Author name(s)

Book title

– sub titulo –

19 de enero de 2012

Apéndice A

Simulador de eventos discretos

Resumen En la computación distribuida han habido pocos trabajos que estudien el rendimiento de algoritmos distribuidos considerando las diferentes condiciones de la red que subyace en las comunicaciones. Los resultados existentes se basan en estudios analíticos que modelan condiciones generales. En contraste con esta aproximación planteamos el uso de la Simulación de Eventos Discretos (SED) como una forma para estudiar algoritmos sobre condiciones particulares tales como cambios en la topología o diferentes retardos en los enlaces. En este apéndice describimos la construcción y operación de una sencilla plataforma que provee un ambiente para el análisis de algoritmos distribuidos. Nuestra propuesta permite escribir algoritmos distribuidos en el lenguaje Python. Iniciamos con una presentación de las necesidades concretas que dieron origen a nuestro simulador, las propiedades que le dan su perfil característico y los modelos que puede soportar. Posteriormente listamos las funciones que debe realizar un simulador de este tipo y revisamos la arquitectura de nuestra solución. Finalmente, describimos un sencillo ejemplo de aplicación.

A.1. Contexto

Imaginemos una red de telecomunicaciones para la que debemos de estudiar algunas de sus operaciones. En muchos casos será suficiente con un monitoreo adecuado que permita caracterizar las tareas que son de nuestro interés. Sin embargo, si la red existiera únicamente en planos, si se quisieran analizar situaciones hipotéticas, si se quisiera predecir sus comportamiento, incluso, si la acción de monitoreo interfiriera con la misma calidad de los servicios, entonces tendríamos que buscar un camino alternativo: un modelo abstracto que representara al sistema bajo estudio y sobre el que pudiéramos efectuar acciones cuyas consecuencias nos permitieran inferir las respuestas que nos interesan.

Construir un modelo como sustituto de un sistema real significa usar un lenguaje formal mediante el cual describimos las partes del sistema que parecen relevantes para nuestro problema y establecemos las relaciones entre estos componentes. Las construcciones de un lenguaje formal pueden manipularse mediante un conjunto de herramientas para producir inferencias sobre las propiedades planteadas por el modelo.

Por otro lado, los métodos analíticos tienen situaciones en las que su aplicación queda rebasada. En estas circunstancias podemos apoyarnos en la simulación de eventos discretos (SED) en donde la descripción se hace por medio de un lenguaje de computadora y la herramienta de inferencia es la computadora misma. El modelo del sistema se describe mediante un programa que se ejecuta para generar una "historia" del sistema que representa. El programa se construye bajo el supuesto de respetar las relaciones causa-efecto que se dan en el sistema real y esto le otorga a dicha historia artificial una validez y un sentido predictivo.

¿Cuáles son las ventajas de un método analítico en comparación con la SED? El primer método tiene la ventaja de producir resultados en término de parámetros de desempeño y por lo tanto sus soluciones son generales. Su inconveniente radica en el número limitado de sistemas reales cuyos modelos tienen solución. En contraste, la SED siempre produce una solución pero, cualquier cambio en los parámetros del sistema real requiere un nuevo programa que lo refleje, es decir no es generalizable. Este método resulta más indicado cuando se necesita describir un sistema muy complejo o con mucho detalle, su mayor inconveniente, como ya hemos dicho, puede ser el tiempo necesario para obtener un resultado con valor predictivo.

A.2. Características

Desarrollamos una plataforma de software que ofrece un ambiente para la simulación y análisis de algoritmos distribuidos. Con esta herramienta, un programador codifica su algoritmo en Python [**?**] con el fin de ligarlo a las bibliotecas de nuestro sistema. El diseño se basa en el concepto de máquina de estados (un recurso teórico de cómputo distribuido) para describir la interacción entre entidades autónomas. Estos principios teóricos unidos a un diseño orientado a objetos nos permiten ofrecer un producto con las siguientes características:

1. Una plataforma de código abierto que permite separar por un lado, el algoritmo y, por otro, la gráfica que representa la red de comunicaciones. Con ello es posible simular el algoritmo sobre diferentes topologías, sin modificar el código de la simulación.
2. Permite simular eventos aleatorios, dejando al programador en libertad para especificar cualquier función de distribución de probabilidad, ya sea para caracterizar el tiempo de un paso de procesamiento, el retardo de transmisión de un mensaje, o el tiempo en que se presenta una falla.
3. Cuenta con un mecanismo de comunicación entre las entidades activas basado en el modelo de paso de mensajes. El usuario puede extender la definición de los mensajes, para establecer sus propias unidades de información.
4. Permite simular la ejecución concurrente del mismo algoritmo, para estudiar, por ejemplo, situaciones de competencia.
5. Permite simular, por ejemplo, situaciones de cooperación entre entidades (e.g. sincronizadores o elección).

