las transacciones con él. Si los dos pares están satisfechos con el intercambio, se incluirán en sus respectivas listas de los cuatro principales y continuarán realizando intercambios hasta que uno de los pares encuentre un socio mejor. El efecto es que los pares capaces de suministrar datos a velocidades compatibles tienden a emparejarse. La selección aleatoria de vecinos también permite a los nuevos pares obtener fragmentos, con el fin de tener algo que intercambiar. Todos los demás pares vecinos excepto estos cinco (los cuatro pares "principales" más el par de prueba) están "filtrados", es decir, no reciben fragmentos de Alicia. BitTorrent dispone de una serie de interesantes mecanismos que no vamos a ver aquí, entre los que se incluyen la gestión de piezas (minifragmentos), el procesamiento en cadena, la selección aleatoria del primer fragmento, el modo *endgame* y el *anti-snubbing* [Cohen 2003].

El mecanismo de incentivos para intercambio que acabamos de describir a menudo se denomina *tit-for-tat* (toma y daca, una estrategia de la teoría de juegos) [Cohen 2003]. Se ha demostrado que este esquema de incentivos puede soslayarse maliciosamente [Liogkas 2006; Locher 2006; Piatek 2007]. No obstante, el ecosistema BitTorrent ha tenido un éxito bárbaro, con millones de pares simultáneos compartiendo activamente archivos en cientos de miles de torrentes. Si BitTorrent se hubiera diseñado sin la estrategia *tit-for-tat* (o una variante), y aunque todo el resto de características fueran las mismas, es posible que BitTorrent no existiera actualmente, ya que la mayor parte de los usuarios hubieran pretendido aprovecharse de los demás [Saroiu 2002].

Vamos a terminar esta exposición sobre P2P mencionando brevemente otra aplicación de P2P, las tablas hash distribuidas (DHT, *Distributed Hast Table*). Una tabla hash distribuida es una base de datos simple, diatribuyéndose los registros de la base de datos a través de los pares de un sistema P2P. Las DHT han sido ampliamente implementadas (por ejemplo, en BitTorrent) y han sido objeto de exhaustivas investigaciones. Proporcionamos una introducción a las mismas en una nota de vídeo disponible en el sitio web de acompañamiento.


Nota de vídeo

Introducción a las tablas hash distribuidas

2.6 Flujos de vídeo y redes de distribución de contenido

La distribución de flujos pregrabados de vídeo representa ya la mayor parte del tráfico en los ISP residenciales de Norteamérica. En concreto, sólo los servicios de Netflix y YouTube consumieron un asombroso 37% y 16%, respectivamente, del tráfico de los ISP residenciales en 2015 [Sandvine 2015]. En esta sección proporcionaremos un resumen del modo en que se implementan en la Internet de hoy en día los servicios más populares de flujos de vídeo. Como veremos, se implementan utilizando protocolos de nivel de aplicación y servidores que funcionan, en cierto modo, como una caché. En el Capítulo 9, dedicado a las redes multimedia, examinaremos más en detalle el tema del vídeo por Internet, así como otros servicios Internet de carácter multimedia.

2.6.1 Vídeo por Internet

En las aplicaciones con flujos de vídeo almacenado, el medio subyacente es un vídeo pregrabado, como por ejemplo una película, un programa de televisión, un evento deportivo en diferido o un vídeo pregrabado generado por el usuario (como los que se pueden ver comúnmente en YouTube). Estos vídeos pregrabados se almacenan en servidores, y los usuarios envían solicitudes a los servidores para ver los vídeos *a la carta*. Muchas empresas de Internet proporcionan hoy en día flujos de vídeo, como por ejemplo Netflix, YouTube (Google), Amazon y Youku.

Pero antes de entrar a analizar los flujos de vídeo, conviene primero echar un rápido vistazo al propio medio subyacente: el vídeo. Un vídeo es una secuencia de imágenes, que normalmente se visualizan a velocidad constante, por ejemplo de 24 o 30 fotogramas por segundo. Una imagen codificada digitalmente y no comprimida, consta de una matriz de píxeles, codificándose cada píxel mediante una serie de bits que representan la luminancia y el color. Una característica importante del vídeo es que se puede comprimir, sacrificando algo de calidad de imagen a cambio de reducir la tasa

de transmisión de bits. Los algoritmos comerciales de compresión existentes hoy en día permiten comprimir un vídeo prácticamente a cualquier tasa de bits que se desee. Por supuesto, cuanto mayor sea la tasa de bits, mayor será la calidad de la imagen y mejor será la experiencia global del usuario, en lo que a visualización se refiere.

Desde la perspectiva de las redes, quizá la característica más destacable del vídeo sea su alta tasa de bits. El vídeo comprimido para Internet suele requerir entre 100 kbps (para vídeo de baja calidad) y más de 3 Mbps (para flujos de películas de alta resolución); los flujos de vídeo en formato 4K prevén una tasa de bits superior a 10 Mbps. Esto se traduce en una enorme cantidad de tráfico y de almacenamiento, particularmente para vídeo de alta gama. Por ejemplo, un único vídeo a 2 Mbps con una duración de 67 minutos consumirá 1 gigabyte de almacenamiento y de tráfico. La medida de rendimiento más importante para los flujos de vídeo es, con mucho, la tasa de transferencia media de extremo a extremo. Para poder garantizar una reproducción continua, la red debe proporcionar a la aplicación de flujos de vídeo una tasa de transferencia media que sea igual o superior a la tasa de bits del vídeo comprimido.

También podemos usar la compresión para crear múltiples versiones del mismo vídeo, cada una con un nivel diferente de calidad. Por ejemplo, podemos usar la compresión para crear tres versiones del mismo vídeo, con tasas de 300 kbps, 1 Mbps y 3 Mbps. Los usuarios puede entonces decidir qué versión quieren ver, en función de su ancho de banda actualmente disponible. Los usuarios con conexiones Internet de alta velocidad podrían seleccionar la versión a 3 Mbps, mientras que los usuarios que vayan a ver el vídeo a través de un teléfono inteligente con tecnología 3G podrían seleccionar la versión a 300 kbps.

2.6.2 Flujos de vídeo HTTP y tecnología DASH

En los flujos multimedia HTTP, el vídeo simplemente se almacena en un servidor HTTP como un archivo normal, con un URL específico. Cuando un usuario quiere ver el vídeo, el cliente establece una conexión TCP con el servidor y emite una solicitud `GET` HTTP para dicho URL. El servidor envía entonces el archivo de vídeo dentro de un mensaje de respuesta HTTP, con la máxima velocidad que permitan los protocolos de red subyacentes y las condiciones existentes de tráfico. En el lado del cliente, los bytes se acumulan en un buffer de la aplicación cliente. En cuanto el número de bytes en el buffer sobrepasa un umbral predeterminado, la aplicación cliente comienza la reproducción: para ser concretos, la aplicación de flujos de vídeo extrae periódicamente fotogramas de vídeo del buffer de la aplicación cliente, descomprime los fotogramas y los muestra en la pantalla del usuario. De ese modo, la aplicación de flujos de vídeo muestra el vídeo al mismo tiempo que recibe y almacena en el buffer otros fotogramas, correspondientes a secuencias posteriores del vídeo.

Aunque los flujos HTTP, tal como se describen en el párrafo anterior, se han implantado ampliamente en la práctica (por ejemplo por parte de YouTube, desde su nacimiento), presentan una carencia fundamental: todos los clientes reciben la misma versión codificada del vídeo, a pesar de las grandes variaciones existentes en cuanto al ancho de banda disponible para cada cliente, e incluso en cuanto al ancho de banda disponible para un cliente concreto a lo largo del tiempo. Esto condujo al desarrollo de un nuevo tipo de flujos de vídeo basados en HTTP, una tecnología a la que se suele denominar **DASH, (*Dynamic Adaptive Streaming over HTTP,* Flujos dinámicos adaptativos sobre HTTP)**. En DASH, el vídeo se codifica en varias versiones diferentes, teniendo cada versión una tasa de bits distinta y, por tanto, un nivel de calidad diferente. El cliente solicita dinámicamente segmentos de vídeo de unos pocos segundos de duración. Cuando el ancho de banda disponible es grande, el cliente selecciona de forma natural los segmentos de una versión de alta tasa de bits; y cuando el ancho de banda disponible es pequeño, selecciona de forma natural los segmentos de una versión con menor tasa de bits. El cliente selecciona los segmentos de uno en uno mediante mensajes de solicitud GET HTTP [Akhshabi 2011].

DASH permite que los clientes con diferentes tasas de acceso a Internet reciban flujos de vídeo con tasas de codificación diferentes. Los clientes con conexiones 3G a baja velocidad pueden recibir una versión con baja tasa de bits (y baja calidad), mientras que los clientes con conexiones de fibra

pueden recibir una versión de alta calidad. DASH también permite a un cliente adaptarse al ancho de banda disponible, si el ancho de banda extremo a extremo varía durante la sesión. Esta característica es particularmente importante para los usuarios móviles, que suelen experimentar fluctuaciones del ancho de banda disponible a medida que se desplazan con respecto a las estaciones base.

