

RESUMEN

Técnicas de Programación Concurrente I

[75.59]

Exámenes **PARCIAL** e **INTEGRADOR**



[Francisco O. Lorda](#)

105554

[github/franorquera](#)

[Carolina Di Matteo](#)

103963

[github/gcc-cdimatteo](#)

Definitions.....	4
Modelos de Concurrencia.....	4
Fork-Join.....	4
Work Stealing.....	5
Rayon.....	5
Crossbeam.....	5
Async.....	6
Future.....	6
Pin.....	6
async fn.....	6
block_on.....	6
async tasks.....	7
Diagrama de Tiempo.....	7
Vectorization.....	7
Operaciones Verticales.....	8
Operaciones Horizontales.....	8
Lock.....	9
Condiciones.....	9
Locks envenenados.....	9
Corrección.....	9
Sección Crítica.....	10
Especificaciones.....	10
Actors.....	10
Messages.....	10
Context.....	11
Arbiter.....	11
Sincronización.....	11
Semaphores.....	11
Operaciones.....	11
Binary Semaphore (Mutex).....	12
! Problema: Productor - Consumidor.....	12
! Problema: Barbero.....	13
! Problema: Fumadores.....	13
! Problema: Filósofos.....	14
Barriers.....	14
fn wait(&self).....	14
is_leader().....	14
! Problema: Banquero.....	15
Condvar.....	15
! Problema: Fumadores.....	15
! Problema: Lector/Escritor.....	15
Normal.....	15

Prioridad.....	16
Monitors.....	17
Semaphore vs Monitor.....	18
Redes de Petri.....	18
Red Ordinaria.....	18
Red General.....	18
Función de Marca.....	19
Funciones de Entrada y Salida.....	19
Interpretaciones.....	19
Ejemplos.....	20
! Mutex.....	20
! Banquero.....	20
! Productor Consumidor con Buffer Infinito.....	21
! Productor Consumidor con Buffer Finito.....	21
! Cliente - Servidor.....	22
! Barbero.....	22
! Fumadores.....	23
! Filósofos.....	23
! Lector/Escritor (arco inhibidor).....	24
! Lector/Escritor (prioridad).....	24
Mensajes.....	24
Modelos de Comunicación.....	24
Canales.....	25
Unix.....	25
Rust (share memory by communicating).....	25
Sincronización en Sistemas Distribuidos.....	25
Sockets.....	25
Tipos.....	26
Algoritmos de Exclusión Mutua.....	26
Centralizado.....	26
Ventajas.....	26
Desventajas.....	27
Distribuido (Ricart and Agrawala).....	27
Ventajas.....	28
Desventajas.....	28
Token Ring.....	29
Centralizado vs. Distribuido vs. Token Ring.....	30
Algoritmos de Elección.....	30
Bully.....	30
Ring.....	30
Transacciones.....	31
Primitivas.....	31
Propiedades ACID.....	31

Implementaciones.....	32
Private Workspace.....	32
Write Ahead Log.....	32
Two-Phase Commit.....	32
Two-Phase Locking.....	32
Concurrencia Optimista (GitHub).....	32
Timestamps.....	33
Detección de Deadlocks.....	33
Algoritmo Centralizado.....	33
Algoritmo distribuido.....	33
Algoritmo Wait-Die.....	33
Algoritmo Wound-Wait.....	34
Ambientes Distribuidos.....	34
Entidad.....	34
Capacidades.....	34
Eventos Externos.....	34
Acciones.....	34
Reglas.....	34
Comportamiento.....	34
Comunicación.....	35
Restricciones de Confiabilidad.....	35
Restricciones Temporales.....	35
Costo y Complejidad.....	35
Tiempo y Eventos.....	35
Tipos de Eventos.....	35
Estados y Configuraciones.....	36
Conocimiento.....	36
Tipos.....	36
Referencias.....	37

Definitions

- ★ **Program**: data declarations & assignment and control-flow statements in a programming language which results in machine instructions executed sequentially from compilation
- ★ **Process**: sequential program that comprise a *concurrent program*; written using a finite set of atomic statements
- ★ **Concurrent Program**: set of sequential processes that can be executed in parallel; executes a sequence of the atomic statements obtained by arbitrarily interleaving the atomic statements from the processes
- ★ **Parallel**: systems in which the executions of several programs overlap in time by running them on *separate processors*
- ★ **Concurrent**: *potential parallelism*, in which the executions may overlap
- ★ **Multitasking**: concurrent execution of *more than one process* scheduled in the processor by the OS in a certain period of time
- ★ **Multithreading**: feature *provided by a programming language* which allows concurrent execution of more than one thread within a single program
- ★ **Computation/Scenario**: execution sequence that can occur as a result of interleaving atomic statements
- ★ **Control Pointer**: indicates the next statement that can be executed by that process

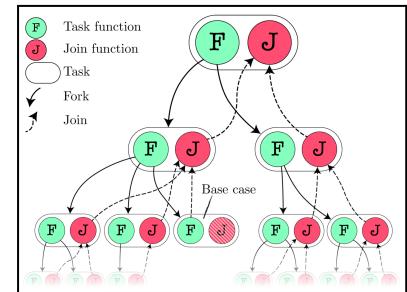
Synchronize the execution of different processes and to enable them to communicate. If the processes were totally independent, the implementation of concurrency would only require a simple scheduler to allocate resources among them.

Modelos de Concurrencia

Especifica cómo los threads colaboran entre sí para completar tareas que les son asignadas. Según el modelo, se comunican y trabajan de formas distintas.

Fork-Join

Modelo de parallelización. El cómputo (**task**) es partido en sub-cómputos menores (**subtasks**). Los resultados se unen (**join**) para construir la solución al cómputo inicial. Se utiliza para un conjunto de problemas donde las tareas son independientes por lo cual se pueden paralelizar. Las tareas sólo se bloquean para esperar el final de las sub-tareas.



Asegura concurrencia sin condiciones de carrera. El resultado del cómputo es *determinístico* (siempre es el mismo) pues los threads son aislados y no dependen entre sí.

Performance (caso ideal): $t_{secuencial} / N_{threads}$.

Work Stealing

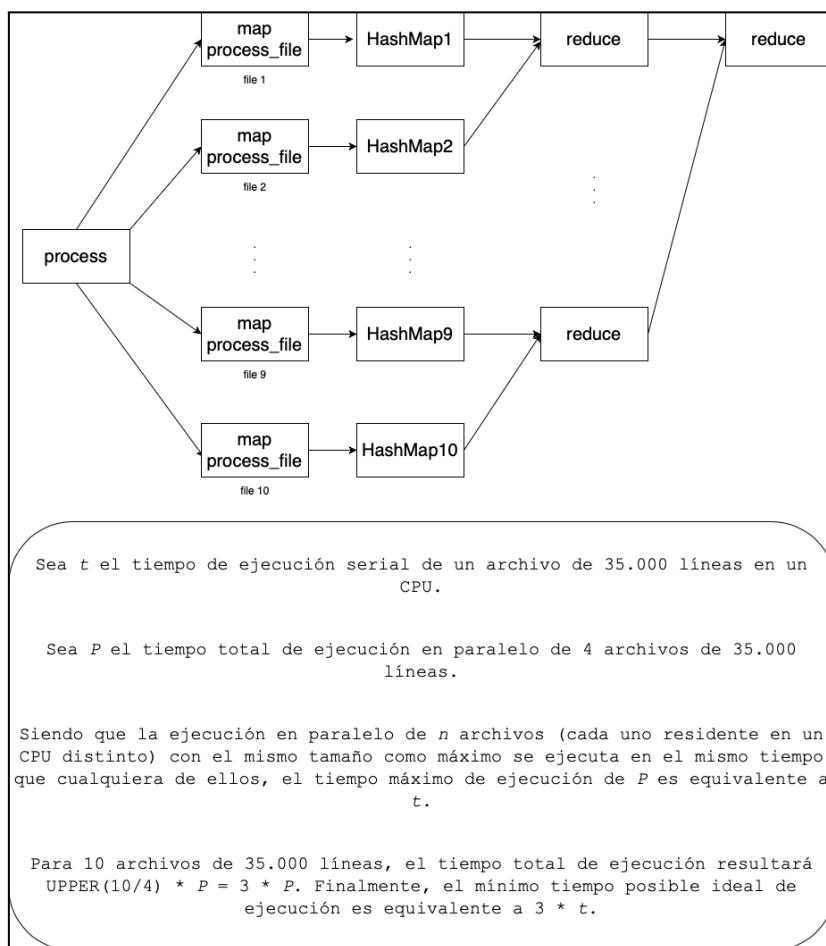
Algoritmo para realizar scheduling de tareas entre threads. Tiene como objetivo mejorar el balanceo de carga de tareas entre threads. Los worker threads inactivos roban trabajo a threads ocupados, para realizar balanceo de carga. Se elige de manera random para que sea equitativo. Minimiza la sincronización entre threads (los workers sólo se comunican cuando lo necesitan).

Rayon

Implementa el modelo fork join de dos formas. Maneja los threads y distribuye el trabajo. Implementa work stealing internamente.

- Realizar dos tareas en paralelo (para funciones)
- Realiza N tareas en paralelo

```
pub fn process(paths: [&str; 10]) -> HashMap<String, i64> {
    paths [&str; 10]
        .par_iter()
        .map(map_op: |p: &str| process_file(filename: format!("./archive/{}.csv", p)))
        .reduce(
            identity: || HashMap::new(),
            op: |h1: HashMap<String, i64>, h2: HashMap<String, i64>| hash_merging(&h1, &h2),
        )
}
```



Crossbeam

Crea un nuevo entorno de thread que garantiza que los threads terminan antes de retornar el closure que se le pasa como argumento a esta función. Todos los

threads que no fueron manualmente esperados (`join`), son esperados antes de que finalice la invocación de la función.

Async

Se usan **tareas asincrónicas** -de Rust- para intercalar tareas en un único thread o en un pool de threads. Las tareas son más livianas que los threads (rápidas de crear, eficientes para pasar el control).

Future

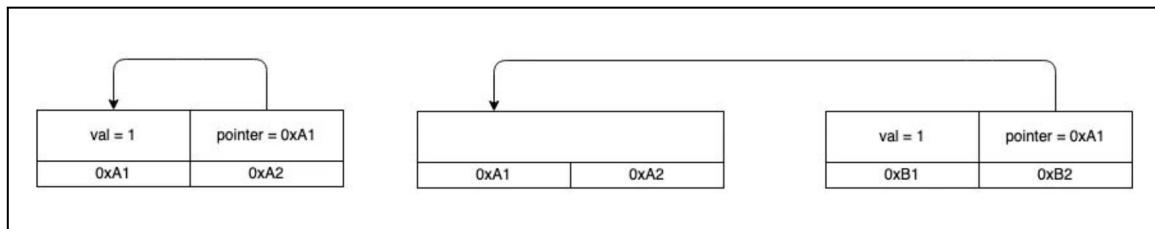
- + estructura sobre la que se puede testear si se completó
- + la operación `poll` (sobre el `future`) nunca bloquea
- + si la operación
 - + se completó → `Poll::Ready(output)`
 - + no se completó → `Pending`
- + modelo piñata → lo único que se puede hacer con un `future` es golpearlo con `poll` hasta que caiga el valor
- + almacena lo necesario para realizar el pedido hecho por la invocación

```
trait Future {
    type Output;
    // por ahora, interpretar 'Pin<&mut Self>' como '&mut Self'.
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>)
        -> Poll<Self::Output>;
}

enum Poll<T> {
    Ready(T),
    Pending,
}
```

Pin

Trait implementado por los `Future` para evitar que su referencia a sí mismos no sea movida en memoria y una vez llamado a `poll()` devuelva el resultado esperado.



