Ricardo Marcelín Jiménez
María Elena Melgar Estrada

Introducción a los algoritmos distribuidos

28 de mayo de 2013

A Calú, Fer y Chrys - Rick

A Israel y Aldahir - Elena

Prólogo

El presente libro comenzó como unas notas que acompañaron al curso de “Algoritmos distribuidos”, ofrecido en la maestría en ciencias y tecnologías de la información, de la UAM Iztapalapa. A lo largo de varios años se ha enriquecido con la retroalimentación ofrecida por los participantes del mismo.

Se trata de un “mapa de ruta” con un conjunto necesario de “paradas” para entrar en contacto con aquellos problemas fundamentales, que todo especialista de la computación distribuida debe conocer. No es un libro acerca de los grandes bloques o componentes de los sistemas distribuidos. En cambio, se consideran lo que podríamos calificar como los pequeños ladrillos, esto es, los problemas elementales que recurrentemente aparecen en la construcción de los sistemas distribuidos.

En la mayoría de los casos, la presentación recurre a la fuente original donde se planteó la solución de cada problema reconocido. A lo largo del libro hemos buscado la claridad y la sencillez de la presentación. Para empezar, desarrollamos un solo estilo de pseudocódigo con el que describimos los algoritmos que se revisan. Este ejercicio es, por sí mismo, una contribución ya que cada autor consultado utiliza sus propias notaciones para describir una solución. Otra característica de la exposición es que puede detenerse en un mismo problema y presentar varias soluciones ordenadas en grado creciente de dificultad.

Aun cuando existen varios modelos con los que se pueden describir las operaciones de un sistema distribuido, en este trabajo nos enfocamos, casi exclusivamente, en el llamado modelo de comunicación asíncrona con intercambio de mensajes.

El libro se desarrolla sobre tres grandes ejes temáticos que corresponden con las tres partes en que se divide el trabajo: espacio, orden y fallas.

1. En la primera parte, se consideran aquellos mecanismos que sirven para recorrer exhaustivamente el grafo formado por el subsistema de comunicaciones, al tiempo que se construye una subgrafo con alguna propiedad requerida.
2. En la segunda parte, se asume la imposibilidad de contar con un reloj global con el que puedan fecharse los eventos que suceden en un sistema, entonces se buscan mecanismos alternativos para construir un orden parcial o total sobre el mismo conjunto de eventos. Visto de otra forma, se busca una forma alternativa de “medir” el tiempo. La noción de causalidad es fundamental para la construcción de las soluciones que se revisan en esta sección.
3. En la tercera y última parte, se considera la posibilidad de que los componentes activos se desvíen de sus especificaciones de operación. Entonces, se presenta el problema de consenso asumiendo diferentes escenarios de fallas y diferentes consideraciones sobre la medición del tiempo. Se revisan las condiciones bajo las que tiene solución, así como el famoso resultado de imposibilidad.

Cada capítulo presenta una tabla con la notación utilizada. Así también, cada capítulo introduce una serie de figuras que sirven para ilustrar los aspectos clave de la solución estudiada. En algunos casos se trata de una especie de "historieta" que describe momentos sucesivos de la ejecución de un algoritmo.

Todos los capítulos cierran con una sección de comentarios donde se describen los escenarios de aplicación del contenido recién expuesto. Si es pertinente, se presenta también algún aspecto del problema que queda pendiente de solución o se sugieren direcciones para mejorar alguna medida de complejidad. En este sentido, se propone también una serie de ejercicios con diferentes grados de dificultad.

Hemos buscado un material autocontenido, por ello mismo decidimos agregar un conjunto de apéndices para complementar el hilo fundamental de la exposición.

En el primer apéndice, incluimos el código de un sencillo simulador de eventos discretos, que escribimos en Python, con el que pueden reproducirse muchas de las situaciones que se exponen a lo largo del curso. En nuestra opinión, el estudio de los algoritmos distribuidos tiene que lograr un balance entre dos formas de abordarlos, por un lado está el análisis de situaciones abstractas. Por otro, necesitamos reconocer los aspectos más finos de cada algoritmo. En esta segunda parte puede resultar de mucha utilidad programar una solución, sin perderse en los detalles de la programación de un sistema real.

El segundo apéndice presenta un conjunto de nociones elementales de la teoría de grafos, las cuales pueden ser útiles en función de los antecedentes académicos de los alumnos.

Finalmente, el tercer apéndice presenta los conceptos mínimos de complejidad computacional, requeridos a lo largo del curso.

En un sistema distribuido, existe una incertidumbre inherente al fechado de sus eventos. Si entendemos que la construcción del conocimiento ocurre sobre un sistema distribuido, entonces debemos aceptar que formamos parte de un sistema con estas características. Por lo anterior, la incertidumbre nos impide ser categóricos, sin embargo, conjeturamos con orgullo que este es el primer libro en español que se escribe sobre el tema.

Iztapalapa, marzo de 2012 *RMJ & EME*

Agradecimientos

Agradecemos el apoyo y colaboración de nuestros compañeros: alumnos y profesores, quienes nos ofrecieron generosamente sus opiniones y puntos de vista para mejorar el contenido de esta obra. En particular reconocemos las sugerencias de Sergio Rajsbaum y Elizabeth Pérez cuyas observaciones nos ayudaron a mejorar la presentación del material en algunas de las secciones del libro. Por otro lado, las varias generaciones de alumnos que han tomado el curso de "Algoritmos distribuidos" en la UAM-I, nos han acompañado en la depuración de este material. En algún momento alguno de ellos nos sugirió que intentaramos reescribir nuestro simulador de eventos discretos, que inicialmente estaba codificado en C++. Fue una agradable sorpresa descubrir todo el potencial de Python. Ricardo agradece también a todos sus tesistas que trabajaron con la vieja versión del simulador.

Contenido

Lista de figuras

Lista de tablas

Lista de algoritmos

Lista de acrónimos

AGM	Árbol Generador Mínimo
AMF	Arista Saliente Mínima
BFS	Búsqueda Primero en Amplitud
DF	Detector de Fallas
DFS	Búsqueda Primero en Profundidad
FIFO	Primero en entrar, primero en salir
I/O	Entrada / Salida
P2P	Par a Par
PI	Propagación de Información
PIF	Propagación con Retroalimentación
SED	Simulación de Eventos Discretos

1 Presentación

1.1. Introducción

En términos generales puede pensarse en un sistema distribuido como en una colección heterogénea de componentes de hardware, software y datos, interconectados por algún tipo de infraestructura de comunicaciones, mediante la que colaboran para ofrecer un servicio relacionado con el manejo de la información. Por ejemplo, transacciones sobre una base de datos distribuida, cálculos científicos, control de procesos en tiempo en real, etc.

Las dificultades de su construcción tienen su origen en las limitaciones de los componentes con los que se integra un sistema distribuido, así como en la diversidad de características que deben reunirse y coordinarse en un todo funcional [47]. Se desea que un sistema distribuido garantice la calidad de sus servicios, a pesar de que un cierto número de componentes falle o se aleje de sus especificaciones de operación.

Antes de intentar definir qué es un algoritmo distribuido, conviene reflexionar sobre los fines del sistema en el que va a desplegarse la ejecución de dicho procedimiento. Como hemos dicho, el sistema tiene como propósito último proporcionar un servicio, sustentado mediante la participación de un cierto número de componentes de cómputo (o entidades de procesamiento), que en conjunto resuelven las tareas elementales en las que puede descomponerse el servicio. Una de estas tareas puede ser, por ejemplo, "garantizar que cada una de las entidades activas del sistema pueda acceder a un recurso compartido de lecto-escritura, pero solo de una a la vez". Este problema se conoce como el problema de exclusión mutua.

La investigación en el campo de la computación distribuida comienza cuando se reconoce un problema de significación práctica y se construye una versión abstracta, que puede abordarse mediante tratamiento matemático. En este momento pasamos del sistema real, al modelo que lo representa. Luego, se propone un algoritmo que resuelve el problema, se le describe y se demuestra que funciona, de acuerdo con las condiciones del modelo. Enseguida se analiza la complejidad de la solución de acuerdo con varias medidas de costo. No solo se desea encontrar un algoritmo que funcione, se espera que lo haga con eficiencia, esto es, utilizando el mínimo de recursos. Por otro lado, el estudio del problema puede conducir a resultados de imposibilidad, así como al establecimiento de cotas en el costo de una solución.

1.2. Sobre las ventajas de trabajar con modelos

El modelo es un ejercicio de abstracción que, desde la perspectiva de quien lo plantea (y también de quien lo usa), recoge las propiedades sobresalientes del sistema real que se busca describir. Entre las ventajas de usar un modelo de sistema distribuido podemos destacar que ofrece escenarios o condiciones de ejecución independientes de la tecnología. De esta forma es posible valorar el desempeño de un procedimiento o una operación y compararlos, en condiciones justas, con otros de su tipo. Podemos decir, por ejemplo, que "bajo el modelo de tipo M, el algoritmo D1 es más eficiente que el algoritmo D2, aun cuando ambos producen el mismo resultado". Sabemos entonces que esta es una propiedad intrínseca del algoritmo, que no depende de su implementación. Por otro lado, el modelo también es capaz de capturar y revelar las limitaciones propias de la construcción de un sistema. Podemos afirmar, por ejemplo, que "el problema C no tiene solución en el modelo de tipo A". Esto quiere decir que todo sistema que opere bajo los supuestos del modelo tipo A, tendrá la misma limitación independientemente de la tecnología que le dé sustento.

La construcción de modelos, por otra parte, establece criterios de clasificación que permiten sistematizar el estudio de los sistemas distribuidos. En esta área ocurre que el modelo describe algunas de las "dimensiones" en que se desarrollan las acciones del sistema. Por tanto, pueden presentarse modelos que se complementan y describen el espacio completo de posibilidades del sistema. Por otro lado, en ocasiones es necesario analizar un mismo sistema distribuido desde diversos puntos de vista y, por tanto, mediante distintos criterios.

Las entidades que componen un sistema distribuido realizan en conjunto una serie de tareas en común, para las que es necesario intercambiar información en el espacio y el tiempo. También se sabe que dichas entidades están expuestas a contingencias que pueden manifestarse de diferentes formas. En suma, un modelo debe ser capaz de describir la dinámica del sistema. Esto es, las posibilidades de interacción de cada una de las partes o componentes activos. En consecuencia, el planteamiento de un modelo comienza por reconocer los elementos encargados del procesamiento de la información. Enseguida deben establecerse los mecanismos que hacen posible el intercambio de información entre estos elementos. Ello implica caracterizar los medios con que se sustenta la comunicación. Por otra parte, modelar al sistema implica también, definir un orden en las interacciones. Finalmente, tratándose de describir el comportamiento en el largo plazo, también es importante considerar la posibilidad de que

algún componente se desvíe de sus especificaciones y muestre un comportamiento no deseado. Visto de otra forma, deben definirse las fallas que pueden presentarse.

1.2.1. La comunicación

Si se atiende a los mecanismos que sustentan el intercambio de información, se puede dividir a los sistemas distribuidos en dos grandes categorías [32, 36]: con *intercambio de mensajes* y *memoria compartida*.

En los modelos con intercambio de mensajes se destaca el papel de la red de comunicaciones. Ésta se describe como un grafo no dirigido cuyos vértices representan a los procesadores de la red, donde residen las entidades activas, y cuyas aristas representan los canales bidireccionales de comunicación que conectan a las máquinas y a través de las que se intercambian mensajes. Los procesadores no cuentan con un recurso de almacenamiento en común, y cada uno de ellos tiene una identidad única. En cada nodo se procesan los mensajes que se reciben desde los vecinos, se realiza un cálculo local y se envían mensajes a algunos vecinos. Todos los mensajes tienen una longitud acotada y solo pueden transportar una cantidad limitada de información. Cada mensaje transmitido se recibe al cabo de un tiempo finito.

En los modelos de memoria compartida, los procesos se comunican efectuando operaciones de almacenamiento y recuperación de la información sobre un espacio común organizado en objetos compartidos, como registros de lectura/escritura, colas o registros con operaciones atómicas de tipo "test&set" [27].

En general, ambas clases de modelos se utilizan para estudiar aspectos distintos de un sistema. El enfoque de mensajes se presta para describir situaciones de intercambio de información entre parejas de nodos. En tanto, en la memoria compartida, se observa la posibilidad de soportar algún tipo de operación de "difusión" de la información. Existe, sin embargo, ciertas condiciones de equivalencia entre ambos modelos de comunicación.

1.2.2. El tiempo

De acuerdo con sus capacidades para medir el tiempo, los sistemas distribuidos pueden clasificarse en *síncronos* y *asíncronos*.

Un sistema síncrono opera en *rondas*. En cada una, los procesadores ejecutan una operación local y envían un mensaje a sus vecinos. Una unidad de tiempo después, todos los mensajes llegan simultáneamente y comienza la siguiente ronda para reiniciar el ciclo de ejecución hasta alcanzar la solución de una tarea determinada.

La sincronía es un atributo de los procesos y la comunicación. Se conocen variantes de esta propiedad:

- cuando existe, y se conoce, un límite superior en el retardo de los mensajes.
- cuando cada procesador tiene un reloj local tal que la derivada de su error con respecto al tiempo está acotada.

- cuando existe una cota máxima y una mínima en el tiempo que un procesador requiere para completar un paso de ejecución.

En contraste, un sistema asíncrono no tiene límites en el retardo de sus mensajes, la deriva de su reloj o el tiempo para ejecutar un paso. Esto es, decir que un sistema es asíncrono significa que no se hace ninguna suposición relativa a sus tiempos de ejecución. También existen modelos parcialmente síncronos, que buscan capturar algunas propiedades de los sistemas reales.

1.2.3. Las fallas

Por otro lado, conforme crece el número de los componentes del sistema distribuido, es claro que también aumenta el riesgo de falla y la vulnerabilidad del sistema. El tipo de fallas que pueden presentarse varía desde las más benignas, como cuando un componente cesa sus operaciones, hasta fallas muy severas como cuando un componente muestra un comportamiento arbitrario, en perjuicio de aquellos con los que se comunica.

1.3. La red asíncrona con intercambio de mensajes

En los capítulos que se presentan en este libro se aborda un conjunto representativo de problemas de cómputo distribuido que admiten solución bajo el modelo de *comunicación asíncrona con intercambio de mensajes*. En realidad se trata de dos modelos complementarios: uno de comunicación (con intercambio de mensajes) y otro de tiempo (asíncrono o sin cotas de duración).

El modelo de comunicación asíncrona con intercambio de mensajes considera un conjunto de entidades activas, denominadas *procesos*, que se ejecutan sobre diferentes *sitios* o *plazas*. Estas entidades se relacionan al intercambiar mensajes a través de los *canales* de comunicación que las unen. Un canal, que comunica a dos procesos, funciona como una estructura funciona como una estructura de datos sobre la que el transmisor guarda su mensaje, sin esperar respuesta, mientras el receptor lo recoge, de acuerdo con una política de servicio que generalmente corresponde al tipo fila o FIFO (*first in first out*). Generalmente, cada sitio tiene asociado un identificador que lo distingue del resto de los componentes con los que intercambia información.

La red de comunicaciones que comprende estos sitios y canales es caracterizada por medio de un grafo $G = (V,E)$, donde V es el conjunto de sitios (también llamados nodos o vértices) y E es el conjunto de enlaces (también llamados aristas o canales) con los que se unen las parejas de sitios entre los que existe una comunicación bidireccional (véase la figura 1.1). Para una revisión de los conceptos relacionados con la teoría de grafos sugerimos que se consulte el apéndice B. Se asume que este grafo es conexo y está compuesto de n vértices y m aristas. En lo sucesivo se utilizan como sinónimos los términos: vértice y nodo. De manera semejante se emplean los términos: arista y enlace. En todos los casos que se revisan se asume que existe un proceso emplazado en cada nodo o vértice. Cada proceso es capaz de reconocer el enlace por donde recibe un mensaje, aun cuando desconozca la identidad del transmisor. Se garantiza que los enlaces transportan sin pérdida los mensajes que manejan en su interior, con retardo finito, pero

impredecible. Este tipo de enlace lo denominaremos *canal asíncrono sin pérdida*. Finalmente, se da por sentado que los mensajes tienen una longitud acotada [34].

En suma, y a menos que se establezca otra cosa, en el modelo de sistema distribuido con el que trabajaremos, se supone un conjunto finito de procesos $\Pi = \{p_1, \ldots, p_n\}$ *que se comunican enviando y recibiendo mensajes de longitud acotada a través de canales. Cada pareja de procesos* p_i *y* p_j *se conecta por un canal asíncrono sin pérdida, denotado como* (i, j)*. Asimismo, se asume que los procesos no comparten ningún tipo de memoria común, ni disponen de una de referencia universal de tiempo [32].*

Para simplificar la notación, y cuando el contexto lo permita, nos referimos a un proceso p_i usando el subíndice i, con el que aparece en Π. Lo anterior implica que este subíndice le confiere identidad. En contraste, si la exposición requiere enfatizar que una acción recae sobre un proceso arbitrario, denotamos a este por una letra, como p o q.

1.4. La construcción colectiva de una solución

El término algoritmo distribuido se aplica a un conjunto de procedimientos concurrentes usados para un amplio espectro de aplicaciones. Originalmente se aceptaban bajo esta definición solo a aquellos algoritmos diseñados para ejecutarse en un conjunto de procesadores repartidos sobre una extensa zona geográfica pero, con el tiempo, su acepción se ha abierto para incluir programas que se ejecutan sobre una red de área local, e incluso sobre un sistema de multiprocesadores con memoria compartida. De esta forma, puede entendérsele como un procedimiento realizado en conjunto por una colección de entidades autónomas que intercambian información para la realización de una tarea común.

El trabajo en algoritmos distribuidos, se distingue por el alto grado de incertidumbre y la independencia en las operaciones con que cada entidad de procesamiento contribuye a la solución de un problema. En todo escenario de ejecución existirán factores impredecibles o desconocidos que vuelven muy complicado el análisis del algoritmo, aun cuando su código sea breve y parezca simple. Incluso para una misma entrada de datos, pueden presentarse condiciones de entrelazado de las operaciones que pueden llevar a resultados diferentes.

Cada una de las tareas elementales del sistema da lugar a un problema que requiere solución. Se necesita por tanto, una forma eficaz de expresar la solución o, mejor dicho, el procedimiento que lo resuelve. En esencia, se trata de describir las operaciones que deberán realizar las entidades de proce-

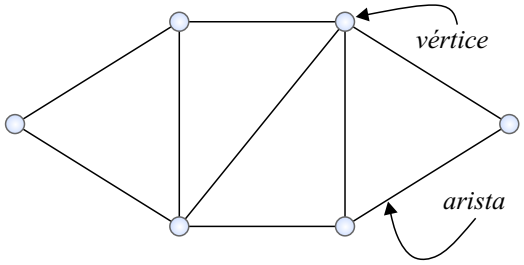


Figura 1.1: Ejemplo de un grafo

samiento, para alcanzar la solución. Parecería sencillo plantear una especie de procedimiento lineal que pueda repetirse siempre de la misma forma y que lleve al resultado deseado. Sin embargo, entre las situaciones peculiares que se presentan en el estudio de los sistemas distribuidos, se observa la posibilidad de un gran número de escenarios de ejecución y la complejidad de su comportamiento externo [51]. A diferencia de los programas secuenciales tradicionales, en los sistemas distribuidos pueden haber muchas ejecuciones posibles y muchas salidas correctas para una misma entrada, como consecuencia de factores como los retardos en las líneas de comunicación, las velocidades de las computadoras, entre otros.

Se trata entonces de reconocer las particularidades de cada nueva edición del problema y, por tanto, plantear un procedimiento reactivo. Lo anterior significa que cada entidad participante debe ser capaz de reconocer sus condiciones particulares, y con base en ellas efectuar una acción local que contribuya a la construcción de la solución global. Se infiere, que cada entidad requiere una memoria propia que le permita caracterizar el estado en que se encuentra y, en función del mismo, reaccionar a los *eventos* de intercambio de información que se presentan en su interacción con el resto del sistema.

Nos encontramos, frente a nuestra primera aproximación al concepto de algoritmo distribuido. Se trata de un procedimiento local, que construye una solución de conjunto o global. Decimos que es local, porque se refiere a las acciones que debe efectuar cada entidad activa en su contexto particular. Sabemos que es un procedimiento reactivo, porque la entidad debe reaccionar a los eventos que se le presentan, en función de su estado, el cual da cuenta de su condición particular. Parece natural pensar que la descripción formal de este procedimiento puede basarse en el uso de algún tipo de autómata. Esta clase de mecanismos utiliza un conjunto de estados, que sirven para registrar la evolución del procedimiento. En función de estos pueden indicarse las acciones locales, así como la transmisión de información, que la entidad debe realizar, en respuesta a sus eventos internos y la información que pueda llegarle desde el exterior.

Informalmente, un algoritmo distribuido es un programa de acciones que se encuentra replicado en cada uno de los sitios que participan en el sistema y a partir del cual se da una secuencia de interacciones entre estos. Por ello se dice que *todos los procesos que intervienen son simétricos*. Es decir, que cada componente se encarga de ejecutar una versión del mismo conjunto de instrucciones a partir de sus circunstancias particulares como son: su identidad, los sitios vecinos con los que interactúa, el orden relativo en que recibe sus mensajes, etc. Normalmente se supone que ninguna de las entidades activas dispone de un conocimiento total o global sobre el sistema al que pertenece. Esto es, sus reacciones obedecen al conocimiento parcial que dispone y que corresponde con su entorno local.

En vista del papel central que juegan las comunicaciones en la ejecución de un algoritmo distribuido, las relaciones de causalidad entre eventos, el conocimiento local y la inferencia, son nociones fundamentales y propias de los ambientes distribuidos.

Cada entidad autónoma que corre su versión local del algoritmo distribuido puede caracterizarse como un conjunto finito de estados, un conjunto finito de eventos de comunicación y un conjunto finito de transiciones atómicas entre estados, disparadas mediante eventos de comunicación [34, 36]. Esto es:

$$(s,e) \rightarrow (s',e'), \tag{1.1}$$

si ocurre un evento e, cuando la entidad se encuentra en su estado s, entonces la entidad transita hasta el estado s' y, si así lo requiere, genera un evento e'.

1.5. Las propiedades de un algoritmo

Luego de construir una descripción precisa de un algoritmo distribuido, hace falta considerar dos aspectos sin los cuales se considera incompleto el diseño: primero, hace falta revisar que el algoritmo sea correcto, esto es, que haga exactamente lo que se espera de él. Segundo, hace falta evaluar los recursos implicados en la solución del problema. No basta saber que el algoritmo resuelve el problema, si para ello requiere una cantidad infinita de recursos. La economía también cuenta.

Para demostrar que un algoritmo es correcto deben inferirse sus propiedades y describirlas en forma de garantías. Sin embargo, ya hemos mencionado varias veces que el problema con los sistemas distribuidos es que las entidades activas pueden interactuar en diferentes secuencias de eventos, que aun así llevan a una solución correcta del problema. Esto es, ocurre un entrelazado de eventos. De todas las combinaciones de eventos que potencialmente pueden presentarse, el análisis de un algoritmo requiere caracterizar aquellas secuencias que pueden presentarse, así como aquellas que nunca pueden ocurrir. Con base en las primeras es posible inferir el avance en la construcción de la solución global. Por ejemplo, refiriéndonos al problema de exclusión muta, es posible establecer que "en cualquier secuencia en la que se observa que una entidad solicita acceder al recurso compartido, se observa también que al cabo de un tiempo finito la entidad recibe dicho recurso". Por su parte, las secuencias imposibles sirven para señalar el conjunto de condiciones que se busca evitar a toda costa. Por ejemplo, se puede decir que "no puede darse una secuencia en la que una entidad reciba el control de un recurso y no lo vuelva a liberar".

Por otra parte, el objetivo del análisis de la complejidad es medir la cantidad de los recursos empleados por un algoritmo distribuido durante su ejecución. A través de esta cuantificación es como se evalúa su desempeño y puede estimarse su costo de operación en términos de tiempo de cálculo, retardo de comunicación, número de mensajes (o bits) intercambiados, requerimientos de almacenamiento, etc. Sin embargo, en el contexto del modelo de intercambio de mensajes, es común suponer que, frente a la duración de las operaciones de comunicaciones, es despreciable el tiempo de cálculo local, así como el retardo en las filas de espera que se forman frente a cada proceso. Bajo estas premisas, las medidas de complejidad de un algoritmo distribuido se establecen preferentemente en función de la cantidad de información intercambiada o el tiempo de transmisión de mensajes.

Una medida de complejidad evalúa el costo de un recurso como una función de las condiciones de entrada del sistema (número de procesadores presentes, capacidades de comunicación, etc.). Además se busca expresar esta medida como una figura de desempeño independiente de la tecnología sobre la que se implanta el algoritmo; con ello se consigue una forma normalizada de comparar la operación de varios algoritmos. Para esto se utiliza la notación asintótica de orden, que expresa la forma en que crece la cantidad de recursos empleados, conforme crece el tamaño de la entrada del problema [24, 34, 50].

Se dice, por ejemplo, que la *complejidad en mensajes* de un algoritmo es $f(n,m)$ si el número total de mensajes enviados es $f(n,m)$, cuando el algoritmo corre sobre un sistema arbitrario cuyo grafo subyacente $G = (V,E)$ tiene $n = |V|$ vértices y $m = |E|$ aristas. Se sigue que la complejidad en mensajes es $O(g(n,m))$, si $f(n,m) = O(g(n,m))$. Es importante observar en este punto que el diseño de un algoritmo supone un modelo de sistema sobre el cual se espera desplegar su ejecución.