Con DASH, cada versión del vídeo se almacena en un servidor HTTP, teniendo cada versión un URL distinto. El servidor HTTP dispone también de un **archivo de manifiesto**, que indica el URL de cada versión, junto con su correspondiente tasa de bits. El cliente solicita primero el archivo de manifiesto y determina cuáles son las diferentes versiones disponibles. Después solicita los segmentos de uno en uno, especificando un URL y un rango de bytes mediante un mensaje de solicitud GET HTTP para cada segmento. Mientras está descargando los segmentos, el cliente también mide el ancho de banda de recepción y ejecuta un algoritmo de determinación de la tasa de bits, para seleccionar el segmento que debe solicitar a continuación. Naturalmente, si el cliente tiene una gran cantidad de vídeo almacenado en el buffer y si el ancho de banda de recepción medido es grande, seleccionará un segmento correspondiente a una versión con alta tasa de bits. E igualmente, si tiene poca cantidad de vídeo almacenado en el buffer y el ancho de banda de recepción medido es pequeño, seleccionará un segmento correspondiente a una versión con baja tasa de bits. DASH permite, de ese modo, que el cliente cambie libremente entre distintos niveles de calidad.

2.6.3 Redes de distribución de contenido

Actualmente, muchas empresas de vídeo a través de Internet están distribuyendo a la carta flujos de vídeo de múltiples Mbps a millones de usuarios diariamente. YouTube, por ejemplo, con una librería de cientos de millones de vídeos, distribuye a diario centenares de millones de flujos de vídeo a usuarios repartidos por todo el mundo. Enviar todo este tráfico a ubicaciones de todo el mundo, al mismo tiempo que se proporciona una reproducción continua y una alta interactividad, constituye claramente un auténtico desafío.

Para una empresa de vídeo a través de Internet, quizá la solución más directa para proporcionar un servicio de flujos de vídeo sea construir un único centro de datos masivo, almacenar todos los vídeos en ese centro de datos y enviar los flujos de vídeo directamente desde el centro de datos a clientes repartidos por todo el mundo. Pero esta solución tiene tres problemas principales. En primer lugar, si el cliente está lejos del centro de datos, los paquetes que viajan desde el servidor al cliente atravesarán muchos enlaces de comunicaciones y probablemente atraviesen muchos ISP, estando algunos de los ISP posiblemente ubicados en continentes distintos. Si uno de esos enlaces proporciona una tasa de transferencia inferior a la velocidad a la que se consume el vídeo, la tasa de transferencia extremo a extremo también será inferior a la tasa de consumo, lo que provocará molestas congelaciones de la imagen de cara al usuario. (Recuerde del Capítulo 1 que la tasa de transferencia extremo a extremo de un flujo de datos está determinada por la tasa de transferencia del enlace que actúe como cuello de botella.) La probabilidad de que esto suceda se incrementa a medida que aumenta el número de enlaces que componen la ruta extremo a extremo. Una segunda desventaja es que un vídeo muy popular será probablemente enviado muchas veces a través de los mismos enlaces de comunicaciones. Esto no solo hace que se desperdicie ancho de banda de la red, sino que la propia empresa de vídeo a través de Internet estará pagando a su ISP proveedor (conectado al centro de datos) por enviar los *mismos* bytes una y otra vez a Internet. Un tercer problema de esta solución es que un único centro de datos representa un punto único de fallo, si se caen el centro de datos o sus enlaces de conexión con Internet, la empresa no podrá distribuir *ningún* flujo de vídeo.

Para poder afrontar el desafío de distribuir cantidades masivas de datos de vídeo a usuarios dispersos por todo el mundo, casi todas las principales empresas de flujos de vídeo utilizan **redes de distribución de contenido** (**CDN, *Content Distribution Network***). Una CDN gestiona servidores situados en múltiples ubicaciones geográficamente distribuidas, almacena copias de los vídeos (y de otros tipos de contenido web, como documentos, imágenes y audio) en sus servidores y trata de dirigir cada solicitud de usuario a una ubicación de la CDN que proporcione la mejor experiencia de usuario posible. La CDN puede ser una **CDN privada**, es decir, propiedad del

propio proveedor de contenido; por ejemplo, la CDN de Google distribuye vídeos de YouTube y otros tipos de contenido. Alternativamente, la CDN puede ser una **CDN comercial** que distribuya contenido por cuenta de múltiples proveedores de contenido; Akamai, Limelight y Level-3, por ejemplo, operan redes CDN comerciales. Un resumen muy legible de las redes CDN modernas es [Leighton 2009; Nygren 2010].

Las redes CDN adoptan normalmente una de dos posibles filosofías de colocación de los servidores [Huang 2008]:

- **Introducción profunda.** Una de las filosofías, de la que Akamai fue pionera, consiste en *introducirse en profundidad* en las redes de acceso de los proveedores de servicios Internet (ISP), implantando clústeres de servidores en proveedores ISP de acceso por todo el mundo. (Las redes de acceso se describen en la Sección 1.3.) Akamai ha adoptado esta solución con clústeres de servidores en aproximadamente 1.700 ubicaciones. El objetivo es acercarse a los usuarios finales, mejorando así el retardo percibido por el usuario y la tasa de transferencia por el procedimiento de reducir el número de enlaces y de routers existentes entre el usuario final y el servidor CDN del que recibe el contenido. Debido a este diseño altamente distribuido, la tarea de mantener y gestionar los clústeres se convierte en todo un desafío.
- **Atraer a los ISP.** Una segunda filosofía de diseño, adoptada por Limelight y muchos otras empresas de redes CDN, consiste en *atraer a los ISP* construyendo grandes clústeres en un número más pequeño (por ejemplo, unas decenas) de lugares. En lugar de introducirse en los ISP de acceso, estas CDN suelen colocar sus clústeres en puntos de intercambio Internet (IXP, *Internet Exchange Point*, véase la Sección 1.3). Comparada con la filosofía de diseño basada en la introducción profunda, el diseño basado en atraer a los ISP suele tener menores costes de mantenimiento y gestión, posiblemente a cambio de un mayor retardo y una menor tasa de transferencia para los usuarios finales.

Una vez implantados los clústeres, la CDN replica el contenido entre todos ellos. La red CDN puede no siempre almacenar una copia de cada vídeo en cada clúster, ya que algunos vídeos solo se visualizan en raras ocasiones o solo son populares en ciertos países. De hecho, muchas CDN no copian activamente los vídeos en sus clústeres, sino que usan una estrategia simple: si un cliente solicita un vídeo de un clúster que no lo tiene almacenado, el clúster extrae el vídeo (de un repositorio central o de otro clúster) y almacena una copia localmente, al mismo tiempo que envía el flujo de vídeo al cliente. De forma similar a lo que ocurre con las cachés web (véase la Sección 2.2.5), cuando el espacio de almacenamiento del clúster se llena, el clúster elimina los vídeos que no son solicitados frecuentemente.

Funcionamiento de una red CDN

Habiendo identificado las dos soluciones principales de implantación de una CDN, profundicemos en el modo en que una de estas redes opera. Cuando se indica al navegador del host de un usuario que extraiga un vídeo concreto (identificado mediante un URL), la CDN debe interceptar la solicitud para (1) determinar un clúster de servidores de la CDN que resulte adecuado para ese cliente en ese preciso instante y (2) redirigir la solicitud del cliente a un servidor situado en dicho clúster. En breve veremos cómo puede la CDN determinar un clúster adecuado. Pero antes, examinemos la mecánica del proceso de interceptación y redirección de una solicitud.

La mayoría de las CDN aprovechan DNS para interceptar y redirigir las solicitudes; un análisis interesante de esa utilización de DNS es [Vixie 2009]. Consideremos un ejemplo simple para ilustrar el modo en que DNS suele participar. Suponga que un proveedor de contenido, NetCinema, utiliza a una empresa proveedora de servicios CDN, KingCDN, para distribuir sus vídeos a sus clientes. En las páginas web de NetCinema, cada uno de los vídeos tiene asignado un URL que incluye la cadena "video" y un identificador unívoco del propio vídeo; por ejemplo, a *Transformers 7* se le podría asignar http://video.netcinema.com/6Y7B23V. Entonces se sucederán seis pasos, como se muestra en la Figura 2.25:

CASO DE ESTUDIO

INFRAESTRUCTURA DE RED DE GOOGLE

Para dar soporte a su amplia variedad de servicios en la nube —incluyendo las búsquedas, Gmail, calendarios, vídeos YouTube, mapas, documentos y redes sociales—, Google ha implantado una amplia red privada y una infraestructura CDN. La infraestructura CDN de Google tiene tres niveles de clústeres de servidores:

- Catorce "mega-centros de datos" (ocho en Norteamérica, cuatro en Europa y dos en Asia [Google Locations 2016]), teniendo cada centro de datos del orden de 100.000 servidores. Estos mega-centros de datos se encargan de servir contenido dinámico (y a menudo personalizado), incluyendo resultados de búsqueda y mensajes Gmail.
- Unos 50 clústeres en IXP dispersos por todo el mundo, consistiendo cada clúster en unos 100-500 servidores [Adhikari 2011a]. Estos clústeres son responsables de servir contenido estático, incluyendo vídeos de YouTube [Adhikari 2011a].
- Varios cientos de clústeres de "introducción profunda", ubicados dentro de proveedores ISP de acceso. En este caso, los clústeres suelen estar compuestos por decenas de servidores, situados en un mismo bastidor. Estos servidores de introducción profunda se encargan de la división TCP (véase la Sección 3.7) y de servir contenido estático [Chen 2011], incluyendo las partes estáticas de las páginas web donde se insertan los resultados de búsqueda.

