

SISTEMAS DISTRIBUIDOS

Resumen completo

Teoría + Seminarios

ÍNDICE :

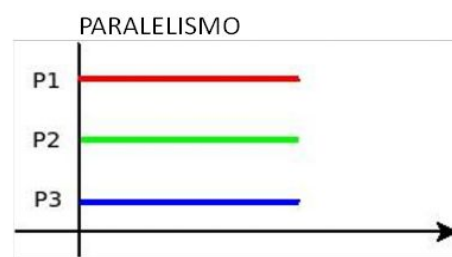
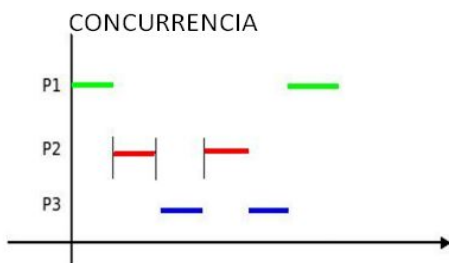
TEORÍA	
1.1. Introducción a los sistemas distribuidos	2
1.2. Errores en los sistemas distribuidos	5
1.3. Desafíos de los sistemas distribuidos	6
2.1. Modelos de sistema	8
2.2. Comunicación entre procesos	11
2.3. Comunicación asíncrona	16
3.1. Sincronización en sistemas distribuidos	22
3.2. Depuración distribuida	27
4. Sistemas peer-to-peer (P2P)	32
5. Sistemas de ficheros distribuidos	36
SEMINARIOS	
1. Administración de SO e introducción a Python	42
2. Sockets en Python	45
3. Servicios Web y REST API en Python	48
4. Node-RED	54
5. Protocolo MQTT	62
6. Arquitecturas orientadas a servicios y WS-BPEL	64
7. Introducción a ESB	68
8. Introducción a Mule con Anypoint	70
9. Introducción a Blockchain	75
10. Git	80

PARTE 1: TEORÍA

TEMA 1.1. - INTRODUCCIÓN A LOS SISTEMAS DISTRIBUIDOS

CONCEPTOS PREVIOS:

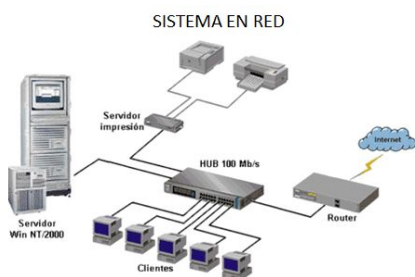
- **Concurrencia:** Se da cuando dos procesos se ejecutan de forma simultánea en una misma máquina, sin llegar a ser al mismo tiempo. Un mismo proceso se divide en hilos, sin afectar al resultado final.
- **Paralelismo:** Capacidad de un sistema de ejecutar dos o más procesos concurrentes al mismo tiempo.



- **Programación distribuida:** Modelo para resolver problemas empleando una red de computadores separados físicamente pero conectados entre sí.

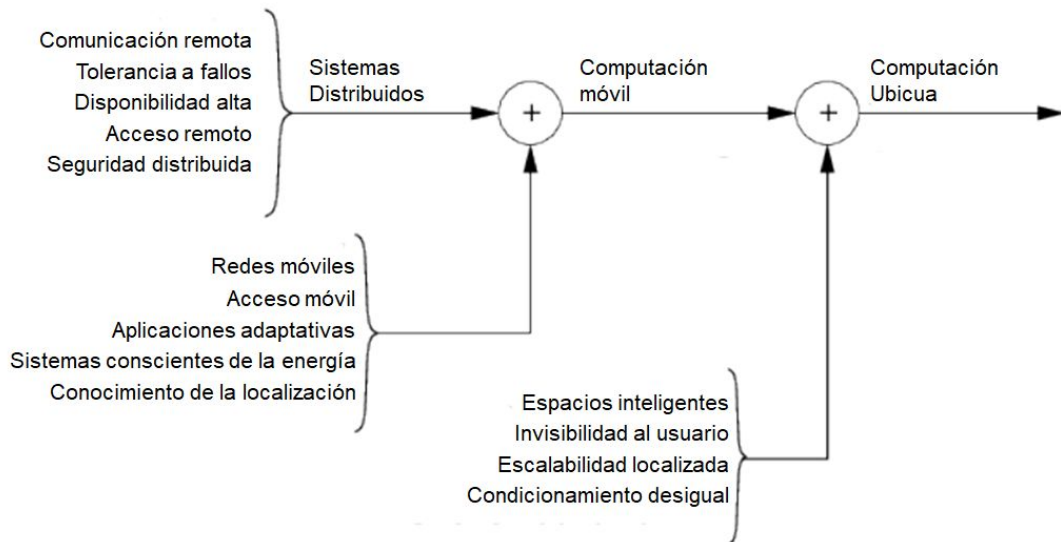
EVOLUCIÓN DE LOS SISTEMAS:

- **Sistemas centralizados:** Existía un único host con todos los recursos centralizados, a los cuales los usuarios accedían mediante terminales de texto, por lo que eran muchos usuarios conectados a un mismo equipo.
- **Sistemas unipersonales:** Un equipo por usuario, cada uno con sus propios recursos diferentes del resto.
- **Sistemas en red:** Equipos independientes unidos entre sí por una red, tal que un usuario puede acceder a los recursos públicos de más de un equipo.
- **Sistemas distribuidos:** Parte de la misma premisa que los sistemas en red, pero de tal forma que el usuario sólo percibe la red de computadores como una sola máquina, y no le interesa conocer la estructura del sistema.



SISTEMAS UBICUOS:

Introducidos como concepto por Mark Weiser en 1991, define la capacidad de un entorno de estar conectado mediante un sistema distribuido sin que el usuario pueda percibirlo. Un claro ejemplo es la domótica, o automatización de las viviendas. Los sistemas ubicuos actualmente son poco comunes, pero están en constante desarrollo.



CARACTERÍSTICAS DE LOS SISTEMAS DISTRIBUIDOS:

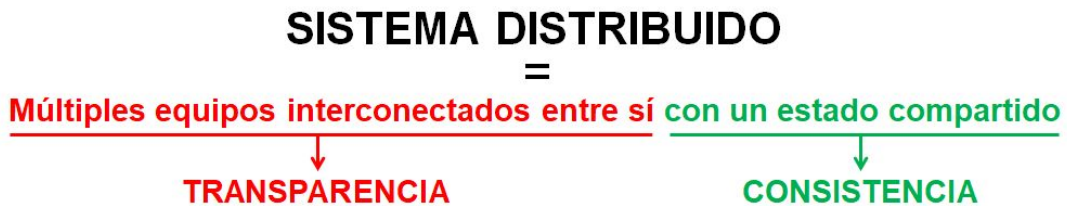
- Las diferencias entre todas las máquinas y la complejidad de los algoritmos de comunicación entre ellas son **ocultas** para el usuario.
- Accedan desde donde accedan, siempre se hará de la **misma forma**.
- Deben ser fácilmente **escalables** (sencillos de ampliar).
- Los servicios ofrecidos por un sistema distribuido deben estar **disponibles** en todo momento aunque estos fallen o estén siendo reparados o sustituidos.
- Normalmente, se estructuran en niveles, siendo una de las más frecuentes la estructuración como **middleware**, es decir, un software que sirve de puente entre el sistema operativo y los servicios del sistema distribuido.

COMPOSICIÓN DE UN SISTEMA DISTRIBUIDO:

- **Múltiples ordenadores:** Un sistema distribuido siempre se compone de varios sistemas independientes, cada uno con su CPU, memoria, etc...
- **Interconexión entre los equipos:** Todos los equipos que componen un sistema distribuido deben sincronizarse y comunicarse entre ellos a través de una línea o red de interconexión.
- **Estado compartido:** Estos equipos realizarán un mismo trabajo de forma conjunta, así que deben tener la misma visión del estado actual del sistema distribuido al que pertenezcan.

OBJETIVOS DE UN SISTEMA DISTRIBUIDO:

Además de la compartición de recursos tanto hardware como software, un sistema distribuido debe buscar otras dos características principales:



TRANSPARENCIA:

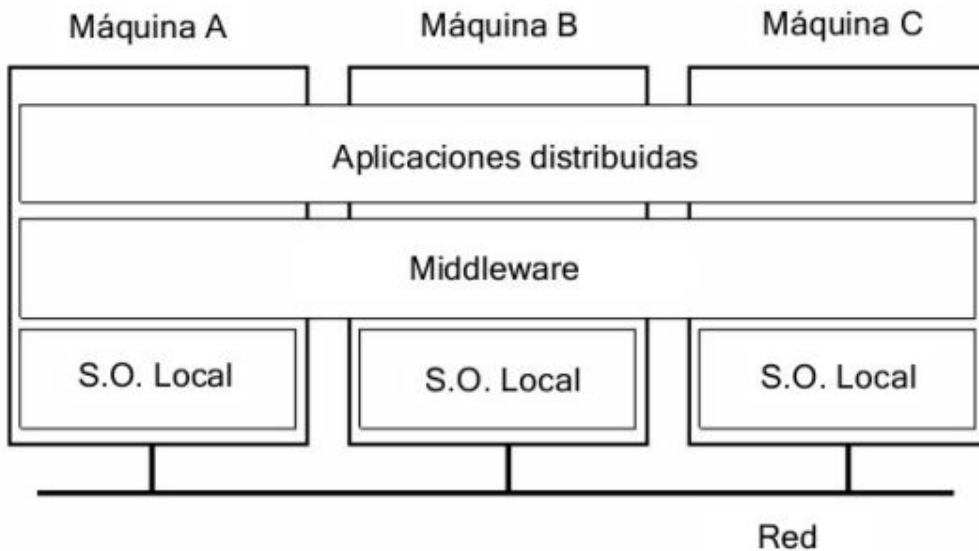
El usuario debe ser capaz de trabajar con el sistema distribuido con total normalidad incluso si se producen fallos o este se modifica para escalarse o mejorarse. Además, no debería conocer donde se encuentran los recursos compartidos ni, en general, el funcionamiento técnico del sistema. Existen varios tipos de transparencia: de acceso, de fallos, de ubicación, de concurrencia...

CONSISTENCIA:

- **De actualización:** Surge por la posibilidad de que se produzcan errores o inconsistencias cuando varios equipos pretenden acceder al mismo recurso al mismo tiempo (exclusión mutua). Para que esto no ocurra, se deben seguir las propiedades ACID mediante el uso de **transacciones**:
 - **Atomicidad:** Asegura que la operación se ha realizado o no.
 - **Consistencia:** Sólo se empiezan las operaciones que pueden acabarse.
 - **Aislamiento:** Una operación no puede afectar a otras (Isolation).
 - **Durabilidad:** Si una operación empieza, esta persiste hasta que acabe.
- **De réplica:** Un conjunto de datos debe mantenerse replicado en muchos equipos a través de multicast. Esto es vital, por ejemplo, en juegos online con más de un jugador en el mismo entorno.
- **De caché:** Surge cuando un usuario modifica ciertos datos que residen en las memorias caché de otros usuarios, volviéndose estas obsoletas. Hay técnicas de gestión de memoria para asegurar la consistencia de las cachés.
- **De reloj:** Muchos eventos realizados en sistemas distribuidos guardan la hora a la que este se ha realizado, pero no se puede asegurar que todos los equipos interconectados tengan la misma hora física. No existe una forma genérica de arreglar esto, ya que el tiempo de transmisión en una red es algo bastante impredecible.
- **De interfaz de usuario:** Toda interacción que realice un usuario sobre un sistema distribuido debe ser percibida por el mismo con un retardo inferior a una décima de segundo.

MIDDLEWARE:

En el contexto de los sistemas distribuidos, un middleware es un software requerido para facilitar las interacciones cliente-servidor y asegurar un acceso transparente a los diferentes recursos no locales del sistema. Un ejemplo de middleware en otro contexto lo encontramos en la máquina virtual de Java, que asegura que todo código pueda ejecutarse en cualquier sistema que posea dicha máquina virtual.



TEMA 1.2. - ERRORES EN LOS SISTEMAS DISTRIBUIDOS

PROBLEMAS DE LOS SISTEMAS DISTRIBUIDOS:

Que un sistema sea distribuido implica diferentes problemas:

- La dificultad de tener una visión global del sistema.
- La complejidad en su gestión y mantenimiento al estar entre varias máquinas.
- Los errores y retrasos en las comunicaciones.
- La baja calidad de los servicios debido a problemas en la red.

EJEMPLOS DE ERRORES EN SISTEMAS DISTRIBUIDOS:

- En 2019, **Airbus** registró retrasos importantes en las entregas de sus aviones ya que las filiales españolas y francesas de la empresa usaban versiones distintas del mismo sistema distribuido, haciendo que muchos formatos de archivos fuesen incompatibles para una de las partes.
- En 2011, **Amazon** usaba algoritmos para detectar cuándo un producto de la competencia subía su precio, para hacerlo ellos también. Sin embargo, el sistema detectó un mismo producto de Amazon, pero de otro vendedor, como de la competencia, haciendo que sus precios subieran rápidamente hasta los casi 24 millones de dólares. El mismo libro se vendía, usado, a 35 dólares.

TEMA 1.3. - DESAFÍOS DE LOS SISTEMAS DISTRIBUIDOS

HETEROGENEIDAD:

El concepto de heterogeneidad aplicado a los sistemas distribuidos se refiere a que varios equipos o software con componentes distintos pueden comunicarse entre sí con medios comunes, sin importar la arquitectura, red de comunicación, sistema operativo, etc...

En este caso, son los **servicios del middleware** los que proveen la heterogeneidad mediante la abstracción de la programación, enmascaramiento de las redes, hardware, sistemas operativos y lenguajes de programación:

- **Servicios de comunicación:** Llamadas o invocaciones a procedimientos remotos y sistemas de notificaciones de eventos.
- **Servicios para sistemas de información:** Permiten gestionar datos de forma distribuida.
- **Servicios de control:** Permite controlar el acceso de las aplicaciones a los datos distribuidos a tiempo real.
- **Servicios de seguridad:** Autenticación y autorización, cifrado, auditorías...

Otro concepto importante es el de **código móvil**, es decir, fragmentos de código que pueden ejecutarse en cualquier equipo sin importar dónde se hayan creado.

EXTENSIBILIDAD:

La extensibilidad es la capacidad de un sistema de actualizarse y, en resumen, poder extenderse. Para que esto sea posible, la especificación y documentación de las interfaces software clave del sistema deben ser visible a los desarrolladores. Un sistema distribuido puede ser extensible a dos niveles:

- **De hardware:** Inclusión de nuevos servidores
- **De software:** Se añaden nuevas funciones o se actualizan las existentes.

Para facilitar la extensibilidad, existen **normas y estándares** por organizaciones reputadas que pueden contribuir a la longevidad, calidad, bajo coste, comprensión e integridad de los sistemas, como la ISO, IEEE, W3C, ITU-T o OMG. Hay algunos estándares, como TCP/IP, considerados de facto por su popularidad.

Si todas las normas y estándares, además de la arquitectura usada y la especificación de las interfaces son públicas, entonces estamos ante un **sistema abierto**. De lo contrario, se trata de un sistema cerrado o propietario. Además, si el código fuente también es público y cualquier usuario puede modificarlo o recompilarlo, es un sistema de **código abierto**. Es el caso, por ejemplo, de Unix.

SEGURIDAD:

La seguridad de un sistema distribuido debe garantizar tres factores fundamentales, además de tener en cuenta los componentes más vulnerables y los tipos de ataque más comunes o dañinos:

- **Confidencialidad:** sólo los elementos autorizados tienen acceso
- **Integridad:** sólo los elementos autorizados pueden modificar los elementos
- **Disponibilidad:** los datos se mantienen disponibles.

ESCALABILIDAD:

Un sistema es escalable si su efectividad no se ve afectada tras un incremento en el número de servidores o usuarios, siempre y cuando la arquitectura y los algoritmos lo permitan. Hacer esto conlleva riesgos que se deben tener en cuenta:

- **Coste** de los recursos físicos
- **Control** de posibles pérdidas de prestaciones del sistema
- **Prevención** de desbordamientos, por ejemplo, en el número de IPs.
- **Evitar** cuellos de botella, es decir, llegan más solicitudes al sistema de las que se pueden administrar, quedando estas en espera.

Algunas formas de solventar estos riesgos involucran el uso de cachés o particionar la información para su distribución.

TOLERANCIA A FALLOS:

En un sistema distribuido los fallos son parciales, es decir, que los componentes no afectados por un fallo siguen funcionando. Se distinguen las siguientes fases:

- **Detección** de los fallos
- **Enmascaramiento** de los fallos, es decir, el proceso de ocultación
- **Tolerancia** de fallos, en caso de que no merezca la pena enmascararlos. Un ejemplo claro son los de página no encontrada (error 404)
- **Recuperación** frente a fallos, especialmente de los datos
- **Redundancia:** La tolerancia a fallos puede conseguirse mediante réplicas de la base de datos o del nombre de dominio, o por múltiples rutas de acceso.

CONCURRENCIA:

Asegura que varios usuarios puedan acceder al mismo recurso compartido **a la vez** y manteniendo la integridad de los datos, que deberán llegar a cada receptor en orden y completos.

La concurrencia es algo que debe conseguir el programa como tal, ya que no hay un reloj global.

TIPOS DE TRANSPARENCIA:

Acceso	Oculto el cómo se accede a cada recurso y su representación.
Ubicación	Oculto el dónde se ubican los recursos.
Migración	Oculto que un recurso migre de una ubicación a otra.
Reubicación	Oculto que un recurso migre mientras se está utilizando.
Replicación	Oculto que puedan existir múltiples réplicas de un recurso.
Concurrencia	Oculto que un recurso pueda ser accedido concurrentemente.
Fallos	Oculto los fallos y la recuperación de los mismos.
Persistencia	Oculto que un recurso esté en memoria volátil o persistente.

TEMA 2.1. - MODELOS DE SISTEMA

DEFINICIÓN:

Cuando una aplicación trabaja con un gran volumen de datos, interesa convertirla en un sistema distribuido. El modelo de un sistema distribuido es una descripción simple pero consistente del diseño de dicho sistema distribuido. Existen diversos tipos de modelo:

- **Arquitectónicos:** Explica las relaciones entre los componentes del sistema, por ejemplo en un modelo cliente-servidor.
- **Fundamentales:**
 - **De interacción:** prestaciones del sistema y dificultad de imponer límites temporales dentro del mismo.
 - **De fallos:** especifica los fallos que podrían producirse en los procesos del sistema o sus canales de comunicación.
 - **De seguridad:** potenciales amenazas para el sistema.

DIFICULTADES Y AMENAZAS DE LOS SISTEMAS DISTRIBUIDOS:

- **Modelos de uso variables:** Los componentes del sistema debe estar listos para soportar variaciones en la carga de trabajo y tener requisitos especiales para comunicaciones además de, a ser posible, estar desconectados.
- **Amplio rango de entornos:** La arquitectura, sistema operativo o tipo de red del usuario abarca una gran cantidad de posibilidades.
- **Problemas internos:** Desincronizaciones, fallos de hardware...
- **Amenazas externas:** Intrusiones y ataques.

MODELOS ARQUITECTÓNICOS:

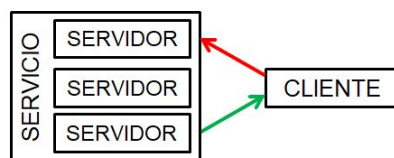
Un modelo arquitectónico simplifica y abstrae las funciones de los componentes del sistema, y posteriormente considera su ubicación e interconexión con el resto de los componentes. En un sistema distribuido, estos modelos clasifican los procesos en **clientes**, **servidores** e **iguales**, aunque existen variantes.

CAPAS DE SOFTWARE:

Aplicaciones y otros servicios	
Middleware: Enmascara las comunicaciones entre equipos y proporciona modelos de programación y bloques útiles para la comunicación entre varios software, además de proporcionar servicios para las aplicaciones.	
Plataforma	Sistema operativo
	Arquitectura y hardware

ARQUITECTURAS DE SISTEMA:

- **Modelo cliente-servidor:** Los procesos clientes interactúan con los procesos servidores para acceder a los recursos compartidos. Un servidor puede ser cliente de otros servidores.
- **Servicios proporcionados por múltiples servidores:** Un servicio de un sistema puede ser proporcionado por varios servidores, al estar distribuido o replicado en varias máquinas.
- **Proxy y cachés:** Una caché puede definirse como un almacén de objetos recientes que se encuentran más próximos a los clientes. Cuando se solicita un servicio, se comprueba antes si está en la caché, que puede estar en un servidor proxy o en el equipo de cada cliente.
- **Proceso peer to peer (P2P):** También llamado de igual a igual, todos los procesos interactúan entre ellos sin distinciones entre clientes y servidores, reduciendo los retardos en la comunicación.

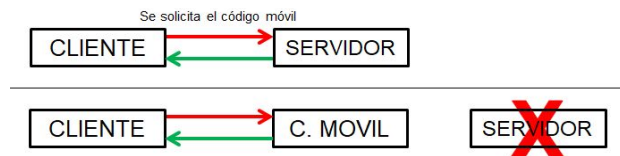


VARIACIONES EN EL MODELO CLIENTE-SERVIDOR:

A la hora de añadir nuevos dispositivos o equipos a nuestro sistema distribuido o trabajar con recursos limitados, existen diversas variaciones dentro del modelo cliente-servidor para permitir el funcionamiento del sistema distribuido con un menor coste y complejidad.

EJEMPLOS DE VARIACIONES DEL MODELO CLIENTE-SERVIDOR:

- **Código móvil:** A través de un navegador o máquina, los usuarios seleccionan un enlace que se descargará y se ejecutará, de tal forma que el usuario no interactúa con el servidor.



- **Agente móvil:** Programa en ejecución que puede trasladarse de un equipo a otro de la misma red, haciendo solicitudes a los recursos locales.
- **Computadora de red:** Usa un servidor de archivos remoto para descargar el sistema operativo y software necesario en el computador del usuario. Ese servidor se encarga de guardar y gestionar los archivos.
- **Cliente ligero:** Soporta una interfaz de usuario sobre un computador local mientras la ejecución de los programas se desarrolla en otro computador remoto. Permite ejecutar muchas más aplicaciones simultáneamente, pero con mayor latencia de red.
- **Dispositivo móvil y enlace espontáneo a red:** Los usuarios usan sus móviles para el uso de servicios locales y remotos de la red. Es más fácil de utilizar, pero más inseguro y con una conectividad limitada.

REQUISITOS DE DISEÑO PARA ARQUITECTURAS DISTRIBUIDAS:

- **Prestaciones:** Balance de cargas, capacidad de respuesta, rapidez...
- **Calidad de servicio:** Fiable, seguro, sencillo y cómodo de usar...
- **Aspectos de fiabilidad:** Tolerancia a fallos, métodos de seguridad...
- **Otros:** Uso de caché y servidores replicados, por ejemplo.

MODELO DE INTERACCIÓN:

Dentro de cualquier sistema distribuido siempre existirán una serie de limitaciones para las prestaciones del mismo. El modelo de interacción expone, de forma teórica y abstracta, la dificultad de imponer límites temporales en el sistema:

- La comunicación en red de por sí establece ciertos límites.
- El retraso en la entrega de un mensaje es impredecible.
- No hay un tiempo global para todo el sistema para medir estos tiempos.
- No determinista, es decir, distintos resultados en distintas ejecuciones.
- Difícil de depurar.

ALGORITMO DISTRIBUIDO:

Define todos los pasos que lleva a cabo cada proceso del sistema, incluyendo los mensajes que se transmiten entre ellos. En resumen, es un algoritmo diseñado para aprovechar las características de los sistemas distribuidos.

PRESTACIONES:

- **Latencia:** Retardo entre el envío de un mensaje y su recepción.
- **Ancho de banda:** Cuánta información puede transmitirse en cierto tiempo.
- **Fluctuación o jitter:** Variación del tiempo en la entrega de varios mensajes.

RELOJES:

Cada equipo tiene su propio reloj interno, que sí puede usarse para medir tiempos en procesos locales. Sin embargo, un sistema distribuido no puede asumir que los relojes de todos los sistemas están sincronizados, por lo que no existe un reloj global, aunque sí se puede usar un reloj de referencia. Se denomina tasa de **deriva de reloj** a la evolución en la diferencia de tiempo entre los relojes de referencia y locales.

MODELOS SÍNCRONOS Y ASÍNCRONOS:

- **Síncronos:** Conoce las características temporales de los relojes locales y establece límites a los procesos globales como la entrega de mensajes o ejecución de procesos. Con estos límites, teóricamente se puede tener una idea de cómo funcionará el sistema, pero en la práctica es imposible garantizar estos límites
- **Asíncronos:** No tiene las limitaciones que establecen los modelos síncronos, por lo que los sistemas distribuidos reales suelen preferir esta opción. Una solución válida en un modelo asíncrono también lo es para uno síncrono, pero no al revés.

MODELOS DE FALLO:

Este modelo estudia las posibles causas de los fallos que pueden darse en un sistema distribuido para predecir y entender sus consecuencias. Este modelo suele distinguir entre varios tipos de fallo:

- **Según la entidad:** Fallos de proceso o comunicación
- **Según el problema:** Una acción que se debería poder hacer no logra llevarse a cabo, o directamente se desconocen sus causas y consecuencias (bizantinos).

TEMA 2.2. - COMUNICACIÓN ENTRE PROCESOS

INTRODUCCIÓN:

Los papeles de las entidades que se comunican entre sí determinan cómo deben comunicarse. Esto es, qué **protocolo** se debe usar (TCP o UDP, entre otros), qué **patrón de comunicación** (por ejemplo, cliente-servidor), y el **tipo de comunicación preferible** (paso de mensajes, recursos compartidos...), suponiendo que el lenguaje y la arquitectura del programa lo permiten.

COMUNICACIÓN BÁSICA:

En el paso de mensajes, siempre se emplean dos operaciones básicas, **send** y **receive**, que envían y reciben los mensajes respectivamente. Además, cada destino tiene asociada una cola de mensajes donde los procesos emisores insertan mensajes que será posteriormente extraídos por los receptores.

