SISTEMA DISTRIBUIDO DE SERVICIOS DE TAQUILLA VIRTUAL

DISEÑO AVANZADO

Aarón Riveiro Vilar
Adrian Cabaleiro Althoff
Iván Pillado Rodríguez
Manuel Martínez Vaamonde

Funcionamiento del algoritmo

- Para el mutex inter-nodo partimos del algoritmo de testigo de Ricart-Agrawala.
- Realizaremos las siguientes modificaciones:
  - Añadimos prioridad
  - Añadimos procesos consulta que pueden leer concurrentemente cuando uno de ellos obtiene el token
  - Añadimos adelantamientos para evitar la inanición de las consultas
  - Añadiremos un algoritmo que nos permita gestionar prioridad, concurrencia de lectores y adelantamientos dentro de los procesos de un nodo

Punto de partida

Algoritmo de testigo

	1	2	3
P	0	0	0

	1	2	3
P	0	0	0

Nodo 2

P -> 0

	1	2	3
P	0	0	0

Nodo 1

P -> 0

	1	2	3
A	0	0	0

QUIERE

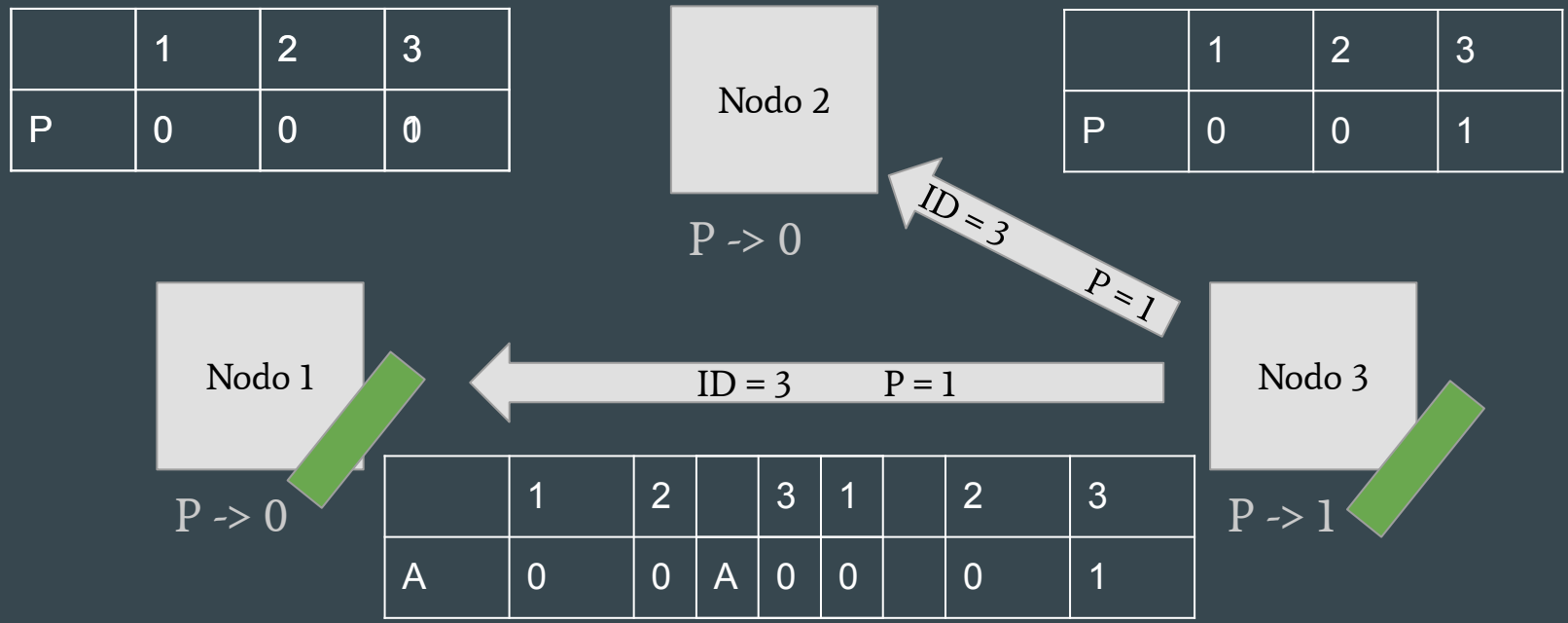
Nodo 3

P -> 0

Punto de partida

Algoritmo de testigo

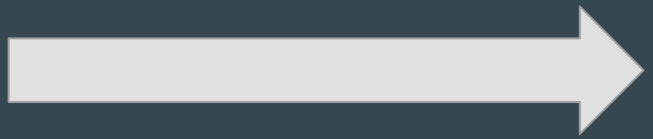
	1	2	3
P	0	0	0



Prioridad

	1	2	3
P	0	0	1

	1	2	3
A	0	0	1



T0	1	2	3
P	0	0	1

T0	1	2	3
A	0	0	1

T1	1	2	3
P	0	0	0

T1	1	2	3
A	0	0	0

T2	1	2	3
P	0	0	0

T2	1	2	3
A	0	0	0

En el caso anterior determinamos el siguiente nodo recorriendo peticiones y atendidas hasta que encontramos una tal que atendidas[i] < peticiones[i]
Añadir prioridad es la modificación más sencilla ya que basta con triplicar los vectores de atendidas y peticiones.
Ahora hay un vector para cada prioridad y cuando es preciso obtener un nodo siguiente se recorren en orden de prioridad.

