



Sistemas Distribuidos 1

Apunte

1. Introducción

Definición

"Colección de computadoras independientes que el usuario ve como un solo sistema coherente" (Tanenbaum)

"Es un sistema de computadoras interconectadas por una red que se comunican y coordinan sus acciones intercambiando mensajes" (Coulouris)

"Aquel en el que el fallo de un computador que ni siquiera sabes que existe, puede dejar tu propio computador inutilizable" (Lamport)

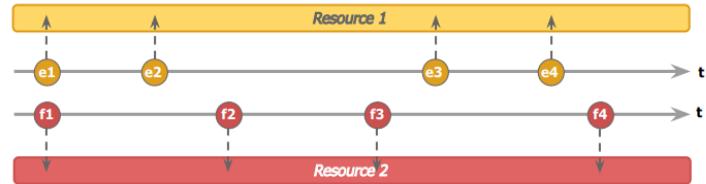
Parámetros de diseño

- Transparencia
- Acceso a recursos compartidos
- Sistemas distribuidos abiertos (interfaces, interoperabilidad, portabilidad)
- Escalabilidad
- Tolerancia a fallos

Paralelismo vs Concurrencia

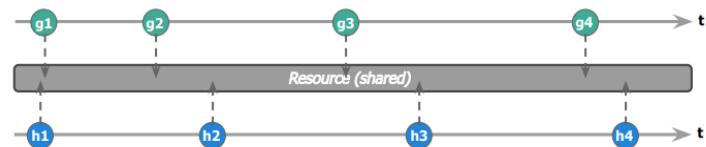
Paralelo

(ejecución simultánea)

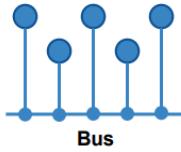


Concurrente

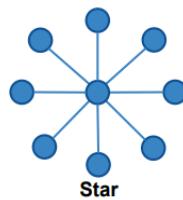
(ejecución intercalada)



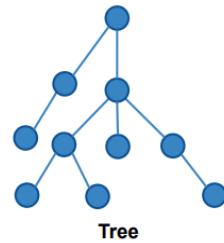
Topologías



Bus



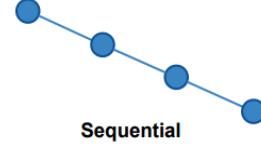
Star



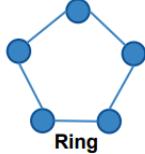
Tree



Mesh



Sequential



Ring

Centralizado vs distribuido

Sist. Centralizados	Sist. Distribuidos
<ul style="list-style-type: none">• Sin conexiones• Con conexiones pero:<ul style="list-style-type: none">• Sin trabajo colaborativo• Sin un objetivo común• Sistemas de tiempo compartido• 'Terminales de conexión'• Muy difíciles de escalar:<ul style="list-style-type: none">• CPUs, Memoria, HD	<ul style="list-style-type: none">• Componentes conectados y:<ul style="list-style-type: none">• realizando trabajo colaborativo• buscando un objetivo común• Escalan distribuyendo trabajo y recursos:<ul style="list-style-type: none">• Nodos, regiones, canales

+ Control

+ Homogeneidad : 2 (2 fuerzas)

+ Consistencia

+ Seguridad : menor superficie de ataque

+ Disponibilidad

+ Escalabilidad

- Reducción de latencia

- Colaboración

- Movilidad

- Costo

Concentrar la autoridad

Coordinación de actividades

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"

(Conway M., *How do committees invent*, Datamation, 1968)

- Este enunciado fue demostrado empíricamente en distintos relevamientos de arquitecturas de software corporativo.
- Corolario: diseñamos de acuerdo a lo que conocemos y estamos acostumbrados a hacer en el día a día.
- No es necesariamente negativo: en su trabajo, el hombre tiende a encontrar soluciones distribuidas y paralelas eficientes (minimizan costo, energía, tiempo, etc.)

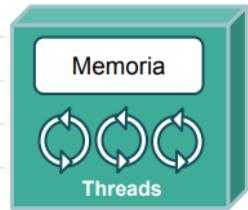
Virtualización

- Independencia entre recursos

2. Multitasking y Comunicaciones

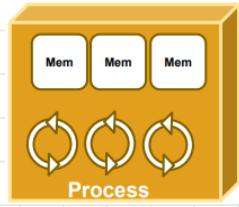
Multithreading

- Recurso compartido (heap, file descriptors, code segment, etc)
- Sincronización (SO, runtime, IPCs)
- Mejor escalabilidad
- Fácil compartir información

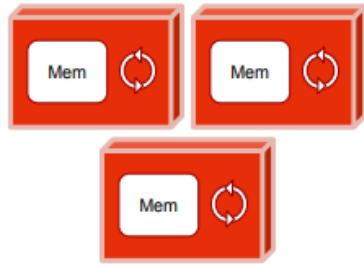


Multiprocessing

- Recurso compartido (code segment)
- Sincronización (IPCs : signals, pipes, queues, locks, sockets, etc)
- Componentes separados
- Muy escalable y estable
- No hay tolerancia a fallas de hardware, OS, etc



Multicomputing



- Recursos compartidos : ninguno
- Sincronización : mensajes entre computadoras
- Comunicación por red : problemas de latencia
- Complejo
- Alta escalabilidad

Comparación

Categoría	Multithreading	Multiprocessing	Multicomputing
Recursos Compartidos	<ul style="list-style-type: none"> • Memoria (heap, data segment) • File descriptors • Code segment (solo lectura) • Soporte del SO (pthread, mutex, etc.) • Soporte del runtime (Java, .NET, etc.) 	<ul style="list-style-type: none"> • Solo code segment (lectura) • IPCs: <ul style="list-style-type: none"> - Shared memory - Pipes/FIFOs - Queues - Semáforos - Sockets - Signals 	<ul style="list-style-type: none"> • Ninguno compartido • Cada nodo tiene su propia memoria y recursos
Sincronización	<ul style="list-style-type: none"> • Compartición de datos sencilla • Alto acoplamiento entre componentes • Difícil de depurar • Un thread defectuoso puede afectar a todos 	<ul style="list-style-type: none"> • Compartición más compleja • Bajo acoplamiento • Mayor aislamiento entre componentes • Más robusto ante fallos internos 	<ul style="list-style-type: none"> • Sincronización explícita mediante mensajes • Mecanismos propios deben ser implementados
Estabilidad	<ul style="list-style-type: none"> • Baja: un fallo puede comprometer todo el sistema 	<ul style="list-style-type: none"> • Media: procesos aislados, pero compartiendo kernel 	<ul style="list-style-type: none"> • Alta: nodos independientes, tolerancia a fallos
Escalabilidad	<ul style="list-style-type: none"> • Limitada por recursos compartidos y sincronización 	<ul style="list-style-type: none"> • Buena: procesos pueden escalar en múltiples núcleos 	<ul style="list-style-type: none"> • Muy alta: cada computadora puede ser un nodo independiente
Comunicación	<ul style="list-style-type: none"> • Memoria compartida interna al proceso 	<ul style="list-style-type: none"> • IPC (inter-procesos dentro de un mismo sistema operativo) 	<ul style="list-style-type: none"> • Comunicación por red (Sockets, RPC, gRPC, etc.) • Problemas de latencia, pérdida, ancho de banda
Uso típico	<ul style="list-style-type: none"> • Aplicaciones ligeras con muchas tareas concurrentes • UI, juegos, servidores web ligeros 	<ul style="list-style-type: none"> • Procesamiento paralelo de datos • Cargas intensivas en CPU o I/O 	<ul style="list-style-type: none"> • Sistemas distribuidos • Microservicios • Computación en la nube

Propiedades

SAFETY → Exclusión mutua

→ Ausencia de deadlocks

LIVENESS → Ausencia de starvation

→ Fairness

BAJADA EN
ALGORITMOS → Busy-waiting

→ Spin-lock

→ Algoritmos de espera

BAJADA EN
ABSTRACCIONES → operaciones atómicas

Mecanismos de sincronización

- Semáforos (Múltex → para acceder a sección crítica)
- Monitores
- Conditional Variables
- Barrier
- Rendezvous

IPCs

- Permiten comunicación entre procesos
- Proveidos por el SO

Mecanismo de sincronización	IPC
Semáforo	Semáforo
??	Shared Memory
Monitor	File Lock
Barrera	??
Rendezvous	Signal
	Queue
	Pipes / Fifos
	Sockets

3. Parallelización, Multiprocesadores y Número

Parallelización de Tareas

- Reducir el tiempo de cálculo (latencia)
- Incrementar la cantidad de tareas (throughput)
- Reducir energía consumida

O
B
S
E
T
I
V
D
S

Camino crítico

- Máxima longitud de tareas secuenciales a computar
- Define el mejor rendimiento

Ley de Amdahl

1 - Todo trabajo de cálculo se divide en fracciones secuenciales y paralelas

2 - Utilizando P unidades de cálculo, el tiempo de ejecución puede reducirse

ley de Gustafson

- Aumentar el paralelismo puede permitir la modificación del problema original para ejecutar más trabajo

Modelo Work-Span

SPEED UP: Ratio de optimización de una operación $S_p = T_1/T_p$

- **Características**
 - Modelo más cercano a la realidad para estimar optimizaciones que el usado por Amdahl
 - Provee una **cota inferior** y una **cota superior** para el **Speedup**
- **Hipótesis**
 - *Paralelismo Imperfecto*: No todo el trabajo paralelizable se puede ejecutar al mismo tiempo
 - *Greedy scheduling*: proceso disponible => tarea ejecutada
 - Tiempo de acceso a memoria despreciable
 - Tiempo de comunicación entre procesos despreciable
 - Posibilidad de analizar la operación/algoritmo en caja blanca
- Definiciones
 - **T_1 (work)**: Tiempo en ejecutar "operación/algoritmo" con 1 sólo proceso
 - **T_{inf} (span)**: Tiempo en ejecutar el **caminio crítico** de la "operación/algoritmo"

Cota	Speedup	Consideraciones
Superior	$\min(P, T_1/T_{inf})$	Se obtiene P en escenarios de Speedup lineal.
Inferior	$(T_1 - T_{inf})/P + T_{inf}$	El trabajo se puede dividir en perfecta e imperfectamente paralelizable

Estrategias de particionamiento

• Descomposición Funcional

$\text{foo(data)} = f(\text{data}) + g(\text{data}) + h(\text{data})$ //1 proceso máximo

vs

$\text{foo(data)} = \text{go } f(\text{data}) + \text{go } g(\text{data}) + \text{go } h(\text{data})$ //3 procesos máx.

• Particionamiento de Datos

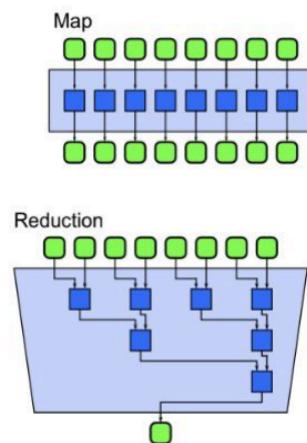
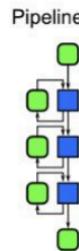
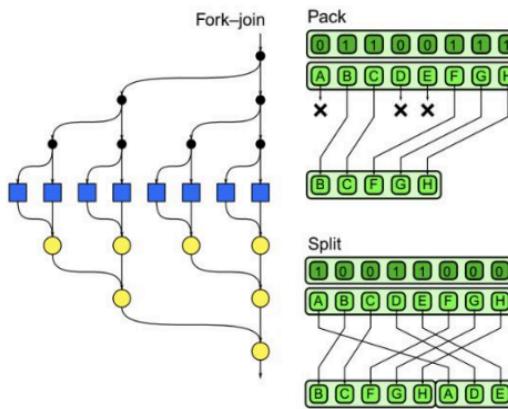
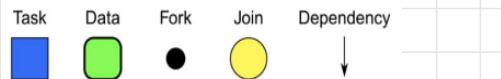
$\text{foo(data)} = f(\text{data})$ //1 proceso máx.

vs

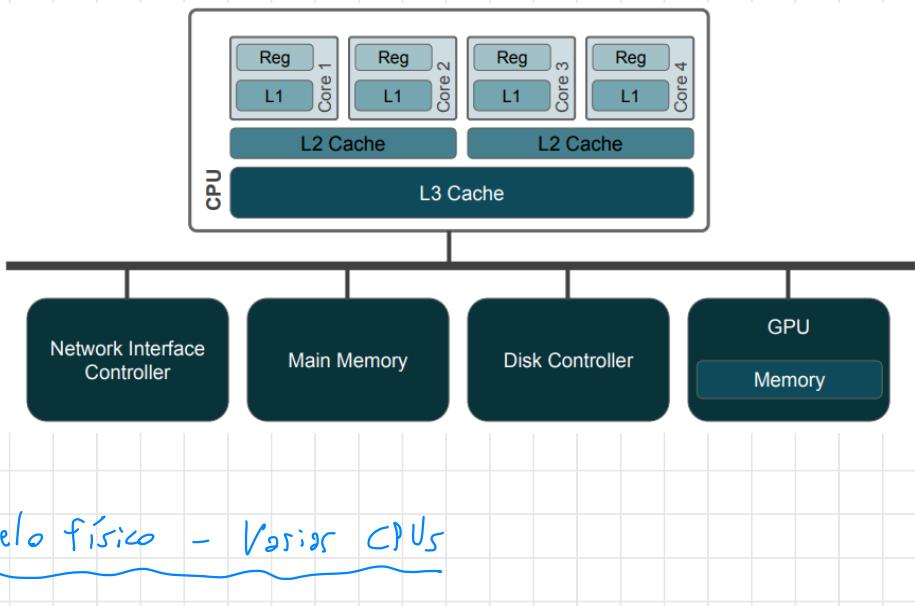
$\text{foo(data)} = \text{go } f(\text{data}[0:N/P-1]) \& \dots \& \text{go } f(\text{data}[(P-1)*N/P:N-1])$
//P procesos máx. (sólo si $f(x)$ es particionable)

