1. Expose the pros and cons of temporal and spatial coupling. Give examples for the four combinations of space and temporal coupling/uncoupling different from the ones that you can find in the book. For these examples justify why this coupled/uncoupled design has been chosen.

Antes de nada, voy a indicar brevemente qué es el desacoplamiento/acoplamiento espacio/tiempo y a qué tipo de comunicación se refiere cada una, si a directa o indirecta. En el libro de referencia para esta asignatura se indica *"Una comunicación indirecta se define como la comunicación entre dos entidades de un sistema distribuido a través de un tercero (intermediario), el cual evita el acoplamiento directo entre emisor y receptor."* Es decir, una comunicación directa es la que hay entre dos nodos sin pasar por un tercero (necesita acoplamiento temporal y espacial), y en la comunicación indirecta para comunicarse un nodo con otro, es necesario que pase por un nodo intermediario (un tercer nodo que se ubica entre ambos, permite desacoplamiento espacial y/o temporal).

Según el tipo de acoplamiento podemos hablar de:

- Desacoplamiento espacial: el emisor no sabe (o no necesita saber) quién es el receptor(es) y viceversa.
- Desacoplamiento temporal: tanto el emisor como el receptor tienen tiempos de vida diferentes, esto es, no necesitan coexistir en un preciso instante para comunicarse.

La siguiente tabla nos da una información resumida y concisa de los conceptos que estamos tratando (se ha obtenido del libro de referencia de la asignatura):

	Acoplamiento temporal	**Desacoplamiento temporal**
Acoplamiento espacial	Comunicación directa dirigida a un receptor o receptores; los receptores deben existir en ese momento. Los participantes de la comunicación deben conocer la dirección de cada miembro antes de iniciar la comunicación	Comunicación dirigida a un receptor o receptores. Remitentes y receptores no tienen por qué existir en el mismo instante del envío.
Desacoplamiento espacial	El remitente no necesita conocer la identidad/dirección del receptor/es; el receptor/es debe existir en ese momento.	El remitente no necesita conocer la identidad/dirección del receptor/es; el remitente/es y el receptor/es no tienen por qué existir en el mismo instante del envío.

Según esta información aportada, podemos indicar las ventajas e inconvenientes del acoplamiento espacial y temporal.

	Ventajas	**Inconvenientes**
Acoplamiento espacial	- Comunicación confiable (uso de ACKs). - Menor consumo de recursos cuando no se utiliza un servidor intermediario (en los servidores y a nivel de red). - Menores latencias. - Integridad de los datos y validación de los mismos (en conexiones *peer-to-peer*).	- Más comunicaciones al generar más conexiones para la comunicación de mensajes. - Menos escalable cuando no se utiliza un servidor intermediario. - Se necesita un entorno homogéneo, sobre todo en la parte de los clientes. - Menor transparencia ante fallos (no se distingue si son de red o de servidor). Para solucionarlos implica implementar medidas a nivel de código. - No gestiona transacciones (o es muy costoso).

		- Se depende directamente de operaciones en memoria. - Posibles cuellos de botella la y un único punto crítico a fallos. - Problemas de escalabilidad.
Acoplamiento temporal	- La mayor parte de las ventajas que se producen en el acoplamiento espacial también se dan aquí, ya que son las que aportan los sistemas punto a punto. - Se garantiza la entrega del mensaje. - Se pueden aplicar sistemas de gestión/corrección de errores en el envío de los datos. - Se puede utilizar uno a uno, pero también uno a muchos. - Con el paradigma/sistema de comunicación grupal (según la implementación puede ser acoplado o desacoplado) se mejora el envío de mensajes (respecto al sistema de comunicación directa): grupos bien definidos, entrega ordenada de mensajes, tolerancia a fallos, gestión de múltiples remitentes…	- La mayor parte de los inconvenientes que se producen en el acoplamiento espacial también se dan aquí, ya que son las que se producen en los sistemas punto a punto. - Es necesario enviar los mensajes en un orden determinado. - Uso de excepciones en código para corregir errores.

Ejemplo 1: Acoplamiento espacial y acoplamiento temporal

El sistema operativo cliente Microsoft Windows utiliza diversos servicios internos mediante mensajes RPC, entre otros el servicio de fax, la cola de impresión, las configuraciones de las conexiones de red… Principalmente se utilizan porque es un sistema fiable y requiere de tiempos de procesamiento cortos. Todos estos servicios necesitan comunicarse con una parte servidora u otro cliente; en todos ellos es necesario conocer el destinatario (acoplamiento espacial) y ejecutarse cuando ambos están disponibles (acoplamiento temporal).

