

Oposiciones cuerpo de secundaria.

Esquemas dos páginas sobre temario oposición profesorado Secundaria.

Especialidad informática

Autor: Sergi García Barea

Actualizado Junio 2025

Licencia



Reconocimiento - NoComercial - CompartirIgual (BY-NC-SA): No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.

Índice	
Introducción	3
Para el buen docente	3
¿Para qué prueba están adaptados estos esquemas?	3
Tema 1. Representación y comunicación de la información.	4
Tema 2. Elementos funcionales de un ordenador digital.	6
Tema 3. Componentes, estructura y funcionamiento de la Unidad Central de Proceso.	8
Tema 4. Memoria interna. Tipos. Direccionamiento. Características y funciones.	10
Tema 5. Microprocesadores. Estructura. Tipos. Comunicación con el exterior.	12
Tema 6. Sistemas de almacenamiento externo. Tipos. Características y funcionamiento.	14
Tema 7. Dispositivos periféricos de entrada/salida. Características y funcionamiento.	16
Tema 8. Hardware comercial de un ordenador. Placa base. Tarjetas controladoras de dispositivos y de entrada/salida.	18
Tema 10. Representación interna de los datos.	20
Tema 11. Organización lógica de los datos. Estructuras estáticas.	22
Tema 12. Organización lógica de los datos. Estructuras dinámicas.	24
Tema 13. Ficheros. Tipos. Características. Organizaciones.	26
Tema 14. Utilización de ficheros según su organización.	28
Tema 15. Sistemas operativos. Componentes. Estructura. Funciones. Tipos.	30
Tema 16. Sistemas operativos: Gestión de procesos.	32
Tema 17. Sistemas operativos: Gestión de memoria	34
Tema 18. Sistemas operativos: Gestión de entradas/salidas.	36
Tema 19. Sistemas operativos: Gestión de archivos y dispositivos	38
Tema 20. Explotación y Administración de sistemas operativos monousuario y multiusuario.	40
Tema 21. Sistemas informáticos. Estructura física y funcional.	42
Tema 22 Planificación y explotación de sistemas informáticos. Configuración. Condiciones de instalación. Medidas de seguridad. Procedimientos de uso.	44
Tema 23. Diseño de algoritmos. Técnicas descriptivas.	46
Tema 24. Lenguajes de programación. Tipos. Características.	48
Tema 25. Programación estructurada. Estructuras básicas. Funciones y Procedimientos.	50
Tema 26. Programación modular. Diseño de funciones. Recursividad. Librerías.	52
Tema 27. Programación orientada a objetos. Objetos. Clases. Herencia. Polimorfismo. Lenguajes.	54
Tema 29. Utilidades para el desarrollo y prueba de programas. Compiladores. Interpretes. Depuradores.	56
Tema 31. Lenguaje C: Características generales. Elementos del lenguaje. Estructura de un programa. Funciones de librería y usuario. Entorno de compilación. Herramientas para la elaboración y depuración de programas en lenguaje C.	58
Tema 32. Lenguaje C: Manipulación de estructuras de datos dinámicas y estáticas. Entrada y salida de datos. Gestión de punteros. Punteros a funciones.	60
Tema 34. Sistemas gestores de base de datos. Funciones. Componentes. Arquitecturas de referencia y operacionales. Tipos de sistemas.	62
Tema 35. La definición de datos. Niveles de descripción. Lenguajes. Diccionario de datos.	64
Tema 36. La manipulación de datos. Operaciones. Lenguajes. Optimización de consultas.	66
Tema 38. Modelo de datos relacional. Estructuras. Operaciones. Álgebra relacional.	68
Tema 39. Lenguajes para la definición y manipulación de datos en sistemas de base de datos relacionales. Tipos. Características. Lenguaje SQL.	71
Tema 40. Diseño de bases de datos relacionales.	73
Tema 44. Técnicas y procedimientos para la seguridad de los datos.	75
TEMA 45: SISTEMAS DE INFORMACIÓN. TIPOS. CARACTERÍSTICAS. SISTEMAS DE INFORMACIÓN EN LA EMPRESA	77
TEMA 47: INSTALACIÓN Y EXPLOTACIÓN DE APLICACIONES INFORMÁTICAS. COMPARTICIÓN DE DATOS	81
TEMA 54: DISEÑO DE INTERFACES GRÁFICAS DE USUARIO (GUI)	83

TEMA 60: SISTEMAS BASADOS EN EL CONOCIMIENTO — REPRESENTACIÓN, COMPONENTES Y ARQUITECTURA	85
TEMA 61: REDES Y SERVICIOS DE COMUNICACIONES	87
TEMA 63: NIVEL FÍSICO EN REDES DE COMUNICACIONES: FUNCIONES, MEDIOS, ADAPTACIÓN, LIMITACIONES Y ESTÁNDARES	91
TEMA 64: FUNCIONES Y SERVICIOS DEL NIVEL DE ENLACE. TÉCNICAS. PROTOCOLOS	93
TEMA 65: FUNCIONES Y SERVICIOS DEL NIVEL DE RED Y DEL NIVEL DE TRANSPORTE. TÉCNICAS. PROTOCOLOS	95
TEMA 66: FUNCIONES Y SERVICIOS EN NIVELES SESIÓN, PRESENTACIÓN Y APLICACIÓN. PROTOCOLOS. ESTÁNDARES	97
TEMA 67: REDES DE ÁREA LOCAL. COMPONENTES. TOPOLOGÍAS. ESTÁNDARES. PROTOCOLOS	99
TEMA 68: SOFTWARE DE SISTEMAS EN RED. COMPONENTES. FUNCIONES. ESTRUCTURA	101
TEMA 70: DISEÑO DE SISTEMAS EN RED LOCAL. PARÁMETROS DE DISEÑO. INSTALACIÓN Y CONFIGURACIÓN DE SISTEMAS EN RED LOCAL	103
TEMA 71: EXPLOTACIÓN Y ADMINISTRACIÓN DE SISTEMAS EN RED LOCAL. FACILIDADES DE GESTIÓN	105
Tema 72. La seguridad en sistemas en red. Servicios de seguridad. Técnicas y sistemas de protección. Estándares.	108
Tema 73. Evaluación y mejora de prestaciones en un sistema en red. Técnicas y procedimientos de medidas.	110
Tema 74. Sistemas multimedia.	112

Introducción

Este documento recoge una serie de **esquemas sintéticos del temario oficial para las oposiciones al cuerpo de profesorado de Secundaria, especialidad Informática**, con el objetivo de ofrecer una herramienta de estudio clara, útil y eficaz. Cada esquema está diseñado para ocupar como máximo **cuatro páginas**, facilitando así su consulta rápida, comprensión global y memorización eficaz.

Para el buen docente

Pero estos esquemas **no son solo para superar una oposición**. Están pensados para ayudarnos a **ser mejores docentes**, personas que entienden la complejidad técnica de su materia, pero también su dimensión educativa, social y ética. Ser docente es una tarea de gran responsabilidad que trasciende un examen: **enseñamos a través de lo que sabemos, pero también a través de lo que somos**.

Por eso, si has llegado hasta aquí, te pido algo importante: lleva contigo el compromiso de ser un buen docente más allá de la oposición. Utiliza estos materiales como base, sí, pero hazlos crecer con tu experiencia, tus reflexiones y tu vocación. Que enseñar sea una decisión consciente, diaria, y no un trámite. Que lo que prepares hoy, lo apliques con compromiso durante toda tu carrera docente, pensando siempre en lo mejor para tu alumnado.

¿Para qué prueba están adaptados estos esquemas?

Estos esquemas están específicamente adaptados para la **prueba de exposición oral del procedimiento selectivo regulado por la ORDEN 1/2025, de 28 de enero**, de la Conselleria de Educación, Cultura, Universidades y Empleo de la Comunitat Valenciana, que establece lo siguiente:

"La exposición tendrá dos partes: la primera versará sobre los aspectos científicos del tema; en la segunda se deberá hacer referencia a la relación del tema con el currículum oficial actualmente vigente en el presente curso escolar en la Comunitat Valenciana, y desarrollará un aspecto didáctico de este aplicado a un determinado nivel previamente establecido por la persona aspirante. Finalizada la exposición, el tribunal podrá realizar un debate con la persona candidata sobre el contenido de su intervención."

No obstante, estos materiales pueden ser también útiles para preparar **otras modalidades de oposición** (como ingreso por estabilización o pruebas de adquisición de especialidades), así como para otras especialidades cercanas, especialmente **la de Sistemas y Aplicaciones Informáticas**, ya que comparten gran parte del temario técnico

Tema 1. Representación y comunicación de la información.

1.1 Introducción

- Definición: transformación de fenómenos del mundo real en estructuras digitales binarias.
- Fundamento para todo procesamiento informático: desde el código hasta el hardware.

1.2 Sistemas de Numeración

Base, dígitos, sistema posicional.

Sistemas: binario, octal, hexadecimal y decimal.

Conversión entre sistemas para debugging, direccionamiento y arquitectura.

1.3 Representación de Datos

- **Enteros:**
 - Códigos: signo+magnitud, CA1 y CA2 (uso de CA2 por su simplicidad en hardware).
- **Punto flotante (IEEE 754):**
 - Precisión simple (32 bits) y doble (64 bits), compuestas por signo (1 bit), exponente (8/11 bits) y mantisa (23/52 bits).
 - Se normaliza desplazando el punto binario para que quede en la forma 1.xxxxx, permitiendo maximizar la precisión y garantizar una representación única.
- **Texto:**
 - ASCII (7 bits, limitado), 8 bits extendido y Unicode (UTF-8/16/32) para múltiples idiomas y emoji.
- **Imágenes:**
 - Representación como matriz de píxeles con RGB + canal alfa.
 - Formatos: BMP, PNG (sin pérdida), JPEG (con pérdida).
- **Vídeo:**
 - Secuencia de imágenes y audio. Compresión intraframe e interframe.
 - Códecs: H.264, H.265 (HEVC), AV1 (compresión espacial e interframe).
- **Audio:**
 - Muestreo (44,1 kHz CD / 48 kHz vídeo); cuantificación (16/24 bits).
 - Formatos: WAV/FLAC (sin pérdida), MP3/AAC (con pérdida).

1.4 Lógica y Operaciones Binarias

- Aritmética: suma, resta, multiplicación, división en base 2.
- Lógica digital: puertas AND, OR, NOT, XOR.
- Aplicaciones: ALU, circuitos combinacionales.

1.5 Detección y Corrección de Errores

- Bit de paridad (detección simple).
- CRC (Cyclic Redundancy Check).
- Código de Hamming (corrige 1 bit; MEM ECC).
- Reed-Solomon (múltiples errores; CDs, RAID, QR).

1.6 Representación en Big Data y Nube

- Formatos JSON/BSON, Avro, Protobuf, Parquet, ORC.
- Integración en pipelines cloud, microservicios y análisis masivo (Spark, Hadoop).

1.7 Comunicación Digital

- Modelo Shannon-Weaver: emisor, codificador, canal (ruido), decodificador receptor + (Compresión y cifrado)
- Señales digitales vs analógicas.
- Protocolos: TCP/IP, UDP/IP, Ethernet, WebRTC.

1.8 Seguridad

- Hashing: SHA-256 (integridad), bcrypt/Argon2 (contraseñas).
- Cifrado:
 - Simétrico: AES.
 - Asimétrico: RSA, ECC.
 - Cifrado simétrico pero compartiendo la clave con cifrado Asimétrico (SSL)
 - Aplicaciones: HTTPS, VPN, BitLocker, TLS.

1.9 Compresión

- Sin pérdida: Huffman, LZW (ZIP, PNG).
- Con pérdida: JPEG, MP3, H.264 (optimizadas para multimedia).

1.10 Conclusión

- La correcta representación y manipulación de datos es básica en informática.
- Aplica desde circuitos y sistemas embebidos hasta servicios cloud y Big Data.

2. PARTE DIDÁCTICA - 1.º DAM – Módulo: Programación

Unidad didáctica: “Estructuras repetitivas aplicadas a la representación y tratamiento de información”

2.1 Contextualización didáctica

- Nivel: 1.º de Ciclo Formativo de Grado Superior – Desarrollo de Aplicaciones Multiplataforma (DAM)
- Módulo profesional: Programación
- **Justificación:** Las estructuras repetitivas permiten automatizar tareas, manipular información y modelar procesos fundamentales en programación profesional.

2.2 Objetivos de aprendizaje

- Aplicar bucles (for, while, do-while) en la codificación de algoritmos.
- Manipular datos numéricos y textuales mediante estructuras de control repetitivas.
- Simular procesos de codificación, verificación y transmisión de datos.

2.3 Metodología y principios pedagógicos

- Metodología activa: basada en tareas prácticas y resolución de problemas reales.
- Progresión: ejercicios con dificultad creciente y trabajo individual seguido de refactorización colaborativa.
- Recursos utilizados: IDE Java (NetBeans, VS Code), pseudocódigo, diagramas de flujo, vídeos explicativos.
- Estrategias metodológicas: trabajo por parejas con roles diferenciados, flipped classroom, revisión entre iguales.

2.4 Inclusión y atención a la diversidad (niveles III y IV):

- Código base parcial con comentarios orientativos.
- Retroalimentación individualizada durante el proceso.

2.5 Aplicación del Diseño Universal para el Aprendizaje (DUA):

- Múltiples formas de representación (textual, visual, audiovisual).
- Variedad de formas de expresión del aprendizaje (código funcional, exposición oral, demos).
- Participación equitativa mediante ajustes de complejidad y agrupaciones heterogéneas.

2.6 Actividad principal: “Procesando datos binarios con bucles”

- Conversión manual de números decimales a binario, octal y hexadecimal mediante bucles.
- Codificación de texto carácter a carácter en binario utilizando la tabla ASCII.
- Cálculo de bit de paridad mediante conteo de unos en cadenas binarias.
- Simulación de un canal de transmisión con errores aleatorios y aplicación del Código de Hamming.
- Verificación de integridad de cadenas binarias utilizando un algoritmo de hash simplificado (XOR).

2.7 Evaluación

- Criterios de evaluación: uso correcto y eficiente de bucles, lógica de control adecuada, limpieza del código, comprensión de los procesos implicados.
- Instrumentos de evaluación: rúbricas detalladas, revisión entre compañeros, evaluación continua con entregas parciales.
- Resultados esperados: desarrollo de programas funcionales que evidencien el dominio de las estructuras repetitivas aplicadas a tareas reales de representación y transmisión de datos.

2.8 Conclusión didáctica

Esta unidad permite al alumnado integrar conocimientos fundamentales de programación y aplicarlos en situaciones prácticas. Fomenta el pensamiento algorítmico, el desarrollo de habilidades técnicas y competencias transversales, incrementando la motivación y la autonomía. Además, establece conexiones claras entre la teoría informática y su aplicación profesional, desarrollando un aprendizaje significativo.

Tema 2. Elementos funcionales de un ordenador digital.

1. INTRODUCCIÓN

2. ELEMENTOS FUNCIONALES

2.1. Unidad Central de Proceso (CPU - Central Processing Unit)

Encargada de ejecutar instrucciones del programa.

- **Registros:** PC, IR, MAR, MDR, FLAGS.
- **ALU / FPU:** operaciones lógicas y en coma flotante.
- **Unidad de Control:** cableada (rápida) o microprogramada (flexible).

2.2. Memoria principal

Memoria de acceso rápido que almacena temporalmente datos e instrucciones.

- **RAM (Random Access Memory):**
 - **DRAM (Dynamic RAM):** económica, necesita refresco constante.
 - **SRAM (Static RAM):** más rápida, usada en cachés.
- **Jerarquía de memoria:** estructura escalonada que optimiza acceso:
 - Registros > Caché (L1, L2, L3) > RAM > SSD/HDD.
- Afecta directamente al **rendimiento:** menor latencia en niveles superiores.

2.3. Subsistema de Entrada/Salida (E/S)

Permite la interacción del procesador con dispositivos externos.

- **Dispositivos periféricos:** teclado, ratón, impresora, disco, red.
- **Modos de transferencia:**
 - **Polling:** la CPU consulta activamente si hay datos disponibles.
 - **Interrupciones:** el periférico avisa al procesador cuando necesita atención.
 - **DMA (Direct Memory Access):** transfiere datos directamente sin CPU.

2.4. Sistema de buses

Canales físicos que interconectan los componentes del sistema.

- **Tipos:** **Bus de datos** (transmite información), **bus de direcciones** (localiza posiciones de memoria) y **bus de control** (gestiona operaciones como lectura o interrupciones).
- **Temporización:** puede ser **síncrona** (con reloj compartido) o **asíncrona** (mediante señales independientes, más flexible).

3. MODELOS DE ARQUITECTURA

3.1. Von Neumann

- Memoria compartida para instrucciones y datos.
- Problema: **cuello de botella** en el acceso a memoria.

3.2. Harvard

- Memoria separada para datos e instrucciones.
- Permite acceso paralelo, más eficiente.

4. TAXONOMÍA DE FLYNN

Clasificación de arquitecturas según número de flujos de instrucciones y datos:

- **SISD (Single Instruction, Single Data):** tradicional, una instrucción opera sobre un dato.
- **SIMD (Single Instruction, Multiple Data):** una instrucción actúa sobre múltiples datos (p. ej., GPU).
- **MISD (Multiple Instruction, Single Data):** redundante, escasa utilidad práctica.
- **MIMD (Multiple Instruction, Multiple Data):** múltiples procesadores ejecutan múltiples instrucciones, típico en sistemas multinúcleo.

5. MEMORIAS: TIPOS Y EVOLUCIÓN

- **ROM (Read-Only Memory):** no volátil, incluye BIOS/UEFI.
 - **EEPROM:** puede reprogramarse eléctricamente.
- **Flash:** memoria no volátil usada en SSD y dispositivos móviles.
- **Tendencias actuales:**
 - **HBM (High Bandwidth Memory):** gran ancho de banda, muy cercana al procesador.
 - **GDDR6:** memoria gráfica usada en GPUs.
 - **Optane:** tecnología de Intel basada en memoria persistente de alta velocidad.
 - **SoC (System on Chip):** integración total en un único chip, común en móviles.

6. CICLO DE INSTRUCCIÓN

Etapas secuenciales que sigue la CPU para ejecutar una instrucción:

1. **Fetch**: se lee la instrucción desde memoria.
2. **Decode**: se interpreta la instrucción.
3. **Execute**: se realiza la operación.
4. **Memory**: acceso a memoria si es necesario.
5. **Write-back**: los resultados se guardan.

Técnicas de optimización:

- **Pipeline**: ejecución en paralelo de etapas.
- **Superescalaridad**: ejecución de múltiples instrucciones por ciclo.
- **Out-of-Order Execution**: reordenamiento dinámico para mejorar rendimiento.
- **SMT (Simultaneous Multithreading)**: varios hilos por núcleo (ej. Hyper-Threading de Intel).
- **Multicore**: varios núcleos físicos en un chip.
- Predicción de saltos:
 - **2-bit saturating counter**: predictor simple con 4 estados (00–11); predice salto tomado (10–11) o no tomado (00–01) y ajusta el contador tras cada salto real. Tolerancia a errores aislados y estabiliza la predicción.
 - **Perceptron predictor (ML)**: usa un perceptrón (red neuronal simple) con historial de saltos para aprender patrones complejos; más preciso que métodos clásicos con bajo coste adicional.
 - **TAGE (Tagged Geometric)**: técnica híbrida basada en múltiples historiales de salto; versiones avanzadas pueden combinarse con ML para optimizar pesos y decisiones.
 - **ML avanzado (Deep Learning, SVM, etc.)**: en investigación; muy preciso pero con latencia y coste alto, por lo que aún no se usa en procesadores comerciales.

7. TENDENCIAS FUTURAS

- **Computación cuántica**: uso de **qubits**, permite paralelismo masivo y algoritmos no clásicos.
- **Arquitecturas neuromórficas**: diseñadas para imitar el cerebro humano.
- **Aceleradores de IA**:
 - **TPU (Tensor Processing Unit)**: de Google, optimizadas para redes neuronales.
 - **NPU (Neural Processing Unit)**: en dispositivos móviles.
 - **FPGAs (Field-Programmable Gate Arrays)**: configurables post-fabricación.
- **Sistemas heterogéneos**: combinación de CPU, GPU y otros aceleradores especializados.

APLICACIÓN DIDÁCTICA (CFGs DAM – Módulo “Programación de procesos y servicios”)

1. REQUISITOS PREVI

2. OBJETIVOS DE APRENDIZAJE

- Analizar el impacto de la arquitectura del sistema en la ejecución de servicios.
- Programar de forma eficiente teniendo en cuenta núcleos, concurrencia y jerarquía de memoria.

3. METODOLOGÍA

- ABR (Aprendizaje Basado en Retos). Simulación de entornos reales.
- Uso de herramientas de análisis: **htop**, **perf**, **taskset**, **systemd**.

4. ATENCIÓN A LA DIVERSIDAD (Niveles III y IV).

5. DISEÑO UNIVERSAL PARA EL APRENDIZAJE (DUA)

- **Representación**: diagramas de arquitectura, vídeos explicativos.
- **Expresión**: scripts, paneles, presentaciones.
- **Compromiso**: retos prácticos contextualizados.

6. ACTIVIDAD PRINCIPAL

“Optimizando procesos según la arquitectura del sistema” Proyecto práctico por equipos Python.

Fases:

1. **Análisis del sistema**: detección de arquitectura (núcleos, RAM) con **os**, **platform**, **psutil**.
2. **Programación concurrente**:
 - a. Con **multiprocessing** y **threading**.
 - b. Afinidad a núcleos con **os.sched_setaffinity()**.
3. **Medición de rendimiento**:
 - a. Scripts instrumentados con **time**, **tracemalloc**.
 - b. Análisis con **perf**, **htop**, comparativa de configuraciones.
4. **Automatización**:
 - a. Scripts como demonios con **systemd**.
 - b. Monitorización de CPU, registro en log.
5. **Defensa y entrega**: Informe técnico + exposición oral con resultados y gráficos.

Tema 3. Componentes, estructura y funcionamiento de la Unidad Central de Proceso.

1. Introducción

- La CPU es el núcleo funcional del ordenador: ejecuta instrucciones, coordina operaciones y gestiona recursos.
- Evolución histórica:
 - Mononúcleo: Intel 8086, primeros Pentium.
 - Multinúcleo: Core 2 Duo, AMD Ryzen.
 - Híbridas: Intel Alder Lake, Apple M1/M2 (big.LITTLE).
- Tendencias actuales:
 - Instrucciones específicas para IA (DL Boost, Neural Engine).
 - Integración CPU+GPU.
 - Alta eficiencia energética: clave en móviles y servidores.

2. Estructura interna de la CPU

2.1 Unidad Aritmético-Lógica (ALU)

- Ejecuta operaciones matemáticas, lógicas y de comparación.
- Registros asociados: acumulador, operandos, flags.
- Unidades especializadas:
 - FPU (coma flotante).
 - SIMD: AVX, SSE, NEON.

2.2 Unidad de Control (UC)

- Decodifica instrucciones y genera señales de control.
- Tipos:
 - Cableada: más rápida.
 - Microprogramada: más flexible.
- Técnicas modernas:
 - Pipelining.
 - Ejecución fuera de orden y especulativa.
 - Predicción de saltos con IA.
 - **2-bit saturating counter**: predictor simple con 4 estados (00–11); predice salto tomado (10–11) o no tomado (00–01) y ajusta el contador tras cada salto real. Tolerancia a errores aislados y estabiliza la predicción.
 - **Perceptron predictor (ML)**: usa un perceptrón (red neuronal simple) con historial de saltos para aprender patrones complejos; más preciso que métodos clásicos con bajo coste adicional.
 - **TAGE (Tagged Geometric)**: técnica híbrida basada en múltiples historiales de salto; versiones avanzadas pueden combinarse con ML para optimizar pesos y decisiones.
 - **ML avanzado (Deep Learning, SVM, etc.)**: en investigación; muy preciso pero con latencia y coste alto, por lo que aún no se usa en procesadores comerciales.

2.3 Memoria interna

Registros

- Ultrarrápidos y limitados.
- Generales y especiales: PC, IR, FLAGS, MAR/MDR, SP (Stack Pointer)

Caché

- L1: núcleo.
- L2: núcleo o compartida.
- L3: compartida global.
- Técnicas:
 - **Prefetching**: cargar datos en caché antes de que se necesiten para reducir latencia.
 - **Coherencia**: mantener actualizados los datos compartidos en cachés múltiples.
 - **Reemplazo adaptativo**: ajustar dinámicamente qué datos se eliminan de la caché según el uso.

RAM

- Área de trabajo externa.
- DDR5, LPDDR5X según entorno.

2.4 Buses internos

- Datos, direcciones, control.

- Evolución: FSB → QPI, Infinity Fabric, Unified Memory.

3. Funcionamiento de la CPU

3.1 Conjunto de instrucciones

CISC

- Instrucciones complejas.
- Arquitecturas: x86, ARMv8-A.

RISC

- Instrucciones simples, eficientes.
- Ejemplos: ARM, RISC-V.

Extensiones modernas

- AVX-512, VT-x/AMD-V, AMX.
- IA integrada, virtualización nativa.
- RISC-V: abierta, modular, en crecimiento.

3.2 Ciclo de instrucción

- Fases: Fetch → Decode → Execute → Memory Access → Write-back.
- Optimización:
 - Multithreading: Hyper-Threading, SMT.
 - Predicción, ejecución especulativa.
 - Soporte IA en la CPU (Apple Neural Engine, Intel AMX).

4. Conclusión

Las CPU modernas integran paralelismo, vectores, control avanzado, y capacidades IA. Su comprensión es clave para programar sistemas eficientes, optimizar procesos y entender el funcionamiento base de cualquier equipo digital.

PROPUESTA DIDÁCTICA: “SIMULANDO UNA CPU: PROGRAMACIÓN DE UN INTÉRPRETE DE INSTRUCCIONES”

A. Contextualización

- Nivel: 1.º FP Grado Superior DAM o DAW.
- Módulo: Programación.
- Perfil: alumnado con dominio básico de estructuras de control y memoria.

B. Objetivos

- Simular mediante programación el ciclo de instrucción de una CPU.
- Representar digitalmente registros, memoria y operaciones básicas.
- Comprender cómo la CPU gestiona y ejecuta código.

C. Metodología

- Aprendizaje basado en proyectos y resolución de problemas.
- Desarrollo incremental con pruebas y visualización.
- Programación individual o en parejas (Python, Java, C++).

D. Actividad principal

- Crear un simulador básico de CPU que:
 - Interprete un pequeño conjunto de instrucciones (LOAD, ADD, JMP...).
 - Emule registros como PC, AC, IR, FLAGS.
 - Visualice el ciclo completo por consola o GUI.
 - Opcional: interrupciones, subrutinas, multithreading.

E. Atención a la diversidad

- Nivel III: código base, guías, plantillas con instrucciones comentadas.
- Nivel IV: objetivos reducidos, apoyo continuo, evaluación formativa.

F. DUA

- Representación: consola paso a paso, GUI opcional, esquemas.
- Acción/expresión: elección libre de lenguaje y estructura.
- Implicación: retos progresivos, gamificación por funcionalidades.

G. Evaluación

- Rúbricas: ejecución correcta, estructura clara, rigor técnico.
- Instrumentos: revisión de código, presentación oral, demo funcional.

H. Conclusión didáctica

- Programar una CPU permite entender su lógica interna desde el rol de programador.
- Favorece el pensamiento lógico, la abstracción computacional y la transferencia de conocimientos entre hardware y software.

Tema 4. Memoria interna. Tipos. Direccionamiento. Características y funciones.

1. Introducción

- La memoria es fundamental para el rendimiento del sistema: almacena instrucciones y datos, afecta a la velocidad de ejecución y coordina con la CPU.
- Una jerarquía eficiente de memoria evita cuellos de botella y maximiza el rendimiento.

2. Conceptos fundamentales

2.1 Elementos clave

- Soporte físico: silicio (RAM, Flash), magnético (HDD), óptico (CD/DVD).
- Acceso: aleatorio (RAM), secuencial (cintas), asociativo (caché).
- Volatilidad: volátil (RAM), no volátil (Flash, ROM, HDD).

2.2 Direccionamiento

- 2D: decodificador único, memorias pequeñas.
- 3D: múltiples decodificadores, alto rendimiento.

2.3 Características

- Velocidad: latencia y ancho de banda.
- Unidad de transferencia: palabra, bloque, línea.
- Modos de direccionamiento lógico: directo, indirecto, paginado, segmentado.

3. Tipos de memoria

3.1 Volátiles

- SRAM: rápida, cara, sin refresco.
- DRAM: necesita refresco, más densa.
- DDR, GDDR, HBM: sincronizadas, especializadas para GPU o IA.

3.2 No volátiles

- ROM: solo lectura.
- Flash: base de SSD.
- NVRAM: combina velocidad y persistencia.

4. Jerarquía de memoria

- Registros: máxima velocidad, mínima capacidad.
- Caché (L1–L3): latencia reducida.
- RAM: datos activos.
- Almacenamiento (SSD, HDD): persistencia.
- Red/Nube: acceso remoto y respaldo.

5. Conexión CPU–Memoria

5. Conexión CPU–Memoria

5.1 Estructura

- **SRAM (Static RAM)**: almacena datos mediante biestables, sin necesidad de refresco, usada en memorias caché por su alta velocidad.
- **DRAM (Dynamic RAM)**: almacena datos en condensadores que deben refrescarse constantemente, utilizada como RAM principal por su alta densidad.
- **ROM (Read-Only Memory)**: contiene datos permanentes o semipermanentes, accesibles por direccionamiento fijo, usada en firmware.

5.2 Acceso

- **Buses**: la CPU accede a la memoria mediante tres buses: **direcciones** (selección), **datos** (información) y **control** (señales de operación).
- **Modos de acceso**: incluyen **lectura**, **escritura** y **modificación** de datos según el tipo de operación que indique la CPU.

5.3 Técnicas

- **Paginación**: divide la memoria en páginas lógicas y marcos físicos para facilitar la gestión y proteger procesos.
- **Acceso por columnas**: optimiza la velocidad accediendo a bloques contiguos dentro de una misma fila de DRAM.
- **Refresco**: en DRAM, consiste en recargar periódicamente los condensadores para evitar la pérdida de datos.

6. Mejora de rendimiento

6.1 Memoria caché

- **L1–L3**: niveles jerárquicos de caché donde L1 es la más rápida y pequeña por núcleo, y L3 es mayor y compartida por la CPU.
- **Mapeo directo**: cada bloque de memoria principal se asigna a una única línea de caché.
- **Mapeo totalmente asociativo**: cualquier bloque puede ubicarse en cualquier línea de caché.

- **Mapeo por conjuntos asociativos:** la caché se divide en conjuntos donde cada bloque puede ocupar cualquier línea dentro de su conjunto.
- **Reemplazo LRU (Least Recently Used):** se elimina el bloque que no se ha usado en más tiempo.
- **Reemplazo FIFO (First In, First Out):** se reemplaza el bloque que lleva más tiempo en la caché.
- **Reemplazo aleatorio:** se sustituye un bloque elegido al azar dentro del conjunto.

6.2 Memoria virtual

- **MMU** (Unidad de Gestión de Memoria) convierte direcciones virtuales en físicas mediante
 - **Paginación:** divide la memoria en bloques fijos (páginas y marcos), lo que permite asignación no contigua y evita la fragmentación externa.
 - **Segmentación:** organiza la memoria en bloques lógicos (código, datos, pila), respetando la estructura del programa pero con riesgo de fragmentación externa.
 - **Segmentación** paginada: combina ambos modelos dividiendo cada segmento lógico en páginas, optimizando espacio y manteniendo organización lógica.
- **TLB:** caché de traducciones recientes rápida.

7. Tecnologías modernas

- **PMEM (Memoria Persistente):** combina la velocidad de la RAM con persistencia similar al almacenamiento, como Intel Optane.
- **Memoria 3D:** apila celdas de memoria verticalmente para aumentar densidad y reducir latencia.
- **PIM (Processing In Memory):** integra procesamiento dentro del chip de memoria para reducir el traslado de datos en tareas como IA y HPC.

PROPUESTA DIDÁCTICA: “SIMULADOR DE JERARQUÍA DE MEMORIA: PROGRAMANDO ACCESOS, LATENCIAS Y CACHÉ”

A. Contextualización

- Nivel: 2.º DAM o DAW.
- Módulo: Programación o Sistemas Informáticos.
- Perfil: alumnado con nociones de estructuras de datos y control de flujo.

B. Objetivos

- Simular la jerarquía de memorias desde registros hasta almacenamiento.
- Comprender cómo la latencia y las políticas de caché afectan al rendimiento.
- Programar comportamientos reales de sistemas modernos de memoria.

C. Metodología

- Proyecto práctico por parejas o grupos pequeños.
- Enfoque incremental: fases de diseño, implementación, testeo y presentación.
- Lenguajes posibles: Python, Java o C++.

D. Actividad principal

- Desarrollo de un programa que simule:
 - Memorias con distinta latencia y capacidad (registros, caché, RAM, disco).
 - Políticas de reemplazo de caché (LRU, FIFO, aleatorio).
 - Mapeo directo y asociativo por conjuntos.
 - Acceso secuencial y aleatorio a datos simulados
 - Estadísticas: tasa de aciertos/fallos, tiempo medio de acceso.
- Interfaz: consola o simple GUI para ver operaciones y resultados.
- Ampliación opcional: paginación, uso de TLB y acceso virtual a disco.

E. Atención a la diversidad

- Nivel III: código base preconfigurado, ayuda estructurada.
- Nivel IV: simulaciones más básicas con componentes seleccionados.

F. DUA

- Representación: animaciones, diagramas, consola paso a paso.
- Acción: desarrollo libre o guiado, estilos de programación variados.
- Implicación: simulación con ejemplos reales, feedback inmediato.

G. Evaluación

- Rúbricas: precisión técnica, eficiencia del simulador, claridad del código.
- Instrumentos: demo, documentación del diseño, revisión por pares.

H. Conclusión didáctica

- Simular memoria refuerza la comprensión del rendimiento real del software.
- El alumnado conecta teoría, arquitectura y programación, desarrollando visión de optimización y eficiencia.

Tema 5. Microprocesadores. Estructura. Tipos. Comunicación con el exterior.

1. INTRODUCCIÓN

- **Microprocesador:** circuito integrado que actúa como la **CPU (Central Processing Unit)** del sistema.
- Ejecuta instrucciones → controla operaciones del sistema.
- Presente en:
 - **PCs y portátiles**
 - **Sistemas embebidos**
 - **Dispositivos móviles**
 - **Servidores y redes**
- Elemento clave para:
 - Procesamiento de datos, Coordinación del hardware e Interacción con periféricos y memoria

2. ESTRUCTURA INTERNA DEL MICROPROCESADOR

2.1 Unidad de control (Control Unit)

- Decodifica instrucciones desde memoria
- Genera señales para coordinar ALU, registros y buses
- Controla el flujo del programa (secuencial o condicional)

2.2 Unidad aritmético-lógica (ALU, Arithmetic Logic Unit)

- Realiza operaciones:
 - Aritméticas: suma, resta, desplazamientos
 - Lógicas: AND, OR, NOT, XOR
- Puede incluir **FPU (Floating Point Unit)** → cálculos reales

2.3 Registros internos

- Memoria interna ultrarrápida
- Tipos:
 - **PC (Program Counter):** apunta a la próxima instrucción
 - **IR (Instruction Register):** almacena la instrucción actual
 - **SP (Stack Pointer):** gestiona la pila
 - **Registro de estado (FLAGS):** indica resultados (Z=Zero, C=Carry, etc.)

2.4 Buses internos

- Conectan las partes del microprocesador
 - **Bus de datos:** transporta información
 - **Bus de direcciones:** identifica ubicación
 - **Bus de control:** señales de lectura, escritura, interrupciones

2.5 Caché (memoria intermedia de alta velocidad)

- Almacena datos/instrucciones frecuentes
- Niveles:
 - **L1:** muy rápida, por núcleo
 - **L2:** mayor, compartida o por núcleo
 - **L3:** más lenta, compartida entre núcleos

2.6 Elementos avanzados

- **Pipeline:** divide la ejecución de instrucciones en etapas que se procesan en paralelo para aumentar el rendimiento.
- **Out-of-Order Execution:** ejecuta instrucciones fuera de su orden original si sus datos están listos, optimizando el uso de la CPU.
- **Branch Prediction:**
 - **2-bit saturating counter, Perceptron predictor (ML), ML avanzado.**
- **SMT (Simultaneous Multithreading):** permite que un solo núcleo físico ejecute múltiples hilos lógicos al compartir recursos internos.

