Introducción a los Sistemas Distribuidos

Motivación

Crecimiento de:

- Nivel de integración e interdependencia entre sistemas,
- Volúmenes de datos a procesar,
- Capacidades de cómputo,
- Paralelismo,
- Virtualización.

Definición

Sistema en el que el fallo de un computador que ni siquiera sabes que existe puede dejar tu propio computador inutilizable.

- Colección de computadoras -> **multiprogramación**
- Independientes -> **autónomos**
- Un sólo sistema -> **distribución transparente al usuario**
- Interconectadas por red -> **sistemas aislados NO SON distribuidos**
- Comunican y coordinan acciones -> **colaborativos**
- Intercambiando mensajes -> **protocolos de comunicación**
- Fallo de un computador -> **problemas no determinísticos**

Parámetros de diseño

- Escalabilidad
- Transparencia
- Tolerancia a Fallos *(Availability, Reliability, Safety, Maintainability)*
- Acceso a Recursos Compartidos
- Sistemas distribuidos abiertos *(Interfaces, Interoperability, Portability)*

Modelos de análisis

- Modelo de **Estados** *(interleaved model)*
- Modelo de **Eventos** *(happened before)*

Características

Centralizados

- **Control.**
- **Homogeneidad.** Estándares p/ software y hardware.
- **Consistencia.** Políticas fuertes.
- **Seguridad.** Menos superficie de ataque.

Distribuidos

- **Disponibilidad.** Tolerancia a Fallos.
- **Escalabilidad.**
- **Reducción de Latencia.** Localidad de recursos.
- **Colaboración.** Interacciones entre sistemas.
- **Movilidad.** No están atados al alcance de una única PC.
- **Costo.** Simplicidad. Delegación.

Descentralizar vs. Distribuir

- *Centralizar* = concentración de autoridad (nivel jerárquico + alto).
- *Descentralizar* = transferir toma de decisiones a eslabones inferiores.
- **Distribuir** = modelo descentralizado de control de computadoras para coordinar actividades con coherencia.

Ley de Conway

"Cualquier organización que diseñe un sistema, inevitablemente producirá un diseño cuya estructura será una **copia de la estructura de comunicación de la organización**."

- Diseñamos s/ lo que conocemos y estamos acostumbrados a hacer.
- Pueden ser soluciones eficientes.

Virtualización

- Necesidad de **independencia real** de recursos.
- Seguridad en los **accesos**.
- Concepto de **Hypervisor**:
  - Manager de VMs.
  - Emula *hardware capabilities*.
  - Administra recursos del `Host OS` -> `Guest OS`.
  - Mecanismos de Seguridad.

Docker

- Soporte de OS:
  - **Namespaces.** Aislamiento de recursos.
  - **Cgroups.** Seguridad.
  - **Union Mount.**
- Engine:
  - **Daemon.** Manejo de recursos.
  - **REST API.** Flexible, simple.
  - **CLI.** REST API a través de socket.

Multithreading y Multiprocessing

Multithreading

- **Recursos compartidos:**
  - Heap,
  - Data Segment,
  - Code Segment (RO),
  - FDs.
- **Sincronización:**
  - Threading SO,
  - Threading runtime,
  - IPC.
- **Características:**
  - Fácil compartir información.
  - Alto **acoplamiento**.
  - **Baja estabilidad**. Thread que falla afecta a todos.
  - Escalabilidad muy limitada.

Multiprocessing

- **Recursos compartidos:** Code Segment (RO).
- **Sincronización:** IPCs.
- **Características:**
  - Complejo compartir información.
  - Componentes **simples** y **separados**.
  - +Escalable y +Estable.
  - Sin tolerancia a fallos.

Propiedades de Sistemas Distribuidos

- **Safety** *(siempre verdadera, "nada malo va a pasar")*:
  - Exclusión mutua.
  - Ausencia de deadlocks.
- **Liveness** *(eventualmente verdadera, "algo bueno va a pasar")*
  - Ausencia de starvation.
  - Fairness.

Asegurar safety: Concurrencia

- Basada en **Algoritmos**
  - Características:
    - Sin abstracciones especiales.
    - Condiciones lógicas simples p/ **Critical Sections**.
  - Técnicas:
    - **Busy-Waiting.** Problemas de performance (ej. spin-lock).
    - **Algoritmos de espera.**
- Basada en **Abstracciones**
  - Características:
    - Provistas por SOs.
    - Construir mecanismos compuestos por combinaciones.
  - Teécnicas:
    - **Operaciones atómicas.** Contadores atómicos, CAS *(Compare and Swap)*.

Mecanismos de Sincronización

- **Semáforos.**
- **Monitores.** Abstracción.
  - **Condition Variables.**
- **Barrera.**
  - Rendezvouz.

IPCs

Características

- Provistos por SO.
- **ABM excede vida del proceso.**
  - Responsabilidad del usuario.
- Identificados por nombre.
- Linux: diferentes tipos de archivo.

Modelos

- Signals.
- Shared Memory.
- File Locks.
- Pipes: unnamed pipes.
  - Jerarquía padre-hijo.
- Fifos: named pipes.
  - Dos procesos cualquiera.
  - Viven en el SO.
- Message Queues.
  - `mtype` identifica el tipo de mensaje.
- Sockets.

Problemas clásicos

- **Productor Consumidor.**
  - Situaciones de bloqueo:
    - Producir paquete con buffer lleno.
    - Consumir paquete con buffer vacío.
  - Acceso al buffer **debe ser sincronizado**.
  - Buffer acotado vs. infinito.
- **Lectores Escritores.**

Paralelización de tareas

- **Objetivos:**
  - Reducir **latencia** *(tiempo de cómputo de una tarea)*.
  - Incrementar **throughput**.
  - Reducir **potencia consumida**.
- **Camino crítico.** Máxima longitud de tareas secuenciales a computar.
- **Ley de Amdahl.** Todo computo se divide en fracciones secuenciales y paralelas.
  - `Tp = Wser + Wpar / P` con `P` unidades de cómputo.
  - Speedup máximo **acotado por fracción de tiempo no paralelizable** (`Smax <= 1/f`).
- **Ley de Gustafson.** Escalar el problema.
  - Parte serial disminuye -> +speedup.
  - Paralelismo aumenta -> +speedup.

Modelo Work-Span

- Provee cota inferior y superior para el speedup.
- Hipótesis:
  - **Paralelismo imperfecto.** No todo lo paralelizable se puede ejecutar al mismo tiempo.
  - **Greedy scheduling.** Proceso disponible == tarea ejecutada.
  - Despreciable:
    - Tiempo de **acceso a memoria**.
    - Tiempo de **comunicación entre procesos**.
- Resultados:
  - `T1`: tiempo en ejecutar operación con 1 proceso.
  - `Tinf`: tiempo en ejecutar su camino crítico (con infinitos procesos).
  - **Cota superior:** `min(P, T1 / Tinf)`
  - **Cota inferior:** `(T1 - Tinf) / P + Tinf`

Estrategias

- Descomposición Funcional.
- Particionamiento de Datos.

Patrones de procesamiento

- Fork-join.
- Pack.
- Split.
- Pipeline.
- Map.
- Reduction.