Ejemplo 2: Acoplamiento espacial y desacoplamiento temporal

Telegram[1] es principalmente una aplicación de mensajería (aunque tiene otras funcionalidades), para el envío masivo de mensajes. El servidor es de código propietario pero el cliente es de código abierto. El diseño de la arquitectura es totalmente *Cloud*, pero por ahora centralizada (están trabajando en un diseño descentralizado). Su funcionamiento, en general, es como el envío de mensajes por SMTP, utiliza una cola de mensajes (por lo que no tiene acoplamiento temporal) y una vez se comunican cliente-servidor se envía los mensajes mediante RPC (acoplamiento temporal).

Ejemplo 3: Desacoplamiento espacial y acoplamiento temporal

Spred[2] es una herramienta que proporciona un sistema de comunicación grupal en sistemas distribuidos, tolerante a fallos, escalable, alto rendimiento… Permite la garantía de entrega de los mensajes ordenados y también la integridad e los datos. Está diseñado para soportar comunicaciones asíncronas y permitir ser utilizado por aplicaciones de terceros para el envío de mensajes a grupos (desacoplamiento espacial); utiliza *multicast* para los procesos de comunicación (acoplamiento temporal).

[1] https://www.telegram.org/
[2] http://www.spread.org/

Ejemplo 4: Desacoplamiento espacial y desacoplamiento temporal

Lime[3] es una aplicación de gestión y coordinación basada en Linda que permite operar con un modelo de espacio compartido de *tuplas*. Se utilizan colas distribuidas (el emisor envía los mensajes a una cola de un servidor intermediario, que es accedida por los receptores), memoria distribuida compartida y espacios de *tuplas* (que son leídas por los consumidores). Por lo tanto, hay desacoplamiento temporal y espacial.

Referencias bibliográficas:

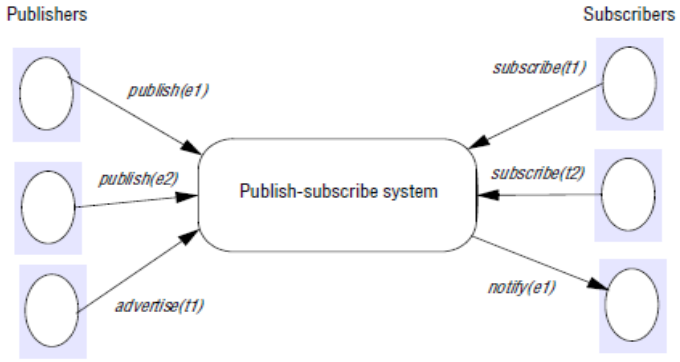
https://districted.wordpress.com/concepts/systems-and-architectures/distributed-systems/
https://people.kth.se/~johanmon/courses/id2201/lectures/indirect.pdf
http://www.inf.fu-berlin.de/inst/ag-se/teaching/V-NETZPR-2015/07_Indirect_Communication%20-%20I.pdf
https://es.slideshare.net/lara_ays/chapter-6-slides-15655567
https://kodu.ut.ee/~dumas/pubs/ccmiddleware.pdf

3

2. Explain the differences between publish/subscribe and message queue systems. Give two examples of free software projects for each one and briefly describe its characteristics.

Publish/subscribe

Un sistema de **publicación/subscripción** se basa en que los publicadores notifican/publican eventos estructurados en un servicio de eventos y los suscriptores se suscriben en éstos (ya que existe un interés determinado en recibirlos). Así, el sistema lo que hace es encontrar suscripciones para las publicaciones y asegurarse de que las notificaciones de los eventos llegan correctamente. Las dos características más notables son: es un sistema heterogéneo para el envío de mensajes y es asíncrono.



Aunque es el ejemplo "más simple/claro" de este tipo de sistemas, considero que es necesario indicarlo. Un ejemplo de uso del modelo *publish/subscribe* se encuentra en la aplicación "*Really Simple Syndication*" (RSS), donde los suscriptores pueden suscribirse a ciertos contenidos y recibir, de manera automática, la información actualizada de estos lugares sin tener que visitarlos uno a uno (basado principalmente en XML, permite el acceso a contenidos mediante lectores de RSS). Se trata de un proveedor de información que obtiene la información de diferentes vías (según las subscripciones que tengamos). Para cada nueva información, ésta es enviada a todos los suscriptores. La persona que tenga interés en un determinado tipo de noticia se suscribirá al evento deseado de forma que cada vez que se actualice le llegan las notificaciones. Se utiliza en muchas aplicaciones como sistemas de información financiera (p.ej. seguimiento de valores en bolsa), blogs, noticias, aplicaciones de monitorización, aplicaciones de trabajo colaborativo…