A.3. Modelos soportados

Las máquinas de estados finitos son el recurso teórico en que se basan nuestro modelo de algoritmo, puesto que sirven para describir la interacción entre entidades, con base en el intercambio discreto de información. Las entidades de las máquinas de estados finitos que mejor se adaptan a nuestros fines son las máquinas comunicantes de estados finitos y los autómatas I/O.

Una máquina comunicante de estados finitos es una entidad abstracta que acepta símbolos de entrada, genera símbolos de salida y cambia a su estado interno de acuerdo con un plan predefinido. Estas entidades pueden comunicarse a través de canales FIFO de capacidad acotada, que mapean la salida de una máquina sobre la entrada de otra.

Por su parte, un autómata I/O modela los componentes de un sistema distribuido y la interacción que se da entre ellos. Es un tipo muy simple de máquina de estados en la que las transiciones están asociadas con acciones designadas. Las acciones se clasifican como entradas, salidas o internas. Las primeras dos se usan para la comunicación entre el autómata y su entorno, mientras que las acciones internas sólo son visibles para el autómata mismo. Las acciones de entrada no están bajo el control del autómata, ya que ocurren desde el exterior, mientras que él mismo especifica las acciones internas y las salidas que debe efectuar.

Nuestra plataforma de simulación trabaja con sistemas cuyas entidades activas pueden modelarse mediante máquinas de estado como las expuestas. Sin embargo, estos modelos sirven para caracterizar cada uno de los componentes activos de un sistema distribuido. También necesitamos de otro modelo para describir a un sistema en su conjunto. Para ello recurrimos al modelo de red asíncrona con intercambio de mensajes, que se representa por una gráfica de comunicaciones $G = (V, E)$. En cada elemento del conjunto de nodos V, se localiza una entidad activa o proceso. En tanto, el conjunto de enlaces E representa los canales mediante los cuales se comunican los procesos. Cada nodo procesa los mensajes que puede recibir de sus vecinos, efectúa un cómputo local y puede enviar, a su vez, mensajes a sus vecinos. Todos los mensajes son de longitud acotada y transportan una cantidad finita de información. Cada mensaje transmitido sobre un canal llega a su destino al cabo de un retardo finito.

Usando nuestra herramienta es posible estudiar la ejecución concurrente de uno o varios algoritmos, sobre los diferentes puntos de una gráfica de comunicaciones. Al mismo tiempo, es posible ejecutar un algoritmo sobre graficas diferentes, ya que estas se consideran condiciones iniciales del experimento.

A.4. Las partes de un simulador de eventos discretos

Los simuladores de eventos discretos tienen una estructura en común, un conjunto de elementos que se muestran a continuación y que, si se utiliza un lenguaje de propósito general deberán ser implementados por el desarrollador:

- Calendarizador de eventos
- Reloj y mecanismo de actualización de tiempo
- Variables de estado
- Rutinas para manejo de eventos

- Rutinas de entrada
- Rutinas de inicialización
- Rutinas para registro de trazas
- Mecanismos para manejo dinámico de memoria
- Programa principal

A.4.1. Procedimiento básico

Un SED utiliza un calendarizador que administra una estructura de datos, o agenda, donde se almacenan los eventos, ordenados de acuerdo con sus tiempos de atención. Esto es, serán despachados por las rutinas correspondientes cuando el reloj de la simulación avance hasta habilitarlos. Se garantiza que un evento o currirá en el tiempo que tiene asociado, siempre que la entidad que lo genera no reciba, antes de este tiempo, otro evento que cancele al primero. En cada ciclo de simulación, el evento fechado con el menor tiempo futuro es removido de la agenda y se simula la recepción del evento, como correspondería en el sistema físico que se representa. La atención de un evento puede, a su vez, causar nuevos eventos en el futuro (que entonces son agregados a la lista de eventos) o la cancelación de eventos previamente programados (que entonces se remueven de la lista). El reloj se avanza entonces, hasta el tiempo del evento cuya simulación acaba de completarse.