COMUNICACIÓN SÍNCRONA Y ASÍNCRONA:

- **Síncrona:** Se usa cuando ambos procesos deben coincidir en el tiempo, como en videollamadas. Emisor y receptor se sincronizan tras cada mensaje, por tanto send y receive son ambas operaciones bloqueantes:
 - **Emisor:** se bloquea hasta que se realiza el receive.
 - **Receptor:** se bloquea hasta que se envía un mensaje.
- **Asíncrona:** El emisor no es bloqueante y copia los mensajes en un búfer local, así que los comunicantes no deben coincidir en el tiempo, como en SMS. El receptor, sin embargo, puede o no ser bloqueante:
 - **Receptor no bloqueante:** El proceso receptor se sigue ejecutando tras realizar el receive, proporcionando un búfer de mensajes.
 - **Receptor bloqueante:** Si el entorno soporta múltiples hilos por cada proceso, uno se encargará del recibo de mensajes mientras el resto ejecutan el programa. Es menos eficiente, pero menos complejo.

DESTINO DE LOS MENSAJES:

Los mensajes se envían a direcciones compuestas por pares, es decir, una IP junto con un puerto determinado. Un **puerto** es el destino de un mensaje dentro de cada máquina. Una máquina puede tener muchos puertos receptores, pero un mensaje sólo puede tener un puerto receptor, aunque tenga muchos emisores.

FIABILIDAD Y ORDENACIÓN:

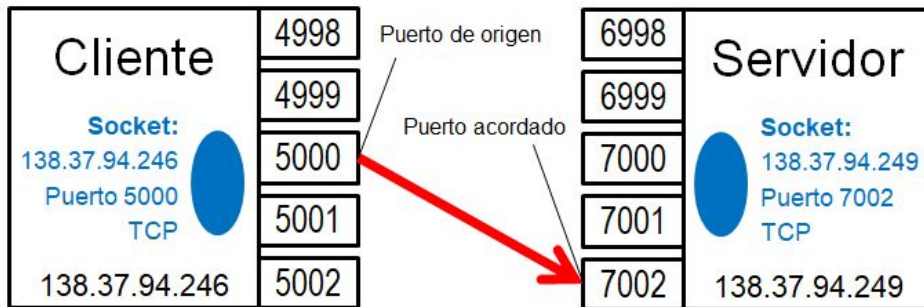
- **Fiabilidad:** Un mensaje siempre se entregará al receptor aunque se pierdan paquetes (bloques entre los que se divide el mensaje) por el camino.
- **Ordenación:** Muchas aplicaciones requieren que los mensajes se entreguen en el orden en el que fueron emitidos.

PIPES:

Era el mecanismo antiguo para constituir el modelo de paso de mensajes, y creaba un cauce unidireccional sin sincronización, por lo cual no resulta ser útil en sistemas distribuidos. Además, los pipes no tienen nombres ni identificadores, además de no poder enviar mensajes discretos, es decir, que constan de varias partes.

SOCKETS:

Un socket se asocia a un puerto local de una máquina y a su dirección IP junto con el protocolo que se empleará (TCP o UDP), de tal forma que pueden enviar y recibir mensajes. La comunicación entre procesos siempre se dará entre dos sockets que actúan de emisor y receptor. Su interfaz se basa en el modelo cliente-servidor.



UDP - COMUNICACIÓN DE DATAGRAMAS:

UDP es un protocolo de comunicación por datagramas no fiable sin garantía de entrega. Los procesos send y receive crean un socket asociado a un puerto de cada máquina, de tal forma que se establece una **comunicación asíncrona con receive bloqueante**, el cual entrega el mensaje y el puerto de origen:

- **Send** toma el control cuando el mensaje llega a las capas UDP o inferiores.
- Estas capas dejan el mensaje en la **cola** del socket asociado al puerto destino.
- **Receive** extrae el mensaje de la cola con bloqueo indefinido o timeout.

Al no ser un protocolo fiable, eso sí, los mensajes podrían perderse en el camino debido a una multitud de errores. Además, provoca grandes cargas de trabajo.

TCP - COMUNICACIÓN DE FLUJOS:

A diferencia de los datagramas, los flujos contienen información oculta sobre los mensajes enviados, como su tamaño, mensajes perdidos o duplicados, control de flujo... convirtiéndolo en un protocolo más lento, pero mucho más fiable que UDP:

- El cliente crea un socket **solicitando** que se establezca una conexión.
- El servidor crea otro socket de **escucha** con una cola de peticiones.
- Al **aceptar** la conexión, se crea un nuevo socket conectado al del cliente.
- Este socket sirve de **cauce**. Cada proceso lee y escribe según sea necesario.
- Si un socket se **cierra**, se transmiten los datos pendientes.

Si no hay datos para leer o se llena la cola, se producirá un bloqueo para impedir la pérdida de datos. Este protocolo también permite atender varios clientes a la vez sin perder fiabilidad. Si hay un error de red, la conexión se rompe.

FUNCIONES UDP Y TCP:

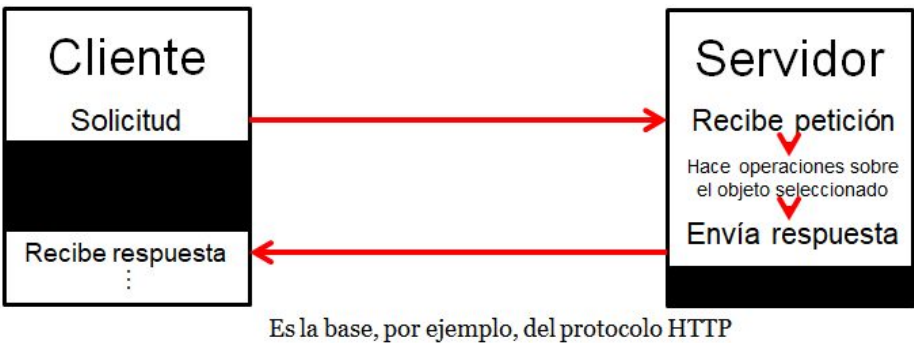
TCP		UDP	
Cliente	Servidor	Cliente	Servidor
Crear socket	Crear socket	Crear socket	Crear socket
Conectarse	Enlazar puertos	Send y Receive	Send y Receive
Send y Receive	Escuchar	Cerrar socket	Cerrar socket
Cerrar socket	Acepta conexión		
	Send y Receive		
	Cerrar socket		

CLASES ÚTILES EN JAVA:

- **DatagramPacket:** Soporta los datagramas enviados durante una conexión **UDP**. El constructor para el emisor recibe el mensaje, su longitud, y la IP y puerto destino, mientras que del receptor sólo recibe un array de bytes para recibir el mensaje.
- **DatagramSocket:** Soporta los sockets de una conexión **UDP**. El constructor para el servidor recibe el puerto que se desea asociar, mientras que el del emisor no recibe nada. Realiza los métodos send y receive, entre otros.
- **ServerSocket:** Acepta solicitudes de conexión del protocolo **TCP**.
- **Socket:** Crea un socket para el protocolo **TCP** que recibe la IP y el puerto del receptor. El servidor recibe el resultado del *accept* de ServerSocket.

PROTOCOLO PETICIÓN-RESPUESTA:

En los datagramas o flujos de datos se asignan identificadores a los mensajes, que se dividirán en peticiones y respuestas. En estos protocolos, la comunicación es síncrona, de tal forma que el cliente se bloqueará mientras espera una respuesta. Pueden ocurrir fallos por tiempo de espera o pérdida de mensajes.



IDENTIFICADORES DE LA INVOCACIÓN:

A la hora de invocar al protocolo petición-respuesta, el cliente genera, junto con la solicitud al servidor, un **idInvocacion** que el servidor añade a su respuesta para que el cliente pueda comprobar si es la esperada. Dicho ID se compone de dos partes:

- Un valor entero que asegura que cada invocación del mismo emisor es **única**.
- Un identificador del receptor para asegurar la **unicidad global**.

MODELO DE FALLOS PETICIÓN-RESPUESTA:

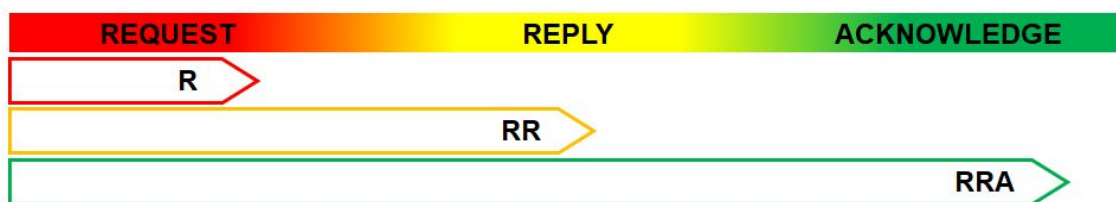
Los fallos abarcan una multitud de posibilidades: mensajes perdidos, fallos en los procesos, fallos de red, modificación de datos... algunos métodos que pueden ayudar a prevenirlos son:

- **Temporizadores:** Cuando se realiza una operación send, se da un tiempo para que se realiza el receive. Si este se acaba, se suele reintentar hasta que se recibe una respuesta o, en algunos casos, se retorna el mensaje de error.
- **Solicitudes duplicadas:** Se pueden reconocer los duplicados y filtrarlos al tener el mismo idInvocacion. Suelen ocurrir debido a reintentos tardíos.
- **Respuestas perdidas:** No es un problema si las operaciones del servidor se realizan repetidamente, dando los mismos resultados.
- **Historial:** Registro de respuestas enviadas para que pueda volver a enviarse al cliente sin necesitar repetir la operación.

PROTOCOLO DE INTERCAMBIO RPC:

RPC significa llamada a procedimiento remoto (**R**emote **P**rocedure **C**all), y se basa en la petición de un programa o servicio localizado en otra máquina de la red. Tres de los protocolos más usados para RPC son:

- **R:** Petición, el cliente no necesita respuesta del servicio.
- **RR:** Petición y respuesta, el servicio responde al cliente, que confirma su llegada al realizar otra petición. Común en entornos cliente-servidor.
- **RRA:** Petición, respuesta y confirmación. Una vez el cliente recibe respuesta, envía confirmación al servicio junto con las ID de la respuesta y la anterior. Además, vacía entradas antiguas del historial.



PROTOCOLO HTTP:

HTTP es el protocolo de petición respuesta más conocido que emplea TCP, ya que debe asegurar la entrega de los datos. Este protocolo ofrece métodos de autenticación del contenido y establece una conexión persistente y abierta al intercambio de mensajes entre el cliente y el servidor.

COMUNICACIÓN MULTICAST:

Este tipo de comunicación envía un único mensaje a todos los clientes de un grupo, aunque sin garantía de entrega ni orden establecido. Proporciona la infraestructura básica para el desarrollo de un sistema distribuido: tolerancia a fallos, búsqueda de servidores, propagación de notificaciones...

TEMA 2.3. - COMUNICACIÓN INDIRECTA

COMUNICACIÓN DIRECTA:

La comunicación entre el cliente y un servidor es **directa**. Obteniendo IP y puertos de destino, se posibilita el paso de mensajes entre dos entidades. Sin embargo, un cambio en la IP produciría errores en la comunicación al no existir intermediarios que notifiquen a los clientes del cambio.

DEFINICIÓN:

Se dice que la comunicación es indirecta cuando no hay acoplamiento directo entre emisor y receptor ya que existen intermediarios de por medio. La comunicación indirecta cumple dos propiedades:

- **Desacoplamiento de espacio:** El emisor no conoce la identidad de los receptores, pudiendo estos ser reemplazados o migrados.
- **Desacoplamiento de tiempo:** Emisor y receptor no tienen por qué coincidir en el tiempo.

VENTAJAS Y DESVENTAJAS:

- Ventajas:
 - Muy útil cuando los cambios son constantes
 - Se suelen gestionar para ofrecer servicios fiables.
 - Envía eventos en sistemas distribuidos, con receptores desconocidos.
 - Usado por grandes infraestructuras como la de Google.
- Desventajas:
 - Sobrecarga de rendimiento.
 - Dificultad para la gestión

TIEMPO Y ESPACIO DESACOPLADO:

		TIEMPO	
		<i>Acoplado</i>	<i>Desacoplado</i>
E S P A C I O	<i>Acoplado</i>	Los receptores deben ser conocidos y existir en ese momento, como en el paso de mensajes.	Los receptores deben ser conocidos, pero no tienen porqué existir en ese momento, como en correo.
	<i>Desacoplado</i>	Los receptores no deben ser conocidos, pero si existir en ese momento, como en multicast.	Los receptores ni deben ser conocidos ni tienen porqué existir en el momento. Esto es comunicación indirecta.

La **diferencia entre la comunicación asíncrona y el desacoplamiento de tiempo** es que la comunicación asíncrona necesita que los receptores **existan** en el momento de enviar el mensaje. Esto no es necesario en desacoplamiento de tiempo.

COMUNICACIÓN EN GRUPO:

Un emisor envía un mensaje que se entrega a un grupo de receptores sin necesidad de conocerlos, funcionando como abstracción de la comunicación multicast, pero más **fiable** y orientado a un rango más amplio de áreas como pueden ser las aplicaciones colaborativas o monitorización de sistemas. Los procesos pueden:

- **Unirse** o dejar un grupo.
- Enviar un mensaje al grupo con entrega **garantizada**.
- Realizar una única operación multicast
 - **Broadcast:** Mensaje enviado a todos los procesos del grupo.
 - **Unicast:** Mensaje enviado a un único proceso del grupo.

GRUPOS DE PROCESOS Y OBJETOS:

En un grupo de procesos, las entidades que se comunican enviándose mensajes entre ellas son procesos del sistema, mientras que los grupos de objetos procesan de forma concurrente (de forma simultánea) el mismo tipo de invocaciones. Los grupos pueden ser de los siguientes tipos:

- **Cerrados:** Solo los miembros del grupo pueden hacer multicast al grupo.
- **Abiertos:** Los procesos externos pueden hacer multicast.
- **Solapados:** Las entidades de un grupo pueden pertenecer a otros grupos.
- **No solapados:** Cada objeto o proceso solo pertenece a un grupo.

FIABILIDAD EN LAS COMUNICACIONES MULTICAST:

Para asegurar una conexión entre procesos fiable, deben cumplirse tres propiedades:

- **Integridad:** El del mensaje no se altera y solo se entrega una vez.
- **Validez:** Se garantiza la entrega del mensaje.
- **Acuerdo:** Al entregar el mensaje a un proceso, este llega a todos los procesos del grupo.

ORDEN DE ENTREGA DE MENSAJES A MÚLTIPLES DESTINOS:

Todos los procesos de un grupo están constantemente enviándose mensajes entre sí, y un sistema puede emplear tres tipos de ordenación distintos:

- **FIFO:** La entrega de los mensajes respeta el orden en el que los procesos del grupo los difundieron a la red.
- **Causal:** La entrega de los mensajes respeta el orden en el que la red los envió al resto de procesos, que en un sistema distribuido no tiene por qué coincidir con el orden en el que los procesos los difundieron.
- **Total:** Si un proceso entrega un mensaje antes que otro, entonces el resto de procesos también respetarán ese orden.

GESTIÓN DE LAS RELACIONES DE GRUPO:

Las entidades siempre podrán unirse o dejar un grupo, o incluso fallar. El servicio de relaciones entre grupos debe realizar 4 tareas fundamentales:

- **Ofrecer una interfaz para cambios:** Operaciones para crear y destruir procesos o añadirlos y eliminarlos de un determinado grupo.
- **Detección de fallos:** Monitoriza miembros inalcanzables o fallos de comunicación para catalogarlos como Suspected (esperados) o Unsuspected.
- **Notificaciones de cambios:** Los miembros de un grupo deben ser notificados cuando otro miembro sea añadido o eliminado.
- **Expandir las direcciones de grupos:** Al extender un mensaje por un grupo, el servicio debe obtener el identificador de las relaciones de grupo, no solo el identificador del propio grupo.

SISTEMAS PUBLICADOR-SUSCRIPTOR:

Sistemas donde los publicadores envían notificaciones sobre una serie de eventos a un servidor, que son sólo recibidos por las personas suscritas a dicho evento. El objetivo es relacionar suscripciones con eventos para, así, asegurar la correcta entrega de las notificaciones. Dentro de un grupo, solo los usuarios o procesos suscritos recibirán la notificación.

CARACTERÍSTICAS DE SISTEMAS PUBLICADOR-SUSCRIPTOR:

- **Heterogeneidad:** Todos los componentes del sistema distribuido trabajan juntos a la hora de tratar con notificaciones de eventos, estén o no diseñados para ello. Los objetos de un grupo se suscriben a determinados patrones de eventos para así tratar las operaciones resultantes.
- **Asincronía:** Los publicadores no están sincronizados con los suscriptores. Una notificación enviada por un publicador le llegará a los suscriptores en cuanto accedan al sistema.

CONJUNTO DE OPERACIONES:

- **publish(*e*):** Un publicador envía un evento *e* al sistema.
- **subscribe(*f*):** Un suscriptor que muestra interés en un tema se suscribe al mismo, con un filtro *f* que determina el tipo de notificaciones que recibirá.
- **unsubscribe(*f*):** Un suscriptor suscrito a un tema con un filtro
- **notify(*f*):** El sistema entrega los eventos con filtro *f* a los suscriptores
- **advertise(*f*):** Un publicador declara un tipo de evento *f* como anuncio.
- **unadvertise(*f*):** Recíproco del anterior.

MODELOS DE SUSCRIPCIÓN:

- **Canal:** Los publicadores envían eventos a un canal concreto y los suscriptores reciben todos los eventos que lleguen al mismo.
- **Tema:** El tema es un atributo de una notificación, así que los suscriptores pueden suscribirse sólo a determinados temas.
- **Contenido:** En lugar de recibir las notificaciones de un tema, las recibe sobre los contenidos que le interesa, siendo más general que los temas.
- **Tipos:** Las notificaciones son instancias de tipos definidos por el publicador, los cuales los suscriptores conocen y pueden suscribirse.
- **Contexto:** Circunstancias físicas como la localización o el tipo de máquina empleada por el suscriptor. Relacionado con la computación ubicua y móvil.

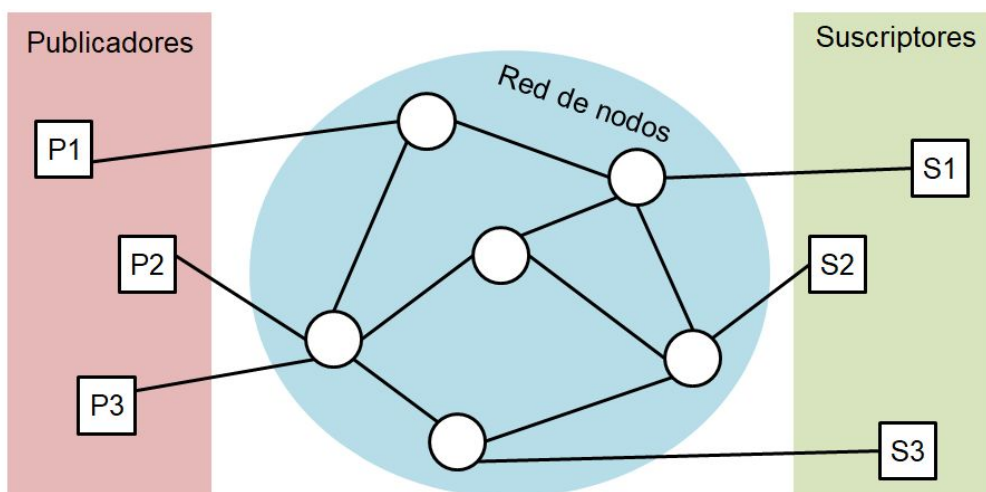
PROCESAMIENTO DE EVENTOS COMPLEJOS (CEP):

En sistemas más complejos como la bolsa de un país, las suscripciones pueden no ser suficientes para notificar determinados eventos más complejo. Por ello, se requiere un sistema que sea capaz de analizar cierto volumen de eventos y llegar a un evento final, partiendo de ciertos patrones predefinidos. Requiere un lenguaje específico.

Por ejemplo, si se detecta un terremoto y una bajada del nivel del mar, entre otros eventos, se puede llegar a notificar una posible alerta de tsunami a los residentes de esa zona, suscritos por contexto al sistema.

IMPLEMENTACIONES DE SISTEMAS PUBLICADOR-SUSCRIPTOR:

- **Centralizadas:** Hay un servidor en un único nodo que gestiona las publicaciones y las suscripciones, a través de una serie de mensajes punto a punto (P2P). Es un sistema fácil de implementar, pero poco escalable e ineficiente si el volumen de peticiones es grande.
- **Distribuidas:** En lugar de un agente o servidor centralizado, hay una red de ellos. Se adaptan a fallos de nodos individuales y son fácilmente escalables:



EJEMPLOS DE IMPLEMENTACIONES DISTRIBUIDAS:

- **Flooding:** Un publicador envía una notificación por broadcast o multicast a todos los nodos de la red, y se reparte a los suscriptores correspondientes.
- **Filtering:** Aquí, cada nodo debe contener una lista con todos sus nodos vecinos, de tal forma que las notificaciones se envían únicamente a los nodos que componen un camino hacia un suscriptor válido.
- **Rendezvous:** Se definen nodos “de encuentro” que actúan como espacio de eventos, de tal forma que una vez llega una cierta cantidad de notificaciones estas son enviadas, desde ahí, a los suscriptores correspondientes, empleado para evitar bloqueos y posibles colapsos en la red. El conjunto de estos nodos forma una red P2P junto con los publicadores y suscriptores.
- **Informed gossip:** Los nodos intercambian entre ellos de forma periódica y a través de cálculos estadísticos la información.
- **Advertisements:** Para evitar el exceso de tráfico, la propagación de suscripciones y el envío de notificaciones se realiza de forma simétrica.

Arquitectura publicador-suscriptor:

	Protocolos que determinan qué suscriptores recibirán las notificaciones
Ruta de eventos	Protocolos que determinan cómo llegarán los eventos a los suscriptores: flooding, filtering...
Interfaz de la red	Protocolos que determinan cómo se distribuirán los datos por la red: multicast, DHT, gossip...
Capa de transporte	Protocolos de transporte como TCP o UDP

COLAS DE MENSAJE:

Las colas de mensaje componen otra categoría dentro de la comunicación indirecta, proporcionando un servicio punto a punto (P2P). Los emisores envían mensajes a dichas colas, que son recogidas por los receptores. Las colas pueden considerarse como un middleware para los mensajes.

RECEPCIÓN DE MENSAJES EN COLAS:

Existen dos métodos por los que los receptores pueden recibir mensajes de las colas:

- **Receive (bloqueante):** La recepción se bloquea hasta que haya disponible un mensaje en la cola
- **Polling (no bloqueante):** La operación poll devolverá un mensaje avisando del estado actual de la cola: disponible o no disponible

Además, una operación **notify** puede enviar notificar a los receptores cuando haya un mensaje disponible en una cola determinada.

MODELO DE PROGRAMACIÓN EN COLAS:

A una misma cola podrán acceder múltiples emisores y múltiples receptores, y la gestión de la cola suele usar una política FIFO, aunque también puede ordenarse por prioridad o directamente seleccionar un mensaje.

Contenido de un mensaje



Una cola almacenará los mensajes indefinidamente hasta que sean consumidos y asegura la entrega e integridad del mismo, independientemente del tiempo que pase entre el envío y el recibo. En estos sistemas de colas, también pueden implementarse métodos de modificación de mensajes y mecanismos de seguridad.

JAVA MESSAGING SERVICE (JMS):

JMS es la especificación de Java para la comunicación indirecta. Toma el concepto de colas, usando los temas de los modelos de suscripción para decidir el destino de los mensajes. Una implementación JMS distingue varios roles:

- **Cliente JMS:** Programa que actúa de productor o consumidor.
- **Proveedor JMS:** El sistema en sí que implementa JMS.
- **Mensaje JMS:** Objeto empleado para la comunicación de información.
- **Destino JMS:** Objeto que soporta comunicación indirecta (colas).

TEMA 3.1. - SINCRONIZACIÓN EN SISTEMAS DISTRIBUIDOS

PROBLEMAS DE TIEMPO:

A la hora de distribuir información por una red, no existe un reloj común que sirva como fuente precisa del tiempo global, dando lugar a posibles fallos en operaciones distribuidas, algoritmos de elección, exclusión mutua, retardos...

Una posible solución sería establecer un único reloj para todo Internet, pero esto resulta costoso y poco práctico, por lo que se suele optar por sincronizar el tiempo físico local de una máquina a un tiempo físico de referencia, existiendo una serie de alternativas:

- **Sincronizar todos:** Todos los relojes locales de una red se sincronizan con un reloj de referencia.
- **Sincronizar entre los nodos:** Se estima el retraso en las conexiones, o bien se emplea un valor medio.
- **No sincronizar:** Se puede emplear un tiempo lógico para, simplemente, ordenar los eventos o los nodos de una red en función de los retardos.

TIEMPO FÍSICO:

Los relojes de los ordenadores son de cuarzo, ya que sus cristales emiten vibraciones a una velocidad constante que el sistema emplea para actualizar el tiempo. Aún así, se suele producir un retardo de unos 90ms al día. Este retardo lo reducen a 9ns los relojes atómicos, pero su fabricación es muy cara.

Para la sincronización, el tiempo de referencia que suelen tomar los sistemas es UTC, o Tiempo Universal Coordinado. UTC mide el tiempo empleando un reloj atómico, donde un segundo equivale a 9.192.631.770 transiciones de un átomo de Cesio-133.