Multicomputing

Muchos procesadores:

- **Comparten** mediante BUS:
  - Network Interface Controller.
  - Main Memory.
  - Disk Controller.
  - GPU (Memory).
- **Taxonomía de Flynn:**
  - SISD: Single Instruction Single Data.
  - SIMD: Array processors.
  - MISD: No son usuales.
  - **MIMD.**

MIMD

Multiprocessors

- CPUs **comparten** memoria y/o clocks.
- **Simétrico vs. Asimétrico** (distintos niveles, conectados por *bridges*).
- **Memory Access:**
  - **Uniform** (UMA, non-NUMA): tiempo identico p/ todos.
  - **Non Uniform** (NUMA): c/ CPU controla un bloque de memoria y se transforma en su *'Home Agent'*.

Multicomputers

- No comparten nada.
- Fallos **independientes**.
- **No hay reloj central** de ejecución de instrucciones.
- **Requieren comunicación** por networking.
- Sincronización mediante mensajes ad-hoc.
- Características:
  - Problemas de comunicación por red *(ancho de banda, latencia, pérdida de mensajes)*.
  - **Comunicación** es compleja y central al diseño del sistema.
  - Alta escalabilidad.
  - Tolerantes a fallos.

Comunicaciones

TCP/IP

- **Application** (user-level, datos).
- **Transport** (punto a punto).
- **Internet** (transmisión).
- **Network Access** (transferencia física confiable).

Relación con Modelo OSI

TCP/IP	OSI
Application	Application, Presentation, Session
Transport	Transport
Internet	Network
Network Access	Data Link, Physical

Middlewares

Capa de software entre el SO y la capa de aplicación/usuario para proveer una vista única del sistema.

Definiciones

- Software de conectividad.
- Ofrece conjunto de servicios.
- Permite operar sobre plataformas heterogeneas.
- Módulo intermedio como conductor entre sistemas.
- Capa de software entre SO y aplicaciones de sistema.
- Permite que múltiples procesos interactúen de un lado a otro (de la red).

Objetivos

- **Transparencia** (respecto de acceso, ubicación, migración, replicación, concurrencia, fallos, persistencia).
- **Tolerancia a Fallos** (availability, reliability, safety, maintainability).
- **Acceso a recursos compartidos** (eficiente, transparente y controlado).
- **Sistemas distribuidos abiertos (Interfaces).**
  - Estándares claros sobre servicios ofrecidos.
  - Interoperabilidad y portabilidad.
- **Comunicación de grupos.**
  - Broadcasting y multicasting.
  - Facilita localización de elementos y coordinación de tareas.

Clasificación

- **Transactional Procedure.** Transaccionalidad respecto a datos.
  - Conectan muchas fuentes de datos.
  - Acceso transparente al grupo.
  - Políticas de retries y tolerancia a fallos.
- **Procedure Oriented.** Servidor de funciones que se pueden invocar.
  - Servicios stateless entre invocaciones (idempotencia).
- **Object Oriented.** Servidor de objetos distribuidos.
  - Marshalling p/ transmitir info.
- **Message Oriented.** Sistema de mensajería.
  - Modo **Information Bus**: tópicos.
  - Modo **Queue**: destinatario definido.
- Reflective Middlewares (config. dinámica).

Documentación

La **arquitectura** representa aquellas **decisiones de importancia** s/ el **costo de modificarlas**.

Características deseadas

Diseño y documentación:

- **Evolutivo:**
  - Rápida adaptación.
  - Tomar feedback.
  - Valor iterativo.
  - No buscar entender todo.
  - No demorar arquitectura.
- Necesario para **coordinación**, **coherencia** y **cohesión**.
  - Diseño preliminar.

Modelos de Documentación

Vistas 4 + 1

- **Vista Lógica:**
  - Estructura y funcionalidad del sistema -> Clases, Estados.
- **Vista de Procesos (o Dinámica):**
  - Descripción de escenarios concurrentes (Actividades).
  - Flujo de mensajes (Colaboración).
  - Flujo temporal de mensajes (Secuencia).
- **Vista de Desarrollo (o de Implementación):**
  - Artefactos que componen al sistema (Componentes, Paquetes).
- **Vista Física (o de Despliegue):**
  - Topología y Conexiones entre componentes físicos (Despliegue).
  - Arquitectura del sistema (Robustez).
- **Escenarios:** Casos de Uso.

C4 Model

1. Contexto.
2. Container.
3. Componente.
4. Código.

Arquitectura de Capas

- Problema **-/> sub-problemas**.
- Fomentan uso **interfaces**.
- Intercambiar componentes **reutilizando conectores y protocolos**.

Layers (capas lógicas)

Agrupación lógica de componentes y funcionalidades de un sistema.

- Verticales y horizontales.
- Suelen hacer `downcalls` (con response).
  - Las `upcalls` son excepcionales.
- **Layer:** módulo con responsabilidades limitadas, coherencia y cohesión.
- C/ capa provee servicios a la capa superior y consume de la inferior.

Tiers (capas físicas)

Describen la distribución física de componentes y funcionalidades de un sistema.

- 2-Tier deployment, 3-Tier.
- Despliegue de Layers dentro de cada tier.

Interfaces

- **Permiten comunicación** entre dos o más componentes/servicios/sistemas.
- Diferentes **contratos**.
- Se expone **una parte** del sistema.
- Esconden implementación.
  - Cambiarla sin modificar contrato.
  - Cambio contrato -> nueva versión.

Tipos de contrato

- **Inter-Aplicaciones.**
  - **API (Application Program Interface).** Interfaz que permite que dos aplicaciones hablen entre sí.
  - Punto de acceso para que una aplicacion que vive por si sola permita que otra aplicación pueda reutilizar la funcionalidad que expone.
  - Eco-sistema entre aplicaciones.
- **Intra-Aplicaciones.**
  - Patrón(es) de diseño (Facades, Mediators, Interfaces).
  - Layers hablandose entre sí.
  - Mensajes entre objetos.

Problemas a resolver

Software es:

- **Dificil de integrar:**
  - Complejidad aumenta exponencialmente con la cantidad de elementos expuestos.
  - Interfaces pequeñas, esconder cosas.
  - Empezar exponiendo poco.
  - Extender con nuevas versiones si es necesario.
- **Dificil de cambiar:**
  - Fina linea entre interfaz flexible vs. cerrada y que no se adapta a los cambios.

  - Ojo con complejizar una interfaz de más.

Modelado de contratos p/ APIs

- Orientados a **Entidades:**
  - Desacoplamiento entre sistemas.
  - Objetivo = flexibilidad.
  - Admite extensiones.
- Orientados a **Procesos:**
  - Alto acoplamiento.
  - Alta performance como objetivo.

Clasificación

- **Web APIs.**
  - REST based (HTTP + JSON).
  - Web Services based (HTTP + SOAP).
- **Remote APIs.** Object-Procedure oriented.
- **Library-based / Frameworks.** Java-Android API.
- **OS related.** POSIX, WinAPI.

Protocolos

Modelo HTTP

- Client-Server.
- Request-Reply.
- Sin estado.

Protocol Data Unit (PDUs)

Encapsulación de PDUs entre capas.

1. **Encapsulación** exacta.
2. **Segmentación** de paquetes.
3. **Blocking** de paquetes.

RESTful

- **Entidades**.
- Web resource **representado por URL**.
- **HTTP/S** protocolo de comunicación.
- **JSON/XML** protocolo de serialización.
- Operaciones **CRUD** p/ cambio de estado.
- **Principios de Arquitectura:**
  - Cliente-Servidor.
  - Cacheability.
  - Interface Uniforme (HATEOAS).
  - Stateless.
  - Layered.
- **Identidad.** Unívocamente entre sistemas.
- **Relaciones.** Integración con otros sistemas.

