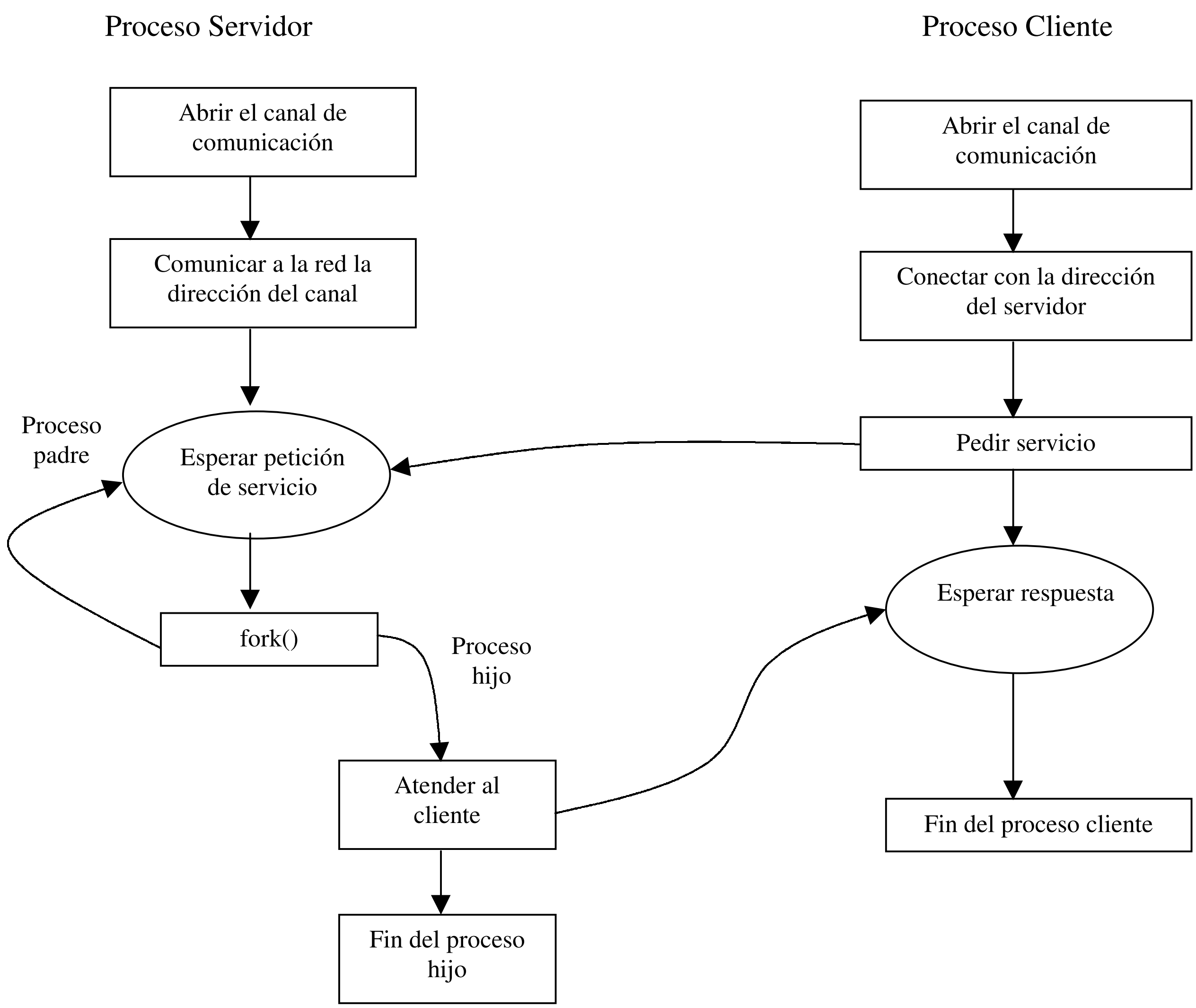
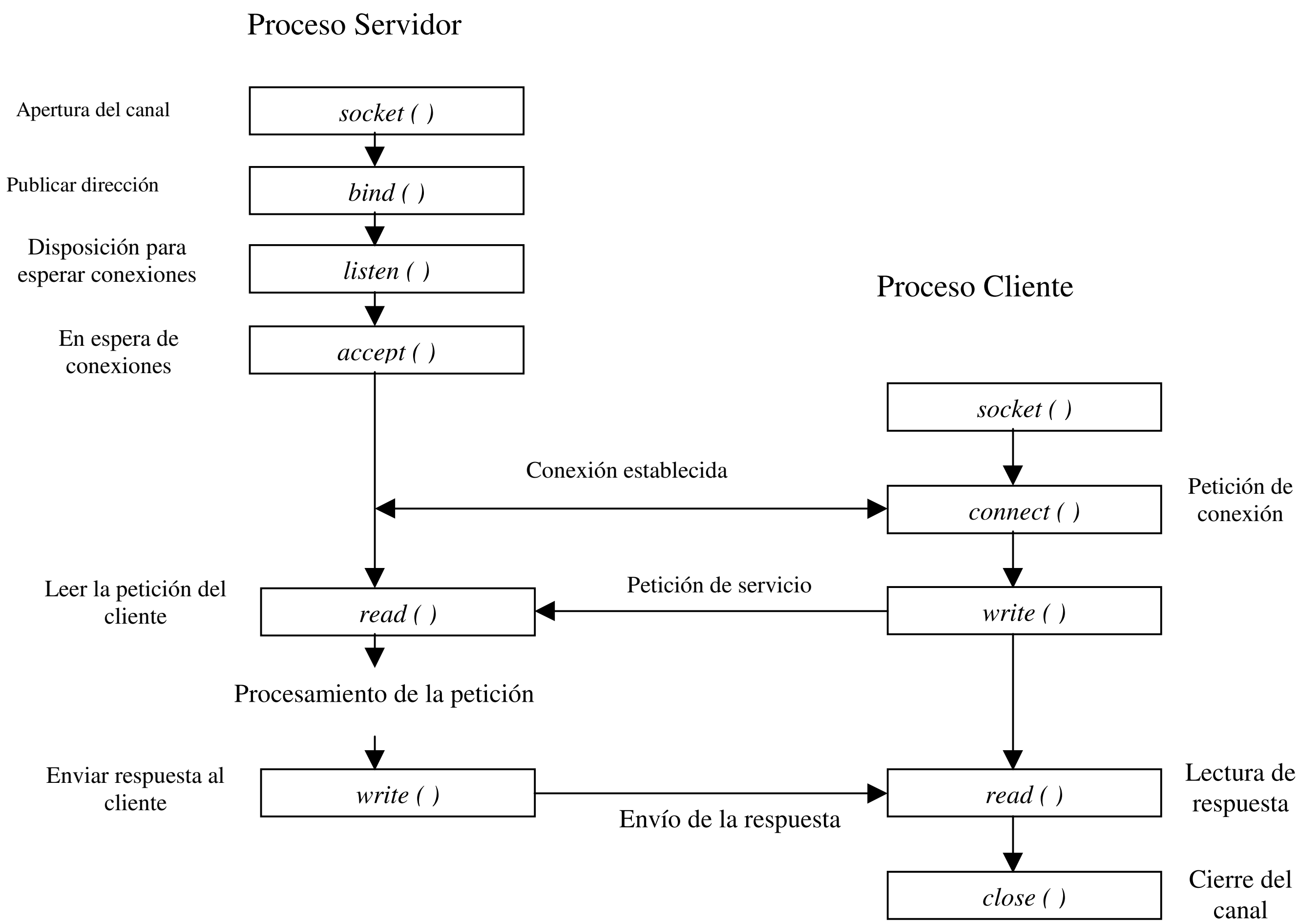
Administración de Procesos en Sistemas Operativos Distribuidos

