Redes y Comunicación 2

**M1.1 – Sockets y Conexiones**

Un protocolo es una secuencia ordenada de interacciones (mensajes), con un formato concreto entre dos o más partes.

Protocolos de red: interaccionan ordenadores, toda la comunicación de internet esa gestionada por protocolos.

Se definen para reglas como mensajes que deben ser enviados y acciones que deben tomarse cuando se recibe un mensaje específico.

Diagrama

Descripción generada automáticamenteLos protocolos definen el formato y orden de los mensajes enviados y recibidos entre los dispositivos en comunicación, así como las acciones a realizar.

**Creando una aplicación de red:**

Escribir programas que ejecutan en sistemas finales diferentes, se comunican por la red.

Por ejemplo, un programa “navegador” se comunica con un programa “servidor web”.

No se necesita escribir software para los elementos de red. Los elementos intermedios de la

**Diagrama

Descripción generada automáticamente**red no ejecutan aplicaciones de usuario. Las aplicaciones en los extremos de la conexión

**Diagrama

Descripción generada automáticamente**permiten un desarrollo rápido y simple.

**Paradigma Cliente-Servidor**

Servidor:

▪ Máquina siempre disponible

▪ Dirección IP permanente

▪ Normalmente ubicada en centros de datos

Clientes:

▪ Cualquier dispositivo

▪ Se conecta de forma intermitente

▪ Puede (suele) tener una IP dinámica

▪ Los clientes no se comunican entre sí

**Arquitectura Peer-to-Peer (P2P)**

▪ Cada cliente es también un servidor (nodos)

▪ Los nodos son todos iguales y se comunican directamente entre sí

Diagrama

Descripción generada automáticamente▪ Características:

+ Descentralizada

+ Redundancia de información

+ Alta disponibilidad

- Gestión de nodos complicada

**Comunicación entre procesos**

Proceso: programa en ejecución

▪En un mismo sistema se comunican utilizando primitivas (IPC) (semáforos, memoria compartida, paso de mensajes)

▪En diferentes sistemas utilizan conexiones por las que intercambian mensajes

Proceso cliente: inicia la comunicación.

Proceso servidor: espera a ser contactado y response solo entonces

**Sockets**

▪ Los procesos envían y reciben mensajes a y desde la red a través de sus sockets.

▪ Un socket es como una puerta:

• Proceso emisor: saca mensajes por la puerta y confía en la infraestructura de transporte (pila TCP/IP), que lleva los mensajes al socket en el proceso receptor

Diagrama

Descripción generada automáticamente

**Sockets y conexiones**

▪Para poder recibir mensajes, un proceso debe tener un identificador único en esa máquina ▪Identificador compuesto de:

○ Dirección IP (IPv4 32 bits, IPv6 128 bits)

○ Número de puerto: 1-65545 ( < 1024 privilegiados)

▪Puertos asociados a aplicaciones estándar:

• Servidor Web (HTTP): 80 y 443

• Servidor de correo: 25

▪Para enviar un mensaje HTTP al servidor uam.es:

• Dirección IP: 150.244.214.237

• Número de puerto: 80

▪ Un socket es como un “enchufe”, compuesto de (dirIP, puerto)

▪ Una conexión es como un cable que une dos sockets 17

▪ Conexión de un navegador a un servidor Web:

Tabla

Descripción generada automáticamente con confianza baja

**Puertos habituales**

Tabla

Descripción generada automáticamente▪Para poder recibir mensajes, un proceso debe tener un identificador único en esa máquina ▪Identificador compuesto de:

○ Dirección IP (IPv4 32 bits, IPv6 128 bits)

○ Número de puerto: 1-65545 ( < 1024 privilegiados)

1. Investiga sobre el ciclo de vida de los estados de un socket, especialmente TIME\_WAIT.

El ciclo de vida de un socket es un proceso que describe cómo se crea, se usa y se cierra una conexión de red entre dos dispositivos. Uno de los estados del ciclo de vida de un socket es TIME\_WAIT, que se produce cuando se cierra una conexión y se espera a que todos los paquetes restantes se envíen o se reciban correctamente. Durante el estado TIME\_WAIT, el socket no puede ser reutilizado para otra conexión hasta que haya transcurrido un tiempo determinado. Esto se hace para asegurarse de que no se produzcan confusiones entre paquetes de diferentes conexiones que puedan tener los mismos números de puerto y secuencia.

2. Por tanto, ¿quién debe empezar el cierre de una conexión, el cliente o el servidor? ¿Por qué?

En general, el cliente suele ser el que inicia el cierre de la conexión, enviando un mensaje de FIN al servidor. El servidor, a su vez, envía un mensaje de ACK al cliente para confirmar que ha recibido el mensaje de FIN y está listo para cerrar la conexión. Luego, el servidor puede enviar su propio mensaje de FIN para cerrar la conexión de manera segura. El cliente confirma este mensaje de FIN con un ACK final y la conexión se cierra definitivamente.

