TP3 - Algoritmos en sistemas distribuidos

Sistemas Operativos - Primer cuatrimestre de 2011

Límite de entrega: lunes 27 de junio, 23:59 hs.

Introducción

Como hemos visto en clase, resolver correcta y eficientemente la exclusión mutua en ausencia de memoria compartida (y sin un coordinador centralizado) nos enfrenta con nuevos desafíos.

En 1981, Ricart y Agrawala publicaron su artículo *An optimal algorithm for mutual exclusion in computer networks* [1], incluyendo una solución –óptima en la cantidad de mensajes, entre otras propiedades interesantes– para el problema de la exclusión mutua en sistemas distribuidos.

El objetivo de este trabajo práctico es implementar una versión del algoritmo que se presenta en las secciones 1 y 2 de [1] utilizando *message-passing* conforme al standard MPI vigente [2].

Procesos, clientes y servidores

Los procesos se dividen en dos clases: los *ranks* impares son clientes, y los pares, servidores. El i-ésimo servidor, con rank $2i$, está al servicio del i-ésimo cliente, con rank $2i+1$ (ver fig. 1a).

Cada cliente representa un programa arbitrario que corre en algún nodo y forma parte de un sistema distribuido. Los detalles de lo que computa no son relevantes aquí; lo que nos interesa es que periódicamente debe ejecutar una sección crítica, bajo garantía de exclusividad global.

Un cliente sólo intercambia mensajes con su servidor asignado. Un servidor se comunica con un único cliente, pero también puede –y debe– comunicarse con otros servidores (ver fig. 1b).

Observar que un proceso servidor implementa, en esta arquitectura, toda la funcionalidad que en el artículo [1] se presenta dividida en tres servicios concurrentes.

Exclusión mutua y `stderr`

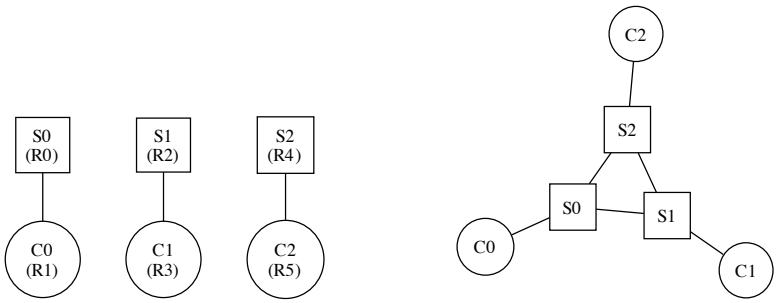
A los efectos de este trabajo, el recurso compartido crucial será uno de los *streams* de salida.

Como abstracción de la serie de instrucciones que conforman la sección crítica de cada cliente, imaginemos que las mismas consisten en escribir un caracter a `stderr`. Por cada instrucción atómica que ejecutaría, el i-ésimo cliente escribe la i-ésima letra minúscula del alfabeto.

Así, por ejemplo, una ejecución de la sección crítica completa del primer cliente podría resultar en el envío a `stderr` de la secuencia "`aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa`".

Cuando más de un cliente logra escribir a `stderr` a la vez, las consecuencias suelen resultar visiblemente catastróficas (ver figs. 2a y 2b). Debe garantizarse que esto nunca suceda.

Nota: el otro stream, `stdout`, puede usarse a discreción para imprimir mensajes de debug o cualquier otra finalidad conveniente, en cualquier orden, con o sin solapamiento.



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

Figura 1: Procesos, clientes y servidores (ejemplos suponiendo `-np 6`)

```
$ mpiexec -np 10 ./tp3 2 100 50 >/dev/null
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
ccccccccccccccccccccccccccccccccccccccccc
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
ddddddddddddddddddddddddddddddddddddddddd
eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee
ccccccccccccccccccccccccccccccccccccccccc
ddddddddddddddddddddddddddddddddddddddddd
eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee
```

(a) Implementación correcta

```
$ mpiexec -np 10 ./tp3_incorrecto 2 100 50 >/dev/null
aaaaaaaabababadabeaabdeabadaeaadaaeadaabbcadea
caaacdebcaabcaabaacbdeabcaabacaba
bcbcddeebbbccdaabcedbcaabbdaceabcaabeaacba
abcaaebacabadceaaaaaaaaadddabbbbccccceeabab cabeabcddaabbcceeaaddb
cceaaadeaaaccdedeccdeccedcdecdccdeedededbcdecdbcde
bcbcdebbccddebbecdbcdbcbbeccebcbecbbcebce
bcbcbcbcbbbccebbbcecebbbccecbebcbcbe
bcbcebcbbcceebddccdedecdcdecdc
eddededededdedededededededdedededededededdedd
ededededededdeded
```

