ILERNA Online

Módulo 9

Programación de servicios y procesos

UF1: Seguridad y criptografía

1. Uso de técnicas de programación segura

1.1. Prácticas de programación segura

A la hora de desarrollar programas, es necesario tener presente que un **código limpio** aportará seguridad a la aplicación.

Para ello es preciso:

- Estar siempre informado de los diferentes tipos de vulnerabilidades, con lo cual es necesario estar al día en ciberseguridad.
- Explorar software libre para conocer cómo se trabaja la seguridad en distintas aplicaciones.

Normalmente, toda aplicación recibirá información por parte del usuario, por lo que es importante el tratamiento de la misma. Los datos, al ser una parte muy sensible, es lo que más se suele atacar.

Es necesario verificar siempre que los datos introducidos son válidos, tanto los parámetros, como la información, como los ficheros que se utilizan.

Cuando se trabaja con URL páginas web, hay que tener presente la **seguridad** y, por ello, comprobar si la URL es fiable o no. Los datos de las webs pueden ser alterados por un usuario, tanto su contenido como la configuración.

En muchas ocasiones, los propios navegadores web avisan de que no se debe confiar en ***cookies* de terceros**, ya que estas también pueden ser modificadas.

Otro de los puntos clave en la programación segura es la **reutilización de código**, una práctica común y muy aconsejada, pero siempre teniendo en cuenta que ese código ha sido revisado y no cuenta con problemas de seguridad.

También se considera práctica de programación segura **eliminar el código obsoleto** para que no interceda con el código bueno, ya que, en las revisiones posteriores, podría llevar a confusión. Por último, se considera fundamental utilizar **herramientas de software** para detectar fallos en el código.

En resumen, durante el desarrollo de un producto software, se **deben tener en cuenta** los siguientes aspectos:

- Los usuarios son los mismos que intentarán vulnerar la aplicación.
- Los archivos que se utilicen en la aplicación deben ser de solo lectura, para evitar que un usuario los modifique.
- Toda información sensible guardada en la base de datos, o que se transmita por la red, debe ir siempre cifrada.
- Comprobar que todas las llamadas al sistema se han realizado con éxito y detener la aplicación si no ha sido así.
- Utilizar las rutas absolutas de los ficheros que se necesiten.
- No abrir nunca terminales desde una aplicación, ni otro software que no sea fiable.

1.2. Criptografía de clave pública y clave privada

La **criptografía** es el conjunto de técnicas que cifran información, con el objetivo de que un usuario no sea capaz de entender el contenido, y, por lo tanto, conseguir confidencialidad en el mensaje.

Para conseguir esta confidencialidad, es necesario transformar el mensaje mediante un algoritmo matemático. La criptología no solo se encarga de cifrar el mensaje, sino también de descifrarlo para que el receptor sea capaz de entenderlo.

Antes de conocer los distintos algoritmos que existen, es necesario conocer algunos **conceptos importantes**:

- **Texto legible:** es el mensaje original.
- **Texto cifrado:** es el resultado de aplicar uno de los siguientes algoritmos sobre un texto legible.

Existen diferentes **tipos de algoritmos**:

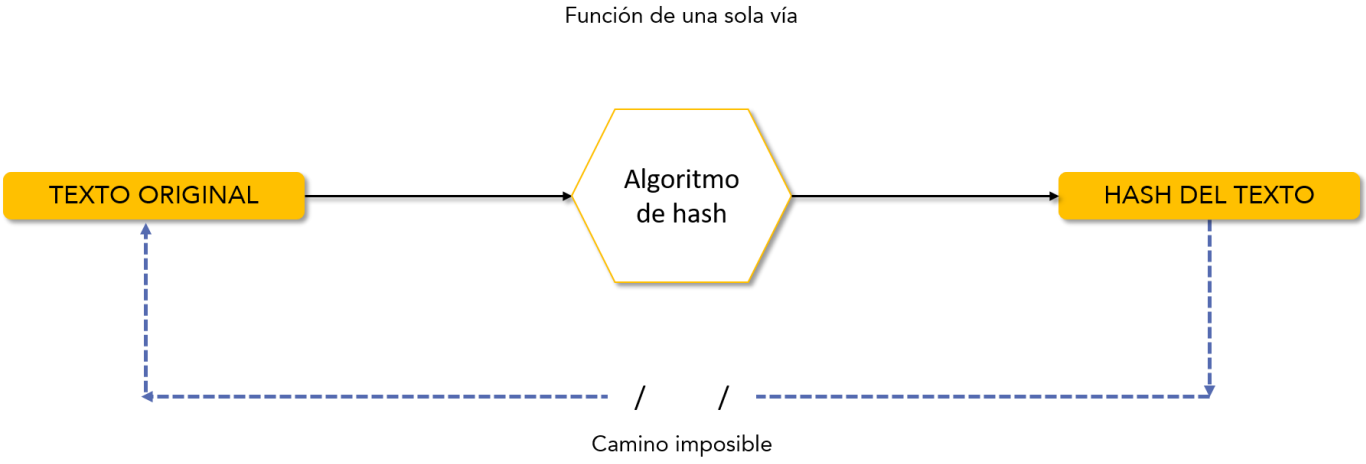
- **Cifrado Cesar**

Cifrado en el que se realiza una sustitución de cada letra por otra. El desplazamiento será el mismo para todas las letras del mensaje. Es un cifrado muy simple, pero bastante vulnerable. Los atacantes pueden conocer el mensaje mediante estadísticas.

Para **más información** de este cifrado:
https://es.wikipedia.org/wiki/Cifrado_C%C3%A9sar

- **Funciones Hash o funciones de una sola vía**

Una **función hash** es una función que, dada cualquier cadena de caracteres, los convierte en otra cadena de longitud fija.



Algoritmos de función *Hash* o funciones de una sola vía:

- MD5: genera resúmenes de 128 bits, es decir, de 32 símbolos hexadecimales. Se suele utilizar para proteger al usuario de troyanos o de cualquier otro software malicioso. Normalmente, cuando se descarga un software, se utiliza una aplicación externa que, mediante este algoritmo, genera un *hash* del instalador. Si coincide con el que ofrece el propio desarrollador, este no habrá sido alterado.

- SHA-1: genera resúmenes de 160 bits, es decir, de 40 símbolos hexadecimales. Desde hace algunos años no se considera un algoritmo seguro, aunque se ha actualizado a versiones posteriores, como es SHA-2, con diferentes longitudes en los resúmenes.

- Algoritmos de clave secreta o de criptografía simétrica

En este tipo de algoritmos, tanto el emisor como el receptor comparten una única clave. El mensaje se cifra y se descifra con esa única clave.

- DES: utiliza una clave de 56 bits, por lo que no se considera seguro. Cifra bloques de datos de 64 bits con la clave de 56, y después de varias iteraciones, muestra 64 bits de salida.

- **Triple DES**: el algoritmo es el mismo que DES, pero, debido a la necesidad de aumentar su seguridad, se aumentó la clave a 112 bits, aunque sigue cifrando bloques de 64 bits.

- **AES**: al igual que DES y Triple DES es un algoritmo cifrador de bloques. En este caso, bloques de 128 bits. La diferencia se encuentra en el tamaño de la clave, puede ser de 128, 192 y 256 bits.

- **Algoritmos de clave pública o de criptografía asimétrica**

En este tipo de algoritmos se generan dos claves, denominadas clave pública y clave privada. El receptor genera estas claves y muestra la clave pública al emisor. El emisor utiliza esta clave para cifrar el mensaje y, posteriormente, el receptor utilizará la clave privada para desencriptarlo.

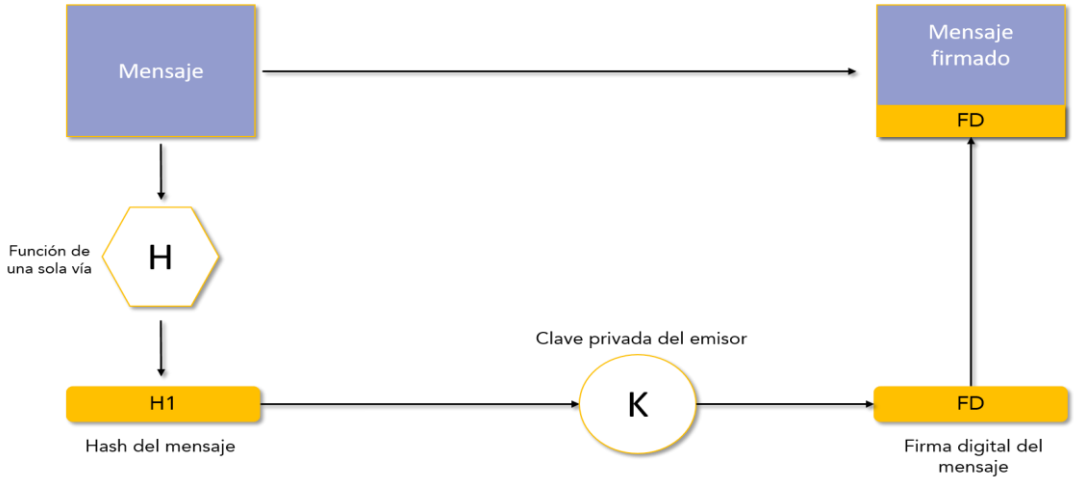
- **RSA**: la seguridad de este algoritmo radica en el problema de la factorización de números enteros. Tanto la clave pública como la clave privada se componen de un par de números.
    - **Pública** (*n*, *e*).
    - **Privada** (*n*, *d*).

Estos números son hallados mediante operaciones a partir de dos números primos, escogidos de forma aleatoria. Hay que tener en cuenta que es imposible conocer *d*, aunque conozcamos *n* y e. Este algoritmo es la base de la **firma digital**.

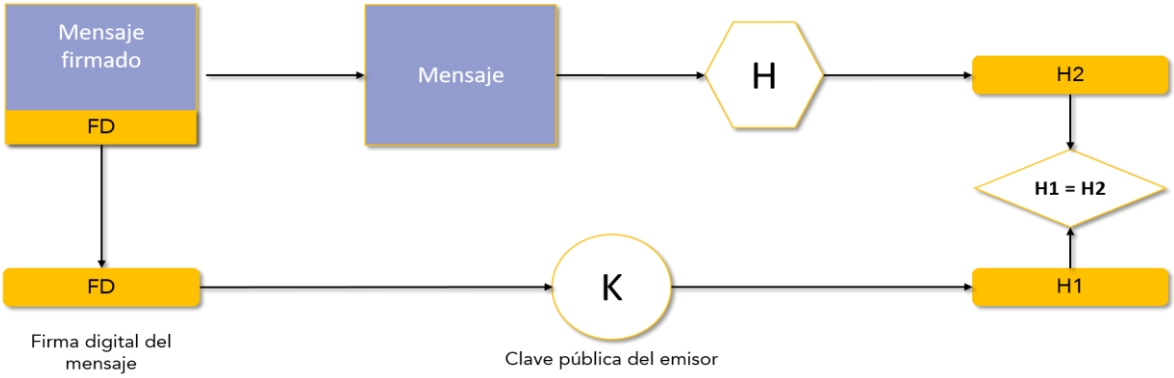
1.3. Principales aplicaciones de la criptografía

- **Seguridad en las comunicaciones**: es la principal aplicación, puesto que se encarga de que los canales en Internet sean seguros.
- **Identificación y autenticación en recursos y sistemas:** mediante la criptografía es posible validar el acceso de los usuarios a la firma digital, tanto si se realiza a través de contraseña, o mediante una técnica más segura.

La **firma digital** es el mecanismo mediante el cual el receptor es capaz de identificar al emisor. Asegura autenticación de origen y no repudio, además de integridad del mensaje.



La **comprobación de la integración del mensaje**, al igual que el de la autenticación del origen, se realiza comprobando la igualdad entre el resumen que se consigue al descifrar el mensaje y el resumen de la firma digital.



- **Certificación**

Este certificado es generado por una **Autoridad Certificadora** (AC), que también es la encargada de realizar la autenticación al usuario.

- **Comercio electrónico**

Mediante la criptografía es posible realizar operaciones sensibles por Internet, con la seguridad de que los datos no serán interceptados por terceros.

Un **certificado digital** es un documento que asocia los datos de identificación a una persona física, empresa o a un organismo, de manera que pueda identificarse en Internet.

1.4. Política de seguridad

Una política de seguridad es un **conjunto de normas que se utilizan para proteger un sistema**.

- **Integridad de los datos:** se asegura de que no hayan sido modificados por terceros, es decir, que se recibe el mensaje tal y como se envió.
- **Disponibilidad:** característica por la cual los datos se encuentran a disposición de quienes deban acceder a ellos.
- **Confidencialidad:** los mensajes solo podrán ser leídos por aquellas personas que han sido autorizadas para ello. De esta forma se garantiza la privacidad de los datos.
- **Autenticidad:** característica mediante la cual el receptor conoce la identidad del emisor.
- **No repudio:** el emisor no puede negar que ha enviado el mensaje, por lo que se evita que se culpe al canal de información de que la información no ha llegado.

Para que un sistema sea seguro **deben cumplirse todas estas características**. Además, hay que tener en cuenta que tanto *no repudio* como *confidencialidad* dependen de la autenticidad para poder cumplirse.

1.5. Programación de mecanismos de control de acceso

El control de acceso se encarga de comprobar si una persona u otra entidad tiene permisos necesarios para acceder al recurso que solicita.

Consta de **tres fases**:

- **Identificación:** es la parte del proceso en la que el usuario indica quién es. Por ejemplo, en el acceso al correo electrónico, el usuario indica su usuario.
- **Autenticación:** es la parte del proceso que realiza el sistema, en la cual se comprueba que el usuario es quien dice ser. En el mismo ejemplo anterior, el servidor de correo verifica que el usuario es quien ha indicado por la contraseña.
- **Autorización:** es la última parte del proceso. Si la autenticación ha resultado con éxito, el sistema ofrece la información requerida, o el acceso al recurso. Para acabar el ejemplo, sería el momento en el que el servidor muestra la bandeja de correo al usuario.

Se ha indicado anteriormente que en el proceso de autenticación el sistema debe verificar que el usuario es quien dice ser. Este proceso es necesario llevarlo a cabo mediante algún tipo de código con el cual únicamente pueda acceder este usuario.

Existen varias **formas:**

- **Contraseña.**
- **Biometría**: entre los que se engloban lectura de retina, huella dactilar, reconocimiento de voz, entre otros.
- **Tarjetas de identificación**: como puede ser el DNI electrónico.

Actualmente estos dispositivos están desbancando al sistema de seguridad por contraseña, ya que ofrecen una mayor seguridad y eliminan el factor humano para medir la seguridad.

A la hora de elegir una contraseña, el usuario suele escoger fechas claves, nombres o incluso su número de teléfono. Estos datos no son privados, por lo que la contraseña es vulnerable. Además, una contraseña poco segura puede ser *hackeada* mediante fuerza bruta en pocos segundos.

1.6. Encriptación de información utilizando protocolos criptográficos

Un protocolo criptográfico es un conjunto de reglas sobre la seguridad en la comunicación de los sistemas informáticos. Mediante estos protocolos se especifica cómo se debe utilizar este algoritmo.

Normalmente, los protocolos criptográficos **tratan los siguientes aspectos**:

- Establecimiento de las claves.
- Autenticación de entidades.
- Autenticación de los mensajes.
- Tipo de cifrado.
- Métodos de no repudio.

HTTPS (*Hypertext Transfer Protocol Secure*)

Es un **protocolo de la capa de aplicación** basado en el protocolo HTTP. Por así decirlo, añade seguridad a dicho protocolo. Utiliza un cifrado basado en SSL/TLS, que se estudiará en el siguiente apartado.

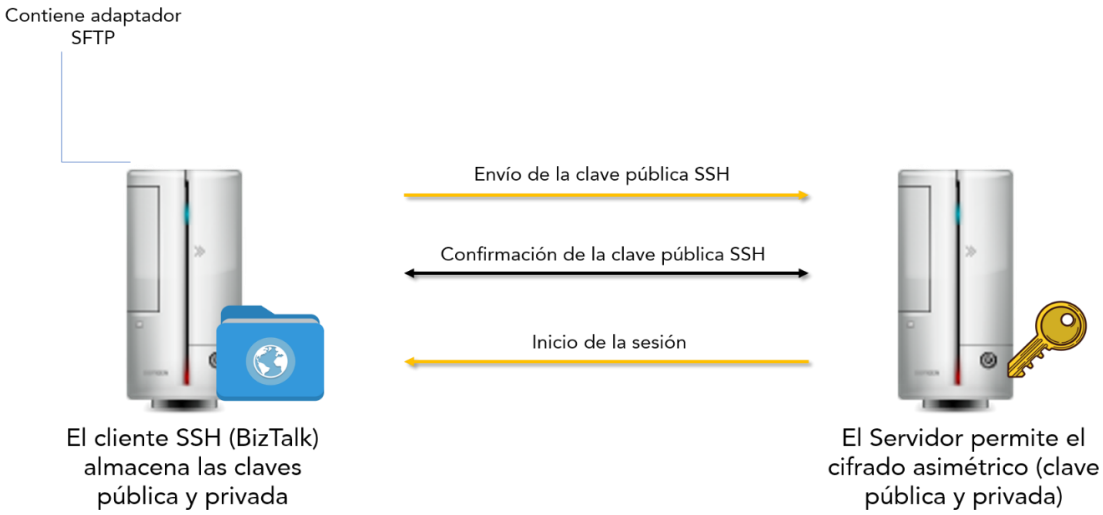
Mientras que el protocolo HTTP utiliza el puerto 80, HTTPS utiliza el 443. Otra diferencia fundamental se encuentra en la URL, que indica qué protocolo se está usando.

1.7. Protocolos seguros de comunicaciones

SSH. También denominado **intérprete de órdenes seguro**. Es un protocolo de seguridad implementado en la capa de aplicación del modelo **OSI**. Este protocolo utiliza el puerto 22 para la comunicación y la realiza de forma similar a **SSL**.