3. Desde el estado ESTABLISHED, dibuja tu propio diagrama de intercambio de mensajes entre cliente y servidor para el cierre de una conexión.

Cliente Servidor

------ --------

ESTABLISHED estado

Envía mensaje de FIN

RECIBE mensaje de FIN

CLOSE\_WAIT estado

RECIBE ACK del mensaje de FIN

FIN\_WAIT\_1 estado

Envía ACK al mensaje de FIN

Envía mensaje de FIN

CLOSING estado

RECIBE ACK del segundo mensaje de FIN

FIN\_WAIT\_2 estado

RECIBE ACK del primer mensaje de FIN

TIME\_WAIT estado

Envía ACK del segundo mensaje de FIN

CLOSED estado

RECIBE ACK del tercer mensaje de FIN

TIME\_WAIT estado (después de un breve período de tiempo)

Datos:

127.0.0.1 es la dirección de loopback (también conocida como localhost).

0.0.0.0 es una metadirección no enrutable que se usa parefera designar un objetivo no válido,

desconocido o no aplicable (un marcador de posición "sin dirección en particular").

En el contexto de una entrada de ruta, generalmente significa la ruta predeterminada.

En el contexto de los servidores, 0.0.0.0 significa *todas las direcciones IPv4 en la máquina local*. Si un host tiene dos direcciones IP, 192.168.1.1 y 10.1.2.1, y un servidor que se ejecuta en el host escucha en 0.0.0.0, se podrá acceder a él en ambas IP.

Puerto filtrado: Un firewall (cortafuegos) bloquea el acceso al puerto.

Puerto cerrado: El puerto no está bloqueado, pero no hay ninguna aplicación escuchando en él.

Puerto abierto: El puerto no está bloqueado y hay una aplicación escuchando en él.

**Servicios sobre capa de transporte**

Servicio sobre TCP:

▪ Transmisión confiable entre los procesos emisor y receptor

▪ Control de flujo: emisor y receptor se sincronizan para no saturarse

▪ Control de congestión: emisor se controla para no saturar la red

▪ Orientado a conexión: la comunicación necesita de una conexión previa entre emisor y receptor

Servicio sobre UDP:

▪ Transmisión no confiable entre emisor y receptor (los datagramas pueden perderse o llegar desordenados)

▪ No orientado a conexión: la comunicación puede empezar inmediatamente, sin necesidad de conexión previa

**Programación de sockets con UDP**

Diagrama

Descripción generada automáticamente

**Cliente UDP**

Escala de tiempo

Descripción generada automáticamente con confianza media

**Servidor UDP**

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

**Programación de sockets con TCP**

El cliente siempre inicia la conexión:

● El proceso servidor debe estar ya corriendo

Cuando la recibe, el servidor crea una nueva conexión a partir del socket “escuchante” para ese cliente en particular:

● Esto permite varios clientes simultáneos

● El puerto de origen de la conexión es elegido de forma aleatoria por el SO y permite distinguir a los clientes

**Programación de sockets con TCP**

Diagrama

Descripción generada automáticamente con confianza media

**Interacción completa: TCP**

Diagrama

Descripción generada automáticamente

**Cliente TCP**

Texto

Descripción generada automáticamente con confianza baja

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

**Tipos de servidores // Concurrencia**

Diagrama, Escala de tiempo

Descripción generada automáticamente

Texto

Descripción generada automáticamenteServidor iterativo

Interfaz de usuario gráfica, Texto

Descripción generada automáticamenteServidor concurrente // Nuevo proceso/thread por conexión

Texto

Descripción generada automáticamenteServidor concurrente // Pool de procesos/threads

Pool de procesos // Tamaño del pool

¿Qué tamaño debe tener el pool?

● Fijo:

✓ Sencillo de implementar

✕ No se adapta a la carga (alta o baja) del sistema

● Dinámico: se crea un número bajo, y si el número de procesos libres está a punto de agotarse se crean nuevos

✓ Se adapta a la carga (siempre con un límite máximo de procesos o DoS)

✕ Dependiendo de la política utilizada, puede ser complejo de implementar

Cada tipo de servidor tiene sus propias ventajas y desventajas en cuanto al rendimiento y el comportamiento en distintos escenarios. A continuación, se describen brevemente los diferentes tipos de servidores y cómo se comportan en algunos escenarios comunes:

* Servidores monolíticos: son aquellos en los que todo el código se ejecuta en un solo proceso. En general, este tipo de servidor puede ofrecer un alto rendimiento en situaciones de baja complejidad, ya que no requieren de comunicación entre procesos y tienen un menor consumo de recursos. Sin embargo, pueden resultar menos escalables y flexibles a medida que crece la complejidad del sistema o aumenta el tráfico.
* Servidores de múltiples hilos o procesos: estos servidores distribuyen el trabajo en múltiples hilos o procesos para poder manejar múltiples solicitudes al mismo tiempo. En general, esto puede mejorar el rendimiento y la escalabilidad en sistemas de mayor complejidad y en situaciones de mayor carga de trabajo. Sin embargo, la comunicación entre hilos o procesos puede ser más compleja y puede haber problemas de sincronización, lo que puede afectar el rendimiento y la estabilidad.
* Servidores de microservicios: estos servidores se basan en una arquitectura en la que los diferentes componentes del sistema se ejecutan como servicios separados, cada uno con su propio proceso y comunicación a través de interfaces de red. Esto permite una mayor escalabilidad y flexibilidad, ya que los diferentes componentes pueden ser escalados y actualizados de forma independiente. Sin embargo, la complejidad de la gestión de múltiples servicios puede ser mayor, y la latencia de red puede ser un factor importante.
* Servidores sin estado (stateless): estos servidores no mantienen ningún estado de sesión entre las solicitudes. Cada solicitud se maneja de forma independiente, lo que permite una alta escalabilidad y una mejor tolerancia a fallos. Sin embargo, la falta de estado puede dificultar la implementación de ciertas funcionalidades, como la autenticación y la autorización.
* Servidores con estado (stateful): estos servidores mantienen un estado de sesión entre las solicitudes, lo que les permite ofrecer funcionalidades más avanzadas, como la autenticación y la autorización. Sin embargo, la gestión del estado puede ser más compleja y puede limitar la escalabilidad y la tolerancia a fallos

**M1.2 - Protocolo HTTP**

**El protocolo HTTP: la navaja suiza de Internet**

▪ Diseñado por Tim Berners-Lee del CERN en 1989, como parte de su visión de una red global: la WorldWideWeb.

▪ El primer servidor Web entró en funcionamiento en 1990, con una única página HTML estática.

▪ Actualmente se utiliza también como protocolo “base” para otras muchas aplicaciones a través de APIs:

○ Mensajería (Whatsapp, Telegram)

○ Correo electrónico

○ Streaming

**Web // Conceptos básicos**

▪ Una página Web está compuesta de objetos (un fichero HTML, una imagen, un video, etc.) que pueden (suelen) estar almacenados en servidores diferentes

Texto

Descripción generada automáticamente▪ Se compone de un fichero HTML base, que incluye varios objetos, cada uno referenciable con una URL:

**HTTP // Conceptos básicos**

▪ Protocolo de nivel de aplicación basado en texto

▪ Esquema cliente/servidor:

• Cliente: aplicación (navegador) que solicita, recibe y renderiza objetos Web

Diagrama

Descripción generada automáticamente• Servidor: aplicación (servidor Web) que devuelve los objetos solicitados en cada petición del cliente

**HTTP // No tiene memoria**

▪Un servidor Web no mantiene información sobre las peticiones anteriores, ni sabe a quién pertenece cada una

▪Necesarios mecanismos “externos”:

○ Cookies

○ Tokens JWT

Texto

Descripción generada automáticamente

Texto, Carta

Descripción generada automáticamente

Escala de tiempo

Descripción generada automáticamenteTexto

Descripción generada automáticamenteDiagrama, Esquemático

Descripción generada automáticamente

Imagen que contiene Tabla

Descripción generada automáticamente

Tabla

Descripción generada automáticamente

Diagrama

Descripción generada automáticamente

Conexiones persistentes vs no persistentes

HTTP no persistente

● Por defecto en HTTP/1.0

● La conexión TCP se establece y cierra para cada objeto

● Los navegadores abren múltiples conexiones simultáneas

HTTP persistente

▪ Por defecto en HTTP/1.1

▪ La conexión no se cierra, de forma que pueden enviarse varios objetos por una única conexión TCP

▪ El cliente envía una petición tan pronto como encuentra un objeto (pipelining)

**Tiempo de respuesta**

RTT (Round-Trip Time): tiempo para que un paquete vaya y vuelva del servidor

Tiempo respuesta HTTP:

▪ Un RTT y medio (¿por qué?) para iniciar conexión TCP

▪ Un RTT por cada petición

▪ Tiempo de transmisión del recurso

**Pipelining**

Modo por defecto en HTTP/1.1

● Los objetos pueden transferirse simultáneamente: en teoría, en (aprox.) un único RTT

● Sin embargo, el servidor debe enviar las respuestas en el orden en el que se reciben, por lo que toda la conexión es FIFO (puede ocurrir un bloqueo HOL)

Diagrama

Descripción generada automáticamente

**Navegadores y HTTP**

● El estándar solo define características del protocolo, pero es decisión de los desarrolladores implementarlas o no, e incorporar algunas nuevas.