(b) Implementación con problemas

Figura 2: Ejemplos de corridas con buena y mala sincronización

Requerimientos

La cátedra provee una implementación ya completa del cliente, el protocolo de comunicación cliente-servidor y un "esqueleto" con los aspectos elementales del servidor. Este último incluye lo mínimo necesario para responder los mensajes del cliente, pero no el protocolo de comunicación *entre* servidores, con lo que las respuestas que se dan al cliente no siempre son correctas.

Se pide completar y corregir la implementación del servidor provista, agregando los mensajes y *handlers* necesarios para la sincronización servidor-servidor, y logrando un correcto arbitraje de la exclusión mutua en el sistema distribuido, de acuerdo con el algoritmo de [1].

La solución no debe violar la exclusión mutua ni exhibir deadlock, livelock o inanición para ninguna combinación de parámetros y `-np` razonable. Ante fallas para algún caso de borde, debe entregarse un breve informe comentando las limitaciones conocidas y conjeturando las causas. Si todo funciona y el código está *claramente* documentado, la entrega de un informe es opcional.

Para el desarrollo puede utilizarse MPICH2 [4], OpenMPI [5] o cualquier otro paquete en versión no-obsoleta. De hecho, una solución correcta debería atenerse al standard vigente [2] sin depender de características específicas de ninguna implementación particular de MPI.

Como siempre, la solución debe remitirse a `sisopdc@gmail.com` bajo la forma de un único archivo comprimido en formato `.tar.gz` o `.zip`, incluyendo un `Makefile` y ningún binario.

Parametrización del cliente

Además de variar la cantidad de procesos del sistema (mediante el `-np` de `mpiexec`), el cliente ofrece algunos parámetros cuyo barrido permite ejercitar con relativa facilidad muchas de las posibles trazas de ejecución. Los tres siguientes pueden controlarse desde la línea de comando:

- La cantidad de iteraciones que realiza en total cada cliente.
- Un coeficiente de demora para el cómputo previo a cada solicitud de acceso exclusivo (modela el tiempo que los clientes invierten *fuera* de la sección crítica en cada iteración).
- Un coeficiente de demora para el tiempo que invierten *dentro* de la sección crítica.

El primero afecta a todos los clientes por igual. Los otros dos tienen un impacto lineal en función del número de cliente: aumentando su valor pueden observarse pequeñas perturbaciones en los clientes con ranks más bajos y grandes cambios en los clientes de ranks superiores.

Notar que, si bien estos parámetros son determinísticos (nada de esto es pseudoaleatorio) aspectos como el *scheduling*, la carga del sistema, la implementación de MPI que se utilice y la arquitectura de hardware, entre muchos otros, introducen su cuota de ruido. Hay combinaciones de parámetros que permiten neutralizar ese ruido; otras permiten amplificarlo deliberadamente.

Sintaxis de invocación

`mpiexec -np N <ejecutable>` [`cant_iters` [`coef_delay_comp` [`coef_delay_crit`]]]

donde `N` es la cantidad de procesos, que debe ser par, y los corchetes significan "opcional".

Algunos ejemplos:

`mpiexec -np 4 ./tp3`	(usar todos los valores por defecto)
`mpiexec -np 8 ./tp3 10`	(10 iteraciones; ambos retardos según los valores por defecto)
`mpiexec -np 8 ./tp3 4 50`	(4 iters.; delay de cómputo 50μs, 100μs, 150μs, ...; delay crít. por defecto)
`mpiexec -np 8 ./tp3 4 0 100`	(4 iters.; cómputo sin delay; retardo por instr. crít. 100μs, 200μs, 300μs...)

Tener presente la utilidad de redireccionar uno o ambos streams:

```
mpiexec -np 4 ./tp3 >/dev/null
mpiexec -np 4 ./tp3 2>/dev/null
mpiexec -np 4 ./tp3 2>/dev/null | tee stdout.log   ...etcétera.
```

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] http://www.mcs.anl.gov/research/projects/mpich2/

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