Todos estos centros de datos y clústeres de servidores están conectados en red mediante la propia red privada de Google. Cuando un usuario hace una búsqueda, a menudo la búsqueda se envía primero a través del ISP local hasta una caché cercana de introducción profunda, de donde se extrae el contenido estático; mientras se proporciona el contenido estático al cliente, la caché cercana reenvía también la consulta a través de la red privada de Google hasta uno de los mega-centros de datos, de donde se extraen los resultados de búsqueda personalizados. Para un vídeo de YouTube, el propio vídeo puede provenir de una de las cachés en los IXP, mientras que parte de la página web que rodea al vídeo puede provenir de la caché cercana de introducción profunda y los anuncios que rodean al vídeo vienen de los centros de datos. Resumiendo: salvo por lo que se refiere al ISP local, los servicios en la nube de Google son proporcionados, en buena medida, por una infraestructura de red que es independiente de la Internet pública.

1. El usuario visita la página web en NetCinema.
2. Cuando el usuario hace clic sobre el vínculo http://video.netcinema.com/6Y7B23V, el host del usuario envía una solicitud DNS preguntando por video.netcinema.com.
3. El servidor DNS local del usuario (al que llamaremos LDNS) retransmite la solicitud DNS a un servidor DNS autoritativo de NetCinema, que observa la cadena "video" en el nombre de host video.netcinema.com. Para "transferir" la consulta DNS a KingCDN, lo que hace el servidor DNS autoritativo de NetCinema es, en vez de devolver una dirección IP, enviar al LDNS un nombre de host perteneciente al dominio de KingCDN, como por ejemplo a1105.kingcdn.com.
4. A partir de ese punto, la consulta DNS entra en la infraestructura DNS privada de KingCDN. El LDNS del usuario envía entonces una segunda consulta, preguntando ahora por a1105.kingcdn.com, y el sistema DNS de KingCDN termina por devolver al LDNS las direcciones IP de un servidor de contenido de KingCDN. Es por tanto aquí, dentro del sistema DNS de KingCDN, donde se especifica el servidor CDN desde el cual recibirá el cliente su contenido.
5. El LDNS reenvía al host del usuario la dirección IP del nodo CDN encargado de servir el contenido.

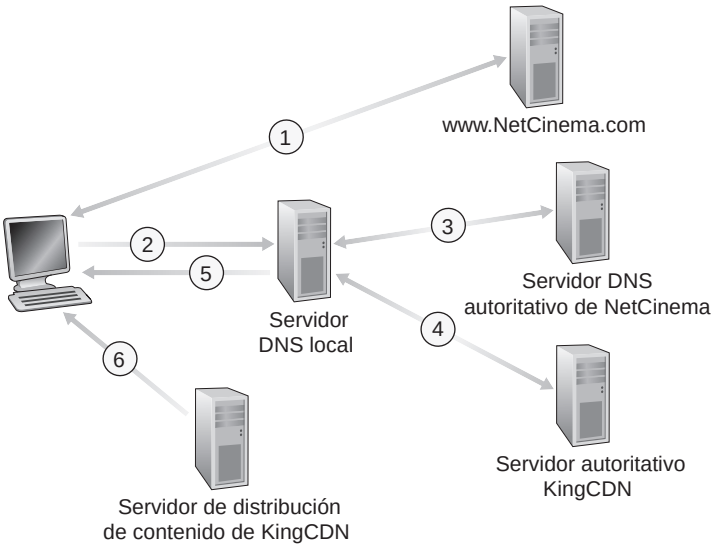

Figura 2.25 ♦ DNS redirige una solicitud de usuario hacia un servidor CDN.

6. Una vez que el cliente recibe la dirección IP de un servidor de contenido de KingCDN, establece una conexión TCP directa con el servidor situado en dicha dirección IP y transmite una solicitud GET HTTP para el vídeo deseado. Si se utiliza DASH, el servidor enviará primero al cliente un archivo de manifiesto con una lista de URL, uno para cada versión del vídeo, y el cliente seleccionará dinámicamente segmentos de las distintas versiones.

Estrategias de selección de clústeres

Uno de los fundamentos de cualquier implantación de una red CDN es la **estrategia de selección de clústeres**, es decir, el mecanismo para dirigir a los clientes dinámicamente hacia un clúster de servidores o un centro de datos pertenecientes a la CDN. Como acabamos de ver, la CDN determina la dirección IP del servidor LDNS del cliente a partir de la búsqueda DNS realizada por el cliente. Después de determinar esta dirección IP, la red CDN necesita seleccionar un clúster apropiado, dependiendo de dicha dirección. Las redes CDN emplean, generalmente, estrategias propietarias de selección de clústeres. Vamos a ver brevemente algunas soluciones, cada una de las cuales tiene sus ventajas y sus desventajas.

Una estrategia sencilla consiste en asignar al cliente al clúster **geográficamente más próximo**. Usando bases de datos comerciales de geolocalización (como Quova [Quova 2016] y MaxMind [MaxMind 2016]), la dirección IP de cada LDNS se hace corresponder con una ubicación geográfica. Cuando se recibe una solicitud DNS de un LDNS concreto, la CDN selecciona el clúster geográficamente más próximo, es decir, el clúster situado a menos kilómetros "a vuelo de pájaro" del LDNS. Una solución de este estilo puede funcionar razonablemente bien para una gran parte de los clientes [Agarwal 2009]. Sin embargo, para algunos clientes la solución puede proporcionar un mal rendimiento, porque el clúster geográficamente más cercano no es necesariamente el clúster más próximo en términos de la longitud o el número de saltos de la ruta de red. Además, un problema inherente a todas las soluciones basadas en DNS es que algunos usuarios finales están configurados para usar servidores LDNS remotos [Shaikh 2001; Mao 2002], en cuyo caso la ubicación del LDNS puede estar lejos de la del cliente. Además, esta estrategia tan simple no tiene en cuenta la variación a lo largo del tiempo del retardo y del ancho de banda disponible de las rutas en Internet, asignando siempre el mismo clúster a cada cliente concreto.

Para determinar el mejor clúster para un cliente basándose en las condiciones *actuales* de tráfico, las redes CDN pueden, alternativamente, realizar periódicamente **medidas en tiempo real** del retardo y del comportamiento de pérdidas entre sus clústeres y los clientes. Por ejemplo, una CDN puede hacer que todos sus clústeres envíen periódicamente mensajes de sondeo (por ejemplo, mensajes ping o consultas DNS) a todos los LDNS de todo el mundo. Una desventaja de esta técnica es que muchos LDNS están configurados para no responder a tales mensajes de sondeo.

2.6.4 Casos de estudio: Netflix, YouTube y Kankan

Terminamos nuestra exposición sobre los flujos de vídeo almacenados echando un vistazo a tres implantaciones a gran escala de enorme éxito: Netflix, YouTube y Kankan. Veremos que cada uno de estos sistemas adopta una solución muy diferente, aunque todos ellos emplean muchos de los principios subyacentes expuestos en esta sección.

Netflix

Netflix, que generó el 37% del tráfico de bajada en los ISP residenciales de Norteamérica en 2015, se ha convertido en el principal proveedor de servicios para películas y series de TV en línea en los Estados Unidos [Sandvine 2015]. Como vamos a ver, la distribución de vídeo de Netflix tiene dos componentes principales: la nube de Amazon y su propia infraestructura CDN privada.

Netflix dispone de un sitio web que se encarga de gestionar numerosas funciones, incluyendo los registros e inicios de sesión de los usuarios, la facturación, el catálogo de películas que puede hojearse o en el que se pueden realizar búsquedas y un sistema de recomendación de películas. Como se muestra en la Figura 2.26, este sitio web (y sus bases de datos *back-end* asociadas) se ejecuta enteramente en servidores de Amazon, dentro de la nube de Amazon. Además, la nube de Amazon se encarga de las siguientes funciones críticas:

- ***Ingesta de contenidos.*** Antes de que Netflix pueda distribuir una película a sus usuarios, debe primero realizar la ingesta y procesar la película. Netflix recibe versiones maestras de estudio de las películas y las carga en hosts situados en la nube de Amazon.

