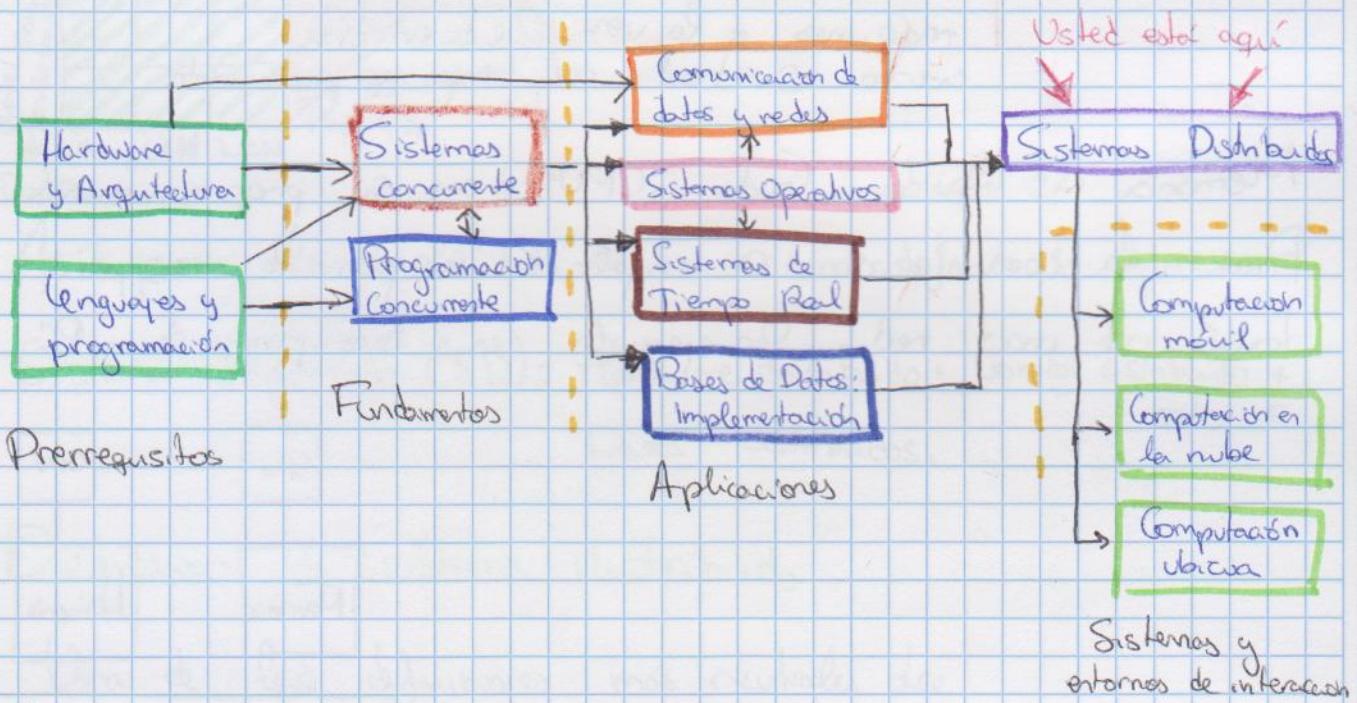


1. Introducción

"Outline"

- 1 Disciplinas relacionadas
- 2 Clasificación y definición
- 3 Características y objetivos
- 4 Paradigmas de aplicaciones distribuidas
- 5 Nuevas características y paradigmas
- 6 Modelos de referencia.

Disciplinas relacionadas



Clasificación de Sistemas Distribuidos

Hardware

Es un rango con 2 extremos según el acoplamiento:

Débilmente acopladas

Fuertemente acopladas

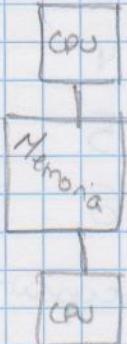
Estos sistemas podrían hacer comunicaciones de milisegundos. A mayor acoplamiento, mejor tiempo de comunicación.

Software

Se desarrollan principalmente tres modelos:

Memoria compartida: Aunque físicamente las memorias estén distribuidas, en el software se da la memoria en común que comparten ambos CPUs.

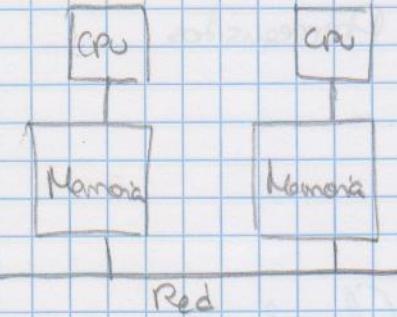
Ventajas | Ambas CPU pueden acceder a la totalidad de la memoria + rápido.



Desventajas | Se añade otro problema a la hora de escribir un dato ambos máquinas a la vez. Puede añadirse cierto Overhead al tiempo de procesamiento.

Memoria distribuida Cada CPU tiene su propia memoria.

Para la comunicación, se empleará un paso de mensajes a través de una red. Un ejemplo es hacer ping de un PC a otro.



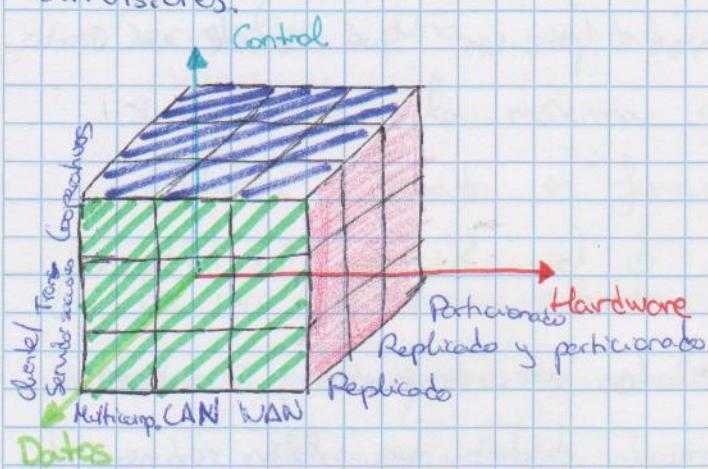
Modelo genérico para tipos de Sistemas Distribuidos

Este modelo para buscar los tipos genera una clasificación en base a tres dimensiones:

El software, que se divide en la lógica / control y los datos.

El propio hardware.

Véase que del modelo más básico se podría desarrrollar más servicios. Es decir, el sistema distribuido más complejo siempre será el más complejo en las tres dimensiones.



Así pues, la complejidad de un sistema definido se define como:

Sistema Distribuido (SD) = Hardware distribuido + Control distribuido + Datos distribuidos.

Definición de sistema distribuido

Una de las definiciones más neutrales dice:

"Conjunto de computadoras autónomas, pero enlazadas por una red y con software diseñado para proteger y facilitar una computación integrada."

Otra definición, basada en la parte hardware:

"Sistema en el que los componentes localizados en computadoras, conectados en red, comunican y coordinan sus acciones vía cable mediante el paso de mensajes"

Otra definición de este dice:

"Colección de computadoras independientes que se presentan ante los usuarios como un único sistema coherente"

Desde el punto de vista hardware, las computadoras son independientes físicamente; pero el software debe conseguir que los usuarios piensen que existe un único sistema.

Características y objetivos de los Sistemas Distribuidos

1. Compartición de recursos: Los recursos pueden ser hardware o software. En sistemas distribuidos, los recursos son gestionados por programas que ofrecen una interfaz de comunicación y se ejecutan en diferentes dispositivos de cómputo.

Un ejemplo sería el cloud computing, en base a la "elasticidad"; es decir, que se adaptan a las necesidades de cómputo, y dar más o menos recursos, sin límites.

Los políticas y métodos de gestión de recursos presentan requisitos comunes:

- Mismo esquema de denominación (URI) para acceder a cualquier lugar.

- Asociación de nombres de recursos a direcciones de comunicación.

- Coordinación de accesos concurrentes para asegurar la consistencia.

Tendencia a un control descentralizado, cada día más tendiente.
Por ej.: Groupware, que da soporte a grupos de usuarios que trabajan de forma cooperativa compartiendo objetos de datos entre aplicaciones.

→ Serverless: Una entidad aún menor que un microservicio, que tiene la máxima granularidad posible. Es un sistema basado en funciones.

2. Es un sistema abierto, es decir, que determine si el sistema puede ser personalizado y ampliado de varias formas, haciendo referencia tanto al hardware como software.

Por ejemplo, Unix es un buen sistema abierto. Esta toda la funcionalidad aquí, siempre y cuando se tenga permiso dentro del sistema.

la utilización de protocolos y estándares de comunicación

(Inter Process Communication)

Se recomienda hacer uso de sistemas abiertos para el sistema distribuido.

