1. Protocolos de aplicación

1.1. Arquitecturas de Red

1.1.1. Cliente servidor

Servidor:

- Servidor siempre en un host
- Dirección IP del servidor tiene que ser estática
- Si quieres más capacidad tienes que comprar más servidores

Cliente:

- No tiene que estar siempre conectado
- Puede tener IP dinámica
- No se comunica con otros clientes

$$t_{\text{C/S}} = \text{máx}\{\frac{Nf}{u_s},\ \frac{F}{\text{mín}\, d_i}\}$$

1.1.2. Peer to peer (P2P)

- Todos los nodos son como clientes y servidores
- Arquitectura distribuida: no hay posibilidad de control central, resistencia a las averías
- Redundancia de la información, equilibrio de la carga…
- No cuesta nada hacer crecer la red
- Fórmulas para la carga en arquitecturas P2P:

$$t_{\text{P2P}} = \text{máx}\{\frac{F}{u_s},\ \frac{F}{\text{mín}\, d_i},\ \frac{NF}{u_s + \sum u_i}\}$$

1.1.3. Híbridas P2P - C/S

- Problema de P2P: localización de un nodo o de la información
- Solución: añadir un servidor central que solo se encarga de poner en contacto a nodos de la red
  - Permite que algunos clientes que estén detrás de NAT puedan utilizar la red P2P
  - Skype

1.2. QoS en la comunicación entre procesos

- Distintas aplicaciones tienen distintos requisitos: velocidad, integridad, seguridad, retardo…
- Los **protocolos** definen tipos de mensajes (sintaxis y semántica) y garantizan distintos requisitos
- Dos protocolos principales a nivel 4 que determinan que requisitos puede ofrecer un protocolo: TCP y UDP

TCP	UDP
Orientado a conexión cada parte fija IP y puertos durante el establecimiento de la conexión.	**No orientado a conexión** Cada parte especifica IP y puerto de destino cada vez que envía datos (uso de `sendto` y `recvfrom`)
Transferencia confiable se garantiza la llegada de toda la información en orden.	**Transferencia no confiable** no se garantiza nada
Control de flujo el emisor no sobrecarga al receptor	**No hay control de flujo** el emisor no sabe como está el receptor
No provee garantías de ancho de banda	No provee garantías de ancho de banda, pero la transferencia de mensajes cortos suele ser más rápida (DNS).

1.3. API de sockets en C

Todo apartado que se conecte a la red utilizará estas dos funciones

- `int socket(direccionamiento, tipo, protocolo)` crea un nuevo socket
  - `direccionamiento` determina el formato de las direcciones, en internet IPv4 es `AF_INET`
  - `tipo` determina TCP (`SOCK_STREAM`) o UDP (`SOCK_DGRAM`)
  - `protocolo` espeficia el protocolo para un tipo (normalmente solo hay uno)
- `int bind(fd, struct sockaddr *local_addr, int addr_len)` registra un socket en un puerto y una IP (un servidor podría tener varias tarjetas de red y por tanto varias IPs)

- `local_addr` especifica IP y puerto, si se quiere escuchar en cualquier IP se pone `0.0.0.0`
- `addr_len` especifica la longitud de la dirección (importante para soportar varios tipos

En este punto un *cliente UDP* ya está listo para enviar con `sendto(fd, data, data_len, flags, dest_addr, dest_addr_len)` y recibir con `recvfrom(fd, buf, buflen, flags, source_addr, source_addr_len)`.
Un servidor UDP solo tendría que hacer `listen(fd, queue_size)` y proceder a recibir datos como un cliente (utilizar `recvfrom`).
Un cliente TCP tendría que conectarse con `connect(fd, dest_addr, dest_addr_len)` y luego ya podría enviar y recibir bytes como si de un fichero se tratara con `read` y `write` o con `send` y `recv`.
Un servidor TCP tendría que esperar conexiones con `accept(fd, source_addr, source_addr_len)` que es bloqueante.

- Se rellena la estructura `source_addr` con la dirección y puerto del cliente.

Tanto cliente como servidor cierran conexiones con `close(fd)`.

1.3.1. Arquitectura de servidores

- **Iterativos**: solo pueden trabajar con una conexión a la vez.
- **Paralelos**: trabajan con más de una conexión a la vez. Se pueden implementar de varias formas:
  - Hacer `fork()` o `pthread_create()` cuando llega una nueva conexión
  - Tener un *pool* de procesos o hilos creado previamente y asignar cada conexión entrante a un miembro del pool. El pool puede redimensionarse dinámicamente.