Partición de procesamiento

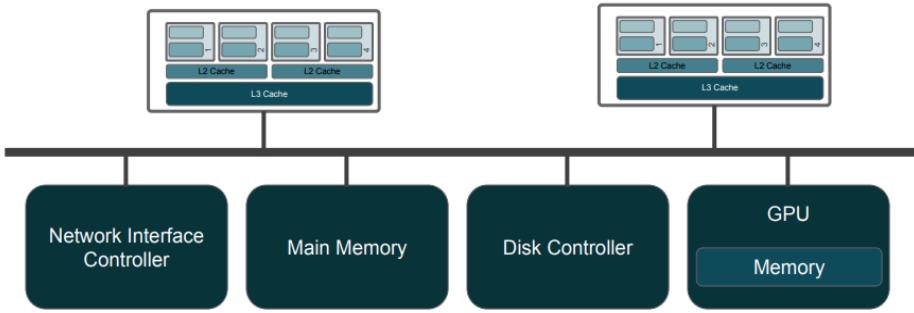
- basados en algoritmos



Modelo físico - UNA CPU



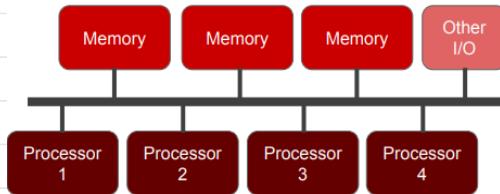
Modelo físico - Varios CPUs



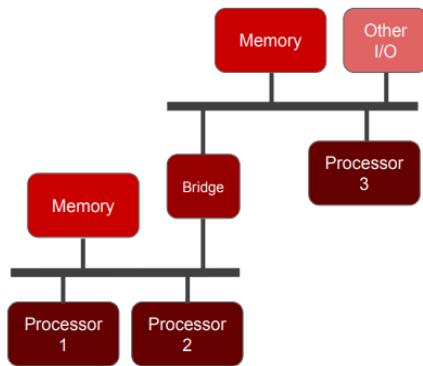
Taxonomía de Flynn

- SISD (Single Instruction Single Data) : modelo estandar de un procesador sin paralelismo
- SIMD : array processor (GPU)
- MIMD : no son usuarios
- MIMD
 - multiprocessors con memoria y/o clock compartidos
 - multicamputers sin recursos compartidos

Symmetric Multiprocessing



Asymmetric Multiprocessing



Multiprocesores UMA vs NUMA

Característica	UMA (Uniform Memory Access)	NUMA (Non-Uniform Memory Access)
Acceso a memoria	Uniforme: todos los procesadores acceden a la memoria al mismo tiempo y velocidad	No uniforme: cada CPU tiene acceso rápido a su memoria local y más lento a la remota
Distribución de memoria	Memoria compartida equitativamente entre todos los procesadores	Memoria dividida en bloques, cada uno asociado a un procesador (home agent)
Ancho de banda	Compartido entre todos los procesadores	Mayor ancho de banda si se accede principalmente a la memoria local
Balance de performance	Equilibrada para aplicaciones generales	Puede ofrecer mejor rendimiento si el software es consciente de la topología de memoria
Complejidad del sistema	Más simple, menos configuración	Más compleja: requiere soporte del sistema operativo y del hardware
Ejemplos de uso	Sistemas tradicionales, desktops, algunos servidores	Servidores de alto rendimiento, virtualización, entornos cloud modernos
Presente en	Arquitecturas tradicionales, SMP (Symmetric Multiprocessing)	Linux Kernel, Windows Server, instancias Cloud (AWS, Azure, etc.)
Historial	Modelo clásico y ampliamente adoptado	Resurgió con fuerza en la nube tras años de menor uso

Múlticomputos

- Cada computador tiene su memoria local
- Cada comp. puede fallar independientemente
- No hay elojo central
- Requieren comunicación

Nombres y direccionamiento

• Nombres

- Permiten **identificar únicamente** a una entidad dentro de un sistema
- Deben describir a la entidad
- Abstraen al recurso de las propiedades que atan al mismo con el sistema (lugar geográfico, direcciones de red)

• Direccionamiento (Addressing)

- **Mapeo entre un nombre y una dirección**
- Dirección de una entidad puede cambiar, **nombre no (*)**
- Dirección puede ser reutilizada

• Domain Name (name) -> IP Address (address)

- Mapeo de un servicio/nodo/otra entidad a una dirección IP
- Traducción a través de protocolo DNS

• IP Address (name) -> Ethernet Address (address)

- IP address identifica a un nodo en un red (sea local o no)
- Ethernet address identifica a NIC (network interface card) de un nodo en una red local
- Resolución se realiza a través de protocolo ARP en IPv4 o **ND** (Neighbor Discovery) en IPv6

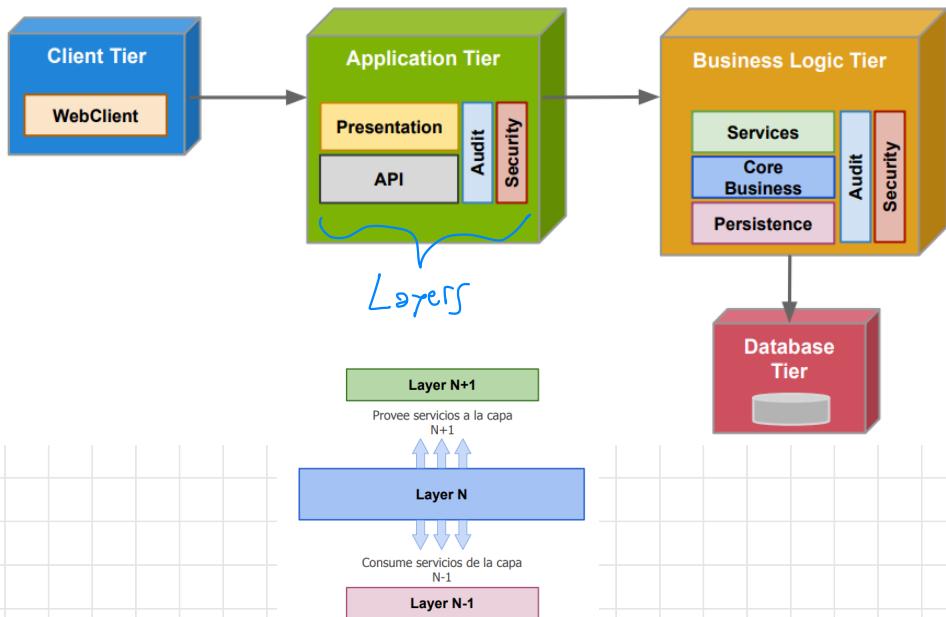
• Service (name) -> Instances (address)

- Mapeo del nombre de un servicio a alguna instancia
- Resolución a través de Service Discovery
- Diferentes implementaciones existentes: Zookeeper, Istio, Linkerd

9. Interfaces y Protocolos

Arquitectura de capas

- Permiten dividir en sub problemas
- Uso de interfaces
- Tipos
 - Layers (lógico)
 - Tiers (física)



Interfaces

- Permiten comunicación entre componentes / sistemas
- Se expone una parte del sistema
- Esconden implementación

Inter-Aplicaciones	Intra-Aplicaciones
Application Programming Interfaces (APIs) Ej: <ul style="list-style-type: none">• Cliente por consola consultando Web Server• JS de navegador consultando servidor• Servicio consultando Servicio	Facades - Mediators - Interfaces Ej: <ul style="list-style-type: none">• Layer 2 consultando Layer 1• Mensaje enviado a un objeto local• Mensaje enviado a un objeto remoto (?)

Clasificación

Web APIs

- Web Services based APIs ([SOAP](#))
- REST based APIs

Library-based / Frameworks

- Java API
 - Ej.: OpenJDK vs Oracle JDK
- Android API

Remote APIs

- Custom TCP/UDP services
- Object oriented: CORBA, JavaRMI
- Procedure oriented: RPC, gRPC

OS related

- POSIX
 - Ej.: Linux vs OpenBSD
- WinAPI

Protocolos

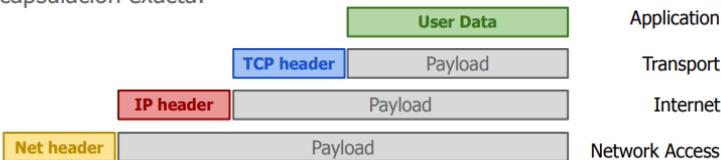
- Modelo client-server / request-reply,
- Sin estado

PDU (Protocol Data Unit)

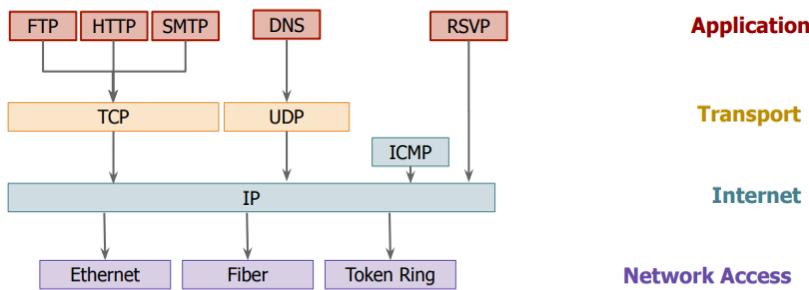
Encapsulación de PDUs entre capas:

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

Ej. Encapsulación exacta:



Ejemplos



Mensajes : BINARIO VS TEXTO

Característica	Binario	Texto Plano
Performance	Alta performance - Tamaño de mensajes eficientes - Compresión puede no ser necesaria	Baja performance - Throughput bajo - Compresión agrega overhead
Serialización	- Uso de formatos como Protobuf, Thrift, ASN.1 - Autogeneración de código disponible - No siempre hay soporte en todos los lenguajes	- Formatos human-readable como JSON, XML - Serialización básica (e.g. HTTP, SMTP) - Fácil de implementar y usar en la mayoría de los lenguajes
Interacción	- Cliente específico por aplicación - Requiere decoder para interpretar los mensajes	- Cliente genérico si se conoce el protocolo (e.g. cURL + REST API) - Fácil de debuggear y analizar
Portabilidad	Limitada por falta de soporte en algunos lenguajes	Alta, gracias a su legibilidad y compatibilidad
Facilidad de uso	Requiere herramientas y conocimiento específico	Intuitivo y accesible para humanos
Debugging	Difícil, requiere herramientas específicas	Simple, se puede leer directamente con cualquier editor de texto

Longitud → Fija → fijo
 ↘ Variable → agrega overhead
 ↘ Mixta

Sincrónicos → emisor espera respuesta

Asincrónicos → emisor hace otras tareas sin esperar respuesta

Restful

- Alta performance
- Escalabilidad
- Confidencialidad
- HTTP/HTTPS
- JSON/XML
- Versionado de APIs

Operación	Verbo HTTP	Equivalente SQL
Create	POST	INSERT
Read	GET	SELECT
Update	PUT	UPDATE
Delete	DELETE	DELETE

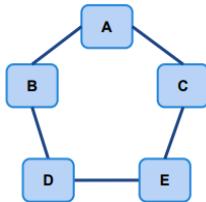
5.1. Mensajes, grupos y middlewares

Grupos

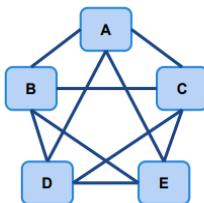
- Permiten ver a una colección de procesos como una abstracción
 - Mensaje a todos o alguno de los entidades
 - Son dinámicos
 - Suscribir > cancelar a grupos
-
- Unicast : punto a punto
 - Anycast : solo uno recibe
 - Multicast : solo un grupo
 - Broadcast : todos
- } Difusión de mensajes

Topología

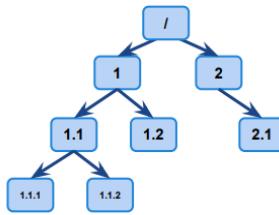
Anillos



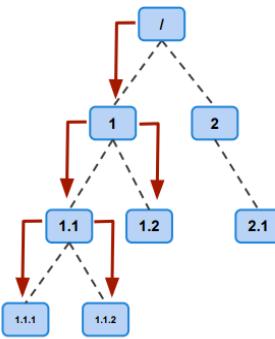
Punto a Punto



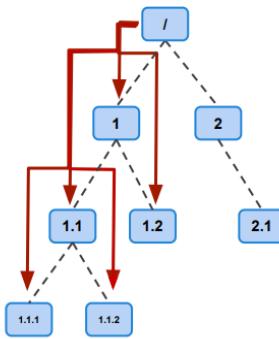
Grupos Jerárquicos



Difusión Descentralizada



Difusión Centralizada



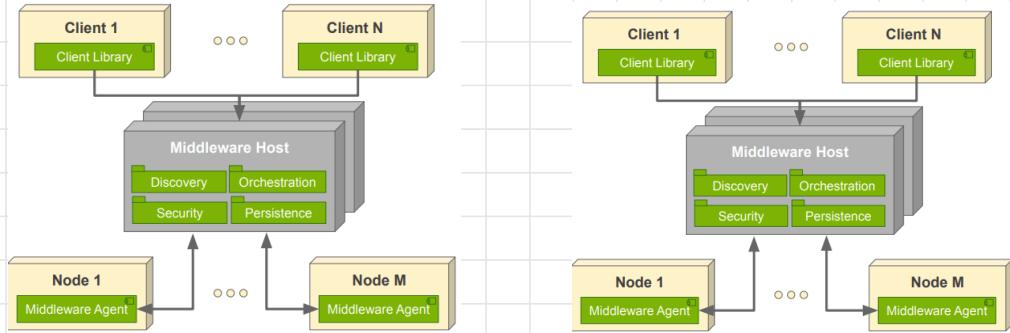
Atomicidad de mensajes

- Se entrega a todos o ninguno
- ACK
- Demora en delivery
- Reintentos

Middleware

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

- Transparencia : Se oculta distribución
 - Tolerancia a fallos
 - Recursos compartidos
 - SD abierto
 - Comunicación en grupo
- Pueden ser centralizadores (host) o distribuidos



