Trabajo Práctico N° 2 - Administración de Sistemas

Integrantes del equipo:

- Lisandro Garcia Seminara
- Bruno “Enzo” Malvasio
- Santino Garcia Corsaro

Profesor: Ignacio Garcia

Fecha de entrega: 2/6

Índice

Investigación sobre BitTorrent

Sistema Peer-to-Peer (P2P)

P2P es un término usado en muchos contextos, a veces con significados levemente diferentes. Es posible encontrar muchas definiciones alternativas, que no son exactamente equivalentes.

Un sistema se considera P2P si los elementos que lo forman comparten sus recursos para proveer el servicio que el sistema fue diseñado para proveer. Los elementos en el sistema:

- Proveen servicios a otros elementos
- Piden servicios de otros elementos

Todos los elementos en el sistema deberían cumplir con los criterios anteriormente mencionados para ser considerados P2P. Sin embargo, en la práctica, un sistema puede tener algunas excepciones y seguir siendo considerado P2P. Por ejemplo, un sistema puede seguir siendo considerado P2P incluso si tiene un servidor de enrolamiento centralizado (entre peers). Por otro lado, algunos sistemas dividen los endpoints entre peers y clientes. Los peers piden y proveen servicios, mientras que los clientes solo piden. Un sistema donde la gran mayoría de endpoints se comportan como clientes podría NO ser considerado P2P estrictamente.

Aunque algunas definiciones de P2P no lo dicen explícitamente, muchas asumen implícitamente que un sistema, para ser P2P, necesita que sus nodos estén envueltos en transacciones relacionadas con los servicios que no benefician directamente a los nodos.
Otros autores agregan que los elementos del sistema P2P, llamados peers, deberían poder comunicarse directamente entre ellos sin pasar por un intermediario. Algunos agregan que el sistema debería ser autoorganizado y tener control descentralizado.

Un servicio complejo puede estar construido por muchos servicios individuales. Algunos de estos pueden ser P2P y otros pueden ser cliente-servidor. Por ejemplo, un cliente para compartir archivos puede incluir un cliente P2P para realizar el compartimiento y un buscador web para acceder a información adicional en un servidor web centralizado. Adicionalmente, hay arquitecturas donde un sistema cliente-servidor puede servir como un backup para un servicio que normalmente proviene de un sistema P2P, o viceversa.

Proveer un servicio normalmente incluye procesar o guardar datos. De acuerdo a la definición mencionada, en un sistema P2P, los peers comparten su procesamiento y su capacidad de almacenamiento (sus recursos de hardware/software) para que el sistema pueda proveer un servicio. Cuando un peer necesita conseguir un archivo específico, el peer descubre primeramente que peer o peers tienen ese archivo y luego lo obtendrá a partir de ellos.

Funciones de un sistema P2P

Los sistemas P2P incluyen bastantes funciones. Las descritas están ancladas a un sistema P2P sea o no BitTorrent. Son proveídas de forma centralizada en algunos sistemas P2P, por ejemplo, a través de un servidor central de enrolamiento y un servidor central de descubrimiento de peers. Las mismas controlan cómo se conectan los peers.

- Función de enrolamiento: Los nodos que se unen a un sistema P2P necesitan obtener credenciales válidas para unirse satisfactoriamente. La función de enrolamiento maneja la autenticación y autorización de los nodos. Esta función no necesariamente se ve implementada en una red BitTorrent a menos que sea privada.
- Descubrimiento de peers: Para unirse a un sistema P2P (para convertirse en un peer) un nodo necesita establecer una conexión con uno o más peers que ya son parte del sistema. Esta función permite que los nodos descubren peers en el sistema para conectarse a ellos. El descubrimiento de peers en BitTorrent se realiza ya sea de forma centralizada, con un tracker, o de forma descentralizada con el uso de las distributed hash tables (DHT) como se verá a continuación.

Las siguientes funciones son implementadas por algunos tipos de servicios P2P, no todos. Dependiendo del tipo de servicio que se provee algunas no serán necesarias.

- Función de indexación de datos: Se encarga de indexar los datos guardados en el sistema.
- Función de guardado de datos: Se encarga de almacenar y recuperar datos del sistema. En BitTorrent los peers almacenan partes del los datos que se comparten.
- Función de cómputo: Se encarga del cómputo hecho por el sistema. Este cómputo puede estar relacionado, entre otras cosas, con procesamiento de datos o procesamiento de medios en tiempo real. BitTorrent no realiza tareas de cómputo distribuido, solo transporta datos.
- Función de transporte de mensajes: Se encarga del intercambio de mensajes entre peers. Dependiendo en cómo esta función se implemente, los peers pueden intercambiar mensajes de protocolo a través de un servidor central o directamente entre ellos.

Consideraciones de seguridad de un sistema P2P

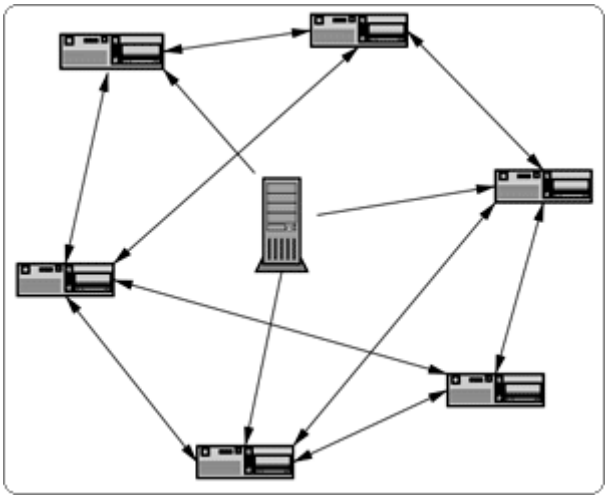
- Primeramente se tiene que tener en cuenta hasta qué punto los nodos en el sistema pueden ser confiables. Si todos lo son, por ejemplo, siendo que están bajo el completo control el operador del sistema y nunca nunca actuarán de una forma maliciosa, un sistema P2P puede conseguir un alto nivel de seguridad. Si no fuesen de completa confianza y puede esperarse que se comporten en formas maliciosas, se vuelve particularmente difícil proveer un nivel aceptable de seguridad en un sistema P2P a diferencia de uno que no lo es, ya que con la propiedad de los datos distributiva y una falta de control centralizado y conocimiento global. El nivel de seguridad depende en la proporción de los nodos que se comporten maliciosamente.

- Debido a cómo funcionan los sistemas P2P, cuando los peers se conectan entre sí para compartir datos revelan su dirección IP. Esto abre oportunidades para que actores maliciosos recolecten direcciones IP sin necesidad de escanear redes activamente ya que un peer está conectado típicamente a múltiples peers. Este tipo de ataque es mucho más eficiente que realizar scans cuando las direcciones que deben ser escaneadas son largas (IPv6). Adicionalmente, existen correlaciones entre una aplicación particular y un sistema operativo en particular (un cliente de BitTorrent que está sumamente presente en Windows/Linux). De esta forma, un atacante puede recolectar IPs que son susceptibles a lanzar ataques que explotan vulnerabilidades específicas de un sistema operativo.

- Los atacantes pueden lanzar ataques de denegación de servicio creando lo conocido como “churn”: la entrada y salida rápida de nodos en una red P2P. Haciendo esto, una cantidad de

atacantes pueden crear mucho trafico de mantenimiento, haciendo que la estructura de ruteo de la red sea inestable. Esto es porque cada vez que un nodo entra y sale de una red deben actualizarse las estructuras de enrutamiento para que la red siga funcionando. Esto, en consecuencia, afecta directamente a la disponibilidad de la red. Una forma fácil de prevenir este ataque es limitando la cantidad de *churn* por nodo.

BitTorrent



BitTorrent es un protocolo de transferencia de datos orientado en una red de procedimientos P2P, ya que estos se conectan entre sí para enviar y recibir partes de un archivo.
Todo esto se realiza por medio de una entidad conocida como “tracker”, el cual coordina las transferencias de todos los usuarios. Sin embargo no tiene ningún acceso al contenido de los paquetes distribuidos, por lo tanto una gran cantidad de usuarios pueden ser soportados por un ancho de banda bastante ligero, facilitando las descargas y aumentando la velocidad de descarga mediante un protocolo más eficiente.

Historia

BitTorrent fue creado por Bram Cohen en 2001. La primera versión del cliente no tenía buscador ni intercambio directo entre pares. Para compartir archivos, se creaba un archivo .torrent que se subía a una página, y los usuarios se conectaban a un rastreador que coordinaba las descargas.

En 2005, se introdujo el seguimiento distribuido con tablas **hash distribuidas** (**DHT**), permitiendo que los clientes se conectaran sin depender de un archivo .torrent.

