`, y se llamará a su método `handle()` dentro de cada thread.

Thread Pool y Worker

Esta entidad representa un conjunto de threads activos y a la espera de una tarea para ejecutar. La cantidad máxima de threads a manejar viene dada por parámetro al momento de su instanciación, para prevenir problemas de recursos del sistema debido a un exceso de solicitudes. En nuestro caso, usaremos 1000 threads para soportar el manejo de hasta 1000 requests simultáneas. Se expone el método `execute()` que recibe un bloque de código a ejecutar y, haciendo uso de la abstracción intermedia `Worker`, ordena la ejecución de ese bloque en paralelo enviándolo al `Worker` a través de un `channel`.

`Worker` es la entidad responsable de recibir código del `ThreadPool` y enviarlo a un `thread`. La razón de ser de esta entidad es encapsular el mecanismo implementado para que un thread no reciba el código a ejecutar ni bien es creado, sino que esté activo esperando a que le llegue un bloque a ejecutar. En lugar de contener un vector de `JoinHandle<()>`, en `ThreadPool` habrá un vector de `Worker` donde cada uno de ellos guardará una única instancia de `JoinHandle<()>`.

En `ThreadPool` habrá un `sender` de un `channel`, y cada instancia de `Worker` tendrá un `receiver` de ese `channel`. Contamos con una estructura llamada `Job` que contendrá los bloques de código que envía el `ThreadPool`. En su respectivo thread, el `Worker` tendrá un ciclo sobre el `receiver` del `channel`, y ejecutará los bloques de código de cada `Job` que reciba. Esto parece ir en contra del modelo *MPSC (Multiple Producer Single Consumer)*, pero para poder mantener múltiples referencias al `receiver` del `channel` y evitar problemas de concurrencia usaremos `Arc<Mutex<T>>` para, mediante la exclusión mutua, asegurar que solo un `Worker` reciba un `Job` del `receiver` por vez.

A su vez, `Job` estará dentro de un `Message`, que es un enum que puede contener un mensaje `NewJob` con un `Job` a ejecutar, o un mensaje `Terminate` que utilizaremos para finalizar todos los workers al momento de ejecutar `drop()` en el `ThreadPool`. De esta manera, en el ciclo dentro del thread de cada `Worker` será necesario distinguir entre estos dos tipos de mensaje.

Request Handler

Se encarga de enrutar el request entrante al endpoint correspondiente.

Primero identifica el método HTTP. Como nuestro tracker expone dos endpoints que son ambos `GET` , retorna un error si el método es otro.

Luego identifica el endpoint en sí (`/announce` o `/stats`) y llama al método respectivo para manejar el request.

Una vez que se recibe la respuesta del respectivo endpoint, se bencodea en caso del **announce** o se crea un JSON en caso de **stats** y se envía por el stream.

HTTP Parser

Tiene la responsabilidad de *parsear* lo leído por el `RequestHandler`, y devolver un struct `Http` que contiene el método HTTP recibido, el endpoint al que se le envió un request y un mapa con los query param recibidos. Si la forma del String recibido no es la esperada, o el método HTTP utilizado no está entre los soportados (en este caso, cualquier método distinto de GET), devuelve un HttpError.

Announce Response

El `AnnouceResponse` es el encargado de estructurar la respuesta del tracker según los **Query Params** pasados por el `RequestHandler`.

Su funcionamiento se divide en 3 principales pasos:

- Lo primero que hace es crear un `AnnounceRequest` con los *QueryParams,* comprobando que se trata de una request válida.
- Luego, crea, a través del `AnnounceRequest`, un `Peer` representando al emisor de la request, con su **IP**, **Peer ID, Puerto** y **Estado Actual**.
- Por último, le notifica al `TrackerStatus` sobre la llegada de un nuevo peer, pasandole el **peer**, el **info_hash** del torrent al cual quiere hacer announce y el ***numwant*** que es el número de peers que le gustaría recibir en la respuesta. El `TrackerStatus` le devuelve la lista de peers pedida y ademas la cantidad de **seeders** y **lecheers** que tiene el torrent actualmente.

Luego de realizar estos 3 pasos correctamente se crea la *Response* con lo devuelto por el *Status* y se lo devuelve al `RequestHandler`.

En caso de error, se devuelve un `AnnounceResponse` pero con el atributo **failure_reason** indicando el error.

Announce Request

El `AnnounceRequest` es el encargado de verificar si una request para el announce es válida. Para saber esto primero revisa que dentro de los **Query Params** estén presentes todos los parámetros obligatorios, los cuales son: **info_hash**, **peer_id**, **port**, **uploaded**, **downlaoded** y **left**. Si alguno de estos no está presente la request no es válida.

Luego para todos los parámetetros, tanto obligatorios como opcionales, verifica que su formato sea correcto. Por ejemplo que **info_hash** sean 20 bytes con **url encoding**.

Una vez verificádos todos los parametros, se forma la request y se la devuelve al `AnnounceResponse`.

Stats Response

Arma la respuesta necesaria para el endpoint `/stats`. Para esto utiliza el estado histórico guardado en el `StatsUpdater`.

Recibe un parámetro `since`, que determina la cantidad de tiempo en horas de estadisticas pedidas desde el momento actual hacia atrás. Teniendo en cuenta dicho parámetro, corta el vector del estado histórico y construye un JSON para, finalmente, devolverlo como respuesta del request.