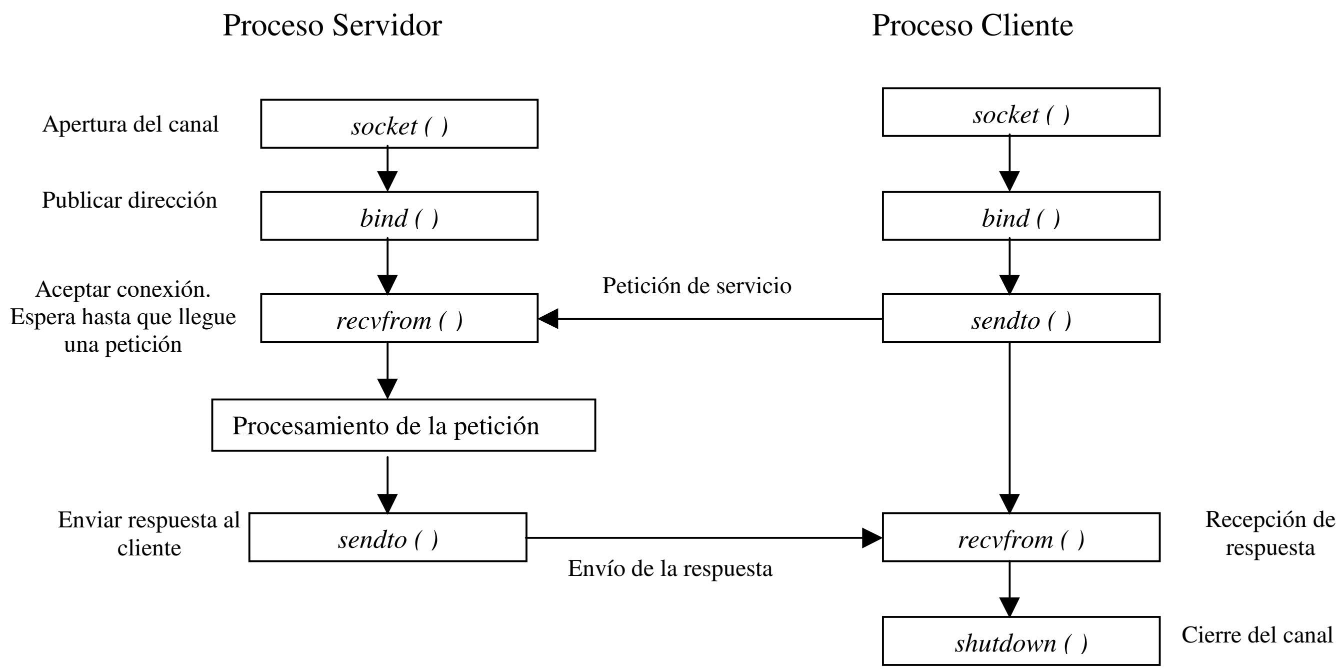
Sockets



Comunicación orientada a conexión (TCP)



Comunicación no orientada a conexión (UDP)



Compilación de programas en C que usan sockets

- Includes:

```
#include <sys/types.h>
#include <sys/socket.h>
```

- Utilizar `cc` con las opciones `-lsocket` y `-lnsl`

p.e.: `$cc -lsocket -lnsl programa.c -o programa`

- Todas las llamadas de sockets regresan -1 si no tuvieron éxito y establecen la variable externa errno

Microkernels y Kernels monolíticos

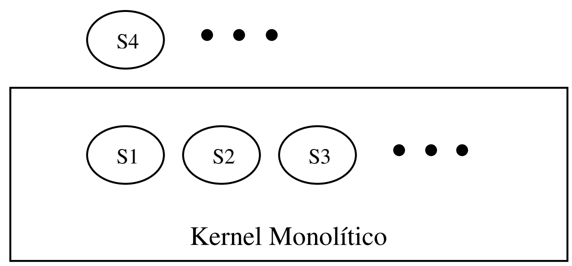
El **núcleo** o **kernel** de un S. O. es un programa que se distingue porque su código es ejecutado con privilegios de acceso a los dispositivos físicos de la computadora host: memoria, registros.

Se encarga también de separar y proteger los espacios de memoria de todos los procesos.

Un proceso usuario puede accesar recursos del núcleo por medio de llamadas al sistema (system call traps).

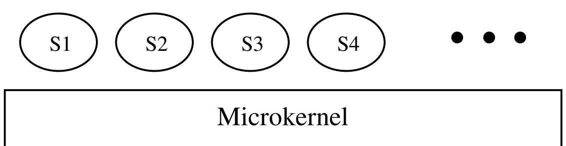
Los núcleos pueden ser:

- **Monolíticos**, cuando realizan todas las funciones básicas del sistema operativo (ej. Unix).