3. Concurrencia Inherente en SD. Surge naturalmente por:

- la actividad independiente de cada usuario.
- la independencia entre recursos hardware
- la localización de distintos procesos en diferentes computadores.

4. Escalabilidad El software de sistema y aplicaciones no debería cambiar cuando el sistema aumenta de tamaño.

El trabajo que implica el procesamiento de una única petición para acceder a un recurso compartido debe ser lo más independiente del tamaño de la red.

El principal desafío es diseñar el software del sistema distribuido de forma que permanezca eficiente y efectivo. Algunas técnicas son:

Uso de cachés

Replicación de datos

Despliegue de servicios

Sistemas abiertos: Estos permiten introducir y reemplazar los servicios, independientemente del proveedor.

5. Tolerancia a fallos El objetivo es que pueda seguir funcionando pese a fallos externos, o incluso internos. Se basa en dos propuestas:

Recuperación software:

Redundancia hardware: El SD proporciona un alto nivel de disponibilidad (medida del tiempo que un recurso esté disponible para su uso)

Las redes, hasta hace poco, no eran redundantes y por tanto se buscaba más bien un diseño seguro. Ahora la tecnología wireless permite crear redes ad hoc, como los MANET. Para ello se basan en soluciones seamless (sin cortes), es decir, usar las tecnologías colaborativamente sin que el usuario tenga por qué saberlo por sí mismo.

Edge computing?
Fog computing?
Servers mirror

6. Transparencia La idea es ver el sistema distribuido como un todo. Es el principal objetivo en el diseño del software de un SD. Algunas formas de transparencia básicas son:

Forma

Acceso Las operaciones son las mismas en entidades locales y remotas.

Localización Las entidades pueden ser accedidas sin importar su localización.

Concurrencia Las operaciones concurrentes sobre la misma entidad no interfieren entre si.

Y avanzadas:

Replicación La existencia de varias instancias de una misma entidad sin conocimiento por parte del programa o del usuario.

(Self-Healing, Self-Configuring, Self-*)

Fallo: Se ocultan fallos hardware y software

Migración Se mueven las entidades en el sistema sin afectar a programas o usuarios.

Rendimiento: Se reconfigura el sistema según la carga.

Escalabilidad: La estructura del sistema no cambia cuando el sistema o las aplicaciones se amplían.

Paradigmas de Aplicaciones Distribuidas

Aplicaciones paralelas de alto rendimiento:

Sus características son:

Decrevenen el tiempo de respuesta.

Da ventaja de escalabilidad a los sistemas distribuidos frente a los sistemas multiprocesador o multihilo.

Se pueden clasificar en cuanto a la granularidad del paralelismo:

Grano grueso (Coarse) Reduce la comunicación pero aumenta el círculo

Grano medio (Medium)

Grano fino (Fine) Reduce el tráfico, pero al haber tantas entidades, hay mucha comunicación (overhead)

En resumen: Cuantas más entidades, menos círculo pero más comunicación.

Aplicaciones tolerantes a fallos Tratan de mantener la fiabilidad y la disponibilidad.

Estos sistemas distribuidos son potencialmente seguros debido a la propiedad de fallo parcial o cuento más débilmente expuesto, más fiabilidad. Además, esta seguridad se refuerza mediante la replicación de funciones / servicios o datos de las aplicaciones en varios nodos.

Aplicaciones con especialización funcional, donde cada aplicación se compone de varios servicios. Una forma natural de diseñar estas aplicaciones es mediante un sistema distribuido con implementaciones alternativas, donde cada servicio puede utilizar uno o más procesadores dedicados. Estos servicios pueden comunicarse entre ellos.

Esto proporciona un alto rendimiento y seguridad, y además las aplicaciones son "fácilmente" escalables.

Aplicaciones inherentemente distribuidas:

Características / Objetivos emergentes en nuevos sistemas

Computadoras portátiles, smartphones, wearables y LAN inalámbricas

Son sistemas distribuidos con clientes (e incluso servicios) móviles. Además de las propiedades anteriores, se suman estos aspectos adicionales.

Es un campo de investigación y desarrollo muy activo. Algunos
áreas de trabajo son:

Redes móviles

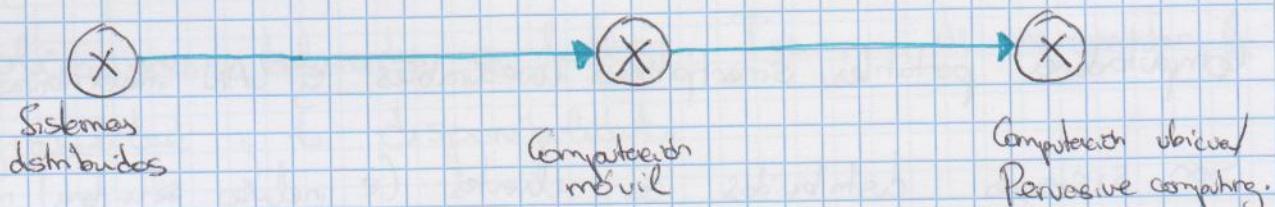
Acceso móvil a la información

Soporte a aplicaciones adaptativas

Técnicas de ahorro de energía

Sensibilidad al contexto / Context awareness.

Evolución de la Computación Distribuida



Espacios inteligentes

Invisibilidad

Escalabilidad localizada /

Locate Scalability

Condicionamiento fijo /

Uneven conditioning

Modelo de referencia para sistemas distribuidos

Un modelo de referencia es un framework para la comprensión de los mecanismos más relevantes entre las entidades de un determinado entorno, independiente de estandares, tecnologías e implementaciones particulares.

El objetivo principal es poder desarrollar arquitecturas específicas

Arquitectura
por capas

2. Comunicación y sincronización de paso de mensajes.

Paso de mensaje

Formar (marshalling) Poner una colección de datos en un formato adecuado para transmitirlos en un mensaje.

Se basa en:

• Poner en plano / To flat las estructuras de datos en estructuras básicas. Es tarea de los Stab.

• Traducir esas estructuras básicas en una representación de datos estandar como XDR

Estas operaciones se pueden generar automáticamente, sobre todo hacer la inversa (o directa) de XDR.

Canal de comunicación. Es la abstracción de una comunicación física que proporciona un camino de comunicación entre procesos y sistemas.

Las notaciones difieren en:

Ambito y denominación de los canales (global, asociado a un conjunto)

Uso

Sincronía (Síncrona / Bloqueante, Síncrona / No bloqueante con buffer, o Asíncrona con buffer limitado)

Operación de comunicación no bloqueante. Su ejecución nunca retrasa el proceso que la invoca.

La mayoría de propuestas de notaciones son equivalentes ya

que con programa se puede hacer con una notación u otra, aunque se deba añadir más o menos código.

La notación más común consta de send, receive y empty.

Hay varios tipos de comunicaciones

Síncrona Las primitivas son bloqueantes

Asíncrona La recepción es bloqueante, el envío no.

Asíncrona con buffer Junto la recepción es bloqueante. Cuando se llena el buffer, se envía send.

Dentro de los mensajes Debe ser conocido por el emisor e independiente de la localización. Varios tipos:

Directo > Proceso Comunicación punto a punto

Indirecto | Enlace Punto a punto

Puerto Muchos a uno

Buzón Muchos a muchos Alguien puede meter una carta, alguien con clave puede leer una carta. Destinatario

Difusión Muchos a muchos Broadcast

Selección Muchos a muchos Multicast

Protocolos de comunicación de grupos

Uso	Tolerancia a fallos en servicios replicados Localización de objetos en servicios distribuidos Mejor rendimiento con servicios replicados Actualización múltiple notificando eventos a varios procesos a la vez.
-----	--

Propiedades	Atomicidad El mensaje es recibido por todos o ninguno Ordenación Ejecución de operaciones en el mismo orden Fidabilidad Los datos que envía el cliente serán recibidos por el servidor sin errores y en el mismo orden el que fueron enviados.
-------------	--

Protocolo petición / respuesta

Es una comunicación típicamente sincrónica, segura y fiable.

Hay varias alternativas para implementar tal protocolo:

Primitivas de comunicación (send/receive). Tres tipos inconvenientes:

Sobreexige: Se usan más canales

Correspondencia entre send y receive

Garantía de reparto de mensajes si los servicios en red no fallan.

Operaciones de comunicación combinan aspectos de montaje y desmontaje de mensajes sincrónico.

Tres tipos primitivas Do Operation, Send Reply y Get Request.

