Examen Sistemas Distribuidos

Índice

- Examen Sistemas Distribuidos
- Índice
- Tema 1 (Computación Distribuida)
  - Historia
  - Cluster Computing
  - Internet Computing
  - Grid Computing
  - Cloud Computing
  - Comparación
- Tema 2 (Sistemas Distribuidos Tipo Cluster)
  - Introducción
    - DDP (Distributed Data Processing)
    - Nuevos paradigmas que aparecen en un sistema distribuido
    - Papel del Sistema Operativo Distribuido
  - Computación Cliente/Servidor (C/S)
  - Middleware
  - Paso de mensajes (Message Passing)
    - Variantes
  - RPC — Remote Procedure Call
    - Ventajas
    - Funcionamiento con "stubs" (esqueletos)
    - Representación de parámetros
  - Clusters
    - Tipos de cluster
    - Configuración
    - Objetivo SO
  - Gestión de procesos distribuidos y migración
    - Migración
    - Por qúe migrar
    - Preguntas clave de migración
    - Estrategias para migrar memoria
    - Migración de Threads y Ficheros Abiertos
    - Negociación de migración
  - Exclusión mutua en distribuido
    - Solución: Cola Distribuida
  - Virtualización
    - Motivos
    - Ventajas
    - Hipervisor / VMM
    - Tipos de Virtualización

- Kubernetes y Docker
  - Kubernetes
    - ¿Qué ofrece?
    - ¿Qué no ofrece?
    - Arquitectura: Cluster de Kubernetes
    - Pods
    - Deployment
    - Red en Kubernetes: CNI (Cluster Network Interface)
  - Docker
    - Contenedor
- Tema 3 (AWS)
  - EC2 (Elastic Compute Cloud)
  - S3 (Simple Storage Service)
  - RDS (Relational Database Service)
  - AWS Lambda
  - Grupos de seguridad (Security Groups)

Tema 1 (Computación Distribuida)

Historia

- **1950s:** separación mainframe-terminal + time-sharing (compartición de recursos).
- **1960s:** McCarthy propone "computación como servicio público".
- **1966:** Parkhill formaliza ideas como elasticidad e "ilusión de suministro infinito".
- **1990s:** Internet + clusters por abaratamiento del hardware.
- **2000s:** Grid computing, recursos heterogéneos distribuidos globalmente.
- **2006+:** Cloud (AWS) consolida elasticidad y enfoque "servicio".

Cluster Computing

Conjunto de máquinas conectadas (normalmente en el mismo centro / organización) que se comportan como un sistema potente para ejecutar tareas.

- Recursos dedicados (infraestructura preparada para eso).
- Alta velocidad interna, administración centralizada.
- Escalable, pero con límites.
- Mantenimiento y ampliación costosa en tamaños grandes.
- Infrautilización: si es dedicado, puede estar sin usarse muchas horas.

Internet Computing

Uso de Internet como infraestructura que habilita computación distribuida a gran escala.

Grid Computing

Modelo que permite acceder a recursos computacionales geográficamente dispersos, heterogéneos y sin control centralizado único.

Cloud Computing

Computación como servicio con elasticidad, aprovisionamiento flexible y orientación comercial en muchos casos.

Cloud retoma la computación como servicio y la lleva a un modelo más práctico, automatizado y comercial.

Comparación

- **Cluster:** Todo en un mismo sitio, controlado, dedicado.

- **Internet computing:** Internet como base/medio.
- **Grid:** Unión de recursos dispersos (heterogéneos).
- **Cloud:** Servicios elásticos aprovisionados "a demanda".

Tema 2 (Sistemas Distribuidos Tipo Cluster)

Introducción

Un sistema distribuido tipo cluster se ve como un conjunto de nodos completos e independientes, unidos por una red. Se suele llamar multicomputador. Su rendimiento depende del S.O. distribuido / software de gestión.