A.5. Estructura y funcionamiento

El simulador que presentamos fue diseñado y construido usando un enfoque orientado a objetos. La reutilización es un concepto clave de nuestro diseño con el que fue posible construir un sistema compacto, con aproximadamente 200 líneas de código escrito en Python.

Las clases implementadas en el sistema se presentan en la figura A.1 y se describen a continuación:

1. Event: Representa un paquete de información que se intercambia entre entidades activas
2. Model: Representa la rutina para la atención de eventos, está basada en un modelo de máquina de estados finitos. También efectua funciones de reporte y generación de trazas.
3. Process: Representa una entidad activa relacionada con un modelo.
4. Simulator: Representa al calendarizador de eventos así como tareas de gestión de reloj.
5. Simulation: Representa al administrador que establece los canales entre procesos y coordina su comunicación. También efectúa funciones de inicialización.

Una instancia de la clase “Simulation”, denominada *myExperiment*, consta de 3 atributos: una gráfica de comunicaciones, una tabla de procesos y un calendarizador o despachador de la simulación. En cada paso, myExperiment invoca al despachador para recoger al próximo evento que debe atenderse. Enseguida, determina la identidad del proceso al que va dirigido, lo busca en su tabla y le envía el evento correspondiente. Dicho proceso entonces consulta su modelo para determinar qué acciones deben