Realmente hay 3 protocolos petición - respuesta con diferentes semánticas en presencia de fallos

R = Request, Re = Reply, A = Acknowledgement

Nombre	Mensaje enviado por	Cliente	→ Servidor	→ Cliente
R	Petición			
RR	Petición		Resposta	
RRA	Petición	Resposta		Reconocimiento

A la hora de realizar un sistema con este tipo de protocolo, deberíamos tener en cuenta:

- Añadir timeouts para evitar esperas innecesarias
- Filtrar peticiones repetidas para aquellos casos en los que una misma petición haya superado el timeout y se repita, pero esté llegando la respuesta del anterior intento. El formato es:
 - Operaciones idempotentes permiten re ejecutar proporcionando los resultados
 - Historia de peticiones para aplicaciones o peticiones "no idempotentes". Para cada petición, guarda la respuesta. Si lo vuelven a pedir, en vez de procesarlo, le devolverán lo mismo.
- Eliminar, en caso de R-ReA, las peticiones resueltas. En otros casos, como R-Re, podría llevar el reconocimiento en el siguiente R-Re.

Todo ello se denomina modelo de gestión de fallos. Así, las garantías de envío sobre errores con la operación

Garantías de envío			Semántica de mensajes
R reintentar peticiones	Filtrar duplicados	Repetir procedimiento / Retransmitir respuesta	No aplicable
	No aplicable		Si más
		Repetir procedimiento	Al menos una vez
		Retransmitir respuesta	Exactamente una vez

RPC (llamada o Procedimiento Remoto).

En el modelo cliente/servidor los servicios proporcionan varias operaciones, basándose en una comunicación con protocolo petición / respuesta.

Los mecanismos de RPC integran esta organización con lenguajes de programación procedurales convencionales. Se diseña así un método cuya llamada parece^a un procedimiento local, pero en realidad se ejecuta remotamente. Así, el servidor se ve como un módulo con una interfaz que expone operaciones y con un tiempo de vida restringido.

Existe así una biblioteca de soporte a servicios para usar cuentas caché, como localizar el servidor, uso de cachés para mayor rendimiento, diferencias entre procedimientos locales y remotos... .

La semántica se compone de:

Parámetros de Entrada/Salida (dando así una comunicación bidireccional).

Usan sólo variables locales. No hay variables globales para servidor y cliente). Así, por ejemplo, los arrays se convierten en listas enlazadas que se usan para reorganizar en la otra punta los otros arrays originales. Como no existen los punteros, se usan diconadores.

- Referencias opacas: Los servidores pueden devolver variables que no necesitan ser interpretadas en el entorno del cliente. Ejemplos: Cookies, Datos de autenticación ...

Algunas questões de diseño:

Clases de sistemas / middlewares: Son mecanismos integrados en el lenguaje de programación que permite que algunos requisitos puedan tratarse con construcciones del lenguaje o de propósito general.

Características del IDL (Lenguaje de Definición de Interfaces):

Deben permitir especificar:

Nombres

Procedimientos

Tipo de parámetros

Dirección Entrada, Salida o Ambos (E/S)

Transparencia: Hay que manejar errores ya que RPC es más vulnerable y toma más tiempo que una red local.

Por ello no debería ser transparente al programador, si bien debería ocultar detalles de bajo nivel al peso de mensajes, no debiera ocultar los fallos o retrasos.

Manejo de excepciones: Hay que notificar los errores de la implementación de RPC es:

Cliente incluye maneja las funciones son del tipo ?-1.

Servidor cliente Correto los procedimientos locales en los para la comunicación.

Servidor servidor Incluye manejo del servidor, extrae los argumentos para tratarse en servidor. Luego de recibir el resultado lo comunica con

- Referencias opacas: los servidores pueden devolver variables que no necesitan ser interpretadas en el entorno del cliente. Ejemplos: Cookies, Datos de autenticación...

Algunas questões de diseño:

Clases de sistemas / middlewares: Son mecanismos integrados en el lenguaje de programación que permite que algunos requisitos puedan tratarse con construcciones del lenguaje o de propósito general.

Características del IDL (Lenguaje de Definición de Interfaces):

Deben permitir especificar:

Nombres

Procedimientos

Tipo de parámetros

Dirección Entrada, Salida o Ambos (E/S)

Transparencia: Hay que manejar errores ya que RPC es más vulnerable y toma más tiempo que una red local.

Por ello no debería ser transparente al programador, si bien debería ocultar detalles de bajo nivel al paso de mensajes, no debería ocultar los fallos o retrasos.

Manejo de excepciones: Hay que notificar los errores de la implementación de RPC es:

Cliente incluye maneja las funciones son del tipo -1.

Stub cliente Correto los procedimientos locales en los para la comunicación.

Stub servidor Incluye maneja del servidor, extrae los argumentos para tratarse en servidor. Luego devuelve el resultado a comunicacion.

Server No hace el mapeo, pero si los procedimientos pedidos por el cliente.

El procesamiento de la interfaz integra el mecanismo RPC con esos programas, extrayendo así argumentos y resultados. Se compila así una especificación escrita en el IDL que genera cabeceras, plantillas y stubs.

El módulo de comunicaciones implementa el protocolo petición - respuesta.

Servicio de ligadura (binding) Es un mecanismo para localizar el servidor, y asociar un nombre a un identificador de comunicación.

El mensaje de la petición se redirige a un puerto concreto, y se evalúa cada vez que el cliente lo requiera ya que el servidor puede ser relocalizado.

Es un servicio del cual dependen otros, debe ser totalmente fiables, ya que los servidores exportan (registan) sus servicios y los clientes importan (crean).

El puerto de binding suele ser el 111.
Las operaciones son del servicio de ligadura son

Registrar <Nombre-Servicio>, <Puerto>, <Version>

Rehivar <Nombre-Servicio>, <Puerto>, <Version>

Buscar (<Nombre-Servicio>, <Version>) : <Puerto>

Para localizar el binder, podemos

Usar una dirección conocida. Si el binder se relocaliza, el cliente y el server deben recomunicarse.

Usar el del Sistema Operativo, que proporcionará la información en tiempo de ejecución (mediante, por ejemplo, variables de entorno)

Usar un broadcast para que el binder pueda responder con la dirección en cuanto recibe el mensaje.

RPCs Asíncronas

Requisitos comunes

El cliente envía muchas peticiones al servidor.
No se necesita una respuesta a cada petición.

Ventajas

El servidor puede planificar operaciones + rápido
El cliente trabaja en paralelo
Se facilita el cálculo de peticiones paralelas
en el caso de varios servidores.

¿Y cómo lo optimizamos?

Con varias peticiones en una sola comunicación. De esta manera se almacenan mensajes hasta que se cumpla un plazo de tiempo o se realice una petición que requiera una respuesta.

Otra manera es haciendo que si la respuesta no se necesita bloquear, hacer que el cliente pueda seguir procediendo a su ejecución.

Citas

También referidos como citas extendidas en entornos distribuidos. Las invocaciones remotas se sirven mediante una instrucción aceptación (call / in). A diferencia de RPC, las citas hacen comunicaciones en el mismo módulo (normalmente) las instrucciones de comunicación están limitadas: A menudo, un proceso desea comunicarse con más de un proceso, que está en puertos diferentes, y no se sabe el orden en el que los otros procesos desean comunicarse con él.

Todo se basa en la programación no determinista Dijkstra, usando para ello instrucciones guardadas

Las instrucciones que permiten la no determinista son de dos tipos: instrucciones de comunicación e instrucciones guardadas.

La guarda es del tipo

Su semántica explica:

Tiene éxito

$B; C \rightarrow S$

Expresión lógica

si

Comunicación

instrucciones

guardadas

donde B es verdadero y C no produce retrasos

La guarda falla si B es falso

bloquea si B es verdadero pero C no se puede ejecutar sin producir retrasos.

No hay variables globales, por lo que B no puede cambiar hasta ejecutar otras instrucciones de asignación.

C puede ser una instrucción de comunicación de entrada o salida.

Estas instrucciones se pueden combinar en construcciones de los tipos.

Alternativas (IF) Si al menos una guarda tiene éxito, una de ellas se escoge de forma no determinista ejecutando C y S.

Tiene sentido
que vayan
en un bucle
y sea posiblemente
que sea el
mismo el
que se repita.

Si todas las guardas fallan, la alternativa falla o termina.

Si hay guardas con éxito PERO algunos están bloqueados,
la ejecución se retrasa hasta que la primera
guarda tenga éxito.

Repetitivas (DO) Funciona igual que IF, pero
es una ejecución iterativa, donde se ejecutan

- UNA SOLA guarda cada vez que damos vuelta al
bucle, hasta que todas fallen.

Citas entendidas

Como se ha mencionado previamente, estos se localizan
en entornos distribuidos. El proceso de como se usan los
citas en entornos cliente-servidor quedaria de forma que:

El servidor exporta operaciones (de forma similar a RPC)

El cliente invoca las operaciones exportadas.

