Redes de Computadoras

Obligatorio 3 - 2017

Facultad de Ingeniería
Instituto de Computación
Departamento de Arquitectura de Sistemas

Nota previa - IMPORTANTE

Se debe cumplir íntegramente el “Reglamento del Instituto de Computación ante Instancias de No Individualidad en los Laboratorios”, disponible en el EVA.

En particular está prohibido utilizar documentación de otros estudiantes, de otros años, de cualquier índole, o hacer público código a través de cualquier medio (EVA, news, correo, papeles sobre la mesa, etc.).

Introducción

Forma de entrega

Una clara, concisa y descriptiva documentación es clave para comprender el trabajo realizado. La entrega de la tarea consiste en un único archivo `obligatorio3GrupoGG.tar.gz` que deberá contener los siguientes archivos:

- Un documento llamado `Obligatorio3GrupoGG.pdf` donde se documente todo lo solicitado en la tarea. `GG` es el número del grupo. La documentación deberá describir claramente su solución, las decisiones tomadas, los problemas encontrados y posibles mejoras
- Los programas solicitados.
- Un directorio `extras` incluyendo cualquier otro archivo que considere relevante.

La entrega se realizará en el sitio del curso, en la plataforma EVA.

Fecha de entrega

Los trabajos deberán ser entregados antes del 12/11/2017 a las 23:30 horas. No se aceptará ningún trabajo pasada la citada fecha y hora. En particular, no se aceptarán trabajos enviados por e-mail a los docentes del curso.

Objetivo del Trabajo

Repasar y fortalecer el aprendizaje de los conceptos teóricos de enrutamiento en una red mediante el diseño y la implementación de algoritmos de enrutamiento sobre un simulador.
Entender las ventajas y desventajas de los algoritmos de *vector de distancias* y de *estado de enlace,* así como sus posibles problemas y las soluciones existentes a estos problemas.
Introducir al estudiante en los conceptos de programación distribuida y asíncrona.

Descripción del problema

La tarea consiste en diseñar, implementar y evaluar un algoritmo de enrutamiento de vector de distancias y un algoritmo de enrutamiento de estado de enlace. Los algoritmos se implementarán como programas Java sobre un simulador orientado a eventos. Los algoritmos deberán de ser asíncronos y distribuidos.

También deberá escribir un informe donde describa y discuta como funcionan los algoritmos implementados, como los probó, que problemas encontró en cada uno de ellos y sus posibles soluciones.

Importante: Antes de comenzar con la programación de este obligatorio, usted debe:

- *leer y entender completamente como se implementan y funcionan los algoritmos de enrutamiento de vector de distancias y de estado de enlace.*
- *entender el problema de los bucles de enrutamiento (conteo a infinito), y como la técnica de reversa envenenada soluciona el problema,*
- *entender la necesidad de los algoritmos de estado de enlace de conocer la topología de la red y las consecuencias de esto,*
- *leer esta letra completamente,*
- *leer el código fuente entregado para entender como funciona la comunicación entre nodos.*

El reto principal de este obligatorio es que su código debe ser asíncrono y distribuido. Para la implementación se utilizará un simulador orientado a eventos que gestiona la comunicación entre *routers*. Debe implementar la parte del algoritmo que se ejecuta en cada router. Aunque los routers están activos en el mismo programa (simulador), sólo pueden comunicarse entre sí mediante el envío de mensajes a través del simulador.

Se utilizarán los siguientes archivos (que se pueden descargar compirmidos desde [1]):

- `RouterNode.java`
- `RouterSimulator.java`
- `RouterPacket.java`
- `GuiTextArea.java`
- `F.java`
- `Makefile`

Después de extraer el archivo anterior, también se crea una carpeta de prueba que contiene tres versiones del simulador que se pueden utilizar para probar su código:

- `RouterSimulator3.java` que simula una red con tres nodos, como se muestra en la Figura 1.
- `RouterSimulator4.java` es el mismo que `RouterSimulator.java` y simula una red con cuatro nodos, como se muestra en la Figura 2.
- `RouterSimulator5.java` que simula una red con cinco nodos, como se muestra en la Figura 3.