Para evaluar la duración de un algoritmo distribuido es necesario incluir en el modelo alguna noción de tiempo, por ejemplo, suponer que:

1. El tiempo de procesamiento local es despreciable.

2. El tiempo que toma un mensaje en llegar a su destino es, a lo más, una unidad.

De este modo, la *complejidad en tiempo* de un algoritmo es $f(n,m)$ si su tiempo total de ejecución nunca excede $f(n,m)$ unidades sobre cualquier grafo subyacente con n vértices y m aristas. De igual forma puede aplicarse la notación asintótica de orden para este caso. Hay que resaltar que las suposiciones (1) y (2) se hacen solamente para analizar la complejidad de tiempo en un supuesto escenario normalizado; sin embargo, para determinar si un algoritmo es correcto se precisa considerar todas sus posibles ejecuciones. Sugerimos al lector referirse al apéndice C para revisar los conceptos utilizados en la descripción de la complejidad de los algoritmos.

Como en el caso de los algoritmos centralizados, los algoritmos distribuidos resuelven problemas que se agrupan en clases de complejidad. Esto significa, entre otras cosas, que la solución de un problema puede transformarse en la solución de otro de la misma clase.

1.6. Guía de lectura

El resto de este libro se divide en tres partes denominadas: *espacio*, *orden* y *fallas*, respectivamente. Todos los algoritmos expuestos en la primera parte resuelven una clase especial de problemas en los que se construye una solución mientras se exploran los vértices del grafo subyacente.

En el capítulo *Búsqueda y propagación*, se revisan tres algoritmos, el primero de ellos realiza una exploración secuencial del grafo de comunicaciones, mientras que los dos últimos sirven para transmitir un mensaje a todos los nodos del grafo, en el menor tiempo posible.

En el capítulo *Elección*, se resuelve el problema de designar a un proceso único y reconocido universalmente por todos los participantes del sistema.

En el capítulo *árbol generador de peso mínimo*, se aborda uno de los problemas más estudiados de la computación distribuida. Sobre un grafo con aristas ponderadas, que en este caso representa a la red de comunicaciones, se busca la construcción de un árbol generador, tal que la suma de los pesos de sus aristas sea mínima.

En la segunda parte, se presenta una colección de algoritmos que permiten construir un orden sobre los eventos que suceden en un sistema distribuido.

En el capítulo *Relojes lógicos*, se presentan las nociones de historia local, historia global, dependencia causal y ejecución distribuida, entonces se describen tres mecanismos para fechar los eventos de la historia global.

En el capítulo *Estado global*, se describe la noción del mismo nombre, así como los conceptos de corte consistente y corrida, luego se presenta un mecanismo distribuido para construir un corte consistente.

Finalmente, la tercera parte está dedicada por completo al problema de consenso y los modelos de tiempo y fallas, bajo los que es o no posible resolverlo.

En el capítulo *Consenso síncrono*, presentamos una vista panorámica del problema. Luego, nos alejamos por única vez del modelo asíncrono, para mostrar cómo el tiempo es una fuente de información que puede contribuir a la solución de este problema fundamental.

En el capítulo *Detectores de falla*, describimos las condiciones mínimas que garantizan la solución del problema de consenso.

En el capítulo *Imposibilidad del consenso asíncrono*, presentamos este importante resultado de la computación distribuida y revisamos las condiciones del modelo de cómputo asíncrono que son su causa fundamental.

En todos los algoritmos se hace énfasis especial en el costo de las soluciones, así como en las propiedades que garantizan su terminación correcta. La exposición de los temas se ha dispuesto de manera que se sucedan en orden creciente de dificultad para ofrecer una presentación gradual.

Para complementar la exposición del libro, se incluyen tres apéndices. El primero de ellos es un sencillo manual de construcción y operación de un simulador de eventos discretos, con el que pueden estudiarse muchas de las situaciones abordadas a lo largo del curso. El segundo, contiene un conjunto mínimo de definiciones y resultados sobre teoría de grafos. El tercero es una breve explicación sobre la notación de complejidad que usamos para describir el costo de un algoritmo.

Parte I
Espacio

Caminante no hay camino, se hace camino al andar

—Antonio Machado

2

Búsqueda y propagación de información

Resumen La exploración del grafo $G = (V, E)$ que representa a la red de comunicaciones, es una operación frecuente de los sistemas distribuidos. Se utiliza, entre otras razones, para reconocer la estructura subyacente y con ello soportar operaciones relacionadas con la comunicación entre procesos. Si G es conexo entonces, mientras se le recorre, puede construirse un subgrafo que incluya a todos sus vértices y utilice el mínimo número de aristas, esto es, construir un *árbol generador*. En el presente capítulo presentamos un conjunto de algoritmos de recorrido que exploran el grafo mientras construyen este tipo de estructura.

2.1. Introducción

Un problema fundamental de los sistemas distribuidos es la exploración exhaustiva del grafo que subyace en la red de comunicaciones. Con ello es posible reconocer a todos los nodos que lo constituyen. Esta exploración puede realizarse de varias formas alternativas. Por principio, puede realizarse de manera secuencial o de manera paralela. En el primer caso, los nodos se visitan uno tras otro. En el segundo caso, en cambio, se visitan tantos nodos como sea posible, de forma simultánea.

Un producto importante de la exploración es la construcción de un árbol generador. La importancia de éste radica en el hecho de que puede utilizarse en operaciones subsecuentes para minimizar el intercambio de información en el sistema. Por ejemplo, si un proceso requiere enviar un mensaje a todos los participantes del sistema y sabe de la existencia de un árbol generador, entonces solo necesita expedir

el mensaje sobre los enlaces que forman parte de este subgrafo. Así, cada proceso recibirá el mensaje exactamente una vez, requiriéndose para ello una cantidad mínima de recursos.

En este capítulo se revisan tres problemas vinculados con la exploración del grafo de comunicaciones. El primer problema se denomina *búsqueda en profundidad* o *DFS* (Depth First Search), se trata de implementar un recorrido secuencial. En el segundo y tercer problema, conocidos como *propagación simple* y *propagación con retroalimentación*, respectivamente, se trata de comunicar un mensaje a todos los vértices del grafo, utilizando el menor tiempo posible. En todos los casos, el recorrido da lugar a un árbol.

En las soluciones que se presentan, cada proceso carece de información sobre la estructura de la red. Estrictamente, le basta con distinguir el enlace o puerto desde donde recibe cada mensaje. Sin embargo, para los fines de nuestra exposición supondremos que cada participante conoce la identidad de los vecinos con los que comparte un enlace.

Como se recordará, el grafo G, para el cual están pensados los algoritmos, consta de n vértices o nodos, m enlaces, y diámetro D. Definimos el diámetro de G como la distancia máxima entre cualquier par de nodos. Considerando G, el algoritmo de búsqueda en profundidad que se presenta termina en un tiempo $O(m)$ y con un costo en mensajes $O(m)$. Por su parte, los algoritmos de propagación tienen una complejidad en tiempo $O(D)$ y una complejidad en mensajes $O(m)$.

La notación utilizada en este capítulo se presenta en la tabla 2.1.

$G = (V, E)$	Grafo G con conjunto de vértices V y conjunto de aristas E
D	Diámetro de la red
n	Número de nodos en la red
m	Número de enlaces en la red
p	El proceso iniciador

`M`
`DESCUBRE`
`REGRESA`
`RECHAZO`
	Mensajes de los algoritmos

i, j, k	Identificador de nodo/proceso
visitado	Estado local del proceso i
vecinos	Conjunto de nodos adyacentes al nodo i
T	Árbol de peso mínimo
l	Nivel en el árbol T
$w_{i,j}$	Retardo de transmisión en el mensaje enviado desde i a j
sin_visitar, *hijo* *padre*, $N(k)$	Variables locales de los algoritmos

Tabla 2.1: Notación

2.2. Búsqueda en profundidad

En principio, se supone que los procesos participantes ejecutan algoritmos locales idénticos, es decir, que poseen las mismas instrucciones de código secuencial. Todos los procesos efectúan el mismo papel, por lo que se dice que hay una condición inicial de simetría en el conjunto. Un método usual en la

programación distribuida consiste en otorgar un privilegio a uno de los procesos del sistema, con lo que se consigue romper la simetría inicial.

Al comenzar el algoritmo DFS [15], se da por hecho que la simetría del conjunto queda rota cuando se designa a un proceso arbitrario p, que será el encargado de arrancar el algoritmo. Se dice que p pone en circulación una "ficha" o mensaje de activación con el que selecciona a otro proceso al que le otorga el control de la ejecución. Solo aquel que posee la "ficha" se considera activo, por tanto, en cualquier instante durante la ejecución del algoritmo, no hay más que un proceso en estas condiciones.

En cada sitio, un proceso marca el enlace desde el que recibe el mensaje de activación por vez primera y selecciona un nuevo camino por el cual debe reexpedir la "ficha" para iniciar el algoritmo en otra parte del sistema, luego detiene su ejecución local y espera noticias de ese vecino a quien cede la acción. Es decir, del proceso con el que comparte el enlace por el que viaja la "ficha".

Cuando un proceso no puede retransmitir la "ficha", ya sea porque no tiene vecinos que visitar o porque ha visitado a todos cuantos podía (que en esencia es lo mismo), entonces envía un nuevo mensaje a través del enlace que tiene marcado, para devolver el control de las operaciones al proceso que le activo por primera vez, al que denomina *padre*; en este momento da por terminada la ejecución local de su algoritmo. En el otro extremo del enlace, el receptor designa como su *hijo* al vecino que le ha devuelto a la acción. Cada vez que esto sucede el proceso intenta reexpedir la "ficha" para activar a otro vecino; si lo consigue detiene su ejecución nuevamente hasta recibir respuesta, en otro caso, regresa el control a su propio padre y termina.

Hay ocasiones en que un proceso recibe la "ficha" por un enlace distinto del original, desde alguno de sus vecinos que ignora sus condiciones locales. El receptor debe responder con otro mensaje donde informe el estado de su ejecución y la imposibilidad de que aquel lo designe como su hijo, puesto que ya hubo otro que lo activó con anterioridad.

En algún momento, la actividad regresa a p y, si no hay más caminos que explorar, entonces aquí se acaba por completo la ejecución. Al terminar, cada proceso está en posibilidad de distinguir a los enlaces que lo comunican con su padre y sus hijos. De esta forma, se han reconocido todos los vértices del grafo asociado al sistema y atravesado todas sus aristas. La exploración comienza y termina en el vértice que corresponde a p y que será la raíz del árbol DFS producido. Sus vértices representan a los procesos de la red y sus aristas, los enlaces que cada proceso ha marcado.

El pseudocódigo se presenta en el algoritmo 2.1. Observese cómo el mismo algoritmo se ejecuta en cada proceso que participa en el sistema, las únicas diferencias (de carácter local) son los identificadores de los procesos. Los procesos que participan en el algoritmo se comunican entre sí intercambiando los siguientes mensajes:

`DESCUBRE`: Llega a un proceso cuando se le visita por primera vez (esta es la "ficha" que desciende el árbol, mientras lo construye).

`REGRESA`: Devuelve el centro de actividad al proceso padre (esta es la "ficha" que remonta el árbol construido).

`RECHAZO`: Respuesta de un proceso al que se visita más de una vez.

Por otro lado, además de su propio identificador i, cada proceso a cargo del algoritmo mantiene una colección local de variables:

Algoritmo 2.1: Algoritmo DFS de Cheung en el nodo i

```
1  al recibir DESCUBRE desde j efectúa
2      sin_visitar ← sin_visitar − {j}
3      si (visitado) entonces envía RECHAZO a j
4      otro
5          visitado ← verdadero, padre ← j
6          continua_exploración()
7  al recibir REGRESA o RECHAZO desde j efectúa
8      continua_exploración()
9  procedimiento continua_exploración()
10     si (existe k en sin_visitar) entonces
11         envía DESCUBRE a k
12         sin_visitar ← sin_visitar − {k}
13     otro
14         si (padre ≠ i) entonces envía REGRESA a padre
15         termina
```

visitado: Es un valor booleano, inicialmente *falso*. Cambia a *verdadero* si se recibió el mensaje `DESCUBRE`.

padre: El nodo del que i recibe `DESCUBRE` por primera vez. Inicialmente, es igual a i.

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

sin_visitar: El conjunto de vecinos de i a los que aún no se les visita. Inicialmente, es igual a *vecinos*.

En las figuras 2.1(a) y 2.1(b) puede observarse un ejemplo de ejecución del algoritmo DFS, los números que acompañan a las flechas representan el orden de secuencia del evento asociado a la transmisión de un mensaje.

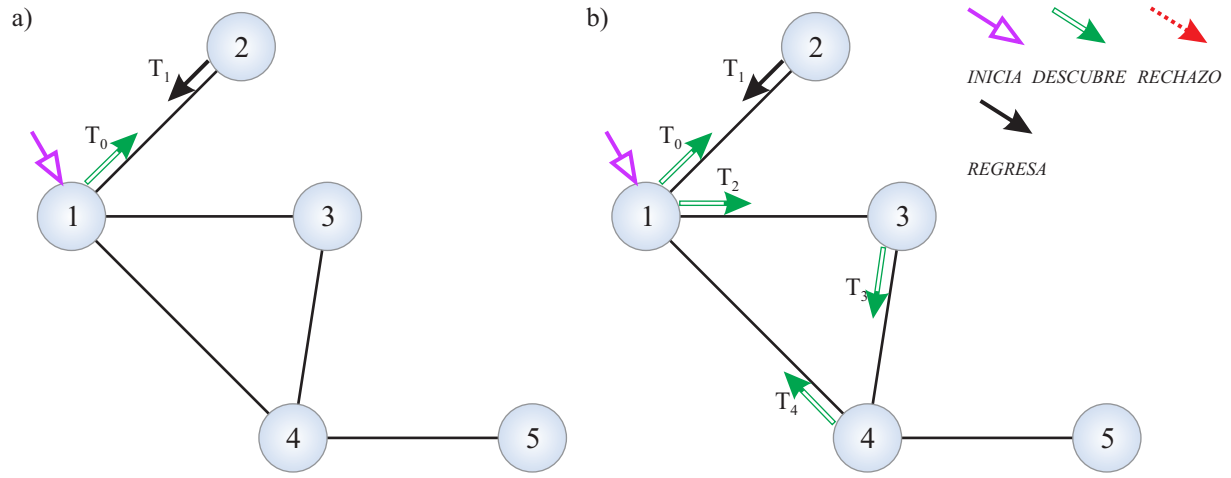
Propiedades del algoritmo DFS

Ahora debemos analizar el algoritmo propuesto con tres objetivos en mente: i) demostrar que termina, ii) que al terminar produce un subgrafo de G que tiene la estructura de un árbol generador, iii) revisar cuál es la cantidad de recursos implicados en esta construcción.

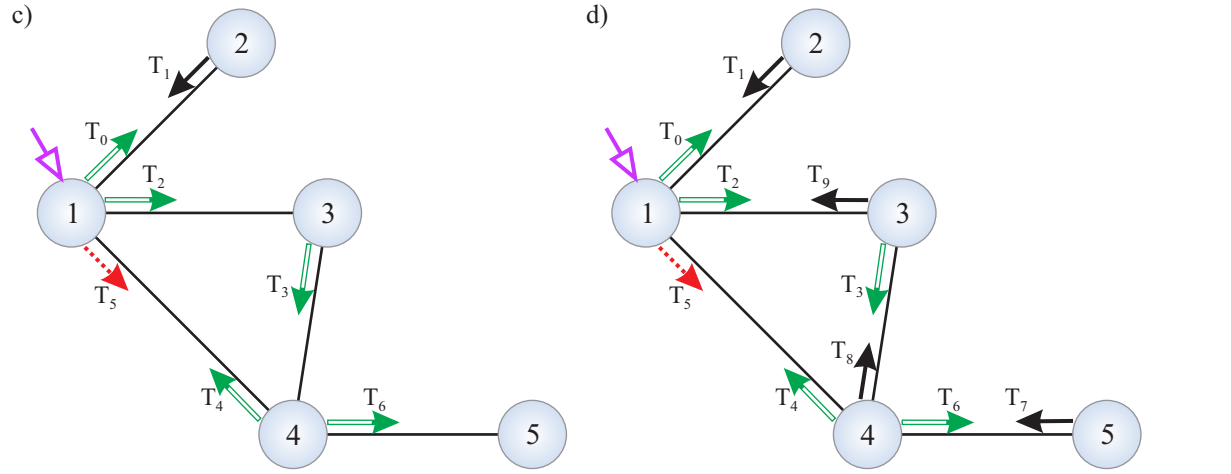
Por principio, queremos inferir que el recorrido que se realiza es exhaustivo, por tanto, afirmamos que

Lema 2.1. *Cada nodo de G se etiqueta como visitado una y exactamente una vez.*

Demostración. Sea $p,\ldots,j,i$ la trayectoria que conecta a p con i. Sabemos que esta trayectoria existe porque G es conexa. Este resultado implica que cada nodo i diferente de p reconoce a un solo nodo j como el que le precede en dicha trayectoria. Se trata de aquel del que recibe el mensaje `DESCUBRE` por primera vez, y al que etiqueta como *padre*, luego de lo cual cambia su estado a *visitado*. Es posible que i reciba posteriormente el mismo mensaje desde otra fuente, en cuyo caso devuelve un mensaje `RECHAZO`, sin cambiar de estado. □



(a) La estampilla de tiempo T_i sugiere un instante de tiempo en cual se puede generar el mensaje. En (a) el nodo 1 despierta espontáneamente, cambia su estado a *visitado*, transfiere el control de operaciones al nodo 2 al transmitirle el mensaje `DESCUBRE` en T_0, y queda en espera. Tras la recepción del mensaje `DESCUBRE`, el nodo 2 marca al nodo 1 como su padre, le devuelve el control a través de un mensaje `REGRESA` en T_1, y concluye su ejecución. En (b) el nodo 1 retoma el control de operaciones y continua con la exploración al expedir un mensaje `DESCUBRE` (en T_2) a su siguiente vecino en *sin_visitar* (i.e., el nodo 3). Tras la recepción del mensaje `DESCUBRE`, el nodo 3 marca al nodo 1 como su padre, y, a diferencia del nodo 2 quien devolvió inmediatamente el control a su padre ya que no tenía vecinos a quienes visitar, el nodo 3 pasa el control al nodo 4 al enviarle un mensaje `DESCUBRE` en T_3. De forma análoga, el nodo 4 marca al nodo 3 como su padre, y pasa el control al nodo 1 al transmitirle en T_4 un mensaje `DESCUBRE`.



(b) En (c), puesto que el estado del nodo 1 es *visitado*, éste elimina al nodo 4 de sus vecinos *sin_visitar* y le devuelve el control de operaciones al enviarle un mensaje de `RECHAZO` en T_5. El nodo 4 continua la exploración y pasa el control al nodo 5 al expedir en T_6 el mensaje `DESCUBRE`. En (d), dado que el nodo 5 no tiene vecinos a quienes visitar, éste devuelve en T_7 el control al nodo 4 a través de un mensaje `REGRESA`, y concluye su ejecución. De forma análoga, el nodo 4 y 3 regresan (a través de un mensaje `REGRESA`) el control de operaciones a sus padres en T_8 y T_9 respectivamente, y concluyen su ejecución. Tras la recepción del mensaje `REGRESA`, el nodo 1 retoma el control, termina la exploración y concluye su ejecución finalizando así el algoritmo DFS.

Figura 2.1: Ejemplo de ejecución del algoritmo DFS de Cheung

Enseguida, vamos a razonar cómo es que la "ficha" se recibe y se devuelve en cada nodo visitado, de donde

Lema 2.2. *El algoritmo termina*

Demostración. Cada vez que un nodo *i*, diferente de *p*, invoca al procedimiento *continua_exploración*(), éste intenta enviar el mensaje `DESCUBRE` a un nodo que forma parte del conjunto *sin_visitar*. Cuando el conjunto queda vacío, entonces devuelve un mensaje `REGRESA` a *padre* y da por terminado su trabajo local. Lo anterior significa que no queda una trayectoria sin explorar, que comience en *p* y pase por *i*.

Por otra parte, cuando *p* invoca al procedimiento *continua_exploración*(), si éste determina que no queda un vecino en *sin_visitar*, entonces significa que han sido exploradas todas las trayectorias posibles y *p* da por terminado el trabajo a nivel global. □

Ahora vamos a razonar sobre las propiedades del subgrafo inducido durante el recorrido de la "ficha".

Lema 2.3. *En cualquier momento de la ejecución del algoritmo, las aristas* (*padre, i*)*, tal que i es diferente de p, forman un subgrafo acíclico.*

Demostración. La relación de descendencia se establece por el orden de designación. Es decir que, un ancestro se reconoce, e incorpora al algoritmo, primero que cualquiera de sus descendientes. Cuando un nodo intenta "descubrir" a un ancestro, éste contesta con un mensaje `RECHAZO`. Con ello se evita que un nodo pueda reconocer como su hijo a otro que ya fue visitado. De esta forma se descarta la formación de una secuencia de aristas que formen un camino cerrado (ciclo). □

Extendemos el resultado anterior para concluir que, al terminar el algoritmo

Corolario 2.1. *La estructura resultante que subyace es un árbol generador.*

Demostración. Por el lema 2.1, sabemos que cada nodo en *G* es incorporado a la construcción. Esto es existe una trayectoria entre *p* e *i*. Por el lema 2.3 las aristas incorporadas forman un grafo acíclico. Tenemos entonces que se ha construido un grafo conexo y sin ciclos, i.e. un árbol. □

Por último, en esta parte de nuestro análisis vamos a revisar la complejidad del algoritmo presentado. Primeramente, debemos establecer el número de mensajes intercambiados.

Lema 2.4. *Por cada enlace se envía exactamente un mensaje en cada dirección.*

Demostración. Por un enlace se envía un mensaje `DESCUBRE` exactamente una vez, y en un solo sentido, porque el canal deja de servir para estos efectos, puesto que el nodo al que se transmite el mensaje se elimina de la lista *sin_visitar*.

En el otro sentido, el nodo receptor devuelve sobre el mismo enlace un mensaje `RECHAZO` o `REGRESA`, según se trate de un proceso que ya fue visitado ó sea la primera vez, respectivamente. En ambos casos el nodo que al que se responde se elimina también de la lista *sin_visitar*. □

Finalmente, vamos a revisar el tiempo que implica la solución de nuestro problema de recorrido secuencial.

Algoritmo 2.2: Algoritmo PI de Segall en el nodo i

```
1 al recibir M desde j efectúa
2     si (visitado = falso) entonces
3         padre ← j
4         visitado ← verdadero
5         envía M a todo k en vecinos − {padre}
6         termina
```

Corolario 2.2. *Asumiendo que cada mensaje se entrega con un retardo máximo de una unidad de tiempo, el algoritmo concluye en un plazo* $\leq 2m$ *y con un costo en mensajes* $\leq 2m$.

Demostración. Se sabe que el algoritmo termina hasta que cada enlace ha transportado un mensaje en cada dirección (no necesariamente en instantes consecutivos). Como no existen mensajes simultáneos, cada mensaje que se transmite cuenta por una unidad de tiempo. Por tanto, el algoritmo termina al cabo de $2m = O(m)$ unidades de tiempo y habiendo intercambiado $2m = O(m)$ mensajes. □

2.3. Propagación simple

Considérese una red cuyo grafo subyacente es $G = (V, E)$, en la que existe un proceso iniciador p que debe transmitir un mensaje a los demás sitios del sistema. Para el modelo de red asíncrona de mensajes con el que se trabaja, solo se requiere que G sea conexa, esto es, que exista un camino entre cualquier pareja de vértices.

Sabemos que el algoritmo presentado en la sección anterior, podría transportar el mensaje que el proceso p requiere enviar. Sin embargo, aunque la búsqueda en profundidad garantiza un recorrido exhaustivo sobre el grafo, también es cierto que se trata de un recorrido secuencial, i.e. donde los procesos son visitados de forma consecutiva. Por tanto, este enfoque no garantiza la forma más rápida de comunicar un mensaje hacia todos los nodos en G. Si queremos mejorar el tiempo de transmisión necesitamos pensar en una solución concurrente, es decir una solución en la que el mayor número de copias del mensaje viajen por la red de manera simultánea.

En el algoritmo *PI* [48], el proceso iniciador p envía su mensaje a cada uno de sus vecinos, estos lo reciben y reexpiden por todas sus líneas de comunicación. A su vez, cada proceso que lo reciba efectuará exactamente la misma operación. Un proceso da por terminada su ejecución, tan pronto como 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 p.

El pseudocódigo que ilustra este procedimiento de propagación de información se presenta en el cuadro del algoritmo 2.2. En este algoritmo, los participantes intercambian un solo mensaje, al que denotamos como `M`. Para evitar el uso de un pseudocódigo diferente para describir la operación de p, asumimos que éste es el único proceso que se puede enviar `M` a sí mismo. Esta suposición refleja el caso en que una entidad externa invoca el arranque del algoritmo.

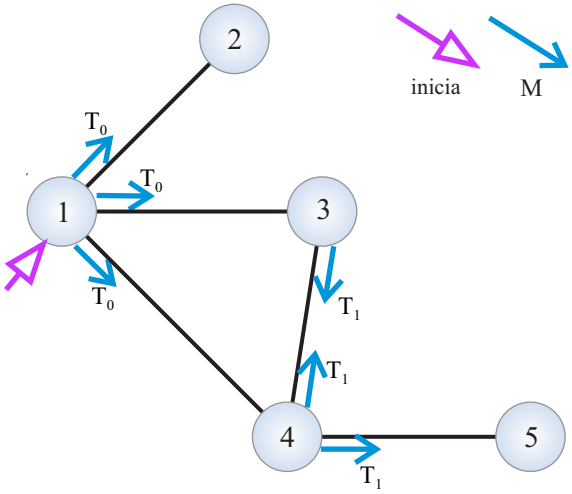


Figura 2.2: En la figura se ilustra la ejecución del algoritmo PI de Segall. En T_0, el nodo 1 transmite el mensaje M a sus vecinos, se marca como visitado y concluye su ejecución. Después de la recepción del mensaje M, los nodos 2, 3, y 4 proceden de la siguiente manera: (i) marcan al nodo 1 como *padre*, (ii) cambian su estado a *visitado*, (iii) transmiten el mensaje M a sus *vecinos* (siempre que $(|vecinos| - \{padre\}) \neq \emptyset$), y (iv) concluyen su ejecución. De forma análoga, el nodo 5 recibe el mensaje M en T_1, marca al nodo 4 como su *padre* y concluye su ejecución al no contar con vecinos a quienes transmitir.