Un ejemplo de aplicación que utiliza este protocolo es ***Putty***. Para realizar una conexión, el cliente envía una petición al servidor por el **puerto 22**. Una vez que el servidor la acepta, envía la clave pública al cliente, para que este pueda cifrar sus mensajes. Por último, una vez que el cliente ha recibido esta clave, puede enviar los mensajes cifrados al servidor.

SSH utiliza tanto cifrado asimétrico como simétrico. Mediante el primero, garantiza la autenticidad del cliente y del servidor, mientras que con el cifrado simétrico se garantiza la confidencialidad y la integridad de los mensajes.



SSL y TLS. El protocolo TLS (*Transport Layer Security*, seguridad de la capa de transporte) actúa en la capa de transporte, al mismo nivel que los *sockets*, y por debajo de HTTP. **Suele utilizarse para la seguridad de otros protocolos de la capa de aplicación**, como pueden ser FTP o SMTP. Es solo una versión actualizada y más segura de SLL.

Ambos protocolos utilizan tanto cifrado simétrico como asimétrico. Debido a que el cifrado asimétrico es demasiado costoso, se utiliza el cifrado simétrico para el intercambio de los mensajes. Como el cifrado asimétrico es más seguro y garantiza la autenticación, el mensaje inicial se realiza mediante este tipo de cifrado. Por último, garantiza la integridad de los mensajes mediante funciones ***hash***.

Te resultará interesante la **lectura del siguiente artículo** *(en inglés)*:
http://blogs.mdaemon.com/index.php/ssl-tls-best-practices/

1.8. Programación de aplicaciones con comunicaciones seguras

Existen **diferentes clases** que sirven para cifrar datos en Java. Entre ellas se encuentran:

- Clase *Cipher*

String getAlgorithm *()*	Devuelve el algoritmo usado en ese objeto.
static Cipher getInstance *(String algorithm)*	Crea un objeto de la clase *Cipher* con el algoritmo recibido por parámetro.
void init *(int opmode, Key key)*	Inicializa el objeto para cifrar o descifrar con la clave recibida. El parámetro *opmode* puede ser *Cipher.ENCRIPT_MODE y Chiper.DECRIPT_MODE*
Byte *[]* **doFinal** *(byte [] input)*	Realiza la encriptación o desencriptación de los datos.

Para más información:
https://docs.oracle.com/javase/7/docs/api/javax/crypto/Cipher.html

- Clase *MessageDigest* para funciones *hash*

static MessageDigest getInstance *(String algorithm)*	Asigna el algoritmo de cifrado para el objeto.
byte*[]* **digest** *()*	Realiza el cálculo del resumen.
static boolean isEqual *(byte[] digestA, byte[] digestB)*	Compara dos resúmenes para comprobar si son iguales.
void update *(byte[] input)*	Actualiza el contenido del objeto.
void reset *()*	Elimina el contenido del objeto.

Para más información:
https://docs.oracle.com/javase/7/docs/api/java/security/MessageDigest.html

- Clase *KeyGenerator* para cifrado simétrico

static KeyGenerator getInstance *(String algorithm)*	Asigna el algoritmo de cifrado para el objeto.
SecretKey generateKey *()*	Genera la clave secreta.
String getAlgorithm *()*	Devuelve el algoritmo usado en ese objeto.
void init *(int keysize)*	Vuelve a generar la clave con el tamaño indicado.

Para más información:
https://docs.oracle.com/javase/7/docs/api/javax/crypto/KeyGenerator.html

- Clase *KeyPair* y *KeyPairGenerator* para cifrado asimétrico

Con la Clase *KeyPair* se crea un objeto que contiene una clave privada y una pública mediante el constructor ***KeyPair*** (*PublicKey publicKey, PrivateKey privateKey*). Es posible recoger estas claves con los siguientes métodos:

PrivateKey getPrivate *()*	Genera la clave privada.
PublicKey getPublic *()*	Genera la clave pública.

Para más información:
https://docs.oracle.com/javase/7/docs/api/java/security/KeyPair.html

KeyPair generateKeyPair *()*	Genera una pareja de claves.
KeyPair genKeyPair *()*	Genera una pareja de claves.
String getAlgorithm *()*	Devuelve el algoritmo que se ha utilizado para la generación del par.
static KeyPairGenerator getInstance *(String algorithm)*	Genera un objeto *KeyPairGenerator* con un par de claves.
void initialize *(int keysize)*	Vuelve a generar las claves con el tamaño indicado.

Para más información:
https://docs.oracle.com/javase/7/docs/api/java/security/KeyPairGenerator.html

1.9. Documentación de aplicaciones desarrolladas

En la documentación generada de cada aplicación desarrollada es necesario detallar la criptografía usada. En la mayor parte de las aplicaciones se trabaja con bases de datos, por lo que la información queda almacenada y podría ser vulnerada.

La primera parte de este proceso consiste en **identificar aquella información considerada sensible**.

Se considera **información sensible** aquella información privada de un usuario.

Tipos de información sensible:

- **Contraseñas**.
- **Datos personales**: como puede ser el DNI o la cuenta bancaria de un usuario. También el número de tarjeta de crédito.
- Incluso, en algunos casos, también es considerada información sensible la **dirección física**.

Toda esta información no es posible guardarla en una base de datos sin cifrar, y hay que tener en cuenta que, si la información navega a través de Internet, debe ir cifrada y mediante un protocolo seguro.

Todo esto es necesario reflejarlo en la documentación de la aplicación, centrándose en la información cifrada y en el tipo de cifrado que se ha realizado.

UF2: Procesos e hilos

1. Programación multiproceso

1.1. Caracterización de la programación concurrente, paralela y distribuida

La **concurrencia** es la propiedad por la cual los sistemas tienen la capacidad de ejecutar diferentes procesos en un mismo tiempo.

Un **proceso** es el conjunto de instrucciones que se van a ejecutar.

La programación concurrente es la que se encarga de ejecutar las tareas simultáneamente. Hay que tener en cuenta que toda ejecución simultanea debe tener unas características especiales, como, por ejemplo, la secuencia de ejecución, las comunicaciones entre los procesos y el acceso coordinado a los recursos.

Ventajas de la programación concurrente:

- Facilita el diseño orientado a objetos.
- Posibilita la compartición de recursos.
- Facilita la programación de aplicaciones en tiempo real.
- Reduce los tiempos de ejecución.

Cada programa se compone de diferentes instrucciones que pueden ser ejecutadas o no simultáneamente. ***Bernstein*** dividió las instrucciones en dos tipos, dependiendo de la operación realizada sobre ellas: de **lectura** y de **salida**.

Para que dos instrucciones se puedan ejecutar concurrentemente, es necesario que se cumplan estas tres **condiciones**:

- $L(S_i) \cap E(S_j) = \emptyset$
- $E(S_i) \cap L(S_j) = \emptyset$
- $E(S_i) \cap E(S_j) = \emptyset$

Veamos un **ejemplo**:

A partir del siguiente conjunto de instrucciones, hay que indicar las que se pueden ejecutar concurrentemente:

Instrucción 1 (I1) → a = x + z
Instrucción 2 (I2) → x = b – 10
Instrucción 3 (I3) → y = b + c

En primer lugar es necesario formar 2 conjuntos de instrucciones:

- **Conjunto de lectura** → Formado por instrucciones que cuentan con variables a las que se accede en modo lectura durante su ejecución.

  L(I1) = {x,z}
  L(I2) = {b}
  L(I3) = {b,c}

- **Conjunto de escritura** →Formado por instrucciones que cuentan con variables a las que se accede en modo escritura durante su ejecución.

  E(I1) = {a}
  E(I2) = {x}
  E(I3) = {y}

Para que dos conjuntos se puedan ejecutar concurrentemente se deben cumplir estas 3 condiciones:

- $L(Si) \cap E(Sj) = \emptyset$
- $E(Si) \cap L(Sj) = \emptyset$
- $E(Si) \cap E(Sj) = \emptyset$

Aplicando estas tres condiciones a cada pareja de instrucciones podemos descubrir cuales se pueden ejecutar concurrentemente y cuáles no.

Conjunto de I1 e I2→ $L(S1) \cap E(S2) \neq \emptyset$

$E(S1) \cap L(S2) = \emptyset$

$E(S1) \cap E(S2) = \emptyset$

Conjunto de I1 e I3→ $L(S1) \cap E(S3) = \emptyset$

$E(S1) \cap L(S3) = \emptyset$

$E(S1) \cap E(S3) = \emptyset$

Conjunto de I2 e I3→ $L(S2) \cap E(S3) = \emptyset$

$E(S2) \cap L(S3) = \emptyset$

$E(S2) \cap E(S3) = \emptyset$

Por lo tanto, las instrucciones que se pueden ejecutar concurrentemente son:

I1 e I3
I2 e I3

- **Problemas de la programación concurrente**

- **Exclusión mutua:** es el resultado de que dos procesos intenten acceder a la misma variable. Esto puede producir inconsistencia, puesto que un proceso puede estar actualizando una variable, mientras que otro proceso está leyendo la misma. Para evitar este problema, en la programación concurrente se utiliza la región crítica, que controla el número de procesos que se encuentra utilizando el recurso.

- **Abrazo mortal:** dos procesos se quedan bloqueados porque están esperando a los recursos que tiene el otro proceso. También es denominado *DeadLock* o *interbloqueo*.
- **Inanición:** un proceso se queda esperando a un recurso compartido, que siempre se le deniega. Sin este recurso el proceso no puede finalizar.

1.2. Identificación de las diferencias entre los paradigmas de programación paralela y distribuida

- **Programación paralela**

Es una técnica de programación en la que **muchas instrucciones se ejecutan simultáneamente**. Este tipo de programación permite dar solución a problemas de grandes dimensiones, para ello, divide en pequeños problemas uno más grande. Estas partes se pueden resolver de forma paralela, por lo que podrán ser ejecutadas a la vez que otras. Esta técnica se utiliza en equipos multiprocesadores en los que cada uno de ellos es el encargado de resolver una tarea. Todos los procesos que se encuentran en ejecución deben estar comunicados entre sí.

Ventajas de la programación paralela:

- Permite la ejecución de tareas de manera simultánea.
- Permite resolver problemas complejos.
- Disminuye el tiempo en ejecución.

Inconvenientes de la programación paralela:

- Mayor dificultad en la programación.
- Mayor complejidad en el acceso a los datos.

Existen diferentes mecanismos de intercambio de información, denominados **modelos**. Según su forma se pueden **clasificar** en:

- **Modelo de memoria compartida**

Este tipo de memoria permite ser accedida por múltiples programas, por lo que es un modo eficaz de transferencia de datos. Esto se consigue creando un espacio de acceso común por parte de cada uno de los procesos en la memoria RAM.

Hay que tener en cuenta que uno de los mayores problemas se da cuando existe un punto de acceso en común, como son las **secciones críticas**.

Es necesario establecer una serie de controles dentro de un punto de acceso a datos común, puesto que estos procesos intentarán realizar lectura y escritura de datos a la vez, dando lugar a datos incoherentes. **Existen diversos mecanismos de control** como son: semáforos, tuberías, monitores, etc. Estos permitirán el acceso de los hilos de manera individual, asegurando que las operaciones realizadas son de forma **atómica**, es decir, hasta que un hilo no termina, otro no puede acceder al mismo espacio de memoria.

- **Modelo de paso de mensaje**

Es el mecanismo más utilizado en la programación orientada a objetos. Cada proceso tiene definidas sus funciones y métodos, por lo que cada uno de ellos ejecuta individualmente las tareas y, si necesita datos de otro, realiza una petición de los resultados que necesita al propietario.

El principal inconveniente es que todos los procesos deben tener implementados métodos que puedan interpretar este intercambio de mensajes.

- **Programación distribuida**

Este tipo de técnica de programación permite que un conjunto de máquinas separadas físicamente, que están interconectadas entre sí por una red de comunicaciones, trabaje como una sola para dar solución a un problema. Cada una procesa de manera independiente una parte del total. Este conjunto de máquinas que operan interconectadas se conoce como ***grid***.

Esta programación se utiliza en la computación de altas prestaciones, en la que las máquinas son configuradas específicamente para cada tipo de escenario y se dan necesidades de computaciones de manera remota. Está enfocada a sistemas distribuidos cliente-servidor que requieren de escalabilidad y con una gran capacidad de tolerancia a fallos, puesto que cada uno opera de manera independiente y en caso de fallo no repercute en el de los demás. La comunicación entre estas máquinas se realiza mediante la llamada de procedimientos remotos (**RPC**), invocación remota de objetos (**RMI**) y ***sockets***.

Características de los sistemas distribuidos:

- **Capacidad de balanceo:** las propias máquinas son capaces de realizar una asignación de recursos concreta para cada proceso, de manera que otro proceso pueda hacer uso de una mayor cantidad de recursos si lo requiere.
- **Alta disponibilidad:** es una de las características más usadas, que permite a este tipo de programación una flexibilidad enorme, en caso de que haya un fallo automáticamente los servicios son asignados en otros servidores.

De la misma manera que si la previsión de recursos necesarios se queda justa, es posible que de manera automática se añada una nueva máquina con las mismas características, para poder hacer frente a la computación de distintos procesos.

Por el contrario, este tipo de programación también cuenta con algunos inconvenientes como puede ser un aumento de la complejidad, ya que es necesario un determinado tipo de software. Asimismo, se incrementa el riesgo de fallos en la seguridad, como pueden ser los ataques de denegación de servicios con sobrecargas de peticiones remotas al servidor.

Ventajas de la programación distribuida:

- Permite escalabilidad, capacidad de crecimiento.
- Permite la compartición de recursos y datos.
- Mayor flexibilidad.
- Alta disponibilidad.

Inconvenientes de la programación distribuida:

- Pérdida de mensajes.
- Está expuesta a diferentes ataques para vulnerar su seguridad.

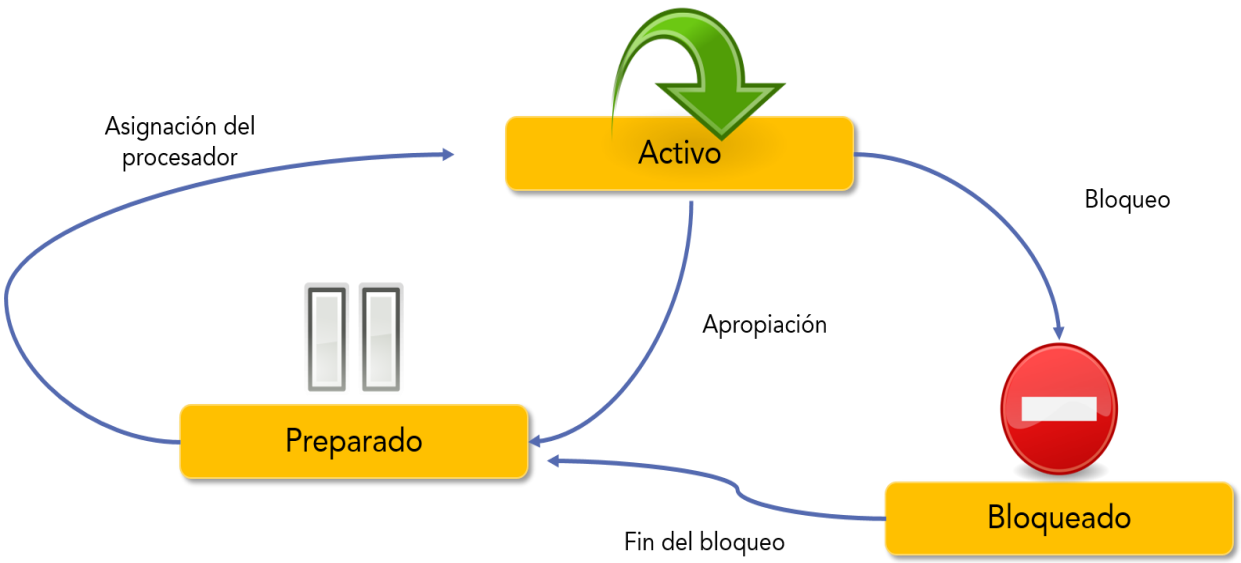
1.3. Identificación de los estados de un proceso

En primer lugar, es necesario definir qué es un **proceso**. Un proceso es un **conjunto de instrucciones que se van a ejecutar dentro de la CPU**.

Todo proceso está caracterizado por un indicador que define la situación o estado en el que se encuentra. Al menos existen **tres estados** diferentes:

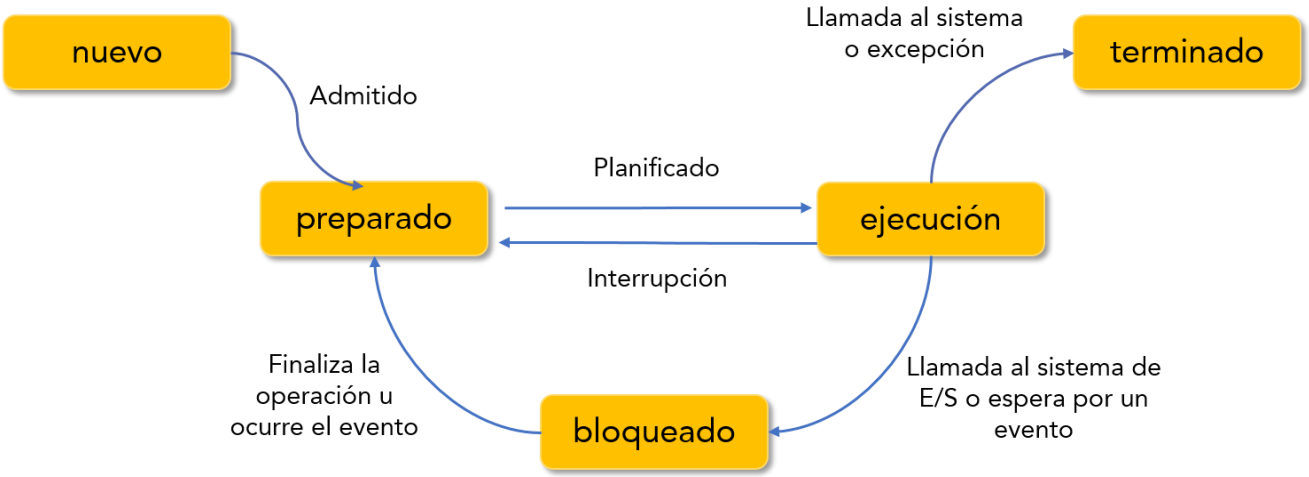
- **Activo o en ejecución:** aquellos procesos que han sido asignados para ejecutarse en el procesador.
- **Bloqueado:** aquellos procesos que han interrumpido su ejecución, y, por lo tanto, se encuentran actualmente a la espera de que termine la operación que los ha dejado bloqueados.
- **Preparado:** aquellos procesos que se encuentran disponibles para entrar a la CPU y ejecutarse.