● En ejercicios y modelos, consideraremos un modelo lo más realista posible del comportamiento de un navegador moderno (Chrome), a no ser que se indique lo contrario:

a. Cargan completamente una página Web y, solo después, solicita los recursos que contiene.

b. Tiene un límite de 6 conexiones simultáneas con un mismo servidor, y 10 en total.

c. Si puede, reutiliza conexiones abiertas.

**HTTP/2**

▪Actualmente, una página media ocupa 1.9 MB, y necesita de unos 100 recursos para visualizarse. HTTP/1.1 no fue diseñado para estos números, y no rinde bien

▪ HTTP/2 es la primera gran actualización del protocolo HTTP en casi 20 años.

▪Actualmente todos los navegadores lo soportan, y la mayoría de los servidores Web lo utilizan

**HTTP/2**

HTTP/1.1 introdujo el pipeline de peticiones sobre una única conexión TCP, pero:

▪El servidor debe responder en orden (FCFS: first-come-first-served, aka FIFO)

▪Con FCFS, los objetos pequeños tienen que esperar mucho detrás de objetos grandes, fenómeno denominado bloqueo HOL (head-of-line blocking)

**HTTP/2: mitigando el bloqueo HOL**

Escenario: un cliente HTTP/1.1 solicita un objeto de gran tamaño (O1 ) y tres pequeños

Imagen que contiene Interfaz de usuario gráfica

Descripción generada automáticamente

HTTP/2: multiplexing

En HTTP/2, los objetos se dividen en frames, que pueden ser intercalados Cliente Servidor

Interfaz de usuario gráfica, Diagrama

Descripción generada automáticamente

Como hemos visto, el multiplexing de HTTP/2 mejora considerablemente el problema del pipelining en HTTP/1.1, que, aunque puede enviar todas las peticiones de objetos simultáneamente, el servidor aún envía las respuestas de forma secuencial

**HTTP/2**

HTTP/2: [RFC 7540, 2015] incrementa la flexibilidad del servidor para enviar recursos al cliente:

▪ Para mantener la compatibilidad al máximo, métodos, códigos de estado y cabeceras son iguales a las de HTTP/1.1

▪ A cada objeto se le puede asignar una prioridad, que influye en el orden en el que se envía (no necesariamente FCFS)

▪ Además, los objetos se dividen en frames, para mitigar el bloqueo HOL ▪ El servidor puede enviar (push) objetos no solicitados (en HTTP/1.1 se simula con websockets)

**Manteniendo el estado**

Recuerda: HTTP es un protocolo sin estado, luego, para un servidor Web, todas las peticiones son independientes

▪ Obviamente, una transacción Web normal necesita “recordar” quiénes somos

▪Para ello, el servidor proporciona al navegador pequeñas piezas de información (cookie) únicas para cada usuario

▪Éste las guarda, y las vuelve a entregar al servidor cuando se visita el mismo dominio que generó la cookie

**Estado HTTP // ¿Para qué?**

Mantener el estado es necesario para: ▪ Autorización de los usuarios ▪ Carritos de la compra ▪ Recomendaciones ▪ Web e-mail, etc.

¿Cómo hacerlo?

▪ Actualmente se utilizan esencialmente dos métodos:

○ Cookies

○ Tokens de autenticación (JWT)

**Cookies y privacidad**

Las cookies se han convertido en una verdadera amenaza a nuestra privacidad en Internet, puesto que permiten rastrear nuestra identidad por múltiples sitios, incluyendo aquellos que no las generaron (y que, en principio, no deberían tener acceso a ellas). Servicios como Facebook, cuyo modelo de negocio se basa en la publicidad, no existirían sin las cookies, o si se establecieran medidas más estrictas para su uso.

**Manteniendo el estado // Cookies**

Tanto las web como los clientes utilizan cookies para mantener el estado entre transacciones Cuatro componentes:

1) Cabecera cookie en respuesta HTTP

2) Cabecera cookie en la siguiente request HTTP

3) Cookie almacenada en el host, gestionada por el navegador del usuario

4) acceso a la base de datos del back-end del sitio web

Example:

▪ Susan utiliza el navegador de su ordenador portátil y visita por primera vez un sitio específico de comercio electrónico.