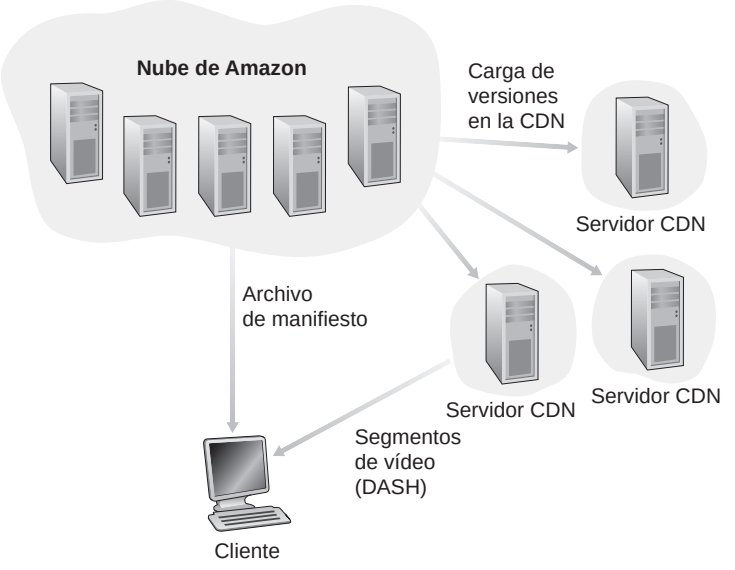

Figura 2.26 ♦ Plataforma de flujos de vídeo de Netflix.

- ***Procesamiento del contenido.*** Las máquinas de la nube de Amazon generan muchos formatos distintos para cada película, adecuados para una amplia variedad de clientes de reproducción de vídeo que se ejecutan en computadoras de sobremesa, teléfonos inteligentes y consolas de juegos conectadas a televisiones. Para cada uno de estos formatos se crea una versión diferente, con múltiples tasas de bits, lo que permite un envío adaptativo de los flujos multimedia a través de HTTP, usando DASH.
- ***Carga de las versiones en su CDN.*** Una vez generadas todas las versiones de una película, los hosts de la nube de Amazon cargan esas versiones en la CDN de Netflix.

Cuando Netflix inauguró su servicio de distribución de flujos de vídeo en 2007, empleaba tres empresas proveedoras de servicios de red CDN para distribuir su contenido de vídeo. Desde entonces, Netflix ha creado su propia CDN privada, desde la cual distribuye ahora todos sus flujos de vídeo. (Sin embargo, Netflix sigue usando Akamai para distribuir sus páginas web.) Para crear su propia CDN, Netflix ha instalado bastidores de servidores tanto en IXP como dentro de los propios ISP residenciales. Netflix dispone actualmente de bastidores de servidores en más de 50 ubicaciones de IXP; en [Netflix Open Connect 2016] puede ver una lista actualizada de IXP que albergan bastidores de Netflix. También hay centenares de ubicaciones de ISP que albergan bastidores de Netflix; véase también [Netflix Open Connect 2016], donde Netflix proporciona a los potenciales ISP asociados instrucciones sobre cómo instalar un bastidor Netflix (gratuito) para sus redes. Cada servidor del bastidor dispone de varios puertos Ethernet a 10 Gbps y de más de 100 terabytes de almacenamiento. El número de servidores de un bastidor es variable: las instalaciones de los IXP suelen tener decenas de servidores y contienen toda la biblioteca de flujos de vídeo de Netflix, incluyendo las múltiples versiones de los vídeos que hacen falta para soportar DASH; los ISP locales pueden disponer de un único servidor y almacenar solo los vídeos más populares. Netflix no utiliza un sistema de caché que se rellena bajo demanda (*pull-caching*) para cargar el contenido en sus servidores CDN ubicados en los IXP e ISP. En lugar de ello, Netflix realiza la distribución cargando activamente los vídeos en sus servidores CDN durante las horas menor tráfico. Para aquellas ubicaciones que no pueden almacenar la biblioteca completa, Netflix carga solo los vídeos más populares, que se determinan diariamente. El diseño de CDN de Netflix se describe con un cierto grado de detalle en los vídeos de Youtube [Netflix Video 1] y [Netflix Video 2].

Habiendo descrito los componentes de la arquitectura Netflix, examinemos más detalladamente la interacción entre el cliente y los distintos servidores implicados en la distribución de las películas. Como hemos dicho anteriormente, las páginas web a través de las que se explora la videoteca de Netflix se sirven desde servidores situados en la nube de Amazon. Cuando un usuario selecciona una película para reproducirla, el software de Netflix, ejecutándose en la nube de Amazon, determina primero cuáles de sus servidores CDN disponen de una copia de la película. A continuación, el software determina cuál de entre los servidores que disponen de la película es el "mejor" para esa solicitud del cliente. Si el cliente está utilizando un ISP residencial que tiene instalado un bastidor de servidores de la CDN de Netflix, y si ese bastidor dispone de una copia de la película solicitada, entonces suele seleccionarse un servidor de ese bastidor. Si no, lo que se suele seleccionar es un servidor de algún IXP cercano.

Una vez que Netflix ha determinado el servidor CDN que tiene que distribuir el contenido, envía al cliente la dirección IP de ese servidor concreto, junto con un archivo de manifiesto, que contiene los URL de las diferentes versiones de la película solicitada. A continuación, el cliente y ese servidor CDN interaccionan directamente, usando una versión propietaria de DASH. Específicamente, como se describe en la Sección 2.6.2, el cliente usa la cabecera de rango de bytes de los mensajes de solicitud GET HTTP para solicitar segmentos de las diferentes versiones de la película. Netflix usa segmentos de una duración aproximada de cuatro segundos Adhikari 2012]. Mientras se descargan los segmentos, el cliente mide la tasa de transferencia de recepción y ejecuta un algoritmo de determinación de la velocidad para identificar la calidad del segmento que debe solicitar a continuación.

Netflix utiliza muchos de los principios básicos que hemos expuesto anteriormente en esta sección, incluyendo los flujos adaptativos y la distribución a través de una CDN. Sin embargo, como Netflix emplea su propia CDN privada, que solo distribuye vídeo (y no páginas web), Netflix ha sido capaz de simplificar y adaptar su diseño de CDN. En particular, Netflix no necesita usar la redirección DNS, explicada en la Sección 2.6.3, para conectar un cliente concreto con un servidor CDN; en lugar de ello, el software de Netflix (que se ejecuta en la nube de Amazon) instruye directamente al cliente para que utilice un servidor CDN concreto. Además, la CDN de Netflix carga el contenido de la caché en sus servidores en momentos planificados (*push-caching*), durante las horas de menor tráfico, en lugar de cargarlo dinámicamente a medida que se producen fallos de localización en caché.

YouTube

Con 300 horas de vídeo cargadas en YouTube cada minuto y varios miles de millones de reproducciones diarias de vídeo [YouTube 2016], YouTube es, sin lugar a dudas, el mayor sitio de compartición de vídeos del mundo. YouTube comenzó a prestar servicio en abril de 2005 y fue adquirido por Google en noviembre de 2006. Aunque el diseño y los protocolos de Google/YouTube son propietarios, podemos hacernos una idea básica de cómo opera YouTube gracias a diversos trabajos de medida independientes [Zink 2009; Torres 2011; Adhikari 2011a]. Al igual que Netflix, YouTube hace un amplio uso de la tecnología CDN para distribuir sus vídeos [Torres 2011]. De forma similar a Netflix, Google utiliza su propia CDN privada para distribuir los vídeos de YouTube y ha instalado clústeres de servidores en muchos cientos de ubicaciones IXP e ISP distintas. Google distribuye los vídeos de YouTube desde estas ubicaciones y directamente desde sus inmensos centros de datos [Adhikari 2011a]. A diferencia de Netflix, sin embargo, Google emplea *pull-caching* y un mecanismo de redirección DNS, como se describe en la Sección 2.6.3. La mayoría de las veces, la estrategia de selección de clústeres de Google dirige al cliente hacia el clúster que tenga un menor RTT entre el clúster y el cliente; sin embargo, para equilibrar la carga entre los clústeres, en ocasiones se dirige al cliente (a través de DNS) a un clúster más distante [Torres 2011].

YouTube emplea flujos HTTP, ofreciendo a menudo un pequeño número de versiones distintas de cada vídeo, cada una con diferente tasa de bits y, correspondientemente, un diferente nivel de calidad. YouTube no utiliza flujos adaptativos (como DASH), sino que exige al cliente que seleccione una versión de forma manual. Para ahorrar ancho de banda y recursos de servidor, que se desperdiciarían en caso de que el usuario efectúe un reposicionamiento o termine la reproducción anticipadamente, YouTube emplea la solicitud de rango de bytes de HTTP para limitar el flujo de datos transmitidos, después de precargar una cierta cantidad predeterminada de vídeo.

