TP3 - Algoritmos en sistemas distribuidos

Sistemas Operativos - Primer cuatrimestre de 2015

Límite de entrega: Miércoles 24 de Junio, 23:59 hs.

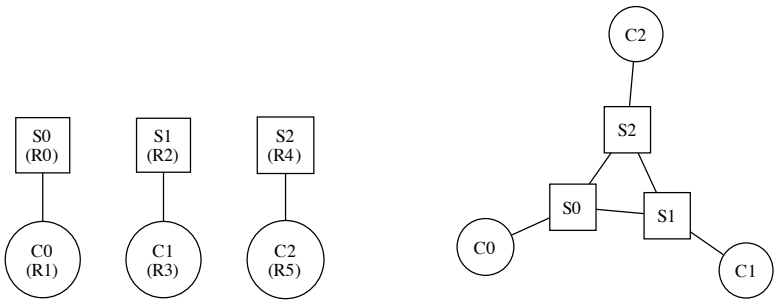
Introducción

Resolver correcta y eficientemente la exclusión mutua en ausencia de memoria compartida (y sin coordinación centralizada) implica nuevos desafíos. En 1981, Ricart y Agrawala publicaron su artículo *An optimal algorithm for mutual exclusion in computer networks* [1] que incluye una solución –óptima en la cantidad de mensajes, entre otras propiedades interesantes– para el problema de la exclusión mutua en tales contextos. Este trabajo práctico consiste en implementar una versión del algoritmo presentado en las secciones 1 y 2 de [1] usando *message-passing*.

Clientes y servidores

Llamaremos *clientes* a los $n \geq 1$ procesos que se ejecutan en $m \geq 1$ máquinas para llevar a cabo el cómputo distribuido propiamente dicho (algún cómputo, cuyos detalles particulares podrían variar según el caso). Sabemos, y esto es lo importante a nuestros efectos, que cada cliente necesitará ejecutar regularmente su sección crítica bajo garantía de exclusividad global.

Llamaremos *servidores* a otros n procesos adicionales[1] cuya tarea consistirá en gestionar acceso exclusivo a pedido de sus respectivos clientes. El i-ésimo servidor, con *rank* $2i$, estará al servicio del i-ésimo cliente, con rank $2i+1$ (ver ej. fig. 1a). Un cliente sólo intercambia mensajes con su servidor y viceversa, pero los servidores deben poder interactuar entre sí (ej. fig. 1b).



(a) Ranks y nomenclatura (b) Canales de comunicación válidos

Figura 1: Procesos, clientes y servidores (ambos ejemplos sup. `mpiexec -np 6`)

[1] Cada servidor provee la funcionalidad que en [1] se explica, por claridad, como tres procesos concurrentes.

Exclusión mutua y *streams*

Como abstracción de las secciones críticas de los clientes, imaginemos que cada instrucción crítica consiste en escribir un byte a `stderr`. Por ejemplo, por cada instrucción atómica que ejecutaría, el i-ésimo cliente podría escribir la i-ésima letra minúscula a dicho *stream*.

Así, una ejecución de la sección crítica completa del primer cliente podría resultar en el envío a `stderr` de la secuencia "`aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa`" (ver ej. fig. 2a).

Esto convierte al stream `stderr` en un recurso crucial cuyo uso debe arbitrarse con cuidado. Si más de un cliente logra escribir en `stderr` a la vez, las consecuencias suelen ser visiblemente catastróficas (ver ej. fig. 2b).

El otro stream, `stdout`, puede usarse a discreción para mensajes informativos o de debug[2] sin restricciones sobre su solapamiento, aunque tampoco garantías sobre su orden relativo.

```
$ mpiexec -np 8 ./tp3_correcto >/dev/null
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
cccccccccccccccccccccccccccccccccccccccc
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
dddddddddddddddddddddddddddddddddddddddd
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
cccccccccccccccccccccccccccccccccccccccc
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
dddddddddddddddddddddddddddddddddddddddd
cccccccccccccccccccccccccccccccccccccccc
...
```

(a) Implementación correcta

```
$ mpiexec -np 8 ./tp3_incorrecto  >/dev/null
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
ccaaaaccaaacaaacaaacaaacaaacaaacaaacaaacaaacaaacaaacaaa
ccccccccccccdcdcdccdcdcdccdcdcdccdc
ddddddddddddddddddddddddddddddddddd
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
ccccccccccccccccccccccccccbcbcbbcbcbbcbcbbcbcbbcbbcbc
bbbbbbbbbbbbbbbbbbbbbbbb
ddddddddddddddddddddddddddddddddddddddddd
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbcbcbb
...
```

(b) Implementación con problemas

Figura 2: Ejemplos de corridas con buena y mala sincronización del acceso a `stderr`.

