Ignacio Méndez Pérez
2023

Resumen Sistemas Distribuidos

Los procesos compartirán **canales**:
- Rutas de comunicación entre procesos.
- Un canal es una abstracción de la red de comunicaciones.

`channel <ch>(<tp_1> <id_1>, …, <tp_n> <id_n>)`

- `<ch>` es el nombre del canal.
- `<tp_i>` y `<id_i>` son los tipos de datos (obligatorios) y los nombre (opcionales) de los campos del mensaje transmitido.

Ejemplos:

```
channel input(char)
channel diskAccess(int cylinder, int block, in count, char* buffer)
channel[n] result(int)
```

Un proceso envía un mensaje al canal `ch` ejecutando:

`send ch(<expr_1>,…,<expr_n>)`

- `<expr_i>` son expresiones cuyos tipos deben corresponder con los tipos de los campos en la declaración de `ch`.

Un proceso recibe un mensaje desde el canal ch ejecutando:

`receive ch(<var_1>,…,<var_n>)`

- `<var_i>` son variables cuyos tipos deben corresponder con los tipos de los campos en la declaración de `ch`.

El desafío es diseñar protocolos de entrada y salida de modo que se cumplan 4 condiciones

1. **Exclusión mutua**: A lo más un proceso a la vez está ejecutando su sección crítica.
2. **No hay *deadlock***: Si dos o más procesos están tratando de entrar a sus secciones críticas, entonces a lo menos uno tendrá éxito.
3. **No hay postergación innecesaria**: Si sólo un proceso está tratando de entrar a su sección crítica, y los otros están en sus secciones NO críticas o terminaron, entonces ese proceso podrá entrar casi de inmediato.
4. **Entrada**: Un proceso que está tratando de entrar a su sección crítica tendrá éxito eventualmente.

Ignacio Méndez Pérez
2023

Uso de un token circulando en un anillo

```
channel[1...n] token(),
   enter(),go(),exit()

process helper[i = 1 ... n]:
   while (true):
      receive token[i]()
      if (!empty(enter[i])):
         receive enter[i]()
         send go[i]()
         receive exit[i]()
      send token[i+1]()

process user[i = 1 ... n]:
   while (true):
      send enter[i]()
      receive go[i]()
      SC
      send exit[i]()
```

La solución es descentralizada y justa.

Como los procesos, `user[i]`, tienen otro trabajo que hacer empleamos procesos adicionales, `helper[i]`, que son los que forman el anillo y se preocupan en de enviar y recibir el token:

- Circula un token —mensaje nulo— entre los procesos **`helper`**
- Cuando `helper[i]` recibe el token, ve si `user[i]` quiere entrar a su SC, en cuyo caso se lo permite
  ... de lo contrario, sólo hace circular el token.

El algoritmo de sacar un número

```
int number ← 1, next ← 1
int[n] turn ← {0 … 0}
process p[i]:
while (true):
   turn[i] ← FA(number, 1)
   while (turn[i] ≠ next);
   SC
   next ← next+1
   sección no crítica
```

Dos pasos fundamentales:

1. Un proceso que quiere entrar a su sección crítica primero saca un número uno más grande que el número que tenga cualquier otro proceso.
2. Luego espera hasta que todos los procesos son números menores hayan pasado por sus secciones críticas.

Las variables:

- El dispensador de números, **`number`**, inicialmente en **`1`**.
- Los números que sacan los procesos: **`turn[i]`** es el número sacado por el proceso **`p[i]`**, inicialmente en **`0`**.
- El número `next`, que tiene el proceso que está en su sección crítica, inicialmente en `1`.

El algoritmo de la panadería

```
int[n] turn ← {0 … 0}

            process p[i]:
while (true):
   turn[i] ← 1
   turn[i] ← max{turn[1],…,turn[n]}+1
   for (j ← 1 … n; st j ≠ i):
      while ((turn[j] ≠ 0)
      && (turn[i],i) > (turn[j],j));
   SC
   turn[i] ← 0
   sección no crítica
```

No usa variables globales como **`number`** o **`next`**.

- No hay un suministrador de números.
- Los procesos chequean entre ellos en lugar de chequear contra un contador central.

Los dos pasos fundamentales:

1. Un proceso que quiere entrar a su sección crítica primero mira los turnos de todos los otros procesos y luego define su propio turno como uno más que el de cualquier otro.
2. El proceso luego espera comparando su turno con el de todos los otros procesos, hasta que el suyo sea el menor de todos.

El algoritmo de Ricart-Agrawala

Usamos dos procesos por nodo (que comparten memoria y variables):

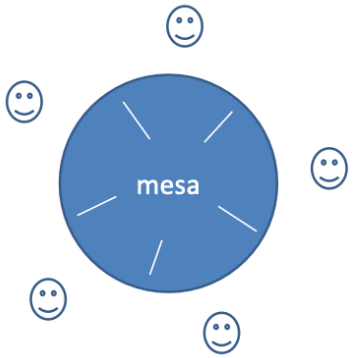
- `main`, el que efectivamente entra a la sección crítica.
- `receive`, el que recibe y procesa solicitudes de otros nodos (en particular, cuando el `main` está ejecutando su protocolo de entrada).

```
int[] requested ← [0 … 0]
int[] granted ← [0 … 0]
int my# ← 0
bool inCS ← false
bool haveToken ← true en nodo 0, false en otros

process main:
   while (true):
      NCS
      if (!haveToken):
         my# ← my# + 1
         for (all other nodes i):
            send rqst[i](myID, my#)
         receive token(granted)
         haveToken ← true
      inCS ← true
      SC
      granted[myID] ← my#
      inCS ← false
      sendToken
```

```
process receive:
   int source, req#
   while (true):
      receive rqst[myID](source, req#)
      requested[source] ← max(requested[source], req#)
      if (haveToken && !inCS):
         sendToken

sendToken:
   if (exists p such that requested[p] > granted[p]):
      for (some such p):
         send token(p, granted)
         haveToken ← false
```

Problema de los filósofos

Imagina que hay 5 filósofos sentados alrededor de una mesa redonda. Entre cada par de filósofos, hay un tenedor. Por lo tanto, hay un total de cinco tenedores. Los filósofos tienen dos actividades principales: pensar y comer. Para comer, un filósofo debe tomar los dos tenedores adyacentes a él, uno a su izquierda y otro a su derecha. Una vez que ha terminado de comer, devuelve los tenedores para que otros filósofos puedan usarlos.



Ignacio Méndez Pérez
2023

El problema es cómo coordinar el comportamiento de los filósofos para evitar situaciones de interbloqueo y muerte mutua, es decir, evitar que los filósofos se queden atascados esperando indefinidamente a que un tenedor esté disponible o que uno o más filósofos nunca puedan comer debido a la ocupación continua de los tenedores por parte de otros.

En el lenguaje de **Go**, una *goroutine* es una forma de ejecutar funciones de manera concurrente y ligera. Permite que múltiples funciones se ejecuten de forma independiente y concurrente dentro del mismo programa.

En el código, cada filósofo intenta tomar primero el tenedor izquierdo y luego el tenedor derecho. Si ambos tenedores están disponibles, el filósofo procede a comer. Sin embargo, si el tenedor izquierdo (o derecho) no está disponible, el filósofo se bloqueará en la llamada a `Lock()` hasta que el tenedor esté disponible.

