DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

Sistemas Distribuidos

Sistemas Operativos

Primer Cuatrimestre de 2017

Integrante	LU	Correo electrónico
Alem Santiago	650/14	santialem.trev@gmail.com
Alliani Federico	183/15	fedealliani@gmail.com
Raposeiras Lucas	034/15	lucas.raposeiras@outlook.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Resumen

El objetivo de este Trabajo Práctico consiste en comprender el uso y funcionamiento de los Sistemas Distribuidos utilizando *MPI*.

En este informe explicaremos cómo realizamos cada ejercicio del Trabajo Práctico, así como también algunas conclusiones obtenidas.

Índice

1. Conceptos básicos **3**

2. Función `void load(list<string>params)` **3**

3. Función `void addAndInc(string key)` **3**

4. Función `void member(string key)` **4**

5. Función `void maximum()` **4**

6. Función `void quit()` **4**

1. Conceptos básicos

La función principal del *Nodo* recibe los mensajes entrantes de forma bloqueante mediante `MPI_Recv`. Cuando se recibe un mensaje, que es del tipo `string`, se guarda en el *buffer* `respuesta`. Luego se lee el primer `char` de `respuesta`, el cual describe la acción que debe realizar el *Nodo* en formato `ASCII`. Finalmente, se convierte el `char ASCII` a `int` para posteriormente realizar un `switch` y así determinar de forma ordenada la acción a realizar.

El código de las acciones que realizan los *Nodos* está diseñado de forma tal que no haya ninguna espera bloqueante en el mismo, y que los *Nodos* siempre esperen el mensaje de la próxima acción a realizar en el ciclo principal. Para salir del ciclo principal, se *setea* en `false` la variable `correr` en la acción `quit`.

Para ejecutar los test se deberán insertan los siguientes comandos del `Makefile`:

```
make test-load-run
make test-add-and-inc-run
make test-member-run
make test-maximum-run
```

2. Función `void load(list<string>params)`

Explicación de la implementación

La función `load` de la *Consola* inicia enviándole el mensaje `COMANDO_LOAD` al primer *Nodo* libre, el cual se obtiene mendiante la función `ProximoNodoLibre()`. La función `ProximoNodoLibre` devuelve el primer *Nodo* libre *desencolándolo* de la *cola* `nodosLibres`, que es inicializada por defecto con todos los *Nodos*. En caso de que no haya ningún *Nodo* libre, la función espera, mediante `MPI_Recv`, un mensaje de algún *Nodo* informando que el mismo finalizó su tarea pendiente y en ese caso se retorna el número de este *Nodo*.

Por lo tanto, si hay menos archivos que *Nodos*, se le envia un archivo a cada *Nodo* hasta que no haya más archivos que cargar. En cambio, si hay más archivos que *Nodos*, se espera a que se vayan desocupando para enviarles los archivos faltantes.

Una vez que se enviaron todos los archivos, se espera a que todos los *Nodos* avisen a la *Consola* que terminaron de hacer la carga.

Cuando el *Nodo* recibe el mensaje `COMANDO_LOAD`, ejecuta la función `load` del `Hashmap` local, y por último envía un mensaje a la *Consola* informando que terminó la tarea.

3. Función `void addAndInc(string key)`

Explicación de la implementación

La función `addAndInc` de la *Consola* inicia enviándole el mensaje `COMANDO_TRY_ADD_AND_INC` mediante `MPI_Isend` a todos los *Nodos*. Para cada *Nodo* dicho mensaje representará la orden de reservarse el derecho de agregar el valor `key` a su `HashMap` local. Al recibir el mensaje `COMANDO_TRY_ADD_AND_INC`, el *Nodo* le reporta a la *Consola* su `rank` a través de un mensaje no bloqueante (`MPI_Isend`).

La *Consola* espera de forma bloqueante la llegada de un mensaje de cualquier *Nodo* mediante la función `MPI_Recv`. En particular, la *Consola* espera la respuesta del mensaje previo (`COMANDO_TRY_ADD_AND_INC`). La *Consola* se liberará de la espera bloqueante cuando llegue la primera de esas respuestas, y definirá al *Nodo* responsable de dicho mensaje como el encargado de realizar efectivamente el `addAndInc`. La *Consola* le comunica a dicho *Nodo* que debe realizar la acción `addAndInc` (`COMANDO_DO_ADD_AND_INC`), indicándole además la palabra que se debe agregar (el `string key` pasado por parámetro), mediante la función `MPI_Isend`.