Por su parte, cada proceso mantiene una colección de registros donde almacena su condición local. Estas variables se denominan:

visitado: Es un valor booleano, inicialmente *falso*. Cambia a *verdadero* si ya se recibió un mensaje.
padre: Indica el nodo del que i recibe M por primera vez. Inicialmente es nulo.
vecinos: Representa el conjunto de nodos con los que i comparte una arista.

Propiedades del algoritmo PI

Enseguida, queremos establecer las siguientes propiedades del algoritmo recién presentado: i) que termina al cabo de un tiempo finito, ii) que al hacerlo se ha conseguido difundir el mensaje a todos los procesos conectados al grafo subyacente, iii) que esta es la solución más rápida posible, y iv) que tiene un costo en término de los mensajes intercambiados y la cantidad de tiempo invertido.

Lema 2.5. *Todos los procesos conectados a la red reciben* M *y terminan su ejecución local.*

Demostración. p envía el mensaje M a cada uno de sus vecinos los que, en vista de sus condiciones iniciales, ejecutan la misma secuencia de instrucciones en sus respectivos sitios y por única vez. Supóngase que existe un proceso i conectado a la red que aún no ejecuta esta secuencia de operaciones, mientras que en el camino que lo une con p, todos los que lo preceden ya han completado el algoritmo. Esto quiere decir que tiene un vecino que le ha enviado un mensaje y, como será la primera vez que se le visita, completará también la misma sección de código que los demás. □

Acerca de esta propiedad, es muy importante comentar que el argumento asume implícitamente dos cosas: por un lado, que estamos ante un modelo de comunicación *asíncrona* con intercambio de mensajes, por otra parte que los canales no tienen fallas por pérdida. Vistas en conjunto, ambas suposiciones implican que un mensaje transmitido puede tardar, pero finalmente llegará a su destino.

Lema 2.6. *La propagación de la información es lo más rápida posible, en el siguiente sentido:*
$\forall (i,j) \in E$, *sea* w_{ij} *el retardo de transmisión del mensaje enviado de* i *a* j. *Entonces, el conjunto de aristas* $\{(\textit{ padre, } i), \forall i \in V, i \neq p\}$, *forma un árbol* T *de pesos mínimos para* G.

Demostración. Por inducción sobre la altura del árbol construido. Para el nivel 0, o sea p que se convierte en la raíz, es evidente que se cumple la hipótesis, si se considera que el peso de la arista de p hacia él mismo es 0.

Ahora, supondremos que se cumple para los vértices del árbol hasta el nivel l. Entonces, sea i un proceso de este nivel que envía M hasta un proceso j, que aún no pertenece al árbol, j será hijo de i siempre que no exista un proceso i' (en el mismo nivel que i) que haya reclamado antes la "paternidad" de j, a través de una arista (i', j), tal que $w_{i',j} < w_{i,j}$. Se concluye que los procesos en el nivel $l+1$ también quedan incorporados al árbol, de acuerdo con el criterio de peso mínimo. □

Esta última propiedad nos dice que la trayectoria $p, \ldots, i$ para todo i en T, representa el camino más rápido de todos los caminos en G que conectan a p con i. Es importante tener en cuenta que el camino más corto no es necesariamente el más rápido.

A continuación vamos a revisar la cantidad de recursos implicados en la solución que hemos presentado.

Corolario 2.3. *Cuando el algoritmo concluye, se ha enviado exactamente un mensaje en cada dirección, de cada enlace.*

Demostración. Sabemos por el lema 2.5, que todo proceso que recibe M, reexpide una copia de éste a sus vecinos, incluyendo al proceso de quien recibe por primera vez. Cada enlace une a dos vértices desde cuyos procesos se envía este mensaje. Por tanto, el algoritmo termina habiendo intercambiado $2m = O(m)$ mensajes □

Corolario 2.4. *Asumiendo que todos los mensajes se entregan con un retardo máximo de una unidad de tiempo, el algoritmo concluye en un plazo* $O(D)$, *donde* D *es el diámetro de* G.

Demostración. En un grafo como G, la distancia entre dos vértices es la longitud del camino más corto entre ellos. El diámetro de G, es la mayor distancia entre dos vértices de la misma. Sean i y j esta pareja, de tal forma que iniciamos el algoritmo en i, es decir $p = i$. Por el lema 2.6, y bajo la hipótesis de tiempo que estamos asumiendo, la altura del árbol resultante será la distancia entre i y j, es decir D. □

2.4. Propagación con retroalimentación

El algoritmo *PI* presenta una propiedad que puede resultar inconveniente en algunos casos: el proceso iniciador p sabe que luego de enviar su mensaje, éste se recibirá en cada proceso del sistema asociado

Algoritmo 2.3: Algoritmo PIF de Segall en el nodo i

```
1  al recibir M desde j efectúa
2      N(j) ← 1
3      si (visitado = falso) entonces
4          visitado ← verdadero
5          padre ← j
6          si (vecinos − {padre} ≠ Ø) entonces envía M a todo k en vecinos − {padre}
7      si (N(k) = 1 para todo k en vecinos) entonces
8          si (padre ≠ i) entonces envía M a padre
9          termina
```

con un vértice de G. Sin embargo, p nunca sabrá cuando esto ocurra. Visto de otra forma, se sabe que el algoritmo termina localmente, esto es, en cada proceso participante, pero el iniciador no puede determinar cuando el algoritmo termina globalmente, es decir, en qué momento el último proceso ha recibido el mensaje.

Para corregir esta limitación, en el protocolo *PIF* [48] se incorpora una etapa de retroalimentación. Cada proceso residente en una hoja del árbol generador, al que se refiere el lema 2.6, termina por recibir un mensaje de todos sus vecinos. Cuando ha recibido el último, devuelve un acuse de recibo a su padre. Por su parte, los procesos de los siguientes niveles aguardan este informe por cada una de sus aristas incidentes, incluyendo a aquellas que los conectan con sus hijos. Una vez que han recibido todos los reportes que esperan, devuelven el mismo reconocimiento a sus respectivos padres. Esta información va remontando el árbol hasta arribar a la raíz, donde reside el proceso iniciador. De este modo, p puede saber cuando se ha completado la propagación. El seudocódigo se ilustra en el algoritmo 2.3. Un ejemplo de su ejecución se muestra en la figura 2.3.

En el algoritmo PIF, cada proceso maneja una colección de variables con las que lleva el registro de su estado:

visitado:	Es un valor booleano, inicialmente *falso*. Cambia a *verdadero* si ya se recibió un mensaje.
N(k):	Inicialmente vale 0. Cambia a 1, si ya se recibió `M` desde k.
padre:	Indica el nodo del que i recibe `M` por primera vez. Inicialmente es nulo.
vecinos:	es el conjunto de procesos que comparten una arista con i.

Como en el caso de PI, para utilizar un mismo seudocódigo, asumimos que p es el único proceso que se puede enviar `M` a sí mismo. Esta suposición refleja el caso en que una entidad externa invoca el arranque del algoritmo.

Propiedades del algoritmo PIF

Ahora utilizaremos algunas de los resultados establecidos previamente para demostrar que el algoritmo ofrece la propiedad de terminación global. Luego, analizaremos los costos de esta solución.

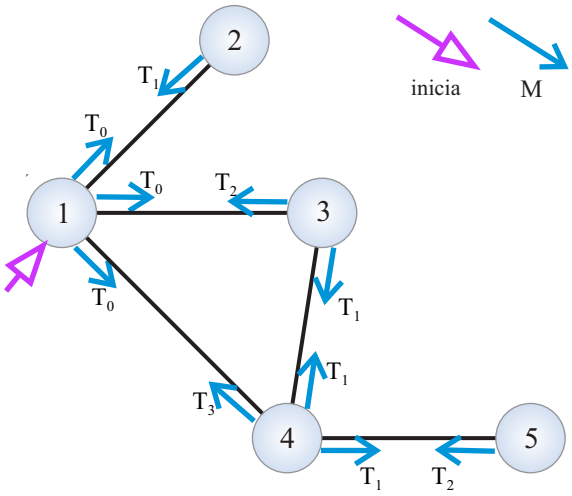


Figura 2.3: En la figura se ilustra la ejecución del algoritmo PIF de Segall. En T_0, el nodo 1 transmite el mensaje M a sus vecinos, se marca como visitado, y queda en espera. Después de la recepción del mensaje M (en T_1), los nodos 2, 3, y 4 proceden de la siguiente manera: (i) marcan al nodo 1 como *padre*, (ii) cambian su estado a *visitado*, y (iii) si $(|vecinos| - \{padre\}) \neq \emptyset$ entonces transmiten el mensaje M a sus *vecinos* y quedan en espera (caso de los nodos 3 y 4). En otro caso, reexpiden M a su padre y concluyen su ejecución (caso del nodo 2). De forma análoga al nodo 2, el nodo 5 recibe el mensaje M en T_1, marca al nodo 4 como su *padre* y concluye su ejecución al no contar con vecinos a quienes enviar el mensaje. Al recibir respuesta de todos sus vecinos, los nodos 3 y 4 transmiten el mensaje M al nodo 1 y terminan su ejecución en los tiempos T_2 y T_3 respectivamente. El nodo 1 recibe estos mensajes y concluye su ejecución, finalizando así el algoritmo.

Lema 2.7. *La propagación de la información es lo más rápida posible en el mismo sentido que se da en el lema 2.6.*

Demostración. Queremos argumentar que la construcción descrita en aquel resultado es idéntica a la que se consigue esta vez, ya que exactamente como sucede en el algoritmo PI, un nodo reconoce a su padre cuando recibe M por primera vez. De inmediato reexpide el mismo mensaje a sus vecinos. Esta secuencia de acciones solo puede presentarse una ocasión porque el nodo cambia de estado. Visto de otra forma, el árbol T de pesos mínimos se construye al tiempo que el mensaje se va alejando de la raíz donde se origina. Vale la pena recordar que los pesos de las aristas a los que se refiere la construcción describen los retardos de comunicación de los mensajes que cruzan por ellas. □

Este último resultado nos dice que, al menos en la primera fase de su desarrollo, el nuevo algoritmo produce los mismos resultados que el anterior. Ahora vamos a aprovechar esta construcción para establecer la propiedad que nos interesa

Lema 2.8. *El algoritmo termina globalmente.*

Demostración. Queremos demostrar que cuando el algoritmo termina en p es porque cualquier otro proceso ha concluido su actividad correspondiente. Para ello haremos un análisis de la actividad sobre el árbol T, descrito en el resultado previo. Aquí es importante observar que aun cuando una arista de G transporta el mensaje, ello no implica que forme parte de T.

Sabemos que todo nodo conectado a p recibe en algún momento el mensaje M y lo reexpide por todas sus aristas. En consecuencia podemos inferir que un nodo hoja recibe el mensaje por cada una de sus aristas. Cuando esto sucede, de inmediato reexpide el mensaje hacia su padre y termina. Podemos afirmar que, para cualquier otro vértice de T que no sea hoja, la ejecución de esta secuencia implica que se ha recibido el mensaje por cada una de sus aristas, incluyendo aquellas que lo conectan con sus hijos. Pero esto, a su vez, implica que todos sus descendientes han ejecutado la misma secuencia, es decir, han terminado. En consecuencia, cuando p cierra su actividad es porque todos sus descendientes, i.e. todos los demás vértices, han terminado previamente. Luego, p es el último proceso activo. □

Hemos argumentado sobre la existencia de una segunda fase del algoritmo PIF, que no existe en PI, la que se conoce como fase de *retroalimentación*. Es importante observar cómo cada proceso transmite los mismos mensajes que se producirían en PI. Sin embargo, la diferencia consiste en retrasar el mensaje que se devuelve al padre, hasta que comienza esta segunda fase.

Corolario 2.5. *Cuando el algoritmo concluye, se ha enviado exactamente un mensaje en cada dirección, de cada enlace.*

Demostración. Todo proceso que termina recibió una copia de M, que reexpidió a cada uno de sus vecinos, incluyendo al proceso del que recibió el mensaje por primera vez. Cada enlace une a dos vértices desde cuyos procesos se envía este mensaje. Por tanto, el algoritmo termina habiendo intercambiado $2m = O(m)$ mensajes □

En términos de complejidad en mensajes los dos algoritmos presentados tienen los mismos costos. El precio que se paga por la propiedad de terminación global debe impactar entonces por el lado de la complejidad en tiempo.

Corolario 2.6. *Asumiendo que cada mensaje se entrega con un retardo máximo de una unidad de tiempo, el algoritmo concluye en un plazo $O(D)$, donde D es el diámetro de G.*

Demostración. Sean i y j la pareja de vértices en G que definen la trayectoria más larga en el grafo, i.e. el diámetro, D. Iniciamos el algoritmo en $p = i$ sabemos, por el lema 2.6, y bajo la hipótesis de tiempo que estamos considerando, que la altura del árbol resultante será la distancia entre i y j, es decir D. Esto significa que la primera fase del algoritmo tomará D unidades de tiempo y, en consecuencia, la segunda fase tomará otro tanto igual. En suma, el algoritmo consume $2D = O(D)$ unidades de tiempo. □

Este último resultado nos dice que el orden de la complejidad en tiempo es el mismo que para PI. Sin embargo, visto en detalle, PIF toma el doble de tiempo que el primer algoritmo.

2.5. Comentarios finales

El algoritmo de búsqueda en profundidad es un procedimiento muy sencillo pero importante, que construye una secuencia de exploración exhaustiva que puede aprovecharse para aplicaciones posteriores. En las redes P2P (Par a Par, en inglés *Peer to Peer*) no estructuradas, por ejemplo, la búsqueda de un archivo

almacenado en un nodo de la red se puede efectuar viajando sobre el árbol DFS, que previamente se ha construido.

Como se recordará, es posible construir un árbol DFS con tan solo 2 mensajes por enlace, con un precio muy caro en tiempo debido a que el algoritmo que presentamos "pierde" tiempo explorando aristas que luego van a quedar podadas, estas son las denominadas aristas de retroceso. Para reducir la complejidad en tiempo es preciso efectuar un compromiso con la comunicación, que eleva el número de mensajes intercambiados, ya que no bastan dos mensajes para podar una arista de retroceso.

Para analizar las cotas de complejidad de un algoritmo DFS que se supusiera óptimo en tiempo y mensajes, hay que observar que cualquier solución funcionará en 2 fases: la primera, en la que los procesos se activan y separan las aristas que forman parte del árbol, de aquellas que quedan fuera. La segunda, de terminación, en que cada proceso concluye su cómputo local y devuelve el control de la ejecución a su padre.

Por lo que concierne al problema de *difusión* o *propagación*, como también se le conoce, éste aparece repetidamente en las operaciones de un sistema distribuido. Para calcular la ruta más corta entre dos sitios arbitrarios, para desarrollar pruebas de conectividad, para reconocer cambios en la topología de la red, el bloque de construcción con el que se resuelven estas tareas es un algoritmo de propagación. Cuando cada proceso que participa en un sistema debe informar a los demás, sobre las condiciones de funcionamiento en el procesador donde reside, nuevamente la solución es un procedimiento de propagación.

Se trata más bien de una operación inherente a la naturaleza de los sistemas distribuidos, que se presenta por la necesidad de intercambiar información entre las entidades participantes. De ahí que repetidamente puede descubrirse esta operación como componente de otras tareas distribuidas de mayor complejidad.

En lo que concierne al algoritmo PIF, en particular, se reconocen dos etapas complementarias: En un principio, mientras se construye el árbol del teorema 2.6 y el mensaje viaja desde p hacia los demás procesos, se habla de una ola que se propaga de la raíz hacia las hojas, esta primera parte también se presenta en PI. Posteriormente, cuando los procesos comienzan a devolver sus mensajes de reconocimiento, se dice que la ola remonta el árbol o se contrae hasta "colapsarse", en el punto en que se originó.

En muchas aplicaciones distribuidas se observa esta secuencia de propagación y contracción (o retro propagación) de una ola. En estos casos, la expansión de la ola, y su regreso, parecen estar controlados por otros factores propios de cada problema particular [41]. Algunas veces, se fija un límite en la profundidad de la ola. Esto significa que solo llega a cubrir una sección del grafo. En otras circunstancias, se puede iniciar la ola desde distintos puntos de la red [48] y de manera simultánea.

Ejercicios

2.1. En realidad, al terminar el algoritmo DFS cada proceso conoce el enlace que lo conecta con su padre, pero ignora qué enlaces lo unen con sus hijos, ¿qué modificaciones deben hacerse al pseudocódigo para conseguir esta información?

RESPUESTA: Al recibir el mensaje `REGRESA`, el receptor (i) debe insertar la identidad del transmisor (j) en una lista que puede llamarse "hijos", inicialmente vacía.

2.2. Programa un algoritmo que transporte una ficha sobre un árbol DFS, previamente construido

SUGERENCIA: Cuando el nodo raíz reconoce a su último hijo, cada nodo del árbol resultante está en posibilidad de visitar a sus hijos. Puede hacerlo en el mismo orden en el que fueron incorporándose. Luego de visitar a sus hijos, el nodo debe devolver el control a su padre.

2.3. Diseñe un algoritmo DFS, con una complejidad en tiempo $O(n)$.

RESPUESTA: Para agilizar el algoritmo, cualquier nodo que es visitado por primera vez debería notificar está condición a sus vecinos (y con ello evitar que le envíen el mismo mensaje otra vez). Para lograr esta condición, envía un nuevo mensaje al que llamaremos `AVISO`, a todos sus vecinos. Al recibirse este mensaje en el nodo i, se elimina a su transmisor, j, de la lista "*sin_visitar*".

2.4. Programe un algoritmo PIF que se repita cada vez que termina, durante 3 ciclos consecutivos. En cada ciclo la ola debe llegar 2 veces más lejos que en la ocasión anterior, comenzando por una distancia de 1 enlace.

SUGERENCIA: El algoritmo PIF crea una ola que se propaga y luego retrocede hasta terminar en el punto donde inició. Podemos usar un contador que incorporamos en el mensaje que viaja en la ola. Este contador se decrementa cada vez que el mensaje es reexpedido un enlace más lejos. La ola sigue viajando hasta que el contador sea 0, o el grafo subyacente no tenga más nodos por donde propagar el mensaje. Cuando la ola regrese a la raíz, incrementamos por 2 el valor inicial del contador durante dos ocasiones consecutivas.

2.5. ¿Cómo podemos implantar un algoritmo DFS sobre un modelo de red con canales que no obedecen a un orden de entrega tipo FIFO?

RESPUESTA: Se puede convertir un canal no FIFO en uno tipo FIFO, si incorporamos un número de secuencia consecutivo, en cada mensaje que se transmite por ese canal.

2.6. Luego de terminado el algoritmo 2.3, ¿qué haría el nodo i para enviar un mensaje a un nodo j arbitrario?, asumiendo que ambos yacen en el mismo árbol construido por el algoritmo.

SUGERENCIA: Existe más de una forma de resolver este problema de encaminamiento, aquí proponemos una muy sencilla, pero un poco costosa en almacenamiento. Preliminarmente, proponemos modificar el algoritmo PIF de manera que, cuando un nodo devuelve el control a su padre, le devuelve también una lista con todos sus descendientes. Entonces, cada nodo lleva un registro no solo de sus hijos sino de todos los nodos que puede alcanzar a través de sus hijos. Luego que el algoritmo PIF ha terminado, un nodo i reconoce si j es su descendiente y lo envía al hijo que lo acerca con j. En otro caso, i reexpide el mensaje hacia su padre.

2.7. ¿Por qué no puede considerarse que el algoritmo PIF produce siempre un árbol en amplitud o BFS (Breadth First Search)?

RESPUESTA: El camino más rápido entre dos vecinos no es necesariamente el enlace que los conecta directamente. El algoritmo PI (y el PIF también) produce un árbol generador en el sentido del lema 2.6, esto es, utiliza los enlaces más rápidos. Ello implica que, dos nodos vecinos podrían resultar conectados por un camino que no incluye al enlace que los conecta directamente.

Se osado y fuerzas poderosas te acompañaran

—Mark Twain

3 Elección

Resumen En muchas operaciones distribuidas se necesita que un proceso actúe como coordinador, iniciador, secuenciador o que desempeñe un papel especial. En general, no importa cuál de los procesos asuma esta responsabilidad, mientras se garantice que exactamente uno de ellos entrará en esta condición. ¿Qué sucede cuando se detecta la ausencia del coordinador?, ¿podría ocurrir que más de un proceso se postulara para ocupar este rol?, en estas circunstancias nos encontramos ante una instancia del problema de elección. En el presente capítulo presentamos un conjunto de algoritmos que resuelven dicho problema. Comenzamos revisando los casos más sencillos sobre redes de tipo anillo y terminamos con un algoritmo general que funciona sobre una red arbitraria. En cada caso mostramos las propiedades que debe exhibir la solución, así como los costos que implica cada una de ellas.

3.1. Introducción

La elección encuentra aplicación en operaciones distribuidas en las que se requiere de un proceso coordinador para resolver, entre otros, problemas de acceso en exclusión mutua, solución de controversias o ciertos tipos de permiso por turnos.

En redes de anillo, por ejemplo, un algoritmo de elección se utiliza cuando se detecta la ausencia de la ficha que otorga el acceso al canal y debe reponerse esta señal pero, con la garantía que solo un proceso resulte a cargo de esta operación de reinicio. Si no se conoce o no existe un participante que asuma esta responsabilidad, entonces debe arrancarse la elección del coordinador, que luego se encargará de restablecer las operaciones en el sistema.

El problema de *elección* se caracteriza por la transformación del estado general del sistema donde, a partir de una configuración inicial en la que todos los procesos activos se encuentran en el mismo estado, se llega a una configuración final, en la que hay un solo proceso con un estado especial que lo distingue del resto. Alternativamente, se puede pensar que cada uno de los participantes cuenta con una variable booleana que, una vez resuelta la elección, tendrá el valor 0, salvo en uno de los procesos al que se designa como vencedor, cuya variable tomará el valor 1.

Cualquier problema distribuido tiene diferentes versiones que quedan determinadas por condiciones particulares como las características del modelo de tiempo, el modelo de intercambio de información, etc. Diferentes combinaciones de estas condiciones pueden producir situaciones donde la solución es más o menos difícil, o incluso, puede no existir.

Para comenzar presentamos el algoritmo de LCR (Le Lann, Chang-Roberts), que resuelve el problema sobre un anillo asíncrono unidireccional con identidades, es decir, donde los mensajes circulan en un solo sentido y cada nodo tiene un identificador que lo distingue del resto. Este algoritmo tiene una complejidad $O(n^2)$ en mensajes y $O(n)$ en tiempo, respectivamente. Enseguida, continuamos con el algoritmo de HS (Hirschberg-Sinclair), que trabaja sobre un anillo asíncrono bidireccional, con identidades. Este algoritmo tiene una complejidad $O(n\log(n))$ en mensajes y $O(n)$ en tiempo, respectivamente. Finalmente, cerramos el capítulo presentando un algoritmo que funciona sobre una red arbitraria (Peleg), y ofrece una complejidad $O(D)$ que es óptima en tiempo, siendo D el diámetro del grafo sobre el que se ejecuta.

En los procedimientos que se presentan se asume que cada nodo cuenta con un identificador único, de modo que los identificadores asignados forman un conjunto totalmente ordenado. Ello implica que los algoritmos operan por comparación de las identidades de los procesos. Sin embargo, es importante mencionar que la comparación de identidades no es la única forma de decidir una elección.

Todo algoritmo de elección, que alcance correctamente su cometido, debe cumplir con dos propiedades básicas: por un lado, se tiene que "a lo más uno de los procesos debe resultar electo"; por otro se tiene que "al cabo de un tiempo finito, todos los procesos entran en un estado final y *al menos* uno de ellos resultará electo". La primera propiedad busca garantizar que nada malo ocurrirá a lo largo de la ejecución. La segunda propiedad busca garantizar que algo bueno o deseable se obtendrá luego de un tiempo finito. Si imaginamos que cada proceso que participa tiene una variable de tipo bandera, que queda levantada cuando el proceso es electo, entonces la primera propiedad nos dice que nunca puede haber dos banderas en alto. En tanto, la segunda nos dice que el algoritmo terminará y cuando esto ocurra habrá, a lo más, una bandera en alto.

La notación utilizada en este capítulo se presenta en la tabla 3.1.

3.2. Elección en anillo unidireccional con identidades

Supongamos un sistema distribuido cuya red de comunicaciones está descrito por un grafo $G = (V,E)$, que forma un ciclo, esto es, un camino cerrado y único que une a todos los nodos. Es importante observar que los enlaces del anillo, como también se le conoce, no necesariamente representan canales físicos, algunos sistemas conectan a sus componentes mediante canales lógicos que producen esta misma estructura.

$G = (V,E)$	Grafo G con conjunto de vértices V y conjunto de aristas E
D	Diámetro de la red
n	Número de nodos en la red
m	Número de enlaces en la red
p	Un proceso
`CANDIDATO`, `REGRESA` `ELECTO`, `FICHA`	Mensajes
i, j, max	Identificador de nodo/proceso
izquierda, *derecha*	Nodos adyacentes al nodo i en un anillo
vecinos	Conjunto de nodos adyacentes al nodo i
estado	Estado local del nodo i
líder	Identificador del nodo ganador de la contienda
$v_1, \ldots, v_n$	Nodos
d	Distancia
r	Número de ronda
R	Última ronda
contador, *señal* *pendientes*, *max_local* *distancia_local*, *espera*	Variables locales

Tabla 3.1: Notación

Nos interesa resolver el problema de elección sobre G, bajo el supuesto que los canales de comunicación son unidireccionales. A diferencia de los modelos de canal hasta ahora considerados, esta condición nos dice que cualquier enlace que une a una pareja de nodos, transporta mensajes en un sentido solamente. En consecuencia, cada nodo tiene un solo vecino del que puede recibir información y un solo vecino al que puede transmitir información. Consideraremos, sin pérdida de generalidad, que los mensajes circulan en el sentido de las manecillas del reloj.