Cada día se cargan en YouTube varios millones de vídeos. No solo se distribuyen a través de HTTP los flujos de vídeo de YouTube de los servidores a los clientes, sino que los usuarios que cargan vídeos en YouTube desde el cliente hacia el servidor también los cargan a través de HTTP. YouTube procesa cada vídeo que recibe, convirtiéndolo a formato de vídeo de YouTube y creando múltiples versiones con diferentes tasas de bits. Este procesamiento se realiza enteramente en los centros de datos de Google (véase el caso de estudio sobre la infraestructura de red de Google en la Sección 2.6.3).

Kankan

Acabamos de ver cómo una serie de servidores dedicados, operados por redes CDN privadas, se encargan de distribuir a los cliente los vídeos de Netflix y YouTube. Ambas empresas tienen que pagar no solo por el hardware del servidor, sino también por el ancho de banda que los servidores usan para distribuir los vídeos. Dada la escala de estos servicios y la cantidad de ancho de banda que consumen, ese tipo de implantación de una CDN puede ser costoso.

Concluiremos esta sección describiendo un enfoque completamente distinto para la provisión a gran escala de vídeos a la carta a través de Internet - un enfoque que permite al proveedor del servicio reducir significativamente sus costes de infraestructura y de ancho de banda. Como el lector estará suponiendo, este enfoque utiliza distribución P2P en lugar de (o además de) distribución cliente-servidor. Desde 2011, Kankan (cuyo propietario y operador es Xunlei) ha estado implantando con gran éxito su sistema P2P de distribución de vídeo, que cuenta con decenas de millones de usuarios cada mes [Zhang 2015].

A alto nivel, los flujos de vídeo P2P son muy similares a la descarga de archivos con BitTorrent. Cuando uno de los participantes quiere ver un vídeo, contacta con un *tracker* para descubrir otros homólogos en el sistema que dispongan de una copia de ese vídeo. El homólogo solicitante pide entonces segmentos de vídeo en paralelo a todos los homólogos que dispongan de él. Sin embargo, a diferencia de lo que sucede con la descarga en BitTorrent, las solicitudes sea realizan preferentemente para segmentos que haya que reproducir en un futuro próximo, con el fin de garantizar una reproducción continua [Dhungel 2012].

Recientemente, Kankan ha efectuado la migración a un sistema de flujos de vídeo híbrido CDN-P2P [Zhang 2015]. Específicamente, Kankan tiene ahora implantados unos pocos cientos de servidores en China y carga de forma activa contenido de vídeo en esos servidores. Esta CDN de Kankan juega un papel principal durante la etapa inicial de transmisión de los flujos de vídeo. En la mayoría de los casos, el cliente solicita el principio del contenido a los servidores de la CDN y en paralelo pide contenido a los homólogos. Cuando el tráfico P2P total es suficiente para la reproducción de vídeo, el cliente deja de descargar de la CDN y descarga sólo de los homólogos. Pero si el tráfico de descarga de flujos de vídeo P2P pasa a ser insuficiente, el cliente restablece las conexiones con la CDN y vuelve al modo de flujos de vídeo híbrido CDN-P2P. De esta manera, Kankan puede garantizar retardos iniciales de arranque cortos, al mismo tiempo que minimiza la utilización de ancho de banda y de una costosa infraestructura de servidores.

2.7 Programación de sockets: creación de aplicaciones de red

Ahora que hemos examinado una serie de importantes aplicaciones de red, vamos a ver cómo se escriben en la práctica los programas de aplicaciones de redes. Recuerde de la Sección 2.1 que muchas aplicaciones de red están compuestas por una pareja de programas (un programa cliente y un programa servidor) que residen en dos sistemas terminales distintos. Cuando se ejecutan estos dos programas, se crean un proceso cliente y un proceso servidor, y estos dos procesos se comunican entre sí leyendo y escribiendo en sockets. Cuando se crea una aplicación de red, la tarea principal del desarrollador es escribir el código para los programas cliente y servidor.

Existen dos tipos de aplicaciones de red. Uno de ellos es una implementación de un estándar de protocolo definido en, por ejemplo, un RFC o algún otro documento relativo a estándares. Para este tipo de implementaciones, los programas cliente y servidor deben adaptarse a las reglas dictadas por ese RFC. Por ejemplo, el programa cliente podría ser una implementación del lado del cliente del protocolo HTTP, descrito en la Sección 2.2 y definido explícitamente en el documento RFC 2616; de forma similar, el programa servidor podría ser una implementación del protocolo de servidor HTTP, que también está definido explícitamente en el documento RFC 2616. Si un desarrollador escribe código para el programa cliente y otro desarrollador independiente escribe código para el programa servidor y ambos desarrolladores siguen cuidadosamente las reglas marcadas en el RFC, entonces los dos programas serán capaces de interoperar. Ciertamente, muchas de las aplicaciones de red actuales implican la comunicación entre programas cliente y servidor que han sido creados por desarrolladores independientes (por ejemplo, un navegador Google Chrome comunicándose con un servidor web Apache, o un cliente BitTorrent comunicándose con un tracker BitTorrent).

El otro tipo de aplicación de red son las aplicaciones propietarias. En este caso, el protocolo de la capa de aplicación utilizado por los programas cliente y servidor *no* tiene que cumplir necesariamente ninguna recomendación RFC existente. Un único desarrollador (o un equipo de desarrollo) crea tanto el programa cliente como el programa servidor, y ese desarrollador tiene el control completo sobre aquello que se incluye en el código. Pero como el código no implementa ningún protocolo abierto, otros desarrolladores independientes no podrán desarrollar código que interopere con esa aplicación.

En esta sección vamos a examinar los problemas fundamentales del desarrollo de aplicaciones propietarias cliente-servidor y echaremos un vistazo al código que implemneta una aplicación cliente-servidor muy sencilla. Durante la fase de desarrollo, una de las primeras decisiones que el desarrollador debe tomar es si la aplicación se ejecutará sobre TCP o sobre UDP. Recuerde que TCP está orientado a la conexión y proporciona un canal fiable de flujo de bytes a través del cual se transmiten los datos entre los dos sistemas terminales. Por su parte, UDP es un protocolo sin conexión, que envía paquetes de datos independientes de un sistema terminal a otro, sin ningún tipo de garantía acerca de la entrega. Recuerde también que cuando un programa cliente o servidor implementa un protocolo definido por un RFC, debe utilizar el número de puerto bien conocido asociado con el protocolo; asimismo, al desarrollar una aplicación propietaria, el desarrollador debe evitar el uso de dichos números de puerto bien conocidos. (Los números de puerto se han explicado brevemente en la Sección 2.1 y los veremos en detalle en el Capítulo 3).

Vamos a presentar la programación de sockets en UDP y TCP mediante una aplicación UDP simple y una aplicación TCP simple. Mostraremos estas sencillas aplicaciones en Python 3. Podríamos haber escrito el código en Java, C o C++, pero hemos elegido Python fundamentalmente porque Python expone de forma clara los conceptos claves de los sockets. Con Python se usan pocas líneas de código, y cada una de ellas se puede explicar a un programador novato sin dificultad, por lo que no debe preocuparse si no está familiarizado con Python. Podrá seguir fácilmente el código si tiene experiencia en programación en Java, C o C++.

Si está interesado en la programación cliente-servidor con Java, le animamos a que consulte el sitio web de acompañamiento del libro; de hecho, allí podrá encontrar todos los ejemplos de esta sección (y las prácticas de laboratorio asociadas) en Java. Para aquellos lectores que estén interesados en la programación cliente-servidor en C, hay disponibles algunas buenas referencias [Donahoo 2001; Stevens 1997; Frost 1994; Kurose 1996]; los ejemplos en Python que proporcionamos a continuación tiene un estilo y aspecto similares a C.

2.7.1 Programación de sockets con UDP

En esta subsección vamos a escribir programas cliente-servidor simples que utilizan UDP. En la siguiente sección, escribiremos programas similares que emplean TCP.

Recuerde de la Sección 2.1 que los procesos que se ejecutan en máquinas diferentes se comunican entre sí enviando mensajes a través de sockets. Dijimos que cada proceso era análogo a una vivienda y que el socket del proceso era análogo a una puerta. La aplicación reside en un lado de la puerta de la vivienda; el protocolo de la capa de transporte reside en el otro lado de la puerta, en el mundo exterior. El desarrollador de la aplicación dispone de control sobre todo lo que está situado en el lado de la capa de aplicación del socket; sin embargo, el control que tiene sobre el lado de la capa de transporte es muy pequeño.

Veamos ahora la interacción existente entre dos procesos que están comunicándose que usan sockets UDP. Si se usa UDP, antes de que un proceso emisor pueda colocar un paquete de datos en la puerta del socket, tiene que asociar en primer lugar una dirección de destino al paquete. Una vez que el paquete atraviesa el socket del emisor, Internet utilizará la dirección de destino para enrutar dicho paquete hacia el socket del proceso receptor, a través de Internet. Cuando el paquete llega al socket de recepción, el proceso receptor recuperará el paquete a través del socket y a continuación inspeccionará el contenido del mismo y tomará las acciones apropiadas.