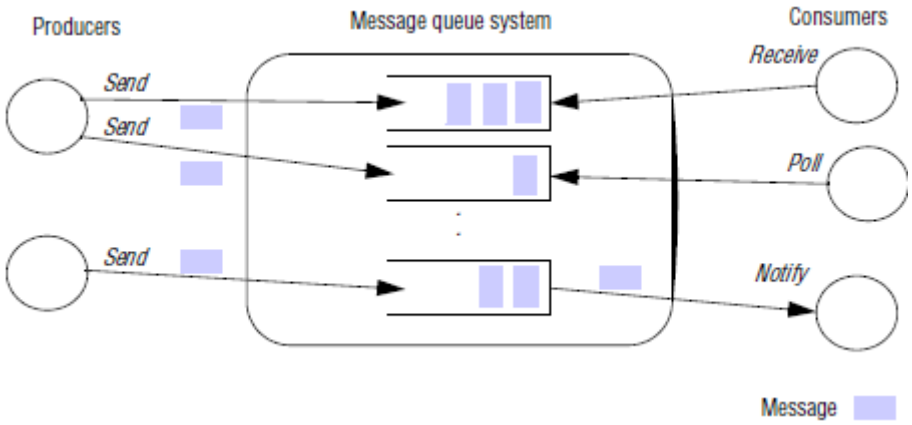
Ejemplo 1: Lector de feeds Awasu[4] (no es software libre, pero tiene versión gratuita, y me ha parecido muy interesante por todas las opciones que pueden llegar a implementar).
Se trata de un lector de *feeds*, principalmente de tipo RSS y Atom, aunque también permite crear *plugins*/complementos personalizados para recibir la información que se desee de un canal que no tenga implementado un *feed*. Tiene un gran número de características que además son personalizables: seguimiento de entradas leídas, elimina anuncios y contenidos no deseados, sincroniza con otros lectores de *feeds*, se pueden personalizar los canales, permite descarga automática de podcast, agrega características de seguridad…

Ejemplo 2: Lector de noticias RSS Bandit[5]
Este proyecto si se trata de una aplicación gratuita y de código libre, que permite leer tanto RSS como Atom, descargar *podcasts*, admite *Network News Transfer Protocol* (NNTP). Aunque el proyecto no está cerrado, solo se corrigen errores menores ya que la personas que lo gestionan así lo han decidido.

[4] https://awasu.com/
[5] https://rssbandit.org/ (actualmente el proyecto se encuentra en Github https://github.com/RssBandit/RssBandit)

Message queue

Un sistema de **colas de mensajes**, consiste en que un emisor envía mensajes/eventos a una cola (elemento intermedio en la comunicación indirecta) y el receptor consume los mensajes de la cola. En este sistema existe desacoplamiento espacial (no tienen por qué conocerse) y también desacoplamiento temporal (la acción de enviar el mensaje a la cola y de consumirlo, no tienen que ser simultáneos); sigue un funcionamiento de consumo similar al *publish/subscribe*.



Una cuestión importante es la persistencia, ya que, hasta que un mensaje no es utilizado por un receptor éste se queda en la cola, de modo que permite garantizar que la entrega del mensaje se realiza. Un ejemplo práctico es un sistema de compra por Internet; al realizar la compra se crea un evento para descontar el dinero de la cuenta del usuario. Este evento se envía a una cola, donde el banco correspondiente irá cogiendo las operaciones que hay en la misma, de manera que cuando llegue el evento descontará el importe de la cuenta.

Ejemplo 1: Plataforma de mensajería NSQ[6]
Se trata de una plataforma de mensajería distribuida en tiempo real que permite la gestión de millones de mensajes por día. Se basa en una topología distribuida y descentralizada, permite alta disponibilidad, escalado horizontal, implementa medidas de seguridad como TLS, envío de mensajes mediante *multicast*, aplica garantías en la entrega de los mensajes…

Ejemplo 2: Redis Simple Message Queue RSMQ[7]
Este proyecto permite implementar colas de mensajes muy ligeras para ser utilizadas por terceras aplicaciones. Se basa en Redis, por lo que se utiliza principalmente una estructura de datos en memoria para su funcionamiento. Entre otras características permite enviar miles de mensajes por segundo, puede utilizar un servidor Redis intermediario y así se utilizan RSMQs para aumentar el flujo de mensajes, se garantiza la entrega del mensaje, permite configurar tiempos de espera de los mensajes, se utiliza un sistema FIFO…

Referencias bibliográficas:
https://www.um.es/docencia/barzana/PRACTICAS/RSS-Google-Reader.html
https://www.actualidadgadget.com/como-subscribirse-a-un-feed-rss-y-que-es-un-feed-rss/
https://www.um.es/actualidad/rss/tut_sindicacion/index.php
https://awesomeopensource.com/projects/message-queue