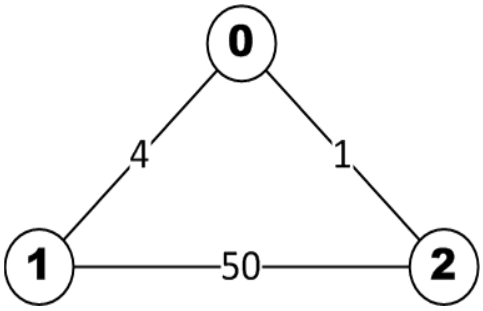


Figura 1

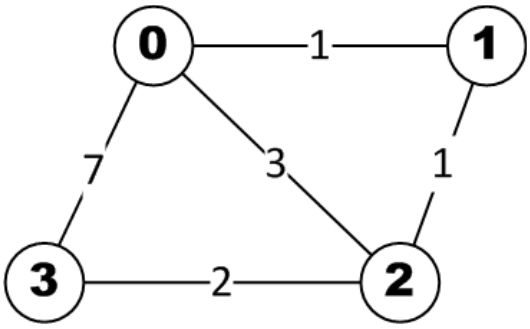


Figura 2

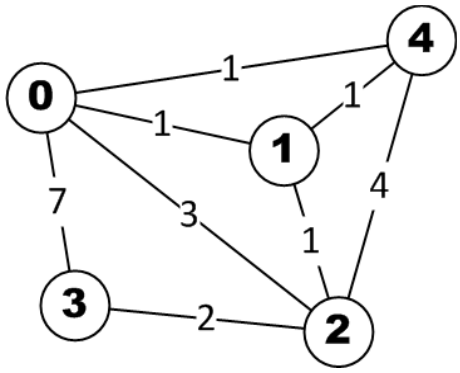


Figura 3

Sólo deberá agregar código al archivo `RouterNode.java`. Para probar el código implementado (en `RouterNode.java`), también puede ser necesario modificar (ligeramente) el código en `RouterSimulator.java`. Puede utilizar la clase `F` en el archivo `F.java` para realizar una salida de texto formateada.

Debe agregar código para el constructor y los métodos de la clase `RouterNode` que se encuentra en el archivo `RouterNode.java`. Sólo puede comunicarse con otros routers a través del método `sendUpdate()`. No debe agregar ningún miembro estático a esta clase, o de alguna manera evadir el requerimiento de que toda la información debe ir a través de `sendUpdate()`. Además, no es aceptable utilizar valores específicos en los costos de los enlaces (tales como números negativos) para indicar a los otros routers que realicen una operación.

Otros métodos en la clase `RouterNode` son:

- `recvUpdate()` que es llamado por el simulador cuando un nodo recibe una actualización o mensaje de uno de sus vecinos,
- `updateLinkCost()` que es llamado por el simulador cuando el costo del enlace de un nodo está a punto de cambiar,
- `printDistanceTable()` que se utiliza para depurar y probar el código, y deberá utilizarlos para mostrar su solución en la defensa. Este método debe imprimir la tabla de enrutamiento en un formato que usted y su docente puedan leer y entender.

Importante: Sus tablas deben ser autoexplicativas, y cuando muestre su código al docente en la defensa, debería ser capaz de explicar exactamente lo que contienen sus tablas.

Se pide:

Parte A

Implemente un algoritmo de vector de distancias basado en la ecuación de Bellman-Ford y siguiendo los lineamientos vistos en teórico.
Cada router solo conoce sus vecinos y a diferencia del algoritmo presentado en el libro no conoce la cantidad total de nodos de la red.

Su solución debe además implementar la técnica de *reversa envenenada* y debe permitir poder habilitar y deshabilitar esta característica, de tal manera que el sistema se pueda comparar con y sin la reversa envenenada (durante la defensa, por ejemplo).

En su implementación, en la ventana de cada uno de los routers, debe mostrarse el vector de distancia mínimo y la ruta (salto siguiente) a tomar para llegar a cada uno de los destinos. Opcionalmente, también se puede mostrar el coste físico para cada uno de los vecinos, así como los vectores de distancia mínimos recibidos de nodos vecinos.

Parte B

Implemente un algoritmo de estado de enlace, utilizando el algoritmo de Dijkstra y basado en el algoritmo visto en teórico.

Será necesario que cada router conozca toda la topología de la red, por lo que deberá disñar e implementar un mecanismo para esta tarea. Se sugiere que implemente un algoritmo de *flooding* controlado donde cada nodo difunde mensajes de *aviso de estado de enlace* (LSA).
Cada router solo conoce sus vecinos y a diferencia del algoritmo presentado en el libro no conoce la cantidad total de nodos de la red.

En su implementación, en la ventana de cada uno de los routers, debe mostrarse la topología aprendida y la ruta (salto siguiente) a tomar para llegar a cada uno de los destinos.

Anexo - Uso del simulador

Ejecución

El simulador comienza inicializando cada router. Aparecen varias ventanas en la pantalla una para la salida del simulador y otra para cada nodo de los routers que se inicializan, vea la Figura 4. El título de la ventana describe lo que está en ella. La interfaz gráfica de usuario se implementa en el archivo `GuiTextArea.java`.

Su código debe garantizar que se producen actualizaciones en la red. Es decir, dado que el simulador se detiene tan pronto como no hay mas transmisiones de paquetes, su código debe ser tal que tan pronto como un nodo se inicializa, envía actualizaciones a sus vecinos. El simulador funcionará hasta que las tablas de enrutamiento hayan convergido, es decir, cuando no haya más mensajes o eventos en el sistema.