async fn

- + retorna inmediatamente antes de que comience a ejecutarse el cuerpo
- + devuelve un `Future` que contiene todo lo necesario para que la función pueda ejecutarse
- + al ejecutar `poll` por primera vez sobre el retorno, se ejecuta el cuerpo de la función hasta el primer `await`
- + `await` toma ownership del `Future` y hace `poll`

block_on

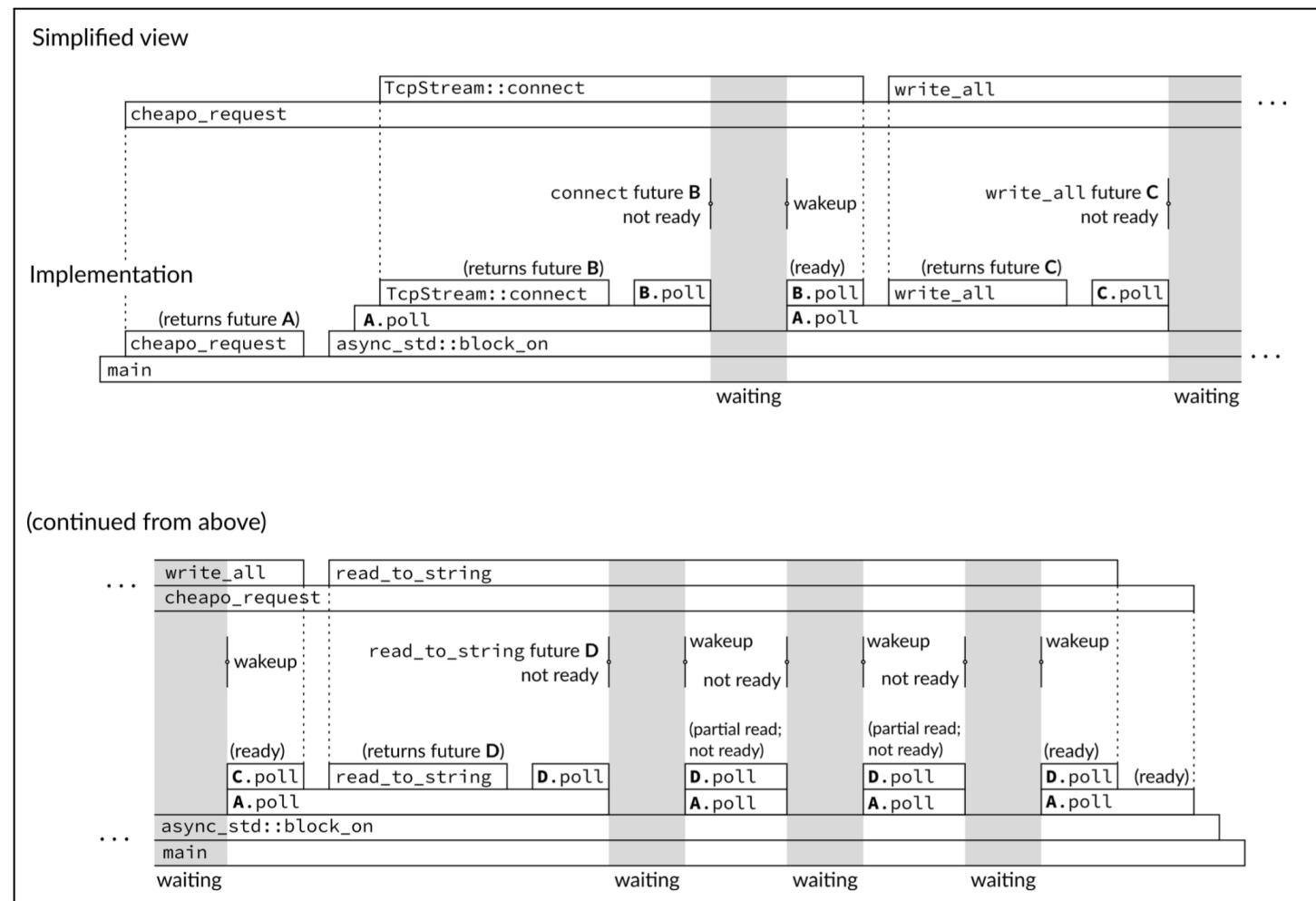
Es una función sincrónica que produce el valor final de la función asincrónica. Espera por el resultado de un valor de una función `async` dentro de un entorno que no es asincrónico. Conoce cuándo hacer `sleep` hasta hacer `poll` de nuevo.

Concurrencia Colaborativa → duerme el thread hasta que se tenga que hacer el *poll* nuevamente.

async tasks

- + recibe un Future y lo agrega a un pool que realizará el polling en el *block_on*
- + lifetimes de variables son static (debe poder ejecutarse hasta el final)
- + todas las ejecuciones pueden realizarse en un único thread
- + el cambio de una tarea a otra ocurre en las expresiones *await*
- + *task::spawn* crea la tarea y la coloca en el pool de threads dedicado a hacer *poll* de los Futures
- + *task::yield_now* favorece el paralelismo, retorna un Future para pasar el control a otra tarea
- + *task::spawn_blocking* coloca la tarea en otro thread del S0 (para realizar un cálculo pesado), dedicado a tareas bloqueantes

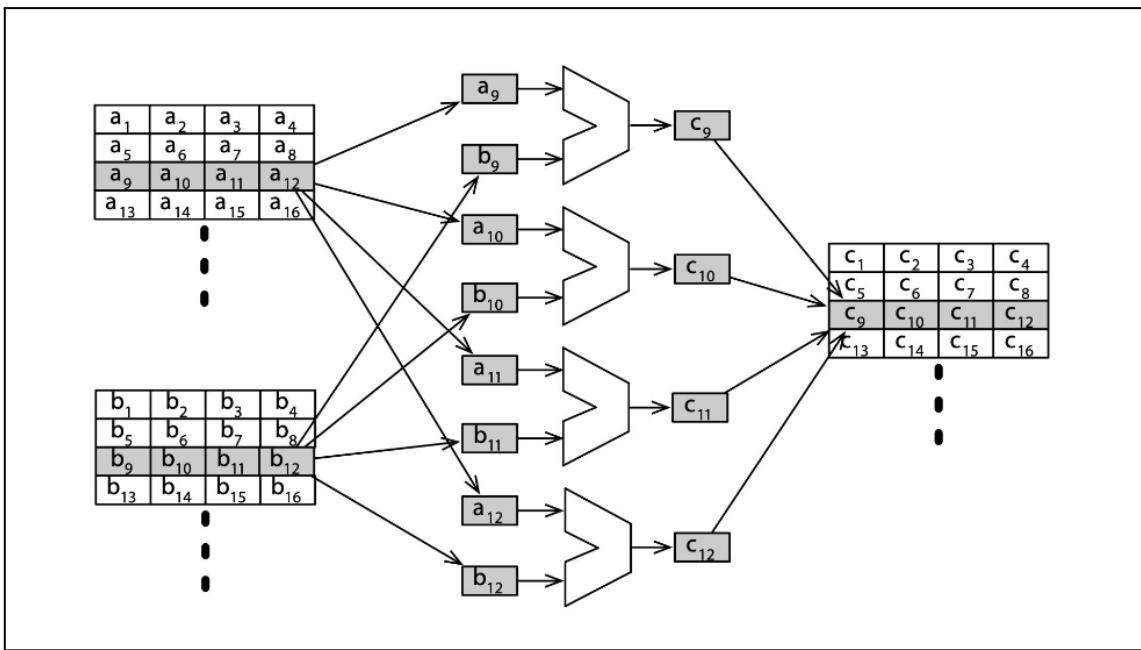
Diagrama de Tiempo



Vectorization

Caso especial de paralelización automática. Procesa una operación en múltiples pares de operandos a la vez. El objetivo es realizar un cálculo simple sobre un

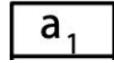
conjunto grande de datos. Por ejemplo, procesar sonido, imágenes, video, redes neuronales. Cada dato es independiente o un pequeño subconjunto lo es.



Cada variable (registro) es un vector de tamaño fijo:

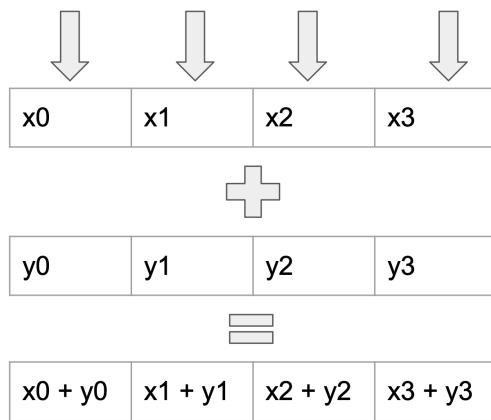


Se divide en N "lanes" (carriles) según el tamaño de los datos (un registro de 512 bits puede alojar 16 enteros de 32 bits, o bien 8 flotantes de 64):



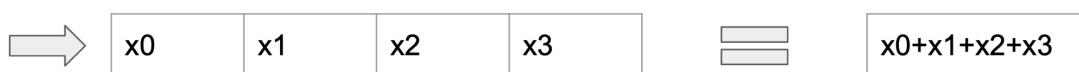
Operaciones Verticales

Entre distintos registros en el mismo lane:



Operaciones Horizontales

En el mismo registro entre distintos lanes. Reducen un vector (registro) a un escalar. Son más lentas porque son secuenciales.



```

rayon::scope(|s| {
    for y in 0..height {
        s.spawn(move |t| {
            // creo una task por cada columna
            for x in 0..width {
                let pixel = input_image.get_pixel(x, y);
                // t.spawn(move |_| {
                // creo una task por cada fila --> como ya estoy parada en una columna, a esta altura es un pixel
                let grayscale_value = (pixel[0] as f32 * 0.299
                    + pixel[1] as f32 * 0.587
                    + pixel[2] as f32 * 0.114) as u8;
                let coord = ((y * width + x) * 3) as isize;
                unsafe {
                    ptr::write_bytes(nasty_output.0.offset(coord), grayscale_value, 3);
                }
            }
        })
    }
});

```

Lock

Sirve para realizar exclusión mutua entre procesos.

- `lock()`: el proceso se bloquea hasta poder obtener el lock.
- `unlock()`: el proceso libera el lock que tomó previamente con `lock`.

Condiciones

Existen dos tipos de locks, de lectura (shared) y de escritura (exclusive).

- Para poder tomar un shared lock el proceso debe esperar hasta que sean liberados todos los exclusive locks.
- Para poder tomar un exclusive lock, el proceso debe esperar hasta que sean liberados todos los locks.

Locks envenenados

Un lock queda en estado envenenado cuando un thread lo toma de forma exclusiva (write lock) y mientras tiene tomado el lock, ejecuta `panic!`.

Las llamadas posteriores a `read()` y `write()` sobre el mismo lock, devolverán error.

Corrección

- Safety: debe ser verdadera siempre
 - ◆ Exclusión Mutua: dos procesos no deben intercalar ciertas (sub)secuencias de instrucciones.
 - ◆ Ausencia de deadlock: un sistema que aún no finalizó debe poder continuar realizando su tarea (avance productivo).
- Liveness: debe volverse verdadera eventualmente
 - ◆ Ausencia de starvation: todo proceso que esté listo para utilizar un recurso debe recibir dicho recurso eventualmente.

- ◆ Fairness: un escenario es débilmente fair (poco justo), si en algún estado en el escenario, una instrucción que está continuamente habilitada, eventualmente aparece en el escenario. (Caso lector/escritor con prioridad de escritura)

Sección Crítica

Cada proceso se ejecuta en un loop infinito cuyo código puede dividirse en parte crítica y parte no crítica. La sección crítica debe progresar (finalizar eventualmente) y la no crítica no requiere progreso (puede terminar o entrar en loop)

Especificaciones

- Exclusión Mutua
- Ausencia de deadlock
- Ausencia de starvation

```

{
  let mut gold_amount: RwLockWriteGuard<'_, i64> = gold.amount.write().unwrap();
  let mut resource_amount: RwLockWriteGuard<'_, i64> = resource.amount.write().unwrap();

  if *gold_amount >= 4 {
    *gold_amount -= 4;
    *resource_amount += 8;
  }
}

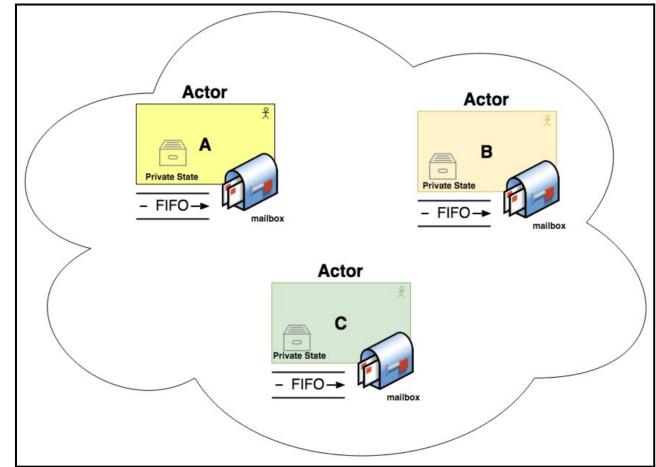
{
  println!("PRINT RESOURCES: {}", *resource.amount.read().unwrap());
}

*gold.amount.lock().unwrap() += 2;
}

