Trabajo práctico - Algoritmos en sistemas distribuidos

Sistemas Operativos - 2do Cuatrimestre 2016

Fecha de entrega: Viernes 18 de noviembre de 2016 a las 23:00hs

1. Introducción

1.1. BitTorrent

El protocolo **BitTorrent**[1] tiene como objetivo el distribuir datos y archivos en *Internet*. Es especialmente utilizado para transferir archivos de gran tamaño.

En lugar de alojar el archivo en un servidor, se alojará en varios, llamados **peers**. De esta manera se evitan cuellos de botella y puntos de falla. A su vez, los archivos se dividen en partes o *piezas* que son identificadas unívocamente por un *hash* y almacenadas en algún nodo de la red (no necesariamente todos en el mismo). De esta manera se pueden compartir esas partes una vez que cada usuario las tiene, aunque no posea aún el archivo completo. Este protocolo tiene una lógica distribuida y mejora mientras más usuarios compartan contenido, por eso es que se premia a los usuarios que lo hagan con una mayor velocidad de descarga.

Existe normalmente un archivo *tracker*, llamado normalmente `.torrent` que identifica todas las *piezas* de un dado elemento a buscar y en qué nodo de la red están. Esta lista se descarga normalmente de algunos sitios web que las publican, generando nuevamente puntos de falla únicos. Como alternativa, existen los los sistemas de trackers descentralizados o *trackerless* en los cuales cada **peer** actúa como un *tracker*. Para realizarlo utiliza una **DHT** o **D**istributed **H**ash **T**able.

1.2. DHT

El objetivo de una **DHT** es otorgar el servicio de diccionario distribuído. Es decir un *filesystem* distribuído.

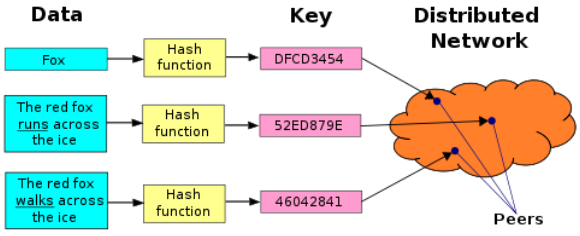

Figura 1: Cómo se distribuye, utilizando un *hash* los datos en un **DHT**

Buscando obtener, dado el *hash* de un archivo, en qué **peer** se encuentra. Es decir que guardaremos ese archivo (o esa pieza del mismo) en ese **peer**.

[1] `bittorrent.org`

Existen varias maneras de encontrar el **peer** indicado para un archivo dado. Una de ellas es realizar un *hash* del id del nodo (que podría ser su IP) y guardar el archivo en el **peer** cuyo *hash(id)* sea más cercano al *hash(archivo)*, según alguna métrica.

De esta manera se genera una red *Overlay* por sobre la red IP, que permitirá encontrar un archivo de una manera eficiente sin la necesidad de buscarlo por todos los nodos. Además contará con otras ventajas, como estar suficientemente distribuída (por las características del *hash*) y así soportar caídas de segmentos de la red.

Para más información leer:

- `http://bittorrent.org/beps/bep_0005.html`
- `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.76.8338&rep=rep1&type=pdf`
- `https://en.wikipedia.org/wiki/Kademlia`

Este TP consiste en implementar una versión simplificada de la **DHT Kademlia** que detallaremos a continuación.

2. Escenario

Generaremos una red que tendrá un nodo distinguido al que llamaremos **consola** y otro que será el nodo de contacto o **Contact**. Luego existirán varios nodos no distinguidos. El nodo **consola NO** será parte de la red.
Cada nodo además guardará una tabla de ruteo, de hasta **K** vecinos más cercanos según su distancia **XOR**. Al iniciar una búsqueda (*Look-up*) de un nodo, ingresar uno nuevo (*Join*) o almacenar un archvio nuevo (*Store*), se realizará siempre por el nodo **Contacto**.

Notas:

- Tendremos 1 nodo de tipo **Contacto** y el resto serán **peers**
- No tendremos separación de nodos en otros tipos
- No borraremos archivos
- Un archivo lo representaremos únicamente por su hash
- Una vez arrancado el sistema, no se caerán los nodos, ni se crearán nuevos, sólo se podrán realizar operaciones con los nodos existentes.