El simulador agrega aleatoriead al tiempo que tardan los mensajes en propagarse por los enlaces para evitar sincronismo. Puede cambiar el valor de la semilla para los números pseudo-aleatorios en el simulador. Por ejemplo, para usar un valor de 123, puede usar -DSeed = 123:

```
java –DSeed=123 RouterSimulator
```

El uso de otro valor inicial para los números pseudo-aleatorios da una secuencia diferente de eventos para los mensajes en su red.

El simulador también permite agregar tiempos de espera entre la ejecución de eventos. De esta forma se puede hacer la ejecución "mas lenta" y visualizar mejor el intercambio de mensajes. Por ejemplo para agregar 1 segundo entre eventos puede hacer:

```
java –DSlow=1000 RouterSimulator
```

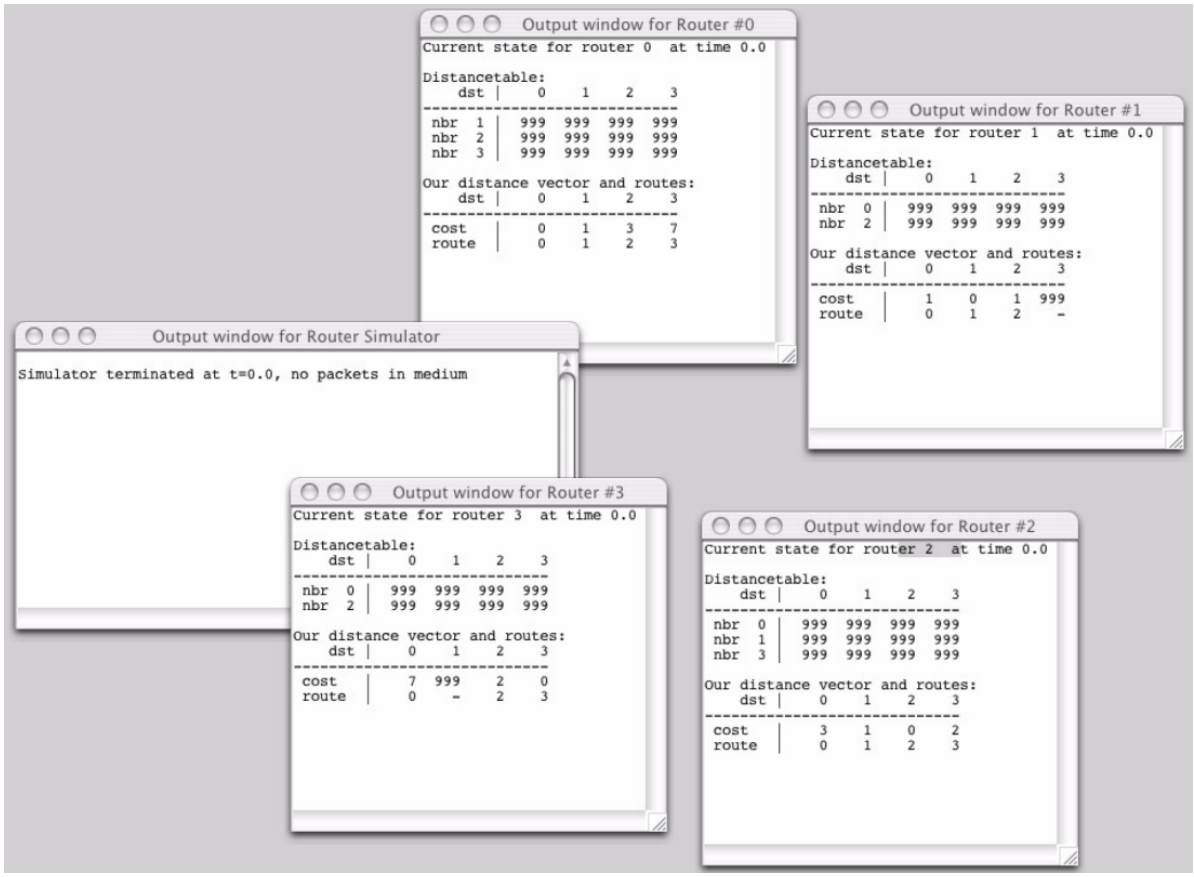


Figura 4

Costos y Eventos

Debe asegurarse de que la topología de red o los costos de los enlaces no cambien como resultado de ejecutar su implementación del algoritmo de enrutamiento. Sin embargo, como se verá más adelante, puede utilizar eventos en la clase `RouterSimulator` para estudiar cómo responde su algoritmo a los cambios en los costos de los enlaces.