▪ cuando las peticiones HTTP iniciales llegan al sitio, se crea:

ID único (aka “cookie”)

• Entrada en la base de datos del backend para esa ID

• Las siguientes peticiones HTTP de Susan a este sitio contendrán el valor ID, permitiendo “identificar” a Susan

Diagrama

Descripción generada automáticamente

**Autenticación y autorización**

● Las aplicaciones Web necesitan autenticar y autorizar a sus usuarios

● Autenticación: comprobar que un usuario es quien dice ser (típicamente, con un usuario/contraseña)

● Autorización: comprobar que un usuario autenticado tiene permisos para realizar cierta acción (p. ej.: un usuario puede publicar en un foro o no)

Imagen que contiene Tabla

Descripción generada automáticamente

**Cookies vs Tokens**

● Las aplicaciones necesitan mantener la identificación del usuario y su estado de autenticación (‘logado o no’) en algún sitio → sesión

● Además, las aplicaciones SPA (Simple Page App), APIs REST u objetos IoT (Internet of Things) no pueden utilizar el esquema tradicional de sesiones y cookies: ○ No es posible acceder a la BD en cada petición, también por rendimiento

**Login de un usuario utilizando cookies**

1. Usuario introduce sus credenciales en un formulario Web

2. El servidor las verifica y crea una sesión en la BD

3. El cliente guarda una cookie en su navegador con el ID de sesión

4. Esta cookie se envía en cada petición posterior y se verifica contra la BD

5. Cuando el usuario hace logout, se destruye la sesión en cliente y servidor

**Tokens de autenticación**

• Cuando se utilizan para autenticar, las cookies deben comprobarse accediendo a la BD en cada petición del usuario → muy mal rendimiento

• Los tokens no tienen este problema: cada token está firmado, lo que permite comprobar su validez sin más información

• Se envía como una cabecera en cada petición:

JSON Web Tokens (JWT)

Diagrama

Descripción generada automáticamente

Texto, Chat o mensaje de texto

Descripción generada automáticamente

**Cachés Web**

▪El tráfico del usuario se configura para que viaje a través de un servidor caché

Web local:

• si objecto en caché: se devuelve el objeto

• sino la caché pide el recurso al servidor, lo guarda para el futuro y lo devuelve al cliente. OBJETIVO: evitar que las peticiones de un cliente tengan que viajar hasta el servidor de destino

Diagrama

Descripción generada automáticamente

**Cachés Web // Servidores proxy**

¿Por qué?

▪ Reduce los tiempos de respuesta

• La caché está más cerca (menos saltos)

▪ Reduce el tráfico en el enlace de salida

• Muchas peticiones no necesitan viajar hasta el servidor de destino

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente con confianza mediaEl servidor decide qué contenido puede cachearse, incluyendo cabeceras en las respuestas HTTP:

**Caché en el navegador // GET condicional**

OBJETIVO: no realizar peticiones si el navegador tiene una versión actualizada del objeto solicitado

▪ Cliente: indica la fecha de la versión cacheada con una cabecera específica en la petición HTTP: If-modified-since:

▪Servidor: si la copia está actualizada, no se devuelve el objeto: HTTP/1.1 304 Not Modified

Diagrama

Descripción generada automáticamente

**Escenario real**

Escenario

▪ Ancho de banda del enlace de acceso: 1.54 Mbps

▪ RTT del router institucional a Internet: 2 seg

▪ Tamaño de los objetos: 100K bits

▪ Tasa medias de peticiones de los usuarios: 15 peticiones/seg

▪ RESULTADO: tráfico generado por los navegadores: 1.50 Mbps

Rendimiento

▪ Utilización del enlace de acceso = 1.5 / 1.54 = .97

▪ Utilización LAN = .0015

▪ Tiempo total = Retraso Internet + Retraso enlace de acceso + Retraso LAN = 2 seg + minutos + usegs

Diagrama

Descripción generada automáticamente

**Opción: instalar una caché Web**

Servidores Internet Red institucional 1 Gbps LAN Enlace de acceso 1.54 Mbps Caché Web local Suponemos un tasa de acierto de la caché de 0.4:

▪ 40% de las peticiones se sirven por la caché, con muy poco retraso (mseg)

▪ 60% deben viajar hasta el servidor

• Tráfico que atraviesa el enlace de acceso = 0.6 \* 1.50 Mbps = .9 Mbps

• Utilización del enlace = 0.9/1.54 = .58

Diagrama

Descripción generada automáticamente▪ Retraso total: = 0.6 \* (retraso Internet) + 0.4 \* (retraso de la caché) = 0.6 (2 segs) + 0.4 (~msecs) = ~ 1.2 secs

**M1.3 – Correo Electrónico**

Kurose, sección 3.5.6, completa

**M1.4 – DNS**

**DNS: Domain Name System**

Cualquier elemento conectado a Internet dispone de:

• Dirección IP, usada por las máquinas

• Nombre, usado por los humanos

**Domain Name System (DNS):**

▪Base de datos distribuida implementada como una jerarquía de servidores de nombres ▪Protocolo de nivel de aplicación: las aplicaciones lo utilizan (normalmente, a petición de un usario) para resolver un nombre

▪ Utiliza el puerto 53 UDP para la comunicación cliente - servidor, y 53 TCP para servidor – servidor

**DNS // Servicios y estructura**

Q: ¿Por qué no un esquema centralizado?