En 2006, se añadió el PEX (peer exchange), mejorando aún más la conectividad: Los clientes ahora también se comunican directamente entre ellos y cuando un cliente se conecta a otro, le pregunta por una lista de otros pares que conoce. De esta forma, los usuarios ayudan a otros a encontrar más usuarios, sin depender tanto del rastreador o de DHT

La versión BitTorrent v2 reemplaza el antiguo sistema de seguridad (SHA-1) por SHA-256, más seguro. Además, usa árboles hash para verificar archivos más rápido y con mayor precisión. También

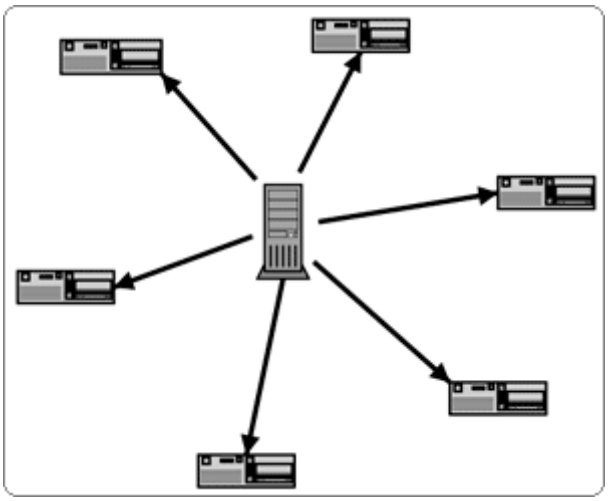
permite compartir archivos duplicados entre diferentes torrents, lo que mejora la eficiencia. Para mantener la compatibilidad, v2 funciona en modo híbrido con v1, incluso en los enlaces magnéticos.

¿Por qué se creó BitTorrent?

Bram Cohen desarrolló BitTorrent en 2001 para resolver un problema importante de internet en ese momento: Distribuir archivos grandes (como vídeos o software) a muchas personas sin saturar los servidores.

Antes de BitTorrent, si muchas personas querían descargar el mismo archivo desde un servidor, eso generaba un gran consumo de ancho de banda para el servidor, lo que lo volvía lento o caro de mantener.

¿Por qué usar bittorrent?



La principal ventaja de BitTorrent es su alta velocidad de descarga gracias a la gran red de usuarios activos que almacenan y seedean paquetes de descarga específicos, utilizando más eficazmente el ancho de banda para descargar archivo, separándolo en partes para facilitar su descarga. Por esto el protocolo se hizo popular para descargar archivos pesados convenientemente.
Además su mantenimiento es mucho más sencillo al estar todo el mecanismo interno del protocolo controlado por el Tracker, por lo tanto se puede prevenir completamente una saturación de los servidores al no depender de un solo servidor, ya que la responsabilidad de almacenar datos se transfiere a todos los usuarios de la red. Las descargas no tienen porqué ser reiniciadas desde cero al ser interrumpidas, reanudando por donde se dejó.

- ☑ Más rápido (con muchos usuarios)
- ☑ Mismo ancho de banda, mejor usado
- ☑ No satura a los servidores
- ☑ No depende de un sólo servidor ya que todos comparten el archivo
- ☑ Simple reanudación (Se puede reanudar desde donde se dejó, sin necesidad de empezar de cero)

¿Por qué no usar bittorrent?

A pesar de las fortalezas del protocolo BitTorrent para la rápida transferencia de datos entre usuarios en una red P2P, existen algunos riesgos y consideraciones a tener en cuenta al trabajar con el mismo.
El protocolo puede resultar inseguro y susceptible a vulnerabilidades debido a su método de encriptación, esté siendo todavía SHA-1 hash, obsoleto para los estándares actuales, y con poco soporte y material online se puede hacer dificultoso investigar y solucionar problemas con el protocolo, demostrado por ser más vulnerable a viruses y otros ataques cibernéticos.
También se suele poner en duda la legalidad del protocolo BitTorrent, ya que por su naturaleza libre y permisiva encontró usos en áreas ilícitas como puede ser la distribución de material pirateado, normalmente de pago. Esto además atrae la atención de diferentes compañías de entretenimiento que buscan tirar abajo servicios que utilicen esta tecnología, o la preocupación por parte de los usuarios por repercusiones legales al asociarse con dichas actividades.
Además, hay que tener en cuenta que los usuarios de la red de Torrent son capaces de ver todos a todos los demás, es decir sus IPs, que accedieron a un cierto archivo torrent, entonces esto es utilizado por compañías de comunicaciones, entretenimiento y gobiernos para detectar casos de pirateria, junto con la ayuda del proveedor de internet o ISP.
El usuario aparte puede encontrar más conveniente la descarga directa en el caso que haya pocos usuarios en la red, ya que la descarga resultaría muy lenta por BitTorrent, al ser un servicio que opera más rápidamente de acuerdo con el número de usuarios que puedan almacenar paquetes/bloques para la descarga.

- ☑ Riesgos de seguridad: Al conectarte con muchos usuarios, tu IP queda expuesta y podés ser blanco de rastreos/ataques
- ☑ Contenido ilegal/pirata (I am not from the US!)
- ☑ Archivos infectados
- ☑ Si hay pocos usuarios, puede ser más conveniente la descarga directa, ya que sería muy lento por BitTorrent

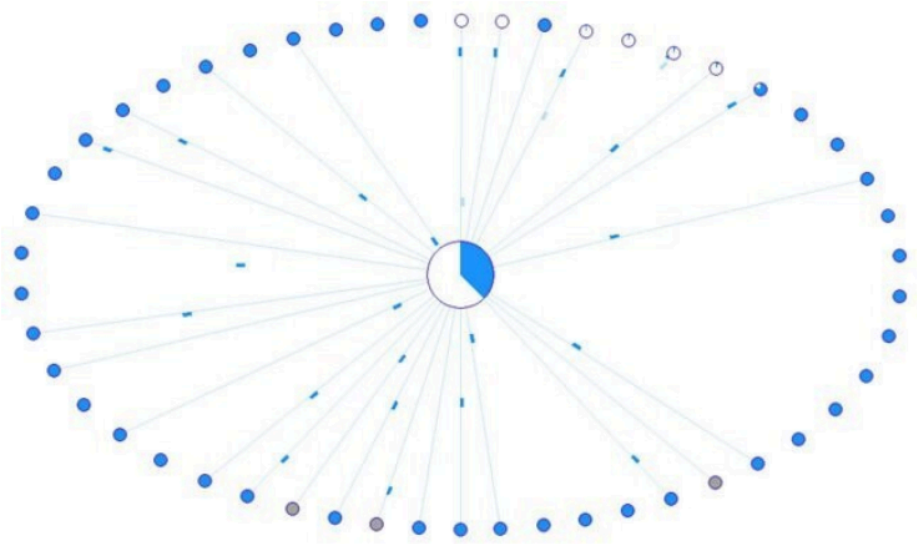
Entidades involucradas

La distribución de un archivo a través del protocolo BitTorrent hace uso de las siguientes entidades:

- Un servidor web
- Un archivo metainfo estático
- Un tracker BitTorrent
- Un descargador original
- Los buscadores web de los usuarios finales
- Los descargadores de los usuarios finales

Idealmente hay muchos usuarios finales para descargar un archivo.

Funcionamiento



BitTorrent divide los contenidos de un archivo en paquetes más pequeños y los envía a otros usuarios, el host solo tiene que descargar las partes desde los diferentes usuarios para obtener el archivo final. Estas particiones del archivo se dividen a su vez en bloques (blocks) para hacer más fácil su distribución entre los clientes y aquel que requiere los datos. Por su parte los usuarios constituyen las computadoras que forman parte de la network, los activos se conocen como “active peers” y pueden tener uno de dos roles:

- **Seeders:** Son aquellos usuarios que tienen los contenidos completos de un archivo en su sistema, el cual están compartiendo activamente, o simplemente partes pequeñas de otros archivos, al igual que los Leechers. Cuanto más Seeders haya más ancho de banda tendrán las descargas ya que recae menos peso, es decir responsabilidad, a cada uno de los cliente de este rol, y si solo existiera un seeder, este sería un modelo clásico de server-usuario.

- **Leecher:** Refieren a los clientes que descargan los bloques de datos de los Seeders, sin embargo, estos también son capaces de almacenar pequeñas cantidades de información para proporcionar a la red.

¿Qué debe hacer un host para comenzar a servir datos?

1. Primeramente se necesita un tracker corriendo, ya sea iniciando uno o ya teniendo uno corriendo previamente.
2. Luego, se debe iniciar un servidor web ordinario, como Apache.
3. Se asocia la extensión .torrent con el mimetype *application/x-bittorrent* en el servidor web. Un ejemplo hecho en un servidor Apache sería el siguiente:

   AddType application/x-bittorrent .torrent

   Un mimetype (Multipurpose Internet Mail Extensions) es un estándar que define el tipo de contenido de los archivos transmitidos por la web. Cada tipo de archivo tiene idealmente un MIME type que le informa al navegador/servidor que tipo de archivo se está manejando. Haciendo esta asociación el navegador puede manejar el archivo .torrent de forma acorde.

4. Se genera un archivo metainfo (.torrent) usando la totalidad del archivo que se servirá y la URL del tracker en cuestión. Un archivo metainfo contiene únicamente metainformación, ya que contiene la información del archivo que se servirá. En la práctica, usando un programa como *mktorrent* se vería de la siguiente manera:
   mktorrent -a http://tracker.example.com/announce -o archivo.torrent perro.jpg
   Este proceso resultará en la creación del archivo *archivo.torrent* conteniendo la metainformación del archivo a servir.
5. Se pone el archivo metainfo en el servidor web.
6. Se realiza un enlace al archivo metainfo (.torrent) en otra página web.
7. Se inicia un descargador que ya tenga el archivo completo, que vendría siendo el origen.