```

Actors

- livianos, se pueden crear miles (en lugar de threads)
- encapsulan comportamiento y estado
- compuestos por:
 - dirección: dónde enviar mensajes
 - casilla de correo (mailbox): FIFO de los últimos mensajes recibidos
- actor supervisor puede crear otros actores hijos
- aislados de otros actores (no comparten memoria)
- el estado privado solo puede cambiarse a partir de procesar mensajes
- pueden manejar un mensaje por vez
- en un sistema distribuido, la dirección del actor puede ser una dirección remota



Messages

- medio de comunicación entre actores
- procesados de forma asincrónica
- estructuras simples inmutables
- `do_send()` (== encola)
 - ◆ ignora errores en el envío del mensaje
 - ◆ si la casilla de mensajes está cerrada, se descarta

- ◆ no retorna el resultado
- `try_send()`: envía el mensaje inmediatamente; si la casilla de mensajes está llena o cerrada retorna `SendError` (== encola y retorna)
- `send()`: retorna un objeto `Future` que resulta del proceso de manejo del mensaje (== manda y retorna `Future`)

Context

- exclusivo de cada actor
- permite al actor
 - determinar su dirección
 - cambiar los límites de la casilla de mensajes
 - detenerse
- los mensajes llegan a la casilla primero, luego el contexto de ejecución llama al `handler` específico

Arbiter

- provee el contexto de ejecución asincrónica para los actores, funciones y `Futures`
- aloja el entorno donde se ejecuta el actor
- realiza varias tareas
 - crear un nuevo Thread del SO
 - ejecutar un `event loop`
 - crear tareas asincrónicas en ese `event loop`

Sincronización

Semaphores

Mecanismos de sincronismo de acceso a un recurso, implementado como una construcción de programación concurrente de más alto nivel. Representa la cantidad de recursos disponibles mediante un contador:

- Si contador $> 0 \rightarrow$ recurso disponible
- Si contador $\leq 0 \rightarrow$ recurso no disponible

Operaciones

- `wait()`: resta uno al contador
- `signal()`: suma uno al contador

Operación *wait(S)*

```
if S.V > 0
  S.V := S.V - 1
else
  S.L add p
  p.state := blocked
```

Operación *signal(S)*

```
if S.L is empty
  S.V := S.V + 1
else
  sea q un elemento arbitrario del conjunto S.L
  S.L remove q
  q.state := ready
```

Binary Semaphore (Mutex)

El contador es cero o uno y se comporta como un lock de escritura.

! Problema: Productor - Consumidor

$buffer := emptyQueue$ $sem notEmpty (0, \emptyset)$	
Productor	Consumidor
<pre>dataType d loop forever p1: append(d, buffer) p2: signal(notEmpty)</pre>	<pre>dataType d loop forever q1: wait(notEmpty) q2: d <- take(buffer)</pre>

$buffer := emptyQueue$ $sem notEmpty (0, \emptyset)$ $sem notFull (N, \emptyset)$

Productor	Consumidor
<pre>dataType d loop forever p1: producir p2: wait(notFull) p3: append(d, buffer) p4: signal(notEmpty)</pre>	<pre>dataType d loop forever q1: wait(notEmpty) q2: d <- take(buffer) q3: signal(notFull) q4: consume(d)</pre>

! Problema: Barbero

```
let barber = thread::spawn(move || loop {
    println!("[Barbero] Esperando cliente");
    customer_waiting_barber.acquire();

    barber_ready_barber.release();
    println!("[Barbero] Cortando pelo");

    thread::sleep(Duration::from_secs(2));

    haircut_done_barber.release();
    println!("[Barbero] Terminé");
});
```

```
thread::spawn(move || loop {
    thread::sleep(Duration::from_secs(thread_rng().gen_range(2, 10)));

    let me = customer_id_customer.fetch_add(1, Ordering::Relaxed);
    println!("[Cliente {}] Entró a la barbería", me);
    customer_waiting_customer.release();

    println!("[Cliente {}] Esperando barbero", me);
    barber_ready_customer.acquire();

    println!("[Cliente {}] Me siento en la silla del barbero", me);

    println!("[Cliente {}] Esperando a que me termine de cortar", me);
    haircut_done_customer.acquire();

    println!("[Cliente {}] Me terminaron de cortar", me);
})
```

! Problema: Fumadores

```
let agent = thread::spawn(move || loop {
    println!("[Agente] Esperando sem");
    agent_sem_a.acquire();

    let mut ings = ALL_INGREDIENTS.to_vec();
    ings.shuffle(&mut thread_rng());
    let selected_ings = &ings[0..N-1];
    for ing in selected_ings {
        println!("[Agente] Pongo {:?}", ing);
        ingredients_sem_a[*ing as usize].release();
    }
});
```

```
let ing = Ingredients::from_usize(ing_id).unwrap();
println!("[Fumador {:?}] Esperando {:?}", me, ing);
ingredient_sems_smoker[ing_id].acquire();
println!("[Fumador {:?}] Obtuve {:?}", me, ing);
```

! Problema: Filósofos

```

if id == (N - 1) {
    // del primero al N-1 filósofo agarran primero el derecho y después el izquierdo
    first_chopstick = &chopsticks[next];
    second_chopstick = &chopsticks[id];
} else {
    // el N filósofo agarra primero el izquierdo y después el derecho
    first_chopstick = &chopsticks[id];
    second_chopstick = &chopsticks[next];
}

thread::sleep(Duration::from_millis(100 * id as u64));

// tratar de forzar tomar el primer palito en el orden de id
loop {
    println!("filosofo {} pensando", id);
    {
        let first_access = first_chopstick.access();
        {
            let second_access = second_chopstick.access();
            println!("filosofo {} comiendo", id);
            thread::sleep(Duration::from_millis(thread_rng().gen_range(500, 1500)));
        }
    }
}

```

Barriers

Permite sincronizar varios threads en puntos determinados de un cálculo o algoritmo.

fn wait(&self)

Bloquea al thread hasta que todos se encuentren en el punto.

is_leader()

Devuelve true en el thread líder, es decir el último de los threads que llama a wait y desbloquea al resto.

! Problema: Banquero

```
// Espero a que todos inicien la semana
barrier.wait();

let prestamo = *lock.read().unwrap() / FRIENDS as f64;
println!("inversor {} inicio semana {} plata {}", id, semana, prestamo);
// Espero a que todos hayan leido el saldo disponible
barrier2.wait();

// Tomo el dinero
```

Condvar

```
► waitC(cond)
    cond.append (p)
    p.state := blocked
    monitor.releaseLock ()
► signalC(cond)
    if ( cond <> empty )
    begin
        q := cond.remove ()
        q.state := ready
    end
► empty(cond)
    return cond = empty
```

- no guarda ningún valor
- tiene asociado un FIFO
- 3 operaciones atómicas
 - waitC(cond)
 - signalC(cond)
 - empty(cond)

```
let pair = Arc::new((Mutex::new([false, false, false]), Condvar::new()));
let (lock, cvar) = &*pair.agent;
```

! Problema: Fumadores

```
let mut state = cvar.wait_while(lock.lock().unwrap(), |ings| {
    let full_table = (*ings).iter().any(|i| *i);
    println!("[Agente] Esperando a que fumen {:?} - {}", ings, full_table);
    full_table
}).unwrap();

let mut _guard = cvar.wait_while(lock.lock().unwrap(), |ings| {
    let my_turn = (0..N).all(|j| j == fumador_id || ings[j]);
    println!("[Fumador {:?}] Chequeando {:?} - {}", me, ings, my_turn);
    !my_turn
}).unwrap();
```

! Problema: Lector/Escritor

Normal

```
let pair = Arc::new((Mutex::new(ReadWrite { readers: 0, writing: false }), Condvar::new()));
```

```

{

    let mut _guard = cvar.wait_while(lock.lock().unwrap(), |state| {
        println!("[Lector {}] Chequeando {:?}", me, state);
        state.writing
    }).unwrap();
    _guard.readers += 1;
}

unsafe {
    println!("[Lector {:?}] Leyendo {}", me, data_reader.data.get().read());
}
thread::sleep(Duration::from_millis(thread_rng().gen_range(500, 1500)));
println!("[Lector {:?}] Terminé", me);

lock.lock().unwrap().readers -= 1;
cvar.notify_all();
}

{
    let mut _guard = cvar.wait_while(lock.lock().unwrap(), |state| {
        println!("[Escrivtor {}] Chequeando {:?}", me, state);
        state.writing || state.readers > 0
    }).unwrap();
    _guard.writing = true;
}

unsafe {
    println!("[Escrivtor {:?}] Escribiendo", me);
    data_writer.data.get().write(me);
}
thread::sleep(Duration::from_millis(thread_rng().gen_range(500, 1500)));
println!("[Escrivtor {:?}] Terminé", me);

lock.lock().unwrap().writing = false;
cvar.notify_all();
}

```

Prioridad

```
let pair = Arc::new((Mutex::new(ReadWrite { readers: 0, writing: false, writers: 0 }), Condvar::new()));
```

```

{
    let mut _guard = cvar.wait_while(lock.lock().unwrap(), |state| {
        println!("[Lector {}] Chequeando {:?}", me, state);
        state.writing || state.writers > 0
    }).unwrap();
    _guard.readers += 1;
}

unsafe {
    println!("[Lector {:?}] Leyendo {}", me, data_reader.data.get().read());
}
thread::sleep(Duration::from_millis(thread_rng().gen_range(500, 1500)));
println!("[Lector {:?}] Terminé", me);

lock.lock().unwrap().readers -= 1;
cvar.notify_all();

lock.lock().unwrap().writers += 1;
{
    let mut _guard = cvar.wait_while(lock.lock().unwrap(), |state| {
        println!("[Escritor {}] Chequeando {:?}", me, state);
        state.writing || state.readers > 0
    }).unwrap();
    _guard.writing = true;
}

unsafe {
    println!("[Escritor {:?}] Escribiendo", me);
    data_writer.data.get().write(me);
}
thread::sleep(Duration::from_millis(thread_rng().gen_range(500, 1500)));
println!("[Escritor {:?}] Terminé", me);

let mut state = lock.lock().unwrap();
state.writing = false;
state.writers -= 1;
cvar.notify_all();

```

Monitors

Permite a los hilos tener exclusión mutua y la posibilidad de esperar (block) a que una condición se vuelva falsa.

Tienen un mecanismo para señalizar otros hilos cuando su condición se cumple.

Los procesos pueden...

- + esperar para entrar al monitor
- + ejecutar el monitor (sólo un proceso a la vez - exclusión mutua)
- + estar bloqueado en FIFO de variable de condición
- + recibir la liberación de la wait condition
- + completar una operación signalC

Semaphore vs Monitor

Semáforo	Monitor
wait puede o no bloquear	waitC siempre bloquea
signal siempre tiene efecto	signalC no tiene efecto si la cola está vacía
signal desbloquea un proceso arbitrario	signalC desbloquea el proceso del tope de la cola
un proceso desbloqueado con signal , puede continuar la ejecución inmediatamente	un proceso desbloqueado con signalC debe esperar que el proceso señalizador deje el monitor

Redes de Petri

Herramienta para modelar concurrencia y sincronización en sistemas distribuidos.
Usadas como una ayuda visual para modelar el comportamiento del sistema.
Construidas sobre una sólida base matemática.