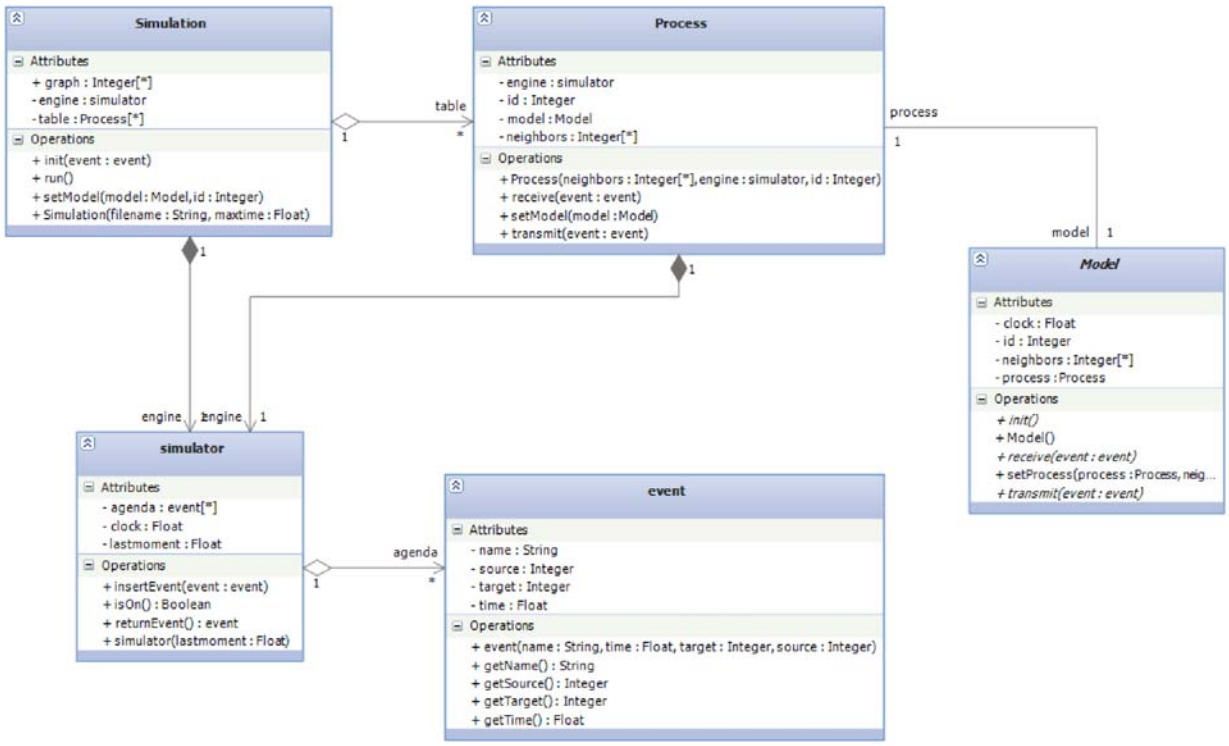

Figura A.1 Diagrama UML de clases del simulador

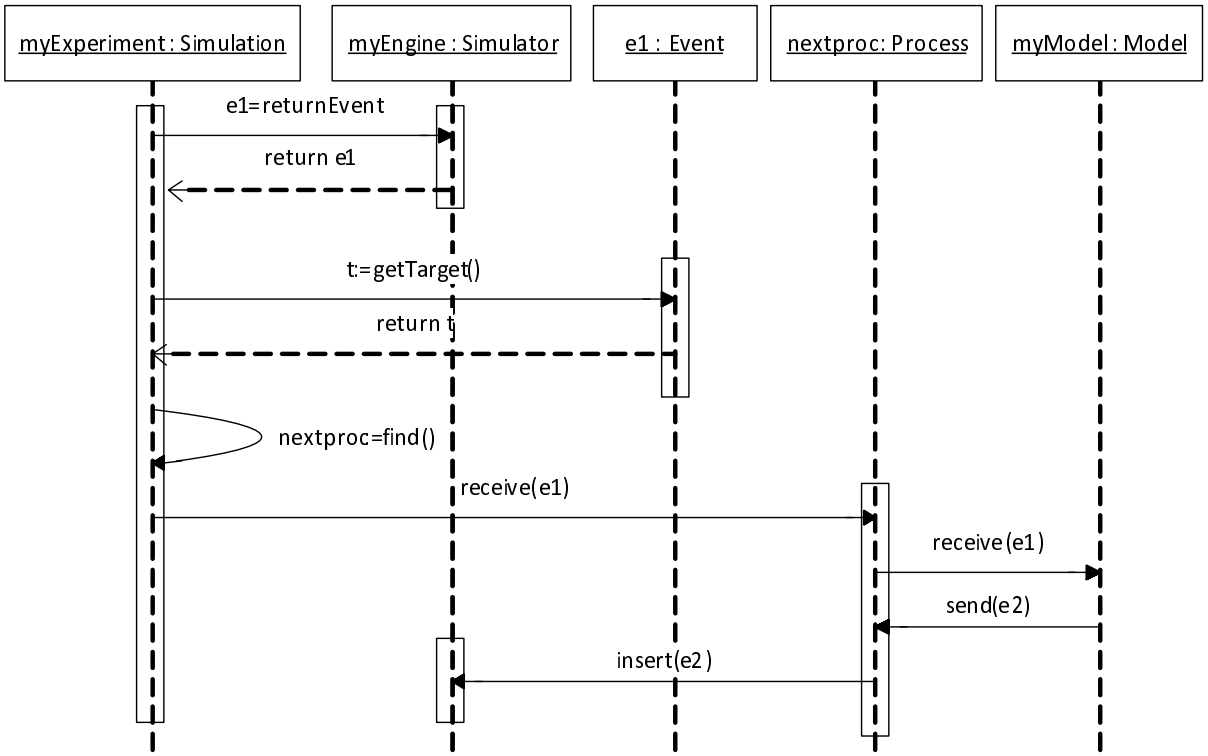

Figura A.2 Diagrama de secuencias

tomarse como respuesta. Si se requiriere, el modelo invocara los métodos de transmisión de su proceso, que se traducirán en la inserción sobre la agenda del despachador (ver figura A.2).

A.5.1. Instalación y operación

En esta sección describimos la instalación del simulador, el resto del capítulo utilizamos un sencillo ejemplo de algoritmo distribuido, a manera de hilo conductor, para mostrar cómo se programa su simulación, y cómo se construyen las redes sobre las que se quiere simular un algoritmo.

Instalación

Para usar nuestro simulador, es necesario cubrir una serie de requisitos que se describen a continuación.