Versionado de API

- **Semantic Versioning (semver).**
- **Incremento:**
  - `+Major` = cambios incompatibles.
  - `+Minor` = agregar funcionalidad manteniendo retrocompatibilidad.
  - `+Patch` (*build*) = correcciones sin afectar interfaz.
- **Tipos:**
  - Explícito en URL.
  - HTTP Custom Header. Incorrecto.
  - HTTP Accept Header. Correcto.
- **!= versionado de objetos!**
  - **Format Versioning.** API puede brindar distintas representaciones de **la misma entidad**.
  - **Historical Versioning.**

Nombres y direccionamiento

Nombres

- **Identifican** unívocamente a una entidad.
- **Describen** a la entidad.
- **Abstraen** al recurso de propiedades que lo atan al sistema.

Direccionamiento (Addressing)

- **Mapeo** entre nombre y dirección.
- Dirección cambia, nombre NO.
- Dirección puede ser reutilizada.
- Ejemplos:
  - `IP -> Ethernet Address`. ARP (IPv4) y ND (IPv6).
  - `Domain Name -> IP`. DNS.
  - `Service -> Instances`. Service Discovery.

Mensajes

Formateo de paquetes

Binario

- **Alta performance:** tamaño eficiente, compresión *innecesaria*.
- **Serialización:** autogeneración, no siempre existe soporte.
- **Interacción:**
  - Acoplamiento.
  - Cliente específico p/ c/ app.
  - Decoder p/ interpretar.

Texto plano

- **Baja performance:** bajo throughput, compresión -> overhead.
- **Serialización:** básica, formatos human-readable.
- **Interacción:** cliente único, fácil de debuggear.

Longitud de paquetes

- Pueden ser:
  - **Bloques fijos.**
    - Fácil de serializar.
    - Subóptimo p/ datos de long. variable.
  - **Bloques dinámicos.** Agrega:
    - Separador p/ delimitar comienzo y terminación.
    - Longitud del tipo p/ delimitar campos.
    - Overhead.
  - **Esquema mixto** (fijos sin delimitadores, variables con).
- Formato: `Type-Length-Value`.

Grupos

- Abstracción p/ **colección de procesos**.
- **Dinámicos**.
- Procesos pueden suscribir y cancelar suscripción a grupos.

  - Primitivas.

Difusión de mensajes

- **Uno a uno:**
  - **Unicast.** Punto a punto.
  - **Anycast.** Uno sólo recibe el mensaje (ej. el más cercano).
- **Uno a muchos:**
  - **Multicast.** Los de un determinado grupo reciben el mensaje.
  - **Broadcast.** Todos.

Atomicidad

- Deben entregarse a todos o a ninguno.
- Necesidad de **ACKs**.
- Necesidad de **demorar delivery** de paquetes recibidos.
- **Reintentos** frente a caídas/no recepción.

Tiempo

Magnitud para medir **duración y separación de eventos**.

- Variable monotónica creciente.
- Discreta.
- No necesariamente vinculada con la hora de la vida real.

Usos

- Ordenar y sincronizar.
- Marcar ocurrencia de un suceso (**timestamps**).
- Contabilizar duración entre sucesos (**timespans**).

Relojes Físicos

- Locales vs. Globales.
- Pueden ser: cuarzo, atómicos, por GPS.
- Referencias globales. GMT, UTC, GPS time, TAI.

Drift

- **Descalibración** x cambios de temperatura, presión, humedad.
- **No son confiables** para comparación.
- **Sincronización periódica necesaria.**
  - Desvío respecto de relojes de referencia.
  - Aplicar corrección cambiando frecuencia.
  - **Nunca atrasar** un reloj.
  - Algoritmo de Cristian: `Tnew = Tserver + (T1 - T0) / 2`

Network Time Protocol (NTP)

Objetivos

- **Sincronización.** Incluso con delays en la red.
- **Alta disponibilidad.** Rutas y servidores redundantes.
- **Escalabilidad.**

Estructura de Servidores

Basada en *stratums* (capas?).

- E0: **Master clocks**.
- E1: Servidores conectados directamente a master clocks.
- E2: Servidores sincronizados con E_1.
- EN: Servidores sincronizados con E_N-1

Modelos de Sincronización

- **Multicast/Broadcast.**
  - LANs de alta velocidad.
  - Eficiente, baja precisión.
- **Cliente-Servidor (RPC).** Grupos de aplicaciones.
- **Simétrico (Peer Mode).**
  - Sincronizados entre sí, backup mutuo.
  - Estratos 1 y 2.

Relojes Lógicos

Conceptos previos

- **Evento:** suceso relativo al proceso `Pi` que modifica su estado.
- **Estado:** valores de todas las variables del proceso `Pi` en un momento dado.
- **Ocurre antes.** Relación de causalidad entre eventos o estados tales que:
  - `a -> b`, si `a,b` pertenecen al mismo proceso `Pi` y `a` ocurre antes de `b`.
  - `a -> b`, si `a` es el envío (en `Pi`) de un mensaje a `Pj`, y `b` es la recepción (en `Pj`).
  - **Transitividad:** `a -> c`, si `a -> b` y `b -> c`.

Definición

Función `C` monotónica creciente que **mapea estados con un número natural**, y garantiza `s -> t => C(s) < C(t)` *(para todos los estados locales posibles del sistema).*

Algoritmo de Lamport

- Conjunto de N procesos, c/u inicia con `reloj = 0`.
- Evento interno -> `reloj += 1`.
- Evento de envío `Pi -> Pj`:
  1. (Pi) `reloj += 1`
  2. (Pi) Envía mensaje a Pj incluyendo el valor actualizado.
  3. (Pj) Recibe mensaje y obtiene `reloj` de Pi.
  4. (Pj) `reloj = Max(reloj_Pi, reloj_anterior) + 1`.

Problema: No cumplen la recíproca (`C(s) < C(t) =/> s -> t`).

Vectores de Relojes

Mapeo de todo estado del sistema compuesto por `k` procesos, con un vector de `k` números naturalez, y garantiza `s -> t sii s.v < t.v` *(con `A.v` vector de relojes de `A`).*

Obs. `s.v < t.v` sii:

- Cada valor del vector `s.v` es `<=` a los de `t.v`.
- Al menos hay una relación de `<` estricta.

Implementación

- Cada proceso incrementa su reloj.
- Cuando recibimos un mensaje:
  1. Agarra ambos vectores y, posición por posición, se queda con el máximo.
  2. Incrementa en 1 su propio contador.

Sincronismo

Algoritmo / protocolo es sincrónico si sus **acciones pueden ser delimitadas en el tiempo**.

- **Sincrónico.** Entrega de msj posee timeout conocido.
- **Parcialmente sincrónico.** No posee timeout conocido o es variable.
- **Asincrónico.** No posee timeout asociado.

Propiedades

- **Tiempo de Delivery:** tiempo que tarda mensaje en ser recibido luego de haber sido enviado.
- **Timeout de Delivery:** todo mensaje enviado va a ser recibido antes de un tiempo conocido.
- **Steadiness:** máxima diferencia entre el mínimo y máximo tiempo de delivery de cualquier mensaje recibido por un proceso.
  - Define varianza con la cual un proceso observa que recibe los msjs.
  - Qué tan constante es la recepción de mensajes.
- **Tightness:** máxima diferencia entre los tiempos de delivery para cualquier mensaje.
  - Define simultaneidad con la cual un mensaje es definido por múltiples procesos.