DDP (Distributed Data Processing)

Para explotar los sistemas distribuidos, se usan estrategias de procesamiento de datos distribuido (DDP):

- Computadores dispersos, conectados por red
- Buscan eficiencia + paralelismo + replicación
- No todos los nodos tienen que ser iguales (heterogeneidad)
- Necesidad de tolerancia a fallos: pérdidas de red, caídas, datos

Características:

- **Disponibilidad:** Si cae un nodo, otro replicado puede asumir carga
- **Compartición de recursos:** Hardware/software costoso compartido, BD distribuida
- **Crecimiento incremental:** Añadir máquinas sin tirar lo viejo
- **Productividad / rendimiento:** Repartir trabajo por nodos

Nuevos paradigmas que aparecen en un sistema distribuido

Cosas importantes que cambian respecto a un sistema "normal":

- No existe una memoria única -> no hay un rango único de direcciones
- Comunicación basada en paso de mensajes
- Gestión/sincronización de recursos compartidos
- Esquemas típicos:
    - Maestro/Esclavo
    - Cliente/Servidor

Papel del Sistema Operativo Distribuido

El SO debe dar soporte a:

- intercambio de datos
- alta disponibilidad y prestaciones

- gestión de procesos distribuida
- seguridad

Computación Cliente/Servidor (C/S)

- Clientes: suelen ser terminales/estaciones con interfaz para el usuario
- Servidor: más complejo, ofrece servicios compartidos

Además aparecen dos piezas clave:

- API: funciones/programas que permiten comunicarse
- Middleware: capa de software + controladores + API para dar visión "global" del sistema

Tipos de aplicaciones:

- **Procesamiento tipo host:** todo en el servidor, cliente "tonto"
- **Procesador tipo servidor:** cliente proporciona GUI y servidor realiza operaciones (web, BD)
- **Basado en cliente:** casi todo en cliente; servidor valida (ej: juegos en red)
- **Cooperativo:** ambos trabajan repartiendo carga (ej: BD + sistema de ficheros distribuido)

Middleware

El middleware aparece porque la evolución C/S crea problemas de estandarización: muchas plataformas, muchas formas distintas de acceso.

Middleware = "software intermedio" entre aplicación, comunicaciones y S.O.

Paso de mensajes (Message Passing)

En distribuido, para comunicar y sincronizar procesos se usa paso de mensajes.

Funciones:

- **Send:** Enviar mensaje con parámetros + nodo destino (o broadcast)
- **Receive:** Recibir normalmente desde buffer, puede indicar nodo o recibir de cualquiera

Variantes

Fiable vs No Fiable:

- **Fiable:** Garantiza entrega si es posible (ACK, reenvío, reordenación…)
- **No fiable:** No garantiza, menos sobrecarga; si se quiere fiabilidad, la implementa la app

Bloqueante vs No Bloqueante:

- **Bloqueante:** La llamada espera a que el envío/recepción se complete

- **No bloqueante:** Vuelve rápido (mensaje en cola); se usa variable estado para saber cuándo termina

RPC — Remote Procedure Call

RPC es una variante del paso de mensajes que **encapsula la comunicación** y permite que el programador "vea" la invocación como una función normal.

Ventajas

- Definir interfaces remotas con operaciones, nombres y tipos -> documentable y comprobable
- El código de comunicación puede generarse automáticamente
- Facilita módulos cliente/servidor portables entre arquitecturas

Funcionamiento con "stubs" (esqueletos)

Ejemplo: `CALL P(X,Y)`

- en el cliente hay un stub/esqueleto
- empaqueta parámetros, identifica nodo remoto, hace send/receive
- en el servidor hay otro stub que recibe, ejecuta localmente y devuelve resultado

Representación de parámetros

Problema: cliente y servidor pueden tener:

- arquitecturas diferentes
- lenguajes distintos
- representaciones distintas de enteros/float/estructuras

Solución: convertir a una representación estándar.

Coste: más trabajo en envío/recepción, beneficio: interoperabilidad.

Clusters

Surgen en universidades, empresas, investigación por:

- procesadores de consumo potentes
- redes y protocolos de alto rendimiento
- herramientas que facilitan configuración y administración

Tipos de cluster

- dedicados / no dedicados
- homogéneos / heterogéneos

- propietarios / Beowulf

Aplicaciones comunes: Internet, bases de datos, cálculo científico.