Red Ordinaria

Grafo dirigido bipartito que cumple

$$PN = (T, P, A)$$

- ❖ $T = t_1, t_2, \dots, t_n$: conjunto de transiciones (eventos que ocasionan los cambios de estado)
- ❖ $P = p_1, p_2, \dots, p_n$: conjunto de lugares (estados del sistema)
- ❖ $A \subseteq (T \times P) \cup (P \times T)$: conjunto de arcos

Red General

Grafo dirigido bipartito que cumple

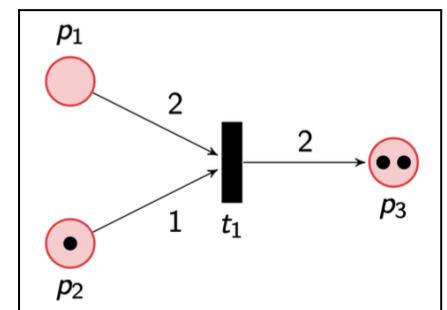
$$PN = (T, P, A, W, M_0)$$

- ❖ $T = t_1, t_2, \dots, t_n$: conjunto de transiciones (eventos que ocasionan los cambios de estado)

- ❖ $P = p_1, p_2, \dots, p_n$: conjunto de lugares (estados del sistema)
- ❖ $A \subseteq (T \times P) \cup (P \times T)$: conjunto de arcos
- ❖ $W : A \rightarrow N \in$ función de peso
- ❖ $M_0 : P \rightarrow N \cup \{0\} \in$ función de marca inicial

La transición t está habilitada si y sólo si $M(p) \geq W(p, t) : \forall p \in I(t)$. Cuando t se dispara:

- + $\forall p \in I(t) : M(p) \leftarrow M(p) - W(p, t)$
- + $\forall p' \in O(t) : M(p') \leftarrow M(p') + W(p', t)$



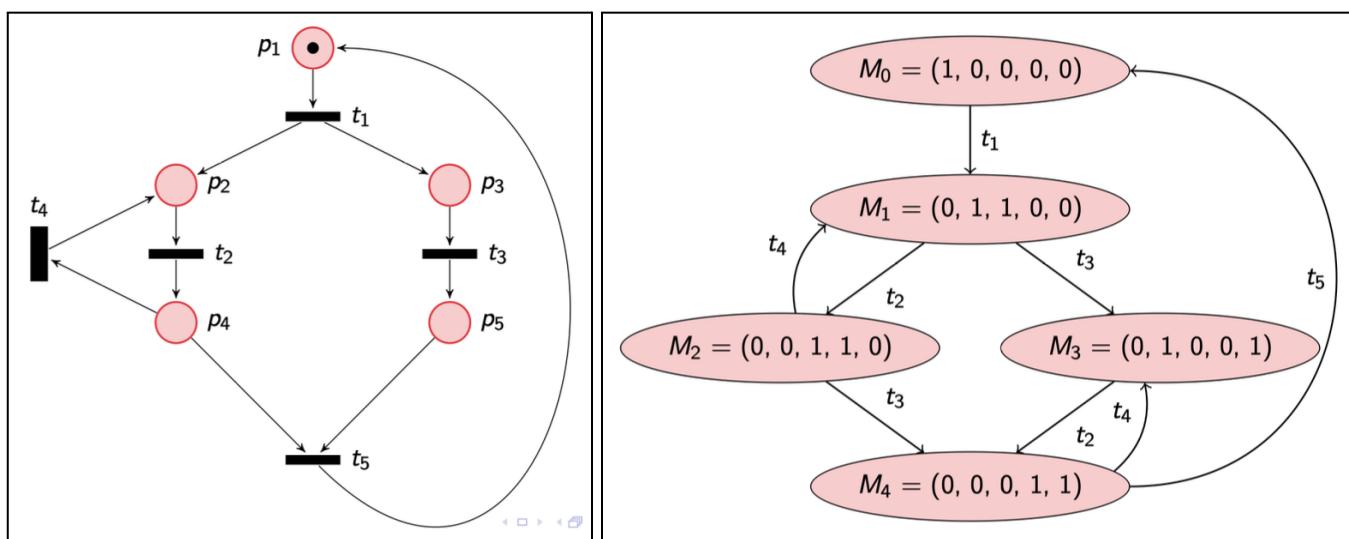
Función de Marca

$$M : P \rightarrow N \cup \{0\}$$

Cuando el token está en el lugar $p_1 \Rightarrow M(p_1) = 1$ y $M(p_2) = 0 \Rightarrow M_0 = (1, 0)$

Funciones de Entrada y Salida

- $I(t) = p / p \in P / (p, t) \in A \subset P$ es la entrada o input de la transición t
- $O(t) = p / p \in P / (t, p) \in A \subset P$ es la salida o output de la transición t

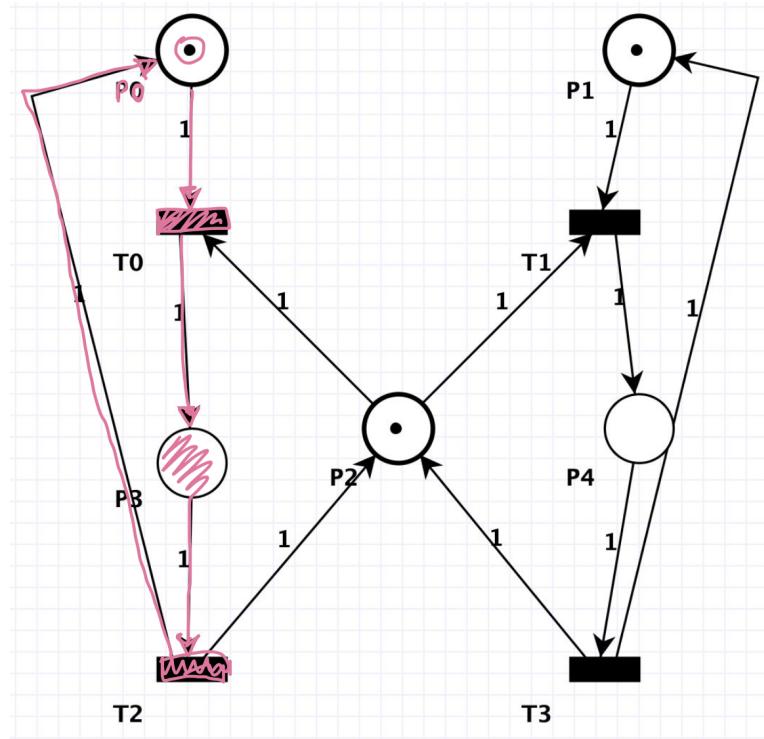


Interpretaciones

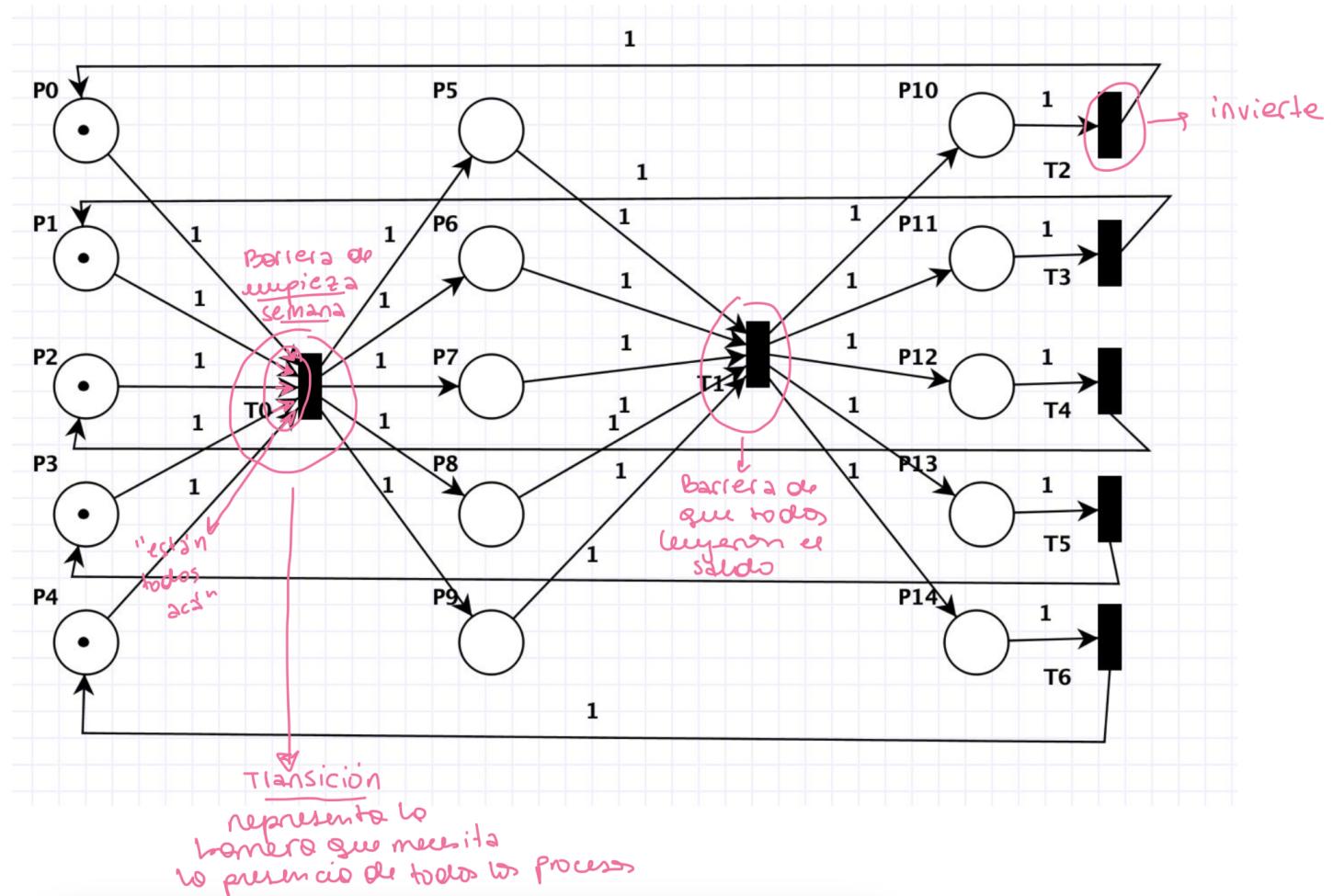
Lugares de entrada	Transiciones	Lugares de salida
Precondiciones	Eventos	Postcondiciones
Datos de entrada	Cómputos	Datos de salida
Señales de entrada	Procesamiento de Señales	Señales de salida
Bufferes de entrada	Procesador	Bufferes de salida