1.4. Protocolos con arquitectura Cliente/Servidor

Protocolo	Puerto(s)	Uso
HTTP/HTTPS	80/443 seguro	Páginas web
SSH	22	Terminal remota
FTP	21 (control) / otro (datos)	Transferencia de archivos
SMTP	25	Envío de correo (servidor - servidor)
IMAP	143/993 seguro	Consulta de correo (cliente - servidor)
POP3	110/995 seguro	Consulta de correo (cliente - servidor)
DNS	53	Traducción dominios -> IPs
DHCP	67 servidores / 68 clientes	Gestión de redes LAN (asignación de IPs dinámicas, configuración de DNS, gateways…)

1.5. Protocolos con arquitectura P2P (peer to peer)

Protocolo	Puerto(s)	Uso
BitTorrent	aleatorios	Transferencia de archivos
Chord		Tablas hash distribuidas

1.6. HTTP

- Protocolo de transferencia de hipertexto.
- Transfiere archivos identificados por URLs o URIs a través de internet.
- Modelo de transferencia: Una **petición - respuesta** por cada recurso que se quiera obtener. La petición siempre la manda el cliente y la respuesta el servidor.
- Normalmente a partir de un fichero principal con formato HTML se generan más peticiones para otros archivos (imágenes, audio) que debe ir incluidos al dibujar la página.
- **Modo no persistente** cada conexión solo se utiliza para una petición/respuesta.
  - Es el por defecto en la versión 1.0.
  - *Overhead* innecesario de establecimiento de conexión para cada recurso que se necesite.
- **Modo persistente** cada conexión se puede utilizar para más de una petición/respuesta.
  - Es el por defecto en la versión 1.1.

- Por separado de lo anterior, los navegadores pueden decidir establecer múltiples conexiones para acelerar la descarga de grupos grandes de recursos. Estas conexiones pueden utilizar cualquier versión.
- Por separado de lo anterior, los navegadores pueden utilizar *pipelining* que consiste en enviar a través de una misma conexión nuevas peticiones sin que se haya recibido la respuesta de la anterior. Es bueno porque aprovecha mejor los paquetes TCP.

Formato de la **cabecera de petición**:

```
VERB /resource?query_string HTTP/VERSION\r\n
Header-Name: header-value\r\n
[more headers]\r\n
\r\n
```

Como mínimo es necesario enviar las cabeceras de `Host` y `Allow`.
Formato de la **cabecera de respuesta**:

```
HTTP/VERSION STATUS-CODE STATUS-MESSAGE\r\n
Header-Name: header-value\r\n
[more headers]\r\n
\r\n
```

Como mínimo es necesario incluir la cabecera `Content-Type` y suelen enviarse además `Last-Modified`, `Server`, `Date` y `Content-Length`.

1.7. FTP

- Transferencia de archivos entre ordenadores.
- Utiliza dos conexiones:
  - Control: la crea el cliente y se envían comandos para autenticarse, inspeccionar y modificar directorios e iniciar transferencias de archivos.
  - Datos: se envían los datos que dicten los comandos enviados por la conexión de control.
    - Modo activo (por defecto): el servidor inicia las conexiones de datos. Da problemas de seguridad y no es posible si el cliente está tras un proxy, firewall o NAT.
    - Modo pasivo (lo solicita el cliente): el cliente inicia las conexiones de datos. Requiere que se habilite un rango de puertos en el servidor para poder recibirlas.
- Servicio de reinicio: algunos servidores soportan la creacion de *marcadores* para que en caso de fallo durante la transferencia de un archivo largo, sea posible transmitir de nuevo solo los datos a partir del último marcador recibido por el destinatario.

1.8. SMTP

- *Simple mail transfer protocol:* envío de correo entre hosts.
- Basado en ASCII, muy similar a HTTP.
- Cada email tiene una serie de cabeceras y diferentes *adjuntos*. El cuerpo del mensaje se codifica como un adjunto.

1.9. POP3 e IMAP

- Protocolos de consulta y gestión del correo.
- POP3: Más antiguo y menos sofisticado, permite la lectura y borrado de mensajes (no mantiene el estado fuera del borrado de mensajes).
- IMAP: Más moderno, permite la organización de mensajes en carpetas, marcar como leido/no leido… (en este sentido mantiene el estado).

1.10. DNS

- Resolución de nombres de dominio
- Sobre UDP entre cliente-servidor y sobre TCP entre servidor-servidor.
- Recursivo: el servidor resuelve las consultas adicionales necesarias para obtener la IP de un dominio.
- Iterativo: el cliente debe preguntar a los distintos servidores (local, root, TLD, dominio) para obtener la IP del dominio.