1. Tener instalador el intérprete de Python
   - En caso de no tenerlo instalado recomendamos consultar la siguiente dirección:

     `http://www.python.org`
2. Copiar la carpeta simulador.v2 incluida en el CD de distribución. Recomendamos la dirección `c:\simulador.v2`, pero es opcional debido a que, si se configuró una variable de entorno en el caso de Windows para su interprete Python, no debe tener problemas para invocar al compilador desde cualquier punto de su sistema de archivos. Esta carpeta debe contener los siguientes archivos: event.py, model.py, process.py, simulation.py, simulator.py, algorithm1.py, algorithm2.py, grafica.txt

Ejemplo de operación: Propagación simple (PI)

Descripción del algoritmo

En esta sección presentamos un sencillo algoritmo distribuido con el fin de desarrollar la primera aplicación de nuestro simulador. Considérese una red cuya grafica subyacente es $G = (V,E)$, en la que existe un proceso $s \in V$ que debe transmitir un mensaje a los demás sitios del sistema. Para el modelo de red asíncrona de mensajes con el que se trabaja, sólo se requiere que G sea conexa, esto es, que exista un camino entre cualquier pareja de vértices. Se utiliza un modelo de comunicación asíncrona con intercambio de mensajes, donde los mensajes se entregan en el mismo orden en que ingresan al canal (canales FIFO) y al cabo de un tiempo finito.

El algoritmo de Propagación Simple (PI), especifica que sobre los canales de comunicaciones cada proces envíe un mensaje a cada uno de sus vecinos, estos lo reciben y rexpiden por todas sus líneas de comunicación. A su vez, los procesos que reciban efectuaran exactamente la misma operación

Un proceso da por terminada su ejecución, en cuanto envía el mensaje a sus vecinos. Es fácil comprobar que la ejecución del algoritmo termina satisfactoriamente y que cualquier proceso acaba recibiendo el mensaje proveniente de s (véase el algoritmo A.1)

Algoritmo A.1: Algoritmo PI

```
1 al recibir M desde j efectua
2     si (visitado(i) == FALSO) entonces
3         si M entonces
4             padre(i) ← j
5             visitado(i) = VERDADERO
6             envia M a todos los vecinos
```

En el algoritmo que se propone, los participantes intercambian el siguiente mensaje:

M: la información o ficha que se propaga por la red.

Al mismo tiempo, un procesador *i* mantiene registros donde almacena su condición local. Estas variables se denominan:

visitado(i): cuyo valor es VERDADERO si ya ha recibido un mensaje y FALSO en otra circunstancia (valor inicial).

vecinos(i): El conjunto de nodos con los que *i* comparte una arista.

Simulando un algoritmo PI

Nuestro programa será codificado en un archivo que llamaremos pi.py que después ejecutaremos mediante el intérprete de Python. Dividiremos la implementación del programa en 2 secciones: la primera, que representa a la función de inicio, donde definiremos las propiedades generales y condiciones iniciales de la simulación. En la segunda sección definiremos el código del algoritmo distribuido que nos interesa estudiar. Se sugiere que la función de inicio se escriba al final del archivo (vease figura A.3). Sin embargo, consideramos que es preferible revisar las partes del archivo en el orden que hemos sugerido.

La función inicial de nuestro ejemplo tiene el parámetro `sys.argv` a través del cual se pueden ingresar cadenas de caracteres al programa al inicio de su ejecución con el propósito de inicializar las condiciones de simulación. La descripción de esta función se muestra a continuación:

- La primer orden en el cuerpo de la función indica que deseamos crear una instancia de la clase “Simulation”, a la que llamaremos “experiment” que construirá su gráfica de comunicaciones a partir del archivo con nombre `sys.argv[1]` y que estará programada para funcionar durante 500 unidades de tiempo simulado (vease figura A.3).
- A continuación, vamos a instalar en cada nodo de la red, una instancia particular de nuestro algoritmo, que estará a cargo de la entidad activa en dicho punto. Esta es la acción del ciclo que se muestra en el código (vease figura A.3).
- Las últimas tres instrucciones deben revisarse en bloque. En primer lugar debemos crear un evento de arranque o semilla que dispare la secuencia que queremos simular. Este evento tiene nombre “C”, está programado para ocurrir en el instante 1.0, su origen es el nodo con id 1 y su destino el mismo nodo. En otras palabras, estamos creando un temporizador que dispare el algoritmo en el nodo 1. En seguida le pedimos a “experiment” que lo inserte en la agenda de eventos que deberá gestionar. Por último, le pedimos que arranque su motor de ejecución (véase figura A.3).

```
# ------------------------------------------------------------------------------
# CODIGO DEL ALGORITMO DISTRIBUIDO
# ------------------------------------------------------------------------------
""" Implementa la simulación del algoritmo de Propagación de Información (Segall)
como ejemplo de aplicación """

import sys
from event import Event
from model import Model
from simulation import Simulation

class Algorithm1(Model):
    """ La clase Algorithm desciende de la clase Model e implementa los métodos
    "init()" y "receive()", que en la clase madre se definen como abstractos """

    def init(self):
        """ Aqui se definen e inicializan los atributos particulares del algoritmo """
        self.visited = False
        print "inicializo algoritmo"

    def receive(self, event):
        """ Aquí se definen las acciones concretas que deben ejecutarse cuando se
        recibe un evento """
        if self.visited == False:
            print "recibo mensaje"
            self.visited = True
            for t in self.neighbors:
                newevent = Event("C", self.clock+1.0, t, self.id)
                self.transmit(newevent)

# ------------------------------------------------------------------------------
# FUNCION DE INICIO
# ------------------------------------------------------------------------------
# construye una instancia de la clase Simulation recibiendo como parámetros
# el nombre del archivo que codifica la lista de adyacencias de la grafica y el
# tiempo max. de simulación

if len(sys.argv) != 2:
    print "Please supply a file name"
    raise SystemExit(1)
experiment = Simulation(sys.argv[1], 500)

# asocia un pareja proceso/modelo con cada nodo de la grafica
for i in range(1,len(experiment.graph)+1):
    m = Algorithm1()
    experiment.setModel(m, i)

# inserta un evento semilla en la agenda y arranca
seed = Event("C", 0.0, 1, 1)
experiment.init(seed)
experiment.run()
```

Figura A.3 Código de la función inicial del algoritmo PI en el lenguaje Phyton

La siguiente sección del archivo “pi.py”, contiene el código del algoritmo que simulará cada nodo (vease figura A.3). Todo algoritmo debe codificarse como una extensión publica de la clase “Model”. De esta forma, la clase “Algorithm1” heredará todos los atributos y métodos de su clase madre. Los atributos que hereda incluyen: un identificador, denominado `id` (de tipo int), un reloj local `clock` (de tipo float), una instancia de la clase “Process”, denominada `process`, una lista de nodos vecinos denominada `neighbors`. Por otro lado, hereda los métodos `init()`, `receive()`, y `transmit()`. Los dos primeros son métodos virtuales, lo que quiere decir que están declarados en la clase madre, pero son las extensiones quienes se encargan de implementarlos.

La nueva clase que definimos introduce un atributo privado denominado `visited` (de tipo booleano) y un método publico denominado `send()`, sus demás métodos son las implementaciones de `init()` y `receive()`, que corren bajo su responsabilidad pero, como ya se mencionó, están declaradas en la clase “Model”. El método `init()` es invocado por el simulador al momento de crear una de las instancias del algoritmo que instala en los nodos de la red (como lo vimos en la función inicial), su tarea es crear las condiciones locales con las que debe arrancar el algoritmo en cada sitio de la red. En este caso particular, la tarea se reduce a fijar en “False” el valor del atributo `visited`.

Por su parte, el método `receive()` simula la llegada de un paquete de información, de la clase “Event”, que se entrega a la entidad activa (proceso) a cargo del algoritmo. Su acción por defecto es consultar al método `receive()`, del modelo que tiene asociado. Cuando un nodo que ejecuta el algoritmo PI recibe un paquete de información, revisa el valor de su variable `visited` para determinar si es la primera vez que lo recibe o no. En el primer caso, puede desplegar información útil para el analista, como su identificador (`id`) y la fecha del evento. Una manera de saber su `id`, es invocando el método `getTarget()` del evento que recibe, (pero también podría utilizarse el atributo `id`). Por otro lado, la fecha de recepción se obtiene consultando el reloj local. Enseguida, se debe reexpedir el mensaje `m` hacia cada uno de los vecinos. Para ello se utiliza la lista `neighbors` que se hereda de la clase “Model”. De esta se obtiene el identificador del nodo hacia donde debe reexpedirse la información y, entonces, se invoca el método `send()`. Por último, se cambia el valor de la variable `visited` a `True`, toda vez que no se debe efectuar más que una vez esta operación de propagación. La próxima vez que se reciba un paquete de información, el algoritmo deberá responder avisando que ha terminado sus operaciones.