Durante la vida de un proceso, se realizan transiciones entre los diferentes estados. En el siguiente diagrama se muestran las distintas **transiciones permitidas**:



- **Transición bloqueo:** cuando un proceso se está ejecutando y produce una llamada al sistema, debe quedarse como bloqueado para evitar consumir CPU. Por ejemplo, cuando un proceso requiere información y pide que se lean datos. En este caso, debe esperar a que se complete esa operación.
- **Transición apropiación:** cuando un proceso se encuentra en ejecución y el gestor de procesos indica que debe dejar de ejecutarse. En este caso, el proceso sale de la CPU hasta que puede volver a estar activo. En el próximo apartado se verán las diferentes técnicas por las cuales un proceso realiza esta transición.
- **Transición asignación:** cuando un proceso entra a ejecutarse en la CPU.
- **Transición fin de bloque:** cuando un proceso está esperando a que acabe la operación por la cual ha pasado a estar en el estado de bloqueo y pasa directamente a seguir su ejecución.

En **Unix** existen más estados para los procesos que los que se han explicado anteriormente. En este diagrama se pueden observar las transiciones entre ellos:



Cabe destacar que existen varios **estados nuevos**, como pueden ser:

- **Nuevo:** aquellos procesos que aún no han sido elegidos para iniciar su procesamiento.
- **Terminado:** aquellos procesos que han finalizado su ejecución.
- **Zombies:** aquellos procesos que han finalizado su ejecución, pero no han liberado los recursos que han utilizado.

El sistema operativo debe agrupar la información de todos los procesos del sistema. Esta información se encuentra en el bloque de control del proceso **(BCP),** que se crea a la vez que el proceso. Cuando el proceso es eliminado se borra también toda esta información.

Esta información se refiere al identificador del proceso, su estado, prioridad, recursos y permisos asignados.

1.4. Ejecutables. Procesos. Servicios

- **Ejecutables**

Son archivos binarios que contienen un conjunto de instrucciones en código fuente que el compilador ha traducido a lenguaje máquina. Este tipo de archivos suele contener instrucciones de llamada a las funciones del sistema operativo.

Este tipo de archivos son incomprensibles para el ser humano. Algunos de los más comunes son **.bat, .com, .exe y .bin**.

- **Procesos**

Conjunto de instrucciones que ejecutará el microprocesador, es lo que se entiende como un **programa en ejecución**. Los procesos son gestionados por el sistema operativo, que es el encargado de crearlos y destruirlos. Requieren de unos recursos de memoria que reservan para su ejecución.

En los sistemas operativos multihilo es posible crear hilos y procesos. La diferencia reside en que un proceso solamente puede crear hilos para sí mismo, y en que dichos hilos comparten toda la memoria reservada para el proceso.

Los procesos están caracterizados por su estado de ejecución en un momento determinado. Estos procesos son el valor de los registros de la CPU para dicho programa.

- **Servicios**

Es un tipo de proceso informático que posee unas características especiales, ya que se ejecutan en segundo plano y no son controlados por el usuario. Pueden permanecer de manera continuada en ejecución dentro del sistema y carecen de interfaz gráfica.

La mayor parte de estos servicios están destinados a realizar tareas de mantenimiento del sistema operativo de forma periódica.

1.5. Caracterización de los hilos y relación con los procesos

El término **hilo** es un concepto reciente y que se podría definir como una **ejecución que forma parte de un proceso**.

Un proceso puede contener un hilo o múltiples hilos de ejecución. Un sistema operativo **monohilo** (un solo hilo de ejecución por proceso) constará de una pila de proceso y una pila de núcleo. En un sistema **multihilo** cada hilo consta de su propia pila de hilo, un bloque de control por cada hilo y una pila del núcleo.

Un hilo posee las siguientes **características**:

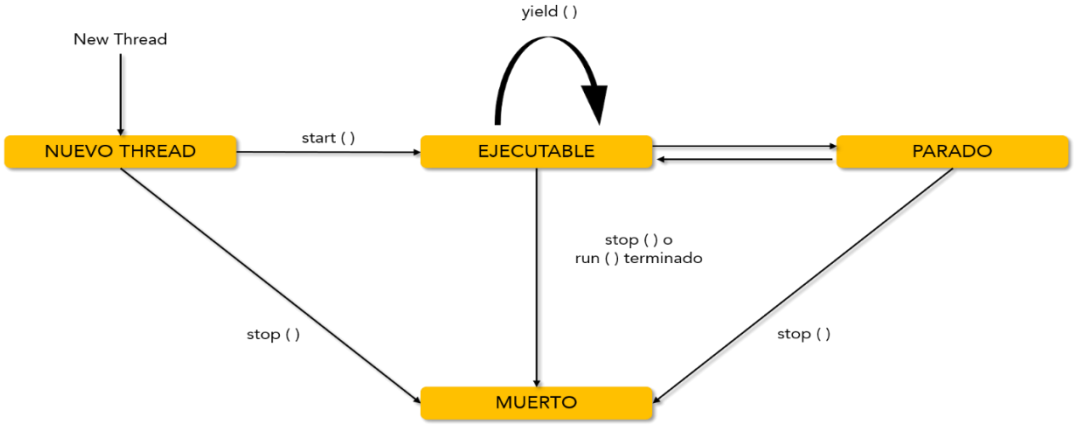
- Contador de programa.
- Juego de registros.
- Espacio de pila.

Los hilos dentro de una misma aplicación disponen de las siguientes **zonas comunes**:

- Sección de código.
- Sección de datos.
- Los recursos del sistema operativo.

Un hilo, al igual que un proceso, está caracterizado además por su estado. Los estados en los que se puede encontrar un hilo son:

- Ejecución.
- Preparado.
- Bloqueado.



Para realizar la transición de un estado a otro existen **cuatro operaciones** básicas:

- **Creación:** cuando se crea un nuevo proceso automáticamente se crea un hilo asociado a dicho proceso.
- **Bloqueo:** un hilo debe esperar a que un evento indique su ejecución. De esta forma el procesador ejecutará otro proceso que esté preparado.
- **Desbloqueo:** evento que indica la salida de un proceso del estado de bloqueo y pasa a la cola de preparados.
- **Terminación:** cuando un hilo finaliza, se liberan los recursos que estaba usando.

En la mayor parte de los casos, los hilos realizan las mismas operaciones que los procesos. **La existencia de un hilo está vinculada a un proceso**. Los hilos permiten que la comunicación entre los distintos procesos sea más rápida, además, se tarda un tiempo menor en crear un nuevo hilo que un nuevo proceso.

1.6. Programación de aplicaciones multiproceso

A la hora de ejecutar una aplicación, el sistema operativo se encarga de asignar en la CPU las distintas tareas que se deben realizar. En muchas ocasiones existen aplicaciones que necesitan que estas tareas se realicen de forma simultánea. De esta necesidad sale el concepto de **multiproceso**, que consiste en el **uso de dos o más procesadores para la ejecución de varios procesos**.

Gracias a esto es posible **mantener en ejecución** distintos programas a la vez.

Existen diferentes mecanismos a la hora de programar este tipo de aplicaciones multiproceso y el más utilizado es la realización, en segundo plano, de tareas que no se encuentran en la memoria principal. Esto permite asignar un mayor número de recursos a aquellos procesos que lo necesitan.

Estas tareas que no se están ejecutando en primer plano suelen ser asignadas a subprocesos del sistema operativo que son iniciados y detenidos de forma controlada. Esto aumenta la velocidad de lectura y escritura de datos para generar una respuesta a la solicitud de un usuario.

1.7. Sincronización y comunicación entre procesos

Tal y como se ha explicado en el apartado anterior, es posible trabajar con procesos que se ejecutan de manera concurrente. Esto ofrece una **mayor velocidad y mejora la comunicación** con la interfaz de usuario. Es muy importante tener en cuenta que se trata de operaciones con datos que acceden, en muchas ocasiones, a los mismos espacios de memoria. Es necesario que exista una comunicación entre procesos para no bloquear el acceso de otros.

La **sincronización entre procesos** es fundamental para conseguir una correcta funcionalidad y para ello se suele hacer uso de señales que definen el estado que tendrá cada uno de esos procesos durante un periodo de tiempo determinado. Estas señales son enviadas de forma independiente a cada proceso y cada señal define un comportamiento.

Las más **comunes** son:

- ***SIGKILL*:** se usa para terminar con un proceso.
- ***SLEEP*:** suspende el proceso durante una cantidad de segundos.
- ***KILL*:** envía una señal.
- ***SIGINT*:** se envía una señal a todos los procesos cuando se pulsan las teclas *Ctrl+C*. Se utiliza para interrumpir la ejecución de un programa.

La comunicación entre procesos se consigue utilizando los mecanismos ya detallados en apartados anteriores y los más importantes son: **tuberías, paso de mensajes, monitores o buzones**, entre otros. Estos asegurarán que el intercambio de datos sea correcto y no existan problemas en la persistencia de datos. Estos procesos se pueden bloquear a la espera de eventos y desbloquear de la misma manera.

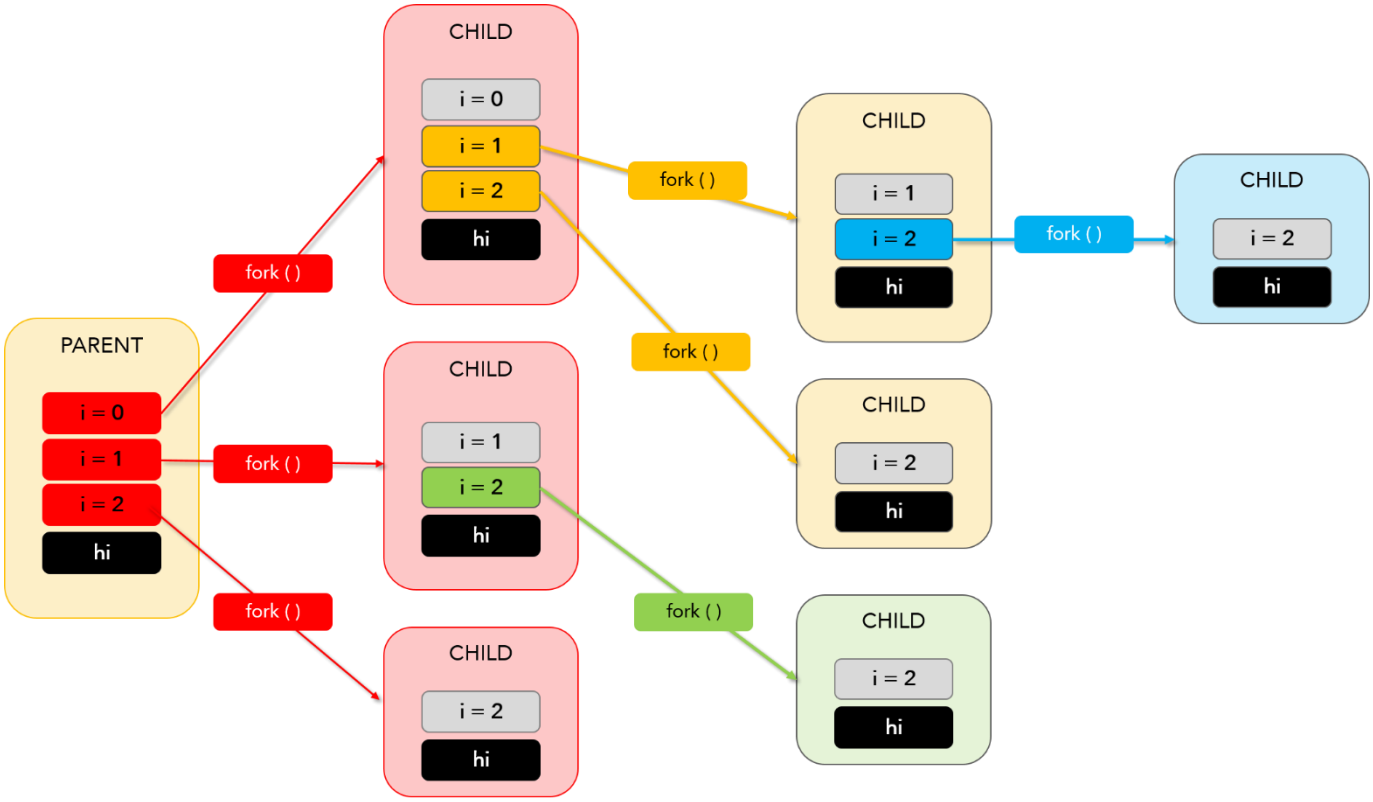
Los **hilos** agilizan las tareas que tienen los procesos, por lo que es importante considerar la posibilidad de que un hilo permanezca bloqueado de forma continuada. Como los hilos no son independientes, sino que **forman parte todos del mismo proceso**, el proceso podrá llegar a bloquearse por completo, así como todos los hilos que componen ese proceso.

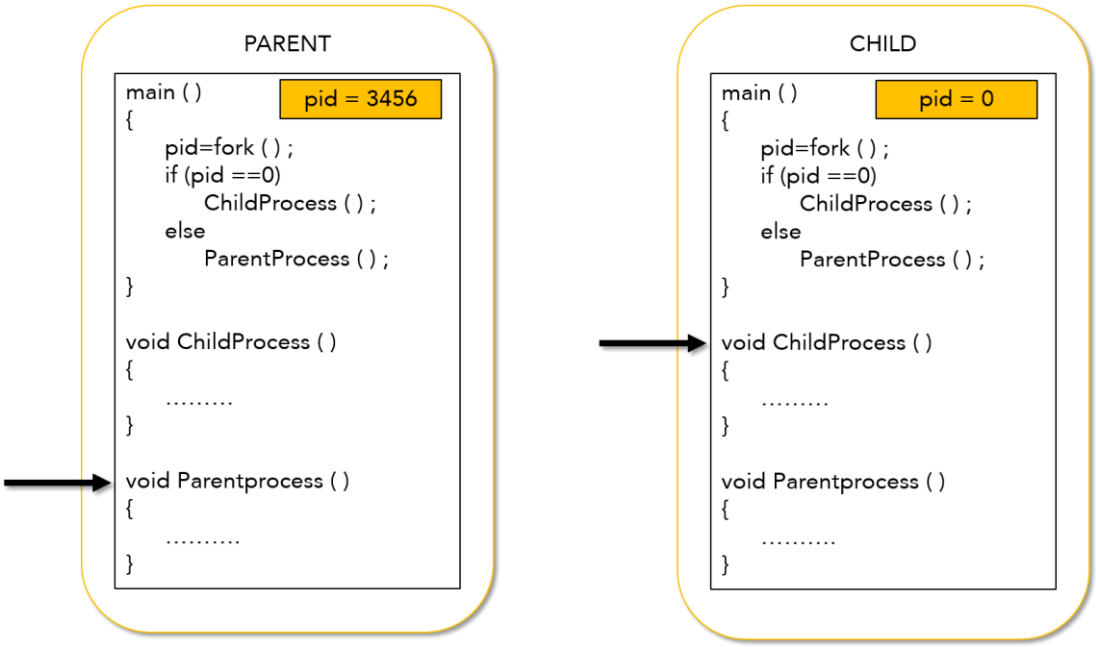
1.8. Gestión de procesos y desarrollo de aplicaciones con fines de computación paralela

La **gestión de procesos** está a cargo del sistema operativo, al igual que la creación de los mismos, en función de las acciones ejecutadas por el usuario.

Un nuevo proceso creado es el encargado de solicitar la creación de otros procesos. Estos nuevos procesos creados dependen del proceso inicial que los creó. Estos conjuntos de procesos dependientes creados reciben el nombre de **árbol de procesos**. Se caracterizan por tener un identificador único y un nombre que los distingue de los demás. El proceso creador se denomina **Padre** y el proceso creado **Hijo**. De la misma forma, estos procesos hijos pueden crear a su vez nuevos hijos. Estos procesos comparten los recursos de la CPU.

Las funciones que permiten crear un proceso son ***createProcess()*** para Windows y ***fork()*** para Linux. Son creadas en el mismo espacio de direcciones y se comunican entre sí mediante lectura y escritura.





Un proceso padre puede finalizar la ejecución de un proceso hijo en cualquier momento, aunque por lo general, es el hijo quien informa al proceso padre de que ha terminado su ejecución solicitando su terminación.

Es importante comprobar que los procesos han terminado de una forma ordenada, es decir, que los procesos hijos finalizan antes de hacerlo el padre. Para obligar a los procesos hijos a finalizar su ejecución se puede hacer uso de la orden ***destroy()*** que realizará una terminación en cascada de todos los procesos hijos.

En lenguajes como *java* ya existe un mecanismo automático denominado ***garbage collector*** (recolector de basura) que se encarga de liberar los recursos cuando un proceso finaliza. En otros lenguajes, un proceso que finaliza su ejecución de forma correcta liberará sus recursos al finalizar su ejecución por medio de la operación ***exit()***.

El control y gestión ordenado de estos procesos es muy importante en la computación paralela. El desarrollo de aplicaciones que gestionan las vías de comunicación entre distintos procesos de forma eficiente permite incrementar mucho la productividad de un software. **Es necesario separar qué tareas requieren de un mayor número de recursos y cuáles se pueden realizar en segundo plano, permitiendo una comunicación fluida con la interfaz de usuario**. De esta forma se evita una gran cantidad de tiempo de espera por la respuesta de algunas de las peticiones generadas por los usuarios durante el uso de la aplicación.