2. Criptografía y seguridad

- Principios básicos: CIA = Confidencialidad, Integridad y Autenticación.
  - Los cifrados nos garantizan la confidencialidad.
  - Las firmas digitales la integridad y autenticación.

2.1. Cifrados simétricos

2.1.1. Tipos de cifrados

- **Cifrados de flujo** (*stream*): cifran cada caracter del alfabeto de manera independiente pero el estado del cifrado puede afectar a cómo se cifra un caracter (ver *Vigenere*)
- **Cifrados de bloque:** cifran el texto plano en bloques de longitud dada, añadiendo padding en caso de que sea necesario. El cifrado define cómo se encripta un bloque. Tienen varios modos de operación atendiendo a cómo se aplica repetidamente la operación criptográfica definida para un bloque a todo un mensaje.
  - **EBC** (o Electronic CodeBook): se trocea el mensaje en bloques del tamaño que soporte el cifrado y se encriptan independientemente. No aporta **confusión** (la misma entrada se encripta siempre de la misma forma y por tanto es vulnerable a **análisis de frecuencias = ataque semántico**).
  - **CBC** (o Cipher Block Chaining): se combina el texto plano del bloque actual con el texto encriptado del bloque anterior mediante una XOR. Para el primer bloque se utiliza un vector de inicialización (IV) que debe ser aleatorio. Aporta **confusión** pero no mucha **difusión** el cambio de un bit en el texto plano provoca el cambio de ese bloque y del siguiente pero no más allá.
  - **CTR** (o Counter Mode): aporta más difusión que **CBC**.

2.1.2. Ejemplos de cifrados

- Cifrados clásicos (todos estos ejemplos son de flujo o *stream*):
  - **Cesar:** sustitución (rotación de las letras del alfabeto). Susceptible de ataques de fuerza bruta y análisis de frecuencias.
  - **Vigenere:** sustitución. Susceptible de ataques de fuerza bruta pero no de análisis de frecuencias.
  - **One-Time Pad:** (*Vernam*, s. XIX) basado en XOR. No susceptible de ser atacado de ninguna manera dada una clave precompartida secreta aleatoria, no reutilizada ni en totalidad ni en parte, y más larga que el mensaje lo que lo hace impráctico.
- Cifrados actuales (todos estos ejemplos son de bloque):
  - **DES:** de los primeros ejemplos de cifrado, hoy en día en desuso porque se puede romper en cuestión de horas. Utilizaba claves de 64 bits (56 útiles + 8 de paridad). **3DES:** es una versión posterior que consiste en aplicar DES 3 veces con claves distintas.
  - **AES:** cifrado en bloque compatible con los modos de operación anteriores. En los ejercicios las longitudes de bloque, IV y clave son iguales. En la práctica no tiene por qué, AES soporta distintas longitudes mientras sean múltiplos de 4 bytes.

2.1.3. Ejercicios sobre cifrados en bloque

- Hay que crearse la **tabla de traducción de bloques**. Esta tabla tiene dos columnas: bloque cifrado y bloque en claro. Si el bloque es de N bits, la tabla tiene 2^N filas y tiene que definir una biyección (si nos falta la codificación de un bloque sabemos que es la secuencia de N bits que falta).
- Si estamos en encadenamiento **ECB** obtenemos los bloques diréctamente de comparar el texto plano con el texto cifrado.
- Si estamos en **CBC** y no tenemos IV pero el tamaño de bloque es pequeño podemos probar con todos los IVs de ese tamaño a ver cuál nos da una biyección en la tabla de traducción de bloques.

2.2. Cifrados asimétricos

- Funciona a partir de un par de claves: lo que una encripta solo lo puede descifrar la otra.
- Una clave se da a conocer para poder recibir mensajes encriptados (clave pública). Otra se mantiene secreta (clave privada).
- Ventajas: no hay que compartir un secreto (la clave) previamente. Todo el mundo puede conocer la clave pública.
- Desventajas: más lentos, limitaciones en el tamaño del mensaje a encriptar.
  - Solución: **esquemas híbridos** = encriptar asimétricamente una clave simétrica.
- Aplicaciones:
  - Encriptación de claves simétricas para cifrado de mensajes largos. La clave simétrica se encripta con la clave pública del destinatario.
  - Firma digital: proporcionan integridad y autenticación. Consiste en la encriptación de un *hash* del mensaje con la clave privada del emisor para que todos los receptores que quieran pueden comprobar si el *hash* de un mensaje que han recibido es el mismo *hash* que el emisor dice que tenía un mensaje.