Ejemplos

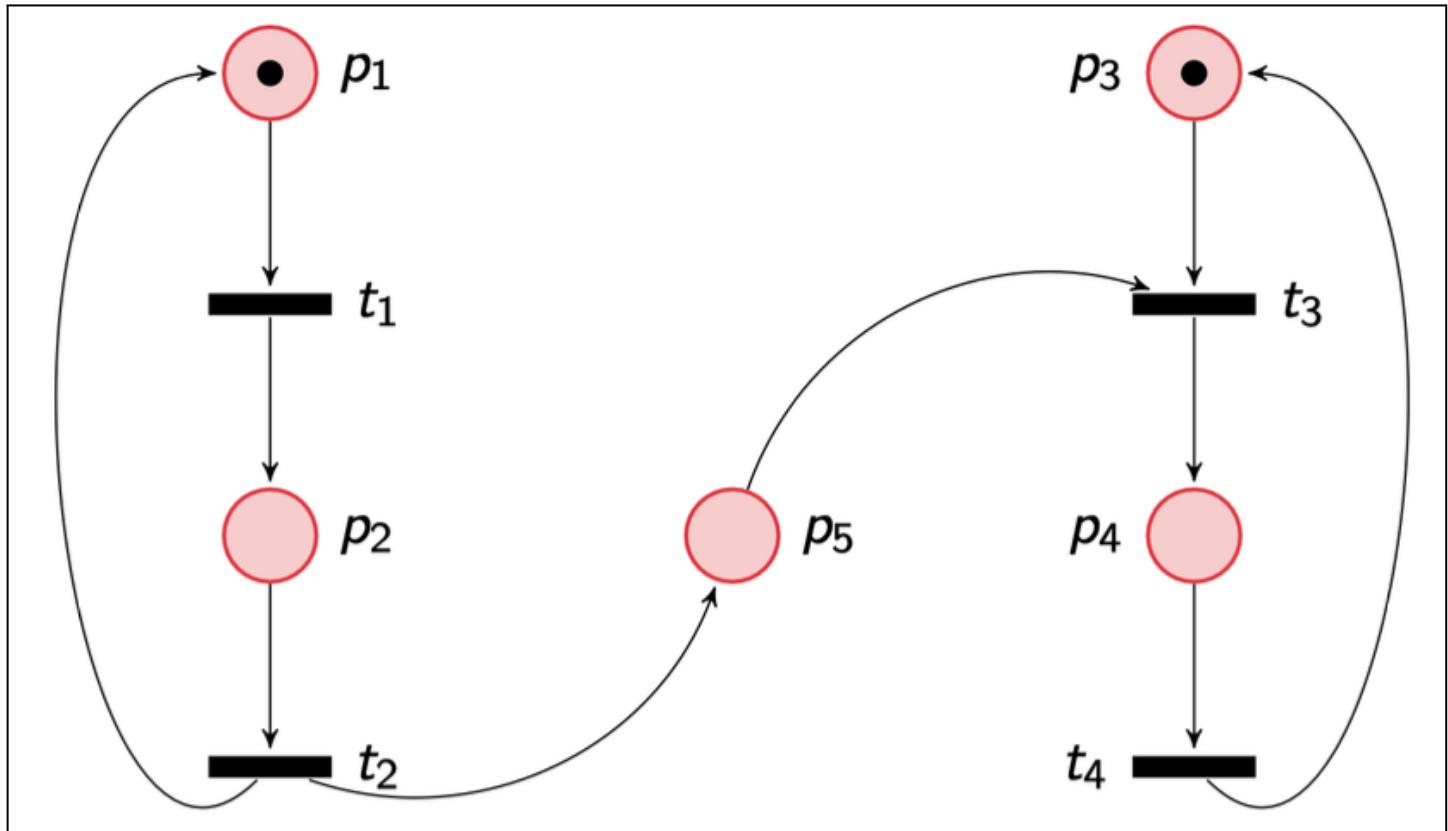
! Mutex



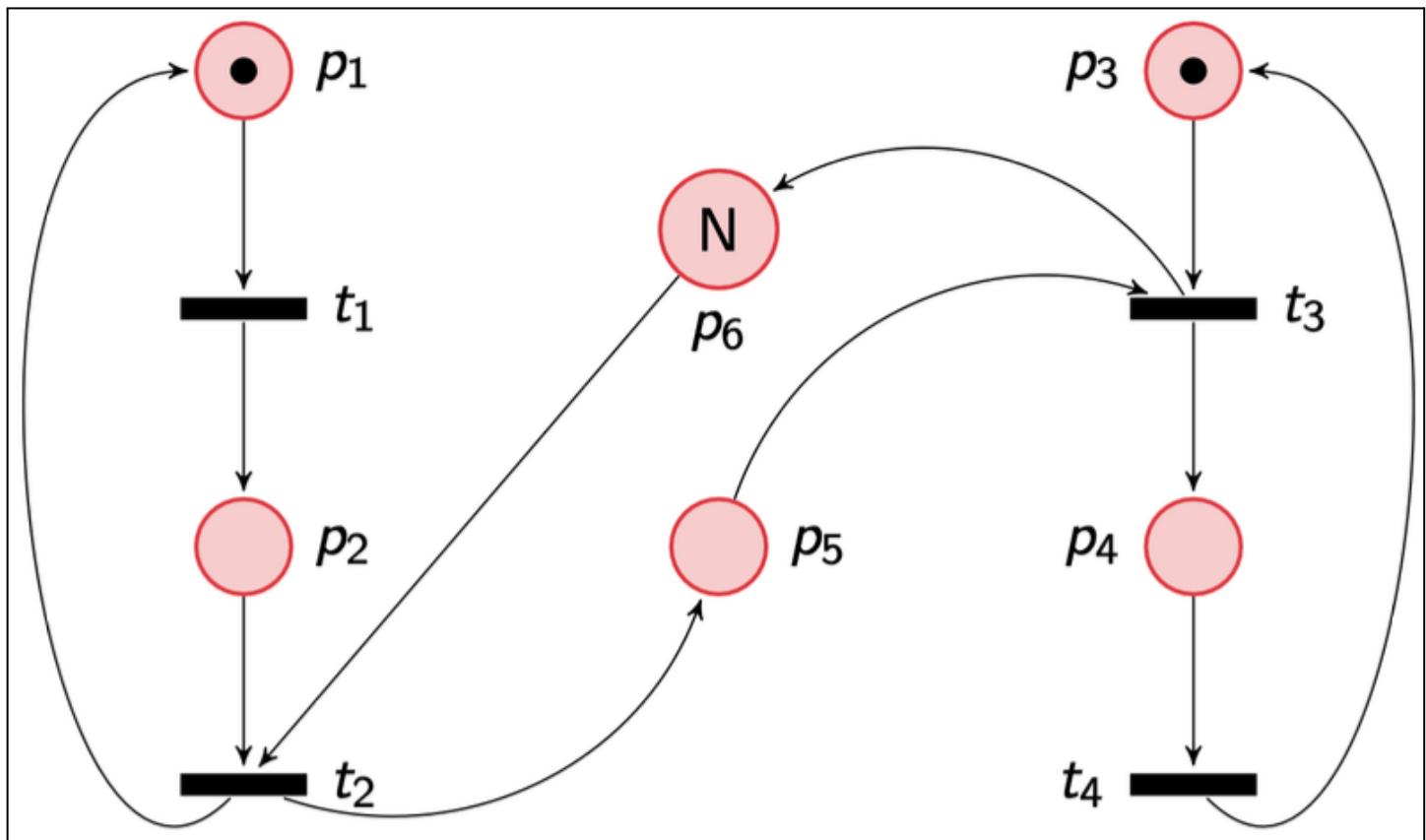
! Banquero



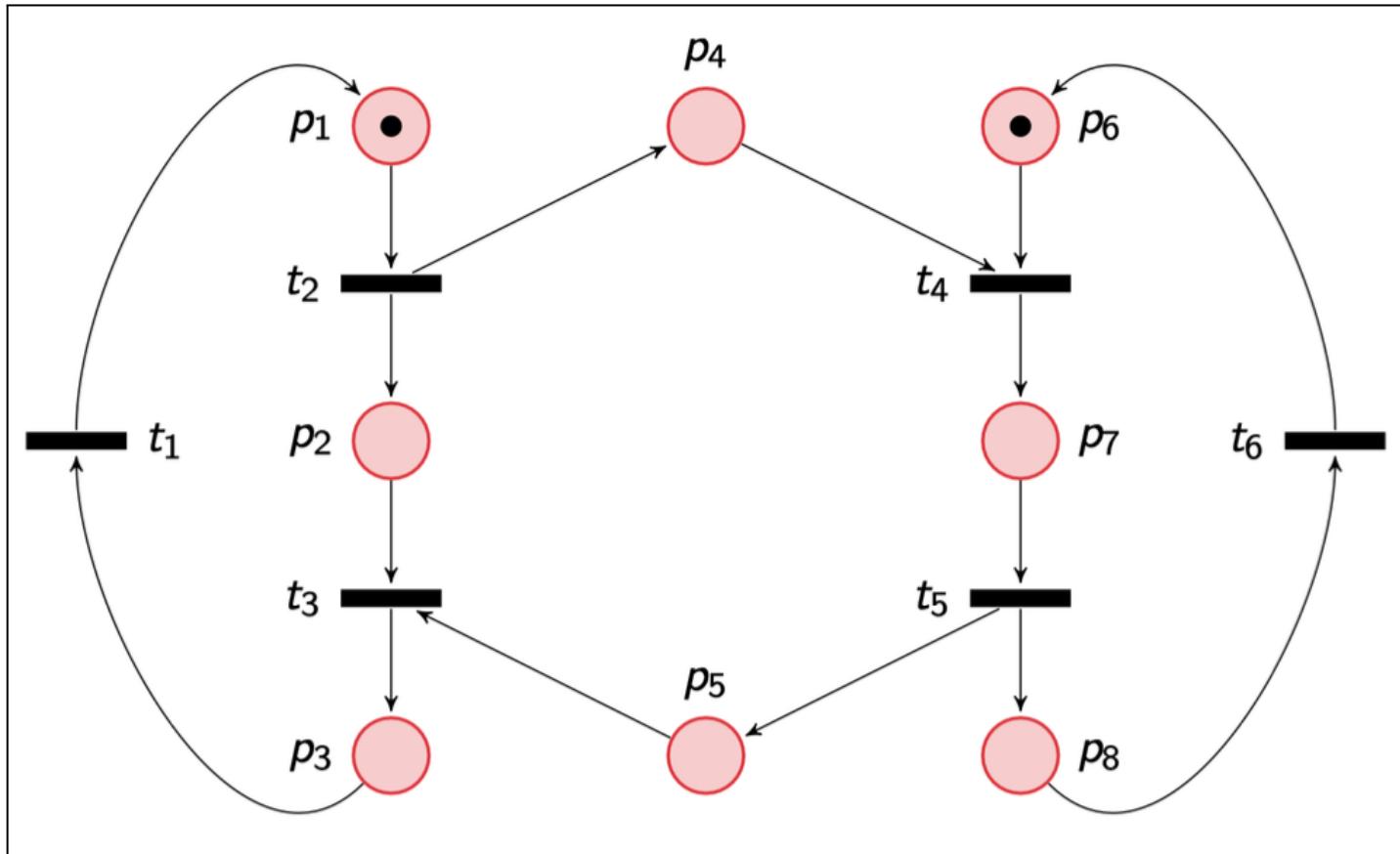
! Productor Consumidor con Buffer Infinito



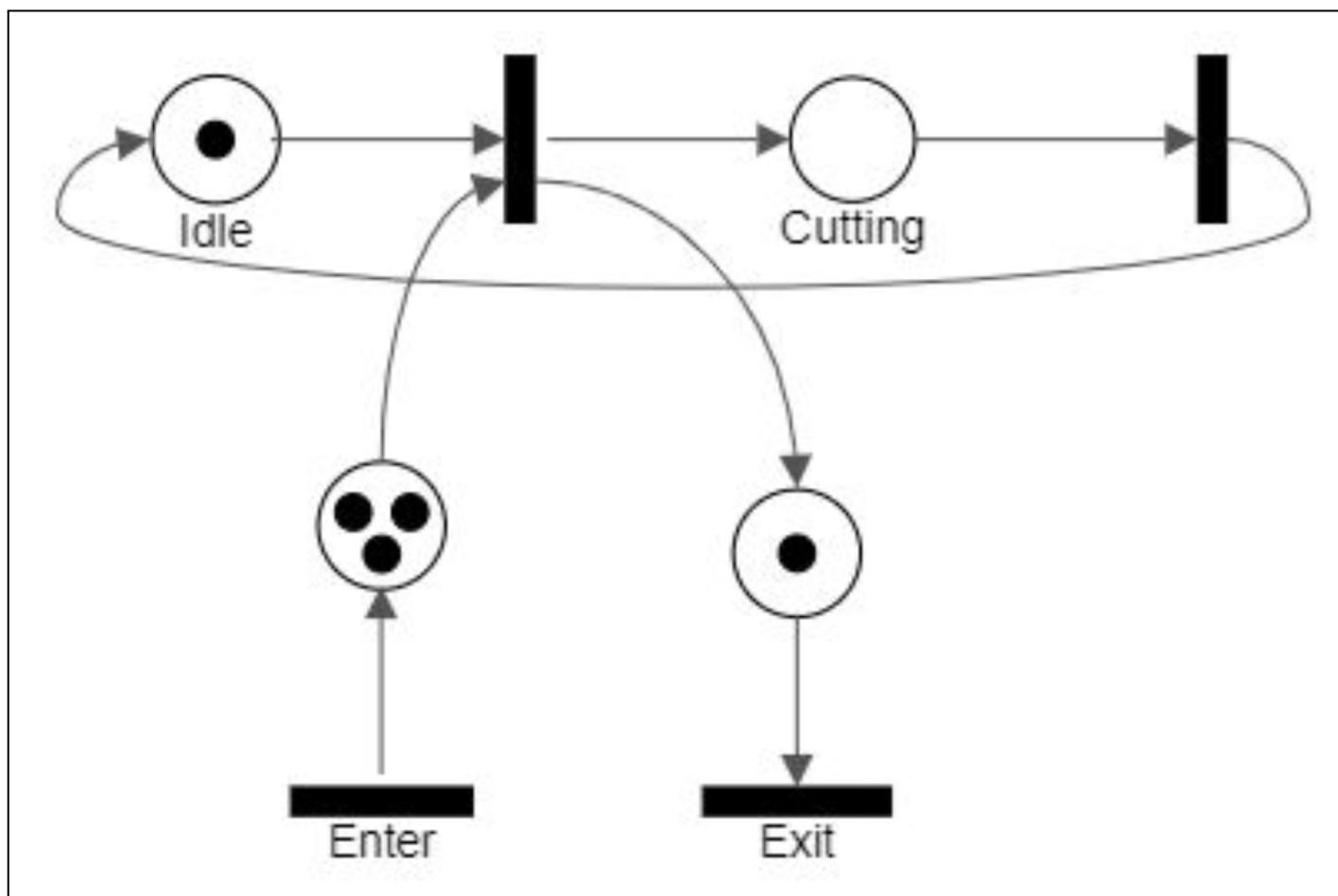
! Productor Consumidor con Buffer Finito