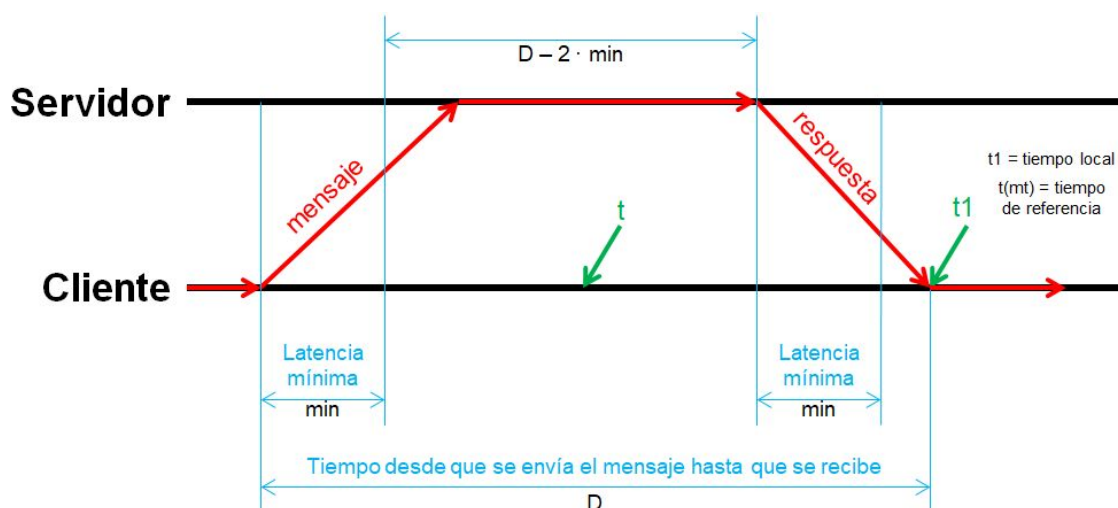
DEFINICIONES:

- **Tiempo físico o de referencia:** Normalmente es UTC.
- **Resolución:** Periodo de tiempo que transcurre entre dos actualizaciones del mismo reloj local.
- **Desviación:** Diferencia entre el tiempo local y el de referencia en un instante. También llamado *offset* o *skew*, se representa con θ .
- **Deriva:** Variación en la desviación en un periodo de tiempo. Es decir, cuánto se adelanta o atrasa un reloj. También llamado *drift*, se representa con δ .
- **Precisión:** Desviación máxima garantizada en la instalación o ajuste de un reloj. También llamado *accuracy*.

SINCRONIZACIÓN:

- **De relojes físicos:** Para sincronizar los relojes de los computadores de un sistema distribuido, suelen emplearse algoritmos de sincronización que aseguren un correcto funcionamiento, aunque este puede mejorarse por GPS.
- **Externa:** Sincronizar un reloj de forma externa es ajustar su valor a un tiempo físico de referencia, junto con la precisión que esta operación conlleve. Las referencias de tiempo UDP se difunden periódicamente por radio y son recibidas por estaciones terrestres o satélites GPS y geoestacionarios.
- **Interna:** En aplicaciones grandes, interesa sincronizar entre sí todos los relojes locales de un sistema. Como implantar receptores para UTC en cada nodo es muy costoso, suelen usarse algoritmos de sincronización interna. Los hay de dos tipos:
 - **Centralizados:** se basan en un servidor específico.
 - **Distribuidos:** toman datos estadísticos.

MÉTODO CRISTIAN:



El método de Cristian emplea la probabilidad a partir de unos datos iniciales para estimar los retrasos en la entrega de mensajes. El cliente recibe la respuesta en el tiempo t_1 , mientras que el servidor conectado a una fuente UTC devuelve $t(\text{mt})$.

- **Desviación:** $|t_1 - D/2 - t(\text{mt})|$ - **Precisión:** $D/2 - \text{min}$

EJEMPLO:

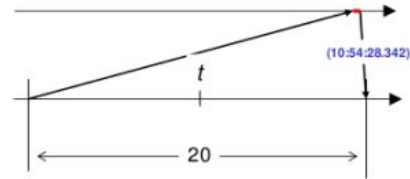
Petición A	$D = 20\text{ms}$	$t_1 = 10:54:26.946$	$t(\text{mt}) = 10:54:28.342$	$\text{min} = 7\text{ms}$
------------	-------------------	----------------------	-------------------------------	---------------------------

Para mayor comodidad, podemos tomar $t_1 = 6949$ y $t(\text{mt}) = 8342$

- **Desviación:** $|6949 - 20/2 - 8342| = 1406\text{ms}$
- **Precisión:** $20/2 - 7 = 3\text{ms}$

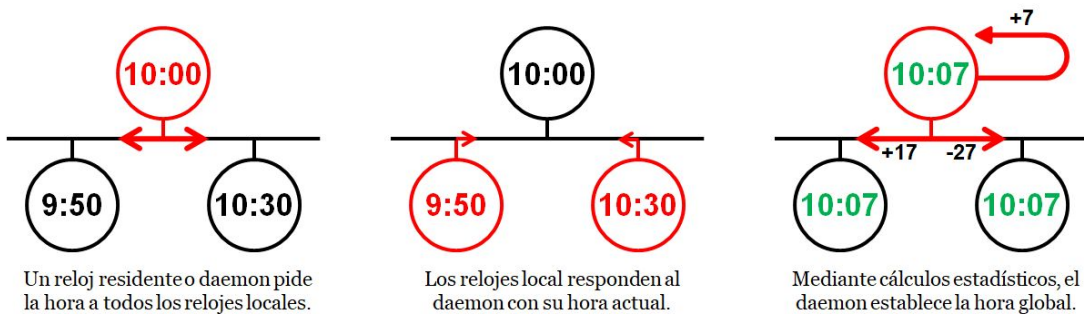
PROBLEMAS DEL MÉTODO CRISTIAN:

El principal problema de este método es que establece una estimación del tiempo para tareas cuya duración depende de muchos factores. Además, no soluciona sobrecargas ni intrusiones por parte de otros servidores, problemas que deben ser tenidos en cuenta aparte.



También, puede darse el caso en el que el reloj de un servidor concreto falle. En ese caso, recurrimos al algoritmo de Berkeley.

ALGORITMO DE BERKELEY:



El algoritmo toma las medias con los tiempos recibidos independientemente del retardo con el que sea recibida. Aún así, si un reloj tiene un retraso bastante superior al resto, puede ser descartado para no alterar la media.

NETWORK TIME PROTOCOL (NTP):

NTP es un estándar de Internet para la sincronización de relojes dentro de cualquier sistema informático tomando el tiempo UDP como referencia a través de otros servidores secundarios. Existen tres modos de operación:

- **Multicast:** para redes locales
- **Llamada a procedimiento:** para mayor precisión
- **Simétrico:** para mejor sincronización interna.

TIEMPO LÓGICO:

El tiempo físico a veces no es suficiente para ordenar los eventos que ocurren en los distintos nodos de un sistema distribuido, algo que siempre debería ser posible si un reloj avanza constantemente en el tiempo. Por tanto, para la ordenación de eventos, hace falta al menos un **reloj lógico**: un contador de software independiente del tiempo físico pero que también avanza de forma constante en el tiempo.

Cada proceso suele ir asociado a un reloj lógico, y estos se centran únicamente en ordenar los eventos y no les interesa tanto saber cuándo ocurren.

MODELO DE EVENTOS:

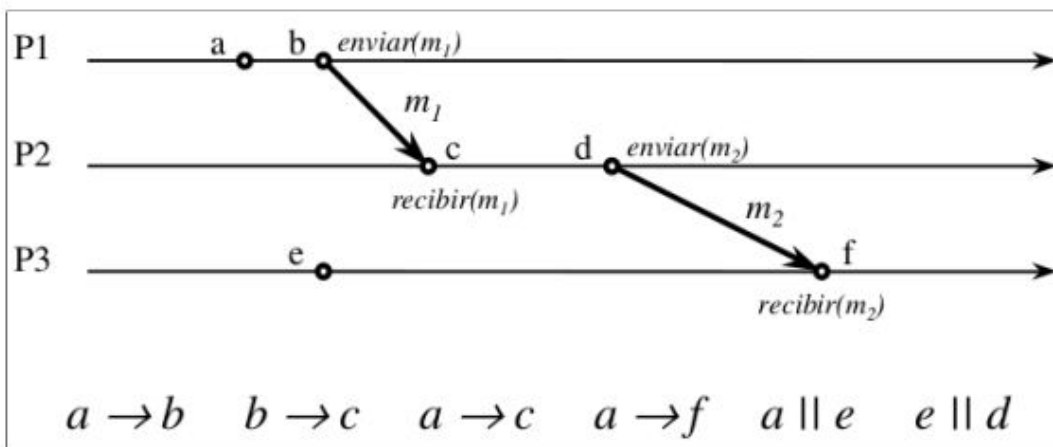
Los procesos sometidos a tiempo lógico se comunican entre ellos mediante paso de mensajes. Para simplificar, se considera que sólo existe un proceso por nodo, cada cual genera una secuencia de eventos que se pueden clasificar en tres tipos:

- **Envío de mensajes:** enviar al ejecutar el evento
- **Recepción de mensajes:** recibir al ejecutar el evento
- **Eventos locales o internos:** otros eventos, la mayoría locales.

Para ordenar estos eventos, si se tiene la resolución suficiente, se asignan marcas temporales locales $T(\text{evento})$ a cada uno de ellos.

Entre dos eventos, que llamaremos “x” e “y”, existe una **relación de causalidad** “ $x \rightarrow y$ ” si “x” ocurre antes que “y”, o lo que es lo mismo, $T(x) < T(y)$. Si “x” e “y” son los métodos **enviar(m)** y **recibir(m)** del mismo mensaje, puede existir un evento “z” tal que “ $x \rightarrow z$ ” y “ $z \rightarrow y$ ”

Si no hay relación de causalidad entre dos eventos, se representa como “ $x \parallel y$ ”.



TIEMPO LÓGICO DE LAMPORT:

Uno de los métodos para calcular el tiempo lógico de un proceso situado en una red de procesos. Según este método, cada proceso P tiene su propio reloj local, al que llamaremos C y que se incrementará en una unidad tras cada evento o envío de mensajes.

Cuando un proceso recibe un mensaje, se actualiza su reloj eligiendo el valor máximo entre su valor actual, y el valor que le envía otro proceso incrementado en una unidad. Es decir:

Si se recibe un mensaje: $C_p = \max(C_p + 1, C(\text{recibido}) + 1)$

Si no: $C_p = C_p + 1$

VECTORES DE TIEMPO:

Se trata de un sistema más potente que el tiempo lógico de Lamport, ya que todos los procesos reciben y envían los valores de los relojes de todos los procesos que conoce. Cada proceso P contiene un vector de tantas posiciones como número de procesos, y en cada posición almacena los valores de los procesos que va recibiendo.

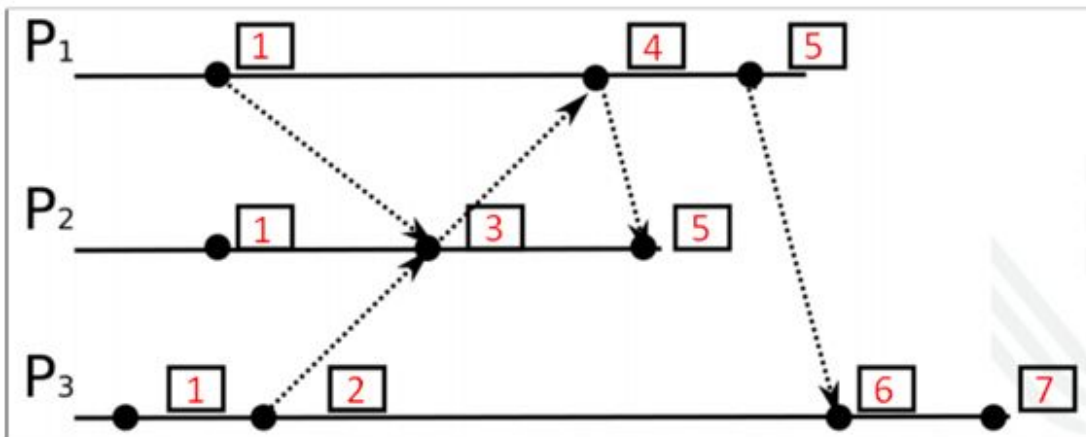
Tras cada evento, el valor de la posición correspondiente al proceso se incrementa en 1, mientras que el resto sigue igual. A la hora de recibir un mensaje de otro proceso, actualiza los valores de su vector con los valores máximos.

Si se recibe un mensaje: $\forall k: V(i)[k] = \max(V(i)[k], V(\text{recibido})[k])$

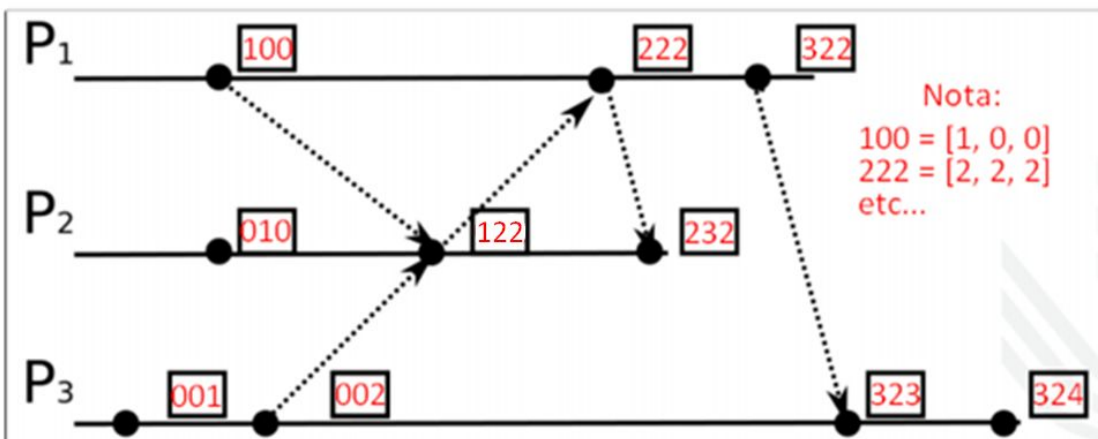
Si no: $V(i)[i] = V(i)[i] + 1$

EJEMPLO LAMPORT Y VECTORES DE TIEMPO:

- **Lamport:**



- **Vectores de tiempo:**



TEMA 3.2. - DEPURACIÓN DISTRIBUIDA

ESTADOS GLOBALES:

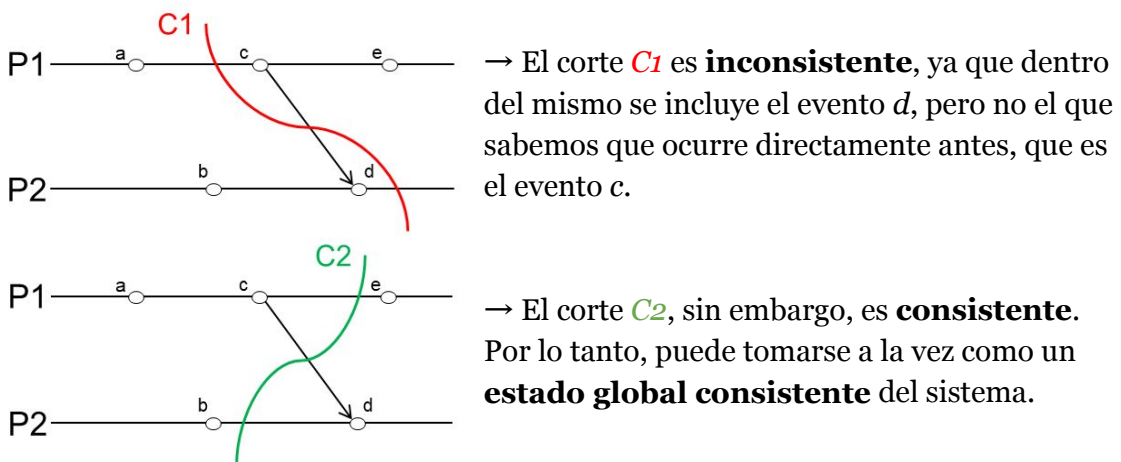
Debido a la imposibilidad de sincronizar los relojes locales en un único tiempo o estado global, resulta muy difícil o imposible especificar en qué estado se encuentra un sistema distribuido en un instante determinado. Aun así, existe una serie de tareas importantes para las que nos interesa conocer el estado global del sistema:

- **Recolección de basura:** Se encarga de liberar las zonas de memoria u objetos ocupados y no utilizados de forma segura. Estas zonas de memoria pueden estar referenciadas en otros procesos o en mensajes aún no recibidos.
- **Detección de interbloqueos:** Dos procesos se quedan, a la vez, esperando un mensaje del otro, por lo que no pueden avanzar.
- **Detección de estados de terminación:** Detecta la terminación de un algoritmo distribuido. Aunque todos los procesos de un algoritmo estén en estado pasivo, hay que comprobar si existe algún mensaje activo entre ellos. Por tanto, se tiene en cuenta el estado del proceso y el canal de comunicación.

CORTES CONSISTENTES:

Puesto que una captura en el tiempo es imposible en un sistema distribuido, el estado de un proceso debe definirse como una sucesión de eventos. Partiendo del modelo de eventos de un determinado sistema, podemos realizar **cortes** en el mismo.

Se dice que un corte es **consistente** cuando, para cada suceso que contiene, también contiene todos los sucesos que ocurrieron anteriormente. Se considera que un suceso está contenido en el corte si está a su izquierda al ser representado en el modelo de eventos. Por ejemplo:

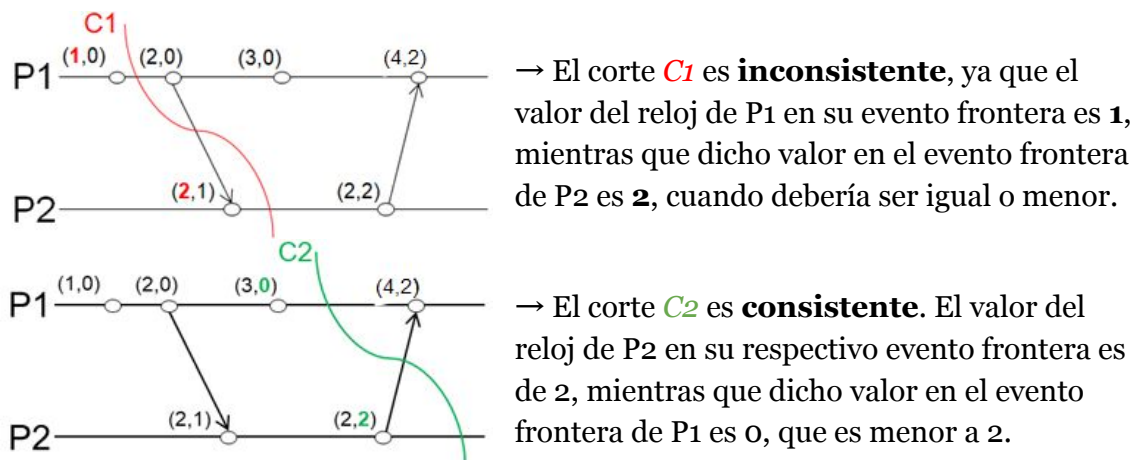


CORTES CONSISTENTES CON RELOJES VECTORIALES:

Para saber si un corte es consistente en un sistema que emplea vectores de tiempo, donde cada proceso tiene una visión parcial del sistema, debe cumplirse la siguiente propiedad con los **eventos frontera** de cada proceso, es decir, los que están más cerca del corte y contenidos en el mismo (a su izquierda):

$$\forall i, j : V_i[i](e_i^{ci}) \geq V_j[i](e_j^{cj})$$

Es decir, el valor en el reloj lógico de un proceso i es mayor o igual que los valores para ese mismo reloj lógico mantenidos en los otros procesos. Un ejemplo:



ALGORITMO DE SNAPSHOT DE CHANDY Y LAMPORT:

Este algoritmo pretende lograr un estado consistente del sistema (corte consistente) incluyendo los canales de comunicación entre los procesos. El estado de un canal entre dos procesos lo constituyen los mensajes enviados pero aún no recibidos. A la hora de realizar un corte, también habrá que anotar los **mensajes en tránsito** de un proceso a otro.

El algoritmo parte de ciertas suposiciones previas:

- Los canales y procesos **no deberían fallar**. El envío de mensajes mantiene las propiedades de integridad y validez.
- Los canales son **unidireccionales** y la entrega es tipo **FIFO**: los primeros mensajes en enviarse serán los primeros en llegar al destino.
- Todo proceso cuenta con canales de comunicación al resto de los procesos.
- Cualquier proceso puede tomar una snapshot en cualquier momento.
- Los procesos **continúan** su ejecución y comunicación de forma paralela mientras se está tomando una instantánea. Esta suposición se mantendrá firme pese a que los procesos son monohilo.

Nota: Snapshot = Instantánea

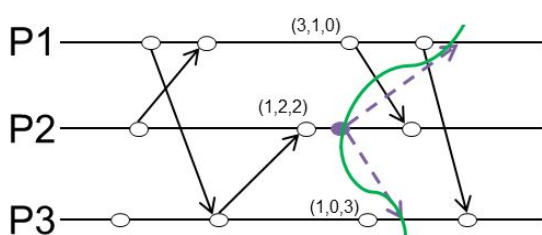
REGLAS DE RECEPCIÓN Y ENVÍO DE SNAPSHOT:

En el momento en el que un proceso registra su estado (toma una snapshot), este **envía un mark**, o mensaje de instantánea, al resto de procesos por sus respectivos canales de comunicación. Cuando un proceso **recibe un mark** por un determinado canal, que llamaremos c , puede ocurrir...

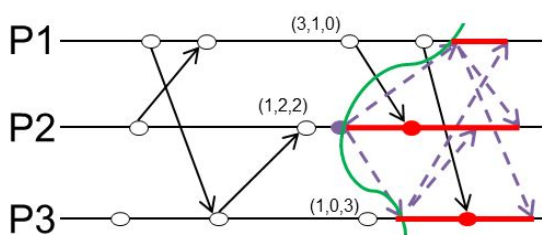
- Si dicho proceso aún **no ha registrado** su estado...
 - registra su **estado**
 - el estado del canal c se registra como **vacío**
 - activa el registro de **mensajes** que lleguen por otros canales
- Si dicho proceso sí ha registrado su estado...
 - el proceso registra el estado del canal c como el **conjunto de mensajes recibidos** tras haber realizado la snapshot.

Al recibir un mark y registrar su estado, dicho proceso volverá a enviar otro mark al resto de procesos que, suponiendo que siguen una entrega tipo FIFO, ya habrán registrado su estado debido al recibo del primer mark.

EJEMPLO DE SNAPSHOT:



→ Supongamos esta red de procesos. En el punto morado, P_2 decide registrar su estado, por lo que envía un **mark** a los otros dos procesos (flechas moradas). Como estos no tienen estado registrado, lo crean, formando un corte consistente.



→ Cuando P_1 y P_3 registran su estado, vuelven a enviar un **mark** al resto de procesos. Ya todos los procesos han registrado su estado, así que registran el estado de los canales de comunicación en función de los **mensajes recibidos** desde que crearon la snapshot (línea roja)

Estado de los canales:

$c_{12} = \{r_{21}\}$	$c_{13} = \{i_{12}\}$	$c_{21} = \{\emptyset\}$	$c_{22} = \{\emptyset\}$	$c_{31} = \{\emptyset\}$	$c_{32} = \{\emptyset\}$
-----------------------	-----------------------	--------------------------	--------------------------	--------------------------	--------------------------

i_{12} = invocación número **2** del proceso **1**

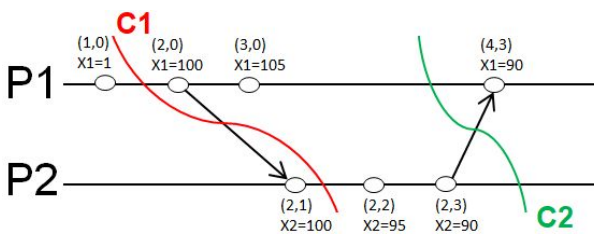
r_{21} = respuesta a la invocación número **1** del proceso **2**.

El estado almacenado por el corte en sí es, por lo tanto, **C = (4,2,3)**, correspondiente a los estados almacenados localmente por cada proceso.

PREDICADOS:

La ejecución de un sistema distribuido puede ser depurada en función de las transiciones entre diferentes estados consistentes, entre las cuales se puede evaluar una determinada condición del sistema, a la que llamaremos **predicado**. Las características más comunes para un predicado son:

- **Estabilidad:** El valor del predicado no varía entre los distintos estados. Útil para detectar interbloqueos o terminaciones.
- **Seguridad:** El valor del predicado es siempre falso para cualquier estado alcanzable. Si fuese verdadero, podemos detectar posibles **errores**. Es el que emplearemos en este tema.
- **Veracidad:** El valor del predicado es verdadero para uno o varios estados alcanzables concreto. Útil para comprobar situaciones necesarias concretas.



Predicado: $\varphi = |x1 - x2| > 50$

Buscamos que el predicado sea falso para comprobar que no hay errores.

En C1: $|x1 - x2| = |1 - 100| = 99 > 50$

En C2: $|x1 - x2| = |105 - 90| = 15 < 50$

MONITORIZACIÓN DE PREDICADOS:

Depurar un sistema distribuido requiere el registro de un estado global, para poder analizarlos y evaluar un determinado predicado en dichos estados. Estos, de forma general, se deberán evaluar teniendo en cuenta estas condiciones:

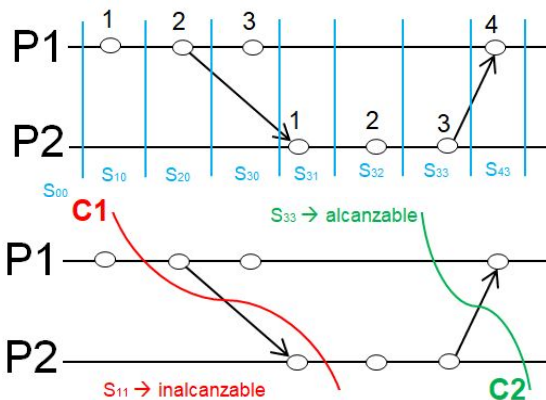
- **Posiblemente** exista un camino donde haya un estado verdadero.
- **Sin ninguna duda** no hay camino donde no haya un estado verdadero.