2.2.1. RSA

- **Generación de claves:**
  1. Elegir primos grandes p y q. Sea $n = p \cdot q$ (el módulo).
  2. Encontrar e que cumpla $1 < e < \varphi(n) = (p-1)(q-1)$ y e coprimo con $\varphi(n)$.
  3. Calcular $d = e^{-1} \mod \varphi(n)$
  4. La **clave pública** es el par (n, e) y la **clave privada** es el par (n, d).
- **Encriptación:** emisor encripta con la clave pública del destinatario. Solo se pueden encriptar $p < n$.

$$c = m^e \mod n$$

- **Descifrado:** utilizando la clave privada del receptor:

$$p = c^d \mod n$$

Atención: si queremos comunicación bidireccional cada agente tiene su par de claves.

2.3. Mecanismos para la compartición de claves

- Hasta ahora la desventaja de todos los cifrados es la compartición de las claves (la clave simétrica o la clave pública).
- Si la clave no se puede compartir no podemos tener integridad.
- Solución: **PKI = public key infrastructure**. Un sistema para asociar identidades (de agentes o de máquinas) a claves basado en criptografía de clave pública.
  - Requiere la compartición previa de la clave pública de una entidad en la que confiamos (autoridad certificadora, hay varias).

2.3.1. PKI

Problema que soluciona: un agente (empresa, máquina o individuo) quiere compartir su clave pública de manera que los destinatarios puedan estar seguros de que dicha clave le pertenece.

1. La entidad genera un par de claves RSA.
2. La entidad crea un **CSR (= certificate signing request)** que incluye su clave pública y metadatos que permiten identificarla (nombre de la entidad, país, etc.) y lo envía a una **RA (= registratión authority)**
3. La RA realiza la verificación de identidad pertinente (por ejemplo, comprueba la identidad de un individuo con el DNI). En caso de éxito, envía el CSR a la **CA (= certification authority)**.
4. La CA crea un certificado **X.509** que contiene los metadatos, la clave pública, fechas de expiración y su firma digital. Como los usuarios confían en la CA, una firma digital válida indica que la clave pública en efecto pertenece al agente que describen los metadatos. La CA envía el certificado al agente solicitante.
5. El agente solicitante facilita su certificado a los usuarios que quieran comunicarse con él de manera que la compartición de claves conyeva autenticación.

Puntos de fallo y soluciones

- Si la CA o la RA están comprometidas o son maliciosas PKI queda inservible. Error poco común pero ha pasado al menos una vez en la historia.

  [Solución (no entra): Certificate transparency logs. Las CAs publican listas con los certificados que emiten. Los agentes pueden comprobar que no se han emitido certificados en su nombre sin su consentimiento. Si un certificado no se ha publicado entonces no se acepta.]

- La clave privada del agente solicitante está comprometida. Entonces, como los certificados son públicos, cualquiera puede hacerse pasar por el agente solicitante.

  Solución 1: **CRL (= certificate revocation lists)**. Nada más obtener el certificado de la CA, el agente solicitante genera otro certificado firmado por él mismo que en caso de ser publicado invalida el certificado de la CA. La publicación se efectúa subiendo el certificado a una CRL. Los clientes deben comprobar si los certificados que utilizan para comunicarse con agentes están revocados (si hay un certificado en una CRL que invalide el que les proporciona el agente para comunicarse).

  Solución 2: **OCSP (= Online Certificate Security Protocol)**. Ya que la comprobación de las CRLs puede ser costosa (puede haber muchas y en ocasiones podemos no saber donde mirar), OCSP centraliza esta comprobación ofreciendo un servicio que lo hace por nosotros. Lo gestionan las **VA (= validation authorities)**.

Nota: en la práctica, las tres *authorities* suelen ser la misma.
Nota 2: cualquiera puede crear una CA (útil para certificados internos en empresas), la cosa está en que todo el mundo confíe en ella.

2.4. Seguridad de contraseñas

2.4.1. Gestión de contraseñas por parte de usuarios

- No utilizar patrones repetidos, ni añadir números o símbolos comunes al final de palabras, ni reutilizar la misma contraseña para varios servicios.
- Métodos para mejorar la seguridad:
  - Utilizar sistemás seguros para generar contraseñas manualmente: P.A.O. (Persona, acción, objeto), patrones de teclado…
  - Utilizar gestores de contraseñas que pueden crear contraseñas únicas y completamente aleatorias para cada servicio.
  - Utilizar autenticación multifactor: algo que sabes (contraseña), algo que tienes (móvil = código por SMS) y algo que eres (biométrica).