Para evitar el deadlock, aseguramos que todos los filósofos tomen los tenedores en el mismo orden, es decir, siempre toman el tenedor izquierdo antes del derecho. De esta manera, si todos los filósofos intentan tomar sus tenedores al mismo tiempo, solo un filósofo tendrá éxito en tomar su tenedor izquierdo, y los demás filósofos esperarán hasta que esté disponible. Esto evita que se bloqueen mutuamente.

Cuando se llama a `Lock()` en un `mutex`, si el `mutex` está libre (es decir, nadie más lo ha bloqueado), la *goroutine* actual adquirirá el `mutex` y continuará su ejecución sin problemas. El `mutex` se marca como "bloqueado" y cualquier otra goroutine que intente adquirirlo se bloqueará hasta que sea liberado.

Si el `mutex` ya está bloqueado por otra *goroutine*, la goroutine actual se bloqueará (esperará) hasta que el `mutex` esté disponible. En este punto, la *goroutine* actual no avanzará más allá de la llamada a `Lock()` hasta que se libere el `mutex` por la goroutine que lo bloqueó inicialmente.

```go
package main

import (
  "fmt"
  "sync"
  "time"
)

const numFilosofos = 5

type Filosofo struct {
  id              int
  tenedorIzquierdo *sync.Mutex
  tenedorDerecho   *sync.Mutex
}

// Función para que un filósofo intente tomar ambos tenedores y comer.
func (f *Filosofo) comer() {
  // Simular el proceso de comer
  fmt.Printf("Filósofo %d está comiendo\n", f.id)
  time.Sleep(time.Second)

  // Liberar los tenedores después de comer
  f.tenedorIzquierdo.Unlock()
  f.tenedorDerecho.Unlock()
}

// Función para que un filósofo intente tomar los tenedores.
func (f *Filosofo) intentarComer() {
  // Intentar tomar el tenedor izquierdo.
  f.tenedorIzquierdo.Lock()
  fmt.Printf("Filósofo %d ha tomado el tenedor izquierdo\n", f.id)

  // Intentar tomar el tenedor derecho.
  f.tenedorDerecho.Lock()
  fmt.Printf("Filósofo %d ha tomado el tenedor derecho\n", f.id)

  // El filósofo tiene ambos tenedores, por lo que puede comer.
  f.comer()
}

func main() {
  // Crear los tenedores (mutexes) y filósofos.
  tenedores := make([]*sync.Mutex, numFilosofos)
  filosofos := make([]*Filosofo, numFilosofos)
  for i := 0; i < numFilosofos; i++ {
    tenedores[i] = &sync.Mutex{}
  }

  for i := 0; i < numFilosofos; i++ {
    // Asignar tenedores izquierdo y derecho a cada filósofo.
    tenedorIzquierdo := tenedores[i]
    tenedorDerecho := tenedores[(i+1)%numFilosofos]

    // Crear e inicializar filósofos.
    filosofo := &Filosofo{
      id:              i + 1,
      tenedorIzquierdo: tenedorIzquierdo,
      tenedorDerecho:   tenedorDerecho,
    }
    filosofos[i] = filosofo

    // Iniciar una goroutine para que cada filósofo intente comer.
    go filosofo.intentarComer()
  }

  // Esperar un tiempo para permitir que los filósofos coman.
  time.Sleep(5 * time.Second)

  fmt.Println("Terminado.")
}
```

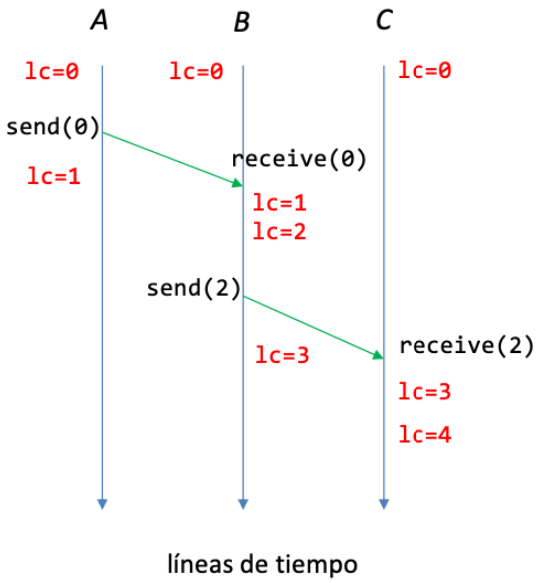
Ignacio Méndez Pérez
2023

Relojes lógicos: simulación de relojes físicos.

Un reloj lógico es un contador que es incrementado cuando ocurren eventos:

- Ejecuciones de **`send`**, **`broadcast`** y **`receive`**.

Suponemos que cada proceso tiene un reloj lógico, que inicialmente vale cero, y que cada mensaje contiene un *timestamp*.



Sea A un proceso, y sea lc un reloj lógico en A.
A actualiza el valor de lc según las siguientes reglas:

- Cuando A **envía** un mensaje, asigna al mensaje un *timestamp* con el valor actual de lc, y luego incrementa lc en 1.
- Cuando A **recibe** un mensaje con un *timestamp* ts, asigna lc el máximo entre lc y $ts + 1$, y luego incrementa lc en 1.

Semáforos distribuidos: ejemplo de descentralización

En sistemas de memoria compartida, representamos un semáforo s como un entero ≥ 0:

- La ejecución de $P(s)$ espera hasta que s sea positivo, y entonces decrementa s.
- La ejecución de $V(s)$ incrementa s.

Para tener semáforos distribuidos, necesitamos 3 cosas:

1. Una manera de contar el número de operaciones P y V.
2. Una manera de postergar la ejecución de las operaciones P.
3. Que los procesos cooperen para mantener el invariante ($s \geq 0$).

En cada proceso:

- Una cola local de mensajes, mq.
- Un reloj lógico lc, que cumple las reglas que vimos.

Para ejecutar una operación P o V, un proceso hace *broadcast* de una mensaje a todos los procesos, incluyéndose.

El mensaje contiene tres campos:

1. Identidad (número) del proceso emisor.
2. Una etiqueta (P o V) que indica el tipo de operación.
3. Un *timestamp* (el valor actual del lc del proceso emisor).

Cuando un proceso recibe un mensaje P o V, lo pone en su cola mq, en orden creciente de *timestamp*.

Detección de terminación e instantáneas globales

No es simple detectar cuando un programa distribuido ha terminado:

- El estado global no es visible para ningún procesador.
- Incluso si todos los procesadores están desocupados, puede haber mensajes en tránsito entre procesadores.

Sin embargo, hay varios algoritmo para hacerlo:

- **Circulación de un token**

  Los procesos forman un **anillo** y toda la comunicación viaja alrededor del anillo.
  El proceso `T[1]` envía a `T[2]`, `T[2]` envía a `T[3]` y `T[n]` envía a `T[1]`.

  En cualquier instante en el tiempo, cada proceso está ya sea activo o desocupado:
  - Está **desocupado** si ha terminado o está suspendido ejecutando una operación **`receive`**. Estado **azul**.
  - Después de recibir un mensaje, un proceso desocupado se vuelve **`activo`**. Estado **rojo**.