El *Nodo* elegido recibe el mensaje `COMANDO_DO_ADD_AND_INC` y se encarga de separar el `string key` del resto del mismo. Posteriormente, ejecuta la función `addAndInc` del `HashMap` local, pasándole como

parámetro dicho `string`. Cuando la ejecución de la función `addAndInc` concluye, el *Nodo* le comunica a la *Consola* que se ha finalizado exitosamente mediante la función `MPI_Isend` por el `TAG` `TAG_ADDANDINC`.

Mientras tanto, mediante un ciclo, la *Consola* espera a la respuesta al pedido del mensaje `COMANDO_TRY_ADD_AND_INC` del resto de los *Nodos*, ya que de esa forma quedará asegurado que la cola de mensajes pendientes de leer se vacíe y no interfiera con el resto de las funciones.

Por último la *Consola* espera la llegada del mensaje indicando que el *Nodo* elegido ha finalizado exitosamente el `addAndInc` por un `TAG` especial (`TAG_ADDANDINC`).

4. Función `void member(string key)`

Explicación de la implementación

La funcion `member` de la *Consola* inicia enviándole el mensaje `COMANDO_MEMBER` a todos los *Nodos*, indicándoles además la palabra que se debe consultar. Para cada *Nodo* dicho mensaje representará la orden de consultar si la palabra especificada es miembro de su `HashMap` local.

Luego la *Consola* espera a que todos los *Nodos* respondan si tienen o no la palabra. En caso de que algún *Nodo* reporte que la palabra se encuentra en su `HashMap` local, se aplica el valor `true` al `bool esta`.

Los nodos únicamente ejecutan la función `member` de su `HashMap` local y reportan el resultado representado en un `int` hacia la *Consola* mediante `MPI_Isend`.

Cuando todos los *Nodos* terminaron de responder, se consulta la variable `esta`. Si su valor de verdad es `true` entonces la palabra pertenece a algún `HashMap`. Por otro lado, si su valor de verdad es `false` entonces la palabra no pertenece a ningún `HashMap`.

Aclaración: se debe esperar a que todos los Nodos respondan por si o por no para asegurase de que la cola de mensajes pendientes de leer quede vacía.

5. Función `void maximum()`

Explicación de la implementación

La función `maximum` de la *Consola* inicia enviándole el mensaje `COMANDO_MAXIMUM` a todos los *Nodos*. Para cada *Nodo* dicho mensaje representará la orden de enviar todas las palabras del `HashMap` local a la *Consola*.

Posteriormente la *Consola* crea un *HashMap* local donde se agregarán las palabras de todos los *HashMaps* de los *Nodos*. La *Consola* comienza a recibir palabras de cualquier *Nodo*, y las va agregando con la función `addAndInc` del *HashMap* temporal recientemente creado.

Los *Nodos*, cuando reciben el mensaje `COMANDO_MAXIMUM`, comienzan a enviar todas las palabras de su `HashMap` local utilizando el *iterador* de `HashMap` provisto por la cátedra. Este *iterador* envía tantas veces la palabra como repeticiones tenga. Cuando termina de enviar todas las palabras envía el `string '0'`.

Cuando la *Consola* recibe el `string '0'` (se utiliza un *cero*, ya que las palabras válidas están compuestas únicamente de letras minúsculas) se incrementa un contador de *Nodos* que terminaron de enviar sus palabras.

Cuando todos los *Nodos* enviaron el `string '0'` la *Consola* deja de recibir palabras y ejecuta la función `maximum` del `HashMap` temporal que contiene todas las palabras de todos los *Nodos*.

6. Función `void quit()`

Explicación de la implementación

La *Consola* libera la *cola* de *Nodos* libres creada para la función `load`. Luego se envía a todos los *Nodos* el mensaje `COMANDO_QUIT`, para que cuando finalicen.

Cuando los *Nodos* reciben el mensaje `COMANDO_QUIT` cambian el valor de verdad del `bool correr` (que comienza en `true`) por `false`.

Por último, se libera la memoria del `HashMap` local.