Protocolos

- Time-driven.
- Clock-driven.

Orden

- **Delivery de Mensajes** != Envío.
- Delivery: **procesar** mensaje, provocando **cambios en el estado**.
- Cola de mensajes, permite:
  - **Hold-back queue** y **delivery queue**.
  - Demorar el **delivery**.
  - Re-ordenar mensajes en la cola.

Órdenes

- **FIFO.** Órden en el que fueron enviados entre un mismo emisor y un mismo receptor.
- **Causal.** Todo mensaje que implique la generación de un nuevo mensaje es entregado **manteniendo esta secuencia de causalidad**, sin importar receptor.
- **Total.** Todo par de mensajes entregados a los mismos receptores es recibido en el mismo órden por esos receptores.

Estado y consistencia

- **Estado Local.** Valores de variables en un momento dado.
- **Estado Global.** Unión de todos los estados locales del sistema.

Máquina de estados

- Modelar el sistema como **serie de estados**.
- Evolución de estados debido a **eventos**.
- Con múltiples procesos hay **estados globales inválidos**.

Historia y Corte

- **Historia (corrida).** Secuencia de todos los eventos procesados por un proceso `Pi`.
- **Corte.** Unión del subconjunto de historias de todos los procesos del sistema hasta cierto evento `k` de cada proceso.
  - **Consistente** si por cada evento que contiene también contiene a aquellos que `ocurren antes`.

Algoritmo de Chandy & Lamport

- **Snapshots de estados globales** en sistemas distribuidos.
- Objetivo: estado global almacenado es **consistente**.

Comunicación reliable

Si se garantiza **integridad, validez y atomicidad** en el **delivery** de los mensajes.

- **Uno a uno.** Trivial sobre TCP/IP y red segura.
- **Uno a muchos.**
  - El grupo debe proveer las 3 propiedades.
  - Bajo TCP/IP, ojo con atomicidad.
  - Definir qué órden de mensajes se garantiza.

Arquitecturas Distribuidas Simples

Cliente Servidor

- Roles:
  - **Servidor:** pasivo, provee servicios.
  - **Cliente:** activo, envía pedidos.
- **Centralización** en toma de decisiones.
- Servidor tiene ubicación conocida.
- Modelos de *callback* posibles: long polling, push notifications.

Peer to Peer

- **Red de nodos pares** entre sí.
- Objetivos de **colaboración**.
  - **Protocolo acordado** entre partes.
  - Lógica distribuida requiere **coherencia entre nodos**.
- Tracker (esquema mixto C-S) como servicio de nombres.

RPC

- **Ejecución remota** de procedimientos.
- Modelo C-S: servidor ejecuta el procedimiento y devuelve resultado.
- Comunicación transparente.
- **Portabilidad** mediante **interfaces**.

IDL

- Diferentes lenguajes se invocan entre sí.
- **Interfaz s/ input y output.**
- **Definición de tipos de mensajes a enviar** como parte de IDL.

Tolerancia a Fallos

- Puede o no ser ejecutado.
- **Garantizar delivery** con estrategias:
  - **REQ-REP** con timeout.
  - Filtrado de duplicados.
  - **Re-transmisión / Re-ejecución** de operación.

Implementación

`Cliente <-> Stub <-> Communications Module (Client-Side) <-> Communications Module (Server-Side) <-> Stub <-> Server`

- **Cliente.** Conectado a stub p/ realizar llamadas al servidor.
- **Servidor.** Conectado a stub p/ recibir parámetros.
  - Posee lógica particular del remote procedure.
- **Stubs.**
  - Administra el marshalling de la información.
  - Envía info de las **calls** al módulo de comunicación y al C-S.
- **Módulo de comunicación.** Abstrae al stub de la comunicación con el server.

Distributed Objects

- Servidores proveen **objetos**.

- **Middleware** p/ ocultar complejidad.
  - Referencias a objetos.
  - Invocaciones de acciones.
  - Errores.
  - Recolección de basura.

CORBA

- Estandar definido.
- Protocolo y serialización.
- Transporte.
- Seguridad.
- Discovery de Objetos.

RMI

- Lo mismo, optimizado en Java.
- **Registry:** directorio de servicios.

MOM

- **Comunicación de grupo** de forma transparente.
- Comunicar mensajes entre apps.
- **Transparencia** respecto de: ubicación, fallos, performance y escalabilidad.

Variantes

- **Centralizado** (Broker) vs. **Distribuido** (Brokerless).
- **BUS** vs. Message **Queues**.
- **Sincrónico:** modelado como conexión punto a punto.
  - No hay transparencia frente a errores.
- **Asincrónico:** modelado con colas.
  - Soporta períodos de discontinuidad del transporte.
  - Complejo recibir respuestas.

Operaciones comunes

- `put`: publicar mensaje.
- `get`: esperar por un mensaje, sacarlo de la cola y retornarlo.
- `poll`: revisar mensajes sin bloquear.
- `notify`: asociar un callback para ser ejecutado por el MOM frente a ciertos msjs.

Brokers

- Proveen **transparencia de localización**.
- **Filtering, Routing.**
- Punto de control y monitoreo.

Ejemplos

ZeroMQ

- **Patrones:** Req-Rep, Pub-Sub, Pipeline (PUSH-PULL), Router-Dealer.
- **Conexiones:** TCP, IPC, Inproc (multithreading).

RabbitMQ

- **Queues.**
  - Nombradas vs. TaskQueues vs. **Anónimas**.
  - ACK.
  - Durabilidad opcional.
- **Exchanges.**
  - Estrategias para transmitir mensajes (`fanout`, `direct`, `topic`, `headers`).
- **Patrones:** Pub-Sub, Routing, **Topics**.

Patrones de Comunicación

Request-Reply

- **Sincrónico** (bloqueante) por defecto.
- ACK trivial (el mismo reply).
- Operación **asincrónica**: dos REQ-REP para mandar solicitud y esperar rta.
- **Estructura:** `messageID | requestID | operationID | argumentos`.
- **Tolerancia a fallos:** timeout con retries.

Publisher-Subscriber

- Comunicación por **eventos**.
- Productores y consumidores.
- Arquitectura basada en:
  - **Tópicos**: indicando tipo de evento (tópico) -> **BUS**.
  - **Canales**: orientadas a canales específicos -> **colas**.

Pipeline

- `Source - Filter(s) - Sink`.
- **Flujo de datos** procesados **secuencialmente** por filtros.

Modelo de Procesamiento

- **Worker por Filter.** Una unidad de procesamiento a cada etapa del pipeline.
- **Worker por Item.** Una unidad de procesamiento a cada item.
  - Un worker acompaña a un dato paso a paso h/ el final del pipeline.

Tipos de etapas (filtros)

- **Paralela:** cada item es **independiente** de los anteriores y posteriores, **admite paralelismo**.
- **Secuencial:** no puede procesar más de uno a la vez.
  - Los puede retornar **ordenados o desordenados**.

Ventajas

- **Algoritmos online.** Iniciar procesamiento antes de que estén todos los datos.
- **Información infinita.**

Direct Acyclic Graphs (DAG)

- **Instrucciones** modeladas mediante un **grafo de flujo de datos**.
  - Nodos = tareas.
  - Aristas = flujo de información.
- **Acíclicos.**
- Calcular **camino crítico**.
- Admite **lazy loading** (nodos requeridos por dependencias).
- Modelar dependencias entre procesos:
  - Dependencia implica posibilidad de bloqueo.
  - Cíclico implica posibilidad de deadlock.