1.9. Depuración y documentación de aplicaciones

En Linux existe el comando ***ps***. Su función es mostrar la información de los procesos que se encuentran activos en el sistema.

```
prueba@PRINCIPAL:~$ ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
prueba      3646    3640  0 12:58 pts/11   00:00:00 bash
prueba      4178    3646  0 13:12 pts/11   00:00:00 ps -f
```

Información:

- **UID:** usuario del proceso.
- **PID:** identificador del proceso.
- **PPID:** PID del padre del proceso.
- **C:** uso del procesador.
- **STIME:** hora de inicio del proceso.
- **TTY:** terminal asociado.
- **TIME:** tiempo de ejecución del proceso.
- **CMD:** nombre del proceso.

2. Programación multihilo

2.1. Elementos relacionados con la programación de hilos. Librerías y clases

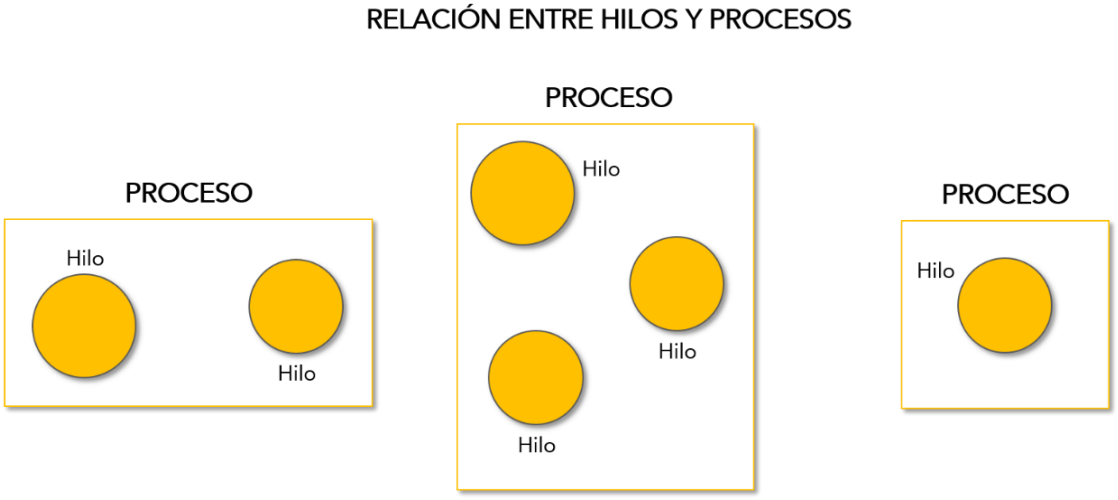
- Clase *Thread*

void start ()	Inicializa el hilo.
void run ()	Comienza la ejecución del hilo.
static Thread currentThread ()	Devuelve la referencia del hilo que se encuentra en ejecución.
long getId ()	Devuelve el identificador del hilo.
static void sleep (long milis)	El hilo se detiene durante el número de milisegundos especificado.
void interrupt ()	Interrumpe la ejecución del hilo.

Para más información:
https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

2.2. Gestión de hilos

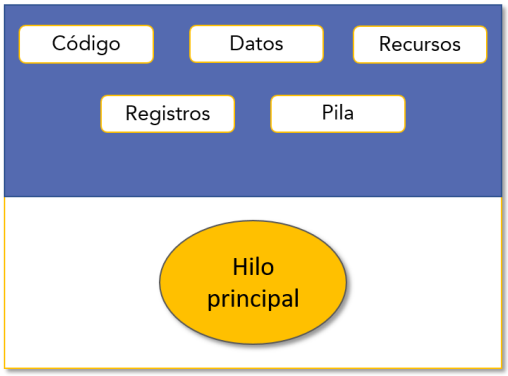
Un proceso es cualquier programa en ejecución y es totalmente independiente de otros procesos. Un proceso puede tener varios hilos de ejecución que realizarán subtareas del proceso principal que los ha creado.



Un **hilo** es un **conjunto de tareas que se ejecutan por el Sistema Operativo.** También se denominan **hebras** o **subprocesos**.

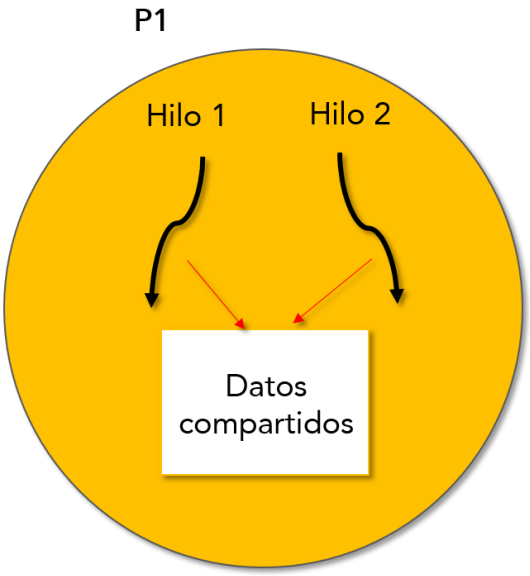
Los hilos contienen información propia, como, por ejemplo, su identificador y toda aquella información que sea necesaria para que la aplicación pueda comunicarse con el sistema, es decir, el contador del programa, la pila de ejecución y el estado de la CPU.

RECURSOS COMPARTIDOS POR HILOS



Proceso con un único thread

Proceso con varios threads



Mientras que los procesos se caracterizan por no compartir la memoria con la que trabajan y ser independientes entre ellos, lo hilos no son iguales. Estos últimos suelen compartir la memoria con la que trabajan, puesto que pertenecen al mismo proceso.

Un ejemplo que puede ayudar a entender este concepto es un **navegador web.** El navegador web es un proceso con un identificador. Dentro de ese navegador es posible crear diferentes pestañas que comparten la memoria.

Cuando un hilo modifica un dato en la memoria, el resto de hilos del proceso pueden obtener este valor modificado.

El uso de hilos en lugar de procesos tiene bastantes **ventajas**, entre las que se destacan dos:

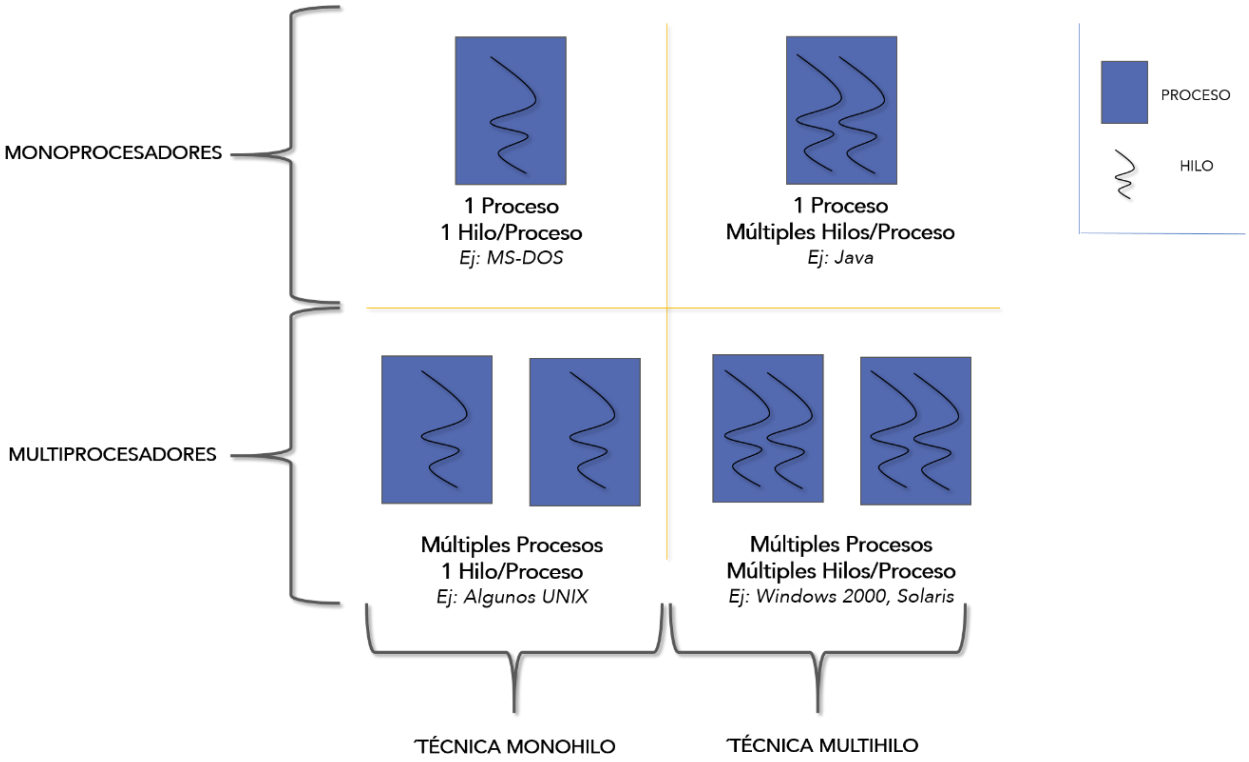
- La **creación** de un hilo tiene un coste menor que la creación de un proceso, puesto que no es necesario reservar memoria para cada uno de ellos porque la comparten.
- Es más rápido **cambiar** de un hilo a otro que de un proceso a otro.

Estos hilos **suelen ser utilizados la mayor parte de las ocasiones para llevar a cabo tareas en segundo plano**, puesto que se reduce mucho el tiempo de realización de diversas tareas de forma simultánea.

Java es uno de los lenguajes de programación que permite trabajar con distintos hilos en el desarrollo de una aplicación.

2.3. Programación de aplicaciones multihilo

A la hora de desarrollar un software es necesario tener en cuenta qué tiempo de ejecución debe de tener como máximo una aplicación. Además, es importante medir el consumo de recursos que esta ha utilizado. Cuando existe solo un proceso, existe la posibilidad de que el consumo de memoria *RAM* y recursos del procesador por parte del proceso en ejecución, sobrecargue el sistema.



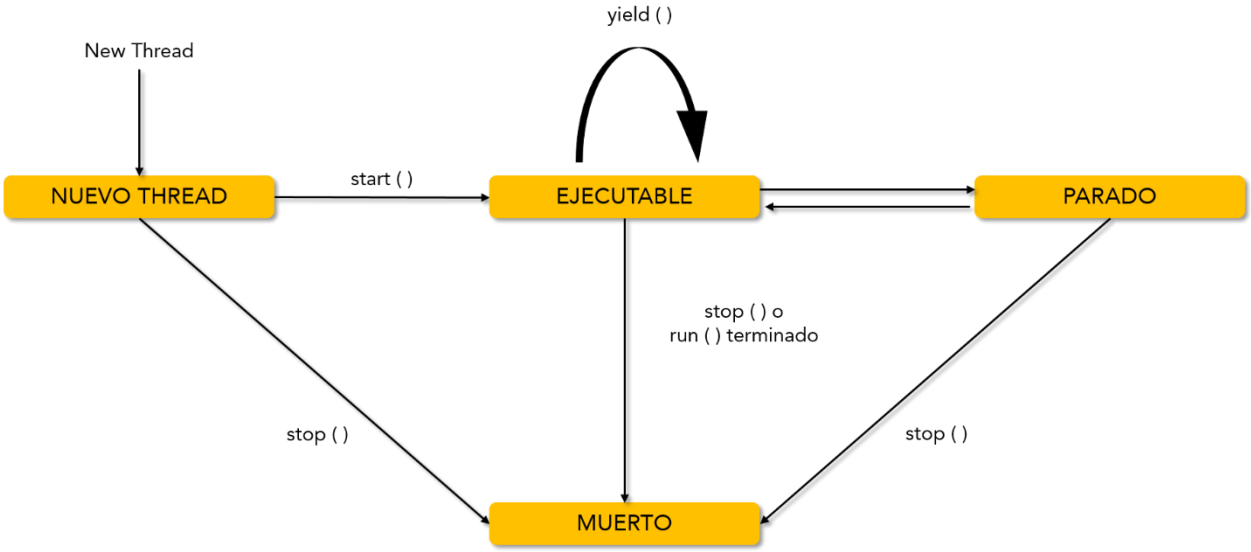
Como solución a esto **se han creado los hilos de ejecución que permiten ser ejecutados dentro de una aplicación de forma paralela y haciendo uso de otra parte del código**. De esta forma, es posible ejecutar un hilo en primer plano que permita la comunicación con el usuario, y, en segundo plano, hilos que vayan ejecutando las diferentes tareas de carga de una aplicación. Con esto se consigue que el rendimiento del equipo esté más optimizado.

Hoy en día **la programación de aplicaciones con estas características es fundamental**.

Además, permiten al usuario trabajar con diferentes aplicaciones al mismo tiempo sin penalizar el rendimiento del equipo. Esto **posibilita una gran cantidad de opciones** como, por ejemplo, el trabajo en múltiples escritorios con diversas aplicaciones de forma paralela.

2.4. Estados de un hilo. Cambios de estado

Al igual que los procesos, los hilos tienen un ciclo de vida en el que van pasando por diferentes estados.



Los **estados que pueden tener un hilo** son:

- **Listo**: se ha creado el hilo, pero aún no ha comenzado su ejecución.
- **Ejecución**: cuando un objeto llama al método ***start()*** comienza la ejecución del hilo, es decir, se empiezan a ejecutar las instrucciones que se encuentran en el método ***run()***.
- **Bloqueado**: el hilo se ha parado pero puede volver a ejecutarse. Normalmente, el hilo se bloquea por alguna de las siguientes acciones:
  - Se invoca al método ***sleep()***.
  - El hilo queda a la espera de que finalice una operación de entrada/salida.
  - Se invoca el método ***wait()***. Cuando un hilo queda bloqueado por este método, otro hilo debe invocar el método ***notify()*** o ***notifyAll()***, dependiendo de si quiere despertar un único hilo, o todos aquellos que estén bloqueados, respectivamente.
  - Otro hilo, externamente, invoca al método ***suspend()***. Cuando un hilo llama al método *suspend(),* el hilo bloqueado debe recibir el mensaje ***resume()*** para poder volver a la ejecución.

- **Muerto**: el hilo muere, ya sea porque ha finalizado el método ***run()*** o porque externamente han decidido pararlo.

Los **cambios de estado de un hilo** vienen dados por las siguientes situaciones:

- **Creación**: cuando se crea un proceso se crea un hilo para ese proceso. Después, este hilo puede crear otros hilos dentro del mismo proceso. El hilo pasará a la cola de los ***Listos***.
- **Bloqueo**: cuando un hilo necesita esperar por un suceso, se bloquea (guardando sus registros de usuario, contador de programa y punteros de pila). Ahora el procesador podrá pasar a ejecutar otro hilo que esté al principio de la cola de ***Listos***, mientras el anterior permanece bloqueado.
- **Desbloqueo**: cuando el evento por el que el hilo permanecía en estado de bloqueo finaliza, el hilo pasa a la cola de los ***Listos***.
- **Terminación**: cuando un hilo finaliza, se liberan todos los recursos utilizados por él.

2.5. Compartición de información entre hilos y gestión de recursos compartidos por los hilos

Cuando varios hilos comparten el mismo espacio de memoria es posible que aparezcan algunos problemas, denominados **problemas de sincronización**:

- **Condición de carrera**: se denomina condición de carrera a la ejecución de un programa en la que su salida depende de la secuencia de eventos que se produzcan.
- **Inconsistencia de memoria**: es aquel problema en el que los hilos, que comparten un dato en memoria, ven diferentes valores para el mismo elemento.
- **Inanición**: es uno de los problemas más graves. Consiste en que se deniegue siempre el acceso a un recurso compartido al mismo hilo, de forma que quede bloqueado a la espera del mismo.
- **Interbloqueo**: es el otro de los problemas más graves. Es aquel en el que un hilo está esperando por un recurso compartido que está asociado a un hilo cuyo estado es bloqueado.

Para poder evitarlos en gran medida, es necesario implementar mecanismos de sincronización entre los hilos de un proceso. De esta forma se consigue que un hilo que quiere acceder al mismo recurso que otro, se quede en espera hasta que se liberan dichos recursos. Este tipo de mecanismos se denominan **zona de exclusión mutua.**

Estos hilos deben saber quiénes de ellos deben esperar y cuáles continuar con su ejecución. Para ello existen diferentes **mecanismos de comunicación** entre hilos:

- **Operaciones atómicas**: son aquellas operaciones que se realizan a la vez, es decir, que forman un pack. De esta forma se evita que los datos compartidos tengan distintos valores para el resto de hilos del proceso.
- **Secciones críticas**: se estructura el código de la aplicación de tal forma que se accede de forma ordenada a aquellos datos compartidos.
- **Semáforos**: este mecanismo solo puede tomar valores 0 o 1. El hilo que accede al recurso inicializa el semáforo a 1 y tras su finalización el valor se queda a 0.
- **Tuberías**: todos los hilos se añaden a una cola que se prioriza por medio de un algoritmo *FIFO*, es decir, el primero en solicitar el acceso será asignado al recurso.
- **Monitores**: garantizan que solo un hilo accederá al recurso con el estado de ejecución. Esto se consigue por medio del envío de señales. El proceso que accede recibe el uso del "candado" y cuando finaliza devuelve este al monitor.
- **Paso de mensajes**: todos los hilos deben tener implementados los métodos para entender los mensajes. Esto supone un mayor coste, aunque si existe seguridad en el envío y recepción de un mensaje, se garantiza que solo un proceso accederá en el mismo momento a un recurso.