  Condición de terminación: **todo proceso está desocupado y no hay mensajes en tránsito**.

  Inicialmente, todos los procesos están activos: rojos.
  Cuando un proceso recibe el token, es porque está desocupado:
  - Se pinta de azul a sí mismo.
  - Pasa el token.
  - Espera a recibir otro mensaje.

  Si el proceso recibe más adelante un mensaje regular:
  - Se pinta de rojo a sí mismo.

  Cuando `T[1]` recibe el token, si está de azul, entonces anuncia terminación y se detiene.

El algoritmo para un grafo completo:

Cada proceso está pintado de rojo o azul, con todos los procesos inicialmente rojos.

Cuando un proceso recibe un mensaje regular, se pinta de rojo.

Cuando un proceso recibe el token, está bloqueado esperando recibir el próximo mensaje:

- Se pinta de azul (si es que no está de azul).
- Pasa el token.

- **El grafo de nodos es un árbol**

  n procesos son los nodos de un grafo no direccional, cuyas aristas representan los canales de comunicación.

  El algoritmo emplea un árbol de cobertura fijo (un subconjunto de las aristas del grafo):

  - El proceso `T[1]`, en la raíz, es responsable de detectar terminación.

  `T[1]` se comunica con los otros procesos para determinar sus estados, los mensajes empleados para esto los llamamos **tokens**.

  Todas los tokens en las hojas son inicialmente **blancos**.

  Si un proceso ha enviado un mensaje regular a otro proceso, al terminar envía un token **gris** a su padre, de lo contrario, envía un token **blanco**.

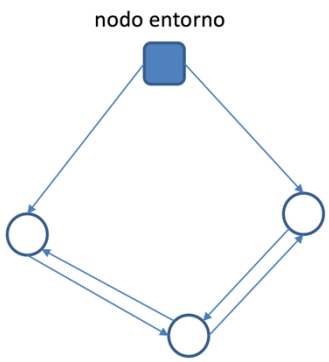
  El padre, al recibir el token **gris**, sabe que su hijo ha enviado un mensaje a otro proceso, y cuando el padre a su vez termina, envía un token **gris** a su padre.

  Finalmente, la raíz sabe que ha habido envío de mensajes cuando recibe un token **gris** de alguno de sus hijos, y pide a todos los nodos reiniciar el algoritmo de terminación.

- **Dijkstra-Scholten**

  Suponemos que los nodos forman un grafo direccional conectado (no necesariamente completo), y suponemos la existencia de un **nodo de entorno**:

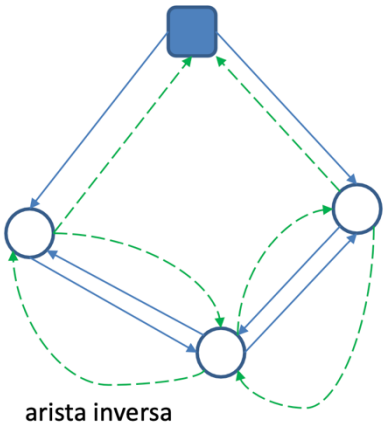
  - Sólo uno.
  - No tiene aristas de entrada.



- Puede llegar a cualquier otro nodo a través de alguna ruta en el grafo.
- Es responsable de detectar terminación.

El algoritmo ejecuta concurrentemente con la computación llevada a cabo en cada nodo:

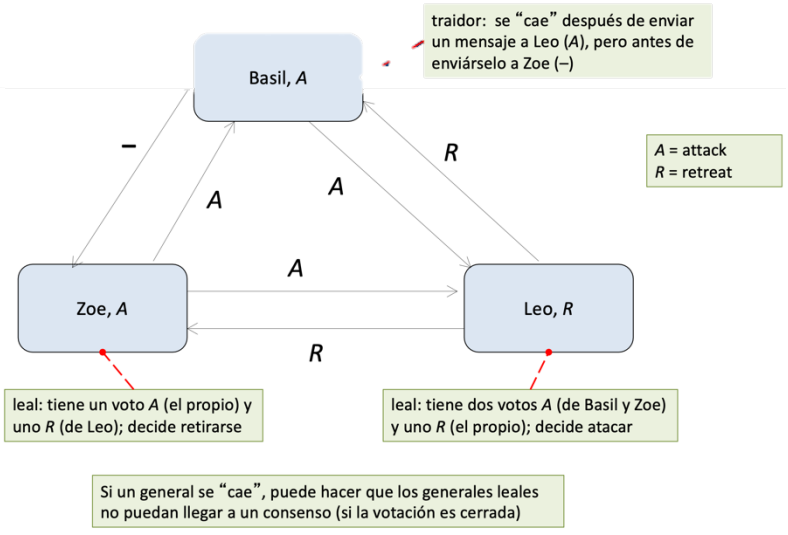
- Especifica sentencias adicionales que deben ser ejecutadas como parte del procesamiento de los mensajes regulares de la computación.
- Supone que por cada arista en el grafo del nodo i al nodo j, hay una arista inversa que transmite un mensaje especial llamado una señal de j a i.
- Supone que todos los nodos, excepto el nodo de entorno, están inicialmente inactivos, solo esperando a recibir mensajes



El problema de los generales bizantinos

La situación plantea que hay varios generales (nodos) en el campo de batalla, y todos ellos deben tomar una decisión común sobre si atacar o retirarse. Sin embargo, algunos de los generales pueden ser traidores y enviar mensajes engañosos a otros generales con el fin de causar la derrota de la fuerza atacante. El objetivo del problema es encontrar un algoritmo que permita a los generales leales llegar a un consenso a pesar de la presencia de generales traidores y mensajes engañosos.

- **Fallas simples ("caídas" de nodos)**:
  - Un nodo (traidor) simplemente **deja de enviar mensajes** en cualquier momento durante la ejecución del algoritmo
  - Suponemos que sabemos que el nodo falló, por ejemplo, hay un *timeout*.
- **Fallas bizantinas**:
  - Un nodo (traidor) puede enviar **mensajes arbitrarios**, no solo los requeridos por el algoritmo.
  - Son más difíciles de manejar que las fallas simples.



Leo debiera dar más peso al plan de Zoe que al de Basil.
En un sistema distribuido, un nodo puede no saber (directamente) quiénes son los traidores, pero lo que importa es que asegure que los traidores no impidan que los generales leales lleguen a consenso.

El algoritmo de 2 ruedas de los generales bizantinos

- **Primera rueda**: Cada general envía su propio plan. Al terminar, **`plan`** tiene los planes de cada general.

- **Segunda rueda**: Cada general envía lo que recibió de los otros generales acerca de sus planes.

Los generales leales siempre envía exactamente lo que recibieron.

Votación en 2 etapas:

1) Para cada general **`G`**, voto de mayoría entre el plan recibido directamente de **`G`** (**`plan[G]`**) y los planes informados para **`G`** recibidos de los otros generales (**`reportedPlan[G]`**).
   El resultado es almacenado en **`majorityPlan[G]`**.

2) La decisión final se obtiene de un segundo voto de mayoría de los valores almacenados en **`majorityPlan`**, incluyendo **`plan[myID]`**.