¿Qué debe hacer un usuario para descargar datos?

1. Instalar un cliente BitTorrent
2. Investigar la web en busca de un torrent
3. Clickear en un link que lleve a un archivo *.torrent*
4. Seleccionar donde guardar el archivo de forma local, o seleccionar una descarga parcial para continuarla
5. Esperar a que la descarga termine
6. Cerrar el cliente BitTorrent o seguir subiendo el archivo

Bencoding

Este es el encoding usado por BitTorrent para los archivos metainfo (*.torrent)*. Soporta cuatro tipos de valores:

- Strings de bytes
- Integers
- Listas
- Diccionarios

Bencode usa caracteres ASCII como delimitadores y dígitos para encodear estructuras de datos en un formato simple y compacto. El algoritmo que usa para encodear es el siguiente:

- **Integers:** Son encodeados como i<base10 integer>e.
  - El integer es encodeado en base 10 y puede ser negativo (indicado por un “-” al principio)
  - Los ceros a la izquierda no son permitidos a menos que el integer sea cero.
  - Ejemplos:
    - Cero es encodeado como i0e.
    - El número 42 es encodeado como i42e.
    - El número 42 negativo es encodeado como i-42e.

- **Strings de bytes:** Son encodeadas como <largo>:<contenidos>.
  - El largo es el número de bytes en la string, encodeado en base 10.
  - Un símbolo ":" separa el largo de los contenidos.
  - Los contenidos son el número exacto de bytes especificados por el largo.
  - Ejemplos:
    - Una string vacía se encodea como 0:.
    - La string "bencode" se encodea como 7:bencode.
- **Listas:** Son encodeadas como l<elementos>e.
  - Comienza con una "l" y termina con una "e".
  - Los elementos son valores bencodeados concatenados sin delimitadores.
  - Ejemplos:
    - Una lista vacía es encodeada como le.
    - Una lista que contiene la string "bencode" y el integer -20 se encodea como l7:bencodei-20ee.
- **Diccionarios:** Son encodeados como d<pares>e.
  - Comienza con una "d" y termina con una "e".
  - Contienen pares de *llave-valor*.
  - Las llaves son strings de bytes y deben aparecer en orden lexicográfico.
  - Cada llave es inmediatamente seguida de su valor, que puede ser cualquier tipo bencodeado.
  - Ejemplos:
    - Un diccionario vacío se encodea como de.
    - Un diccionario con las llaves "wiki":"bencode" y "meaning":42 se encodea como d7:meaningi42e4:wiki7:bencodee.

No hay restricciones en los tipos de valores guardados en las listas y diccionarios. Pueden contener otras listas y diccionarios, permitiendo estructuras de datos complejas.

Archivos Metainfo

Como ya sabemos, BitTorrent usa un archivo *.torrent* llamado metainfo file. Este se vería de esta forma siguiendo el ejemplo dado anteriormente:

d8:announce35:http://tracker.example.com/announce10:created by13:mktorrent 1.113:creation datei1748799493e4:infod6:lengthi65703e4:name9:perro.jpg12:piece lengthi262144e6:pieces20:D%'ÓÞ$_h'éf/Ød¿ee

Todos los datos en un archivo metainfo están bencodeados. El contenido del archivo metainfo (como se comentó antes siendo el que termina en *.torrent*) es un diccionario bencodeado que contiene las llaves especificadas a continuación. Todos los valores de las cadenas de caracteres están encodeados en UTF-8.

- **announce**: Incluye la URL del tracker. Este representa un servidor HTTP que organiza y monitorea a todos los usuarios y que pieza o bloque de datos tiene cada uno.
- **announce-list** (opcional): Es una extensión de la especificación de BitTorrent oficial (actualmente aceptada), siendo el BitTorrent enhancement proposal número 12. Es una llave que refiere a una lista de listas de URLs, y contendrá una lista de tiers de anuncios. Si el cliente es compatible con la especificación multitracker, y si la llave está presente, el cliente podrá ignorar la llave announce y solo usar las URLs presentes en announce-list.
- **creation-date** (opcional): Es el tiempo de creación del torrent en el formato standard UNIX epoch (integer, segundos desde enero 1 1970 00:00:00 UTC).
- **comment** (opcional): Es una forma libre de comentarios textuales del autor (string).
- **created by** (opcional): Es el nombre y la versión del programa usado para crear el archivo metainfo (string).
- **encoding** (opcional): Es la string que especifica el formato de encoding usado para generar las piezas que son parte del diccionario info en el *.torrent* (string).
- **info**: Es un diccionario que describe el o los archivos del torrent. Hay dos formas posibles:
    - Una para el caso de un torrent de un solo archivo, que no tiene estructura de directorios.
    - Otra para el caso del torrent de múltiples archivos.

Diccionario info

A continuación se describen las llaves que se encuentran en el diccionario info de los archivos metainfo:

- **piece length**: Mapea el número de bytes en cada pieza que el archivo se divide. Por motivos de transferencia, los archivos son divididos en piezas de tamaño definido que son del mismo largo excepto, quizás, por la última que podría ser truncada. Casi siempre es una potencia de dos, más comúnmente 218 = 256 k (BitTorrent antes de la versión 3.2 usaba 220 = 1M como default).
- **pieces**: Cadena que consiste en la concatenación de los valores de hash SHA1 de 20 bytes, uno por cada pieza. Es una cadena de bytes.
- **private** (opcional): Es un integer. Si se setea a 1, el cliente debe publicar su presencia para conseguir otros peers únicamente a través de los trackers explícitamente descritos en el archivo metainfo. Si este campo está seteado a 0 o no está presente, el cliente debe tener peers por otros medios (como PEX, DHT). Private se interpreta como “no hay fuente de peers externa”. Hay mucho debate en torno a los trackers privados (se hablará luego de eso). Existe un BEP siendo el número 27 (actualmente aceptado).

En el caso que esté en modo single-file (un solo archivo):

- **name**: Mapea a una string encodeada en UTF-8 que es el nombre sugerido para guardar el archivo. Es puramente en forma de consejo.
- **length**: Es el largo del archivo en bytes (integer).

En el caso que esté en modo multiple-file (muchos archivos):

- **name**: Es el nombre del directorio en el que se guardarán los archivos. Es puramente en forma de consejo. (string).
- **files**: Una lista de diccionarios, uno por cada archivo. Cada diccionario en esta lista contiene las siguientes llaves:

- **length**: El largo del archivo en bytes. (integer).
- **path**: Una lista que contiene una o más strings que juntas representan el path y el filename. Cada elemento en la lista representa un nombre de directorio o (siendo el caso del último elemento) el filename. Por ejemplo, el archivo "dir1/dir2/file.txt" consistirá de tres strings. Esto es encodeado como una lista bencodeada de strings: l4:dir14:dir28:file.txte

BitTorrent soporta magnet links para efectuar las descargas remotas desde los hosts de los paquetes de datos. Estos consisten en un link de texto el cual incluye toda la información necesaria y sin necesidad de incorporar el archivo ".torrent", independientemente del funcionamiento del Tracker del protocolo. Esto asemeja su comportamiento a un verdadero servicio de usuario-a-usuario, ya que se prescindirá del Tracker para establecer la conexión entre tu máquina y los archivos que quieres descargar.
https://bittorrent.org/beps/bep_0009.html

Trackers

Como se mencionó brevemente al comienzo de esta sección, un tracker es un servidor que asiste en la comunicación entre peers. El tracker se encarga de mantener registro de donde residen las copias de los archivos en las computadoras de los peers, cuales están disponibles en el momento de la petición del cliente, y ayuda a coordinar transmisión y reensamblamiento del archivo copiado de forma eficiente.

Los clientes que ya comenzaron a descargar un archivo se comunican con el tracker periódicamente para negociar transferencia de archivos a más velocidad con nuevos peers, y proveer estadísticas de performance de la red. Sin embargo, después de la descarga P2P inicial del archivo, la comunicación entre peers puede continuar sin la conexión de un tracker.

Actualmente, los clientes BitTorrent implementan una tabla distribuida de hashes (DHT) y el protocolo de peer exchange (PEX) para descubrir peers sin la necesidad de trackers; sin embargo, estos se siguen incluyendo con los torrents para mejorar la velocidad del descubrimiento de peers (que afecta directamente en la velocidad). Este tema se expandirá más adelante en el documento.