2.6. Programas multihilo, que permitan la sincronización entre ellos

Cuando en un programa tenemos varios hilos ejecutándose a la vez es posible que varios hilos intenten acceder a la vez al mismo sitio (fichero, conexión, array de datos…) y es posible que la acción de uno de ellos entorpezca la del otro. Para solucionar estos problemas, hay que sincronizar los hilos. Por ejemplo si Hilo A escribe en fichero "Hola" y el Hilo B escribe "Adiós" , al final quedarán las letras entremezcladas. La finalidad es que el Hilo A escriba primero "Hola" y luego el Hilo B escriba "Adiós". Así que cuando un hilo escriba en el fichero, debe marcar de alguna manera que el fichero está ocupado y el otro hilo deberá esperar que esté libre.

La forma de comunicarse consiste en compartir un objeto. Imaginemos que escribimos en un fichero usando una variable *fichero* de tipo ***PrintWriter***. Para escribir el Hilo A hará esto:

```
fichero.println("Hola");
```

Mientras el Hilo B hará esto:

```
fichero.println("Adiós");
```

Si los dos hilos lo hacen a la vez, sin ningún tipo de sincronización, el fichero final puede tener esto:

```
HoAldiaós
```

Para evitar este problema debemos sincronizar los hilos. En Java el código sería así:

```
synchronized (fichero)
{
   fichero.println("Hola");
}
```

Y el otro hilo:

```
synchronized (fichero)
{
   fichero.println("Adiós");
}
```

Al poner ***synchronized(fichero)*** marcamos fichero como ocupado desde que se abren las llaves hasta que se cierran. Cuando el Hilo B intenta también su *synchronized(fichero)*, se queda ahí bloqueado, en espera de que el Hilo A termine con *fichero*.

2.7. Gestión de hilos por parte del sistema operativo. Planificación y acceso a su prioridad

Los sistemas operativos tienen dos **formas de implementar hilos**:

- **Multihilo apropiativo***:* permite al sistema operativo determinar cuándo debe haber un cambio de contexto. Pero esto tiene una desventaja, y es que el sistema puede hacer un cambio de contexto en un momento inoportuno, causando una confusión de prioridades, creando una serie de problemas.
- **Multihilo cooperativo***:* depende del mismo hilo abandonar el control el control cuando llega a un punto de detección, lo cual puede traer problemas cuando el hilo espera la disponibilidad de un recurso.

En cuanto a la gestión de las prioridades, en el lenguaje Java, cada tiene una preferencia. Predeterminadamente, un hilo hereda la prioridad del hilo padre que lo crea, aunque puede aumentar o disminuir mediante el método ***setPriority()***. Para obtener la prioridad del hilo se utiliza el método ***getPriority()***.

La prioridad se mide mediante un rango de valores enteros entre 1 y 10, siendo 1 la mínima prioridad (MIN_PRIORITY) y 10 el valor de máxima (MAX_PRIORITY). Si varios hilos tienen la misma prioridad, la máquina virtual va cediendo control de forma cíclica.

2.8. Depuración y documentación de aplicaciones

Durante todo el proceso de desarrollo de una aplicación es necesario hacer **depuraciones constantes y comprobaciones del comportamiento y resultados** obtenidos.

La mejor forma de realizarlas es a través del uso de **puntos de interrupción**. Estos puntos indican al depurador del software que ejecuta la aplicación qué debe parar o pausar su ejecución en un punto determinado.

Estos puntos suelen usarse para mostrar los valores obtenidos y mensajes necesarios para comprobar que los datos son los correctos y la ejecución hasta ese punto es correcta. Además, este tipo de puntos se utilizan como herramienta de trazabilidad de la aplicación.

Con esto se consiguen **focalizar los errores y poder resolver** las incidencias más rápidamente.

Toda la documentación debe detallar cuál es la funcionalidad de dicha aplicación. En ella deben de aparecer de forma esquemática todos los puntos relevantes, que son necesarios para comprender y conocer el correcto funcionamiento de la misma.

UF3: Sockets y servicios

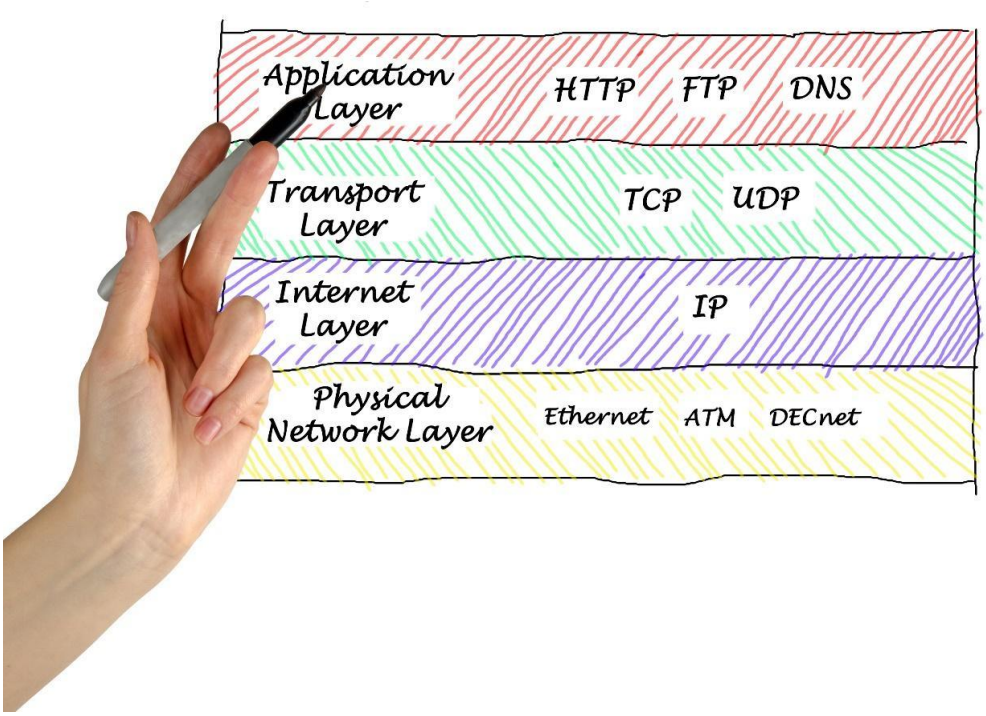
1. Programación de comunicaciones en red

1.1. Comunicación entre aplicaciones

Las **redes de ordenadores** están formadas por un conjunto de dispositivos que se encuentran conectados entre sí para poder intercambiar información entre ellos.

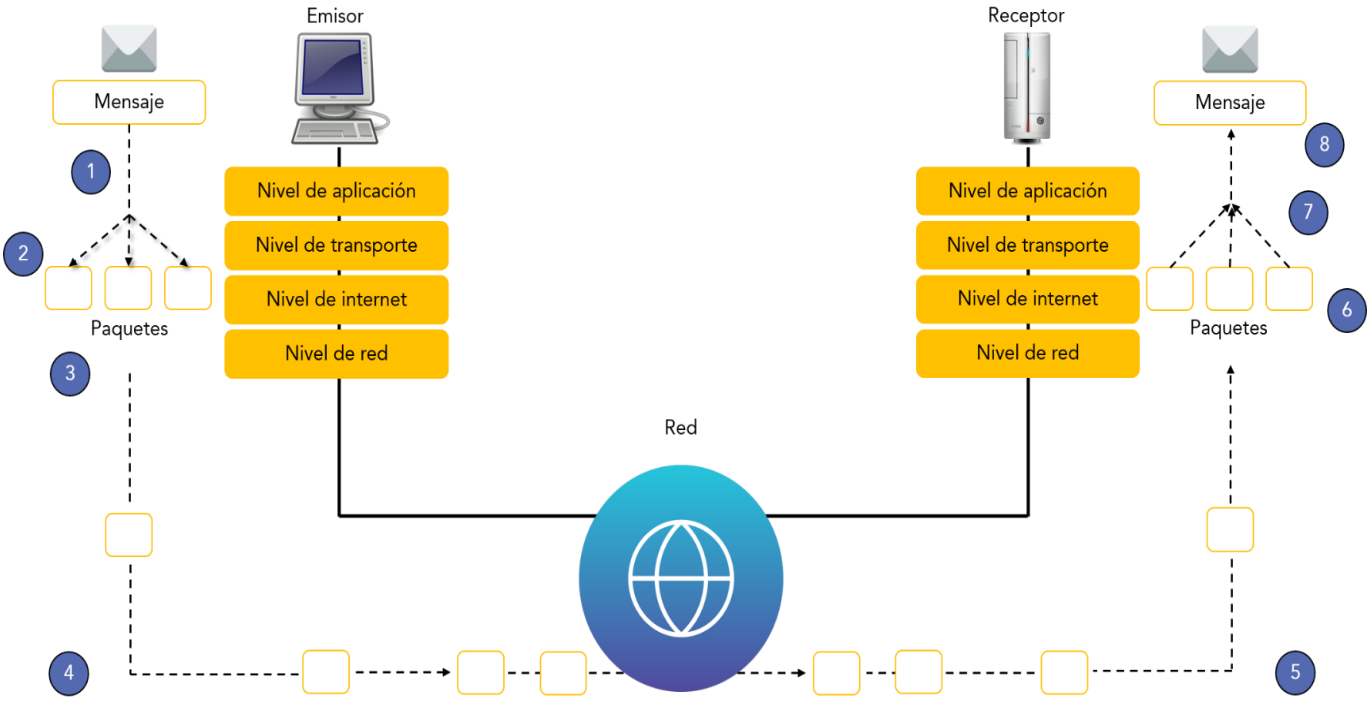
Para implementar estas redes, se utilizan unas tecnologías bastante complejas que dividen en capas para simplificar un poco las diferentes funciones que se deben llevar a cabo. Cada una de estas capas será la encargada de una tarea determinada, poniendo en práctica los diferentes recursos (software y hardware) disponibles.

A continuación, es posible ver que se presentan unas encima de otras, como si fueran una pila, de tal forma que, cada capa se puede comunicar con las que tiene encima y debajo. En el ***modelo TCP/IP***, se divide la comunicación en cuatro capas.



Cuando una **aplicación quiere enviar un mensaje** se siguen los siguientes pasos:

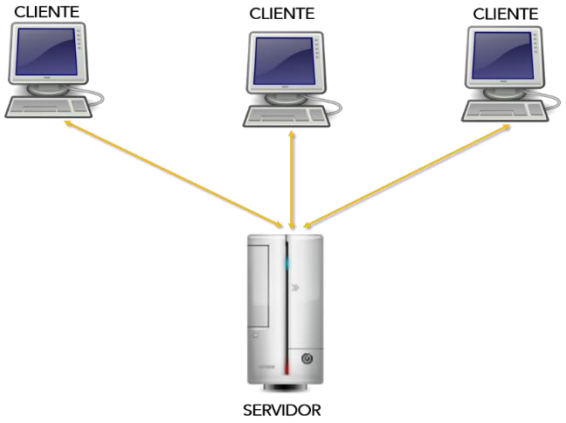
1. El **emisor** envía el mensaje, lo que significa que el mensaje pasa de la capa de aplicación a la capa de transporte.
2. En el **nivel de transporte** se divide el mensaje en paquetes para realizar el envío y lo pasa a la capa inferior, la capa de Internet.
3. En el **nivel de Internet** se comprueba el destino de los paquetes y se calcula la ruta que deben seguir; después se envían los paquetes al nivel de red.
4. El **nivel de red** se encarga de transmitir el paquete al receptor del mensaje.
5. El **receptor** recibe los paquetes en el nivel de red, en el más bajo, y los envía a la siguiente capa, que se corresponde con el nivel de Internet.
6. El **nivel de Internet del receptor** es el encargado de comprobar que son correctos. Si es así, los reenvía al nivel de transporte.
7. El **nivel de transporte** es el encargado de formar el mensaje con los paquetes recibidos y envía el mensaje a la última capa, el nivel de aplicación.
8. El **nivel de aplicación** recibe el mensaje correctamente.



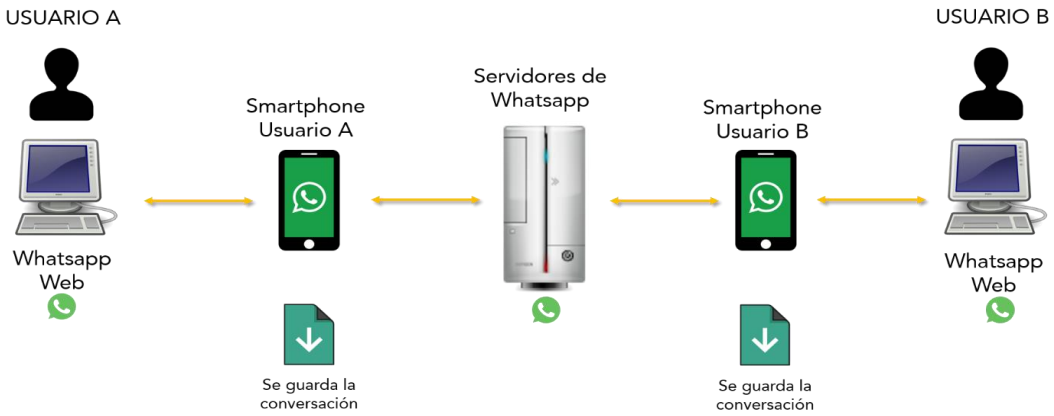
1.2. Roles cliente y servidor

El **modelo cliente-servidor** se basa en una arquitectura en la que existen distintos recursos, a los que se denomina servidores, y un determinado número de clientes, es decir, de sistemas que requieren de esos recursos. Los servidores, además de proveer de recursos a los clientes, también ofrecen una serie de servicios que se estudiarán en el siguiente capítulo.

En este modelo, los clientes son los encargados de **realizar peticiones a los servidores**, y estos responden con la información necesaria.



En la actualidad existen distintas aplicaciones que utilizan este modelo, como pueden ser el **correo electrónico, mensajería instantánea y el servicio de Internet**, es decir, se utiliza siempre que se accede a una página web.



Se debe tener en cuenta que, en este modelo, no se intercambian los roles entre clientes y servidores.

Los ***sockets*** también son un ejemplo del modelo cliente-servidor y se estudiarán en este capítulo.

1.3. Elementos de programación de aplicaciones en red. Librerías

- Clase *InetAddress*

Es la clase que representa las **direcciones IP**.

byte[] getAddress ()	Devuelve la dirección IP sin procesar como objeto
static InetAddress getLocalHost ()	Devuelve la dirección IP de la máquina
static InetAddress getLocalHost (String host)	Devuelve la dirección IP de la máquina especificada
String toString ()	Convierte la dirección IP en una cadena

Para más información:
https://docs.oracle.com/javase/7/docs/api/java/net/InetAddress.html

- Clase *URL*

Es la clase que representa un localizador de recursos uniforme (URL), es decir, representa la dirección del recurso en Internet.

Una URL es un conjunto de caracteres que permiten denominar de forma única los recursos en Internet, de esta forma facilita el acceso a ellos.

El **formato de una URL** es el siguiente:
protocolo://máquina:puerto/ruta_fichero

Si es necesario la identificación para acceder a ese recurso, el usuario y contraseña se deben indicar delante de la máquina, de forma que la URL quedaría:

protocolo:/usuario:contraseña@máquina:puerto/ruta_fichero

Existen distintas formas de crear un objeto de esta clase, dependiendo de los parámetros de los que se disponga. Por ello, existen diferentes constructores:

URL (String url)	Crea un objeto URL de la cadena recibida.
URL (String protocol, String host, int port, String file)	Crea un objeto URL con los datos recibidos.
URL (String protocol, String host, String file)	Crea un objeto URL con los datos recibidos.

Para más información:
https://docs.oracle.com/javase/7/docs/api/java/net/URL.html

Los **métodos** son:

String getFile ()	Devuelve el nombre del archivo
String getHost ()	Devuelve el nombre de la máquina
String getPath ()	Devuelve el nombre de la ruta de acceso
int getPort ()	Devuelve el puerto que ocupa
String getProtocol ()	Devuelve el protocolo
URI toURI ()	Devuelve el URI equivalente a esta URL
URLConnection openConnection ()	Crea y devuelve una conexión al objeto remoto de esta URL

- Clase *URLConnection*

Esta clase permite trabajar con conexiones realizadas a la URL especificada. Para ello, es necesario crear un objeto de la clase **URL** e invocar al método *openConnection()*.

```
URL url = new URL ("https://www.ilerna.es/ca/fp-a-distancia");
URLConnection urlCon = url.openConnection();
```

Con esto obtenemos una conexión al objeto URL referenciado. Las instancias se esta clase se pueden utilizar tanto para leer como para escribir al recurso referenciado por la URL.