▪ Punto único de fallo: es un servicio crítico para Internet

▪ Volumen de tráfico inmanejable Servicios DNS básicos:

▪Tradución nombre <-> dirección IP

▪Host aliasing • Múltiples nombres para un mismo host

• P. ej: permite alojar varios sitios Web en un único servidor

▪ Distribución de carga

• Servidores Web replicados: varias direcciones IP corresponden al mismo nombre A: ¡No escala!

Diagrama

Descripción generada automáticamente

**DNS // Servidores raíz**

Último recurso para servidores DNS, que no conocen a otros servidores que necesitan

▪ Función increíblemente importante para Internet

• No funcionaría sin ellos

▪ Gestionados por la ICANN (Internet Corporation for Assigned Names and Numbers)

Forma

Descripción generada automáticamente

**DNS // Proceso de resolución**

Máquina en asterix.uam.es necesita la dirección IP de [www.rediris.es](http://www.rediris.es)

0. Navegador solicita al resolver (proceso local) la resolución.

1. El resolver se comunica con el DNS local (normalmente, configurado a mano). Éste comprueba si la petición está en su caché. Como es una petición recursiva, él se encarga del resto del proceso.

2. Si no conoce la IP del TLD correspondiente (.es), pregunta a un servidor raíz aleatorio. Si la conoce, pregunta al TLD directamente (paso 4)

3. El servidor raíz devuelve una lista con TLDs para el dominio .es

4. El servidor DNS elige una entrada de la lista y le consulta.

5. El servidor TLD correspondiente responde con la dirección IP del servidor primario del dominio .rediris.es

6. El servidor DNS pregunta finalmente al servidor primario por el host ‘www’

7. Se devuelve la respuesta correcta, que se pasa al solictante en el paso 8

Diagrama, Esquemático

Descripción generada automáticamente

**Registros DNS**

Tipo A (Address) Se utilizan para una resolución simple dominio > dirección IP 13

Tipo NS (Name Server) Sirven para obtener el servidor primario correspondiente a un dominio Tipo CNAME (Canonical NAME) Sirven para crear “alias” (punteros) diferentes para un mismo dominio

Tipo MX (eMailer eXchange) Permiten obtener el servidor de correo asociado a un dominio

**Caché DNS**

Texto

Descripción generada automáticamente▪ La infraestructura DNS cachea a varios niveles las traducciones para reducir los tiempos de respuesta

**Caché DNS**

Las entradas cacheadas DEBEN caducar (tienen un TTL):

•Si no, si una máquina cambia su IP, los DNS devolverían siempre respuestas incorrectas.

• Un TTL típico para servidores DNS es de 48 horas.

**DNS // Formato de los mensajes**

Tanto las consultas como respuestas DNS tienen el mismo formato Cabecera

▪ ID: identificador de 16 bits, para que el cliente pueda mapear la respuesta

▪ flags:

• Petición o respuesta

• Solicitud de recursión

• recursion available

• reply is authoritative

Tabla

Descripción generada automáticamente

**¿Cómo se generan nuevos dominios?**

Escenario: queremos crear una web para una nueva empresa llamada “UAM Utopia”

▪ Registrar el nombre de dominio (‘uamutopia.com’) en un registrador DNS:

• Datos personales del responsable del dominio

• Crear los registros A y NS el TLD .com (se encarga el registrador): (dns1.uamutopia.com, NS) (uamutopia.com, 212.212.212.1, A) (dns1.uamutopia.com, 212.212.212.1, A)

▪ Instalar y configurar un servidor DNS en la máquina con IP 212.212.212.1

• Registro A para www.uamutopia.com, para el servidor Web

• Registro MX para uamutopia.com, para poder recibir y enviar correo

**DNS // Seguridad**

Ataques DDoS (Distributed Denial of Service)

▪Inundar servidores raíz con tráfico

• Sin éxito hasta la fecha

• Contramedida: servidores locales cachean las IPs de los TLDs, evitando consultas a los servidores raíz

Ataques de suplantación (spoofing)

▪ Interceptan las peticiones DNS (que no están protegidas criptográficamente), y devuelven respuestas incorrectas, para redirigiar el tráfico a un servidor del atacante

▪ Contramedida: RFC 4033 (Servicios de autenticación DNSSEC)

**DNS // DNS spoofing**

● El atacante debe:

○ Ser más rápido en su respuesta que el servidor real

○ Adivinar el ID de transacción original

○ La petición no debe estar ya cacheada por el DNS local

● Más info: How Hackers Spoof DNS Requests With DNS Cache PoisoningDiagrama

Descripción generada automáticamente con confianza baja

**M1.5-Arquitectura Peer-to-Peer (P2P)**

Cada cliente es también un servidor (nodos)

▪ Los nodos son todos iguales y se comunican directamente entre sí

▪ Los nodos se conectan de forma intermitente, cambiando de IP • Gestión compleja ▪ Ejemplos: intercambio de ficheros (BitTorrent), criptomonedas (Bitcoin), VoIP (Skype)

**Diagrama

Descripción generada automáticamente**

**P2P // Operaciones básicas**

Registro (Bootstrap): Proceso por el que un nodo se une a la red. En un entorno descentralizado, es un problema más difícil.