3. TIPOS DE MICROPROCESADORES

3.1 Según arquitectura (ISA - Instruction Set Architecture)

- **CISC (Complex Instruction Set Computing)**
 - Instrucciones complejas y variadas
 - Ej.: x86 (Intel, AMD)
- **RISC (Reduced Instruction Set Computing)**
 - Instrucciones simples y rápidas
 - Ej.: ARM, RISC-V

3.2 Según número de núcleos

- **Single-core:** un solo núcleo / **Multi-core:** varios núcleos paralelos

3.3 Según aplicación

- **CPU (uso general):** PCs, servidores
- **Microcontroladores (MCU, Microcontroller Unit):** CPU + memoria + periféricos → sistemas embebidos
- **Procesadores de señales digitales (DSP, Digital Signal Processor):** operaciones matemáticas rápidas
- **Procesadores gráficos (GPU, Graphics Processing Unit):** cálculos gráficos y paralelos

4. COMUNICACIÓN CON EL EXTERIOR

4.1 Buses del sistema

- **Bus de datos:** transfiere información
- **Bus de direcciones:** localiza destino/origen
- **Bus de control:** señales (lectura, escritura, interrupción, reloj)

4.2 Modos de acceso

- **Memoria compartida:** CPU y periféricos acceden a RAM
- **E/S mapeada (I/O mapeada):** direcciones reservadas para periféricos
- **DMA (Direct Memory Access):** el periférico transfiere datos directamente a memoria sin usar CPU

4.3 Interrupciones

- **Mecanismo para responder a eventos externos**
- Tipos:
 - **IRQ (Interrupt Request):** interrupciones normales por hardware
 - **NMI (Non-Maskable Interrupt):** no pueden ser ignoradas
- Controladores:
 - **PIC (Programmable Interrupt Controller):** gestiona interrupciones
 - **APIC (Advanced PIC):** en multiprocesadores

5. CICLO DE EJECUCIÓN DE INSTRUCCIONES

1. **Fetch:** buscar instrucción (según PC)
2. **Decode:** decodificar (CU interpreta)
3. **Execute:** ejecutar (ALU o FPU actúan)
4. **Write-back:** guardar resultado
5. **Actualizar PC:** preparar siguiente ciclo

6. JERARQUÍA DE MEMORIA

Nivel	Ubicación	Ejemplo	Acceso (ciclos)
Registros	Dentro del núcleo	PC, IR	1
Caché (L1-L3)	En chip	L1, L2, L3	1–20
RAM	Externa	DDR4/DDR5	100–200
Almacenamiento	Disco / SSD	SSD, HDD	Miles

PROPUESTA DIDÁCTICA

Actividad: Programa una CPU básica - Programación 1º DAM

1. **Contexto:** Actividad inicial para comprender cómo se ejecuta el código. Se simula una CPU.

2. **Actividad:** Escribir programa que simule una CPU.

3. Objetivos:

- Programar una CPU básica (ciclo de instrucción, registros, ALU).
- Entender cómo se interpretan y ejecutan instrucciones.
- Relacionar programación con arquitectura.

4. Metodología:

Trabajo práctico: desarrollo de una simulación por software que interprete instrucciones simples (ej. ADD, SUB, JMP). Se implementa un contador de programa, registros y lógica de control.

5. Evaluación. Instrumentos y criterios.

Funcionalidad del programa, claridad del código y comprensión del proceso simulado.

6. Inclusión, diversidad y DUA

Tema 6. Sistemas de almacenamiento externo. Tipos. Características y funcionamiento.

1. Introducción

- El almacenamiento externo permite persistencia de datos fuera de la RAM.
- Clave para seguridad, portabilidad, rendimiento y escalabilidad del sistema.
- Su evolución ha sido marcada por mejoras en capacidad, velocidad y miniaturización.

2. Clasificación de los sistemas de almacenamiento

2.1. Por tecnología

- **Magnéticos:** HDD, cintas (gran capacidad, acceso secuencial).
- **Ópticos:** CD/DVD/Blu-ray (lectura por láser, bajo coste).
- **Semiconductores:** SSD, USB, SD, eMMC, UFS (sin partes móviles, acceso rápido).

2.2. Por ubicación

- Interno: SATA, M.2 (SATA/NVMe).
- Externo: USB, eSATA, Thunderbolt.
- En red: NAS, SAN, almacenamiento en nube.

2.3. Por tipo de acceso

- Secuencial: cintas magnéticas.
- Aleatorio: SSD, HDD, USB.

3. Comparativa técnica

Tipo	Capacidad	Velocidad	Durabilidad	Coste/GB
HDD	Alta	Media	Media	Bajo
SSD SATA	Alta	Alta	Alta	Medio
SSD NVMe (Non-Volatile Memory Express)	Media	Muy alta	Alta	Medio-Alto
USB / SD	Variable	Baja	Alta	Bajo
Óptico	Baja	Baja	Alta	Muy bajo
Cinta mag.	Muy alta	Muy baja	Muy alta	Muy bajo

4. Funcionamiento interno

4.1. HDD

- Platos giratorios, cabezales móviles.
- Lectura magnética, afectado por latencia.

4.2. SSD

- Memoria NAND, sin partes móviles.
- Tipos de celdas: SLC, MLC, TLC, QLC.
- Protocolos: SATA, NVMe (PCIe).

4.3. Cintas magnéticas

- Acceso secuencial, uso en backups masivos.

4.4. Ópticos

- Lectura por láser, bajo coste, almacenamiento físico.

4.5. Tecnologías complementarias

- **RAID:** rendimiento y redundancia.
- **TRIM:** mantenimiento del rendimiento SSD.

5. Interfaces y conectividad

Interfaz	Velocidad teórica	Aplicación
SATA III	600 MB/s	HDD/SSD internos
USB 3.2	5–10 Gbps	Discos externos

NVMe PCIe	8.000 MB/s	SSD M.2/U.2
Thunderbolt	40 Gbps	Almac. profesional
eSATA	3 Gbps	Conexión SATA ext.

6. Aplicaciones

- **Doméstico:** se combina un SSD interno rápido con un HDD externo para ampliar capacidad y hacer copias de seguridad.
- **Educativo:** se utilizan memorias USB por su portabilidad y almacenamiento en la nube para acceso remoto y colaboración.
- **Empresarial:** se usan NAS para compartir archivos en red local y SAN como red dedicada de alto rendimiento para conectar servidores con almacenamiento centralizado; las cintas se emplean para copias de seguridad a largo plazo.
- **Móviles:** integran eMMC (más lento y secuencial) o UFS (más rápido y paralelo) como almacenamiento interno soldado a placa, y pueden usar microSD como almacenamiento externo extraíble y más lento.

PROPUESTA DIDÁCTICA – Módulo: Programación (CFGS DAM)

1. Contexto

Integración en DAM para comprender la interacción software-hardware.

2. Objetivos

- Simular comportamiento de almacenamiento.
- Aplicar acceso secuencial y aleatorio.
- Programar operaciones básicas: guardar, borrar, listar.

3. Actividad: *Simula un dispositivo de almacenamiento*

- Implementación en Python/Java/C# con estructuras (arrays/listas).
- Funciones:
 - Leer/escribir bloques.
 - Acceso secuencial y aleatorio.
 - Simulación de fragmentación/borrado.
 - Mejora opcional: compresión o acceso paralelo.

4. Metodología.

5. Evaluación

- **Instrumentos:** revisión código, cuestionario técnico, exposición.
- **Criterios:**
 - Relación con conceptos reales.
 - Código estructurado.
 - Claridad en exposición.

5. Inclusión, diversidad y DUA

- **Nivel III:** andamiaje, plantillas de código.
- **Nivel IV:** retos adaptados y soporte extra.
- DUA: variedad de formatos, estrategias personalizadas, roles activos.

6. Conclusión

Actividad competencial que relaciona teoría y práctica, refuerza arquitectura, programación y conciencia hardware-software.

Tema 7. Dispositivos periféricos de entrada/salida. Características y funcionamiento.

1. Introducción

Los dispositivos periféricos de entrada/salida (E/S) constituyen el canal fundamental de interacción entre el sistema informático y el entorno físico. Son clave en el desarrollo de software de propósito general, sistemas embebidos y programación de bajo nivel.

2. Clasificación funcional

2.1. Periféricos de entrada

- Capturan información del entorno o del usuario.
- Ejemplos: teclado, ratón, escáner, sensores, cámara, lector NFC.

2.2. Periféricos de salida

- Devuelven información procesada al usuario o a otros sistemas.
- Ejemplos: pantalla, impresora, impresora 3D, altavoz.

2.3. Periféricos mixtos

- Realizan ambas funciones.
- Ejemplos: pantalla táctil, módem, memoria USB, interfaz háptica.

2.4. Dispositivos menos comunes o especializados

- **Tableta digitalizadora** (entrada): permite dibujar con lápiz digital; usada en diseño gráfico.
- **Actuador lineal** (salida): convierte señales eléctricas en movimiento físico. Robótica.
- **Pantalla háptica** (mixto): devuelve respuesta táctil (vibración o presión) según la interacción.
- **Cámara térmica** (entrada): capta radiación infrarroja para detectar calor; usada en industria.
- **Casco de realidad virtual (VR)** (mixto): combina sensores de posición, acelerómetros y pantallas para simular entornos virtuales.
- **Pantalla Braille dinámica** (salida): representa caracteres Braille físicamente mediante pines móviles para personas con discapacidad visual.
- **Sensor LIDAR** (entrada): mide distancias por láser; usado en topografía.
- **Biosensor** (entrada): mide variables biológicas (temperatura, pulso, glucosa); wearables.

3. Características técnicas

Tipo	Parámetros esenciales
Entrada	Resolución, tasa de muestreo, latencia, tecnología de sensor
Salida	Velocidad de respuesta, resolución, fidelidad de señal
Mixtos	Tasa de transferencia bidireccional, sincronización, compatibilidad

- Interfaces: PS/2, USB 2.0/3.x, HDMI, SATA, I2C, SPI.
- Modos de transmisión: síncrono/asíncrono, serie/paralelo.

4. Arquitectura de funcionamiento

4.1. Elementos comunes

- **Controlador (driver)**: software intermedio que traduce instrucciones del sistema operativo al lenguaje del dispositivo.
- **Registro de control y buffer de datos** en el hardware.

4.2. Entrada

- Estímulo físico → señal analógica → ADC → digitalización → buffer → CPU.
- Ej.: micrófono → señal eléctrica → ADC → datos de audio.

4.3. Salida

- CPU → datos digitales → DAC → señal analógica → actuador.
- Ej.: imagen digital → DAC → señal HDMI → monitor.

4.4. E/S programada vs por interrupciones vs DMA

- **Programada**: CPU consulta continuamente el periférico (ineficiente).
- **Interrupciones**: el periférico notifica a la CPU cuando requiere atención; pueden ser enmascarables (la CPU puede ignorarlas temporalmente) o no enmascarables.
- **DMA (acceso directo a memoria)**: transfiere datos sin intervención de la CPU.

5. Protocolos e interfaces

Interfaz	Tipo	Aplicación frecuente
USB	Serie	Teclado, ratón, almacenamiento
HDMI/VGA	Vídeo	Salida de vídeo

Bluetooth	Inalámbrico	Auriculares, periféricos móviles
I2C/SPI	Embebido	Sensores, microcontroladores
PCIe	Alta velocidad interna	Tarjetas gráficas, capturadoras

6. Gestión desde software

6.1. Llamadas al sistema

- Acceso a dispositivos como archivos especiales (`/dev`) en sistemas UNIX/Linux.
- Funciones típicas: `open()`, `read()`, `write()`, `ioctl()`.

6.2. Buffers de E/S

- Memoria intermedia que permite desacoplar velocidades de CPU y periféricos.
- Tipos:
 - **FIFO**: cola donde el primer dato en entrar es el primero en salir (First In, First Out).
 - **Cola circular**: buffer que se reutiliza en forma circular para optimizar espacio.
 - **Doble búfer**: se alternan dos áreas de memoria para lectura y escritura sin interferencias.
 - **Triple búfer**: añade un tercer búfer para minimizar esperas en sistemas con gráficos o vídeo.

6.3. Gestión de eventos

- Arquitecturas reactivas: `event listeners`, `callbacks`.
- Uso extendido en interfaces gráficas (GUIs), dispositivos interactivos y juegos.

6.4. Caching

- Memoria rápida entre periférico y sistema.
- Minimiza accesos repetidos a E/S costosa.
- **Técnicas**: **write-through** (escritura inmediata en caché y dispositivo) y **write-back** (escritura diferida desde caché al dispositivo).
- Importante en discos, sistemas multimedia, impresión.

6.5. Modelado orientado a objetos

- Abstracción de periféricos como clases.
- Enfoque útil en simulaciones, frameworks de E/S y arquitectura software modular.

PROPUESTA DIDÁCTICA – MÓDULO: PROGRAMACIÓN (CFGs DAM)

1. Contexto

Simulación y programación de periféricos como proyecto integrado. Permite afianzar conceptos de estructuras de datos, eventos y orientación a objetos.

2. Objetivos

- Comprender la abstracción software de un periférico.
- Programar su lógica interna usando clases y eventos.
- Implementar buffers, interrupciones simuladas y estados.

3. Actividad principal: “Simula un periférico interactivo”

Tarea: desarrollar una aplicación que simule dispositivos de E/S.

- Clases `Teclado`, `Ratón`, `Pantalla`, `Impresora`, `USB`, `PantallaTáctil`.
- Simulación de flujo de datos, estados, interrupciones, colas de eventos.
- Implementación de errores comunes: desconexión, latencia, sobrecarga.

4. Metodología

- Trabajo cooperativo en parejas.
- Diseño incremental con pruebas intermedias.
- Integración final de varios periféricos en una aplicación unificada.

5. Atención a la diversidad y DUA

- **Nivel III**: materiales guiados, soporte visual, pseudocódigo.
- **Nivel IV**: plantillas de clases, descomposición de tareas.
- DUA:
 - Representación: diagramas, vídeos, texto.
 - Acción y expresión: código, simulación visual, presentación oral.
 - Implicación: elección del periférico a simular.

6. Evaluación

Instrumentos: rúbrica, control del código, autoevaluación.

Criterios:

- Correcta simulación funcional del periférico.
- Aplicación rigurosa de conceptos de E/S.
- Código estructurado y comentado.

Tema 8. Hardware comercial de un ordenador. Placa base. Tarjetas controladoras de dispositivos y de entrada/salida.

1. Introducción

El hardware comercial se refiere a componentes físicos estandarizados (placa base, tarjetas, memoria, almacenamiento, etc.) que conforman un ordenador. Facilitan el ensamblaje modular, la compatibilidad entre fabricantes y generaciones, y la evolución tecnológica. Para el programador, entender la arquitectura física permite optimizar recursos, adaptar código y anticipar limitaciones del sistema.

2. Placa base

2.1 Función

Es el elemento estructural que integra y comunica todos los componentes del sistema: CPU, RAM, almacenamiento, tarjetas de expansión y periféricos.

2.2 Componentes principales

- **Socket CPU:** LGA 1700 (Intel), AM5 (AMD); define compatibilidad física y eléctrica.
- **Chipset (PCH):** gestiona periféricos; el puente norte está integrado en la CPU moderna. Ejemplos: Z790 (DDR4/DDR5, PCIe 5.0), B650 (DDR5, PCIe 5.0).
- **Buses internos:** DMI (Intel) e Infinity Fabric (AMD); determinan la velocidad entre CPU y otros componentes.
- **RAM:** Slots DDR4/DDR5 con soporte dual/triple channel; norma JEDEC.
- **UEFI/BIOS:** firmware configurable, interfaz gráfica, arranque seguro.
- **VRM:** regula tensión a CPU y RAM; clave en rendimiento y estabilidad.
- **Puertos de expansión:** PCIe x1/x4/x8/x16 (versión 3.0 a 5.0, hasta 4 GB/s por línea).
- **Conectores internos:** SATA III (6 Gb/s), M.2/U.2 (NVMe); soporte RAID.
- **Puertos externos:** USB 2.0/3.x/USB-C, HDMI, DisplayPort, RJ-45, audio jack/óptico.

2.3 Factores de forma

Formato	Tamaño	Aplicación
ATX	305×244 mm	Sobremesa estándar
microATX	244×244 mm	Equipos compactos
Mini-ITX	170×170 mm	HTPC, sistemas reducidos

3. Tarjetas controladoras

3.1 Función

Extienden o especializan funciones del sistema: gráficos, red, audio, almacenamiento, captura.

3.2 Tipos

- **GPU:** PCIe x16, GDDR6/6X, APIs (DirectX, Vulkan); núcleos CUDA/Stream.
- **Sonido:** PCIe x1, DAC/ADC dedicados; mejoran calidad y latencia.
- **NIC:** Ethernet 1/2.5/10 GbE, Wi-Fi 6/6E, BT; funciones Wake-on-LAN, VLAN.
- **Almacenamiento:** controladoras RAID (0,1,5,10), SAS, NVMe.
- **Captura:** Full HD/4K, HDMI loop, bitrate 120 Mbps, codecs H.264/H.265.

3.3 Compatibilidad

- Espacio físico (longitud, disipadores).
- Alimentación (6/8/12 pines PCIe).
- Versión PCIe compatible con placa y CPU.

4. Normas y estándares

Norma	Regula
JEDEC	DDR4/DDR5
PCI-SIG	PCIe 3.0–5.0
SATA-IO	Interfaz y velocidades SATA
ATX	Dimensiones de placas y fuentes
USB-IF	Protocolos USB (2.0 a USB4)

5. Relación con software

- **Procesamiento:** influencia de CPU, RAM, buses.
- **Acceso a datos:** SATA vs. NVMe.
- **Carga gráfica:** GPU dedicada vs. integrada.
- **Conectividad:** NIC/Wi-Fi.
- **Virtualización y paralelismo:** compatibilidad según arquitectura.
-

6. Cuellos de botella

6.1 Qué es

- Componente que limita el rendimiento global del sistema.
- Actúa como “tapón” en la cadena de procesamiento.

6.2 Casos típicos

- **CPU limitada:** el resto del sistema es potente pero la CPU no rinde.
- **GPU limitada:** la CPU va sobrada pero los gráficos no fluyen.
- **RAM lenta/escasa:** el sistema usa disco como memoria (muy lento).
- **Almacenamiento lento:** sobre todo con HDD en tareas de acceso intensivo.
- **Buses saturados:** PCIe antiguo, DMI colapsado.
- **Red/I/O lenta:** conexiones lentas limitan transferencias o virtualización.

6.3 Cómo detectarlo y solucionarlo

- Monitorización: Task Manager, HWInfo, benchmarks.
- Equilibrio en la configuración de componentes.
- Actualización dirigida (solo el componente cuello).
- Optimización de software (menos acceso a disco, más paralelismo, compresión, buffering, etc.).

Conclusión técnica

Conocer el hardware comercial capacita al programador para optimizar código, adaptar soluciones a distintos entornos físicos (gaming, oficina, servidores), y prever cuellos de botella. Este conocimiento técnico refuerza la estabilidad, eficiencia y compatibilidad del software profesional.

PROPUESTA DIDÁCTICA – MÓDULO DE PROGRAMACIÓN (CFGs DAM)

1. Contextualización

Relaciona arquitectura hardware con programación orientada a objetos. El alumnado modela un ordenador simulando componentes físicos y su interacción, favoreciendo la comprensión sistémica.

2. Objetivos

- Representar estructuras hardware mediante clases.
- Simular ejecución y flujo de datos.
- Analizar impacto del hardware en el rendimiento del software.

3. Metodología

- Programación por parejas de clases **CPU**, **RAM**, **PlacaBase**, **GPU**, **NIC**.
- Simulación, reflexión y adaptación del código a diferentes configuraciones.

4. Atención a la diversidad y DUA

- **Nivel III:** plantillas y ayudas visuales, guía paso a paso.
- **Nivel IV:** clases predefinidas, comentarios explicativos.
- **DUA:**
 - Entrada múltiple (texto, visual, funcional).
 - Roles diferenciados: modelador, codificador, tester.
 - Productos variados: código, presentación, demo.

5. Actividad principal: “Simula un sistema físico”

- Desarrollo de un programa OO que modela un sistema real.
- Métodos como **procesar()**, **cargarDatos()**, **mostrarInfo()**, **conectar()**.
- Escenarios variables: oficina, servidor, multimedia.
- Integración opcional de **psutil** o **System.getProperty()** para comparar simulación y sistema real.

6. Evaluación

- **Instrumentos:** código funcional, demo explicativa, cuestionario técnico.
- **Criterios:**
 - Coherencia del modelo hardware.
 - Eficiencia, claridad y documentación del código.
 - Relación hardware-software bien fundamentada.

7. Conclusión didáctica

Tema 10. Representación interna de los datos.

1. Introducción

- Toda información digital se representa internamente en binario (base 2).
- También se utilizan otras bases para facilitar tareas específicas:
 - Octal (8): sintaxis compacta.
 - Decimal (10): interfaz humana.
 - Hexadecimal (16): dirección de memoria, colores, instrucciones.
- Comprender estas representaciones es esencial en programación, redes y sistemas.

2. Representación de caracteres

- **ASCII**: estándar de 7 u 8 bits.
- **UNICODE (UTF-8, UTF-16)**: codifica miles de símbolos, compatible con la web.
- Conversión texto ↔ binario esencial en programación, bases de datos y redes.

3. Representación de booleanos

- 1 bit: 0 = falso, 1 = verdadero.
- Usos en condiciones, lógica digital, estructuras de control.
- Aplicación en puertas lógicas y simplificación con mapas de Karnaugh.

4. Representación de números enteros

- **Signo y magnitud, CA1, CA2 (complemento a 2)**: estándar en programación.
- **Exceso-Z**: usado en representación de exponentes (coma flotante).

5. Representación de reales

- **Coma fija**: poco precisa.
- **Coma flotante (IEEE 754)**: 32/64/128 bits.
- Componentes: signo, exponente, mantisa.
- Problemas comunes: redondeo, desbordamientos.

6. Números complejos

- Dos flotantes: parte real + imaginaria. Representado con estructura, objeto, etc.
- Usos: audio, señales, simulación física, computación cuántica.

7. Representación de estructuras

7.1 Lineales

- Vectores, matrices, listas enlazadas: estructuras fundamentales.

7.2 Jerárquicas y grafos

- Árboles: Árboles: BST (búsqueda binaria), AVL (balanceo estricto), B+ (claves en hojas, ideal para BBDD), R-B (rojo-negro, balanceo por colores).
- Grafos: listas/matrices de adyacencia.
- Aplicaciones en rutas, grafos sociales, IA.

7.3 Tablas hash y punteros

- Acceso $O(1)$, gestión dinámica de memoria.
- Punteros: manipulación directa de direcciones (C/C++).

8. Multimedia y datos complejos

Imagen

- Raster vs. vectorial.
- Compresión con y sin pérdida (JPEG, PNG).

Sonido y vídeo

- Muestreo, resolución, formatos (MP3, FLAC, MP4, H.265).
- Codificación por frames (compresión intraframe e interframe), códecs para streaming.

3D

- Modelado por vértices, formatos: OBJ, GLTF.
- Aplicaciones: videojuegos, simulación física, realidad aumentada.

9. Seguridad y compresión

Cifrado

- Simétrico, asimétrico, combinación de ambas
- AES, RSA, ECC.
- Criptografía post-cuántica en desarrollo.

Compresión

- ZIP, PNG (sin pérdida).

- MP3, JPEG (con pérdida).

Hash

- SHA, MD5.
- Aplicaciones: autenticación, integridad, búsqueda.

PROPUESTA DIDÁCTICA: “CREA TU CONVERTOR BINARIO MULTI-TIPO: EL ORDENADOR DESDE DENTRO”

A. Contextualización

- Nivel: 1.º DAM o DAW.
- Módulo: Programación.
- Perfil: alumnado con competencias en codificación, estructuras de datos y desarrollo de interfaces.

B. Objetivos

- Simular mediante programación la conversión binaria de diversos tipos de datos.
- Visualizar estructuras internas y codificación real.
- Integrar teoría de representación con desarrollo de software funcional.

C. Metodología

- Aprendizaje basado en proyectos.
- Desarrollo individual o en parejas.
- Iteración por funcionalidades, pruebas y presentación.

D. Actividad principal

Proyecto: Programa “BinarioTotal”

- Desarrollo de una aplicación que permita:
 - **Codificar/decodificar:**
 - Caracteres (ASCII, UTF-8).
 - Enteros (CA2).
 - Reales (IEEE 754).
 - Booleanos.
 - **Visualizar:**
 - Codificación interna en binario y hexadecimal.
 - Comparación entre formatos.
 - **Simular estructuras:**
 - Arrays, listas enlazadas, árboles (con impresión en memoria).
 - **Extra** (opcional):
 - Codificar imágenes o sonidos simples.
 - Comprimir o cifrar una cadena o archivo.
- Lenguajes recomendados: Python (Tkinter), Java (Swing/FX), C++ (CLI/GUI simple).
- Interfaz: consola o ventana gráfica básica.

E. Atención a la diversidad

- Nivel III: plantillas base, guía paso a paso.
- Nivel IV: estructura modular, evaluación progresiva, refuerzo individual.

F. DUA

- Representación: visualización binaria, esquemas gráficos, interfaces.
- Expresión: código, documentación, presentación del proyecto.
- Implicación: simulador personalizado, trabajo en equipo, reto final.

G. Evaluación

- Rúbricas: exactitud técnica, calidad del código, visualización, documentación.
- Instrumentos: pruebas funcionales, demo, defensa oral.

H. Conclusión didáctica

- El alumnado transforma la abstracción binaria en lógica aplicada.
- Se potencia la comprensión de la arquitectura digital desde el desarrollo de software

Tema 11. Organización lógica de los datos. Estructuras estáticas.

1. Introducción

- Organización lógica: definición abstracta de cómo se agrupan, relacionan y manipulan los datos sin referirse a su almacenamiento físico.
- Importancia: permite diseñar algoritmos eficientes, reutilizables y orientados a tipos.
- Abstracción: separación entre interfaz lógica y estructura física.

2. Organización lógica de los datos

- Es la forma abstracta de estructurar y acceder a los datos independientemente del hardware o del almacenamiento físico.
- Permite representar información de forma coherente, optimizar el uso de memoria y definir algoritmos eficientes.
- Ejemplos: pilas, colas, listas, árboles, grafos, tablas hash.
- Clave en la programación estructurada, orientada a objetos y funcional.

3. Tipos Abstractos de Datos (TAD)

- Un TAD define: conjunto de valores, operaciones permitidas y sus propiedades semánticas.
- Ejemplos:
 - **Pila (Stack):** LIFO.
 - **Cola (Queue):** FIFO.
 - **Lista:** secuencia ordenada con inserción/borrado dinámico.
 - **Árbol:** estructura jerárquica padre-hijo.
 - **Grafo:** nodos y aristas, relaciones complejas.
 - **Tabla hash:** acceso $O(1)$ mediante clave.

4. Tipos de datos escalares

- **Entero:** complemento a 2.
- **Real:** IEEE 754 (simple/doble precisión).
- **Carácter:** Unicode (UTF-8, UTF-16).
- **Booleano:** 0/1.
- **Enumeración:** valores cerrados.
- **Rango:** subset de un tipo base.

5. Tipos de datos estructurados

5.1 Vectores

- Arrays unidimensionales o multidimensionales para datos indexados.

5.2 Conjuntos

- Manejo de elementos únicos; operaciones: unión, intersección, diferencia.

5.3 Registros y tuplas

- Agrupan datos heterogéneos; equivalente a `struct` en C/C++.

6. Implementación estática

6.1 Pilas

- Array + puntero. Operaciones: `push()`, `pop()`, `top()`. Usos: llamadas, backtracking.

6.2 Colas y deque

- Array circular; en deque se permite inserción por ambos extremos. Usos: gestión de procesos, buffers.

6.3 Listas

- Estáticas en array o dinámicas enlazadas; arrays: acceso $O(1)$, inserciones costosas; listas: acceso secuencial $O(n)$, inserciones eficientes $O(1)$.

6.4 Árboles

- **BST:** elemento izquierdo < nodo < elemento derecho; búsqueda en $O(\log n)$, se puede desacoplar.
- **AVL, B+, R-B:** balanceados automáticamente, garantizan $O(\log n)$.
- **Heap:** árbol completo implementado en array; usado en colas de prioridad, heapsort.

6.5 Grafos

- Representación por lista de adyacencia (espacio eficiente) o matriz (acceso rápido). Aplicaciones: rutas, IA, topología.

6.6 Tabla hash

- Array indexado vía función hash. Colisiones: encadenamiento o direccionamiento abierto. Usos: caché, bases de datos, almacenamiento rápido, contraseñas, tablas rainbow.

6.7 Diccionarios

- También llamados HashMap. Par clave/valor. Utilizan una tabla de Hash internamente.

7. Utilidad en programación y concursos

- Las estructuras estáticas son eficientes y predecibles.
- Su estudio es esencial para entrevistas técnicas, competiciones (OIE, ProgramaMe).
- Plataformas de práctica: Codeforces, LeetCode, AtCoder, HackerRank.

PROPUESTA DIDÁCTICA: “DISEÑA Y DEFIENDE TU ESTRUCTURA: IMPLEMENTACIÓN EN PROGRAMA”

A. Contextualización

- Nivel: 1.º DAM o DAW. Módulo: Programación.
- Perfil: alumnado con dominio de estructuras de control y programación modular.

B. Objetivos

- Identificar el TAD óptimo para resolver un problema real.
- Implementar y defender su estructura mediante código funcional.
- Relacionar su elección con eficiencia y uso de memoria.

C. Metodología

- Trabajo en grupo, enfoque práctico.
- Pasos: análisis → diseño lógico → implementación estática → presentación.

D. Actividad principal

1. Cada grupo selecciona un problema aplicable a una estructura estática (ej. historial, planificación, rutas).
2. Justifica la elección de la estructura (pila, cola, árbol, grafo, hash).
3. Implementa la estructura usando arrays (estática):
 - **Stack**: array + tope.
 - **Queue**: array circular.
 - **BST/AVL/B+**: nodos y punteros.
 - **Heap**: array con operaciones de inserción/eliminación.
 - **Hash Table**: array + función hash + manejo de colisiones.
4. Diseña funciones básicas (insertar, buscar, eliminar, recorrer).
5. Prepara un breve informe en pseudocódigo y organigrama visual.
6. Defiende la solución en una exposición oral, explicando eficiencia y cómo resuelve el problema.
7. Opcional: analiza casos peores y discute mejoras posibles.

E. Atención a la diversidad

- Nivel III: plantillas de estructura y operaciones básicas.
- Nivel IV: planificación por fases, tutoría, roles técnicos dentro del grupo.

F. DUA

- Representación: pseudocódigo, diagramas, esquemas y mapas mentales.
- Expresión: implementación en código, documentación, exposición oral.
- Implicación: cada grupo contextualiza la solución, promueve el debate técnico.

G. Evaluación

- Rúbricas:
 1. Adecuación de la estructura al problema.
 2. Correcta implementación y documentación.
 3. Claridad en la presentación: complejidad, diseño, justificación.
- Instrumentos: observación, demo funcional, revisión del código y autoevaluación.

H. Conclusión didáctica

- Integrar teoría de TADs con implementación estática favorece la comprensión profunda de la programación eficiente.
- El alumnado desarrolla habilidades lógicas, sintácticas y de argumentación técnica.
- La actividad conecta diseño, código y defensa profesional, preparándolos para retos tecnológicos reales.

Tema 12. Organización lógica de los datos. Estructuras dinámicas.

1. Introducción

- Las estructuras dinámicas permiten gestionar datos sin tamaño fijo, adaptándose al crecimiento o reducción en tiempo de ejecución.
- Utilizan punteros/referencias para enlazar nodos en memoria y habilitan inserciones y borrados eficientes.

2. Organización lógica de los datos

- Es la forma abstracta de estructurar y acceder a los datos independientemente del hardware o del almacenamiento físico.
- Permite representar información de forma coherente, optimizar el uso de memoria y definir algoritmos eficientes.
- **Ejemplos:** pilas, colas, listas, árboles, grafos, tablas hash.

3. Fundamentos y características

3.1 Características clave

- Asignación de memoria en tiempo real.
- Flexibilidad estructural mediante enlaces.
- Uso intensivo de punteros o referencias.

3.2 Ventajas e inconvenientes

- **Ventajas:** eficiente en operaciones frecuentes de inserción/borrado.
- **Inconvenientes:** más complejas de implementar, acceso más lento, requerimiento de gestión de memoria manual o automática.

4. Listas dinámicas

4.1 Listas enlazadas simples

- Nodo: dato + puntero al siguiente.
- Operaciones: insertar, eliminar, buscar, recorrer.
- Ideal para estructuras flexibles.

4.2 Listas doblemente enlazadas

- Punteros a siguiente y anterior.
- Navegación bidireccional.
- Base para deques o editores de texto.

4.3 Listas circulares

- El último nodo apunta al primero.
- Útiles en estructuras cíclicas o buffers circulares.

5. Pilas y colas dinámicas

5.1 Pilas (Stack)

- Implementadas con listas.
- Modelo LIFO: operaciones **push()**, **pop()**, **top()**.
- Aplicaciones: backtracking, llamadas recursivas.

5.2 Colas (Queue)

- FIFO: operaciones **enqueue()**, **dequeue()**, **front()**.
- Comunes en planificación de tareas.

4.3 Deques

- Inserción y eliminación por ambos extremos.
- Versátiles para estructuras flexibles.

5. Árboles dinámicos

5.1 Árbol binario

- Nodo con máximo dos hijos.
- Recorridos: inorden, preorden, postorden.
- Útil en búsqueda y expresión de datos.

5.2 BST

- Nodo izquierdo < nodo < nodo derecho.
- Búsqueda eficiente (si el árbol está equilibrado).

5.3 Árboles balanceados

- **AVL y R-B (rojo-negro):** mantienen equilibrio eficiente.
- Mejora de rendimiento en inserciones y búsquedas.

5.4 Árboles n-arios

- Más de dos hijos por nodo.
- Adecuados para DOM, jerarquías, expresiones múltiples.

5.5 Heap (montículo)

- Árbol binario completo.

- **Max-heap:** padres \geq hijos;
 - **Min-heap:** padres \leq hijos.
 - Implementados en arrays (índices: $2i+1$, $2i+2$).
 - Aplicaciones: colas de prioridad, heapsort.
- 6. Grafos dinámicos**
- 6.1 Listas de adyacencia**
- Cada nodo mantiene lista de vecinos.
 - Eficiente para grafos dispersos.
- 6.2 Nodos enlazados**
- Representan vértices con listas de aristas; permiten grafos dirigidos, ponderados.
- 6.3 Aplicaciones**
- Redes de comunicación, rutas, redes sociales, algoritmos de IA (Dijkstra, A*).
- 7. Tablas hash con listas encadenadas**
- Utilizan array + función hash. Colisiones resueltas mediante listas enlazadas. Útiles en diccionarios, caches, bases de datos.
- 8. Gestión dinámica de memoria**
- 8.1 Asignación y liberación**
- En C/C++: **malloc/free**, **new/delete**.
 - En Java/Python: recolector de basura automático.
- 8.2 Problemas comunes**
- **Memory leak:** fallos en liberación.
 - **Doble liberación:** errores críticos.
 - **Fragmentación:** ineficiencia en uso de memoria.
- 9. Aplicaciones reales**
- SO: colas de procesos. Compiladores: AST dinámicos.
 - Editores: estructura de líneas. Juegos/simulación: IA y comportamiento dinámico.
 - Bases de datos: índices en árboles B/B+.
- 10. Conclusión**
- Las estructuras dinámicas son esenciales para programas adaptables y funcionales.
 - Su implementación requiere comprensión de punteros, memoria y eficiencia algorítmica.
 - Son fundamentales para sistemas escalables y seguros.
- PROPUESTA DIDÁCTICA: “SIMULA UNA ESTRUCTURA VIVA: PROGRAMA CON NODOS”**
- A. Contextualización**
- Nivel: 1.º DAM o DAW. Módulo: Programación.
 - Perfil: alumnado con experiencia en estructuras de control y manejo de memoria.
- B. Objetivos**
- Implementar una estructura dinámica en código (lista, árbol, cola, grafo).
 - Visualizar su comportamiento (inserción, eliminación, recorrido).
 - Relacionar teoría con ejecución dinámica real.
- C. Metodología**
- Trabajo por parejas o grupos pequeños.
 - Enfoque por fases: elección, diseño, implementación, visualización y exposición.
 - Lenguaje recomendado: Java, C++, Python (con GUI opcional).
- D. Actividad principal**
1. Seleccionar un caso real (playlist, cola, jerarquía, red).
 2. Diseñar la estructura lógica y la interfaz textual o gráfica.
 3. Implementar nodos con punteros/referencias y las operaciones básicas:
 - Listas: insertar, eliminar, buscar, recorrer.
 - Árbol: insertar, buscar, balancear, mostrar ordenamientos.
 - Grafos: añadir vértices/aristas, recorrido BFS/DFS.
 4. Crear visualización: consola con pasos detallados o GUI sencilla (Tkinter o JavaFX).
 5. Presentar ejemplos: cómo cambia la estructura al ejecutar operaciones.
 6. Defensoría oral: eficiencia, ventajas y posibles mejoras.
- E. Atención a la diversidad**
- Nivel III: estructura base proporcionada, operaciones guiadas.
 - Nivel IV: planificación por fases, práctica más autónoma.
- F. DUA**
- Representación: diagramas, código interactivo, visualizador de nodos.
 - Acción: desarrollo personalizado, elección de casos de uso.
 - Implicación: proyecto significativo, defensa colaborativa.

Tema 13. Ficheros. Tipos. Características. Organizaciones.

1.1 Introducción

- El fichero es la unidad básica de almacenamiento digital.
- Encapsula datos organizados, esenciales para interoperabilidad, seguridad y eficiencia del sistema.
- Ejemplos actuales: ficheros de configuración, logs, multimedia, backups, modelos IA.

1.2 Estructura lógica

- **Campo:** unidad mínima (e.g., "precio").
- **Registro:** conjunto coherente de campos (e.g., ficha de producto).
- **Fichero:** colección organizada de registros.
- Ilustración visual: nombre, precio, categoría → Registro = "Ratón", 25€, "Electrónica".