Para atender a las invocaciones, el servidor usa instancias

de aceptación. Su sintaxis es: [in <Nombre-Operación> (<Parámetros>) → \$n] formales

El ámbito de los parámetros formales es el de la operación
(Scope). En este caso, la guarda tiene éxito si:

- , & {Se invoca la operación
- {La expresión lógica se evalúa a verdad.

Así la ejecución se retrasa hasta que haya éxito con una de las guardas. Si hay varias, la ejecución será no determinista.

Al igual que en las citas, se permiten construcciones IF y DO

Estas citas extendidas adquieren las siguientes características

A diferencia de RPC, el servidor es un proceso activo. Se ejecuta antes y después de servir una invocación remota.

Sin parámetros no hay comunicación, pero sí sincronización.

Las invocaciones se sirven en los instantes que el servidor deseé.

Las operaciones se dan en el contexto del proceso que especifican puntos de comunicación muchos a uno.

El servidor puede definir y puede definir distintas guardias para la misma operación, causando efectos diferentes entre las invocaciones de un mismo servicio.

3. Coordinación

Tiempo lógico

Hay dos cuestiones de tiempo importantes en Sistemas Distribuidos.

Sincronización externa Cuando ocurrió un evento concreto. Para ello es necesario sincronizar la hora de la máquina donde ocurre el evento mediante un reloj o fuente autorizada externa.

Sincronización interna Trata de obtener las mismas referencias de tiempo o intervalo entre los eventos ocurriendo en dos computadoras diferentes conociendo su precisión.

Para ello, requiere la precisión de una medida.

El otro son los problemas lógicos debidos a la distribución de los sistemas.

Se define por evento aquella acción que parece ocurrir inmediatamente. El orden de ocurrencia de estos eventos puede ser crítico en aplicaciones distribuidas.

Hay varios tipos de aplicaciones en base al tiempo, con sus propios requisitos!

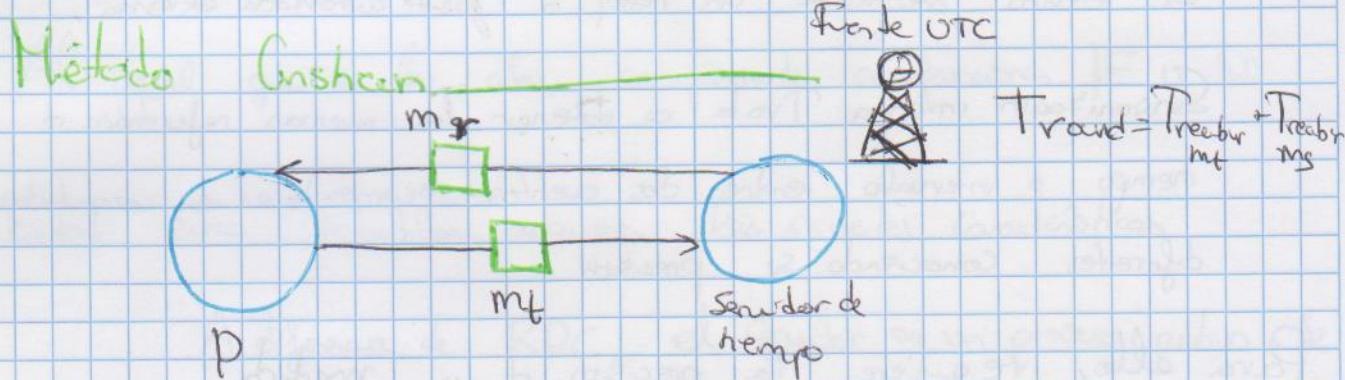
Centralizadas Sólo necesitan conocer el orden de los eventos. Solo basta asociar un reloj o contador a cada evento.

Distribuidas Tienen estos requisitos y opciones

Se debe conocer el desplazamiento relativo del reloj de una máquina con respecto a otra, e igual velocidad del pulso. Esta opción es casi imposible de implementar.

Sistemas sincronos Existe un tiempo físico compartido por el sistema entero.

Sistema asincrono Se hace un servidor de tiempo sobre las peticiones. A través del método Cushing se podrán sincronizar los relojes basados en UTC estandarizados. Esto acarrea problemas derivados del reloj del servidor o si hay un servidor impositivo.



Asumiendo iguales tiempos de envío y recepción, P puede fijar su reloj a $t + T_{round}/2$. Es una aproximación, ya que en el envío y recepción puede haber otras variables.

Relación de orden

Relación ocurrió - antes Es un esquema de ordenación de eventos basado en dos puntos:

Si dos eventos ocurrieron en el mismo proceso, entonces ocurren en el orden que se observan.

Si se envía un mensaje, entonces el evento asociado al envío ocurre antes que el evento de recepción de dicho mensaje.

Lamport generalizó estas dos relaciones con la relación
ocurrió antes (\rightarrow)

Si $\exists p: x \xrightarrow{p} y$ (en p), $x \rightarrow y$

$\forall m \in \text{Mensajes}, \text{Send}(m) \rightarrow \text{receive}(m)$

Siendo $x, y, z \in \text{Eventos}$, si $x \rightarrow y \wedge y \rightarrow z$; $x \rightarrow z$.

Para capturarlos numéricamente, se hace uso de mecanismos
denominados relojes lógicos. Un reloj lógico es un contador
software que se incrementa monótonamente.

C_p : Noda el reloj lógico C del proceso p

$C_p(a)$: Denota la marca de tiempo del evento a en el proceso p .

$C_p(b)$: Denota la marca de tiempo del evento b , independien-
temente del proceso.

Para capturar la relación, los procesos actualizan sus relojes
lógicos y transmitir sus valores en los mensajes.

C_p se incrementa antes de cada evento que ocurre en p

Cuando un proceso p envía un mensaje m añade el valor $t_p(m)$

Cuando un proceso q recibe un mensaje

computamos $C_q = \max(C_q, t_p(m))$ y

aplicar $\Delta C_q = 1$ antes de marcar el evento receive.

Es fácil demostrar que si $a > b$, $C_p(a) \leq C_p(b)$

y puede extender a una relación de orden total

$$(a) < (b) \Leftrightarrow (p(a) < p(b)) \vee ((p(a) = p(b)) \wedge p < q)$$

Algoritmos distribuidos

Exclusión mutua

Este tipo de algoritmos se usan cuando no existe un núcleo central local para basar la exclusión mutua en variables u otras facultades compartidas.

Los requisitos básicos y comunes son:

Propiedades de seguridad y inviolabilidad

Orden causal en la entrada a la sección crítica.

Podemos ofrecer 3 soluciones:

un servidor centralizado, un algoritmo distribuido basado en relays lógicos o un algoritmo basado en anillo.

En un servidor centralizado,

para entrar en la sección crítica se envía una petición al sistema y se espera la respuesta en forma de testigo o token.

El servidor encola peticiones cuando no se dispone de testigo

Cuando un proceso sale de la sección crítica, devuelve el token al servidor, y se da al siguiente la cedencia.

El problema es que puede convertirse en el wellio de botella del sistema, y además es punto crítico de fallo ya que si el servidor falla, el sistema se para. Además

habrá que regenerar el testigo si el cliente falla, además de que el servidor pudiere fallar.

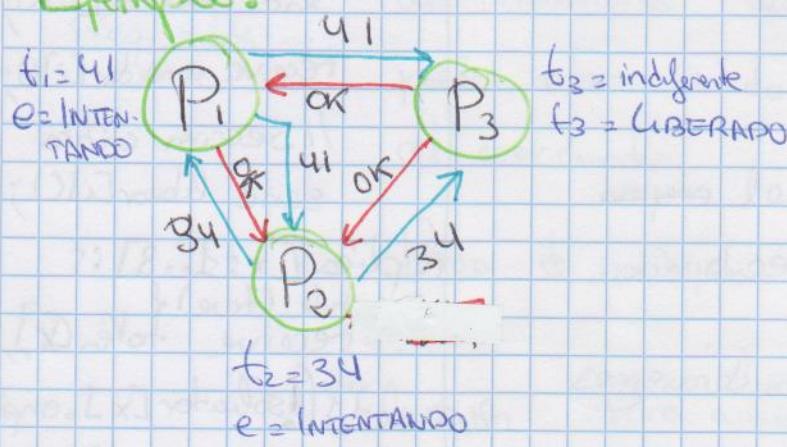
El algoritmo tiene una variante denominada basada en relojes lógicos (Ricart / Agrawala). La idea básica es que los procesos que deseen entrar en la sección crítica envíen un multicast a los $n-1$ procesos. Si recibe de ellos n respuestas, podrá entrar.