Para monitorizar estos estados globales, se ha visto el algoritmo de Chandy y Lamport para sistemas distribuidos. Para sistemas centralizados, existe el **algoritmo de Marzullo y Neiger**:

- Existe un proceso **monitor** al que el resto de procesos envían su estado local.
- Cada cierto tiempo, vuelve a recibir **nuevos estados**.
 - El monitor solo registrará los estados consistentes.
- Los mensajes de estado son almacenados por el monitor en **colas** de proceso.
 - Existe una cola de proceso por cada proceso asociado.

Monitorizar estados globales es un proceso **costoso**, tanto por ancho de banda como por almacenamiento. Por ello, puede ser útil tener en cuenta qué predicado vamos a enviar, reduciendo su **tamaño** (no incluyendo partes no importantes del mensaje) o su **número** (limitando los casos en los que se genera y envía un estado).

RED DE ESTADOS GLOBALES:



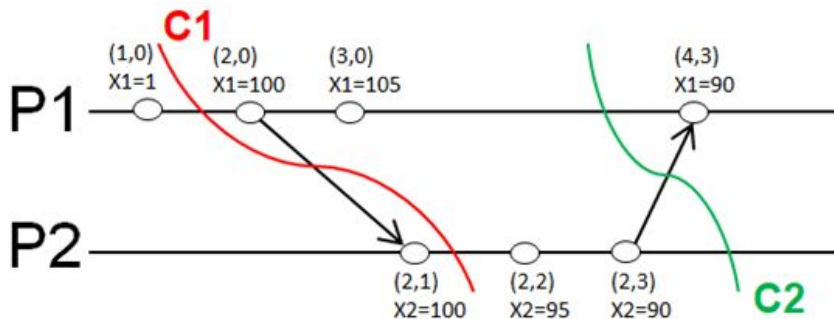
→ Los estados se representan de la forma S_{xy} , siendo (x, y) el valor local de los dos procesos de la red. Si fuesen tres en lugar de dos, los estados se representarían de la forma S_{xyz} .

→ Hay que tener en cuenta que no todos los estados desde S_{00} serán alcanzables. Los asociados a cortes **inconsistentes** no son estados globales alcanzables.

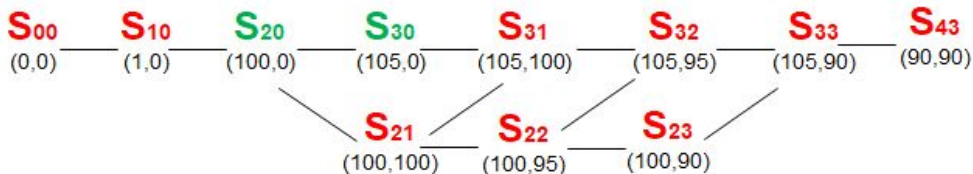
Los estados alcanzables lo serán desde S_{00} , pero para llegar a ellos habrá que pasar obligatoriamente por todos los demás estados alcanzables que existan por el camino.

EVALUACIÓN DE ESTADOS CONSISTENTES:

Evaluar toda una red de estados globales es un proceso complejo, pues es necesario obtener todas las **linealizaciones** o caminos posibles para determinar si estos tienen inconsistencias posiblemente o sin duda alguna:



Predicado: $\varphi = |x1 - x2| > 50$



En la red de caminos, se representan en rojo aquellos donde el predicado es **falso**, y en verde aquellos donde es **verdadero**. En este caso, el camino tendrá estados de error sin ninguna duda, ya que es obligatorio pasar por el estado S_{20} .

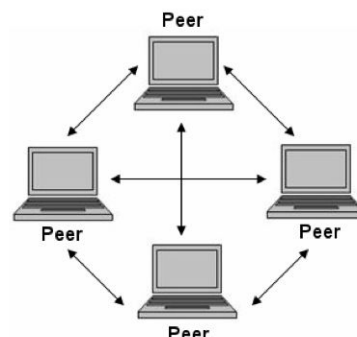
Si arreglásemos la inconsistencia en el estado S_{20} , la red pasaría a tener estados de error posiblemente, ya que solo algunos caminos pasan por el estado S_{30} . Dentro de un camino, la única condición es que *no se puede recorrer la misma línea dos veces*.

TEMA 4 - SISTEMAS PEER-TO-PEER (P2P)

PEER-TO-PEER:

Un sistema *peer-to-peer*, o sistema **P2P**, es un sistema auto-organizado de entidades **iguales** y **autónomas**, llamadas **peers** (según Oram y otros autores, en 2001), entre los que existe un uso compartido de recursos distribuidos.

Un sistema *P2P* no distingue entre clientes y servidores, sino que trata a todas las partes como iguales, evitando así ser un servicio centralizado.



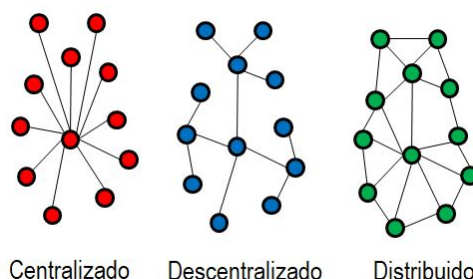
DIFERENCIAS CON UNA RED CLIENTE-SERVIDOR:

En una red cliente-servidor, los clientes solicitan recursos a uno o más servidores, por lo que cada parte tiene un rol definido y no comparten recursos, siendo un sistema casi siempre **centralizado**. Además, según se aumenta el número de usuarios, disminuye la tasa de transferencia (*bitrate*) debido al tráfico intenso.



CARACTERÍSTICAS DE LOS SISTEMAS P2P (I):

- **Descentralización:** Pese a que existen distintas clases de sistemas P2P, la mayoría son **descentralizados** y no distribuidos, ya que los peers suelen ser propietarios de sus propios recursos **locales**. Los nodos descargan recursos de los peers que los aportan.
- **Escalabilidad:** Un sistema P2P soportará más operaciones que un sistema centralizado en el que todos los clientes intentan acceder a los mismos servidores, dando lugar a una mejora seria en la escalabilidad.
- **Auto-organización:** En lugar de tener un nodo central con una visión total de la red, cada peer es el encargado de **distribuir la carga** como considere. Por ello, añadir un nuevo nodo a una red P2P conlleva un esfuerzo mínimo.
- **Coste de la propiedad:** Administrar y mantener todo el hardware de una red cliente-servidor tiene un gran coste para, por ejemplo, una empresa. Como cada peer en un sistema P2P es autónomo, cada cliente lo mantendrá de forma individual y el único coste recae sobre la implementación y el mantenimiento del sistema P2P en sí.



Centralizado Descentralizado Distribuido

CARACTERÍSTICAS DE LOS SISTEMAS P2P (II):

- **Conectividad puntual:** También referida como *ad-hoc connectivity*, se refiere a que los peers **no están conectados entre sí todo el rato**, sino que únicamente se conectan para realizar actividades concretas.
- **Rendimiento:** Existen muchos métodos para mejorar el rendimiento en una red P2P, ya que los peers pueden aportar capacidad de almacenamiento o de procesamiento al sistema:
 - **Replicación:** Los ficheros se copian entre peers por si uno falla.
 - Uso de **memorias caché**
 - **Encaminamiento inteligente** entre pares.
- **Seguridad:** La mayoría de los problemas de seguridad de los sistemas P2P también son problemas para la mayoría de sistemas distribuidos:
 - **Cifrado** multiclave para el acceso a los recursos.
 - Seguridad del **código** de la red.
 - Gestión de la **propiedad intelectual** (copyright) de los recursos.
 - **Reputación** y contabilización, es decir, fiarse más de los iguales que más recursos y de mayor calidad aportan.
- **Transparencia y usabilidad:** La red P2P funcionará igual sin importar ni el **dispositivo** desde el que se acceda (móvil, PC...) ni el **lugar** desde el cual se conecten.
- **Resistencia a fallos:** Se da por hecho que, si falla uno de los peers de una red, el resto seguirá funcionando y comunicándose sin problema alguno.

ATAQUES EN REDES P2P:

Puesto que estas redes se basan en la libre compartición de recursos, hay aspectos donde son más vulnerables que otro tipo de redes. Algunos ataques comunes son:

- **Poisoning:** También llamado *envenenamiento*, consiste en la distribución de archivos sin relación alguna con su descripción o lo que prometen ser.
- **Polluting:** También llamado *contaminación*, se inyectan o se infiltran paquetes malignos entre los paquetes legítimos de un archivo.
- **DoS o choking:** Se provoca un tráfico excesivo dentro de una red P2P de forma deliberada con el fin de *saturarla*.
- **Identificación:** Se *identifican* los usuarios de una red para localizar usuarios que puedan estar causando problemas, como violar los derechos de autor o aportando ficheros no deseados.
- **Scamming:** *Estafas* de cualquier tipo dentro de una red P2P, como solicitar pagos para el uso o distribución de un servicio gratuito, o la falsa autorización para añadir ficheros no solicitados a los recursos compartidos de una red.

ALMACENAMIENTO EN REDES P2P:

Una entidad normalmente organiza sus archivos y ficheros de forma **centralizada**, o directamente son distribuidos entre dispositivos sin contar con un repositorio central. Un sistema P2P creará un **repositorio conectado** a partir de los datos locales de cada peer individual. Este repositorio permite la categorización de los datos según un criterio determinado y, además, proporciona **auto-organización** a la hora de agregar documentos.

Los archivos compartidos en un servidor P2P se guardan en **nodos individuales**, en lugar de en un nodo central. Además, un peer que descargue un recurso **también puede ofrecerlo al resto de peers**, reduciendo considerablemente el tráfico de intercambio de archivos en la red (~70% del tráfico total de Internet). Esto, sin embargo, dificulta la **localización exacta** de los recursos que queremos buscar.

Pese a esto, el almacenamiento de archivos es el atractivo principal para las entidades que deciden implementar una red P2P.

DISTRIBUCIÓN DEL PROCESAMIENTO EN REDES P2P:

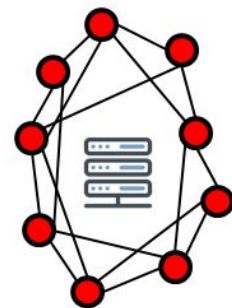
Otra área de aplicación menos popular pero no menos importante de las redes P2P es la distribución de la **capacidad de procesamiento** de cada peer para **aumentar los ciclos de procesador** y, en general, el poder de computación de la red. Existen diversas aplicaciones para esta estrategia:

- Se forma un cluster de peers de tal forma que dicho conjunto forma una única **computadora lógica en red**, transparente para todos los peers.
- Conseguir un poder de computación comparable o incluso superior al de un superordenador.
- **Grid Computing:** Se envía o se distribuye una información entre varios peers con el fin común de procesarla (*ejemplo: [Folding@Home](#), voluntarios ofrecen su capacidad de cómputo para realizar una multitud de cálculos con el fin de hallar sus resultados en algún peer concreto*)

MODELO P2P CENTRALIZADO:

Una de las posibles **arquitecturas** de una red P2P es la arquitectura centralizada. Pese a que todos los peers están conectados entre sí, existe un **servidor** de coordinación central que atiende peticiones de **búsqueda** e indexación. Cuando un peer sabe de qué peers puede obtener un determinado archivo, los solicita. (Ejemplo: *Napster*).

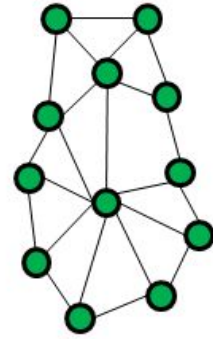
Normalmente, ambos peers que participan en un intercambio notifican al servidor cuando finaliza la transferencia.



MODELO P2P DESCENTRALIZADO O PURO:

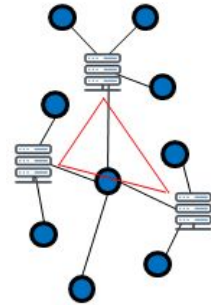
Otra posible arquitectura de una red P2P. En este caso, no tenemos una entidad central, sino que **todos los peers son iguales**, sin ninguna jerarquía.

La búsqueda de un documento pasará **de peer a peer** hasta que se localice uno del que podamos obtenerlo y establecer una conexión con él, limitado hasta cierto nivel de profundidad para no tener que recorrer la red completa, convirtiéndolo en un modelo **poco escalable** (Ejemplo: *Gnutella*).



MODELO P2P HÍBRIDO:

Este modelo introduce una capa **jerárquica y dinámica** compuesta de varios servidores centrales que no almacenan ningún recurso, sino que simplemente ayudan a **encaminar el tráfico** y administrar los recursos, sin conocer la identidad de cada nodo para no perder confiabilidad. Estos servidores pueden comunicarse entre sí para ayudar a localizar los recursos. (Ejemplos: *BitTorrent*, *eMule*...).



ESTRUCTURAS DE LAS REDES P2P:

Según como se enlazan unos nodos a otros, las redes P2P pueden ser:

- **No estructuradas:** Los recursos no se localizan en nodos concretos, sino que cada uno contiene una serie de **enlaces** a otros nodos que va actualizando según se crean, eliminan... La búsqueda en estas redes es **no arbitraria**, es decir, no se asegura que se localice el recurso aunque esté en la red. (*Es el caso de la mayoría de redes P2P que usan las 3 arquitecturas vistas antes*)
- **Estructuradas:** Los recursos están en nodos concretos, y cada nodo contiene una **tabla de hash** propia que lo hace responsable del contenido de una parte específica de la red a la que pertenece.

FUNCIONAMIENTO Y PROBLEMAS DE LAS REDES ESTRUCTURADAS:

Las redes P2P estructuradas **nunca tendrán entidad central** de localización ni archivos almacenados fuera de la red, sino que desde el principio se asigna un conjunto de archivos a cada peer, y se emplea un servicio **DHT (tablas hash distribuidas)** para localizar un archivo cuando sea solicitado con una función ya definida por la red. El problema es que se requiere **conocer el nombre exacto** del archivo para realizar la consulta a la tabla hash y, al conectar o desconectar elementos, se produce un proceso de sincronización en los nodos vecinos que, en casos concretos, podría **colapsar la red P2P**.

TEMA 5 - SISTEMAS DE FICHEROS DISTRIBUIDOS

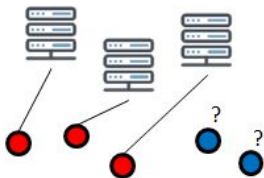
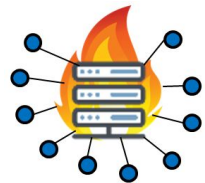
OBJETIVO:

A veces, cuando se trabaja con un gran volumen de archivos, una forma de tenerlos todos almacenados es estableciendo distintas localizaciones distribuidas, de tal forma que no sea necesario conocer dónde se sitúa un archivo concreto exactamente. Estos sistemas de ficheros distribuidos pueden tener diferentes enfoques:

- **No persistentes:** Se pierden cuando se apaga la máquina.
 - Almacenamiento en *memoria compartida*
 - Almacenamiento de *objetos CORBA, EJB...* (pueden comunicarse)
- **Persistentes:** Se mantienen guardados al apagar la máquina
 - Datos *no estructurados*, como ficheros
 - Datos *estructurados*: bases de datos distribuidas o paralelas.

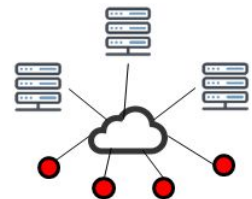
POR QUÉ USAR SISTEMAS DE FICHEROS DISTRIBUIDOS:

- **Situación A:** Se un único servidor para todos los clientes de una red. Sin embargo, esta comienza a tener tanta **actividad**, incluyendo subidas y descargas de archivos, que empiezan a producirse **fallas** como cuellos de botella.



- **Situación B:** El problema anterior se intenta solucionar instalando más servidores, creando así un **sistema distribuido**. Sin embargo, los usuarios comienzan a quejarse de que desconocen en qué servidor se encuentra el archivo que buscan, ya que cada fichero se sitúa en un único servidor concreto.

- **Situación C:** Se establece en nuestro sistema distribuido una nube que es capaz de visualizar y localizar todos los ficheros de cada servidor de la red. Así se establece un **sistema de ficheros distribuidos**.



CARACTERIZACIÓN:

Un sistema de ficheros distribuido pretende emular el funcionamiento de un sistema no distribuido, de tal forma que los clientes puedan acceder a los **recursos del sistema** como si estuviesen en su nodo local:

- **Ficheros:** elementos que se almacenan junto a sus datos (permisos, tipo...)
- **Directorios:** ficheros especiales que contienen otros ficheros (carpetas)
- **Metadatos:** datos ocultos al usuario, pero útiles para la gestión de ficheros.
- **Operaciones:** crear, abrir, cerrar, leer, eliminar, modificar, renombrar...

REQUISITOS DE UN SISTEMA DE FICHEROS DISTRIBUIDO:

- **Transparencia:** Como se mencionó anteriormente, se pretende simular un sistema no distribuido o local pese a que, por detrás, exista una completa arquitectura distribuida. Se debe notar a la vista del usuario como un sistema flexible, pero escalable. En este contexto, existen tipos de transparencia:
 - **De acceso** → el usuario no debe ser consciente de que los ficheros realmente están distribuidos entre varios servidores.
 - **De localización** → el usuario debe ver un único espacio de nombres uniforme para los ficheros (no hay cambios en el pathname)
 - **De movilidad** → el usuario no nota modificaciones en un fichero si este se trasladara o migrara de un servidor a otro.
 - **De rendimiento** → el usuario debe notar el mismo rendimiento al usar el sistema incluso si hay aumentos en la carga del servicio.
 - **De escalabilidad** → el sistema no debe reducir sus capacidades cuando este se aumente, por ejemplo, añadiendo más servidores.
- **Concurrencia:** Los cambios hechos en un fichero por un cliente no debe afectar ni a la vista ni al acceso del fichero por parte del resto de clientes. Por ejemplo, Google Drive siempre muestra la misma vista de un documento a todos los usuarios.
- **Replicación:** Un mismo nombre de fichero puede referirse a varios ficheros replicados en varios servidores., para asegurar el acceso rápido.
 - **Ventajas:** se reparte la carga de trabajo, mayor tolerancia a fallos.
 - **Desventajas:** costoso mantenimiento y actualización de las copias.
- **Heterogeneidad:** El sistema debe ser accesible independientemente del sistema operativo o plataforma empleada por el cliente, proporcionando una interfaz, diseños y API compatibles.
- **Tolerancia a fallos:** recuperación ante fallas de un servidor o un cliente, no debe haber caídas del servidor.
- **Consistencia:** Si se emplea replicación de ficheros, todas las copias de un mismo archivo deben ser idénticas. Los cambios realizados sobre una copia deben propagarse al resto para que no existan inconsistencias, incluso si los cambios se realizan de forma concurrente.
- **Seguridad:** Se deben implementar mecanismos de control de acceso a los ficheros, como autenticación y autorización.
- **Eficiencia:** Ya que se emula el funcionamiento de un sistema de ficheros local o no distribuido, la eficiencia de un sistema de ficheros distribuidos debe ser idéntica o, como mínimo, comparable.

EVOLUCIÓN:

Durante las **primeras generaciones** de sistemas distribuidos, estos sistemas de ficheros no eran más que sistemas de almacenamiento en red. Hoy en día, gracias al avance en sistemas de **objetos distribuidos** (programas en diferentes lenguajes que pueden comunicarse entre sí) y en la web, los sistemas son más complejos y a mucha mayor escala, como el almacenamiento en la nube.

COMPARATIVA DE SISTEMAS DE ALMACENAMIENTO:

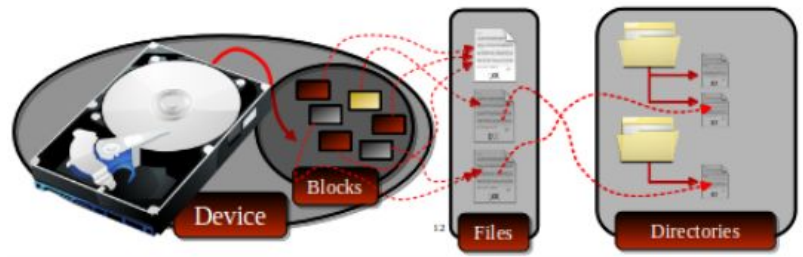
	Compartición	Persistencia	Replicación	Consistencia
Memoria principal				Sólo una copia
Sistema de ficheros				Sólo una copia
Ficheros distribuidos				
Web				Se hace manualmente
Memoria distribuida				Sólo una copia
Objetos remotos				Sólo una copia
P2P				Poco fiable

ROJO = No VERDE = Sí AMARILLO = Con limitaciones

MÓDULOS DE UN SISTEMA NO DISTRIBUIDO:

Los sistemas de ficheros distribuidos imitarán, en la medida de lo posible, el funcionamiento de los sistemas no distribuidos, como los locales:

MÓDULO	FUNCIONAMIENTO
<i>Dispositivo</i>	Realiza operaciones de lectura y/o escritura en buffer
<i>Bloque</i>	Accede a los bloques (=servidores) del disco duro (=nube)
<i>Acceso a fichero</i>	Escribe o lee datos de los ficheros
<i>Control de acceso</i>	Comprueba los permisos para operar sobre ficheros
<i>Fichero</i>	Asocia identificadores con sus correspondientes archivos
<i>Directorio</i>	Asocia rutas y archivos a los identificadores definidos.



ESTRUCTURA DE LOS FICHEROS:

Los sistemas de ficheros, distribuidos o no, se encargan de almacenar, recuperar, proteger o compartir los archivos, los cuales tienen la siguiente estructura:

MANTENIDA POR...	NOMBRE DE LA CAPA
<i>El sistema</i>	Longitud del archivo
	Hora (<i>timestamp</i>) de creación
	Hora de lectura
	Hora de escritura
	Conteo de referencias (enlaces)
<i>El usuario</i>	Dueño
	Tipo de archivo
	Control de acceso (<i>rw-rw-r--</i>)

EJEMPLOS DE SISTEMAS DE FICHEROS DISTRIBUIDOS:

- **Arquitectura de servicio de ficheros:** Modelo genérico basado en la división de responsabilidades entre cliente y servidor. Se basan NFS y AFS.
- **Network File System (NFS):** Desarrollado por Sun Microsystems en 1984. Cliente y servidor tienen una relación simétrica (multiplataforma).
- **Andrew File System (AFS):** Se centra en la transferencia de grandes bloques de datos y el uso de cachés.

ARQUITECTURA DE SERVICIO DE FICHEROS:

Recomienda estructurar el servicio en tres componentes esenciales, centrados en el manejo y acceso de los ficheros distribuidos:

- **Servicio de Ficheros Planos:** Implementa las operaciones de lectura y escritura para los ficheros individuales, a los cuales se les asigna un identificador único (UFID) para representarlos.
- **Servicios de directorio:** Mapea o asocia rutas o nombres de archivos a identificadores únicos. Además, gestiona los directorios y sus archivos.
- **Módulo cliente:** Ofrece estas operaciones de acceso a los ficheros mediante una API o una interfaz.



PROBLEMAS DE LA ARQUITECTURA DE SERVICIO DE FICHEROS:

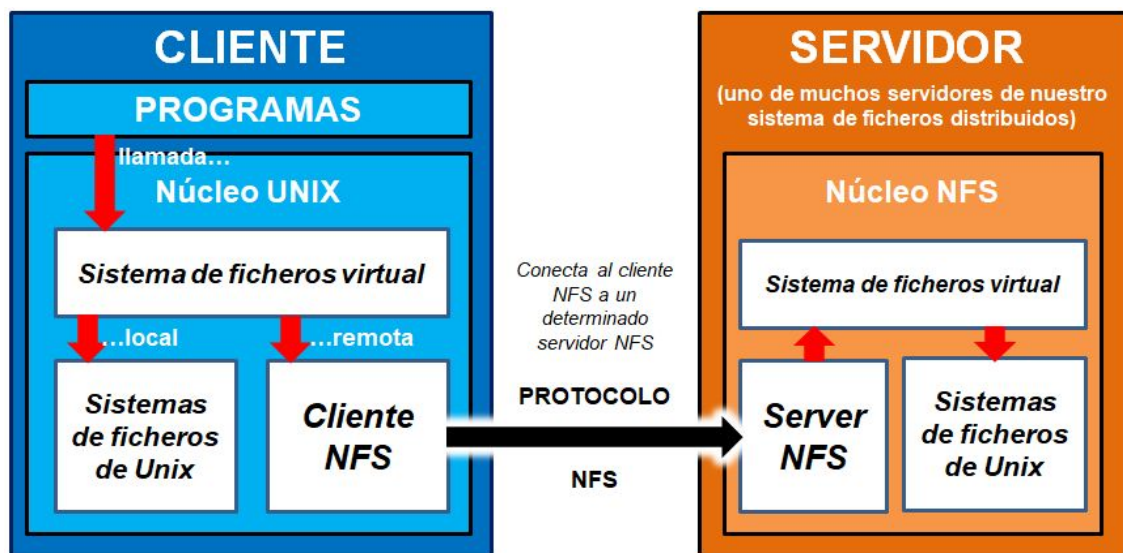
Los principales problemas de este modelo de sistemas recaen sobre la distribución de la carga de trabajo entre cliente y servidor, sobre todo en lo referente al acceso a los datos y resoluciones de nombre de fichero. Además, no asegura una correcta migración y replicación transparente de los datos.

NETWORK FILE SYSTEM (NFS):

Estándar basado en *UNIX* desarrollado por Sun Microsystems en 1985, muy popular y de código abierto. Soporta una **amplia mayoría** de los requisitos mencionados para los sistemas de ficheros distribuidos, aunque existen algunas **excepciones con limitaciones** como son la concurrencia, replicación, consistencia o seguridad.

Emplea su propio **protocolo**, también llamado NFS, que aunque no ofrece concurrencia, se añade con el uso de otro protocolo, NLM (*Network Lock Manager*).