Protocolo HTTP/HTTPS

El tracker es un servicio HTTP/HTTPS que responde a HTTP GET request. Estas requests incluyen métricas de clientes que ayudan al tracker mantener estadísticas generales del torrent.
La respuesta incluye una lista de peers que ayuda al cliente a participar en el torrent.
La URL base consiste del *announce URL* como esta definida en el archivo metainfo (*.torrent*). Los parámetros son luego agregados a esta URL, usando métodos CGI estándar (por ejemplo, un "?" después de la URL del *announce*, seguida de secuencias de "param=value" separadas por "&"). Un ejemplo de esto seria: https://tracker.example.com/announce?param1=value1&param2=value2

Todos los datos binarios en la URL (particularmente *info_hash* y *peer_id*) deben ser correctamente escapados. Esto es que cualquier byt que no entre en “0-9”, “a-z”, “A-Z”, “.”, “-”, “_” y “~” deben ser encodeados usando el formato “%nn”, donde nn es el valor hexadecimal del byte. Un ejemplo de esto sería:

- Para un hash de 20 bytes de:
  \x12\x34\x56\x78\x9a\xbc\xde\xf1\x23\x45\x67\x89\xab\xcd\xef\x12\x34\x56\x78\x9a
- La forma correcta encodeada es:
  %124Vx%9A%BC%DE%F1%23Eg%89%AB%CD%EF%124Vx%9A

Parámetros Request del tracker

El método GET realizado del cliente al tracker usa los siguientes parámetros:

- **info_hash**: URL codificado en un SHA1 de 20 bytes de acuerdo al valor de la llave *info* en el archivo metainfo (*.torrent*). El valor será un diccionario bencodeado, a partir de la definición de la llave *info*.
- **peer_id**: URL codificado en un string de 20 bytes usado como un ID único asociado al cliente al comienzo de la descarga. No hay reglas específicas para generar peer_ids, pueden ser valores binarios, pero tienen que ser únicos por cada máquina local, por lo tanto deberían incluir elementos como ID de proceso u hora de comienzo de la descarga como referencia.
- **port**: El número del puerto el cual el cliente está escuchando actualmente. Los puertos reservados exclusivamente a BitTorrent suelen estar en un rango entre 6881-6889. Los clientes pueden rendirse si no pueden establecer un puerto dentro de este rango.
- **uploaded**: La cantidad total de paquetes subidos (desde que el cliente envió el evento *started* al tracker) en base diez ASCII. El consenso es que esto debería resultar en el número de bytes totales subidos.
- **downloaded**: La cantidad total de paquetes descargados (desde que el cliente envió el evento *started* al tracker) en base diez ASCII. Al igual que el anterior debería ser el número de bytes totales descargados.
- **left**: La cantidad de bytes que el cliente aún tiene que descargar para finalizar la descarga en base diez ASCII. Representando los datos necesarios para que la descarga se considere realizada en un 100%.
- **compact**: Estableciendo este valor a 1 indica que el cliente acepta una respuesta compacta. La lista de peers es reemplazada por una string de peers con 6 bytes por peer. Los primeros cuatro bytes son el host y los últimos dos son el puerto. Debe ser considerado que algunos trackers solo soportan respuestas compactas para salvar ancho de banda y quizás rechazan respuestas sin “compact=1” o simplemente envían una respuesta completa a menos que la request contenga “compact=0” en cuyo caso simplemente la rechazan.
- **no_peer_id**: Indica si el tracker puede omitir el campo de *peer_id* dentro del diccionario de peers. Esta opción es omitida si la opción de compact es habilitada.
- **event**: Si está especificado, puede ser una de tres opciones, *started*, *completed*, *stopped* (o vacío el cual es lo mismo que no lo especifiquen). Si no se especifica, esta request es realizada en intervalos regulares.
  - **started**: La primera request al tracker debe incluir este valor en el campo *event*.
  - **stopped**: Debe ser enviada al Tracker si el cliente se está apagando de forma exitosa.

    - **completed**: Debe ser enviada al Tracker cuando se completa la descarga. Sin embargo, no debe ser enviada si la descarga ya estaba 100% completada al momento que el cliente inicio. Esto sería, presumiblemente, para que el tracker incremente la métrica de descargas completadas basada únicamente en este *event*.

  ip: Carácter opcional. La dirección ip real del sistema del cliente, en formato punteado o rfc3513 definido en una dirección hexadecimal de IPv6. Opcional ya que en la práctica se puede extraer este campo desde la dirección donde se envió la request HTTP original, y solo se especifica adicionalmente en el caso que esta dirección no sea la real del cliente, como puede ser en el caso donde el usuario utilice un cliente proxy. Para evitar devolver una dirección de terminal arbitraria la cual no es ruteable. Por lo tanto debe ser especificada la IP del cliente (externa y ruteable) para ser enviada a peers externos.
- **numwant**: Carácter opcional. Número de peers que el cliente le gustaría recibir del tracker, puede ser 0, pero suele ser ajustado a 50 por defecto.
- **key**: Carácter opcional. Consiste en una identificación adicional la cual no es compartida con ningún otro peer, con el fin de validar la identidad del cliente en el caso que su dirección IP cambie.
- **trackerid**: Carácter opcional. Si un *announce* previo contenía un tracker id, debe estar almacenado aquí.

Respuestas del tracker

El Tracker responde con un documento en “text/plain” el cual consiste en un diccionario bencodeado compuesto por las siguientes llaves:

- **failure reason**: En el caso que esté presente, quiere decir que no hay ninguna otra llave presente. Este siendo un mensaje de error para explicar la razón por el fallo de la request (string).
- **warning message**: Opcional, similar al anterior pero la request es procesada normalmente sin interrumpir el proceso, y la advertencia es mostrada como un error.
- **interval**: Intervalo en segundos el cual el cliente debe esperar antes de seguir enviando datos en forma de requests al tracker.
- **min interval**: Opcional. Intervalo mínimo de anuncio o transferencia, si este está presente los clientes no deberían reanunciarse más veces que el mismo.
- **tracker id**: Un valor string que el cliente devuelve en sus anuncios próximos. Siempre es necesario tener al menos un valor de este campo, aunque sea de un anuncio previo.
- **complete**: Número de peers con el archivo completo, es decir seeders, en formato de integer.
- **incomplete**: Número de peers no-seeders, AKA leechers. (integer).
- **peers (modelo de diccionario)**: Es una lista de diccionarios que incluyen las siguientes llaves:
    - **peer id**: La ID autoseleccionada del peer, como se describe más arriba para la request para el tracker (string).
    - **ip**: La IP del peer, ya sea IPv6 o IPv4 o el DNS name (string).
    - **port**: El número del puerto del peer (integer).
- **peers (modelo binario)**: En vez de usar el modelo del diccionario, el valor de *peers* puede ser una string que consiste en múltiples de 6 bytes. Los primeros 4 siendo la dirección IP y los últimos 2 el puerto.

Como se mencionó anteriormente, la lista de peers tiene una longitud de 50 por defecto. Si hay menos peers en el torrent, entonces la lista será más chica. De otra forma, el tracker seleccionará peers de forma random para incluir en la respuesta.

Los clientes pueden enviar una request al tracker más seguido que el intervalo especificado, si un *event* ocurre o si el cliente necesita saber de más peers. Sin embargo, es mala práctica "hammerear" a un tracker para conseguir múltiples peers: si un cliente quiere una lista larga de peers en la respuesta entonces tendría que especificar el parámetro *numwant.*

La convención scrape

Por convención, la mayoría de los trackers soportan otra forma de request, conocida como *scrape*, que consulta el estado de uno o más torrents que el tracker está manejando. El objetivo es automatizar la recolección de estadísticas que de otro modo implicaría hacer screen scraping de la página del tracker.

El método *scrape* se realiza mediante una request HTTP GET. La URL base de *scrape* se deriva a partir de la announce URL de la siguiente manera:

1. Se identifica el último / en la URL del announce.
2. Si el texto inmediatamente después no es announce, se considera que el tracker **no** soporta scrape.
3. Si sí lo es, se reemplaza announce por scrape.

Ejemplos (announce URL → scrape URL):

- http://example.com/announce → http://example.com/scrape
- http://example.com/x/announce → http://example.com/x/scrape
- http://example.com/announce.php → http://example.com/scrape.ph
- http://example.com/a → (scrape no soportado)
- http://example.com/announce?x2%0644 → http://example.com/scrape?x2%0644
- http://example.com/announce?x=2/4 → (scrape no soportado)
- http://example.com/x%064announce → (scrape no soportado)

No se debe hacer unquoting de entidades URL codificadas.

El parámetro opcional info_hash (20 bytes) puede ser incluido para consultar datos de un solo torrent. Si no se especifica, se devuelven estadísticas de todos los torrents manejados por el tracker.

Algunos trackers aceptan múltiples info_hash en la misma URL, aunque no es parte del estándar oficial.

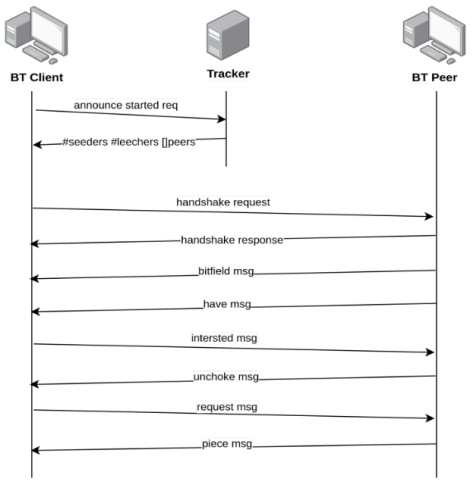
La respuesta del tracker es un documento "text/plain" (a veces comprimido en gzip), que contiene un diccionario bencodeado con la siguiente estructura:

- files: Diccionario que contiene una entrada por cada torrent del que se proveen estadísticas.

    - **Clave:** el info_hash de 20 bytes (binario).
    - **Valor:** un diccionario con las siguientes llaves:
        - complete: número de peers con el archivo completo (seeders), tipo integer.
        - downloaded: número total de veces que se ha completado la descarga, tipo integer.
        - incomplete: número de peers sin el archivo completo (leechers), tipo integer.
        - name: (opcional) nombre interno del torrent según el campo name del .torrent.