T0	1	2	3
P	0	0	1

T0	1	2	3
A	0	0	0

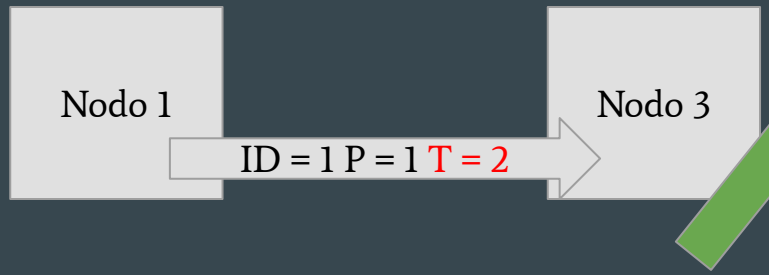
T1	1	2	3
P	0	0	0

T1	1	2	3
A	0	0	0

T2	1	2	3
P	1	0	0

T2	1	2	3
A	0	0	0

También es necesario que los nodos comuniquen la prioridad de sus peticiones cuando las transmiten para que estas se apunten en el array correcto



T0	1	2	3
P	0	0	1

T0	1	2	3
A	0	0	1

T1	1	2	3
P	0	0	0

T1	1	2	3
A	0	0	0

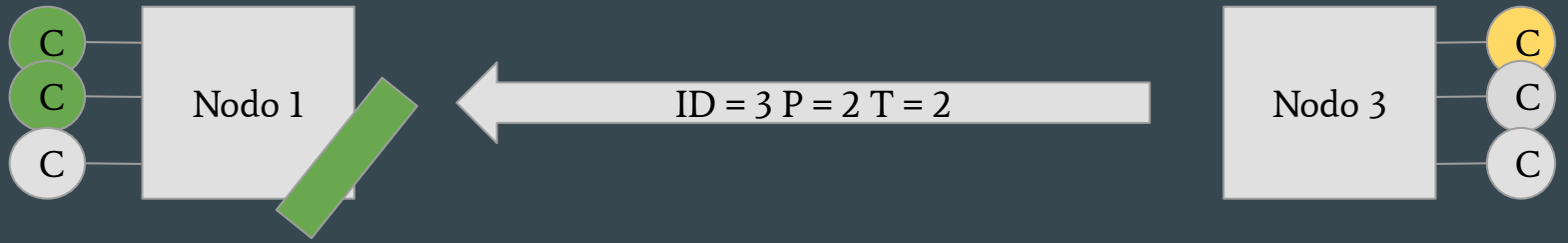
T2	1	2	3
P	1	0	0

T2	1	2	3
A	0	0	0

Procesos consulta (Procesos concurrentes inter e intra nodo)

Problema:

- El token es único
- Para permitir que dos consultas de nodos diferentes lean concurrentemente debemos duplicar el token
- Si hemos duplicado el token debemos asegurarnos de que todas las copias han sido recuperadas antes de enviar el token real



Procesos consulta (Procesos concurrentes inter e intra nodo)

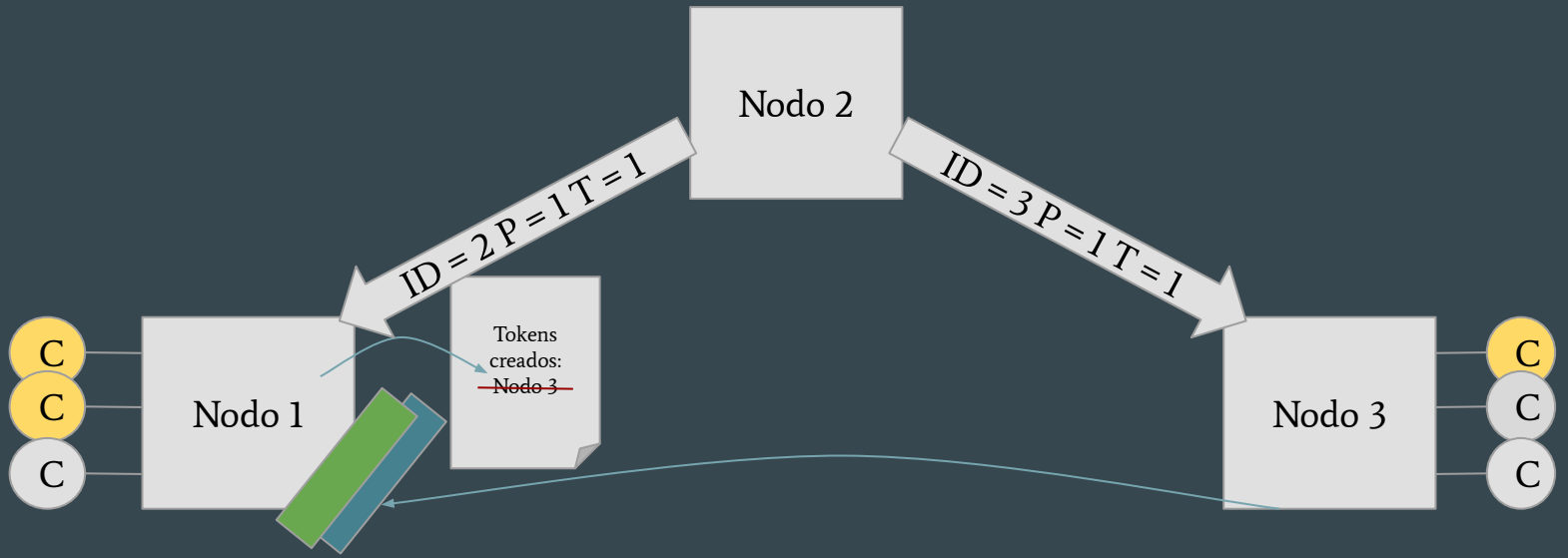
Solución:

- Si el nodo que sostiene el token tiene procesos consulta en SC crea un token de consulta cuando recibe una petición de un nodo con prioridad 2 (Consulta)
- Este token no puede ser transferido, sólo devuelto y sólo es válido para consultas
- El nodo que crea el token de consulta apunta el ID del nodo al que ha enviado el token y sólo soltará el token real cuando todas las instancias de tokens de consulta hayan sido devueltas

Procesos consulta (Procesos concurrentes inter e intra nodo)

Esta situación continúa hasta que un proceso más prioritario quiere entrar en SC

Cuando esto sucede las consultas paran de competir por SC y los token de consulta son devueltos



Adelantamientos

T0: Anulaciones, Pagos
T1: Reservas, Administración
T2: Consultas

Con el reparto de prioridades anterior T0 no puede causar inanición a T1 ya que el número de anulaciones es limitado. Los usuarios de Administración están bajo control del cliente por lo que asumimos que no causarán inanición a T2 y que el volumen de pagos fuese suficiente como para causar inanición a T2 no sería un problema.