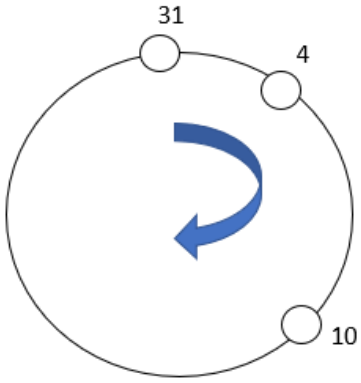
[6] https://nsq.io/
[7] https://github.com/smrchy/rsmq

3. Give the routing tables of a Chord overlay with nodes N4, N10, N31.

Chord es un protocolo sencillo y escalable de búsquedas distribuidas en redes P2P que relaciona claves (*keys*) con nodos y está diseñado para funcionar en redes descentralizadas (es decir, sin nodos privilegiados). Utiliza una topología en anillo (en el sentido de las agujas del reloj) con una función Hash, como SHA-1, para asignar las claves tanto a los recursos como a los *peers* mediante el hash de su dirección IP.

Las claves tienen un tamaño de "m" bits para alcanzar el número máximo de identificadores; se utiliza un rango de claves de 0 a 2^m-1. Está basado en Distributed Hash Tables (DHT), que son tablas en forma clave-valor, que suministran un índice para localizar los recursos.

Se considera que la red está formada por 2^m nodos, que pueden estar activos o ausentes de la red; el protocolo se encarga de asignar las claves a los nodos activos y mantener esas asignaciones dinámicamente, es decir, a medida que los nodos van entrando y saliendo de la red se van actualizando sobre los nodos existentes.



Cada nodo tiene y administra una tabla de rutas de “m” filas, *finger table*. En cada fila se almacena el valor del nodo sucesor al valor que se obtiene de sumar el identificador del nodo más 2^i, siendo “i” un valor desde 0 hasta m-1. Cada fila de la tabla contiene una referencia a un nodo situado cada vez más alejado. Datos a tener en cuenta:

a) “i”: identificador desde el que se buscará el nodo sucesor.
b) Nodo: identificador del nodo.
c) Cálculo: Su valor será $(n+2^{i-1}) \bmod 2^i$, con $1 \leq i \leq m$.
d) Intervalo: desde el identificador actual hasta el de la siguiente fila.
e) Sucesor: será el nodo sucesor.

La representación (de forma más visual y clara) podría ser:

i	NODO	CÁLCULO	INTERVALO	SUCESOR
$1 \leq i \leq m$	N_n	$(n+2^{i-1}) \bmod 2^i = k$	[k, k tal que i+1)	$N_n \geq k$

Según los datos proporcionados pueden participar 31 nodos. Por lo tanto, se necesitan m=5 bits para codificar el identificador (permitirá llegar a 32 identificadores, con un valor de 0 hasta 2^m-1, que es 31).

N4

i	NODO	CÁLCULO	INTERVALO	SUCESOR
1	N4	$4+2^0 = 5$	[5,6)	N10
2	N4	$4+2^1 = 6$	[6,8)	N10
3	N4	$4+2^2 = 8$	[8,12)	N10
4	N4	$4+2^3 = 12$	[1,.20)	N31
5	N4	$4+2^4 = 20$	[20,4)	N31

N10

i	NODO	CÁLCULO	INTERVALO	SUCESOR
1	N10	$10+2^0 = 11$	[11,12)	N31
2	N10	$10+2^1 = 12$	[12,14)	N31
3	N10	$10+2^2 = 14$	[14,18)	N31
4	N10	$10+2^3 = 18$	[18,26)	N31
5	N10	$10+2^4 = 26$	[26,10)	N31

N31

i	NODO	CÁLCULO	INTERVALO	SUCESOR
1	N31	$31+2^0 = 32 \bmod 32 = 0$	[0,1)	N4
2	N31	$31+2^1 = 33 \bmod 32 = 1$	[1,3)	N4
3	N31	$31+2^2 = 35 \bmod 32 = 3$	[3,7)	N4
4	N31	$31+2^3 = 39 \bmod 32 = 7$	[7,15)	N10
5	N31	$31+2^4 = 47 \bmod 32 = 15$	[15,31)	N31

Describe the process of looking up for key 30 from N4.

El proceso es el siguiente:

- En primer lugar, el nodo N4 comprueba si la clave está en su tabla, como no está busca la clave más cercana a la clave 30 pero por debajo (la clave más cercana sería la 20, que apunta al nodo N31).
- La búsqueda pasa al nodo N31 y vuelve a comprobar si se encuentra en su tabla. En este caso, el nodo N31 sí que la tiene y la búsqueda finaliza.
- El nodo N31 envía un mensaje al nodo que solicita la clave 30, que es N4, indicándole que es él, el nodo N31, quien tiene la clave 30.