Requerimientos

El cliente genérico y el protocolo cliente-servidor provistos no deben modificarse.

El servidor provisto es un ejemplo: incluye apenas lo mínimo necesario para responder algo, correcto o no, a cada mensaje del cliente. Su uso como base para la implementación es opcional.

El objetivo principal del trabajo es implementar correctamente el servidor, agregando los tags y *handlers* necesarios para la comunicación servidor-servidor, aplicando las ideas de [1] y logrando un correcto arbitraje de la exclusión mutua en el sistema distribuido.

Cumplido lo antedicho, como objetivo secundario también es deseable revisar el criterio de parada del sistema. Un protocolo de terminación correcto debería garantizar que cada rank llegue a su `MPI_Finalize()` sin abandonar mensajes en tránsito ni perder otros recursos.

La solución no debe violar la exclusión mutua ni exhibir deadlock, livelock o inanición para ninguna combinación de parámetros y `-np` razonable. Tampoco debe introducir componentes centralizados ni suponer la existencia de relojes globalmente sincronizados.

Otros problemas menores no serán causal de reentrega si están documentados en un breve informe (listar limitaciones conocidas y conjeturar posibles causas). Si la solución cumple todos los requerimientos y el código es muy claramente legible, entregar un informe no es necesario.

Para el desarrollo puede usarse MPICH2 [4], OpenMPI [5] o cualquier otra implementación del standard MPI-2 [2] en versión no-obsoleta. Como parte del *testing* final es deseable probar

[2] El lector atento notará que este uso de `stdout` y `stderr` es el inverso del esperable. ¿Por qué?

al menos dos distintas: una solución correcta debería atenerse al standard vigente sin depender de aspectos específicos de ninguna implementación o plataforma en particular.

Parametrización

Además de poder variarse la cantidad total de ranks participantes (el `-np` de `mpiexec`), el sistema ofrece algunos parámetros cuyo barrido permite ejercitar con relativa facilidad muchas posibles trazas de ejecución. Los siguientes pueden controlarse desde la línea de comando:

1. El caracter que imprime cada cliente para identificarse.
   Por defecto, el primer cliente imprime `a`, el segundo `b`, el tercero `c`, etc.
2. La cantidad de iteraciones que realiza en total cada cliente.
   Por defecto el primer cliente realiza 2, el segundo 4, el tercero 6, etc.
3. El tiempo de cómputo previo a solicitar acceso exclusivo de cada cliente.
   Por defecto 200 ms/iter, 400 ms/iter, 600 ms/iter, 800 ms/iter, etc.
4. El tiempo de cómputo dentro de la sección crítica de cada cliente.
   Por defecto 100 ms/iter, 200 ms/iter, 300 ms/iter, 400 ms/iter, etc.

La línea de comando puede incluir cero o más grupos de cuatro argumentos, por ej.:

```
mpiexec -np  2 ./tp3
mpiexec -np  2 ./tp3  a 5 1000 1000
mpiexec -np  4 ./tp3  a 5 0 100    b 3 1000 100
mpiexec -np  6 ./tp3  @ 4 0 1000   b 9 250 0   c 9 500 0
```

El primer grupo, de haberlo, se aplica al primer cliente, el segundo grupo al segundo cliente y así sucesivamente. De haber más grupos de cuatro argumentos que clientes, los grupos sobrantes se ignoran. De haber más clientes que grupos, los clientes subespecificados reciben los mismos parámetros por defecto que les corresponderían en una invocación sin argumentos.

Observar que estos parámetros son determinísticos (nada de esto es pseudoaleatorio) pero aspectos como el *scheduling*, la carga del sistema, la implementación de MPI que se utilice y la arquitectura de hardware, entre otros, introducen su cuota de ruido. Ciertas combinaciones de parámetros son útiles para compensar ese ruido; otras, para amplificarlo deliberadamente.

Referencias

[1] *An optimal algorithm for mutual exclusion in computer networks.* Glenn Ricart y Ashok K. Agrawala, publicado en Communications of the ACM, 24(1), enero 1981, pp. 9–17.
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.62.7872&rep=rep1&type=pdf

[2] *MPI: A Message Passing Interface Standard.*
http://www.mpi-forum.org/docs/docs.html

[3] MPI Tutorial @ LNLL (muy recomendable)
https://www.llnl.gov/computing/tutorials/mpi/

[4] MPICH2, http://www.mcs.anl.gov/research/projects/mpich2/

[5] OpenMPI, http://www.open-mpi.org/