El algoritmo de LCR 3.1, inicialmente propuesto por Le Lann [35] y posteriormente mejorado por el equipo de Chang y Roberts [13], comprende el intercambio de dos tipos de mensajes.

`CANDIDATO`(j): Transporta el identificador de un nodo que participa en la competencia. Solo una instancia de este mensaje podrá dar la vuelta completa al anillo. Aquella con el mayor identificador de la red.

`ELECTO`(j): Transporta el identificador del nodo ganador de la competencia. Completa una vuelta para notificar a todos los nodos el resultado de la elección

Por otro lado, además de su propio identificador i, cada proceso a cargo del algoritmo mantiene una colección local de variables:

estado: $\in \{$*despierto, dormido*$\}$. Su valor inicial es *dormido* cambia a *despierto* cuando el proceso inicia su participación, ya sea de forma espontánea o bien, porque su vecino lo despierta.

líder : Es un valor entero, inicialmente 0. Al terminar el algoritmo guarda el identificador del proceso electo.

Un proceso puede arrancar espontáneamente, o despertar por la recepción de un mensaje de tipo `CANDIDATO`. En ambas situaciones cambia su estado a *despierto*. En el primer caso, el proceso se "postula" transmitiendo un mensaje `CANDIDATO`, que contiene además el valor de su identificador. En

Algoritmo 3.1: Algoritmo de elección de Le Lann, Chang, Roberts (LCR), en el nodo i

```
1  al recibir CANDIDATO(j) desde izquierda efectúa
2      si (j > i) entonces envía CANDIDATO(j) a derecha
3      otro
4          si (j ≤ i) & (estado ≠ despierto) entonces
5              envía CANDIDATO(i) a derecha
6              estado ← despierto
7          otro
8              si (j = i) entonces envía ELECTO(i) a derecha
9  al recibir ELECTO(j) desde izquierda efectúa
10     líder ← j
11     si (j ≠ i) entonces envía ELECTO(j) a derecha
12     termina
```

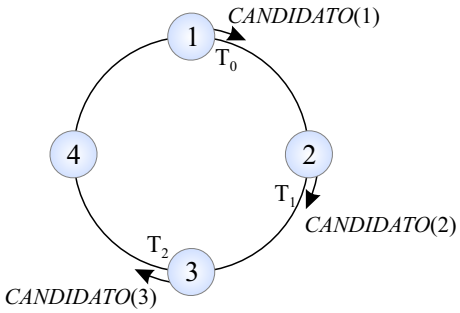
el segundo caso, el proceso debe comparar su identificador con el valor transportado por el mensaje que recién ha recibido. Si su identificador es mayor, entonces reexpide el mismo mensaje pero le reemplaza su contenido con su propio valor. Si su identificador es menor, entonces reexpide exactamente el mismo mensaje. Si su valor es igual al que recibe, y ya está despierto, entonces significa que el mensaje ha completado una vuelta al anillo sin haber encontrado otro proceso con un identificador mayor. Por tanto, es el vencedor de la contienda y ahora reexpide un nuevo mensaje de tipo `ELECTO` para informar a todos sobre el identificador del ganador y el fin del algoritmo. Todo proceso que recibe este segundo tipo de mensaje actualiza su variable *líder* con el valor del identificador transportado. En la figura 3.1 se ilustra una ejecución en particular del algoritmo.

Propiedades del algoritmo LCR

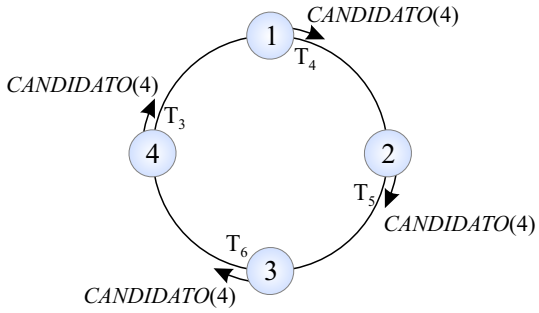
Para analizar el algoritmo debemos demostrar que termina y, cuando esto ocurre, existe uno y solamente un proceso cuyo variable *líder* coincide con su identificador. Para terminar determinaremos cuál es su complejidad en mensajes y tiempo.

Lema 3.1. *Todos los procesos del anillo terminan el algoritmo. Cuando esto ocurre, existe exactamente un proceso al que todos reconocen como líder.*

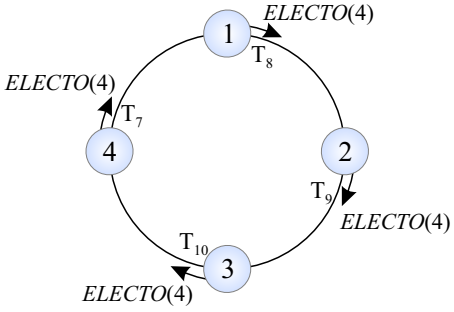
Demostración. Para comenzar debemos recordar que cada proceso tiene un identificador que toma del nodo en el que reside. También, que cada identificador es único y diferente de los demás. Luego, existe un valor que corresponde al máximo identificador asignado a un nodo. Cuando el proceso con este valor expide un mensaje `CANDIDATO`, ya sea porque se despierta espontáneamente o su vecino lo saca de ese estado, entonces este mensaje podrá completar una vuelta al anillo, ya que no encontrará otro identificador que lo reemplace. Cuando este mensaje llegue de regreso al punto de donde partió, el proceso de origen enviará un nuevo mensaje de tipo `ELECTO`, que viajará por todo el anillo hasta completar un ciclo. A su paso por cada nodo, este mensaje cierra la actividad. Sabemos, por otra parte,



(a) La estampilla de tiempo T_i corresponde a un posible instante de tiempo en el que se genera el mensaje. El nodo 1 despierta espontáneamente y transmite el mensaje `CANDIDATO`(1) con su identificador a cuestas. El nodo 2 recibe este mensaje y, debido a que cuenta con un identificador mayor que el del mensaje recibido, reexpide el mensaje `CANDIDATO`(2) sustituyendo el valor del mensaje con su propio identificador. De forma análoga lo hace el nodo 3 y el nodo 4.



(b) Debido a que el nodo 4 tiene el identificador más grande, el mensaje `CANDIDATO`(4) dará una vuelta completa al anillo.



(c) Tras la recepción del mensaje `CANDIDATO`(4) el nodo 4 resulta vencedor y transmite el mensaje `ELECTO`(4) para dar a conocer su condición de líder.

Figura 3.1: Un ejemplo del algoritmo de elección LCR

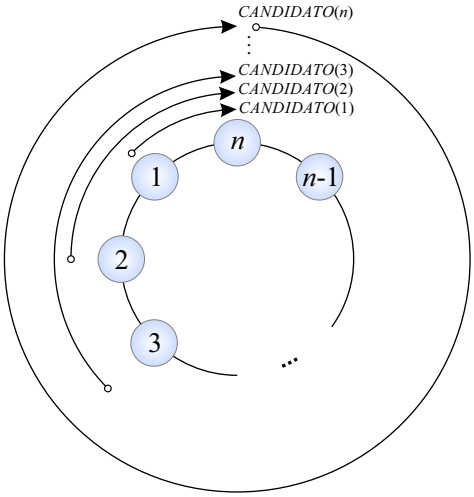


Figura 3.2: El peor caso considerando los mensajes transmitidos en el algoritmo de elección LCR. Supongamos, sin perdida de generalidad, que los identificadores van de 1 a n y que están asignados en orden decreciente en el sentido de las manecillas. Supongamos también, que el algoritmo de elección LCR arranca de forma simultánea en todos los nodos. El mensaje originado en el nodo 1 habrá viajado solo una vez, el del nodo 2 dos saltos, y así sucesivamente para los demás mensajes originados en los nodos subsecuentes. De esta forma, el número total de mensajes es la suma $\{1+2+,\ldots,+n\}$, lo cual resulta en una complejidad en mensajes de $O(n^2)$.

que todos los identificadores son diferentes, por tanto, solo hay un nodo cuyo identificador coincide con el valor de su variable *líder*. □

En el lema 3.2 mostramos cuántos mensajes deben intercambiarse, en el peor caso, para resolver el problema de elección mediante LCR. Y finalmente, en el lema 3.3 vamos a evaluar cuánto tiempo tardaría el algoritmo, en el peor caso.

Lema 3.2. *El costo en mensajes del algoritmo de LCR tiene una complejidad $O(n^2)$.*

Demostración. Sea G un anillo de n nodos. Designaremos como v_1 a un nodo arbitrario de G. Enseguida, v_2 será el nodo a la derecha de v_1, es decir el que le sigue en el sentido de las manecillas del reloj. De manera general, v_l será el que se encuentre a $l-1$ enlaces a la derecha de v_1, para $l = 2,\ldots,n$. Supongamos una situación en la que ordenamos de mayor a menor los posibles identificadores con los que se designan los nodos de G. Luego, asignamos el primer valor a v_1, el segundo a v_2 y así sucesivamente. Ahora, imaginemos que el algoritmo de elección se arranca de forma simultánea, en cada nodo del anillo. Podemos prever que todos los mensajes de tipo `CANDIDATO` irán a "morir" en v_1. El mensaje originado en v_n viajará solo una vez. El que salió de v_l habrá viajado $n+1-l$ saltos. Solo el mensaje que salió de v_1 podrá completar el ciclo. Entonces, el costo de la comunicación en esta primera fase será de $1+2+\ldots+n = (n+1)(n)/2$ mensajes. En tanto, para la segunda fase del algoritmo, se expide un mensaje único (`ELECTO`), que completa una vuelta al anillo. En total, se han transmitido $(n+1)(n)/2+(n)$ mensajes. En consecuencia, el peor caso tiene una complejidad $O(n^2)$ mensajes, véase la figura 3.2. □

Lema 3.3. *Asumiendo que cada mensaje se entrega con un retardo máximo de una unidad de tiempo, el algoritmo concluye en un plazo $O(n)$.*

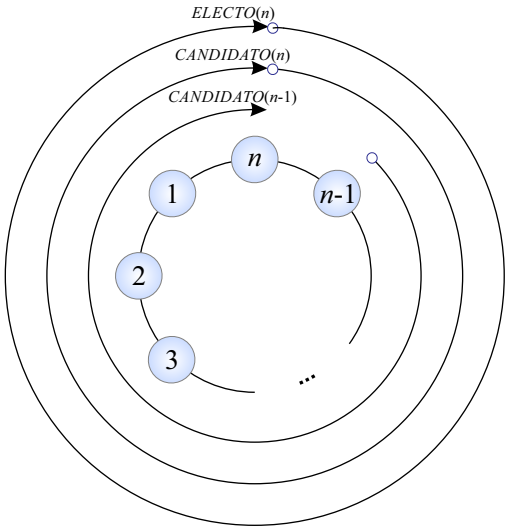


Figura 3.3: El peor caso considerando el tiempo consumido en el algoritmo de elección LCR. Como en la figura 3.2, supongamos que los identificadores se asignan en orden decreciente de n a 1, en el sentido de las manecillas del reloj. Supongamos también, que el nodo $n-1$ es el único en arrancar el algoritmo de manera espontánea. Después de la transmisión del mensaje `CANDIDATO`$(n-1)$ sucede lo siguiente: i) El mensaje `CANDIDATO`$(n-1)$ es retransmitido por cado nodo subsecuente hasta llegar al nodo n donde es reemplazado por el mensaje candidato de este último. ii) El mensaje `CANDIDATO`(n) es retransmitido por cada nodo subsecuente hasta regresar al nodo n. Y, iii) al recibir este mensaje, el nodo n se sabe ganador de la contienda por lo que transmite el mensaje `ELECTO`(n) que también será retransmitido por cada nodo subsecuente hasta regresar a n. En consecuencia, se tienen $(n-1)$ saltos del mensaje originado en el nodo $n-1$ y $2n$ de los mensajes originados en el nodo n. Sumando el número de saltos se tiene un total de $3n-1$ obteniéndose una complejidad en tiempo de $O(n)$.

Demostración. Sea v_1 el nodo que tiene asignado el máximo identificador. Supongamos ahora que el segundo mayor identificador se encuentra asignado a un nodo v_2, a la derecha de v_1. Ahora supongamos también que el proceso en v_2 es el único en arrancar el algoritmo de manera espontánea. Su mensaje `CANDIDATO` dará casi una vuelta completa al anillo pero, cuando llegue a v_1 el valor que acarrea a cuestas será reemplazado por el máximo identificador. Por su parte, este nuevo mensaje podrá completar la vuelta al anillo, luego de lo cual dará lugar al mensaje `ELECTO`, que también completará una vuelta. Si suponemos que cada mensaje se entrega con un retardo máximo de una unidad de tiempo, entonces el algoritmo termina al cabo de $n-1+n+n=3n-1$ unidades de tiempo. Visto de otro modo, tiene una complejidad $O(n)$ en tiempo, véase la figura 3.3. □

3.3. Elección en anillo bidireccional con identidades

En esta sección presentamos un nuevo algoritmo de elección sobre anillos. Pero ahora, suponiendo que los canales de comunicación son bidireccionales. Esta condición indica que cada proceso puede intercambiar mensajes con un vecino a la derecha y otro a la izquierda. La idea principal del algoritmo HS, propuesto por Hirschberg y Sinclair [28], consiste en llevar a cabo elecciones mediante rondas

Algoritmo 3.2: Algoritmo de elección Hirschberg-Sinclair (HS), en el nodo i

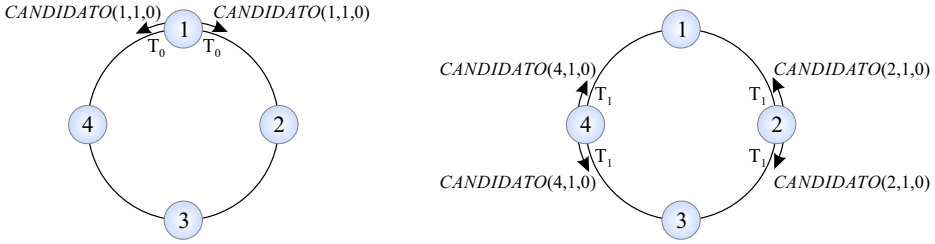
```
1  al recibir CANDIDATO(j,d,r) desde derecha(izquierda) efectúa
2      si (j > i) & (d < 2^r) entonces
3          envía CANDIDATO(j,d+1,r) a izquierda(derecha)
4      otro
5          si (j > i) & (d = 2^r) entonces
6              envía REGRESA(j,r) a derecha(izquierda)
7          otro
8              si (j ≤ i)&(estado ≠ despierto) entonces inicia(i,1,0)
9              otro
10                 si (j = i) entonces envía ELECTO(i) a derecha
11     estado ← despierto
12 al recibir REGRESA(j,r) desde derecha(izquierda) efectúa
13     si (i ≠ j) entonces
14         envía REGRESA(j,r) a izquierda(derecha)
15     otro
16         contador++
17         si contador = 2 entonces inicia(i,1,r+1)
18 al recibir ELECTO(j) desde izquierda efectúa
19     líder ← j
20     si (j ≠ i) entonces envía ELECTO(j) a derecha
21     termina
22 procedimiento inicia(id,distancia,ronda)
23     envía CANDIDATO(id,distancia,ronda) a derecha & izquierda
24     contador = 0
```

asíncronas. Un proceso que se propone como candidato comienza compitiendo dentro de una zona o "arco" del anillo, que incluye a aquellos procesos que se encuentran en sus cercanías. El ganador de la ronda, pasa a competir contra los ganadores de zonas aledañas. Lo que a su vez define una nueva zona. En la última ronda, la zona comprende a todo el anillo. El ganador final define al líder. Otro aspecto clave del algoritmo es que, en cada nueva ronda, la zona de competencia duplica su tamaño, esto significa que un proceso i ganador de una ronda r tiene el mayor identificador de todos los nodos que están, a lo más, a una distancia de 2^r saltos de i. Visto de otra forma, en cada ronda disminuye a la mitad el número de participantes.

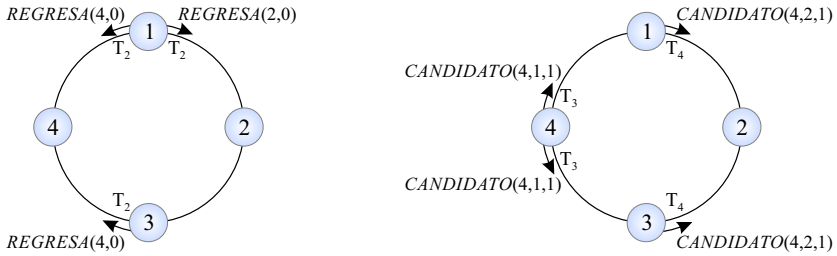
El algoritmo HS 3.2, comprende el intercambio de tres tipos de mensajes:

`CANDIDATO(j,d,r)`:	Transporta el identificador (j) de un nodo que participa en la competencia, así como la distancia en saltos (d) desde el nodo en que se originó, y la ronda (r) en la que "compite".
`REGRESA(j,r)`:	Transporta el identificador (j) del nodo ganador de la última ronda (r), de vuelta al lugar en donde se originó.
`ELECTO(j)`:	Notifica la identidad (j) del proceso que resultó electo como líder.

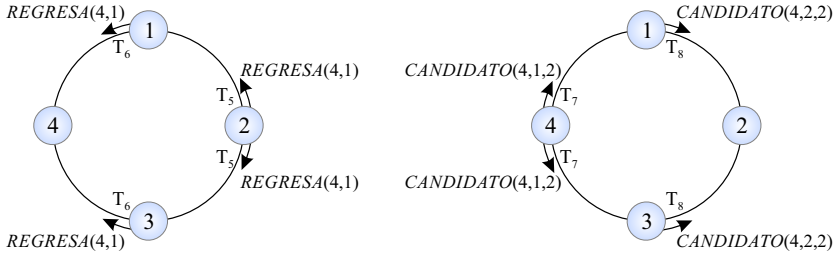
Por otro lado, además de su propio identificador *i*, cada proceso a cargo del algoritmo mantiene una colección local de variables: *estado* y *líder*, que se utilizan de la misma forma que en el algoritmo LCR. también se utiliza una variable *contador*, que ayuda a un proceso a reconocer cuando ha pasado a la siguiente ronda. La figura 3.4 ilustra una ejecución en particular del algoritmo HS.



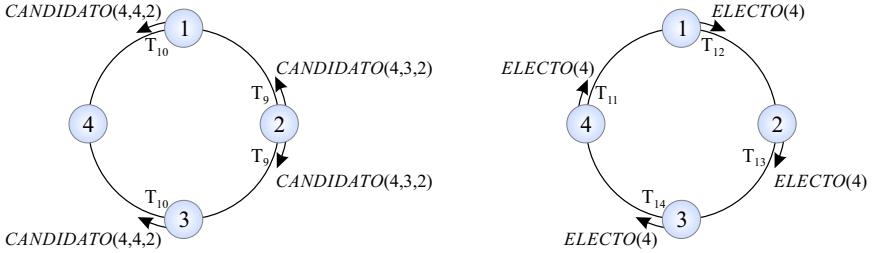
(a) La estampilla de tiempo T_i corresponde a un posible instante de tiempo en el que se genera el mensaje. El nodo 1 despierta espontáneamente y se postula como candidato al transmitir el mensaje `CANDIDATO`(1,1,0). El nodo 4 recibe dicho mensaje y debido a que cuenta con un identificador mayor al del nodo 1 este se postula como candidato y transmite el mensaje `CANDIDATO`(4,1,0). De forma análoga lo hace el nodo 2.



(b) Como respuesta a la candidatura del nodo 4, los nodos 1 y 3 le responden con un mensaje `REGRESA`(4,0). Como respuesta a la candidatura del nodo 2 solo el nodo 1 responde debido a que su identificador es menor. Ante la recepción de ambos mensajes `REGRESA`, el nodo 4 se sabe ganador de esa ronda y se postula otra vez como candidato pero ahora abarcando una distancia mayor.



(c) A través de ambos mensajes `REGRESA`(4,1) el nodo 4 sabe que fue vencedor en esa ronda ($r = 1$) por lo cual se postula como candidato en la ronda siguiente.



(d) El nodo 4 recibe el mensaje `CANDIDATO`(4,4,2) y debido a que el identificador de dicho mensaje coincide con su identificador, el nodo 4 es electo líder noticia que difunde a través del mensaje `ELECTO`(4)

Figura 3.4: Un ejemplo del algoritmo de elección HS

Propiedades del algoritmo de HS

Para analizar el algoritmo debemos demostrar que termina y, cuando esto sucede, existe uno y solamente un proceso cuya variable *líder* coincide con su identificador. Luego revisaremos cuál es su complejidad en mensajes y tiempo.

Lema 3.4. *Todos los procesos del anillo terminan el algoritmo. Cuando esto ocurre, existe exactamente un proceso al que todos reconocen como líder.*

Demostración. Sabemos que existe exactamente un nodo con el máximo identificador. Éste despierta espontáneamente, o es despertado por un vecino. En cualquier caso, será el único que completará todas las rondas del algoritmo y el único cuyos mensajes de tipo `CANDIDATO` darán la vuelta al anillo. Esto es, "lanza" el mensaje por la derecha (izquierda) y lo recibe por la izquierda (derecha)[1]. Luego de lo cual, pone en circulación un mensaje de tipo `ELECTO`, que completa una vuelta al anillo para informar a todos que existe un líder y que el algoritmo puede concluir. □

El siguiente lema nos dice que el número de candidatos que participan en el algoritmo decrece de manera exponencial.

Lema 3.5. *Suponiendo que en la ronda 0 inician n candidatos, para toda ronda $r \geq 1$ el número de procesos que llegarán a la ronda r será a lo más $\lceil \frac{n}{2^{r-1}+1} \rceil$*

Demostración. Sabemos que la mínima distancia entre dos ganadores de la ronda $r-1$ será $2^{r-1}+1$. Sin embargo, solamente uno de ellos llegará a la siguiente ronda. Entonces, dividimos la longitud del anillo en arcos de $2^{r-1}+1$ nodos cada uno, y tenemos el resultado. □

Por otra parte, el número de mensajes que se transmiten en cada ronda es una función lineal del orden del anillo.

Lema 3.6. *El número de mensajes transmitidos por el algoritmo durante cada ronda tiene una complejidad $O(n)$.*

Demostración. Vamos a distinguir entre dos tipos de ronda, las de contienda, que abarcan hasta la penúltima, y la de notificación, que en esencia es el trabajo de la última, es decir, informar que el algoritmo ha terminado y que hay un líder electo. Para toda ronda de contienda, $r \geq 0$, se transmiten a lo más 4×2^r mensajes asociados con cada candidato. Esto es, 2^r mensajes de tipo `CANDIDATO` a la derecha y otro tanto igual a la izquierda. Luego, se devuelve el mismo número de mensajes de tipo `REGRESA`. Sabemos por el lema 3.5 que en la ronda hay, a lo sumo, $\lceil \frac{n}{2^{r-1}+1} \rceil$ candidatos. En consecuencia, se transmitirán en total $4 \times 2^r \lceil \frac{n}{2^{r-1}+1} \rceil < 8n$ mensajes.

Por último, en la ronda de notificación se envía un único mensaje `ELECTO`, que da la vuelta a todo el anillo, es decir, recorre n enlaces. En conclusión, para cualquier ronda se transmiten $O(n)$ mensajes. □

El resultado siguiente es una consecuencia de los dos lemas que lo preceden, véase la figura 3.5

[1] imaginemos un pequeño asteroide como el del Principito de Saint-Exupéry, tiramos una piedra hacia el frente y, por efecto de la gravedad, esta da la vuelta y nos pega en la nuca

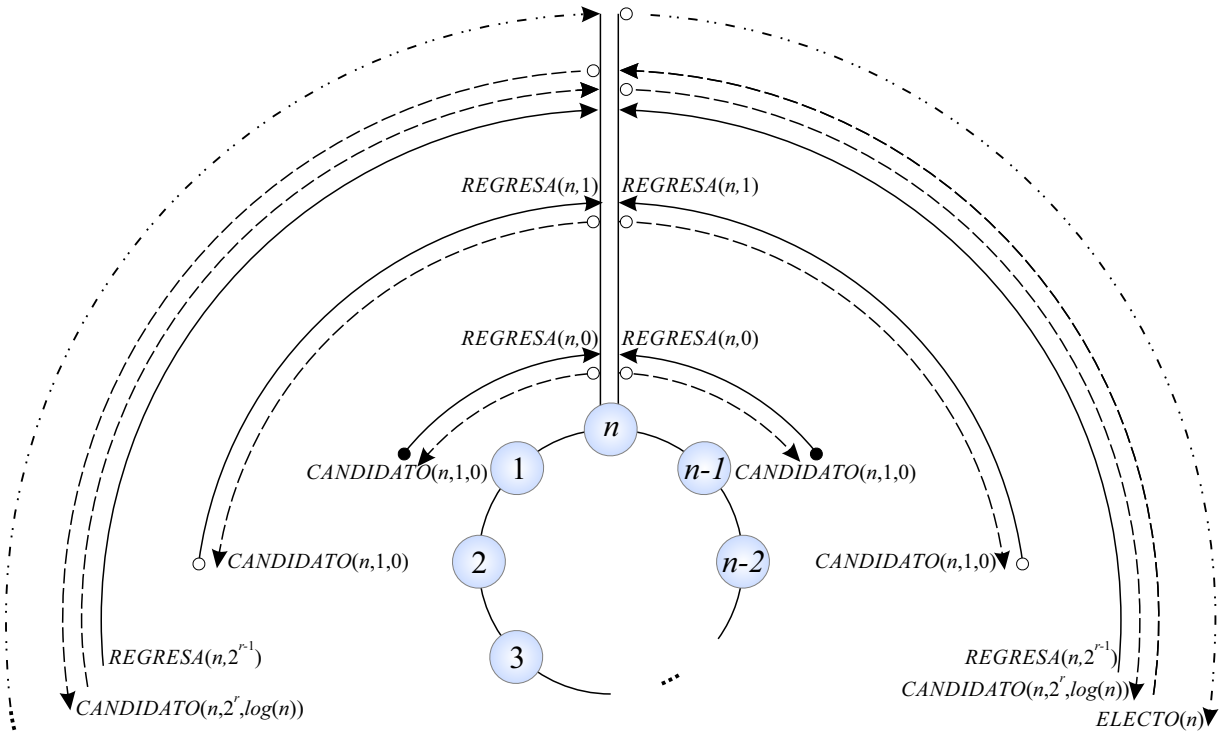


Figura 3.5: El total de mensajes transmitidos en el algoritmo de elección HS

Corolario 3.1. *El total de mensajes transmitidos es* $O(n \log n)$.