ARQUITECTURA NFS:



- **Protocolo NFS:** Conjunto de llamadas a procedimiento remoto para realizar operaciones sobre un almacenamiento de ficheros remoto.
- **Clientes y servidores NFS:** Se comunican por el protocolo NFS y a través de operaciones síncronas, por TCP, UDP o protocolos por debajo.

La única limitación es que el protocolo exige que las peticiones estén debidamente formuladas y que el usuario esté autenticado y autorizado adecuadamente.

Por lo general, NFS se trata de un sistema de ficheros distribuidos **robusto y de alto rendimiento**, pero no por ello complejo, lo que explica su extendido uso en sistemas de todo el mundo.

CARACTERÍSTICAS DE NFS:

<i>Acceso</i>	Excelente. API integrado en la interfaz de llamadas
<i>Localización</i>	No se asegura, ya que depende en gran parte del cliente
<i>Concurrencia</i>	Limitada, sobre todo para ficheros compartidos
<i>Réplica</i>	Sólo para ficheros de lectura, usa otro protocolo para escritura.
<i>T. a fallos</i>	El servicio se suspende si un servidor falla. Limitado pero eficaz.
<i>Movilidad</i>	Apenas conseguido, se debe hacer casi todo de forma manual.
<i>Rendimiento</i>	Los servidores son multiprocesador, rendimiento alto.
<i>Escalabilidad</i>	Es fácil aumentar o subdividir los servidores o procesadores.

ROJO = No

VERDE = Sí

AMARILLO = Con limitaciones

OTROS SISTEMAS DE FICHEROS DISTRIBUIDOS:

- **Google File System (GFS):** Sistema propietario desarrollado por Google, empleado para sus propias aplicaciones en la nube como Google Drive. Se enfoca sobre todo al desarrollo de aplicaciones y sistemas.
- **Hadoop Distributed File System (HDFS):** Sistema abierto implementado para Hadoop, una famosa arquitectura de computación distribuida, con objetivos parecidos a GFS.

PARTE 2: SEMINARIOS

SEMINARIO 1 - ADMINISTRACIÓN DE SISTEMAS OPERATIVOS E INTRODUCCIÓN A PYTHON

INSTALACIÓN Y PARTICIONES DE LINUX:

Una **partición** es una división en la unidad física de almacenamiento de datos. A la hora de instalar un sistema Linux, se requieren como mínimo estas dos particiones:

- **Root:** Es la raíz del sistema. Todos los directorios de nuestro sistema parten de aquí. Se representa con “/”.
- **Swap:** Área de intercambio. Aquí se almacenarán áreas de código inactivas en procesos grandes. Es una especie de RAM virtual, pero algo más lenta.

Además, se recomienda crear también estas dos particiones:

- **Boot:** Partición de arranque, contiene los archivos utilizados al arrancar el sistema, además del núcleo.
- **Home:** Contiene la información personal de cada usuario. Se recomienda guardarla en una partición por si hay que reinstalar Linux.

DIRECTORIOS IMPORTANTES DEL SISTEMA DE FICHEROS:

- **/bin:** Contiene los ejecutables básicos del sistema, que involucran comandos importantes como cp, mv, mkdir... en /sbin se encuentran los ejecutables exclusivos para el superusuario.
- **/dev:** A partir de este directorio se pueden acceder a las diferentes particiones del disco duro y a los dispositivos conectados.
 - **Disco primario:** /dev/hda, **particiones** /dev/hda1, /dev/hda2...
 - **Disco secundario:** /dev/hdb, **particiones** /dev/hdb1, /dev/hdb2...
- **/etc:** Reservado para ficheros de configuración de los programas instalados y scripts de inicio del sistema, como /etc/passwd.

COMANDOS BÁSICOS DE LINUX:

ls	Lista contenido de directorio	pwd	Muestra dónde estamos
cp	Copiar un archivo	kill	Finalizar un proceso
mv	Mover un archivo	ps	Muestra info de procesos
rm	Eliminar un archivo	mkdir	Crear un directorio
rmdir	Eliminar un directorio	touch	Crear un archivo vacío

EJEMPLOS PRÁCTICOS:

- **Ver en qué partición está la swap:** `sudo fdisk -l /dev/sda`
- **Detener el programa Apache2:** `sudo /etc/init.d/apache2 stop`

USUARIOS Y GRUPOS:

Unix es un sistema **multiusuario**, lo que significa que muchos usuarios pueden usarlo de forma simultánea, de tal forma que los archivos de cada usuario sean privados y el resto no pueda verlos. Para ello, se establece un nombre de usuario y contraseña por usuario, que se almacena de forma cifrada. Encontramos estos tipos de cuenta:

- **Administrador:** se identifica con root, y cuenta con todos los permisos y accesos, así que debe emplearse para funciones de administración.
- **Usuario:** cuenta normal y personal de un usuario concreto.
- **Especiales de los servicios:** por ejemplo, de noticias a tiempo real.

FICHEROS DE INFORMACIÓN DE USUARIOS:

- **/etc/passwd:** Información sobre usuarios
- **/etc/group:** Información sobre grupos
- **/etc/shadow:** Contiene las contraseñas de usuario cifradas
- **/etc/gshadow:** Contiene las contraseñas de grupo cifradas (desusado)

En cada fichero, se almacena la información de cada usuario en una línea, con los datos o campos separados por un carácter especial, generalmente “:”.

COMANDOS DE GESTIÓN DE USUARIOS:

- **useradd:** Añade un nuevo usuario al sistema
- **userdel:** Borra la cuenta de un usuario del sistema
- **usermod:** Modifica las opciones de un usuario
- **groupadd:** Añade un nuevo grupo al sistema
- **groupdel:** Borra un grupo del sistema

INTRODUCCIÓN A PYTHON:

Python es un lenguaje de **muy alto nivel** nacido en 1991 que prioriza la legibilidad del código para el programador, gracias a su sintaxis sencilla y el uso de **indexación** en lugar de corchetes y puntos y coma. Se trata de un lenguaje **multiplataforma** y **multiparadigma** que soporta tanto programación orientada a objetos como estructural. Además, destaca por su **tipado dinámico**, de tal forma que no hay que asignarle un tipo a las variables creadas. El IDE recomendado que se utilizará para los seminarios es **PyCharm**.

IF ELSE:

```
1 num = 4
2
3 if num % 3 == 0:
4     print("El número da resto 0 al dividirlo entre 3")
5 elif num % 3 == 1:
6     print("El número da resto 1 al dividirlo entre 3")
7 else:
8     print("El número da resto 2 al dividirlo entre 3")
```

Imprime: El número da resto 1 al dividirlo entre 3

LISTAS:

```
1 a = list()           #Forma 1 de definirla
2 b = []               #Forma 2 de definirla
3 c = [1, 2, 3, 4]     #Forma 3 de definirla
4 print(c[0], c[2], c[-1])
```

Imprime: 1 3 4

```
1 a = [1, 2, 3, 4]
2 a.append(5)          # Añade el valor 5
3 print(a[0:5:2])      # Imprime de c[0] a c[4] de 2 en 2
```

Imprime: 1, 3, 5

DICCIONARIOS:

```
1 dct = {"Silla": "Chair", "Mesa": "Table"}
2 dct["Libro"] = "Book"
3 print(dct.values())
4 print(dct.keys())
```

Imprime: dict_values(['Chair', 'Table', 'Book'])
dict_keys(['Silla', 'Mesa', 'Libro'])

ITERAR:

```
1 lista = [1, 1, 2, 3, 5, 8, 13, 21]
2 dct = {"Silla": "Chair", "Mesa": "Table", "Libro": "Book"}
3
4 for i in lista:
5     print(i)
6
7 for j in dct.keys():
8     print(str(j) + " - " + str(dct[j]))
```

Imprime: 1 1 2 3 5 8 13 21
Silla - Chair Mesa - Table Libro - Book

FUNCIONES:

```
1 def suma(a, b):
2     return a+b
3
4 def suma_resta(a, b):
5     return suma(a, b), a-b
6
7 a = 5
8 b = 3
9 sum_, res_ = suma_resta(a, b)
10 print(sum_, res_)
```

Imprime: 8 2

SEMINARIO 2 - SOCKETS EN PYTHON

SOCKETS:

Los sockets son una **API** para la comunicación entre procesos y la base para la construcción de sistemas distribuidos complejos. Prácticamente todo Internet y todos los sistemas operativos los usan, haciéndolo estándar de facto. Cada socket se asocia a un puerto, haciendo que otros procesos puedan comunicarse con él.

LA API SOCKET:

Cuando se crea un socket, este va asociado a una dirección IP y a un puerto que no sea inferior al 1024, ya que estos están reservados. Se pueden usar los protocolos TCP o UDP:

- **TCP: Socket Stream** - Los paquetes llegan ordenados y se asegura que son recibidos, pero al usar más ancho de banda suele ser más lento.
- **UDP: Socket Datagram** - Al no estar orientado a conexión, los mensajes llegan desordenados o se pierden, pero llegan más rápido y con menor coste.

CREACIÓN SOCKET EN PYTHON:

```
import socket
sock = socket.socket(<familia>, <tipo>)
```

La familia del socket puede ser:

- **AF_UNIX:** Familia de direcciones IP de Unix, útil para memoria compartida
- **AF_INET:** Familia de direcciones de Internet (IPv4)

El tipo del socket puede ser:

- **SOCK_STREAM:** Para TCP
- **SOCK_DGRAM:** Para UDP. Puede implementar multicast.

SOCKETS UDP:

Partiendo del socket creado, los comandos para enviar y recibir mensajes con un socket UDP (SOCK_DGRAM) son los siguientes:

```
/*ENVIAR: */ socket.sendto(mensaje, (HOST [IP], PUERTO))  
/*RECIBIR: */ mensaje = socket.recvfrom(tamaño del buffer)
```

Al enviar el mensaje, si no hay un puerto a la escucha, este puede perderse. Además, el receive es bloqueante (síncrono). Queda parado hasta que recibe algo. Buffer size es el tamaño máximo en bytes del mensaje recibido.

SERVIDOR UDP:

#Se ejecuta primero

```
import socket
```

```
HOST = 'localhost'
```

```
PORT = 5000
```

```
s_udp = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

```
s_udp.bind(HOST, PORT)
```

```
print("Esperando mensaje")
```

```
mensaje = s_udp.recvfrom(1024)
```

```
print("Mensaje recibido: " + str(mensaje))
```

```
s_udp.close()
```

CLIENTE UDP:

#Se ejecuta después

```
import socket
```

```
HOST = 'localhost'
```

```
PORT = 5000
```

```
s_udp = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

```
s_udp.sendto("Soy el cliente", (HOST, PORT))
```

```
s_udp.close()
```

SOCKETS TCP:

Partiendo del socket creado, los comandos para conectar, enviar y recibir mensajes con un socket TCP (SOCK_STREAM) son los siguientes:

```
/*CONECTAR: */ socket.connect(HOST, PORT) #conectar a servidor
```

```
/*ESCUCHAR: */ socket.listen(mensaje) #aceptar clientes
```

```
/*ACEPTAR: */ socketCliente, addr = s_tcp.accept()
```

```
/*ENVIAR: */ socket.send(mensaje)
```

```
/*RECIBIR: */ mensaje = socket_cliente.recv(tamaño del buffer)
```

SERVIDOR TCP:

#Se ejecuta primero

```
import socket
HOST = 'localhost'
PORT = 5000

s_tcp = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s_tcp.bind(HOST, PORT)
s_tcp.listen(1)
print("Esperando mensaje")
socketCliente, addr = s_tcp.accept()      #nuevo socket que
                                           atiende al cliente

mensaje = socketCliente.recv(1024)
print("Mensaje recibido: " + mensaje + " de " + str(addr))
s_cliente.send("Hola, soy el servidor")
socketCliente.close()
s_udp.close()
```

CLIENTE TCP:

#Se ejecuta después

```
import socket
HOST = 'localhost'
PORT = 5000

s_tcp = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s_tcp.connect(HOST, PORT)
s_tcp.send("Hola, soy el cliente")
mensaje = s_tcp.recv(1024)
print("Mensaje recibido: " + mensaje + " del servidor")
s_tcp.close()
```

CODIFICACIÓN Y DECODIFICACIÓN A BYTES:

En algunas versiones de Python como Python3, es necesario codificar los mensajes a bytes y luego decodificarlos para el correcto funcionamiento del programa, sin importar el tipo del socket. Se emplean los siguientes comandos:

```
/*ENVIAR: */ socket.send(bytes(mensaje, "utf-8"))
/*RECIBIR: */ print(mensaje.decode("utf-8"))
```

UTF-8 es un formato de codificación capaz de representar cualquier caracter Unicode como cadena de 1 a 4 bytes.

SEMINARIO 3 - SERVICIOS WEB Y REST API EN PYTHON

TIPOS DE LENGUAJES:

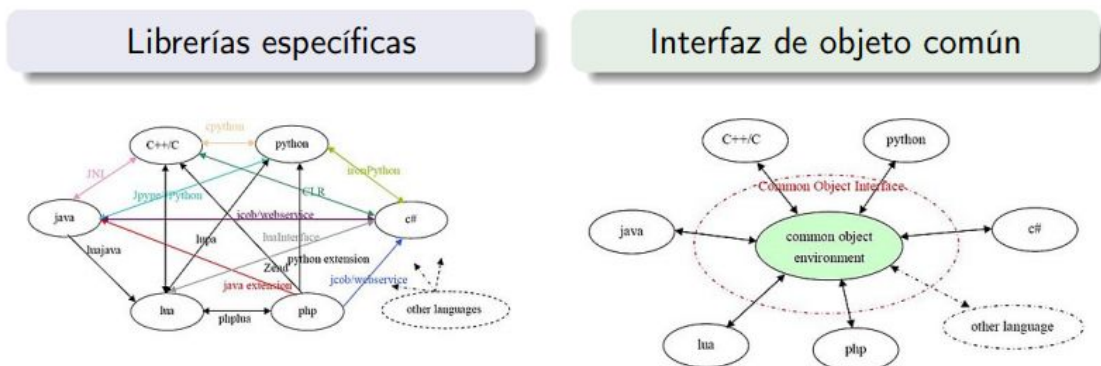
Dentro de un sistema distribuido existen diversas partes y componentes, pudiendo cada una estar implementada en lenguajes de programación o frameworks distintos. Algunos tipos de lenguaje son:

- De bajo, medio y alto nivel
- Paralelismo y concurrencia
- Multiplataforma como Java
- Para plataformas web como PHP o JavaScript
- A tiempo real como Ada
- Para cálculo científico como Scala

PROGRAMACIÓN MULTILENGUAJE:

Puesto que cada lenguaje tiene bases y objetivos diferentes, es común que dentro de un sistema existan componentes escritos en una variedad de lenguajes. Si queremos usar el mismo lenguaje para todo, este debe ser multiplataforma y contener librerías para todo lo que queramos hacer.

Por ello, existen diversos métodos para comunicar componentes escritos en distintos lenguajes, como pueden ser **librerías específicas para cada lenguaje** (poco escalable y difícil de mantener) o **interfaces de objeto común** (más sencillo).



SERVICIOS WEB:

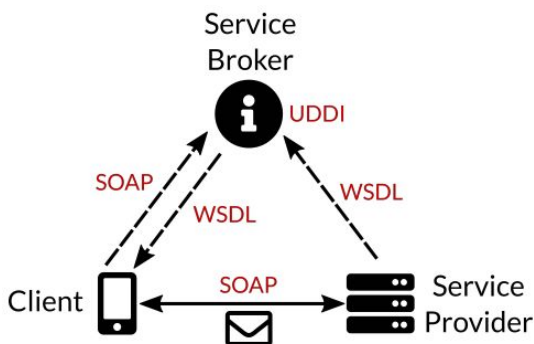
Un servicio web es un sistema para la comunicación de datos entre aplicaciones mediante entornos distribuidos en una red, por lo que resulta ideal para la programación multilinguaje. Estos servicios web están estandarizados por W3C y ofrecen un enfoque para la interoperación de diferentes aplicaciones sobre diferentes plataformas, lenguajes o frameworks.



TIPOS DE SERVICIOS WEB:

- **SOAP:** Basado en XML, ofrece formatos para el intercambio de datos a través de procedimientos y ejecutables remotos
- **REST:** Surgió para solucionar los problemas de flexibilidad de SOAP, dando opción a formatos basados en HTML, JSON, texto plano... además de basarse en la arquitectura web y el protocolo HTML.

INTERACCIÓN SOAP:



En primer lugar, el proveedor del servicio web lo debe registrar en un broker mediante WSDL.

Tras ello, el broker lo registra a través del estándar UDDI.

Un cliente que desee acceder a un servicio primero debe consultar al broker mediante SOAP. El broker devuelve el **contrato** del servicio mediante WSDL.

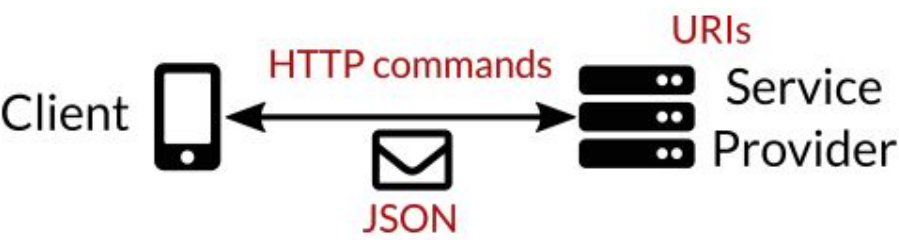
A partir de aquí, cliente y proveedor ya pueden intercambiar mensajes por SOAP. Se trata de un estándar robusto, pero tedioso de implementar y trabajar ya que se basa únicamente en XML y no permite una comunicación directa entre cliente y proveedor sin la intervención del broker.

- **Contrato:** Especificación de los métodos y operaciones, entre otros elementos, del servicio con el que el cliente quiere intercambiar mensajes. Un contrato no se puede modificar una vez publicado (la implementación sí).

ESTÁNDARES EMPLEADOS EN SOAP:

- **UDDI:** *Universal Description Discovery and Integration*. Registro público de Servicios Web (como unas Páginas Amarillas).
- **WSDL:** *Web Services Description Language*. Describe la interfaz pública de los servicios web (operaciones, métodos, formatos...)
- **SOAP:** *Simple Object Access Protocol*. Establece protocolos de comunicación de mensajes entre cliente y servicio para el envío y recibo de mensajes. Un mensaje por SOAP contiene:
 - **Envelope:** identifica al mensaje y es obligatorio.
 - **Header:** permite extender un mensaje con más atributos y es opcional.
 - **Body:** contiene el mensaje en sí, es obligatorio.
 - **Fault:** contiene información de estado y es opcional.

INTERACCIÓN REST:



Simplemente conociendo la **URI**, podemos establecer conexión directa con el servicio Web a través de comandos HTTP, JSON, etc. Necesitamos conocer de antemano las especificaciones del servicio (lo que sería el contrato en SOAP).

ESTÁNDARES EMPLEADOS EN REST:

- **URL:** identifica un recurso por su ubicación en Internet.
- **URN:** identifica un recurso electrónico por su nombre.
- **URI:** engloba URL y URN en un mismo concepto.
- **HTTP:** ofrece operaciones para el intercambio de mensajes:
 - **GET:** solicita un recurso en una URL.
 - **HEAD:** solicita el encabezado de un recurso en una URL.
 - **POST:** envía datos a un programa o recurso concreto en una URL.
 - **PUT:** envía datos a una URL.
 - **DELETE:** borra un recurso ubicado en una URL.

POST suele usarse para modificaciones en recursos ya existentes o conectados, mientras que PUT se usa para operaciones más importantes como registros.

- **JSON:** *JavaScript Object Notation*. Otro formato de intercambio de mensajes independiente al lenguaje de programación.

SOAP VS REST:

	SOAP	REST
Diseño	Estandarizado	Pautas y recomendaciones flexibles
Seguridad	Soporte SSL	HTTPS y SSL
Rendimiento	Requiere más recursos	Requiere menos recursos
Mensajes	XML	Texto plano, HTML, XML, JSON, YAML, y otros
Protocolos de Transferencia	HTTP, SMTP, UDP, y otros	HTTP
Recomendado para	Aplicaciones de alta seguridad, servicios financieros, pasarelas de pago	APIs públicas, servicios móviles, redes sociales
Ventajas	Seguridad y extensibilidad	Flexibilidad, escalabilidad y rendimiento

CLIENTE REST EN PYTHON:

En Python, un cliente REST utilizará operaciones HTTP, incluidas en la librería requests, que lo hace sencillo de implementar:

```
import requests

URL = "http://maps.googleapis.com/maps/api/geocode/json"
location = "delhi technological university"
PARAMS = {'address': location}
r = requests.get(url = URL, params = PARAMS)
data = r.json()

latitude = data['results'][0]['geometry']['location']['lat']
longitude = data['results'][0]['geometry']['location']['lng']
formatted_address = data['results'][0]['formatted_address']

print("Latitude:%s\nLongitude:%s\nFormatted Address:%s"
      %(latitude, longitude, formatted_address))
```

Este código representa una operación **GET** empleando la API de Google Maps. El objetivo es obtener los datos geográficos de la Universidad Tecnológica de Delhi, para lo cual hacemos uso de los diferentes métodos de la API. La consulta, en este caso, requiere una clave **address** con el valor del lugar que queremos localizar. La recepción se realiza mediante **JSON**, almacenandose en la variable data.

```
import requests

API_ENDPOINT = "http://pastebin.com/api/api_post.php"
API_KEY = "XXXXXXXXXXXXXXXXXXXX"

source_code = '''
print("Hello, world!")
a = 1
b = 2
print(a + b)
'''

data = {'api_dev_key': API_KEY,
        'api_option': 'paste',
        'api_paste_code': source_code,
        'api_paste_format': 'python'}
r = requests.post(url = API_ENDPOINT, data = data)

pastebin_url = r.text
print("The pastebin URL is:%s"%pastebin_url)
```

Este código representa una operación **POST** empleando la API de pastebin.com, sitio para subir y compartir trozos de texto plano o código. En concreto, queremos crear un “pastebin” cuyo contenido sea *source_code* e imprimir su URL. De nuevo, nos guiamos por las especificaciones definidas en la API de Pastebin.

FRAMEWORKS DE DESARROLLO WEB:

Un servidor REST es mucho más complejo de programar que un cliente, ya que es necesario controlar el paralelismo, salidas, particiones, etc. No basta con crear sockets. Para simplificar el trabajo, se emplean **frameworks de desarrollo web**. Algunos de los más conocidos son:

- **Pyramid:** Flexible, rápido y minimalista. Ideal para webs grandes.
- **Django:** Mayor framework actual para Python con una comunidad activa.
- **Flask:** Emplea un único archivo, con varias extensiones disponibles.
- **Bottle:** Framework simple que proporciona las herramientas mínimas.

El que usaremos en la asignatura será Bottle, el más recomendado para principiantes. Se instala escribiendo el comando `pip install bottle` en una terminal.

HELLO WORLD EN BOTTLE:

```
from bottle import Bottle, run

app = Bottle()

@route('/hello/<name>')
def index(name):
    return template('<b>Hello {{name}}</b>!', name=name)

run(app, host='localhost', port=8080)
```

Con `Bottle()` inicializamos el framework, con `@route` establecemos la URL de acceso al servicio y, justo debajo, definimos las funciones pertinentes. Con `run()`, se inicia la ejecución del programa en un determinado host y puerto.

ROUTE:

El comando **route** permite asociar una función a una o varias rutas o URL. Los parámetros que toman las funciones se definen entre `<>`, pudiendo indicar los tipos de parámetro con **:int**, **:float**... si fuese necesario. También soporta el uso de expresiones regulares con **:re** (por ejemplo, `:re:[a-z]+` para almacenar cualquier cadena cuyos caracteres sean letras de la “a” a la “z” minúsculas).

```
@route('/')
@route('/hello/<name>')
def greet(name='Stranger'):
    return template('Hello {{name}}, how are you?', name=name)

@route('/object/<id:int>')
def callback(id):
    assert isinstance(id, int)

@route('/show/<name:re:[a-z]+>')
def callback(name):
    assert name.isalpha()
```

EJEMPLOS DE PETICIONES CON BOTTLE:

```
from bottle import get, request # o route

@get('/cars') # o @route('/cars')
def getcars():
    cars = [ {'name': 'Audi', 'price': 52642},
              {'name': 'Mercedes', 'price': 57127},
              {'name': 'Skoda', 'price': 9000},
              {'name': 'Volkswagen', 'price': 21600} ]

    return dict(data=cars)
```

Usando **@get** o **@route** (@route, por defecto, es un @get), se procesa una petición GET que devuelve el diccionario de coches especificado

```
from bottle import post, request # o route

@post('/login') # o @route('/login', method='POST')
def do_login():
    try:
        data = request.json()
    except:
        raise ValueError
    if data is None:
        raise ValueError

    username = data['username']
    password = data['password']
    if check_login(username, password):
        return "<p>Your login information was correct.</p>"
    else:
        return "<p>Login failed.</p>"
```

Usando **@post**, se recibe un JSON por parte del cliente que, en caso de recibirse correctamente, incluirá un usuario y contraseña que serán comprobadas por nuestro programa. El resultado de la comprobación es lo que se devuelve al cliente.

```
from bottle import put, request # o route

# o @route('/names/<oldname>', method='PUT')
@put('/names/<oldname>')
def update_handler(name):
    try:
        data = json.load(utf8reader(request.body))
    except:
        raise ValueError

    newname = data['name']
    _names.remove(oldname)
    _names.add(newname)

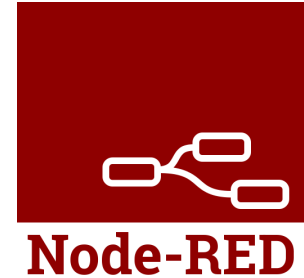
    # return 200 Success
    response.headers['Content-Type'] = 'application/json'
    return json.dumps({'name': newname})
```

Usando **@put**, se recibe un JSON por parte del cliente que, en caso de recibirse correctamente, incluirá un nuevo nombre de usuario de un cliente que desea cambiarlo. En este caso, debe eliminarse el nombre antiguo y enviar confirmación.