Problema:

- Reservas puede causar inanición a T2
- La inanición de T2 abre la posibilidad de que un ataque DoS a Reservas deje consultas inaccesible a los clientes y evite que estas se conviertan en pagos

Adelantamientos

Solución:

- Limitar el número de reservas pueden pasar a SC cuando hay una consulta a la espera

Para ello cuando se procesa una reserva se verifica si hay consultas a la espera en el nodo a través de una variable de memoria compartida y también si hay consultas a la espera en otros nodos si vec_peticiones[2][n] > vec_atendidas[2][n] para n $\in$ [0,N-1]

Prioridad consultas

Si lo hay se incrementa un contador de adelantamientos. Cuando este llega a MAX_ADELANTAMIENTOS se da paso a una consulta y se pone el contador a 0

Adelantamientos

Para persistir el estado del contador entre nodos se añade una variable adelantamientos al token

El nodo que recibe el token pone su variable adelantamientos al valor recibido en el token

Concurrencia intra-nodo

Para permitir que múltiples procesos de tipos diferentes se ejecuten en un mismo nodo es necesario implementar un algoritmo que gestione prioridad, concurrencia de lectores y adelantamientos.

En esencia este es un problema de lectores y escritores con una serie de modificaciones importantes:

- Espera por token
- Existen 3 niveles de prioridad
- Sincronizaciones cuando se pierde el token

Concurrencia intra-nodo - Espera por token

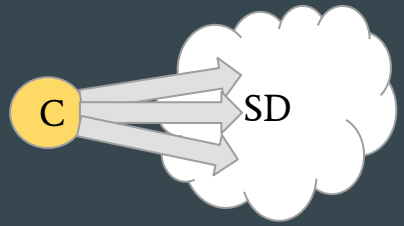
A diferencia del problema clásico de lectores-escritores es preciso verificar si tenemos el token antes de ejecutar la lógica de sincronización

En caso negativo el primer proceso que quiere SC broadcastea una petición al SD y se suspende a la espera del token

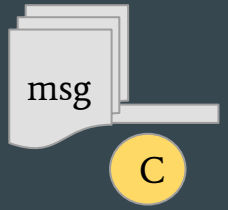
Los procesos que quieran entrar a SC mientas es primer proceso espera comprueban si ya se ha broadcasteado una petición al SD de su nivel de prioridad. Si no se ha hecho la hacen y acto seguido se suspenden en un semáforo de paso controlado por el proceso que se suspendió esperando al token

Concurrencia intra-nodo - Espera por token

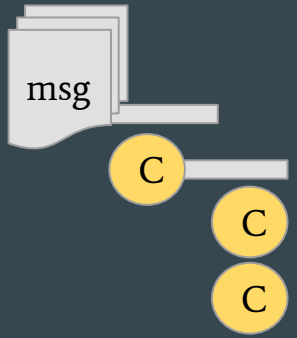
Si token == 0



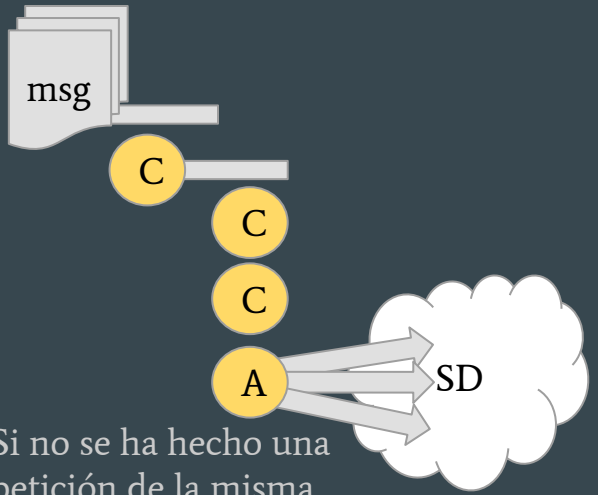
El primer proceso en querer del nodo broadcastea peticiones al SD



Se suspende esperando el token en la cola de mensajes



Si otros procesos quieren de manera previa a la obtención del token se suspenden en un semáforo de paso controlado por el primero



Si no se ha hecho una petición de la misma prioridad que un proceso que quiere este la hace y posteriormente se suspende en la sincronización con el primero

Concurrencia intra-nodo - Prioridad

En el problema de lectores y escritores cuando uno tiene prioridad sobre el otro el menos prioritario comprueba si el más prioritario quiere y en caso afirmativo se suspende en una sincronización con el último del grupo prioritario

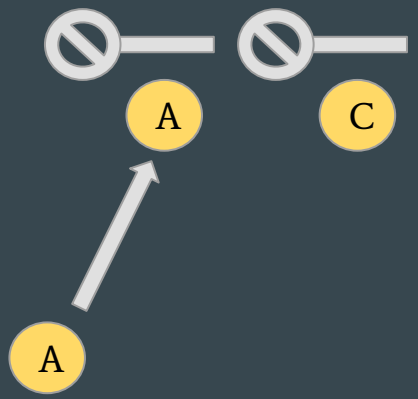
En este caso existen múltiple posibilidades ya que un proceso T2 puede estar a la espera de un grupo de procesos T0 o T1 por lo que el último en salir de SC del grupo T0 debe comprobar si T1 quiere, en caso afirmativo hay procesos suspendidos esperando a que T0 los despierten. Si de lo contrario T1 no quiere se repite esta comprobación para T2.

Cuando el último del grupo T1 sale de SC comprueba si T2 quiere y en caso afirmativo despierta los procesos T2 suspendidos esperando a que T1 o T0 los despierten.

Concurrencia intra-nodo - Prioridad

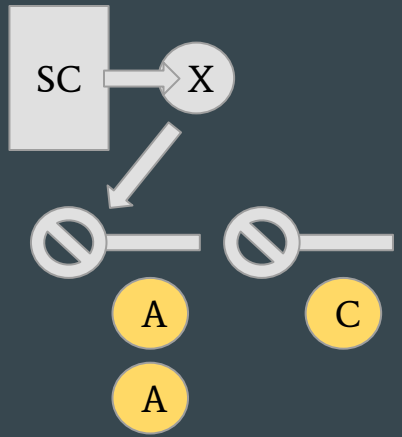
Vector quiere

T0	T1	T2
3	2	1



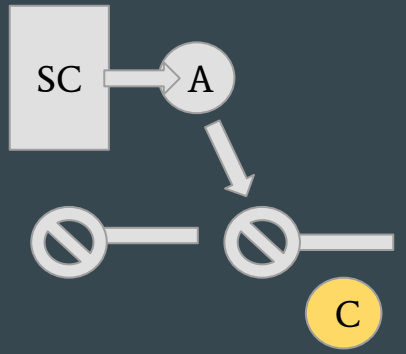
Vector quiere

T0	T1	T2
0	2	1



Vector quiere

T0	T1	T2
0	0	1



Concurrencia intra-nodo - Sincronizaciones pérdida token

La lógica del apartado anterior asume que siempre que al menos un proceso quiere dentro del nodo se retiene el token. Se debe tener en cuenta que si se reciben peticiones de prioridad superior a los procesos que están a la espera el token se enviará a otro nodo y los procesos deben ser despertados no para entrar en SC sino para esperar por el token.