1.3 Tipos de ficheros

- La extensión no garantiza el tipo real: se usa el *magic number* o encabezado binario para validarlo.
- **A. Por codificación**
 - **Texto:** legibles por humanos (ASCII/Unicode). Ej.: `.txt`, `.csv`, `.html`, `.json`
 - **Binario:** no legibles directamente; compactos, eficientes. Ej.: `.exe`, `.jpg`, `.mp3`, `.class`
- **B. Por función**
 - **Ejecutables:** contienen instrucciones para el sistema. Ej.: `.exe`, `.jar`, `.out`
 - **Configuración:** parámetros de usuario o sistema. Ej.: `.ini`, `.yaml`, `.conf`, `.env`
 - **Datos estructurados:** información organizada. Ej.: `.csv`, `.json`, `.xml`, `.db`
 - **Registros (logs):** trazas de actividad. Ej.: `.log`, `.audit`
 - **Scripts y código fuente:** instrucciones interpretables. Ej.: `.py`, `.sh`, `.sql`, `.ps1`
- **C. Por aplicación**
 - **Multimedia:** imagen, audio, vídeo (comprimidos o no). Ej.: `.mp4`, `.png`, `.mp3`, `.flac`
 - **Documentos:** texto + formato + recursos. Ej.: `.pdf`, `.docx`, `.odt`
 - **Intercambio/portabilidad:** agrupan o empaquetan archivos. Ej.: `.zip`, `.tar.gz`, `.iso`
 - **Técnico/científico:** datos especializados. Ej.: `.mat`, `.stl`, `.gltf`, `.nc`, `.gguf` (IA)
- **D. Contenedores**
 - Agrupan varios tipos (datos + metadatos) Ej.: `.mkv`, `.mp4`, `.apk`, `.docx`

1.4 Características

- **Nombre/extensión,** tamaño lógico/físico, **metadatos** (timestamps, permisos).
- Ubicación (ruta), atributos: cifrado, inmutabilidad, enlaces.
- Sistemas de archivos, información de integridad, accesos concurrentes, dispositivos.
- Enlaces duros y enlaces simbólicos.

1.5 Organización física

Organización	Acceso	Rendimiento	Usos típicos
Secuencial	Lineal	Bajo	Logs
Indexada	Parcial	Medio	Búsquedas
Directa (hash)	Directo	Alto	OLTP, caches
Encadenada	Dinámico	Irregular	Blockchain

1.6 Sistemas de archivos

- **Seguridad:** NTFS (ACL, EFS), APFS, ZFS.
- **Rendimiento:** EXT4, XFS, tmpfs.
- **Distribuidos/Cloud:** S3, HDFS, CephFS.
- **Compatibilidad:** FAT32, exFAT.

1.7 Aplicaciones actuales

- OS: logs, arranque, configuración.
- Bases de datos: ficheros de índices/tablas.
- Web/Móviles: multimedia, JSON.
- IoT, DevOps, IA: binarios, modelos versionados (`.pkl`, `.onnx`).

1.8 Tendencias

- Cifrado por defecto (BitLocker, LUKS).
- Clasificación inteligente por IA.

- Cloud-native: sincronización, control de versiones.
- Optimización (S3 lifecycle, edge caching).

1.9 Seguridad y versiones

- Cifrado simétrico (AES-256), asimétrico (PGP).
- Control de versiones: Git, Git LFS, MLflow.
- Ejemplos: backups cifrados, scripts versionados.

1.10 Ficheros en la nube

- Plataformas: Google Drive, S3, Azure Blob.
- Características: accesibilidad, sincronización, versión.
- Riesgos: pérdida de control, mitigación con cifrado y MFA.

1.11 Conclusión

- Los ficheros siguen siendo eje esencial en el almacenamiento moderno.
- Su gestión eficaz garantiza sistemas robustos, seguros y adaptables.

2. PARTE DIDÁCTICA

2.1 Contextualización

- **Nivel:** CFGM 2º SMR **Módulo:** Servicios en red.
- **Perfil:** alumnado con conocimientos básicos en redes y administración, orientación práctica y profesional.

2.2 Objetivos de aprendizaje

- Comprender el papel de los ficheros y su organización lógica/física en un servicio de red.
- Instalar y configurar un servidor FTP (vsftpd) y SFTP en Linux.
- Diseñar una estructura de carpetas con tipos de ficheros diferenciados (configuración, datos, logs, backups).
- Asignar correctamente permisos, rutas, usuarios y cifrado según la finalidad del fichero.
- Valorar la importancia de la seguridad, integridad y control de acceso en el intercambio remoto de ficheros.

2.3 Metodología

- **Aprendizaje basado en tareas:** montaje y prueba de un servidor real.
- **Trabajo por proyectos:** simulación de un entorno empresarial.
- Aprendizaje colaborativo: roles técnicos en grupo (admin, usuario, auditor).

2.4 Atención a la diversidad (niveles III y IV)

- Desdoblamiento del grupo para tareas prácticas.
- Rúbricas con niveles de logro diferenciados.
- Apoyos visuales y guías paso a paso.
- Tiempo flexible en ejecución de prácticas.

2.5 DUA (Diseño Universal para el Aprendizaje)

- Representación: videotutoriales, esquemas, consola práctica.
- Acción: demostración por CLI y GUI (e.g., FileZilla).
- Motivación: práctica conectada con casos reales de empresas.

2.6 Actividad principal

Supuesto didáctico: "Implementación de un servidor FTP/SFTP seguro para una pequeña empresa"

- Como técnico de redes en una pequeña empresa, debes instalar y configurar un servidor FTP/SFTP en un sistema Linux para ofrecer acceso remoto a los ficheros de varios departamentos (Administración, Comercial, Técnica).
- Cada departamento tiene distintos tipos de ficheros (configuración, datos, logs, backups, multimedia o IA) y necesita una estructura lógica de carpetas, con permisos diferenciados, usuarios aislados, y acceso cifrado mediante SFTP.

2.7 Evaluación

- **Instrumentos:** rúbricas, observación directa, test técnico.
- **Criterios:**
 - Instalación y configuración funcional.
 - Uso correcto de rutas, permisos y cifrado.
 - Documentación técnica de configuración.
 - Resolución de incidencias simuladas.

2.8 Conclusión didáctica

- Comprender los ficheros como elementos clave en servicios de red forma parte del perfil técnico del alumnado SMR.
- La práctica con FTP/SFTP refuerza competencias en seguridad, organización de datos y administración básica de sistemas, aplicables directamente en el entorno laboral.

Tema 14. Utilización de ficheros según su organización.

1. Introducción

- Un fichero es una unidad lógica de almacenamiento en memoria secundaria.
- Su **organización física y lógica** determina cómo se almacenan, acceden y gestionan los datos.
- Una buena organización mejora el **rendimiento**, la **seguridad**, la **accesibilidad remota** y la **eficiencia del sistema**.
- En entornos de red, los ficheros organizados correctamente garantizan integridad y control.

2. Organización física vs lógica

◆ Organización física

- **Qué es:** cómo se almacena el fichero en el disco (bloques, sectores).
- **Quién la gestiona:** sistema operativo, administrador.
- **Ejemplos:** fragmentación en **ext4**, **NTFS**, estructuras **FAT/inodo**.
- **Herramientas:** **fsck**, **iostat**, **defrag**.
- **Importancia:** afecta al rendimiento y acceso físico a los datos.

◆ Organización lógica

- **Qué es:** estructura interna de los datos (campos, registros, claves).
- **Quién la usa:** programadores, SGBD, scripts.
- **Ejemplos:** **.json**, **.csv**, **.sqlite**, **.yaml**.
- **Herramientas:** **jq**, **sqlite3**, editores.
- **Importancia:** facilita lectura, búsqueda y procesamiento.

3. Tipos de organización de ficheros

3.1 Ficheros secuenciales

- Registros almacenados uno tras otro, en orden.
- Acceso lineal, desde el inicio.
- Ventajas: sencillos, eficientes para escritura continua.
- Usos: logs, backups, trazas.
- **Ejemplo real:** **/var/log/vsftpd.log** (registro de accesos FTP).
- **Herramientas:** **tail**, **cat**, **head**.

3.2 Ficheros secuenciales indexados

- Añaden un índice para búsquedas rápidas sin perder orden lógico.
- **Ventajas:** buena relación entre rendimiento y estructura.
- **Usos:** bases de datos simples, catálogos, usuarios de red.
- **Ejemplo real:** base de datos SQLite para usuarios FTP con índice por login.

3.3 Ficheros directos (aleatorios)

- Acceso directo al registro por clave o posición (hash, dirección).
- Muy rápidos, ideales para configuraciones o binarios.
- Usos: sistemas críticos, backups cifrados, parámetros de red.
- **Ejemplo real:** acceso binario a **.pt** o **.gguf** (modelos IA); **seek()** en Python.

3.4 Ficheros relacionales

- Datos organizados en tablas, gestionados por SGBD.
- Soportan múltiples usuarios, SQL, control de acceso, integridad.
- Usos: ERP, gestión de usuarios, registros de red.
- **Ejemplo real:** tabla MySQL con usuarios FTP, permisos y rutas.

3.5 Ficheros jerárquicos / semiestructurados







- Estructura flexible por claves, etiquetas o indentación.
- Ideales para configuraciones, modelos, APIs, servicios.
- Usos: scripts, contenedores, definición de servicios en red.
- **Ejemplo real:**
 - **vsftpd.conf** (clave-valor) **docker-compose.yaml** (estructura YAML de servicios)

4. Criterios de elección

Necesidad	Organización recomendada
Escritura masiva	Secuencial
Búsqueda por clave	Indexada o directa
Configuración flexible	Jerárquica (JSON/YAML)

Multiusuario y seguridad	Relacional
Acceso a modelos IA / binarios	Directa
Análisis de eventos	Secuencial + herramientas CLI

5. Actividades prácticas

Área de trabajo	Actividad general
 Organización de ficheros	Diseñar estructura lógica de directorios por función o departamento.
 Análisis de registros	Leer y filtrar información en ficheros secuenciales (ej. logs o historiales).
 Gestión de usuarios	Almacenar información de acceso en ficheros estructurados con índice.
 Seguridad en almacenamiento	Aplicar cifrado a ficheros según su tipo o nivel de sensibilidad.
 Acceso binario	Leer registros en ficheros binarios mediante desplazamiento directo.
 Configuración estructurada	Editar ficheros jerárquicos (ej. <code>.yaml</code> , <code>.json</code>) para definir servicios o rutas.

6. Conclusión

La organización de ficheros no es solo una cuestión técnica, sino una herramienta para garantizar **eficiencia, seguridad, escalabilidad y control** en entornos reales.

PROPUESTA DIDÁCTICA

Asignatura: Bases de Datos y Programación (CFGs DAM)

1. Contextualización

- Nivel: alumnado con base en programación y estructuras
- Conexión: ficheros, rendimiento, almacenamiento, servicios en red

2. Objetivos

- Aplicar distintos tipos de organización de ficheros
- Programar lectura/escritura eficiente de datos externos
- Evaluar rendimiento y uso adecuado según contexto




3. Metodología

Enfoque a la Diversidad / DUA

- **Nivel III:** código base, ejemplos guiados, esquemas
- **Nivel IV:** pseudocódigo, depuración visual, reflexión autónoma
- **DUA:**
 - Entrada: vídeo, consola, esquemas
 - Salida: código, demo, presentación
 - Flexibilidad: entregas por fases o roles

4. Actividad principal

Título: Comparador de estructuras de fichero

1. Programar 3 versiones de una misma app (ej. agenda):
 -  Secuencial (`.csv`),  Indexada (SQLite),  Acceso directo (binario con clave)
2. Medir: tiempo de acceso, volumen, flexibilidad
3. Justificar elección de la mejor estructura según el caso

5. Evaluación

- **Rúbrica código:** funcionalidad, estructura, buenas prácticas **Test funcional:** ejecución y tiempos. **Presentación:** reflexión técnica breve


Tema 15. Sistemas operativos. Componentes. Estructura. Funciones. Tipos.

1. Introducción


- El sistema operativo (SO) es el software base que actúa como intermediario entre el **hardware** y el **usuario** o **aplicaciones**.
- Administra **recursos físicos y lógicos**: CPU, memoria, dispositivos, procesos y ficheros.
- Permite ejecutar programas de forma **segura, eficiente y concurrente**.
- Su conocimiento es clave para tareas de **administración de sistemas, desarrollo de software, redes y seguridad**.

2. Funciones principales del sistema operativo


2.1 Gestión de procesos

- Carga, planificación, ejecución, sincronización y finalización de procesos.
- Planificadores: Round Robin, FIFO, SJF, multilevel queue.
- Estados: nuevo, listo, ejecutando, bloqueado, terminado.
-  Ejemplo práctico: **ps, top, kill, nice, htop**.


2.2 Gestión de memoria

- Asignación y liberación dinámica de espacio a procesos.
- Aislamiento, protección, paginación, segmentación, swapping.
- Memoria virtual: espacio lógico mayor al físico.
-  Comandos: **free, vmstat, cat /proc/meminfo**.


2.3 Gestión de dispositivos de entrada/salida (E/S)

- Abstracción y control de periféricos mediante drivers.
- Buffers, colas de E/S, interrupciones, polling.
- Interfaces: USB, SATA, PCIe.
-  Ejemplo práctico: **lsusb, lspci, dmesg**.

2.4 Gestión de archivos

- Organización jerárquica de datos en unidades lógicas (ficheros).
- Operaciones: crear, leer, escribir, eliminar, montar, permisos.
- Sistemas de archivos: FAT32, NTFS, ext4, XFS, APFS.
-  Comandos: **ls, cp, mv, chmod, mount, df**.

2.5 Seguridad y control de acceso

- Control de usuarios, políticas de permisos, autenticación.
- Protección frente a acceso indebido o malicioso.
- Mecanismos: cifrado, listas de control de acceso (ACL), SELinux, AppArmor.
-  Comandos: **passwd, chown, chmod, umask, su, sudo**.

3. Componentes del sistema operativo

3.1 Núcleo (Kernel)

- Componente central que gestiona hardware, interrupciones, memoria y procesos.
- Tipos: monolítico (Linux), microkernel (Minix), híbrido (Windows).

3.2 Gestores / módulos

- Submódulos funcionales: gestor de procesos, memoria, archivos, E/S, red.

3.3 Shell

- Interfaz con el usuario:
 - CLI (Command-Line Interface): **bash, PowerShell**.
 - GUI (Graphical User Interface): GNOME, KDE, Windows Explorer.

3.4 Controladores (drivers)

- Código específico que traduce órdenes del SO para un dispositivo concreto.
- Puede ser cargado dinámicamente (**modprobe, .inf, .sys**).

4. Estructura del sistema operativo

- **Monolítico**: un solo bloque de código (Linux clásico).
- **Microkernel**: funciones mínimas en núcleo, el resto como servicios (Minix, QNX).
- **Modular**: carga de componentes bajo demanda (**systemd**, módulos Linux).
- **Cliente-servidor**: SO distribuye funciones entre procesos (en red).
- **Híbrido**: combina varios enfoques (Windows NT, macOS).

5. Tipos de sistemas operativos

5.1 Por dispositivo

- Escritorio: Windows, Ubuntu, macOS.
- Móvil: Android, iOS.
- Servidor: Debian, CentOS, Windows Server.
- Embebido: RTOS, OpenWRT, Raspbian.
- Tiempo real: FreeRTOS, QNX.

5.2 Por arquitectura

- Monousuario vs multiusuario
- Monotarea vs multitarea
- Distribuidos y virtualizados: Proxmox, VMware ESXi
- En la nube: SO como servicio, contenedores (Alpine, distroless)

6. Tendencias actuales y evolución

- Contenerización: uso de SO mínimos para Docker/Kubernetes.
- Virtualización nativa: hipervisores tipo 1 y 2.
- Seguridad reforzada: SELinux, cifrado de disco, TPM 2.0.
- Automatización: scripts de administración, DevOps, Salt Project
- SO en la nube y en dispositivos IoT.

PROPUESTA DIDÁCTICA

Asignatura: Programación de Servicios y Procesos (2.º DAM)

1. Contextualización

Aula equipada con sistemas Windows y Linux, acceso a terminales, editores de código y herramientas de administración.

Alumnado con perfil de desarrollo backend, conocimientos previos en concurrencia, procesos y fundamentos de SO.

2. Objetivos

- Programar tareas relacionadas con la gestión de procesos del sistema.
- Aplicar conceptos del sistema operativo a nivel de usuario y programador.
- Analizar el comportamiento de procesos y recursos en tiempo real.

3. Metodología

- Proyecto técnico individual o por parejas.
- Exploración práctica del sistema con programación de scripts/aplicaciones.
- Análisis funcional a partir de pruebas reales con procesos, memoria y permisos.

4. Atención a la Diversidad y DUA

- **Nivel III:** base de código inicial, apoyo con vídeos y plantillas.
- **Nivel IV:** análisis guiado sin programación profunda, opción de simulación.
- **DUA:**
 - Entrada: documentación + vídeo + demo de comandos.
 - Salida: informe, presentación o ejecución comentada.
 - Flexibilidad en lenguaje (Python, Java, bash).

5. Actividad Principal

“Monitor y gestor de procesos multiplataforma”

Tareas:

- Desarrollar una aplicación concurrente que monitorice procesos y memoria del sistema y que mande información a través de una API.
- Mostrar PID, estado, consumo de CPU y memoria, usuario propietario.
- Permitir al usuario enviar señales (**kill**, **nice**, **suspend**, etc.) a procesos activos.
- Registrar las acciones realizadas en un fichero de log.
- (Opcional) Añadir interfaz gráfica o exportación de datos.

Tecnologías sugeridas: Python (**psutil**), Java (**ProcessHandle**), bash.

6. Evaluación

- **Instrumentos:** rúbrica de proyecto, seguimiento individual, test final de conocimientos.
- **Criterios:**
 - Funcionalidad de la aplicación desarrollada.
 - Uso correcto de conceptos del sistema operativo.
 - Aplicación de permisos y gestión de procesos reales.
 - Claridad y justificación técnica en la presentación.

7. Conclusión Didáctica

La actividad conecta la teoría del sistema operativo con su programación directa, desarrollando habilidades técnicas útiles para el entorno profesional. El alumnado integra competencias en procesos, concurrencia y control del sistema, mejorando su perfil como programadores orientados a servicios y administración avanzada.

Tema 16. Sistemas operativos: Gestión de procesos.

1.1 Introducción

- Un proceso es un programa en ejecución con recursos del sistema operativo: CPU, RAM, archivos, E/S.
- La gestión eficiente permite multitarea, aislamiento y rendimiento.
- Clave en: servidores cloud, IA, contenedores, microservicios.

1.2 Ciclo de Vida del Proceso

- Modelo de 5 estados: Nuevo, Listo, Ejecución, Bloqueado, Terminado.
- Transiciones: dispatch, I/O, interrupciones.

1.3 Modelo de 7 Estados (con swap)

- Añade: Listo suspendido y Bloqueado suspendido (disco).
- Mejora la gestión de memoria y escalabilidad.

1.4 PCB (Process Control Block)

- Contiene: PID, registros, punteros a memoria, estadísticas, UID/GID.

1.5 Hilos (Threads)

- Subprocesos dentro del mismo espacio de direcciones.
- Modelos: 1:1 (Linux), N:1 (limitado), M:N (Go, Erlang).
- Problemas: condiciones de carrera, bloqueos.
- Soluciones: mutex, semáforos, monitores.

1.6 Planificación de CPU

- Criterios: turnaround, waiting, response time, throughput.
- Algoritmos: FCFS, SJF, Round Robin, prioridades, multicolos.
- **Linux CFS**: justa distribución con árbol rojo-negro ($O(\log n)$).

1.7 Concurrencia y Sincronización

- Sincronización esencial ante acceso compartido a recursos.
- Mecanismos: semáforos (wait/signal), monitores, spinlocks.
- Ejemplo: filósofos comensales.

1.8 Comunicación entre Procesos (IPC)

- Técnicas: memoria compartida, pipes, colas, sockets.
- Uso en pipelines de IA, microservicios, servicios distribuidos.

1.9 Interbloqueos

- Condiciones de Coffman.
- **Mutua exclusión**: Al menos un recurso no puede ser compartido; solo un proceso puede usarlo a la vez.
- **Retención y espera**: Un proceso con recursos retenidos puede solicitar otros y quedar bloqueado.
- **No expropiación**: Los recursos no pueden ser forzadamente retirados; solo se liberan voluntariamente.
- **Espera circular**: Existe una cadena de procesos donde cada uno espera un recurso del siguiente.
- Estrategias: prevención, evitación (banquero), detección y recuperación.

1.10 Casuística en S.O.

SO	Gestión de procesos	Herramientas
Linux	fork/exec, /proc, CFS	ps, htop, strace
Windows	CreateProcess, scheduler	TaskMgr, PowerShell
Contenedores	namespaces, cgroups	Docker, Podman

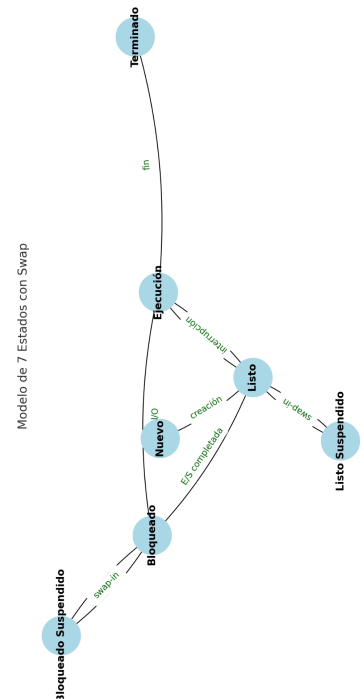
1.11 Seguridad

- Riesgos: procesos zombie, condiciones de carrera, IPC insegura.
- Mitigación: ASLR, UID, sandbox, auditorías.

2. PARTE DIDÁCTICA (MÓDULO: PROGRAMACIÓN DE SERVICIOS Y PROCESOS)

2.1 Contextualización

- **Nivel**: Grado Superior en DAM/ASIR.
- **Módulo**: Programación de Servicios y Procesos.
- **Perfil**: alumnado con conocimientos en programación y administración de sistemas. Enfoque



profesional y automatización.

2.2 Objetivos de aprendizaje

- Programar servicios como procesos o demonios.
- Gestionar subprocesos/hilos para tareas concurrentes.
- Implementar sincronización segura entre procesos.
- Utilizar planificación adecuada y comunicación eficiente.

2.3 Metodología

- Aprendizaje activo basado en proyectos (ABP).
- Simulación de sistemas reales (servicios Linux, hilos Java o Python).
- Evaluación mediante prácticas funcionales y análisis de código.

2.4 Atención a la diversidad (niveles III y IV)

- Prácticas guiadas con plantillas.
- Tareas escalonadas según nivel.
- Apoyo personalizado en lógica concurrente.

2.5 DUA

- Representación: vídeos, depuradores, visualizadores de procesos.
- Acción: scripting, programación, pruebas automatizadas.
- Compromiso: proyectos aplicados (chat, servidor de colas, servicios REST).

2.6 Actividad principal

Supuesto didáctico: “Desarrollo de un servicio concurrente de atención de tareas”

- Desarrollar una aplicación concurrente que monitorice procesos y memoria del sistema y que mande información a través de una API.
- Objetivo: desarrollar un servicio en Python/Java que atienda múltiples tareas simulando peticiones de clientes.
- Implementa multihilo, colas de mensajes, sincronización con semáforos.
- Control de estado de procesos y uso de `psutil` (Python) o `Thread` (Java).
- Incluye logs, planificación simple y monitoreo.

2.7 Evaluación

- **Instrumentos:** rúbrica de código, test técnico, checklist de funcionalidades.
- **Criterios:**
 - Correcta creación y control de procesos e hilos.
 - Implementación de sincronización y manejo de condiciones de carrera.
 - Estructura y calidad del código.

2.8 Conclusión didáctica

- La programación y gestión de procesos es esencial para servicios concurrentes.
- Capacita al alumnado para diseñar y mantener software robusto, seguro y eficiente en entornos reales de servidor.
- Este módulo vincula la teoría de los sistemas operativos con la práctica profesional de la programación de servicios.

Tema 17. Sistemas operativos: Gestión de memoria

1. Introducción

- La gestión de memoria es una función crítica del sistema operativo.
- Objetivos: eficiencia, protección, asignación dinámica, multitarea, optimización del rendimiento.

2. Tipos de Memoria

- **RAM (Memoria principal):** volátil, rápida, almacena datos en uso.
- **ROM:** no volátil, solo lectura, contiene instrucciones básicas (BIOS/UEFI).
- **Memoria caché:** intermedia entre CPU y RAM; muy rápida.
- **Memoria virtual:** usa disco como extensión de la RAM.
- **Memoria secundaria:** almacenamiento permanente (HDD, SSD).
- **Memoria flash:** usada en SSD, USB; rápida y no volátil.

3. Funciones del SO en la Gestión de Memoria

3.1 Asignación de Memoria

- **Estática:** al inicio del proceso.
- **Dinámica:** durante la ejecución.
- A procesos del usuario y al sistema operativo.

3.2 Protección y Aislamiento

- Cada proceso accede solo a su espacio de memoria.
- Previene corrupción o acceso no autorizado.

3.3 Liberación de Memoria

- Al finalizar el proceso. También, con funciones propias dentro del programa.
- **Garbage Collector:** libera memoria automáticamente (Java, Python...).

3.4 Intercambio (Swapping)

- Mueve procesos entre RAM y disco para liberar espacio.
- Mejora la multitarea, pero puede impactar el rendimiento.

4. Técnicas de Gestión

4.1 Fragmentación

- **Interna:** pérdida de espacio **dentro de bloques asignados** (paginación).
- **Externa:** pérdida de espacio en **huecos dispersos** (segmentación, asignación contigua).
- Técnicas para reducirla: paginación (reduce externa), segmentación (reduce interna), compactación (requiere coste computacional).

4.2 Segmentación

- Divide la memoria lógica de un proceso en segmentos de tamaño variable, como código, datos o pila. Usa tabla de segmentos.
- Facilita la protección (cada segmento tiene atributos) y la modularidad (segmentos independientes y reutilizables).

4.3 Paginación

- Divide la memoria lógica en páginas y la memoria física en marcos (frames) del mismo tamaño.
- Evita la fragmentación externa, aunque puede causar fragmentación interna.
- La traducción de direcciones se realiza mediante una **tabla de páginas**; para mejorar el rendimiento se emplea una caché llamada **TLB (Translation Lookaside Buffer)**.

4.4 Segmentación paginada

- **Cada segmento** se subdivide en **páginas**. Combina las ventajas de segmentación y paginación.
- Cada dirección lógica se interpreta como:
[Número de segmento] → [Número de página] → [Desplazamiento].
- Uso común en arquitecturas modernas para gestión eficiente y flexible.
- **Un segmento es una parte lógica del programa (como código o datos), que puede dividirse en páginas, y cada página se almacena en un marco de memoria física del mismo tamaño.**

4.5 Memoria Compartida

- Permite que varios procesos accedan al mismo espacio de memoria.
- Usada para comunicación entre procesos (IPC).

5. Aspectos Avanzados

5.1 Paginación por Demanda

- Las páginas se cargan solo cuando se necesitan.
- Si no está en RAM → **page fault** → se carga desde disco.

5.2 Thrashing

- Exceso de carga y descarga de páginas. Disminuye el rendimiento drásticamente.
- Soluciones: control de procesos, más RAM, algoritmos de reemplazo.

5.3 Algoritmos de Reemplazo de Página

- **FIFO**: reemplaza la página más antigua
- **LRU**: reemplaza la menos usada recientemente.
- **Óptimo**: ideal pero no aplicable en la práctica.
- **Clock**: versión eficiente del LRU. Usa un puntero circular y un bit de uso para decidir qué página reemplazar.

5.4 Gestión en Sistemas Empotrados

- Recursos muy limitados.
- Sin memoria virtual; asignación estática, precisa y eficiente.

5.5 Herramientas de Análisis y Depuración

- Detección de fugas de memoria y errores:
 - **Valgrind** (C/C++), **VisualVM** (Java), **tracemalloc** (Python).

6. Tendencias Modernas

6.1 Memoria Persistente (NVDIMM)

- Combina velocidad de RAM con persistencia de disco

6.2 Optimización con IA

- Predicción de uso para liberar o asignar memoria anticipadamente.

6.3 Contenedores y Virtualización

- Aislamiento y gestión de memoria en entornos como Docker o Kubernetes.
- Uso de **cgroups** para limitar y controlar recursos.

PROPUESTA DIDÁCTICA

Asignatura: Programación (1.º DAM)

Unidad temática: Gestión de memoria: segmentación, paginación y segmentación paginada

1. Contextualización

El alumnado se inicia en conceptos clave de arquitectura y funcionamiento de sistemas, con conocimientos básicos de programación y estructuras de datos.

Se utilizará software visual o simuladores interactivos para representar y comprender los esquemas de asignación de memoria.

2. Objetivos

- Comprender los fundamentos de la segmentación, paginación y segmentación paginada.
- Simular la asignación, traducción y liberación de memoria con ejemplos paso a paso.
- Relacionar los conceptos con estructuras de datos (arrays, pilas, funciones...) de sus programas.

3. Metodología

- Aprendizaje **activo y visual**, mediante simulaciones y ejemplos guiados.
- Uso de representaciones gráficas y mapas de memoria.
- Actividades de **resolución de problemas prácticos** y trabajo en parejas para reforzar la comprensión.

4. Atención a la Diversidad y DUA

- **Nivel III (refuerzo)**: explicaciones paso a paso, vídeos y esquemas visuales interactivos.
- **Nivel IV (ampliación)**: resolución de problemas tipo examen, desafíos con errores intencionados a detectar.
- **DUA**:
 - **Múltiples formas de entrada**: esquemas, simuladores, vídeos, ejemplos con código.
 - **Múltiples formas de expresión**: entregas en forma de dibujos, pseudocódigo o explicaciones orales/escritas.
 - **Flexibilidad temporal**: tiempo extra o repeticiones si es necesario.

5. Actividad Principal

Programación de un simulador de gestión de memoria

- Modelos a implementar:
 - **Segmentación paginada**: segmentos subdivididos en páginas.
- El simulador debe mostrar:
 - Tablas generadas, mapa de memoria, traducción de direcciones lógicas a físicas.

6. Evaluación

- **Instrumentos**: entrega del código, rúbrica técnica, exposición final.
- **Criterios**: funcionamiento del simulador, claridad del diseño, uso de estructuras adecuadas, comprensión del modelo.

7. Conclusión Didáctica

Tema 18. Sistemas operativos: Gestión de entradas/salidas.

1. Introducción

- La gestión de E/S permite que el sistema operativo se comunique con los dispositivos periféricos.
- Su objetivo es asegurar un **intercambio eficiente, seguro y controlado** de datos entre el procesador, la memoria y el exterior (teclado, disco, red...).
- Ejemplo: leer un archivo desde disco y mostrarlo por pantalla implica múltiples niveles de E/S.

2. Objetivos del Subsistema de E/S

- **Abstracción del hardware:** el usuario/programador no necesita conocer detalles técnicos del dispositivo.
- **Asignación y liberación de dispositivos:** control del uso de periféricos compartidos.
- **Planificación del acceso:** ordenación y gestión de peticiones de múltiples procesos.
- **Protección y sincronización:** evita conflictos y accesos simultáneos inseguros.

3. Componentes del Sistema de E/S

- **Controladores (drivers):** traducen las órdenes del SO al lenguaje específico del dispositivo.
- **Subsistema de E/S en el kernel:** gestiona las peticiones y coordina los controladores.
- **Buffering:** almacenamiento temporal para sincronizar velocidades entre dispositivo y CPU.
- **Caching:** guarda datos usados recientemente para acelerar accesos repetidos.

4. Técnicas de Gestión de E/S

4.1 E/S Programada

- La CPU controla directamente el dispositivo y espera a que termine la operación.
- Muy **ineficiente**: el procesador queda ocupado todo el tiempo.

4.2 E/S Mediante Interrupciones

- El dispositivo **interrumpe** a la CPU cuando necesita atención. Interrupciones enmascarables y no enmascarables.
- Mejora la eficiencia: permite ejecutar otras tareas mientras se espera.

4.3 DMA (Acceso Directo a Memoria)

- El dispositivo transfiere datos directamente a la RAM sin intervención continua de la CPU.
- Muy usado en discos, tarjetas de red, audio/video.

5. Clasificación de dispositivos y modos de acceso

5.1 Tipos de dispositivos

- **Entrada:** teclado, ratón.
- **Salida:** pantalla, impresora.
- **Entrada/Salida:** discos, USB, red.

5.2 Por tipo de acceso

- **De carácter:** flujo de datos (teclado, serie).
- **De bloque:** acceden a bloques fijos (discos).

5.3 Modo de operación

- **Sincrónico:** el proceso se bloquea hasta que finaliza la E/S.
- **Asincrónico:** el proceso continúa y se notifica cuando la E/S termina.

6. Planificación de E/S (especialmente en discos mecánicos sin acceso aleatorio)

- **FCFS (First Come First Served):** orden de llegada.
- **SSTF (Shortest Seek Time First):** menor distancia del cabezal.
- **SCAN (ascensor):** el cabezal recorre el disco ida y vuelta.
- **LOOK:** similar al SCAN, pero se detiene si no hay más peticiones.

8. Monitorización y rendimiento de E/S

- **Herramientas reales:** *iotstat*, *iotop*, *dstat*, *Task Manager*.
- **Indicadores clave:**
 - **Latencia:** tiempo de respuesta.
 - **Throughput:** cantidad de datos procesados por segundo.
 - **Cuellos de botella:** procesos que ralentizan el sistema.
- **Aplicación práctica:** análisis de rendimiento de un sistema real en laboratorio.

9. Conclusión

- La gestión de E/S es esencial para que el sistema operativo interactúe con el mundo exterior de forma eficaz.
- Entender sus componentes y técnicas permite optimizar el rendimiento, evitar bloqueos y diseñar software compatible con múltiples dispositivos.
- La evolución hacia modelos eficientes como **DMA** y el uso de **interrupciones** demuestra la importancia de liberar a la CPU y mejorar la multitarea.

Módulo: Programación (1.º DAM) Simulación de técnicas de Entrada/Salida (E/S) y análisis de rendimiento

1. Contextualización

Entorno con ordenadores personales o simuladores simples.

Alumnado con conocimientos básicos de programación y estructuras de control.

Se abordará la simulación de técnicas de E/S desde un enfoque algorítmico para comprender su impacto en el rendimiento.

2. Objetivos

- Comprender el funcionamiento lógico de las técnicas de E/S: programada, con interrupciones y DMA.
- Programar una **simulación por software** que represente cada técnica.
- Medir y comparar el coste (tiempo y eficiencia) de cada una.
- Relacionar la teoría con su aplicación práctica mediante código.

3. Metodología

- Enfoque práctico: desarrollo por fases de un programa simulador.
- Análisis comparativo con **medición de tiempos**, uso de funciones `sleep()` o contadores simulados.
- Actividad cooperativa o individual según nivel.
- Uso de gráficos o tablas para visualizar los resultados.

4. Atención a la Diversidad y DUA

- **Nivel III:** simuladores guiados paso a paso, pseudocódigo base y ayuda visual.
- **Nivel IV:** implementación libre del simulador, reto opcional con interfaz gráfica o comparación estadística.
- **DUA:**
 - **Múltiples representaciones:** esquemas de flujo, animaciones, código comentado.
 - **Múltiples formas de expresión:** entregas como informe técnico, defensa oral o visualización gráfica.
 - **Andamiaje progresivo:** material escalado según nivel del grupo.

5. Actividad Principal

“Simulación de técnicas de E/S”

- El alumnado programará un simulador sencillo que represente:
 - **E/S Programada:** bucle de espera activo (polling).
 - **E/S con interrupciones:** respuesta reactiva tras señal.
 - **DMA:** transferencia simulada paralela con mínima intervención del "CPU".
- Para cada técnica se medirán:
 - **Tiempo total de uso del “CPU”.**
 - **Número de ciclos ocupados.**
 - **Comparación gráfica de eficiencia.**
- Entrega final: **informe comparativo** + simulador funcional + visualización de resultados.

6. Evaluación

- **Instrumentos:** lista de cotejo técnica, entrega del simulador, presentación breve.
- **Criterios:**
 - Implementación funcional de cada técnica.
 - Claridad y justificación en la comparación de costes.
 - Uso adecuado de estructuras de control y medición.
 - Capacidad de explicar y defender las decisiones tomadas.

7. Conclusión Didáctica

Simular las técnicas de E/S desde la programación permite al alumnado entender cómo afectan al uso del procesador y al rendimiento del sistema. Este enfoque favorece la conexión entre los conceptos de sistemas operativos y la práctica del diseño algorítmico, fomentando una visión crítica y aplicada del uso de recursos computacionales.

Tema 19. Sistemas operativos: Gestión de archivos y dispositivos

1. Introducción

El sistema operativo organiza y gestiona la información mediante un sistema de archivos y controla el acceso a los dispositivos. Esto permite que usuarios y programas manipulen datos sin preocuparse por los detalles del hardware.

Ejemplo: abrir un documento implica localizarlo en el sistema de archivos, verificar permisos, acceder al disco y al controlador.

2. Objetivos de la gestión de archivos y dispositivos

- Proporcionar una **estructura lógica** para almacenar y acceder a datos.
- Garantizar la **integridad, seguridad y disponibilidad** de los archivos.
- Administrar el acceso a **periféricos físicos** mediante controladores.
- Controlar el uso de **recursos compartidos** entre procesos.
- Optimizar el **rendimiento y la eficiencia del almacenamiento**.

3. Estructura del sistema de archivos

- **Archivo**: unidad lógica de almacenamiento (texto, binario...).
- **Directorio**: agrupación lógica de archivos.
- **Ruta**: ubicación dentro del árbol de directorios.
 - Absoluta: `/home/user/doc.txt`
 - Relativa: `../doc.txt`
- **Descriptor de archivo**: identificador interno usado por el sistema para acceder a un archivo abierto

4. Tipos y características de sistemas de archivos

Sistema	Plataforma	Características	Limitaciones
FAT32	Windows	Compatible, simple	Máx. 4 GB por archivo
NTFS	Windows	Permisos, cifrado, journaling	No nativo en Linux sin drivers
EXT4	Linux	Estable, rápido, journaling	-
Btrfs	Linux	Snapshots, copy-on-write	Más complejo de administrar
ZFS	Unix/Linux	Integridad, compresión	Alto consumo de RAM, licencia

5. Operaciones y permisos sobre archivos

- **Operaciones básicas**: crear, abrir, leer, escribir, cerrar, eliminar.
- **Permisos en Linux (rwx)**:
 1. Usuario, grupo, otros: `-rwxr-xr--`
 2. Gestión: `chmod`, `chown`, `umask`
- **ACLs (Access Control Lists)**:
 1. Permiten definir permisos más granulares por usuario/grupo.
- **Metadatos**: nombre, tamaño, tipo, propietario, fecha de modificación.
- **Archivo oculto**: en Unix/Linux comienza con `.` (ej.: `.bashrc`).
- **Flujo de uso típico**:
 1. Crear → 2. Abrir → 3. Leer/Escribir → 4. Cerrar

6. Gestión de dispositivos

6.1 Tipos de dispositivos

- **De carácter**: datos en flujo (teclado, ratón, serie).
- **De bloque**: acceso por bloques (discos, USB, SSD).