Búsqueda de información (Lookup): La información suele dividirse en trozos (“chunks”) que son más fáciles de intercambiar y almacenar. Esta operación localiza en qué nodo se encuentra cada porción que necesita.

Descarga de información (Retrieval): Una vez localizada la información, el participante se conecta directamente a los nodos necesarios y comienza la descarga. Puede (suele) descargar chunks de un mismo fichero de distintos nodos a la vez.

**P2P // Registro**

Existen dos soluciones básicas:

● Servidor de descubrimiento público: sirve como “referencia” inicial, y proporciona las direcciones IP de otros nodos para poder empezar a conectarse a ellos.

○ Enfoque centralizado: punto de fallo

○ Mantenimiento costoso: no puede fallar o toda la operación queda inoperativa

○ Un atacante puede aprender la topología de la red preguntando a este servidor

● Lista de nodos “habituales” embebidos en el software

○ Necesaria una actualización muy frecuente

**Distribución de ficheros // C-S vs P2P**

Q: ¿Cuánto tarda la distribución de un fichero de tamaño F de un servidor a N nodos?

Diagrama

Descripción generada automáticamente

**Distribución de ficheros // C-S**

▪Servidor: suponemos que el servidor envia N copias del fichero de forma secuencial:

• Tiempo para mandar una copia: F/u s

• Tiempo para mandar N copias: NF/u s

▪ Cliente: cada cliente i tarda en descargar el fichero F/di

• min(di ) tiempo de descarga del cliente más lento

Texto

Descripción generada automáticamente

**Distribución de ficheros // P2P**

▪ Servidor: debe subir al menos una copia antes de que la distribución P2P se puedo poner en marcha: F/u s

▪ Clientes: cada cliente debe descargar una copia: F/min(di ) u s Red di ui F

▪ Red: la red, como un todo, debe descargar un total de NF a una tasa máxima u s + Σui , considerando que todos los nodos transmiten a la vez

Imagen que contiene Diagrama

Descripción generada automáticamente

**P2P // Clasificación**

Clasificación en base a las técnicas de búsqueda de información:

● Centralizada: existe algún tipo de servidor (servidor de descubrimiento), o super-nodos (como los groups-leaders de KaaZaa) que ayudan en la localización de la información

● Descentralizada: no hay topología ni estructura específica, y la petición de información debe llegar a todos los nodos (flooding)

**Arquitectura centralizada // Napster**

Napster: primer ejemplo de red P2P centralizada

● Diseñada para la compartición de archivos musicales (MP3)

● Debido a su diseño centralizado, tenía información sobre los archivos que se compartían, por lo que fue denunciada por las compañías discográficas

● En 2002 fue obligada a cerrar

● Actualmente sigue existiendo como una plataforma de pago, al estilo de Spotify

**P2P // Arquitectura centralizada**

Alice busca un fichero (contenido en el nodo de Bob)

1. Alice entra en la red conectándose al servidor de descubrimiento, y registra su contenido

Diagrama

Descripción generada automáticamente2. Envia una petición buscando el fichero “Sweet child of mine”

3. El servidor le proporciona la dirección IP del nodo que la contiene

4. Alice se conecta directamente a Bob y descarga el contenido

**P2P // Arquitectura centralizada**

Desventajas**:**

● Punto único de fallo

● Rendimiento limitado

● Problemas de copyright en caso de compartir contenido ilegal

Características:

● Transferencia de datos descentralizada

● Búsqueda de información centralizada

**P2P // Arquitectura descentralizada**

En función de la topología de la red, las redes P2P descentralizadas se pueden clasificar en:

● Estructuradas: organizadas de acuerdo a una topología específica, con un protocolo que asegura una búsqueda eficiente de datos, incluso si el recurso es extremadamente raro (poco compartido)

● No estructuradas: no imponen una estructura particular, sino que se auto-organiza conforme los nodos se conectan a la red y establecen conexiones aleatorias entre ellos (ej.: Bitcoin, Gnutella, Kazaa)

**Descentralizada // No estructurada**

Debido a su estructura, tienen algunas ventajas:

● Son fáciles de “construir”, puesto que se auto-organizan. No es necesario un protocolo que ordene los nodos

● Como todos los nodos tienen el mismo papel (no hay super-nodos ni servidores de descubrimiento), estas redes son extremadamente robustas

Sin embargo:

● Esta (falta de) estructura limita el rendimiento de la búsqueda de información, que debe hacerse por inundamiento (flooding)

● Cuando un nodo quiere encontrar un dato en la red, la consulta de búsqueda debe ser propagada a través de la red para para encontrar tantos pares como sea posible que compartan los datos (elevado tráfico de señalización, uso de CPU)

● Además, al no existir correlación entre un nodo y el contenido gestionado por él, no hay garantía de que encuentre un par que tenga los datos deseados.