What is the complexity of a lookup? What happens when the nodes are not uniformly distributed?

En Chord, la complejidad del proceso de búsqueda es de O(log(n)), siendo n el número de nodos de la red. Esto implica que sea un proceso de búsqueda más eficiente en comparación a otros sistemas, ya que se puede reducir enormemente el número de saltos entre nodos para cada búsqueda.

Cuando los nodos no se distribuyen de una manera uniforme, un gran número de nodos o gran parte de ellos, apuntan al mismo nodo sucesor. Si los nodos no se distribuyen de manera uniforme, la distancia entre un nodo y el correspondiente predecesor será mayor entre unos nodos que entre otros (se puede ver muy claramente y de forma gráfica en el anillo dibujado en este ejercicio, ya que se aproxima bastante en escala a los datos de este ejercicio; el nodo 31 tiene una porción predecesora mucho más grande que los otros nodos). Esta desigualdad provocará que los nodos que presentan mayores distancias respecto a sus predecesores deberán soportar una carga de trabajo más elevada en relación a los nodos donde esta distancia sea menor (tendrán más claves a su cargo). En este sentido, los nodos con más claves a su cargo almacenarán más información (tendrán más filas en su tabla), por lo que tardarán más en buscar claves dentro de su tabla, ya que esta será más grande que la de otros nodos; esta circunstancia, puede provocar puntos de congestión y cuellos de botella en el anillo.

Referencias bibliográficas:

https://es.wikipedia.org/wiki/Chord
https://en.wikipedia.org/wiki/Chord_(peer-to-peer)
http://homepage.divms.uiowa.edu/~ghosh/9.Chord.pdf
https://www.youtube.com/watch?v=Dlz1jMrrez8&ab_channel=slideshowthis

4. Describe the components of the BitTorrent architecture and how they help it to achieve its purposes. Argue for each of the following cases how would them affect to Bittorrent performance and/or scalability.

BitTorrent es un protocolo diseñado para el intercambio de archivos punto a punto (*peer-to-peer*) en Internet. Es uno de los protocolos más comunes para la transferencia de archivos de gran tamaño (como los ficheros multimedia). Existe un gran número de aplicaciones clientes que utilizan dicho protocolo para la descarga y compartición de archivos (conocidos como clientes Bittorrent), los cuales no pretenden ser una aplicación para la transmisión en tiempo real de contenido, sino para la descarga de los archivos y su posterior reproducción.

Los principales componentes que integran esta arquitectura, los cuales son necesarios y empleados en la transferencia de datos son:

- **Archivo .torrent**: contiene metadatos asociados con el fichero que se quiere obtener:
  - El nombre y la duración del archivo.
  - La ubicación de un *tracker* (especificado como una URL).
  - Una suma de control asociado a cada trozo, que se genera utilizando el algoritmo de hash SHA-1 (permite verificar el contenido para la siguiente descarga).
- **Chunk** (pedazo): una porción de tamaño fijo de un archivo determinado. Cada archivo completo se trocea en partes más pequeñas para poder operar con ellas de forma más fácil y dinámica; cada una de estas partes/trozos es un *chunk*.
- **Peers** (puntos): Se denomina así a todos los usuarios/participantes que integran la red. Ya sea *leecher* o *seeder*, se comparte siempre los *chunks* que se tenga.
- **Leechers** (sanguijuelas): son todos los usuarios que están en la red descargando o que quieren descargar un archivo pero que todavía no lo tienen completo; una vez tienen todos los trozos asociados con un archivo, puede convertirse en *seeders* para descargas posteriores de otros participantes.
- **Seeders** (semillas): son los usuarios de la red que poseen una versión completa de un archivo (todos los *chunks*); ponen la semilla inicial para la distribución de archivos.
- **Trackers** (rastreadores): servidor especial que contiene la información necesaria para que los *peers* se conecten unos con otros; contiene información sobre las descargas en curso. Inicialmente es la única forma de localizar qué usuarios contienen el archivo que se quiere descargar.
- **Torrent** (torrente o enjambre): conjunto de sitios/elementos involucrados en la descarga de un archivo. El enjambre son los usuarios en general que el *tracker* se encarga de buscar. Basado en esto, el *tracker* mantiene la información sobre el estado actual de descargas de un archivo determinado en función de los *seeders* y *leechers* asociados. Por lo tanto, estos 3 elementos y roles asociados en BitTorrent (*tracker*, *seeders* y *leechers*), se conocen como el torrente (o enjambre) para ese archivo.
- **Tit-for-tat** (ojo por ojo): mecanismo de incentivo que premia las descargas en BitTorrent a los usuarios que más comparten, priorizando más ancho de banda.
- **Optimistic unchoking** (desenganche optimista): mecanismo de reserva de ancho de banda para permitir que nuevos participantes puedan iniciar descargas.
- **Rarest first** (más raro primero): esquema de programación por el cual se prioriza los *chunks* menos compartidos.

a) Very large chunks.