6.2 Acceso a dispositivos

- A través de archivos especiales:
 - Linux: `/dev/sda1`, `/dev/tty`, `/dev/null`
 - Windows: `COM1`, `LPT1`, `C:\`

6.3 Controladores (drivers)

- Traducen órdenes del SO al lenguaje del hardware.
- Pueden cargarse automáticamente o manualmente.

7. Montaje y administración de dispositivos

- **Montaje**: integración del dispositivo en el sistema de archivos.
- **Punto de montaje**: carpeta en la que se accede al dispositivo.

- **Comandos Linux:**
 - Ver dispositivos: `lsblk`, `blkid`
 - Montar: `mount /dev/sdb1 /mnt/usb`
 - Automontaje: `/etc/fstab`
- **Windows:**
 - Letras de unidad (D:\, E:\). Herramienta Administración de discos

8. Seguridad, fiabilidad y backups

- **Journaling:** previene corrupción (EXT4, NTFS, ZFS).
- **Snapshots:** estados inmutables del sistema de archivos (Btrfs, ZFS).
- **Cifrado:**
 - Windows: BitLocker
 - Linux: LUKS, ZFS
- **Copias de seguridad:**
 - Manuales: `rsync`, `tar`, 7-Zip
 - Automáticas: `cron`, Timeshift, Historial de archivos
 - Regla 3-2-1: 3 copias, 2 soportes, 1 en otro lugar

9. Sistemas de archivos en RAM y virtuales

9.1 En memoria (RAM-based)

- **tmpfs** en Linux: se monta en RAM (`/tmp`, `/dev/shm`)
- **Ventajas:** velocidad muy alta, ideal para pruebas o cachés
- **Inconvenientes:** datos volátiles, limitados por tamaño de RAM

Ejemplo: `mount -t tmpfs tmpfs /mnt/ramdisk`

9.2 Virtuales

- Simulan archivos para acceder a información del sistema.
- **Linux:**
 - `/proc`: información de procesos y hardware
 - `/sys`: parámetros del kernel
 - `/dev`: dispositivos como archivos
- **Windows:** `NUL`, `CON`, `PRN`, `WMI`
- **Aplicación práctica:**
 - Leer `/proc/cpuinfo`, `/proc/meminfo` para extraer info del sistema

Propuesta Didáctica – Módulo: Programación (1.º DAM)

1. Contextualización

Se simula un sistema de archivos y acceso a dispositivos para aplicar lógica de programación estructurada.

2. Objetivos

- Simular operaciones básicas del sistema de archivos.
- Aplicar estructuras (listas, árboles, mapas) para representar jerarquía, permisos y bloqueos.
- Comparar velocidad o restricciones entre RAM, disco y dispositivos.

3. Metodología

- Trabajo por fases: diseño, codificación, prueba.
- Enfoque progresivo: CRUD, permisos, acceso a dispositivos.
- Posibilidad de comparar simulación con comportamiento real del sistema.

4. Atención a la Diversidad y DUA

- **Nivel III:** pseudocódigo guiado, ejemplos en pareja.
- **Nivel IV:** implementación libre, uso de estructuras avanzadas.
- **UDL:**
 - Esquemas visuales del árbol de directorios.
 - Explicaciones orales, informes o presentaciones.

5. Actividad Principal

“Simulador de sistema de archivos y acceso a dispositivos”

- **Simular:**
 - Estructura de carpetas y archivos.
 - Operaciones CRUD.
 - Sistema básico de permisos (lectura/escritura).
 - Acceso concurrente (bloqueo).
 - Dispositivos simulados (bloque y carácter).

6. Evaluación

- **Instrumentos:** rúbrica técnica, presentación, código entregado.
- **Criterios**

Tema 20. Explotación y Administración de sistemas operativos monousuario y multiusuario.

1. Introducción

El sistema operativo (SO) permite la interacción entre usuario y hardware. Administra procesos, memoria, dispositivos, tareas y seguridad, adaptándose a distintos contextos: doméstico, profesional o empresarial.

2. Clasificación de sistemas operativos

- **Por número de usuarios:**
 - *Monousuario*: un usuario activo por vez. Ej.: Windows Home, macOS.
 - *Multiusuario*: múltiples sesiones simultáneas. Ej.: Linux, Windows Server.
- **Por tareas:**
 - *Monotarea*: obsoleto. *Multitarea*: alternancia (concurrente) o simultaneidad real (paralela).
- **Por núcleo (kernel):**
 - *Monolítico*: alto rendimiento, difícil de mantener (Linux).
 - *Microkernel*: modular, más seguro (Minix, QNX).
 - *Híbrido*: rendimiento y seguridad (Windows NT, macOS).
- **Por arquitectura:**
 - *Sistemas en red*: comparten recursos (FreeBSD, Windows Server).
 - *Distribuidos*: múltiples máquinas como un único sistema lógico (Hadoop, Mesos).
 - *Cloud*: escalabilidad y ejecución bajo demanda (AWS, Azure).
- **Por procesadores:**
 - *SMP (Symmetric Multiprocessing)*: núcleos comparten memoria.
 - *NUMA (Non-Uniform Memory Access)*: varias CPU cada una su propia memoria.
- **RTOS (tiempo real)**: QNX, VxWorks.
- **Emergentes**: SO para IoT (Zephyr, RIOT).

3. Explotación de sistemas monousuario

- Instalación del SO, drivers y software básico.
- Gestión de cuentas locales y configuraciones del sistema.
- **Copias de seguridad automáticas** con herramientas como Time Machine.
- **Medidas de seguridad básicas**: actualización automática, antivirus, cifrado de disco (ej.: BitLocker, FileVault).

4. Administración de sistemas multiusuario

4.1 Procesos y planificación

- Procesos ejecutados en modo usuario, gestionados por el SO.
- Planificación mediante algoritmos como FIFO, Round Robin (RR), prioridades.
- Comunicación entre procesos (IPC) con *pipes*, *sockets*, *memoria compartida*.

4.2 Memoria

- Técnicas: **paginación**, **segmentación**, **swapping** (intercambio entre RAM y disco).
- Separación de memoria en **modo usuario** y **modo kernel** por seguridad.

4.3 Servicios y demonios

- **Linux**: **systemd** administra servicios (daemons), **cron** para tareas programadas.
- **Windows**: **Task Scheduler**, servicios en segundo plano.

4.4 Almacenamiento

- **Sistemas de archivos**:
 - NTFS (Windows), EXT4, Btrfs (Linux con snapshots), ZFS (integridad, compresión, replicación).
- **Volúmenes lógicos**: LVM en Linux, Storage Spaces en Windows.
- Permiten reorganizar, ampliar y gestionar mejor los discos.

4.5 Gestión avanzada

- **Linux**: **sudo**: permisos temporales de administrador. ACLs: control de permisos a nivel fino. **journalctl**: visualización de eventos y logs del sistema. **cgroups**: limitan recursos como CPU, RAM o red a procesos.
- **Windows Server**: Active Directory: gestión centralizada de usuarios y recursos. GPOs: políticas de grupo aplicadas a usuarios o equipos.

4.6 Copias de seguridad (Backup)

Las copias de seguridad permiten recuperar datos ante fallos, errores humanos o ataques. Deben formar parte de toda estrategia de administración.

- **Tipos**: Completa, Incremental, Diferencial. Regla 3-2-1: 3 copias, 2 medios, 1 externa.
- **Herramientas**: **Linux**: **rsync**, **tar**, **Deja Dup**, **borgbackup**, automatizado con **cron**.

5. Virtualización y contenedores

5.1 Máquinas virtuales (VMs)

- Emulan hardware completo. Cada VM tiene su propio sistema operativo.
- Ej.: VirtualBox, VMware.

5.2 Contenedores

- Comparten el **kernel del host** pero aíslan aplicaciones.
- Uso de **namespaces** (PID, red, sistema de archivos...) y **cgroups** (limitación de recursos).
- Ej.: Docker, LXC. Seguridad mejorada con AppArmor o SELinux.

5.3 Orquestación

- **Kubernetes**: despliegue, escalado y balanceo automático de contenedores.
- **Helm**: gestor de paquetes para Kubernetes.

6. Seguridad y monitorización

6.1 Seguridad

- **Modelos de control de acceso**: de acceso mediante DAC (Discretionary), MAC (Mandatory) y RBAC (Role-Based), uso de ACLs, autenticación multifactor
- **Cifrado**: BitLocker (Windows), LUKS (Linux), ZFS.
- **Firewalls**: **iptables**, **nftables** en Linux; Windows Defender Firewall.
- **Autenticación avanzada**:
 - Claves seguras, autenticación multifactor (2FA), certificados digitales.

6.2 Monitorización

- **Linux**:
 - **htop**: visualización de procesos. **journalctl**: consulta de logs del sistema.
 - Prometheus + Grafana: monitorización en tiempo real con paneles.
- **Windows**:
 - Sysinternals: herramientas avanzadas de diagnóstico.
 - Event Viewer: visualización de eventos del sistema.

7. Automatización de tareas administrativas

7.1 Scripts

- **Linux**: scripts bash con **cron** o temporizadores **systemd**.
- **Windows**: scripts PowerShell o **.bat** con **Task Scheduler**.

7.2 Tareas comunes

- Backups, limpieza de archivos, reinicios, informes de estado.

7.3 Herramientas avanzadas

- **Ansible**, **SaltStack**, **Puppet**: automatización de configuración en varios sistemas desde un único punto.

2. PARTE DIDÁCTICA

Actividad: Despliegue de Salt Project

Etapas educativas: 2.º SMR **Módulo profesional:** Servicios en Red

1. Contextualización

Instalación y configuración de Salt Project (automatización y administración remota) en un entorno virtualizado con Ubuntu Server. Se trabaja con el modelo master/minion.

2. Objetivos de aprendizaje

- Comprender el funcionamiento de Salt como sistema de gestión remota.
- Configurar correctamente la comunicación entre el master y los minions.
- Ejecutar comandos y aplicar configuraciones desde el master.

3. Metodología

- Trabajo por parejas o individual.
- Entorno virtualizado (VirtualBox, red interna).
- Desarrollo guiado con pruebas reales usando **salt** y **state.apply**.

4. Atención a la diversidad y DUA

- Nivel III: entornos preconfigurados, guías detalladas.
- Nivel IV: instalación libre, creación de estados propios.
- UDL: contenido accesible por línea de comandos, documentación y vídeos; entregas prácticas y escritas.

5. Actividad principal

- Crear una VM master y una o más minions (Ubuntu Server).
- Instalar y configurar Salt en ambos extremos.
- Verificar la conexión (**test.ping**).
- Ejecutar comandos remotos (**cmd.run**, **service.status**).
- Crear un archivo **.sls** que instale y active un servicio (ej. nginx).

6. Evaluación. Instrumentos y criterios.

Tema 21. Sistemas informáticos. Estructura física y funcional.

1. Introducción

- Un **sistema informático** es la combinación de hardware, software y comunicaciones para procesar, almacenar y transmitir información.
- Evolución: **mainframes** → **PCs** → **cloud** → **edge computing** → **IA distribuida**.
- Papel clave en servicios críticos (salud, educación, industria, transporte).

2. Clasificación y tendencias

2.1 Por tamaño

- Microcomputadoras, servidores, supercomputadoras.

2.2 Por arquitectura

- Cliente-servidor, embebido, cloud, IoT, sistemas distribuidos.

2.3 Tendencias actuales

- **Serverless computing** (AWS Lambda).
- **Contenedores y microservicios** (Docker, Kubernetes).
- **Edge computing, MicroVMs, SoCs (System on Chip)**.
- **IA distribuida, computación cuántica y neuromórfica**.

3. Impacto social y ético

- **Digitalización**: industria 4.0, smart cities, salud digital.
- **Riesgos y desafíos**: brecha digital, sostenibilidad (Green IT), sesgos algorítmicos.
- **Marco legal y ético**: RGPD, soberanía digital, ética computacional.

4. Arquitectura física

- **Modelo Von Neumann**: CPU, memoria, E/S, buses. **Arquitectura Harvard**.
- **CPU y coprocesadores**: ALU, registros, GPU, TPU (Tensores).
- **Memoria**: RAM (DDR5), caché, almacenamiento SSD (NVMe).
- **Periféricos**: entrada/salida clásicos y avanzados (VR, biometría).
- **Redes**: Ethernet, Wi-Fi 6, 5G, SDN (redes definidas por software).
- **Eficiencia energética**: refrigeración, energías renovables, centros de datos verdes.

5. Arquitectura lógica y software

5.1. Estructura funcional interna del sistema

- **Gestión de procesos**: planificación, estados de ejecución, prioridades.
- **Gestión de memoria**: asignación dinámica, paginación, segmentación, swapping.
- **Comunicación E/S**: controladores, interrupciones, acceso a periféricos.
- **Jerarquía de almacenamiento**: Nivel 1: caché (rápida, volátil) Nivel 2: RAM (principal, volátil). Nivel 3: almacenamiento persistente (SSD, HDD).

5.2. Capas lógicas del sistema

- Hardware → Sistema Operativo → Middleware → Aplicaciones.

5.3. Software y herramientas

- SO: Linux, Windows, Android, RTOS.
- Lenguajes: Python, Java, Rust.
- Herramientas: Git, CI/CD, virtualización, contenedores.
- DevOps, microservicios, cloud-native..

6. Gestión de recursos

- **Procesos y multitarea**: planificación de CPU.
- **Memoria**: paginación, segmentación, swapping.
- **Herramientas de monitorización**: top, htop, logs, Grafana, métricas.

7. Seguridad informática

- **Principios**: confidencialidad, integridad, disponibilidad (CIA).
- **Técnicas**: TLS 1.3, autenticación (OAuth2), backups, Zero Trust.
- **Amenazas**: malware, phishing, ransomware, ataques DDoS.
- **Prevención**: firewall, copias de seguridad, segmentación de red.
- **Copia seguridad**: Completa, Incremental, Diferencial. Regla 3-2-1: 3 copias, 2 medios, 1 externa.

8. Sistemas Cloud híbrido

- Infraestructura que combina recursos **locales y en la nube**.
- Casos comunes: bases de datos locales + backups en cloud.
- Herramientas: **Azure Arc, AWS Outposts, Proxmox + S3**.
- Ventajas: flexibilidad, escalado, continuidad operativa.

9. Resiliencia y tolerancia a fallos

- **Redundancia**: duplicación de componentes críticos.
- **Clustering**: varios servidores colaborando (alta disponibilidad).

- **Failover:** cambio automático a un nodo de respaldo.
- **Ejemplos:** RAID, balanceadores de carga, centros de datos tolerantes a fallos.

10. Aplicaciones reales

- **Sanidad:** historia clínica electrónica, IA diagnóstica.
- **Educación:** LMS, aulas virtuales, contenidos digitales.
- **Industria:** IoT, control de procesos, mantenimiento predictivo.
- **Transporte:** sensores, logística inteligente, conducción autónoma.

11. Conclusión

- Los sistemas informáticos son la **base tecnológica** de la sociedad actual.
- Su evolución va hacia la **inteligencia distribuida**, la **eficiencia energética** y la **gestión ética** de la información.
- Comprender su estructura física y lógica permite diseñar, administrar y proteger infraestructuras tecnológicas reales.

2. PARTE DIDÁCTICA

Módulo profesional: Servicios en Red (2.º SMR)

2.1 Contextualización

Este módulo pertenece al segundo curso del Ciclo Formativo de Grado Medio en Sistemas Microinformáticos y Redes (SMR).

El alumnado trabaja sobre redes reales y virtuales en un entorno orientado a la práctica profesional. Se parte de conocimientos básicos de redes, sistemas operativos y configuración de servicios.

2.2 Objetivos de aprendizaje

- Comprender la **estructura física y funcional** de un sistema informático en red.
- Diseñar e implementar una **red local con servicios básicos**.
- Aplicar principios de **organización, seguridad y documentación técnica**.
- Usar entornos virtuales para probar, corregir y optimizar configuraciones.

2.3 Metodología

- Aprendizaje basado en tareas prácticas y simulación.
- Trabajo cooperativo por grupos reducidos.
- Uso de herramientas como **VirtualBox, GNS3** o redes reales del aula.
- Desarrollo guiado con fases: planificación → montaje → pruebas → entrega

2.4 Atención a la diversidad (niveles III y IV)

- **Nivel III:** esquemas guiados, simuladores preconfigurados, checklist técnico paso a paso.
- **Nivel IV:** autonomía en el diseño, configuración avanzada y reto opcional (ej.: añadir control remoto).
- Materiales adaptados: vídeos, tutoriales, apoyo personalizado.

2.5 Diseño Universal para el Aprendizaje (DUA)

- **Representación múltiple:** diagramas de red, mapas IP, visualización de servicios.
- **Expresión variada:** presentación oral del proyecto, documentación técnica o maqueta virtual.
- **Implicación realista:** simulación contextualizada (empresa ficticia)

2.6 Actividad principal

Título: “Planifica y configura la red de tu empresa”

Descripción:

Cada grupo diseña e implementa una red básica para una empresa simulada (taller, tienda, academia...).

Deberán:

- Planificar:
 - Dirección IP (rango IPv4 privado).
 - Esquema físico y lógico de red: switches, routers, servidores.
- Implementar (simulado o real):
 - Servicios: DHCP, DNS, FTP/Samba, servidor web o de correo local.
 - Acceso compartido entre equipos.
 - Medidas básicas de seguridad: firewall, contraseñas, permisos.
- Documentar:
 - Justificación técnica, esquemas, pasos de configuración.
- Presentar el resultado al grupo clase.

2.7 Evaluación. Instrumentos y criterios

Tema 22 Planificación y explotación de sistemas informáticos. Configuración. Condiciones de instalación. Medidas de seguridad. Procedimientos de uso.

1. Introducción

Los sistemas informáticos requieren una planificación cuidadosa, una implantación bien estructurada y una explotación mantenible. Se aborda el ciclo completo: desde el diseño hasta la gestión diaria del sistema, incluyendo la configuración, instalación física, medidas de seguridad y normas de uso.

2. Planificación del sistema informático

- **Análisis de necesidades:** usuarios, servicios, red, software.
- **Diseño físico y lógico:** topología, direccionamiento IP, servicios.
- **Selección de arquitectura:** cliente-servidor, híbrida o distribuida.
- **Presupuesto, escalabilidad, licencias, software según el contexto:**
 - Ofimática, programación, diseño gráfico, navegación, gestión.
- **Planificación de software libre y/o propietario.**
- **Planificación de usuarios y estructura organizativa:** departamentos, permisos, grupos.

3. Condiciones de instalación

- **Ubicación de equipos:** racks, salas ventiladas, puestos ergonómicos.
- **Cableado estructurado y conectividad:** routers, switches, puntos de acceso.
- **Alimentación eléctrica:** SAI, organización del cableado.
- **Instalación de sistemas operativos** (Windows, Linux, dual boot).
- **Instalación del software base** según perfil de usuario.

4. Configuración del sistema

- **Configuración de red:** DNS, DHCP, rutas.
- **Servicios de archivos:** FTP, Samba, Torrent.
- **Servidores de contenido:** web, correo, herramientas internas.
- **Virtualización básica** (VirtualBox, Proxmox, Docker).
- **Contenedores con Docker** (despliegue rápido de servicios).
- **Automatización de configuraciones con Salt Project.**
- **Integración con servicios externos** (nube, DNS público, copias remotas).

5. Explotación y mantenimiento

- **Creación de usuarios:** cuentas locales y en red.
- **Active Directory:** dominio, políticas de grupo, perfiles.
- **Mantenimiento preventivo:** limpieza, revisión de logs, actualizaciones.
- **Copias de seguridad:** planificación y verificación.
- **Monitorización:** herramientas (ping, top, dashboards), análisis de métricas.
- **Automatización de tareas rutinarias** (cron, scripts, Salt).
- **Optimización del rendimiento**
 - Control de procesos activos y servicios innecesarios, Ajustes del arranque y limpieza de archivos temporales. Monitorización del uso de CPU, RAM y disco

6. Medidas de seguridad

- **Seguridad lógica:** contraseñas seguras, permisos, actualizaciones.
- **Seguridad de red:** firewall, NAT, puertos, segmentación (VLAN, Wi-Fi invitados).
- **Seguridad física:** acceso restringido, cámaras, SAI, control de periféricos.
- **Registro de eventos:** logs de sistema, alertas, historial de accesos.
- **Protección frente a amenazas:** antivirus, copias de seguridad, políticas de uso.
- **Copias de seguridad:** Incremental, Diferencial, Completa, Técnica 3 copias, 2 dispositivas, 1 lugar diferente.
- **Plan de contingencia**
 - Restauración desde copias de seguridad previamente verificadas.
 - Uso temporal de sistemas alternativos (servidores secundarios, contenedores).
 - Guía básica de actuación en caso de caída de red o servicios críticos.
 - Designación clara de responsables ante incidencias.

7. Procedimientos de uso

- **Normas internas:** uso del sistema, horarios, navegación, correo.
- **Gestión de usuarios y permisos:** altas, bajas, asignación de funciones.
- **Guías de usuario:** acceso a carpetas, impresión, correo interno.
- **Registro y gestión de incidencias:** tickets, escalado, documentación.
- **Documentación técnica:**
 - Esquemas físicos y lógicos. Ficha de software y configuración.
 - Inventario de hardware.
 - Manual del administrador y del usuario.

8. Conclusión

La planificación y explotación eficaz de un sistema informático requiere una visión integral: técnica, organizativa y de seguridad. La configuración de servicios debe adaptarse al contexto, mientras que las condiciones de instalación y los procedimientos internos garantizan un entorno funcional y sostenible. El uso de herramientas como Docker y Salt Project aporta automatización, eficiencia y profesionalidad.

2. PARTE DIDÁCTICA

Finalidad: Comprender cómo planificar, instalar, configurar y mantener los servicios de red necesarios para el funcionamiento de un sistema informático completo en un entorno simulado.

2.2 Objetivos de aprendizaje

- Identificar necesidades técnicas en una red local.
- Planificar la instalación y condiciones del sistema informático.
- Desplegar servicios funcionales (DHCP, DNS, FTP, compartición de archivos...).
- Aplicar medidas de seguridad y realizar una correcta documentación técnica.
- Simular procedimientos de uso y recuperación ante fallos.

2.3 Metodología

- Aprendizaje por proyectos técnicos reales.
- Trabajo en parejas o pequeños grupos.
- Uso de **VirtualBox**, **Debian** y **Windows Server** como entornos virtuales.
- Incorporación de herramientas como **Docker** (servicios ligeros) y **Salt Project** (automatización básica) según nivel.
- Evaluación continua a través del desarrollo del proyecto y documentación.

2.4 Atención a la diversidad (niveles III y IV)

- **Nivel III:** prácticas guiadas paso a paso, configuraciones predefinidas, esquemas base.
- **Nivel IV:** configuración autónoma, reto opcional (desplegar un servicio con Docker o aplicar automatización con Salt).
- Adaptación de ritmos y acompañamiento técnico diferenciado.

2.5 DUA (Diseño Universal para el Aprendizaje)

Representación: esquemas de red, diagramas funcionales, videotutoriales y plantillas.

Acción y expresión: instalación y configuración práctica, entrega de documentación, defensa técnica del trabajo.

Implicación: contexto realista (empresa ficticia), con roles definidos (técnico de red, responsable de seguridad...).

2.6 Actividad principal

Supuesto didáctico: “Despliegue y mantenimiento de servicios para una empresa ficticia”

Contexto: La clase simula una empresa (tienda, oficina, clínica...) que necesita una red informática con servicios esenciales.

Fases del proyecto:

Diseño técnico

- Plano de red física (topología, cableado, puntos de red).
- Plan de direccionamiento IP.
- Determinación de servicios requeridos.

Instalación

- Montaje en VirtualBox de servidores Linux o Windows.
- Instalación del sistema operativo y software requerido.
- **Configuración de servicios**
 - Servidor **DHCP y DNS** para la red local.
 - Servidor de **archivos (FTP o Samba)** para el personal.
 - **Servidor web** o servicio contenedorizado (Docker).
 - **Automatización básica** de configuración con Salt (opcional).

Medidas de seguridad

- Activación de cortafuegos.
- Gestión de usuarios y permisos.
- Plan básico de contingencia (restauración desde backup).

2.7 Evaluación. Instrumentos. Criterios

Tema 23. Diseño de algoritmos. Técnicas descriptivas.

1. Introducción al diseño algorítmico

- Un **algoritmo** es una secuencia ordenada y finita de pasos para resolver un problema.
- **Propiedades esenciales:**
 - **Precisión:** cada paso debe estar definido sin ambigüedad.
 - **Determinismo:** mismo resultado ante las mismas entradas.
 - **Efectividad:** cada paso debe ser ejecutable en un tiempo finito.
 - **Finito:** termina en un número limitado de pasos.
- **Ámbitos de aplicación:** IA, videojuegos, Big Data, ciberseguridad, sistemas embebidos, automatización.

2. Construcción y estructura de algoritmos

- **Elementos básicos:**
 - Instrucciones: asignación, E/S, operadores.
 - Control de flujo: secuencia, condicionales (**if**, **switch**), bucles (**for**, **while**).
 - Modularidad: uso de funciones y procedimientos.
 - Validación de datos: comprobar entradas antes de procesarlas.
- **Buenas prácticas:**
 - Evitar redundancias.
 - Comentar código.
 - Dividir el algoritmo en pasos o módulos lógicos.

3. Representación de algoritmos

3.1 Pseudocódigo

- Lenguaje intermedio entre lenguaje natural y programación.
- Fácil de entender, sin sintaxis formal.

3.2 Diagramas de flujo

- Representación visual de decisiones, bucles y operaciones.
- Útil en fases iniciales del diseño.

3.3 Diagramas Nassi-Shneiderman

- Alternativa estructurada a los diagramas de flujo.
- Favorece el diseño modular

3.4 Tablas de decisión

- Útiles cuando hay múltiples condiciones que afectan las decisiones..

4. Herramientas de apoyo al diseño algorítmico

- **Entornos visuales educativos:** Scratch, Blockly, App Inventor.
- **IA y LLMs:** herramientas como ChatGPT o GitHub Copilot ayudan a generar, explicar y depurar código.
- **Prototipado interactivo:** herramientas como Figma permiten simular el flujo algorítmico de pantallas e interacciones (para apps o sistemas interactivos).

5. Metodología descriptiva de diseño

Diseñar algoritmos implica analizar el problema y describir la solución paso a paso de forma clara:

- **Análisis:** identificar entradas, salidas y restricciones.
- **Estructuración lógica:** usar secuencia, condicionales (**if**, **switch**) y bucles (**for**, **while**).
- **Modularidad** (Top-Down): dividir el problema en funciones pequeñas y comprensibles.
- **Selección de estructuras de datos:** arrays, listas, pilas, árboles, según convenga al problema.
- **Pruebas:** verificar con casos típicos, extremos y errores. Se pueden usar técnicas como prueba de caja negra (entradas/salidas) y caja blanca (seguimiento interno de que todo el código vaya como toca).

6. Representación de estrategias algorítmicas comunes

Algunas estrategias algorítmicas siguen patrones comunes que pueden describirse fácilmente con pseudocódigo, diagramas o ejemplos paso a paso.

La elección adecuada de estructuras de datos (arrays, listas, pilas, diccionarios...) es clave para representar la lógica del algoritmo de forma clara y comprensible.

- **Divide y vencerás:** se representa mediante funciones recursivas bien estructuradas, con llamadas a subproblemas (ej.: ordenación rápida).
- **Programación dinámica:** se describe con tablas o vectores que almacenan subresultados, destacando la reutilización eficiente (ej.: Fibonacci).
- **Voraz (Greedy):** se puede mostrar como una secuencia de decisiones locales óptimas con condiciones claras (ej.: Dijkstra).

- **Backtracking:** se representa como un árbol de decisiones, marcando rutas exploradas y retrocesos (ej.: N-reinas, Sudoku).
- **Evolutivos y heurísticos:** se explican mejor mediante esquemas de evolución progresiva de soluciones, usando poblaciones o reglas aproximadas.

7. Eficiencia algorítmica

- **Notación Big-O:** mide el crecimiento del tiempo o espacio requerido en función del tamaño de entrada n .
- **Casos comunes:**
 - $O(1)$: acceso a un array.
 - $O(n)$: recorrer una lista.
 - $O(n \log n)$: ordenaciones eficientes (MergeSort).
 - $O(n^2)$: algoritmos ingenuos (burbujas).
 - $O(n!)$: permutaciones (Backtracking).
- Considerar:
 - **Peor caso, mejor caso, caso promedio.**
 - **Complejidad temporal vs espacial.**

8. Ética y limitaciones de los algoritmos

- **Sesgos** en los datos → decisiones injustas. Transparencia y explicabilidad.
- Responsabilidad del programador en entornos sensibles (salud, justicia, finanzas)

9. Conclusión

- Pensar algorítmicamente es esencial para resolver problemas de forma estructurada, escalable y eficiente.
- La conexión entre **problema** → **lógica** → **código** → **experiencia de usuario** es clave.
- Las tendencias actuales (IA, computación cuántica, programación visual) redefinen el modo en que diseñamos y utilizamos algoritmos.

2. PARTE DIDÁCTICA (MÓDULO: PROGRAMACIÓN)

2.1 Contextualización

- **Nivel:** 1º DAM. **Módulo:** Programación.
- **Perfil:** alumnado en formación inicial, enfocado en resolver problemas lógicos mediante código y representación visual.

2.2 Objetivos de aprendizaje

- Comprender qué es un algoritmo y su utilidad práctica.
- Representar soluciones con pseudocódigo, diagramas y entornos visuales.
- Aplicar la lógica algorítmica a tareas reales de programación.
- Relacionar algoritmo → código → experiencia visual (prototipo).

2.3 Metodología

- Aprendizaje activo con resolución de problemas contextualizados.
- Secuencia guiada: planteamiento → pseudocódigo → diagrama → código/prototipo.
- Alternancia entre papel, pizarra, herramientas digitales (Scratch, Python, Figma).

2.4 Atención a la diversidad (niveles III y IV)

- Actividades escalonadas (resolución básica vs. lógica compleja).
- Soporte visual y verbal: rúbricas, videotutoriales, plantillas de diagramas.
- Ritmos diferenciados en la representación y programación.

2.5 DUA

- Múltiples formas de representación (diagrama, código, prototipo).
- Expresión: oral, escrita, gráfica.
- Compromiso: problemas cercanos al alumnado (apps, juegos, decisiones).

2.6 Actividad principal

Supuesto didáctico: “Del problema a la pantalla”

- Fase 1: plantear un problema cotidiano (e.g., calcular descuentos, gestión de turnos, app de reservas).
- Fase 2: representar solución en:
 - Pseudocódigo
 - Diagrama de flujo o Nassi-Shneiderman
- Fase 3: codificar en Python).
- Fase 4: diseñar un prototipo en Figma que represente el flujo visual del algoritmo (pantallas, botones, decisiones).

2.7 Evaluación. Instrumentos. Criterios

Tema 24. Lenguajes de programación. Tipos. Características.

1.1 Introducción

- Un lenguaje de programación es un sistema formal para expresar algoritmos.
- Sirve de puente entre la lógica del programador y el funcionamiento de la máquina.
- Su evolución responde a nuevas necesidades: eficiencia, mantenibilidad, inteligencia artificial, desarrollo web, etc.

1.2 Elementos fundamentales

- **Sintaxis:** reglas que determinan cómo debe escribirse un programa.
- **Semántica:** significado de las instrucciones escritas.
- **Estructuras de control:**
 - Secuencia
 - Condicionales: **if**, **switch**
 - Bucles: **for**, **while**
- **Tipos de datos:**
 - Primitivos: **int**, **char**, **bool**
 - Estructurados: **arrays**, **struct**
 - Abstractos: listas, pilas, árboles, diccionarios
- **Tipado:**
 - Estático: definido en tiempo de compilación (C, Java)
 - Dinámico: definido en tiempo de ejecución (Python, JS)
 - Fuerte: no permite conversiones implícitas inseguras (Python)
 - Débil: permite conversiones implícitas (JS)
- **Cualidades deseables en un lenguaje:**
 - Legibilidad
 - Seguridad
 - Eficiencia
 - Portabilidad

1.3 Paradigmas de programación

- **Imperativo:** indica cómo debe hacerse una tarea (C, Python)
- **Declarativo:** indica qué se desea lograr sin especificar cómo (SQL)
- **Funcional:** se basa en funciones matemáticas puras, sin efectos secundarios ni variables mutables (ej: Haskell).
- **Lógico:** se programa mediante hechos y reglas, dejando que el motor lógico deduzca soluciones (ej: Prolog).
- **Orientado a objetos:** estructura el código en objetos que combinan datos y métodos (Java, Python)
- **Reactivo:** responde a eventos y cambios (JavaScript con Vue.js)
- **Tiempo real:** requiere respuestas inmediatas y predecibles (C embebido)
- **Cuántico:** trabaja con qubits, superposición y entrelazamiento (Q#)

1.4 Clasificación

- **Por nivel de abstracción:**
 - Bajo nivel: ensamblador
 - Medio nivel: C, Rust
 - Alto nivel: Python, Java
- **Por forma de ejecución:**
 - Compilados: traducidos antes de ejecutarse (C, C++)
 - Interpretados: ejecutados línea a línea (Python, JS)
 - Híbridos: combinan compilación e interpretación (Java, C#)
 - Transpilados: convertidos a otro lenguaje (TypeScript → JavaScript)
- **Por generación:**
 - Lenguajes clásicos: Pascal, COBOL
 - Lenguajes modernos: Kotlin, Go, Rust
 - Lenguajes emergentes: Q#, Cirq, Mojo

1.5 Lenguajes y usos

- **C/C++:** desarrollo de sistemas embebidos, controladores, software de alto rendimiento
- **Java:** aplicaciones empresariales, desarrollo Android
- **Python:** inteligencia artificial, ciencia de datos, automatización
- **JavaScript:** desarrollo web frontend y backend
- **SQL:** gestión de bases de datos relacionales
- **Q#:** programación cuántica

1.6 Herramientas

- **Compiladores:** gcc, javac
- **Intérpretes:** python, node
- **Entornos de desarrollo (IDEs):** Visual Studio Code, IntelliJ, Replit (en la nube)

1.7 Tendencias actuales

- **Cloud-native:** desarrollo pensado para la nube (Python, Go)
- **Inteligencia artificial y machine learning:** uso de bibliotecas como TensorFlow (Python)
- **Plataformas no-code/low-code:** herramientas visuales como Appgyver o Glide
- **Programación cuántica:** desarrollo con lenguajes como Q# y Cirq
- **Colaboración remota:** trabajo en equipo en línea con herramientas como GitHub Codespaces, CoLab y Live Share (Visual Studio Code)

1.8 Conclusión

- No existe un lenguaje único para todo; la elección depende del objetivo.
- Hoy en día conviven múltiples paradigmas de programación.
- La tendencia es hacia herramientas accesibles y entornos colaborativos.

1.9 Criterios para elegir un lenguaje

- **Tipo de proyecto:** web, IA, sistemas, móviles, etc.
- **Rendimiento:** si se requiere eficiencia y control del hardware.
- **Curva de aprendizaje:** facilidad de uso para el programador.
- **Portabilidad:** posibilidad de usarlo en múltiples plataformas.
- **Productividad y ecosistema:** existencia de herramientas, bibliotecas y frameworks.
- **Comunidad y soporte:** documentación, foros y actualizaciones.
- **Mantenibilidad:** claridad y estructura del código.
- **Seguridad:** capacidad para prevenir errores y gestionar memoria.

2. PARTE DIDÁCTICA

Nivel: 1.º DAM **Módulo:** Programación

Perfil: alumnado con conocimientos básicos en algoritmia y estructuras de control, en fase de exploración de distintos lenguajes y entornos, con orientación a desarrollar soluciones en lenguajes estructurados como Java.

2.2 Objetivos de aprendizaje

- Comprender los tipos y características de los lenguajes de programación.
- Identificar sus paradigmas, niveles de abstracción y ámbitos de aplicación.
- Aplicar estructuras de control en la resolución de problemas mediante Java.
- Desarrollar soluciones funcionales, claras y mantenibles.
- Simular un proceso técnico realista (entrevista + prueba práctica).

2.3 Metodología

- Simulación de un proceso técnico profesional: entrevista a un lenguaje + encargo de desarrollo.
- Representación de conocimientos en formato guion técnico y programa funcional.
- Programación en Java como lenguaje base.
- Trabajo cooperativo: análisis, desarrollo, presentación y revisión crítica.
- Aprendizaje activo: “aprender haciendo” mediante reto realista.

2.4 Atención a la diversidad (niveles III y IV)

- Fichas-guía para guiar la entrevista y la estructura del programa.
- Actividad dividida en fases con complejidad creciente.
- Apoyo visual y plantillas para facilitar la documentación y codificación.
- Seguimiento individual y revisión entre pares según necesidades detectadas.

2.5 DUA

- **Representación:** entrevista como analogía laboral, código comentado, esquemas del flujo del programa.

2.6 Actividades principales

Actividad: “Entrevista técnica a Java + reto de gestión de turnos”

- El alumnado crea y simula una entrevista técnica al lenguaje Java, presentando sus características (paradigma, tipado, herramientas, aplicaciones) de forma crítica y creativa. Después, desarrolla en Java un programa de gestión de turnos con funcionalidades básicas, que debe adaptar tras recibir una petición de mejora, como búsqueda por nombre o priorización de turnos.

