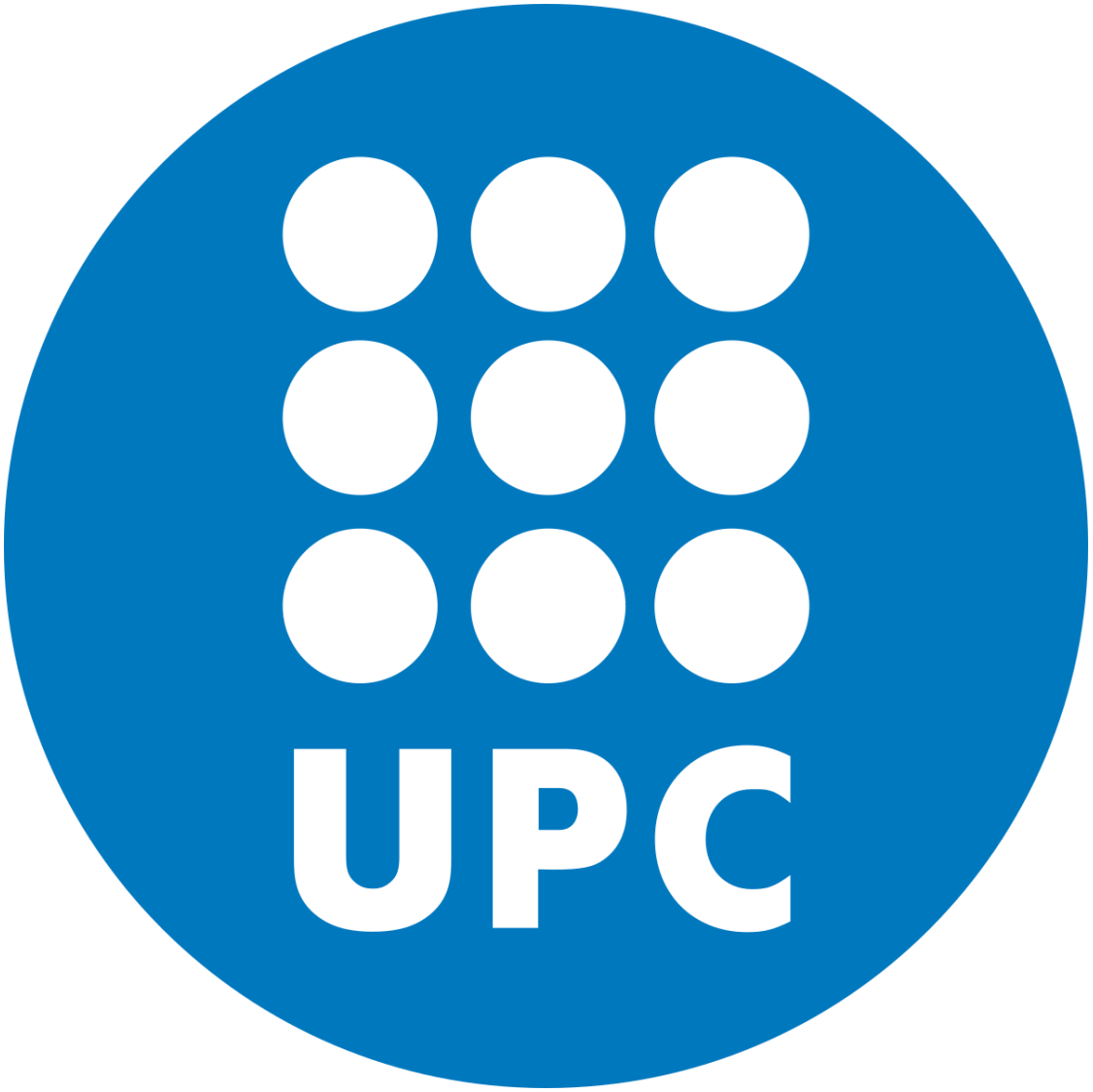
Sistemas operativos distribuidos y en redes

Muty: a distributed mutual-exclusion lock



Alumnes: Alejandra Lisette Rocha i Massimo Wang

Curs: 2024/2025 Q5

1. The architecture

Experiments

i) Make tests with different Sleep and Work parameters to analyze how this lock implementation responds to different contention degrees.