Tipos de middleware

Transactional processing:

- Garantizan transaccionabilidad de operaciones
- Confiables
- Tolerancia a fallos

Object oriented

- Manejadores hacia objetos distribuidos que viven en el middleware

Procedure oriented

- El middleware trabaja como un servidor de funciones

Message oriented (rabbit)

- Mensajes entre apps

Moms (message oriented middleware)

- Comunicación transparente entre apps
- Mensajes
- Tolerante a fallos
- Bus de información o colas

Sincrónico

- + Conexión punto a punto
- + Respuestas instantáneas
 - No hay transparencia frente a errores

Asincrónico

- + Colas
- + Soporta colas
 - Complejo recibir respuestas

Características de mensajes

- Multiples
- Definidos en el MOM
- Nombre y longitud
- Garantía de que el mensaje sera entregado

Brokers

- Provee transparencia
- Lógica del middleware
- Punto de control y monitoreo

6.1. Rabbit

Queues

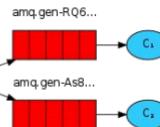
- Nombradas vs TaskQueue vs Anónimas
- Actuación autónoma por defecto
- Durabilidad definida en la cola y mensaje

Exchanges

- Implementan estrategias para transmitir mensajes
- Fanout, direct, topic, headers

Publisher - Subscriber

- **Productor**
 - envía mensajes a un exchange de tipo **fanout**
- **Consumidores**
 - crean colas anónimas para recibir mensajes del productor
 - colas anónimas son *bindeadas* a exchange del productor para comenzar a recibir mensajes
- **Exchange Fanout:** Realiza un *broadcast* de todos los mensajes recibidos a todas las colas conocidas



Routing

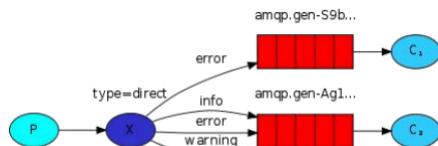
- **Productor**

- Envía mensajes a un exchange de tipo *direct*
- Adosa al mensaje un identificador de routeo (*routing_key*)

- **Consumidor**

- Realiza binding a exchange *direct* con los *routing_keys* que desea recibir

- **Exchange Direct:** Redirige mensajes con una *routing_key* específica a aquellas colas que se encuentran *bindeadas* a la misma



Topic

- **Productor**

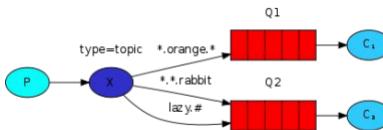
- Envía mensajes a un exchange de tipo *topic*
- Adosa al mensaje un identificador de routeo (*routing_key*)

- **Consumidor**

- Realiza binding a exchange *topic* con los *patrones* que desea recibir

- **Exchange Topic:** Soporta patrones de búsqueda basados en palabras.

- routing_key* es un conjunto de palabras separadas por punto
 - *: Permite sustituir una palabra
 - #: Permite sustituir una o más palabras

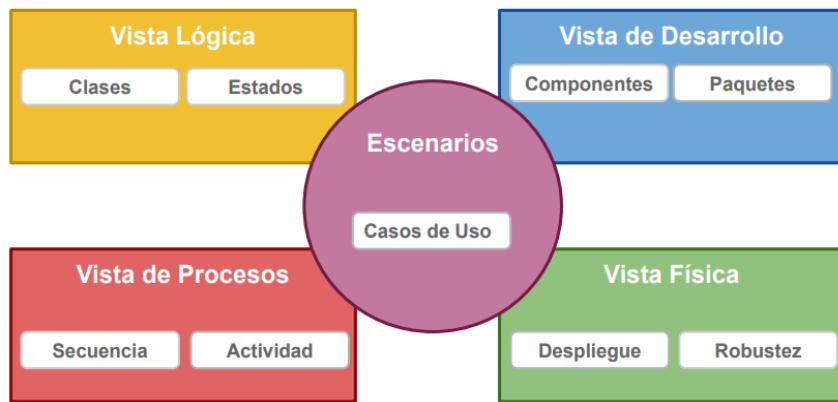


6.2. Documentación

Diseño y Documentación:

- Evolutivo:
 - Adaptarse rápido, tomar feedback y aportar valor iterativamente
 - No buscar el entendimiento del todo y ni demorar la arquitectura
- Necesario para coordinación, coherencia y cohesión
 - Sin un diseño preliminar, probablemente jamás haya diseño.

4 + 1



Vista Lógica

- Estructura y funcionalidad del sistema (Clases, Estados)

Vista de Física (o Despliegue)

- Topología y Conexiones entre componentes físicos (Despliegue)
- Expone la arquitectura del sistema (Robustez)

Vista de Desarrollo (o de Implementación)

- Artefactos que componen al sistema (Paquetes, Componentes)

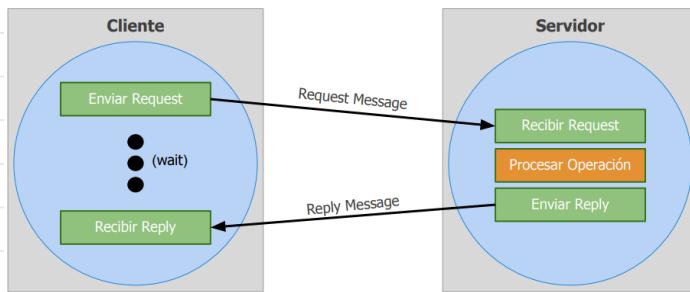
Vista de Procesos (o Dinámica)

- Descripción de escenarios concurrentes (Actividades)
- Flujo de mensajes en el sistema (Colaboración)
- Flujo temporal de mensajes en el sistema (Secuencia)

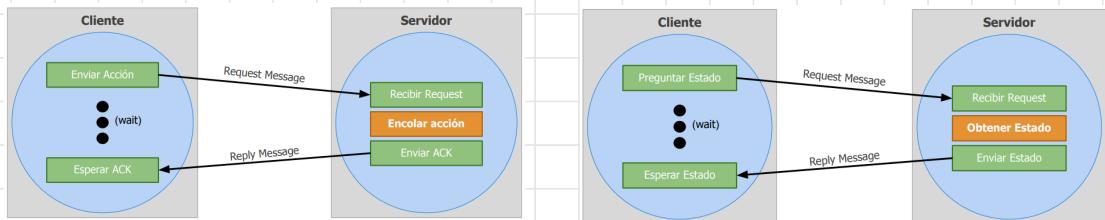
7.1. Patrones de Comunicación

Request - Reply

- Poco cliente - servidores
- Sincrónico (bloqueante) por defecto
- ACKs necesarios (esenciales)



↓ Y si quiero que sea asíncronico? Usar 2 Request - Reply



Estructura de mensajes

- Message Id ($0 = \text{Request}$, $1 = \text{Reply}$)
- Request Id (incremental o uuid)
- Operation Id
- args

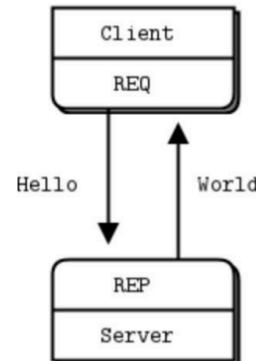
Tolerancia a fallos → Timers con retries

- ¿Qué pasa si un Request o un Reply se pierde?

Estrategia	Tipo de Control	Retry - Request	Filtro Duplicados	¿Mensaje recibido?
#1	Sin control	No	No implementable	<i>Maybe</i>
#2	Re-ejecución	Si	No	<i>At Least Once</i>
#3	Retransmisión	Si	Si	<i>Exactly Once</i>

Zer0MQ

- Cliente-Servidor convencional (aunque no tanto...)
- No posee primitiva *accept*
- Primitiva *bind* funciona como *bind + accept*
- Primitiva *send* es **no bloqueante**
- Cliente no necesita esperar a que el servidor esté corriendo para enviar mensajes
- Cómo funciona *under the hood*?
 - I/O threads: 1 thread per GB/s (in or out)
 - Buffering



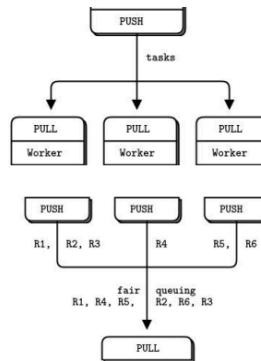
Producer - Consumer

Modelo basado en comunicación por tareas entre productores y consumidores

- **Producers:** son los emisores. Componentes que generan cierta información que se considera la materia prima para cierto procesamiento posterior.
- **Consumers:** son los receptores. Esperan la aparición de cierta información para efectuar un procesamiento particular.

ZeroMQ

- Comunicación de tareas de un productor a un consumidor.
- Admite múltiples consumidores y/o múltiples productores.
- Garantiza *fairness* en la entrega de mensajes: round robin
- Utiliza los sockets PUSH/PULL para marcar el rol de cada extremo.



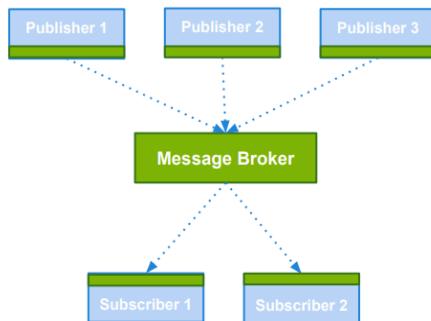
Publisher - Subscriber

Modelo basado en comunicación por eventos entre productores y consumidores

- **Publishers:** son los emisores. Componentes que tienen la posibilidad de generar algún **elemento de interés**.
- **Subscribers:** son los receptores. Esperan la aparición de algún **evento de su propio interés** sobre el cual efectuarán alguna acción.

Dos posibles arquitecturas

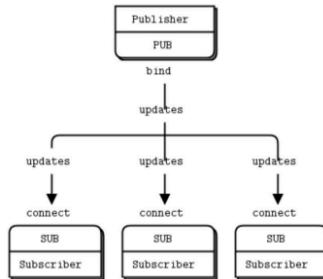
- **Basada en tópicos:** publicación y suscripción indicando el tipo de evento, tópico o tag.
- **Basada en Canales:** publicaciones y suscripciones orientadas a canales específicos.



• Con Bus o Colas

ZeroMQ

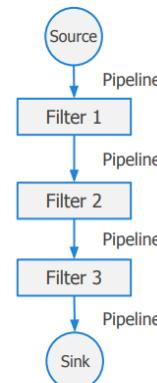
- Un ZMQ PUB socket publica mensajes
- Message pattern: `id field1 field2 ...fieldN`
- N ZMQ SUB sockets se suscriben a los Eventos que desean recibir suscribiéndose al ID del evento
- Suscripción puede ser cancelada en cualquier momento
- Mensaje es enviado a todos los sockets suscriptos a un evento determinado
- Múltiples publishers? [XPUB-XSUB pattern](#)



Pipelines

- En arquitecturas de software se lo conoce como '*Pipelines and Filters*'
- Los datos de entrada forman un flujo donde distintos *filters* (o *processors*) se conectan entre sí para procesarlos de manera secuencial
- Inspirado en patrones de procesamiento de señales, es muy utilizado en entornos Unix:

```
cat in | grep pattern | sort | uniq > out
```



Admite dos modelos de procesamiento:

- **Worker por Filter:** se asigna una unidad de procesamiento a cada etapa del pipeline. Los items son recibidos por el *worker*, procesados y enviados a la próxima etapa.
- **Worker por Item:** se asigna una unidad de procesamiento a cada item. Un *worker* toma al item ingresado y lo acompaña hasta el final del pipeline, aplicándole los *filters* paso a paso.

Cada uno de los *processors* funciona como una etapa, pudiendo ser del tipo:

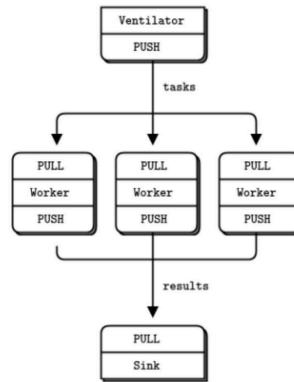
- **Paralela:** cada item a procesar es independiente de anteriores y posteriores por lo que admite paralelismo.
- **Secuencial:** no puede procesar más de un item a la vez. Ya procesados, los puede retornar:
 - Ordenados
 - Desordenados

Ventajas

- + Algoritmos ONLINE : permite iniciar el procesamiento sin tener todos los datos (items)
- + Información infinita

ZC50 MQ

- Patrón Productor-Consumidor
- Chaining de Productores-Consumidores da como resultado un pipeline
- Mensajes son consumidos de forma Equitativa (**fairness**)
 - Qué lógica utiliza para decidir esto?
- Combinaciones
 - 1 PUSH -> N PULL
 - N PUSH -> 1 PULL

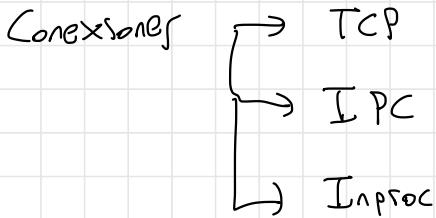


DAG (Directed Acyclic Graphs)

- Muestra el flujo de datos
- Nodos = Coseas
- Aristas = flujo de datos
- Acíclicos
- Muestra dependencia entre procesos y bloques (deadlock)

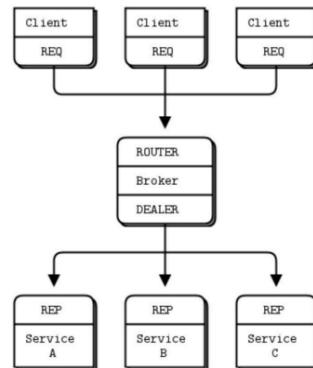
7.2. ZeroMQ

- Sockets en asteroides
- Para crear brokers less middlewares
- El usuario se encarga de serializar
- Soporte para varios patrones (vistas anteriores)