2.7. Evaluación. Instrumentos. Criterios.

Tema 25. Programación estructurada. Estructuras básicas. Funciones y Procedimientos.

1.1 Introducción

- Paradigma que organiza el código de forma lógica, secuencial y modular.
- Base técnica y pedagógica para aprender POO, desarrollo web o scripting.
- Aporta claridad, facilidad de mantenimiento, comprensión y depuración.

1.2 Origen y objetivos

- Propuesta por Dijkstra en los años 70 como alternativa al uso de `goto`.
- Favorece código legible, reusable y testeable.
- Beneficios: reducción de errores, mayor control del flujo, colaboración efectiva.

1.3 Estructuras básicas de control

- **Secuencia:** ejecución ordenada de instrucciones.
- **Selección:** decisiones (`if`, `else`, `switch-case`, ternario).
- **Iteración:** bucles (`for`, `while`, `do-while`)

Ejemplo en Java:

```
if (nota >= 5) {  
    System.out.println("Aprobado");  
} else {  
    System.out.println("Suspenso");  
}
```

1.4 Funciones y procedimientos

- **Funciones:** devuelven un valor con `return`, encapsulan lógica reusable.
- **Procedimientos:** tareas sin retorno (métodos `void` en Java).
- **Parámetros:**
 - En Java, siempre por valor (los objetos permiten modificar contenido).
- **Ámbito:** variables locales y globales (nivel de clase o bloque).
- **Recursividad:** la función se llama a sí misma; requiere caso base.

1.5 Diseño modular

- Organización en métodos con tareas concretas.
- Reutilización, mantenimiento y pruebas facilitadas.
- **Buenas prácticas:**
 - Funciones pequeñas y bien nombradas.
 - Evitar duplicación de código.
 - Uso coherente de `return` para controlar el flujo.

1.6 Diseño top-down, cohesión y acoplamiento (nuevo)

- **Diseño top-down:**
 - Se parte de una visión general del problema y se divide en subproblemas.
 - Se diseña de lo general a lo específico, creando funciones que resuelven partes concretas.
 - Facilita la planificación del código antes de escribirlo (diagramas, pseudocódigo por niveles).
- **Cohesión:**
 - Mide cuán relacionadas están las tareas dentro de una función.
 - Alta cohesión: cada función hace una sola cosa clara → facilita lectura y mantenimiento.
- **Acoplamiento:**
 - Mide la dependencia entre funciones o módulos.
 - Bajo acoplamiento: funciones independientes, bien aisladas → mejora reutilización y prueba.
- **Relación con la estructuración:** ambos principios permiten construir programas modulares, comprensibles y escalables.

1.7 Trazado y depuración estructurada

- La estructura lógica del código permite seguir el flujo paso a paso.
- Herramientas: `System.out.println`, depurador del IDE.
- Identificación de errores mediante revisión de condiciones, valores y recorridos.
- Indentación y comentarios como apoyo a la comprensión.

1.8 Buenas prácticas en programación estructurada

- Evitar funciones extensas o muy anidadas.
- Nombrar bien variables y métodos.
- Modularizar para facilitar pruebas y mantenimiento.
- Documentar el código cuando sea necesario, sin exceso.

1.9 Eficiencia algorítmica

- Comparación de soluciones estructuradas: iterativa vs. recursiva.
- Ejemplos típicos: factorial, búsqueda en listas, cálculo de sumas.
- Introducción a la **notación Big-O** para estimar el coste computacional.
- Técnicas básicas: **refactorización** de código para optimizar recursos, y **memoización** para evitar cálculos repetidos.
- Evaluar cuándo una solución clara y estructurada también es eficiente.

1.10 Comparación con otros paradigmas

- **Estructurada:**
 - Unidad: funciones
 - Uso: lógica y algoritmos
 - Datos: acceso directo
- **POO:**
 - Unidad: clases y objetos
 - Uso: aplicaciones grandes
 - Datos: encapsulados y distribuidos

2. PARTE DIDÁCTICA (MÓDULO: PROGRAMACIÓN – 1º DAM)

Nivel: 1º DAM. Módulo: Programación.

- **Perfil:** alumnado en formación inicial en desarrollo de software, que necesita dominar la lógica estructurada antes de pasar a la POO.

2.2 Objetivos de aprendizaje

- Comprender y aplicar estructuras básicas.
- Implementar funciones con parámetros y retorno.
- Dividir el código en bloques reutilizables y legibles.
- Evaluar soluciones eficientes con control de errores.

2.3 Metodología

- Codificación práctica en Java desde el primer momento.
- Enfoque descendente: problema → algoritmo → código modular.
- Actividades individuales y en grupo con retos progresivos.

2.4 Atención a la diversidad (niveles III y IV)

- Proyectos escalables por dificultad.
- Plantillas de código base.
- Seguimiento con feedback personalizado en IDE.

2.5 DUA

- Representación: pseudocódigo + código real + diagramas.
- Expresión: defensa de funciones, documentación técnica.
- Compromiso: gamificación, retos competitivos y cercanos.

2.6 Actividad principal

Supuesto didáctico: “Diseña tu algoritmo estrella: el reality show de la eficiencia”

Supuesto didáctico: “Diseña tu algoritmo estrella: el reality show de la eficiencia”

- **Fase 1:** Se asigna a cada grupo un **reto cotidiano** (por ejemplo: sistema de gestión de turnos, reservas de sala o control de acceso).
- **Fase 2:** Diseño del algoritmo mediante estructuras de control (**if**, **for**, **while**) y funciones. Se incluye validación de entradas y planificación modular.
- **Fase 3:** Codificación de la solución en **Java**, aplicando principios de programación estructurada: funciones reutilizables, variables locales, retorno de valores, etc.
- **Fase 4: Simulación de entrevista técnica en Java:**
 - Cada grupo elige **una o varias funciones clave** de su programa y las presenta como si fueran “candidatos a ser contratados”.
 - Se responde a preguntas como:
 - ¿Qué hace esta función?
 - ¿Por qué está bien diseñada (cohesión, claridad)?
 - ¿Es eficiente? ¿Iterativa o recursiva?
 - ¿Qué la hace especial frente a otras soluciones?
 - Se fomenta la **justificación técnica y la reflexión crítica** sobre las decisiones de diseño.
- **Fase 5:** Ejecución y prueba en un **entorno de desarrollo real** (Eclipse o IntelliJ), permitiendo mostrar el funcionamiento del algoritmo y validar su comportamiento con datos de prueba.

2.7 Evaluación. Instrumentos. Criterios

Tema 26. Programación modular. Diseño de funciones. Recursividad. Librerías.

1.1 Introducción a la modularidad

- Divide proyectos en **módulos independientes y cohesivos**.
- Facilita **mantenimiento, reutilización, tests y colaboración entre equipos**.
- Aumenta la claridad del código y permite aislar errores más fácilmente.
- 🧠 *Anécdota didáctica*: error de pila por recursión sin caso base → importancia de controlar bien los flujos y las llamadas.

1.2 Fundamentos de módulos

- Un **módulo** es una unidad de código con una **interfaz pública clara** y una lógica interna **aislada**.

```
# modulo_usuario.py
def registrar(usuario):
    return True
```

- **Alta cohesión**: cada módulo cumple un propósito claro y concreto.
- **Bajo acoplamiento**: los módulos dependen poco entre sí.
- La comunicación entre módulos se realiza mediante **interfaces limpias**: parámetros, eventos o retorno de valores.

1.3 Diseño de funciones

- Aplicación de los principios **DRY (Don't Repeat Yourself)** y **SRP (Single Responsibility Principle)**: cada función debe hacer **una sola cosa**.
- Usar **parámetros claros (preferiblemente ≤ 3)**. Si hay más, agruparlos en objetos.
- Preferencia por **variables locales** para evitar efectos colaterales invisibles.
- **Refactorización**: dividir funciones complejas en subrutinas bien nombradas y especializadas.
- Organización del proyecto en carpetas lógicas (sugerencias):
 - **controllers**: lógica de control
 - **services**: funciones intermedias
 - **utils**: utilidades y funciones auxiliares

1.4 Recursividad

- Técnica de definición de funciones que **se llaman a sí mismas** para resolver subproblemas.
- Estructura básica:
 - **Caso base**: condición que detiene la recursión.
 - **Caso recursivo**: divide el problema en una versión más simple de sí mismo

```
def factorial(n):
    return 1 if n == 0 else n * factorial(n - 1)
```

- **Optimización de recursividad**:
 - **Memorización**: almacenamiento de resultados ya calculados para evitar.
- **Comparativa: recursiva vs. iterativa**
- La **recursividad** es más clara para problemas como árboles o algoritmos de backtracking, donde se repite una estructura similar. La **iteración**, en cambio, es más eficiente y adecuada para tareas repetitivas simples, como bucles. Mientras la recursión puede consumir más memoria, la iteración ofrece mayor control y estabilidad en ejecución

1.5 Librerías y reutilización

- **Librerías estándar** (integradas) y **externas** (instalables con gestores como **pip** o **npm**).
- Buenas prácticas de documentación:
 - **README.md**, docstrings, comentarios útiles
 - Versionado semántico: **mayor.menor.parche** (ej.: **2.3.1**)
- **Empaquetado** por ecosistema:
 - Python: **setup.py**, **__init__.py**, **PIP**
 - JavaScript: **package.json**, **NPM**
- Consejos de seguridad y mantenimiento:
 - Evitar dependencias innecesarias
 - Verificar actualizaciones y reputación de paquetes externos

1.6 Modularidad en sistemas reales

- Aplicación de principios modulares en arquitecturas modernas:
 - **Capas lógicas**: presentación, lógica de negocio, acceso a datos
 - **Microservicios**: cada servicio cumple una función única y se comunica por red
 - **REST/API**: comunicación entre servicios o componentes por HTTP
 - **Tests unitarios**: herramientas como **pytest**, **Jest**, **unittest**

1.7 Diseño algorítmico modular

- **Aplicación directa de modularidad a la resolución de problemas algorítmicos.**
- Separar el algoritmo principal en funciones que cumplen sub-tareas:
 - **Divide y vencerás** (ej.: mergesort, búsqueda binaria)
 - **Backtracking** (ej.: sudoku, laberinto)
 - **Greedy** (ej.: cambio óptimo, planificación)
- Técnica de diseño:
 - Función principal como “controladora”
 - Funciones auxiliares con tareas específicas
- Apoyarse en pseudocódigo o diagramas modulares antes de codificar.
- Beneficios: claridad, eficiencia, fácil prueba y mantenimiento.

1.8 Pruebas y verificación de funciones

- Cada función diseñada debe ser **comprobable individualmente**.
- Introducción al **testing unitario**:
 - En Python: **assert**, **pytest**, **unittest**
 - En JavaScript: **describe**, **test**, **expect** (Jest)
- Probar distintos tipos de entradas:
 - Casos típicos. Casos límite. Casos inválidos
- Relación directa con modularidad: **funciones pequeñas son más fáciles de testear**.

1.9 Conclusión científica

- La **modularidad** es una competencia clave en el desarrollo de software profesional.
- La **recursividad** es una herramienta poderosa cuando se domina su uso.

2. PARTE DIDÁCTICA

- **Nivel:** 1º DAM. **Módulo:** Programación.
- **Perfil:** futuro desarrollador de software con formación en arquitectura limpia y buenas prácticas de construcción de aplicaciones modulares.

2.2 Objetivos de aprendizaje

- Diseñar módulos con alta cohesión y bajo acoplamiento.
- Crear funciones limpias, eficientes y fáciles de mantener.
- Aplicar recursividad de forma controlada.
- Utilizar y construir librerías propias.

2.3 Metodología

- Talleres: de monolito a módulos.
- Desarrollo iterativo con herramientas reales (VS Code, Python).
- Revisión de código colaborativa (peer review).

2.4 Atención a la diversidad (niveles III y IV)

- Pauta por capas: modularización progresiva.
- Recursos de ayuda: plantillas, tutoriales modular en vídeo.
- Feedback directo en IDE para mejorar diseño.

2.5 DUA

- Representación: diagramas de componentes, composición modular.
- Acción: desarrollo de librerías y pruebas.
- Expresión: documentación técnica, defensa de decisiones.
- Compromiso: retos con gamificación (“función estrella”).

2.6 Actividad principal

Concurso “¡Modula y vencerás!”

- **Fase 1:** Cada grupo recibe un encargo realista (gestión de reservas, cálculo de descuentos, validación de datos, generación de informes o exploración de estructuras anidadas).
- **Fase 2:** Diseñan la solución modular, definiendo componentes funcionales (usuarios, pagos, reservas...) y aplicando el principio SRP en funciones. Se promueve el uso de **recursividad** en funciones donde sea adecuada, como búsqueda jerárquica, análisis de rutas o cálculos acumulativos.
- **Fase 3:** Seleccionan y presentan su “**función estrella**”, destacando qué hace, cómo está optimizada, y por qué es un ejemplo de buen diseño modular.
- **Fase 4:** Justifican la elección entre **recursividad o iteración**, y explican su impacto en la claridad y eficiencia del código.
- **Fase 5:** Demuestran la ejecución real del sistema y presentan la arquitectura modular con apoyo gráfico o visual.

2.7 Evaluación. Instrumentos. Criterios de evaluación.

Tema 27. Programación orientada a objetos. Objetos. Clases. Herencia. Polimorfismo. Lenguajes.

1.1 Introducción y motivación

- La programación orientada a objetos (POO) modela el mundo real agrupando **estado (atributos)** y **comportamiento (métodos)** en estructuras llamadas objetos.
- Permite desarrollar sistemas **más escalables, mantenibles y reutilizables** que la programación estructurada..

1.2 Evolución histórica

- En la programación **pre-POO** (C, Pascal), los datos y la lógica estaban separados.
- La POO surgió con lenguajes como **Smalltalk, C++ y Objective-C**, que permitían combinar ambos conceptos.
- Se consolidó con lenguajes modernos como **Java, C#, Kotlin**, que incluyen **gestión automática de memoria** y fuertes modelos de tipos.

1.3 Ventajas clave

- **Abstracción:** permite representar entidades del mundo real como clases.
- **Encapsulamiento:** oculta los detalles internos del objeto, protegiendo su estado.
- **Herencia:** permite reutilizar código mediante especialización.
- **Polimorfismo:** permite usar objetos diferentes a través de la misma interfaz.

1.4 Clases y objetos

- Una **clase** es un molde o plantilla; un **objeto** es una instancia concreta.
 - Ejemplo: `Coche miTesla = new Coche();`
- **Atributos:**
 - De instancia: pertenecen a cada objeto.
 - Estáticos: comunes a todos los objetos de la clase.
 - Constantes: valores fijos (`final`, `const`).
- **Métodos:**
 - De instancia: operan sobre atributos del objeto.
 - Estáticos: no dependen del objeto.
 - Getters/setters: permiten acceder y modificar atributos de forma controlada.

1.5 Visibilidad y encapsulamiento

- Modificadores de acceso:
 - `public`: accesible desde cualquier parte.
 - `private`: solo accesible desde la propia clase.
 - `protected`: accesible desde la clase y sus subclases

```
private int velocidad;
public void setVelocidad(int v) {
    if (v >= 0) velocidad = v;
}
```

- Permite proteger el estado y controlar cómo se modifican los datos.

1.6 Ciclo de vida de un objeto

- **Creación:** mediante constructor (por defecto o personalizado).
- **Inicialización:** se asignan valores iniciales.
- **Uso:** el objeto ejecuta métodos, modifica atributos.
- **Finalización:** el objeto es eliminado cuando ya no se usa (recolector de basura en Java, `__del__` en Python).

1.7 Relaciones entre clases

- **Asociación:** relación general entre clases (ej.: un `Profesor` da clases a un `Alumno`).
- **Agregación:** una clase contiene a otra, pero cada una puede existir por separado (ej.: una `Universidad` tiene `Departamentos`).
- **Composición:** una clase contiene a otra que no puede existir sin ella (ej.: un `Coche` tiene un `Motor`).

💡 **Analogía:** Composición es como un corazón dentro del cuerpo; agregación es como libros dentro de una mochila.

1.8 Herencia y polimorfismo

- **Herencia:** permite que una clase (hija) herede atributos y métodos de otra (padre).
 - Tipos: simple, múltiple (C++), multinivel. Java simple y multinivel con **extends**.
 - Debe usarse con moderación para evitar acoplamientos innecesarios.
- **Polimorfismo:**
 - **Sobrecarga (overload):** mismo nombre, diferentes funciones :
`void mover(); void mover(int pasos);`

- **Sobrescritura (override):** redefinir el comportamiento en una subclase:

```
class Animal { void hacerSonido() { System.out.println("..."); } }
class Perro extends Animal { void hacerSonido() { System.out.println("Guau!"); } }
```
- **Enlace dinámico:** la llamada al método se resuelve en tiempo de ejecución:

```
Animal a = new Perro(); a.hacerSonido(); // "Guau!"
```

1.9 Interfaces y abstracción en Java

- Una **interfaz** define un contrato: un conjunto de métodos que una clase se compromete a implementar.
- Permiten una forma de **polimorfismo sin herencia directa** de clases.
- **Ventajas:**
 - Separan **lo que hace** un objeto de **cómo lo hace**.
 - Fomentan el diseño desacoplado y la extensibilidad.
 - Permiten implementar **herencia múltiple de comportamiento**.

```
interface Volador { void volar(); } class Pajaro implements Volador {
public void volar() {System.out.println("Estoy volando");}}
```

1.10 Principios SOLID

- Conjunto de buenas prácticas para el diseño de clases y objetos:
 - **S: Responsabilidad única** – cada clase debe hacer una sola cosa.
 - **O: Abierto/cerrado** – abierto a extensión, cerrado a modificación.
 - **L: Sustitución de Liskov** – las subclases deben poder reemplazar a sus padres sin errores.
 - **I: Segregación de interfaces** – mejor muchas interfaces pequeñas que una grande.
 - **D: Inversión de dependencias** – depender de abstracciones, no de implementaciones.

1.11 Lenguajes y ejemplos

- **Java:** OO puro, fuerte tipado, sin herencia múltiple.
- **C++:** OO con herencia múltiple, manejo manual de memoria.
- **Python:** flexible, admite POO parcial (múltiples paradigmas), permite mixins.
- **C#:** fuertemente orientado a objetos, con interfaces, propiedades y LINQ.

1.12 Comparación con programación estructurada (nuevo)

- La programación estructurada separa datos y funciones, siendo adecuada para tareas simples, mientras que la orientada a objetos los agrupa en clases, facilitando el desarrollo de aplicaciones complejas, escalables y mantenibles.

2. PARTE DIDÁCTICA

- **Nivel:** 1º DAM. **Módulo:** Programación. **Perfil:** alumnado que debe dominar diseño OO como herramienta profesional.

2.2 Objetivos de aprendizaje

- Modelar entidades del mundo real con clases.
- Implementar herencia, polimorfismo y encapsulamiento.
- Aplicar principios SOLID y patrones básicos.

2.3 Metodología

- Desarrollo por fases: análisis → clases → relaciones → código en Java.
- Talleres colaborativos con refactorización y revisión conjunta de código.

2.4 Atención a la diversidad


- Plantillas base y guías activas. IDE con análisis estático (como SonarLint).

2.5 DUA

- Representaciones: UML, diagramas de clases, ejemplos en código.

2.6 Actividad principal

Proyecto “La startup de los objetos”

- Cada grupo define su **caso de uso principal** y diseña el **modelo de dominio**, identificando clases como **Usuario**, **Producto**, **Pedido**, etc., aplicando conceptos de **herencia**, **composición y relaciones entre objetos**.
- El modelo se implementa en **Java o Python**, estructurado en **clases, paquetes y/o interfaces**, respetando principios de **encapsulamiento y modularidad**.
- Se justifica el diseño mediante principios como **SOLID**.
- El proyecto se presenta en formato de **reunión técnica simulada**, como si los estudiantes expusieran la solución ante un equipo de stakeholders (clientes o inversores).
-  **Opcional:** integrar una librería ORM para la persistencia de datos (ej.: **Hibernate** en Java o **SQLAlchemy** en Python).

2.7 Evaluación. Instrumentos. Criterios.

Tema 29. Utilidades para el desarrollo y prueba de programas. Compiladores. Interpretes. Depuradores.

1. Introducción

- El desarrollo profesional de software requiere herramientas que permitan **traducir, ejecutar y verificar** programas. Los entornos modernos integran **editores, compiladores, intérpretes, linters y depuradores**, lo que permite un flujo de trabajo eficiente y una mejora continua de la calidad del código.

2. Compiladores

2.1 Definición

- Un **compilador** traduce el código fuente a código máquina antes de su ejecución. Esto permite una ejecución más rápida, aunque exige recompilación tras cada cambio.

2.2 Fases de un compilador

1. **Análisis léxico**: identifica tokens en el texto fuente.
2. **Análisis sintáctico**: valida la estructura gramatical.
3. **Análisis semántico**: comprueba tipos, nombres y coherencia.
4. **Generación de código intermedio**: crea una representación abstracta.
5. **Optimización**: mejora el rendimiento del código.
6. **Generación de código objeto y enlazado**: crea el ejecutable final.

2.3 Ejemplos

- **GCC** (C/C++), **javac** (Java), **Kotlin compiler**, **Rustc**.

2.4 Ventajas

- **Alto rendimiento**, validación previa de errores.
- **Protección del código fuente**: se distribuye el ejecutable, no el texto.

3. Intérpretes

3.1 Definición

- Ejecutan el código fuente línea a línea, sin generar archivo binario. Son ideales para scripting, prototipado y enseñanza.

3.2 Ejemplos

- Python, Ruby, PHP, bash, Node.js.

3.3 Ventajas

- **Ejecución inmediata**, ideal para pruebas y ajustes rápidos.
- Menor tiempo de preparación, mayor flexibilidad

4. Depuradores

4.1 ¿Qué es un depurador?

- Herramienta que permite **analizar la ejecución del programa paso a paso**, para identificar y corregir errores lógicos o de flujo.

4.2 Funcionalidades principales

- Puntos de interrupción (breakpoints).
- Ejecución paso a paso (**step into**, **step over**).
- Inspección de variables y estructuras.
- Modificación del flujo en tiempo real.
- Seguimiento de llamadas (**stack trace**).

4.3 Ejemplos

- **gdb** (C/C++), **pdb** (Python), depuradores integrados en IDEs (VSCode, PyCharm, NetBeans).

5. Linters y análisis estático de código

5.1 ¿Qué es un linter?

- Herramienta que analiza el código sin ejecutarlo, detectando **errores de estilo, duplicaciones, variables no usadas y malas prácticas**.

5.2 Finalidad

- Prevenir errores antes de ejecutar.
- Mejorar la legibilidad y mantenibilidad del código.
- Asegurar coherencia en equipos de desarrollo.

5.3 Ejemplos

- **pylint**, **flake8** (Python), **eslint**, **prettier** (JavaScript), **cpplint** (C++), SonarLint.

5.4 Diferencia con el compilador

- El **compilador** se centra en errores de sintaxis y semántica.
- El **linter** se enfoca en estilo, redundancia y coherencia.

5.5 Relación con el flujo de desarrollo

- Se usa antes de compilar o interpretar. Ideal para mantener **código limpio y profesional** desde el principio.

6. Conclusión científica

- El uso de compiladores, intérpretes, depuradores y linters no solo mejora la productividad y calidad del desarrollo, sino que permite al programador comprender **cómo se transforma el código en instrucciones reales para la máquina**. Estas herramientas constituyen una **infraestructura técnica y cognitiva fundamental** en cualquier entorno profesional. Además, el análisis estático (linters) y la depuración estructurada son claves en la prevención de errores y en la **escalabilidad de proyectos colaborativos**.

Propuesta didáctica

1, Contexto. Asignatura: **Programación** Niveles: **CFGM DAM / DAW**

Aula con IDEs variados (VSCode, NetBeans, PyCharm).

Alumnado con conocimientos básicos de programación.

2. Objetivos

- Diferenciar entre compilación, interpretación y análisis estático.
- Aplicar herramientas reales de desarrollo y depuración.
- Fomentar buenas prácticas de prueba y corrección estructurada.

3. Metodología

- Aprendizaje basado en problemas (ABP).
- Programación incremental con revisión continua.
- Trabajo colaborativo con roles rotativos

4. Atención a la diversidad y DUA

- **Nivel III:** ejemplos guiados, esquemas paso a paso.
- **Nivel IV:** apoyo técnico individual, simplificación de código.
- **DUA:** soportes múltiples (visual, textual, práctico); tareas diferenciadas por nivel.

5. Actividad principal

“Auditoría técnica: análisis comparado de código compilado vs interpretado”

- Desarrollar un pequeño programa funcional en **C** (compilado) y su equivalente en **Python** (interpretado), incorporando intencionadamente errores de distintos tipos: **sintácticos, lógicos y de estilo**.
- El código será intercambiado entre grupos, de modo que cada equipo intente **detectar, analizar y corregir** los errores introducidos por otro grupo, aplicando herramientas como **gdb, pdb, linters** y depuración paso a paso.
- Elaborar un **informe comparativo** que incluya:
 - Ventajas e inconvenientes de cada enfoque (Compilador vs Interpretado)
 - Observaciones sobre la facilidad de depuración
 - Reflexión sobre el uso de linters y su impacto en el trabajo en equipo
- Elaborar una presentación con los bugs detectados en el código de otros equipos.

6. Evaluación

- **Instrumentos:** rúbrica de práctica, revisión del informe, autoevaluación técnica.
- **Criterios:** uso correcto de herramientas, comprensión de los procesos, calidad del código, reflexión crítica.

Tema 31. Lenguaje C: Características generales. Elementos del lenguaje. Estructura de un programa. Funciones de librería y usuario. Entorno de compilación. Herramientas para la elaboración y depuración de programas en lenguaje C.

1.1 Introducción y características

- Creado por **Dennis Ritchie (1972)**, base de UNIX y precursor de C++, Java y muchos otros.
- Es un **lenguaje de medio nivel**: permite trabajar con memoria y hardware, pero con abstracción suficiente para programación estructurada.
- Se destaca por su **eficiencia, portabilidad, sintaxis compacta y control detallado** del sistema.
- C sigue siendo ampliamente usado en: Sistemas embebidos, Kernels (Linux), Desarrollo de compiladores, Firmware y microcontroladores
- Estándares oficiales: **ANSI C, C89, C99, C11, C17**.

1.2 Elementos del lenguaje C

- **Directivas de preprocesador**: `#include`, `#define`, `#ifndef`, etc.

Estructura básica:

```
#include <stdio.h>
```

```
int main() {  
    printf("Hola mundo");  
    return 0;  
}
```

- **Tipos de datos**: `int`, `char`, `float`, `double`
 - Modificadores: `short`, `long`, `unsigned`, `signed`
- **Operadores**:
 - Aritméticos: `+`, `-`, `*`, `/`, `%`
 - Lógicos: `&&`, `||`, `!`
 - Relacionales: `<`, `>`, `==`, `!=`
 - Bit a bit: `&`, `|`, `^`, `<<`, `>>`

1.3 Estructura modular

- Código dividido en:
 - **Archivos .h** (cabeceras): declaraciones de funciones y estructuras
 - **Archivos .c**: implementación

Uso del **preprocesador**:

```
// operaciones.h  
int sumar(int, int);
```

```
// operaciones.c  
int sumar(int a, int b) { return a + b; }
```

Inclusión condicional:

```
#ifndef OPERACIONES_H  
#define OPERACIONES_H  
...  
#endif
```

1.4 Funciones

- **Bibliotecas estándar**:
 - `stdio.h` → `printf`, `scanf`
 - `stdlib.h` → `malloc`, `exit`
 - `string.h` → `strlen`, `strcpy`
 - `math.h` → `sqrt`, `pow`
- **Funciones definidas por el usuario**:
 - Prototipo en `.h`, definición en `.c`
 - Paso de argumentos por **valor** o por **puntero**

Recursividad:

```
int factorial(int n) {  
    if (n == 0) return 1;  
    return n * factorial(n - 1);  
}
```

1.5 Punteros y memoria

Un puntero almacena la **dirección** de una variable:

```
int a = 5;
int *p = &a;
```

- Acceso indirecto: `*p` da el valor apuntado.
- **Memoria dinámica:**
 - `malloc`, `calloc`, `realloc`, `free`
- **Errores comunes:**
 - *Memory leaks*
 - *Segmentation faults*
 - *Buffer overflows*
- Herramienta clave: `Valgrind`

1.6 Entorno y proceso de compilación en C

- **Fases:**
 1. **Preprocesado** (`#include`, macros)
 2. **Compilación** (traducción a código objeto)
 3. **Enlazado** (vinculación de múltiples objetos o librerías)
 4. **Ejecución**
- Herramientas: `gcc`, `clang`, `make`

Ejemplo de Makefile:

```
programa: main.o operaciones.o
gcc -o programa main.o operaciones.o
```

1.7 Depuración y testing

- **Depuración:**
 - `gdb`: puntos de ruptura, inspección de variables, seguimiento de pila.
 - `Valgrind`: análisis de errores de memoria.
 - `Sanitizers`: `-fsanitize=address`, `-fsanitize=undefined`
- **Testing:**
 - `assert()`: comprobación de condiciones.
 - Integración con `make` para pruebas automatizadas y `cmocka` para test unitarios.
- **Optimización:**
 - Compilación optimizada con `-O2`, `-O3`
 - Análisis de rendimiento con `gprof`
- Linters: `cppcheck`, `clang-tidy`, `splint`, `SonarLint`

2. PARTE DIDÁCTICA

2.1 Contextualización. Nivel: 1º DAM. Módulo: Programación.

- Lenguaje C se emplea para formar bases sólidas en estructuras de datos, gestión de memoria y desarrollo profesional de bajo nivel.

2.2 Objetivos de aprendizaje

- Comprender la estructura de programas en C.
- Usar punteros, funciones, compilación modular y librerías.
- Aplicar herramientas de depuración y testing profesional.

2.3 Metodología

- Enfoque práctico: codificación diaria, ejercicios dirigidos y proyectos grupales.
- Apoyo visual (diagrama de compilación, árbol de dependencias).
- IDEs ligeros: Code::Blocks, Geany + consola.

2.4 Atención a la diversidad

- Plantillas con código base comentado.
- Pares tutores para seguimiento.
- Ajustes en número de funciones/módulos según nivel (III y IV).

2.5 DUA

- **Representación:** UML, código comentado, ejecución paso a paso.
- **Acción:** prácticas con funciones, Makefile, GDB.
- **Motivación:** desafíos entre equipos tipo “debugging hunt”.

2.6 Actividad principal

- Desarrolla una aplicación modular en lenguaje C que permita **gestionar una colección de canciones** (título, artista, duración...).
- Deberá incluir funciones de **alta, búsqueda, listado y eliminación**, haciendo uso de estructuras, punteros y memoria dinámica.
- Fase final: defensa técnica + demo + validación de errores.

2.7 Evaluación. Instrumentos. Criterios.

Tema 32. Lenguaje C: Manipulación de estructuras de datos dinámicas y estáticas. Entrada y salida de datos. Gestión de punteros. Punteros a funciones.

1.1 Introducción

C permite construir estructuras de datos manualmente, gestionar punteros, modularizar código y controlar la memoria. Ideal para desarrollar lógica algorítmica de bajo nivel.

1.2 Estructuras del lenguaje (estáticas)

Construcciones nativas con memoria fija.

- **Arrays:** homogéneos, acceso rápido, tamaño fijo. Ej: `int lista[10];` ⚠ No verifican límites.

struct: datos heterogéneos. Usar `typedef` para claridad:

`typedef struct { char nombre[30]; int edad; } Alumno;`

- **union:** comparte memoria entre campos.
- **enum:** constantes simbólicas legibles. Ej: `enum Estado {ACTIVO, INACTIVO};`

1.3 Estructuras algorítmicas

Definidas con `struct`, `typedef`, punteros. Según su tipo de memoria:

1.3.1 Estáticas

- Arrays de `struct`, tablas simples.
- Pilas/colas simuladas con índices.
- ✓ Simples y eficientes, pero sin crecimiento dinámico.

1.3.2 Dinámicas

- **Lineales:** listas enlazadas, pilas, colas (`malloc`, `free`, `next`).
- **Jerárquicas:** árboles binarios con `left/right`, recorridos recursivos.
- **Asociativas:** hash (clave → valor), grafos con punteros cruzados.
- **Genéricas:** listas con `void *`, punteros a funciones para liberar/comparar.

1.4 Entrada y salida de datos

- **Estándar:** `scanf`, `fgets`, `getchar`. ⚠ Preferir `fgets`.
- **Archivos:** `fopen`, `fprintf`, `fclose`, validar errores.
- **Binario:** `fwrite`, `fread` para guardar/cargar estructuras (`Alumno[]`).

1.5 Punteros

- **Básicos:** `int *p = &x;`, `*p` accede al valor.
- **Referencia:** modificar desde función: `void cambiar(int *p)`
- **Dobles:** matrices, listas dobles, pasar puntero por referencia.
- **Errores comunes:** `NULL`, uso tras `free`, punteros colgantes, doble `free`.

1.6 Punteros a funciones

- **Declaración:** `int (*pf)(int,int) = suma;`
- **Aplicaciones:**
 - Arrays de funciones: `void (*menu[])() = {ver, editar};`
 - Callbacks: `qsort(..., comparar);`
 - ✓ Útiles para menús dinámicos o lógica desacoplada.

1.7 Modularización y compilación

- `.h`: declaraciones; `.c`: implementación. Uso de `extern`.

Makefile básico para automatizar:

prog: main.o lista.o

`gcc -o prog main.o lista.o`

- Estructura de carpetas proyectos: `/src`, `/include`, `/tests`

1.8 Testing y depuración

- **Testing:** `assert`, frameworks (`CMocka`, `Unity`).
- **Depuración:** `gdb`, `Valgrind`, `-fsanitize=address`.
- **Visualización:** imprimir estructuras, simular recorridos.

1.9 Buenas prácticas

- Inicializar punteros, verificar `malloc`.
- Liberar memoria en orden.
- Documentar quién libera qué.
- Separar lógica, E/S y validación.
- Dividir código en funciones pequeñas.

2. PARTE DIDÁCTICA

2.1 Contextualización. Nivel: 1º DAM. Módulo: Programación.

- El uso de estructuras dinámicas y punteros entrena la lógica algorítmica, la organización modular y la gestión avanzada de memoria.

2.2 Objetivos de aprendizaje

- Manipular estructuras de datos dinámicas y estáticas en C.
- Usar punteros y funciones para modularizar programas.
- Validar programas con herramientas de testing profesional.

2.3 Metodología

- **Aprendizaje por proyectos:** desarrollar sistemas pequeños con listas, árboles o colas.
- Prácticas guiadas de implementación, pruebas y depuración.
- Simulación de problemas reales: menú de opciones, carga desde fichero, etc.

2.4 Atención a la diversidad

- Desdoblamiento de funciones: versión básica vs. avanzada.
- Guías paso a paso + comentarios orientativos.
- Evaluación formativa progresiva, con seguimiento individual.

2.5 DUA

- **Representación:** diagramas de estructuras dinámicas, videos paso a paso.
- **Acción:** implementación modular, menús dinámicos, debugging gamificado.
- **Motivación:** retos entre grupos, “modo cazador de bugs”.

2.6 Actividad principal

“Misión: estructura viva – diseña, enlaza y da vida a tus datos”

Simula el desarrollo de una pequeña biblioteca de estructuras dinámicas en C, desde el diseño hasta la validación profesional.

- **Fase 1: Diseño y planificación**
 - Selecciona una estructura dinámica (lista, pila, árbol, cola...).
 - Elabora su modelo lógico en papel: nodos, enlaces, operaciones clave.
- **Fase 2: Implementación modular**
 - Codifica usando `.h/.c`, `typedef`, `malloc`, `free` y punteros dobles.
 - Añade operaciones de E/S básicas (carga, visualización, persistencia).
- **Fase 3: Interfaz interactiva**
 - Implementa un **menú dinámico** con punteros a funciones para cada operación (insertar, buscar, eliminar, etc.).
- **Fase 4: Verificación técnica**
 - Prueba el programa con `assert`.
 - Analiza errores de memoria con **Valgrind**.
 - Si es posible, automatiza tests con **CMocka**.
- **Fase 5: Presentación final**
 - Realiza una **defensa técnica oral** del diseño, las decisiones tomadas y las herramientas utilizadas.
 - Incluye una demo funcional y muestra evidencias de testing y depuración.

2.7 Evaluación

- **Instrumentos:** rúbrica técnica + validación funcional.
- **Criterios:**
 - Estructura de datos funcional, uso correcto de memoria.
 - Menú modularizado con punteros a funciones.
 - Uso efectivo de herramientas (Valgrind, assert...).
 - Defensa clara y técnica del código.

2.8 Conclusión didáctica

Este tema desarrolla en el alumnado competencias fundamentales para el desarrollo profesional: diseño eficiente de estructuras, modularidad, control total de la memoria y depuración avanzada. Su dominio es esencial como base para asignaturas futuras (ED, Programación OO, Acceso a Datos).

Tema 34. Sistemas gestores de base de datos. Funciones. Componentes. Arquitecturas de referencia y operacionales. Tipos de sistemas.

1.1 1 Funciones de un sistema gestor de bases de datos

Un SGBD es un sistema software que gestiona grandes volúmenes de datos de forma **eficiente, segura y estructurada**.