Peers

Protocolo peer por TCP



El protocolo peer facilita el intercambio de las piezas como se describe en el archivo metainfo (*.torrent)*.

Un cliente debe mantener información de estado por cada conexión que tiene con un peer remoto:

- **choked**: Si el peer remoto ahogó al cliente o no. Cuando un peer ahoga al cliente, es una notificación de que no se responderá ninguna request hasta que el cliente sea desahogado. El cliente no debería intentar mandar requests para obtener bloques (partes de información como se especificó más arriba en el documento), y deberia considerar que todas las requests pendientes (no respondidas) serán descartadas por el peer remoto.
- **interested**: Si el peer remoto está interesado o no en algo que el cliente tiene para ofrecer. Es una notificación que deja en claro que el peer remoto comenzará a pedir bloques cuando el cliente los desahoga.

Esto implica que el cliente también deberá mantener registro de si está interesado en el peer remoto, y de si tiene ahogado o desahogado al peer remoto. La lista real se vería algo así:

- **am_choking:** Este cliente está ahogando al peer.

- **am_interested:** Este cliente está interesado en el peer.
- **peer_choking:** El peer está ahogando a este cliente.
- **peer_interested:** El peer está interesado en este cliente.

Las conexiones del cliente comienzan como ahogadas y no interesadas. Es decir:

- **am_choking:** 1
- **am_interested:** 0
- **peer_choking:** 1
- **peer_interested:** 0

Se descarga un bloque por el cliente cuando el cliente está interesado en un peer, y si el peer no está ahogando al cliente.
Un bloque se sube cuando el cliente no está ahogando a un peer, y si ese peer está interesado en el cliente.

Es importante para el cliente mantener a sus peers informados sobre si está interesado en ellos o no. La información de estado debería mantenerse actualidad con cada peer, incluso cuando el cliente está ahogado. Esto le permite a los peers saber si el cliente comenzará a descargar cuando esté desahogado y viceversa.

Tipos de datos

A menos que se lo especifique de otra forma, todos los integers en el protocolo peer TCP están encodeados como valores de cuatro bytes en formato big-endian. Esto incluye el prefijo de longitud de todos los mensajes que vienen después del handshake.

Flujo de mensajes

El protocolo peer por TCP consiste de un handshake inicial. Después de eso, los peers se comunican intercambiando mensajes con un largo prefijado. El prefijo de la longitud es un integer como se lo describe arriba,

Handshake

El handshake es un mensaje requerido y debe ser el primer mensaje transmitido por el cliente. Tiene (49+len(pstr)) bytes de longitud.

El handshake se ve de la siguiente forma:
<pstrlen><pstr><reserved><info_hash><peer_id>

- **pstrlen**: String del largo de <pstr>, como un único byte crudo.
- **pstr**: String identificadora del protocolo.
- **reserved**: 8 bytes reservados. Todas las implementaciones actuales usan todos ceros. Cada bit en estos bytes puede ser usado para cambiar el comportamiento del protocolo.
- **info_hash**: Hash SHA1 de 20 bytes de la llave *info* en el archivo metainfo (*.torrent)*. Es la misma *info_hash* transmitida en las requests a los trackers.

- **peer_id**: Es una string de 20 bytes usada como un ID único para el cliente. Usualmente es el mismo *peer_id* que se transmite en las requests a los trackers, pero no siempre, dependiendo del cliente y de sus capacidades.

El iniciador de una conexión debería transmitir su handshake de forma inmediata. El recipiente debería esperar al handshake del iniciador, si es capaz de servir muchos torrents simultáneamente (los torrents son identificados de forma única por su infohash). De todos modos, el recipiente debe responder tan pronto como vea la parte de *info_hash* en el handshake.

Si un cliente recibe un handshake con un *info_hash* que no está compartiendo actualmente, entonces el cliente debe dropear la conexión.

Si el iniciador de la conexión recibe un handshake en el cual el *peer_id* no matchee con el que se espera, entonces el iniciador debe dropear la conexión. Se debe notar que el iniciador presumiblemente recibió la información del peer por el tracker, que incluye el *peer_id* registrado por el peer. El *peer_id* del tracker y el del handshake deberían coincidir.

El *peer_id* tiene 20 bytes (caracteres) de largo. Hay dos convenciones con las cuales encodear el cliente y la versión del cliente en el *peer_id*:

- "-", dos caracteres para el id del cliente, cuatro dígitos ASCII para el número de versión, "-", seguido de números random. (Es la convención más usada, algunos clientes que la usan son Transmission, qBittorrent, Vuze, Deluge, µTorrent (hell no!), libtorrent (el mismo cliente de la implementación que se mostrará hacia el final del documento), etc)
- un valor alfanumérico ASCII para la identificación del cliente, hasta cinco caracteres para la versión, seguido de tres caracteres, seguido de caracteres random. Cada carácter en la versión representa un número de 0 a 63.

side-note del escritor (Licha): En la especificación se pasa muchísimo tiempo hablando de las distintas formas que tienen los peers de identificarse según su cliente y según el standard, así que lo recorté un montón porque sencillamente era bloat del tipo ADHD.

Mensajes peer-to-peer

Todos los mensajes restantes en el protocolo toman la forma de <length prefix><message ID><payload>, El prefijo del largo es un valor big-endian de 4 bytes. El id del mensaje es un solo byte decimal. El payload (la carga para los no entendidos en materia de Team Fortress) es dependiente del mensaje.

keep-alive: <len=0000>

El mensaje *keep-alive* es un mensaje con cero bytes, especificado con el prefijo de longitud seteado a cero. No hay id de mensaje ni payload. Los peers podrían cerrar una conexión si no reciben mensajes (keep-alive o cualquier otro) por un cierto periodo de tiempo, así que este mensaje debe ser para mantener la conexión viva (alive) si ningún comando ha sido enviado durante un tiempo determinado. Este tiempo es generalmente dos minutos.

choke: <len=0001><id=0>

El mensaje de ahogamiento tiene un largo prefijado y no tiene payload.

unchoke: <len=0001><id=1>
El mensaje de desahogamiento tiene un largo prefijado y no tiene payload.

interested: <len=0001><id=2>
El mensaje de interesado tiene un largo prefijado y no tiene payload.

not interested: <len=0001><id=3>
El mensaje de no interesado tiene un largo prefijado y no tiene payload.

have: <len=0005><id=4><piece index>
El mensaje have tiene un largo prefijado. La carga es el index basado en cero (comienza en cero) de una *piece* que fue recientemente descargada de forma exitosa y verificada con el hash.

(En la realidad no es tan así. Esto pasa porque no se espera que los peers descarguen piezas que ya tienen, y un peer puede elegir no "publicar" tener una piece a otro peer que ya lo tiene. Al mismo tiempo, podría ser valioso enviar un mensaje *have* a un peer que ya tiene un piece específico dado a que será útil en determinar qué piece es raro. Un peer malicioso podría elegir publicar tener piezas que sabe que el peer nunca descargará.)

bitfield: <len=0001+X><id=5><bitfield>
El mensaje bitfield solo será enviado inmediatamente después de que la secuencia de handshake sea completada, y antes de que cualquier otro mensaje se envíe. Es opcional, y no tiene que ser enviada si un cliente no tiene piezas.

Este mensaje tiene un largo variable, donde X es el largo del bitfield. El payload es un bitfield representando las piezas que fueron descargadas exitosamente. El bit alto en el primer byte corresponde a piezas de index 0. Los bits que están limpios indican piezas faltantes, y los bits seteados indican piezas válidos y disponibles. Los bits que quedan al final (spare bits) son seteados a cero. Algunos clientes envían bitfield con piezas faltantes incluso si tiene todos los datos. Luego, envía el resto de los piezas como mensajes have. Esta práctica es llamada lazy bitfield, y ayuda a combatir contra el filtro de los ISP aplicado al protocolo BitTorrent (te odio Movistar).

Un bitfield del largo equivocado es considerado un error. Los clientes deberían dropear la conexión si reciben bitfields que no son del largo correcto, o si el bitfield tiene alguno de los spare bits seteados.

request: <len=0013><id=6><index><begin><length>
El mensaje request tiene un largo prefijado, y es usado para pedir (request) un bloque. El payload contiene la siguiente información:

- **index**: Integer que especifica el index del piece.
- **begin**: Integer que especifica el byte offset dentro del piece en cuestión.
- **largo**: Integer que especifica el largo pedido.