SEMINARIO 4 - NODE-RED

INTRODUCCIÓN A NODE-RED:

Node-RED es un entorno de desarrollo para desarrollar aplicaciones JavaScript de una forma intuitiva y sencilla, cuyo objetivo es simplificar lo máximo posible el desarrollo de aplicaciones orientadas al manejo de eventos asíncronos. Fue diseñado por IBM y es de código abierto, pero sin ser software libre. La versión 1.0 se publicó en 2019. Se basa en Node.js.



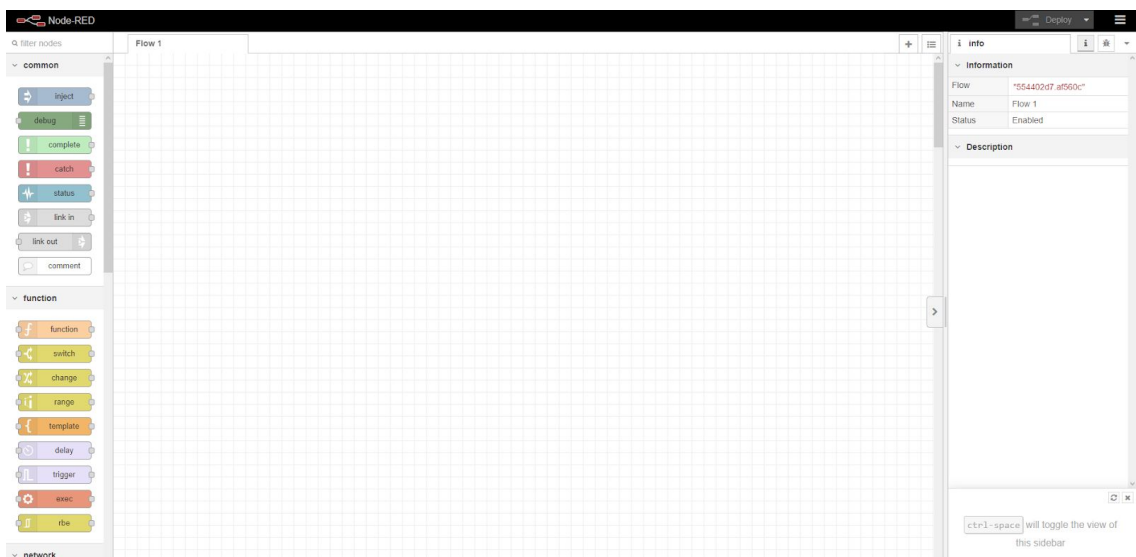
INSTALACIÓN Y EJECUCIÓN:

- **En Windows** (desde cmd):
 - Descargar e instalar [Node.js](https://nodejs.org/)
 - `node --version && npm --version` para comprobar instalación
 - `npm install -g --unsafe-perm node-red`
- **En Linux** (desde terminal):
 - `sudo apt-get update`
 - `sudo apt-get install npm`
 - `sudo npm install -g --unsafe-perm node-red`

Nota: La opción -g añade el comando node-red al PATH.

Una vez finalice la instalación, ejecutamos el comando `node-red` y copiamos en un navegador la dirección local que nos proporciona. Por ejemplo, `127.0.0.1:1880`.

INTERFAZ DE NODE-RED:




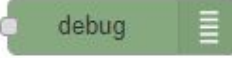

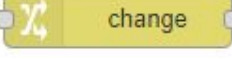
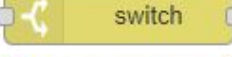
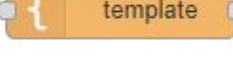
Nodos

Espacio de trabajo

Información

NODOS Y TIPOS DE NODO:

En Node-RED, un nodo es la unidad básica de construcción. Se trata de **bloques** de software, con entradas y salidas, capaces de procesar mensajes. Un nodo puede o no tener una o más entradas y/o salidas. Los tipos de nodo más básicos son:

	Nodo de salida que genera mensajes sin necesidad de entrada.
	Nodo de entrada que muestra mensajes de depuración.
	Ejecuta código JavaScript sobre los mensajes que pasan por él.
	Modifica propiedades de los mensajes sin recurrir a funciones.
	Enruta un mensaje a diferentes salidas según una condición
	Genera textos personalizados por propiedades de un mensaje.

Este último nodo, *template*, emplea el **lenguaje de plantillas** [Mustache](#).

FLUJO:

Un **flow** o flujo es una colección de nodos conectados entre sí que componen una aplicación o programa. La parte izquierda de un nodo siempre será para entradas, y la parte derecha para salidas, de tal forma que el flujo completo se leerá de izquierda a derecha. Cada flujo puede tener un nombre y una descripción.

Los nodos se conectan entre sí mediante **wires**, o cables.

MENSAJES:

Un mensaje en Node-RED es la información que se pasa entre los nodos de un flujo (por sus wires), que no son más que **JavaScript Objects** (objetos escritos siguiendo el formato JSON) con un conjunto de propiedades o **payload**. Una propiedad puede ser cualquier tipo JavaScript válido, como Boolean, Number, String, Array...


En el editor de Node-RED, los mensajes se representan como **msg**.

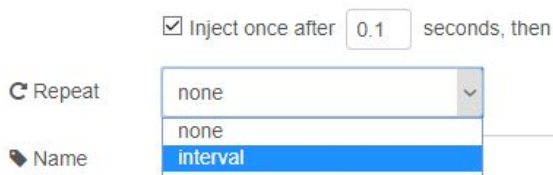
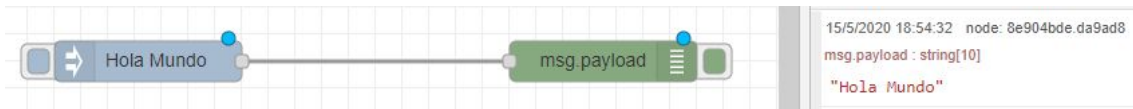
CONTEXTO:

Los nodos de un flujo o de todo el programa pueden compartir el mismo **contexto** de flujo, es decir, variables o información que se almacenan en memoria (se eliminan al cerrar Node-RED), sin tener que usar mensajes. El alcance puede ser:

- **Node:** Visible al nodo que almacena el valor.
- **Flow:** Visible a todos los nodos de un flujo.
- **Global:** Visible a todos los nodos.

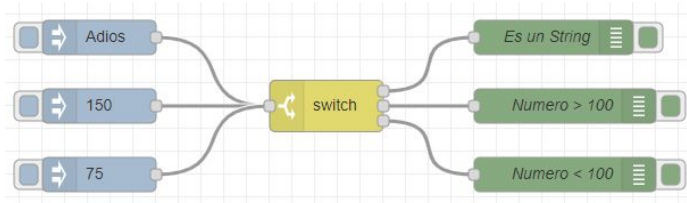
HOLA MUNDO:

1. Creamos un nodo inject, cambiamos su payload a string.
2. Escribimos “hola mundo” o la cadena que queramos enviar.
3. Creamos un nodo debug, y creamos un wire desde el nodo inject
4. Hacemos clic en Deploy, y luego en el icono del menú de depuración: 
5. Pulsamos el botón a la izquierda del nodo inject.

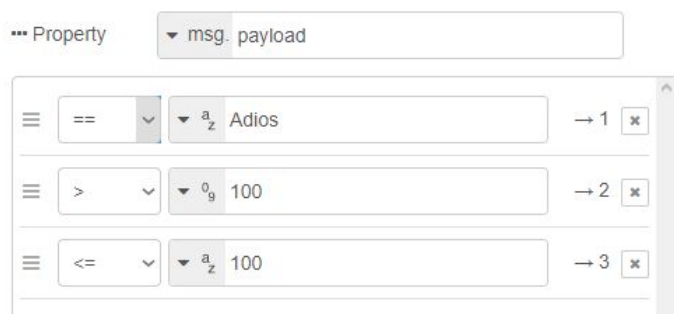



Podemos hacer que el mensaje se inyecte tras una cierta cantidad de segundos, o que se repita cada x segundos.

EJEMPLO CON SWITCH:



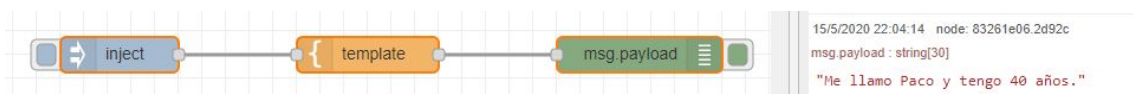
→ Podemos usar un nodo Switch para determinar las salidas de un conjunto de entradas de esta manera. El nodo Switch se configura así:



Añadiremos más condiciones de salida haciendo click en 

EJEMPLO CON TEMPLATE:

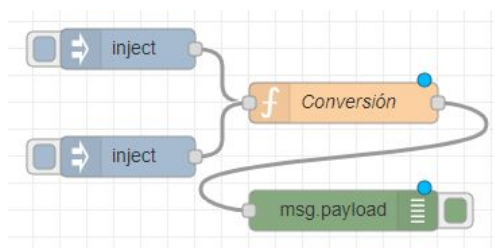
Usando un mensaje inicial de tipo JSON, podemos imprimir mensajes personalizados en función de las claves y valores empleados gracias a template:



- **JSON:** `{"nombre":"Paco","edad":40}`
- **Template:** `Me llamo {{payload.nombre}} y tengo {{payload.edad}} años.`
- **Imprime:** `Me llamo Paco y tengo 40 años.`

EJEMPLO CON FUNCTION:

Con el nodo **Function**, podemos modificar el contenido de un mensaje. Se emplea el lenguaje **JavaScript**, aunque generalmente suelen ser códigos muy simples. En este ejemplo, crearemos un programa simple que nos convierta una cantidad de grados de **Celsius a Fahrenheit**, y viceversa.



Se parte de los siguientes payloads en formato JSON:

- `{"tipo": "celsius", "valor": 30}`
- `{"tipo": "fahrenheit", "valor": 86}`

El código del nodo Function y su resultado para el primer mensaje serán:

Function

```
1 if(msg.payload.tipo=="celsius"){
2   msg.payload.cambio =
3   {"tipo":"fahrenheit",
4    "valor":msg.payload.valor*(9/5)+32}
5 }
6 else{
7   msg.payload.cambio =
8   {"tipo":"celsius",
9    "valor":(msg.payload.valor-32)*5/9}
10 }
11
12 return msg;
```

15/5/2020 22:49:54 node: 1b73294d.539e37

msg.payload : Object

▼ object

tipo: "celsius"

valor: 30

▼ cambio: object

tipo: "fahrenheit"

valor: 86

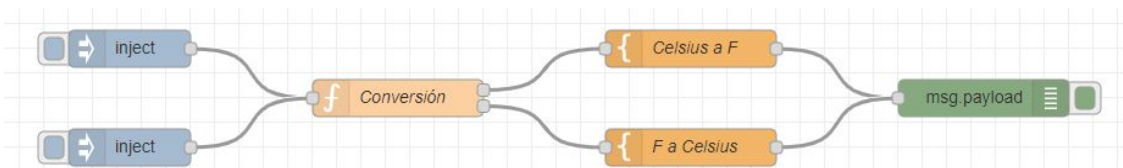
DIFERENTES SALIDAS EN FUNCTION:

Un nodo Function permite, al igual que un Switch, diversas salidas dependiendo de una condición. Esto nos puede ser útil para mejorar el flujo anterior ya que el nodo Template, necesario para mejorar el mensaje, no puede evaluar condiciones.

```
1 if(msg.payload.tipo=="celsius"){
2   msg.payload.cambio =
3   {"tipo":"fahrenheit",
4    "valor":msg.payload.valor*(9/5)+32}
5   return [msg, null];
6 }
7 else{
8   msg.payload.cambio =
9   {"tipo":"celsius",
10    "valor":(msg.payload.valor-32)*5/9}
11   return [null, msg];
12 }
```

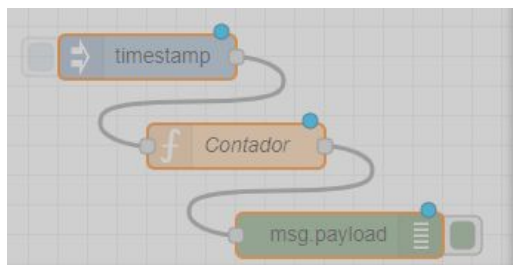
→ Nuestro código Javascript retornará valores por una salida u otra mediante un **vector con tantas posiciones como salidas haya**. En este caso, tenemos dos salidas.

Nuestro nuevo flujo mejorado queda así:



CONTEXTO:

Supongamos que queremos incluir una función en un flujo que cuente cuántos mensajes pasan por ella. Una variable local no nos sirve, ya que esta se reiniciará por cada mensaje que llegue. Por ello, tenemos que asignarle un contexto de esta forma:



Function

```
1 var c = context.get('contador') || 0
2 i++;
3 msg.payload = "Mensajes: " + i;
4 context.set('contador', i);
5
6 return msg;
```

El método **context.get()** obtiene el valor guardado de una variable de contexto. Con “|| o”, le decimos que si dicha variable no existe, la cree inicializada a 0. El contexto se guardará con **context.set()**.

Usando **context**, la variable será global únicamente para ese nodo función. Si la queremos hacer global para todo el flujo, usaremos **flow.get()** y **flow.set()**, con una sintaxis igual a la de los métodos de context. De la misma forma, si queremos que la variable sea global a todos los flujos, se usará **global.get()** y **global.set()**.

Nodo	Flujo	Todo
context	flow	global

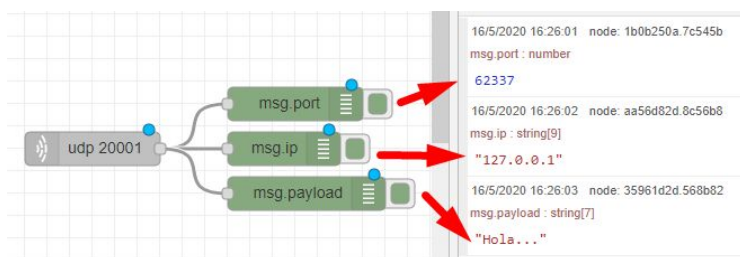
CLIENTE UDP:

```
import socket

serverAddressPort = ("127.0.0.1", 20001)
bufferSize = 1024
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

s.sendto("Hola...", serverAddressPort)
msg, addressfrom = s.recvfrom(bufferSize)
print("Server: " + msg)
```

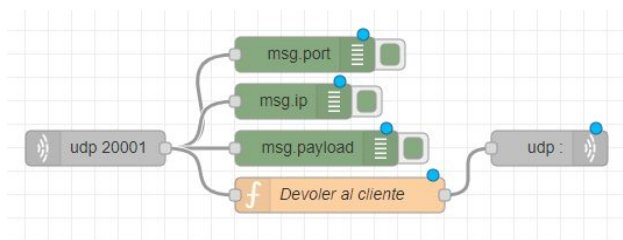
→ Con un nodo UDP podemos crear un socket UDP de entrada. Este se quedará escuchando en un determinado puerto las peticiones de un **cliente UDP** creado aparte (en nuestro caso, usando Python).



→ La operación **sendto** enviará los mensajes por un puerto libre aleatorio. Si queremos obtener dicho puerto y su IP, usamos los comandos **msg.port** y **msg.ip**.

A continuación, configuramos el servidor que responderá la petición del cliente.

SERVIDOR UDP:



→ Para poder devolverle al cliente la respuesta que espera, simplemente añadimos un nodo Function y un nodo de salida UDP. En este último no hará falta configurar nada, pues ya emplea automáticamente los valores msg.ip y msg.port.

Contenido nodo función:

```
msg.payload = "...mundo"; return msg;
```

OPERACIÓN GET A API EXTERNA:

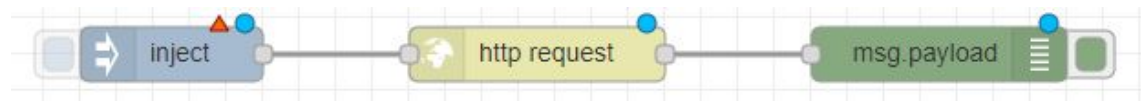
Para este ejemplo, vamos a utilizar la API de [OpenWeather](#). Debemos crearnos una cuenta y, posteriormente, dirigirnos a *API* → *API Keys*. Guardamos el valor que aparece, puesto que es nuestra **clave personal** para realizar consultas a la API.

Key	Name
333d866d5f2f10c907d0e9b0296d9ff5	Default

Si consultamos de nuevo la [API](#) de OpenWeather, vemos que las llamadas son así:

```
api.openweathermap.org/data/2.5/weather?q={city name}&appid={your api key}
```

Donde *{city name}* es la ciudad que queremos consultar y, además, hay que sustituir *{your api key}* por el valor previamente guardado. Ahora, creamos el siguiente flujo:



```
1 {  
2   "city": "Cádiz",  
3   "key": "333d866d5f2f10c907d0e9b0296d9ff5"  
4 }
```

→ El nodo inject será de tipo **JSON**, con este contenido (como mínimo).

URL
☒ Append msg.payload as query string parameters

weather?q={{payload.city}}&appid={{payload.key}}

→ En el nodo **HTTP** request, cambiamos los valores de la consulta a la API.

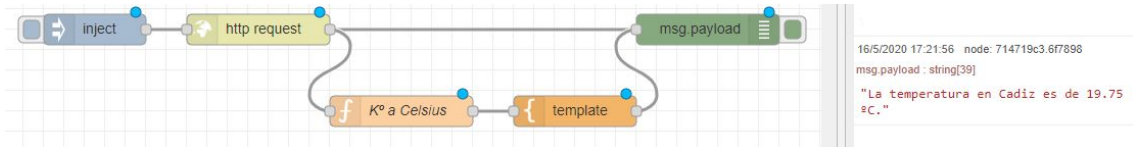
Además, en el nodo HTTP, se puede configurar el tipo de retorno del mismo para que, directamente, nos lo imprima en formato JSON:



Nota: La API de OpenWeather también nos explicará qué significa cada dato obtenido.

MODIFICACIÓN DE MENSAJES DE LA API:

Explorando la API de OpenWeather, nos daremos cuenta de que la temperatura, por defecto, se nos imprime en grados Kelvin. A través de un nodo Function y un nodo Template, podemos realizar e imprimir su conversión a Celsius:



Contenido de function:

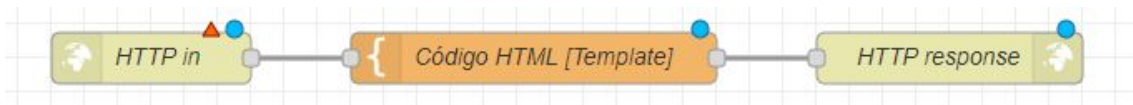
```
msg.payload.main.temp = msg.payload.main.temp - 273.15;  
return msg;
```

Contenido de template:

La temperatura en `{{payload.name}}` es de `{{payload.main.temp}}` °C.

CREAR ENDPOINT GET CON RESPUESTA HTML:

Usando HTML, podemos crear una página web simple que responda a un usuario que realice una petición GET. Para ello, hacemos uso de los nodos *HTTP in* y *HTTP response*, creando un esquema así:



- En HTTP in, definimos el **endpoint**. Por ejemplo, `/hola/:nombre`, donde los dos puntos (:) denotan los parámetros.
- En el nodo template, escribimos el código HTML de nuestra página.



→ Para acceder a la página, añadimos a la dirección y puerto en la que usamos Node-RED el endpoint `/hola/:nombre`, y sustituimos *nombre* por lo que queramos. Por ejemplo, `http://127.0.0.1:1880/hola/Roberto`

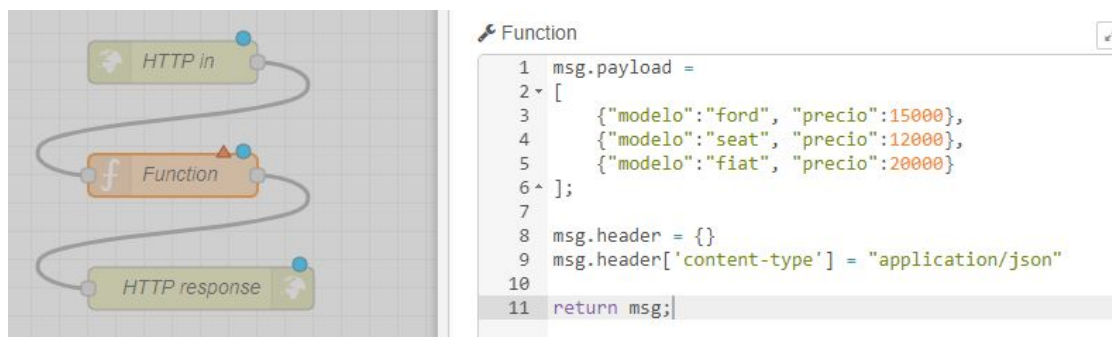
CREAR ENDPOINT GET CON RESPUESTA JSON:

Podemos devolver un JSON, al igual que la API de OpenWeather, en lugar de una web HTML, de la siguiente manera, usando un nodo *Change*:



SIMPLIFICACIÓN CON FUNCTION:

El flujo anterior puede simplificarse reemplazando los nodos Template y Change por un único nodo **Function**, que realizará las tareas de ambos nodos a la vez. En este ejemplo, creamos un endpoint `/cars` que devolverá los datos de varios coches:



COMANDO CURL PARA PETICIONES GET:

El comando **curl**, disponible tanto para la terminal de Linux como para la línea de comandos de Windows, permite realizar peticiones GET a una URL concreta de una forma muy sencilla, a través del siguiente comando:

```
curl -i <URL> (ejemplo: curl -i http://127.0.0.1:1880/hola/Roberto)
```

CREAR ENDPOINT POST CON RESPUESTA JSON:



Vamos a volver a imprimir un mensaje que salude al usuario, pero esta vez el nombre será pasado como **parámetro** en una petición **POST**, en lugar de formar parte de la URL como en GET.

Una vez formado el flujo, introducimos un comando como este en la terminal:

```
curl -X POST -d '{"name":"Roberto"}' -H "Content-type: application/json" http://127.0.0.1:1880/hello
```

Contenido de function:

```
var name = msg.payload.name;
msg = ";Hola, " + name;
return msg;
```

PLUG-INS ADICIONALES:

Podemos añadir más nodos a Node-RED para realizar tareas concretas como, por ejemplo, crear un bot para Telegram, haciendo click en las **tres barras** (a la derecha del botón Deploy) → Manage palette → Install y buscando el plug-in que queramos instalar. En el caso del bot de Telegram, tendremos que buscar *telegambot* en la barra de búsqueda y seleccionar el primer resultado.

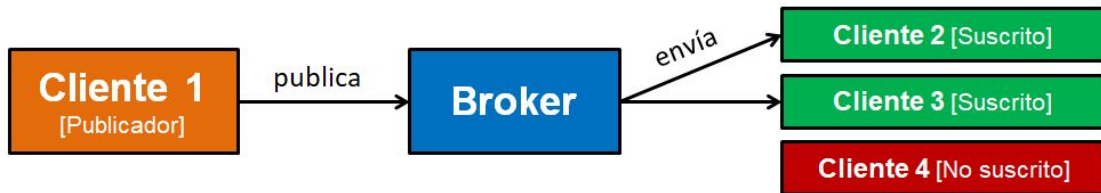
SEMINARIO 5 - PROTOCOLO MQTT

SISTEMAS PUBLICADOR-SUSCRIPTOR:

Sistemas donde los *publicadores* envían notificaciones sobre una serie de eventos a un servidor, que son sólo recibidos por las personas suscritas a dicho evento (los *suscriptores*). El objetivo es relacionar suscripciones con eventos para, así, asegurar la correcta entrega de las notificaciones. Algunos ejemplos incluyen:

- Sistemas de información financiera
- Feeds de noticias RSS
- Apoyo en un sistema de trabajo colaborativo
- Sistemas de computación ubicua
- Monitorización de redes

EJEMPLO DE IMPLEMENTACIÓN CENTRALIZADA:



En un modelo centralizado, el más sencillo de implementar, podemos establecer un **broker** que recibirá los mensajes de los publicadores y las suscripciones de los suscriptores para que, cuando un publicador publique un mensaje, pueda enviarlo a los clientes suscritos a dicho tema.

La interacción con el broker es mediante una serie de mensajes punto a punto (P2P) implementados por paso de mensajes o invocación remota, lo que lo hace propenso a **cuellos de botella** para redes grandes si fallase el *broker* (poco escalable)

NOTA: Se produce un cuello de botella si llegan más peticiones de las que el broker puede administrar

INTRODUCCIÓN A MQTT:

El protocolo publicador-suscriptor [MQTT](https://mqtt.org) (*Message Queuing Telemetry Transport*) facilita el envío de mensajes entre dispositivos. Se adapta con mayor facilidad a sistemas que consumen pocos recursos y suele operar en el protocolo TCP/IP.



MQTT tiene a su **broker** MQTT como elemento central. Este se encarga de recibir los mensajes de publicadores y suscriptores, filtrarlos, enviar los mensajes de los publicadores a los suscriptores correspondientes... además, es responsable de la autenticación y autorización de los clientes.

EJEMPLO MQTT:

Esquema de una aplicación para domótica (*Internet of Things*)



MOSQUITTO Y PAHO:

Mosquitto es una implementación de un **broker MQTT** creada por la fundación Eclipse. Mosquitto fue diseñado como elemento ligero para su uso en dispositivos de cualquier capacidad. Se puede [descargar](#) por separado, aunque distribuciones como Linux o Debian ya lo tienen en sus repositorios.



Paho, a su vez, es una implementación de un **cliente MQTT** también creada por la fundación Eclipse, especialmente pensada para el *Internet of Things* (IoT). Permite definir a los publicadores y a los suscriptores.



NOTA: Internet of Things (IoT) se refiere a la interconexión a través de Internet de objetos que utilizamos en nuestro día a día. Pueden variar desde portátiles o teléfonos móviles hasta incluso bombillas inteligentes o frigoríficos.