El método `send()` recibe como parámetros el nombre de un mensaje y el identificador del nodo a quien se debe enviar. Para crear una instancia de la clase .Event”(cuya definición revisaremos enseguida), debe incluir su nombre, el tiempo en el que debe ocurrir, el id del nodo destino y el id del nodo de origen. El nombre y el destino están dados desde la invocación del método. El tiempo se fija incrementando en 1 el reloj local. En una aplicación más elaborada, el analista podría incrementar el reloj local en una cantidad aleatoria para simular retardos variables en la transmisión. Finalmente, el id del nodo que envía se obtiene del atributo `id`. Cuando el mensaje se termina de construir, se invoca el método `transmit()`, heredado de “Model”.

El siguiente paso es preparar el archivo de texto que codifique a la gráfica sobre la que queremos experimentar nuestro algoritmo.

Codificando una gráfica

Como sabemos, se puede simular un mismo algoritmo sobre diferentes redes sin modificar una línea de su código. Esto se debe a que cada red se almacena como un archivo de texto que puede proporcionarse a la ejecución en el momento de su arranque. Este archivo contiene la lista de adyacencias de la red que se quiere estudiar. Utilizaremos como ejemplo la red que se muestra en la figura A.4.

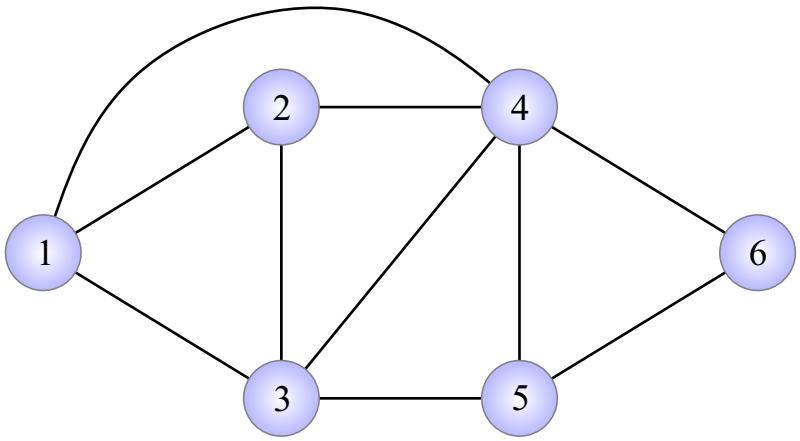

Figura A.4 Red de comunicación

Para producir nuestro archivo debemos utilizar cualquier editor que pueda generar caracteres tipo ASCII. El archivo generado está formado por renglones, donde el i-ésimo renglón del archivo contiene la lista de identificadores de los nodos con los que el proceso *i* comparte una arista. Cada identificador debe estar precedido de un espacio en blanco y el renglón debe finalizar con un carácter de retorno de carro. Por ejemplo, el archivo “graf1.txt”, tendría el contenido mostrado en la figura A.5.

```
2 3 4
1 3 4
1 2 4 5
1 2 3 5 6
3 4 6
4 5
```

Figura A.5 Contenido del archivo “graf1.txt”

El primer renglón, por ejemplo, nos dice que el nodo 1 tiene conexión con los nodos 2, 3 y 4. De manera semejante se codifica cada uno de los renglones de “graf1.txt”

Ejecución del simulador

Para ejecutar el simulador, el usuario debe invocar la siguiente instrucción desde la línea de comandos de DOS (o LINUX):

```
C:\>python pi.py graf1.txt
```

Ello disparara la ejecución del simulador del algoritmo PI, el que recibirá como parámetro el archivo “graf1.txt”. El resultado será la traza de la simulación que se despliega directamente en modo texto, sobre la consola de la computadora. Alternativamente, se puede ejecutar la siguiente instrucción:

```
C:\>python pi.py graf1.txt > pruebaSimulacion.txt
```

La cual produce el mismo resultado, redirigiendo la traza de la simulación hacia un archivo denominado “pruebaSimulacion.txt”. Este puede servir a un analista para estudiar con detalle el resultado de su simulación.

Bibliografía

Índice alfabético