2.4.2. Gestión de contraseñas por parte de servidores

- Contraseñas en reposo (en la base de datos de un servidor). Maneras de guardarlas y qué pasa si nos roban la base de datos:

  Nota: si nos roban la base de datos, nosotros ya hemos perdido, la cosa es minimizar el daño a los usuarios teniendo en cuenta que probablemente utilicen la misma contraseña para todo.

  - Guardarlas en texto plano. Liada parda.
  - Guardarlas *hasheadas*. Menos liada parda porque no se puede obener la contraseña diréctamente. Las **rainbow tables** asocian contraseñas comunes a sus *hashes* con lo que puede ser fácil recuperar las contraseñas.
  - Guardar contraseñas con **salt y hash**. Consiste en generar una cadena aleatoria que se concatena a la contraseña antes de aplicar el *hash*. En la base de datos se guarda tanto el *salt* como el *hash*. Al concatenar el *salt* a la contraseña, conseguimos que esta nueva cadena ya no esté entre las contraseñas comunes y por tanto no se pueden utilizar rainbow tables.
- Contraseñas en tránsito (cuando se transmiten por redes). Maneras de transmitirlas y vulnerabilidades
  - En claro. Liada parda.
  - Cifradas. Mejor pero se sigue siendo vulnerable a ataques **MITM (= Man in the middle)** si la clave utilizada no se autentica. Un atacante podría interceptar el tráfico y hacer creer a cada extremo que se está comunicando con el otro suplantando sus claves públicas por otras suyas.
  - Cifradas con autenticación = **HTTPS = HTTP + TLS (antiguo SSL)**. No es posible suplantar la identidad del servidor porque el usuario verifica su identidad mediante PKI.
- Contraseñas en las personas. Posibles ataques de ingeniería social (manipular a las personas para que nos digan sus contraseñas): *phising*, …

2.5. Seguridad en redes

2.5.1. TLS = Transpor Layer Security

- Versión actual (v 1.3) soluciona los problemas de la versión original SSL.
- Basado en un esquema híbrido para la transmisión de mensajes. Tras el establecimiento de la conexión TCP se verifican las identidades mediante PKI y se comparte un *master secret* = unos cuantos bytes aleatorios a partir de los cuales se derivan todos los parámetros del protocolo (IVs, números de unicidad, claves simétricas, etc…). Además, cliente y servidor se ponen de acuerdo en los cifrados a utilizar.
- Dos funciones:
  - Encriptación: se cifra todo el tráfico a nivel 4 (TLS = Transpor Layer Security).
  - Autenticación: las claves públicas se autentican mediante PKI.

2.5.2. SSH = Secure Shell

- Una terminal remota que además permite la transmisión de tráfico arbitrario mediante túneles (por ejemplo, podríamos transmitir la interfaz de usuario, el protocolo GIT…).
- Se encarga cifrar dicho tráfico de manera simétrica.
- Para la autenticación del cliente hay varios modos:

- Usuario y contraseña (se envían cifrados).
- Clave pública (que ha de ser pre-compartida con el servidor)
- Restricción a un conjunto de IPs….

  [En realidad SSH no define la autenticación sino que la deja abierta para que usemos lo que queramos (LDAP, /etc/shadow, PKI, PAM, SAML…)]

- Para la autenticación del servidor se puede utilizar PKI o comprobar manualmente la clave pública que nos mande.

2.5.3. Firewalls

- Se definen reglas de permisión o prohibición sobre rangos o valores fijos de direcciones IP y puertos.
- Las reglas se aplican ordenadas y cuando hay coincidencia con una se aplica allow o deny y se deja de buscar. Por eso se suele poner una última regla que prohibe todo.
- Los firewalls con *filtrado con estado* tienen en cuenta el estado de la conexión.

descripción	action	protocol	source IP	dest IP	source port	dest port	flag bit	check connection
prohibir conexiones entrantes en el puerto 80 (HTTP)	allow	TCP	*	*	*	80	*	-
permitir tráfico entrante UDP desde dentro de una subred	allow	UDP	169.72.23.0/24	*	*	*	*	-
denegar tráfico saliente SMTP desde un equipo de la LAN	deny	TCP	10.0.2.23	*	*	25	*	-
denegar tráfico entrante a una subred que vaya dirigido a puertos protegidos	deny	*	*	10.0.2.0/24	*	< 1024	*	-
última regla que deniega el tráfico no permitido explícitamente	deny	*	*	*	*	*	*	-