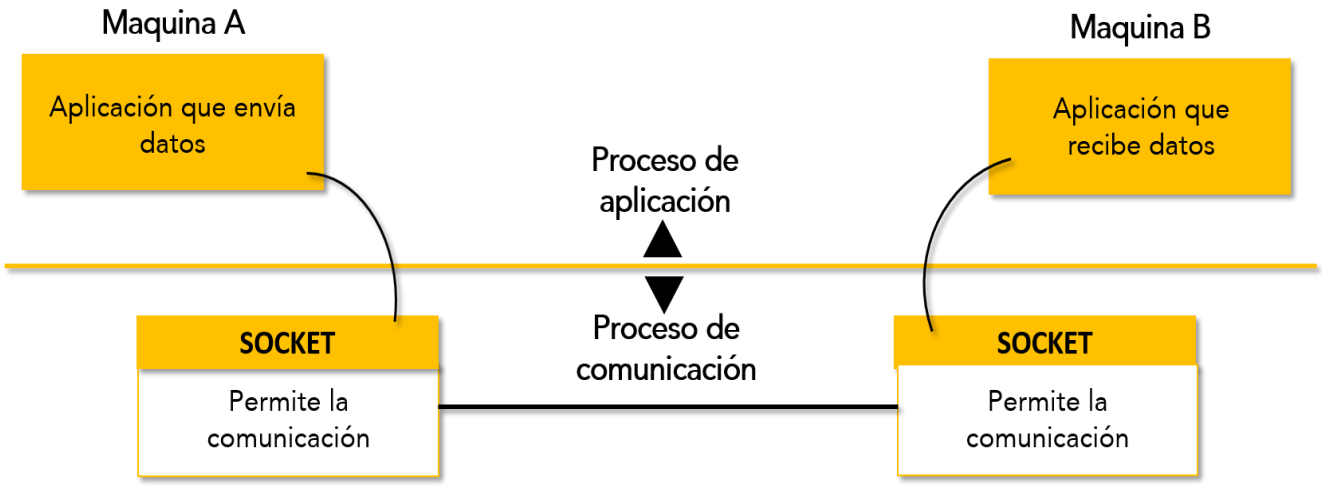
Algunos **métodos** de esta clase son:

abstract void connect ()	Establece la conexión con el recurso de la URL. Una vez que se obtiene la conexión se pueden leer y escribir datos en dicha conexión
InputStream getInputStream ()	Devuelve un flujo de entrada para leer los datos del recurso
OutputStream getOutourStream ()	Devuelve un flujo de salida para escribir datos en el recurso
URL getURL()	Devuelve la URL de la conexión

Para más información:
https://docs.oracle.com/javase/7/docs/api/java/net/URLConnection.html

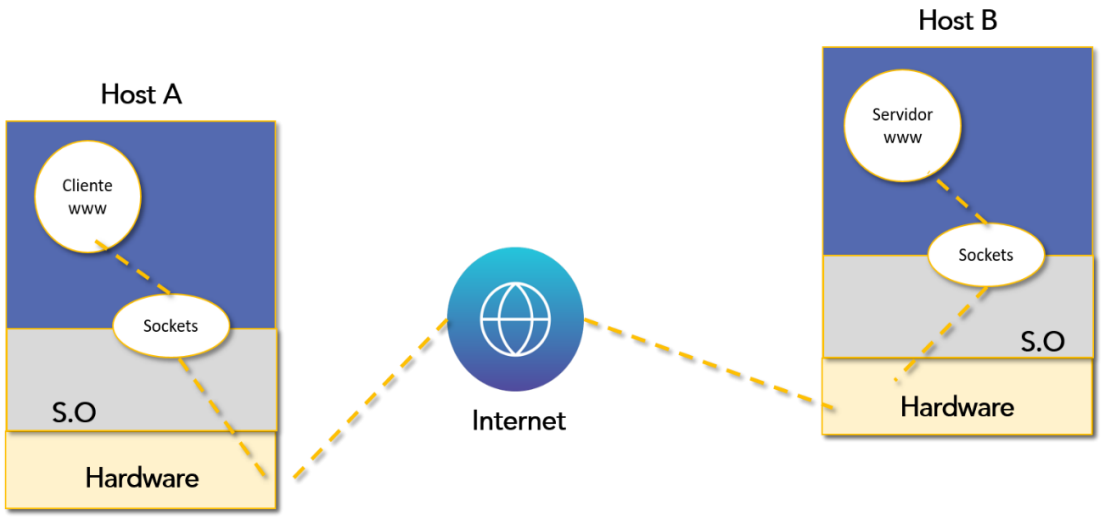
1.4. Sockets

Un socket es un mecanismo que **permite la comunicación entre aplicaciones a través de la red,** es decir, abstrae al usuario del paso de la información entre las distintas capas. Su función principal es crear un canal de comunicación entre las aplicaciones y simplificar el intercambio de mensajes.



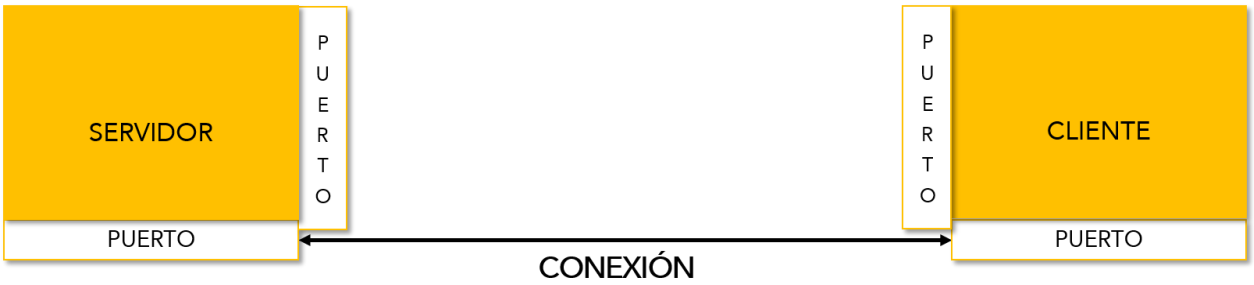
Un ***socket*** se define mediante la dirección IP de los dos dispositivos, el protocolo de transporte, y, por último, el puerto de cada uno de los dispositivos por el que se conectan. **Un puerto es el identificador que permite la comunicación entre los dispositivos de una red**. Una máquina solo puede tener un único puerto asignado.

La máquina denominada servidor tiene un puerto asignado, que es el encargado de quedarse a la espera de que algún cliente realice cualquier petición.



Para que un cliente pueda comunicarse con un servidor, debe conocer su dirección IP y el puerto asignado, y realizar la petición. Cuando el servidor la recibe, si decide aceptarla, asigna el puerto para la comunicación, de forma que el socket conocido por el cliente queda libre para recibir a nuevos clientes.

Como la petición la realiza desde el puerto del cliente, el servidor ya conoce su puerto, y es el cliente el que aceptará la conexión, y después, asignará el puerto para el envío de mensajes.



Existen **dos tipos de sockets**: los orientados a conexión y los no orientados a conexión.

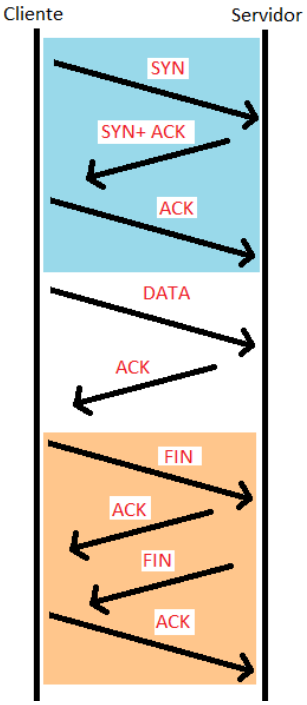
- ***Sockets* orientados a conexión**

Este tipo de *sockets* utilizan el protocolo **TCP**. Se utilizan para aquellas aplicaciones que requieren una alta fiabilidad en el envío de mensajes, puesto que, mediante este protocolo, aseguran la entrega de todos y cada uno de los paquetes, en el mismo orden en el que fueron enviados. Para ello, se utiliza una verificación de los paquetes en el receptor, enviando un acuse de recibo. Si el emisor del mensaje no lo recibe, se procederá entonces al reenvío del paquete.

Otra de las principales características de este tipo de *sockets* es el **establecimiento de la conexión**. Para realizarla es necesario el envío de varios mensajes antes de comenzar el intercambio de mensajes:

- Petición del cliente de establecer conexión (**SYN**).
- Confirmación (**ACK**) por parte del servidor para establecer la conexión (**SYN**).
- Confirmación de cliente del mensaje anterior (**ACK**).

Una vez que estos tres mensajes han sido enviados, es posible empezar a enviar mensajes entre los dispositivos.



Cuando termina el envío de mensajes, el cliente es el que decide cerrar la conexión, para ello existen otro tipo de mensajes:

- El cliente envía el mensaje para cerrar la conexión (**FIN**).
- El servidor confirma el cierre de conexión (**ACK**).
- El servidor envía el cierre de la conexión (la otra vía) (**FIN**).
- El cliente confirma el cierre de conexión (**ACK**).

Los **sockets orientados a conexión** se utilizan en distintos servicios como, por ejemplo, *FTP, Telnet, HTTP y SMTP* que se estudiarán en el próximo capítulo.

- ***Sockets* no orientados a conexión**

Este tipo de sockets utilizan el protocolo **UDP**. Al contrario que los anteriores, este tipo de sockets no garantizan que los mensajes enviados lleguen a su destino, por lo que no es un protocolo fiable. Además, tampoco garantizan que los paquetes vayan a llegar en el orden enviado. A cambio de esto, ofrecen una mayor velocidad en el intercambio de los mensajes, puesto que no tienen que establecer conexión para ello, ni controlar los mensajes que han llegado.

Los *sockets* no orientados a conexión también son utilizados en los servicios. En este caso son utilizados por ***SNTP, DNS y NFS.***

1.5. Utilización de sockets para la transmisión y recepción de información

En la capa de transporte hay que destacar dos protocolos, **TCP y UDP**, que son muy importantes en la comunicación en red.

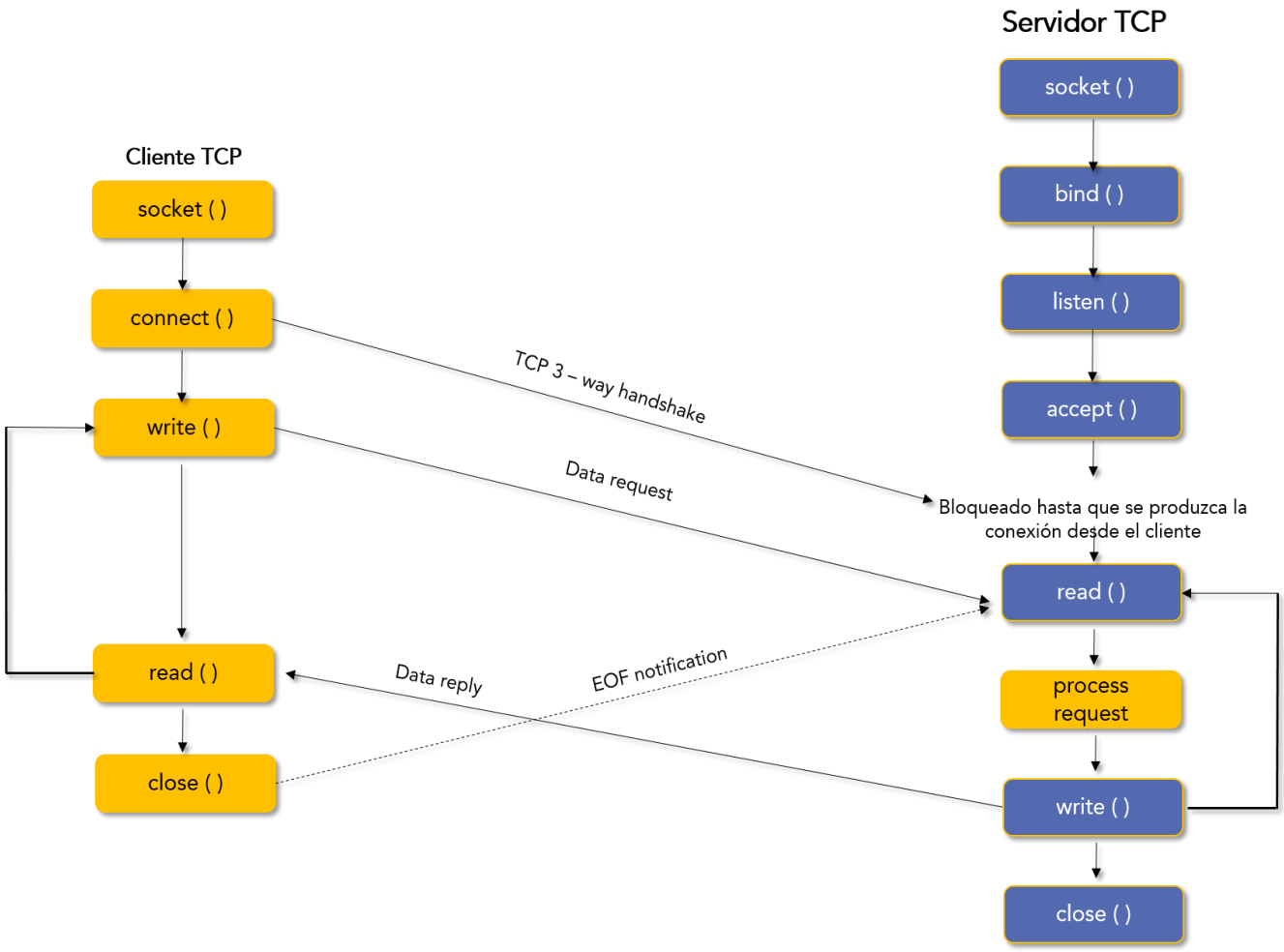
Aunque ambos sean protocolos de la capa de transporte, tienen una gran cantidad de **diferencias**:

	TCP	UDP
Conexión	Protocolo orientado a conexiones.	Protocolo sin conexiones.
Función	Se usa para enviar mensajes por Internet de una computadora a otra, mediante conexiones virtuales	Se usa para transporte de mensajes y/o transferencias. Al no estar basada en conexiones, un programa puede enviar una carga de paquetes y recibirse en el destino.
Uso	Aplicaciones que requieren confiablidad alta y donde el tiempo de transmisión es menos crítico.	Aplicaciones que necesitan transmisión rápida y efectiva. Servidores que reciben una gran cantidad de peticiones pequeñas de un alto número de clientes.
Uso por otros protocolos	HTTP, HTTPS, SMTP, Telnet	DNS, DHCP, TFTP, SNMP, RIP, VoIP
Ordenar por paquetes de data	Orden especificado.	No tienen orden inherente. Los paquetes son independientes unos de los otros. Si requieren un orden, esto se maneja a nivel de aplicación.
Velocidad de transferencia	Más lento .	No hace falta verificación de errores por paquete, por lo que su velocidad es mayor.
Confiabilidad	Ofrece una garantía absoluta de que los datos llegarán en el mismo orden en que se enviaron.	No hay garantía de que los paquetes de datos lleguen.
Tamaño del Título	20 bits	8 bits
Campos comunes de títulos	Puerto de origen, puerto de destino, *checksum.*	

Fluidez	Los datos se leen como una secuencia de bits y no se transmiten indicadores para los límites de segmentos de los mensajes.	Los paquetes son enviados individualmente; se verifica su integridad solo si llegan. Los paquetes tienen límites definidos, que comprueban que el mensaje está completo.
Peso	TCP es pesado. Requiere tres paquetes para establecer una conexión antes de transmitir. TCP maneja confiabilidad y control de congestión.	UDP es liviano. No hay ordenamiento de mensajes ni conexiones de verificación.
Control de flujo de data	Sí realiza control de flujo. Requiere tres paquetes para establecer una conexión antes de transmitir. Maneja confiabilidad y control de congestión.	No se realiza control de flujo.
Verificación de errores	Sí hay verificación de errores.	Sí hay verificación de errores, pero no tiene opciones para recuperar los datos perdidos.
Campos	1. Número de secuencia 2. Número de ACK 3. Índice data 4. Reservado 5. Bit de control 6. Ventana 7. Indicador de urgencia 8. Opciones 9. Relleno *10. CheckSum* 11. Puerto de origen 12. Puerto de destino	1. Largo 2. Puerto de origen 3. Puerto de destino *4. CheckSum*
Reconocimiento	Hay segmentos de reconocimiento.	No hace reconocimiento.
"Handshake" (verifica conexiones en tres tiempos)	SYN SYN-ACK ACK	No hace esta verificación.
Checksum	Completo	Solo para detectar errores.

1.6. Creación de sockets

- *Sockets* orientados a conexión



- Clase *ServerSocket*

Es la clase que se debe instanciar en la parte del servidor para crear el puerto que se queda esperando a la conexión por parte de los clientes.

Existen **distintos constructores**:

ServerSocket ()	Crea un socket no enlazado.
ServerSocket (int port)	Crea un socket enlazado al puerto especificado.
ServerSocket (int port, int baklog)	Crea un socket enlazado al puerto especificado, indicando el número máximo de peticiones que pueden estar en cola.

ServerSocket (int port, int baklog, InetAdress dirección)	Crea un socket enlazado al puerto especificado y a una dirección IP, indicando el número máximo de peticiones que pueden estar en cola.

Para más información:
https://docs.oracle.com/javase/7/docs/api/java/net/ServerSocket.html

- Clase *Socket*

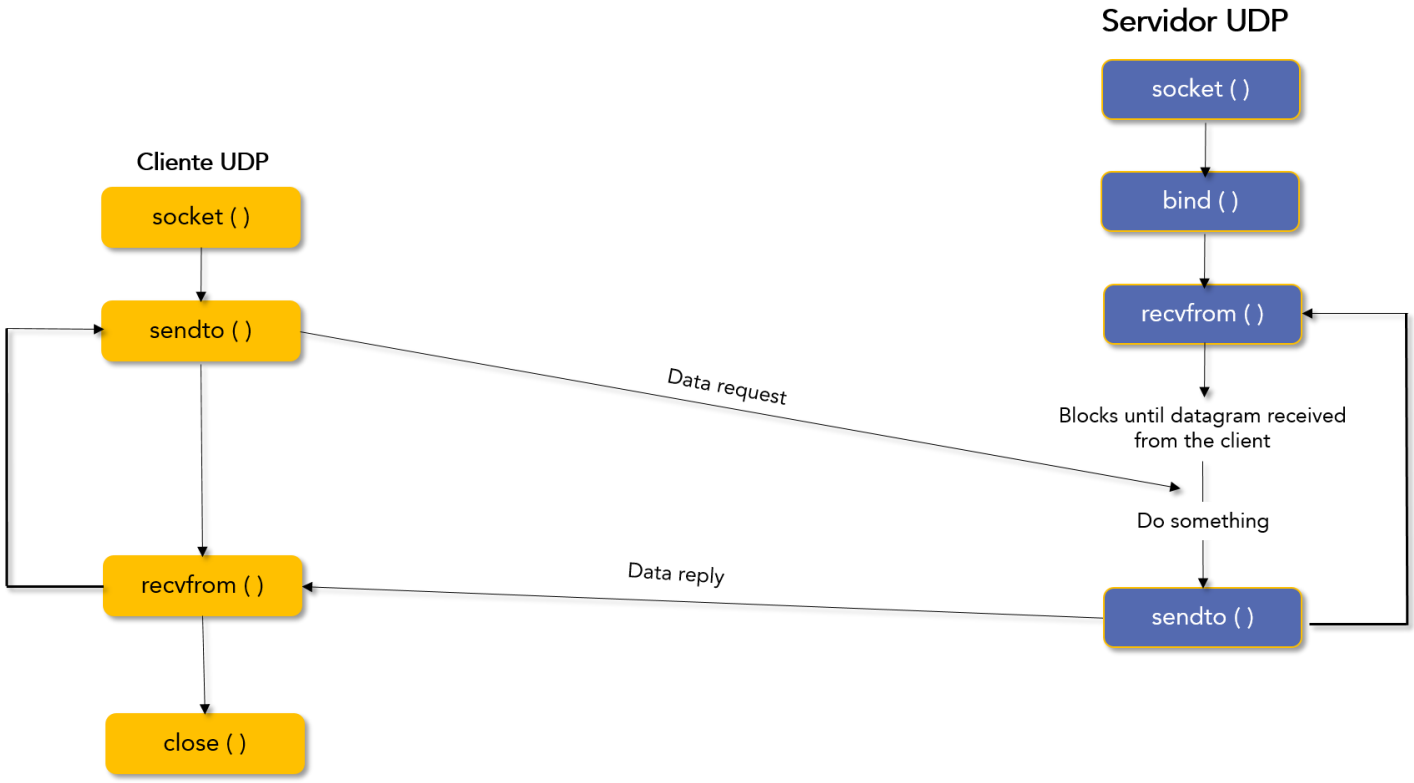
Es la clase que se debe instanciar en la parte del cliente.