Configuración

- red rápida (ej. InfiniBand)
- suele existir algún sistema de ficheros compartido:
    - nodo servidor de ficheros
    - sistema distribuido con RAID

Objetivo SO

- **Gestión de fallos:**
    - Alta disponibilidad: servicio continúa, pero si cae un nodo se pierden sus peticiones
    - Tolerancia a fallos: todo disponible siempre -> redundancia + replicación/migración
- **Equilibrado de carga:** el SO debe distribuir trabajo e incluir nodos nuevos
- **Computación paralela:**
    - Compilación paralela (decidir en ejecución qué va en paralelo)
    - Aplicaciones paralelas (programador las escribe usando paso de mensajes)

Gestión de procesos distribuidos y migración

Migración

Transferir suficiente estado de un proceso de un nodo a otro para que siga ejecutándose.

Por qúe migrar

- **Compartición de carga:** mover de nodos saturados a libres
- **Optimizar comunicaciones:** procesos que se hablan mucho -> mismo nodo
- **Disponibilidad:** mover procesos de "alta durabilidad" si se apaga un nodo
- **Uso de recursos especiales:** mover a nodos con HW específico

Preguntas clave de migración

¿Quién inicia la migración?

- S.O./monitor (si busca equilibrado)
- la propia aplicación (si busca un recurso específico)

¿Qué se migra?

- se destruye origen y se recrea destino
- mínimo: **PCB (bloque de control de proceso)**
- problema serio: **memoria + ficheros abiertos + mensajes pendientes**

Estrategias para migrar memoria

- **Ambicioso (completo):** copiar todo el espacio de memoria en el momento
  - no quedan restos
  - puede tardar mucho
- **Precopia:** copiar mientras el proceso sigue ejecutándose
  - reduce el tiempo parado
  - hay que recopiar lo modificado
- **Ambicioso (no completo):** enviar solo páginas recientes/en RAM y resto bajo demanda
  - minimiza datos transferidos
  - quedan restos en origen
- **Copiar al referenciar:** todo bajo demanda
  - coste inicial bajo
  - depende de accesos posteriores
- **Volcado (si hay Sistema de Ficheros compartido):** volcar páginas de memoria al disco y luego recuperarlas
  - libera origen
  - ayuda con memoria restante

Migración de Threads y Ficheros Abiertos

- Con hilos se complica mucho: no se puede mover toda la memoria si no se migran todos los hilos -> mejor “bajo demanda”
- Ficheros abiertos: duplicación, coherencia de caché, o compartir bloques para no saturar red

Negociación de migración

Sistema para decidir si "D acepta a P" (migración acordada entre nodos):

- cada máquina tiene proceso Starter que monitoriza carga
- migración se decide negociando entre starters y kernels

Exclusión mutua en distribuido

Varios nodos acceden a recursos compartidos -> se necesita exclusión mutua garantizando:

- no bloqueo indefinido

- si está libre, que pueda entrar sin retrasos
- no asumir velocidades relativas ni nº procesadores
- estancia finita en sección crítica

Además: en distribuido los mensajes pueden llegar desordenados, así que se necesita ordenación (ej. timestamp).

Solución: Cola Distribuida

Suposiciones:

- cada nodo tiene un proceso "árbitro"
- mensajes llegan en el orden enviado
- red completamente conectada

Funcionamiento (resumen):

- Proceso Pi encola su petición local y la anuncia a todos
- los demás la encolan y contestan
- cuando Pi está al principio -> entra a sección crítica
- al salir -> avisa para que lo quiten
- siguiente proceso entra

Virtualización

La virtualización permite ejecutar múltiples instancias de SO o aplicaciones simultáneamente, con cierta transparencia al SO.

Motivos

- baja utilización (10–15% por servidor)
- costes altos (infraestructura + mantenimiento + energía)
- más personal TI
- protección ante contingencias insuficiente
- mantenimiento de escritorios difícil y seguridad complicada

Ventajas

- menos inversión y mantenimiento
- más flexibilidad y aprovechamiento
- más seguridad y tolerancia a fallos

Hipervisor / VMM

Software de virtualización. Tipos:

- **Tipo 1 (bare metal):** sobre hardware (ej: Xen, ESXi, EC2)
- **Tipo 2 (hosted):** sobre un SO (ej: VMware Workstation)

Tipos de Virtualización

1. **Virtualización completa (full)**

- abstracción total, SO sin modificar
- VMM emula/captura instrucciones privilegiadas
- más sobrecarga -> peor rendimiento