Coordinación de Actividades

- **Coordinación.**
  1. División/Despacho/Difusión.
  2. Ejecución.
  3. Consolidación.
- **Replicación.**
  1. Difusión.
  2. Ejecución (todos lo mismo).
  3. Consolidación.
- **Acceso a Recursos Compartidos.**
  1. Serialización de requests.
  2. Ejecución serial.

Open MPI

- Transmisión y recepción de mensajes.
- **Ejecución transparente** de 1 a N nodos.
- Librería con **abstracciones** de uso general.
  - Foco en **cómputo distribuido**.
- Implementa middleware de comunicación de grupos.
  - `MPI_Send`, `MPI_Recv`, `MPI_Bcast`.
  - `MPI_Scatter`, `MPI_Gather`.
  - `MPI_Allgather`, `MPI_Reduce`.

Apache Flink

- Plataforma p/ **procesamiento distribuido** de datos.
- Motor de ejecución de **pipelines de transformación**.
- Framework p/ crear pipelines.
- **Ventanas** para streaming.
- Casos de uso: ETL, Data Pipelines.

Dataflow

DAG de operaciones sobre un flujo de datos.

- **Streams:** flujo de información que eventualmente no finaliza.
- **Batchs:** dataset de tamaño conocido.

Bloques de pipeline

- **Source:** inyecta datos.
- **Transformation (operador):** modifica o filtra datos.
- **Sink:** almacenamiento final

Apache Beam

Modelo de **definición de pipelines** de procesamiento de datos con **portabilidad de lenguajes** y **motores de ejecucución**.

- **Runners:**
  - Ejecución directa.
  - Motores de cluster.
  - Plataformas cloud.

Bloques de un pipeline

- **Input** y **Output** *(source y sink)*.
- **PCollection:** colecciones paralelizables de elementos *(streams)*.
- **Transformations:** modificaciones elemento a elemento *(operators)*.

Map Reduce

Introducción

- **Parallel Computing.** Partir procesamiento en partes que pueden ser ejecutadas concurrentemente en múltiples cores.
- **Idea:**
  - Identificar **tareas** que pueden ejecutarse de forma concurrente,
  - Identificar **grupos de datos** que puedan ser procesados de forma concurrente.

Caso ideal (Master-Worker)

- **No existe dependencia** entre los datos.
- Datos partidos en **chunks del mismo tamaño**.
- C/ proceso puede trabajar con un **chunk/shard**.
- **Master.**
  - Parte la data en chunks,
  - Envía ubicación de los chunks a los workers,
  - Recibe ubicación de los resultados de todos los workers.
- **Workers.**
  - Recibe ubicación de los chunks,
  - Procesa el chunk,
  - Envia ubicación del resultado al Master.

Función Map

`Map: (input shard) -> intermediate(k:v pairs)`

- Función proporcionada por el usuario.
- Ejecutada en todos los chunks de data por `M` *map workers*.
- Usuario decide **criterio de filtrado**.
- Agrupa todos los valores asociados con una **misma key**.

Función Reduce

`Reduce: intermediate(k:v pairs) -> result files`

- Realiza una **agregación** de los datos.
- Llamada p/ c/ **Unique Key**.
- Función distribuida, particionando las keys en `R` *reduce workers*.

Algoritmo

1. **Partir datos** de entrada en **N chunks**.
2. **Fork de Procesos** en Cluster.
   - 1 master.
   - `M` mappers (tantos como chunks).
   - `R` reducers (configurado por el usuario)
3. **Map** de Shards en Mappers.
4. Creación de **Intermediate Files** (IFs).
   - Data particionada en `R` regiones.
   - **Función de Partición**.
     - Decide cuál de los workers va a trabajar con cada key.
     - Default: `hash(key) mod R`
     - Map workers `particionan data por Keys` con esta función.
5. **Reducers leen data** (por RPC) de los IF a procesar.

- Ordenan data por key.
- Agrupan ocurrencias de la misma key.

6. **Reducer aplica función** sobre c/ set de datos (agrupado por key).
7. Almacenamiento de resultados en **Output files**.

Data Intensive Applications

Datos en Sistemas de Gran Escala

- **NoSQL mejor** que SQL. **Flexibilidad** de esquemas.
- Enfoques transaccionales:
  - **Online Transaction Processing (OLTP).** Orientado a unidades lógicas, transacciones.
  - **Online Analytics Processing (OLAP).** Orientado a análisis del conjunto de datos.
- **Almacenamiento:**
  - Relacional.
  - Columnar.
  - Cubos de Información.
    - Vistas con pre-cálculos estadísticos.
    - Agrupaciones por diferentes dimensiones.

Replicación

- **Leader based:**
  - **1 lider RW.**
  - **N followers RO.**
  - Replicación (**mirroring**) síncrona vs. asíncrona.
  - Problemas:
    - *read your own writes*,
    - *monotonic reads*.
- **Multi-leader based:**
  - **+Tolerancia a fallos.**
  - Problema agregado: *conflictos por occurencia*.
- **Leaderless based:**
  - Totalmente **distribuido**.
  - Sincronización mutua.
  - Topologías.
  - **Conflictos muy frecuentes** si no hay particionado.
  - Alternativa: **consenso** para escrituras (paxos?).

Particionamiento

Motivaciones

- **Performance.** Aumentar velocidades de R/W.
- **Conflictos.** Evitar colisiones/resoluciones de conflictos.
- **Redundancia.** Tolerancia a fallos.

Tipos de particionado

- **Horizontal** (por filas). Registro en UNA partición a la vez.
- **Vertical** (por atributos/dimensiones/campos). Registro en TODAS las particiones a la vez.

Función de partición

- Por **Key-Value**.
- Por **Key-Range**.
- Por **Hash-of-Key**.
  - Escribir todo a la vez.
  - Buscar buena distribución de datos entre todas las particiones.
- Mixtos:
  - Generar **N shards** para cada key. Da igual *en cuál* guardo.

  - Partición por claves secundarias.

Enrutamiento

- Si **conozco la función de particionamiento**, ir directamente.
  - Ojo: implica conocer la dirección de cada partición, etc etc.
- Si **no la conozco**, algo extra del lado del usuario.
  - Opción 1: ir a cualquier partición, si es miss, **ser redireccionado**.
  - Opción 2: ir a un **centinela** que sabe donde esta la data.

Distributed Shared Memory (DSM)

- **Objetivo:** ilusión de **memoria compartida centralizada**.
- **Ventajas:**
  - Intuitivo (facilita distribuir algoritmos no distribuidos).
  - Compartir información entre nodos que no se conocen.
- **Desventajas:**
  - Desalienta la distribución,
  - Genera latencia,
  - Cuello único de botella (SPOF).

Implementación naive

- Información en **memoria** de un **servidor** (*memory pages*).
- Clientes acceden mediante **requests**.
- Consistencia = **serialización de requests**.
- Baja performance p/ clientes.

Migración de Memory Pages

- Información en memoria del servidor, **delegada en los clientes**.
- Optimizar **localidad de acceso** pidiendo memory page **prestada**.
  - Otro pedido de la misma página queda bloqueado.
  - Alternativa: permitir sub-delegación.
- Consistencia garantizada.

Replicación de Memory Pages (RO)

- Favorece escenario **muchas lecturas pocas escrituras**.
- **Leer** => replicación en modo RO.
- **Escrituras coordinadas** por servidor.
  - Ante cambios, invalida réplicas.