Hemos distinguido diferentes casos según los valores que le pasamos como parámetros de ejecución y a parte hemos decidido que para cada caso estuviera ejecutándose alrededor de 20 segundos:

- Tiempo de sleep mayor que work, sleep: 2000 y work: 1000.

```
14> muty:stop().
John: 18 locks taken, 817.8333333333334 ms (avg) for taking, 0 withdrawals
Ringo: 19 locks taken, 679.1578947368421 ms (avg) for taking, 0 withdrawals
Paul: 18 locks taken, 756.5555555555555 ms (avg) for taking, 0 withdrawals
George: 18 locks taken, 629.5555555555555 ms (avg) for taking, 0 withdrawals
stop
```

Al ser el tiempo de Sleep mayor que Work, es bastante menos probable que se produzcan deadlocks, ya que todos los workers se encontraran en Open, entonces cuando queramos coger el lock, casi siempre estará libre. Es decir, los procesos siempre encontrarán el lock disponible para entrar. Cuando el tiempo de sleep es mayor que el tiempo de work, el sistema se vuelve menos competitivo y más equilibrado:

- Tiempo de sleep menor que work, sleep: 500 y work: 5000.

```
32> muty:stop().
John: 6 locks taken, 5338.333333333333 ms (avg) for taking, 1 withdrawals
George: 7 locks taken, 5063.714285714285 ms (avg) for taking, 1 withdrawals
Ringo: 5 locks taken, 6145.2 ms (avg) for taking, 3 withdrawals
Paul: 6 locks taken, 5985.333333333333 ms (avg) for taking, 1 withdrawals
stop
```

Dado que el tiempo de trabajo(5000 ms) es mucho mayor que el tiempo de sueño(500 ms), cada trabajador permanece en la sección crítica mucho tiempo, generando una alta contención. Esto provoca que haya altos tiempos de espera para adquirir el bloqueo, variación en la cantidad de bloqueos tomados y retiros como se puede ver en la imagen de arriba, Ringo tiene más retiros (3), lo que sugiere que intentaron obtener el bloqueo pero se retiraron después de un tiempo de espera, probablemente debido a un conflicto continuo para acceder al bloqueo. Esto ocurre porque lock1 no implementa ninguna política avanzada para evitar interbloqueos o mejorar la justicia en el acceso al bloqueo. Simplemente permite que un trabajador acceda a la sección crítica si el bloqueo está libre. Además podemos observar que el

tiempo de bloqueo es menor como mayor sea la diferencia entre el trabajo y el tiempo de espera, cuanto menos trabajo haya entonces más tiempo de espera.

- Tiempo de sleep igual que work, sleep: 1000 y work: 1000.

```
36> muty:stop().
John: 9 locks taken, 1092.4444444444443 ms (avg) for taking, 0 withdrawals
Ringo: 9 locks taken, 986.4444444444445 ms (avg) for taking, 0 withdrawals
Paul: 10 locks taken, 871.9 ms (avg) for taking, 0 withdrawals
George: 10 locks taken, 1007.2 ms (avg) for taking, 0 withdrawals
stop
```

Cuando el tiempo de sueño (sleep) es igual al tiempo de trabajo (work), los trabajadores pueden acceder al bloqueo de manera justa y equitativa, lo que resulta en un sistema más balanceado, es decir, no hay retiros, los tiempos de espera son moderados y similares entre los trabajadores y la cantidad de accesos al bloqueo está equilibrada.

ii) Split the muty module and make the needed adaptations to enable each worker-lock pair to run in different machines (that is, john and l1 should run in a machine, ringo and l2 in another, and so on).
Remember how names registered in remote nodes are referred and how Erlang runtime should be started to run distributed programs.