Existen **distintos constructores**:

Socket ()	Crea un socket no conectado.
Socket (InetAddress address, int port)	Crea un socket conectado al puerto indicado en la dirección IP especificada.
Socket (InetAddress address, int port, InetAddress localAddr, int localPort)	Crea un socket y lo conecta a la dirección remota especificada en el puerto remoto especificado.
Socket (String host, int port)	Crea un socket y lo conecta al puerto especificado en el host.
Socket (String host, int port, InetAddress localAddr, int localPort)	Crea un socket y lo conecta al host remoto especificado en el puerto especificado.

Para más información:
https://docs.oracle.com/javase/7/docs/api/java/net/Socket.html

- **Sockets no orientados a conexión**



- Clase *DatagramSocket*

Es la clase que se debe instanciar tanto en la parte del cliente como en el servidor.

Existen **distintos constructores**:

DatagramSocket ()	Crea un socket UDP y lo conecta a cualquier puerto disponible de la máquina host remota.
DatagramSocket (int port)	Crea un socket UDP y lo conecta con el puerto especificado.
DatagramSocket (int port, InnetAddress laddr)	Crea un socket UDP y lo conecta a la dirección local especificada.
DatagramSocket (SocketAddress bindaddr)	Crea un socket UDP y lo conecta a la dirección de socket local especificada.

Para más información:
https://docs.oracle.com/javase/7/docs/api/java/net/DatagramSocket.html

- Clase *DatagramPacket*

Es la clase que permite crear paquetes para enviarlos a través del socket UDP.

Existen **distintos constructores**:

DatagramPacket (byte [] buf, int longitud)	Crea un datagrama que recibe paquetes de la longitud especificada.
DatagramPacket (byte [] buf, int longitud, InetAddress address, int port)	Crea un datagrama que envía paquetes de la longitud especificada al puerto especificado.
DatagramPacket (byte [] buf, int offset, int longitud)	Crea un datagrama que recibe paquetes de la longitud especificada del puerto especificado.
DatagramPacket (byte [] buf, int longitud, SocketAddress address)	Crea un datagrama que envía paquetes de la longitud especificada al host especificado.

Para más información:
https://docs.oracle.com/javase/7/docs/api/java/net/DatagramPacket.html

1.7. Enlazamiento y establecimiento de conexiones

- *Sockets* orientados a conexión
- Clase *ServerSocket*

Algunos de sus **métodos más importantes**:

Socket accept ()	Acepta una petición de conexión por parte del cliente.
void bind (SocketAddress endpoint)	Vincula el *socket* a una dirección IP y a un puerto determinado.
void close ()	Cierra el *socket*.
InetAddress getInetAddress ()	Devuelve la dirección local del *socket.*
int getPort ()	Devuelve el puerto del *socket.*

Para más información:
https://docs.oracle.com/javase/7/docs/api/java/net/ServerSocket.html

- Clase *Socket*

Algunos de sus **métodos más importantes**:

void bind (SocketAddress endpoint)	Vincula el socket a una dirección local.
void close ()	Cierra el *socket.*
void connect (SocketAddress endpoint)	Conecta el *socket* con el servidor.
InputStream getInputStream ()	Devuelve el flujo de entrada para el *socket*.
InetAddress getLocalAddress ()	Devuelve la dirección a la que está conectado el *socket*.
int getLocalPort ()	Devuelve el número de puerto local al está vinculado el *socket.*
OutputStream getOutputStream ()	Devuelve el flujo de salida para el *socket.*
int getPort ()	Devuelve el número de puerto remoto al que está conectado el *socket.*
boolean isClosed ()	Comprueba si está cerrado el *socket*.
boolean isConnected ()	Comprueba si está conectado el *socket*.

Para más información:
https://docs.oracle.com/javase/7/docs/api/java/net/Socket.html

- **Servicios no orientados a conexión**
- Clase *DatagramSocket*

Algunos de sus **métodos más importantes**:

void bind (SocketAddress addr)	Vincula el *socket* a una dirección y puerto especificado.
void close ()	Cierra el *socket.*
void connect (InnetAddress address, int port)	Conecta el *socket* a una dirección remota.
void disconnect ()	Desconecta el *socket.*
InetAddress getInetAddress ()	Devuelve la dirección local a la que está vinculado el *socket.*

int getPort ()	Devuelve el número de puerto al que está conectado este *socket.*
void receive (DatagramPacket p)	Recibe un paquete de datagramas del *socket.*
void send (DatagramPacket p)	Envía un paquete de datagramas al *socket.*
setSoTimeout (int time)	Habilita o deshabilita un tiempo de espera especificado, en milisegundos.

Para más información:
https://docs.oracle.com/javase/7/docs/api/java/net/DatagramSocket.html

- Clase *DatagramPacket*

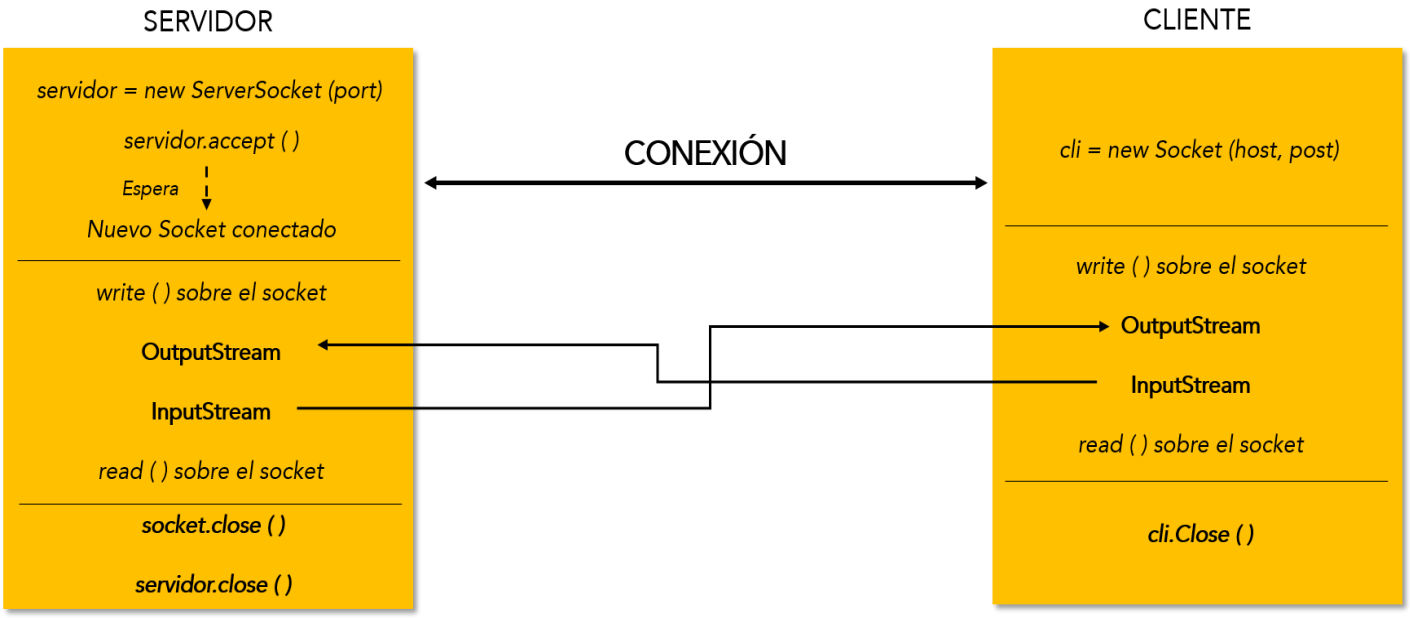
Algunos de sus **métodos más importantes**:

InetAddress getAddress ()	Devuelve la dirección IP de la máquina con la que se ha realizado una transmisión de paquetes.
byte [] getData ()	Devuelve el búfer de datos.
int getLength ()	Devuelve la longitud de los datos.
int getPort ()	Devuelve el puerto del host remoto con el que se ha realizado una transmisión de paquetes.
void setAddress (InetAddres address)	Establece la dirección IP de la máquina a la que se envía este datagrama.
void setData (byte [] data)	Establece el búfer de datos para un paquete.
void setLength (int longitud)	Establece la longitud del paquete.
void setPort (int port)	Establece el puerto del host remoto al que se envía el datagrama.

Para más información:
https://docs.oracle.com/javase/7/docs/api/java/net/DatagramPacket.html

1.8. Programación de aplicaciones cliente y servidor

- *Socket* TCP

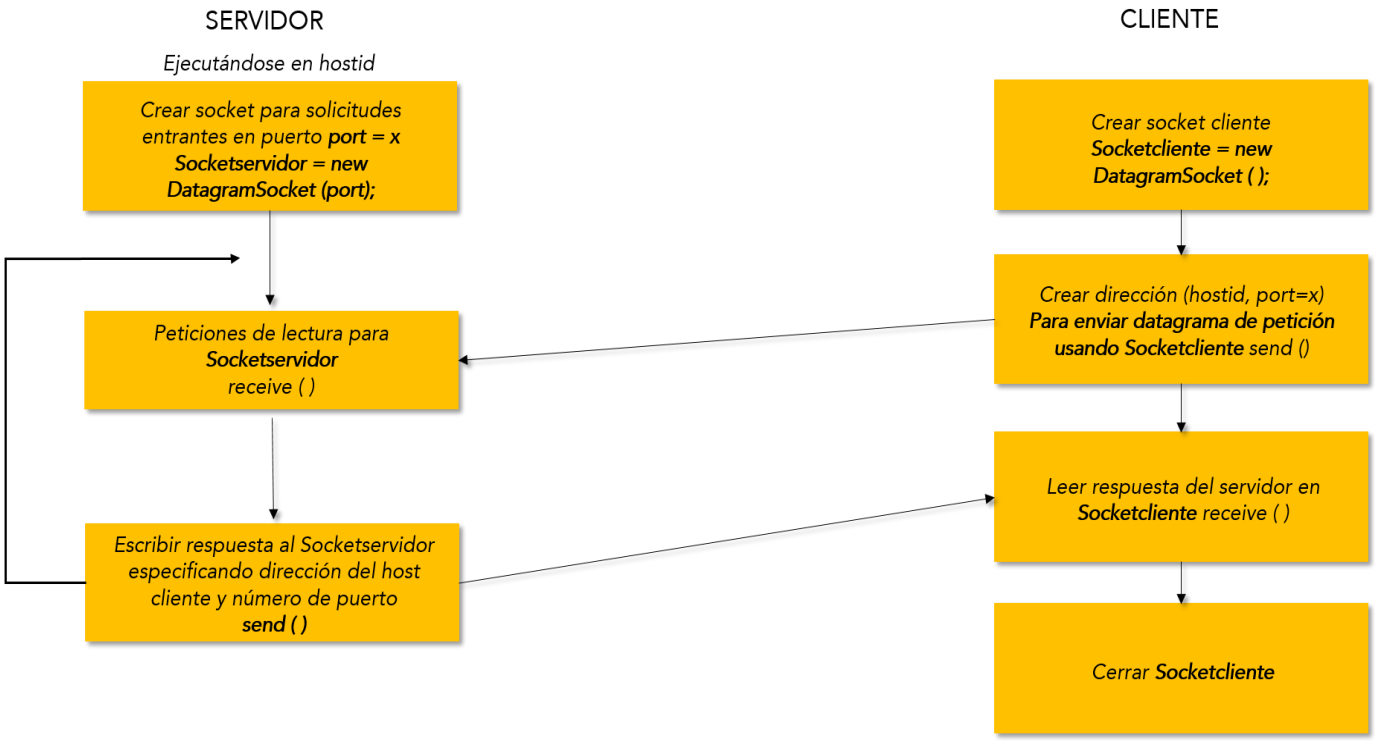


Cuando se crea un *socket* en el servidor, se queda esperando a que los clientes se conecten. Esto se realiza mediante la función ***accept( )***. Cuando se crea un socket en el cliente, indicando el puerto y la dirección del socket servidor, se realiza la conexión entre el cliente y el servidor.

Una vez que existe conexión, comienza la transmisión de los datos, mediante funciones de *write* y *read*. Estas operaciones se realizan mediante las clases *DataInputStream* y *DataOutputStream*, que permiten utilizar diversos métodos de lectura y de escritura.

Cuando se termina la transmisión de los datos, se cierra la conexión, y también el *socket* del cliente. Después, cuando el servidor acaba su función, también se cierra.

- Socket UDP



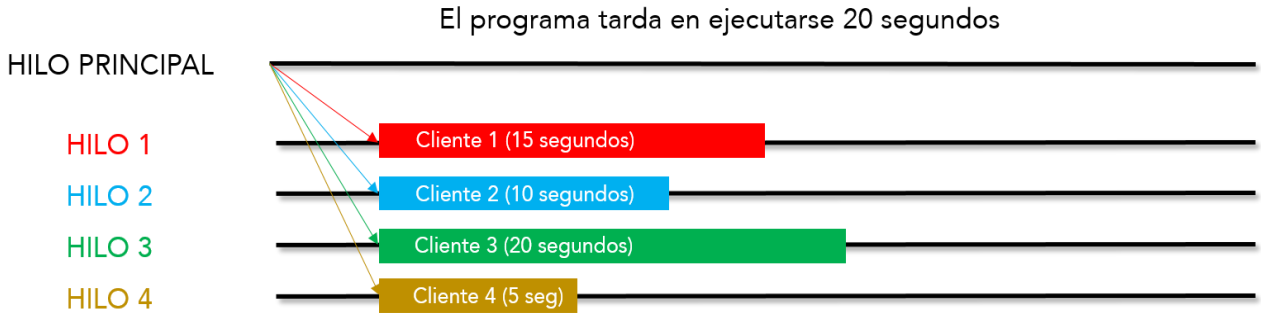
Se crea un *socket* en el servidor que se queda a la espera de peticiones de clientes. Cuando se crea un *socket* en el cliente, se conecta al *socket* del servidor. Una vez realizada la conexión se envían los distintos datagramas mediante los métodos ***send*** y ***receive***.

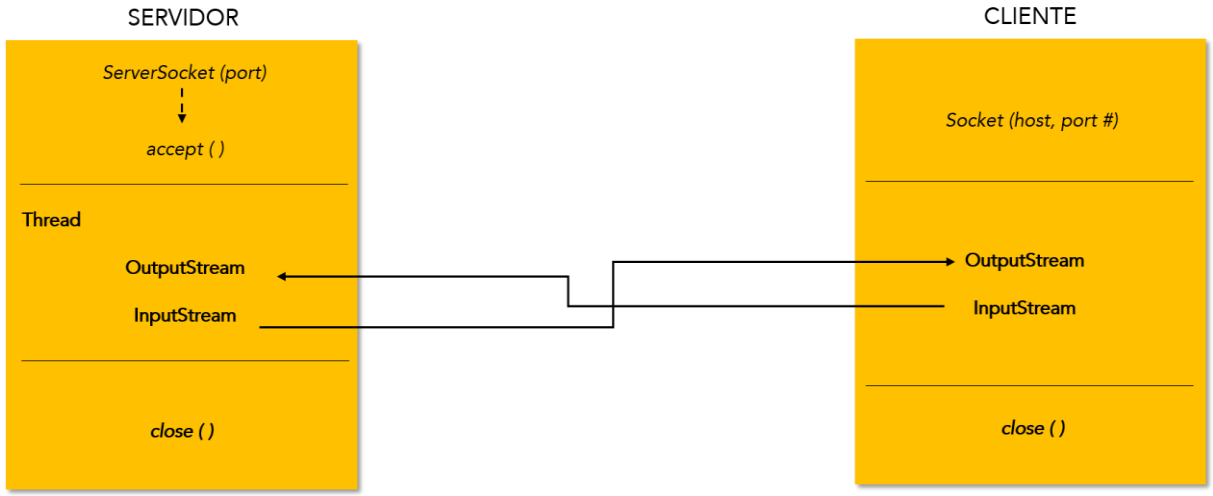
Cuando se termina la transmisión de los datos, se cierra el *socket* del cliente. El servidor se puede quedar a la espera de otros clientes, pero cuando el servidor termina su función, también hay que cerrar su socket.