Replicación de Memory Pages (RW)

- Servidor tiene las páginas en memoria h/ ser pedidas.
- Cliente toma **control total** sobre la página replicada.
- Servidor => **secuenciador de operaciones**.
- Servidor aplica cambios ante caídas.

Distributed File Systems (DFS)

Motivaciones

- **Esquema centralizado de información persistente.**
  - Control de backups, acceso, y monitoreo.

- **Optimización de recursos** gracias a concentración.

Factores de diseño

- **Transparencia a los clientes.**
  - **Acceso** con credenciales.
  - **Localización.** Operar archivos como si fueran locales.
  - **Movilidad** interna no debe ser percibida.
  - **Performance y Escala.** Optimizaciones no afectan.
- **Concurrencia.** Sin operaciones adicionales.
- **Heterogeneidad de Hardware.** Adaptación.
- **Tolerancia a Fallos.** Permitir `at-least-once` o `at-most-once`.

Network File System (NFS)

- Objetivo: **independencia de plataformas**.
- Requiere abstracción del kernel: **VFS (Virtual File System)**.
- Arquitectura C-S utilizando RPC sobre TCP/UDP.
- Acceso a archivos mediante VFS.
  - Requiere invocación remota.
- Soporta POSIX.

Hadoop DFS (HDFS)

- Objetivo: **hardware de bajo costo**.
- Basada en GFS.
- No soporta POSIX => considerado **Data Storage**.

Factores de diseño

"Moving computation is cheaper than moving data." (a computation requested by an application is much more efficient if it is **executed near the data** it operates on).

- **Tolerancia a fallos.** Adaptarse a fallos de HW.
- **Volumen y Latencia.** Favorece *streaming* y archivos volumétricos.
- **Portabilidad.** Utiliza TCP entre servidores y RPC con clientes.
- **Performance.** Favorece operaciones de lectura.

Arquitectura

- **Master-Slave.**
- **Namenode:** contiene metadata de archivo, coordina los datanodes.
- **Datanodes:** almacena datos de archivo *(file blocks)*.
- Cliente consulta namenode por el FS y la ubicación.
  - Luego, comunicación directa.

Almacenamiento

- Bloques de **tamaño fijo**.
- **Replicados** en distintos datanodes.
- Metadata mantenida en memoria, con **log de transacciones**.
- Cluster permite **re-balanceo de bloques**.

Big Table

- Claves, datos, columnas.

  - Almacena Clave-Datos.
  - Datos = conjunto de columnas.
  - Conjunto **disperso**.
- **Tablets:** conjunto de filas consecutivas s/ clave.
  - Unidad de balanceo.
  - Permite escalar el sistema.
  - Auto-splitting.
- **Jerarquía de tres niveles:** root metadata.

Arquitectura

- **Master.**
  - Locks en Chubby.
  - Asignación de tablets.
- **Tablet Servers** (R/W directo con clientes).

Auto-splitting

- Exitoso: **randomly distributed keys**.
  - R/W se handlean por rangos, por ej.
- No exitoso: **monotically increasing keys**.
  - Writes siguen yendo al mismo!

Escalabilidad

Capacidad de un sistema p/ **adaptarse** a diferentes ambientes **modificando los recursos** del sistema.

Objetivo

Crecimiento respecto de:

- **Tamaño.** +Usuarios/recursos.
- **Distribución geográfica.** Dispersión.
- **Objetivos administrativos del sistema.**

Características de plataformas

- **Plataformas p/ alta concurrencia.**
  - Patrones conocidos.
  - Escalamiento automático.
  - Fuerte vínculo con infra/producto.
- **Arquitecturas ad-hoc y personalizadas.**
  - Necesidad de configuración y soporte.
  - Escalamiento manual / automatizado x humanos.
  - Posibilidad de migraciones.

Patrones de carga

- Predictable Burst.
- Unpredictable Burst.
- Periodic Processing (avg usage + períodos de inactividad).
- Start Small, Grow Fast.

Limitantes

- Arquitectura y algoritmos.
- Datos.
- Red (latencia, ancho de banda).
- Restricciones de negocio / locales.
- **Presupuesto.**

Técnicas

- Escalamiento **vertical**. +Recursos.
- Escalamiento **horizontal**.
  - Redundancia,
  - Balanceadores de carga,
  - Proximidad geográfica.
- **Fragmentación de datos.**
- **Optimizar algoritmos.** Performance, mensajería.
- **Asincronismo.** Limitado por negocio.
- Componentización -> separar servicios.

Elasticidad

Capacidad de un sistema p/ **modificar dinámicamente los recursos adaptándose a patrones** de carga.

Componentes

- **Application Load Balancer.** Ver a qué servicio/instancia le mandamos tráfico.
- **Autoscaler.**
  - **Scale In:** decrementar instancias.
  - **Scale Out:** incrementar instancias.
  - En función de **métricas**.
- **Monitoring Automático.** Métricas sobre CPU, memoria, I/O, networking, etc. p/ c/ servicio/instancia.
- Ejemplos:
  - **AWS:** Amazon ELB, Amazon Autoscaling, Amazon CloudWatch.
  - **K8s:** K8s Service, Horizontal Pod Autoscale, K8s Metrics Server.

Alta Disponibilidad

- **SLA (Agreement):** disponibilidad pactada con cliente.
  - Define qué sucede si no se cumple.
- **SLO (Objectives):** lo que se debe cumplir para no invalidar SLA.
  - Ej: availability $> 99.95\%$
- **SLI (Indicators):** métricas a comparar con SLOs.
  - Plataforma de **observability**.
  - Analizar **impacto del despliegue**.

CAP

- **Consistency.** Repetibilidad de respuesta de todos los nodos frente a mismo pedido.
- **Availability.** Responder a todo.
- **Partition Tolerance.** Lidiar con formación de grupos aislados de nodos.

Arquitecturas orientadas a Servicio

Monolíticas

`-Request-> ReverseProxy (<-> Static Files) -> WebServer -Query-> DBServer`

Escalables

- Web Requests: **+Web Servers**.
  - (+) Routeo y Escalabilidad
  - (-) SPOF.
- Data Queries: **+DB Servers**.
  - (+) Throughput lectura.
  - (-) Throughput escritura.

Service Oriented (SOA)

- **Paradigma** orientado al **ámbito corporativo**.
- **Business Process Management (BPM).** Disciplina involucrando modelado, automatización, ejecución, control, métricas y optimización de los flujos de negocio p/ objetivos de empresa.

Características

- **Tecnologías:**
  - Web Services (SOAP + HTTP).
  - **Enterprise Service Bus** para **eventos**.
  - Service **Repository** & **Discovery**.
    - Comunicación punto a punto.
- **Servicios y Procesos:**
  - Interfaces.
  - Contratos.
  - Implementación: business logic + data management.

Microservicios

- +Granularidad.
- Escalabilidad Horizontal.
- Flexibilidad de Negocio.
- Monitoreo y Disponibilidad parcial.