Para esto el último proceso en ejecutar SC antes de que se pierda el token pone la variable token a 0 y despierta los procesos que estaban suspendidos en los semáforos de paso del apartado anterior. Al no haber token comprueban si ya hay una petición para su prioridad y en caso negativo la broadcastean al SD posteriormente el primero se suspende en la cola de mensajes esperando el token y el resto se suspenden en una sincronización con este.

Concurrencia intra-nodo - Sincronizaciones pérdida token

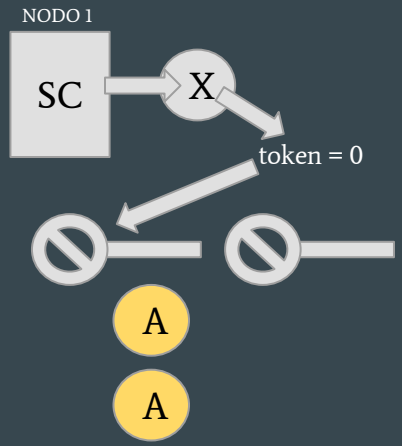
Vector quiere

T0	T1	T2
0	2	0

Peticiones/Atendidas

T0	1	2	3
P	0	0	2

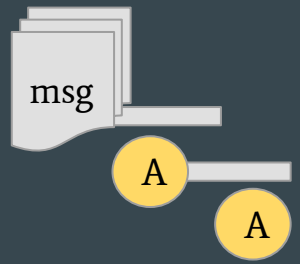
T0	1	2	3
A	0	0	1



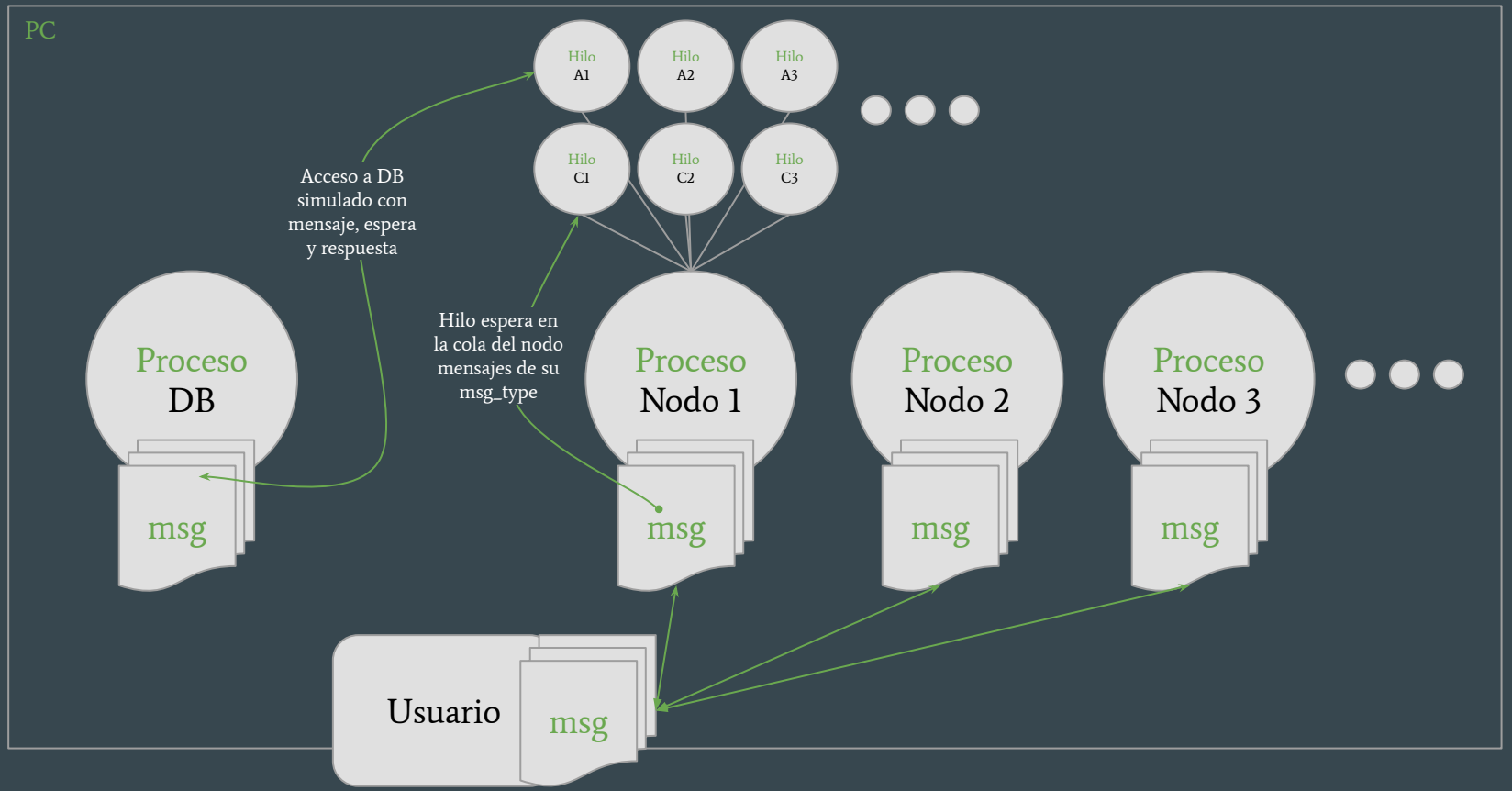