EJEMPLO PRÁCTICO - INICIAR MOSQUITTO:

Una vez instalado Mosquitto, bien desde la página o desde terminal, lo inicializamos en una terminal **con permisos de administrador**:

- **En Linux:** `sudo mosquitto`
- **En Windows (cmd):** `net start mosquitto`

Una vez hecho esto, no tenemos que volver a tocar Mosquitto. Ya está programado para tratar las solicitudes de los clientes, bien sean suscripciones o publicaciones de mensajes.

CÓDIGO PUBLICADOR EN PYTHON:

```
1 import paho.mqtt.client as mqtt
2
3 client = mqtt.Client("Publicador")
4 client.connect('localhost')
5 topic = "universidad/andalucia/cadiz"
6
7 client.publish(topic, "La UCA ya está abierta")
```

El publicador publica un mensaje sobre un determinado tema, o topic.

CÓDIGO SUScriptor EN PYTHON:

```
1 import paho.mqtt.client as mqtt
2 import time
3
4 # Decodifica el mensaje y lo muestra en pantalla como string, junto a su tema
5 def mostrar_mensaje(client, userdata, message):
6     print("Evento recibido: " + format(str(message.payload.decode('utf-8'))))
7     print("Tema: " + format(message.topic))
8
9
10 # Define el cliente, lo conecta, y llama a mostrar_mensaje si el broker le envía uno
11 client = mqtt.Client("Suscriptor")
12 client.on_message = mostrar_mensaje
13 client.connect("localhost")
14
15 # Se suscribe al tema y espera a recibir mensajes
16 client.loop_start()
17
18 client.subscribe("universidad/andalucia/uca")
19 time.sleep(60)
20
21 client.loop_stop()
```

El suscriptor se suscribe al mismo tema al que pertenece el mensaje del publicador que hemos definido antes. Primero se ejecutará el suscriptor, y luego el publicador, siempre con Mosquitto en segundo plano.

SEMINARIO 6 - ARQUITECTURAS ORIENTADAS A SERVICIOS Y EL LENGUAJE WS-BPEL

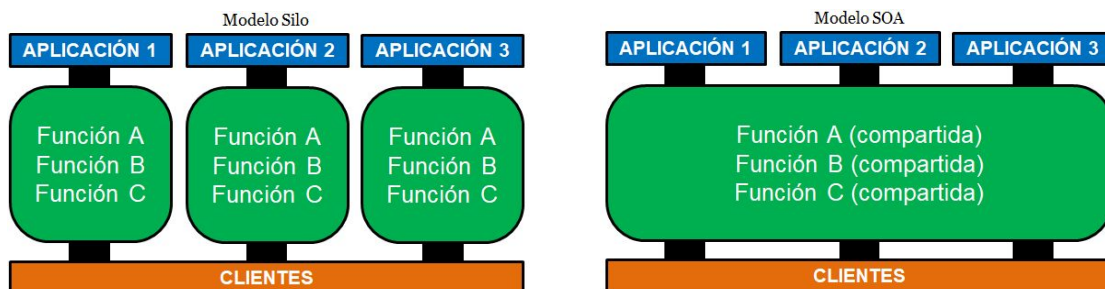
ARQUITECTURAS ORIENTADAS A SERVICIOS:

Dentro del contexto de las *Service-Oriented Architectures*, o **SOA**, un **servicio** es un contrato hacia el usuario que le ofrece una serie de **prestaciones** concretas bajo una o varias condiciones. Cada servicio tiene una funcionalidad concreta que describe al cliente qué puede hacer y cómo se usa el servicio.

Por otro lado, una Arquitectura Orientada a Servicios es una arquitectura software que define un **modelo desacoplado** de los servicios que nos proporciona un servidor. Los servicios desacoplados pueden juntarse o reutilizarse para crear la **lógica de negocio** que se quiere ofrecer a los clientes. Muy empleado en **servicios web** como tiendas online, sobre todo.

En resumidas cuentas, normalmente tendríamos varias aplicaciones, cada una con sus propios servicios aunque estos estén **duplicados**, pero utilizando SOA, podemos desacoplar estos servicios de tal forma que las diferentes aplicaciones puedan **compartir las funcionalidades comunes** para sus propósitos concretos.

MODELO SILO VS MODELO SOA:



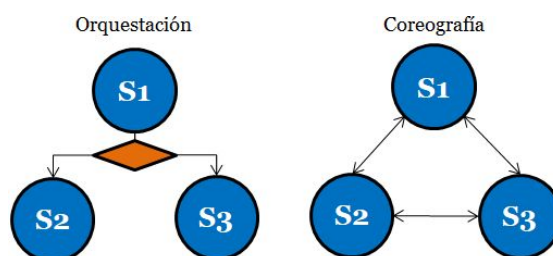
Así, el usuario puede centrarse en los servicios o funciones que se ofrecen, en lugar de en la aplicación.

ORQUESTACIÓN Y COREOGRAFÍA:

Los servicios dentro de un SOA pueden organizarse de dos formas distintas:

- **Orquestación:** Existe un controlador central que dirige las actividades del resto de servicios. Este controlador se denomina servicio **orquestador**. Cada servicio reciben y responden los pedidos del orquestador. *(en una orquesta, los músicos siguen las instrucciones del director)*
- **Coreografía:** No existe un controlador central, sino que los servicios se organizan entre ellos. La ejecución de las actividades es responsabilidad de cada servicio, y no de un controlador. *(en una coreografía, no hay director)*

La orquestación es, por lo tanto, más restrictiva que la coreografía. Los algoritmos que usan orquestación son comparables al funcionamiento de un **semáforo**, mientras que los que usan coreografía pueden compararse con una **rotonda**.



La orquestación se basa, en mayor parte, en la ejecución de **procesos de negocio**, en los que es importante un único **flujo de control** controlado por un orquestador. Sin embargo, la coreografía está más bien orientada a la **colaboración entre actores**, que ofrece una visión más abstracta y menos detallada del sistema.

WS-BPEL:

WS-BPEL (*Web Service Business Process Execution Language*) es un lenguaje para la implementación de SOA para servicios web mediante **orquestación**. Está basado en **XML** y puede mantener estructuras de datos en variables. BPEL define un modelo y una gramática para el comportamiento de un proceso de negocio, basándose en las interacciones con los clientes, además de la manera de coordinar estas interacciones.

CARACTERÍSTICAS DE WS-BPEL:

WS-BPEL, como lenguaje que es, nos proporciona:

- **Constructores condicionales** y de time-out
- Condiciones de **excepción** o de error
- Se **coordina** con las partes para saber si un proceso ha terminado o no.
- Un mecanismo de **conurrencia**.
- **Exclusión mutua** con variables compartidas por concurrencia.

WSDL:

WSDL (*Web Service Definition Language*) es un lenguaje basado en XML y que se emplea para describir la **funcionalidad concreta** de un servicio web, mientras que WS-BPEL nos permite implementar el proceso en sí y manejar los contratos. Fue desarrollado por la W3C.

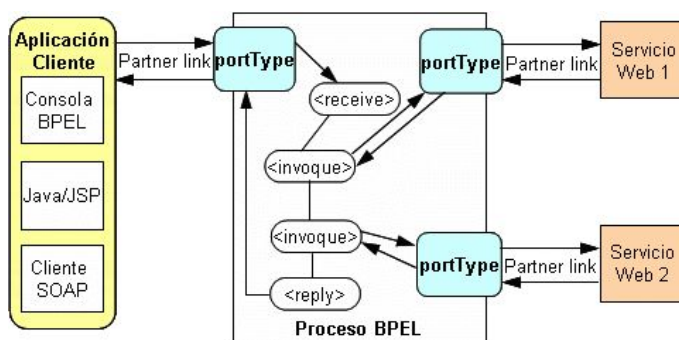
Un **proceso** se modela, empleando WS-BPEL, como una colección o colaboración de **servicios**, cada uno definido en WSDL. En un documento WSDL, se **definen** los servicios que usarán los procesos WS-BPEL.

DESCRIPCIÓN DE WS-BPEL 2.0:

La estructura de cualquier proceso en WS-BPEL se divide en actividades, que pueden ser **básicas**, o **estructuradas** (conjunto de básicas). Cada actividad contiene sus propias variables y contenedores. Por otro lado, los procesos WS-BPEL se suelen estructurar siguiendo esta jerarquía:

1. Declaración de las **relaciones** con los partners o clientes
2. Declaración de **variables**
3. Declaración de **manejadores**
4. Descripción del **comportamiento** del proceso.

ESQUEMA BPEL:



A la vista de la aplicación y los servicios, el proceso BPEL es un servicio más que actúa como orquestador de la red.

Los portType son las interfaces que definen qué operaciones puede realizar cada cliente o servicio en el proceso.

DEFINICIÓN DE UN PROCESO BPEL:

Antes de trabajar en la implementación de un proceso, este puede contener alguno de estos elementos en su primer nivel, como parte de su cabecera:

- **<import>**: importación de paquetes y ficheros, como en C++ o Python.
- **<partner_links>**: se declaran enlaces a otros servicios.
- **<variables>**: definición de variables globales.
- **<faultHandlers>**: manejadores de fallos.
- **<eventHandlers>**: manejadores de eventos.

Tras definir estos elementos, se procede a implementar la actividad principal.

ALGUNAS ACTIVIDADES BÁSICAS DE UN PROCESO BPEL:

- **<assign>**: Se asigna un valor a una o varias variables
- **<empty>**: No hace nada, pero puede contribuir a la sincronización.
- **<exit>**: Fuerza la salida del proceso.
- **<invoke>**: Invoca un servicio web.
- **<receive>**: Espera la llegada de un mensaje en una operación específica.
- **<reply>**: Envía una respuesta al mensaje previamente recibido.
- **<throw>**: Lanza una excepción o fallo
- **<wait>**: El proceso se para durante un periodo de tiempo.

ALGUNAS ACTIVIDADES ESTRUCTURADAS DE UN PROCESO BPEL:

- **<sequence>**: Comienza una ejecución secuencial de las actividades que se encuentren dentro del bloque
- **<flow>**: Proporciona concurrencia a las actividades que se encuentren dentro del bloque
- **<scope>**: Define variables locales para una actividad concreta.
- **<if><condition> ... <else> ...**
- **<while> <condition>**
- **<foreach>**
- **<repeatUntil>**: La actividad contenida dentro del bloque se ejecuta repetidamente hasta que se cumple una determinada condición.

BPEL PARA ECLIPSE:

El entorno de desarrollo Eclipse contiene un plug-in para el desarrollo de SOA con BPEL. Ofrece una interfaz gráfica como alternativa a tener que redactar el código BPEL en XML.

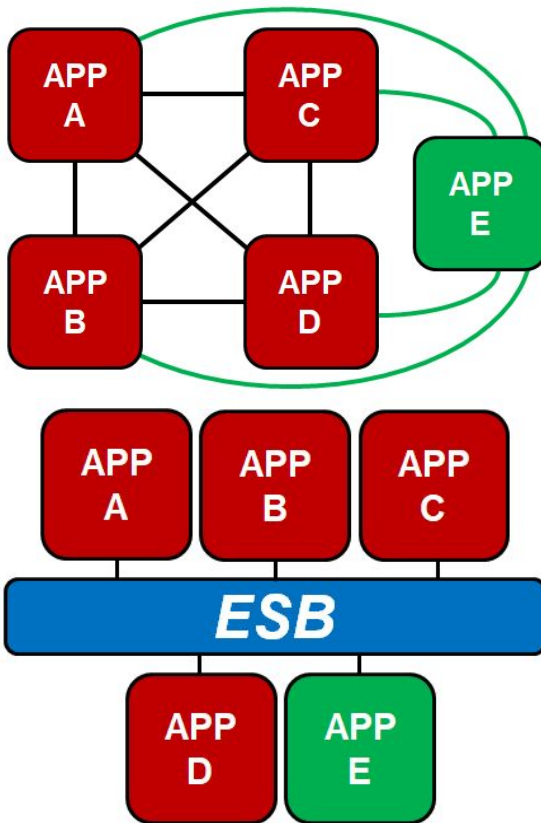


SEMINARIO 7 - INTRODUCCIÓN A ESB

DEFINICIÓN:

Un ESB (Enterprise Service Bus) se define, dentro de las Arquitecturas Orientadas a Servicios (*ver seminario 6*), como un **elemento de integración** para las mismas, al igual que WS-BPEL. Esto quiere decir que permite exponer a los clientes diferentes funciones, recursos o aplicaciones como **servicios**, ofreciendo soluciones a diversos problemas de integración que podrían ocurrir usando BPEL.

¿CUÁNDO ES ÚTIL UN ESB?:



→ Tenemos una arquitectura P2P con las aplicaciones **A, B, C y D**. Queremos añadir a la red una nueva aplicación, **E**.

En este caso, esto podría suponer una **complejidad** de implementación elevada, ya que **E** podría integrarse con cada aplicación de forma distinta, puede que haya que modificar aspectos individuales de cada aplicación, etc...

→ Añadiendo un ESB al sistema, no desaparece la complejidad de integración pero las herramientas que nos ofrece el ESB nos **facilitan** mucho el trabajo.

Además, se **reducen los costes** de actualización y mantenimiento.

ESB es ideal para trabajar en entornos **heterogéneos**, es decir, que utilizan distintas tecnologías, protocolos, lenguajes, etc...

FUNCIONES DE UN ESB (I):

- **Transparencia de localización:** El consumidor de un servicio no tiene por qué conocer la localización del proveedor. El ESB hace el enrutamiento, aunque el proveedor cambie su localización (*desacoplamiento de espacio*)
- **Conversión de protocolos:** Un ESB se adapta a distintos protocolos de transporte (HTTP, JMS, TCP...), útil si el cliente de un servicio utiliza un protocolo distinto al que usa el proveedor.
- **Transformación de mensajes:** Igual que la anterior, pero con formatos de mensaje como pueden ser XML, JSON, HTML...

FUNCIONES DE UN ESB (II):

- **Encaminamiento de mensajes:** Una serie de lógicas y reglas previamente implementadas dirige los mensajes de un proveedor a diferentes clientes.
- **Enriquecimiento de mensajes:** Un ESB puede añadir información extra a los mensajes, como detalles del cliente que se comunica con un proveedor empleando una base de datos externa.
- **Seguridad:** Protocolos de autenticación, autorización de usuarios, encriptación de mensajes...
- **Administración y monitorización:** El ESB puede monitorizarse en tiempo de ejecución, al ser una herramienta compleja y potente.

ALGUNOS ESB DE CÓDIGO ABIERTO:

- **Apache ServiceMix y Tuscany**
- **JBoss ESB**
- **Mule**
- **OpenESB**
- **Petals ESB**
- **Spring Integration**
- **WSO2 ESB**

MULE:

Según *Rademakers y Dirksen*, de todos estos ESB, **Mule** es el mejor para su uso, dando estas razones, entre muchas otras:

- Incorpora todas las funciones relevantes de un ESB
- Bien documentado y con visibilidad en el mercado (se usa mucho)
- Desarrollo y comunidad activas, en constante actualización
- Altamente flexible y extensible
- Soporta una gran cantidad de protocolos de transporte
- Se integra fácilmente con otros proyectos de código abierto
- **Tiene IDE propio: MuleStudio.**



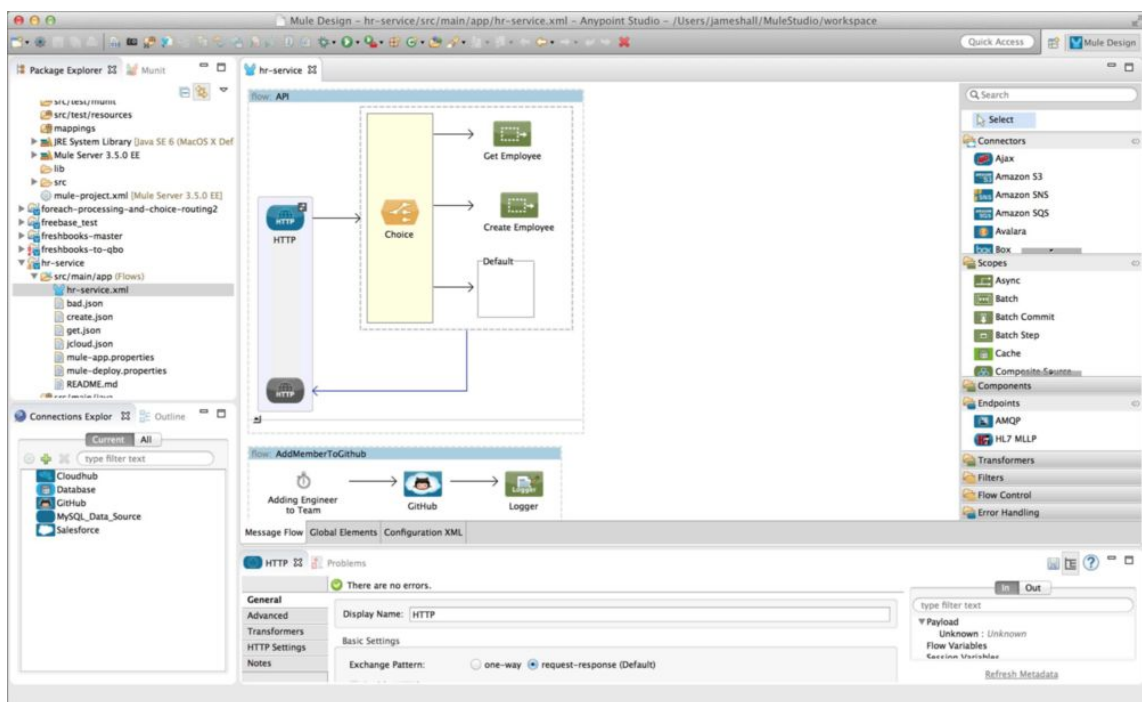
SEMINARIO 8 - INTRODUCCIÓN A MULE CON ANYPOINT

ANYPOINT STUDIO:

Anypoint es una interfaz gráfica basada en Eclipse para el desarrollo de **ESB**, que abstrae al usuario de los detalles más técnicos de Mule para conseguir un manejo **sencillo e intuitivo**. Anypoint se encarga de escribir los códigos a partir de un esquema de elementos *drag-and-drop* que es lo que construye el usuario manualmente, aunque también puede modificar el código XML si lo desea.

Lo que se desarrolle tanto en el esquema como en el editor XML se actualizará automáticamente en el otro entorno. Además, las aplicaciones desarrolladas pueden desplegarse en la nube.

INTERFAZ DE ANYPOINT STUDIO:



DESCARGAR ANYPOINT STUDIO:

Actualmente, Anypoint Studio consta de una versión **comercial** de pago con 30 días de prueba. La versión basada en Eclipse está **actualmente obsoleta**, por lo que es necesaria una **versión más antigua de Eclipse** para que la versión comunitaria funcione sin fallos de compatibilidad. Para ello, usamos el [instalador de Eclipse](#).

Try the Eclipse **Installer** 2020-03 R

The easiest way to install and update your Eclipse Development Environment

±2,677,056 Downloads

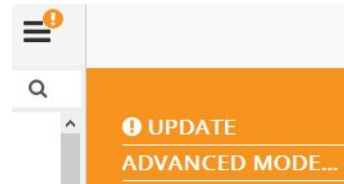
Download

Mac OS X 64 bit
Windows 64 bit
Linux 64 bit

INSTALACIÓN ECLIPSE:

IMPORTANTE: Si ya está Eclipse instalado, es recomendable realizar esta instalación en una máquina virtual. Como vamos a instalar una versión más antigua, es posible que algunas opciones dejen de funcionar.

1. Descargamos el instalador de Eclipse para nuestro sistema operativo
2. Una vez se abra el instalador, pinchamos en las tres barras de arriba a la derecha y, seguidamente, en **Advanced Mode**.
3. En Product Version, marcamos **Oxygen** y, en Java 1.8+ VM, un kit de desarrollo para **Java 1.8**

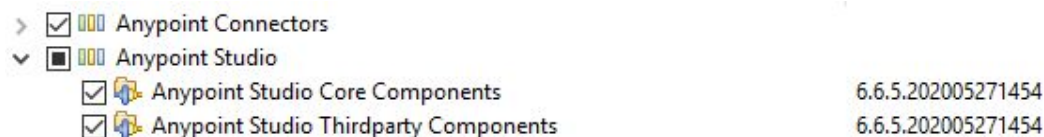


Product Version:	Oxygen
Java 1.8+ VM:	C:\Program Files\Java\jre1.8.0_241 (Current)

4. Pinchamos en Next todo el rato hasta finalizar la instalación.

INSTALACIÓN ANYPOINT Y MULE:

Una vez ejecutemos Eclipse, nos vamos a *Help* (barra superior), y pinchamos en *Install New Software*. En el primer campo, llamado *Work with:*, introducimos la dirección <http://studio.mulesoft.org/r6/plugin> y pulsamos Intro. A continuación, marcamos *Anypoint Connectors* y, dentro de *Anypoint Studio*, sólo las dos primeras opciones que aparecen.



A mitad de la instalación, nos aparecerá un mensaje de aviso en el que tendremos que pinchar en *Install Anyway*. Una vez instalado, reiniciamos Eclipse. Una vez reiniciado, para abrir un proyecto de Anypoint, cerramos la pestaña de Welcome, pinchamos en *Open Perspective* y seleccionamos *Mule Design*.



OBJETIVO Y ACTIVEMQ:

Nuestro objetivo es crear un sistema publicador-suscriptor muy simple, creando un ESB que actúe como intermediario entre el cliente y el sistema. Para instalar el sistema publicador-suscriptor en nuestra máquina, usaremos el software de Apache [ActiveMQ](#). Nos vamos a *Components > ActiveMQ 5* y descargamos para nuestro SO.

Una vez instalado, descomprimos el archivo comprimido, abrimos un terminal dentro de la carpeta `/bin` y escribimos `./activemq start`. Para que esta instrucción funcione, es necesario tener instalada la última versión de [Java](#) y añadir la variable `JAVA_HOME` al `PATH`.

TIPOS DE ELEMENTOS EN ANYPOINT:

- **Connectors:** Comunican a nuestra aplicación de Eclipse Anypoint con el mundo exterior, como un cliente desde una terminal. Pueden recibir información (inbound) o enviarla (outbound). Se representan en nuestros esquemas con el color azul.
- **Scopes:** Permiten facilitar la legibilidad del código XML o implementar procesamiento paralelo agrupando varios procesadores de lenguaje, es decir, bloques que pueden leer, modificar o escribir sobre los mensajes. Se representan con el color verde.
- **Components:** Añaden una funcionalidad específica para nuestro flujo, como la impresión por pantalla o el logueo de usuarios, permitiendo la integración de Software como Servicio (SaaS). Estos se invocan cuando reciben un mensaje, y suelen devolver otro nuevo o el mismo modificado. También se representan con el color verde.
- **Transformers:** Enriquecen o modifican los mensajes, convirtiéndolos de un tipo a otro. De nuevo, se representan en color verde.
- **Filters:** Determinan si un mensaje puede, o no, continuar a través del flujo de la aplicación. En caso contrario, se rechazan. Se representan en nuestro esquema con el color amarillo.
- **Flow Controls:** Especifican cómo se encaminarán los mensajes dentro de nuestro flujo. Esto incluye agrupación de mensajes, separación, enrutamiento tipo switch... También aparecen en color amarillo.
- **Error Handlers:** Ofrecen procedimientos para el manejo de excepciones dentro de Anypoint. Se representan en nuestro esquema con el color naranja o el color rojo.

ESQUEMA DE FLUJO TÍPICO:

1. Una **fuentes** de mensajes que se activan cada vez que reciben uno.
2. Un **filtro** que acepta o rechaza el mensaje.
3. Un **transformador** que, en caso de que se acepte el mensaje, convierte el tipo del mensaje a uno más legible (por ejemplo, array de bytes a string).
4. Podemos usar el componente **Logger** para mostrar en consola el contenido del mensaje y así monitorizarlo.
5. Un **enriquecedor de mensajes** que añade información importante o incluso metadatos al mensaje convertido.
6. Un **componente** que se encarga de redirigir el mensaje a una aplicación o servicio determinado, dependiendo de su composición.
7. Dicho servicio puede retornar la respuesta al cliente directamente, o bien se almacenan en una base de datos.

CREAR PROYECTO MULE EN ECLIPSE:

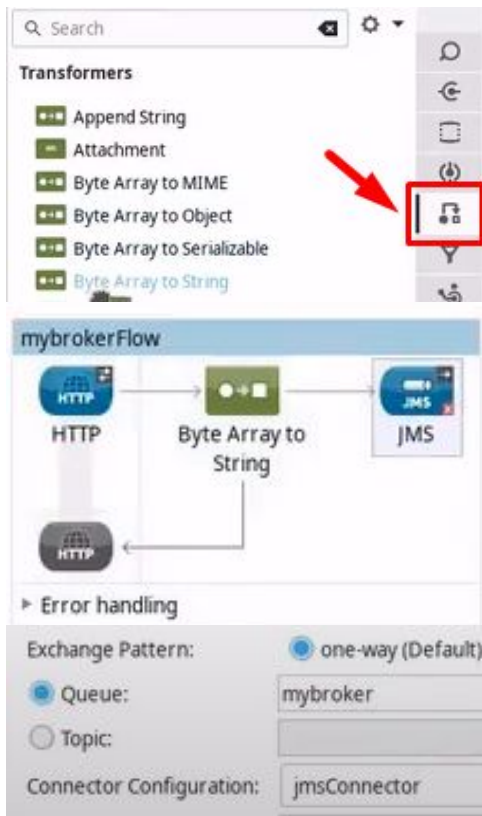
1. Desde la perspectiva de Mule, vamos a *File > New > Mule Project*
2. Le damos un nombre al proyecto, y hacemos click en *Install Runtimes*
3. Buscamos *Mule*, e instalamos la versión más reciente de los paquetes que están bajo *Anypoint Studio Community Runtimes*.
4. Una vez finalice la instalación, reiniciamos Eclipse, volvemos a crear el proyecto seleccionando el runtime previamente instalado.
5. Le damos a Next hasta finalizar todos los pasos.