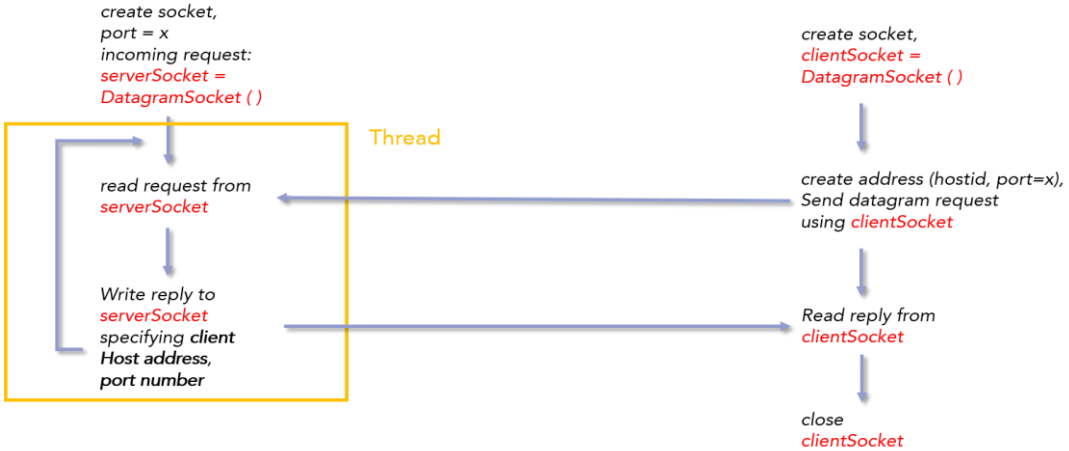
1.9. Utilización de hilos en la programación de aplicaciones en red

Mediante los *sockets* creados hasta ahora, el servidor únicamente es capaz de trabajar con un cliente simultáneamente. La solución para esto pasa por la creación de hilos para contestar a cada cliente. Es decir, cuando se crea el socket del servidor y se queda esperando clientes y recibe la petición de un cliente, debe crear un Hilo con el resto del funcionamiento del servidor.

De esta forma, el servidor podrá atender a todos los clientes que realicen la petición.







2. Generación de servicios de red

2.1. Programación y verificación de aplicaciones cliente de protocolos estándar

Las aplicaciones cliente son aquellas que realizan peticiones a un servidor, solicitando cualquier tipo de recurso o información que esté alojado en el servidor.

El ejemplo más claro de aplicaciones cliente son los **navegadores**, a través de los cuales se realizan la mayor parte de las peticiones. Las aplicaciones instaladas en cualquier tipo de dispositivo que requieran de Internet para su funcionamiento son, de igual manera, un tipo de cliente.

Mediante **Java** se pueden crear aplicaciones que se conecten a distintos servicios, como pueden ser clientes *FTP*, *Telnet* o *SMTP*, entre otras.

El **modelo TCP/IP** está compuesto por cuatro capas, donde la capa de aplicación maneja protocolos de alto nivel que implementa **servicios** como:

- **Conexión remota** → Telmet
- **Correo electrónico** → SMTP
- **Acceso a ficheros remotos** → FTP, NFS, TFTP
- **Resolución de nombres de ordenadores** → DNS, WINS
- **World Wide Web** → HTTP

Capas TCP/IP
Aplicación
Transporte
Internet
Interfaz Red

Capas y protocolos TCP/IP	
SMTP, Telnet, FTP, HTTP	NFS, SNMP, DNS
TCP	UDP
IP	
Protocolos de subred	

A lo largo del capítulo se estudiarán las distintas clases que ofrece el *API* de *Java* para realizar este tipo de conexiones, y así poder lograr una comunicación con los distintos servicios.

2.2. Programación de servidores

Para dar servicio a las peticiones que realizan los clientes desde cualquier tipo de aplicación es necesario almacenar en una máquina todos los datos que se van a solicitar. Para esto se utilizan los **servidores**, que, como se ha comentado anteriormente, son **equipos informáticos que dan servicio a otros equipos.**

Debido al gran volumen de peticiones que pueden llegar a tener este tipo de equipos, los servidores se caracterizan por tener unas prestaciones técnicas muy superiores al resto de ordenadores que se comercializan.

Existen **dos tipos de servidores**:

- **Servidor compartido**: servidores con prestaciones muy altas, que son compartidos entre diferentes empresas por su elevado coste.
- **Servidor dedicado**: servidores únicos para cada empresa.

Los servidores son los encargados de proporcionar ciertos servicios a los clientes, es decir, de ofrecer una serie de **programas que permiten gestionar los recursos**:

- **DNS**: cuando un cliente solicita una dirección web desde un navegador, el servidor se encarga de determinar en qué lugar se encuentra esta web y le indica la respuesta al cliente.
- **DHCP**: cuando un cliente se conecta a un servidor es necesario conocer quién es el cliente a través de su IP. El protocolo DHCP configurado en el servidor permite asignar de manera dinámica una IP a cada cliente que se conecta. Al contrario que los clientes, los servidores siempre tienen una IP fija.
- **FTP**: protocolo de transferencia de archivos. Es un protocolo utilizado para poder enviar un fichero de un cliente a un servidor. El servidor debe ser configurado para poder permitir este tipo de transferencias. La mayor parte de los servidores permiten esto. FTP no lleva ningún tipo de encriptación, por lo que si se desea un servicio de transferencia encriptada es necesario utilizar SFTP.
- **TELNET**: protocolo que permite acceder a otro equipo remoto a través de la terminal. Actualmente lo más habitual es utilizar una de sus variantes que es SSH, que permite hacer uso de una conexión remota cifrada.

- **HTTP**: es uno de los protocolos más conocidos y más usados en las aplicaciones cliente. Permite hacer uso del intercambio de información en Internet (World Wide Web). Emplea un esquema de petición-respuesta, petición del cliente y respuesta del servidor web.
- **NFS**: este protocolo permite que distintos equipos que forman parte de una misma red puedan acceder a ficheros como si estuvieran almacenados de forma local en el equipo.
- **SMTP**: protocolo simple de transferencia de correo electrónico. Permite a los clientes el envío de correo entre distintos dispositivos, mientras que mediante los protocolos POP e IMAP se reciben.

2.3. Análisis de librerías de clases y componentes

- **Telnet**

- **Constructores:**

TelnetClient ()	Constructor por defecto. Su terminal *type* es VT100.
TelnetClient (String terminalType)	Constructor con el terminal *type* recibido por parámetro.

- **Métodos:**

void disconnect ()	Desconecta la sesión Telnet.
InputStream getInputStream ()	Devuelve el *stream* de entrada de la conexión Telnet.
OutputStream getOutputStream ()	Devuelve el *stream* de salida de la conexión Telnet.

Para más información:
https://commons.apache.org/proper/commons-net/javadocs/api-3.6/org/apache/commons/net/telnet/TelnetClient.html

- FTP

- Constructor:

FTPClient ()	Crea un cliente FTP

- Métodos:

void disconnect ()	Cierra la conexión al servidor FTP.
void abort ()	Interrumpir una transferencia que se encuentra en proceso.
boolean deleteFile ()	Elimina un fichero del servidor FTP.
boolean changeToParentDirectory ()	Cambia al directorio padre de donde se encuentra.
boolean changeWorkingDirectory (String pathname)	Cambia el directorio de trabajo por el que se indica en la ruta.
boolean deleteFile (String pathname)	Elimina el archivo indicado.
FTPFile[] listDirectories ()	Devuelve la lista de directorios que se encuentran en el directorio de trabajo.
FTPFile[] listDirectories (String pathname)	Devuelve la lista de directorios que se encuentra en la ruta indicada.
FTPFile[] listFiles ()	Devuelve la lista de archivos que se encuentran en el directorio de trabajo.
FTPFile[] listFiles (String pathname)	Devuelve la lista de archivos que se encuentran en la ruta indicada.
String[] listNames ()	Devuelve la lista de nombres de los archivos que se encuentran en el directorio de trabajo.
String[]listNames (String pathname)	Devuelve la lista de nombres de los archivos que se encuentran en la ruta indicada.
boolean login (String username, String password)	Entra en el servidor FTP mediante el usuario y contraseña.
boolean logout ()	Cierra sesión en el servidor FTP.
boolean makeDirectory (String pathname)	Crea un nuevo directorio en la ruta especificada.
String printWorkingDirectory ()	Devuelve el nombre del directorio de trabajo.
boolean removeDirectory (String pathname)	Elimina el directorio especificado.

boolean rename (String from, String to)	Cambia el nombre de un fichero del servidor.
boolean retrieveFile (String remote, OutputStream local)	Recupera el contenido del fichero del servidor en un flujo de datos.
boolean storeFile (String remote, InputStream local)	Almacena el contenido del flujo en un fichero del servidor.

Para más información:
https://commons.apache.org/proper/commons-net/apidocs/org/apache/commons/net/ftp/FTPClient.html

Algunos de estos métodos devuelven un *array* de objetos de la clase **FTPFile**. Esta clase se utiliza para mostrar la información de los ficheros almacenados en un servidor FTP.

String getName ()	Devuelve el nombre de un fichero.
long getSize ()	Devuelve el tamaño del fichero en bytes.
int getType ()	Devuelve el tipo de fichero. Puede ser directorio, fichero o enlace simbólico.
String getUser ()	Devuelve el usuario propietario del fichero.
boolean isDirectory ()	Comprueba si es un directorio.
boolean isFile ()	Comprueba si es un fichero.
void setName (String name)	Establece el nombre de un archivo.
void setUser (String user)	Establece el usuario propietario de un archivo.
String toString ()	Devuelve en forma de cadena el contenido del fichero.

Para más información:
https://commons.apache.org/proper/commons-net/apidocs/org/apache/commons/net/ftp/FTPFile.html

- HTTP

- Constructor:

HttpURLConnection(URL u)	Crea la conexión.

- Métodos:

abstract void disconnect ()	Indica que serán improbables próximas peticiones al servidor.
String getRequestMethod ()	Devuelve el método de solicitud.
int getResponseCode ()	Devuelve el código del estado de un mensaje HTTP.
String getResponseMessage ()	Devuelve el mensaje de respuesta HTTP.

Para más información:
https://docs.oracle.com/javase/7/docs/api/java/net/HttpURLConnection.html

- SMTP

- Constructor:

SMTPClient ()	Constructor por defecto.
SMTPClient (String codificación)	Se establece una codificación en el constructor.

- Métodos:

boolean login ()	Inicia sesión en el servidor SMTP.
boolean login (String hostname)	Inicia sesión en el servidor SMTP del host indicado.

boolean logout ()	Cierra la sesión con el servidor.
Writer sendMessageData ()	Envía el mensaje de correo.
boolean verify (String username)	Comprueba que la dirección de correo es válida.

Para más información:
https://commons.apache.org/proper/commons-net/javadocs/api-3.6/org/apache/commons/net/smtp/SMTPClient.html

2.4. Análisis de requisitos para servidores concurrentes

El avance de las tecnologías ha supuesto que la comunicación y la transferencia de datos a través de la red sea fundamental a nivel mundial. Esto supone que la infraestructura informática sea capaz de gestionar y procesar un gran número de peticiones a la vez.

Cuando se va a proceder a instalar y configurar un servidor, es esencial analizar cuál va a ser el servicio que va a ofrecer y cuántas conexiones será capaz de atender.

Si el servidor está destinado a **recibir un número pequeño de procesos y cuya atención supone emplear una gran cantidad de recursos,** es más aconsejable configurarlo como **servidor iterativo**. Este solo atenderá una petición y hasta que no finalice no cogerá la siguiente. Estos servidores suelen hacer uso del ***protocolo UDP***.

Si el servidor tiene que **soportar una gran cantidad de peticiones y además se tiene que garantizar que sean resueltas correctamente, es mejor hacer uso de un servidor concurrente**. Este permite atender a varios clientes de forma simultánea. Estos servidores suelen hacer uso del ***protocolo TCP***.

Ambos están continuamente esperando por la llegada de una petición por parte del cliente.

En la actualidad, la mayor parte de servidores hacen uso de los servidores concurrentes. Esto supone que no exista un colapso cuando se realizan miles de peticiones a un mismo servicio web, por ejemplo, desde diversos dispositivos.

2.5. Implementación de comunicaciones simultáneas

Existen diferentes tecnologías para llevar a cabo la comunicación entre cliente y servidor. A partir del mecanismo más básico que son los ***sockets*** existen algunas técnicas que permiten implementar dicha comunicación simultánea a un más alto nivel.

- RMI

La técnica más utilizada es la invocación a métodos remotos, RMI (*Remote Method Invocation*). Dicha invocación implica que el objeto solicitado estará en una red diferente. Consiste en la petición de un objeto por parte del cliente al servidor. El cliente debe realizar dicha petición cumpliendo con todos los parámetros definidos en el servidor para este servicio. Esta petición es procesada por el servidor que comprueba dicho acceso. El servidor, una vez verifica dicho acceso, envía la respuesta con el objeto solicitado al cliente.

Los **parámetros obligatorios para realizar una invocación** son:

- **Objeto servidor o remoto**: es el encargado de recibir la petición de acceso al método y la procesa.
- **Objeto cliente**: es el objeto que realiza la invocación al método remoto. Este realiza una petición al objeto servidor, quien le dará una respuesta.
- **Método invocado**: es el servicio invocado al que se accede y debe de ser realizado con todos sus parámetros. Este es quien comunica por mensajes la petición al objeto servidor.
- **Valor de retorno**: valor que se envía al objeto cliente como finalización del proceso de invocación.

- RPC

Existe una variante de esta técnica cuyo concepto es similar al RMI, es el **RPC** (***Remote Procedure Call***). La diferencia con el RMI es la estructura de programación de cada una de ellas. El RMI está basado en la programación orientada a objetos y el RPC en la programación estructurada. El cliente en el RPC realiza una petición directamente al método del servidor.

- Servicios web

Este tipo de técnicas de comunicación son estándares para cualquier tipo de acceso entre cliente y servidor. Además, existen algunas que definen la comunicación propia bajo entornos web, como son los servicios web. La mayor parte de los servicios web basan su comunicación mediante el uso de los protocolos SOAP y REST.

- **SOAP** (*Simple Object Access Protocol*): este protocolo define cómo dos objetos van a comunicarse. En SOAP esta comunicación se hará usando el lenguaje *XML*. Tanto los mensajes como el contenido de los mismos se realizan usando este lenguaje. Debido a que el lenguaje no cambia, y tanto cliente como servidor implementan esta misma comunicación, la petición se procesa de forma automática.
- **REST** (*Representational State Transfer*): estos servicios no se ven obligados a utilizar XML como lenguaje, sino que permiten otros como *JSON*, cuyo uso está muy expandido.

El uso de este viene definido por el tipo de operaciones que se quieren llevar a cabo.

Las diferentes operaciones son: ***POST, GET, PUT y DELETE***.

Una de las ventajas que ofrece **REST** frente a **SOAP** es que los mensajes que son enviados ya definen todo lo necesario para ser procesados en el servidor, por lo que ni cliente ni servidor deben almacenar el estado de cada uno de ellos.

2.6. Verificación de la disponibilidad del servicio

A la hora de prestar un servicio es importante comprobar si este se encuentra disponible para el acceso de los clientes. En este caso, este procedimiento es tarea del **administrador de red**. Estos suelen estar sometidos a una disponibilidad de prácticamente un 100% durante mucho tiempo.

Esta exigencia puede provocar que ocurran fallos, por lo que este tipo de servidores cuentan con diferentes herramientas de monitorización de su estado. Así, es posible comprobar la disponibilidad de forma inmediata y realizar las acciones necesarias de mantenimiento.

Una de las formas que existen para comprobar su estado es realizar **peticiones programadas a través de la URL del servicio web**. Periódicamente se analizará el tiempo de respuesta del mismo y se comprobará el tiempo de carga de los recursos creados en dicho servicio.

Hoy en día existen servicios que contienen peticiones a datos que son especialmente sensibles a caídas. **Existen, por tanto, herramientas que permiten garantizar una disponibilidad de prácticamente el 100% sin pérdidas de datos. Esto se conoce como alta disponibilidad**.

Estos servidores permanecen activos junto con unas réplicas de sí mismos que permiten que, en caso de fallos, automáticamente se encargue el otro servidor de procesar los servicios.

Además, aquellos servicios sometidos a peticiones de forma masiva pueden hacer uso de **balanceadores de carga**, que se encargarán de distribuir las peticiones entre los distintos servidores para evitar un colapso de respuesta en el servicio. Esto, además de evitar colapsos, optimiza el rendimiento y añade una mayor velocidad y flexibilidad al servicio.

2.7. Depuración y documentación de aplicaciones

A lo largo de este módulo se han visto diferentes clases y algunos de los métodos que proporciona *JAVA* para trabajar con los servicios en red. En cada uno de los apartados se ha indicado el enlace correspondiente para poder estudiar con más detalle dichas clases. Siempre que se desarrollen aplicaciones para *java* es necesario trabajar con el API de *java* al lado. Esto permitirá que la tarea resulte más sencilla.

Este tipo de documentación denominada API no solo está para JAVA, sino para cualquier aplicación o servicio de red. **Una API proporciona un conjunto de métodos o funciones accesibles para los desarrolladores de software de cualquier tipo de aplicaciones**. Es importante tener en cuenta la versión de Java con la que se está trabajando, puesto que no es posible utilizar métodos de las versiones posteriores y algunos de versiones anteriores que se hayan quedado obsoletos.

Estas API contienen un catálogo documentado y detallado de funciones y parámetros definidos, que indican la forma correcta de acceder a dichas características. Son publicadas normalmente en los sitios web de sus desarrolladores.

Cuando se quiere desarrollar, es importante apoyarse en una buena documentación. Durante el desarrollo es aún más importante depurar correctamente la aplicación o servicio que se está programando. La generación de excepciones controladas y las distintas herramientas de depuración ofrecidas por los IDE de desarrollo ayudarán a optimizar y mantener de una forma más eficiente dichas aplicaciones.

Bibliografía

Sánchez campos, Alberto; Montes Sánchez, Jesús. (2013). *Programación de servicios y procesos. CFGS*. España: Ra-ma Editorial.

Ramos Martín, Mª Jesús. (2013). *Programación de Servicios y Procesos*. España: Garceta.

ILERNA Online