Router - dealer

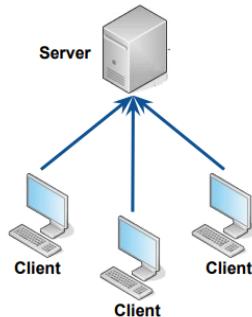
- **ROUTER socket**
 - Agrega al mensaje recibido un ID de destinatario
- **DEALER socket**
 - Rutea los mensajes de forma justa (**fair**)
 - Propaga el ID de origen del mensaje
- Ambos sockets permiten recibir mensajes de **múltiples sockets a la vez**
- Ambos sockets son **asincrónicos**
 - Poll para recibir mensajes



8. Arquitecturas Distribuidas Simples

Cliente - Servidor

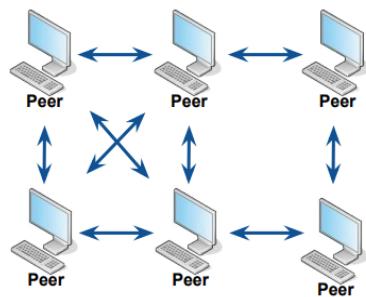
- Se definen roles para los participantes:
 - Servidor como elemento pasivo y provee servicios
 - Clientes activos que envían pedidos al servidor
- Permite centralización en toma de decisiones
- Suele asumirse que los servidores tienen más capacidades de *hardware* que los clientes



• Los clientes No se comunican entre si

Peer to peer

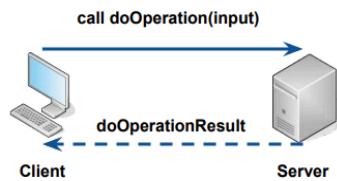
- Se establece una red de nodos que se consideran pares entre sí
- Asume capacidades de recursos similares entre los pares
- Muy útil cuando existen objetivos de colaboración por parte del negocio
 - Protocolo acordado entre partes
 - La lógica distribuida requiere coherencia entre los nodos
- Auge en internet a partir de la invención de Napster, BitTorrent, etc



• Difícil establecer comunicación
• Requiere mucho networking

RPC (remote procedure call)

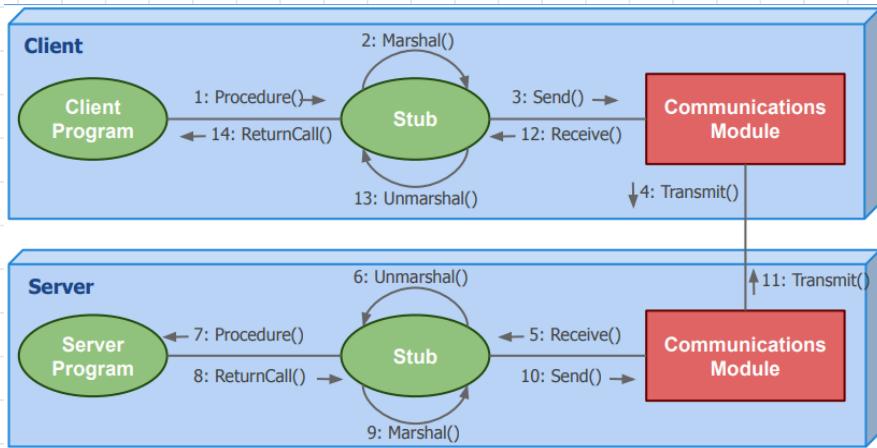
- Ejecución remota de procedimientos
- Modelo Cliente-Servidor
 - Cliente realiza una llamada a un Procedimiento
 - Servidor responde con el resultado de la operación
- Comunicación remota transparente para el usuario
- Portabilidad a través de implementación de interfaces bien definidas



- Requiere IDL (Interface definition language)
 - Para que distintos lenguajes se puedan llamar
- Se deben manejar fallas con:
 - Retries con timeout
 - Filtros de duplicados
 - Retransmisión

Estrategia	Tipo de Control	Retry - Request	Filtro Duplicados	¿Mensaje recibido?
#1	Sin control	No	No implementable	Maybe
#2	Re-ejecución	Si	No	At Least Once
#3	Retransmisión	Si	Si	Exactly Once

- **Cliente**
 - Se encuentra conectado a un *stub*
 - Realiza llamadas de forma transparente al servidor (o no tanto)
- **Servidor**
 - Se encuentra conectado a un *stub* del cual recibe parámetros
 - Posee lógica particular del *remote procedure*
- **Stubs**
 - Administra el *marshalling* de la información
 - Envía información de llamadas (*calls*) al módulo de comunicación y al cliente / servidor
- **Módulo de comunicación**
 - Abstacta al *stub* de la comunicación con el servidor



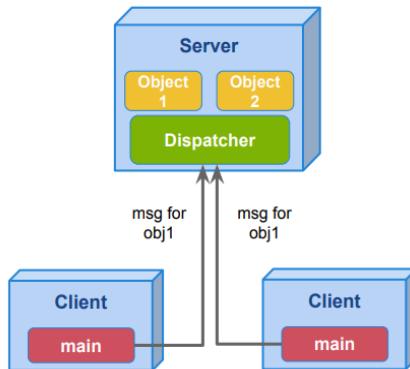
gRPC

- HTTP2 para Transporte
- Protocol buffers para encoding
- Punto a punto
- Servicios y mensajes en archivos .proto
- Alta performance y para microservicios

Objetos Distribuidos

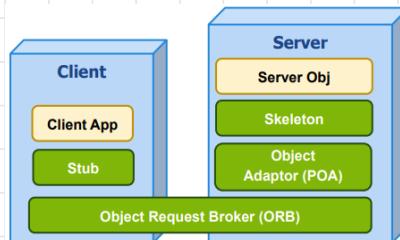
- Los servidores ya no proveen servicios sino objetos
- Existe un middleware que oculta la complejidad de:
 - Referencias a Obj. remotos
 - Invocación de acciones
 - Errores (excepciones)
 - Recolección de basura

- STATEFUL



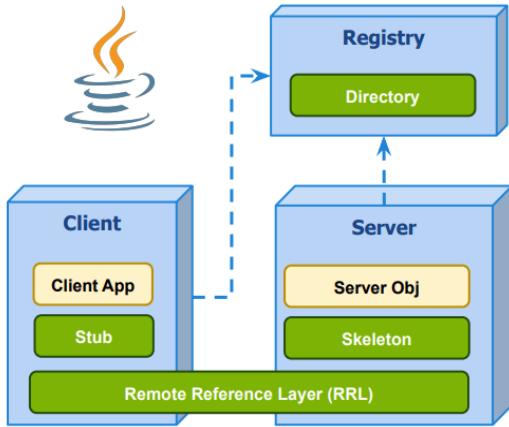
Corba

- Protocolo y serialización
- Transporte
- Seguridad
- Discovery de Objetos



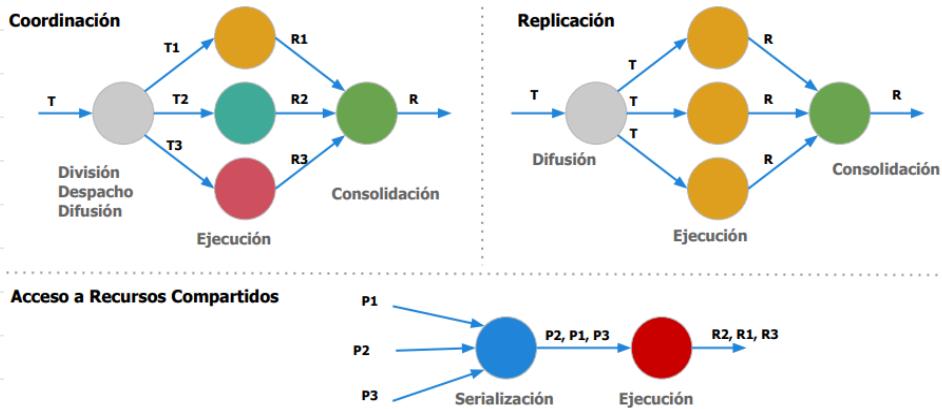
RMI

- Versión optimizada de Distributed Objects en Java
- Requiere los siguientes pasos:
 - 1- Registro del servidor en un directorio de servicios.
 - 2- Consulta del registro por parte del cliente
 - 3- Invocación desde el cliente al servidor



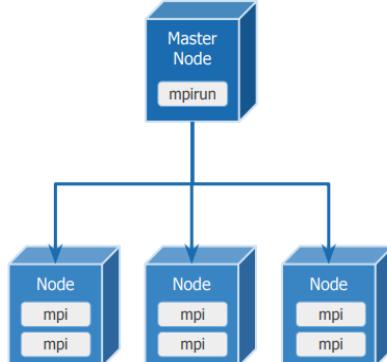
9. Distribución y coordinación de procesos

Coordinación de actividades



Open MPI

- Basado en transmisión y recepción de mensajes.
- Ejecución transparente de 1 a N nodos.
- Se utiliza como una librería con abstracciones de uso general con **foco en el cómputo distribuido**
- Implementa un middleware de comunicación de grupos:
 - MPI_Recv, MPI_Send, MPI_Bcast, MPI_gather, etc.



Flint

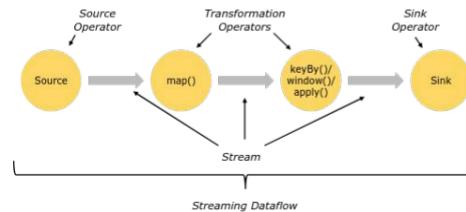
- Plataforma para procesamiento distribuido de datos.
 - Incluye motor de ejecución de pipelines de transformación.
 - Define *framework Java/Scala* para crear pipelines:
 - SQL y Table API permiten definir tablas dinámicas (lógicas) con los flujos de datos y utilizar álgebra relacional
 - Dataset y DataStream API permiten definir secuencias de procesamiento con formato DAG



- **Dataflow:** DAG de operaciones sobre un flujo de datos
 - **Streams:** un flujo de información que puede no finalizar
 - **Batches:** un conjunto de datos (*dataset*) de tamaño conocido

```
DataStream<String> lines = env.addSource(  
    new FlinkKafkaConsumer<>(...)); } Source  
  
DataStream<Event> events = lines.map((line) -> parse(line)); } Transformation  
  
DataStream<Statistics> stats = events  
    .keyBy("id")  
    .timeWindow(Time.seconds(10))  
    .apply(new MyWindowAggregationFunction()); } Transformation  
  
stats.addSink(new RollingSink(path)); } Sink
```

- **Source:** bloque capaz de injectar datos al *pipeline*
 - **Transformation:** también conocido como operador. Es un **nodo de modificación de datos o filtrado de los mismos**
 - **Sink:** bloque de destino de la información. Almacenamiento final del *pipeline*



Casos de uso



- **Extract Transform Load (ETL):** operaciones programadas de carga y modificación de datos para posterior análisis con origen y destino definidos en una DB.
 - **Data Pipelines:** tareas de procesamiento recurrentes, basadas en la ocurrencia de eventos.

Beam

Modelo de definición de **pipelines** de procesamiento de datos con portabilidad de lenguajes y motores de ejecución (*runners*).

Soporta distintos lenguajes de programación:

- Java
- Python
- Go

Soporta distintos *Runners*:

- Ejecución directa (*Stand-alone o DirectRunner*)
- Motores de *cluster*: Apache Hadoop, Apache Flink, Apache Spark
- Plataformas *cloud*: Google Dataflow, IBM Streams



beam

Los componentes básicos son similares a los definidos en Flink:

- **Input y Output:** origen y destino respectivamente de los datos (símil *Source* y *Sink*).
- **PCollection:** colecciones paralelizables de elementos (símil *Streams*)
- **Transformations:** modificaciones aplicadas a las PCollections, elemento a elemento (símil *Operators*)

10. Modelado de Cómputo distribuido

Map Reduce

- Parallel computing: parte el procesamiento en partes que se pueden ejecutar concurrentemente
 - map $f[a,b,c] \Rightarrow [f(a), f(b), f(c)]$
 - map $\text{sqrt}[a,b,c] \Rightarrow [\text{sqrt}(a), \text{sqrt}(b), \text{sqrt}(c)]$
 - reduce $f[a,b,c] \Rightarrow f(a,b,c)$
 - reduce $\text{sum}[1,2,3] \Rightarrow \text{sum}(\text{sum}(\text{sum}(1,2),\text{sum}(3,\text{sum}(\text{NULL}))))$

Caso ideal (Master - worker)

- No existe dependencia entre los datos
- Datos pueden ser partidos en **chunks** del mismo tamaño
- Cada proceso pueden trabajar con un **chunk/shard**
- **Master**
 - Encargado de **partir la data en #chunks**
 - Envía ubicación de los **chunks** a los Workers
 - Recibe ubicación de los resultados de todos los Worker
- **Workers**
 - Recibe ubicación de los **chunks** del Master
 - **Procesa el chunk**
 - Envía el ubicación del resultado de procesamiento al Master

Map

- **Map:** (input shard) → intermediate(key/value pairs)
 - Data es particionada automáticamente en K chunks y procesada por M workers ejecutando la función map
 - Función Map proporcionada por el usuario es ejecutada en todos los chunks de data
 - Usuario decide cómo filtrar la data provista en los chunks
 - Librería MapReduce agrupa todos los valores asociados con una misma key y envía ubicación de los datos al Master Process

Reduce

- **Reduce:** intermediate(key/value pairs) → result files
 - Función Reduce realiza una agregación de los datos para obtener un resultado final (result file)
 - Función Reduce es llamada por cada Unique Key
 - Realiza un merge de los datos recibidos para formar un set de datos menor
 - Función Reduce es distribuida partiendo las keys en R Reduce workers
 - La cantidad de R workers es especificada por el usuario

Tiempo

Magnitud para medir duración y separación de eventos.