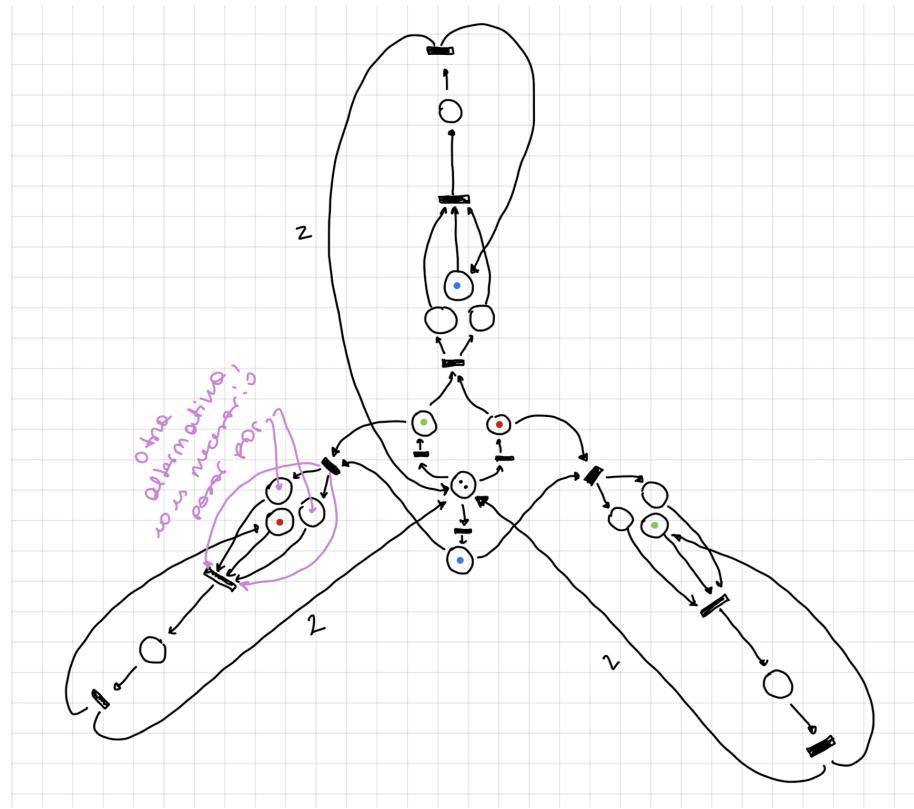
! Cliente - Servidor



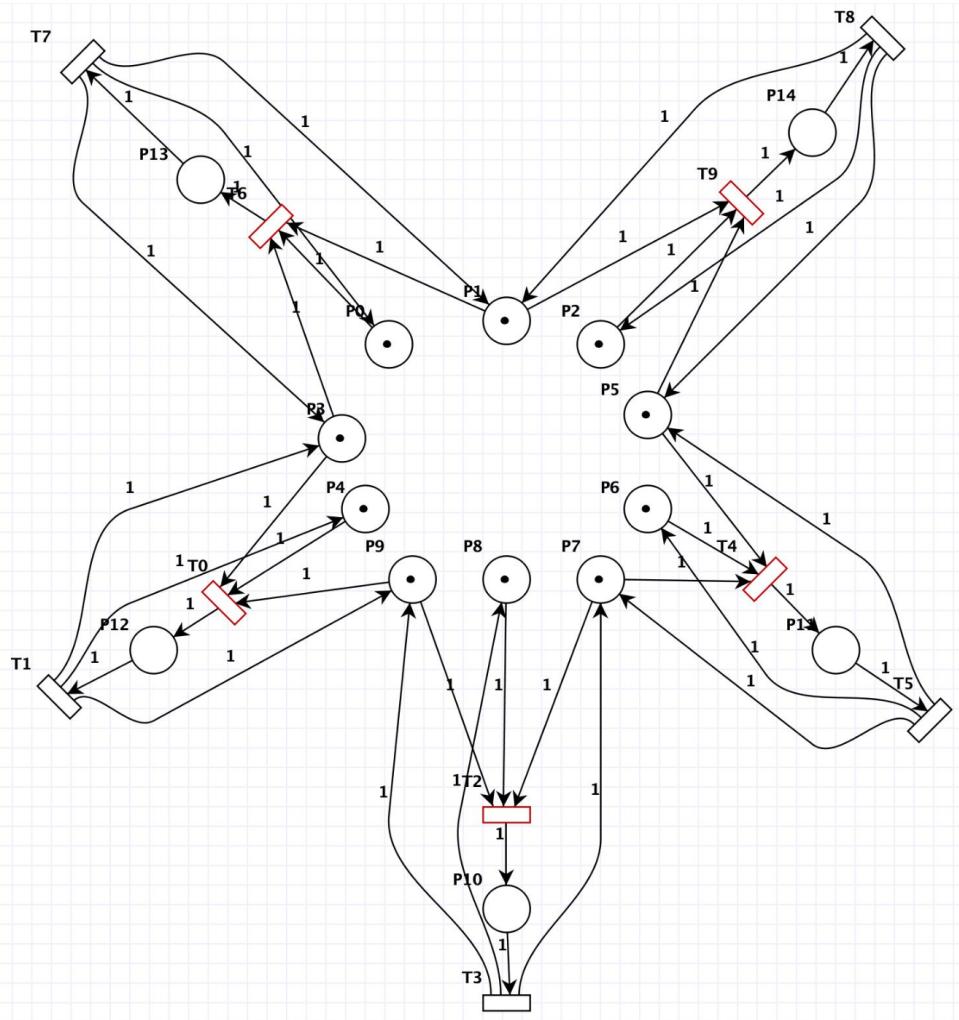
! Barbero



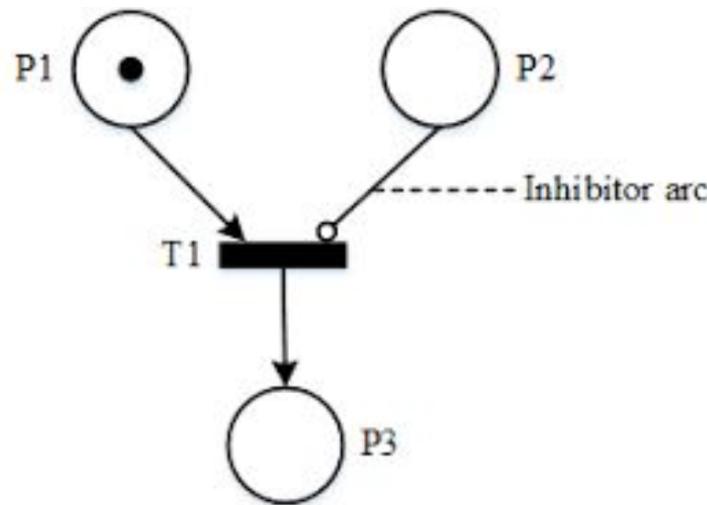
! Fumadores



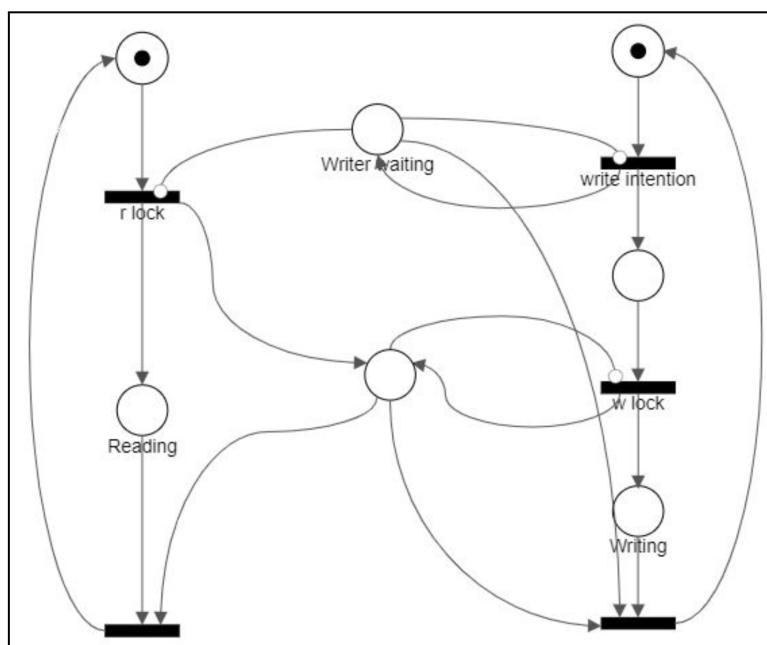
! Filósofos



! Lector/Escritor (arco inhibidor)



! Lector/Escritor (prioridad)



Mensajes

Modelos de Comunicación

Comunicación

- sincrónica
- asincrónica

Direccionamiento

- simétrico
- asimétrico
- sin direccionamiento

Flujo de Datos

- direccional
- bidireccional

Canales

- conectan un proceso emisor con un proceso receptor
- pueden ser **sincrónicos** o **asincrónicos**
- son **unidireccionales**

<i>channel of Integer ch</i>	
Productor	Consumidor
<pre>task body Producer is I : Integer begin loop Produce(I); ch <= I; end loop end Producer</pre>	<pre>task body Producer is I : Integer begin loop ch => I ; Consume(I); end loop end Consumer</pre>

Unix

- *Pipes* y *FIFOs* → conecta dos procesos independientes, orientados a bytes
- *Message Queues* → mensajes como unidades independientes

Rust (*share memory by communicating*)

- 2 extremos
 - emisor → una parte del código invoca métodos sobre el transmisor, con los datos que se quieren enviar
 - receptor → otra parte del código chequea el extremo de recepción por la existencia de mensajes
- múltiples productores, un consumidor
- transfieren el ownership del elemento enviado
- para crear múltiples productores, se clona el extremo de envío