S1, S2, S3... : Servidores estáticos dentro del kernel (espacio de sistema)
S4 ... : Servidores fuera del kernel (espacio de usuario) cargados dinámicamente

- **Microkernels**, cuando en él existen solamente las funciones más básicas: comunicación entre procesos, servidores y memoria. El resto es provisto por servidores cargados dinámicamente (ej. Mach).



S1, S2, S3, S4 ... : Servidores fuera del kernel (espacio de usuario) cargados dinámicamente

Algunas ventajas:

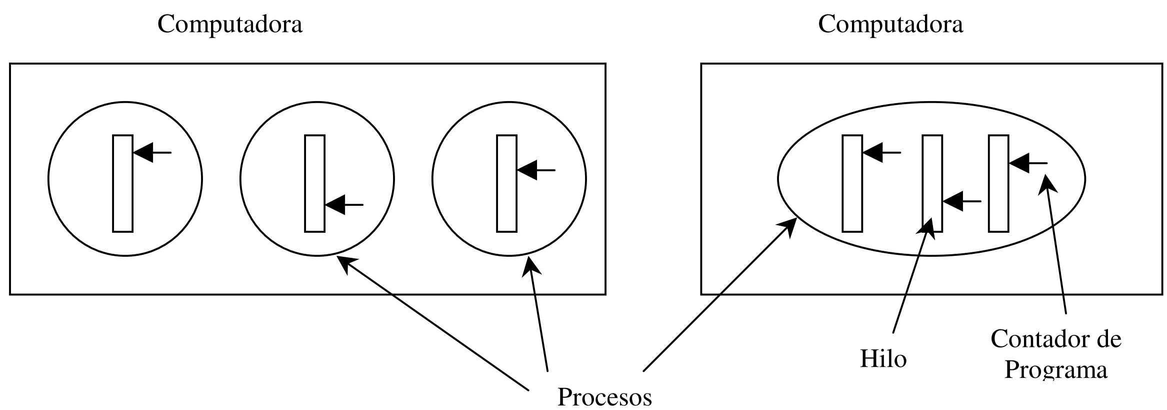
- Refuerza la modularidad
- Facilita la distribución de las tareas en diferentes máquinas
- Facilita la depuración por tener componentes más pequeños
- Permite una mejor utilización del paradigma de objetos
- Facilita la portabilidad (puede programarse en lenguajes de alto nivel, como C/C++)

Procesos e hilos

Un **proceso pesado** (o proceso) es un programa que incluye un medio ambiente de programación:

+ Un espacio de direcciones.
+ Recursos de comunicación y sincronización (semáforos, puertos) entre hilos.

Un **proceso multi-hilos** es aquel formado por varios **hilos** (o procesos ligeros), que son actividades secuenciales que comparten el mismo ambiente de programación que su padre.



Es preferible tener un proceso multi-hilos que múltiples procesos:

+ Más barato crear hilos que procesos.
+ Más barato cambio de contexto entre hilos.
+ Los hilos dentro de un proceso pueden compartir datos y recursos.

-Los hilos no están protegidos dentro de un mismo proceso.

Paquetes implementando hilos: C Threads (Mach), GNU Threads (SunOS), P(osix) Threads (OSF).

Esquemas de trabajo con hilos en servidores:

Modelo servidor-trabajador : un hilo principal se encarga de atender las peticiones externas y distribuye el trabajo a otro(s) hilo(s)

Modelo de equipo: Todos los hilos resuelven el trabajo en forma paralela

Modelo de entubamiento: Los hilos se ejecutan en pipe-lining

Hilos en el espacio de usuario

+Rápido cambio de contexto
+Diferentes algoritmos de planificación por proceso (conteniendo hilos)
+Buena escalabilidad

-Malos para implantar llamadas al sistema con bloqueos
-No puede implantarse planificación Round Robin

Hilos en el espacio del kernel

+Mejor implantación para un hardware específico
+Los hilos pueden "reciclarse"
+Pueden implantarse buenos algoritmos de planificación

-El cambio de contexto es más lento por ser una llamada al sistema
-La sincronización es más costosa (semáforos)
-Poca flexibilidad en la planificación (buenos, pero pocos)

Sincronización entre procesos

Sincronización

Necesaria para:

+ Ordenar eventos.
+ Consistencia de datos distribuidos.
+ Protocolos.
+ Varios algoritmos distribuidos.