Supondremos para ello
los procesos

Paso de mensajes fiable

Cada proceso mantiene su propio reloj lógico

Example:



Por ello, obtener el testigo requiere $\frac{1}{2}(n-1)$ mensajes.

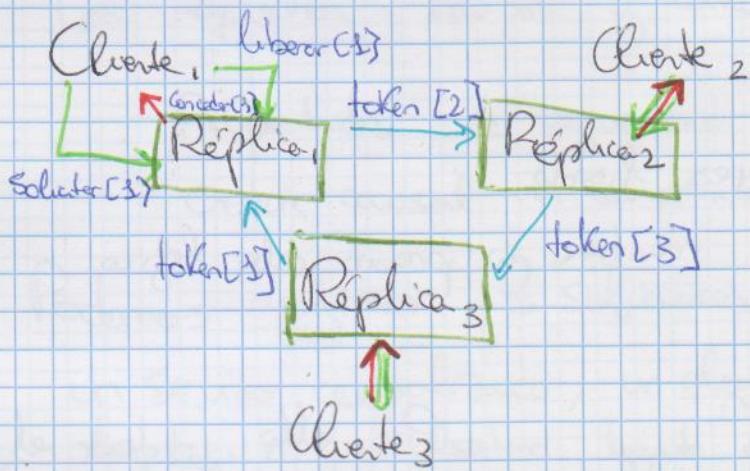
Es más costoso que el algoritmo centralizado y cualquier proceso es

un punto crítico de fallo. Cualquier proceso recibe peticiones y envía respuestas, por lo que puede darse colas de botella.

La exclusión mutua con algoritmo basado en anillo

Hace que los procesos se configuren en un anillo lógico donde cada proceso se conoce a sus vecinos, de forma que el token circula continuamente.

Si un proceso falla, hay que reconfigurar el anillo y si el token se pierde, hay que regenerarlo. Por tanto, no es posible asegurar el cumplimiento de la relación ocumó-antes.



Por con Sevenera, en el mejor
peor de los casos, recibirá

n mensajes. No consideraremos
que los clientes cajan el token.

Pseudo-C

```
char token[1..3]();  
char solicitar[1..3]();  
char conceder[1..3]();  
char liberar[1..3]();  
  
cliente[i][1..3]::  
//...  
send solicitar[i]();  
receive conceder[i]();  
// Sección crítica  
send liberar[i]();  
  
replica[x][1..3]::  
while (true) {  
    receive token(x);  
    if (!solicitar[x].empty)  
        receive solicitar(x);  
        send conceder[x];  
        receive liberar(x);  
        send (token[x % n] + 1);  
    }  
}
```

Algoritmo de elección

Trata de elegir un único proceso de un conjunto de ellos para comandar a los demás, por ejemplo, si uno de ellos falla.

El principal requisito es que el proceso sea único inclusivo. Si varios procesos entran en el proceso de elección simultáneamente. Normalmente se elegirá el de mayor identificador o PID.

Algoritmo Bully

También conocido como algoritmo del valiente

Requisitos

los miembros del grupo se conocen, aunque puede haber algún más de un proceso aparte del coordinador.

Paso de mensajes fiable

Elección Para anunciar una elección

Responde Se envía como respuesta a un mensaje de elección

Coordinador Se envía para anunciar el ID del nuevo proceso coordinador.

Se requieren hasta n^2 mensajes.

El proceso es iniciado cuando detecta que un proceso ha fallado. En ese caso:

Envía un mensaje de elección a los procesos (de mayor ID?)

Espera mensaje de respuesta

Si no llega el mensaje de respuesta

Se programa coordinador y envía mensaje
Coordinador a los procesos de menor ID.

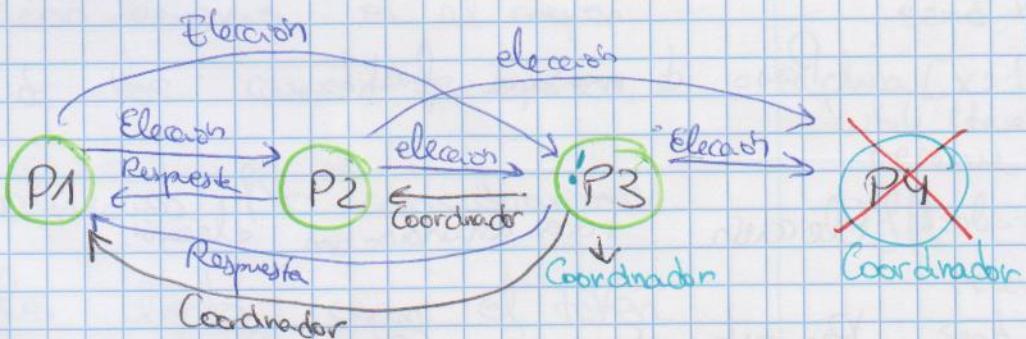
Si llega,

espera a que llegue un mensaje coordinador, si no
hacer nueva elección

Si un proceso recibe un mensaje coordinador graba
la id del coordinador, devuelve un mensaje de
respuesta y comienza una nueva elección

Cuando el proceso fallido se restablece, comienza una nueva elección.

P4



Una cosa curiosa es que si el proceso se restablece durante un tiempo habría dos o más procesos coordinadores que mandan mensajes coordinador.

Algoritmo basado en anillo.

Requisitos

Los procesos se organizan en un anillo lógico aunque sin considerar el orden con su identificación.

Paso de mensajes fiable y los procesos no fallan durante la ejecución.

Tipos de mensaje

Elegido Anuncia que hay uno elegido en curso

Coordinador Se envía un mensaje con el ID del coordinador.

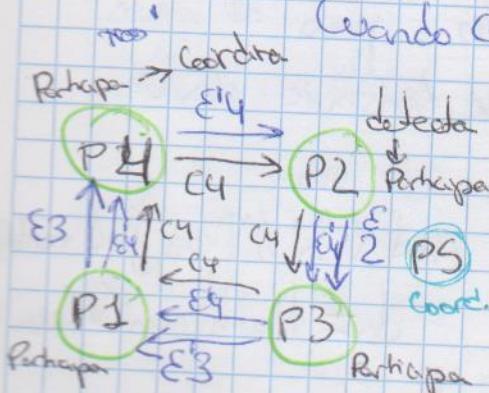
Al participante Los procesos son no participante.

Mensajes

Pueden activar la elección manteniéndose como participante y enviando el ID correspondiente.

Si recibe un mensaje de elección, compara ID con el del participante. Si se marca como no participante, entra a participar y pasa el mensaje. Si es menor, pasa el mensaje a su vecino. Si es mayor,

Cuando coincide el ID con el del proceso, se convierte en coordinador.



Si se recibe un mensaje de coordinador, se marca como no participante y se pasa el mensaje.

En el peor de los casos, el costo es $3n-1$.

Consenso

Trata de resolver de forma precisa y generalizada problemas de consenso y relacionados. Suelen acordar valores tras enviar diferentes propuestas. Tras establecer el valor no se cambia.

Ejemplos. Solución del problema de los generales Birchenes, Consistencia intencional, multicast ordenado.

Todos los anteriores tipos, de una forma u otra, son procesos de consenso. Transacciones, exclusión mutua, elección y comunicaciones en grupo.

Comunicaciones fiables

Requisitos

Los procesos pueden fallar arbitrariamente.

Los procesos correctos acabarán eventualmente, deberán decidir el mismo valor, y si dentro de los correctos proponen el mismo valor, cualquier correcto afirma el mismo valor.

Algoritmo general.

Todos proponen valores hasta qj se cumpla el multicast. Si los procesos fallan, pero al menos uno recoge ese valor, se consume otra vez hasta llegar al fallar quienes aún son.

receive-loop (r_{final})

if (receive(p_j, V_j)

Valores- $i \leftarrow \text{sig. Refresh}(V_i)$

end () // Tras final ready

$d_j = \text{Math.min}(\text{valores}_i - \text{final})$,

start () {

valores - i - 1 E] = { vi };

valores - i - 0 [] = { };

sendLoop (r) { // En ronda r, $1 \leq r \leq f+1$

foreach (valor : valores - i - r) {

if (!valores - i - r - anterior.contains (valor))
multicast (valor);

valores - i - r - sig = valores - i - r;

}

El algoritmo general de consenso posee las siguientes propiedades:

Terminación es algo obvio, ya que el sistema es sincrónico por los timeouts.

Acordio Cada proceso llega al mismo punto de valores cuando termina la ronda final.

Integridad. Se aplica a los valores propuestos por todos los procesos, y los procesos correctos llegan al mismo punto de valores cuando termina la ronda final y aplica la misma función.