Los clientes pueden descargar varios fragmentos en paralelo desde diferentes sitios. Lo que se trata es de utilizar trozos pequeños para poder ser compartidos de forma más rápida entre los participantes. Con esto se consigue reducir la carga en un sitio en particular para

repartirla entre todos los participantes. Si se utilizan *chunks* de tamaño muy grande, será más difícil de compartir los trozos. Por ejemplo, si un *leecher* tiene problemas en la red y se cancela la descarga, tendría que empezar a descargar el *chunk* completo. Además, un *seeder* estaría mucho tiempo realizando la función de compartir y los *leechers* no podrían compartir sus *chunks* con otros participantes rápidamente, disminuyendo la carga en el *seeder*, para que este comparta los *chunks* que aún no tiene ningún otro *peer*.

Por lo tanto, con un tamaño pequeño de *chunk* la red bittorrent es más escalable porque permite que más *peers* puedan compartir *chunks* en todo momento. Además, permite distribuir mejor la carga para que todos estén compartiendo en todo momento y que no haya unos pocos participantes que sean *seeders* y muchos participantes que sean *leechers*.

b) Selecting certain peers and assign more traffic to those.

El problema en este caso es similar al indicado anteriormente, cuando se utilizan trozos de archivos muy grandes. Lo principal es fomentar que todos participen en el torrente y que todos puedan recibir trozos y compartirlos. Si no se utilizan unos métodos que favorezcan estas premisas, el sistema es menos escalable y la carga será muy dispar (unos pocos mucha carga y muchos poca carga); por eso se utilizan técnicas como *optimistic unchoking*,

Si solamente unos pocos participantes tienen un gran ancho de banda, a muchos otros les quedará muy poco, por lo tanto, se conseguiría que unos pocos tengan el archivo completo pronto, pero que muchos otros tengan que esperar más tiempo. Este hecho implica que es menos escalable ya que muchos participantes no asumirán tener que esperar mucho tiempo. Además, el primer *seeder* tendrá una carga elevadísima y en cambio muchos participantes ni llegarán a ser *seeders* en mucho tiempo, ya que no obtendrán ningún *chunk* en bastante tiempo.

c) Removing tit-for-tat.

El término *tit-for-tat* (ojo por ojo) es utilizado a la hora de compartir ficheros en una red P2P, incluyendo las redes Bittorrent, a la hora de cooperar y compartir entre *peers*. De forma sencilla podría entenderse como un mecanismo de cooperación del tipo "yo te doy si tú me das". Esta característica se aplica a los usuarios que utilizan un cliente Bittorrent cuando se comparte el ancho de banda. Como el ancho de banda suele ser limitado, y el de subida aún más (con las nuevas líneas de fibra simétricas se podría decir que esta característica no aplica de la misma manera), se trata de garantizar la compartición de los ficheros (tráfico de subida para el que comparte) con los otros participantes. Esta característica premia a los participantes de los que recibimos datos; si ellos nos envían entonces se prioriza el envío sobre ellos. Por lo tanto, *tit-for-tat* trata de buscar un intercambio equitativo, pero también premiar a los que más comparten.

Si se elimina esta característica los clientes pueden aplicar una visión más egoísta, ya que cada vez que se descarguen lo que deseen, ya no les importaría tomar el rol de *seeder*, ya que en las próximas descargan podrán descargar igualmente (con un gran ancho de banda). A la larga, esto no sería un entorno escalable ya que los *seeders* pueden estar un tiempo compartiendo, pero no infinitamente; no fomentaría la participación activa. Por otro lado, si los *leechers* no toman el rol de *seeder* una vez han conseguido el fichero completo, todos estarán descargando de muy pocos *seeders*, lo que implica un rendimiento muy malo ya que su ancho de banda será repartido para muchos más usuarios.

Por el contrario, si la política *tit-for-tat* se volviera demasiado estricta, los nuevos participantes estarían en una enorme desventaja al no poder descargar (ya que aún no han podido compartir nada). Para evitar este tipo de situaciones se utiliza el mecanismo de *optimistic unchoking*, así cada cliente reserva determinado ancho de banda y lo aplica a participantes al azar, lo que permite conectarse a nuevos participantes con un "buen ancho de banda" y también para que los nuevos usuarios puedan entrar en este sistema de compartición.