Demostración. Sabemos que el proceso con el máximo identificador duplica el tamaño de su arco en cada nueva ronda. Sea R la última ronda de competencia, entonces podemos inferir que $2^{R-1} < n$, mientras que $2^R \geq n$. En consecuencia, le toma $R = \lceil \log n \rceil$ rondas para reconocerse como el ganador absoluto. Por tanto, el total de mensajes transmitidos será, en el peor caso: $\sum_{r=0}^{R} 8n = O(n \log n)$. Aun si consideramos los mensajes aportados en la ronda de notificación, la complejidad total seguirá siendo $O(n \log n)$. □

Por último, queremos averiguar cuál es la complejidad en tiempo del algoritmo.

Corolario 3.2. *Asumiendo que cada mensaje se entrega con un retardo máximo de una unidad de tiempo, el algoritmo concluye en un plazo* $O(n)$.

Demostración. Esta vez, comenzaremos analizando la duración de la ronda de notificación. Sabemos que ésta consiste en un mensaje que se transmite secuencialmente sobre cada enlace del anillo. Por tanto, esta ronda dura n unidades. Por cuanto se refiere a las rondas de competencia, afirmamos que la ronda $r = 0, \ldots, R$, dura 2×2^r unidades de tiempo. Luego, el tiempo que éstas consuman será $2\{2^0 + 2^1 + \ldots + 2^R\} = 2\{2^{R+1} - 1\} = 2(2^{\lceil \log n \rceil + 1} - 1)$. El tiempo total será entonces $2(2^{\lceil \log n \rceil + 1} - 1) + n = O(n)$. □

3.4. Un algoritmo general de elección, óptimo en tiempo

El algoritmo que ahora se considera resuelve el problema de elección sobre una red arbitraria, cuyo grafo subyacente es $G = (V, E)$, con $n = |V|$ y $m = |E|$. Es una variante de un algoritmo de elección por recorrido en amplitud propuesto por Peleg [41]. Se sabe también que es un algoritmo óptimo en tiempo.

En forma espontánea e independiente, un conjunto arbitrario de procesos arranca el algoritmo, sin que esto deba considerarse (necesariamente) como un inicio simultáneo de sus operaciones. Cuando esto ocurre los procesos activos se encargan de propagar un mensaje sobre la red, con el que se echa a andar el procedimiento de elección en los elementos hasta entonces inactivos. Se da por hecho que cada proceso posee un identificador que lo distingue de los demás y que en el conjunto de todos los identificadores existe un orden total. Cuando un participante inicia una nueva ronda del algoritmo transmite a sus vecinos un mensaje donde reporta el identificador más grande que conoce hasta ese momento. La ronda se considera terminada cuando todos los vecinos del proceso le entregan sus propios mensajes, donde cada uno reporta a su vez el identificador más grande que ha "visto" en lo que lleva de su ejecución. En este sentido la operación es similar a un algoritmo PIF de profundidad 1. Cada vez que se completa una operación de tipo PIF, se puede asumir que se ha completado una nueva ronda. Podemos imaginar esta operación como si fuera un "latido", compuesto por una fase de expansión, seguida de una retropropagación (ver el algoritmo de Peleg 3.3 y 3.4).

El mensaje intercambiado entre los participantes, es de un solo tipo, pero contiene dos campos:

`FICHA` (max, d) Un proceso recibe este mensaje desde un vecino que reporta el mayor identificador que ha "visto", *max*, así como la mayor distancia, d, del nodo más alejado a *max* de acuerdo a su conocimiento.

En este algoritmo ocurre que el proceso con el mayor identificador podría no llegar a convertirse en el ganador si no despierta espontáneamente y, en vez de ello, es activado por un vecino. Entonces, el ganador de la contienda, será el proceso con el mayor identificador de entre aquellos que se activen por cuenta propia. Desde la perspectiva de éste, cada "latido" le garantiza que su valor se ha propagado hasta un salto más lejos que la vez anterior. Si se conociera el diámetro D del grafo subyacente, entonces el proceso que se convertirá en el ganador recibiría su propio identificador como el mayor valor reconocido, durante un número de rondas consecutivas acotadas por D. Si no se conociera D, se puede construir una estimación de este valor, a partir de la mayor distancia reconocida en los mensajes que recibe, mientras recibe el "eco" de su propio identificador. Cuando el proceso sigue recibiendo su propio "eco", pero observa que su estimación del diámetro ha dejado de incrementarse por 3 veces consecutivas, significa que su candidatura ha llegado hasta los confines del grafo. Entonces, el sitio puede estar seguro que es el ganador del algoritmo y pasa a la etapa de notificación.

Por otro lado, además de su propio identificador i, cada proceso mantiene una colección local de variables, donde registra el desarrollo del algoritmo y su condición propia:

pendientes: Es una cola donde se almacenan los mensajes cuya atención debe posponerse hasta que cambien las condiciones del nodo. Inicialmente es vacía.

contador: El proceso registra cuando sigue recibiendo un "eco" de su propio identificador pero, si su estimación del diámetro ha dejado de incrementarse por 3 veces consecutivas,

Algoritmo 3.3: Algoritmo de elección de Peleg en el nodo i. Parte 1/2

```
1  al recibir FICHA⟨max, d⟩ desde j efectúa
2      si (estado = dormido) entonces
3          si ( i = max ) entonces
4              max_local ← max, estado ← candidato
5              transmite()
6          otro
7              guarda mensaje en pendientes
8              max_local ← max, estado ← vencido
9              transmite()
10             actualiza()
11     otro
12         guarda mensaje en temporal
13         espera ← espera − 1
14         si (d = −1) entonces
15             señal ← verdadero, líder ← max
16         si (espera = 0) entonces
17             espera ← |vecinos|, pendientes ← temporal, temporal ← ∅
18             transmite()
19             actualiza()
   /*Un nodo procesa los mensajes recibidos durante la ronda actual y si es el caso
     elige un líder*/
20 procedimiento actualiza()
21     r ← r + 1
22     si (señal = falso) entonces
23         candidato ← obtenerMejorCandidato()
24         descartaPeores(candidato)
25         distancia ← obtenerDistanciaMaxima()
26         si (candidato > max_local) entonces
27             estado ← vencido
28             max_local ← candidato
29             distancia_local ← r
30         si (estado = candidato) entonces
31             si (candidato < max_local) entonces contador ← 1
32             otro
33                 si (distancia > distancia_local) entonces
34                     distancia_local ← distancia
35                     contador ← 0
36                 otro
37                     contador ← contador + 1
38                 si (contador = 3) entonces
39                     estado ← electo
40                     distancia_local ← −1
41         otro
42             si (distancia > distancia_local) entonces distancia_local ← distancia
43     otro
44         transmite(max_local, −1)
45         termina
```

Algoritmo 3.4: Algoritmo de elección de Peleg en el nodo i. Parte 2

```
   /*El nodo i transmite a todos sus vecinos el candidato y la distancia del nodo más
     lejano al candidato de acuerdo a su conocimiento*/
1  procedimiento transmite()
2      para todo k en vecinos haz
3          envía FICHA(max_local, distancia_local) a k
   /*El nodo i selecciona al mejor candidato reportado en los mensajes recibidos durante
     la ronda actual*/
4  procedimiento obtenerMejorCandidato()
5      mejorCandidato ← 0
6      para todo mensaje m en pendientes haz
7          Sea max el valor transportado en m
8          si (max > mejorCandidato) entonces mejorCandidato ← max
9      regresa mejorCandidato
   /*El nodo i selecciona únicamente los mensajes correspondientes al mejor candidato
     reportado en los mensajes recibidos durante la ronda actual y descarta los mensajes
     de los otros candidatos.*/
10 procedimiento descartaPeores(mejorCandidato)
11     mejores ← ∅
12     para todo mensaje m en pendientes haz
13         Sea max el valor transportado en m
14         si (max = mejorCandidato) entonces guarda mensaje m en mejores
15     pendientes ← mejores
   /*El nodo i selecciona la distancia máxima reportada en los mensajes recibidos
     durante la ronda actual*/
16 procedimiento obtenerDistanciaMaxima()
17     distanciaMaxima ← 0
18     para todo mensaje m en pendientes haz
19         Sea dist el valor transportado en m
20         si (dist > distancia_local) entonces distanciaMaxima ← dist
21     regresa distanciaMaxima
```

significa que su candidatura ha llegado hasta los confines del grafo. Inicialmente tiene un valor igual a cero.

vecinos: Conjunto de procesos con los que i comparte un enlace.

max_local: El identificador más grande del que el proceso tiene noticia. Inicialmente tiene un valor igual a cero.

distancia_local: Longitud de la distancia más grande desde cualquier nodo a *max_local* de acuerdo al conocimiento de i. Inicialmente tiene un valor igual a cero.

estado: $\in \{$*despierto, dormido,candidato, vencido, electo*$\}$. Su valor inicial es *dormido*.

r: Registra el número de rondas en las que ha participado el proceso. Un proceso en $estado = candidato$ sabe que, si sigue recibiendo su identificador al término de cada ronda, entonces su candidatura se ha propagado un enlace más lejos que en la ocasión anterior. Inicialmente tiene un valor igual a cero.

espera: Cuenta el número de vecinos que faltan de comunicarse para iniciar la fase de actualización. Su valor inicial es $|vecinos|$.

señal: Es un valor booleano, inicialmente es *verdadero*. Cambia a *falso* para dar a conocer que un proceso ha sido electo y terminar el algoritmo.

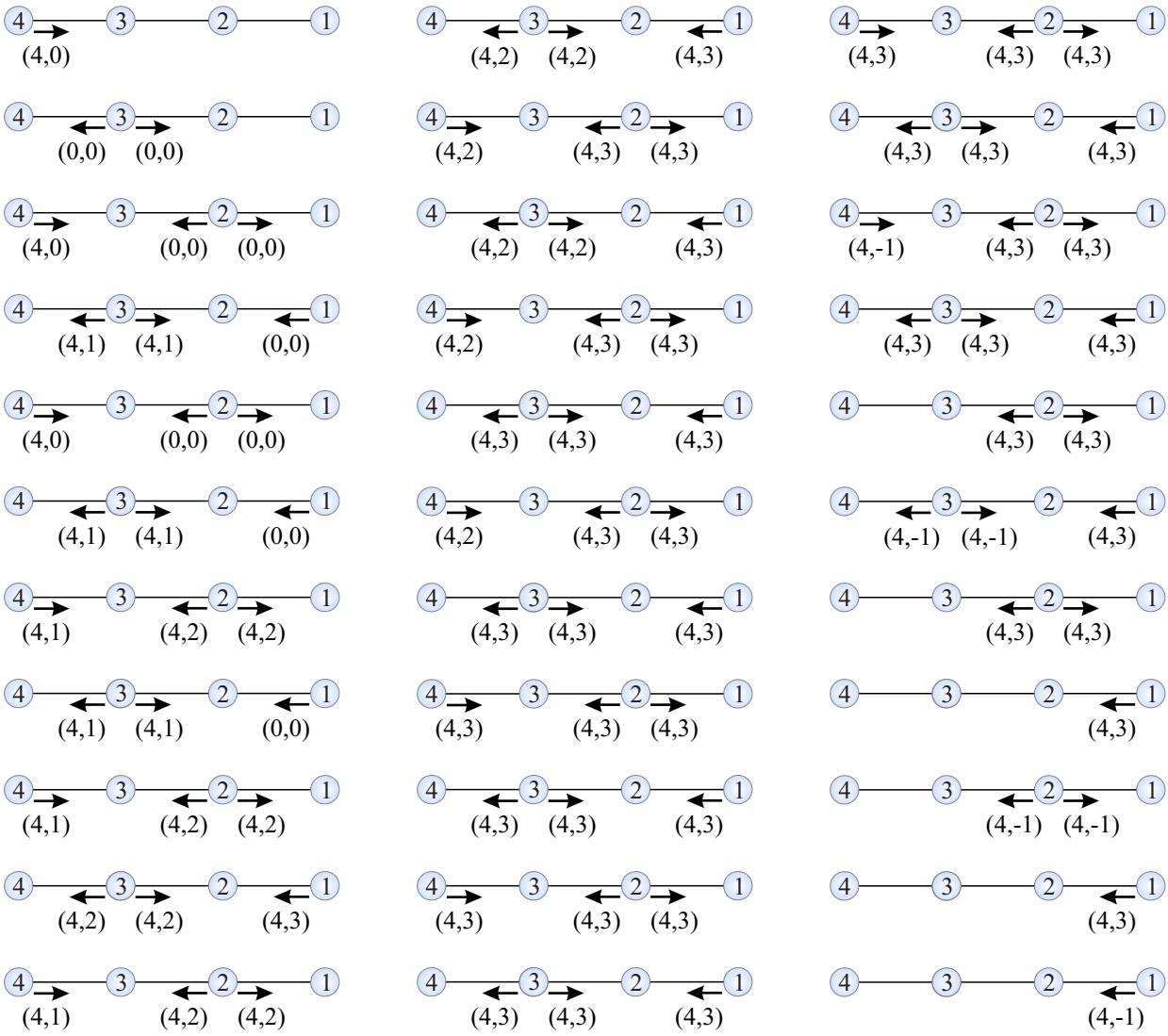


Figura 3.6: Ejemplo de ejecución del algoritmo de Peleg. El único que se propone como candidato es el nodo 4, quien inicia al difundir la ficha $(4,0)$. Obsérvese cómo los mensajes que regresan a 4 llegan con distancias incrementales cada vez que se encuentra un nodo más alejado del cual éste no tenía conocimiento. Cuando el nodo 4 recibe por cuatro veces consecutivas el mismo valor, su estado cambia a *electo* y notifica su condición de *líder* a través del mensaje $(4,-1)$.

Propiedades del algoritmo de Peleg

La duración del algoritmo se mide a partir del momento en que se activa espontáneamente el primer proceso participante. Si todos los mensajes se entregan con un retardo máximo de una unidad de tiempo, entonces todos los procesos estarán despiertos en un tiempo no mayor que $O(D)$ unidades. Donde D es el diámetro del grafo G. Es decir que, para entonces, todos los procesos habrán producido su primera fase.

Sea i el mayor identificador de un proceso participante y t_0 el momento en que éste se activa. Si d_0 es la profundidad del árbol en amplitud con raíz en i, entonces todo proceso $j \neq i$ tendrá un valor max $= i$ al tiempo $t_0 + d_0$ a lo más. Por tanto, a partir del instante $t_0 + 2d_0$, y durante los siguientes 3 pulsos consecutivos, i recibe su propio identificador como el de mayor valor y d_0 como la mayor distancia de la que se tiene conocimiento.

Teorema 3.1. *El algoritmo tiene una complejidad en tiempo $O(D)$ y una complejidad en mensajes $O(mD)$.*

Demostración. Puesto que, tanto t_0 como d_0 son menores o iguales que D, se tiene que el algoritmo dura a lo más $t_0 + 2d_0 + 2 \leq 2D + 2$ unidades de tiempo, esto es, $O(D)$. Por otro lado, si un proceso transmite un mensaje a cada uno de sus vecinos durante una fase, entonces el total de mensajes tiene un orden $mO(D) = O(mD)$. □

3.5. Comentarios finales

La elección es un problema muy rico, por cuanto las características del sistema pueden combinarse para dar lugar a muchas variantes con diferentes grados de dificultad. Tan solo en el caso de elección en anillo se pueden considerar aspectos tales como: la direccionalidad del anillo (¿la información viaja en un solo sentido, o en ambos?), la existencia (o ausencia) de los identificadores de los procesos participantes que forman un conjunto ordenado, el conocimiento común de alguna medida relativa al grafo subyacente (orden, diámetro, etc.), el uso de señales de sincronización (para desarrollar las operaciones en rondas o fases). Es importante remarcar que la elección sobre un anillo no implica que el grafo subyacente deba necesariamente obedecer a esta topología, puede tratarse de un anillo lógico. Si fuera el caso de un sistema con topología arbitraria, que debe ejecutar repetidamente esta tarea, podría precederla la construcción de un anillo lógico, para facilitar la solución de la elección.

Para el problema de elección sobre redes asíncronas con paso de mensajes, se sabe cuál es la complejidad óptima, tanto en tiempo, como en el número de mensajes. Sin embargo, no se conoce un procedimiento general que sea óptimo en las dos medidas: $\Theta(D)$ y $\Theta(m + n\log n)$, respectivamente. Cuando hablamos de un procedimiento general nos referimos a un algoritmo que funcione sobre una topología arbitraria. Donde D es el diámetro del grafo subyacente. Para ciertas clases de redes es posible conseguir una elección óptima en ambos casos, usando el algoritmo de Peleg, lo que podría indicar una posible dirección en el diseño del algoritmo "final". Obsérvese que para una red cuyo grafo subyacente G fuera tal que $m = O(n)$ y $D = O(\log(n))$ se tendrían costos de $\Theta(D)$ y $\Theta(n\log n)$ en tiempo y mensajes, respectivamente.

Ejercicios

3.1. ¿Qué sucedería en un algoritmo de elección si uno de los nodos participantes interrumpiera sus actividades de manera súbita?

RESPUESTA: El algoritmo quedaría estancado, i.e. sin terminar

3.2. ¿Cuál es el máximo número de bits que se necesitan para codificar un mensaje completo del algoritmo HS?

SUGERENCIA: Se necesitan considerar cuántos mensajes distintos pueden enviarse (`CANDIDATO`, `REGRESA`, `ELECTO`), cuántos bits se necesitan para codificar el mayor identificador posible, cuál es la máxima distancia que un mensaje puede viajar y, por último, cuántas rondas puede durar el algoritmo.

3.3. Diseñe un algoritmo para construir un anillo lógico sobre una red arbitraria.

RESPUESTA: Se necesita un algoritmo que recorra todo el grafo subyacente de manera exhaustiva al tiempo que impone un orden secuencial y cíclico de visita. Entonces, un algoritmo de DFS es la solución.

3.4. ¿De qué manera puede un nodo determinar que ha resultado vencedor en el algoritmo de Peleg, si se desconoce el diámetro de la red que subyace en el sistema?

SUGERENCIA: El nodo recibe “ecos” de su propia identidad por un número máximo de veces, luego del cual sabe que no puede esperar cambios en la información de retroalimentación”que recibirá.

3.5. Se sabe que para una red G tal que $m = O(n)$ y $D = O(\log n)$ se tendrían el mejor desempeño en un algoritmo que resolviera el problema de elección, ¿qué grafos tienen estas propiedades?

RESPUESTA: Los árboles binarios balanceados.

...fue mi padre quien lo trajo, yo tenía cinco años y el apenas una rama.

—Alberto Cortez

4

Árbol generador de peso mínimo

Resumen Como en todo el libro, en este capítulo se considera conexo al grafo que modela las comunicaciones de un sistema. Además, en esta ocasión se asume también que cada una de las aristas que lo componen tiene un peso único, que la hace diferente del resto. En estas condiciones, el costo de un árbol generador o abarcador del grafo original, sería igual a la suma de los pesos de las aristas que forman parte del árbol. Buscamos entonces un algoritmo distribuido que induzca el árbol generador de costo mínimo. El procedimiento más sencillo consistiría en construir el árbol incorporando una tras otra las aristas de menor peso. Sin embargo esta solución inicial sería muy costosa en término de los mensajes y el tiempo necesarios para completar la construcción. Un procedimiento más eficiente debe permitir que la construcción avance simultáneamente en varias zonas del grafo original. Esto es, desarrollar un bosque (i.e., una colección de árboles), comenzando con algunos vértices que se despiertan de manera espontánea y que se consideran árboles de orden mínimo. Los árboles se combinan por parejas incorporando una arista del grafo original, para formar un nuevo árbol de mayor orden. Este procedimiento se repite hasta tener un solo árbol que representa la solución buscada, ¿cómo se garantiza que en cada ocasión se incorpora la arista correcta?, ¿cómo se logra la solución con economía de recursos?

4.1. Introducción

En ciertos problemas de computación distribuida ocurre que, partiendo del grafo que subyace en la red de comunicaciones, necesitamos construir un árbol generador que satisfaga la propiedad de costo mínimo. El peso con que se ponderan las aristas del grafo original representa un costo que se desea

$G = (V,E)$	Grafo G con conjunto de vértices V y conjunto de aristas E
D	Diámetro de la red
n	Número de nodos en la red
m	Número de enlaces en la red
$e = uv = vu$	Una arista incidente a los nodos u, v
$w(e)$	Costo asociado a la arista e
AGM	Árbol generador de costo mínimo
T	Un árbol
F	Subárbol del *AGM*
$F(L,w)$	Fragmento F con nivel L y núcleo peso w
`ACEPTA`, `CONECTA`	
`PRUEBA`, `RECHAZO`	Mensajes
`REPORTE`, `INICIO`, `UNION`	
estado	Estado local del nodo i
i, j	Identificador de nodo/proceso
candidato, *cuenta*	
mínimo, *mejor_peso*	Variables locales
mejor_arista, *nivel*	
padre, *peso_núcleo*	

Tabla 4.1: Notación

optimizar. Por ejemplo, podríamos asignar un peso que fuera proporcional al ancho de banda del canal que representa cada arista. Igualmente, el peso podría representar una medida de distancia entre los vértices que cada arista une. Debemos observar que el árbol que buscamos no nos ofrece el camino de menor costo entre cualquier pareja de vértices, sino el conjunto de enlaces de menor costo, suficientes para mantener unidos a los vértices del grafo original.

Existen, desde hace varias décadas, algoritmos centralizados que resuelven este problema. Se reconocen como clásicos dos enfoques de solución extensamente estudiados, estos son: el algoritmo de Prim-Dijkstra [43] y el algoritmo de Kruskal [30]. En el primero, el algoritmo comienza con un solo nodo y en cada paso incorpora la arista libre de menor peso y al nodo al otro extremo de ésta. El algoritmo concluye cuando ha incluido a todos los vértices del grafo. En el segundo, el algoritmo comienza con todos los nodos y va agregando de forma sucesiva la arista libre de menor peso que no genere un ciclo en el árbol resultante. Esta operación se repite hasta construir el árbol generador de costo mínimo.

No fue sino hasta los años 80 que se comenzó a estudiar el caso desde la perspectiva de la computación distribuida. El algoritmo de Gallager, Humblet, y Spira (GHS) [23] es uno de los más representativos en esta área. Este algoritmo puede verse como una generalización de la versión centralizada del algoritmo de Kruskal. Mientras que éste último permite el crecimiento de varios subárboles de forma paralela en distintas regiones del grafo, el algoritmo es aún secuencial puesto que en cada paso solo se combinan dos de ellos. Por el contrario, el algoritmo GHS permite por un lado el paralelismo de las combinaciones entre subárboles y por otro, realizar la tarea de construcción del árbol generador de peso mínimo de forma distribuida.

En este capítulo se presentarán algunas de las soluciones al problema del árbol generador de costo mínimo. Por razones pedagógicas, primero se abordarán las versiones centralizadas de los algoritmos Prim-Dijkstra y Kruskal y después la versión distribuida del algoritmo GHS. Con ello buscamos abordar en forma gradual las limitaciones que deben superarse para construir una solución eficiente. La notación utilizada en este capítulo se presenta en la tabla 4.1.

4.2. Condiciones del problema

A continuación presentamos algunas definiciones que nos permitirán establecer de forma precisa el problema del árbol generador de costo mínimo. La red de comunicaciones se modela a través de un grafo, $G = (V,E)$, no dirigido y conexo. El número de vértices es $n = |V|$ y el número de aristas es $m = |E|$. Cada arista $e \in E$ tiene asociado un peso $w(e)$. La arista entre los nodos u y v se denota como uv o vu. Un *árbol generador* de G es un subgrafo, conexo y sin ciclos, que conecta todos los vértices de V; su costo es la suma de los pesos de las aristas que lo componen. Un *árbol generador de costo mínimo* (AGM(G) o simplemente AGM), es aquel cuyo costo es el más pequeño entre todos los posibles árboles generadores de G. Un *fragmento*, es un subárbol del AGM. Una arista uv se dice *arista saliente* de un fragmento F si $u \in F$ y $v \notin F$, o viceversa (i.e., $v \in F$ y $u \notin F$). La *arista saliente de peso mínimo* es la arista saliente de un fragmento F cuyo peso es el menor de entre todas las aristas salientes.

La figura 4.1 ilustra estos conceptos. En 4.1(a) presentamos un grafo cuyas aristas tienen un peso asociado. Por ejemplo, el peso de la arista v_1v_2 es igual a 40. (i.e., $w(v_1v_2) = 40$). En la figura 4.1(b), las líneas remarcadas (aristas) y cada par de vértices que estas unen conforman un árbol generador cuyo costo es igual a 210 (i.e., $w(v_1v_2)+w(v_1v_3)+w(v_2v_5)+w(v_5v_6)+w(v_6v_4) = 60+40+80+20+10 = 210$). En la figura 4.1(c) se presenta el AGM. Las partes sombreadas representan dos fragmentos (F, F') del AGM. F tiene 4 aristas salientes (i.e., v_1v_3, v_2v_3, v_2v_5, y v_2v_4) de las cuales la arista v_2v_3 es la de menor peso. F' tiene 3 aristas salientes donde la arista v_5v_3 es la de menor peso.

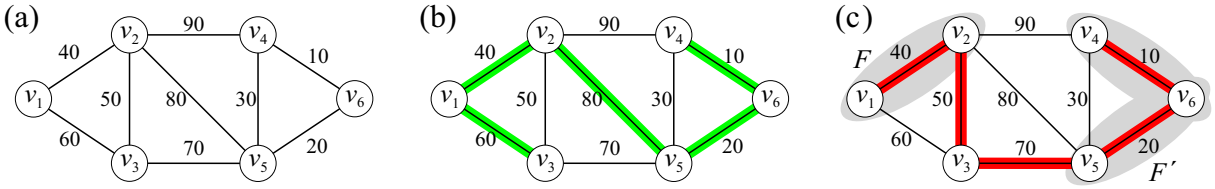


Figura 4.1: (a) Grafo ponderado (b) Árbol generador (c) Fragmentos y árbol generador mínimo.