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("Hola");
        tx.send(val).unwrap();
    });
    let received = rx.recv().unwrap();
    println!("Recibido: {}", received);
}
```

Sincronización en Sistemas Distribuidos

Sockets

Comunicación entre dos procesos diferentes

- en la misma máquina

- en dos máquinas diferentes

Se usan en aplicaciones que implementan el modelo *cliente - servidor*:

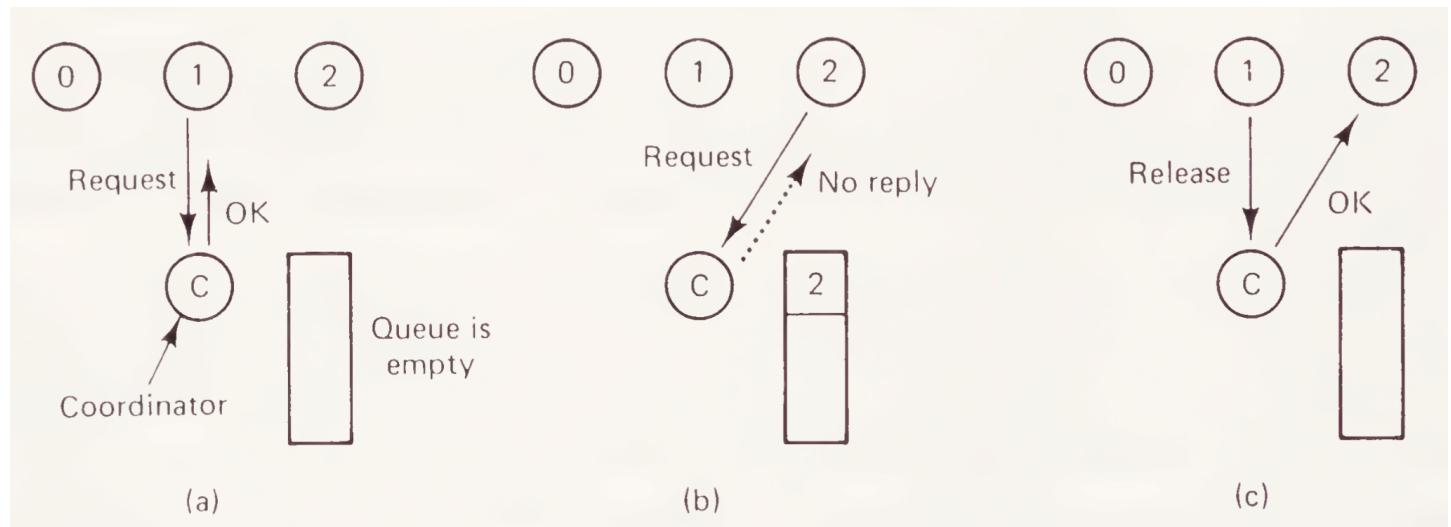
- ❖ **Cliente** → activo porque inicia la interacción con el servidor
- ❖ **Servidor** → pasivo porque espera recibir las peticiones de los clientes

Tipos

- ★ **STREAM** → protocolo TCP (entrega garantizada del flujo de bytes)
- ★ **DATAGRAM** → protocolo UDP (entrega no garantizada; servicio sin conexión)
- ★ **RAW** → permiten a las aplicaciones enviar paquetes IP
- ★ **SEQUENCED PACKET** → similares a stream sockets, pero preservan los delimitadores de registro; utilizan el protocolo SPP (Sequenced Packet Protocol)

Algoritmos de Exclusión Mutua

Centralizado



Un proceso es elegido coordinador. Cuando otro proceso quiere entrar a la sección crítica, envía un mensaje de request al coordinador pidiendo permiso:

- Si ningún otro proceso está en la sección crítica, el coordinador da el permiso. Cuando el proceso inicial recibe la respuesta de acceso, entra a la sección crítica y empieza a trabajar. Cuando termina, envía el último mensaje al coordinador devolviendo el acceso exclusivo.
- Si algún otro proceso está en la sección crítica, el coordinador no responde o rechaza el permiso, y encola el pedido según el orden de llegada. Cuando el proceso que tenía tomado el acceso lo libera, desencola al primer proceso que tenía en la cola y le da permiso a entrar a la sección crítica.

Ventajas

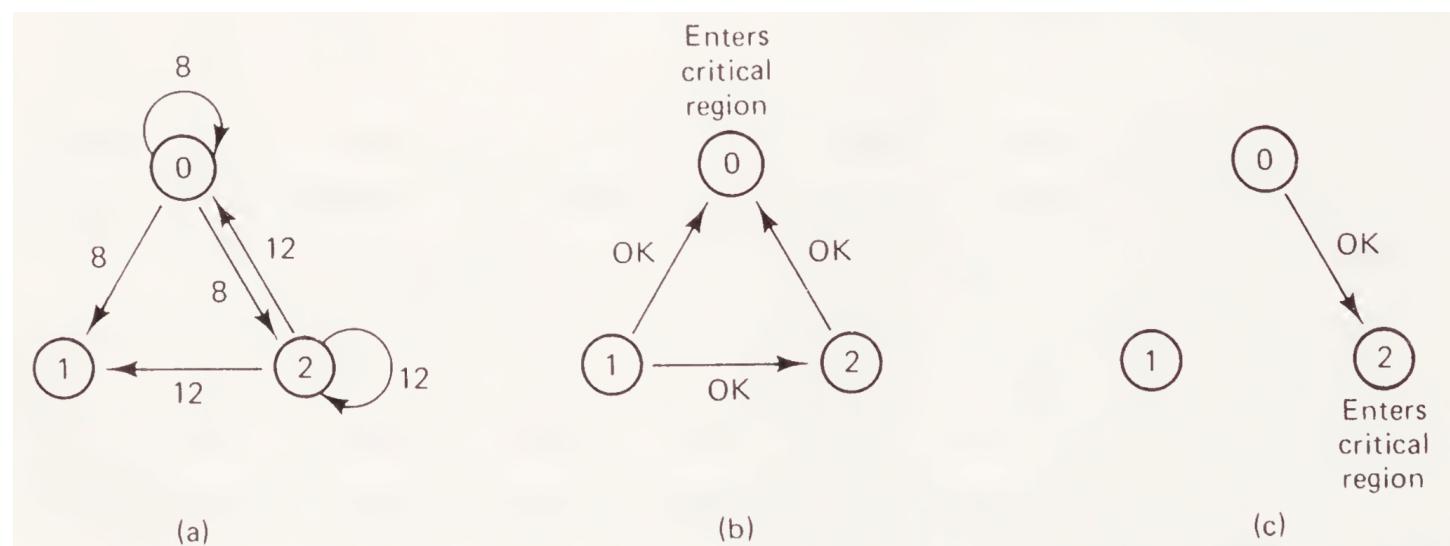
- + **Garantiza la exclusión mutua** → el coordinador sólo deja entrar a la sección crítica a un proceso a la vez

- + **Es justo** → los pedidos son garantizados en el orden en el que llegan al coordinador y ningún proceso muere de starvation
- + **El esquema es simple de implementar**
- + **La conexión sólo requiere de 3 mensajes** → request, grant, release
- + Puede ser utilizado para alojamiento de recursos más generales

Desventajas

- El coordinador es un único punto de falla (**single point of failure**) → si se cae, todo el sistema lo hace
- Si los procesos se bloquean después de hacer un request, **no pueden distinguir entre un coordinador caído de un permiso denegado** ya que no se recibe ningún mensaje
- En un sistema grande, un sólo coordinador puede conllevar a un **cuello de botella**

Distribuido (Ricart and Agrawala)



Cuando un proceso quiere entrar en una sección crítica, construye un mensaje con: el nombre de la sección crítica a la cual quiere acceder, su número de proceso y el tiempo actual (timestamp). Envía dicho mensaje al resto de los procesos, incluido él mismo (asumiendo que todos los mensajes se reciben).

Cuando un proceso recibe un request de otro, la acción que toma depende de su estado con respecto a la sección crítica que se menciona en el mensaje:

1. Si el receptor no está en la sección crítica ni quiere acceder a ella, envía al emisor inicial un mensaje de OK.
2. Si el receptor está en la sección crítica, no responde y encola el pedido.
3. Si el receptor quiere entrar a la sección crítica pero todavía no lo hizo, compara el timestamp del mensaje que llega con el mensaje que envió a todos los demás. El timestamp más chico gana la sección crítica primero.
 - a. Si el proceso emisor inicial tiene el timestamp más pequeño, el receptor envía un mensaje de OK.
 - b. Si el proceso receptor tiene el timestamp más pequeño, encola el pedido y no envía nada al emisor.

Luego de enviar el pedido de permiso para entrar a una sección crítica, el proceso espera a que todos los demás le den permiso. Cuando se reciben todos, entra a la sección crítica. Cuando sale de la sección crítica, envía OK a todos los procesos que encoló y los borra de la cola.

Ventajas

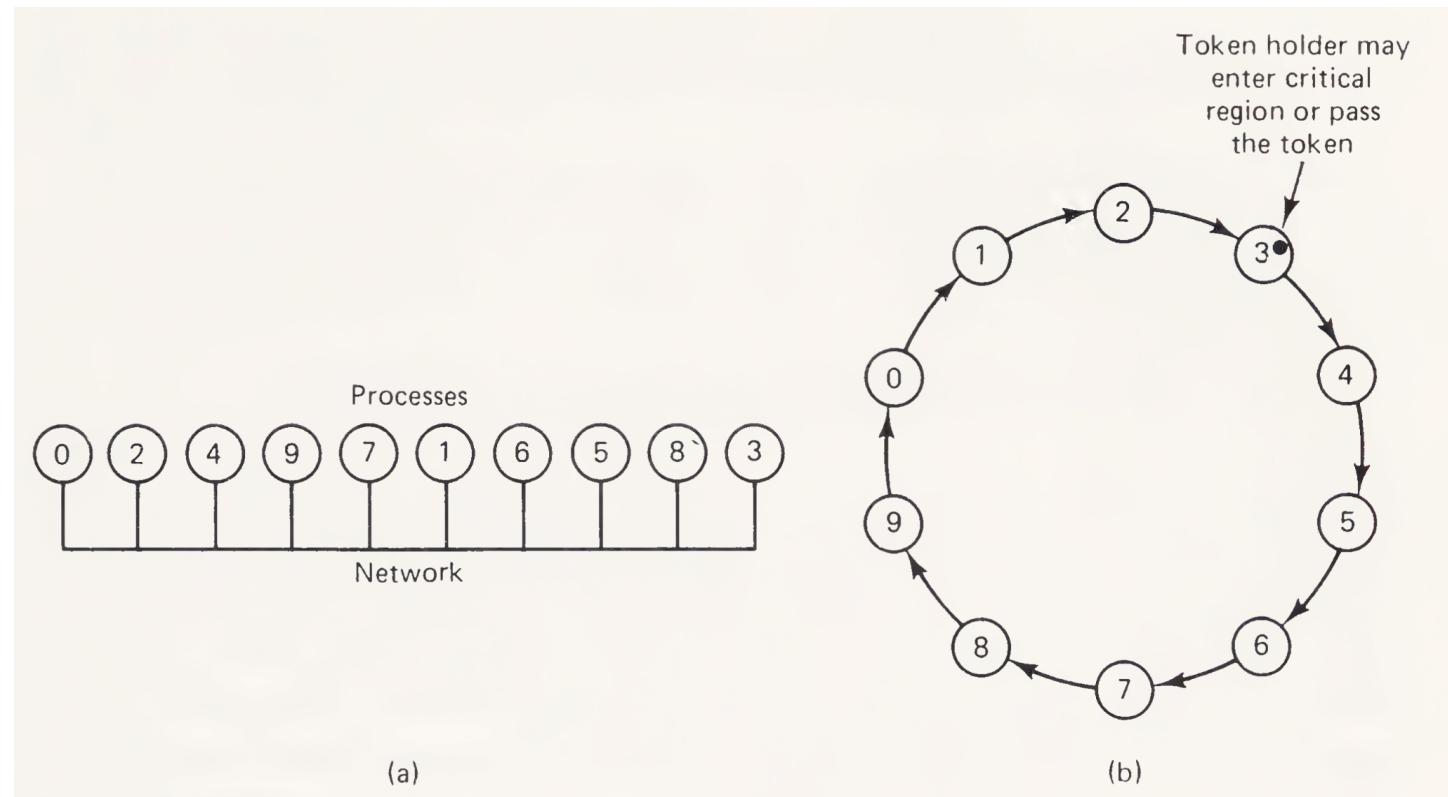
- + **Garantiza la exclusión mutua sin deadlock ni starvation**

Desventajas

- Se reemplaza el **single point of failure** por **n points of failure** → si algún proceso cae, falla en responder los pedidos y el silencio será interpretado incorrectamente como una denegación de permiso bloqueando todos los intentos subsecuentes de otros procesos para entrar a las secciones críticas. Se puede arreglar respondiendo todos los mensajes, tanto de autorización como de denegación de permiso. Siempre que un pedido o una respuesta estén perdidos, el emisor sigue intentando establecer una conexión hasta que se lance un timeout y el emisor concluya que el destinatario está caído. Luego de que un pedido haya sido denegado, el emisor debe bloquearse esperando por un mensaje OK.
- Tiene que existir un **grupo de comunicación primitivo**, o cada proceso debe mantener una lista de miembros incluyendo los procesos entrantes, salientes y caídos. El método funciona mejor con grupos pequeños de procesos que nunca cambian sus grupos.
- Empeora el cuello de botella → todos los procesos están envueltos en todas las decisiones que incluyen entrar o no a una sección crítica.

Se pueden hacer algunas mejoras pero el algoritmo es muy malo, aún contra el centralizado.

Token Ring



Se tiene una red de tipo bus sin ningún tipo de ordenamiento para los procesos, pero cada uno sabe quién es su sucesor. Cuando el anillo es inicializado, se le da un token al proceso 0. El token circula alrededor del anillo y pasa desde el proceso k al proceso $k+1$ en mensajes point-to-point. Cuando un proceso adquiere el token de su vecino, chequea para ver si quiere entrar a una región crítica. Si quiere entrar a una región crítica, el proceso entra a la región, hace todo el trabajo que necesita y la deja. Luego de salir, pasa el token a lo largo del anillo. No está permitido entrar a la sección crítica dos veces usando el mismo token. Si un proceso tiene el token pero no quiere entrar a la sección crítica, pasa el mensaje.

Sólo un proceso tiene el token en cada momento, de modo que sólo un proceso puede entrar a la región crítica. Como el token pasa a lo largo de todo el anillo, no puede haber starvation. Una vez que un proceso decide que quiere entrar a una región crítica, tiene que esperar a que le den el token.

Si el token se pierde tiene que ser regenerado, y detectar la pérdida es difícil: no se puede saber con exactitud el tiempo que le toma a un proceso tener el token. Un proceso caído es detectado cuando no recibimos el ACK del receptor.

Centralizado vs. Distribuido vs. Token Ring

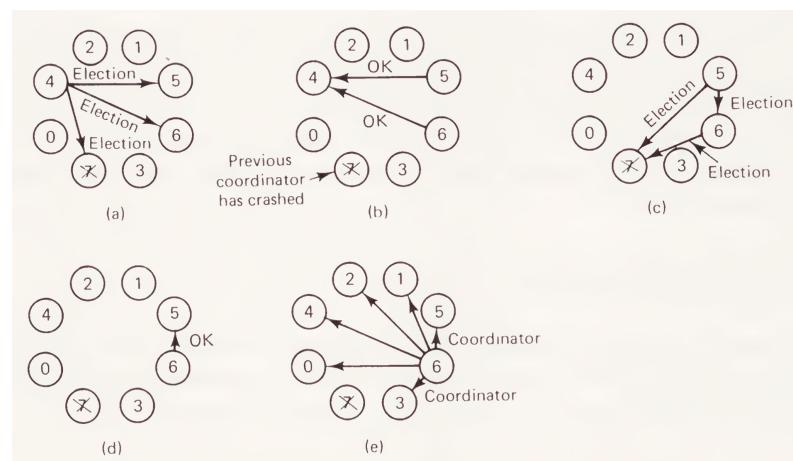
Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Distributed	$2(n - 1)$	$2(n - 1)$	Crash of any process
Token ring	1 to ∞	0 to $n - 1$	Lost token, process crash

El algoritmo centralizado es el más simple y el más eficiente.

Algoritmos de Elección

Se asegura que cuando una elección empieza, termina con todos los procesos aceptando quién será el nuevo coordinador.

Bully



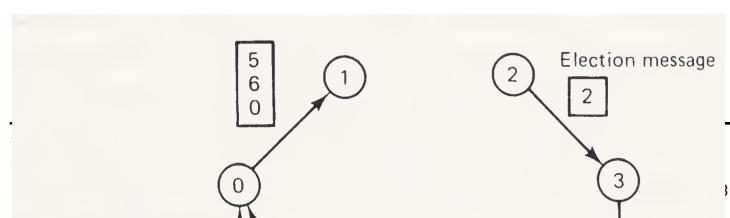
Cuando un proceso se da cuenta que el coordinador se cayó (no responde a sus requests) inicia una elección:

1. P envía un mensaje de elección a todos los procesos con número mayor a él
2. Si nadie responde, P gana la elección y se convierte en el coordinador
3. Si algún proceso contesta termina el trabajo de P

Un proceso puede recibir un mensaje de elección en cualquier momento de alguno de los procesos con números más pequeños que él. Cuando el mensaje llega, el receptor envía un mensaje de OK al emisor para indicar que está vivo y se va a encargar de la elección. El receptor empieza una elección si es que no está llevando a cabo una. Eventualmente todos los procesos terminan la elección menos uno, que será el nuevo coordinador. El coordinador anuncia su victoria enviando a todos los demás un mensaje avisándoles que él es el nuevo coordinador.

Si un proceso previamente caído vuelve a la red, comienza una elección. Si sucede que tiene el número más grande de procesos ejecutando, va a ganar la elección y tomar el trabajo del coordinador.

Ring



Asumimos que los procesos están físicamente o lógicamente ordenados de forma tal

que cada uno conoce quién es su sucesor. Cuando algún proceso se da cuenta que el coordinador se cayó, construye un mensaje de elección conteniendo su propio número de proceso y envía el mensaje a su sucesor. Si el sucesor está caído, el emisor lo salta y envía el mensaje al siguiente miembro del anillo, o el siguiente a él hasta que algún proceso en ejecución sea localizado. En cada paso, el emisor agrega su propio número de proceso a la lista en el mensaje. Eventualmente el mensaje vuelve al proceso que comenzó la elección, quien reconoce el evento cuando recibe un mensaje conteniendo su propio número de proceso. En ese punto, envía un mensaje de tipo coordinator y lo hace circular una vez más, esta vez para informarle al resto quién es el nuevo coordinador y quiénes son los procesos que pertenecen al anillo. Cuando este mensaje circula una vez, es eliminado y todos los procesos vuelven a trabajar.