Unset

```
-module(muty_new).
-export([start/3, stop/0]).

% We use the name of the module (i.e. lock3) as a parameter to the start
procedure. We also provide the average time (in milliseconds) the worker is
going to sleep before trying to get the lock (Sleep) and work with the lock
taken (Work).

start(Lock, Sleep, Work) ->
    Node1 = 'node1@127.0.0.1',
    Node2 = 'node2@127.0.0.1',
    Node3 = 'node3@127.0.0.1',
    Node4 = 'node4@127.0.0.1',

    spawn(Node1, fun() ->
```

```
        register(l1, apply(Lock, start, [1])),
        register(w1, worker:start("PeppaPig", l1, Sleep, Work)) end),

    spawn(Node2, fun() ->
        register(l2, apply(Lock, start, [2])),
        register(w2, worker:start("Ringo", l2, Sleep, Work)) end),

    spawn(Node3, fun() ->
        register(l3, apply(Lock, start, [3])),
        register(w3, worker:start("Paul", l3, Sleep, Work)) end),

    spawn(Node4, fun() ->
        register(l4, apply(Lock, start, [4])),
        register(w4, worker:start("George", l4, Sleep, Work)) end),
    timer:sleep(1000),
    {l1, Node1} ! {peers, [{l2,Node2}, {l3,Node3}, {l4,Node4}]},
    {l2, Node2} ! {peers, [{l1,Node1}, {l3,Node3}, {l4,Node4}]},
    {l3, Node3} ! {peers, [{l2,Node2}, {l1,Node1}, {l4,Node4}]},
    {l4, Node4} ! {peers, [{l2,Node2}, {l3,Node3}, {l1,Node1}]},

    ok.

stop() ->
    Node1 = 'node1@127.0.0.1',
    Node2 = 'node2@127.0.0.1',
    Node3 = 'node3@127.0.0.1',
    Node4 = 'node4@127.0.0.1',
    {w1, Node1} ! stop,
    {w2, Node2} ! stop,
    {w3, Node3} ! stop,
    {w4, Node4} ! stop.
```

Para ejecutar cada par worker-lock en diferentes nodos/máquinas hemos realizado lo siguiente:

1. Definición de nodos remotos. Hemos definido cuatro nodos diferentes, cada uno con un nombre (node1, node2, node3, node4) y una dirección IP local (127.0.0.1).
2. Creación de procesos remotos. Se ha usado la función spawn(Node, Fun) para iniciar procesos en diferentes nodos. Luego se han registrado los nombres de los procesos con register.
3. Comunicación entre nodos. Después de un pequeño retraso, se envían mensajes a cada lock registrado en los nodos, informándoles sobre sus compañeros (peers), que configura la comunicación entre los procesos de bloqueo (locks) en diferentes nodos.

4. Finalización de procesos. Para terminar se usó la función stop() que envía mensajes stop a los procesos workers en los nodos remotos, pidiéndoles que se detengan.

Open Questions

What is the behavior of the lock when you increase the risk of a conflict?

Como ya hemos observado anteriormente, a medida que aumenta la probabilidad de interferencias o conflictos entre diferentes procesos o nodos, es crucial implementar estrategias de control de la concurrencia para paliar estas eventualidades. Esto puede generar competencia por la sección crítica o recurso compartido con varios componentes del sistema intentado modificar o acceder a la misma vez el cierre. Esto puede dar como resultado tiempos de espera más prolongados así como tasas de retiros más altas, acceso desigual o injusto a los bloqueos y una menor eficiencia general. Todo esto a causa de que en lock1 no se dispone de controles de concurrencia o implementación alguna para el manejo de prioridades o como los relojes Lamport, cosa que podría ayudar a administrar, coordinar el acceso a escenarios de alta contención.