Serverless

Cloud

- Todo lo que se puede consumir más allá del firewall.
- `Networking + Infra + Nuevas plataformas + Servicios`

Niveles de abstracción

- **IaaS.**
  - Almacenamiento y **virtualización de equipos**.
  - Definir redes y adaptación frente a carga.
  - **Customer Managed:** Apps, Security, Databases, OS.
  - *Ej. Google Cloud Storage.*
- **PaaS.**

  - Frameworks y plataformas p/ desarrollar aplicaciones *Cloud Ready*.
  - Recursos expuestos como **servicios p/ desarrollo** y **manejo de ciclo de vida** (logs,monitoreo).
  - **Customer Managed:** Apps.
  - *Ej. Google AppEngine.*
- **SaaS.**
  - Alquiler de servicios, **software a demanda**.
  - Soluciones **genéricas** y **adaptables**.
  - Arquitecturas pensadas p/ **integración**.
  - **Customer Managed:** NADA.
  - *Ej. Google Apps.*

Beneficios

- **Accesibilidad.** Movilidad y visibilidad constante.
- **Time-to-Market.** Recursos instantaneos.
- **Escalabilidad.** Capacidad *ilimitada* de recursos.
- **Costos.** Pay-as-you-go. Controles de gasto.

Resistencia al cambio

- Factores **políticos**:
  - Locación y jurisdicción de datos,
  - Incapacidad de influir sobre decisiones de HW.
- Factores **técnicos**:
  - Costos p/ migraciones,
  - Exposición de datos sensibles,

PaaS

Tener una plataforma para desarrollar. Viene con:

- **Infraestructura.** Ya está en la plataforma, todo como IaaS.
- **Plataforma de Desarrollo.** SOs, librerías especiales, middlewares.
- **Persistencia.** Bases de datos, archivos blobs, colas de mensajes.
- **Monitoreo.**
- **Escalabilidad.** Balanceo, elasticidad.

Google AppEngine

Plataforma basada en buenas prácticas (forzadas).

Buenas prácticas

- Sistemas **granulares**.
- Escalamiento **horizontal**.
- **Requests breves**, los largos son encolables.
- Independencia de SO / HW.

Servicios integrados

- Cache.
- Colas de mensajes.
- Elasticidad.
- Versionado.
- Herramientas de log, debugging, monitoreo.

- Modelos No-Relacionales (Datastore, BigTable).

Microservicios

- **Aplicaciones**:
  - **Servicios**, pueden o no hablar entre ellos.
  - C/ servicio tiene capa de:
    - Memcached.
    - Datastore.
    - Task Queues.

Componentes

- **Servicios:** módulos.
- **Instancias (AppServers):** servidores de backend.
  - Unidad de procesamiento.
  - **Dinámicas:** creadas x requests.
  - **Residentes:** escaladas manualmente.
  - Depende de:
    - Alguien que se encargue de bufferear cosas,
    - Que no sea importante el estado dentro de las instancias.
  - Evidentemente BUENO tener **stateless**.
  - Si mi req tarda mucho, tiran la instancia.

Arquitectura

- Primer Data Center.
  - Google **front end**.
    - 1st level validations.
  - **Edge Cache.**
- Segundo Data Center.
  - Tiene a la App en sí.
  - **AppEngine front end.**
    - CDN.
    - App Servers.
      - Instancias.

Comunicación interna

- **Push Queues:** cuando algo llega, la **cola es algo activo**.
  - Puede invocar cosas.
  - La cola creaba los handlers si no estaban.
  - Si habían pocos workers, la cola creaba más.
  - La cola hacía el balanceo.
  - El mensaje debe tener una URL (p/ atenderlo).
- **Pull Queues:** las de siempre.
  - Procesamientos largos.
  - Payload, Etiqueta.
- Se comparte **Datastore** y **Memcached**.
  - Todo el resto es **serverless**.
- Mensajes en modo **leasing** (ACK de Rabbit).

Almacenamiento

- Datastore.

- Basado en BigTable.
- Clave-Valores.
- Atomicidad? **Particionado**.
  - Guardar las cosas que component mi transaccion todas juntas.
  - Transacciones con poco delay.
  - Localidad espacial p/ acceso.

Diseño de Arquitecturas de Gran Escala

Procedimiento

1. **Endpoints.**
2. **Análisis de volumen.** Entender dónde está la carga del sistema.
   - Storages.
   - Networking.
   - Seguridad.
3. **Diseño.**
   - Documentación: DAG, Robustez, etc.
   - MOM.

Notas de clase

- **Scope** del sistema?
- Importante:
  - **Endpoints.**
  - **Boundaries.**
  - **Datos.**
- TL;DR:
  - Analizamos la información.
  - Extraemos datos.
  - Entender scope y boundaries.
  - Endpoints, assumptions y avanzar.

Tolerancia a Fallos

En presencia de fallos, el sistema distribuido continúa operando en **forma aceptable**.

- **Dependable Systems.**
- **Garantizar** comportamiento en distintas condiciones.
- Prevenir **de cara al usuario**.
- Nivel de tolerancia.
- **Fallo** (parcial) -> **Error** (en el estado del sistema) -> **Falla/Avería** (comportamiento incorrecto).

Clasificación de fallos

Según **frecuencia**:

- **Transientes.** Una vez y desaparecen. Repetir lo arregla.
- **Intermitentes.**
- **Permanentes.** H/ reemplazar componente defectuoso.

Según **tipo**:

- **Crash.**
- **Timing.**
- **Omisión.**
- **Respuesta.** Valor incorrecto.
- **Arbitraria o Bizantina.**
  - En tiempos y respuesta.
  - Distinta info para distintos consumidores.

Condiciones

- **Del entorno.**
  - Entorno físico del hardware.
  - Interferencia y ruido.
  - Drifts de relojes.
- **Operacionales.**
  - Especificaciones
  - Networking.
  - Protocolos.

Detección de errores

- **Fault Removal.** Removerlos antes de que pasen.
- **Fault Forecasting.** Probabilidad de que un componente falle.
- **Fault Prevention/Avoidance.** Evitar condiciones que llevan a generarlos.
- **Fault Tolerance.** Aceptar los errores y tratarlos en el sistema.

Frente a errores

- **Resiliencia:** mantener nivel aceptable en presencia de fallos y desafíos.
- **Degradación suave:** difiere del comportamiento normal pero sigue siendo aceptable.
- **Enmascarado de errores.**
  - Tolerar mediante **redundancia**.
    - Física = **replicación**.
    - De información = **valor**.
    - De tiempo = **retries**.
- **Replicación.** Evitar SPOF.
- **Recuperación** de un error y llevarlo a estado correcto.

  - Almacenamiento estable.
  - Checkpointing (periódico).
  - Message logging (repetir desde checkpoint).
  - Consenso.

Tipos de Replicación

- **Pasiva.** Una primaria y varias secundarias/backup.
- **Activa.** Múltiples máquinas hacen lo mismo. Orden total.
- **Semi-activa (Leader-Follower).** Un lider toma decisiones no determinísticas.

Confiabilidad

Dependability. Medida de confianza en el sistema.

- **Availability.**
- **Reliability.**
- **Maintainability.** Ciclo de despliegue, provee:
  - Inmutabilidad.
  - Resiliencia.
  - Desacoplamiento.
- **Safety.** El sistema debe poder ser recuperado automática o manualmente ante cualquier falla.

Coordinación y Acuerdo

Exclusión mutua distribuida

- Obtener **acceso exclusivo** a un **recurso disponible p/ la red**.
- Pasaje de mensajes.
- Requerido:
  - **Safety:** solo un proceso a la vez.
  - **Liveness:** evitar starvation, espera eterna de mensajes.
  - **Fairness:** c/ proceso misma prioridad. In-order processing.