Transacciones

- conjunto de procesos independientes, cada uno puede fallar aleatoriamente
- los errores en la comunicación son manejados transparentemente por la capa de comunicación
- *Stable Storage*
 - se implementa con discos
 - la probabilidad de perder los datos es extremadamente pequeña

Primitivas

- ❖ **BEGIN TRANSACTION** → inicia la transacción
- ❖ **END TRANSACTION** → finaliza la transacción e intenta hacer commit
- ❖ **ABORT TRANSACTION** → finaliza forzosamente la transacción y restaura los valores anteriores
- ❖ **READ** → lee datos de un archivo u otro objeto
- ❖ **WRITE** → escribe datos a un archivo u otro objeto

Propiedades ACID

- ★ **Atomicity (Atómicas):** la transacción no puede ser dividida
- ★ **Consistency (Consistentes):** la transacción cumple con todos los invariantes del sistema
- ★ **Isolation (Aisladas o Serializadas):** las transacciones concurrentes no interfieren con ellas mismas
- ★ **Durability (Durables):** una vez que se commitean los cambios, son permanentes

Implementaciones

Private Workspace

- al iniciar una transacción, el proceso recibe una copia de todos los archivos a los cuales tiene acceso
- hasta que hace commit, el proceso trabaja con la copia
- al hacer commit, se persisten los cambios
- extremadamente costoso salvo por optimizaciones

Write Ahead Log

- los archivos se modifican *in place*, pero se mantiene una lista de los cambios aplicados (primero se escribe la lista y luego se modifica el archivo)
- al commitear la transacción, se escribe un registro commit en el log
- si la transacción se aborta, se lee el log de atrás hacia adelante para deshacer los cambios (*rollback*)

Two-Phase Commit

- el coordinador es aquel proceso que ejecuta la transacción
1.
 - a. el coordinador escribe *prepare* en su log y envía el mensaje al resto de los procesos
 - b. los procesos que reciben el mensaje, escriben *ready* en el log y envían el mensaje al coordinador
 2.
 - a. el coordinador hace los cambios y envía el mensaje *commit* al resto de los procesos
 - b. los procesos que reciben el mensaje, escriben *commit* en el log y envían *finished* al coordinador

Two-Phase Locking

1. Expansión
 - a. se toman todos los locks a usar
 2. Contracción
 - a. se liberan todos los locks (no se pueden tomar nuevos)
- + Garantiza propiedad serializable para las transacciones
- Pueden ocurrir deadlocks
- Strict two-phase locking → la contracción ocurre después del commit

Concurrencia Optimista (GitHub)

- el proceso modifica los archivos sin ningún control, esperando que no haya conflictos

- al commitear, se verifica si el resto de las transacciones modificó los mismos archivos en cuyo caso se aborta la transacción (== conflictos)
- + libre de deadlocks y favorece el paralelismo
- rehacer todo puede ser costoso en condiciones de alta carga

Timestamps

- timestamps únicos globales para garantizar orden
- cada archivo tiene dos timestamps
 - lectura y escritura
 - qué transacción hizo la última operación en cada caso
- cada transacción al iniciarse recibe un timestamp
- se compara el timestamp de la transacción con los timestamps del archivo
 - si es mayor la transacción está en orden y se procede con la operación
 - si es menor la transacción se aborta
- al commitear se actualizan los timestamps del archivo

Detección de Deadlocks

Algoritmo Centralizado

- el proceso coordinador mantiene el grafo de uso de recursos
- los procesos envían mensajes al coordinador cuando obtienen / liberan un recurso y el coordinador actualiza el grafo
- los mensajes pueden llegar desordenados y generar falsos deadlocks
- posible solución: utilizar timestamps globales para ordenar los mensajes (algoritmo de Lamport)

Algoritmo distribuido

- cuando un proceso debe esperar por un recurso, envía un *probe message* al proceso que tiene el recurso con:
 - id del proceso que se bloquea
 - id del proceso que envía el mensaje
 - id del proceso destinatario
- al recibir el mensaje, el proceso actualiza el id del proceso que envía y el id del destinatario y lo envía a los procesos que tienen el recurso que necesita
- si el mensaje llega al proceso original, hay un ciclo en el grafo

Algoritmo Wait-Die

- se asigna un timestamp único y global a cada transacción al iniciar (algoritmo de Lamport)
- cuando un proceso está por bloquearse en un recurso (T1) que tiene otro proceso (T2), se comparan los timestamps...
 - $T1 < T2 \rightarrow T1$ espera (== el viejo espera al nuevo)

- $T_1 > T_2 \rightarrow T_1$ aborta (== el nuevo NO espera al viejo, aborta)

Algoritmo Wound-Wait

- se asigna un timestamp único y global a cada transacción al iniciar (algoritmo de Lamport)
- cuando un proceso está por bloquearse en un recurso (T_1) que tiene otro proceso (T_2), se comparan los timestamps...
 - $T_1 < T_2 \rightarrow T_2$ aborta (== el nuevo aborta para que el viejo lo tome)
 - $T_1 > T_2 \rightarrow T_1$ espera (== el nuevo espera al viejo)

Ambientes Distribuidos

Entidad

Unidad de cómputo de ambiente informático distribuido (proceso, procesador, etc.).

Capacidades

- acceso de lectura y escritura a una memoria local (no compartida con otras)
 - registro de estado: $status(x)$
 - registro de valor de entrada: $value(x)$
- procesamiento local
- comunicación: preparación, transmisión y recepción de mensajes
- Setear y Resetear un reloj local

Eventos Externos

Solamente responde a eventos externos: es reactiva

- llegada de un mensaje
- activación del reloj
- impulso espontáneo

Acciones

Secuencia finita e indivisible de operaciones. Es atómica porque se ejecuta sin interrupciones.

Reglas

Relación entre el evento que ocurre y el estado en el que se encuentra la entidad cuando ocurre dicho evento, de modo tal que $estado \times evento \rightarrow acción$.

Comportamiento

Conjunto de todas las reglas que obedece una entidad

- para cada posible evento y estado debe existir una única regla

- se llama también protocolo o algoritmo distribuido de la entidad

Es homogéneo si todas las entidades que lo componen tienen el mismo comportamiento.

Todo comportamiento colectivo se puede transformar en homogéneo.

Comunicación

Mediante mensajes (un mensaje es una secuencia finita de bits). Puede ocurrir que una entidad sólo pueda comunicarse con un subconjunto del resto de las entidades.

Restricciones de Confiabilidad

- Entrega garantizada → cualquier mensaje enviado será recibido con su contenido intacto
- Confiabilidad parcial → no ocurrirán fallas
- Confiabilidad total → no han ocurrido ni ocurrirán fallas

Restricciones Temporales

- Delays de comunicación acotados: existe una constante Δ tal que en ausencia de fallas el delay de cualquier mensaje en el enlace es a lo sumo Δ
- Delays de comunicación unitarios: en ausencia de fallas, el delay de cualquier mensaje en un enlace es igual a una unidad de tiempo
- Reloj sincronizados: todos los relojes locales se incrementan simultáneamente y el intervalo de incremento es constante

Costo y Complejidad

Medidas de comparación de los algoritmos distribuidos:

- **Cantidad de Actividades de Comunicación**
 - Cantidad de transmisiones o costo de mensajes, M
 - Carga de trabajo por entidad y carga de transmisión
- **Tiempo**
 - Tiempo total de ejecución del protocolo
 - Tiempo ideal de ejecución: tiempo medido bajo ciertas condiciones, como delays de comunicación unitarios y relojes sincronizados

Tiempo y Eventos

Tipos de Eventos

- Impulso espontáneo
- Recepción de un mensaje
- Alarma del reloj activada

Los eventos desencadenan acciones en un tiempo futuro. Los distintos delays resultan en distintas ejecuciones del protocolo con posibles resultados diferentes.

- Los eventos disparan acciones que pueden generar nuevos eventos
- Si suceden, los nuevos eventos ocurrirán en un tiempo futuro
- Una ejecución se describe por la secuencia de eventos que ocurrieron

Estados y Configuraciones

- Estado interno de x en el instante t $\sigma(x, t)$: contenido de los registros de x y el valor del reloj c_x en el instante t
- El estado interno de una entidad cambia con la ocurrencia de eventos

Sea una entidad x que recibe el mismo evento en dos ejecuciones distintas, y σ_1 y σ_2 los estados internos, si $\sigma_1 = \sigma_2 \Rightarrow$ el nuevo estado interno de x será el mismo en ambas ejecuciones.

Conocimiento

Conocimiento Local \rightarrow contenido de la memoria local de x y la información que se deriva. En ausencia de fallas, el conocimiento no puede perderse.

Tipos

- Información Métrica: información numérica sobre la red (número de nodos del grafo, número de arcos del grafo, diámetro del grafo, etc)
- Propiedades Topológicas: conocimiento sobre las propiedades de la topología (el grafo es un anillo/acíclico/etc)
- Mapas Topológicos: un mapa de la vecindad de la entidad hasta una distancia d (matriz de adyacencia del grafo)

Referencias

-  → importante, revisar
-  → ignorar
- **definición**
- aclaración, prestar atención
- **aclaración, prestar atención**