Dado G, el problema del árbol generador de costo mínimo consiste en construir su AGM. Los algoritmos que presentamos en este capítulo (entre otros) dan solución a este problema. En general, en estas soluciones subyacen dos ideas que son el punto de partida, y que se expresan en los siguientes lemas:

Lema 4.1. *Si las aristas de un grafo conexo $G = (V,E)$ tienen pesos diferentes, entonces G tiene un AGM único.*

Demostración. Supóngase, por contradicción, que G tiene dos árboles generadores de costo mínimo $T_1 = (V_1,E_1)$ y $T_2 = (V_2,E_2)$. Ahora, ordenamos de forma decreciente los pesos de las aristas que aparecen en cada uno de los árboles. Sea e la arista de costo mínimo que solo pertenece a uno de ellos. Sin pérdida de generalidad, e pertenece a E_1 y no pertenece a E_2. Entonces, $E_2 \cup \{e\}$ debe contener un ciclo y al menos una arista de este ciclo, e', no se encuentra en E_1 puesto que T_1 no contiene ciclos. Como e' está en E_2 y no en E_1 y como los pesos de las aristas son distintos, se sigue que $w(e) < w(e')$. En consecuencia, $E_2 \cup \{e\} - \{e'\}$ representa un árbol generador con un costo menor que el de T_2 (!). □

Lema 4.2. *Sea $G = (V,E)$ un grafo conexo cuyas aristas se ponderan con pesos distintos y sea $T = (V,E')$ su único AGM. Para cualquier fragmento F de T, la arista saliente mínima de F estará en T.*

Demostración. Supóngase, nuevamente por contradicción, que existe un fragmento $F = (V_1,E_1)$ en $T = (V,E')$ cuya arista saliente mínima e, no pertenece a E'. Entonces, $E' \cup \{e\}$ contiene un ciclo. Este ciclo posee al menos, otra arista de F, denominada e'. Puesto que e tiene el peso más pequeño entre todas las aristas salientes de F, se sigue que $w(e') > w(e)$. En consecuencia, $E' \cup \{e\} - \{e'\}$ forma un árbol generador con un costo menor que el de T(!). □

4.3. Algoritmo de Prim-Dijkstra

Este algoritmo construye un AGM comenzando con un fragmento constituido por un solo nodo, u, y extiende este fragmento al incorporar de forma sucesiva una arista a la vez. Más precisamente, considere el conjunto de vértices $V' \subseteq V$ y el conjunto de aristas $E' \subseteq E$ sobre los cuales se construye el árbol generador de costo mínimo $T = (V',E')$. Inicialmente $V' = \{u\}$. Mientras $V' \neq V$, el árbol T crece al agregar la arista saliente $e = v_i v_j$ en E' y su extremo v_j en V' si y solo si e minimiza el peso $\min_{e=v_i v_j:\, v_i \in T} w(e)$.

Ilustramos el algoritmo de Prim sobre el grafo que utilizamos como ejemplo en la figura 1.1 de la sección anterior. En la figura 4.2(a), se comienza con un fragmento que contiene solo al nodo v_1 (i.e., $V' = \{1\}$ y $E' = \emptyset$). El fragmento tiene dos aristas salientes, v_1v_2 y v_1v_3. El algoritmo agrega la arista saliente de menor peso (i.e., $e = v_1v_2$), y al nodo v_2 (figura 4.2(b)). En este punto el fragmento está constituido por $V' = \{1,2\}$ y $E' = \{v_1v_2\}$. Ahora, las aristas salientes del fragmento son v_1v_3, v_2v_3, v_2v_5, y v_2v_4. El algoritmo agrega la arista saliente de menor peso (i.e., v_2v_3) y al nodo v_3 (figura 4.2(c)). En las próximas 3 iteraciones el algoritmo agrega las aristas v_3v_5, v_5v_6, y v_6v_4 a E' y los vértices v_5, v_6, y v_4 a V' (en ese mismo orden). Las figuras 4.2(d), (e), y (f) muestran los detalles de estas iteraciones. En la figura 4.2(f) se presenta el AGM.

El análisis de tiempo de ejecución del algoritmo de Prim-Dijkstra considera lo siguiente: (i) el tiempo requerido en el crecimiento del fragmento hasta completar el AGM, y (ii) el tiempo requerido en buscar la arista saliente de peso mínimo cada vez que el fragmento crece. Puesto que el algoritmo agrega una

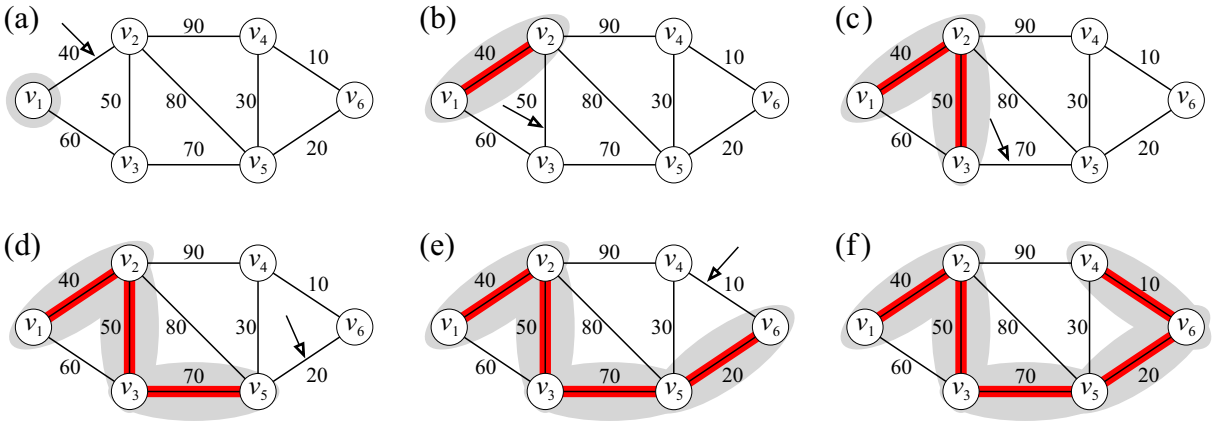


Figura 4.2: Ejemplo de ejecución del algoritmo de Prim-Dijkstra.

arista en cada iteración, el número de pasos requerido en (i) es $O(n)$, esto es porque el AGM contiene $n-1$ aristas en total. En cada una de estas iteraciones se busca la arista saliente de peso mínimo del fragmento sobre el conjunto de aristas E por lo cual (ii) requiere un número de pasos de $O(m)$. En consecuencia, el tiempo total requerido por el algoritmo es $O(mn)$.

4.4. Algoritmo de Kruskal

El algoritmo de Kruskal construye un AGM comenzando con fragmentos que solo contienen un nodo, en consecuencia, inicialmente se tienen n fragmentos. Después, estos fragmentos se combinan de forma sucesiva dando lugar a fragmentos de mayor orden. En cada iteración, un par de fragmentos es unido por la arista libre cuyo peso es el menor. Concretamente, considere el conjunto de vértices $V' = V$ y conjunto de aristas $E' \subseteq E$ sobre los cuales se construye de forma sucesiva el árbol generador de costo mínimo $T = (V', E')$. Inicialmente $E' = \emptyset$. Primero, el algoritmo ordena todas las aristas en E tal que $w(e_i) < w(e_{i+1})$. Después, mientras recorre las aristas en este orden, inserta la arista e_i en E' si y solo si e_i no genera un ciclo con las aristas ya existentes en E'. En otro caso, simplemente descarta e_i y continua con e_{i+1}. El algoritmo termina cuando $|E'| = n-1$.

Utilizamos el mismo grafo de la figura 1.1 para ilustrar el algoritmo de Kruskal. En la figura 4.3(a) el algoritmo comienza con 6 fragmentos y ordena las aristas de menor a mayor peso (i.e., v_4v_6, v_6v_5, v_4v_5, v_1v_2, v_2v_3, v_1v_3, v_3v_5, v_2v_5, y v_2v_4). En las dos iteraciones siguientes, inserta las aristas v_4v_6, v_6v_5 en E' (véanse las figuras 4.3(b) y (c)). En este punto $V' = V$ y $E' = \{v_4v_6, v_6v_5\}$. En la próxima iteración, el algoritmo descarta la arista v_4v_5 ya que ésta forma un ciclo con las ya existentes en E'. Después, continua su ejecución e inserta las aristas v_1v2 y v_2v3 en las siguientes dos iteraciones (véanse las figuras 4.3(d) y (e)) y descarta la arista v_1v_6 en la siguiente. El algoritmo concluye al agregar la arista v_3v_5 pues en este punto $|E'| = n-1$. En la figura 4.3(f) se presenta el AGM.

El análisis de tiempo de ejecución del algoritmo de Kruskal considera lo siguiente: (i) el tiempo requerido en ordenar las aristas del grafo G, y (ii) el tiempo requerido en detectar los ciclos. Puesto que la red cuenta con m aristas el número esperado de pasos requeridos por un algoritmo de ordenamiento

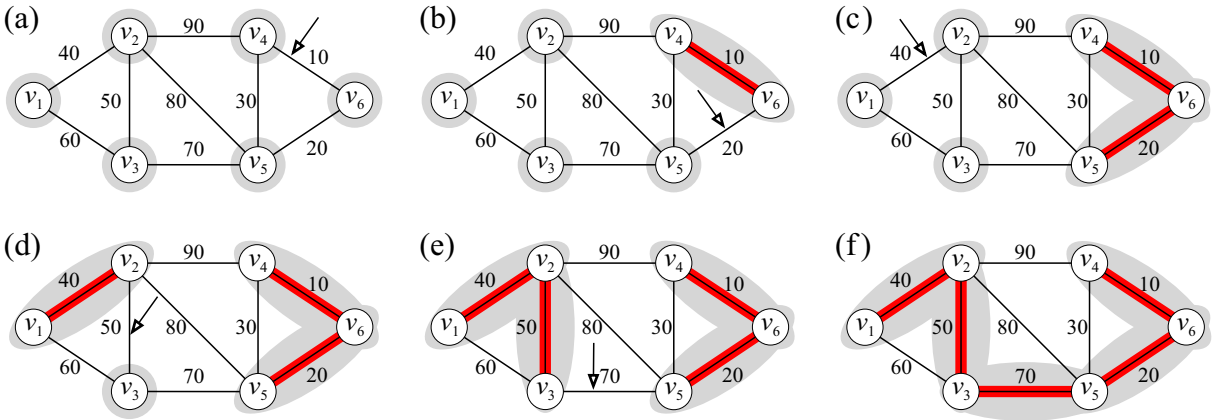


Figura 4.3: Ejemplo de ejecución del algoritmo de Kruskal.

será de $O(m \log m)$ (p.ej., Quick-Sort). El tiempo requerido para detectar ciclos dependerá del método utilizado. Por ejemplo, una forma simple de verificar que $e = (u,v)$ no genera un ciclo es: agregar e al fragmento F si y solo si $u,v \notin F$. Otro método, consiste en ejecutar un algoritmo BFS (Breadth-First Search) sobre F cada vez que se intenta agregar una arista e. Si el algoritmo BFS encuentra un nodo ya visitado en F, entonces descarta e, pues en este caso se ha encontrado un ciclo. Este último método requiere un tiempo de $O(n)$ por cada arista que se agrega al AGM. En consecuencia, el tiempo de ejecución del algoritmo es $O(nm)$.

4.5. El algoritmo GHS

El algoritmo GHS puede verse como una generalización de la versión centralizada del algoritmo de Kruskal. Si bien es cierto que éste último podría aceptar el crecimiento de varios fragmentos de forma simultánea, el algoritmo es aún secuencial puesto que en cada paso combina un par de fragmentos a la vez. En contraste, el algoritmo GHS permite el crecimiento simultáneo de varios fragmentos.

Para lograr su cometido, en el algoritmo GHS, cada nodo que arranca espontáneamente se considera a sí mismo un fragmento independiente. Después, cada fragmento busca su arista saliente de peso mínimo y cuando la encuentra intenta combinarse con el fragmento en el otro extremo de la misma. Algunos fragmentos lograrán de inmediato esta acción de crecimiento, mientras otros deben esperar. Esto dependerá del estado actual de cada fragmento, i.e., las etapas de crecimiento por las que haya atravesado, así como del estado de aquellos fragmentos con los que intente combinarse. Sin embargo, el resultado final será un solo fragmento i.e., el AGM.

El algoritmo distribuido toma en cuenta las siguientes condiciones:

- Existe un solo proceso por cada nodo del grafo G.
- Cada nodo en G ejecuta el mismo algoritmo, conoce el peso de sus aristas incidentes, y solo puede comunicarse con los nodos en el otro extremo de éstas.
- Inicialmente, todos los nodos se encuentran en un estado de reposo. Uno o más nodos arrancan su ejecución local del algoritmo al despertar de forma espontánea o después de la recepción de un mensaje.
- Los mensajes siguen una política FIFO y pueden ser transmitidos de forma independiente en ambas direcciones del canal. Se asume que no hay pérdida, ni errores en los mensajes, y que estos llegan a su destino en un tiempo indeterminado pero finito

Al aplicar conceptos probados en modo secuencial para la solución distribuida del problema del AGM se observan los siguientes retos:

- *Reto 1:* ¿Cómo y cuándo se combinan dos fragmentos?
  El orden del fragmento determina los costos asociados con la búsqueda de la arista saliente de peso mínimo. Según crezca el número de nodos incorporados al fragmento, tomará más tiempo y mensajes completar este paso. Por otro lado, el crecimiento de los fragmentos es un proceso dinámico y habrá nodos que actualicen su información antes que otros. En consecuencia, es posible que dos nodos se enteren en instantes distintos que pertenecen al mismo fragmento. Deben evitarse los riesgos que

pudieran surgir de este problema de sincronización. También es necesario considerar las condiciones bajo las que es conveniente la combinación de fragmentos con órdenes muy dispares.

- *Reto 2:* ¿Cómo determinar la arista saliente de peso mínimo de un fragmento?
  Si se trata de un nodo que acaba de despertar i.e., un fragmento de orden mínimo, el procedimiento es relativamente fácil porque éste elige a su arista incidente de menor peso. En cualquier otra circunstancia, el fragmento está formado por más de un nodo que dispone de información parcial. Dicho de otra forma, cada nodo reconoce a sus aristas incidentes que ya fueron incorporadas al fragmento, pero puede ignorar si el resto de sus aristas salen del fragmento o lo conectan con otros nodos del mismo subgrafo. Se requiere, por un lado, que cada nodo sea capaz de identificar la condición de cada una de sus aristas incidentes y, por otro, que pueda coordinarse con los demás para elegir solo una, la arista saliente de peso mínimo del fragmento.

4.5.1. Combinación de fragmentos

Al arrancar su ejecución local, un proceso se considera a sí mismo un fragmento individual F, al que se asocia un entero L denominado *nivel*, con un valor inicial 0. En cualquier etapa posterior, un fragmento F es un subárbol del AGM que, además de contar con un nivel, se caracteriza por una arista única, e, llamada *núcleo* o *arista principal*. Juntos, el nivel y el peso del núcleo, le confieren a F una identidad $(L, w(e))$. Los nodos adyacentes de la arista principal, o *nodos principales*, coordinan la actividad del fragmento.

Continuemos ahora con las reglas que deben seguirse para la combinación de fragmentos. Sean F y F' dos fragmentos del AGM, con identidades (L, w) y (L', w'), respectivamente. La combinación entre ellos puede realizarse de dos maneras alternativas: por *fusión* o por *absorción*. Esto dependerá de la relación entre sus respectivos niveles. Más precisamente, la combinación entre F y F' obedece las siguientes reglas:

1. *Combinación por fusión.* Si $L = L'$ y ambos fragmentos comparten la misma arista saliente de peso mínimo, e, entonces, el nuevo fragmento F'' tendrá nivel $L+1$ y núcleo e. En otras palabras, F y F' se fusionarán en un nuevo fragmento, cuyo núcleo será la arista saliente de peso mínimo que los une, y su nivel se incrementará en una unidad.
2. *Combinación por absorción.* Si $L > L'$ y F' comunica a F su intención de combinarse, entonces el nuevo fragmento F'' tendrá la misma identidad que F (i.e., (L, w)). Esto es, el fragmento F' será absorbido por el fragmento F, en cuyo caso el nivel y el núcleo de F se propagará sobre todos los nodos en F'.

En la figura 4.4 se presenta un ejemplo que ilustra la aplicación de estas reglas. Inicialmente cada fragmento tiene nivel 0 (figura 4.4(a)). Un tiempo después, hay dos fragmentos F y F', ambos con nivel 1 y núcleos v_1v_2 y v_4v_6, respectivamente. Ambos fragmentos son el resultado de la combinación por fusión (figura 4.4(b)). Luego, estos fragmentos continúan creciendo al absorber F al nodo v_3 y F' al nodo v_5 (figura 4.4(c)). Finalmente, F y F' se fusionan dando lugar a un fragmento de nivel 2 (figura 4.4(d)) y núcleo v_3v_5.

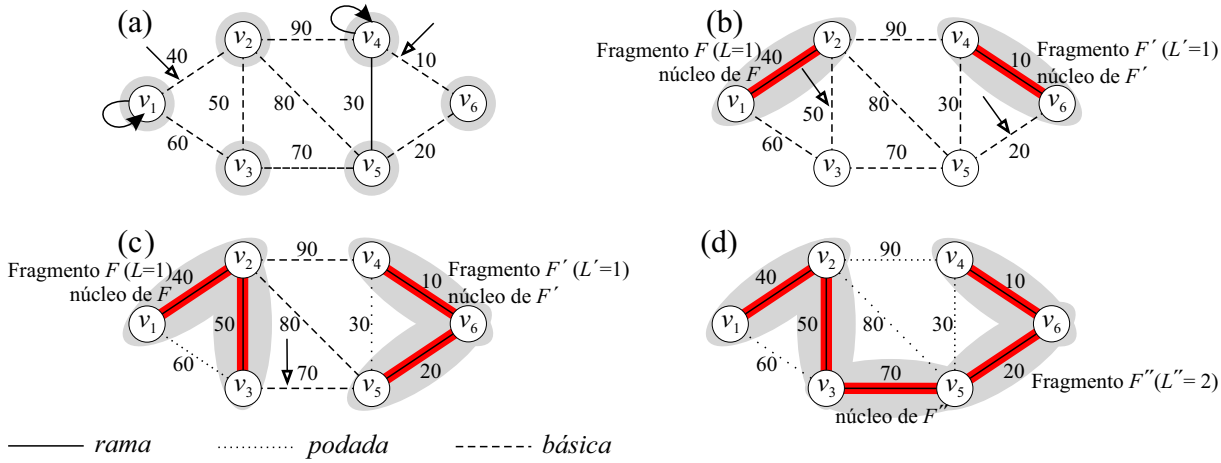


Figura 4.4: Ejemplo de ejecución del algoritmo GHS. (a) Todos los fragmentos tienen nivel 0. Los nodos v_1 y v_4 despiertan espontáneamente y arrancan su ejecución local del algoritmo. (b) y (d) ilustran la combinación por fusión y (c) la combinación por absorción.

Más adelante revisamos en detalle el conjunto completo de mensajes que se intercambian en los diferentes momentos del algoritmo. Sin embargo, de manera preliminar, adelantamos aquellos mensajes utilizados durante la combinación de dos fragmentos. Sean $i \in F(L,w)$ y $j \in F'(L',w')$, dos nodos adyacentes cuyos fragmentos han determinado unirse a través de la arista ij que ambos comparten. En la combinación por fusión los nodos intercambian los mensajes `CONECTA`(L) y `CONECTA`(L'), respectivamente. Este mensaje es la solicitud de conexión que hace un nodo cuando el fragmento ha encontrado la arista saliente de peso mínimo. En tanto, para la combinación por absorción, el nodo j envía primero el mensaje `CONECTA`(L') al nodo i que responde con un mensaje `INICIO`$(L, w(e), s)$.

4.5.2. Buscando la arista saliente de peso mínimo de un fragmento

En esta sección revisaremos el conjunto de actividades que desarrolla un fragmento para encontrar su arista saliente de peso mínimo. Comenzamos presentando algunos conceptos que usaremos en la exposición.

Estados de un nodo. Según la etapa del algoritmo que ejecute, un nodo puede encontrarse en uno de tres estados posibles: *dormido*, *búsqueda*, y *reporte*. El primero es el estado inicial, el segundo se presenta cuando un nodo busca su mejor arista saliente (i.e., la de menor peso), y el tercero cuando ya la encontró y está reportando el peso de la misma.

Tipos de aristas. Cada nodo clasifica sus aristas incidentes en uno de los siguientes tres tipos: *rama*, *podada*, y *básica*. El primero indica si la arista está incorporada al fragmento, el segundo si se sabe que incide sobre dos nodos que pertenecen al mismo fragmento, y el tercero si aún no se ha explorado. Inicialmente todas las aristas son clasificadas como básicas. En la figura 4.4(a) todas las aristas son *básicas*. A medida que el algoritmo se ejecuta, los nodos van explorando sus aristas y reclasificándolas.

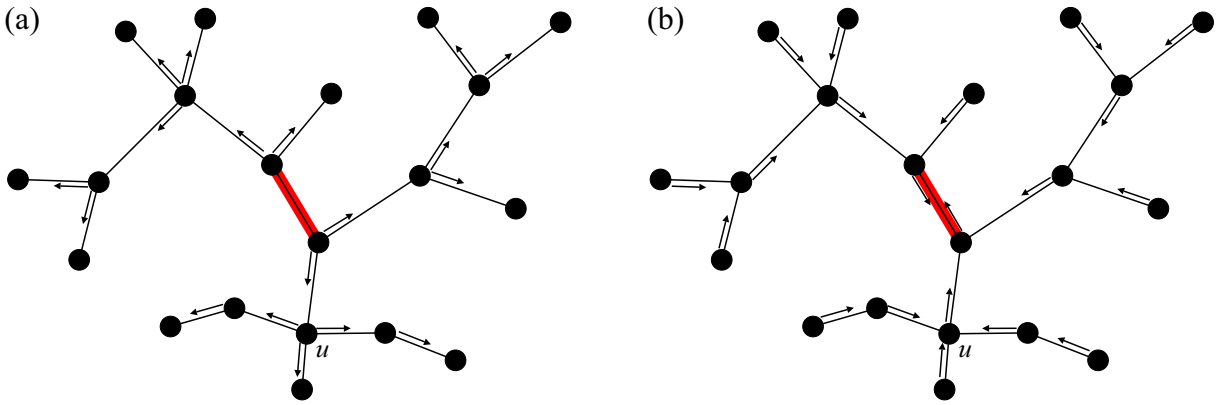


Figura 4.5: Operación de propagación (a) y retroalimentación (b) en un fragmento.

Por ejemplo, en la figura 4.4(c) las aristas v_1v_2, v_2v_3, v_4v_6 y v_6v_5 son ramas, las aristas v_2v_4, v_2v_5, y v_3v_5 son básicas, y las aristas v_1v_3 y v_4v_5 son podadas. Al final de la ejecución del algoritmo todas las aristas son ramas o podadas (p.ej., figura 4.4(d))

Propagación y retroalimentación en un fragmento. Los nodos principales de un fragmento coordinan el crecimiento del mismo, esto es, se encargan de determinar la siguiente arista por la que el fragmento debe combinarse. Para conseguirlo deben lograr que cada nodo bajo su administración encuentre y reporte a su arista básica de menor peso, de entre las cuales elegirán a la arista mínima del fragmento. Imaginemos una operación de propagación de información con retroalimentación, como la que explicamos en un capítulo precedente. Durante la propagación, los nodos principales envían un mensaje hacia todos los nodos para llevarlos al estado de *búsqueda*. Luego, en la fase de retroalimentación, los nodos cambian al estado de *reporte*, y su información remonta el fragmento hasta llegar de regreso a los nodos principales.

La figura 4.5 ilustra la propagación (a) y retroalimentación (b) de información en un fragmento. La línea remarcada representa el núcleo del fragmento y los nodos adyacentes a ésta son los nodos principales. Para señalar un ejemplo en particular, centremos nuestra atención en el nodo u. En la propagación, u reexpide el mensaje que proviene de un nodo principal (véase la figura 4.5(a)). En la retroalimentación, u reexpide un mensaje que remonta el fragmento hacia su nodo principal, pero solo hasta que tiene la certeza que no recibirá más mensajes de este tipo (véase la figura 4.5(b)).

Procedamos ahora con la descripción completa del conjunto de actividades que desarrolla un fragmento para encontrar su arista saliente de menor peso. En general, esta tarea puede describirse como una secuencia de tres etapas: *inicialización, identificación*, y *selección*. La primera, coincide con la operación de propagación que recién presentamos, y tiene el doble propósito de actualizar en cada nodo la información del fragmento al que pertenece, y ponerlo en un estado de *búsqueda*. En la segunda etapa, cada nodo determina su arista básica de menor peso (si acaso tiene alguna). Por último, la tercera etapa, que coincide con la operación de retroalimentación, consiste en reunir toda la información procedente de los nodos, a fin de seleccionar a la arista por donde el fragmento intentará seguir creciendo.

4.5.2.1. Iniciando la búsqueda

Recordemos que todo fragmento con un nivel mayor que cero puede originarse de dos formas alternativas: ya sea como resultado de la fusión de dos fragmentos, o bien, como resultado de un fragmento que absorbió a otro. Para el primer caso, supongamos que el fragmento F con nivel L acaba de formarse a partir de dos fragmentos con nivel $L-1$ que coincidieron en la misma arista saliente de peso mínimo e. El núcleo de F es ahora e y su peso es utilizado para darle identidad al fragmento. Los nodos adyacentes al núcleo comienzan un nuevo ciclo al propagar un mensaje $\texttt{INICIO}(L, w(e), s)$, como describimos anteriormente. Para el segundo caso, supongamos un nodo $i \in F$ que recibe un mensaje $\texttt{CONECTA}(L')$ desde el nodo j en el fragmento F'. Si el nivel $L > L'$ entonces i manda un mensaje $\texttt{INICIO}(L, w(e), s)$ a j que continua con la propagación del mensaje sobre el fragmento F'.