Leo

general	plan	informado por Basil	informado por Zoe	mayoría
Basil	*A*		–	*A*
Leo	*R*			*R*
Zoe	*A*	–		*A*
decisión				*A*

Zoe

general	plan	informado por Basil	informado por Leo	mayoría
Basil	–		***A***	*A*
Leo	*R*	–		*R*
Zoe	*A*			*A*
decisión				*A*

Ignacio Méndez Pérez
2023

RPC (*Remote Procedure Call*)

La idea es permitir a los programas llamar a procedimientos que residen en otros computadores **transparentemente**:

- El procedimiento que hace la llamada no debería tener que saber que el procedimiento llamado se está ejecutando en otro computador, o viceversa.
- El paso de mensaje necesario no tiene que ser visible para el programador.

Supongamos que un programa tiene acceso a una base de datos que permite agregar (*append*) datos a una lista, y luego devuelve una referencia a la lista actualizada:

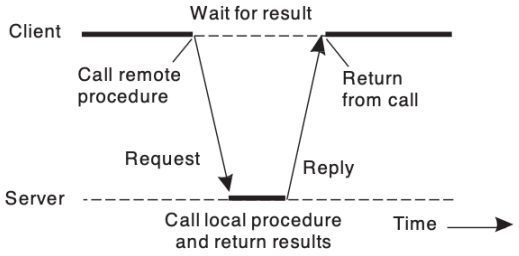
```
newlist ← append(data, dbList)
```

- En un sistema tradicional, **append** es tomada desde una librería por el *linker* e insertada en el programa objeto.
- Luego es llamada de la manera habitual → colocando sus parámetros en el *stack*.
- El programador no conoce los detalles de implementación de **append**, que es como debería ser.

En **RPC**, cuando **append** es un procedimiento remoto, una versión especial de **append** es ofrecida al cliente que llama.

Un ***stub* cliente**, que es llamado usando la secuencia de llamada normal, pero que no ejecuta la operación *append*:

- Empaca los parámetros en un mensaje, y pide que el mensaje sea enviado al servidor.
- Luego de llamar a **`send`**, el *stub* llama a **`receive`** y se bloquea hasta que llegue la respuesta.



Cuando el mensaje llega al servidor, el sistema operativo local pasa el mensaje a un ***stub* servidor**:

- Código que transforma solicitudes que llegan por la red en llamadas a procedimientos locales (típicamente, el *stub* ha llamado a **`receive`** y está bloqueado esperando que lleguen mensaje).

El stub desempaca los parámetros y llama al procedimiento del servidor de la manera usual:

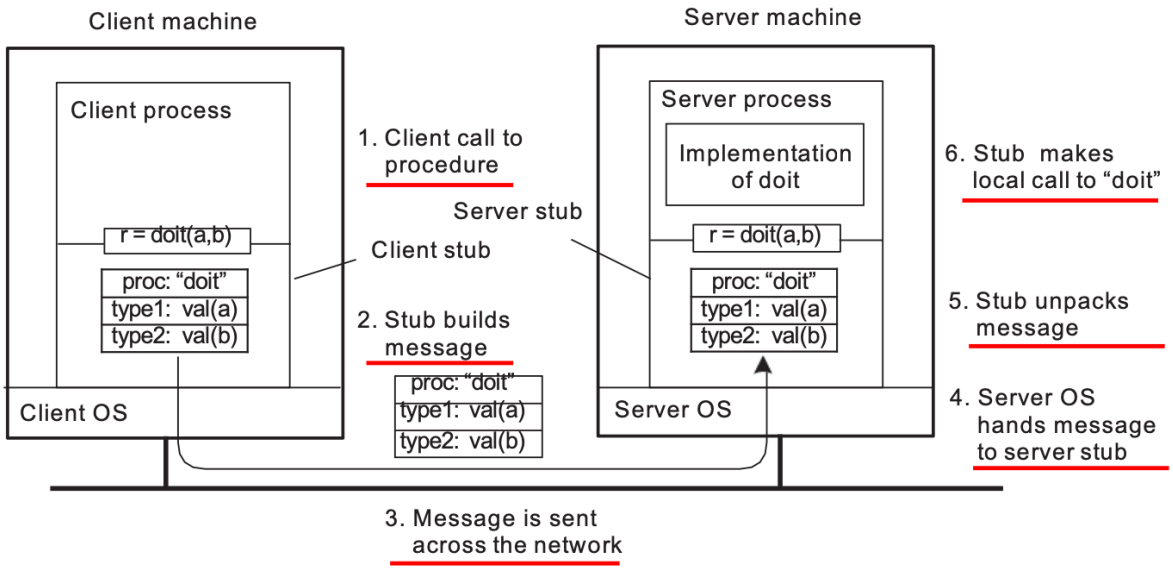
- Para el servidor, es como si estuviera siendo llamado directamente por el cliente: los parámetros y dirección de retorno están en el *stack*.

El servidor hace su trabajo y luego devuelve los resultados al *stub*.

El stub empaca el resultado en un mensaje y llama a **`send`** para enviarlo de vuelta al cliente:

- Y luego llama a **`receive`** de nuevo para esperar a la próxima solicitud.

Pasos del "viaje de ida" al llamar al procedimiento remoto **`doit(a,b)`**, en que el parámetro **`a`** es de tipo **`type1`** y **`b`** es de tipo **`type2`**.



Pasos del "viaje de vuelta", que no se muestran en la figura:

7. Server stub packs result in a message and calls its local OS.
8. Server's OS sends message to client's OS.
9. Client's OS gives message to client stub.
10. Client stub unpacks result and returns it to client.

Cuando el mensaje llega al cliente, el sistema operativo local lo pasa al *stub* cliente a través de la operación **`receive`**, que fue llamada antes, y el cliente es desbloqueado.

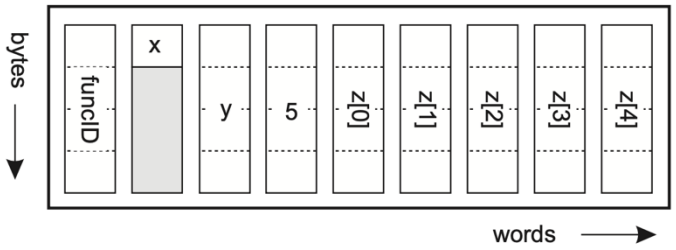
El stub inspecciona el mensaje, desempaca el resultado, lo pasa al programa que hizo la llamada y retorna de la forma usual.

En este punto, el programa cliente que hizo la llamada sabe que agregó datos a la lista, y no sabe nada más:

- No tiene idea de que el trabajo fue hecho remotamente en otro computador.
- Desde su punto de vista, a los servicios remotos se tiene acceso haciendo llamadas a procedimientos ordinarias (locales), y no llamando a **`send`** y **`receive`**.

El protocolo RPC indica que:

- Un carácter se transmite en el byte de más a la derecha de una palabra.
- Un *float*, como una palabra entera.
- Un arreglo, como un grupo de palabras igual al largo del arreglo, precedido por una palabra que da el largo.