En conclusión, lock1 funciona razonablemente bien en condiciones de baja contención, pero a medida que aumenta el riesgo de conflicto, sus limitaciones se vuelven evidentes, lo que lleva a una distribución de bloqueos ineficiente e injusta.

2. Resolving deadlock

Experiments

Repeat the previous tests to compare the behavior of this lock with respect to the previous one.

- Tiempo de sleep mayor que work, sleep: 2000 y work: 1000.

```
7> muty:stop().
John: 17 locks taken, 305.8235294117647 ms (avg) for taking, 0 withdrawals
Ringo: 15 locks taken, 505.26666666666665 ms (avg) for taking, 0 withdrawals
Paul: 12 locks taken, 971.9166666666666 ms (avg) for taking, 0 withdrawals
George: 9 locks taken, 1530.7777777777778 ms (avg) for taking, 0 withdrawals
stop
```

Podemos observar que hay una distribución más equitativa de los locks que ha tomado cada uno. Esto se debe a que, como el tiempo de sleep es grande y el de work pequeño, cuando alguien toma el recurso y termina, se queda esperando en sleep mucho tiempo, lo que da tiempo a que los demás tomen el recurso y hagan el trabajo.

- Tiempo de sleep menor que work, sleep: 500 y work: 5000.

```
9> muty:stop().
John: 11 locks taken, 2460.6363636363635 ms (avg) for taking, 0 withdrawals
Ringo: 11 locks taken, 2596.0 ms (avg) for taking, 1 withdrawals
Paul: 2 locks taken, 2492.5 ms (avg) for taking, 6 withdrawals
George: 0 locks taken, 0 ms (avg) for taking, 7 withdrawals
stop
```

Vemos en la captura que John y Ringo han tomado muchos más locks que los otros dos. Esto se debe a que son los que tienen el ID más pequeño y tienen prioridad sobre los demás. Además, al tener un sleep pequeño y un work grande, normalmente pasarán menos tiempo esperando para volver a solicitar el recurso que trabajando con él, lo que explica que solo dos workers hayan sido los que tomaron el recurso de la región crítica.

- Tiempo de sleep igual que work, sleep: 1000 y work: 1000

```
13> muty:stop().
John: 39 locks taken, 318.7435897435897 ms (avg) for taking, 0 withdrawals
Ringo: 38 locks taken, 514.2105263157895 ms (avg) for taking, 0 withdrawals
Paul: 19 locks taken, 1815.0 ms (avg) for taking, 0 withdrawals
George: 16 locks taken, 2108.75 ms (avg) for taking, 0 withdrawals
stop
```

En este caso, vemos que paso algo similar al caso anterior, la mayor parte de los locks lo han tomado John y Ringo, pero a diferencia del anterior es que al tener el tiempo de sleep y work iguales los otros dos también han adquirido locks pero dando prioridad a John y Ringo ya que son los que tienen las IDs más pequeñas.
Por último, podemos notar que el tiempo de bloqueo es más corto a medida que aumenta la diferencia entre el tiempo de espera y el de trabajo (más espera y menos trabajo), lo cual tiene sentido ya que si pasamos mucho tiempo esperando y poco trabajando, es más probable que el proceso que posee el lock que deseamos ya haya finalizado su tarea y esté libre. Esto ya lo veníamos observando en el caso anterior.
En este caso, como se priorizan los procesos en función de su ID (un ID más bajo tiene mayor prioridad), observamos que los procesos con ID más bajo tardan menos en acceder a la sección crítica en todos los escenarios (ya que tienen más preferencia). En cambio, los procesos con un ID más alto tardan mucho más. Incluso podría generarse un problema de starving si la sección crítica permanece constantemente ocupada por procesos con IDs pequeños, lo que podría impedir que los procesos con IDs mayores puedan ejecutarse o lo hagan después de un tiempo excesivo.

Open Questions.

i) Justify how your code guarantees that only one process is in the critical section at any time.