CONFIGURAR CONECTOR HTTP:

El cliente enviará array de bytes con datos a nuestro ESB mediante mensajes HTTP. Desde la perspectiva de *Message Flow*, arrastramos un conector HTTP al canvas. En la pestaña bajo el canvas, seleccionamos HTTP:



1. En *Path*, escribimos el nombre que le hemos dado a nuestro proyecto.
2. En *Connector Configuration*, elegimos crear una nueva configuración.
3. Cambiamos el *host* a *localhost*. El puerto puede quedarse en 8081.



Los mensajes que nos enviará el cliente llegan como array de bytes, así que para que sean legibles nos interesa transformarlos a string. Para ello, arrastramos un *Transformer* de tipo *Byte Array to String* a dentro del *conector HTTP* previamente configurado.

Para imprimir esta salida, arrastramos un Conector JMS también al conector HTTP, como se muestra abajo a la izquierda.

Este conector JMS debe configurarse desde la ventana bajo el canvas. Para ello, vamos a la pestaña JMS y:

1. Marcamos *Queue* y escribimos el nombre de nuestro proyecto.
2. Añadimos una nueva configuración desde *Connector Configuration*.
3. Seleccionamos la configuración de *ActiveMQ* y la guardamos.

Para que esto funcione, es necesario añadir la librería de Java incluida con la descarga de ActiveMQ al proyecto.

AGREGAR LIBRERÍA DE JAVA:

1. A la izquierda de la pantalla, hacemos click derecho en el nombre de nuestro proyecto, justo abajo de donde pone *Package Explorer*.
2. Vamos a *Properties > Java Build Path > Libraries > Add External JARs*.
3. Vamos a la carpeta que descomprimos al descargar ActiveMQ, y hacemos doble click en el archivo .jar.

Una vez hecho esto. Podemos realizar una primera ejecución de nuestra aplicación Mule. Para ello, volvemos a hacer click derecho en el nombre de nuestro proyecto bajo *Package Explorer > Run as > Mule Application*. Si todo ha salido bien y está *ActiveMQ* iniciado, veremos esto en consola:

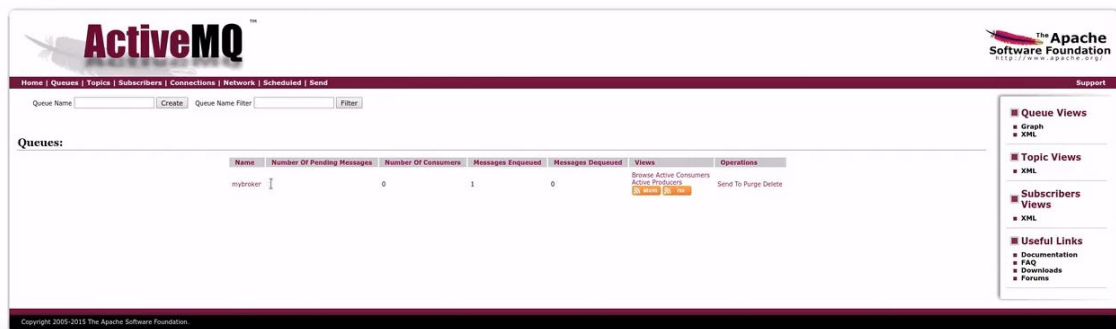
```
*****
*      - - + APPLICATION + - -      *      - - + DOMAIN + - -      *      - - + STATUS + - - *
*****
* mybroker                          * default                          * DEPLOYED                          *
*****
```

ENVÍO DE MENSAJE POST AL SERVIDOR:

Desde terminal y usando el comando curl, escribimos el siguiente comando:

```
curl -X POST -H "Content-type: application/json" -d
"{\"origen\":\"loquequemos\", \"destino\":\"loquequemos\",
\"mensaje\":\"loquequemos\"}" http://localhost:8081/<nombre de
nuestro broker>
```

Para comprobar que el mensaje se ha enviado correctamente, vamos al panel de administración de ActiveMQ situado en *localhost:8081* (debería haberse abierto de forma automática al iniciar ActiveMQ) y pinchamos en la sección *Queues*.



Nuestro mensaje, en formato JSON, debería aparecer en la pestaña “Message Details”, situada abajo del todo.

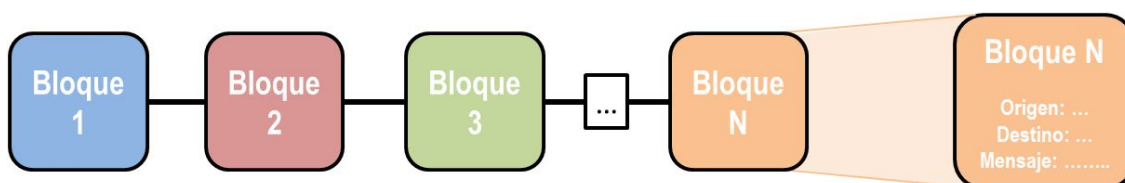
En resumen, hemos enviado un mensaje como clientes usando el comando curl, este ha sido recibido por el ESB que hemos creado y estaba funcionando, y este lo ha enviado a la dirección de ActiveMQ especificada, que es nuestro sistema publicador y suscriptor.

SEMINARIO 9 - INTRODUCCIÓN A BLOCKCHAIN

ARQUITECTURA BLOCKCHAIN:

Blockchain, tal y como indica su nombre, es una cadena de bloques que componen una **información** completa, y donde cada bloque contiene una serie de **datos** concretos. Al no existir un nodo central, es un sistema **seguro** y poco susceptible a la modificación no autorizada de los datos. Por ello, se emplea con mucha frecuencia en sistemas basados en transacciones, como los bancos.

Respecto a las transacciones, la cadena completa supervisa los nuevos bloques que se añaden y su información, sin necesidad de recurrir a terceros que puedan fallar. A su vez, **blockchain es la tecnología detrás de Bitcoin**, pues con blockchain se tiene el registro de las transacciones, como si fuera un libro de cuentas.

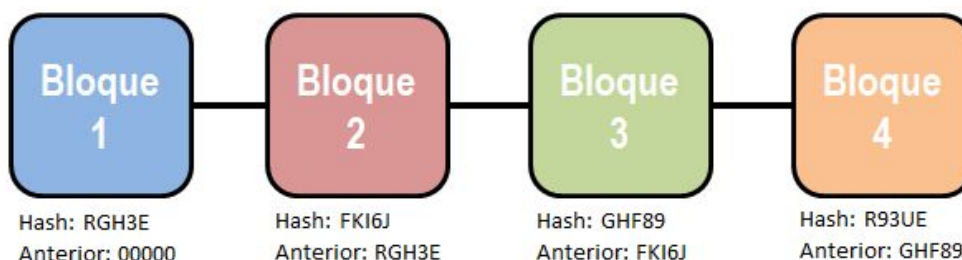


La información que se almacena en cada bloque dependerá del diseño de nuestra blockchain y el caso de uso.

HASH ASOCIADO AL BLOQUE:

Un hash es un **número único** asociado a cada bloque, que se puede interpretar como su “huella dactilar”, y es una de las razones por las que blockchain es una arquitectura segura. Este hash permite **identificar a un bloque su contenido** sin riesgo de confusión con el contenido o el nombre de otro. Por ello, si un bloque se modifica, su hash también cambia.

Cada bloque también almacena el **hash del bloque anterior** en la cadena (excepto el primero, llamado bloque génesis). Así, si un atacante modifica el contenido de un bloque y, por tanto, cambia el hash, el resto de la cadena quedará **invalidada** y no registrará el cambio.



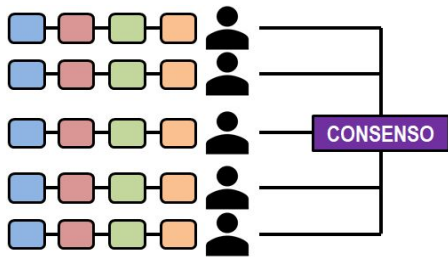
En este ejemplo, si un atacante modifica el bloque 2, cambia su hash, pero sería diferente al registrado por el bloque 3, que habría detectado el ataque.

PRUEBA DE TRABAJO:

Suponiendo el ataque mencionado anteriormente, el mismo atacante podría modificar el resto de bloques antes de que sea detectado. Para evitar esto, una gran parte de las blockchain introducen el concepto de **prueba de trabajo**, que consiste en la resolución de una **tarea por parte del cliente** para poder crear los bloques.

Estas tareas suelen tardar 5-10 minutos en realizarse, dando a la cadena tiempo de sobra para realizar comprobaciones antes de que el atacante cree termine de modificar todos los bloques de la cadena.

RED P2P DISTRIBUIDA:



→ Además de las prueba de trabajo, se suele implementar una blockchain en una **red P2P distribuida**. De esta manera, cada vez que alguien se conecta a la red, recibe una copia de la blockchain con todos sus bloques, convirtiéndose así en un **nodo** de la red P2P.

Cada vez que se crea un bloque, todos los nodos comprueban que no ha sido modificado de forma no autorizada y, en caso positivo, se añade a las cadenas de todos los nodos. Se dice que los nodos componen un **consenso** entre ellos.

TRANSACCIONES BLOCKCHAIN:

Puesto que blockchain es una arquitectura muy orientada a la seguridad e integridad de los datos, el proceso de una transacción en blockchain es el siguiente:

1. **Solicitud:** Un usuario solicita una transacción. Esto es, intercambio de criptomonedas, contratos, registros, etc...
2. **Transmisión:** La solicitud se transmite a todos los nodos de la red P2P que está asociada a dicha cadena.
3. **Validación:** Utilizando algoritmos públicos, libres y conocidos, los nodos validan la transacción. Esto es, comprobar que no se ha modificado nada desde el momento de su creación (consenso).
4. **Se añade:** Una vez se verifica la transacción, esta se añade a la cadena, que luego será replicada y enviada a todos los nodos de la red. Este nuevo bloque es **permanente** (no se puede borrar) e **inalterable** (no se puede modificar).



VENTAJAS DE BLOCKCHAIN:

- **Resiliencia:** Puesto que blockchain se basa en arquitectura replicada, un ataque a un nodo no supondría ningún problema puesto que la cadena no alterada seguirá estando en una gran mayoría de los otros nodos. Tener el control de una amplia parte de una red P2P tiene un coste desmesurado.
- **Reducción de tiempos:** Al existir una única versión (o “libro de cuentas”) de las transacciones, no es necesario un largo proceso de verificación que requiera la asistencia de terceros. La blockchain replicada se encarga de todo.
- **Fiabilidad:** Blockchain siempre verifica y certifica la identidad de las partes involucradas en una transacción.
- **Transacciones inalterables:** Como las transacciones se registran en orden cronológico, la inalterabilidad de los bloques está certificada. Si se modificase un bloque, también habría que hacerlo con todos los posteriores.
- **Prevención de fraude:** El consenso entre los nodos evita que se pierda información por fraude o malversación. Por ello, blockchain también supone un mecanismo de monitorización de coste reducido.
- **Seguridad:** Normalmente, un ataque informático tiene un objetivo específico pero, en una blockchain, todos esos objetivos específicos cuentan con una copia de la cadena, manteniendo el sistema activo en caso de ataque.
- **Transparencia:** Los cambios en una blockchain son siempre públicos, haciendo aún más inalterables las transacciones.
- **Colaboración:** Las partes involucradas en una transacción no necesitan la intervención de ningún tercero como en los bancos tradicionales.
- **Descentralización:** Existen reglas de estándares que determinan cómo se intercambia la información entre los nodos, asegurando que todas las transacciones son validadas y, en dicho caso, añadidas a la blockchain.

VERSIONES DE BLOCKCHAIN:

- **Blockchain 1.0:** La creación de DLT (Distributed Ledger Technology), es decir, el libro de cuentas (**ledger**) distribuido que caracteriza a blockchain, fue aplicado sólo para el campo de las **criptomonedas**, permitiendo transacciones financieras con, como ejemplo claro, Bitcoin
- **Blockchain 2.0:** Introdujo el concepto de **contrato inteligente**, que es un software que reside dentro de las blockchain que están comprobando de forma constante el cumplimiento y verificación de los bloques. Nacen como alternativa a los contratos tradicionales.
- **Blockchain 3.0:** Introdujo las aplicaciones descentralizadas o **DApps**, que distribuye la ejecución (*backend*) de un software entre distintos bloques de la cadena, pese a que el usuario lo vea como una aplicación normal (*frontend*).

VARIANTES DE BLOCKCHAIN:

Dependiendo de la implementación de una blockchain, puede existir distintas variantes de las mismas:

- **Públicas:** Los ledgers son visibles para todo el mundo y cualquiera puede verificar o añadir un nuevo bloque a la cadena. Ofrecen incentivos para que las personas se unan, como recompensas en bitcoin, y son de uso gratuito.
- **Privadas:** Suelen ser exclusivas de organizaciones o empresas. Aunque son visibles al público, sólo los miembros autorizados de la entidad pueden añadir y verificar bloques.
- **Consorcio:** Sólo los miembros autorizados pueden ver, verificar y añadir.

CASOS DE USO DE BLOCKCHAIN:

- **Mercados:** Facturación y supervisión de los datos, gestión de cuotas...
- **Gobiernos:** Votaciones, digitalización de documentos de identidad...
- **IOT:** Redes de sensores, domótica, ciudades inteligentes, asistentes...
- **Salud:** Gestión de datos médicos, fichas de salud, cartera de salud...
- **Arte y ciencia:** Supercomputación, análisis de ficheros, recursos P2P...
- **Finanzas:** Pagos en moneda digital, contabilidad, mercados de capital...
- **Entretenimiento:** Juegos como [CryptoKitties](#), [oxUniverse](#)...

CRIPTOMONEDAS:

Las criptomonedas nacen como alternativa digital a las monedas tradicionales como los dólares o los euros. El nombre deriva de los métodos criptográficos empleados para el intercambio de información con estas divisas digitales.

A diferencia de las monedas tradicionales, el titular de una moneda digital es a su vez el propietario de esta, en lugar del banco que las emite. Cualquiera puede utilizar criptomonedas sin pagar cuotas de proceso. Además, puesto que las criptomonedas utilizan la tecnología de blockchain, el remitente y destinatario de una transacción no necesitarán recurrir ni a un banco ni a ningún otro tercero.

BITCOIN:

En 1998, se publicó un artículo que definía un sistema anónimo de dinero electrónico y distribuido llamado “B-Money”. En 2009, el usuario anónimo *Satoshi Nakamoto* lanzó Bitcoin, basado en dicho artículo.

Bitcoin es una tecnología P2P no regulada por ningún banco ni gobierno, puesto que su emisión se realiza por la red de forma colectiva. Es la criptomoneda dominante en el mundo y es de código abierto, de tal forma que nadie posee el control de Bitcoin. A día de hoy, se han emitido unos 21 millones de Bitcoin. (1 Bitcoin = ~8.500€)

BLOCKCHAIN Y BITCOIN:

Blockchain es la tecnología detrás de Bitcoin. Es decir, Bitcoin es el **token** o moneda digital en sí, y blockchain es el **ledger** o libro de cuentas que lleva el registro de las transacciones. Se puede tener blockchain sin Bitcoin, pero no se puede tener Bitcoin sin blockchain.



MITOS SOBRE BLOCKCHAIN:

- **Resuelve todos los problemas**
 - No, solo los que puedan solucionarse con una base de datos distribuida.
- **No es fiable**
 - Al contrario, lo que puede no ser fiable son las tecnologías que usen blockchain, como con cualquier otro recurso.
- **Es privada**
 - No, solo asegura la integridad pero no la confidencialidad.
- **Es completamente inmutable**
 - No, solo es prácticamente imposible con las tecnologías actuales.

DESVENTAJAS DE BLOCKCHAIN:

- **Riesgo de error:** El factor humano puede derivar en errores relacionados con el almacenamiento de datos. Blockchain no puede asegurar que todas sus aplicaciones estén correctamente implementadas.
- **Desperdiciable:** Que todos o casi todos los nodos de una red P2P tengan que verificar constantemente los bloques que se añaden, pese a que suponga un tiempo mínimo de inactividad, al final es un desperdicio de la capacidad de cómputo. Se necesitan muchas verificaciones para llegar al consenso entre los nodos.

OTRAS CRIPTOMONEDAS POPULARES:

- Ethereum
- Bitcoin Cash
- Ripple
- Litecoin
- Dogecoin
- [*Pesetacoin*](#)

SEMINARIO 10 - GIT

SISTEMAS DE CONTROL DE VERSIONES:

Un Sistema de Control de Versiones, o SCV, es realmente un mecanismo para compartir ficheros entre distintos participantes, pero que a su vez gestiona las versiones creadas del mismo. Entre sus funcionalidades, se destacan:

- Creación de **copias de seguridad** de los ficheros.
- En relación con esta última, se pueden **restaurar** versiones anteriores.
- **Sincronizar** archivos.
- Controlar la **autoría** de los archivos.
- Si se tratase de código, permite realizar **pruebas aisladas** del mismo.

ELEMENTOS DE UN SCV:

Los elementos básicos de un SCV son:

- **Repositorio:** Contiene el conjunto de archivos que forman el proyecto.
- **Servidor:** Allí se aloja nuestro repositorio
- **Copias locales del proyecto**
- **Ramas:** Una rama es una localización dentro de un repositorio, como si fuese un sistema de directorios. La rama principal de desarrollo se llama **master**.

OPERACIONES DE UN SCV:

Las operaciones básicas y más importantes de un SCV son:

- **Add:** añade un nuevo archivo al SCV
- **Revision:** permite controlar el histórico de versiones de un archivo
- **Head:** versión más reciente del repositorio o una rama concreta
- **Clone/checkout:** Crea una copia de trabajo local de un repositorio
- **Commit/checkin:** envía cambios locales al repositorio y cambia su versión.
- **Log:** histórico de cambios en el repositorio
- **Update/synchronize:** sincroniza un repositorio local con su última versión.
- **Revert/reset:** deshace cambios para volver al último estado del repositorio.

Dentro de estos sistemas, también encontramos algunas operaciones igual de importantes, pero más avanzadas:

- **Branches:** Crea una copia (*rama*) de un recurso para trabajar con diferentes versiones de un repositorio a la vez. Permite que un recurso pueda ser desarrollado de formas distintas e independientes.
- **Diff/change:** encuentra diferencias entre dos versiones del repositorio.
- **Merge/patch:** une o fusiona ramas.
- **Conflict:** registra y muestra cambios solapados sobre el mismo recurso.

SCV CENTRALIZADOS Y DISTRIBUIDOS:

CENTRALIZADOS	DISTRIBUIDOS
Un único servidor, sobre el que todos los participantes colaboran, almacena todo el repositorio. Las modificaciones se deben hacer de forma secuencial , pero son los más fáciles de usar. Un ejemplo es <i>Subversion</i> .	Cada participante posee una copia de todo el repositorio, permitiendo así hacer modificaciones paralelas . Es más complejo de usar, pero mucho más flexible al contar con más comandos para las ramas. Un ejemplo es <i>Git</i> .

GIT:

Git es un Sistema de Control de Versiones **distribuido** para el seguimiento de los cambios que se producen en los archivos de un proyecto. Puede gestionar proyectos de casi cualquier tipo y lenguaje, aunque para usar Git como tal no es necesario tener ningún conocimiento sobre programación.



Al ser un sistema creado por *Linus Torvalds*, también creador del sistema operativo Linux, es de código abierto. De hecho, al principio se creó como una herramienta sólo para máquinas Linux. Pese a ser un sistema rápido y sin limitaciones de tamaño, **garantiza la integridad** de los archivos. No se puede modificar el contenido de un archivo de un repositorio sin que este cambio quede registrado (*checksum*).

CARACTERÍSTICAS PRINCIPALES DE GIT:

- Permite que muchos desarrolladores puedan **coordinar** el trabajo y conocer la persona, fecha y motivo responsable de un cambio, sin necesidad de que todos los involucrados estén conectados en la misma red.
- Permite **recuperar** cualquier versión de cualquier repositorio siempre que esta versión haya sido guardada y registrada.
- Existen repositorios **locales y remotos**. Desde los repositorios locales se puede actualizar el repositorio remoto de un proyecto (operación *push*).
- Cuando se registra una nueva versión (commit), se registra:
 - Una copia del estado del repositorio
 - Quién realizó cada modificación específica
 - Motivos y notas de cada cambio
- Git gestiona **conflictos** entre versiones y todas las ramas, o branches, de un proyecto, de tal forma que nunca se pierda información.

A diferencia de otros SCV, Git captura el estado de **todos los documentos** o archivos del repositorio a la hora de actualizarlo (commit). En la mayoría de SCV distintos a Git, los commit **solo almacenan los cambios** que se producen.

ESTADO DE LOS ARCHIVOS EN GIT:

Los ficheros involucrados en un repositorio Git pueden tener 4 estados:

- **Untracked:** El archivo no existe o Git no tiene constancia de que exista.
- **Unmodified:** El archivo está en el repositorio, pero no ha sido modificado desde el último commit.
- **Modified:** Se ha editado el archivo desde el último commit, pero los cambios no serán añadidos al repositorio al hacer commit.
- **Staged:** Los ficheros en este estado pasarán al estado Unmodified cuando se haga commit, ya que se actualizará su estado en el repositorio.

SECCIONES DE UN PROYECTO GIT:

Directorio de trabajo	“Staged Area”	Repositorio
Al hacer <i>checkout</i> o clone, se crea una copia local del repositorio sobre la que se realizan modificaciones.	Los archivos modificados pasan al estado Staged y se añaden al repositorio cuando se hace <i>commit</i> .	Almacén de todas las versiones de los archivos de los que se haya hecho commit.

METODOLOGÍA DE TRABAJO:

Normalmente, los proyectos en Git siguen el siguiente flujo:

1. **Creamos** el proyecto, importando documentos locales ya existentes o creando un nuevo repositorio sin contenido y sus respectivas copias de trabajo (*repositorios locales*)
2. Realizamos los **cambios** convenientes (añadir, modificar, eliminar ficheros...)
3. Enviamos los cambios al repositorio (**commit**).

COMANDOS BÁSICOS DE GIT (I):

<code>git init</code>	Inicializa el repositorio local en nuestra máquina
<code>git add</code>	Registrar cambios (o creaciones) de archivos
<code>git status</code>	Muestra el estado actual del repositorio
<code>git commit</code>	Guardamos cambios en nuestro repositorio local
<code>git commit -m</code>	Añadimos un mensaje al cambio, explicando motivos
<code>git push</code>	Los cambios en un repositorio local se envían al remoto
<code>git pull</code>	Los datos del repositorio remoto se actualizan en el local
<code>git clone</code>	Clona un repositorio en un nuevo directorio

COMANDOS BÁSICOS DE GIT (II):

git help	Obtener ayuda sobre un determinado comando de git
git diff	Muestra los archivos en estado Modified, pero no los Staged.
git rm	Eliminar un archivo
git mv	Renombrar o cambiar de localización un archivo
git log	Visualizar el historial de commits
git blame	Muestra quién ha modificado por última vez un archivo

EL FICHERO .GITIGNORE:

Git, por defecto, **no tendrá en cuenta carpetas vacías** a la hora de utilizar sus comandos (se puede evitar creando un fichero vacío dentro). Sin embargo, existe otro fichero, llamado `.gitignore`, donde se introducen qué otros archivos tampoco deberán ser tenidos en cuenta por *git*, siguiendo una determinada sintaxis (*ejemplos*):

- Ignorar todos los ejecutables: `*.exe`
- Ignorar todos los archivos temporales: `temp/`
- Ignorar ejecutables, salvo los de la carpeta X: `*.exe, !X/*.exe`

TAGGING:

Un **tag** es una **marca asociada a todo un proyecto** git, que suele utilizarse para asignar versiones a los estados del proyecto (por ejemplo, `v1.0`, `v1.6.4...`). Una vez creado, un tag no puede modificarse, pero sí eliminarse (*no recomendado*).

Por ejemplo, para crear la versión `v1.0.0` de nuestro proyecto, escribimos:

```
git tag v1.0.0 -m "primera versión"
```

Escribiendo únicamente `git tag`, podemos mostrar todos los tag del proyecto y, con `git show`, podemos mostrar el mensaje asociado a una versión concreta. Finalmente, con `git push --tags`, enviamos los tags creados en el repositorio local al repositorio remoto.

INSTALACIÓN DE GIT:

Para todos los sistemas operativos, Git puede instalarse directamente descargando el ejecutable desde su [página principal](#).

De forma alternativa, también puede instalarse desde la terminal de Linux escribiendo el comando:

```
sudo apt-get install git
```

PRIMEROS PASOS CON GIT:

1. Una vez instalado, lanzamos Git (*click derecho > Git Bash*)
2. Configuramos nuestro nombre y nuestro e-mail. Esto no son credenciales, si no datos que usará git para registrar nuestros commits públicamente.
 - `git config --global user.name <nombre de usuario>`
 - `git config --global user.email <nuestro correo>`
3. Creamos nuestro proyecto, supongamos un archivo Python prueba.py
4. Inicializamos el repositorio con `git init`
5. Añadimos dicho archivo al repositorio git
 - `git add prueba.py`
6. Comprobamos el estado del proyecto con `git status`
7. Registramos los cambios en nuestro repositorio con commit:
 - `git commit -m "primer commit"`
8. Asociamos nuestro repositorio local al repositorio remoto:
 - `git remote add origin <enlace al repositorio>`
9. Actualizamos el repositorio remoto (en concreto, su rama master) con los cambios del repositorio local:
 - `git push -u origin master`

COMANDOS BÁSICOS CON BRANCHES:

<code>git branch</code>	Creamos una rama desde la rama actual
<code>git checkout</code>	Cambiar rama de trabajo actual
<code>git checkout -b</code>	Crear y cambiar rama de trabajo actual
<code>git merge</code>	Fusiona una rama con la actual
<code>git branch -v</code>	Muestra el último commit de una rama
<code>git branch --merged</code>	Muestra las ramas que han sido unidas
<code>git rebase</code>	Mueve la secuencia de commits de una rama a otra. Tiene el mismo efecto que merge, pero muestra los cambios de una forma más lineal y ordenada.