El problema de los generales bizarros _____

El problema de los generales bizarros se nos describe de la siguiente manera:

"Tres o más generales deben acordar entre atacar o rehuirse, pero uno o más de los generales pueden ser traidores. Por tanto,

Si el comandante emite la orden a los tenientes para ejecutar su orden puede pasar dos cosas:

1. Que el comandante sea un traidorero, quien propone a uno atacar y al otro defender

2. Que el teniente sea traidorero de forma que a otros dos procesos tenientes que uno defienda y otro ataque."

esta desconfiguración es un caso especial de consenso, porque solo un proceso propone el valor y se pueden dar fallos arbitrarios de los procesos.

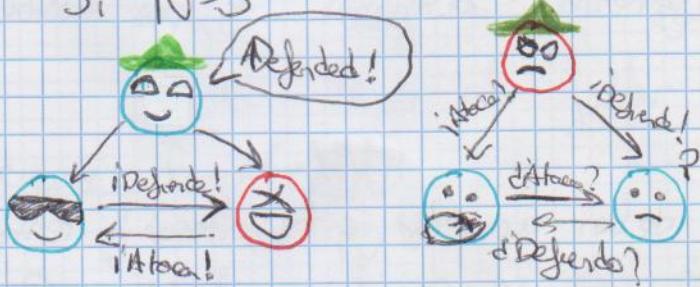
El requisito de autenticidad se podría probar, ya que si el comandante es correcto, todos acuerdan el valor.

En esta situación, hay 2 rondas de mensajes a los valores enviados por el comandante.

Una característica es que no se podía llegar a un consenso para $N=3$ o $N \leq 3f$.

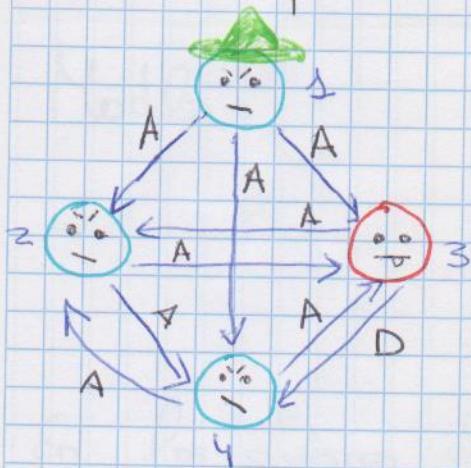
Si $N > 3f$, los procesos aplican la mayoría al conjunto de valores recibidos, implicando $f+1$ rondas.

Si $N=3$



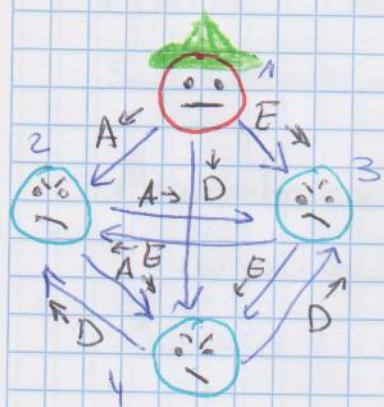
Si tienen que decidir que hacer, no llegan a ningún acuerdo.

Con 4 procesos...



A = Atacar
D = Defender
E = Esperar.

El comandante envía la orden de atacar.
Entre 2 y 4 hay consenso, atacan.
Entre 2 y 3 hay consenso, atacan.
Entre 3 y 4, uno dice de atacar y otro de defender, pero como entre 2 y 4 dicen de atacar ~~pero~~ y el comandante también, se concuerda atacar. También venían que 3 es sospechoso de fraude.



El comandante envía una orden distinta a cada 1.

Entre 2 y 3 se recogen las órdenes de atacar y esperar.

Entre 2 y 4 se recogen las órdenes de atacar y defender.

Entre 3 y 4 se recogen las órdenes de defender y esperar.

2, 3 y 4 notan que tienen tres órdenes distintas puestas en común, por lo que saben que tendrán que ponese de acuerdo entre ellos ya que el comandante es sospechoso de fraude.

Tema 4. Sistemas Cliente/Servidor y P2P

Introducción.

Modelo

En los sistemas distribuidos, hay varios tipos de modelos.

Físicas Forma explícita de describir el sistema capturando su composición hardware.

Arquitectónicas Descripción en términos de fases de computación y comunicación que deben satisfacer requisitos actuales y futuros.

Fundamentales Es la perspectiva abstracta de aspectos individuales como la interacción, la tolerancia a fallos y la seguridad.

Modelo físico

Modelo base Componentes: hardware y software esenciales

Primeros sistemas distribuidos Redes locales con computadoras homogéneas y servicios básicos

Internet Grandes sistemas distribuidos en la red de redes con elementos heterogéneos

Entorpecionales Son más heterogéneos, dinámicos y cooperativos.

llevan a ser Sistemas de Sistemas (SoS) como el Internet de las cosas (IoT)

Modelos arquitectónicos

Los elementos arquitectónicos son:

Entidades, a nivel de sistema y modelo de programación.

Diferencia entre objeto y componente: Un componente agrupa varios objetos y las relaciones entre tales objetos, tanto de su estructura como de su comportamiento.

Paradigmas de comunicación: IPC, invocación remota, o comunicación indirecta con desacople espacial y temporal.

Desacople temporal

Desacople espacial

Roles

Clientes/Servidores donde hay servidores que podrían ser clientes

Igual a Igual (P2P), escala mejor al compartir más computadoras o comunicaciones porque es más complejo.

Estrategias de despliegue] Cómo organizar el sistema? Con servicios para múltiples servidores, cachés, código móvil y agentes móviles.

A partir de ahí, podemos usar varios patrones arquitectónicos:

Por capas / layers, con una organización vertical por niveles de abstracción.

Por etapas / tiers, es complementario al anterior porque organiza la funcionalidad dentro de cada capa.

Thin Client / Server Da entrada a mover complejidad a servicios (como Clad), y por tanto simplificar superposiciones en clientes (donde tienen menos recursos)

Intermediarios (Proxy) con mismo fin pero para transparencia en distribución, replicación, cache...

Gestores / Brokers que soporta interoperabilidad en infraestructuras complejas. Por ej., binders y descubrimiento de servicios

Reflexión Soporta cambios persistentes y adaptación / evolución cambiando estructura y comportamiento.

Mito de los fundamentales

Se basan en las siguientes características:

Interacción Varios réplicas de servicios o procesos P2P que cooperan, necesitando para ello algoritmos distribuidos.

Rendimiento de canales Se mide si el ~~capacidad de~~ tener un establecimiento permanente (streaming) o no.
Reloj es y tiempo de eventos

Variantes

Síncrono	Cada reloj físico, proceso y mensaje tiene una cota máxima de tiempo
Asíncrono	No se conoce la velocidad de reloj, procesos ni retrasos de los mensajes.

Ordenación de eventos a través de relojes logicos

Fallos que pueden pasar en mensajes y procesos.

Diseño

Servicios

Un servicio es una colección de atributos y comportamientos que pueden ser proporcionados por un recurso empresarial para su uso a través de interfaces bien definidas.

Este concepto resulta de separar el comportamiento externo e interno de ~~la empresa~~ sistema, en base a las siguientes perspectivas:

Entorno

Se asocian a los principios de diseño de los servicios

Abstracción Los contratos de servicios sólo contienen información esencial, y la información se limita a lo establecido por contrato.

Contrato estandarizado Servicios dentro del mismo marco de servicios cumplen con los mismos estándares de diseño del contrato.

Rebal acoplamiento Los contratos de servicio imparten requisitos de acoplamiento de consumidores bajos y están diseñados de su forma.

Reutilización Contienen y expresan lógica, y pueden posicionarse como recursos reutilizables.

Autonomía Ejercen un alto nivel de control sobre su entorno de ejecución subyacente.

En estado Minimizan el consumo de recursos delegando la gestión de la información de estado a otro recurso.

Desacoplamiento Se complementan con metadatos para ser descubiertos e interpretados eficazmente.

Componibles Son participantes eficaces de la composición, sin importar el tamaño y la complejidad de la composición.

Para componer hay 2 estrategias

Orquestación Un "director" controla las decisiones a realizar y se va comunicando con los demás.

Coreografía No hay un "director", sino que dinámicamente se van realizando las peticiones según las capacidades de cada servidor.

El SOA permite que el software esté disponible bajo demanda. Gracias al principio de abstractioón, el servicio esconde los detalles subyacentes del servicio para permitir o preservar la relación de bajo acoplamiento.

Hay tres ejemplos de instancias:

Encapsulamiento de sistemas legacy Permite interactuar sistemas antiguos y modernos

Encapsulamiento de servicios en un servicio Permite abstractear servicios ajenos en un servicio común.