Cada nodo receptor del mensaje $\texttt{INICIO}(L, w(e), s)$ comienza la búsqueda de su mejor arista saliente (como describiremos más adelante) solo si $s =$ *búsqueda*. Los dos primeros argumentos de este mensaje, L y $w(e)$, proveen a cada nodo receptor con la información de identidad del fragmento que acaba de formarse, y el tercero, es una variable que permite que los nodos cambien su estado actual a *búsqueda* o $reporte$.

Asimismo, cada nodo del fragmento reconoce a una arista de tipo rama, a la que denominaremos ascendente, a través de la que recibe el último mensaje de tipo $\texttt{INICIO}(L, w(e), s)$. Si fuera el caso, el mismo nodo propaga el mensaje por sus otras ramas, a las que denominaremos descendentes.

4.5.2.2. Identificando aristas salientes

Como mencionamos, la arista saliente uv de un nodo $u \in F(L,w)$ es aquella cuyo extremo, v, no está en F. Si $L = 0$ (i.e., F solo tiene un nodo) entonces, u simplemente elige de entre sus aristas incidentes a la de menor peso. Por otro lado, si $L > 0$, es necesario desarrollar un mecanismo a través del cual u pueda identificar a su arista saliente de menor peso. Por las premisas del modelo, sabemos que u desconoce cuáles de sus aristas, inicialmente clasificadas como básicas, lo conectan con otros nodos que también yacen en F y que, por tanto, deberían reclasificarse como podadas.

Cuando $L > 0$, u selecciona a su arista de menor peso, en estado básico, y envía un mensaje hacia el otro extremo de la misma. Si, a partir de la respuesta que recibe, u reconoce que el otro nodo no está en su fragmento, entonces habrá concluido su tarea. En otro caso, seleccionará la siguiente arista básica de menor peso. Esta operación se repite hasta encontrar una arista saliente o agotar sus posibilidades.

En detalle, un nodo, $u \in F$, obedece las siguientes reglas para identificar su arista saliente de menor peso:

1. Si $L = 0$, entonces u selecciona la arista incidente de menor peso.
2. Si $L > 0$, entonces u selecciona de entre sus aristas en estado *básico* a la de menor peso y envía un mensaje $\texttt{PRUEBA}(L,w)$ al nodo al otro extremo de ésta. Si los identificadores en ambos fragmentos coinciden (figura 4.6(a)) entonces, el nodo receptor (v) responde con un mensaje $\texttt{RECHAZO}$. Luego, u continúa probando su siguiente mejor arista. Si los identificadores son diferentes y el nivel de u es menor o igual que el nivel de v (figura 4.6(b)), entonces v responde con un mensaje $\texttt{ACEPTA}$. En otro caso, v retarda su respuesta hasta que la condición $(L > L')$ se cumpla (figura 4.6(c)).

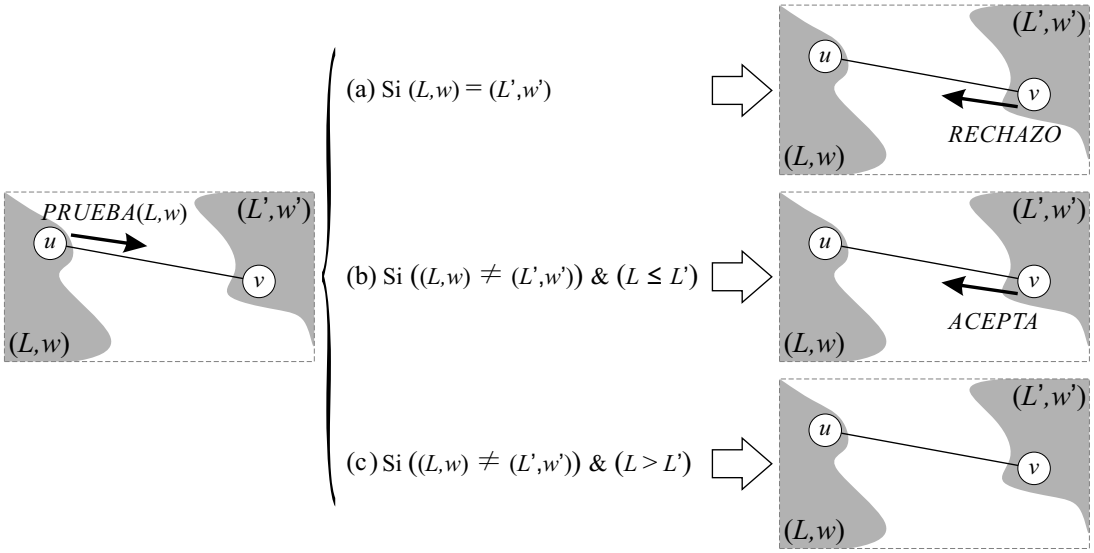


Figura 4.6: Secuencia de operaciones que se siguen después de la transmisión de un mensaje `PRUEBA`

4.5.2.3. Seleccionando la mejor arista saliente

Es importante recordar que todo fragmento es un árbol. Entonces, las operaciones de inicialización y selección coinciden con la propagación y retroalimentación de información sobre esta misma estructura. La primera transporta información desde los nodos principales hacia las hojas, mientras la segunda comienza en las hojas del fragmento y remonta el árbol hasta concluir en los nodos principales. Como mencionamos, el objetivo de esta última etapa es determinar cuál es la arista saliente de peso mínimo en el fragmento.

Ahora, en la etapa de selección, un nodo hoja envía un mensaje `REPORTE`(W) sobre su rama ascendente. El argumento W es el peso de su mejor arista saliente y puede ser infinito si no tiene alguna.

De forma similar, un nodo en el interior del fragmento envía un mensaje `REPORTE`(W) sobre su rama ascendente, pero solo hasta que ha identificado su propia arista saliente de menor peso y ha recibido un mensaje desde cada una de sus ramas descendentes. Debemos observar en este caso, que el argumento W corresponde con el menor peso del cual se tiene noticia. Dicho de otra manera, el menor valor que el nodo ha conocido de manera directa o mediante un mensaje.

Después de enviar su mensaje `REPORTE`, un nodo cambia su estado a *reporte* para indicar que su búsqueda ha concluido. Así también, registra en una variable local denominada como *mejor_peso*, el menor peso encontrado. Por último, el nodo denota como *mejor_arista*, a aquella arista saliente cuyo peso envió en un mensaje, o bien a aquella rama por la que recibió el mensaje que transportaba el *mejor_peso*.

Por último, los nodos principales también intercambian mensajes de tipo `REPORTE`. Hay que observar que el fragmento puede considerarse formado por dos subárboles cuyas raíces se encuentran conectadas mediante la arista principal. Cuando un nodo principal concluye su selección y recibe un mensaje `REPORTE`(W) a través del núcleo del fragmento, verifica si el valor W es mayor que su *mejor_peso*. De ser así, sabe que la arista saliente de peso mínimo del fragmento está de su lado. Luego, envía el mensaje `UNION` a través del camino formado por la *mejor_arista* que cada nodo registró. Este camino conecta

Algoritmo 4.1: Algoritmo de Gallager, Humblet y Spira (GHS), en el nodo *i*. Parte 1/3

```
   /*Solo se ejecuta una vez.*/
 1 procedimiento despierta()
 2     sea k la arista adyacente de peso mínimo
 3     tipo(k) ← rama
 4     nivel ← 0
 5     estado ← reporte
 6     cuenta ← 0
 7     envía CONECTA(nivel) por k
   /*Comienza la fase de retropropagación, cuando tiene noticia de todos sus hijos y de
     su propia arista básica de peso mínimo*/
 8 procedimiento reporta()
 9     si (cuenta = 0 & candidato = nulo) entonces
10         estado ← reporte
11         envía REPORTE(mejor_peso) por padre
   /*Prueba si existe su arista básica, en otro caso intenta arrancar la fase de
     retropropagación.*/
12 procedimiento busca_AMF()
13     si (existen aristas adyacentes de tipo básico) entonces
14         candidato ← la arista básica de peso mínimo
15         envia PRUEBA(nivel, peso_núcleo) por candidato
16     otro
17         candidato ← nulo
18         reporta()
   /*Envía una orden hasta el nodo con la AMF, para que la reclasifique e inicie el
     procedimiento de unión.*/
19 procedimiento cambia_raíz()
20     si (tipo(mejor_arista) = rama) entonces
21         envía UNION por mejor_arista
22     otro
23         envía CONECTA(nivel) por mejor_arista
24         tipo(mejor_arista) ← rama
```

al nodo principal con el nodo que tiene la arista saliente de peso mínimo. Cuando el mensaje alcanza su destino final, i.e., el nodo sobre el que incide la arista mínima del fragmento, éste envía un mensaje $\texttt{CONECTA}(L)$ a través de su *mejor_arista* (L es el nivel actual del fragmento al que pertenece). Por otro lado, si ($W = mejor\_peso = \infty$) entonces no hay ninguna arista saliente y el AGM es el fragmento mismo.

4.5.3. Los mensajes que soportan al algoritmo

A lo largo del algoritmo se utilizan los siguientes mensajes:

`ACEPTA`: Un nodo responde con este mensaje, si pertenece a un fragmento distinto de aquel al que pertenece el nodo del que recibió un mensaje de $\texttt{PRUEBA}(L, w)$.

Algoritmo 4.2: Algoritmo de Gallager, Humblet y Spira (GHS), en el nodo i. Parte 2/3

```
   /*Dependiendo de la relación entre los niveles del fragmento que manda y del que
     recibe, se da lugar a una operación de absorción o de fusión, esta última solo
     ocurre de común acuerdo.*/
 1 al recibir CONECTA(L) desde j efectúa
 2     si (estado = descanso) entonces despierta()
 3     si (L < nivel) entonces
 4         tipo(j) ← rama
 5         envía INICIO(nivel, peso_núcleo, estado) por j
 6         si (estado = búsqueda) entonces
 7             cuenta ← cuenta + 1
 8     otro
 9         si (tipo(j) = básica) entonces
10             guarda mensaje en cola de pendientes
11         otro
12             envía INICIO(nivel+1, w(j), búsqueda) por j

       /*El nodo que recibe el mensaje asume la identidad y el estado que contiene este
         mismo.*/
13 al recibir INICIO(L,w,s) desde j efectúa
14     nivel ← L
15     peso_núcleo ← w
16     estado ← s
17     padre ← j
18     mejor_arista ← nulo
19     mejor_peso ← infinito
20     para toda arista j' ≠ j , tipo(j') = rama haz
21         envía  INICIO(L,w,s) sobre j'
22         si (s = búsqueda ) entonces cuenta ← cuenta + 1
23     si (s = búsqueda ) entonces
24         busca_AMF()
   /*El nodo que envía este mensaje pertenece al mismo fragmento del nodo que lo recibe,
     este último debe buscar un nuevo candidato.*/
25 al recibir RECHAZO desde j efectúa
26     si (tipo(j) = básica) entonces tipo(j) ← podada
27     busca_AMF()
   /*un mensaje de la fase de retropropagación que remonta el árbol*/
28 al recibir REPORTE(w) desde j efectúa
29     si (j ≠ padre) entonces
30         cuenta ← cuenta − 1
31         si (w < mejor_peso) entonces
32             mejor_peso ← w; mejor_arista ← j
33             reporta()
34     otro
35         si (estado = búsqueda) entonces guarda mensaje en cola de pendientes
36         otro
37             si (w > mejor_peso) entonces cambia_raíz()
38             otro si (w = mejor_peso = ∞) entonces termina
```

Algoritmo 4.3: Algoritmo de Gallager, Humblet y Spira (GHS), en el nodo *i*. Parte 3

```
   /*Solo se responde si el nodo que recibe pertenece a un fragmento con un nivel mayor
     o igual que el del fragmento del nodo que envía.*/
 1 al recibir PRUEBA(L,w) desde j efectúa
 2     si (estado = descanso) entonces despierta()
 3     si (L > nivel) entonces guarda mensaje en cola de pendientes
 4     otro
 5         si (w ≠ peso_núcleo) entonces envía ACEPTA por j
 6         otro
 7             si (tipo(j) =básica) entonces tipo(j) ← podada
 8             si (candidato ≠ j) entonces envía RECHAZO por j
 9             otro busca_AMF()
   /*El nodo que envía este mensaje pertenece a un fragmento diferente del nodo que lo
     recibe. Este último actualiza el peso y el camino que lo conecta con la arista
     básica de menor peso, de la que tiene noticia.*/
10 al recibir ACEPTA desde j efectúa
11     candidato ← nulo
12     si (w(j) < mejor_peso) entonces
13         mejor_arista ← j
14         mejor_peso ← w(j)
15     reporta()
```

`CONECTA`(L): Un nodo envía este mensaje hacia el otro extremo de su arista saliente mínima, como una petición para formar un nuevo fragmento.

`INICIO`(L,w,s): Tan pronto como se forma un nuevo fragmento, o se actualiza su identificador, los nodos de la arista principal, o nodos principales, envían este mensaje a todo el fragmento, para iniciar la búsqueda de la siguiente arista saliente mínima. L es el nivel del fragmento, w es el peso de la arista principal y s el estado de los nodos principales. Por ser el primer mensaje que cruza la arista con que se unen dos fragmentos, hasta entonces independientes, también se le puede interpretar como la confirmación o reconocimiento de la unión de estos.

`PRUEBA`(L,w): Un nodo en estado de *búsqueda*, envía este mensaje a través de su arista *básica* de menor peso, para averiguar si lo conecta con otro fragmento, es decir, si se trata de su arista saliente mínima.

`RECHAZO`: Un nodo responde con este mensaje, si pertenece al mismo fragmento que el nodo del que recibió un mensaje de `PRUEBA`(L,w).

`REPORTE`(w): En el árbol que constituye a un fragmento, un nodo devuelve este mensaje a su padre, para reportarle el peso de la arista saliente de menor costo, de la que tiene conocimiento. No se trata necesariamente de una arista que incida en el nodo mismo, sino que puede hallarse en alguno de sus descendientes. Este reporte remonta el árbol hasta llegar al núcleo del fragmento, donde se concentra la información y se elige la siguiente AMF.

`UNION`: Un nodo principal envía este mensaje al nodo en que incide la arista saliente mínima del fragmento, para indicarle que debe iniciar el proceso de unión, con el fragmento en el otro extremo de la arista.

Además de su *estado* y su propio identificador *i*, el nodo mantiene también una colección local de variables, donde registra el desarrollo del algoritmo:

candidato: la arista adyacente a *i*, de tipo básica, con el menor peso. Inicialmente es nula.
cuenta: registra el número de mensajes de tipo `REPORTE` que aún están pendientes de recibirse. Inicialmente vale 0.
mejor_arista: acerca al nodo hacia su descendiente sobre el que incide la arista con el menor peso. Inicialmente es nula
mejor_peso: el menor peso del que el nodo tiene noticia. Inicialmente tiene un peso infinito.
nivel: describe el nivel del fragmento al que pertenece el nodo. Inicialmente es 0.
padre: el nodo a través del que puede enviar un mensaje hacia el núcleo del fragmento, al que pertenece. Inicialmente es nulo.
pendientes: es una cola donde se guardan los mensajes cuya atención debe posponerse hasta que cambien las condiciones del nodo.
peso_núcleo: describe el peso de la arista principal que coordina las operaciones del fragmento al que pertenece el nodo. Inicialmente vale 0.

4.5.4. Propiedades del algoritmo GHS

Debido a que estamos asumiendo un modelo de intercambio asíncrono de información, los mensajes pueden experimentar retardos arbitrarios, parecería que existe el riesgo de que el algoritmo caiga en una condición de interbloqueo (*deadlock*). Se argumenta a continuación, que esta contingencia es imposible.

Lema 4.3. *Desde cualquier configuración, con al menos dos fragmentos, en algún momento ocurre una fusión o una absorción.*

Demostración. Sea L el nivel mínimo que puede tener un fragmento de esta configuración y sea F el fragmento de nivel L, con la arista saliente de menor peso, entre todos los fragmentos del mismo nivel. Un mensaje de `PRUEBA` transmitido desde F puede recibirse en un vértice que pertenece a un fragmento F' de nivel $L' \geq L$, o bien, despertar a un nodo hasta entonces *dormido*. En el primer caso, se responde el mensaje inmediatamente. En el segundo caso, el nodo que despierta se considera un fragmento de nivel 0 y, en consecuencia, debe revisarse si L sigue siendo el menor nivel de la configuración y F el fragmento de nivel L, con la menor arista saliente, o si se actualizan estos valores para repetir la argumentación. Puesto en otras palabras, necesitamos asumir la existencia del fragmento con el menor nivel posible. Si ocurriera que este fragmento llegara a despertar a un nodo solitario, entonces éste pasaría a ser el nuevo fragmento de nivel mínimo, en el que se basa nuestra argumentación.

Puesto que el número de nodos es acotado, en algún momento se llega a una configuración en la que todo mensaje de `PRUEBA`, que F transmite, se responde inmediatamente, porque el vértice receptor pertenece a un fragmento con un nivel mayor o igual que L. Esto implica que, en algún instante, F

encuentra su arista saliente mínima (AMF) e, y transmite un mensaje `CONECTA` hacia otro fragmento F'. Si F' es un fragmento de nivel $L' > L$, entonces absorbe inmediatamente a F. En otro caso, ambos fragmentos son del mismo nivel y, por la forma en que se escogió a F, comparten la misma AMF. Luego, tiene lugar la fusión de estos. □

Debe observarse que, si el algoritmo llega a un estado en el que solo queda un fragmento, entonces terminará. Por tanto, si el algoritmo no ha terminado es que, al menos, hay dos fragmentos que aún no se unen.

El lema 4.3 implica que el número de fragmentos decrece al menos en uno, en cada etapa hasta que, en algún momento, no queda más que un solo fragmento, esto es, el AGM que se busca. Con esto puede concluirse que:

Corolario 4.1. *El algoritmo en algún momento termina y la construcción que produce es un AGM.*

Por otra parte, a causa de los retardos no acotados en la transmisión de mensajes, puede ocurrir que un nodo posea información sin actualizar sobre el fragmento al que pertenece. Por ejemplo, cuando ocurre la absorción de éste, pueden pasar varias unidades de tiempo antes de que un nodo reciba un mensaje `INICIO`, con el que se entere que pertenece a un fragmento diferente del que tiene registrado. Parecería que no se puede confiar en la información local para responder un mensaje de `PRUEBA`, lo que es crucial para identificar las aristas salientes. Se mostrará cómo, en algunas situaciones, puede responderse este mensaje aun cuando se tenga incertidumbre en la información local.

Las siguientes afirmaciones establecen que, durante el desarrollo del algoritmo, una arista solo puede ser el núcleo de un fragmento y que el nivel de un fragmento solo puede incrementarse.

Afirmación 4.5.1 *Sea e el núcleo de un fragmento F. Entonces e no puede ser el núcleo de un fragmento $F' \neq F$.*

Demostración. Supongamos lo contrario, es decir, que e es el núcleo de un fragmento $F' \neq F$. Esto significaría que F participó en una fusión o una absorción, de la que resultó un fragmento distinto que, sin embargo, conservó su núcleo original. En ninguno de los dos casos es esto posible. □

Como se ha dicho, la información que un nodo posee sobre su fragmento puede no estar actualizada si el fragmento está absorbiéndose. No obstante:

Afirmación 4.5.2 *Un nodo cuyo identificador de fragmento es (L,w), pertenece a un fragmento de nivel $L' \geq L$.*

Demostración. Solo si el fragmento al que pertenece el nodo estuviera participando en una fusión o una absorción, el identificador que conoce podría no ser exacto. En el primer caso el nivel exacto será $L' = L+1$. En el segundo caso, el nivel L' sería el mismo que el del fragmento absorbente. En ambos casos, $L' \geq L$. □

Considérese el procedimiento para determinar una arista saliente. Un nodo i en el fragmento F_1, con identificador (L_1, w_1), envía un mensaje de `PRUEBA`, sobre su arista e, hacia el nodo j en el fragmento F_2, con identificador (L_2, w_2). De acuerdo con el algoritmo, si $(L_1, w_1) = (L_2, w_2)$, entonces j responde inmediatamente con un mensaje de `RECHAZO`. Si $(L_1, w_1) \neq (L_2, w_2)$ y $L_2 \geq L_1$, entonces j responde

inmediatamente con un mensaje `ACEPTA`. Si $L_2 < L_1$, entonces retarda su respuesta hasta que cambie la relación entre L_1 y L_2.

En el intervalo entre el mensaje de `PRUEBA` de i y la respuesta de j, el identificador de i no puede cambiar. Esto es porque el mensaje de `PRUEBA` de i es consecuencia del mensaje `INICIO` que procede de un nodo principal. Mientras los nodos principales no reciban el reporte, desde cada uno de los nodos del fragmento, no podrá emitirse un mensaje `CONECTA`. Luego, F_1 no se funde o absorbe a otro fragmento y por tanto, el identificador de F_1 no puede cambiar. Por tanto, el único identificador que puede ser incorrecto es el de j.

Se argumenta a continuación que, si j responde a i, entonces su respuesta será correcta, aun cuando L_2 y w_2 no estén actualizados.

Primeramente, si dos nodos pertenecen al mismo fragmento, permanecerán en él por el resto del algoritmo. Como el identificador de su fragmento no puede cambiar durante la búsqueda, $(L_1, w_1) = (L_2, w_2)$ y e no es una arista saliente. Luego, está justificado el rechazo de j. A continuación se considera el caso en que i y j no pertenecen al mismo fragmento.

Afirmación 4.5.3 *Si j devuelve el mensaje* `ACEPTA`*, entonces i y j no pertenecen al mismo fragmento.*

Demostración. j devuelve un mensaje `ACEPTA` a i, solo si $(L_1, w_1) \neq (L_2, w_2)$ y $L_2 \geq L_1$. Si $L_2 > L_1$, entonces por la afirmación 4.5.2, el nivel real de F_2 solo puede ser mayor o igual a L_2 y, por tanto, mayor que L_1. Luego, por la afirmación 4.5.1, F_1 y F_2 tienen distintos núcleos y, en consecuencia, i y j están en fragmentos diferentes.

De otra forma, si $L_1 = L_2$ y $(L_1, w_1) \neq (L_2, w_2)$, entonces el nivel real de F_2 solo puede ser mayor o igual que L_2. Si es igual, entonces w_2 sigue siendo el peso del núcleo de F_2, por la afirmación 4.5.1. Si el nivel real de F_2 es mayor que L_2, también será mayor que L_1, entonces i y j no pertenecen al mismo fragmento. □

Si $L_2 < L_1$, podría ocurrir que algún otro nodo de F_2 enviara un mensaje `CONECTA` hacia algún nodo en F_1 que resulte en la absorción de F_2, por parte de F_1. Un mensaje de `INICIO` sería enviado hacia F_2. Si el mensaje tardara en alcanzar a j, i y j pertenecerían al mismo fragmento sin saberlo. Para evitar esta contingencia, j debe retardar la respuesta del mensaje de `PRUEBA` de i.

Es posible que un fragmento F_1 absorba a otro fragmento F_2, mientras busca su arista mínima saliente y que este hecho modifique la búsqueda de F_1 resultando, por ejemplo, que la arista mínima saliente de F_2 también sea la AMF del nuevo fragmento $F_1 \cup F_2$. Supóngase que el nodo i, del fragmento F_1 con identificador (L_1, w_1), recibe un mensaje `CONECTA` desde el nodo j, en el fragmento F_2 con identificador (L_2, w_2), sobre la arista e. Supóngase también que $L_1 > L_2$. En este caso, la absorción ocurre del siguiente modo:

El nodo i envía el mensaje `INICIO`(w_1, L_1, s_i) a j (que a su vez lo retransmite al resto del fragmento F_2), donde s_i es el estado del nodo i. Si $s_i =$*búsqueda*, entonces F_2 tomará parte en la búsqueda de la arista mínima saliente y la arista que resulte será la AMF del fragmento $F_1 \cup F_2$. En otro caso, si $s_i =$*reporte*, i ya habrá enviado su mensaje de `REPORTE` y la AMF de F_1 será la misma que para $F_1 \cup F_2$. Luego, no se efectuará una búsqueda en F_2.

Para acotar el número de mensajes enviados durante la ejecución del algoritmo, se demostrará primero el siguiente lema:

Lema 4.4. *Un fragmento de nivel L contiene, al menos,* 2^L *nodos.*

Demostración. Por inducción sobre L: El caso base es inmediato, un fragmento de nivel $L = 0$ contiene un solo nodo. Supóngase que el lema es cierto para fragmentos de nivel $\leq L-1$ y considérese un fragmento F de nivel L. F es el resultado de la combinación de dos fragmentos de nivel $L-1$ y quizá, de la absorción de algunos fragmentos. Según la hipótesis de inducción, cada uno de los fragmentos de nivel $L-1$ contiene al menos $2^{(L-1)}$ nodos y, por tanto, F contiene al menos 2^L nodos. □

Ahora vamos a establecer los límites superiores que acotan la complejidad, en mensajes y en tiempo, del algoritmo GHS.

Por principio de cuentas, una arista solo puede ser rechazada una vez y esto implica la transmisión de un mensaje de `PRUEBA`, que se responde con otro de `RECHAZO`. Entonces, este intercambio nos reporta a lo más $2m$ mensajes.

Por otra parte, mientras que un nodo se encuentra en un nivel diferente del primero o el último, puede recibir a lo más un mensaje `INICIO` y un mensaje `ACEPTA`. Puede transmitir, a lo más, un mensaje exitoso de `PRUEBA`, uno de `REPORTE` y, según sea un nodo en el interior del fragmento o en la frontera del mismo, un mensaje `UNION` ó `CONECTA`, respectivamente. Por el lema 4.4, podemos inferir que un nodo puede pasar por $\log n$ niveles, cuando mucho. Entonces, si descontamos el primero y el último nivel, el nodo puede intercambiar $5n(\log n - 1)$ mensajes.