Según la especificación oficial, el tamaño recomendado para los bloques solicitados es 2^15 (32KB), pero la mayoría de las implementaciones actuales utilizan bloques de 2^14 (16KB).
A partir de la versión 3 (2004), se cambió el tamaño de los bloques a 16KB. Desde la versión 4.0 (mediados de 2005), se comenzó a desconectar a los clientes que solicitaban bloques de más de 16KB. Aunque la especificación oficial permite bloques de 32KB, la mayoría de los clientes modernos utilizan bloques de 16KB debido a la mejora en el rendimiento (menos overhead de solicitudes).

piece: <len=0009+X><id=7><index><begin><block>
El mensaje piece es de largo variable, donde X es el largo del bloque. El payload contiene la siguiente información:

- **index**: Integer que especifica el index del piece.
- **begin**: Integer que especifica el byte offset dentro del piece.
- **block**: El bloque de datos, que es un subset de la *piece* especificada por index. (nótese que el bloque representa una parte más pequeña de la *piece* en la que se divide lo que se quiere descargar).

cancel: <len=0013><id=8><index><begin><length>
El mensaje cancel tiene un largo prefijado, y se usa para cancelar requests de bloques. El payload es igual al del mensaje de request. Es típicamente usado en la fase de final de juego.

port: <len=0003><id=9><listen-port>
Por último, el mensaje puerto es enviado por los clientes que implementan un tracker DHT. El listen-port es el puerto de escucha que el nodo DHT del peer está escuchando. Este peer debería ser insertado en la tabla de ruteo local (si el tracker DHT es soportado).

Dependencias

Para el funcionamiento de BitTorrent, depende directamente de protocolos como TCP, a diferencia de protocolos cliente-servidor tradicionales que utilizan FTP o HTTP.

BitTorrent usa TCP para las conexiones principales de datos, asegurando la entrega fiable de los paquetes. UDP puede ser utilizado para protocolos de control como DHT (Distributed Hash Table) o PEX (Peer Exchange), que no requieren garantías de entrega.

BitTorrent utiliza funciones hash como SHA-1 para crear identificadores únicos para los archivos (en este caso el infohash que identifica el archivo torrent) y verificar la integridad de los datos. Aunque SHA-1 está siendo reemplazado en algunas implementaciones por SHA-256 debido a sus vulnerabilidades.

BitTorrent utiliza puertos como el 6881-6889 por defecto (aunque muchos clientes permiten configurarlos para evitar restricciones).

El propio modelo de peer-to-peer es fundamental, ya que los peers se conectan entre sí para intercambiar fragmentos de archivos.

Extensiones aceptadas para la especificación original

Como ya se mencionó brevemente en el documento, la comunidad de BitTorrent coordinaba el desarrollo del protocolo proponiendo las llamadas BitTorrent Enhancement Proposals (BEPs). Estas son propuestas de mejoras al protocolo original de todo tipo. Muchas de estas fueron oficialmente aceptadas y agregadas a la implementación original de BitTorrent (también conocida como la Mainline). Las mismas llegan a ser tan recientes como hace 5 años (como se puede ver en el repositorio oficial de las BEPs en Github).

DHT

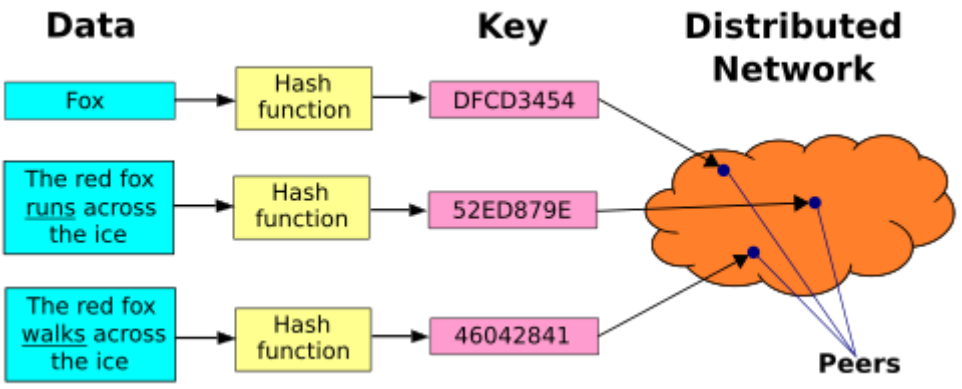
Antes de que existieran las tablas hash distribuidas (DHT) en BitTorrent, la única forma de encontrar otros peers para descargar un archivo era mediante trackers centralizados. Esto implicaba que si un tracker dejaba de funcionar, también lo hacía toda la red de compartición asociada a ese archivo.

En mayo de 2005, el cliente de BitTorrent Azureus (hoy conocido como Vuze) fue el primero en implementar una solución descentralizada utilizando una DHT basada en el diseño Kademlia, un algoritmo para organizar y buscar datos de forma eficiente en redes distribuidas. Poco después, de forma independiente, la empresa BitTorrent, Inc. desarrolló su propia implementación de DHT, basada en Kademlia y la incluyó en su cliente oficial. Esta versión se conoció como Mainline DHT (Mainline refiriéndose a la implementación oficial de BitTorrent).

El uso de DHT permitió eliminar la **dependencia** de los trackers, favoreciendo un sistema más resistente y escalable. En lugar de preguntar a un servidor central por los peers disponibles, los clientes podían encontrarse entre sí a través de esta red distribuida.

En el Mainline DHT cada cliente elige aleatoriamente un ID de 160 bits y utiliza operaciones XOR para calcular distancias entre nodos. A través de estructuras llamadas k-buckets, cada cliente mantiene una lista de nodos conocidos y puede buscar otros usuarios que compartan el archivo usando el *info_hash* del torrent.

El éxito de esta tecnología fue tan grande que para 2013 se estimaba que entre 16 y 28 millones de usuarios estaban conectados simultáneamente a Mainline DHT, con variaciones de más de 10 millones a lo largo del día.



Es importante aclarar que **no todas las DHT son iguales**, aunque muchas se basen en Kademlia, existen distintos diseños y adaptaciones según el uso y el protocolo. Además, con el paso del tiempo, los investigadores descubrieron que muchas DHT eran vulnerables a un tipo particular de amenaza: el **ataque Sybil**. Este ataque consiste en que un atacante crea una gran cantidad de identidades falsas dentro de la red para obtener control o influenciar el sistema de una manera maliciosa. Frente a este problema, surgieron propuestas de DHTs resistentes a **fallos bizantinos**, es decir, diseñadas para seguir funcionando incluso cuando parte de los nodos se comportan de manera maliciosa.

Tablas de enrutamiento de PHT

Cada nodo mantiene una tabla de enrutamiento con nodos denominados como "buenos". Estos nodos se usan como punto de partida para consultas en la DHT (Distributed Hash Table) y también se devuelven en las respuestas a consultas de otros nodos.

No todos los nodos que descubrimos son iguales. Algunos son buenos, y otros no. Un nodo se considera bueno si ha respondido a una de nuestras consultas en los últimos 15 minutos. También se considera bueno si ha respondido a una consulta nuestra alguna vez y nos ha enviado una consulta en los últimos 15 minutos. Si no hay actividad por 15 minutos, el nodo se vuelve cuestionable. Si falla múltiples veces al responder, se considera malo. Los nodos buenos tienen prioridad en la tabla.

La tabla de enrutamiento cubre todo el espacio de IDs de 0 a 2^{160}. Está dividida en "buckets" (cubos), cada uno cubriendo una parte del espacio. Una tabla vacía empieza con un solo bucket que cubre todo el rango. Al insertar un nodo con ID "N", se coloca en el bucket cuyo rango incluye a "N". Cada bucket puede contener hasta K = 8 nodos. Si el bucket está lleno de nodos buenos y nuestro ID no está en su rango, no se puede añadir más nodos.

Si nuestro ID sí está en el rango del bucket lleno, se divide en dos buckets con la mitad del rango cada uno. Los nodos se redistribuyen entre los nuevos buckets. Por ejemplo, el primer bucket lleno siempre se divide en los rangos $0–2^{159}$ y $2^{159}–2^{160}$.

Si un bucket está lleno de nodos buenos, se descartan los nuevos nodos. Pero si hay nodos malos, uno puede ser reemplazado. Si hay nodos cuestionables (inactivos por más de 15 minutos), se les hace un ping. Si responden, se continúa con el siguiente menos reciente. Si alguno no responde, se intenta una segunda vez antes de reemplazarlo. Así se mantiene una tabla estable con nodos activos.

Cada bucket debe mantener un registro de "último cambio". Si se añade un nodo, se reemplaza uno, o se hace ping exitosamente, se actualiza este valor. Si un bucket no cambia en 15 minutos, debe refrescarse haciendo una búsqueda find_node con un ID aleatorio dentro de su rango. Los nodos que no pueden recibir consultas deben refrescar más seguido su tabla.

Finalmente, al insertar el primer nodo o al arrancar el software, el nodo debe intentar encontrar los nodos más cercanos a sí mismo en la DHT usando find_node de manera iterativa. Además, la tabla de enrutamiento debe guardarse entre ejecuciones del cliente para mantener la información.

Extensión del protocolo BitTorrent

El protocolo BitTorrent se ha extendido para intercambiar números de puerto UDP de los nodos entre pares que son presentados por un tracker. De esta forma, los clientes pueden sembrar automáticamente sus tablas de enrutamiento al descargar torrents normales.

Cuando un cliente es nuevo y su primer intento es descargar un torrent sin tracker (trackerless), no tendrá nodos en su tabla de enrutamiento. Por eso, necesita los contactos que vienen incluidos en el archivo .torrent para comenzar.

Los pares que soportan la DHT (Distributed Hash Table) activan el último bit de los 8 bytes reservados en el handshake del protocolo BitTorrent. Cuando un par recibe un handshake que indica que el otro par soporta DHT, debe enviar un mensaje PORT.