**No estructurada // Gnutella**

Uno de las primeras redes en no depender de un servidor central:

● Cada nodo se conecta a un número limitado de nodos que utilizan el protocolo Gnutella también

● Existen dos tipos básicos de mensajes:

○ Petición (query) : enviado por un nodo a sus nodos directamente conectados con la identificación de un objeto que buscan

○ Respuesta (query response): si un nodo que recibe una petición tiene el contenido solicitado, responde indicando su dirección IP y puerto de conexión.

1. Los pares envían mensajes QUERY a sus vecinos directos de la red.

2. Los receptores reenvían el mensaje de consulta a todos sus vecinos, que a su vez lo reenvían a todos sus vecinos, etc.

3. Cuando un nodo recibe un mensaje QUERY, comprueba si la palabra clave coincide con alguno de los archivos que tiene a su disposición para compartir.

4. Una vez encontrada una coincidencia, devuelve un mensaje QUERY\_HIT, que contiene el nombre y el tamaño del archivo coincidente, junto con su dirección IP y puerto.

5. El mensaje QUERY\_HIT sigue la ruta inversa a la del mensaje QUERY, por lo que utiliza las conexiones TCP preexistentes. Si se reciben varios mensajes de respuesta, el usuario decide qué archivo quiere descargar

6. El nodo inicial establece entonces una conexión TCP directa con el usuario deseado y envía un mensaje HTTP GET que incluye el nombre específico del archivo:

a. El archivo se envía con un mensaje de respuesta HTTP

b. Una vez recibido el archivo completo, se finaliza la conexión TCP directa

**Descentralizada // Estructurada**

● La red se organiza en una topología específica, normalmente utilizando variaciones del esquema de tablas hash distribuidas (DHT), que clasifican y organizan a los nodos utilizando pares (clave, valor).

● Así se consigue un alto grado de descentralización y, simultáneamente, eficacia y garantía de resultados.

Diagrama

Descripción generada automáticamente

Tabla

Descripción generada automáticamente

**DHT (Distributed Hash Table) // Descripción**

● Organizan tanto los nodos (direcciones IP) como las claves (identificadores de contenido), para vincular unos con otros, asignando un identificador en el rango [0, 2n − 1]

● El identificador se calcula utilizando una función hash:

○ node\_id = SHA256(Dir. IP, i)

○ key\_id = SHA256(Nombre clave)

● Ejemplo: Clave (“Guns ‘N Roses”)

○ key\_id = SHA256(“Guns ‘N Roses”) = 4ff5dbbbc250c2ba7eed50b90b2ef8c3e9b27b3a789d5e25a286d2d69c843218

**HT // Protocolo Chord**

● Los nodos se distribuyen en un anillo ordenado en función de su node\_id

● El contenido se almacena en el sucesor del nodo cuyo node\_id es igual o mayor a key\_id.

● Ejemplo: la clave K10 se almacenará en el nodo sucesor con ID igual o inmediatamente superior a 10 (N14).

**Chord // Búsqueda de información**

● El nodo N8 busca una información con clave K54:

○ Propaga su petición hasta que el nodo que contiene la información la recibe

○ Se conecta directamente a él y la descarga

● Aunque elegante y garantiza encontrar el contenido, esta solución tiene un rendimiento pobre:

○ En el caso peor, todos los nodos (N) de la red deben transmitir un mensaje. En término medio, se transmiten N/2 mensajes

**Chord // Fingers**

● Para mejorar la búsqueda, se utilizan una serie de “atajos” (denominados fingers) que reducen el número de nodos que deben ser consultados

● Cada nodo tiene su propia tabla de atajos, de forma que el número total de mensajes se reduce

● Las entradas para un nodo se calculan con un incremento exponencial: finger[i] = sucesor(node\_id + 2i ) mod n donde n es el número máximo de nodos en la red

Diagrama

Descripción generada automáticamente

**Chord // Gestión de nodos**

Cuando un nuevo node se une al sistema, los punteros de algunos nodos tendrán que cambiar, o las búsquedas no funcionarán correctamente.

Chord utiliza un protocolo de estabilización que corre periódicamente en background para actualizar estos puntos y la tabla de fingers.

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

Descripción generada automáticamente

Diagrama

Descripción generada automáticamente

Diagrama

Descripción generada automáticamente