Encapsulamiento de componentes personalizados Trata la virtualización de los sistemas, donde la misma imagen en el exterior a través de una interfaz común, si bien en el interior están de forma heterogénea

Es importante la clasificación de los servicios. La infraestructura se puede clasificar como:

Transporte

Manejo de recursos de información

Administración

La interconectividad en computadoras distribuidas implica resolverlo entre máquinas, redes y aplicaciones. Al usar servicios de

alto nivel, las aplicaciones recibirán su código.

Desde el punto de vista de la gestión, se deben explotar los servicios de más alto nivel, y así hacia abajo.

Eso sí, no es fácil interconectar aplicaciones que ofrecen servicios a diferentes niveles.

Los servicios completos pueden ser utilizados directamente por los usuarios finales mediante órdenes y las aplicaciones se pueden construir en base a estos servicios lo que implica minimizar el fuero de desarrollo.

Los servicios IPC están generalmente disponibles sólo para desarrolladores de aplicaciones invocados a través de APIs, lo que permite más flexibilidad y eficiencia.

Si se ofrece una API, ésta por ser usada,

las aplicaciones distribuidas (bien por P2P o Cliente/Servidor) pueden usar diferentes paradigmas para el intercambio de mensajes.

Petición / Respuesta. Cada petición / respuesta va como una unidad separada. Cumple estrechamente con el modelo Cliente / Servidor.

Conversacional. Cada interacción no está autocentrada ni es una unidad independiente. Lo utilizan C/S y P2P.

Procesamiento de mensajes encabezados. El receptor almacena los mensajes de pehichi en una cola, y atiende los mensajes cuando esté desocupado; y así el emisor puede enviar pedidos asincrónicos.

Los modelos más usuales para interconectar aplicaciones Cliente/Servidor y Peer2Peer (P2P). Notese que C/S se puede implementar sobre P2P pero lo contrario no es necesariamente verdad. Ambas no escalan bien igual, en particular horizontalmente.

Se hace ahora a un procesamiento distribuido cooperativo, usando C/S o P2P. Puede ser implementado con diferentes configuraciones, y C/S y P2P resultan ser ^{algo} categorías

Cliente/Servidor

Modelo clásico de 2 etapas / 2-Tiers

GUI + Lógica → Lógica + Datos

GUI → Lógica + Datos

GUI + Lógica → Datos

Modelo solapado

, de tres etapas o
3-Tiers

GUI → Lógica → Datos

Los módulos de 2 etapas deben distribuir la autoridad, responsabilidad e inteligencia. Algunas configuraciones:

Thin Client - Fat Server: Permite clientes muy ligeros con muy fuerte CRUD. Tienen menor carga por la red

Thin Server - Fat Client: Permite que el server sea muy estable.

Modelos intermedios: Tienen las ventajas y desventajas de ambos, en proporción.

Modelos de n etapas:

Son más avanzados y flexibles al tener mayor autonomía, más robustos por sus partes independientes. Están adaptados para aquellos sistemas con datos distribuidos.

Como inconveniente, es difícil construir sistemas fiables y eficientes con más de 3 etapas.

Servidor de aplicaciones
de 3 etapas con una separación entre sus propiedades. Es una categoría middleware basada en componentes

Proporcionan soporte directo a modelos separados lógico y datos, separando categorías middleware basado en componentes

Mejores técnicas

Evitar el rendimiento del sistema será una fuerza en tanto orden. Para ello se presentan varios simuladores para analizar referidos en la red y tiempo de procesamiento en nodos.

Otra consideración será minimizar mensajes, ya que el tráfico de red sigue patrones impredecibles, con tiempos de respuesta, latencias de calcular por el entorno y problemas de fiabilidad.

desarrollar a protocolos y algoritmos

Anexo. Sintaxis de citas en Ada

Creación de tareas

En Ada, la unidad de proceso secuencial que se puede ejecutar de forma paralela se denomina tarea o task.

Las tareas se definen con: task <nombre>

Y se define su funcionamiento con

Ada | task <body> <nombre> is
begin
-- Cuerpo
end <nombre>

Citas

Las tareas interactúan entre sí a través de citas. Estas actúan como señales de la llamada de una tarea a un punto de entrada declarado en otra tarea. Por tanto, hay que definir puntos de entrada.

Los puntos de entrada se definen por entre <nombre>

Las citas hay que aceptarlas. Al hacerlo podemos definir cómo funciona ese procedimiento. Así pues, para aceptar una cita tenemos la siguiente sintaxis

Ada | accept <nombre> do
-- Cuerpo
end <nombre>

Entre las dos sentencias accept se admite poner otro código.

Selección de casos: DO e IF en Ada

La definición de una construcción alternativa IF requeriría el constructo Select / de sintaxis:

Ada

Select

(when <condición>)

accept <opción 1> do

-- Código

end <opción 1>

Or

(when <condición>)

accept <opción 2> do

-- Código

end <opción 2>

Or

...

Or

(when <condición>)

accept <opción N> do

-- Código

end <opción N>

end select

La definición de la construcción repetitiva DO consistiría en introducir un select entre un loop / end loop.

Se pueden introducir esperas con delay ().

Se puede escribir texto con Ada.Text_IO.Put_Line.

Los puntos de entrada pueden ser también

1 Con tiempo límite (llama a un punto de entrada que se cancela si

2 Condicionales no se acepta antes de tiempo

3 Asíncronas llama inmediatamente a un punto de entrada que se cancela si no se acepta

se abren las secciones que se están ejecutando

se está ejecutando una sección de sentencias. Es decir, si se

acepta la llamada al punto de entrada se cumple el tiempo,

Ada

Select

delay 5.0;

raise FunctionNotOverridable;

end select

Select

Controlador.Rebano (Model);

or

delay 50.0;

Put ("Controlador actual");

1

Ada

Select

Procesador.Auso

else

raise Error

end select;

2

Ada

Select

Procesador.Auso

else

raise Error

end select;

Anexo • poll y select en POSIX

Los servidores deben manejar miles de descriptores de archivos. En un servidor se tienen miles de conexiones abiertas a la vez, que se conectan con la llamada a sistema accept donde como resultado un descriptor de archivo.

El problema con ello es que consume mucho tiempo de CPU. Por ello, el Kernel de Linux, en vez de que le vayamos preguntando "¿Está ya?", se le pide que avise cuando una de los descriptores cambie su estado.

Para monitorizar esos descriptores, están `select()` y `poll()`. En esencia, hacen casi lo mismo:

1 Se le da una lista de los descriptores de archivos de los que queremos sacar información.

2 El Kernel dice qué descriptores tienen datos para leer/escibir.

Las diferencias son por lo que cogen:

`poll (pollfd *fds)`: Se le pasa como argumento una lista de descriptores de archivo.

`select (int n_fds, fd-set *read_fds, fd-set *write_fds, fd-set *except_fds)`: Pasa por argumentos:

`n_fds` Número de descriptores de fichero.

`read_fds` Bitset de descriptores de fichero ALEGR

`write_fds` " " " " PARA ESCRIBIR

`except_fds` " " " " PARA EXCEPCIONES

Anexo. Blockchain, Bitcoin y el Problema de los generales bizantinos

Desarrollo de Sistemas Distribuidos.

1 El problema de los generales, revisado

el problema de los generales bizantinos es un problema derivado de algoritmos de consenso comentado por Leslie Lamport.

Dice lo siguiente:

Supongamos un ejército donde 1 comandante envía a $N-1$ tenientes una orden, y se ponen de acuerdo para atacar o defenderse. Pero... ¿y si uno de los tenientes, o el mismo comandante fueran traidores de la otra facción? "

Según lo visto en clase, no tendría solución si $N=3$ y/o $N \leq 3$.
(Ojo, N incluye al comandante y los tenientes)

2 Bitcoin y los blockchain

2.1 Transacciones

El concepto básico de los blockchain son las transacciones, que indican un movimiento de un registro de una dirección de origen a otra de destino.

Tales direcciones se representan mediante pares de clave pública/privada. Con tal de poder hacer la transacción se necesita la clave privada asociada a su clave pública correspondiente. Así, la transacción se sube a la red tras haberse firmado con la clave privada.