Referencias bibliográficas:

https://es.wikipedia.org/wiki/BitTorrent
https://tv.unir.net/videos/10233/47/972/0/0/Arquitectura-P2P-El-caso-de-BitTorrent
https://es.slideshare.net/Heleniodearica/bit-torrent-48825798

5. Apache Kafka is a free software distributed messaging system for log processing. Read the following paper on Kafka and answer the following exercises.
http://notes.stephenholiday.com/Kafka.pdf

a) Discuss its main design and architectural characteristics. Explain what are "brokers", "consumers" and "producers" in this context.

A continuación, explico que es cada uno de los siguientes componentes, para entender mejor el diseño y la arquitectura de Kafka:

- **Producers (productores)**: es un cliente de Kafka que se encarga de publicar mensajes de forma optimizada (por ejemplo, envía datos y las comunicaciones no esperan ACKs).
- **Brokers (intermediarios)**: es cada uno de los nodos intermedios (instancia o servidor Kafka), que almacena los mensajes que gestiona el sistema y su principal función es la de intercomunicador de mensajes entre los productores y los consumidores (lo más común es que formen un *cluster* para mejoras de rendimiento, escalabilidad, alta disponibilidad…). Contiene la/s partición/es de uno o varios *topics*; únicamente un *broker* puede tener la partición principal de un *topic* y los demás tendrán una réplica (si cae el *broker* que tiene la partición principal, una réplica asume ese rol). Cada *broker* puede tener un rendimiento diferente según sus características hardware y la configuración que se le asigne. El *broker* con la partición principal recibe las peticiones de escritura de los mensajes de lo productores, determina el *offset*, aplica los *commit* en el almacenamiento y solicita la actualización de las réplicas.
  Cada vez que los productores envían mensajes al *broker*, este los agrupa al último segmento recibido con los mensajes recibidos previamente; una vez que el segmento cumple un requisito (según un tamaño fijado, pasado un tiempo fijado…) se pasa al almacenamiento; los consumidores solo podrán hacer uso de los mensajes cuando se hayan ubicado en disco.
- **Consumers (consumidores)**: es un cliente de Kafka que se encarga de consumir mensajes. Mediante el diseño de Kafka, permite consumir mensajes de una partición concreta, ya que, al conocer los *offset*, puede solicitar al *broker* el envío de los segmentos de una partición que le faltan (que pueden ser enviados de forma agrupa). Todo esto implica que se mejore el rendimiento en gran cantidad de aspectos (red, carga de los *brokers*…)
- **Otros componentes**
  Aunque en esta pregunta no se solicita que se describan más componentes, únicamente los voy a indicar brevemente para comprender mejor la arquitectura y el diseño de Kafka:
  - Zookeper: software/servicio gestor de coordinación para las aplicaciones distribuidas: es un servicio centralizado para la gestión de la configuración, realiza descubrimientos, actualiza los *brokers*, revisa el estado de las particiones…
  - Clúster: es una agrupación de *brokers*, para proveer de tolerancia a fallos, HA…
  - Mensaje: unidad de datos en Kafka, no tienen estructura fija, se pueden agrupar, formados por una clave y un valor, y una marca de tiempo.
  - Esquema: estructura opcional que se aplica a los mensajes, para "estandarizar" la forma de los mensajes (asegura una información en cada mensaje).
  - *Topic*: tema o categoría en la que se pueden agrupar los mensajes enviados por los productores, se utiliza un *stream* para enviarlos y funciona como un "contenedor".
  - Partición: es cada una de las partes en las que se puede dividir un *topic*; es una secuencia de mensajes fija y es la unidad de replicación entre los *brokers*. Es la principal figura en Kafka para aportar redundancia, escalabilidad horizontal, mejora de rendimiento…
  - Offset: es el identificador único que tiene cada mensaje (según la ubicación en una partición), esto permite no tener que depender de índices, terceros nodos/servicios…

En Kafka no es necesario tratar cada mensaje de forma individual, se utilizan "*topics*" (temas) para tratar registros de un mismo tipo (es personalizable). A grandes rasgos, se definen flujos de mensajes de un determinado tema; los productores de registros publican mensajes de un determinado tema y los envían a unos servidores intermedios llamados *brokers* (quedan almacenados en estos servidores intermedios), de esta forma, los consumidores se subscriben a uno o más temas de los *brokers* para consumir/extraer los mensajes.

El sistema Kafka soporta tanto comunicaciones punto a punto como un modelo de publicación/subscripción, aunque realmente cuando se le quita partido a este sistema es con las de tipo publicación/subscripción (permite a múltiples consumidores recibir su propia copia de un *topic* en paralelo). Esto a su vez permite que múltiples productores y múltiples consumidores puedan publicar y recibir mensajes al mismo tiempo.