En esta nueva implementación lock2, se programa un sistema de bloqueo distribuido que garantiza el acceso exclusivo de un solo proceso a la vez en la sección crítica en cualquier momento usando un sistema de prioridad basado en identificadores únicos para cada instancia de bloqueo que se genere (MyId). Cuando una instancia de bloqueo recibe una solicitud para acceder a la sección crítica (`{take, Master, Ref}`), envía solicitudes (`request`) a las demás instancias de bloqueo, incluyendo su propio identificador (`MyId`) en la solicitud. Las respuestas (`ok`) se reciben solo de las instancias que no están intentando acceder a la sección crítica o que tienen una prioridad menor (un `MyId` más alto) y, por lo tanto, ceden el paso. Si otra instancia de bloqueo tiene una prioridad más alta (un `MyId` menor), esta responderá con `ok`, y la instancia con la prioridad menor procederá a la sección crítica primero. De este modo, el código garantiza que, en cualquier momento, solo la instancia de bloqueo con la mayor prioridad (el menor `MyId`) entre las que están esperando obtenga el acceso a la sección crítica, evitando que múltiples procesos entren al mismo tiempo. Cabe destacar que la función wait es la encargada de gestionar las solicitudes dando prioridad con IDs más bajos y manteniéndolos en una cola de espera. La prioridad se determina a partir de identificadores únicos asignados a cada proceso y se gestiona la cola de espera en base a estos identificadores. La liberación del bloqueo se realiza mediante un mensaje de liberación permitiendo así a los procesos a la espera de continuar. Esta nueva implementación asegura que la entrada crítica se haga de forma ordenada según la prioridad asignada a los procesos mediante los IDs (tiene más prioridad aquél con un ID menor).

ii) What is the main drawback of lock2 implementation?

El principal inconveniente de la nueva implementación lock2 es la injusticia en el acceso debido al sistema de prioridad fija utilizando MyId, una instancia de bloqueo con un identificador bajo como mayor prioridad , puede obtener acceso más drecuentemente en situaciones de contención. Esto puede ocasionar a una situación en la que aquellas instancias con identificadores más altos con prioridad más baja esperen más tiempo y que tengan menos oportunidades para acceder a la sección crítica. Otro inconveniente a destacar es que en entornos con alta contención y muchas instancias de bloqueo, las instancias con prioridad baja pueden experimentar hambre, es decir, que se queden esperando indefinidamente. Otro inconveniente es la posibilidad de bloqueo mutuo o deadlock que ocurre cuando dos o más procesos se quedan bloqueados, uno esperando al otro para que libere el bloqueo. La función wait puede ocasionar a una espera circular entre procesos prpovocando bloqueo mutuo y no ser suficiente el criterio de priorización de solicitudes para evitar este suceso. Esto se podría prevenir usando un mecanismo de bloqueo más sofisticado como el uso de un orden global de bloqueo o técnicas de detección y recuperación de bloqueo mutuo.

3. Lamport time

Experiments

Repeat the previous tests to compare this version with the former ones.

- Tiempo de sleep mayor que work, sleep: 2000 y work: 1000.

```
6> muty:stop().
John: 21 locks taken, 620.4285714285714 ms (avg) for taking, 0 withdrawals
Ringo: 21 locks taken, 636.7142857142857 ms (avg) for taking, 0 withdrawals
Paul: 19 locks taken, 776.2105263157895 ms (avg) for taking, 0 withdrawals
George: 20 locks taken, 690.15 ms (avg) for taking, 0 withdrawals
stop
```

En este caso en concreto, también podemos observar esa distribución más equitativa de los locks que ha tomado cada uno que se mencionaba para el caso anterior. A parte de que los resultados son muy similares a los de lock1 y 2, ya que el tiempo de espera (sleep) es grande, hay pocas probabilidades de que todos los procesos pidan acceso al mismo tiempo.