Funciones esenciales:

- Almacenamiento y recuperación de datos
- Consulta y actualización
- Control de concurrencia
- Seguridad e integridad
- Recuperación ante fallos
- Copias de respaldo y auditoría

1.2 Componentes internos de un sistema gestor de bases de datos

- **Motor de almacenamiento:** gestiona acceso físico a datos, bloques, índices, caché y transacciones.
 - **Procesador de consultas:** analiza, optimiza y ejecuta instrucciones SQL.
 - **Gestor de transacciones:**
 - Aplica el modelo **ACID** (→ ver más abajo).
 - Puede usar **MVCC** (un mecanismo que permite que varias transacciones accedan a los mismos datos al mismo tiempo sin bloquearse entre sí, manteniendo la consistencia) y bloqueos para concurrencia.
 - **Catálogo (diccionario de datos):** mantiene metadatos sobre tablas, vistas, permisos, esquemas.
 - **Interfaz de usuario:** acceso a través de SQL, APIs, GUI o drivers.
 - **Módulo de recuperación:** gestiona logs, journaling, punto de restauración (**restore**, **rollback**).
- 🔍 **ACID** (propiedades de las transacciones):
- **Atomicidad:** todo o nada.
 - **Consistencia:** mantiene reglas de integridad.
 - **Aislamiento:** transacciones concurrentes no interfieren.
 - **Durabilidad:** los cambios persisten tras un fallo.

1.3 Arquitecturas de referencia

- **Monolítico:** todo en un solo proceso. Ej: SQLite.
- **Cliente-servidor:** cliente SQL se conecta a un servidor central (PostgreSQL, MySQL).
- **Distribuido:** datos en nodos remotos, con mecanismos de replicación y consenso (Cassandra, CockroachDB).
- **En la nube:** servicio gestionado (Amazon RDS, Azure SQL, MongoDB Atlas).

1.4 Arquitectura operacional: capas físicas y lógicas

- **Capa física:** bloques, páginas, ficheros binarios.
- **Capa de almacenamiento:** índices, buffer cache, tablas.
- **Capa lógica:** esquemas, vistas, relaciones.
- **Capa de acceso:** interfaces SQL, ORMs, APIs, conectores.

1.5 Tipos de sistemas gestores de bases de datos

- ♦ **Relacionales (SQL)**
 - Modelo tabular, normalizado, con claves y relaciones.
 - Ejemplos: MySQL, PostgreSQL, Oracle, SQL Server, SQLite.
- ♦ **NoSQL**
 - **Documentales** (MongoDB, CouchDB): JSON flexible, ideal para datos semi-estructurados.
 - **Clave-valor** (Redis, Memcached): rápidos, uso en cacheo o sesiones.
 - **Columnar** (Cassandra, HBase): optimizados para lecturas por columna.
 - **Grafos** (Neo4j, JanusGraph): nodos y relaciones, útil para redes o mapas semánticos.
- ♦ **NewSQL**
 - Combinan **SQL clásico** con **escalabilidad horizontal**.
 - Ej: CockroachDB, Google Spanner.
- ♦ **Sistemas embebidos**
 - Ligeros, sin servidor, integrables en aplicaciones.
 - Ej: SQLite, Berkeley DB.

1.6 Comparativa y elección del SGBD

Modelo	Ventajas	Limitaciones
--------	----------	--------------

SQL	ACID, relaciones, SQL poderoso	Menor escalabilidad sin partición
NoSQL	Escalable, rápido, flexible	Menos garantías de integridad
NewSQL	ACID + escalabilidad	Más complejo, menos maduro
Embebido	Ligero, sin servidor	Sin concurrencia, sin escalado real

✚ Elección depende de:

- Necesidad de relaciones y joins → SQL
- Gran volumen + esquema flexible → NoSQL
- Alta disponibilidad distribuida → NewSQL
- Entorno cerrado o app móvil → Embebido

1.7 Conclusión técnica

Un SGBD moderno es una arquitectura de múltiples capas que gestiona datos, control de acceso, transacciones, rendimiento y fallos.

La elección del sistema adecuado afecta directamente a la **eficiencia, fiabilidad, coste y escalabilidad** del software que lo utiliza.

2. PARTE DIDÁCTICA

2.1 Contextualización (MÓDULO: Bases de datos– 1.º DAM)

- El alumnado usa SGBD en prácticamente todos los módulos: backend web, móvil, sensores, microservicios y Big Data.
- Imprescindible dominar estructura interna, funciones y tipos de SGBD.

2.2 Objetivos de aprendizaje

- Reconocer y describir las funciones internas del SGBD.
- Identificar sus componentes y capas operacionales.
- Comparar tipos según arquitectura y uso.
- Tomar decisiones adecuadas según requisitos de proyecto.

2.3 Metodología

- Clases teórico-prácticas con esquemas visuales.
- Estudio de casos reales (PostgreSQL, MongoDB, Cassandra).
- Ejercicios comparativos de sistemas y configuraciones.

2.4 Atención a la diversidad

- Textos divididos por niveles de profundidad.
- Fases diseñadas: básico (SQL relacional), intermedio (NoSQL documental), avanzado (NewSQL/distribuido).
- Tutorías específicas para aclarar conceptos complejos.

2.5 DUA

- Representación: diagramas capa-función.
- Acción: configuración real de instancias locales, monitorización, ejecución de consultas.
- Motivación: simular decisiones técnicas reales en selección/configuración de SGBD.

2.6 Actividad principal

Los equipos actúan como consultoras que reciben un caso (IoT, e-commerce, red social...). Analizan necesidades y comparan tres opciones:

- SQL (PostgreSQL)
- NoSQL (MongoDB)
- Sistema Embebido (SQLite)

✚ Tareas:

- Diseñar el modelo de datos y arquitectura para cada opción.
- Simular operaciones clave (insert/select masivo, fallos simples).
- Comparar rendimiento, escalabilidad y consistencia.
- Defender la mejor elección en una presentación final.

2.7 Evaluación

- **Instrumentos:** rúbrica técnica, informe al estilo “technical review”, exposición.
- **Criterios:**
 - Cobertura de funciones internas y arquitecturas.
 - Comparativa argumentada de sistemas.

Tema 35. La definición de datos. Niveles de descripción. Lenguajes. Diccionario de datos.



1.1 Introducción

- “El proceso de definición de datos se basa en describirlos en distintos niveles de abstracción (conceptual, lógico y físico), mediante lenguajes formales (como SQL-DDL), y apoyarse en diccionarios de datos que aseguran integridad, trazabilidad y reutilización de la información.”

1.2 La definición formal de los datos

- Definir datos implica describir **qué** se almacena y **cómo**, clave para la integridad y usabilidad de bases de datos.
- Desde perspectiva de diseño y ejecución, los datos deben documentarse, estructurarse y controlar su uso.

1.3 Niveles de descripción de datos (conceptual, lógico, físico)

1. **Nivel conceptual:** vista global del sistema: entidades, relaciones y restricciones generales.
 2. **Nivel lógico:** estructuras propias del modelo de datos usado (tablas y columnas en SQL, documentos en NoSQL).
 3. **Nivel físico:** organización interna de los datos: ficheros, índices, páginas de disco, compresión, particionado.
-  Esta separación permite modificar el almacenamiento sin alterar la lógica de negocio ni la visión del usuario.
 -  **Conexión con el modelo E-R:**
El nivel conceptual suele derivarse del modelo entidad-relación (E-R), que identifica las entidades, atributos y relaciones que luego se formalizan en el diccionario de datos.

1.4 Lenguajes y modelos de definición de datos

- **SQL-DDL:**
`CREATE, ALTER, DROP, TRUNCATE...`
Definición de tipos (`INT, VARCHAR`), restricciones (`NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, CHECK`).
- **NoSQL-DDL:**
Uso de validadores JSON Schema (ej.: en MongoDB), estructuras definidas en la aplicación o en herramientas externas.
- **DDL orientado a objetos:**
Generación a partir de clases usando **ORMs** (Hibernate, SQLAlchemy, Entity Framework).
El esquema se deriva del modelo de clases.

1.5 Diccionario de datos: estructura, contenido y funciones

- Catálogo central que describe cada elemento: nombre, tipo, longitud, valores permitidos, significado, propietario, restricciones, etiquetas, origen.
- Permite coherencia semántica, gobernanza y entendimiento compartido.
- Suele incluir metadatos: fechas, responsables, nivel de confidencialidad y ejemplos.

1.6 Metadatos adicionales

- Trazabilidad: quién y cuándo creó/modificó datos.
- Uso: campos obligatorios/opcionales, clasificaciones, relaciones cruzadas.
- Soporta auditoría, control de calidad y seguridad.

1.7 Uso en ingeniería de datos

- Diseño común entre aplicaciones heterogéneas, APIs, servicios, microservicios.
- Base para generación automática de código, validación de datos y documentación interactiva.

1.8 Conclusión técnica

- La definición formal de datos garantiza interoperabilidad, calidad y mantenimiento eficiente de sistemas.
- Hace del diccionario un artefacto clave en proyectos reales de software y bases de datos.

2. PARTE DIDÁCTICA

2.1 Contextualización (MÓDULO: Bases de datos– 1.º DAM)

- El objetivo es que el alumnado entienda cómo estructurar y documentar datos de forma profesional, habilidad esencial para desarrollo de aplicaciones y servicios.

2.2 Objetivos de aprendizaje

- Aprender los tres niveles de descripción de datos.
- Usar lenguajes DDL en SQL y ejemplos NoSQL.
- Crear y mantener un diccionario de datos colaborativo.

2.3 Metodología

- Desarrollo de una base de datos pequeña (ej. tienda online, registro académico).
- Trabajo por fases:

1. Diagrama entidad-relación (nivel conceptual).
 2. Generación de scripts DDL (nivel lógico).
 3. Implementación de índice, particiones, replicación (nivel físico).
- Registro sistemático de campos en un diccionario compartido (Hoja de cálculo o herramienta colaborativa).


2.4 Atención a la diversidad

- Plantillas progresivas: de básico (nombre, tipo) a completo (restricciones, origen, nivel confidencial).
- Ajuste de metas según capacidad técnica (niveles III y IV).
- Tutoría individualizada en elaboración de diccionario.






2.5 DUA

- Representaciones gráficas (ERD, tablas).
- Acción: práctica de definición y documentación de datos.
- Motivación: simular trabajo real de analista/arquitecto de datos.

2.6 Actividad principal

 **Reto:** Diseña el modelo de datos completo para una app real (biblioteca, tienda online, reservas...), **desde la lógica hasta el diccionario**, como si fueras parte de un equipo de desarrollo profesional.

¿Qué harás?

1.  **Mapa conceptual:** Crea el esquema E-R y define los **tres niveles** del modelo: conceptual (E-R), lógico (tablas), físico (tipos, índices).
2.  **Código DDL:** Escribe el script SQL con **CREATE TABLE**, claves y restricciones. ¡Debe funcionar!
3.  **Diccionario técnico:** Documenta los campos: nombre, tipo, qué significan, si son obligatorios... (puedes usar Excel o Notion).
4.  **Versionado como un pro** Sube todo a GitHub con mensajes de commit reales:
add: tabla pedidos + FK a clientes
5.  **Exposición final** Presenta tu modelo en clase: explica cómo lo estructuraste, qué decisiones tomaste y cómo lo documentaste. Tu grupo será el “equipo de datos” ante el cliente.

2.7 Evaluación

- **Instrumentos:** rúbrica para evaluación del ERD, scripts, diccionario y exposición.
- **Criterios:**
 - Completitud y coherencia semántica del diccionario.
 - Calidad de scripts DDL y justificación técnica.
 - Claridad en la exposición y capacidad argumentativa.

2.8 Conclusión didáctica

El trabajo sistemático en diseño y documentación de datos implica responsabilidad técnica y facilita futuras integraciones y mantenimiento. El dominio de los tres niveles y el diccionario de datos prepara para roles reales como desarrollador, analista o arquitecto de datos.

Tema 36. La manipulación de datos. Operaciones. Lenguajes. Optimización de consultas.

1.1 Introducción

- Esencial en cualquier SGBD: almacenar, consultar, actualizar, borrar datos de forma eficiente.
- Base para aplicaciones web, móviles, IoT y Big Data.
- El rendimiento depende del diseño de consultas y estructura de datos.

1.2 Modelos de datos

Modelo	Ejemplo	Características
Relacional (SQL)	Tablas + claves → CREATE TABLE clientes (...)	Normalización, integridad referencial
Documental (NoSQL)	MongoDB con documentos JSON	Agilidad, esquemas flexibles
Clave-valor	Redis cache	Alta velocidad, ideal para sesiones
Columnares	Cassandra	Esquemas densos por columnas
Grafos	Neo4j	Relaciones complejas (social, rutas)
Multimodelo	ArangoDB	Combina documentos, grafos y relaciones
NewSQL	CockroachDB	SQL distribuido con ACID

1.3 Lenguajes de manipulación

- **SQL:**
 - DML: **SELECT, INSERT, UPDATE, DELETE**
 - DDL: **CREATE, ALTER, DROP**
 - DCL: **GRANT, REVOKE**
 - TCL: **BEGIN, COMMIT, ROLLBACK**
- **NoSQL:**
 - MongoDB (MQL): **db.clientes.find({ ... })**, agregaciones
 - Cassandra (CQL): consultas simples por partición
- **SQL con JSON:**
 - PostgreSQL: **SELECT datos->>'nombre' FROM empleados WHERE datos->>'pais' = 'España';**

1.4 Operaciones de datos

- **Básicas:** **SELECT, INSERT, UPDATE, DELETE**
- **Avanzadas:**
 - **JOIN**, subconsultas, funciones agregadas (**GROUP BY, HAVING**)
- **Transacciones:** ACID, aislamiento, MVCC

1.5 NoSQL

- MongoDB: inserciones y agregaciones eficientes
- Cassandra: consultas clave-primaria, sin JOINS

1.6 Optimización de consultas

- **SQL:**
 - Índices: B-Tree, Hash, GIN
 - EXPLAIN/ANALYZE
 - Reescritura de consultas (evitar **SELECT ***, subconsultas invalidas...)
⚠ caso típico: evitar producto cartesiano mediante subconsulta
- **NoSQL:** índices compuestos, TTL, diseño con enfoque “query-first”
- **Herramientas:**
 - PostgreSQL: **pg_stat_statements, auto_explain**
 - MongoDB: **.explain()**, Atlas

- Cassandra: **nodetool**, tracing

1.7 Configuración y escalabilidad

- Ajustes: caches, buffer sizes, conexiones
- Alta disponibilidad: replicación, sharding, tolerancia a fallos
- En la nube: RDS, MongoDB Atlas, escalado gestionado

1.8 Tendencias

- Bases autogestionadas (IA integrada)
- Enfoque multimodelo
- Integración con Big Data (Spark, Kafka)
- Columnar para analítica masiva (ClickHouse)

1.9 Conclusión técnica

- Modelar datos, saber manipularlos y optimizar consultas garantiza sistemas robustos.
- SQL y NoSQL son complementarios en arquitecturas modernas.

2. PARTE DIDÁCTICA

(MÓDULO: PROGRAMACIÓN – 1.º DAM)

2.1 Contextualización

- **Aplicación** de conceptos en desarrollo real de aplicaciones web y móviles.
- Permite entender impacto del diseño de datos y rendimiento.

2.2 Objetivos de aprendizaje

- Diseñar modelos relacionales y NoSQL.
- Escribir y optimizar consultas SQL y MQL.
- Aplicar índices, revisando planes de ejecución.
- Comparar ventajas y limitaciones de cada enfoque.

2.3 Metodología

- Talleres con bases reales: SQLite, PostgreSQL y MongoDB local
- Ejercicios progresivos: de consultas simples a operaciones complejas.
- Análisis de caso y reescritura de consultas subóptimas.

2.4 Atención a la diversidad

- Niveles diferenciados: grupos en esquema relacional, documental, híbrido.
- Enfoque práctico con guías paso a paso.
- Laboratorio accesible para resolución independiente.

2.5 DUA

- Representación: diagramas de entidad-relación, sketch de documentos JSON
- Acción: modelado y pruebas de consultas, optimización, monitorización
- Compromiso: realizar pruebas en tiempo real del impacto en rendimiento

2.6 Actividad principal

Proyecto “Diseña tu Base de Datos del Mundo Real”

- Grupos eligen un dominio (tienda en línea, red social, etc.)
- Diseñan modelo: tablas SQL + documentos NoSQL
- Implementan operaciones CRUD, agregaciones y transacciones/multi-documento
- Optimizan con índices, EXPLAIN, reestructuran consultas
- Presentan resultados y justificación de decisiones

2.7 Evaluación

- **Instrumentos:** rúbrica técnica + presentación del proyecto
- **Criterios destacables:**
 - Calidad del modelo y escalabilidad
 - Eficiencia de consultas y uso de índices
 - Entendimiento de la configuración y entorno seleccionado
 - Comunicación técnica clara y coherente

2.8 Conclusión didáctica

Este tema forma al alumnado en el manejo competente de datos, combinando teoría de bases, consultas avanzadas, modelos alternativos y visión de arquitecturas modernas. Les dota de herramientas clave para desarrollar servicios eficientes y escalables.

Tema 38. Modelo de datos relacional. Estructuras. Operaciones. Álgebra relacional.

1 Introducción

- Propuesto por **E. F. Codd (1970)**.
- Basado en **teoría de conjuntos** y **lógica de primer orden**.
- Representa la información como **relaciones** (tablas) con **tuplas** (filas) y **atributos** (columnas).
- Introduce independencia lógica/física y claves:
 - **Clave primaria (PK)**: identifica unívocamente cada tupla.
 - **Clave foránea (FK)**: mantiene la **integridad referencial** entre tablas.

1.2 Estructura del modelo relacional

1.2.1 Componentes básicos

- **Relación**: conjunto de tuplas.
- **Atributo**: columna con dominio de valores.
- **Tupla**: fila que representa un registro.
- **Dominio**: conjunto de valores válidos.
- **Esquema de relación**: nombre de la tabla y atributos.
- **Instancia de relación**: conjunto de tuplas existentes en un momento dado.

1.2.2 Restricciones de integridad

- **Dominio** (tipo correcto de dato)
- **NOT NULL, UNIQUE, CHECK**
- **Integridad de entidad**: PK no nula ni duplicada.
- **Integridad referencial**: FK debe existir en la tabla referida.

📌 Ejemplo:

clientes(id_cliente [PK], nombre, ciudad)

pedidos(id_pedido [PK], id_cliente [FK], total)

1.3 Normalización

Proceso de mejora del diseño de relaciones para reducir redundancia y mejorar integridad.

Forma normal	Qué evita
1FN	Atributos multivaluados o repetidos
2FN	Dependencias parciales en claves compuestas
3FN	Dependencias transitivas entre atributos

Cada forma normal mejora la calidad del esquema y facilita mantenimiento, pero puede afectar el rendimiento si se fragmenta en exceso.

1.4 Operaciones del álgebra relacional

1.4.1 Operaciones fundamentales

- **Selección (σ)**: filtra filas por condición
 $\sigma_{\text{ciudad}='Madrid'}(\text{clientes})$
- **Proyección (π)**: extrae columnas
 $\pi_{\text{nombre,email}}(\text{clientes})$
- **Producto cartesiano (\times)**: combina todas las tuplas de dos relaciones

1.4.2 Operaciones derivadas

- **Unión (\cup), Intersección (\cap), Diferencia ($-$)**
Solo entre relaciones compatibles (mismo número y tipo de atributos)
- **Renombrado (ρ)**: cambia nombre de relación o atributo
 $\rho_{C1 \rightarrow \text{nombre}}(\text{cliente})$
- **JOINS**:
 - θ -join: unión con condición
 - equi-join: condición de igualdad
 - natural join (\bowtie): elimina atributos duplicados
- **División (\div)**: casos con "para todo"; compleja pero útil
Ej.: "clientes que han comprado todos los productos"

1.5 Álgebra relacional y SQL

El álgebra relacional es formal; **SQL lo implementa de forma práctica**. Comprender álgebra permite optimizar consultas y depurar errores.

Ejemplo de traducción:

```
SELECT nombre FROM clientes
JOIN pedidos ON clientes.id = pedidos.cliente_id
WHERE pedidos.total > 100;
```

→ Álgebra:

```
 $\pi_{\text{nombre}}(\text{clientes} \bowtie \sigma_{\text{total} > 100}(\text{pedidos}))$ 
```

Tabla de correspondencias:

Álgebra	SQL
σ	WHERE
π	SELECT columnas
\bowtie	JOIN ON
U	UNION
–	EXCEPT / MINUS
\div	Subconsulta correlada

1.8 Conclusión técnica

El modelo relacional es:

- **Robusto y universal**
- Base teórica de todos los SGBD clásicos
- Clave para formular y optimizar consultas
- Imprescindible para el diseño lógico y profesional de bases de datos

2. PARTE DIDÁCTICA**Módulo: Bases de Datos — 1.º DAM****2.1 Contextualización**

- Aplicación directa en **desarrollo backend, administración de datos y consultas SQL**.
- Fundamenta operaciones en gestores como SQLite, MySQL o PostgreSQL.

2.2 Objetivos de aprendizaje

- Comprender estructuras relacionales: tablas, atributos, claves.
- Aplicar operadores del álgebra relacional.
- Traducir álgebra \leftrightarrow SQL y razonar sobre optimización.
- Normalizar esquemas y detectar redundancia.

2.3 Metodología

- Actividades prácticas en papel + SQL real.
- Ejercicios de traducción entre álgebra y SQL.
- Simulación de esquemas con DBDesigner, SQLite o db-fiddle.

2.4 Atención a la diversidad

- Ejercicios escalonados: selección \rightarrow join \rightarrow subconsultas.
- Visualización de consultas con símbolos π , σ , \bowtie .
- Retos para alumnado avanzado: consultas con división y renombramiento.

2.5 Principios DUA

- Representación: esquemas visuales, tablas, pseudocódigo.
- Acción: ejecución práctica en SGBD.
- Motivación: mostrar cómo SQL “piensa” como el álgebra relacional.

2.6 Actividad principal**“Del papel a la base de datos: consultas en dos lenguajes”**

1. Dado un esquema (clientes, pedidos), redactar consultas naturales.
2. Expresarlas en álgebra relacional formal.
3. Traducirlas a SQL y ejecutarlas.
4. Analizar con EXPLAIN y justificar decisiones.

2.7 Evaluación

- Formativa + rúbrica por fases:
 - Precisión en álgebra.
 - Corrección sintáctica y lógica de SQL.
 - Justificación del diseño y comparación de eficiencia.

2.8 Conclusión didáctica

- El álgebra relacional refuerza el pensamiento lógico y estructurado.
- Favorece la transición de “problema real” → “consulta formal” → “ejecución eficiente”.

Tema 39. Lenguajes para la definición y manipulación de datos en sistemas de base de datos relacionales. Tipos. Características. Lenguaje SQL.

1.1 Introducción

- **SQL** (Structured Query Language) fue creado en los años 70 y sigue siendo el **lenguaje estándar declarativo** para bases de datos relacionales.
- Es **declarativo**, describes *qué* resultados quieres, no *cómo* obtenerlos.
- Se integra con lenguajes como Python, JavaScript, R o Spark, siendo el “idioma universal” para bases de datos estructuradas.

1.2 Componentes del lenguaje SQL

Subtipo	Función	Ejemplos
DDL	Definición de estructuras	CREATE TABLE, ALTER TABLE
DML	Manipulación de datos	INSERT, UPDATE, DELETE, window functions
DQL	Consultas	SELECT, con JOIN, GROUP BY, CTE, HAVING
DCL	Control de permisos	GRANT, REVOKE, roles, RLS, column masking
TCL	Transacciones ACID	BEGIN, COMMIT, ROLLBACK

1.3 DDL — Lenguaje de Definición

- Crea y modifica la **estructura** de BD: tablas, vistas, tipos.

```
CREATE TABLE empleados (  
  id INT PRIMARY KEY,  
  nombre VARCHAR(100),  
  salario DECIMAL(8,2)  
);
```

- Permite tipos avanzados (JSON, GEOMETRY), vistas materializadas, triggers.
- Rol: *arquitecto de la base de datos*.

1.4 DML — Manipulación de datos

- Inserta, actualiza y elimina registros.

```
INSERT INTO empleados (...) VALUES (...);
```

- Incluye **window functions**:

```
SELECT nombre, salario, AVG(salario) OVER (PARTITION BY depto) FROM empleados;
```

1.5 DQL — Consultas

- **SELECT**: extracción de datos mediante combinaciones (JOIN), agregaciones, WHERE, HAVING, CTE.
- Ejemplo clave para eficiencia y usabilidad.

1.6 DCL — Control de acceso

- Define **quién puede hacer qué** sobre la BD.

```
GRANT SELECT ON clientes TO usuario_lectura;
```

- Incluye técnicas como **Row-Level Security** y **Column Masking**.

1.7 TCL — Transacciones

- Agrupa operaciones para garantizar atomicidad.

```
BEGIN;
```

```
UPDATE cuentas SET saldo = saldo - 100 WHERE id = 1;
```

```
UPDATE cuentas SET saldo = saldo + 100 WHERE id = 2;
```

```
COMMIT;
```

- Crucial en operaciones críticas como transferencias bancarias.

1.8 Integración e interfaces externas

- SQL embebido en Python, C, Java, etc.
- Compatible con ORMs (Django, Hibernate).
- Conecta lógica de negocio con manejo de datos.

1.9 Alternativas y extensiones a SQL

- **NoSQL:** MongoDB (documentos), Cassandra (columnas). Escalabilidad, consistencia final (BASE).
- **NewSQL:** CockroachDB, TiDB. Ofrecen **SQL + escalabilidad ACID distribuidas**.
- **GraphQL:** consulta flexible desde frontends.
- **DSLs:** Pandas (Python), dplyr (R), Flink SQL, Kafka Streams.

1.10 Aplicaciones modernas

- **BI:** Power BI, Tableau.
- **Learning Analytics:** analizar comportamiento del alumnado.
- **IA y Big Data:** preparar y combinar datos para análisis.
- Alta demanda en sectores variados (salud, educación, fintech).

1.11 Conclusión técnica

- SQL es robusto, universal y continúa evolucionando.
- Entender sus sublenguajes, transacciones y seguridad es esencial.
- Su dominio abre puertas en múltiples ámbitos técnicos.

2. PARTE DIDÁCTICA

Módulo: Bases de Datos / Desarrollo – 1.º DAM

2.1 Contextualización

- SQL es una **herramienta profesional** en backend, análisis de datos y BI.
- Sus sublenguajes cubren desde la creación de la estructura hasta la gestión de accesos y transacciones.

2.2 Objetivos de aprendizaje

- Distinguir entre DDL, DML, DQL, DCL y TCL.
- Crear y modificar estructura de BD.
- Escribir consultas y manipular datos.
- Implementar y controlar transacciones.
- Gestionar permisos en entornos multiusuario.

2.3 Metodología

- Clases prácticas guiadas con conexión a SGBD (SQLite, PostgreSQL).
- Ejercicios en cuaderno y ejecución real de código.
- Comparativa entre SQL puro, ORMs y DSLs.

2.4 Atención a la diversidad

- Ejercicios diferenciados por nivel de complejidad.
- Apoyo visual (tablas, sublenguajes en colores).
- Desafíos para alumnos avanzados (triggers, window functions, RLS).

2.5 Diseño Universal para el Aprendizaje (DUA)

- **Representación:** diagramas de flujo, ejemplos visuales.
- **Acción:** escribir, ejecutar y verificar.
- **Motivación:** demostrar cómo SQL afecta resultados reales y rendimiento.

2.6 Actividad principal

“Ingeniería de consultas: optimiza y asegura tu base de datos”

1. Se proporciona una BD incompleta y consultas ineficientes.
2. Los alumnos deben:
 - Identificar errores o malas prácticas (faltan índices, subconsultas innecesarias).
 - Reescribir código, justificando mejoras.
 - Aplicar permisos, triggers o RLS.
 - Probar y documentar resultados comparativos.

2.7 Evaluación

- Evaluación continua con rúbrica según:
 - Capacidad de distinguir y aplicar sublenguajes.
 - Corrección sintáctica y lógica.
 - Justificación técnica de soluciones.
 - Calidad de la documentación y demostraciones.

2.8 Conclusión didáctica

- SQL enseña lógica estructurada, seguridad y buenas prácticas profesionales.
- Empodera al alumnado para diseñar, consultar y proteger bases de datos reales.
- Prepara para el mercado laboral y roles en data engineering, administración y desarrollo.

Tema 40. Diseño de bases de datos relacionales.

1. Introducción

Las bases de datos relacionales permiten **almacenar, organizar y consultar datos estructurados** de forma eficiente, coherente y segura.

Un diseño sólido garantiza:

- **Integridad** (reglas que aseguran datos correctos)
- **Eficiencia** (consultas optimizadas)
- **Escalabilidad** (crece sin pérdida de rendimiento)
- **Seguridad y gobernanza del dato**



Fases del diseño de una BDR:

1. **Recolección de requisitos**
2. **Modelo conceptual (E-R)**
3. **Modelo lógico (relacional, normalizado)**
4. **Modelo físico (ajustado al SGBD)**
5. **Validación, documentación y mantenimiento**

2. Fundamentos del Modelo Relacional

- Propuesto por **Edgar F. Codd (1970)**.
- Representa los datos en **relaciones (tablas)**, con **tuplas (filas)** y **atributos (columnas)**.
- Operaciones basadas en **álgebra relacional y cálculo relacional**.
- Introduce claves primarias y foráneas para mantener la integridad.

3. Proceso de Diseño Relacional

3.1 Recolección de Requisitos

- Entrevistas, casos de uso, reglas de negocio.
- Identificar entidades, relaciones, atributos clave.
- Estimar volumen, frecuencia de uso y crecimiento futuro.
- Considerar seguridad, concurrencia y consistencia necesarias.

3.2 Modelo Conceptual (MER)

Representación abstracta del dominio con el modelo Entidad-Relación.

- Elementos:
 - **Entidades** y sus atributos
 - **Relaciones** (1:1, 1:N, N:M)
 - **Claves primarias**
 - **Atributos compuestos y derivados**
 - **Cardinalidades** y restricciones
- Puede incluir: especialización, generalización, agregación.

3.3 Modelo Lógico Relacional

Traducción del modelo E-R a tablas con reglas formales.

- **Entidad** → **tabla**, **atributo** → **columna**, **relación** → **FK**
- Se definen:
 - Claves primarias y foráneas
 - Tipos de datos
 - Reglas de integridad (NOT NULL, UNIQUE, CHECK, etc.)

♦ Normalización

Proceso formal para eliminar redundancias y anomalías de modificación.

Forma normal	Requisito principal	Qué elimina	Acción a realizar
FNN	Tabla sin estructura	Datos mal organizados	Detectar errores de diseño
1FN	Valores atómicos	Grupos repetidos	Separar valores en filas
2FN	En 1FN + sin dependencias parciales	Atributos que dependen solo de parte de la PK	Crear nuevas tablas

3FN	En 2FN + sin dependencias transitivas	Atributos no clave dependientes de otros no clave	Separar en nuevas relaciones
BCNF	Todo determinante es superclave	Dependencias no válidas	Descomponer si es necesario

✚ Opcional: formas normales superiores (4FN, 5FN) para casos complejos (atributos multivaluados, dependencia de unión).

3.4 Modelo Físico

Representación concreta del diseño en el SGBD elegido.

- **Definición exacta** de:
 - Tipos de datos (**INT**, **VARCHAR**, **DATE**, **BOOLEAN**)
 - Longitudes, codificaciones (UTF-8, ISO)
 - Claves primarias, foráneas, restricciones
 - Índices simples y compuestos (optimización de búsquedas)
 - Estrategias de particionado o compresión

✓ Se tienen en cuenta factores como rendimiento, tamaño, concurrencia o replicación.

4. Etapas complementarias del diseño

4.1 Validación del diseño

- Revisión con usuarios o analistas: ¿cubre los requisitos?
- Casos de prueba: inserción, modificación, integridad referencial
- Simulación de consultas clave para evaluar rendimiento
- Detección de ambigüedades o redundancias

4.2 Documentación del modelo

- **Diccionario de datos:** nombre de campos, tipo, restricciones, descripción.

4.3 Mantenimiento y evolución

Un buen diseño debe poder adaptarse sin romper el sistema.

- Versionado del esquema (**CREATE/ALTER**, migraciones, Git)
- Registro de cambios (**changelog.sql**, tags, ramas)
- Refactorizaciones controladas (renombrado de columnas, separación de tablas)
- Estrategias de desnormalización parcial si el rendimiento lo requirer

5. Conclusión

Un diseño relacional de calidad parte de una buena comprensión del dominio, Se construye paso a paso desde lo conceptual a lo físico y se valida, documenta y mantiene activamente

PROPUESTA DIDÁCTICA

1. Contextualización. Bases de Datos (CFGs DAM)

- Aula con acceso a SGBD (MySQL, PostgreSQL, SQLite).
- Alumnado con conocimientos básicos de lógica y programación.

2. Objetivos

- Comprender y aplicar los principios del diseño relacional.
- Elaborar modelos conceptuales, lógicos y físicos de bases de datos.
- Detectar y corregir errores de diseño.

3. Metodología

- **Aprendizaje basado en proyectos y casos reales.**
- Diseño guiado paso a paso con ejercicios aplicados.

4. Atención a la Diversidad y DUA

- **Nivel III:** plantillas visuales de modelos ER, diagramas preconstruidos.
- **Nivel IV:** fragmentación de tareas, validación paso a paso.
- **DUA:** representación múltiple (visual, simbólica, textual), entrega en distintos formatos (PDF, maqueta, exposición).

5. Actividad Principal

“Diseña la base de datos de tu aula virtual”

- Análisis de requisitos: alumnado, profesorado, módulos, calificaciones.
- Creación del modelo ER, normalización y paso a relacional.
- Implementación parcial en SGBD y prueba con datos reales.
- Documentación técnica completa y defensa en grupo.

6. Evaluación

- **Instrumentos:** rúbrica del diseño, funcionalidad del prototipo, defensa oral.
- **Criterios:** consistencia lógica, normalización correcta, utilidad y presentación clara.

Tema 44. Técnicas y procedimientos para la seguridad de los datos.

1.1 Contexto y principios fundamentales

- **Reputación, economía y control:** pérdida de datos equivale a pérdidas graves.
- Principios esenciales: **Confidencialidad, Integridad, Disponibilidad, Autenticidad, No repudio.**
- Dos líneas de defensa:
 - **Seguridad activa:** prevención, detección.
 - **Seguridad pasiva:** respuesta, recuperación.

1.2 Servicios clave de seguridad

- **Confidencialidad**
 - Cifrado en tránsito (*TLS/SSL*) y en reposo (*AES, RSA*).
 - Clasificación de datos (público, interno, confidencial).
 - Modelos de acceso: **RBAC, Zero Trust.**
- **Integridad**
 - Hashes (*SHA-256*), firmas digitales, herramientas como *Tripwire*.
 - Restricciones en BD: **CHECK, UNIQUE.**
- **Disponibilidad**
 - Estrategia 3-2-1 en backups.
 - Clústeres, replicación y planes de recuperación ante desastres.
- **Autenticidad y No repudio**
 - **MFA**, biometría.
 - **Timestamping** con firma digital.

1.3 Técnicas avanzadas de protección

- **Cifrado en la nube:** *BYOK, HSM.*
- Protección en BD:
 - **TDE** (cifrado total).
 - **Always Encrypted** (columnas sensibles).
- **DLP:** bloqueo de fugas por correo o USB.
- **Enmascaramiento/pseudonimización** para cumplir normativas.
- **Principio de menor privilegio**, auditoría y revisión periódica de accesos.

1.4 Sistemas de protección de datos

- **Backups:** completos, incrementales, snapshots.
- **Monitorización y auditoría:** plataformas tipo *Wazuh, Splunk.*
- **Base de datos:**
 - ACID, triggers de auditoría, consultas parametrizadas, protección XSS/SQLi.
 - NoSQL: control de acceso, cifrado y validación.
- **Sistemas de ficheros:**
 - BitLocker, **ACL**, snapshots.
- **Replicación:** síncrona (consistencia) vs. asíncrona (rendimiento).

1.5 Marcos normativos y estándares

- **ISO 27001/27002, NIST 800-53 / 800-207.**
- Legislación: **RGPD, LOPDGDD, ENS.**
- Herramientas de análisis de riesgos: **MAGERIT, PILAR.**

1.6 Amenazas habituales

- **Ransomware**, malware, empleados desleales, errores humanos, *Shadow IT.*
- Técnicas de ataque: **SQLi, XSS, MITM.**

1.7 Seguridad en entornos cloud

- Riesgos: buckets públicos, exposición de claves.
- Controles: **CloudTrail, IAM**, cifrado cliente, soluciones **CSPM** (*Prisma Cloud*).

2. PARTE DIDÁCTICA

2.1 Contextualización

- Módulo: *Seguridad y Alta Disponibilidad* de ASIR.
- Enfoque práctico: diseño y despliegue de sistemas seguros.

2.2 Objetivos de aprendizaje

- Identificar y aplicar los cinco pilares de la seguridad de datos.
- Implementar medidas técnicas (cifrado, backups, autenticación).
- Conocer marcos legales: RGPD y ENS.
- Desarrollar cultura de seguridad y responsabilidad profesional.

2.3 Metodología

- Formación teórica y práctica:
 - Configuración de **TLS** y **AES**.
 - Simulación de backups y recuperación con snapshots.
 - Implementación de roles y cifrado en base de datos.
 - Análisis de ataques conocidos (vulnerabilidades y mitigación).

2.4 Atención a la diversidad

- Escalabilidad de retos:
 - Nivel básico: cifrado de ficheros y backups.
 - Nivel avanzado: gestión de certificados, automatización, infraestructura CPI.
- Recursos visuales: esquemas de capas de seguridad, auditorías.

2.5 Principios del DUA

- **Representación:** flujogramas, comparativas entre modelos activos y pasivos.
- **Acción:** laboratorio con *Wazuh*, *GPG*, configuración de cifrado y roles.
- **Motivación:** relación directa entre fallos reales y políticas preventivas.

2.6 Actividad principal

Actividad: “Protege tus datos: construye y defiende tu sistema”

💡 Tu misión es crear un sistema de datos sencillo pero seguro. Primero lo vas a proteger, luego alguien intentará atacarlo (¡simulado!), y por último, deberás mejorarlo.