Se lo puede definir mediante una **variable monótonica creciente**:

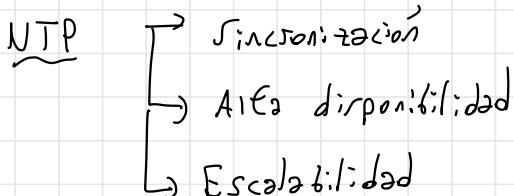
- En sistemas de computación será **discreta**
- **No necesariamente vinculada con la hora** de la vida real

La medición del tiempo permite:

- **Ordenar y sincronizar**
- **Marcar la ocurrencia de un suceso**
 - **Timestamps**: puntos absolutos en la línea de tiempo
- **Contabilizar duración entre sucesos**
 - **Timespans**: intervalo en la línea de tiempo

Relojes físicos

- Se pueden descalibrar
- GMT, UTC, TAI, etc
- No son confiables



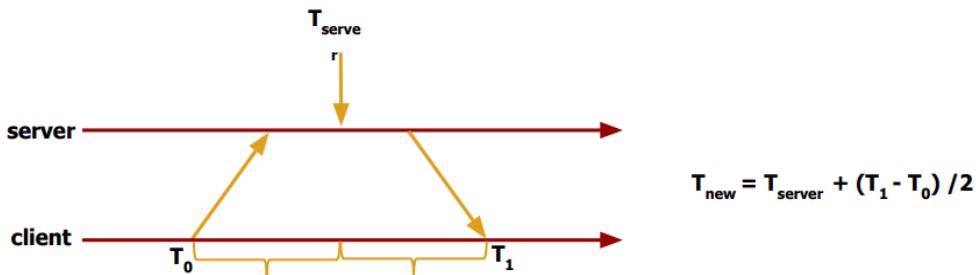
Drift

- Los relojes físicos no son confiables para su comparación por efecto del drift
- Hay que sincronizarlos periódicamente:
 - Medir el desvío respecto de un reloj de referencia (UTC, GPS, etc.)
 - Aplicar corrección o compensación lineal cambiando la frecuencia del reloj local
 - Nunca atrasar un reloj
- Hay que sincronizarlos al despertar la computadora

Para mitigar el drift se utiliza el algoritmo de Cristian

Realiza una compensación del delay existente al obtener la medida de tiempo

- T_0 : Cliente envía request
- T_1 : Cliente recibe respuesta
- Hipótesis: *Delays en la red son constantes*

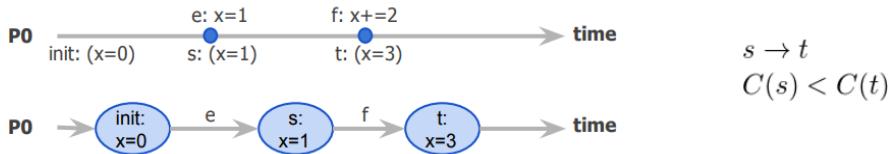


Reloj lógico

Dado S , el conjunto de todos los estados locales posibles del sistema y → la relación temporal de implicancia 'ocurre antes' (o *happened before*), un reloj lógico es una función C monótonica creciente que mapea estados con un número Natural y garantiza:

$$\forall s, t \in S : s \rightarrow t \Rightarrow C(s) < C(t)$$

Ej.:



L'import → no sirve para conservación

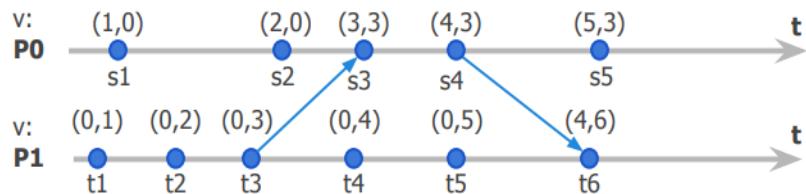
Vector de relojes

Un vector de relojes es el mapeo de todo estado del sistema compuesto por k procesos, con un vector de k números Naturales y garantiza:

$$\forall s, t \in S : s \rightarrow t \Leftrightarrow s.v < t.v$$

donde $s.v$ y $t.v$ son los vectores de k componentes para los estados s y t respectivamente y

$$s.v < t.v \Leftrightarrow \forall k : s.v[k] \leq t.v[k] \wedge \exists j : s.v[j] < t.v[j]$$

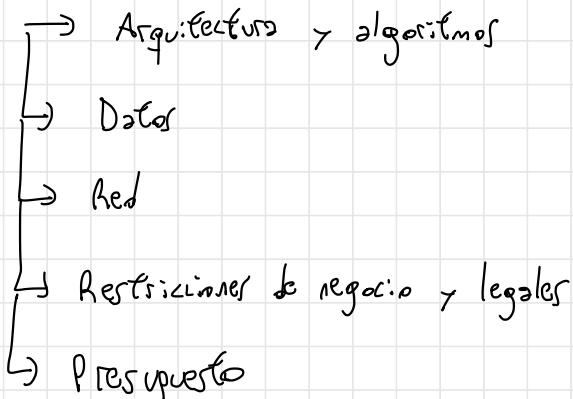


11. Sistemas clásicos y de alta disponibilidad

Escalabilidad

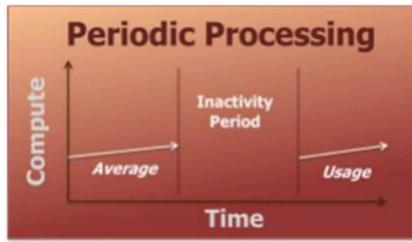
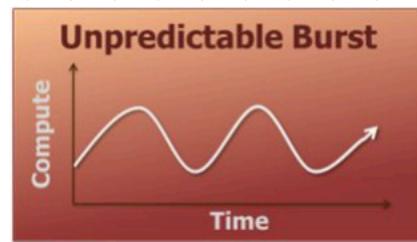
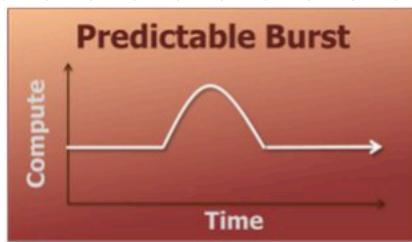
- Para alta concurrencia
- Arquitecturas Ad-hoc y personalizadas

Limitantes



- Escalamiento vertical
 - Agregar recursos a un nodo
- Escalamiento horizontal
 - Redundancia
 - Balanceadores de carga
 - Proximidad geográfica
- Fragmentación de datos
 - Fraccionar para optimizar.
 - Mantener juntos datos "cercaos"
- Componentización
 - Separar servicios
- Optimizar algoritmos
 - Performance
 - Mensajería
- Asincronismo
 - Mantener sincrónico sólo lo estrictamente necesario
 - Limitado por el negocio

Patrones de carga de aplicaciones web



Elastичidad

- Capacidad de un sistema para poder **modificar dinámicamente los recursos del sistema** adaptándose a patrones de carga
- Término utilizado en Arquitecturas Cloud. Requiere soporte de la infraestructura

- Application Load Balancers

- AutoScaler

- Monitoring automático

Alc disponibilidad

Aún cuando ciertas propiedades de confiabilidad fallen para sistemas públicos, nos interesa que siempre estén disponibles:

$$P(\text{system available}) = 1 - P(\text{failure})$$

Como $P(\text{system available}) < 1$ siempre, lo importante es definir cuán cerca de 1 se encuentra:

- $P(\text{system available}) \sim 0,9 \Rightarrow 36,5$ días caídos al año
- $P(\text{system available}) \sim 0,999 \Rightarrow 8,76$ horas caídas al año

La disponibilidad de un sistema se mide por la cantidad de 9's

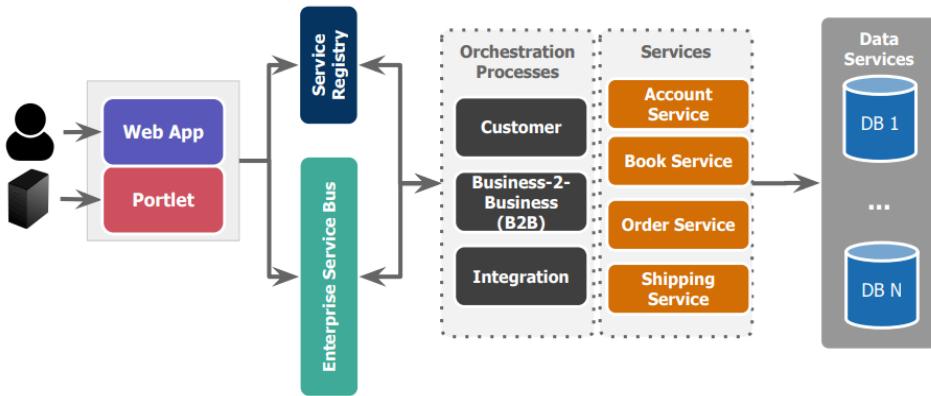
- **SLA: Service Level Agreement**
 - Contrato / Acuerdo de disponibilidad pactado con el cliente
 - Definición de qué sucede si el mismo no se respeta (e.g. [BigQuery SLA](#))
- **SLO: Service Level Objectives**
 - Lo que se debe cumplir para no invalidar el SLA
 - E.g. Availability > 99.95%
- **SLI: Service Level Indicators**
 - Métricas a ser comparadas con los SLOs
 - Siempre deben ser superiores al threshold del SLO
 - Por lo general requiere una plataforma de [observability](#)
 - Analizar impacto del [despliegue de los servicios](#)

Teorema CAP

- **Consistency:** también conocido como **repetibilidad de respuesta** de todos los nodos frente a un mismo pedido.
- **Availability:** **capacidad del sistema de responder a todo pedido.**
- **Partition Tolerance:** **capacidad de lidiar con la formación grupos aislados de nodos.**

Solo se
pueden garantizar
2 de 3

SOA (Service Oriented Architecture)



Tecnologías

- WebServices = SOAP + HTTP
- ESB preponderante para eventos
- Service Repository & Discovery para comunicación punto a punto

Procesos y Servicios

- Contract
- Interface
- Implementation: Business Logic + Data Management

12. Sincronismo, Orden y Corte de Estado

Sincronismo

Un algoritmo / protocolo es **sincrónico** si sus acciones pueden ser delimitadas en el tiempo

- **Sincrónico:** Entrega de un mensaje posee un *timeout* conocido
- **Parcialmente sincrónico:** Entrega de un mensaje no posee un timeout conocido o bien el mismo es variable
- **Asincrónico:** Entrega de un mensaje no posee *timeout* asociado

¿Cómo se generaliza esto para un sistema?

- **Steadiness**
- **Tightness**

Tiempo de Delivery ($t_D^p(m)$)

- Es el tiempo que tarda un mensaje **m** en ser **recibido** luego de haber sido enviado hacia **p**

Timeout de Delivery (T_{Dmax})

- Todo mensaje enviado va a ser recibido antes de un tiempo T_{Dmax} conocido

Steadiness (σ)

- Máxima diferencia entre el mínimo y máximo tiempo de delivery de cualquier mensaje recibido por un proceso

$$\sigma = \max(T_{Dmax} - T_{Dmin})$$

Tightness (τ)

- Máxima diferencia entre los tiempos de delivery para cualquier mensaje **m**

$$\forall p, q: \tau = \max_{m, p, q} (t_D^p(m) - t_D^q(m))$$

Steadiness (σ)

- Define la varianza con la cual un proceso observa que recibe los mensajes
- Define que tan constante (*steady*) es la **Recepción de mensajes**

Tightness (τ)

- Define la simultaneidad con la cual un mensaje es **recibido por múltiples procesos**

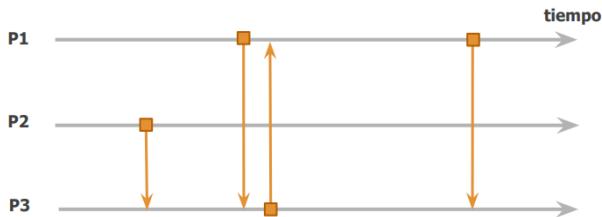
Delivery de mensajes

El delivery consiste en procesar el mensaje provocando, eventualmente, cambios en el estado del proceso

- Los mensajes se mantienen en una cola que permite controlar el momento en que se lo libera

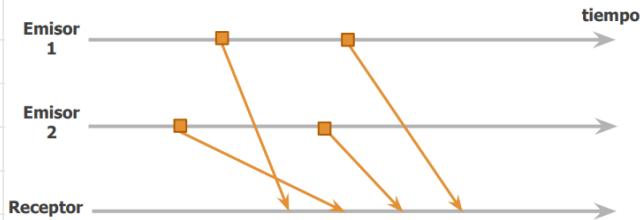
Orden sincrónico

Todo mensaje posee el mismo timestamp tanto en el proceso emisor como en el receptor. Es decir, la transmisión de mensajes insume tiempo nulo.



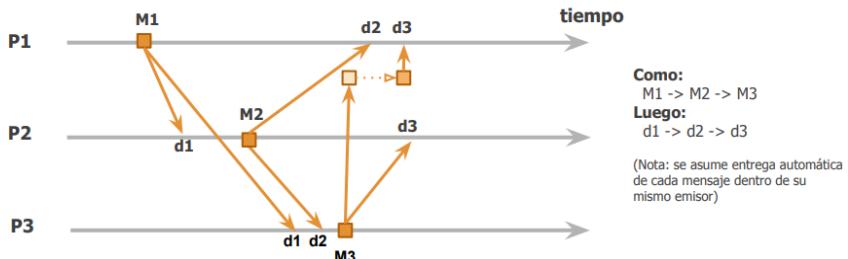
Orden FIFO

Todos los pares de mensajes entre un mismo emisor y receptor se entregan en el orden en que fueron enviados.



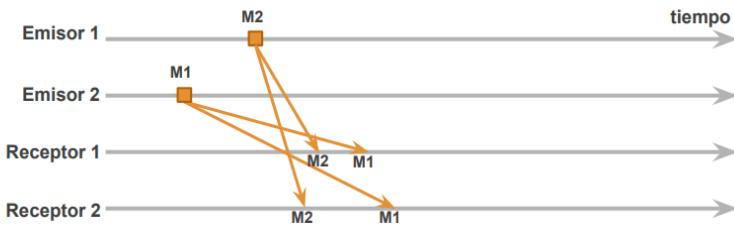
Orden causal

Todo mensaje que implique la generación de un nuevo mensaje, es entregado manteniendo esta secuencia de causalidad sin importar el receptor.