Supongamos una transacción recibida.

$$T_0 = \{ \text{Input}_0, \text{Output}_0 \}$$

$$\text{Input}_0 = \{ \dots \}$$

$$\text{Output}_0 = \{ \text{Addr}(\text{PK}_A), <\text{Cuerpo}> \}$$

↓
Dirección destinada
a clave

↓
Clave pública

Y otra para enviar

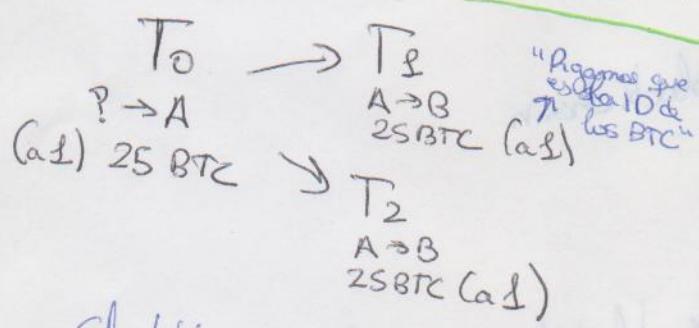
$$T_1 = \{ \text{Input}_1, \text{Output}_1 \}$$

$$\text{Input}_1 = \{ \text{Hash}^{\text{firmacode}}_{\text{privada}}, \text{Firma}^{\text{privada}}_{\text{privada}}, \text{Clave}^{\text{pública}}_{\text{pública}} \}$$

$$\text{Output}_1 = \{ \text{Addr}(\text{PK}_A), <\text{Cuerpo}> \}$$

2.2 La cadena de bloques

Un problema de las transacciones es que se podría hacer varias mensajes validez una transacción anterior. En si podría ser una ventaja, pero si lo fueses en el caso de los Bitcoins y las criptomonedas, podría generar lo que se conoce como un doble gasto.



Si fisiéramos hacer nuestra moneda, deberíamos evitar este tipo de cosas. ¿Pero cómo?

La respuesta: la cadena de bloques o blockchain.

En esencia, el blockchain es un registro de transacciones público. De esta manera, al recibir una transacción, se comprueba que esa transacción no tenga el mismo input que otra transacción. Evidentemente sería un requisito indispensable que el hash resultante sea siempre único para que todo ello pueda ser válido, pero con algunos que veremos más adelante resolvemos ese problema.

Al final, ese registro será un conjunto secuencial de bloques, de los cuales se irán creando más y se irán uniendo a la cadena existente.

En el caso de Bitcoin, la tarea de verificar las transacciones y paserlas por bloques es tarea de los mineros, quienes ^{verifican} los bloques (si son válidos se descartan).

2.3

Proof of work. ¿Qué tiene que ver el problema de los generales en esto?

Con el tema de los mineros, nos puede dar a pensar: ¿Y si hay mineros corruptos, o traidores, que aceptan esos bloques con dobles gastos?

Para ello, se requiere que los bloques que incluyan las transacciones pasen por una secuencia denominada prueba de trabajo o proof of work para considerarse válidos.

Esta prueba demuestra que se ha gastado un tiempo de computación en generar ese bloque de transacciones. De esa forma, se trata de encontrar un valor "nonce" de forma que el hash del bloque sea inferior al valor objetivo.

Así, un minero honesto tratará de realizar varios intentos hasta dar con el hash correcto, tras lo cual se anexa al registro siendo el último ~~el~~ bloque de este.

¿Y si dos mineros llegan al mismo resultado? Se coge la cadena más larga.

De esta manera, si el nº de transacciones es pequeño, no podrá pasar la prueba de trabajo.

Al ser principal el rol de mineros en el caso de los criptomonedas.

se suelen dar recompensas en ese momento a los animales
que realizan una actividad en ese momento. La otra forma es
que se premia con una recompensa al animal que realizó
una actividad deseada y se evita la recompensa a los animales
(animales o individuos no deseados)

funcionamiento de los animales. Hay lo siguiente: Es
(el animal deseado)

que el animal que está deseado tiene que ser recompensado
y el animal que no es deseado tiene que ser castigado.

Algunos animales tienen más facilidad para adaptarse a las
condiciones de vida que otros. Los animales que tienen más
facilidad para adaptarse a las condiciones de vida son los que
tienen más facilidad para vivir en diferentes tipos de entornos.

Los animales que tienen más facilidad para adaptarse a las
condiciones de vida tienen más facilidad para vivir en diferentes
tipos de entornos. Los animales que tienen más facilidad para
adaptarse a las condiciones de vida tienen más facilidad para
vivir en diferentes tipos de entornos.

Los animales que tienen más facilidad para adaptarse a las
condiciones de vida tienen más facilidad para vivir en diferentes

tipos de entornos. Los animales que tienen más facilidad para
adaptarse a las condiciones de vida tienen más facilidad para
vivir en diferentes tipos de entornos.

Liberos de algoritmos distribuidos,
en pseudo C

Algoritmo distribuido basado en relojes lógicos

```
process [ ] : Process; → Process {
    id : Integer;
    state : String;
    Time-mark : Integer;
    petitions : Map[Integer, Message];
}

procedure start ( proceso : Process ) {
    proceso.state = "LIBERADO";
}
```

```
procedure obtain_token ( proceso : Process ) {
    proceso.state = "INTENTANDO";
    Time-mark K = Date.now; // Integer
    foreach ( proceso - externo in procesos ) {
        if ( proceso - externo != proceso )
            send ( proceso - externo, Time-mark );
    }
}
```

```
Responses : Integer = 0;
while ( Responses != procesos.length - 1 ) {
    foreach ( proceso - externo ! = proceso ) {
        if ( proceso.externo.Time-mark > proceso.Time-mark ) {
            proceso.state = "INTENTANDO";
            petitions.add ( receive ( proceso.externo ), processo );
        } else
            send ( proceso.externo, "OK" );
        if ( receive ( proceso.externo, "OK" ) )
            Responses += 1;
    }
}
process.state = "EN-SECCION-CRITICA";
```

```

procedure liberate - token (processo : Process) {
    processo.state = "LIBERADO";
    foreach (petición in pedidos) {
        send (pedido.senor, "OK");
        petidos.delete (pedido);
    }
}

Algoritmo basado en anillos
// Dejamos los procedimientos genericos (que sirven
// de canales de comunicación.

msg-channel token [NUM_SERVERS];
msg-channel solicitor [NUM_SERVUBRS];
msg-channel concord [NUM_SERVERS];
msg-channel liberar [NUM_SERVERS];

// Procedimientos del cliente
procedure Cliente :: loop (id-server : Integer) {
    while (true) {
        i = id-server;
        send (solicitor[i], i); // i, para mandar msg.
        receive (concede[i], token);
        // Eventos secuencia critica
        send (liberar[i], token);
        // ...
    }
}

// Procedimientos del servidor
procedure Server :: loop (id-server : Integer) {
    while (true) {
        i = id-server;
        receive (token[i], token);
        if (!solicitor[i].empty()) {
            receive (solicitar[i], i);
            send (concede[i], token);
            receive (liberar[i], token);
        }
        send (token[i] % NUM_SERVERS] token);
    }
}

Algoritmo del voluntario / Bully
processes : Process[E];
on - event (P ACCESS-FINAL, bully-election);
on - event (PROCESS-NICER-READY, bully-election);
procedure bully-election (process : Process);
for each ( process-extero in Processos )
    if ( process-extero != proceso & & proceso.estado =
        > proceso.id ) {
        proceso.send (process-extero, ELECTION);
    }
msg = any-receive-thread ("RESPUESTA", 20000);
if (msg == null) {
    process.estado = COORDINADOR;
    foreach (process-extero in processos)
        if (process-extero.id < proceso.id)
            proceso.send (process-extero, (coordinador,
                id-coordinador));
    else
        if (process-extero.receive ("COORDINADOR",
            id-coordinador));
}
}

```

else is receive (ELECTION) {
processo. send (ELECTION.size(), RESPUESTA);

else if (ELECCION .id = process . id)
process . status = NO_PARTICIPANTE;
} = COORDINADOR . id = proceso . id;
process . send (vecino , COORDINADOR);

Algoritmo de elección basado en calle

```

process : Process [ ]
procedure start( process : Process ) {
    process $status = NO_PARTICIPANTE;
}
Process {
    status < String;
    varno : Process;
    i.e : Integer;
}

```

On-event (process-Fail, ring-decoration);

```

Procedure ring-election (process : Process) {
    if (I receive (Election) and
        process.Status = PARTICIPANTE;
        process.send (vacio, id));
}

```

Algoritmo de comando general.
procedure start();

```

    procedure end();
    d = min(volumes - r);
    }  

    volumes - r - prev = f();
    volumes - r - f() = volumes - r;
    volumes - r = f();
    volumes - r - f() = volumes - r;
}

```

Please send (vacine) (vacine);

```

procedure send-loop (r : Integer) {
    foreach (valor in values - r) {
        if (!values - r - prev . contains (valor))
            multicast (r);
    }
    values - r - sig = values - r;
}

procedure receive-loop (r : Integer) {
    if (pi . receive (V_i)) {
        values - i - r - sig . add (V_i);
    }
}

```