2. **Virtualización asistida por HW**

- similar a la completa pero usa soporte Intel VT / AMD-V
- menos sobrecarga

3. **Paravirtualización**

- SO "sabe" que está virtualizado -> requiere modificaciones
- usa hiperllamadas
- mejor rendimiento, pero menos compatible

4. **Virtualización a nivel de SO (containers/jails)**

- múltiples instancias del mismo SO (containers)
- poco impacto en rendimiento
- poca flexibilidad
- uso típico: hosting, múltiples servicios, balanceo
- sistema de almacenamiento copy-on-write

Kubernetes y Docker

Kubernetes

Kubernetes (k8s) es una herramienta open-source diseñada para facilitar el **despliegue de contenidos y servicios**. Se basa en contenedores que **aíslan aplicaciones** reutilizando el **S.O. host**.

¿Qué ofrece?

1. **Descubrimiento de servicios y balanceo de carga**

- Las apps se anuncian con IP propia o DNS.
- Balanceo entre instancias dentro de la red.

2. **Gestión y configuración del almacenamiento**

- Montaje automático "bajo demanda".
- Soporta almacenamiento local/remoto/cloud.

3. **Rollback (vuelta atrás a estados anteriores)**

- Describir estado deseado, modificar y volver atrás si hace falta.

4. **Empaquetado automático de contenedores**

- Despliega contenedores en nodos según recursos.
- Se pueden fijar CPU y RAM por contenedor.

5. **Auto-reparado (self-healing)**

- Detecta fallos/caídas.
- Para y reinicia automáticamente contenedores problemáticos.

6. **Accesos seguros**

- Gestión de contraseñas y acceso seguro.

¿Qué no ofrece?

- No despliega código fuente ni compila apps.
- No es middleware de programación (no ofrece paso de mensajes, almacenamiento paralelo, etc).
- No recolecta logs, no monitorea, no alerta (por defecto).

Arquitectura: Cluster de Kubernetes

Un clúster tiene nodos de trabajo que ejecutan aplicaciones en contenedores. Tiene mínimo 1 nodo de trabajo.

Componentes principales:

- **Worker Nodes (nodos de trabajo):** donde viven los pods.
- **Control Plane (plano de control):** administra nodos y pods.

En producción, el control plane suele estar replicado -> alta disponibilidad.

Pods

Los **pods** son las **unidades de despliegue** manejadas por Kubernetes:

Grupo de **1 o más contenedores** que comparten red/recursos.

Los pods suelen ser **volátiles**: se crean bajo demanda y se destruyen cuando no se necesitan.

Deployment

Un **deployment** representa el despliegue de una aplicación:

Está formado por **uno o varios pods**, ejecutándose en una o varias máquinas. Además sirve como punto de entrada e interacción con la app.

- Un deployment permite **réplicas** (varios pods iguales).
- Si hay réplicas, intenta **balancear peticiones** entre pods/nodos.

Red en Kubernetes: CNI (Cluster Network Interface)

Kubernetes no gestiona directamente la red virtual: provee una interfaz CNI para que otros proveedores la implementen.

Sin red (CNI), no hay comunicación normalizada entre pods/nodos.

Docker

Docker permite crear contenedores con todas las librerías necesarias para ejecutar una aplicación de forma aislada, facilitando migración y despliegue sin reconfigurar todo el entorno.

Contenedor

Unidad básica de despliegue: aplicaciones empaquetadas listas para ejecutarse, con librerías y configuración incluidas, movibles entre sistemas.

Tema 3 (AWS)

EC2 (Elastic Compute Cloud)

EC2 es el servicio de AWS para crear **máquinas virtuales** (instancias) en la nube

S3 (Simple Storage Service)

S3 es un servicio de **almacenamiento de objetos** (no es un disco tradicional).

Guarda archivos como "objetos" dentro de un **bucket**.

RDS (Relational Database Service)

RDS es un servicio para crear y administrar una **base de datos relacional** en AWS

AWS Lambda

Lambda es el servicio **serverless** de AWS para ejecutar funciones bajo demanda **sin gestionar servidores**.

Grupos de seguridad (Security Groups)

Un **Security Group** es un "firewall virtual" que controla el tráfico de red de entrada y salida.