Para probar su código exhaustivamente deberá cambiar los costos de los enlaces y probar con varias configuraciones. Esto lo puede hacer simplemente cambiando la estructura de datos `connectcosts[][]` en `RouterSimulator()`, vea la Figura 5. Asegúrese de poner los mismos costos de enlace en ambas direcciones en cada enlace, ya que en este obligatorio solo se considerarám costos de enlace simétricos. También puede experimentar con nuevas topologías cambiando el número de nodos (si es necesario) y modificando el array `connectcosts[][]` según sea necesario. Para cambiar el número de nodos, debe modificar el valor asignado a `NUM_NODES` en el constructor de la clase `RouterSimulator`.

```
/* set initial costs */
// remember that in java everything defaults to 0
connectcosts[0][1]=1;
connectcosts[0][2]=3;
connectcosts[0][3]=7;
connectcosts[1][0]=1;
connectcosts[1][2]=1;
connectcosts[1][3]=INFINITY;
connectcosts[2][0]=3;
connectcosts[2][1]=1;
connectcosts[2][3]=2;
connectcosts[3][0]=7;
connectcosts[3][1]=INFINITY;
connectcosts[3][2]=2;
```

Figura 5

El simulador nunca enviará mensajes `RouterPacket` por sí mismo. Es su código el que debe garantizar que la información se intercambia entre routers en la red.

El archivo `RouterNode.java` tiene una variable llamada `costs` que se inicializa en el constructor. El simulador establece los costos de enlace asociados con un nodo a través de la variable `costs`. Por ejemplo, en la red mostrada en la Figura 2, los costos de enlace para el nodo 1 se ajusta mediante el vector `{1, 0, 1, RouterSimulator.Infinity}`, lo que significa que el costo del enlace (desde el Nodo 1) al nodo 0 es 1, a sí mismo es 0, al nodo 2 es 1, y al nodo 3 es infinito.

Los costos de todos los enlaces se establecen en el constructor para la clase `RouterSimulator` como se discutió en la Figura 5. Más adelante en ese código, se añaden dos eventos que cambian los costos del enlace en momentos específicos durante el período de simulación. El enlace entre `eventity` y `dest` cambia a `cost` en tiempo `evtime`. Un código de ejemplo para añadir un evento que cambia tales costos se muestra en la Figura 6. Tenga en cuenta que los costos de los enlaces no van a cambiar si no se cambia el valor de las constantes `LINKCHANGES` en `true` en el principio del archivo `RouterSimulator.java`.

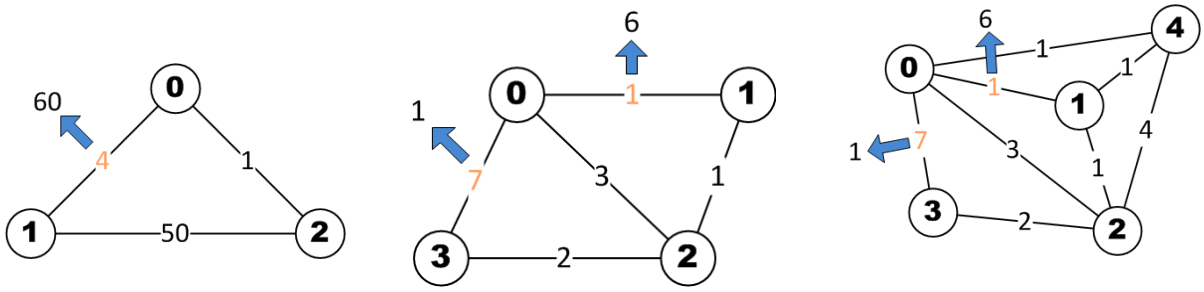
Por favor, cambie los costos del enlace y los eventos para probar su código. Cuando ocurre un cambio de coste de enlace (que se programa mediante el uso de los eventos mencionados anteriormente), el simulador notifica a los routers llamando al método `updateLinkCost()`.

```
/* initialize future link changes */
if (LINKCHANGES)    {
  evptr = new Event();
  evptr.evtime =  40;
  evptr.evtype =  LINK_CHANGE;
  evptr.eventity =  0;
  evptr.rtpktptr =  null;
  evptr.dest = 1;
  evptr.cost = 60;
  insertevent(evptr);
}
```

Figura 6

Cada una de las tres copias del simulador (para las redes ilustradas en las figuras 1, 2 y 3) tiene

programado al menos un evento de cambio de costo de enlace. Las siguientes figuras muestran cómo en cada una de las topologías cambian los costos de enlace:



Referencias y Bibliografía Recomendada

[1] https://eva.fing.edu.uy/mod/resource/view.php?id=71140
[2] RFC 2453. RIP v2. https://tools.ietf.org/html/rfc2453
[3] RFC 1142. IS-IS. https://tools.ietf.org/html/rfc2453
[4] RFC 2328. OSPF v2. https://www.ietf.org/rfc/rfc2328.txt