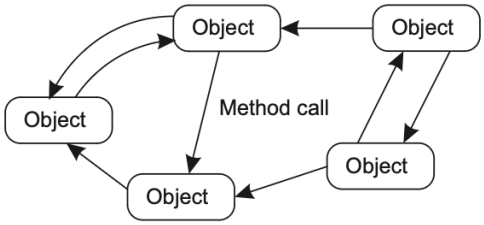


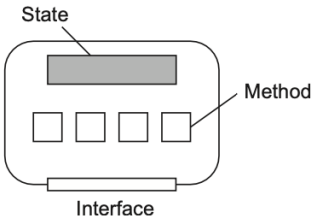
El stub cliente sabe que debe usar el formato de la figura, y el stub servidor sabe que los mensajes que lleguen tendrán ese formato.

Referencias globales: Referencias que tienen significado tanto para el proceso que llama como para el que es llamado.

Arquitectura basada en objetos:

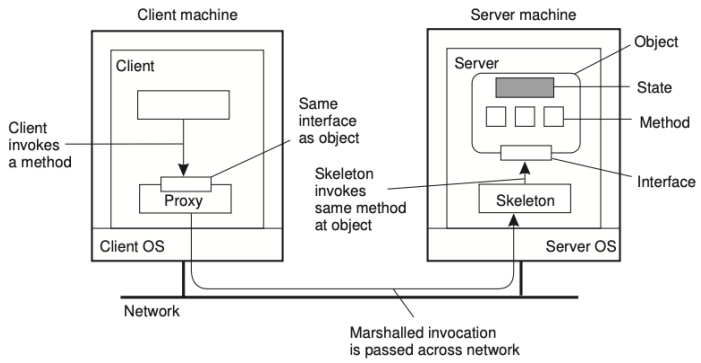
- Cada objeto es un componente.
- Los componentes están conectados a través de un mecanismo de llamadas a procedimientos.
- En un sistema distribuido, la llamada puede ocurrir a través de una red.





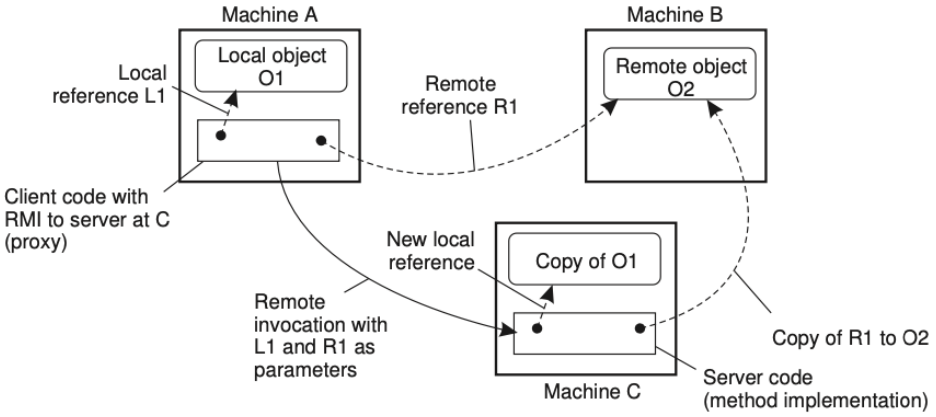
Hacer la separación entre la interfaz y el objeto que la implementa, permite que la interfaz esté en un computador y el objeto mismo en otro, un objeto distribuido, como en la fig.

Cuando un cliente **es asociado** (*binds*) a un objeto distribuido, se carga, en el espacio de direcciones del cliente, una implementación de la interfaz del objeto, un **proxy** (*stub* cliente, en RPC), a cargo del *marshaling* de la llamada, el objeto mismo reside en el servidor y ofrece la misma interfaz, que en este caso es llamada por un *skeleton* (stub servidor), que primero hizo *unmarshaling* del mensaje del cliente y luego también envía el resultado de vuelta al cliente.



Usar sólo objetos distribuidos y referencias globales puede ser ineficiente:

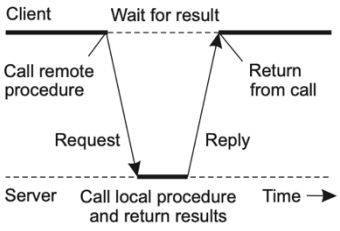
- Cada llamada genera una solicitud entre diferentes espacios de direcciones, o computadores (pensemos en objetos pequeños, p.ej., *Booleans*)
- Al hacer una llamada con una referencia a un objeto como parámetro, la referencia es pasada como parámetro por valor sólo si se refiere a un objeto remoto (y el objeto es pasado por referencia)
  ... en cambio, si es a un objeto local al cliente, se pasa todo el objeto por valor
  P.ej., en la fig., el cliente está en A y el servidor en C
  ... el cliente tiene una referencia local a O1 y otra a O2, que reside en B
  ... y usa ambas como parámetros al llamar al programa servidor en C, pasando una copia de O1 y una copia de la referencia a O2.

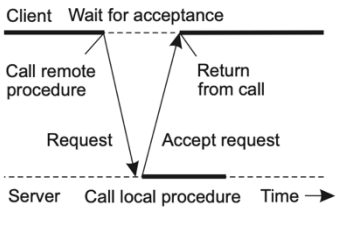


Arquitecturas orientadas a servicios (SOA)

RPC sólo de ida:

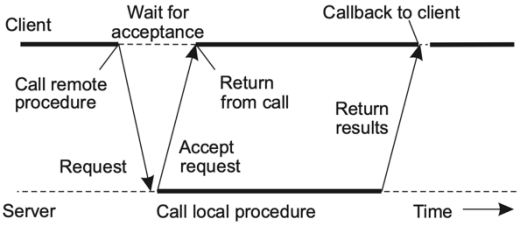
- El cliente sigue ejecutando inmediatamente después de hacer la llamada, sin esperar la aceptación del servidor.
- No se sabe si llamada será procesada.





RPC asíncrono diferido

- El cliente llama al servidor, espera su aceptación y continúa.
- Cuando el resultado está disponible, el servidor envía una respuesta que produce callback en el cliente.

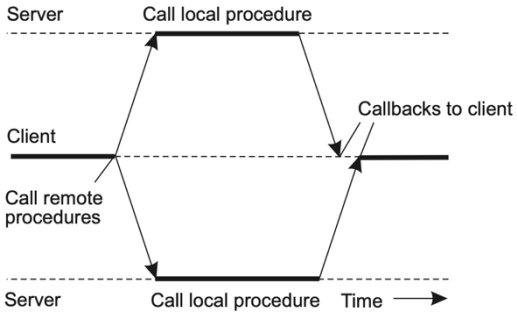


Un callback es sólo una función definida por el usuario que es llamada cuando ocurre un evento especial, p.ej., llega un mensaje.

Ignacio Méndez Pérez
2023

RPC *Multicast*

El cliente envía solicitud a dos servidores, que la procesan independientemente y el paralelo. Al terminar, el resultado es enviado al cliente que ejecuta un callback útil.



Rendezvous

Mecanismo o técnica que permite que varios procesos o entidades se encuentren y coordinen su actividad en un punto específico en el espacio o el tiempo. El objetivo del *rendezvous* es sincronizar y asegurar que ciertos eventos ocurran de manera ordenada y controlada entre los diferentes componentes de un sistema distribuido.