Algoritmos

- **Servidor central.** Un coordinador de la sección crítica.
  - Se sabe identificar el recurso.
  - Requests encolados (FIFOs).
  - Acceso *time-bounded*.
- **Token Ring.**
  - El token circula por el anillo.
  - Acceso "por turnos".
- **Ricart & Agrawala.**
  - Cuando querés acceder:
    1. `Request` con **timestamp** del proceso, **ID** y **nombre** del recurso.
    2. Enviar a todos.
    3. Esperar `OK` de todos.
    4. Entrar.
  - Cuando recibis `Request`:
    1. Envía `OK` si no está interesado.
    2. Si tiene la seccion, no responde y lo encola.
    3. Si está esperando, se comparan timestamps. El del **timestamp menor gana**.
       * El perdedor envía `OK`.

    * El ganador encola request.
- Cuando terminas de usar la sección, mandás `OK` a todos los encolados.

Elección de Líder

- Cualquier proceso puede iniciar elección.
- Nunca más de un lider.
- Resultado de elección **único y repetible**.
- Estado: Identificador (`P`), indefinido (`@`).

Ring

- Inicio: marcarse como `participando`, mandar un msj con su ID.
- Al recibir mensaje de elección:
  - Si estado es `no participando`: se marca como `participando`, y manda el msj con el max ID entre el suyo y el que venía.
  - Si estado es `participando`: reenvía el mensaje si el ID es mayor al suyo.
    - Si el ID es el suyo, **se identifica como líder**.
- Al reconocerse como lider: se marca como `no participando` y envía msj de `lider elegido`.
- Al recibir msj de `líder elegido`: se marca como `no participando`, settea el líder y lo retransmite si el ID no es el suyo.

Bully

Hipótesis

- Canales **reliable**.
- Cada proceso **conoce el ID** asociado del resto de procesos.
- Uso de **timeouts** para detectar procesos muertos.
  - `T = 2 * T_max_transmisión + T_max_process`
- **Todos** se pueden **comunicar** entre sí.

Mensajes

- `Election`: iniciar elección.
- `Answer`: ACK.
- `Coordinator`: quién fue elegido.

Algoritmo

- Al detectar que el líder se cayó, se manda `Election` a procesos con ID mayor.
- Si un proceso recibe `Election`, responde con `Answer` y comienza una nueva elección.
- Si un proceso recibe `Coordiator`, setea a su emisor como líder.
- Si un proceso que comenzó no recibe `Answer` tras `T` tiempo, se autoproclama líder.

Consenso

Dado un **conjunto de procesos distribuidos** y un punto de **decisión**, todos deben **acordar** en el mismo valor.

Propiedades necesarias

- **Agreement.** El valor de la variable `decided` es el mismo en todos los procesos correctos.
- **Integrity.** Si procesos correctos propusieron el mismo valor, entonces tienen la misma `decisón variable`.
- **Termination.** Todos los procesos activos setean eventualmente su `decisión variable`.

Generales Bizantinos

- Un proceso líder y varios followers.
  - Líder emite un valor.
  - Followers lo reenvían al resto.
- Procesos *traicioneros*: pueden enviar valores incorrectos.
- `N >= 3f + 1`

Paxos

- Objetivo: **consensuar valor** aunque hayan **diferentes propuestas**.
- **Tolerante a fallos.** Progresa si hay mayoría de procesos vivos.
  - Quorum: `N >= 2f + 1`
- Posible rechazar propuestas.
- Asegura **órden consistente en un cluster**.
  - Eventos almacenados incrementalmente por ID.

Actores

- **Proposer.** Recibe request e inicia protocolo.
- **Acceptor.**
  - Mantiene estado del protocolo en **almacenamiento estable**.
  - Quorum si la mayoría están vivos.
- **Learner.** Cuando hay acuerdo, se ejecuta el request.

Algoritmo

0. Request del cliente.
1. Fase 1: primera propuesta.
   1. **Prepare.**
      - Proposer envía propuesta `#N`.
      - `N` mayor a cualquier propuesta previa.
   2. **Promise.**
      - Si ID es mayor al último recibido, se rechaza cualquier request con `ID < N`.
      - Envía `Promise` al proposer.
2. Fase 2: si recibí promise de la mayoría.
   1. **Propose.**
      - Rechaza requests con `ID < N`.
      - Envía `Propose` con el `N` recibido y un valor `v`.
   2. **Accept.**
      - Si la propuesta aun es válida, anuncia nuevo valor `v`.
      - Envia `accepts` a todos los learners y al proposer.
   3. El **learner responde** al cliente.

Sistemas de Tiempo Real

- Sistemas cuya evolución se específica en términos de **requerimientos temporales** requeridos por el entorno.
  - Lo importante es que se indica el paso a paso.
  - Cual es el tiempo definido para ejecutar cada paso.
- **Correctitud** del sistema = **respuestas correctas** en **tiempo correcto**.
- Ejemplos: electrodomésticos digitales, medidores de señales, mediciones por sensores, control de automóviles, control en aeronaves, marcapasos, etc.
- Sistema **previsible**, NO performante.
  - Sobre temporalidad.
  - Correcto **scheduling**.

Tipos

- **Hard RT:** se debe evitar todo fallo relacionado con el tiempo de delivery.
  - Perder un deadline es fallo total.
- **Soft RT:** pueden ser admitidos ocasionalmente / nivel de tolerancia.
  - Utilidad de resultado disminuye tras deadline.

Comunicación

- Requiere comunicación **fiable** y **sincrónica**.
  - **Deadlines** definidos.
  - TCP no cumple. No hay garantías de tiempo.
- Comunicación Serial: **Profibus**.
- Utilizar **Ethernet**: rediseñar protocolo de capas superiores.
  - **Profinet**.

Fault Tolerance

- **Previsibilidad.** Todo tiene que estar escrito y bien definido.
- En hard RT, **hard resets**.
  - Muy importante revisar **maintainability**: recuperarse de forma barata, rápida y consistente.

Paradigmas de Trabajo

- **Event-Triggered.**
  - El cliente lo espera de forma **bloqueante**.
  - El cliente debe poder controlar tiempos de inactividad.
- **Time-Triggered.**
  - Definición de **time slots**.
  - En c/ time slot se pueden emitir eventos.
  - Cuándo tengo respuesta?

Sistemas de Control

Escenarios donde un sistema intenta **controlar** de forma manual o automática alguna **realidad del medio físico**.

- **Compatibilidad** entre especificaciones.
- No todo sistema RT es de control.
- Ejemplos.
  - En la industria: procesos químicos, líneas de producción, etc.
  - En la vida: termostatos, ascensores, control de luminosidad, etc.

Nociones

- **Control.** Capacidad de actuar para mover cosas y buscar que algo pase.
- **Proceso.** Sucesión de cosas que quiero controlar.
- **Variable controlada.** Valor/cantidad que mido/controlo. **Salida del sistema.**
- **Variable manipulada.** Cantidad/condicion que modifico para afectar el valor de la controlada.
- **Perturbación.** Señal que afecta negativamente al valor de la salida del sistema.
- **Planta.** Sistema físico sobre el cual se trabaja.
- **Controlador (referencia).** Sistema encargado de determinar qué hay que hacer.
- **Actuador.** Sensores físicos.

Ciclos

- **Lazo Abierto.** Manual, no automatizado.
  - No hay feedback.
  - No hay retroalimentación.
  - No considera lo que pasa en la realidad.
- **Lazo Cerrado.** Realimentado, feedback.
  - Medir error entre lo que quería hacer y lo que obtuve.

Programación

- Arquitecturas dirigidas por eventos o por tiempo.
- Scheduling importante.
  - Non-preemptive, esquema de prioridades => garantizar deadlines.
- Protocolos de comunicación específicos.