- **Fase 1: Diseña tu sistema seguro**
 - Decide qué datos hay (**mínimo:** usuarios con sus contraseñas, correos, amigos, etc.).
 - Clasifica los datos: ¿cuáles son privados?
 - Elige medidas de seguridad:
 - ¿Qué datos se cifran?
 - ¿Quién puede ver cada cosa?
 - ¿Se hacen copias de seguridad?
- **Fase 2: Prueba tu sistema (¡y ataca!)**
 - Implementa una pequeña base de datos (ejemplo: usuarios con contraseñas).
 - Simula un ataque:
 - Intenta ver datos sin permiso a través de Crear un usuario "fantasma" (sin contraseña y con rol=admin por error).
 - Detecta los fallos y **corrige** el sistema:
 - Añade validación, roles, protección de campos
- **Fase 3: Comprueba y explica**
 - Haz una prueba final para ver si los fallos están arreglados.
 - Mira los logs o mensajes del sistema.
 - Explica con tu grupo:
 - Qué aprendiste
 - Como podías evitar el problema.
 - Si no podías evitar el problema, como podías mitigarlo (copia, encriptación)
 - Qué harías mejor la próxima vez

🎤 **Entrega:** documentación de todo el proceso.

2.7 Evaluación

- Rúbrica ponderando:
 - Calidad técnica (90 %).
 - Documentación y argumentación (10 %).
- Criterios:
 - Aplicación de técnicas de cifrado y autenticación.
 - Eficacia frente a incidencias simuladas.
 - Claridad técnica y uso adecuado de estándares normativos.

2.8 Conclusión didáctica

- La seguridad de datos es un **proceso continuo**, no una función aislada.
- Desarrolla en el alumnado competencias técnicas, éticas y de decisión.
- Les prepara para roles profesionales en seguridad, gestión y administración de sistemas.

TEMA 45: SISTEMAS DE INFORMACIÓN. TIPOS. CARACTERÍSTICAS. SISTEMAS DE INFORMACIÓN EN LA EMPRESA

1. Introducción

- Un **Sistema de Información (SI)** es un conjunto organizado de recursos humanos, tecnológicos y procedimentales que permite recopilar, procesar, almacenar y difundir información para apoyar la toma de decisiones y el control en una organización.
- Los SI son fundamentales en la **gestión, operativa y estrategia empresarial**.

2. Componentes de un Sistema de Información

- **Hardware:** servidores, redes, equipos de usuario.
- **Software:** aplicaciones, SGBD, sistemas operativos.
- **Datos:** materia prima del sistema.
- **Personas:** usuarios, analistas, técnicos.
- **Procesos:** métodos organizativos y normativas internas.

3. Tipos de Sistemas de Información

Tipo	Función principal	Usuarios principales
TPS (Transaction Processing System)	Procesamiento diario de transacciones	Operativos
MIS (Management Information System)	Informes periódicos y control	Mandos intermedios
DSS (Decision Support System)	Apoyo a decisiones no estructuradas	Dirección
ERP (Enterprise Resource Planning)	Integración total de áreas funcionales	Toda la empresa
CRM (Customer Relationship Management)	Gestión de relaciones con clientes	Comercial y marketing
SCM (Supply Chain Management)	Gestión de la cadena de suministro	Logística y compras
BI (Business Intelligence)	Análisis avanzado de datos	Dirección estratégica

PROPUESTA DIDÁCTICA

Asignatura: Sistemas de Información / Administración de Sistemas (CFGS DAM / ASIR)

1. Contextualización

- Aula equipada con simuladores de SI empresariales y acceso a software ERP/CRM.
- Alumnado orientado a informática aplicada al entorno empresarial.

2. Objetivos

- Comprender la estructura y función de los SI.
- Diferenciar sus tipos y aplicaciones.
- Analizar cómo los SI apoyan a la empresa en su operativa y estrategia.

3. Metodología

- **Aprendizaje por simulación y análisis de casos reales.**
- Uso de entornos ERP/CRM y ejercicios de toma de decisiones.

4. Atención a la Diversidad y DUA

- **Nivel III:** presentación guiada de tipos de SI con ejemplos visuales.
- **Nivel IV:** tareas simplificadas, acompañamiento activo.
- **DUA:** entrada múltiple (vídeos, mapas mentales, software), producción diversa (presentaciones, infografías, casos).

5. Actividad Principal

“Escoge y justifica tu sistema”

- Simulación de necesidades de una empresa ficticia.
- Elección justificada de los tipos de SI que necesita.
- Presentación de cómo se integran en su operativa diaria.
- Comparativa entre soluciones tecnológicas reales (SAP, Salesforce, Odoo, etc.).

6. Evaluación

- **Instrumentos:** rúbrica del análisis, exposición del caso, cuestionario técnico.
- **Criterios:** clasificación correcta, argumentación fundamentada, vinculación con procesos empresariales.

7. Conclusión Didáctica

Los SI son el pilar de la gestión empresarial moderna. Su conocimiento permite al alumnado participar activamente en la mejora de procesos, la transformación digital y la eficiencia operativa de cualquier organización.

TEMA 46: APLICACIONES INFORMÁTICAS DE PROPÓSITO GENERAL Y PARA LA GESTIÓN EMPRESARIAL. TIPOS. FUNCIONES. CARACTERÍSTICAS

1. Introducción

- Las aplicaciones informáticas permiten **automatizar tareas, gestionar información y mejorar la productividad** tanto en contextos generales como en entornos empresariales específicos.
- Se dividen entre herramientas **de propósito general** (ofimática, navegación, comunicación) y **de gestión empresarial** (ERP, CRM, BI).

2. Aplicaciones de Propósito General

2.1 Características

- Uso cotidiano, interfaz amigable, funcionalidad transversal.
- No específicas de un sector.

2.2 Tipos y Funciones

Tipo	Funciones principales	Ejemplos
Procesadores de texto	Edición de documentos	Microsoft Word, LibreOffice Writer
Hojas de cálculo	Cálculo, gráficos, análisis de datos	Excel, Calc
Presentaciones	Diseño de diapositivas, comunicación	PowerPoint, Impress
Navegadores web	Acceso a internet	Chrome, Firefox
Cientes de correo	Gestión de mensajes	Outlook, Thunderbird
Comunicación	Videollamadas, mensajería	Zoom, Teams, Slack

3. Aplicaciones para la Gestión Empresarial

3.1 Características

- Integran procesos y datos clave del negocio.
- Adaptables a distintas áreas funcionales.

3.2 Tipos y Funciones

Tipo	Función principal	Ejemplos
ERP	Gestión integral (compras, ventas, RRHH)	SAP, Odoo, Microsoft Dynamics
CRM	Gestión de relaciones con clientes	Salesforce, Zoho, HubSpot
BI	Análisis de datos y apoyo a decisiones	Power BI, Qlik, Tableau
SCM	Gestión de cadena de suministro	Oracle SCM, SAP SCM
Contabilidad	Registro y control financiero	Contasol, Sage 50

PROPUESTA DIDÁCTICA

Asignatura: Aplicaciones Ofimáticas / Sistemas de Gestión Empresarial (CFGM SMR / CFGS DAM/DAW)

1. Contextualización

- Aula con paquetes ofimáticos y software empresarial (ERP/CRM online).

- Alumnado técnico con perfil mixto (administrativo e informático).

2. Objetivos

- Diferenciar entre aplicaciones generales y empresariales.
- Utilizar herramientas reales en contextos simulados.
- Relacionar las TIC con la mejora de procesos empresariales.

3. Metodología

- **Aprendizaje funcional y comparativo.**
- Práctica directa con software + análisis de necesidades de empresas.

4. Atención a la Diversidad y DUA

- **Nivel III:** entornos simplificados (modo asistente, plantillas).
- **Nivel IV:** acompañamiento personalizado, reducción de funciones.
- **DUA:** diversidad de soportes (vídeo, texto, simulación), entrega flexible (documento, exposición, infografía).

5. Actividad Principal

“Diseña tu solución empresarial”

- Caso práctico de una empresa simulada.
- Selección y justificación de herramientas: general (ofimática) y empresarial (ERP o CRM).
- Prueba real (registro de ventas, análisis de datos, informe final).
- Defensa técnica ante el grupo.

6. Evaluación

- **Instrumentos:** rúbrica del caso, análisis funcional, prueba práctica.
- **Criterios:** adecuación de herramientas, uso correcto, justificación argumentada.

7. Conclusión Didáctica

Dominar aplicaciones generales y empresariales capacita al alumnado para enfrentar escenarios profesionales reales, automatizar procesos y mejorar la toma de decisiones en cualquier sector.

TEMA 47: INSTALACIÓN Y EXPLOTACIÓN DE APLICACIONES INFORMÁTICAS.

COMPARTICIÓN DE DATOS

1. Introducción

- La instalación y explotación de software son procesos fundamentales en el ciclo de vida de las aplicaciones informáticas.
- La compartición de datos facilita la **colaboración, interoperabilidad y centralización de la información**.

2. Instalación de Aplicaciones

2.1 Fases del Proceso

- Revisión de requisitos del sistema.
- Elección del tipo de instalación: local, cliente-servidor, en red o virtualizada.
- Configuración inicial (idioma, rutas, licencias).
- Verificación post-instalación (comprobación de servicios, logs).

2.2 Métodos de Instalación

- Asistida (GUI): instaladores con interfaz gráfica.
- Desatendida: mediante scripts, automatizada (ej. MSI, .deb, .msu).
- En red: instalación centralizada desde servidor.

3. Explotación de Aplicaciones

3.1 Objetivos

- Garantizar el uso eficaz del software.
- Asegurar disponibilidad, rendimiento y mantenimiento.

3.2 Acciones Clave

- Actualizaciones y parches.
- Gestión de usuarios y roles.
- Monitorización de logs y rendimiento.
- Backup y recuperación.

PROPUESTA DIDÁCTICA

Asignatura: Implantación de Aplicaciones / Administración de Sistemas (CFGM SMR / CFGS ASIR)

1. Contextualización

- Entorno con SO Windows y Linux, servidores y estaciones cliente.
- Alumnado orientado a instalación, mantenimiento y soporte técnico.

2. Objetivos

- Automatizar instalaciones y configurar entornos correctamente.
- Gestionar permisos y servicios de aplicaciones en producción.
- Implementar la compartición segura de datos.

3. Metodología

- **Aprendizaje procedimental con enfoque práctico.**
- Uso de scripts, máquinas virtuales y servicios compartidos reales.

4. Atención a la Diversidad y DUA

- **Nivel III:** instalación guiada, entornos simulados.
- **Nivel IV:** simplificación de comandos y validación visual.
- **DUA:** contenidos accesibles (video-tutoriales, esquemas), respuesta diversificada (script, informe, presentación).

5. Actividad Principal

“Instala, configura y comparte”

- Instalación de una aplicación cliente-servidor (ej. ERP, gestor documental).
- Configuración de usuarios y carpetas compartidas con permisos.
- Documentación del proceso y resolución de incidencias típicas.
- Simulación de acceso remoto y uso colaborativo.

6. Evaluación

- **Instrumentos:** rúbrica de práctica, checklist de verificación, informe técnico.
- **Criterios:** proceso de instalación correcto, configuración funcional, seguridad aplicada en la compartición.

7. Conclusión Didáctica

Instalar y explotar aplicaciones con criterio técnico permite al alumnado asegurar sistemas fiables y colaborativos. La compartición segura de datos es un pilar en la eficiencia de cualquier entorno informático.

TEMA 54: DISEÑO DE INTERFACES GRÁFICAS DE USUARIO (GUI)

1.1 Fundamentos del Diseño Centrado en el Usuario

- La interfaz es el canal de comunicación entre humano y máquina: desde GUIs tradicionales hasta voz, gestos, AR/VR, y BCI.
- El diseño debe facilitar interacción clara, eficaz y comprensible.

1.2 Tipos de Interfaces

- **GUI clásicas** (móvil, web, escritorio).
- **Conversacionales** (chatbots, Siri).
- **Gestuales** (control sin contacto).
- **AR/VR/MR** (aumento/mundo virtual).
- **BCI** (control mental emergente).

1.3 Usabilidad, UX y Accesibilidad

- **Usabilidad**: eficacia, eficiencia y satisfacción.
- **UX**: impresión emocional global (¿volvería a usarla?).
- **Accesibilidad**: WCAG 2.2; obligatorio para entornos públicos.
- **Dark Patterns**: prácticas engañosas; deben evitarse.

1.4 Principios de diseño

- **Heurísticas de Nielsen**: feedback, control, reversibilidad, evitación de errores.
- **Leyes cognitivas**:
 - Fitts: botones grandes y accesibles.
 - Hick: menos opciones = decisiones más rápidas.
 - Gestalt: cohesión visual por agrupación y forma.

1.5 Diseño visual y microinteracciones

- **Color**: transmite emociones.
- **Tipografía**: guía visual.
- **Microinteracciones**: animaciones que dan contexto (hover, carga).

1.6 Proceso UX/UI

1. **Design Thinking**: empathize, define, ideate, prototype, test.
2. **Lean UX**: validación rápida e iterativa.
3. Técnicas: personas, user journeys, wireframes, mockups, prototipos interactivos.

2. PARTE DIDÁCTICA

(Módulo: Entornos de Desarrollo – DAM y Desarrollo Multiplataforma)

2.1 Contextualización

- Conecta programación, diseño visual y accesibilidad.
- Preparación de interfaces reales aplicables en industria digital.

2.2 Objetivos de aprendizaje

- Diseñar GUI claras, usables y accesibles.
- Crear prototipos interactivos y medir su eficacia.
- Reflexionar sobre ética en el diseño: evitar patrones oscuros.

2.3 Metodología

- **Teórica**: heurísticas, leyes cognitivas, accesibilidad, análisis de Dark Patterns.
- **Práctica**:
 - Prototipado en **Figma**.
 - Desarrollo con **React Native** o **Flutter**.
 - Test con usuarios reales desde el primer prototipo.

2.4 Atención a la diversidad (niveles III y IV)

- Material adaptado: desde wireframes básicos hasta prototipos interactivos completos.
- Apoyo visual (diagramas UX, contraste accesible) para estudiantes con dificultades.
- Reto avanzado: crear interfaces gestuales accesibles.

2.5 Aplicación DUA

- **Representar**: prototipos visuales y herramientas de diseño.
- **Acción**: test de usabilidad activo con usuarios reales.
- **Motivación**: análisis de interfaces éticas vs manipuladoras; promover diseño responsable.

2.6 Actividad principal: “Del Boceto al Código”

- **Reto grupal**: diseñar e implementar una interfaz real (web o móvil).
 1. Entender al usuario: entrevistas, creación de personas, user journey.
 2. Iterar prototipos: wireframes → mockups → prototipos interactivos.
 3. Codificar: React Native o Flutter, accesibilidad incluida (roles, labels, semántica, contrastes, navegación teclado).

4. Validar con pruebas de usabilidad y recoger métricas (eficacia, tiempo, SUS).
5. Presentación final: demo funcional + defensa del diseño, decisiones visuales y éticas.

2.7 Evaluación

- *Criterios ponderados:*
 - Usabilidad y accesibilidad técnica -> 40 %
 - Desarrollo visual y microinteracciones -> 25 %
 - Metodología UX aplicada -> 20 %
 - Justificación ética y detección de dark patterns -> 15 %

2.8 Conclusión didáctica

- En DAM, esta propuesta integra competencias de programación, diseño, pruebas de usuario y ética profesional.
- Simula un entorno real: desde briefing, prototipos y test UX, hasta entrega técnica y presentación.
- Destaca habilidades clave: creatividad, comunicación, colaboración interdisciplinar y responsabilidad digital.

TEMA 60: SISTEMAS BASADOS EN EL CONOCIMIENTO — REPRESENTACIÓN, COMPONENTES Y ARQUITECTURA

1.1 Introducción y evolución

- **Definición:** Un Sistema Basado en Conocimiento (SBC) emplea conocimiento explícito (reglas, ontologías, grafos) para razonar y tomar decisiones de forma justificable.
- **Evolución histórica:**
 - *Primera generación:* sistemas expertos clásicos (p. ej. MYCIN, DENDRAL), basados en reglas “si... entonces”. Rígidos, costosos de mantener, sin aprendizaje.
 - *Segunda generación (híbrida):* integran Machine Learning (ML) con lógica simbólica → sistemas neuro-simbólicos más adaptables y explicables.

1.2 Representación del conocimiento

- **Enfoque simbólico:**
 - *Ontologías* (OWL, SNOMED CT)
 - *Reglas lógicas* (CLIPS, Prolog)
 - *Grafos de conocimiento* (Google, Facebook)
- **Representación semántica/ML:**
 - *Embeddings* (Word2Vec, BERT, RDF2Vec) traducen conceptos a vectores, integrando razonamiento lógico y estadístico.

1.3 Arquitectura híbrida del SBC

- **Componentes:**
 1. Captura del conocimiento (manual o automática).
 2. Base de Conocimiento (hechos, reglas, estructuras).
 3. Motor de inferencia (lógica formal).
 4. Módulo ML (clasificación, clustering, extracción).
 5. Interfaz explicativa (XAI: LIME, SHAP, Trepan).
- **Flujo de trabajo:** extracción → inferencia simbólica → predicción ML → fusión de resultados → generación de explicación.

1.4 Sinergias ML + simbólica

- Reglas generadas automáticamente (árboles, clustering, LLMs).
- Inferencia probabilística (Pyro, Stan, DeepProbLog).
- NLP para convertir lenguaje natural en reglas o grafos (spaCy, ChatGPT).
- Visualización mediante Graphviz o D3.js.

1.5 Aplicaciones destacadas

- **Salud:** sistemas mixtos que mejoran diagnósticos en un 37 %.
- **Banca y seguros:** scoring crediticio con XAI.
- **Industria 4.0:** mantenimiento predictivo (LSTM + reglas), gemelos digitales.

1.6 Futuro y tendencias

- Generative AI + RAG (GPT + Pinecone, Weaviate).
- Aprendizaje continuo con detección de drifts y adaptación de reglas.
- Exploración de lógica cuántica en SBC (proyectos IBM Q).

1.7 Ejemplo de implementación (Python)

```
from pyke import knowledge_engine
from sklearn.ensemble import RandomForestClassifier
```

```
engine = knowledge_engine.engine(__file__)
engine.activate('diagnostico')
```

```
model = RandomForestClassifier().fit(X_train, y_train)
```

```
def diagnosticar(paciente):
    reglas = engine.prove_1('diagnostico', paciente)
    pred = model.predict([paciente['datos']])
    return combinar(reglas, pred)
```

1.8 Comparativa: SBC Clásico vs Híbrido

Métrica	SBC Clásico	SBC + ML Híbrido
Precisión	~70 %	> 90 %

Adaptabilidad	Baja	Alta
Coste de mantenimiento	Alto	Medio
Explicabilidad	Muy alta	Alta
Escalabilidad	Limitada	Alta

1.9 Conclusión técnica

Los SBC híbridos ofrecen precisión, adaptabilidad y capacidad de explicación, esenciales en sistemas donde la transparencia es un requisito.

2. PARTE DIDÁCTICA (página 2)

(Módulo: *Inteligencia Artificial / Sistemas Inteligentes – 1.º DAM*)

2.1 Contextualización

- Aplicación real en salud, finanzas e industria.
- Comprensión integrada de lógica simbólica, ML y XAI.

2.2 Objetivos de aprendizaje

1. Representar conocimiento con reglas, ontologías o grafos.
2. Desarrollar un sistema híbrido con reglas + ML.
3. Implementar explicaciones mediante XAI.
4. Reflexionar sobre ética y transparencia en sistemas inteligentes.

2.3 Metodología

- Sesiones mixtas teoría/práctica.
- Taller en Python: desarrollo completo de un SBC híbrido.
- Debates sobre ética de sistemas inteligentes.

2.4 Atención a la diversidad

- *Nivel III*: reglas simples y explicaciones paso a paso.
- *Nivel IV*: integración avanzada con embeddings y ontologías.
- Apoyo gráfico mediante diagramas de flujo y gráficos de conocimiento.

2.5 Adaptación al DUA

- **Representación múltiple**: reglas textuales, grafos, vectores.
- **Acción práctica**: programación real y generación de explicaciones.
- **Motivación**: casos reales con impacto social y reflexiones críticas.

2.6 Actividad principal

“Sistemas que Aprenden y Explican”

- **Descripción**: crear un SBC híbrido en Python para diagnóstico médico o scoring financiero.
- **Pasos**:
 1. Definición de reglas simbólicas.
 2. Entrenamiento de un modelo ML (Scikit-learn).
 3. Integración en función `diagnosticar()`.
 4. Explicaciones con LIME o SHAP.
 5. Presentación y demo de resultados.

2.7 Criterios de evaluación

- Diseño simbólico y reglas (25 %)
- Modelo ML funcional (25 %)
- Calidad explicativa con XAI (30 %)
- Presentación técnica y reflexión ética (20 %)

TEMA 61: REDES Y SERVICIOS DE COMUNICACIONES

1. Introducción

- Las redes son esenciales para la conectividad digital, soporte de servicios críticos (salud, automoción, entretenimiento) y habilitadoras de tecnologías como cloud e IA.
- Ejemplos: transmisión médica en tiempo real, coches conectados, plataformas OTT.

2. Arquitectura de Red

2.1 Modelos de Capas

- **Modelo OSI (ISO):** 7 capas (Física, Enlace, Red, Transporte, Sesión, Presentación, Aplicación).
- **Modelo TCP/IP:** 4 capas prácticas (Acceso, Internet, Transporte, Aplicación).
- Funciones clave: transporte fiable (TCP), direccionamiento (IP), segmentación y encapsulación de datos.

2.2 Capas Funcionales y Protocolos

- **Física y Enlace:** transmisión de bits (WiFi, UTP, 5G), control de acceso (Ethernet, MAC).
- **Red y Transporte:** IP, TCP (fiabilidad), UDP (rapidez), ICMP, QoS (latencia, jitter).
- **Aplicación:** protocolos visibles al usuario (HTTP, FTP, DNS, SIP, MQTT, WebRTC, servicios cloud).

3. Evolución Tecnológica

- De ARPANET a redes cuánticas y 6G.
- **SDN:** control centralizado de red.
- **IA en redes:** detección de anomalías, ajuste predictivo.
- Ejemplo: Google B4 y balanceo inteligente.

4. Seguridad en Redes

- **Amenazas:** DDoS, sniffing, spoofing, ransomware.
- **Defensas:** firewalls, IDS/IPS, cifrado (TLS, IPsec), 802.1X.
- **Zero Trust y Microsegmentación.**
- **Normativas:** GDPR, ENS, ISO 27001, NIST.

5. Gobernanza y Legislación

- **Ley General de Telecomunicaciones,** neutralidad de red, soberanía digital.
- Retos: localización de datos, legislación aplicable, independencia tecnológica.

6. Futuro y Conclusión

- Redes autónomas, ciudades inteligentes, IA distribuida, redes cuánticas.
- Las redes no son solo infraestructura: son el tejido del desarrollo socioeconómico.

PROPUESTA DIDÁCTICA

Asignatura: Redes Locales (Ciclo Formativo de Grado Medio en SMR)

1. Contextualización

- Aula de informática con simuladores (Packet Tracer) y equipamiento de red.
- Alumnado entre 16-20 años, grupo heterogéneo.

2. Objetivos

- Comprender y aplicar los modelos OSI y TCP/IP.
- Diseñar y simular una red básica segura y funcional.
- Desarrollar competencias digitales, trabajo en equipo y resolución de problemas.

3. Metodología

- **Aprendizaje basado en proyectos (ABP).**
- Simulación progresiva: diseño, configuración, testeo.
- Uso de recursos multimedia, rúbricas y portfolios digitales.

4. Atención a la Diversidad y DUA

- Nivel III: descomposición de tareas complejas, guías visuales, tutorización individual.
- Nivel IV: adaptaciones metodológicas, software lector, trabajo por parejas.
- **DUA:** múltiples formas de representación (infografías, vídeos), expresión (presentaciones, esquemas), implicación (gamificación, retos colaborativos).

5. Actividad Principal

"Misión: Conecta tu Mundo – Diseña tu Red"

- Simulación de red de campus educativo o ciudad inteligente.
- Elección de dispositivos, topología, servicios (DNS, VoIP, correo) y políticas de seguridad.
- Evaluación del rendimiento, resolución de fallos simulados y presentación final.

6. Evaluación

- **Instrumentos:** rúbrica de proyecto, observación directa, cuestionario de autoevaluación.

- **Criterios:** diseño correcto, funcionamiento lógico, seguridad aplicada, creatividad, defensa oral.

7. Conclusión Didáctica

El diseño y simulación de redes desarrolla pensamiento lógico, habilidades técnicas y visión sistémica. Aporta un enfoque activo, inclusivo y competencial, conectando la teoría con la práctica real.

TEMA 62: ARQUITECTURAS DE SISTEMAS DE COMUNICACIONES: CAPAS Y ESTÁNDARES

1. Introducción

- Las arquitecturas permiten la interoperabilidad entre dispositivos heterogéneos mediante modelos comunes que estructuran la comunicación en capas.
- Garantizan que un mensaje pueda viajar desde cualquier origen hasta cualquier destino, con independencia del hardware, fabricante o ubicación.

2. Modelo de Capas: Abstracción Funcional

- División modular que simplifica el diseño, diagnóstico y evolución de redes.
- Cada capa tiene funciones específicas y se comunica con las capas contiguas.
- **Analogía postal:** desde escribir una carta hasta su entrega física.

3. Modelos de Referencia

3.1 Modelo OSI (ISO/IEC 7498)

- Propuesto por la ISO en los 80 como modelo teórico de referencia (7 capas).
- **Capas:**
 - Física: transmisión binaria (UTP, fibra).
 - Enlace: control de acceso y errores (Ethernet, WiFi).
 - Red: direccionamiento y rutas (IP, ICMP, OSPF).
 - Transporte: fiabilidad y flujo (TCP, UDP).
 - Sesión: control de diálogo (RPC, NetBIOS).
 - Presentación: cifrado y formato (SSL/TLS, JPEG).
 - Aplicación: interacción usuario (HTTP, SMTP, DNS).
- **Utilidad:** enseñanza, diseño de sistemas, resolución de incidencias.

3.2 Modelo TCP/IP

- Base real de Internet (4 capas): Aplicación, Transporte, Internet, Acceso a Red.
- Fusión de capas OSI: simplificación y orientación práctica.
- Protocolos clave: IP, TCP, DNS, DHCP, HTTP.
- Descrito por RFCs, enfocado en interoperabilidad y robustez.

4. Arquitecturas por Entorno

4.1 Redes LAN/WAN

- Ethernet, VLAN, NAT, routing empresarial.

4.2 Redes Móviles

- 4G (EPC), 5G (slicing, edge computing).

4.3 Redes Industriales / IoT

- ISA-95, MQTT, OPC-UA: eficiencia y jerarquía funcional.

4.4 Redes Definidas por Software (SDN)

- Separación control/datos, APIs abiertas, flexibilidad total.

5. Estandarización

5.1 Finalidad

- Comunicación entre dispositivos diversos, innovación modular, estabilidad global.

5.2 Organismos

- ISO, IEEE (802.x), IETF (IP, TCP), ITU-T (VoIP), ETSI (5G).

5.3 Estándares Relevantes

- IEEE 802.3 (Ethernet), 802.11 (WiFi), TLS, IPsec, MQTT, CoAP, WPA3.

PROPUESTA DIDÁCTICA

Asignatura: Redes de Área Local (CFGM SMR) o Redes Avanzadas (CFGS ASIR)

1. Contextualización

- Aula equipada con recursos físicos y digitales.
- Estudiantes entre 16-22 años, con perfiles diversos y necesidades específicas.

2. Objetivos de Aprendizaje

- Comprender la estructura y función de las arquitecturas en red.
- Diferenciar entre modelos OSI y TCP/IP.
- Aplicar estándares y protocolos a entornos reales.

3. Metodología

- **Gamificación y aprendizaje vivencial.**
- Representación física de las capas del modelo OSI/TCP/IP.
- Role-playing, uso de objetos simbólicos y dinámicas colaborativas.

4. Atención a la Diversidad y DUA

- **Nivel III:** adaptación de roles, soporte visual y textual, pausas programadas.

- **Nivel IV:** simplificación de tareas, apoyo continuo, herramientas accesibles.
- **DUA:** distintas formas de presentación (gráficos, verbal, kinestésica), opciones de participación activa, implicación a través del juego.

5. Actividad Principal

“El Gran Juego de las Capas: Simula Internet en el Aula”

- Alumnado representa las capas de red físicamente.
- Se simulan errores, cifrados, redirecciones.
- Se integran elementos reales: IPs, protocolos, detección de errores.

6. Evaluación

- **Instrumentos:** listas de cotejo, vídeos de la actividad, rúbrica de roles.
- **Criterios:** comprensión de funciones por capa, participación activa, resolución de fallos.

7. Conclusión Didáctica

Comprender las arquitecturas por capas permite al alumnado abstraer y dominar la complejidad de las redes. Esta propuesta hace tangible lo invisible, fomenta la cooperación, el pensamiento sistémico y conecta teoría y práctica en una experiencia significativa.

TEMA 63: NIVEL FÍSICO EN REDES DE COMUNICACIONES: FUNCIONES, MEDIOS, ADAPTACIÓN, LIMITACIONES Y ESTÁNDARES

1. Introducción

- El nivel físico (capa 1 del modelo OSI) es responsable de la **transmisión bruta de bits** sobre un medio físico.
- No interpreta datos ni protocolos: solo transforma información binaria en señales eléctricas, ópticas o de radio.

2. Funciones del Nivel Físico

- **Codificación y modulación:** conversión de bits en señales.
- **Transmisión y recepción** de bits en tiempo real.
- **Sincronización** emisor-receptor (ej. codificación Manchester).
- **Topología física:** estrella, bus, anillo, malla.
- **Interfaz física:** conectores, voltajes, frecuencias.
- **Control del medio:** acceso compartido (ej. CSMA/CD en Ethernet).

3. Medios de Transmisión

3.1. Medios Guiados

- **Par trenzado (UTP/STP):** económico, limitado a <100m.
- **Coaxial:** mayor inmunidad, uso en redes antiguas.
- **Fibra óptica:** alta velocidad, inmunidad EMI, ideal para largas distancias.

3.2. Medios No Guiados

- **Radiofrecuencia:** WiFi, Bluetooth.
- **Microondas, infrarrojo:** comunicación corta o dirigida.
- **Satélite y redes móviles (4G/5G):** cobertura amplia, latencia variable.



Comparativa de medios:

Medio	Velocidad	Distancia	Coste	Interferencia
UTP	Media	<100m	Bajo	Alta
Coaxial	Media	100–500m	Medio	Baja
Fibra óptica	Alta	km	Alta	Nula
Radiofrecuencia	Media	Variable	Medio	Alta

4. Adaptación al Medio

- **Modulación:** digital-analógica (ej. módem).
- **Repetidores/amplificadores:** compensan atenuación.
- **Multiplexores:** optimizan canales.
- **Codificación de línea:** reduce errores y garantiza sincronía.

5. Limitaciones de la Transmisión

- **Atenuación:** pérdida de señal → uso de repetidores.
- **Interferencias EMI:** evitables con fibra.
- **Ruido y diafonía:** distorsión → técnicas de corrección.
- **Retardo de propagación:** afecta tiempos de respuesta.
- **Capacidad del canal:** limita velocidad máxima.

6. Estándares del Nivel Físico

- **IEEE 802.3 (Ethernet):** UTP/fibra, hasta 400 Gbps.
- **IEEE 802.11 (WiFi):** Gbps, radiofrecuencia.
- **ITU G.652:** fibra óptica monomodo.
- **RS-232, USB, Bluetooth:** diversos usos y velocidades.

PROPUESTA DIDÁCTICA

Asignatura: Instalación de Redes (CFGM SMR)

1. Contextualización

- Aula-taller con materiales de red (cables, testers, conectores, WiFi).
- Alumnado de 16-20 años, nivel inicial técnico.

2. Objetivos

- Comprender los medios físicos y sus características.

- Fabricar cables de red y verificar su funcionalidad.
- Evaluar medios según criterios técnicos y contextuales.

3. Metodología

- **Aprendizaje basado en investigación y práctica (ABI + ABP).**
- Combinación de teoría, experimentación, comparación y exposición.

4. Atención a la Diversidad y DUA

- **Nivel III:** infografías, acompañamiento, videotutoriales.
- **Nivel IV:** roles diferenciados, apoyo visual y táctil, tiempos flexibles.
- **DUA:** contenidos multimodales (esquemas, vídeos, práctica), medios diversos de expresión (tablas, exposición oral), andamiaje personalizado.

5. Actividad Principal

“Comparador de Medios y Montaje de Cableado”

- Investigar y comparar tecnologías físicas (Ethernet, WiFi, fibra).
- Elaborar tabla comparativa.
- Fabricar cables UTP con conectores RJ-45.
- Verificar funcionalidad con tester y proponer usos reales:
 - Oficina (Ethernet).
 - Hospital (fibra óptica).
 - Videovigilancia urbana (WiFi 6 + fibra).

6. Evaluación

- **Instrumentos:** rúbrica técnica, observación directa, autoevaluación.
- **Criterios:** precisión en el montaje, rigor técnico en la comparación, presentación clara y justificada.

7. Conclusión Didáctica

El nivel físico, base de toda red, se convierte en un conocimiento tangible mediante prácticas reales. El enfoque activo fomenta la comprensión técnica, la autonomía y la toma de decisiones fundamentadas.

TEMA 64: FUNCIONES Y SERVICIOS DEL NIVEL DE ENLACE. TÉCNICAS. PROTOCOLOS

1. Introducción

- El nivel de enlace (capa 2 del modelo OSI) conecta el nivel físico con el nivel de red.
- Su misión: asegurar una transmisión libre de errores entre nodos directamente conectados, mediante la estructuración en tramas, detección de errores y control del medio.

2. Funciones Principales

2.1 Encapsulación en Tramas

- Inserta datos de usuario + dirección MAC + control de errores.
- Diferente según tecnología (Ethernet, WiFi, PPP).

2.2 Direccionamiento Físico

- Uso de direcciones MAC únicas.
- Comunicación en entornos LAN.

2.3 Detección y Corrección de Errores

- CRC, checksum, paridad.
- Retransmisión si hay errores (según protocolo).

2.4 Control de Flujo

- Evita saturación del receptor.
- Técnicas: ventanas deslizantes, ACK/NACK, XON/XOFF.

2.5 Control de Acceso al Medio (MAC)

- Determina quién transmite y cuándo (CSMA/CD, CSMA/CA, Token).

2.6 Reconocimiento de Recepción

- Servicios confirmados (ACK) o no confirmados (Ethernet).

3. Servicios al Nivel de Red

Tipo de Servicio	Descripción
No orientado a conexión	Sin sesión previa (Ethernet).
Orientado a conexión	Comunicación lógica (HDLC).
No confirmado	Sin acuse de recibo (rápido, no fiable).
Confirmado	Incluye ACK/NACK (más fiable).

4. Técnicas Relevantes

4.1 Control de Acceso

- **CSMA/CD**: Ethernet.
- **CSMA/CA**: WiFi.
- **Token Passing**: redes deterministas (FDDI).
- **MAC determinista**: redes industriales.

4.2 Control de Errores

- **CRC**: verificación potente.
- **Paridad, Hamming, ARQ**: corrección y retransmisión.

4.3 Control de Flujo

- Ventanas deslizantes, XON/XOFF, ACK/NACK.

PROPUESTA DIDÁCTICA

Asignatura: Redes Locales (CFGs ASIR)

1. Contextualización

- Aula TIC con acceso a red Ethernet y WiFi.
- Alumnado de 18-25 años, en formación técnica profesional.

2. Objetivos de Aprendizaje

- Comprender la estructura y función del nivel de enlace.
- Analizar tramas reales.
- Identificar funciones de control y acceso al medio.

3. Metodología

- **Aprendizaje activo y analítico.**

- Uso de Wireshark, escenarios simulados, retos comparativos.
- Trabajo cooperativo e investigación guiada.

4. Atención a la Diversidad y DUA

- **Nivel III:** visualización guiada de tramas, plantillas comparativas.
- **Nivel IV:** simplificación de tareas, apoyo técnico personalizado.
- **DUA:** representación múltiple (diagramas, análisis, práctica), implicación a través de exploración real, productos diversos (tabla, informe, vídeo explicativo).

5. Actividad Principal

“Analizador de Tramas Ethernet y WiFi”

- Captura y análisis de tramas en red local.
- Comparación entre tecnologías cableadas e inalámbricas.
- Estudio de campos: MAC, tipo, longitud, CRC.
- Reflexión sobre fiabilidad, eficiencia y seguridad.

6. Evaluación

- **Instrumentos:** rúbrica analítica, entrega comparativa, exposición oral.
- **Criterios:** correcta identificación de campos, análisis razonado, comprensión técnica y reflexiva.

7. Conclusión Didáctica

El nivel de enlace asegura la fiabilidad local de la red, y su estudio con herramientas reales como Wireshark permite al alumnado adquirir competencias técnicas fundamentales, pensamiento analítico y comprensión profunda del funcionamiento de redes modernas.

TEMA 65: FUNCIONES Y SERVICIOS DEL NIVEL DE RED Y DEL NIVEL DE TRANSPORTE. TÉCNICAS. PROTOCOLOS

1. Nivel de Red (Capa 3 del modelo OSI)

1.1 Funciones principales

- **Direccionamiento lógico:** uso de IP (IPv4/IPv6), jerárquico y escalable.
- **Encapsulación en paquetes:** encabezados con IP origen/destino, TTL, protocolo.
- **Enrutamiento:** estático o dinámico (RIP, OSPF, BGP).
- **Fragmentación y reensamblaje:** divide grandes paquetes según MTU (solo en origen en IPv6).
- **QoS y control de congestión:** priorización del tráfico crítico.