Stats Updater

Esta entidad es la encargada de mantener un **Historial de Estadísticas** del tracker. En su creación es necesario pasarle por parametro un `Duration` para indicarle cada cuanto tiempo quiere hacerse el guardado.

Su funcionamiento consiste en:

- Primero pedirle al `TrackerStatus` que elimine a todos sus *peers inactivos*.
- Luego, fijarse si se llegó al máximo de instancias de estadísticas que se permite guardar. El límite esta para que no pueda crecer indefinidamente.

  Si se llegó al límite, lo que se hace es eliminar la estadística más antigua y remplazarla por la mas reciente, que la obtiene pidiendosela al `TrackerStatus`.
- Por último, el thread se pone a ‘dormir’ hasta que se cumpla su **Timeout** y vuelva a realizar los pasos de arriba.

El `StatsUpdater` es *thread-safe* y *bloqueante*, por lo tanto, si se quiere acceder a una copia del **Historial**, cuando llega una request para el endpoint ***/*stats**, por ejemplo, se deberá esperar a que el **Updater** esté durmiendo.

Tracker Status

El `Tracker Status` es la estructura que se encarga de mantener el estado actual de todos los `Peers` registrados en el *tracker.* Podría pensarse como una “base de datos” en memoria, en donde se almacena toda la información del sistema. Internamente, el `Tracker Status` mantiene un diccionario de `Swarms`, que representan al conjunto de `Peers` relacionados a un *torrent* específico.

La decisión de que los datos del `Tracker Status` se guarden únicamente en memoria y no se persistan en el disco se tomó en base a que la información almacenada es de carácter efímero: solo tiene valor por alrededor de una hora. Pasado ese tiempo, los *peers* que no se vuelvan a anunciar serán eliminados del sistema, ya que es poco probable que sigan conectados y que sigan teniendo valor para una nueva persona que intenta conectarse. Por otro lado, en el caso de que el servidor se caiga y se vuelva a iniciar, la información se volverá a reconstruir a medida que se conecten nuevamente los *peers*. De esta manera nos ahorramos los problemas relacionados a mantener una base de datos real, y el acceso a memoria es mucho más rápido.

Esta estructura brinda soporte para los dos *endpoints* del servidor:

- Cuando un nuevo peer se conecta con el servidor mediante el `/announce`, el `Announce Response` llama al método `#incoming_peer` del `Tracker Status`. Allí se busca el *swarm* correspondiente mediante el *info_hash* del *torrent,* luego se anuncia al nuevo *peer*, y se devuelve una lista de *peers* activos del *swarm*.
- Para poder manejar las *requests* que llegan al `/stats`, el `Tracker Status` provee los métodos `#get_global_statistics`, que se encarga de obtener la suma total de *seeders* y *leechers* en el *tracker,* y `#remove_inactive_peers`, que recorre todos los *swarms* y elimina aquellos *peers* cuya última actualización haya ocurrido hace más de una hora. Este último método es llamado por el `Stats Updater` una vez por minuto, manteniendo la estructura actualizada.

El `Tracker Status` es *thread-safe*, ya que debe ser accedido desde cada uno de los hilos que manejan las *requests*. Esto se logra con un `Mutex` sobre el diccionario de *swarms*.

Swarm

Un `Swarm` es una estructura que representa al conjunto de *peers* de un *torrent*. Se encarga de mantener un diccionario de `Peers` accedidos por su *peer id*, y las estadísticas de cuántos de ellos son *seeders* y cuántos son *leechers.*

Posee métodos para:

- Manejar un peer entrante mediante el mensaje `#announce`, actualizando el estado de un *peer* ya registrado previamente, o registrando uno nuevo.
- Para obtener una lista aleatoria de *peers* activos en el `Swarm` mediante el mensaje `#get_active_peers`.
- Eliminar los *peers* inactivos, es decir, los *peers* cuyo último *announce* se haya realizado antes del *timeout* elegido.

Peer

Estructura que representa un *peer* conectado al *tracker*. Almacena su *IP*, su *peer id*, su puerto de conexión, y opcionalmente una *key* usada para identificar al peer en caso de que cambie su *IP*. También mantiene un **Peer Status** que contiene los datos de su último *announce* con el *tracker* (cantidad de bytes subidos, cantidad de bytes descargados, cantidad de bytes que le faltan descargar para completar la descarga del *torrent,* opcionalmente un *event*, y el horario de su última conexión).

Un peer puede crearse en base a una `AnnounceRequest`, y tiene la capacidad de convertirse a *bencoding* mediante el trait `ToBencode`.

UI

Se desarrolló un frontend con HTML y Javascript para mostrar las estadísticas del tracker en tiempo real. Realiza un polling de los datos de estadísticas, mediante requests periódicos (cada medio segundo, por ejemplo) al endpoint `/stats` del tracker, pasando por query param la cantidad de horas de estadisticas que se quieren mostrar, y una vez recibidas se actualiza el gráfico en pantalla.

La cantidad de tiempo de estadísticas a mostrar en el gráfico puede ser cambiada por el usuario mediante una lista desplegable con opciones de tiempo predefinidas. A su vez, mediante otra lista desplegable, es posible cambiar la granularidad del tiempo, es decir, si las estadísticas están divididas por minuto o por hora.