Separación entre procesamiento y coordinación

La coordinación ocurre de manera directa:
- El acoplamiento referencial se manifiesta, p.ej., cuando un proceso sólo puede comunicarse cuando conoce el nombre o identificador del otro proceso.
- El acoplamiento temporal significa que ambos procesos deben estar corriendo (p.ej., hablar por teléfono).

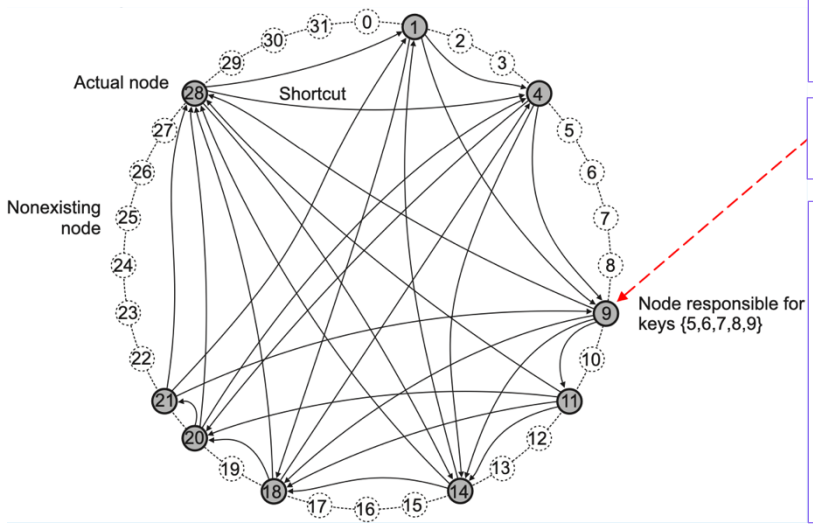
Los procesos no necesitan estar ejecutándose al mismo tiempo para que puedan comunicarse:
- Los mensajes se ponen en un *mailbox* compartido
- Es necesario direccionar explícitamente el *mailbox*.

	Temporally coupled	Temporally decoupled
Referentially coupled	Direct	Mailbox
Referentially decoupled	Event-Based	Shared data space

Los procesos no se conocen mutuamente:
- Un proceso sólo puede **publicar** una **notificación** describiendo la ocurrencia de un evento.
- Los procesos se pueden **suscribir** a un tipo específico de notificaciones.

Los procesos se comunican sólo a través de **tuplas**:
- Datos estructurados con un cierto número de campos, similar a una fila en una tabla de una base de datos.
- Los procesos pueden poner cualquier tipo de tupla en el espacio compartido.

Red P2P sistema CHORD:



Un dato con clave k, de m bits, es almacenado en el nodo con menor identificador $id \geq k$, llamado el **sucesor** de la clave k. En este ejemplo, $m = 5$ y los nodos almacenados son $\{1,4,9,11,14,18,20,21,28\}$, en la práctica, claves e identificadores pueden llegar a tener entre 100 y 200 bits de largo.

El nodo con clave 9 es el sucesor de las claves 5, 6, 7, 8 y 9.

Si el nodo 9 busca al nodo que almacena la clave 3 (el nodo 4), entonces:
- El nodo 9 tiene atajos, a los nodos 11, 14, 18 y 28.
- El nodo 9 envía la solicitud al nodo conocido más lejano, el 28.
- El nodo 28 tiene 3 atajos, a los nodos 1, 4 y 14, por lo que envía la solicitud al nodo 1.
- El nodo 1 sabe que el sucesor es el nodo 4 y que por lo tanto éste debe almacenar la clave 3, por lo que le envía la solicitud al nodo 4.

- **Asignación de claves y nodos**: Cada nodo y recurso en el sistema Chord se asigna una clave de identificación mediante una función de hash. Las claves se organizan en un anillo lógico unidimensional, y cada nodo en el anillo es responsable de un rango de claves.

- **Determinar el nodo sucesor**: El nodo que inicia la búsqueda primero debe determinar su sucesor en el anillo. Para ello, puede comparar su clave de identificación con la clave de los demás nodos en el anillo hasta encontrar el nodo cuya clave sea la más cercana a la suya en sentido de las agujas del reloj. Una vez que encuentra el sucesor, puede iniciar la búsqueda en ese nodo.

- **Enrutamiento**: El nodo emisor envía la solicitud de búsqueda a su sucesor y, si el recurso no está en el nodo sucesor, este reenviará la solicitud al nodo cuya clave de identificación sea la más cercana al recurso buscado. El proceso de enrutamiento continúa hasta que la solicitud llegue al nodo responsable del recurso o, en caso de que el recurso no exista en la red, hasta que vuelva al nodo que inició la búsqueda. Este proceso de enrutamiento se realiza utilizando una búsqueda binaria, lo que garantiza una complejidad de tiempo de $O(\log n)$, donde n es el número de nodos en el sistema.

- **Devolución del resultado**: Una vez que la búsqueda llega al nodo responsable del recurso, ese nodo proporciona el recurso solicitado al nodo que inició la búsqueda a través de la misma ruta utilizada para la búsqueda.

Procesos

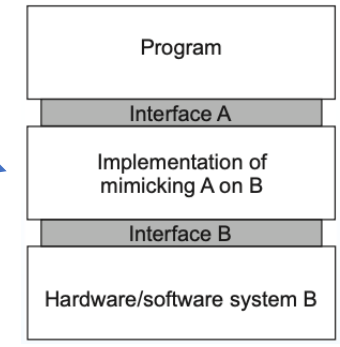
Virtualización: Tecnología que permite la ejecución de varias máquinas virtuales sobre una máquina física con el objetivo de aprovechar al máximo los recursos de un sistema y que su rendimiento sea mayor.

Los procesos y los *threads* son una forma de hacer más cosas al mismo tiempo:

- Nos permiten construir programas que parecen estar ejecutándose simultáneamente (en un computador con una única CPU esta ejecución simultánea es una ilusión).

La **virtualización** permite extender o reemplazar una interfaz existente para imitar el comportamiento de otro sistema.

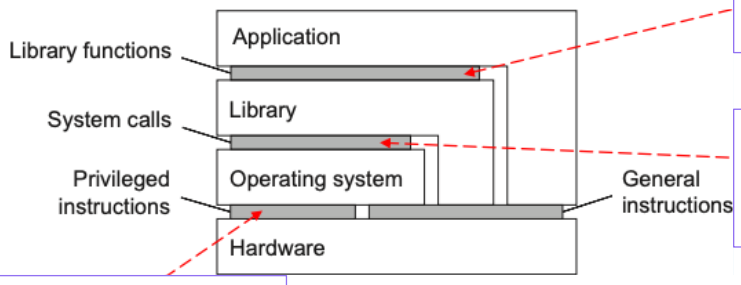
Por ejemplo, virtualización del sistema A sobre el sistema B.



Organización general de virtualizar el sistema A sobre B

Ignacio Méndez Pérez
2023

Los sistemas computacionales generalmente ofrecen cuatro tipos de **interfaces**, en tres niveles.



