Práctica 2
Repositorio datos asignatura

Eduardo Gimeno Soriano 721615

Sergio Álvarez Peiró 740241

5 de noviembre de 2019

Introducción

El sistema desarrollado es un repositorio de datos distribuido utilizando el algoritmo de exclusión mútua de Ricart-Agrawala y los relojes lógicos de Lamport.

Sección Principal

Resumen

En el sistema se identifican dos roles, lector y escritor, existe un único programa en el cual se diferencian por un argumento introducido. Para reducir el tamaño del fichero se ha creado una biblioteca con funciones auxiliares.

Para la implementación del algoritmo de Ricart-Agrawala, se ha traducido el algoritmo modificado del enunciado de la forma más exacta posible.

Arquitectura del sistema

Cada proceso del sistema está formado por tres subprocesos. Un proceso principal que ejecuta el protocolo de acceso a la sección crítica, así como el acceso a la misma, un proceso encargado de recibir request (request process) y un proceso encargado de recibir permission (permission process). Entre estos tres subprocesos también existe comunicación.

Proceso principal

En primer lugar se ejecuta la operación begin_protocol, la cual recibe como parámetros el identificador del proceso dentro del sistema y el número total de procesos del sistema. Con esta operación se obtienen los pids del resto de request process y permission process del sistema, así como enviar los propios al resto. Enviar los propios al resto implica haber creado dichos procesos.

```
IO.puts("SPAWN REQUEST Y PERMISSION")
# Crear subprocesos encargados de recibir request y permission
rr_pid = spawn(Common, :request, [proc_id, lrd, clock, cs_state, [], :nil, [], true])
rp_pid = spawn(Common, :permission, [waiting_from, waiting_from, self])

# Enviar al resto de procesos princiaples los pids de los subprocesos encargados de recibir request y permission y recibir sus análogos
filtered_list = Enum.filter(Node.list, fn(x) -> Atom.to_string(x) =~ "alumno" || Atom.to_string(x) =~ "profesor" end)
num_msg = length(filtered_list)
Common.enviar(filtered_list, rr_pid, proc_id)
rr_list = Common.recibir(num_msg, [])

filtered_list = Enum.filter(Node.list, fn(x) -> Atom.to_string(x) =~ "alumno" || Atom.to_string(x) =~ "profesor" end)
num_msg = length(filtered_list)
Common.enviar(filtered_list, rp_pid, proc_id)
rp_list = Common.recibir(num_msg,[])
send(rr_pid, {:update, rp_list})
```

Imagen 1: Creación de subprocesos e intercambio de PIDs en begin_protocol

Una vez realizado esto, se duerme al proceso durante un tiempo pseudoaleatorio y se invoca a la función principal protocol, la cual recibe clock, el cual ha sido necesario iniciarlizar antes para que el request process lo conozca, el identificador del proceso en el sistema, los pids del request process y permission process y dos listas que contienen los pids de los request process y permission process del sistema, los cuales tienen asociados en la lista el identificador dentro del sistema.

En esta operación se realizan begin_op(), end_op() y el acceso a la sección crítica. En el caso del lector la operación que ejecutará sobre el repositorio se generá aleatoriamente, en el escritor además de esto, la desripción se generá también aleatoriamente haciendo uso de un módulo externo denominado Randomizer [1]. Una vez terminado el protocolo se duerme al proceso durante un tiempo pseudoaleatorio y se vuelve a invocar a esta función, de esta forma cada lector y escritor operan continuadamente.

```
IO.puts("INICIO DEL PROTOCOLO")
cs_state = :trying
lrd = clock + 1
op_type = Common.generar_operacion_lector
# Actualizar datos en subproceso encargado de recibir request
send(rr_pid, {:update, cs_state, lrd, op_type})
# Enviar peticiones de acceso al resto
Common.enviar_request(lrd, proc_id, op_type, rr_list)
# Recibir luz verde del subproceso encargado de recibir permission
receive do
    {:ok} -> cs_state = :in
             send(rr_pid, {:update, cs_state})
end
```

Imagen 2: Traducción de begin_op() en protocol

```
IO.puts("SECCION CRITICA")
# Pedir al repositorio los datos y mostrarlos por pantalla
repositorio = hd(Enum.filter(Node.list, fn(x) -> Atom.to_string(x) =~ "repositorio" end))
send({:pprincipal, repositorio}, {op_type, self})
op_type_s = Atom.to_string(op_type)
receive do
    {:reply, content} -> IO.puts(op_type_s)
                         IO.puts(content)
end
```

Imagen 3: Acceso a la sección crítica en protocol

```
cs_state = :out
# Actualizar datos en subproceso encargado de recibir request
send(rr_pid, {:update, cs_state})
# Enviar permission al resto de procesos bloqueados
send(rr_pid, {:need_perm_delayed, self})
receive do
    {:perm_delayed, perm_delayed} -> Common.enviar_permission(proc_id, rp_list, perm_delayed)
end
send(rr_pid, {:reset_perm_delayed})
send(rr_pid, {:need_clock, self})
clock_n = 0
receive do
    {:clock, new_clock} -> clock_n = new_clock
end

IO.puts("FUERA SECCION CRITICA")
```

Imagen 4: Traducción de end_op() en protocol

Request process

Este proceso ejecuta la operación request, la cual se encarga de recibir request del resto del sistema y se comunica con el proceso principal para intercambiar variables

utilizadas por ambos. Implementa when REQUEST is recieved del algoritmo de Ricart-Agrawala.

Permission process

Este prceso ejecuta la operación permission, la cual se encarga de recibir permission del resto del sistema y una vez a recibido todas le comunica al proceso principal que puede acceder a la sección crítica. Implementa when PERMISSION is recieved del algoritmo de Ricart-Agrawala.

Vistas del sistema

Vistas del sistema disponibles en:

https://drive.google.com/open?id=1tsEdRmKIkWb9kNsDGaEnP94ZylC_n7qE

Validación Experimental

Para comprobar el funcionamiento del sistema se han realizado en primer lugar pruebas básicas, de estas destacar la siguiente, existe un lector y un escritor, el segundo con menor identificador, ambos quieren entrar en la sección crítica a la vez, por tanto, observar que por tener menor identificador, el escritor entra antes.
En cuanto a pruebas de mayor tamaño se ha probado con número diferente de cada tipo de actor en el sistema, es decir, dos lectores y un escritor y viceversa. Una vez realizadas se ha probado con tres lectores y tres ecritores y por último con cinco lectores y cinco escritores.

Para comprobar que ocurría en el sistema durante la ejecución se han introducido mensajes de debug en determinados puntos del código.

Conclusiones

El algoritmo de Ricart-Agrawala es eficaz a la hora de garantizar la exclusión mútua si no se utiliza un proceso coordinador. El problema más representativo de este algoritmo es que si cualquier nodo del sistema falla se produce un bloqueo del sistema.

Bibliografía

[1] Randomizer utilizado para generar cadenas aleatorias para los escritores https://gist.github.com/ahmadshah/8d978bbc550128cca12dd917a09ddfb7

[2] Material de la asignatura de Sistemas Distribuidos del grado de Ingeniería Informática de UNIZAR.

[3] Documentación del lenguaje Elixir https://elixir-lang.org/docs.html