Peticiones/Atendidas

T1	1	2	3
P	2	0	0

T1	1	2	3
A	1	0	0



Topología Real



Topología simulada

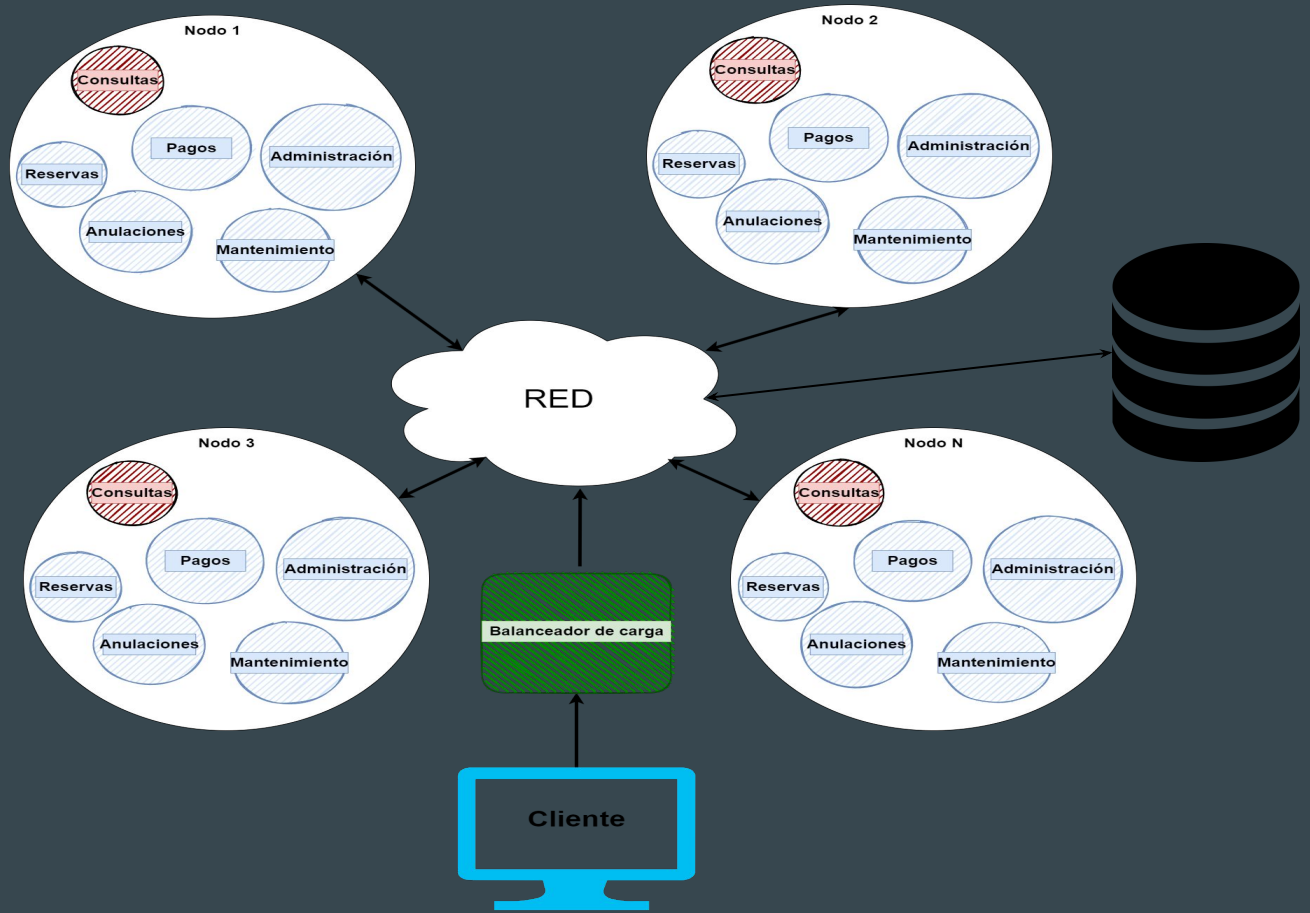


Diagrama de Flujo - Proceso Escritor

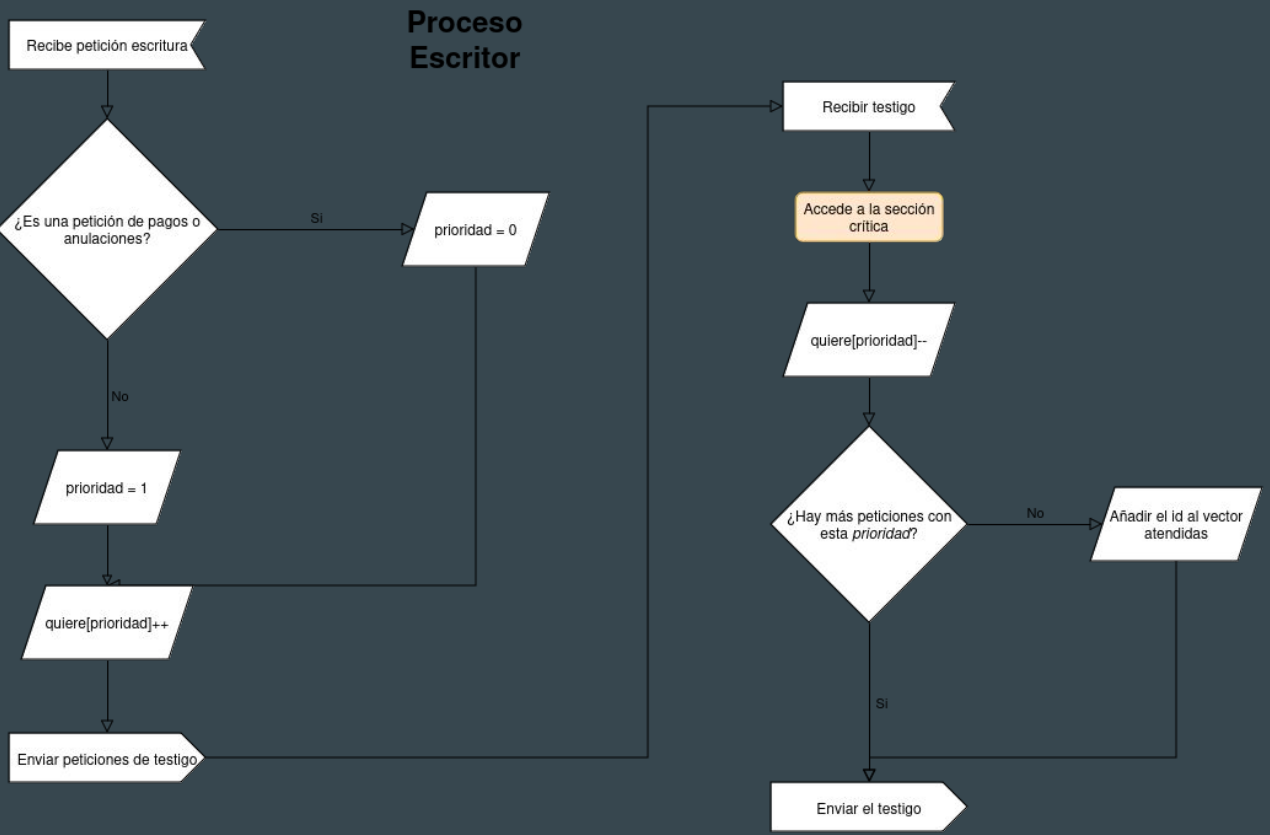


Diagrama de Flujo - Proceso Lector

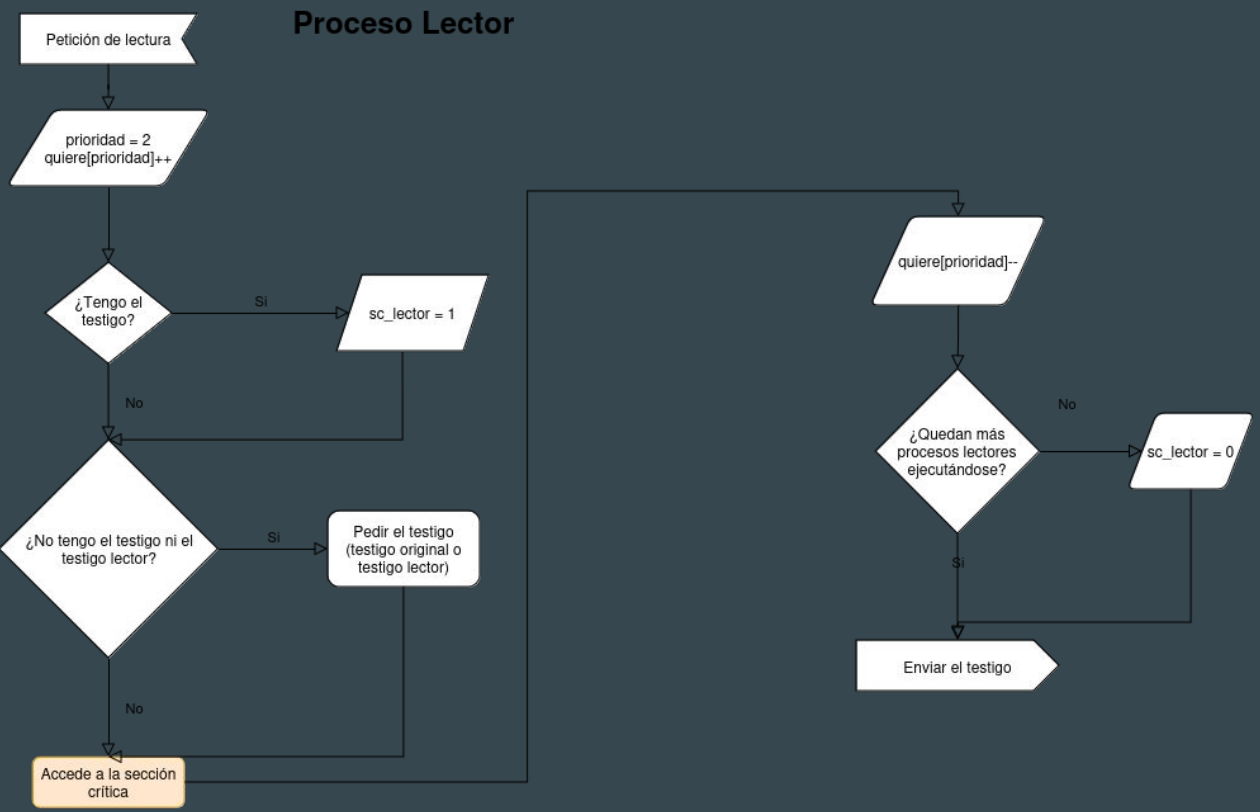
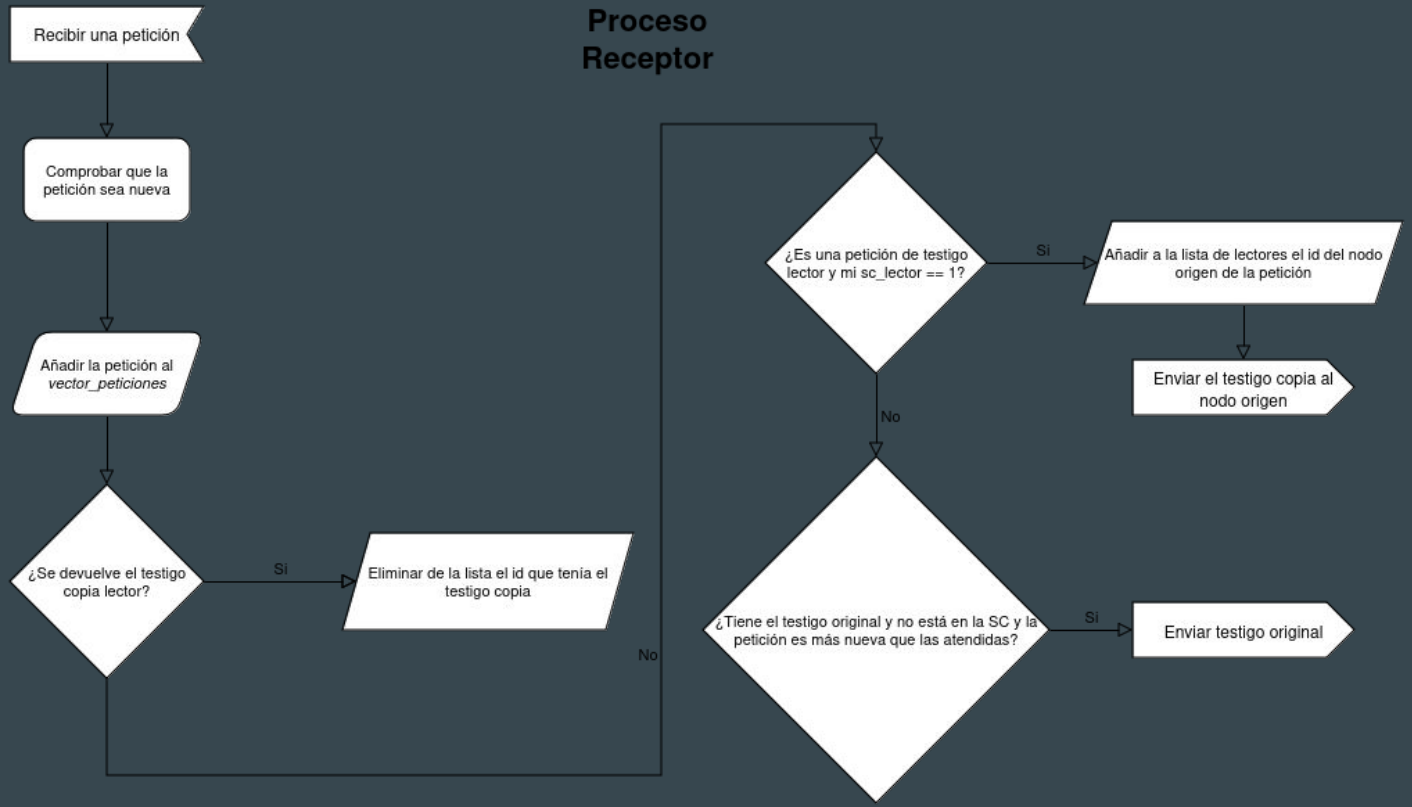