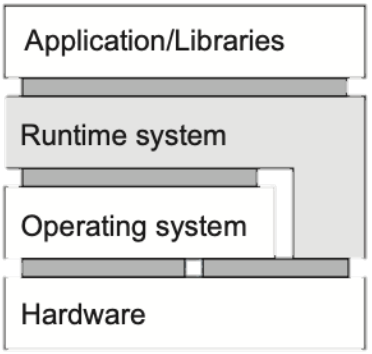
Una **API** formada por llamadas a funciones de biblioteca.

Los ***system calls*** son la interfaz ofrecida por el sistema operativo (a veces escondidas por una API).

La ***ISA*** (*instruction ser architecture*) es la interfaz entre el *software* y el *hardware*, compuesta por las instrucciones privilegiadas y las instrucciones generales.

Podemos implementar un sistema de *runtime* que esencialmente ofrezca a las aplicaciones un set de instrucciones abstracto:

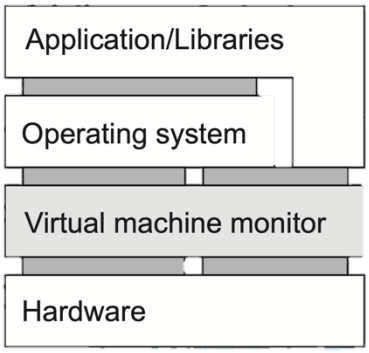
- Las instrucciones pueden ser interpretadas (como en el caso del entorno *runtime* de Java)
- ... o emuladas (p.ej., al correr aplicaciones Windows en Unix); el emulador tiene que imitar el comportamiento de los *system calls* (nada trivial)



Esta virtualización es conocida como *máquina virtual para un proceso* (***PVM***)

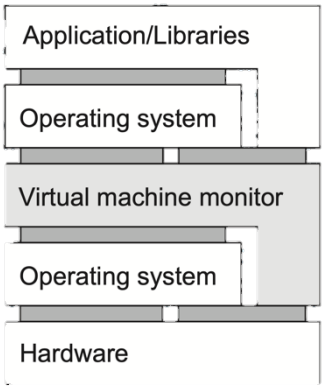
También podemos ofrecer un sistema implementado como una capa que cubre al hardware original, y que ofrece como interfaz el set de instrucciones completo de ese mismo hardware—**monitor de máquina virtual nativa** ("nativa", porque es implementada directamente sobre el hardware)

La interfaz puede ser ofrecida simultáneamente a diferentes programas → es posible tener diferentes sistemas operativos invitados corriendo independiente y concurrentemente en la misma plataforma.



Un monitor de máquina virtual nativa tiene que ofrecer y regular el acceso a varios recursos (p.ej., almacenamiento externo y redes) debe implementar drivers para esos dispositivos.
Un **monitor de máquina virtual *hosted*** corre encima de un sistema operativo anfitrión:

- El monitor puede hacer uso de las facilidades existentes ofrecidas por este anfitrión.
- en lugar de correr como una aplicación a nivel de usuario , se le dan privilegios especiales.



Ignacio Méndez Pérez
2023

Desde el punto de vista de los sistemas distribuidos, la aplicación más importante de la virtualización está en ***cloud computing***:

- En lugar de arrendar una máquina física, el proveedor de cloud arrienda una máquina virtual (IaaS) que puede, o no, estar compartiendo una máquina física con otros clientes.
- La virtualización permite que haya un aislamiento casi completo entre clientes, quienes tienen la ilusión de que arrendaron una máquina física dedicada.

- Un **servidor iterativo** maneja directamente la solicitud y, si es necesario, devuelve una respuesta al cliente.
- Un **servidor concurrente** pasa la solicitud a un *thread* separado u otro proceso, que es responsable de atender la solicitud y enviar la respuesta. Después queda inmediatamente a la espera de la próxima solicitud.

Daemon: Tipo de programa de *software* que se ejecuta en segundo plano en un sistema operativo y generalmente no tiene una interfaz de usuario directa. Los *daemons* son procesos que realizan diversas tareas y servicios esenciales para el funcionamiento del sistema, como administrar servicios de red, tareas de programación, procesamiento de solicitudes, entre otras funciones.

Objetos transitorios existen sólo mientras el servidor existe, pero posiblemente por menos tiempo; p.ej., una calculadora, o una copia en memoria de un archivo *read-only.*

Una política es crear el objeto transitorio a la primera solicitud de llamada y destruirlo en cuanto deje de haber clientes vinculados al objeto:

- el objeto ocupará los recursos del servidor sólo mientras sea realmente necesario.
- ... pero una invocación va a tomar más tiempo porque hay que crear el objeto primero.

Otra política es crear todos los objetos transitorios cuando el servidor es inicializado:

- el costo es que se consumen recursos aun cuando ningún cliente use el objeto.

La **política de activación** son las decisiones sobre cómo invocar a un objeto:

- p.ej., el objeto debe ser traído al espacio de direcciones del servidor (activado), antes de que pueda ser invocado.
- ... o el objeto puede seguir existiendo después de que su adaptador ha terminado.

Ignacio Méndez Pérez
2023

Computación basada en *cluster*

Un supercomputador construido usando tecnología *off-the-shelf*, conectando una colección de computadores a través de una red de alta velocidad (desde que esto se volvió económica y técnicamente atractivo).

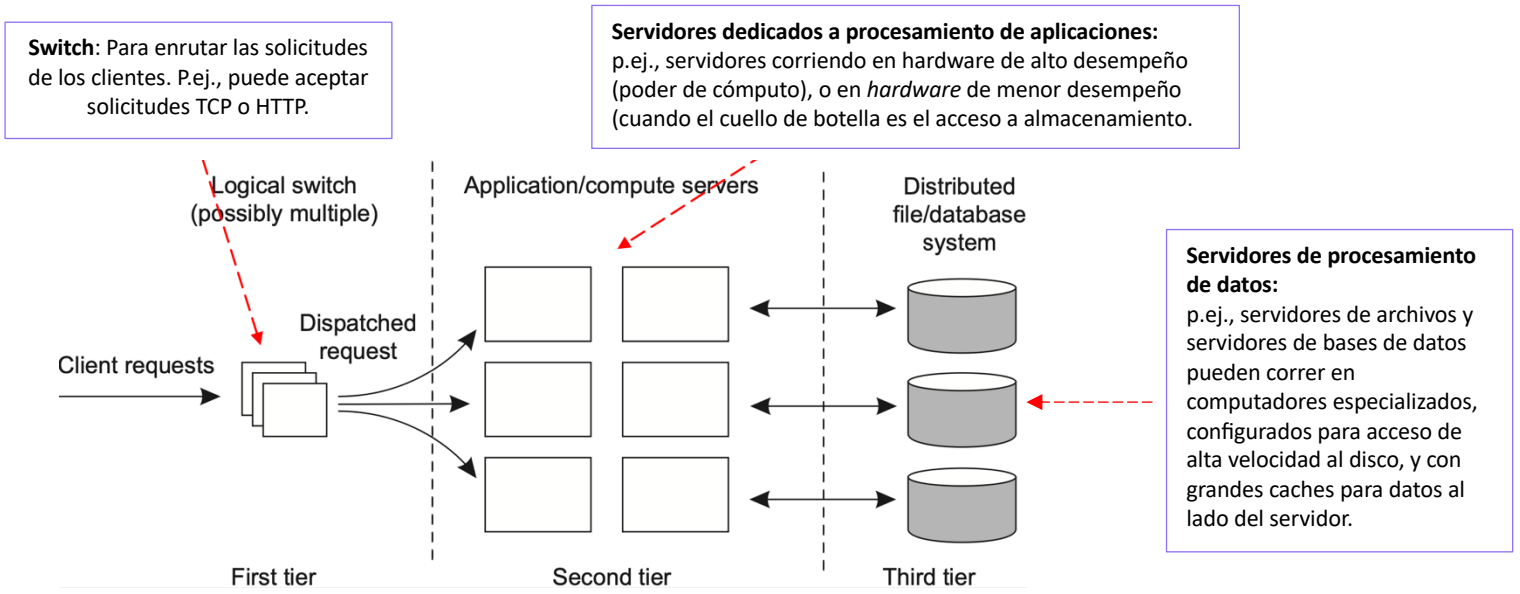
Para programación paralela: un mismo programa es ejecutado en paralelo en múltiples computadores.
P.ej., una **colección de nodos de cómputo** controlados y accedidos mediante un único **nodo maestro**.