Orden Total

Todo par de mensajes entregado a los mismos receptores es recibido en el mismo orden por esos receptores.



Estado

Estado Local

Se define s_j , el estado local del proceso j en el instante t, según

$$s_j = (X_0(t), X_1(t), \dots, X_n(t))$$

con X_i = variable i del proceso j

Estado Global

Se define S , el estado global del sistema en el instante t, según

$$S = s_0 \cup s_1 \cup \dots \cup s_n$$

con s_i = estado local del proceso j

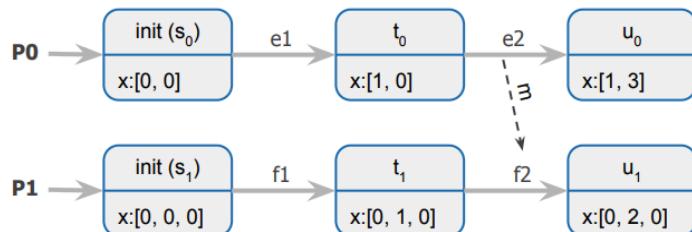
Sistema como máquina de estados

- Consiste en modelar el sistema como una serie de estados
- Un estado evoluciona al siguiente estado por la ocurrencia de un evento
- Asumiendo instrucciones determinísticas, el procesamiento de cualquier evento bajo el estado actual se puede reproducir

Es más simple de hacer cuando hay un único proceso. Ej:



Con múltiples procesos, es más difícil determinar el estado global S. Ej.:



Estado Globales Válidos: $S = s_0 \cup s_1 \cup s_2 \cup s_3$, etc

Estado Globales Inválidos: $S = u_1 \cup t_0$

Historia y Corte

Historia (o corrida)

Caracteriza al proceso P_i , considerándolo una máquina de estados, mediante la secuencia de todos los eventos procesados:

$$h_i = (e_0, e_1, e_2, \dots)$$

con e_j = todo evento aceptado por P_i

Corte

Unión del **subconjunto de historias** de todos los procesos del sistema **hasta cierto evento k** de cada proceso:

$$C = h_0 \cup \dots \cup h_n = (e_0, \dots, e_k) \cup \dots \cup (e'_0, \dots, e'_k)$$

Un **corte** es **consistente** si por cada evento que contiene, **también contiene a aquellos que 'ocurren antes' que dicho evento.**

Algoritmo de Chandy y Lamport

- **Algoritmo que permite obtener snapshots** de estados globales en sistemas distribuidos
- El objetivo del algoritmo es almacenar estados de un conjunto de procesos y estados de canales (snapshots) de forma que, aunque los estados no hayan ocurrido al mismo tiempo, **el estado global almacenado sea consistente**
- **Hipótesis**
 - Los procesos y los canales de comunicación no fallan
 - Canales son unidireccionales y poseen orden FIFO
 - Grafo fuertemente conexo (caminos de ida y vuelta definidos)
 - Cada proceso puede iniciar un snapshot en cualquier momento

- Pasos:
- 1) El proceso observador guarda su estado
 - 2) Este envía un marcador a todos los procesos
 - 3) Los procesos que reciben el marcador por primera vez envían su propio estado y propagan el marcador en el resto de sus mensajes
 - 4) Si un proceso recibe un mensaje sin un marcador que ya recibió, significa que ese mensaje fue creado antes y debe formar parte del snapshot

Comunicación reliable

Comunicación *Reliable* (o Confiable): si se garantiza integridad, validez y atomicidad en el *delivery* de mensajes.

Uno a uno

- Trivial si contamos con protocolos sobre TCP/IP y una red segura

Uno a Muchos

- El grupo debe proveer las 3 propiedades: bajo TCP/IP, atomicidad requiere atención especial
- Definir el orden entre mensajes garantizado: FIFO, causal, total, etc.

13. Data intensive applications

. Relacional vs NoSQL

↳ se adapta mejor a modelos con esquemas cambiantes

Transacional (OLTP) vs Analytics (OLAP)

- OLTP: orientado a grupos de reads/writes
- OLAP: orientado a análisis de conjunto de datos

	OLTP	OLAP
Patrón de Read	Pocos registros. Búsqueda por clave.	Agregación de muchos registros.
Patrón de Write	Acceso aleatorio. Registros pequeños.	Importaciones batch (ETLs) o Streams.
Uso Principal	Info maestra y transaccional p/usuarios	Exploración de datos. Análisis estadístico.
Datos	Instantánea de los datos en el momento actual. Tamaños de MBs-GBs.	Histórico de los datos. Tamaños de TBs-PBs.

Relacional

- Normalmente: un archivo de almacenamiento por tabla.
- Relaciones entre tablas por *foreign-keys*
- Lectura de toda la fila para retornar proyecciones.

Columnas

- Normalmente: Un archivos de almacenamiento por columna.
- Lectura de cada columna para retornar proyecciones.
- Grandes beneficios para compresión, lectura y agregaciones.

Sales

Columna	Contenido
date_id	20100101, 20100101, 20100102, 20100102, 20100102
product_id	100, 101, 100, 103, 103
quantity	2, 1, 3, 2, 10
price	200.00, 400.00, 300.00, 100.00, 500.00
discount	NULL, 50.00, NULL, NULL

Products

Columna	Contenido
product_id	100, 101, 103
name	Prod1, Prod2, Prod3
price	100.00, 450.00, 50.00

Cubos de información

- Normalmente: mantienen vistas materializadas con pre-cálculos estadísticos.
- Se crean grillas agrupadas por diferentes dimensiones.
- Operaciones como SUM, COUNT, MAX, MIN, AVG se consultan a estos cubos.

Ej.: Cubo: date_id x product_id, quantity

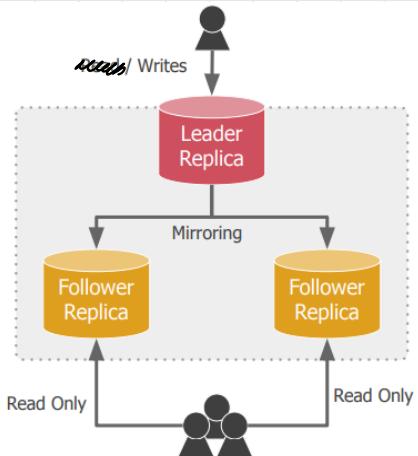
date_id	product_id	quantity	1	2	3	10	total	...
			100	101	103	total	...	
20100101		200.00	400.00	0.00	600.00		...	
20100102		300.00	0.00	600.00		900.00		...
	total	500.00	400.00	600.00		1500.00		...

Términos de Comparación

Característica	Almacenamiento Relacional	Almacenamiento Columnar	Cubos de Información
Unidad de almacenamiento	Un archivo por tabla	Un archivo por columna	Vistas materializadas preprocesadas
Estructura de datos	Tablas con relaciones vía foreign keys	Tablas con datos separados por columna	Grillas multidimensionales precalculadas
Acceso a datos (proyecciones)	Se lee toda la fila	Se leen solo las columnas necesarias	Acceso directo a agregados precalculados
Eficiencia de lectura	Menor eficiencia para consultas analíticas	Alta eficiencia para lectura y agregaciones	Muy alta, ya que consulta resultados precalculados
Ventajas en compresión	Limitadas	Alta compresión por tipo homogéneo	Depende del motor, pero suelen ser optimizados
Uso típico	OLTP (transaccional)	OLAP (analítico)	OLAP (analítico con énfasis en visualización)
Operaciones comunes	JOINS, SELECT, INSERT, UPDATE	SELECT con filtros, agregaciones	SUM, COUNT, MAX, MIN, AVG sobre dimensiones
Ejemplos	PostgreSQL, MySQL, Oracle	Apache Parquet, Amazon Redshift, ClickHouse	Microsoft SSAS, SAP BW, Apache Flink

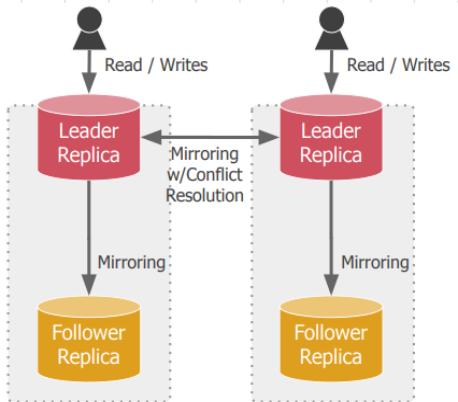
Replicación : LEADER BASED

- Una réplica se designa como **master** o **leader**.
- Otras réplicas se designan *mirrors*, *slaves* o *followers*.
- Sólo se aceptan **escrituras en el leader**.
- Tanto *leader* como *followers* aceptan lecturas.
- La replicación puede ser **síncrona** o **asíncrona**.
- Problemas de la replicación:
 - *Read your own writes, Monotonic reads*



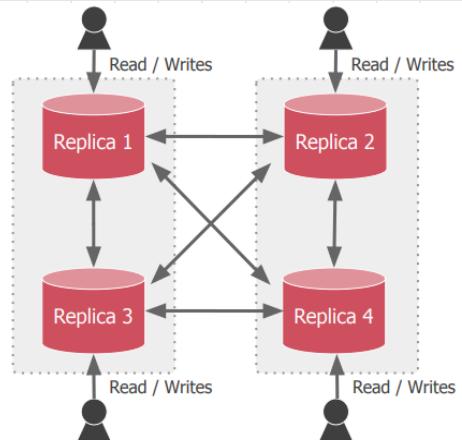
MUTI-LEADER BASED

- Modelo normal en escenarios de múltiples data-centers.
- Frente a caídas de un data-center, se puede promover al otro como líder global.
- Problemas de la replicación:
 - Idem a leader based más la posibilidad de conflictos por concurrencia.
- Otros inconvenientes:
 - manejo de triggers, claves incrementales, integridad de relaciones.



LEADERLESS

- Sistema de replicación totalmente distribuido.
- Las réplicas deben sincronizarse mutuamente.
- Se puede definir topologías para la sincronización:
 - anillo, jerárquicas, todos contra todos
- Los **conflictos son muy frecuentes** a menos que se particione.
- Otra alternativa es conseguir un consenso entre las réplicas para aplicar escrituras



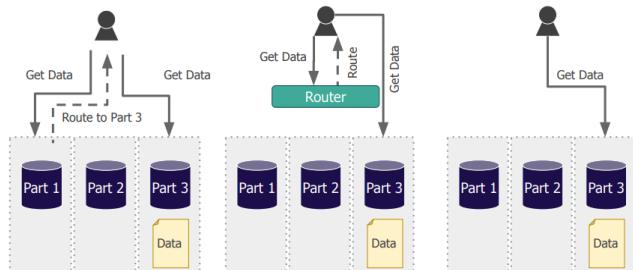
Particionamiento

+ Performance

+ Conflictos

+ Redundancia

Enrutamiento ↴



Horizontal

- La información se segregá por registros entre cada partición.
- El registro se encuentra en UNA partición a la vez.

date_id	product_id	quantity	price
20100101	100	2	200.00
20100101	101	1	400.00
20100102	100	3	300.00
20100102	103	2	100.00
20100102	103	10	500.00

date_id	product_id	quantity	price
20100101	100	2	200.00
20100101	101	1	400.00

date_id	product_id	quantity	price
20100102	100	3	300.00
20100102	103	2	100.00
20100102	103	10	500.00

Vertical

- La información se segregá respecto de sus atributos/dimensiones/campos entre cada partición.
- El registro se encuentra en TODAS las particiones.

date_id	product_id	qty	price	disc
20100101	100	2	200.00	NULL
20100101	101	1	400.00	50.00
20100102	100	3	300.00	NULL
20100102	103	2	100.00	NULL
20100102	103	10	500.00	NULL

date_id	product_id	qty	price	date_id	product_id	disc
				100	101	100
20100101	100	2	200.00	100	50.00	NULL
20100101	101	1	400.00	101	NULL	NULL
20100102	100	3	300.00	103	NULL	NULL
20100102	103	2	100.00	103	NULL	NULL
20100102	103	10	500.00	103	50.00	NULL

Distributed Shared Memory (DSM)

Objetivo

- Brindar la **ilusión** de una **memoria compartida centralizada**

Ventajas

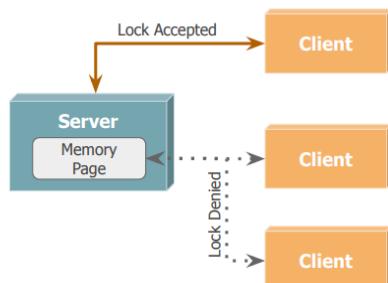
- Muy **intuitivo** para el desarrollo de sistemas distribuidos: Los algoritmos no distribuidos pueden ser traducidos fácilmente
- Información compartida entre nodos** sin que requieran conocerse

Desventajas

- Desalienta la distribución**, genera latencia, **cuello de botella** y **punto único de falla** (arquitectura cliente-servidor)

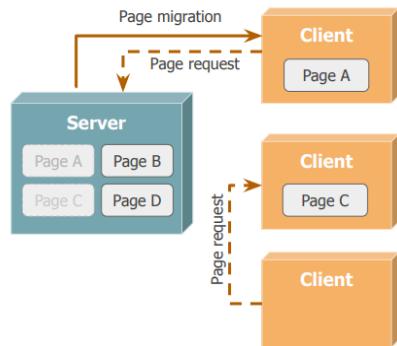
Enfoque naïve

- La información es almacenada en memoria por el servidor
- Los clientes acceden mediante requests a escribir o leer las páginas
- El servidor puede garantizar la consistencia muy fácilmente serializando los requests
- Muy baja performance** para las aplicaciones cliente