Diagrama de Flujo - Proceso Receptor



Pseudocódigo - Proceso Escritor

```
while (1) {
        receive(SOLICITUD_ESCRITOR);

        int prioridad;
        if (SOLICITUD_ESCRITOR == 'pago' || SOLICITUD_ESCRITOR == 'anulacion') {
                prioridad = 0;
        } else {
                prioridad = 1;
        }

        quiere[prioridad]++;                                    mutex
        if ((NOT token) OR (prioridad > token_prioridad) OR (token_prioridad == 2)) {

                peticion = peticion + 1;

                for (i = 0; i < N-1; i++) {
                        send(REQUEST, id[i], {id, peticion, prioridad, 'escritor'});
                }

                receive(TOKEN, {&vector_atendidas, &token_prioridad});


                for (i = 0; i < N-1; i++) {                                     mutex
                        vector_atendidas[prioridad][i] = msg_token.vector_atendidas[prioridad][i];
                }

                token = 1;                                      mutex
        }

        seccion_critica = 1;                                    mutex
```

```
SECCIÓN CRÍTICA

quiere[prioridad]--;                                    mutex
salir = 1;
for (i=0; i<=prioridad; i++) {
        if (quiere[i] != 0) {salir = 0; break}
}
if (salir) {
        vector_atendidas[prioridad][id] = peticion;     mutex
}
seccion_critica = 0;                                    mutex

int token_enviado = 0;

for(j = 0; j < 3; j++) {
        for (i = (id + 1) % N; i != id; i = (i + 1) % N) {
                if(vector_peticiones[j][i] > vector_atendidas[j][i]) {   mutex
                        token_enviado = 1;
                        enviar_token(i,0,j);
                        break;
                }
        }
        if(token_enviado)
                break;
        }
}
```

Pseudocódigo - Proceso Lector

```
while (1) {
        receive(SOLICITUD_LECTOR);

        int prioridad = 2;

        quiere[prioridad]++;

        if (token) {                                    mutex
                sc_lector = 1;
        }

        if ((NOT token) AND (NOT token_lector)) OR (prioridad > token_prioridad)) {    mutex

                peticion = peticion + 1;

                for (i = 0; i < N-1; i++) {
                        send(REQUEST, id[i], {id, peticion, prioridad, 'lector'});
                }

                receive(TOKEN, {&vector_atendidas, tipo, id_origen});
                token_prioridad = 2;
                parar = 0;

                for (i = 0; i < N-1; i++) {
                        vector_atendidas[prioridad][i] = msg_token.vector_atendidas[prioridad][i];    mutex
                }
                if(msg_token.tipo == 0)
                        token = 1;                      mutex
                else {
                        id_origen_testigo = msg_token.id_origen;
                        token_lector = 1;               mutex
                }
        }

        seccion_critica = 1;                            mutex
```

SECCIÓN CRÍTICA

```
quiere[prioridad]--;                                    mutex
if (quiere[prioridad] == 0) {
        vector_atendidas[prioridad][id] = peticion;     mutex
}

seccion_critica = 0;                                    mutex

if (lista_vacia()) {
        sc_lector = 0;                                  mutex
}

if (token_lector == 1) {                                mutex
        send(REQUEST, {id_nodo_origen, null, 2, 'token'});
        token_lector = 0;                               mutex
}

int token_enviado = 0;

for(j = 0; j < 3; j++) {
        for (i = (id + 1) % N; i != id; i = (i + 1) % N) {
                if(vector_peticiones[j][i] > vector_atendidas[j][i]) {    mutex
                        token_enviado = 1;
                        enviar_token(i,0,j);
                        break;
                }
        }
        if(token_enviado)
                break;
}
```

Pseudocódigo - Proceso Receptor

```
int id_nodo_origen, num_peticion_nodo_origen, prioridad_origen, tipo_origen;

while (1) {
        receive(REQUEST, {&id_nodo_origen, &num_peticion_nodo_origen, prioridad_origen, tipo_origen});

        id_nodo_origen = msg_peticion.id_nodo_origen;
        num_peticion_nodo_origen = msg_peticion.num_peticion_nodo_origen;
        prioridad_origen = msg_peticion.prioridad_origen;
        tipo_origen = msg_peticion.tipo_origen;

        vector_peticiones[prioridad_origen][id_nodo_origen] = max(vector_peticiones[prioridad_origen][id_nodo_origen],
        num_peticion_nodo_origen);

        if (tipo_origen == 'token') {
                eliminar_lista(id_nodo_origen);
        } else if (tipo_origen == 'lector' && sc_lector == 1) {
                lista = id_nodo_origen;
        enviar_token(id_nodo_origen , 1);
        } else if (token AND (NOT seccion_critica) AND vector_peticiones[prioridad_origen][id_nodo_origen] >
        vector_atendidas[prioridad_origen][id_nodo_origen]) {
        enviar_token(id_nodo_origen, 0)
        }
}
```