- Tiempo de sleep menor que work, sleep: 500 y work: 5000.

```
17> muty:stop().
John: 5 locks taken, 4759.6 ms (avg) for taking, 1 withdrawals
Ringo: 4 locks taken, 4562.25 ms (avg) for taking, 2 withdrawals
Paul: 3 locks taken, 7226.666666666667 ms (avg) for taking, 1 withdrawals
George: 3 locks taken, 3881.3333333333335 ms (avg) for taking, 2 withdrawals
stop
```

En este caso observamos que el número de deadlocks es menor que en lock2 y lock1, ya que actualmente el acceso se reparte de manera más equitativa, evitando que un proceso sufra starvation y se rinda.

- Tiempo de sleep igual que work, sleep: 1000 y work: 1000

```
19> muty:stop().
John: 15 locks taken, 1046.7333333333333 ms (avg) for taking, 0 withdrawals
Ringo: 14 locks taken, 964.3571428571429 ms (avg) for taking, 0 withdrawals
Paul: 13 locks taken, 922.0769230769231 ms (avg) for taking, 0 withdrawals
George: 16 locks taken, 850.4375 ms (avg) for taking, 0 withdrawals
stop
```

Podemos ver que hay un reparto bastante equitativo de locks y sin ningún deadlock al tener el tiempo de sleep y work iguales.

Y para finalizar, al igual que en el caso de lock2, el tiempo de bloqueo se observa como más corto cuando la diferencia entre el tiempo de espera y el

de trabajo es mayor, es decir, cuando hay más tiempo de espera y menos de trabajo. Esto es lógico, ya que si se espera más y se trabaja menos, es más probable que el proceso que posee el lock que necesitamos ya haya terminado su tarea y esté disponible.

En comparación con el caso anterior, aquí el acceso a la sección crítica se gestiona según el reloj de Lamport de cada proceso. Es decir, la prioridad se determina por el orden en que los procesos solicitan el acceso, no por su ID. Este enfoque hace que el sistema sea mucho más justo y ayuda a evitar la starvation, que podría ocurrir en el caso anterior. Como resultado, el tiempo de espera entre los nodos es bastante equilibrado (no hay un proceso con prioridad permanente, sino que la prioridad es temporal). Este comportamiento mejora la similitud con el primer caso, aunque el tiempo total sigue siendo prácticamente el mismo.

Open Questions.

Note that the workers are not involved in the Lamport clock. According to this, would it be possible that a worker is given access to a critical section prior to another worker that issued a request to its lock instance before (assuming real-time order)?

Sí, es posible que a un trabajador se le conceda acceso a la sección crítica antes que a otro trabajador que emitió una solicitud antes en orden de tiempo real. Esto se debe a la naturaleza de los relojes Lamport, que se basan en un orden lógico en lugar de en tiempo real.

En lock3, cada instancia de bloqueo utiliza un reloj Lamport para ordenar las solicitudes y garantizar el acceso a la sección crítica. Sin embargo, este orden es puramente lógico, es decir, no tiene en cuenta la secuencia real de solicitudes en tiempo real. Cabe recalcar que el reloj Lamport solo garantiza que los eventos se ordenen de una manera que sea consistente en todo el sistema distribuido, pero no refleja la secuencia exacta de eventos en tiempo real. Por ejemplo, una solicitud con una marca de tiempo Lamport más alta (emitida más tarde de manera lógica) puede haberse emitido un poco antes en tiempo real que otra solicitud con una marca de tiempo más baja.

Otro motivo de que esto suceda puede ser a la no participación de los trabajadores dado que los trabajadores en sí no forman parte de la sincronización del reloj de Lamport, dependen de sus respectivas instancias de bloqueo para administrar el acceso a la sección crítica.

También puede deberse a retrasos en la actualización del reloj ya que en sistemas distribuidos nada es perfecto y puede haber retrasos de la red o diferencias en la velocidad con la que se procesan los mensajes, cosa que puede generar situaciones en las que una solicitud emitida tarde tenga una marca de tiempo de Lamport más baja, cosa que le permita acceso antes a la situación crítica.