Así que puede que se esté preguntando ahora: ¿qué es lo que se introduce en la dirección de destino asociada al paquete? Como cabría esperar, la dirección IP del host de destino es parte de esa dirección de destino. Al incluir la dirección IP de destino en el paquete, los routers de Internet serán capaces de enrutar el paquete hasta el host de destino. Pero, dado que un host puede estar ejecutando muchos procesos de aplicaciones de red, cada uno de ellos con uno o más sockets, también es necesario identificar el socket concreto dentro del host de destino. Cuando se crea un socket, se le asigna un identificador, al que se denomina **número de puerto**. Por tanto, como cabría esperar, la dirección de destino del paquete también incluye el número de puerto del socket. En resumen, el proceso emisor asocia con el paquete una dirección de destino que está compuesta de la dirección IP del host de destino y del número de puerto del socket de destino. Además, como veremos enseguida, también se asocia al paquete la dirección de origen del emisor —compuesta por la dirección IP del host de origen y por el número de puerto del socket de origen—. Sin embargo, la asociación de la dirección de origen al paquete *no* suele ser realizada por el código de aplicación UDP; en lugar de ello, lo realiza automáticamente el sistema operativo subyacente.

Vamos a utilizar la siguiente aplicación cliente-servidor simple para demostrar cómo programar un socket tanto para UDP como para TCP:

1. El cliente lee una línea de caracteres (datos) de su teclado y envía los datos al servidor.
2. El servidor recibe los datos y convierte los caracteres a mayúsculas.
3. El servidor envía los datos modificados al cliente.
4. El cliente recibe los datos modificados y muestra la línea en su pantalla.

La Figura 2.27 muestra la actividad principal relativa a los sockets del cliente y del servidor que se comunican a través del servicio de transporte UDP.

A continuación proporcionamos la pareja de programas cliente-servidor para una implementación UDP de esta sencilla aplicación. Realizaremos un análisis detallado línea a línea de cada

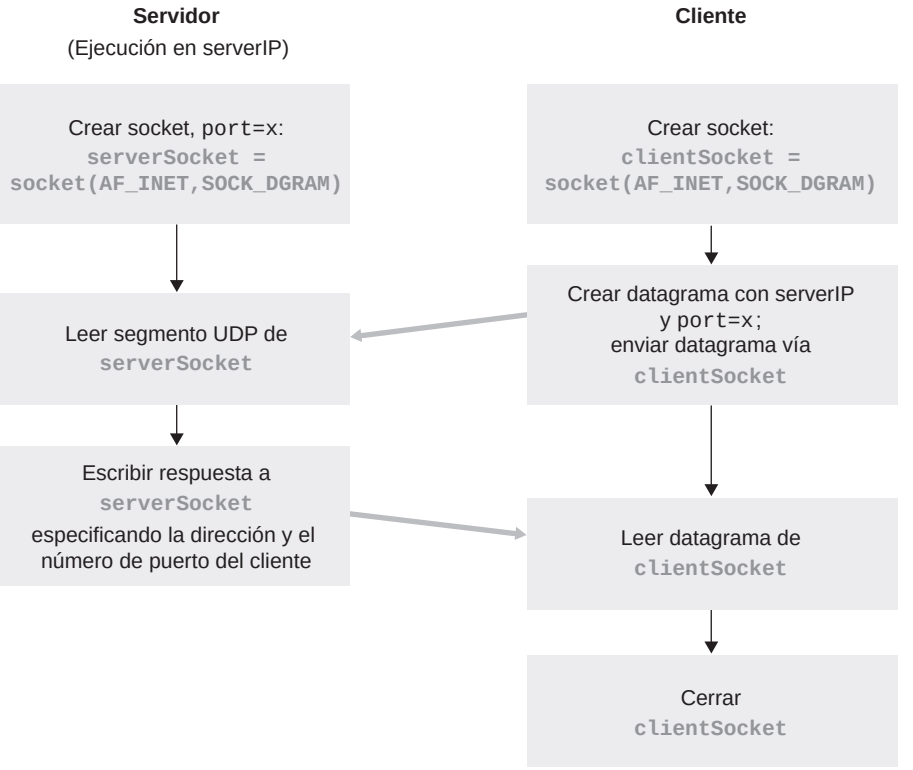

Figura 2.27 ♦ Aplicación cliente-servidor usando UDP.

uno de los programas. Comenzaremos con el cliente UDP, que enviará un mensaje del nivel de aplicación simple al servidor. Con el fin de que el servidor sea capaz de recibir y responder al mensaje del cliente, este debe estar listo y ejecutándose; es decir, debe estar ejecutándose como un proceso antes de que cliente envíe su mensaje.

El nombre del programa cliente es UDPCliente.py y el nombre del programa servidor es UDPServidor.py. Con el fin de poner el énfasis en las cuestiones fundamentales, hemos proporcionado de manera intencionada código que funciona correctamente pero que es mínimo. Un "código realmente bueno" tendría unas pocas más líneas auxiliares, en concreto aquellas destinadas al tratamiento de errores. Para esta aplicación, hemos seleccionado de forma arbitraria el número de puerto de servidor 12000.

UDPCliente.py

He aquí el código para el lado del cliente de la aplicación:

```
from socket import *
serverName = 'hostname'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_DGRAM)
message = raw_input('Escriba una frase en minúsculas:')
clientSocket.sendto(message.encode(),(serverName, serverPort))
modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
print(modifiedMessage.decode())
clientSocket.close()
```

Veamos ahora las distintas líneas de código del programa UDPClient.py.

```
from socket import *
```

El módulo `socket` constituye la base de todas las comunicaciones de red en Python. Incluyendo esta línea, podemos crear sockets dentro de nuestro programa.

```
serverName = 'hostname'
serverPort = 12000
```

La primera línea define la variable `serverName` como la cadena 'hostname'. Aquí, se proporciona una cadena de caracteres que contiene la dirección IP del servidor (como por ejemplo, "128.138.32.126") o el nombre de host del servidor (por ejemplo, "cis.poly.edu"). Si utilizamos el nombre de host, entonces se llevará a cabo automáticamente una búsqueda DNS para obtener la dirección IP.) La segunda línea asigna el valor 12000 a la variable entera `serverPort`.

```
clientSocket = socket(AF_INET, SOCK_DGRAM)
```

Esta línea crea el socket de cliente denominado `clientSocket`. El primer parámetro indica la familia de direcciones; en particular, `AF_INET` indica que la red subyacente está utilizando IPv4. (No se preocupe en este momento por esto, en el Capítulo 4 abordaremos el tema de IPv4.) El segundo parámetro especifica que el socket es de tipo `SOCK_DGRAM`, lo que significa que se trata de un socket UDP (en lugar de un socket TCP). Observe que no se especifica el número de puerto del socket del cliente al crearlo; en lugar de ello, dejamos al sistema operativo que lo haga por nosotros. Una vez que hemos creado la puerta del proceso del cliente, querremos crear un mensaje para enviarlo a través de la puerta.

```
message = raw_input('Escriba una frase en minúsculas:')
```

`raw_input()` es una función incorporada de Python. Cuando este comando se ejecuta, se solicita al usuario que se encuentra en el cliente que introduzca un texto en minúsculas con la frase "Escriba

una frase en minúsculas:". El usuario utiliza entonces su teclado para escribir una línea, la cual se guarda en la variable `message`. Ahora que ya tenemos un socket y un mensaje, desearemos enviar el mensaje a través del socket hacia el host de destino.

```
clientSocket.sendto(message.encode(),(serverName, serverPort))
```

En la línea anterior, en primer lugar convertimos el mensaje de tipo cadena a tipo byte, ya que necesitamos enviar bytes por el socket; esto se hace mediante el método `encode()`. El método `sendto()` asocia la dirección de destino (`serverName, serverPort`) al mensaje y envía el paquete resultante por el socket de proceso, `clientSocket`. (Como hemos mencionado anteriormente, la dirección de origen también se asocia al paquete, aunque esto se realiza de forma automática en lugar de explícitamente a través del código.) ¡Enviar un mensaje de un cliente al servidor a través de un socket UDP es así de sencillo! Una vez enviado el paquete, el cliente espera recibir los datos procedentes del servidor.

```
modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
```

Con esta línea, cuando un paquete procedente de Internet llega al socket del cliente, los datos del paquete se colocan en la variable `modifiedMessage` (mensaje modificado) y la dirección de origen del paquete se almacena en la variable `serverAddress`. Esta variable contiene tanto la dirección IP del servidor como el número del puerto del mismo. El programa UDPClient realmente no necesita esta información de dirección del servidor, puesto que ya la conoce, pero no obstante esta línea de Python proporciona dicha dirección. El método `recvfrom` también especifica el tamaño de buffer de 2048 como entrada. (Este tamaño de buffer es adecuado para prácticamente todos los propósitos.)

```
print(modifiedMessage.decode())
```

Esta línea muestra el mensaje modificado (`modifiedMessage`) en la pantalla del usuario, después de convertir el mensaje en bytes a un mensaje de tipo cadena, que tiene que ser la línea original que escribió el usuario, pero ahora escrito en letras mayúsculas.

```
clientSocket.close()
```

Esta línea cierra el socket y el proceso termina.