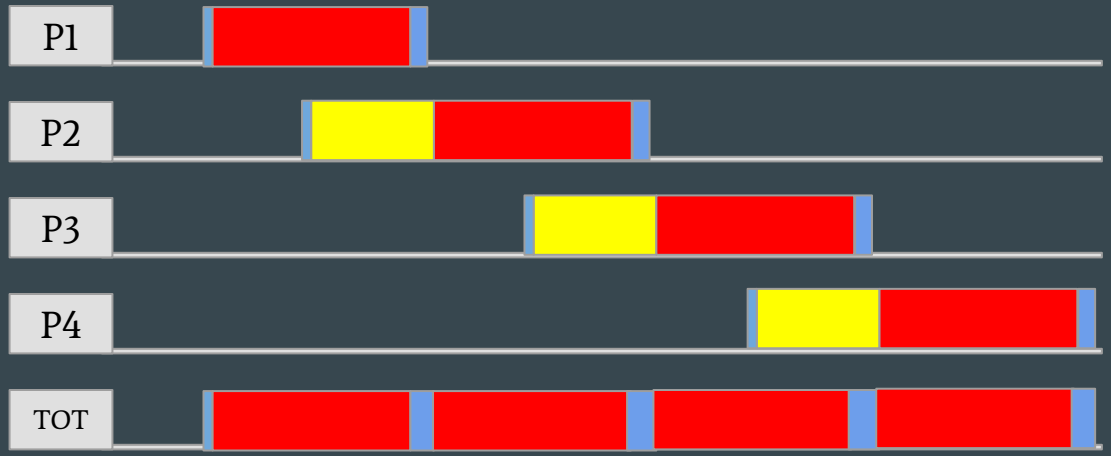
mutex

mutex

Metodología métricas

1. Tiempo ejecución código de sincronización
2. Peticiones / segundo - nodo
3. Tiempo de respuesta de petición
4. Factor de utilización promedio

Metodología métricas - Tiempo de ejecución de código de sincronización



En azul - Tiempo código sincronización
En amarillo - Tiempo espera
En rojo - Tiempo SC

Stat - tiempo código sincronización/tiempo SC
(Menos es mejor)

¿Cómo realizar la métrica?
Los procesos toman timestamps cuando:

- Empiezan/paran de ejecutar código sincronización
- Entran/salen de CS
- Entran/salen de espera

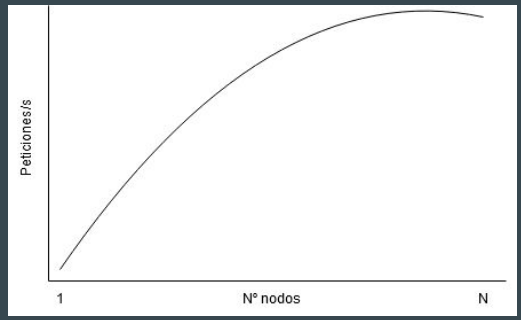
Con todos los timestamps de todos los procesos de un SD podemos derivar una gráfica similar a la de la izquierda.
Esta métrica es un indicador de eficiencia en el algoritmo de sincronización.

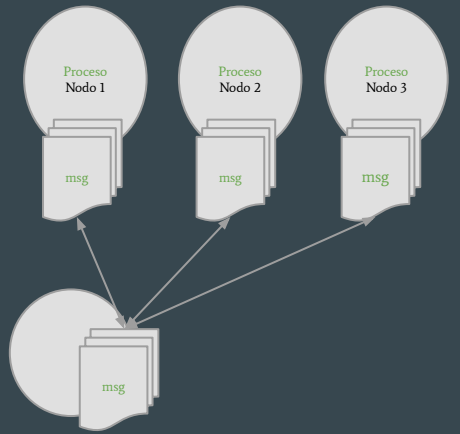
Metodología métricas - Peticiones / segundo - nodo

Como se ha explicado en la parte de topología real simulamos la red como el envío de mensajes a las colas de mensajes de los nodos.

Para poder determinar cual es el número máximo de peticiones que un número de nodos puede atender iniciamos el SD con el número de nodos que estamos perfilando y haciendo uso de un programa escrito para este fin mantenemos las colas de todos los nodos con al menos una peticion de cada tipo de esta forma nos aseguramos de que todos los procesos del SD tienen una petición que atender.

Una vez iniciado el programa que floodea el DS contamos el número de peticiones atendidas que recibimos cada segundo.



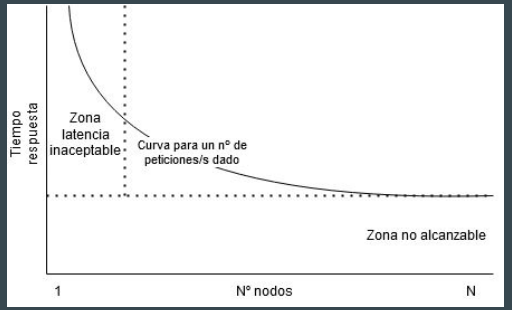


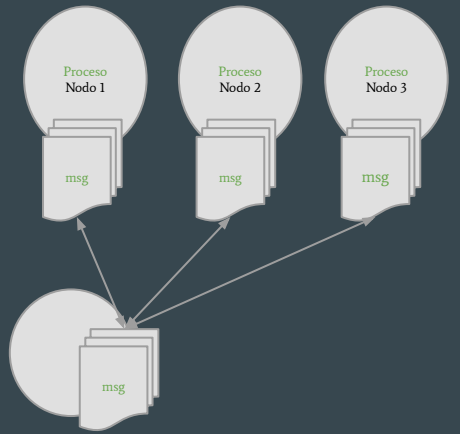
Metodología métricas - Tiempo de respuesta de petición

El método para obtener esta estadística es similar al anterior.

En lugar de floodear el SD introduciendo más peticiones de las que puede atender ahora introduciremos peticiones con una frecuencia dada.

En lugar de contar cuantas peticiones son atendidas cada segundo tomaremos nota del tiempo que tarda cada petición en ser atendida y tomaremos el valor medio.





Metodología métricas - Factor de utilización promedio

Los procesos toman un timestamp cada vez que:

- Se entran/salen de espera
- Empiezan/terminan su ejecución

El factor de utilización es la media entre todos los procesos de todos los nodos de el tiempo de ejecución menos el sumatorio de todos los tiempos de espera partido por el tiempo de ejecución

FU = sum((t_ejecucion_pi - sum(tiempo_espera_i_pi))/t_ejecucion_pi)/i