A continuación, indico algunas características de sistemas de gestión de mensajes que no se consideran correctas, o que al menos en Kafka se utiliza otra opción para mejorar el sistema respecto a estos otros sistemas "tradicionales":

- Se basan en asegurar/garantizar la entrega a costa de sacrificar otras características muy importantes, cuando no pasaría nada si se perdiese algún tipo de mensaje entre millones de registros que se capturan.
- No permiten agrupar mensajes (se necesita una comunicación completa TCP/IP para cada mensaje).
- No permiten o no es sencillo enviar mensajes particionados a un grupo de máquinas; tienen un componente distribuido muy débil.
- Gestionan eficazmente las colas de consumo siempre y cuando no empiecen a almacenarse/encolarse mensajes (como en ingestas de muchos datos de ficheros almacenados); cuando es así su gestión es ineficaz.
- Se basan principalmente en sistemas "push", lo cual puede provocar desbordamiento de datos en las colas de las aplicaciones consumidoras (un modelo "pull" gestiona los registros a máximo rendimiento, pero sin desbordamientos, ya que se solicitan los datos y no se reciben de terceros).

Teniendo en cuenta todos estos datos, las mejoras en Kafka son muchas y notables: más eficiencia en la transmisión de datos (menos operaciones de conexión y más eficientes las que se realizan), fácilmente escalable horizontalmente, bajas latencias en la transmisión de datos, mayor rendimiento, no realiza accesos aleatorios a datos, realiza escritura secuencial en disco, aplica mecanismos para evitar pérdida de datos y tolerancia a fallos, es un sistema distribuido, replicación de información entre *brokers*, aplica sistemas para garantizar la integridad de los datos, persistencia de mensajes en sistema de ficheros…

b) Discuss how the design in the communication affects the throughput and how it affects to the capacity to process a larger volume of log data.

Considero que en parte ya he respondido esta pregunta en el punto anterior, tratando de describir la arquitectura/diseño de Kafka y lo que aporta respecto a otros modelos más "tradicionales". De todos modos, voy indicar unas cuantas características muy concretas que influyen directamente en el mejor rendimiento de la comunicación y como esas características ayudan a tratar volúmenes grandes de datos. Así, mediante el uso de *topics*, particiones y *offsets*, se consiguen principalmente las siguientes ventajas:

- Bajas latencias: permite mayor número de transmisiones en menos tiempo.

- Envío de más datos en cada conexión de red completa entre productor-*broker*-consumidor.
- Permite ingestas de grandes volúmenes de datos online y offline al utilizar un sistema *pull* y no *push* (trabaja en todo momento al máximo sin ser saturado por un tercero).
- No utiliza el mensaje como unidad de transferencia sino inicialmente la partición (más datos en cada transferencia) y en caso de ser necesario se envían mensajes en grupo según el último offset que se haya enviado (más granularidad a la hora de realizar un reenvío y ajustando exactamente los datos que faltan; no repite el envío de un mismo dato dos veces).
- Uso de *brokers* sin estado: los *brokers* no gestionan la información de lo que se consume, eso recae en el propio consumidor; si no se consume algo por un consumidor, el *broker* lo elimina pasado un tiempo fijado (esto no afecta a las grandes ingestas de datos ya que suele tener una configuración de retención de 7 días; tiempo de sobra para realizar grandes ingestas).
- Los envíos de datos de los productores a los *brokers* no esperan ACKs.
- El uso de un sistema de ficheros más eficiente (se utiliza la escritura secuencial y se evitan las lecturas/accesos aleatorios).

Referencias bibliográficas:

https://kafka.apache.org/documentation/#gettingStarted
https://enmilocalfunciona.io/aprendiendo-apache-kafka-parte-1/
https://enmilocalfunciona.io/aprendiendo-apache-kafka-parte-2-2/
https://enmilocalfunciona.io/aprendiendo-apache-kafka-parte-3-conceptos-basicos-extra/
https://aprenderbigdata.com/introduccion-apache-kafka/
https://www.ionos.es/digitalguide/servidores/know-how/que-es-apache-kafka/

ANEXO BIBLIOGRÁFICO

- Coulouris, George F. ...[et al.]. Distributed Systems: Concepts and Design, 5th Edition. Harlow [etc.] : Addison-Wesley/Pearson Education, cop. 2012. 1063 p.
  ISBN 9780273760597

- Kreps, J., Narkhede, N. & Rao, J. (2011). Kafka: a Distributed Messaging System for Log Processing. *Linkedin*. URL: http://notes.stephenholiday.com/Kafka.pdf

- **Referencias documentales indicadas específicamente en un apartado**:
  Se ha indicado únicamente la URL de los documentos leídos, pero no se ha indicado la referencia bibliográfica en el formato formal (por ejemplo, en formato APA), ya que han sido utilizados para aumentar el conocimiento sobre diversos conceptos, más que ser utilizados para realizar alguno de los apartados de los ejercicios propuestos.