Existen 2 tipos de sincronización:

+ Externa, cuando hay una fuente externa de tiempo.
+ Interna, cuando la sincronización se hace en base a los relojes locales.

! Problema sincronización en una red: el tiempo de envío de un mensaje a través de la red es variable !

Métodos de sincronización de relojes físicos

1. Método de Cristian.

   Un servidor central provee el tiempo por petición del usuario.

   img

Si P pide el tiempo y recibe el valor t, puede poner su tiempo en $t + T_{trans}$.

Problema: $T_{trans} = T_{min} + x$ es variable.

Método:
a) Calcular T_{round} al enviar m_r y recibir m_t .

b) Escoger un valor entre [t + Tmin ; t + T round –Tmin]

Problema: si falla el servidor

2. Algoritmo de Berkeley.

   Una computadora maestra pide a sus esclavos sus relojes y efectúa la sincronización

img

Método:

a) El maestro pide varios relojes
b) Usando T round , el maestro estima los relojes locales.

c) El promedio de los relojes es tomado como base, si éste no se diferencía más de un valor máximo.

d) El maestro envía los desfases de tiempo a los esclavos.

3. Protocolo de tiempo de red (NTP).

Características:

+ Provee sincronización de tiempos en Internet con UTC (Coordinated Universal Time).
+ Provee servicio confiable tolerante a fallas.
+ Permite a los clientes el acceso frecuente.
+ Provee protección contra interferencias.

El servicio es proporcionado por diversos servidores.

img

Método:

(a) Los servidores se sincronizan entre sí en 3 modos:

+ Multicast.
+ Llamada a procedimiento.
+ Simétrico.

(b) El desfase entre los dos relojes vale

$$o_i - \frac{d_i}{2} \le o = o_i + \frac{t'-t}{2} \le o_i + \frac{d_i}{2}$$

donde

t	Tiempo re transmisión de *m*
t'	Tiempo de transmisión de *m'*
$o_i = \dfrac{T_{i-2} - T_{i-3} + T_{i-1} - T_i}{2}$	Estimación simple offset
di = t + t'	Precisión de la estimación

(c) Varios pares <*oi, di*> son retenidos para estimar *o*

Métodos de sincronización de relojes lógicos

Tiempos lógicos

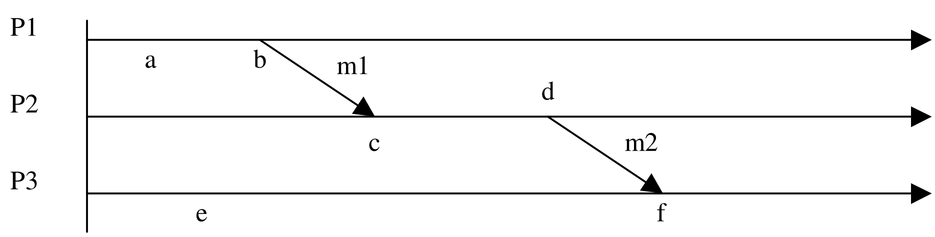
Definición: La relación de causalidad $x \xrightarrow{p} y$ indica que los eventos *x* y *y* ocurren en el proceso *p* y *x* ocurre antes de y.

Definición. La relación *happenedbefore* ($\rightarrow$) está definida por:

HB1 : Si ∃ proceso *p* | $x \xrightarrow{p} y$ entonces $x \rightarrow y$

HB2 : Para cualquier mensaje *m*, *send(m)* $\rightarrow$ *rcv(m).*

HB3 : Si *x; y; z* son eventos tales que x $\rightarrow$ y; y $\rightarrow$ z, entonces x $\rightarrow$ z. (transitividad)



Nota: La relación $x \rightarrow y$ NO implica que el evento *x* genera al evento *y*. La causalidad es potencial.

Definición. Si la relación *happenedbefor* e no se cumple entre *x* y *y*, se dice que $x \not\rightarrow y$.
Si $x \not\rightarrow y$ y $y \not\rightarrow x$ se dice que *x* y *y* son concurrentes: x || y.