Nodo maestro:
- Maneja la asignación de nodos a un programa particular.
- Mantiene una cola de trabajos (*jobs*) por hacer.
- Ofrece una interfaz a los usuarios.
- Ejecuta el middleware necesario para la ejecución de programas y la gestión del *cluster*.

Nodos de cómputo:
- Sistema operativo estándar extendido con funciones de *middleware* (comunicación, almacenamiento, tolerancia a fallas, etc.).

***Clusters* de servidores**

Colección de computadores conectados a través de una red, en que cada computador "corre" uno o más servidores.



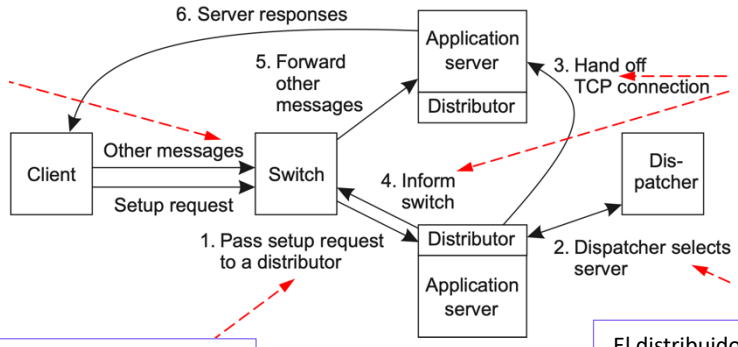
Ignacio Méndez Pérez
2023

Switch

Un objetivo de diseño de un cluster de servidores es esconder el hecho de que hay múltiples servidores.
Esta transparencia de acceso se ofrece mediante un único punto de acceso, el **switch**, un computador dedicado que tiene una única dirección de red
(Por escalabilidad y disponibilidad un cluster podría tener múltiples puntos de acceso).

El switch juega un papel importante en distribuir la carga entre los servidores ("*load balancer*").
Puede utilizar la política *round robin*.

El diseño del switch es crucial; puede convertirse en un cuello de botella.
Conviene distribuir la tarea de inspeccionar el contenido entre varios servidores.



La tarea del distribuidor e manejar el TCP *handoff* una vez que el servidor ha sido seleccionado.

Cuando llega una solicitud, se la pasa a un distribuidor que corre en uno de los servidores.

El distribuidor pasa la solicitud a un despachador, que inspecciona el contenido para decidir cuál servidor de aplicaciones es el más apropiado.

Coordinación

Los algoritmo de elección tratan de encontrar al proceso con el identificador numéricamente mayor para designarlo como **coordinador**.

Los que los procesos no saben es cuáles procesos están actualmente funcionando y cuáles han dejado de funcionar.

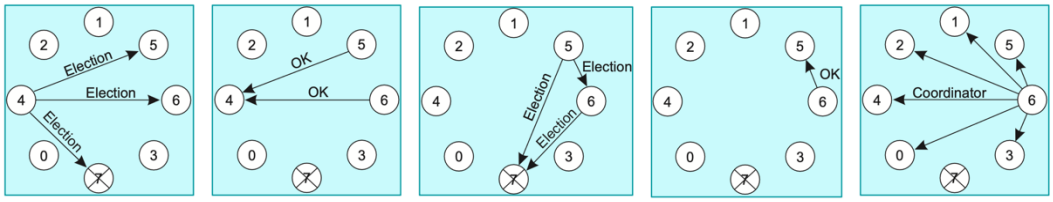
El objetivo de un algoritmo de elección es que al finalizar, todos los procesos estén de acuerdo en cuál es el **nuevo coordinador**.

Ignacio Méndez Pérez
2023

Algoritmo del *bully*

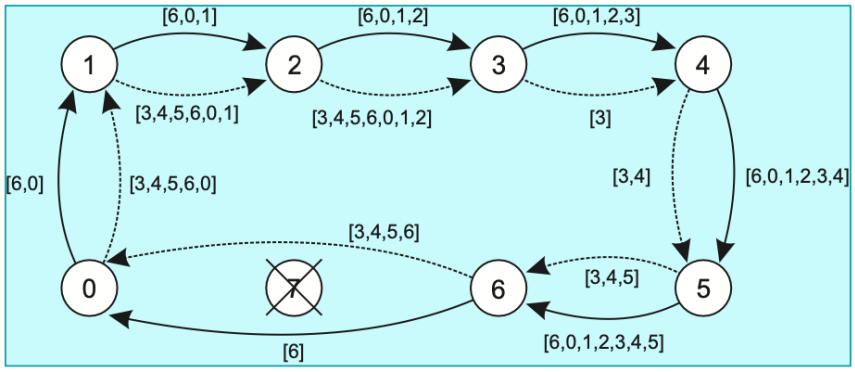
Cuando cualquier proceso k se da cuenta de que el coordinador no está respondiendo, inicia una elección:

1. k envía un mensaje $Election$ a todos los procesos con identificadores $> k$.
2. Si nadie responde, entonces k gana la elección y se vuelve coordinador.
3. Si uno de los procesos responde, entonces ese proceso se hace cargo y la tarea de k está terminada.



Un algoritmo basado en un anillo lógico (que usa un token)

- Cada proceso sabe quién es su sucesor.
- Cuando un proceso se da cuenta de que el coordinador no está funcionando, construye un mensaje $Election$ que contiene una lista con su propio ID y lo envía a su sucesor.
- Si el sucesor no está funcionando, entonces el emisor lo salta y va al siguiente proceso en el anillo, y así hasta encontrar un proceso que esté funcionando.
- Cada proceso que recibe un mensaje lo reenvía a su sucesor (o al sucesor de su sucesor, etc.), agregando su ID a la lista.
- Finalmente, el mensaje vuelve al proceso que inició la elección, que reconoce su propio ID en la lista.
- El proceso cambia el mensaje a $Coordinator$ y lo envía nuevamente, para informar a todos quién es el nuevo coordinador y cuáles son los procesos funcionando.



El algoritmo iniciado por los procesos 3 y 6 simultáneamente.