UDPServidor.py

Echemos ahora un vistazo al lado del servidor de la aplicación:

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))
print("El servidor está listo para recibir")
while True:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.decode().upper()
    serverSocket.sendto(modifiedMessage.encode(), clientAddress)
```

Observe que el principio de UDPServidor es similar a UDPCliente. También importa el módulo socket, asigna el valor 12000 a la variable entera `serverPort` y crea también un socket de tipo `SOCK_DGRAM` (un socket UDP). La primera línea de código que es significativamente diferente de UDPCliente es:

```
serverSocket.bind(('', serverPort))
```

La línea anterior asocia (es decir, asigna) el número de puerto 12000 al socket del servidor. Así, en UDPServidor, el código (escrito por el desarrollador de la aplicación) asigna explícitamente un número de puerto al socket. De este modo, cuando alguien envía un paquete al puerto 12000 en la dirección IP del servidor, dicho paquete será dirigido a este socket. A continuación, UDPServidor entra en un bucle `while`; este bucle permite a UDPServidor recibir y procesar paquetes de los clientes de manera indefinida. En el bucle `while`, UDPServidor espera a que llegue un paquete.

```
message, clientAddress = serverSocket.recvfrom(2048)
```

Esta línea de código es similar a la que hemos visto en UDPCliente. Cuando un paquete llega al socket del servidor, los datos del paquete se almacenan en la variable `message` y la dirección de origen del paquete se coloca en la variable `clientAddress`. La variable `clientAddress` contiene tanto la dirección IP del cliente como el número de puerto del cliente. Aquí, UDPServidor *hará uso* de esta información de dirección, puesto que proporciona una dirección de retorno, de forma similar a la dirección del remitente en una carta postal ordinaria. Con esta información de la dirección de origen, el servidor ahora sabe dónde dirigir su respuesta.

```
modifiedMessage = message.decode().upper()
```

Esta línea es la más importante de nuestra sencilla aplicación, ya que toma la línea enviada por el cliente y, después de convertir el mensaje en una cadena, utiliza el método `upper()` para pasarlo a mayúsculas.

```
serverSocket.sendto(modifiedMessage.encode(),clientAddress)
```

Esta última línea asocia la dirección del cliente (dirección IP y número de puerto) al mensaje escrito en mayúsculas (después de convertir la cadena a bytes) y envía el paquete resultante al socket del servidor. (Como hemos mencionado anteriormente, la dirección del servidor también se asocia con el paquete, aunque esto se hace de forma automática en lugar de explícitamente mediante el código.) A continuación, se suministrará el paquete a esa dirección de cliente a través de Internet. Una vez que el servidor envía el paquete, permanece en el bucle `while`, esperando la llegada de otro paquete UDP (de cualquier cliente que se esté ejecutando en cualquier host).

Para probar ambos programas, ejecute UDPCliente.py en un host y UDPServidor.py en otro host. Asegúrese de incluir el nombre de host o la dirección apropiados del servidor en UDPCliente.py. A continuación, ejecute UDPServidor.py, el programa del servidor compilado, en el host servidor. De este modo se crea un proceso en el servidor que está a la espera hasta que es contactado por algún cliente. Después, ejecute UDPCliente.py, el programa del cliente compilado, en el cliente. Esto crea un proceso en el cliente. Por último, utilice la aplicación en el cliente, escriba una frase seguida de un retorno de carro.

Para crear su propia aplicación cliente-servidor UDP, puede empezar modificando ligeramente los programas de cliente o de servidor. Por ejemplo, en lugar de pasar todas las letras a mayúsculas, el servidor podría contar el número de veces que aparece la letra *s* y devolver dicho número. O puede modificar el cliente de modo que después de recibir la frase en mayúsculas, el usuario pueda continuar enviando más frases al servidor.

2.7.2 Programación de sockets con TCP

A diferencia de UDP, TCP es un protocolo orientado a la conexión. Esto significa que antes de que el cliente y el servidor puedan empezar a enviarse datos entre sí, tienen que seguir un proceso de acuerdo en tres fases y establecer una conexión TCP. Un extremo de la conexión TCP se conecta al socket del cliente y el otro extremo se conecta a un socket de servidor. Cuando creamos la conexión TCP, asociamos con ella la dirección del socket de cliente (dirección IP y número de puerto) y la dirección del socket de servidor (dirección IP y número de puerto). Una vez establecida la conexión

TCP, cuando un lado desea enviar datos al otro lado, basta con colocar los datos en la conexión TCP a través de su socket. Esto es distinto al caso de UDP, en el que el servidor tiene que tener asociada al paquete una dirección de destino antes de colocarlo en el socket.

Ahora, examinemos más en detalle la interacción entre los programas cliente y servidor en TCP. Al cliente le corresponde iniciar el contacto con el servidor. Para que este puede reaccionar al contacto inicial del cliente, tendrá que estar preparado, lo que implica dos cosas. En primer lugar, como en el caso de UDP, el servidor TCP tiene que estar ejecutándose como proceso antes de que el cliente trate de iniciar el contacto. En segundo lugar, el programa servidor debe disponer de algún tipo de puerta especial (o, más precisamente, un socket especial) que acepte algún contacto inicial procedente de un proceso cliente que se esté ejecutando en un host arbitrario. Utilizando nuestra analogía de la vivienda/puerta para un proceso/socket, en ocasiones nos referiremos a este contacto inicial del cliente diciendo que es equivalente a "llamar a la puerta de entrada".

Con el proceso servidor ejecutándose, el proceso cliente puede iniciar una conexión TCP con el servidor. Esto se hace en el programa cliente creando un socket TCP. Cuando el cliente crea su socket TCP, especifica la dirección del socket de acogida (*wellcoming socket*) en el servidor, es decir, la dirección IP del host servidor y el número de puerto del socket. Una vez creado el socket en el programa cliente, el cliente inicia un proceso de acuerdo en tres fases y establece una conexión TCP con el servidor. El proceso de acuerdo en tres fases, que tiene lugar en la capa de transporte, es completamente transparente para los programas cliente y servidor.

Durante el proceso de acuerdo en tres fases, el proceso cliente llama a la puerta de entrada del proceso servidor. Cuando el servidor "escucha" la llamada, crea una nueva puerta (o de forma más precisa, un *nuevo* socket) que estará dedicado a ese cliente concreto. En el ejemplo que sigue, nuestra puerta de entrada es un objeto socket TCP que denominamos `serverSocket`; el socket que acabamos de crear dedicado al cliente que hace la conexión se denomina `connectionSocket`. Los estudiantes que se topan por primera vez con los sockets TCP confunden en ocasiones el socket de acogida (que es el punto inicial de contacto para todos los clientes que esperan para comunicarse con el servidor) con cada socket de conexión de nueva creación del lado del servidor que se crea posteriormente para comunicarse con cada cliente.

Desde la perspectiva de la aplicación, el socket del cliente y el socket de conexión del servidor están conectados directamente a través de un conducto. Como se muestra en la Figura 2.28, el proceso cliente puede enviar bytes arbitrarios a través de su socket, y TCP garantiza que

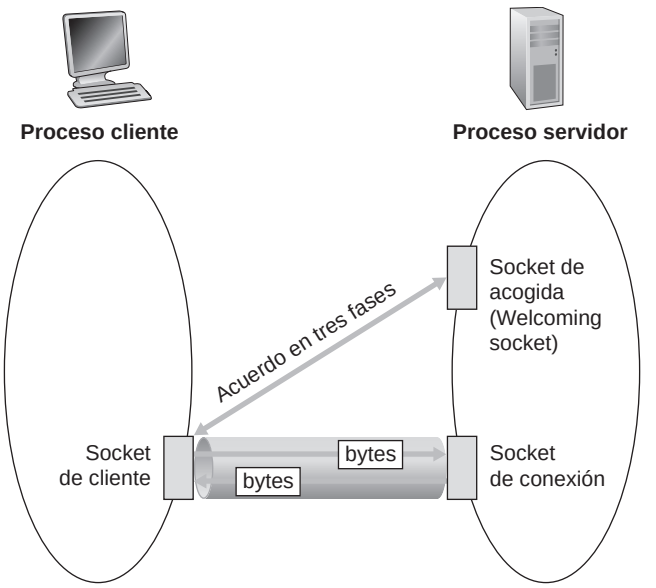

Figura 2.28 ♦ El proceso TCPServidor tiene dos sockets.

el proceso servidor recibirá (a través del socket de conexión) cada byte en el orden en que ha sido enviado. Por tanto, TCP proporciona un servicio fiable entre los procesos cliente y servidor. Además, al igual que las personas pueden entrar y salir a través de una misma puerta, el proceso cliente no sólo envía bytes a través de su socket, sino que también puede recibirlos; de forma similar, el proceso servidor no sólo puede recibir bytes, sino también enviar bytes a través de su socket de conexión.

Vamos a utilizar la misma aplicación cliente-servidor simple para mostrar la programación de sockets con TCP: el cliente envía una línea de datos al servidor, el servidor pone en mayúsculas esa línea y se la devuelve al cliente. En la Figura 2.29 se ha resaltado la actividad principal relativa al socket del cliente y el servidor que se comunican a través del servicio de transporte de TCP.

TCPCliente.py

He aquí el código para el lado del cliente de la aplicación:

```
from socket import *
serverName = 'servername'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
```

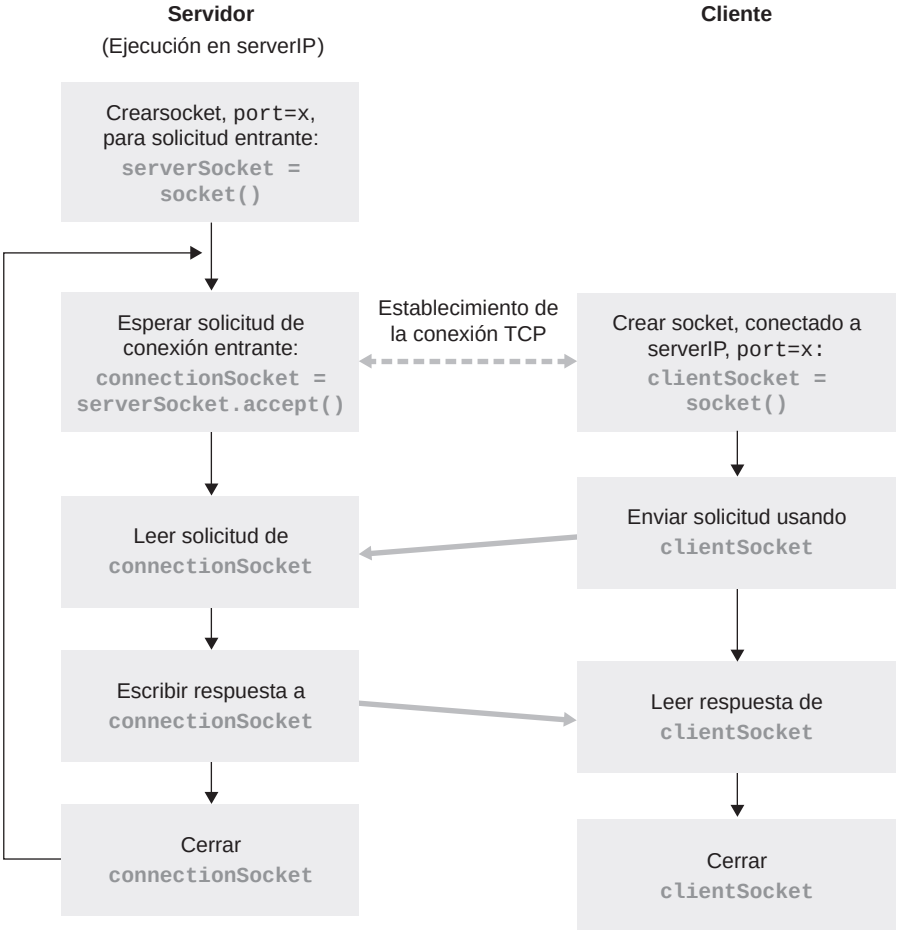

Figura 2.29 ♦ La aplicación cliente-servidor utilizando TCP.

```
sentence = raw_input('Escriba una frase en minúsculas:')
clientSocket.send(sentence.encode())
modifiedSentence = clientSocket.recv(1024)
print('From Server: ', modifiedSentence.decode())
clientSocket.close()
```

Fijémonos ahora en las líneas del código que difieren significativamente de la implementación para UDP. La primera de estas líneas es la de creación del socket de cliente.

```
clientSocket = socket(AF_INET, SOCK_STREAM)
```

Esta línea crea el socket de cliente, denominado `clientSocket`. De nuevo, el primer parámetro indica que la red subyacente está utilizando IPv4. El segundo parámetro indica que el socket es de tipo `SOCK_STREAM`, lo que significa que se trata de un socket TCP (en lugar de un socket UDP). Observe que de nuevo no especificamos el número de puerto del socket de cliente al crearlo; en lugar de ello, dejamos que sea el sistema operativo el que lo haga por nosotros. La siguiente línea de código es muy diferente de la que hemos visto en UDPClient:

```
clientSocket.connect((serverName,serverPort))
```

Recuerde que antes de que el cliente pueda enviar datos al servidor (o viceversa) empleando un socket TCP, debe establecerse primero una conexión TCP entre el cliente y el servidor. La línea anterior inicia la conexión TCP entre el cliente y el servidor. El parámetro del método `connect()` es la dirección del lado de servidor de la conexión. Después de ejecutarse esta línea, se lleva a cabo el proceso de acuerdo en tres fases y se establece una conexión TCP entre el cliente y el servidor.

```
sentence = raw_input('Escriba una frase en minúsculas:')
```

Como con UDPClient, la línea anterior obtiene una frase del usuario. La cadena `sentence` recopila los caracteres hasta que el usuario termina la línea con un retorno de carro. La siguiente línea de código también es muy diferente a la utilizada en UDPClient:

```
clientSocket.send(sentence.encode())
```

La línea anterior envia la cadena `sentence` a través del socket de cliente y la conexión TCP. Observe que el programa *no* crea explícitamente un paquete y asocia la dirección de destino al paquete, como sucedía en el caso de los sockets UDP. En su lugar, el programa cliente simplemente coloca los bytes de la cadena `sentence` en la conexión TCP. El cliente espera entonces a recibir los bytes procedentes del servidor.

```
modifiedSentence = clientSocket.recv(2048)
```

Cuando llegan los caracteres de servidor, estos se colocan en la cadena `modifiedSentence`. Los caracteres continúan acumulándose en `modifiedSentence` hasta que la línea termina con un carácter de retorno de carro. Después de mostrar la frase en mayúsculas, se cierra el socket de cliente:

```
clientSocket.close()
```

Esta última línea cierra el socket y, por tanto, la conexión TCP entre el cliente y el servidor. Esto hace que TCP en el cliente envíe un mensaje TCP al proceso TCP del servidor (véase la Sección 3.5).

TCPServidor.py

Veamos ahora el programa del servidor.

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
print('El servidor está listo para recibir')
while True:
    connectionSocket, addr = serverSocket.accept()
    sentence = connectionSocket.recv(1024).decode()
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence.encode())
    connectionSocket.close()
```

Estudiemos ahora las líneas que difieren significativamente en UDPServidor y TCPCliente. Como en el caso de TCPCliente, el servidor crea un socket TCP con:

```
serverSocket=socket(AF_INET,SOCK_STREAM)
```

De forma similar a UDPServidor, asociamos el número de puerto de servidor, `serverPort`, con este socket:

```
serverSocket.bind(('',serverPort))
```

Pero con TCP, `serverSocket` será nuestro socket de acogida. Después de establecer esta puerta de entrada, esperaremos hasta escuchar que algún cliente llama a la puerta:

```
serverSocket.listen(1)
```

Esta línea hace que el servidor esté a la escucha de solicitudes de conexión TCP del cliente. El parámetro especifica el número máximo de conexiones en cola (al menos, 1).

```
connectionSocket, addr = serverSocket.accept()
```

Cuando un cliente llama a esta puerta, el programa invoca el método `accept()` para el serverSocket, el cual crea un nuevo socket en el servidor, denominado `connectionSocket`, dedicado a este cliente concreto. El cliente y el servidor completan entonces el acuerdo en tres fases, creando una conexión TCP entre el socket `clientSocket` del cliente y el socket `connectionSocket` del servidor. Con la conexión TCP establecida, el cliente y el servidor ahora pueden enviarse bytes entre sí a través de la misma. Con TCP, no solo está garantizado que todos los bytes enviados desde un lado llegan al otro lado, sino que también queda garantizado que llegarán en orden.

```
connectionSocket.close()
```

En este programa, después de enviar la frase modificada al cliente, se cierra el socket de conexión. Pero puesto que `serverSocket` permanece abierto, otro cliente puede llamar a la puerta y enviar una frase al servidor para su modificación.

Esto completa nuestra exposición acerca de la programación de sockets en TCP. Le animamos a que ejecute los dos programas en dos hosts distintos y a que los modifique para obtener objetivos ligeramente diferentes. Debería comparar la pareja de programas para UDP con los programas para TCP y ver en qué se diferencian. También le aconsejamos que realice los ejercicios sobre programación de sockets descritos al final de los Capítulos 2, 4 y 9. Por último, esperamos que algún día, después de dominar estos y otros programas de sockets más complejos, escriba sus propia aplicación de red popular, se haga muy rico y famoso, y recuerde a los autores de este libro de texto.