Definición. Un reloj lógico es un contador que se incrementa monotónicamente según la relación *happenedbefore.*

Algoritmo de Lamport para cálculo de relojes lógicos

Sea *Cp* el reloj lógico del proceso *p*, *Cp (a)* la estampilla de tiempo del evento *a* en *p* y *C(b)* la estampilla de tiempo del evento *b* en cualquier evento.

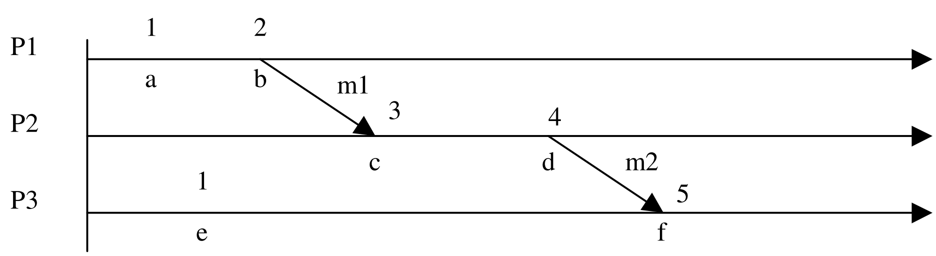
LC1 : *Cp* es incrementado antes de cualquier evento en el proceso *p*: $C\,p = C\,p + 1$.

LC2 :

1. Cuando un proceso *p* envía un mensaje *m*, añade a éste la estampilla de tiempo $t = C\,p$.

2. Cuando un proceso *q* recibe un mensaje (*m, t*), se hace $C\,q = max(\ C\,q\ ,\ t\ )$ y se aplica LC1 antes del evento *rcv(m)*.

Con este algoritmo se puede demostrar que $a \longrightarrow b \Longrightarrow C(a) < C(b)$. (Lo converso no es necesariamente cierto).



Concurrencia y sincronización distribuida

Algoritmo de servidor central

- Un servidor da acceso a la sección crítica al poseedor de un token

Problemas: Fallo del servidor, fallo del poseedor del token

Algoritmo de Ricart y Agrawala

- Está basado en un consenso distribuido
- Un proceso que requiere entrar a S.C. manda un mensaje multicast con marcas de tiempo de su reloj, y puede entrar si el resto de los procesos se lo permite.
- Si un proceso que quiere entrar recibe un mensaje de petición compara la marca de tiempo y sólo cuando posee la marca de tiempo menor puede entrar

Img

Problemas: El algoritmo requiere muchos más mensajes que el de servidor central. No hay tolerancia a fallas.

Algoritmo de anillo de tokens

- Se basa en la existencia de un anillo físico o lógico entre los procesos.
- Un token es pasado en un solo sentido entre los procesos y el que lo posee puede entrar en sección crítica
- Si un proceso no desea entrar en sección crítica, simplemente pasa el token al siguienre

Img

Problemas: Puede fallar si el token se pierde o un proceso no funciona.
Solución: Enviar varios tokens.

Candados

- Es una variante del uso de tokens.
- Cuando un proceso está en sección crítica genera un archivo (candado o lock).
- Cualquier proceso que desea entrar en S.C. verifica primero la existencia del candado. Si éste existe el proceso espera.
- El control puede ser centralizado o distribuido

Img.

Problemas: Si un proceso falla y no borra el candado, los demás quedan bloqueados

Transacciones y Atomicidad

Son una forma de más alto nivel para manejar la sincronización y depende de la existencia de operaciones primitivas sincronizadas (p. e. BEGIN_TRANSACTION, END_TRANSACTION, ABORT_TRANSACTION, READ, WRITE).

Una transacción (o transacción atómica) es aquella que no puede dividirse y se ejecuta totalmente o se aborta por completo

Requiere de la existencia de almacenamiento estable de información (replicación y/o redundancia)

Propiedades de las transacciones: (operaciones ACIDas)

1. (Atomicity) Atómicidad.- Externamente las transacciones son indivisibles
2. (Consistency) Consistencia.- No se violan los invariantes del sistema
3. (Isolation) Aislamiento.- Las transacciones concurrentes no interfieren entre sí
4. (Durability) Durabilidad.- Una vez realizada una transacción, los cambios son permanentes