Este mensaje comienza con el byte 0x09 y contiene una carga útil de 2 bytes que representa el puerto UDP del nodo DHT, en orden de red (big-endian).

Al recibir este mensaje, el par debe intentar hacer un ping al nodo usando el puerto recibido y la dirección IP del par remoto. Si se recibe una respuesta al ping, el nodo debe intentar insertar esta nueva información de contacto en su tabla de enrutamiento, siguiendo las reglas normales.

Extensiones de Archivos torrent

Un diccionario de un torrent sin Tracker no tiene una llave para “publicar”, en vez posee una llave “nodos”. Esta llave debe tener un valor de k nodos más cercanos al torrent generando la tabla enrutadora del cliente. Alternativamente la llave puede ser un valor conocido de un nodo “bueno” como puede ser el de la persona generando el torrent. Se recomienda no agregar “router.bittorrent.com” al final de archivos torrent o agregar este nodo a las tablas de enrutamiento de los clientes.

PEX

“Peer exchange”, comúnmente abreviado como PEX es un protocolo de comunicaciones que expande el alcance de BitTorrent, permitiendo que un grupo de usuarios (peers) colaboren a la descarga de un archivo de manera más eficiente.

En un principio los usuarios compartían un grupo conocido como “swarm”, el cual su funcionamiento dependía directamente del tracker para permitir a cada peer actualizar información con el resto del grupo.

Mientras tanto PEX alivia la fuerte dependencia al tracker permitiendo que cada peer pueda actualizar su estado directamente a los demás del grupo, incrementando la velocidad, eficiencia y seguridad del protocolo BitTorrent. Sin embargo no hay que olvidar que este es simplemente otro medio para hallar direcciones IPs, y su funcionalidad no se extiende a acciones como introducir un nuevo peer a un swarm, para hacer este contacto inicial con el grupo cada peer debe estar conectado a un tracker mediante un archivo .torrent o conectarse mediante un router llamado “bootstrap node” para encontrar la tabla hash distribuida (DHT) la cual contenga la lista de peers dentro del swarm.

Para la mayor parte de los usuarios de BitTorrent tanto PEX como DHT comenzarán a trabajar automáticamente una vez que el usuario inicialice un cliente de BitTorrent y abra un archivo .torrent. Cabe destacar que los torrents privados son una excepción, ya que como estos no están libremente disponibles suelen tener desactivado el algoritmo DHT

Estas son algunas de las convenciones más utilizadas para el uso de Peer exchange:

Extenciones como AZMP o LTEP son típicamente utilizadas para la implementación de PEX, ambos consisten en enviar paquetes conteniendo un grupo de peers para ser agregados a la swarm y otro grupo de peers para ser removidos.

Está acordado entre clientes y compañías como Azureus y desarrolladores de µTorrent que cualquier cliente que quiera implementar cualquiera de las funcionalidades previamente mencionadas tienen que regirse a los siguientes límites a la hora de enviar mensajes PEX:

- No deben haber más de 50 peers agregados o removidos mandados en cualquier mensaje PEX
- Un mensaje de peer exchange no debe ser enviado más frecuentemente que uno por minuto.

Los clientes pueden elegir si quieren regirse por estos límites y cortar conexiones con clientes que los ignoren.

Ahora, hay que tener en cuenta algunas consideraciones de seguridad a la hora de implementar el intercambio de peers.

La información transmitida por mensajes PEX debe ser considerada desconocida o incluso potencialmente maliciosa, ya que un atacante exterior puede intentar sabotear la operación de una swarm llenándola con información irrelevante o introduciendo peers inútiles.

PEX también puede ser utilizado para lanzar ataques DDOS mediante clientes de BitTorrent que intentan conectarse a las direcciones IPs de sus víctimas.

Para mitigar estas posibles inseguridades un cliente debería evitar recibir todas sus conexiones de una sola fuente PEX, ignorando IPs duplicadas incluso en puertos diferentes. Adicionalmente implementar un sistema de prioridad de peers puede ayudar a esparcir los intentos de conexión a otras subredes, minimizando el impacto de cualquier ataque o interferencia.

Fast

Fast refiere a varias extensiones que cumplen múltiples propósitos. Permiten que un peer se integre más rápidamente a un swarm al proporcionar un conjunto específico de piezas que podrá descargar independientemente de su estado de chokeado. También reducen la sobrecarga de mensajes al agregar los mensajes *HaveAll* y *HaveNone*, y permiten el rechazo explícito de solicitudes de piezas, mientras que anteriormente solo era posible un rechazo implícito, lo que significaba que un par podía quedar esperando una pieza que nunca sería entregada.

La extension Fast modifica la semántica de los mensajes existentes de *request, chocke, unchoke* y *cancel*, y añade una *reject request*. Cada request tiene la garantía de resultar en exactamente una respuesta la cual es el correspondiente mensaje *reject* o *piece*. Incluso cuando una *request* es cancelada, el peer que recibe la cancelación debería responder con la *reject* correspondiente o la *piece*: las requests que están siendo procesadas se permite que completen.

Choke no rechaza más implicitamente todas las requests pendientes, elimiando algunas race conditions que podrían causar que algunas piezas sean requesteadas muchas veces de forma innecesaria. Adicionalmente, si un peer recibe una pieza que nunca fue requesteada, el peer debe cerrar la conexión.

Aquí he algunas de estas extensiones y su principal método para llevar a cabo este fin.

Have all / Have none:
Esta extensión especifica que el responsable de enviar el mensaje tenga todas o ninguna de las piezas respectivamente. Esto reemplazaría el campo “Have bitfield”, y un valor de Have all, Have none, o Have bitfield debe aparecer sólo inmediatamente después de concretarse el handshake. El fin de estos mensajes es guardar ancho de banda y remover parcialmente la idiosincrasia de no enviar un mensaje cuando un peer no tiene una pieza.

Suggest Piece:
Su nombre es un mensaje el cual quiere decir “a usted le gustaría descargar esta pieza” y fue diseñado para ser utilizado al hacer “super-seeding” sin pérdidas de rendimiento, al no realizar descargas innecesarias. Para esto cada seed debe intentar mantener la misma cantidad de copias de cada pieza en la network, y un peer puede mandar más que un mensaje de “suggest piece” en cualquier momento. Y un peers que recibe muchas copias de este mensaje puede interpretar que las piezas sugeridas son igual de apropiadas.

Reject Request:
Notifica al peer que realiza la request que la misma no será satisfecha. El mismo debe cerrar la conexión poco después de recibir este mensaje en el caso que esté deshabilitada la extensión FAST. Pero si está activada:

- Si el peer recibe una request que nunca fue enviada el mismo debe cerrar la conexión.
- Si el peer envía un choke, debe rechazar todas las requests del peer al cual le fue enviado el choke, excepto que sean requests por piezas que formen parte de la extensión “allowed fast”. Un peer debe realizar un choke y después rechazar las requests para que el otro peer no repita la request de las piezas.
- Si un peer recibe una request de un peer que le realizó choking, el peer recibiendo debe rechazar a menos que la pieza forme parte de “allowed fast”.
- Si un peer recibe una cantidad excesiva de requests de un peer que está haciendo choking, el peer recibiendo puede optar por cerrar la conexión antes que rechazar la request. Sin embargo se tiene que tener en cuenta que puede tomar varios segundos para que los buffers drenen y propagen los mensajes una vez que se realiza choke a un peer.

Allowed Fast:
Su nombre representa un mensaje que dice “si pedis esta pieza, te la daré incluso si estas choked”. Esto ayuda a acortar el proceso de salir del estado de choke, en donde un peer se puede encontrar alternando constantemente entre estos dos estados. Las piezas permitidas por Allowed fast son especificadas en una lista fija, la misma es generada a partir de un algoritmo canónico que produces index de piezas únicos para el receptor del mensaje, para en el caso que dos peers ofrezcan k piezas fast serán la misma k, y si uno pasa a ser k+1 será lo mismo más uno más. En este ejemplo k debería ser lo suficientemente chico para evitar abusos de requests, actualmente se setea por defecto a 10, pero los peers son libres de cambiarlo de acuerdo a su respectiva demanda.

El responsable del mensaje puede listar piezas que el receptor no tiene y este no debe interpretar un mensaje Allowed fast como sinónimo de que el primero tenga la pieza. Esto les permite a los peers generar y comunicar listas de Allowed fast al comienzo de una conexión. Sin embargo un peer es capaz de enviar este tipo de mensajes en cualquier momento.

Un peer puede rechazar requests de Allowed fast solo en el caso que no tenga los recursos necesarios, en el caso que la pieza demandada ya fue enviada al peer o si el peer que realizó la request no es un peer original. Algunas implementaciones eligen rechazar request de mensajes Allowed fast cuando el peer en cuestión tiene más de k piezas.

Magnet links

El propósito principal de esta extensión es permitir unir los clientes a una swarm y completar una descarga sin necesidad de descargar un archivo .torrent. En su lugar se opta por descargar la metadata desde los peers directamente mediante magnet links, el cual consiste en un link web con toda la información necesaria para unirse a la swarm deseada.

La extensión de la metadata solo transfiere la información en forma de diccionario dentro del archivo .torrent. Esta operación puede ser validada por el info-hash.