1.2 Técnicas clave

- **ARP:** resolución IP → MAC.
- **ICMP:** diagnóstico (ping, traceroute).
- **MPLS:** encaminamiento eficiente con etiquetas.

1.3 Protocolos destacados

- **IPv4:** dominante, uso de NAT.
- **IPv6:** más direcciones, sin NAT, mejor seguridad.
- **IPsec:** cifrado/autenticación de paquetes IP.

2. Nivel de Transporte (Capa 4 del modelo OSI)

2.1 Funciones principales

- **Multiplexación/demultiplexación:** mediante puertos (HTTP:80, HTTPS:443).
- **Establecimiento y cierre de conexión:** TCP (three-way handshake).
- **Control de flujo y errores:** ventanas deslizantes, checksum, retransmisión.
- **Segmentación y reensamblaje:** división en segmentos ordenados.

2.2 Técnicas clave

- **ACK/NACK, ventanas dinámicas, timeouts.**

2.3 Protocolos

- **TCP:** fiable, ordenado, conexión (navegación, FTP, email).
- **UDP:** rápido, sin conexión (VoIP, streaming).
- **QUIC:** sobre UDP con TLS integrado (usado en HTTP/3).
- **SCTP:** multistreaming y corrección avanzada.

2.4 Interacción con capa de red

- IP lleva los paquetes.
- TCP/UDP asegura entrega y orden extremo a extremo.

PROPUESTA DIDÁCTICA

Asignatura: Seguridad en Redes (CFGS ASIR)

1. Contextualización

- Aula con Wireshark, red local, acceso web.
- Alumnado de 18-25 años, perfil técnico-avanzado.

2. Objetivos

- Analizar funciones de los niveles 3 y 4.
- Observar seguridad extremo a extremo (IP, TCP, TLS).
- Representar conexiones y capas implicadas.

3. Metodología

- **Aprendizaje por indagación y demostración técnica.**
- Captura, análisis y representación gráfica de protocolos.

4. Atención a la Diversidad y DUA

- **Nivel III:** pautas paso a paso, plantillas de análisis.
- **Nivel IV:** análisis asistido, revisión personalizada.
- **DUA:** entrada múltiple (visual, práctica, textual), respuesta flexible (informe, infografía, vídeo).

5. Actividad Principal

“Rastreo de una conexión segura extremo a extremo”

- Acceso a una web HTTPS.
- Captura del tráfico con Wireshark.
- Análisis: DNS, IP, TCP, TLS.
- Creación de un diagrama de flujo con explicación de cada capa.

6. Evaluación

- **Instrumentos:** rúbrica de análisis, presentación de flujo, reflexión técnica.

- **Criterios:** identificación correcta de capas, comprensión del cifrado, claridad en representación.

7. Conclusión Didáctica

Entender los niveles de red y transporte empodera al alumnado para diagnosticar, optimizar y securizar redes. La práctica basada en análisis real conecta la teoría con la experiencia del mundo digital moderno.

TEMA 66: FUNCIONES Y SERVICIOS EN NIVELES SESIÓN, PRESENTACIÓN Y APLICACIÓN. PROTOCOLOS. ESTÁNDARES

1. Introducción

- Las capas superiores del modelo OSI (5 a 7) permiten que los servicios de red sean comprensibles, estructurados y seguros.
- Aunque en TCP/IP se integran en la capa de aplicación, su análisis independiente facilita el diagnóstico y diseño de comunicaciones avanzadas.

2. Capa de Aplicación (Nivel 7 OSI)

2.1 Funciones

- Interfaz entre usuario y red.
- Autenticación, acceso a recursos, gestión de servicios.
- Proporciona protocolos para aplicaciones concretas.

2.2 Protocolos Clave

- **HTTP/HTTPS**: navegación web.
- **SMTP/IMAP/POP3**: correo electrónico.
- **DNS/DHCP**: resolución de nombres y configuración.
- **FTP/SFTP**: transferencia de archivos.
- **SNMP**: monitorización de red.

2.3 Estándares

- RFCs de IETF, ISO/IEC 9594 (X.500).

3. Capa de Presentación (Nivel 6 OSI)

3.1 Funciones

- Traducción de formatos (ASCII, JSON, XML).
- Cifrado/descifrado (TLS/SSL).
- Compresión/descompresión.

3.2 Protocolos/Tecnologías

- **TLS/SSL**: seguridad extremo a extremo.
- **MIME**: contenidos multimedia.
- **ASN.1, XML, JSON**: definición estructural de datos.

4. Capa de Sesión (Nivel 5 OSI)

4.1 Funciones

- Establecimiento y finalización de sesiones.
- Sincronización y control de diálogo.
- Control de flujo de sesión.

4.2 Protocolos

- **NetBIOS, RPC, SMB.**

5. Interacción entre Capas y Unificación (TCP/IP)

- En TCP/IP, las capas 5-7 se integran como una sola.
- Ejemplo: HTTPS combina sesión (TLS handshake), presentación (cifrado) y aplicación (HTTP).

PROPUESTA DIDÁCTICA

Asignatura: Servicios de Red e Internet (CFGs ASIR)

1. Contextualización

- Aula con acceso a red y Wireshark.
- Alumnado técnico con conocimientos de protocolos básicos.

2. Objetivos

- Comprender el papel de las capas altas del modelo OSI.
- Observar y analizar la integración de sesión, presentación y aplicación en conexiones reales.
- Desarrollar competencias analíticas para diagnóstico de servicios.

3. Metodología

- **Estudio de casos reales + análisis con herramientas técnicas.**
- Aprendizaje basado en indagación estructurada (ABI).

4. Atención a la Diversidad y DUA

- **Nivel III:** guías paso a paso para capturas y tablas.
- **Nivel IV:** acompañamiento técnico, plantillas de análisis visual.
- **DUA:** múltiples representaciones (diagramas, tablas, texto), opciones de entrega diversa, andamiaje digital personalizado.

5. Actividad Principal

“Análisis de una comunicación HTTPS: descomponiendo la capa de aplicación”

- Captura de tráfico HTTPS con Wireshark.
- Identificación: handshake TLS (sesión), cifrado (presentación), HTTP (aplicación).
- Elaboración de tabla y diagrama de flujo.
- Reflexión sobre roles de cada capa en la seguridad y funcionalidad.

6. Evaluación

- **Instrumentos:** rúbrica de análisis, tabla técnica, autoevaluación reflexiva.
- **Criterios:** identificación precisa, comprensión conceptual, claridad explicativa.

7. Conclusión Didáctica

Estas capas son las más cercanas al usuario y fundamentales para garantizar una experiencia segura, eficiente y transparente. Comprender cómo interactúan es clave en la administración moderna de servicios y redes.

TEMA 67: REDES DE ÁREA LOCAL. COMPONENTES. TOPOLOGÍAS. ESTÁNDARES. PROTOCOLOS

1. Introducción

- Una **Red de Área Local (LAN)** conecta dispositivos en un área geográfica limitada (oficina, centro educativo).
- Permite compartir recursos, servicios y datos con alta velocidad y baja latencia.

2. Componentes de una LAN

2.1 Hardware

- **Estaciones de trabajo** (clientes), **servidores**, **dispositivos de red**.
- **Switches**: interconexión a nivel de enlace.
- **Routers**: interconexión con otras redes (WAN).
- **Puntos de acceso**: conectividad inalámbrica.
- **Cables UTP/STP**, conectores RJ-45, fibra óptica.

2.2 Software

- **Sistemas operativos de red** (Windows Server, Linux).
- **Protocolos y servicios de red**: DHCP, DNS, Samba, Active Directory.

3. Topologías de Red

Tipo	Descripción	Ventajas	Inconvenientes
Bus	Un solo cable, dispositivos en serie	Económica, sencilla	Colisiones, difícil de escalar
Estrella	Todos conectados a un switch	Fiable, fácil diagnóstico	Dependencia del nodo central
Anillo	Transmisión circular	Ordenada, sin colisiones	Poco flexible
Malla	Conexión redundante entre nodos	Alta disponibilidad	Coste elevado, compleja
Mixta	Combinación de las anteriores	Escalable, adaptable	Requiere diseño técnico

PROPUESTA DIDÁCTICA

Asignatura: Redes Locales / Implantación de Redes (CFGM SMR / CFGS ASIR)

1. Contextualización

- Aula-taller con equipamiento de red (switches, routers, cableado, PCs).
- Alumnado técnico-práctico, formación orientada al montaje y diagnóstico de redes.

2. Objetivos

- Identificar componentes físicos y lógicos de una LAN.
- Diferenciar topologías y estándares.
- Configurar redes funcionales y eficientes.

3. Metodología

- **Aprendizaje práctico por montaje y simulación de escenarios reales.**
- Uso de Packet Tracer, Wireshark y hardware físico.

4. Atención a la Diversidad y DUA

- **Nivel III**: simulaciones guiadas, prácticas paso a paso.
- **Nivel IV**: dinámicas por equipos, trabajo con plantillas.

- **DUA:** entrada múltiple (esquemas, vídeos, simuladores), expresión libre (informe, maqueta, vídeo explicativo).

5. Actividad Principal

“Diseña tu red local: desde el plano hasta la conexión”

- Diseño de topología (con o sin redundancia).
- Selección de componentes adecuados.
- Configuración IP, DHCP, DNS en entorno simulado o real.
- Comprobación de conectividad y rendimiento.

6. Evaluación

- **Instrumentos:** rúbrica de diseño, checklist técnico, prueba funcional.
- **Criterios:** coherencia del diseño, funcionalidad, documentación técnica.

7. Conclusión Didáctica

Dominar las LAN es esencial para cualquier profesional TIC. Conocer sus componentes, estándares y protocolos permite diseñar, implantar y mantener infraestructuras fiables y escalables.

TEMA 68: SOFTWARE DE SISTEMAS EN RED. COMPONENTES. FUNCIONES. ESTRUCTURA

1. Introducción

- El **software de sistemas en red** es el conjunto de programas que permiten la administración, control, seguridad y comunicación entre dispositivos conectados a una red.
- Su correcta implementación es clave para garantizar el funcionamiento eficiente y seguro de una infraestructura informática.

2. Funciones Principales

- **Comunicación entre dispositivos:** compartir datos, impresoras, servicios.
- **Gestión de recursos de red:** discos, usuarios, permisos.
- **Seguridad y control de acceso:** autenticación, cifrado, cortafuegos.
- **Supervisión y mantenimiento:** monitoreo, registro de eventos, actualizaciones.

3. Componentes del Software de Red

Componente	Función principal
Sistema operativo de red	Controla recursos compartidos y conexiones (ej. Windows Server, Linux).
Servicios de red	DHCP, DNS, FTP, HTTP, LDAP.
Protocolos de red	TCP/IP, SMB, NFS, SNMP, SSH.
Controladores (drivers)	Interfaz entre hardware y software de red.
Herramientas de gestión	Monitorización (Nagios), configuración (Ansible, Puppet).

PROPUESTA DIDÁCTICA

Asignatura: Administración de Sistemas / Redes Locales (CFGM SMR / CFGS ASIR)

1. Contextualización

- Entorno de prácticas con máquinas virtuales y software de servidor.
- Alumnado con perfil técnico, orientado a instalación y mantenimiento.

2. Objetivos

- Identificar y configurar los componentes clave del software de red.
- Entender la estructura lógica del sistema en red.
- Asegurar conectividad, funcionalidad y seguridad en un entorno real.

3. Metodología

- **Aprendizaje procedimental y basado en tareas.**
- Instalación y configuración paso a paso de servicios y roles.

4. Atención a la Diversidad y DUA

- **Nivel III:** interfaz gráfica, guías visuales, comandos simplificados.
- **Nivel IV:** soporte directo, tareas adaptadas.
- **DUA:** múltiples representaciones (diagramas, vídeos, guías), opciones de trabajo (individual/grupal, simulación/montaje real).

5. Actividad Principal

“Implanta tu sistema de red básico”

- Instalación de un SO de red (ej. Ubuntu Server, Windows Server).
- Configuración de servicios básicos (DHCP, DNS, compartición de archivos).
- Implementación de seguridad básica (firewall, SSH, control de usuarios).
- Verificación y prueba con clientes simulados.

6. Evaluación

- **Instrumentos:** rúbrica de instalación/configuración, diario de prácticas, cuestionario técnico.
- **Criterios:** funcionalidad, seguridad mínima aplicada, organización y documentación del sistema.

7. Conclusión Didáctica

El software de sistemas en red permite gestionar de forma eficiente los recursos tecnológicos de una organización. Su dominio capacita al alumnado para diseñar entornos estables, escalables y seguros en el ámbito profesional TIC.

TEMA 70: DISEÑO DE SISTEMAS EN RED LOCAL. PARÁMETROS DE DISEÑO. INSTALACIÓN Y CONFIGURACIÓN DE SISTEMAS EN RED LOCAL

1. Introducción

- El diseño de un sistema de red local (LAN) **determina su eficiencia, escalabilidad y seguridad.**
- Incluye fases desde el análisis de requisitos hasta la implementación, implicando decisiones técnicas y estratégicas.

2. Fases del Diseño de una Red LAN

2.1 Análisis de necesidades

- Recuento de dispositivos, usuarios, servicios y seguridad.
- Previsión de crecimiento y disponibilidad requerida.

2.2 Diseño lógico

- Selección de topología (estrella, malla, híbrida).
- Segmentación mediante subredes, VLANs.
- Asignación de direcciones IP (estáticas, dinámicas).

2.3 Diseño físico

- Distribución de puntos de red, armarios de comunicaciones.
- Elección de medios (UTP, fibra), switches, routers.
- Planificación de redundancia y alimentación.

2.4 Plan de seguridad

- Control de acceso físico y lógico.
- Políticas de firewall, DMZ, autenticación.

PROPUESTA DIDÁCTICA

Asignatura: Implantación de Redes / Administración de Sistemas (CFGS ASIR / CFGM SMR)

1. Contextualización

- Aula técnica con racks, switches, routers, cableado y software de simulación.
- Alumnado con perfil técnico que requiere integrar teoría y práctica.

2. Objetivos

- Diseñar redes coherentes, escalables y seguras.
- Instalar y configurar una LAN desde cero.
- Verificar su funcionalidad con herramientas profesionales.

3. Metodología

- **Aprendizaje basado en proyectos reales y resolución de problemas.**
- Trabajo en equipos con roles técnicos diferenciados.

4. Atención a la Diversidad y DUA

- **Nivel III:** simulación con Packet Tracer, uso de esquemas y plantillas.
- **Nivel IV:** maquetas físicas simples, apoyo guiado.
- **DUA:** variedad de materiales (manuales, vídeos, visual thinking), expresión múltiple (presentación, infografía, vídeo tutorial).

5. Actividad Principal

“Diseña e implanta tu red de aula”

- Diseño físico y lógico completo de una red para un aula o pequeña empresa.
- Instalación de cableado y montaje de switches, configuración IP, DNS, DHCP, usuarios.
- Simulación de pruebas de conectividad, seguridad y rendimiento.
- Documentación técnica y presentación de la red implementada.

6. Evaluación

- **Instrumentos:** rúbrica de diseño, checklist de instalación, observación directa.
- **Criterios:** coherencia técnica, documentación precisa, red funcional y segura.

7. Conclusión Didáctica

Diseñar e instalar una red local con criterio profesional es una competencia clave en FP TIC. Permite al alumnado integrar conocimientos técnicos y desarrollar la capacidad de planificación, análisis y trabajo en equipo aplicable al entorno real.

TEMA 71: EXPLOTACIÓN Y ADMINISTRACIÓN DE SISTEMAS EN RED LOCAL. FACILIDADES DE GESTIÓN

1. Introducción

La **explotación** de una red local se refiere a su uso operativo diario: disponibilidad de servicios, acceso a recursos, soporte a usuarios y resolución de incidencias.

La **administración** implica tareas más profundas: planificación, configuración, seguridad, supervisión, documentación y mejora continua.

Ambas dimensiones son esenciales para mantener redes funcionales, seguras, escalables y adaptadas a las necesidades del entorno (educativo, empresarial, etc.).

2. Objetivos de la Gestión en Redes LAN

Objetivo	Descripción y ejemplos
2.1 Disponibilidad y supervisión	Supervisión de dispositivos (CPU, RAM, conectividad), notificación de caídas o degradaciones. Ej.: alertas SNMP por fallo de red o servidor.
2.2 Rendimiento y optimización	Control de tráfico (QoS), balanceo de carga, análisis de uso para prevenir cuellos de botella.
2.3 Seguridad	Gestión de usuarios y permisos, cortafuegos (UFW), detección de intrusos (Snort), bloqueo automático (Fail2Ban).
2.4 Escalabilidad	Organización IP, VLANs, segmentación de red, planificación documentada para crecimiento.
2.5 Resiliencia	Copias de seguridad, alta disponibilidad, redundancia mediante RAID, clustering o servidores en espejo.
2.6 Gestión de actualizaciones	Aplicación automática de parches y actualizaciones mediante WSUS, APT, YUM, Chocolatey, etc.
2.7 Automatización	Ejecución de tareas programadas: reinicios, limpieza de logs, backups, mediante cron, Bash, PowerShell.
2.8 Documentación técnica	Inventariado, topologías, configuraciones, cambios documentados. Herramientas como NetBox, CMDB, diagramas de red.
2.9 Acceso remoto seguro	Conexiones cifradas por SSH, RDP, VPN, VNC. Control y auditoría de accesos remotos.
2.10 Gestión de contenedores y virtualización ligera	Uso de tecnologías como Docker para desplegar servicios en entornos aislados, facilitando pruebas, migración y escalabilidad.
2.11 Administración centralizada con GPOs	En entornos Windows, aplicación de políticas de grupo (GPO) para controlar configuraciones, seguridad y comportamiento de los usuarios.

3. Herramientas de Gestión y Administración

Área	Herramientas principales	Aplicaciones prácticas

Monitorización	Zabbix, Nagios, PRTG, Grafana	Supervisión de recursos, notificaciones por SNMP, paneles visuales
Configuración central	Ansible, Puppet, RMM	Instalación de software, cambios masivos en múltiples equipos
Seguridad	Snort, Fail2Ban, UFW, antivirus	Prevención y respuesta ante intrusiones
Automatización	Bash, PowerShell, cron	Scripts de mantenimiento, tareas recurrentes
Gestión de actualizaciones	WSUS, APT, YUM, Chocolatey	Aplicación centralizada de parches del sistema
Documentación	NetBox, CMDB, diagramas de red	Topología de red, seguimiento de cambios, inventario de activos
Acceso remoto	SSH, RDP, VPN, VNC	Administración desde ubicaciones externas, soporte remoto
Contenedores	Docker, Docker Compose	Despliegue de servicios en contenedores aislados y portables
Administración Windows	Active Directory, GPOs, RSAT	Control de usuarios, configuración automatizada, políticas globales

4. Protocolos y conceptos clave

- **SNMP**: protocolo estándar para monitorizar y gestionar dispositivos de red.
- **SSH/RDP/VPN**: canales seguros para administración remota.
- **CLI vs GUI**: gestión por línea de comandos permite mayor control y automatización; las interfaces gráficas facilitan el acceso visual.
- **Gestión centralizada vs local**: en redes grandes se centralizan configuraciones, registros y usuarios.
- **Eventos y alertas**: herramientas como Graylog o ELK Stack permiten centralizar logs y establecer alertas automáticas.

5. Implicaciones profesionales y buenas prácticas

- La administración eficaz reduce tiempos de inactividad, mejora la productividad y fortalece la seguridad de la organización.
- Documentar cambios, utilizar nomenclatura coherente y mantener inventarios actualizados son prácticas clave.
- El trabajo colaborativo en redes implica establecer roles técnicos, coordinar incidencias y registrar decisiones.

6. Conclusión

La explotación y administración de sistemas en red local exige combinar conocimientos técnicos con planificación, organización y visión preventiva. El uso de herramientas especializadas, protocolos adecuados y buenas prácticas permite a los técnicos mantener infraestructuras robustas, seguras y listas para crecer.

PROPUESTA DIDÁCTICA

Asignatura: Implantación de Sistemas Operativos (CFGM SMR)

1. Contextualización

- Aula de sistemas con máquinas virtuales LAN simuladas.
- Alumnado con perfil práctico, iniciándose en administración.

2. Objetivos

- Diagnosticar y resolver incidencias comunes.
- Utilizar herramientas de monitorización y automatización.
- Documentar y aplicar medidas preventivas.

3. Metodología

- **Aprendizaje basado en retos** con fallos simulados.
- Exploración activa y resolución colaborativa.

4. Atención a la Diversidad y DUA

- **Nivel III:** guía con instrucciones paso a paso, soporte visual.
- **Nivel IV:** simplificación de comandos, trabajo tutorizado.
- **DUA:** múltiples formas de representar resultados (gráficos, informes), trabajo en pareja, aprendizaje adaptativo.

5. Actividad Principal

“Simulación de administración de red con fallo controlado”

- En parejas, gestionar una LAN virtual.
- Resolver fallos introducidos por el docente (corte DHCP, cambio IP, tráfico anómalo).
- Uso de herramientas: ping, netstat, ip, htop, nmap.
- Elaboración de informe técnico con medidas de mejora.

6. Evaluación

- **Instrumentos:** rúbrica de solución de incidencias, diario técnico, defensa oral.
- **Criterios:** identificación del fallo, solución eficaz, documentación clara, trabajo en equipo.

7. Conclusión Didáctica

Gestionar una red es combinar conocimiento técnico, organización y previsión. Esta práctica fomenta la autonomía técnica y prepara al alumnado para entornos reales de administración con enfoque proactivo y seguro.

Tema 72. La seguridad en sistemas en red. Servicios de seguridad. Técnicas y sistemas de protección. Estándares.

1. Fundamentos de la seguridad en red

1.1. Importancia de la seguridad en red

- Las redes son vectores de ataque constantes en entornos conectados.
- Riesgos: pérdida de disponibilidad, integridad y confidencialidad.

1.2. Necesidad de protección

- La ciberseguridad comienza en la red: proteger el canal de datos es esencial.

2. Servicios de seguridad

2.1. Autenticación y autorización

- Métodos: contraseñas, autenticación multifactor (MFA), biometría.
- Protocolos: Kerberos, OAuth2, SSO (Single Sign-On).

2.2. Control de acceso

- Modelos: RBAC (control basado en roles), ABAC (basado en atributos).
- Tecnologías: VLANs, NAC (Network Access Control), listas de control de acceso (ACL).

2.3. Cifrado y no repudio

- Herramientas: HTTPS, VPN, cifrado de discos, firmas digitales.

2.4. Auditoría y SIEM

- SIEM: gestión centralizada de eventos e información de seguridad.
- Herramientas: Wazuh, Splunk; detección de anomalías mediante análisis en tiempo real.

3. Técnicas de protección

3.1. Segmentación de red

- Uso de VLANs, microsegmentación y redes definidas por software (SDN).

3.2. Bastionado (hardening)

- Eliminación de servicios innecesarios, refuerzo de configuraciones, automatización con Ansible.

3.3. Prevención de amenazas

- Herramientas: EDR (Endpoint Detection and Response), DNSSEC, bloqueo de direcciones IP.

4. Defensa en profundidad

4.1. Firewalls de nueva generación (NGFW)

- Inspección profunda de paquetes, filtrado por aplicación, bloqueo en tiempo real.

4.2. Sistemas IDS e IPS

- IDS: detección de intrusiones. IPS: prevención activa.

4.3. Copias de seguridad

- Regla 3-2-1: 3 copias, en 2 soportes diferentes, 1 externa.

5. Normativa y estándares

5.1. Estándares técnicos

- ISO 27001, NIST SP 800-53, COBIT, MITRE ATT&CK.

5.2. Legislación vigente

- RGPD, LOPDGDD, ENS (Esquema Nacional de Seguridad), Directiva NIS2.

5.3. Evaluación de riesgos

- MAGERIT y PILAR: análisis detallado de activos, amenazas y vulnerabilidades.

6. Amenazas actuales

- Man-in-the-Middle (MITM): interceptación si el tráfico no está cifrado.
- Ransomware: secuestro de datos mediante cifrado.
- Fallos de configuración en entornos cloud.
- Ataques DDoS: saturación de servicios mediante tráfico masivo.

7. Concienciación y formación

- El usuario como primera línea de defensa.
- Simulacros y campañas de concienciación: phishing, ransomware, ingeniería social.
- Actividades de ciberseguridad: CyberCamp, CTFs, test de impacto.

PROPUESTA DIDÁCTICA: “CIBERDEFENSORES EN RED”

A. Contextualización

- Nivel educativo: 1.º FP Grado Superior en Administración de Sistemas Informáticos en Red (ASIR).
- Módulo: Seguridad y alta disponibilidad.

B. Objetivos de aprendizaje

- Aplicar medidas de protección de red y respuesta a incidentes.

- Identificar amenazas y aplicar estándares y buenas prácticas.
- Fomentar la responsabilidad digital y el trabajo colaborativo.

C. Metodología

- Aprendizaje basado en proyectos (ABP) y gamificación.
- Dinámica de roles: analista SIEM, responsable de red, backup, hardening.
- Aprendizaje activo mediante retos progresivos.

D. Actividad principal

El alumnado trabajará en equipos simulando un **Centro de Operaciones de Seguridad (SOC)**, con el objetivo de diseñar, implementar y defender una infraestructura de red ante posibles ciberataques, en un entorno virtual controlado.

Entorno y herramientas

- **TryHackMe**: para escenarios prácticos de detección de amenazas y análisis forense.
- **VirtualBox**: para montar sistemas operativos vulnerables, firewalls, y estaciones de monitorización y en el simular red interna.

Fases de la actividad

1. **Diseño de la infraestructura**
 - Creación de una red segmentada con zonas críticas (intranet, DMZ, servidores).
 - Configuración de firewalls, sistemas IDS/IPS (simulados o conceptuales), control de acceso y monitorización.
2. **Implementación de medidas defensivas**
 - Hardening básico de sistemas virtualizados.
 - Simulación de reglas de cortafuegos, ACLs, autenticación, y políticas de red.
3. **Simulación de amenazas**
 - Recepción de un escenario de ataque simulado (ransomware, escaneo de puertos, escalada de privilegios, etc.).
 - Análisis del ataque, detección temprana, respuesta y recuperación.
4. **Evaluación continua y retroalimentación**
 - Análisis técnico del rendimiento (resiliencia, tiempos de respuesta).
 - Valoración organizativa (rol asumido por cada miembro, comunicación del equipo, documentación técnica).
 - Autoevaluación y coevaluación mediante rúbricas técnicas y de trabajo en grupo.

Resultados esperados

- Infraestructura defendida con éxito ante un ataque simulado.
- Registro y documentación técnica del incidente.
- Presentación final justificando las decisiones técnicas, respuestas y medidas correctoras.

E. Atención a la diversidad (niveles III y IV)

- Nivel III: apoyo visual, guías paso a paso, grupos heterogéneos.
- Nivel IV: adaptación de tareas, refuerzo individual, recursos accesibles.

F. Diseño Universal para el Aprendizaje (DUA)

- Representación: vídeos, esquemas, simulaciones.
- Acción y expresión: elección de herramientas, roles diferenciados.
- Implicación: enfoque competitivo, trabajo por equipos, retroalimentación constante.

G. Evaluación

- Rúbricas por competencias técnicas y actitudinales.
- Instrumentos: observación directa, diarios de aprendizaje, checklist de configuración.
- Criterios: efectividad defensiva, trabajo en equipo, resolución de incidentes.

H. Conclusión didáctica

- La ciberseguridad es una competencia transversal y crítica.
- Simular un SOC permite integrar teoría, práctica y conciencia ética.
- El alumnado se convierte en protagonista de su aprendizaje, desarrollando competencias digitales avanzadas.

Tema 73. Evaluación y mejora de prestaciones en un sistema en red. Técnicas y procedimientos de medidas.

1. Introducción

Evaluar el rendimiento de una red es esencial para garantizar su estabilidad, calidad de servicio y capacidad de crecimiento. Este proceso permite identificar cuellos de botella, prever fallos y aplicar mejoras técnicas. Involucra el uso coordinado de métricas, herramientas de análisis y estrategias de optimización.

2. Parámetros Clave de Evaluación

Métrica	Descripción	Unidad o Indicador
Ancho de banda	Capacidad máxima de transmisión de datos	Mbps / Gbps
Latencia	Tiempo de ida y vuelta de un paquete	ms (milisegundos)
Jitter	Variabilidad en la latencia entre paquetes	ms
Pérdida de paquetes	Porcentaje de paquetes que no llegan correctamente	%
Uso de CPU/RAM	Carga de recursos en servidores o dispositivos de red	%
Conexiones activas	Número simultáneo de sesiones en la red	unidades
Tiempo de respuesta	Velocidad de servicios clave (DNS, DHCP, web...)	ms

3. Técnicas de Medición y Herramientas

3.1 Pruebas activas

Generan tráfico sintético para evaluar el comportamiento de la red en condiciones controladas.

Herramientas: [iPerf](#), [Ping](#), [Traceroute](#), [NetStress](#).

3.2 Monitorización pasiva

Observan el tráfico real sin añadir carga. Utilizan protocolos como **SNMP** para obtener métricas desde routers, switches o servidores.

Herramientas: [Wireshark](#), [Nagios](#), [PRTG](#), [Zabbix](#).

3.3 Logs y análisis histórico

Analizan registros para detectar patrones de fallo o tendencias de uso. Se pueden automatizar con scripts para generar alertas e informes.

Fuentes: [/var/log](#), [event viewer](#), [syslog](#), gráficas y dashboards.

3.4 Simulación de carga

Evalúan el rendimiento bajo estrés y validan el comportamiento ante picos de demanda.

Herramientas: [JMeter](#), [Cisco Packet Tracer](#), [GNS3](#).

3.5 Métricas avanzadas y visualización

Se incluyen indicadores como número de conexiones activas, congestión en enlaces troncales o tiempos de servicios. La visualización mediante dashboards interactivos (ej. en [Grafana](#)) permite una interpretación clara y profesional.

3.6 Calidad de Servicio (QoS)

La mejora del rendimiento puede implicar políticas de QoS para priorizar tráfico crítico (voz, vídeo), lo que mejora la experiencia del usuario en redes saturadas.

3.7 Escenarios mixtos y entornos reales

El análisis se aplica también a redes con dispositivos cableados, inalámbricos o virtuales, simulando entornos corporativos reales y complejos.

PROPUESTA DIDÁCTICA

Asignatura: *Redes de Área Local* (CFGM SMR)

1. Contextualización

El aula dispone de red física y simulada, equipos de medición y servidores para prácticas. El alumnado tiene conocimientos básicos en redes, sistemas y seguridad.

2. Objetivos

- Aplicar herramientas de diagnóstico de red.
- Interpretar métricas de rendimiento y detectar cuellos de botella.
- Configurar sistemas de monitorización activa y pasiva.
- Proponer y justificar mejoras técnicas fundamentadas.

3. Metodología

Aprendizaje activo basado en la detección y resolución de problemas reales. Se realiza evaluación comparativa antes y después de las mejoras aplicadas.

4. Atención a la Diversidad y DUA

- **Nivel III:** plantillas guiadas, herramientas visuales interactivas.
- **Nivel IV:** tareas paso a paso, trabajo tutorizado en parejas.
- **DUA:** recursos accesibles (vídeos, esquemas, simuladores), formatos libres de entrega (informe, presentación, panel digital).

5. Actividad Principal: “Audita y mejora tu red”

El alumnado llevará a cabo una auditoría completa sobre una red simulada, utilizando **Cisco Packet Tracer** como entorno de prácticas. La actividad combina análisis técnico, toma de decisiones fundamentadas y documentación profesional:

- **Monitorización inicial:** análisis de parámetros clave (latencia, tráfico, pérdida de paquetes) mediante comandos y herramientas del simulador.
- **Diagnóstico de rendimiento:** identificación de cuellos de botella o configuraciones ineficientes a partir de los datos obtenidos.
- **Propuesta de optimización:** diseño de un plan de mejora con acciones como reconfiguración de switches, modificación de topologías o aplicación básica de QoS.
- **Implementación y validación:** aplicación de las mejoras y comprobación de su impacto mediante pruebas comparativas.
- **Documentación y defensa:** elaboración de un informe técnico y exposición oral de resultados, justificando cada intervención.

6. Evaluación

- **Instrumentos:** checklist técnico, análisis de métricas, rúbrica de intervención.
- **Criterios:** identificación rigurosa de problemas, aplicación de soluciones justificadas, claridad y profesionalidad en la presentación de resultados.

7. Conclusión Didáctica

La capacidad de evaluar y optimizar redes es esencial para cualquier técnico en sistemas. Esta competencia fomenta una actitud proactiva, mejora la empleabilidad y prepara al alumnado para enfrentar entornos conectados complejos con criterios técnicos sólidos y autonomía.

Tema 74. Sistemas multimedia.

1. Definición y contexto

1.1. ¿Qué es un sistema multimedia?

- Conjunto de tecnologías que integran texto, imagen, audio, vídeo, animación y datos en tiempo real.
- Aplicaciones: educación (pizarras digitales), medicina (imagen diagnóstica), control remoto (drones), entretenimiento (videojuegos, RA).
- Incorporación de inteligencia artificial: reconocimiento facial, generación de voz e imagen.

2. Representación digital de medios

2.1. Imagen

- Formatos RAW, BMP, sin compresión.
- Raster: JPEG, PNG (píxeles, pierden calidad al escalar).
- Vectorial: SVG (formas matemáticas, calidad escalable).
- Canal alfa: gestión de transparencia.

2.2. Audio

- Tasa de muestreo: frecuencia de captura (ej. 44.1 kHz).
- Bitrate: calidad versus tamaño.
- Formatos: WAV (sin compresión), FLAC (sin pérdida), MP3/AAC (con pérdida).

2.3. Vídeo

- FPS: fluidez (30 fps estándar, 60 fps mayor realismo).
- Códec: compresión (H.264); contenedor: empaquetado (MP4).

3. Procesamiento multimedia

3.1. Transformadas

- Fourier: análisis de frecuencias (audio).
- DCT: base de JPEG.
- Wavelets: compresión multiescala (JPEG2000).

3.2. Convoluciones

- Aplicación de filtros a imágenes.
- Base de redes neuronales convolucionales (visión artificial).

4. Transmisión multimedia

- Protocolos adaptativos: HLS, DASH (ajuste de calidad).
- Protocolos en tiempo real: RTMP, RTSP (baja latencia).

5. Inteligencia Artificial en multimedia

5.1. Generación

- DALL·E, Stable Diffusion, voice cloning, NeRF.

5.2. Análisis

- YOLO, DETR (detección objetos).
- CLIP, GPT-4V (relación imagen-texto).

6. Herramientas

- FFmpeg: conversión, edición por línea de comandos.
- OpenCV: visión artificial.
- MediaPipe: detección de gestos en móviles.

7. Tendencias futuras

- Codificación neural, vídeo volumétrico, edge computing, interfaces adaptativas.

8. Ética y legislación

- Deepfakes y manipulación audiovisual.
- Sesgos algorítmicos.
- IA Act (UE): marco legal según nivel de riesgo.

9. Conclusión

- Los sistemas multimedia evolucionan hacia la comprensión y generación inteligentes de contenido.
- Su diseño debe equilibrar eficiencia técnica, ética y usabilidad.

PROPUESTA DIDÁCTICA: “ENTRENADOR MULTIMEDIA: CREA UNA APP DE FITNESS INTERACTIVO”

A. Contextualización

- Nivel educativo: 2.º curso de Grado Superior en DAM.
- Módulo: Multimedia y dispositivos móviles.

B. Objetivos de aprendizaje

- Integrar medios audiovisuales en apps Android.

- Optimizar la compresión y la reproducción multimedia.
- Diseñar experiencias interactivas y accesibles.

C. Metodología

- Aprendizaje basado en proyectos (ABP).
- Trabajo en equipo con división de roles técnicos.






D. Actividad principal

Proyecto: “Mi entrenador personal: diseña tu app fitness”

Objetivo general

Desarrollar una aplicación Android funcional que simule el comportamiento de un **entrenador personal digital**, integrando **contenidos multimedia**, **control de rutinas**, **motivación sonora** y **experiencia personalizada**, con recursos tanto online como offline.

Funcionalidades obligatorias mínimas

1.  **Reproductor de vídeos de ejercicios**
 - Uso de vídeos propios (grabados con móvil/OBS) o libres (p. ej., de Pixabay o Pexels).
 - Organización por tipo: fuerza, cardio, estiramiento.
2.  **Audios de instrucciones y motivación**
 - Integración de audios personalizados grabados por el alumnado o generados con IA/texto a voz.
 - Asociados a ejercicios o intervalos específicos.
3.  **Temporizador de rutinas configurable**
 - Selección de duración, descanso, número de ciclos.
 - Posible modo Tabata, HIIT, etc.
 - Cuenta atrás visual y sonora.
4.  **Dinamizador virtual**
 - Mensajes motivadores automáticos en puntos clave (“¡Vamos!”, “Último ejercicio...”).
 - Se pueden mostrar en pantalla o por audio/texto hablado.
5.  **Modo offline + control de calidad**
 - Opción para guardar vídeos/audio en memoria local.
 - Control de bitrate/resolución para dispositivos con poca capacidad o conexión lenta.

E. Atención a la diversidad

- Nivel III: plantillas, videotutoriales, apoyo técnico continuo.
- Nivel IV: descomposición de tareas, soporte individualizado.

F. DUA

- Representación: vídeos subtitrados, interfaces intuitivas.
- Expresión: variedad de herramientas y temas.
- Implicación: aplicación real, presentación gamificada.

G. Evaluación

- Rúbricas: integración técnica, diseño, accesibilidad y documentación.
- Instrumentos: presentación oral, prueba funcional, memoria técnica.

H. Conclusión didáctica

- Desarrollo de competencias en programación, tratamiento multimedia y ética digital.
- La app como producto funcional, motivador y aplicable en contextos reales.