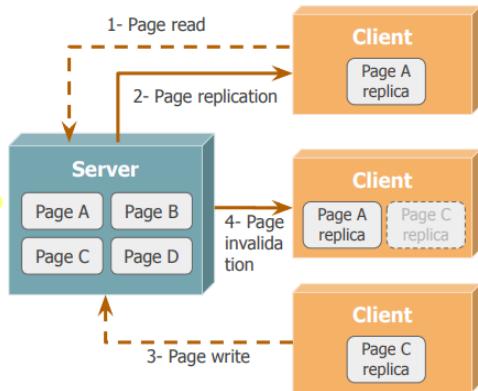
Memory pages (migración)

- La información es almacenada en memoria por el servidor y **delegada en los clientes**
- Los clientes pueden optimizar la localidad de acceso **pidiendo una memory page prestada**
- Otros clientes pueden pedir la misma página y quedar bloqueados, salvo que se permita una sub-delegación
- Garantiza consistencia. **No se accede concurrentemente a las páginas**



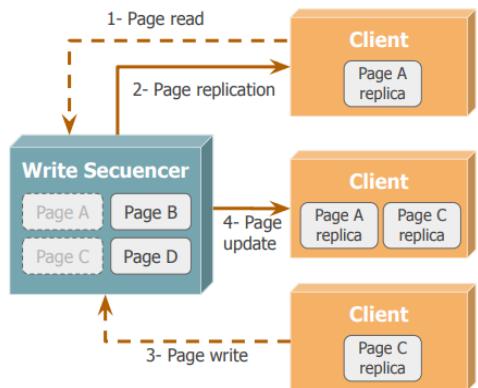
Memory pages (replicación solo lectura)

- Favorece escenarios con muchas lecturas y pocas escrituras
- Las escrituras son coordinadas por el servidor
- Las lecturas implican una replicación de la página en modo *read-only*
- El servidor invalida las réplicas frente a cambios



Memory pages (replicación lectura - escritura)

- El servidor mantiene las páginas de memoria hasta que los clientes las requieren
- Los clientes toman control total de las réplicas
- El servidor se transforma en un secuenciador de las operaciones
- El servidor aplica también los cambios ante caídas de los clientes



Distributed File Systems

- Compartir archivos en redes locales e intranets
- Poseer un esquema centralizado de información persistente
 - Control de Backups
 - Control de acceso y monitoreo
- Optimización de recursos por la concentración:
 - Discos de mayor capacidad permitían economizar
 - El costo de administración se reduce

Diseño

- **Transparencia a los clientes:**
 - Acceso: obtención de los recursos con credenciales usuales
 - Localización: operación de los archivos como si fueran locales
 - Movilidad: el movimiento interno de archivos no debe ser percibido
 - Performance y Escala: las optimizaciones no deben afectar al cliente
- **Concurrencia:** el acceso concurrente no debe requerir operaciones particulares al cliente
- **Heterogeneidad de Hardware:** adaptación automática a diferentes HWS
- **Tolerancia a Fallos:** capacidad de ocultar o minimizar los fallos (permitir operaciones como at-least-once o at-most-once)

NFS (Network file system)

- Requiere una nueva abstracción en el kernel: Virtual File System
- Arquitectura de cliente-servidor utilizando RPC sobre TCP o UDP
- Las aplicaciones utilizan el VFS para acceder a los archivos, lo que requiere una invocación remota
- Los servidores proveen operaciones idénticas a las requeridas por Posix:
 - Soporta ser montado como una unidad virtual

Hadoop DFS (HDFS)

- Sistema de archivos distribuido diseñado para utilizar hardware de bajo costo
- Implementación de Apache basada en el diseño de Google File System (GFS)
- No soporta POSIX por lo que se lo considera un Data Storage en lugar de FS
- Base del ecosistema de tecnologías Hadoop para procesamiento distribuido.

Diseño

- **Tolerancia a Fallos:** Los fallos de HW son normales. Es más económico adaptarse que defenderse
- **Volumen y Latencia:** Favorece las operaciones de *streaming* y los archivos volumétricos frente operaciones de usuarios finales de baja latencia
- **Portabilidad:** Preparado para ser utilizado en hardware de bajo costo. Utiliza TCP entre servidores y RPC con clientes
- **Performance:** Favorece operaciones de lectura. Política de *write-once-read-many*

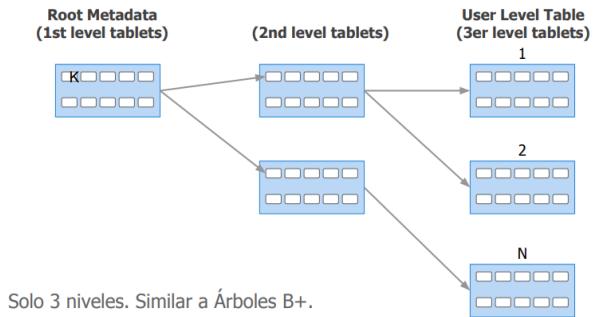
Big Table

- Solo almacena pares clave-datos
- Los datos son un conjunto de valores (columnas)
- No se almacenan en un orden definido

Tablets → Conjunto de filas consecutivas de acuerdo a la clave

date_id	product_id	quantity	price
20100101	100	2	200.00
20100101	101	1	400.00
20100102	100	3	300.00
20100102	103	2	100.00
20100102	103	10	500.00

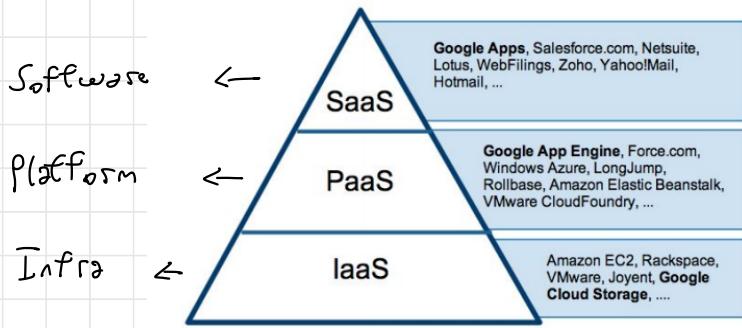
Key: 20100101, [Column: Product_id, Data: 100, Column: Quar
Data: 2, Column: price: 200.00]



16. Arquitecturas Cloud

Cloud

- Una nueva forma de ofrecer recursos de IT.



Ventajas

+ Accesibilidad

+ Velocidad

+ Escalabilidad

• Costos → pagar lo que usar

Pública vs Privada

Cloud vs Datacenter propio

Flex - ar - go vs Fijo

Desventajas

- Regulaciones
- Nulo control sobre el hardware
- Alta exposición de datos sensibles

PaaS

- Lambda, Cloud functions, etc

SOA (Service Oriented Architecture)

- Es un paradigma orientado al ámbito corporativo
- Las funcionalidades del sistema están organizadas como servicios independientes

Cada servicio expone una funcionalidad clara.

Service Registry y Discovery

- El cliente consulta a un service registry para obtener la lista de instancias
- Se suele utilizar para DNS

17. Tolerancia a Fallos

Introducción

- Estudia las necesidades de sistemas confiables

- Recuperación
- Redundancia
- Concurso

Fault (fallo) → condición no cumplida
error → efecto incorrecto en el sistema
Pausa → comportamiento incorrecto de cara al usuario

Fallo parcial

- Cuando un componente de un sistema tiene un error
- Puede generar una reacción en cadena

Clasificación

- Transientes : ocurren una vez y desaparecen
- Intermittentes : difíciles de diagnosticar
- Permanentes :

Condiciones

- Para definir que fallas tolerar hay que entender las condiciones

Condiciones del entorno

- Entorno físico del hardware (temperatura, resistencia a vibraciones y polvo, ubicación)
- Interferencia y ruido
- Clock drift

Condiciones operacionales

- Especificaciones, valores límites y tiempos de respuesta (ej. de sensores)
- Networking (ancho de banda, latencia)
- Protocolos soportados

Estrategias

- Fault removal : remover errores antes de que sucedan
- Fault forecasting : determinar la probabilidad de falla de un componente
- Fault prevention : evitar condiciones que lleven a errores
- Fault tolerance : manejar errores

Degradación suave

- El comportamiento empeora pero sigue siendo aceptable

Recuperación

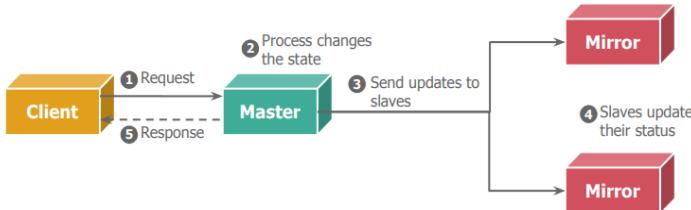
- Recuperarse de un error
 - Almacenamiento estable
 - Check pointing
 - Message logging
 - Consenso

Redundancia

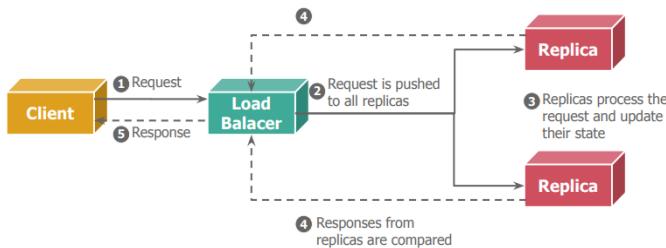
- Física (replicación)
- De información (valores)
- De tiempo (reintentos)

Replicación

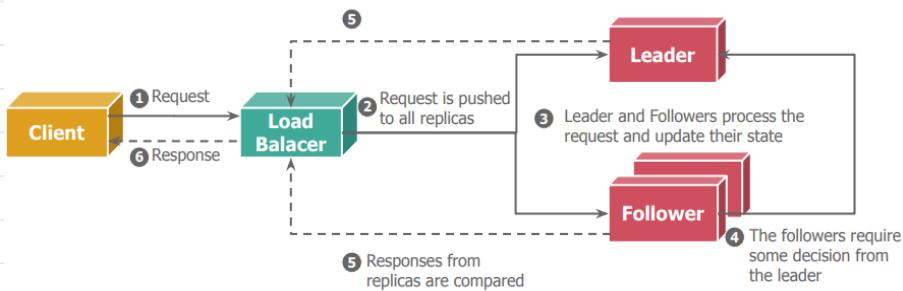
Pasiva: una primaria y varios de backup



Activa: múltiples replicas que ejecutan operaciones al mismo tiempo



Semi activa: solo el líder toma las decisiones



Resiliencia

• Mantener un nivel aceptable

- Errores de configuración u operacionales
- Desastre naturales (terremotos, inundaciones, huracanes)
- Factores políticos, económicos, sociales, del negocio
- Ataques maliciosos

Dependability (confianza): medida de la confianza en el sistema.

- **Availability (disponibilidad):** la probabilidad de que el sistema esté operando correctamente.
- **Reliability (fiabilidad):** capacidad del sistema para dar servicio correcto en forma continuada.
- **Durability:** probabilidad que un dato persistido se pueda recuperar
- **Safety (seguridad):** en presencia de fallos no ocurre nada catastrófico.
- **Maintainability (mantenibilidad):** la cantidad de tiempo que se requiere para recuperar el sistema (p. ej. repararlo o actualizarlo).

Depende de :

- Presupuesto
- Necesidades (performance y escalabilidad)
- Componentes

Maintainability

- Usar imágenes docker
- Ambiente de testing
- Trazabilidad
- Health checks
- Desacoplamiento

Safety

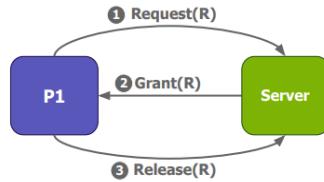
- DRP (disaster recovery process) definido
- SLAs
- Testing

Exclusión Mutua Distribuida

- Algoritmo para obtener acceso exclusivo a un recurso
 - Safety : solo un proceso a la vez
 - Liveness : sin espera eterna
 - Fairness : equilibrado

Centralizado

- Alguno de los procesos en el sistema es elegido como el coordinador de la sección crítica
- Se sabe de antemano la forma en la que el recurso será identificado
 - Ej. string utilizado como ID
- Si el recurso se encuentra tomado, los requests son encolados (FIFO)
 - Procesos esperan a que el server les de acceso a la sección crítica
 - Acceso a sección crítica *time-bounded*



+ Orden

- Único punto de fallo

+ Fácil de implementar

- Cuello de botella

+ Menos conexiones

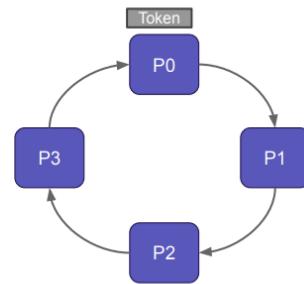
,

+ Solo necesita conocer al servidor

,

Token ring

- Anillo es construido ordenando a los procesos involucrados por algún atributo (MAC, IP, PID)
- Inicialización
 - Proceso 0 crea un token
- Token circula alrededor del anillo
 - De P_i a $P_{(i+1) \bmod N}$
- Cuando un proceso recibe el token
 - Proceso chequea si requiere sección crítica
 - En caso negativo, pasa el token a su vecino
 - En caso afirmativo, accede al recurso reteniendo el token



+ Facil

- Implementación de protocolo

+ No hay coordinadores

- Cada impl. se regenera anillo

+ Es garantizado

- Token se puede perder

Ricart & Agrawala

- Utiliza multicaso y relojes lógicos

Cuando un proceso intenta acceder a la sección crítica

- Crea un **request** con un **timestamp** asociado al proceso, un **ID** y el nombre del recurso ($<T, P_i, R>$)
- Envía el **request** a todos los procesos en el **grupo**
- Espera hasta que todos los procesos le den permiso de ingresar a la sección crítica (**OK response**)
- **Entre a la sección crítica**