3. Algoritmos

- **Join**: Ingresa un nuevo **peer** a la red
  - El nodo **consola** le enviará al nodo que quiere ingresar a la red su **Rank** y cuál es el nodo de **Contacto**
  - Si el nodo que está ingresando es el contacto, queda ingresado (es el primero)
  - Si no lo es, asume que el **contacto** es el más cercano a él.
  - Para cada uno de los nodos más cercanos

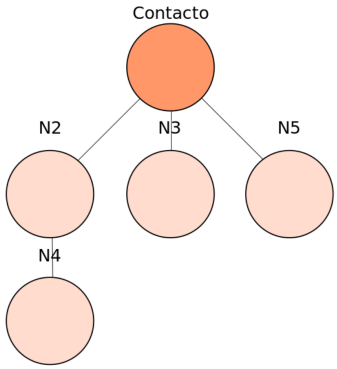

Figura 2: Esquema de ejemplo de una red y sus vecinos

- Le envía al nodo su hash y rank.
- Recibe o bien ok, o sea queda agregado a la lista de contactos de ese nodo, o bien una lista de nodos más cercanos a él, que son agregados
- Una lista de archivos que debe agregar a su base porque son migrados desde ese nodo hacia él (por ser el nuevo más cercano a esos archivos)

- **Look-up**: Busco un archivo en la red. Toma como parámetros el **Rank** del nodo que busca el archivo y el **filename** buscado

  - El nodo **consola** realiza el **hash** del archivo
  - Le enviará al nodo **Rank** esos datos.
  - Si lo tiene lo devuelve, sino para cada nodo cercano de su base al hash (Busca el/los nodo/s que lo debe/n tener:

    - Le envía al nodo el hash
    - Recibe o bien ok, o sea lo tiene ese nodo y devuelve ese valor
    - O bien la lista de vecinos del nodo, entre los cuales agrega a ser buscados los más cercanos al hash.

- **Store**: Guarda un archivo en la red. Toma como parámetros el **Rank** del nodo que guarda el archivo y el **filename**

  - El nodo **consola** realiza el **hash** del archivo
  - Le enviará al nodo **Rank** esos datos.
  - Si lo tiene lo devuelve, sino para cada nodo cercano de su base al hash (Busca el/los nodo/s que lo debe/n tener:

    - Le envía al nodo el hash
    - Recibe o bien ok, o sea lo debe tener ese nodo, luego lo agrega a la lista de los que lo deben tener.
    - O bien la lista de vecinos del nodo, entre los cuales agrega a ser buscados los más cercanos al hash.

  - Finalmente, envía el archivo a ser guardado en los nodos más cercanos al hash buscado.

4. Implementación

Para implementar el TP utilizaremos MPI[2], pero más precisamente trabajaremos en python[3] Para hacerlo, utilizaremos las siguientes funciones de MPI:

- `MPI.COMM_WORLD.send()` Envío bloqueante de mensajes.
- `MPI.COMM_WORLD.recv()` Recepción bloqueante de mensajes.
- `MPI.COMM_WORLD.isend()` Envío No bloqueante de mensajes.
- `MPI.COMM_WORLD.irecv()` Lectura no bloqueante de mensajes.
- `MPI.COMM_WORLD.iprobe()` Indica si hay o no mensajes, de manera no bloqueante.
- `MPI.COMM_WORLD.wait()` Espera de manera bloqueante un mensaje / grupo

Una buena descripción de ellas se encuentra en `https://computing.llnl.gov/tutorials/mpi` y en python en `http://pythonhosted.org/mpi4py/`

5. Ejecución

Para ejecutar el programa compilado debe usarse:

```
mpiexec -np N python <programa.py>
```

donde `N` es la cantidad de procesos a lanzar.

6. Entregar

Se utilizará como base el archivo tp3.py entregado por la cátedra.
El mismo al correr permitirá ingresar las siguientes opciones:

```
* j|join <node_rank>
* s|store <node_rank> <file_name>
* l|look-up <node_rank> <file_name>
* q|quit
```

Se pide:

1. Implementar de manera bloqueante los métodos:

   A) `__find_nodes`
   B) `__handle_console_look_up`
   C) `__find_nodes_join`
   D) `__handle_console_store`

2. Generar casos de prueba de los mismos y mostrar que funciona

[2] `http://www.mpi-forum.org/docs/docs.html`
[3] `https://pythonhosted.org/mpi4py/`

3. Implementar de manera no bloqueante los métodos. No hace falta hacerlos 100 % no bloqueantes, pero justificar adecuadamente las partes que no se realicen de esa manera.

   A) `__find_nodes`

   B) `__find_nodes_join`

4. Generar casos de prueba de los ítems anteriores y mostrar que funciona