La metadata es manejada en bloques de 16Kib (16384). Los bloques de metadata son indexados comenzando desde el 0, todos tienen el mismo peso excepto el último el cual puede ser más chico.

La extensión utilizada por la metada usa un protocolo para publicar su existencia a otros. Agregando “ut_metadata” al lugar correspondiente de “m” en el diccionario como parte del mensaje tras realizar el handshake dentro del header de la extensión. Exponiendo el código del mensaje usado para ese mensaje. También puede agregar “metadata_size” al mensaje del handshake para especificar un valor íntegro como número de bytes de la metadata.

Los mensajes de la extensión está bencodeados, existen 3 tipos diferentes de mensajes:

- 0. request
- 1. data
- 2. reject

Estos mensajes poseen una llave “msg_type” el cual su valor indica el tipo de mensaje. También tienen otra llave “piece”, para indicar que parte de la metadata se está refiriendo al mensaje en cuestión.
Para ser compatible con expansiones futuras, se tienen que ignorar los IDs de mensajes no identificados.

Los mensajes de request no tienen ninguna llave adicional dentro de su diccionario, ya que la respuesta de un peer usando la extensión es de rechazo o “data message”. La respuesta debe tener la misma pieza que tuvo la request.

Un peer debe verificar que cualquier pieza que envía cumpla con la verificación estipuladas en el info-hash, por lo menos hasta que el cliente acceda a toda la metadata. Por su parte los peers que no tengan toda la metadata deben responder con un mensaje de rechazo a cualquier request de metadata.

Ejemplo:
{'msg_type': 0, 'piece': 0}

d8:msg_typei0e5:piecei0ee

Los mensajes de data agregar otro documento de información al diccionario, indicado por "total_size". Esta llave tiene la misma semántica que "metadata_size" en la extensión del header, expresado como íntegro.
La pieza de metadata es fusionada con el diccionario bencodeado, pero no es parte del mismo, sino del mensaje. La pieza de data debe pesar 16Kib a menos que sea la última pieza de la metadata.

Ejemplo:
{'msg_type': 1, 'piece': 0, 'total_size': 3425}
d8:msg_typei1e5:piecei0e10:total_sizei34256eexxxxxxxxx...

Los mensajes de reject no poseen ninguna llave adicional en su mensaje. Esto da a entender que el peer no tiene la pieza de metadata que estaba pidiendo.
Los clientes pueden implementar flood protection al rechazar mensajes de request luego de que un cierto número de ellos hayan sido recibidos, el cual suele ser el número de piezas de metadata por un determinado factor.

Ejemplo:
{'msg_type': 2, 'piece': 0}
d8:msg_typei1e5:piecei0ee

El formato de magnets URI es:

v1:
magnet:?xt=urn:btih:<info-hash>&dn=<name>&tr=<tracker-url>&x.pe=<peer-address>

v2: magnet:?xt=urn:btmh:<tagged-info-hash>&dn=<name>&tr=<tracker-url>&x.pe=<peer-addr ss>

<info-hash>
Es el info-hash codificado, para un total de 40 caracteres. Para que sea más compatible con links ya existentes, los clientes deben también aceptar el estándar de 32 caracteres propio del info-hash codificado en base32.

<tagged-info-hash>
Es el multihash formateado, es decir info-hash codificado en hex para torrents con el nuevo formato de metadata. Los campos "btmh" y "btih" pueden existir en el mismo magnet sí describen el mismo torrent híbrido.

<peer-address>
Representa la dirección de un peer expresada como hostname:port, ipv4-literal:port or [ipv6-literal]:port.

Este parámetro puede ser incluido al inicializar una transferencia de metadata directa entre dos clientes reduciendo la necesidad de fuentes externas. Solo debe ser incluida si el cliente puede descubrir su IP pública y determinar su alcance.

Después tenemos xt, el cual es el único parámetro obligatorio.

dn refiere al nombre que muestra mientras el cliente está esperando la metadata.

tr es la url del Tracker si hay uno, o pueden haber muchos si son múltiples.

x.pe es lo mismo que tr pero para los peers.

Si no hay un Tracker especificado, el cliente debe usar el DHT para adquirir peers.

uTorrent transfer protocol

El protocolo de transporte uTorrent (uTP) se creó para que los clientes de BitTorrent no interrumpan conexiones de internet, pero que sigan usando todo el ancho de banda no usado de forma completa.

El problema es que los módems de cable y de DSL tienen típicamente un buffer de envío desproporcional a su frecuencia de envío máxima, que puede retener muchos segundos que podrían ser paquetes. El tráfico de BitTorrent es generalmente transferencias en segundo plano, y debería tener menos prioridad que chequear email o surfear la web, pero cuando se usan conexiones TCP regulares BitTorrent llena rápidamente el buffer de envio, añadiendo muchos segundos de delay a todo el tráfico interactivo.

El hecho de que BitTorrent usa muchas conexiones TCP le da una ventaja injusta cuando compite con otros servicios por ancho de banda, lo cual exagera el efecto de BitTorrent llenando la pipe de subida. La razón de esto es porque TCP distribuye el ancho de banda disponible de forma pareja entre las conexiones, y mientras más conexiones use una aplicación, más largo será el ancho de banda que obtiene.

uTP resuelve este problema usando el tamaño de la cola del módem como un controlador para su frecuencia de envío. Cuando la cola se hace muy grande, se desacelera. Esto permite utilizar la máxima capacidad de subida cuando no hay competencia por ella, y permite que se desacelere a la nada absoluta cuando hay mucha cantidad de tráfico interactivo.

No explicaremos mucho más de esta extensión ya que es un protocolo de transporte en sí, pero podemos agregar que es un protocolo de transporte construido encima de UDP.

Holepunch Extension

Esta extensión provee una forma de conectarse a los peers que no pueden recibir conexiones entrantes, ya sea porque están detrás de un NAT filtrador o por un firewall que bloquea conexiones entrantes.

Agrega un nuevo mensaje *ut_holepunch*. Se usa para coordinar una conexión uTP (utorrent transfer protocol) entre dos peers a través de un tercero (el peer rele).

Los tipos de mensajes son los siguientes:

msg_type	name	description
0x00	rendezvous	Mensaje inicial enviado por el peer que quiere iniciar la conexión.
0x01	connect	El peer relé informa a ambos peers cómo conectarse.
0x02	error	Algo salió mal (peer destino no válido, no conectado, etc.).

El peer que inicia la conexión envía un mensaje rendezvous al peer relé, conteniendo el endpoint (IP y puerto) del peer objetivo. Si el peer relé está conectado al peer objetivo, y el peer objetivo soporta esta extensión, el peer relé envía un mensaje connect a el peer inicializador y el peer objetivo, cada uno conteniendo el endopoint del otro. Cuando se recibe dicho mensaje, cada peer inicia una conexión uTP al otro peer. Si no puede ser enviado exitosamente el mensaje rendezvous el peer relé debería responder al peer inicializador con un error de mensaje.

Torrents privados

Un tracker privado restringe el acceso a los torrents que trackea. Un torrent con acceso restringido es llamado torrent privado. Todos los otros torrents son públicos. Para promover el compartimiento, los trackers privados suelen mantener estadísticas sobre usuarios registrados y restringe el acceso a algunos o a todos los torrents a los clientes que no suben adecuadamente.

Cuando se genera el archivo metainfo (*.torrent*) los usuarios denotan un torrent como privado incluyendo el par llave-valor "*private=1"* en el diccionario *info* del archivo metainfo.

Cuando un cliente BitTorrent obtiene un archivo metainfo conteniendo la llave "private=1" debe solo anunciarse al tracker privado, y debe solamente iniciar conexiones con los peers que devuelve dicho tracker.

Cuando muchos trackers aparecen en la *announce-list* dentro del archivo metainfo de un torrent privado, cada peer debe usar solamente un tracker a la vez y solo cambiar entre trackers cuando dicho tracker falla. Cuando cambia entre trackers, el peer debe desconectarse de todos los peers actuales y solo conectarse a los que provee el nuevo tracker.

Bibliografía

https://wiki.theory.org/BitTorrentSpecification
https://www.xataka.com/basics/bittorrent-que-como-funcionan-torrents
https://www.rfc-editor.org/rfc/rfc5694#section-2.4
https://www.youtube.com/watch?v=hkA85I5m7xo
https://www.youtube.com/watch?v=EkkFT1bRCT0
https://en.wikipedia.org/wiki/Mainline_DHT
https://en.wikipedia.org/wiki/Distributed_hash_table
https://www.geeksforgeeks.org/sybil-attack/
https://en.wikipedia.org/wiki/Sybil_attack
https://en.wikipedia.org/wiki/Peer_exchange
https://en.wikipedia.org/wiki/Bencode
bep_0003.rst_post
bep_0000.rst_post
How to Write a Bittorrent Client, Part 1 | kristenwidman
How to Write a Bittorrent Client – Part 2 | kristenwidman
How to make your own bittorrent client
https://en.wikipedia.org/wiki/BitTorrent_tracker
https://taller-1-fiuba-rust.github.io/proyecto/22C1/proyecto.html
https://bittorrent.org/beps/bep_0011.html
https://bittorrent.org/beps/bep_0009.html
https://bittorrent.org/beps/bep_0005.html
https://bittorrent.org/beps/bep_0029.html
https://bittorrent.org/beps/bep_0006.html