Cuando un proceso recibe un request

- Si el **receiver** no está interesado, envía un **OK** al **Sender**
- Si el **receiver** posee la sección crítica, **no responde** al proceso **Sender** y encola el mensaje
- Si el **receiver** envió también un **request** para acceder a la sección crítica
 - Se deben **comparar** los **timestamp** del mensaje enviado y recibido
 - Aquel que tenga un **timestamp** menor **gana**
 - Si el **receiver** es el perdedor, envía un **OK** al **Sender**
 - Si el **receiver** es el ganador, encola **request**

+ No hay coordinador

- Muchas conexiones

- Muchos mensajes

- No se diferencia proceso caído de proceso en SC

19. Algoritmos de Consenso

Elección de líder

Objetivo

- Elegir a un proceso en un grupo para que desempeñe un rol particular
- Permitir reelecciones en caso de que proceso líder decida darse de baja
- Permitir reelecciones en caso de que proceso líder se encuentre caído

Características

- Cualquier proceso puede comenzar una nueva elección de líder
- En ningún momento puede haber más de un líder
- El resultado de la elección de un nuevo líder debe ser única y repetible

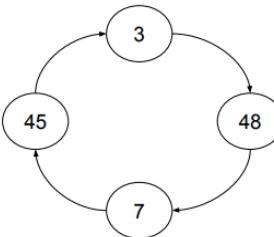
Propiedades

- Cada proceso debe tener un identificador único
- Todos los procesos poseen un array que indica el estado del algoritmo de elección de Líder
- Estados posibles: Identificador (P), indefinido (@)
 - Indefinido (@) es el estado inicial que posee un proceso P , al comenzar a participar del algoritmo de elección de líder
- Safety
 - Un proceso participante P_i posee el estado $elected_i = P$ o $elected_i = @$
- Liveness
 - Todos los procesos participan de la elección de líder y bien terminan con un estado $elected_i \neq @$ o comienzan una nueva elección de líder

Algoritmo del anillo

Condiciones Iniciales

- Cada proceso se comunica solamente con su vecino
- Mensaje son enviados siempre en la misma dirección (por ejemplo, sentido horario)
- Al comienzo del algoritmo, todos los procesos son marcados como *no participantes*



1 - Inicio de la elección

- Algún proceso P_j se marca como *participando* y envía un mensaje en sentido horario indicando que él es el líder.

2 - Proceso P_j recibe mensaje de elección de líder y...

- Si se encuentra en estado *no participando*:
 - Cambia su estado a *participando*.
 - Compara el identificador del líder con el suyo, lo reemplaza en caso de ser mayor y reenvía el mensaje al nodo siguiente.
- Si se encuentra en estado *participando*:
 - Si el ID recibido es menor al suyo, no reenvía el mensaje.
 - Si el ID recibido es mayor al suyo, reenvía el mensaje.
 - Si el ID recibido es igual al suyo, entonces **es el líder!!**

3 - Proceso P_j reconoce que es el líder y...

- Se setea como *no participando*.
- Envía un mensaje de **líder elegido**.

4 - Proceso P_j recibe un mensaje de líder elegido y...

- Cambia su estado a *no participando*.
- Setea la variable elegido con el identificador del mensaje recibido.
- Si el identificador recibido es distinto al suyo, retransmite el mensaje.

Algoritmo Bully

- Todas las procesos se hablan y se conocen
- Mensajes : Election, Answer y Coordinator
 - Proceso con mayor ID puede **identificarse como leader** y mandar un **Coordinator Message** a todos los procesos del sistema
 - Si un proceso detecta que el **Líder está caído**, envía **Election Messages** a procesos que tengan **un ID mayor al suyo**
 - Si un proceso recibe un **Election Message**, responde con un **Answer Message** y **comienza una nueva elección**
 - Si un proceso recibe un **Coordinator Message**, elige al proceso que envió el mensaje como líder
 - Si un proceso que comenzó una elección no recibe **Answer Messages** después de transcurrido tiempo T , **el proceso se autopropone como líder**
 - Si **un proceso caído vuelve a la vida**, **comienza una nueva elección de líder**
 - Si este proceso es el que posee mayor ID, será elegido como el nuevo líder
 - Por esta razón este algoritmo es llamado **Bully Algorithm**

Consenso

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

Problema complejo, requiere acotar variables:

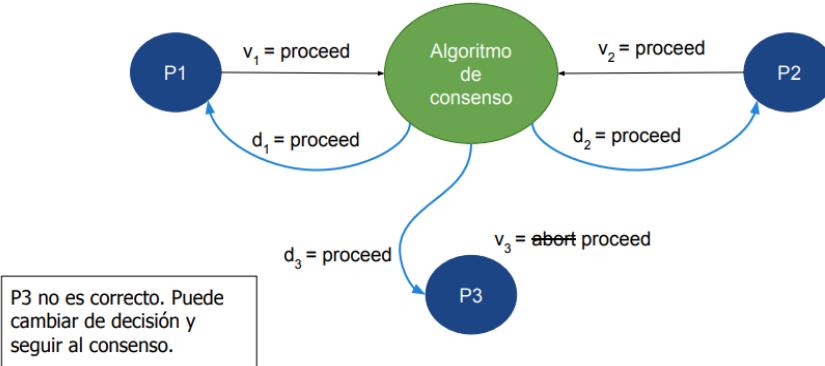
- Canales de comunicación son *reliables*
- Todos los procesos pueden comunicarse entre sí
- Única falla a considerar es la caída de un proceso
- Caída de un proceso no puede ocasionar la caída de otro

Propiedades necesarias de los algoritmos de consenso:

- *Agreement* : el valor de la variable es igual para todos
- *Integrity* : si todos proponen el mismo valor, entonces queda así
- *Termination* : eventualmente todos selean su variable

Definición

- Conjuntos de P_i ($i = 1 \dots N$) procesos desean llegar a un acuerdo
- Cada proceso comienza en el estado **undecided**
- Cada proceso posee una **decision variable** d_i ($i = 1 \dots N$)
- Cada proceso propone un valor v_i
- Procesos se comunican entre sí a través de mensajes
- Luego de haber recibido mensajes de otros procesos, proceso P_i setea su **decision variable** d_i y cambia su estado a **decided**



Algoritmo sincronico

```
Init
    Values(i, 0) = {}, Values(i, 1) = { v }

For each round r, 1 < r <= f+1
    broadcast Values(i, r) - Values(i, r-1)
    Values(i, r+1) = Values(i, r)
    While round r still open
        receive Values from j in Values(j, r)
        Values(i, r+1) = Values(i, r+1) U Values(j, r)

After f+1 rounds
    decide d = aggregation over Values(i, f+1)
```

Paxos

Objetivo

- Consensuar un valor aunque múltiples procesos realicen diferentes propuestas

Características

- Tolerante a Fallos
 - Algoritmo progresiva siempre que haya una mayoría de procesos vivos
 - $N \geq 2f + 1$ (Fórmula de Quorum)
- Posible Rechazar Propuestas
 - Pedido de un cliente puede ser rechazado
 - Cliente puede reintentar una propuesta las veces que desee



- Cliente intenta setear/modificar un valor R (key/value)
 - Si puede hacerlo, recibe un OK
 - En caso contrario, debe volver a intentar
- Paxos asegura orden consistente en un cluster
 - Eventos realizados por clientes son almacenados de forma incremental, por ID

Proposer

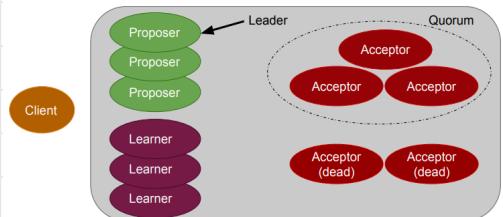
- Reciben requests de clientes y comienzan el protocolo
- Leader debe ser elegido para evitar starvation

Acceptor

- Reciben prepare/propose de los Proposers
- Mantienen el estado del protocolo en almacenamiento estable
- Quorum: Existe cuando la mayoría de Acceptors se encuentran vivos

Learner

- Cuando el protocolo llega a un acuerdo, Learner ejecuta el request y envia respuesta al cliente



Pasos:

1)

- Client realiza un Request

2)

- **Proposer:** Crea una propuesta #N donde N es mayor a cualquier propuesta realizada previamente por el Proposer
- Enviar propuesta a Acceptors esperando obtener Quorum (mensaje llegue a la mayoría de Acceptors)

3)

- Si ID recibido del Proposer es mayor al último recibido, Acceptors prometen rechazar cualquier Requests con ID < N
- Envío de Promesa a Proposer
- Acceptors no contestan si llega una propuesta que no cumpla que $N > N'$

4)

- **Proposer:** Si recibe Promesas de la mayoría:
 - Rechaza todos los requests con un ID < N
 - Envía Propose con el N recibido por los acceptors y un valor v: Propose(N,v)

5)

- **Acceptor:** Si la promesa aún es mantenida, anunciar el nuevo valor v
 - Enviar **accepts** a todos los Learners y al Proposer que envió el request inicial
 - No enviar **accepts** si un ID superior a N fue recibido

6)

- **Learner:** Responde al cliente y se toman acciones respecto del valor acordado

Generales Bizantinos

Definición

- Tres o más **generales** deben decidir si atacan o se retiran
- Un **comandante** envía la orden de ataque/retirada
- **Generales** pueden ser *traicioneros*
 - Le indica a los otros generales que deben atacar si el comandante le indicó que se retiren y viceversa
- **Comandante** puede ser *traicionero*
 - Envía diferentes órdenes a diferentes generales
- **Comandante / General** *traicionero* viene a emular a un proceso que posee fallas

Requerimientos

- *Agreement*
 - El valor de la variable **decided** es el mismo en todos los procesos activos
 - Si P_i y P_j son procesos activos entonces $d_i = d_j$ cuando su estado es **decided**
- *Integrity*
 - Si el comandante *no es traicionero*, todos los procesos deben setear su **decision variable** al valor enviado por el comandante
- *Termination*
 - Eventualmente todos los procesos activos setean su **decision variable**

Fórmula de Quorum

- $N \geq 3f + 1$

20. Intro a Sistemas de tiempo real

Sistemas de tiempo real

- Son aquellos sistemas cuya evolución se especifica en términos de requerimientos temporales requeridos por el entorno
- La correctitud del sistema depende de que entregue respuestas correctas y en tiempo correcto.
- Un sistema es RT si tiene al menos un servicio RT
- Ejemplos de sistemas RT:
 - Electrodomésticos digitales, medidores de señales (presión, pulsaciones, ritmo cardíaco, etc.) mediciones por sensores, control de automóviles, control en aeronaves, marcapasos, etc.

Hard RT:

- se debe evitar todo fallo relacionado con el tiempo de *delivery*
- perder un *deadline* o plazo de respuesta es un fallo total

Soft RT:

- fallos relacionados con el tiempo de *delivery* pueden ser admitidos ocasionalmente*.
- la utilidad de un resultado disminuye tras el *deadline*

- RT implica
PREVISIBILIDAD

↓
Scheduling correcto

Comunicación

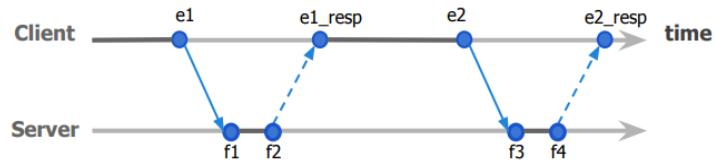
- TCP / IP : no asegura fiabilidad → sincronización
- Serial : si asegura
- Ethernet : para capa física

RT : Tolerancia a fallos

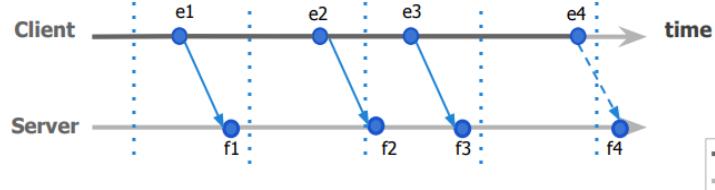
- Los sistemas ahora deben ser tolerantes a fallos de tiempo
- Distintos tipos de estrategias:
 - **Soft RT:**
 - Ej.: Sistemas web de gran escala
 - El 99% de los requests deben responderse en 2 seg. El 1% restante se deben responder en 10 seg. Se admite 1 outliers cada 1M reqs.
 - **Hard RT:**
 - Ej.: Misión crítica
 - El 100% de los requests debe resolverse en 1 seg. Frente a errores, se asume un fallo catastrófico y se recomienda hard reset.
 - Muy importante revisar el factor de *Maintainability*.

Pasadigmas de trabajo

Event - Triggered



Time - Triggered



— Active
— Idle

Sistemas de control

- Controlar un sistema de forma manual o automática

Control

capacidad de actuar para garantizar el comportamiento de un suceso.

Variable controlada

cantidad o condición que se mide o controla.
Salida del sistema.

Perturbación

señal que tiende a afectar negativamente el valor de la salida del sistema.

Controlador (referencia)

sistema encargado de determinar la actuación para conseguir cierto objetivo del proceso

Proceso

toda sucesión de operaciones que se desea controlar.

Variable manipulada

cantidad o condición que se modifica para afectar el valor de la variable controlada.

Planta

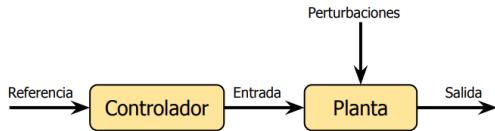
cualquier sistema físico que se desea controlar.

Actuador

elemento físico de la planta que frente a señales del controlador opera sobre el proceso

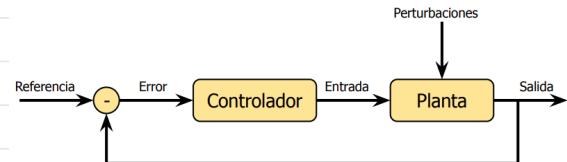
Lazo abierto

- la salida del sistema no afecta la acción de control.



Lazo cerrado

- se utiliza información sobre el estado del sistema para actuar sobre el sistema y llevar la salida del mismo a los valores deseados.



Programación

- Event o Time Triggered
- Scheduling
- Protocols