Programación Distribuida: Diseño e Implementación de un Servicio Distribuido de Semáforos utilizando Suzuki-Kasami.

Raúl Monge Anwandter Rodolfo Castillo Mateluna

November 15, 2017

1 Objetivos de la Tarea

1. Implementar el Algoritmo de Suzuki-Kasami para implementar una aplicación de Semáforos.
2. Utilizar RMI de Java para trasmitir objetos remotos e invocar remotamente.

2 Descripción de la Tarea

Se deben crear procesos que participen en el sistema de Semáforo Distribuidos basado en el Algoritmo de Suzuki-Kasami. Sea de forma local o remota, se crearan N procesos que formarán parte de este algoritmo. El programa mostrará como salida: un color (por interfaz gráfica o salida estándar) que representa el estado de ese proceso/nodo de acuerdo al algoritmo implementado y el arreglo RN_i del proceso actual i. Un semáforo puede tener los siguientes estados, así como lo muestra la Figura 1:

- Rojo: El proceso tiene el Token y se encuentra en la sección crítica o a punto de terminar.
- Amarillo: El proceso está esperando por el Token para ingresar a su sección crítica.
- Verde: El proceso está ocioso; y puede o no tener el Token con él.

El sistema de semáforos necesita implementar un solo Token; es decir, existe una sola instancia para todo el sistema, por lo que usted debe hacer uso de RMI para lograr este objetivo (hint: Serializando el objeto).

3 Consideraciones de Diseño

3.1 Semáforo

El semáforo representa los estados posibles que el Algoritmo Suzuki-Kasami puede tener en cada proceso/nodo que desea entrar a su S.C. El Semáforo debe implementar los siguientes métodos:

Figure 1: Diagrama del problema

- `request(id, seq)`: Método que registra un *request* de un proceso remoto. Recibe el *id* del proceso que hace la petición y el número de petición del proceso.
- `waitToken()`: Método que le indica a un proceso remoto que debe esperar por el token para realizar la sección crítica.
- `takeToken(token)`: Método que toma posesión del *token* en el proceso.
- `kill()`: Método que mata el proceso remoto. Debe usar este método para detener el algoritmo de S-K una vez que el token haya pasado por todos los nodos del sistema.

P.D: Si desea implementar más métodos que ayuden en su trabajo, usted es libre de hacerlo en la medida que explique su utilidad detalladamente en el archivo `README`.

4 Consideraciones adicionales

1. Usar el lenguaje de Programación Java
2. Para efectos prácticos, todos los procesos se encontrarán en una misma red IP o red local.
3. Por simplicidad considere que cada nodo tendrá una única sección crítica. Sin embargo, puede optar a una bonificación de puntaje si es que diseña un sistema que soporte múltiples secciones críticas por nodo.

4. Toda operación que involucre comunicación entre procesos será realizada por medio de `RMI` utilizando *al menos* los métodos expuestos anteriormente.

5. Los procesos deben ser ejecutados con 4 argumentos de la forma:

   ```
   process <id> <n> <initialDelay> <bearer>
   ```

   Donde:

   - `id` es el *id* del proceso ejecutado, éste debe ser único e incremental en unidades enteras comenzando desde 0.
   - `n` es el número de procesos totales que existirán en la aplicación distribuida, este número debe ser el mismo para todos los procesos ejecutados.
   - `initialDelay` es el tiempo (en ms) que el proceso esperará para intentar entrar a la sección crítica.
   - `bearer` es un *boolean* que indica si el nodo es el portador inicial del *token* de modo que debe existir un único proceso con este argumento posicional con el valor *true*.

   Importante! Como será indicado más adelante, la ejecución será realizada mediante un *target* de `Makefile`, por lo que deberá investigar como pasar argumentos al ejecutable utilizando este mecanismo.

6. Se realizará una ayudantía especial (a coordinar) para explicar el enunciado de la tarea y una demostración de como utilizar `Java RMI`.

5 Caso de Pruebas

Para probar que el sistema cumpla con las condiciones del problema, se pide a los estudiantes que expongan en un log la información necesaria con un formato bien definido mostrando claramente una línea temporal en cada proceso (de modo que pueda ser verificado un orden lógico). Esto puede hacerlo entregando los valores de las estructuras de datos del algoritmo de forma ordenada y el color de cada semáforo. **Se recomienda que implemente un feedback gráfico**.

6 Condiciones de Entrega de la Tarea

- La Entrega deberá realizarse por Moodle y debe llevar el nombre: **Tarea2SD-Apellido1-Apellido2.tar.gz**.
- El tar.gz debe contener todos los archivos necesarios para correr su tarea. Evite incluir archivos basura. Entre ellos se encuentran y no están limitados a:

  - `README.md`: Usted puede realizar supuestos respecto a su tarea en el archivo README, serán considerados. Recuerde que los supuestos son para simplificar su tarea, no para evitar hacer alguna parte de esta. Además, se pide que se explique la estrategia usada, la cual debe ser concisa y al punto, no se esfuerce con testamentos ya que en lugar de favorecer la nota podrían empeorarla.
  - `Makefile`: El comando `make` deberá iniciar la compilación y generación de los archivos necesarios para correr el programa. Además este deberá incluir *targets* que ejecuten su implementación. Los detalles de su uso deben ser especificados en el `README`.

- La tarea debe ser realizada en parejas. En el caso de tener un grupo, continúe con el mismo.
- Por cada día de atraso se descuentan 20 puntos, hasta un máximo de dos días de retraso, posterior a ese plazo no se moleste en entregarla pues la nota final será 0.
- Cualquier duda, inquietud o reclamo, no dude en hacerla llegar a través del Moodle.
- La tarea debe entregarse antes de las 23:55 hrs del día **1 de diciembre de 2017**.

7 Evaluación de código

- Estructura y Estilo de código (20%): Código bien documentado (comentarios). Una parte importante dentro de la vida profesional como desarrollador es la adopción de **convenciones**, por lo que se exige, para esta ocasión utilizar la **Java code convention**.
- README (5%): Debe contener todo lo especificado en la sección anterior.
- Makefile (10%): comandos necesarios para compilar y ejecutar.
- Diseño de la Solución (30%): Solución que aborda todas los requerimientos de la tarea, elección de estructuras y una arquitectura adecuada, interfaces claramente definidas, manejo adecuado de conexiones y direccionamiento, manejo adecuado de concurrencia. El Diseño es propio, pero debe cumplir con lo especificado en el Enunciado.
- Test, Correctitud y buen funcionamiento de la solución (35%): Que la solución pueda desplegarse correctamente y muestre el correcto funcionamiento de cada solución en tiempo de ejecución.

Agradecimientos

Se agradece a Paulina Silva Ghio por la preparación de una versión anterior de la tarea.

References

[1] R. Monge, *Capítulo 4: Algoritmos Distribuidos*, Apuntes de Curso de Sistemas Distribuidos, 2014.