Finalmente, en el nivel 0 un nodo puede recibir a lo más un mensaje `INICIO` y transmitir un mensaje `CONECTA`. En el último nivel, cada nodo envía un mensaje de tipo `REPORTE`. Estos casos nos producen $3n$ mensajes.

Si ahora reunimos todas nuestras cuentas parciales totalizaremos $2m + 5n(\log n - 1) + 3n < 2m + 5n \log n$ mensajes.

Para abordar la complejidad de tiempo supondremos, como en otros casos, que la transmisión de un mensaje toma una unidad de tiempo. La duración del algoritmo dependerá del número de participantes independientes (no necesariamente simultáneos) que arranquen la construcción. Se pueden encontrar casos en los que los participantes se despiertan de forma secuencial y cada uno prueba $O(n)$ aristas antes de encontrar su mejor arista, entonces el algoritmo puede tardarse hasta $O(n^2)$ unidades de tiempo. Si el tiempo es un recurso a considerar, parecería más indicado utilizar algún algoritmo de propagación, por ejemplo, para despertar a todos los nodos y luego arrancar la construcción del AGM. En estas circunstancias se puede garantizar que todos los nodos estarán despiertos, a lo más, en el instante $n-1$. Para el tiempo n cada nodo habrá enviado un mensaje `CONECTA` y, para el tiempo $2n$, cada nodo se encontrará formando parte de un fragmento de nivel 1.

Afirmación 4.5.4 *Suponiendo que todos los nodos inician espontáneamente el algoritmo, antes del instante n, al llegar al tiempo* $5Ln - 3n$*, a lo más, cada nodo se encuentra en el nivel L*

Demostración. Demostraremos por inducción. Se observa que está afirmación es verdadera para $L-1$ con $L-1 \geq 0$. Ahora, supondremos que todos los nodos se encuentran en un nivel L, al llegar al tiempo $5Ln-3n$. En estas circunstancias, cada nodo puede enviar hasta n mensajes de `PRUEBA` (uno a la vez), a los que corresponden n respuestas en n instantes distintos. Todos los nodos envían su `REPORTE` hacia el núcleo del fragmento, lo que vuelve a implicar n unidades de tiempo distintas. Los nodos principales envían su mensaje de `UNION`, que puede tardar en viajar otro tanto igual de tiempo. Finalmente ocurre

un mensaje de `INICIO` que se propaga por todos los nodos en n unidades otra vez. Desde el momento en que los nodos se encontraban en el nivel L, hasta ahora que se encuentran en el nivel $L+1$, han transcurrido $5n$ unidades de tiempo, lo que significa que ahora nos encontramos en el instante $5Ln - 3n + 5n = 5(L+1)n - 3n$. □

En la última etapa, solo se envían los mensajes de `PRUEBA`, `RECHAZO` y `REPORTE` en, a lo más, $3n$ unidades. Por tanto, como $L \leq \log n$, el algoritmo concluye en $5Ln - 3n + 3n \geq 5n \log n$, unidades de tiempo.

4.6. Comentarios finales

Se sabe que [2], para una red arbitraria, se requieren $m + n \log n$ mensajes para construir el AGM, lo que demuestra que el algoritmo GHS es óptimo, salvo constantes, respecto a esta medida. En contraste, su complejidad en tiempo, para el peor caso es $O(n^2)$. Awerbuch propuso un algoritmo que es, a la vez, óptimo en mensajes y en tiempo $\Theta(m + n \log n)$ y $\Theta(n)$, respectivamente [5].

El algoritmo GHS funciona solo si se garantiza que todos los pesos de las aristas son diferentes. Sin embargo, es posible construir una variante del algoritmo que funcione aun si existen aristas con pesos iguales, siempre y cuando el conjunto de los pesos esté totalmente ordenado, al igual que los identificadores de los procesos del sistema. En estas circunstancias, todo participante define por cada una de sus aristas una tupla formada por el peso de la arista y el identificador del proceso con el que se conecta, cada tupla caracteriza exactamente a una arista pero además, el conjunto de las tuplas puede ordenarse lexicográficamente. Si los procesos del sistema ignoran los identificadores de sus vecinos, entonces cada uno deberá transmitir su propio identificador sobre todos sus enlaces incidentes, antes de arrancar el algoritmo.

Ejercicios

4.1. ¿Qué diferencias hay entre el árbol de peso mínimo que se construye con el algoritmo PIF y el que se construye con el algoritmo GHS?, dicho de otro modo ¿en qué circunstancias se aplica cada uno de ellos?

RESPUESTA: La diferencia se encuentra en el significado que se le otorga a los pesos con que se ponderan las aristas en el algoritmo PIF. Solo cuando el peso se refiere al retardo de propagación de la información, puede asegurarse que PIF (y PI también), construyen un AGM. En cualquier otra circunstancia debe usarse GHS.

4.2. Supongamos un grafo tal que los pesos de sus aristas no son únicos, pero los pesos al igual que los identificadores de los vértices forman conjuntos totalmente ordenados. ¿Cómo puede aprovecharse esta circunstancia para aplicar GHS sobre éste y producir un AGM único?

SUGERENCIA: Cada participante define, por cada una de sus aristas, una tupla formada por el peso de la arista y el identificador del nodo con el que se conecta. De esta forma, cada tupla caracteriza exactamente a una arista. Además, el conjunto de las tuplas puede ordenarse lexicográficamente.

4.3. ¿Cómo podría usarse el algoritmo GHS para resolver el problema de elección sobre una red asíncrona arbitraria?

RESPUESTA: El algoritmo garantiza que al terminar hay una sola arista tipo núcleo, de cuyos vértices puede elegirse al líder. La decisión final puede tomarse atendiendo al orden que existe entre los identificadores de los vértices adyacentes al núcleo.

4.4. Se dice que, para optimizar tiempo, se deben despertar a todos los nodos antes de arrancar el algoritmo GHS, ¿cuáles son los costos adicionales de esta operación preliminar?

SUGERENCIA: Los costos asociados con un algoritmo tipo PI o PIF.

4.5. ¿Cómo puede explicarse que detener el crecimiento de un fragmento trae ahorros de tiempo en el largo plazo?, ¿cómo se codifica está condición?

RESPUESTA: Porque esta acción de paro no es indefinida, solo se difiere hasta el momento en que, el fragmento está en posibilidad de duplicar su orden en una sola operación. Esta condición se da cuando se presenta un mensaje de `PRUEBA` que no puede responderse hasta que cambie la relación de niveles entre el fragmento que envía y el que recibe. Mientras tanto, el mensaje se guarda en una cola denominada “pendientes”.

Parte II
Orden

Sabia virtud, de conocer el tiempo.

—Renato Leduc

5 Relojes lógicos

Resumen Ahora estamos interesados en construir un orden entre los eventos que se presentan en un sistema distribuido. Se trata de una tarea muy importante, a partir de la cual se soporta el análisis de las operaciones del propio sistema, y sirve para resolver problemas como la evaluación de alguna propiedad global, la construcción de puntos de verificación (checkpoints), la depuración o detección de errores, entre otros. En este capítulo presentamos un conjunto de técnicas para fechar los eventos de un sistema distribuido, cuando el uso de un reloj físico queda descontado.

5.1. Introducción

En la vida cotidiana, se utiliza una norma de tiempo universal para establecer una relación de precedencia entre eventos, a partir de un orden cronológico fijado con relojes débilmente sincronizados, como los relojes de pulsera y pared. En contraste, en un sistema distribuido es muy difícil disponer de un tiempo único pues, en vista de la rapidez con la que se suceden los eventos y las limitaciones del subsistema de comunicaciones, es casi imposible sincronizar todos los relojes físicos con la precisión necesaria. Alternativamente, existen algoritmos denominados *sistemas de reloj lógico*, con los que pueden fecharse los eventos como mecanismo de base para establecer un orden entre ellos [46].

En este capítulo se presenta la teoría de los algoritmos de reloj lógico. Como en otras ocasiones, se comienza por destacar los atributos del modelo de sistema que subyacen en el planteamiento del problema. Enseguida se desarrollan varias soluciones alternativas y se establece una comparación entre estas. Para terminar, se discuten las posibilidades de implantación y aplicación de este tipo de algoritmos.

La notación utilizada en este capítulo se presenta en la tabla 5.1.

e	Un evento
h	Historia local del proceso i
H	Historia global
$\rightarrow$	Relación sucedio antes
$\prec$	Orden total
$C(e)$	Estampilla de tiempo del evento e
(t_i, i)	Estampilla de un evento en el proceso i
t_i	Tiempo lógico local del proceso i
$\mathbf{t}_i$	Reloj lógico vectorial del proceso i
$\mathbf{T}_i$	Reloj lógico matricial del proceso i
M	Mensaje del algoritmo

Tabla 5.1: Notación

5.2. Dependencia causal

Se definen tres tipos diferentes de *eventos* que un proceso puede atender: *internos*, de *transmisión de mensajes* y de *recepción de mensajes*. Un evento interno solo afecta el estado del proceso en que ocurre. Se trata, por tanto, de una acción local. Por otra parte, los eventos de transmisión y recepción representan el flujo de información entre procesos y establecen una relación causa-efecto entre la transmisión de un mensaje y su correspondiente recepción.

La *historia local* del proceso i, durante una ejecución del sistema, es una secuencia (posiblemente infinita) de eventos:

$$h_i = e_i^1, e_i^2, \ldots$$

ocurridos en i. El superíndice de la notación, denominada *numeración canónica*, corresponde con el orden total impuesto por la ejecución secuencial de los eventos locales. Sea $h_i^k = e_i^1, e_i^2, \ldots, e_i^k$ el prefijo de la historia local conteniendo sus primeros k eventos. Se define h_i^0, como la secuencia vacía. La *his-*

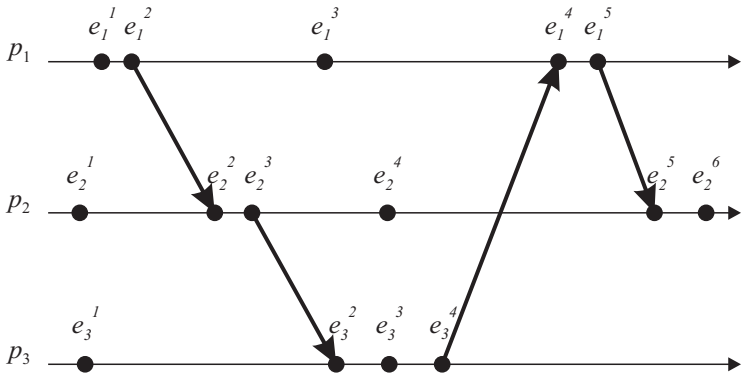


Figura 5.1: Diagrama espacio-tiempo con los eventos de un sistema distribuido.

toria global de la ejecución es el conjunto $H = h_1 \cup \ldots \cup h_n$, conteniendo la historia local de todos los participantes del sistema.

Obsérvese que la historia global no especifica los tiempos relativos entre sus eventos. En el contexto de un sistema asíncrono, los eventos pueden ordenarse con base en la noción de "causa y efecto", previamente mencionada. En otras palabras, dos eventos están obligados a suceder en cierto orden, solo si la ocurrencia de uno puede afectar la existencia del otro. La información fluye de un evento a otro, ya sea porque ambos eventos ocurren en el mismo proceso, y por tanto inciden sobre un mismo estado local, o bien porque los eventos ocurren en procesos diferentes y corresponden al intercambio de un mensaje.

Se pueden formalizar estas ideas definiendo la relación $\rightarrow$ de *dependencia causal*, también conocida como "*sucedió antes*", sobre los eventos de una ejecución a partir de las siguientes propiedades [31]:

1. Si $e_i^k, e_i^l \in h_i$ y $k < l$, entonces $e_i^k \rightarrow e_i^l$.
2. Si e_i es un evento de transmisión y e_j es su recepción correspondiente, entonces $e_i \rightarrow e_j$.
3. Si $e \rightarrow e'$ y $e' \rightarrow e''$, entonces $e \rightarrow e''$.

La dependencia causal permite caracterizar una *ejecución distribuida*, como un conjunto parcialmente ordenado definido por la pareja $(H, \rightarrow)$.

Hay que entender que la relación causa-efecto solo se cumple cuando una pareja de eventos representan el segundo caso que define a $\rightarrow$, en cualquier otra circunstancia, la única conclusión que puede inferirse de la relación $e \rightarrow e'$ es: que la ocurrencia de *e podría* haber afectado la ocurrencia de e'.

Algunos eventos de la historia global pueden no estar causalmente relacionados, en otras palabras, es posible que para algunos e y e' no se cumpla que $e \rightarrow e'$ y tampoco $e' \rightarrow e$. En tal circunstancia se dice que son eventos *concurrentes*, lo que se denota como $e||e'$. La figura 5.1, denominada *diagrama espacio-tiempo*, muestra una ejecución distribuida en tres procesos. Cada línea horizontal representa el progreso de un proceso diferente. Un punto indica un evento y una flecha indica la transferencia de un mensaje. En esta ejecución se tiene que $e_1^2 \rightarrow e_2^2$, $e_2^2 \rightarrow e_3^2$ y $e_2^2||e_1^3$.

Un sistema de reloj lógico consiste en un conjunto T y una función C que asocia a un evento e, de una ejecución distribuida $(H, \rightarrow)$, con un elemento $C(e)$, denominado estampilla de e, en el conjunto T. Los elementos de T forman un conjunto parcialmente ordenado, sobre la relación $<$. El reloj se define como:

$$C : H \rightarrow T,$$

de modo que, dados dos eventos arbitrarios e_1 y e_2, se satisfaga la siguiente propiedad:

$$e_1 \rightarrow e_2 \Rightarrow C(e_1) < C(e_2).$$

Las estampillas asociadas a los eventos obedecen la propiedad fundamental de monotonicidad. Esto es, si un evento e_1 puede afectar causalmente a un evento e_2, la estampilla $C(e_1)$ será menor que $C(e_2)$. Esta propiedad se denomina condición de *consistencia débil* de reloj. Si T y C satisfacen la siguiente condición:

$$e_1 \rightarrow e_2 \Leftrightarrow C(e_1) < C(e_2),$$

entonces se dice que el sistema cumple una condición de *consistencia fuerte*.

5.3. Modalidades de reloj lógico

En la implantación de un reloj lógico, deben resolverse dos tareas igualmente importantes. En primer lugar, el diseño de una estructura de datos local con la que cada proceso pueda representar su tiempo lógico. En segundo lugar, el diseño de un protocolo o conjunto de reglas, mediante las cuales puedan actualizarse las estructuras de datos para garantizar la condición de consistencia.

Cada proceso i mantendrá una estructura de datos que le ofrezca dos capacidades:

- Un reloj lógico local lc, con el que el proceso pueda medir su propio progreso.
- Un reloj lógico global gc, que represente la imagen particular que i mantiene, sobre el tiempo global y que le permite asignar estampillas consistentes. Típicamente lc forma parte de gc.

Por su parte, el protocolo garantiza que el reloj lógico del proceso y su visión del tiempo global, se manejen consistentemente. Cualquier protocolo de este tipo debe incluir las siguientes dos reglas:

R1. Que establece la forma en que el proceso actualiza su reloj lógico local, cada vez que se ejecuta un evento sea de transmisión, recepción o interno.

R2. Que establece la forma en que un proceso actualiza su reloj lógico global para mantener su visión del tiempo y progreso global. Determina la información concerniente al tiempo local que el proceso habrá de acarrear sobre los mensaje que transmita y define también, la manera en que el proceso receptor habrá de beneficiarse con los mensajes que reciba, para actualizar su tiempo global.

Existen tres tipos de sistemas de reloj lógico: *escalares, vectoriales* y *matriciales*. Su clasificación atiende a la forma como representan su tiempo lógico y actualizan sus relojes lógicos. Sin embargo, todos incluyen un conjunto de reglas para garantizar la propiedad fundamental de monotonicidad, asociada con la causalidad. Además, cada uno de estos sistemas ofrecen al usuario, otras propiedades interesantes.

5.3.1. Relojes escalares

En un sistema de reloj lógico de tipo escalar, el dominio del tiempo es el conjunto de los enteros no negativos. El reloj lógico local del proceso i, así como su visión del tiempo global, se encuentran condensados en una sola variable entera t_i [31].

Las reglas con que se maneja el reloj son las siguientes:

R1. Antes de la ejecución de un evento (transmisión, recepción o interno), i ejecuta la instrucción

$$t_i = t_i + d \quad (d > 0)$$

En el caso más general, d puede tener un valor diferente, cada vez que se ejecuta R1 y este valor es dependiente de la aplicación. Sin embargo, en muchos casos prácticos, d mantiene un valor constante e igual a 1, pues con ello, cada proceso es capaz de identificar unívocamente cada evento que ocurre y al mismo tiempo se minimiza la tasa de crecimiento de d.

Algoritmo 5.1: Algoritmo de reloj escalar de Lamport en el nodo i

```
1 al recibir M(t') efectúa
2     t_i ← max(t_i, t')
3     Ejecuta R1
4     Entrega M
```

R2. Cada mensaje que cruza la red, lleva a cuestas el valor del reloj de su transmisor al momento de la expedición. Cuando i recibe un mensaje M con la estampilla t', entonces ejecuta las instrucciones del algoritmo 5.1.

Claramente, los relojes escalares satisfacen la monotonicidad y, por tanto, la propiedad de consistencia. Además, en un sistema distribuido se pueden usar relojes escalares para establecer un orden total de los eventos.

Un problema de este esquema de ordenamiento es que dos o más eventos, localizados sobre diferentes procesos, pueden tener estampillas idénticas de tiempo. Si esto ocurre, el empate se rompería utilizando los propios identificadores de proceso. Se redefine una estampilla como la tupla (t_i, i), donde t_i es el tiempo al que ocurre un evento, mientras i es el identificador del proceso en que se localiza. La relación de orden total $\prec$, sobre dos eventos e y e' con estampillas (t_i, i) y (t_j, j), se define de la siguiente manera:

$$e \prec e' \Leftrightarrow (t_i < t_j) \vee (t_i = t_j \wedge i < j)$$

Por otra parte, el inconveniente más grave de los relojes escalares es que no son fuertemente consistentes. Esto es, para dos eventos e_1 y e_2,

$$C(e_1) < C(e_2) \not\Rightarrow \ e_1 \rightarrow e_2$$

Por ejemplo, en la figura 5.2, el tercer evento del proceso p_1 tiene una estampilla escalar menor que la estampilla del tercer evento en p_2. Sin embargo, el evento de p_1 no precede causalmente al evento de p_2.

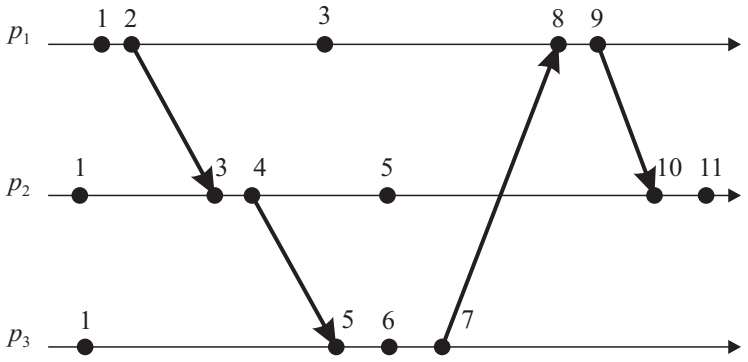


Figura 5.2: Evolución del tiempo escalar en una ejecución distribuida.

Algoritmo 5.2: Algoritmo de reloj vectorial de Fidge-Mattern en el nodo i

```
1 al recibir M(t') efectúa
2     Actualiza su tiempo lógico global 1 ≤ k ≤ n : t_i[k] := max(t_i[k], t'[k])
3     Ejecuta R1.
4     Entrega M.
```

5.3.2. *Relojes vectoriales*

En un sistema de reloj lógico vectorial, el dominio del tiempo se representa como un conjunto n-dimensional de vectores enteros no negativos. Cada proceso i mantiene un vector $\mathbf{t}_i[1,\ldots,n]$, donde $\mathbf{t}_i[i]$ es el reloj lógico local de i y describe el progreso del tiempo lógico local. En tanto, $\mathbf{t}_i[j]$ con $j \neq i$, representa la última noticia que i posee sobre el tiempo lógico de j. El vector $\mathbf{t}_i$, en su totalidad, representa la imagen que i posee sobre el tiempo lógico global y que utiliza para fechar los eventos [19] [37].

Un proceso i utiliza las siguientes dos reglas para actualizar su reloj:

R1. Antes de ejecutar un evento, i, actualiza su tiempo lógico local con la instrucción

$$\mathbf{t}_i[i] := \mathbf{t}_i[i] + d \quad (d > 0)$$

R2. Cada proceso transmisor guarda, dentro del mensaje M que expide, el valor de su reloj vectorial al instante de la transmisión. Al momento de recibir el mensaje $M(\mathbf{t}')$, el proceso i ejecuta las instrucciones del algoritmo 5.2.

De esta forma, la estampilla de un evento será el valor del reloj vectorial que reporta el proceso, en el instante en que se ejecuta.

Para dos estampillas vectoriales arbitrarias $\mathbf{t}$ y $\mathbf{t}'$, se cumplen las siguientes propiedades:

- $\mathbf{t} \leq \mathbf{t}' \Leftrightarrow \forall j : \mathbf{t}[j] \leq \mathbf{t}'[j]$.
- $\mathbf{t} < \mathbf{t}' \Leftrightarrow \mathbf{t} \leq \mathbf{t}' \wedge \exists j : \mathbf{t}[j] < \mathbf{t}'[j]$.
- $\mathbf{t} || \mathbf{t}' \Leftrightarrow \neg(\mathbf{t} < \mathbf{t}') \wedge \neg(\mathbf{t}' < \mathbf{t})$.

Como se estableció, la relación $\rightarrow$ induce un orden parcial sobre el conjunto de eventos producidos por una ejecución distribuida. Sean dos eventos e_1 y e_2. Si se utiliza un sistema de reloj lógico vectorial para fecharlos con las estampillas $\mathbf{t}$ y $\mathbf{t}'$, respectivamente, entonces

- $e_1 \rightarrow e_2 \Leftrightarrow \mathbf{t} < \mathbf{t}'$.
- $e_1 || e_2 \Leftrightarrow \mathbf{t} || \mathbf{t}'$.

Por otra parte, un sistema de relojes vectoriales cumple la condición de consistencia fuerte. En consecuencia, puede determinarse si dos eventos se encuentran causalmente relacionados, comparando sus estampillas vectoriales. Sin embargo, la dimensión del reloj vectorial no puede ser menor que n para poder aplicar esta útil propiedad [14].

Adicionalmente, si el valor de d se mantiene constante e igual a 1, en la regla R1, el i-ésimo componente del reloj vectorial en i, $\mathbf{t}_i[i]$, denota el número de eventos en i ocurridos hasta ese momento. De

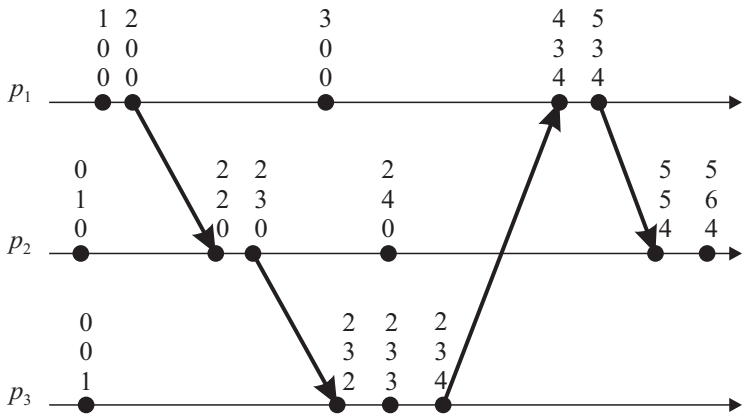


Figura 5.3: Evolución del tiempo vectorial en una ejecución distribuida.

la misma forma, cuando un evento e tiene una estampilla $\mathbf{t}$, entonces $\mathbf{t}[j]$ denota el número de eventos ejecutados por j, que suceden antes que e, como se observa en la figura 5.3. Evidentemente, $\sum \mathbf{t}[j] - 1$ representa el total de eventos de la ejecución distribuida, que preceden causalmente a e.

5.3.3. Relojes matriciales

En un sistema de relojes matriciales, el tiempo se representa como un conjunto de matrices de enteros no negativos de tamaño $n \times n$. Un proceso i mantiene una matriz $\mathbf{T}_i[1 \ldots n, 1 \ldots n]$, donde

- $\mathbf{T}_i[i,i]$ denota el reloj lógico local del proceso i y registra el progreso de la ejecución en el mismo;
- $\mathbf{T}_i[i,j]$ denota la última noticia que el proceso i tiene sobre el reloj lógico local, $\mathbf{T}_j[j,j]$, de j (de la sección anterior, obsérvese que el renglón $\mathbf{T}_i[i,.]$ es igual al vector $\mathbf{T}_i[.]$ y exhibe todas las propiedades de los relojes lógicos vectoriales);
- $\mathbf{T}_i[j,k]$ representa el conocimiento del proceso i sobre lo último que j sabe acerca del reloj lógico local, $\mathbf{T}_k[k,k]$, de k.

La matriz completa $\mathbf{T}_i$ denota la visión particular que i mantiene sobre el tiempo lógico global. La estampilla matricial de un evento es el valor del reloj matricial de su proceso, en el momento en que se ejecuta [22].

Un proceso i maneja su reloj de acuerdo con las siguientes reglas:

R1. Antes de ejecutar un evento, i actualiza su tiempo lógico local efectuando la instrucción

$$\mathbf{T}_i[i,i] := vecT_i[i,i] + d \quad (d > 0)$$

R2. Cada mensaje M transmitido, lleva a cuestas una matriz de tiempo $\mathbf{T}'$. Cuando i recibe el mensaje compuesto $\mathbf{T}'$, procedente de j, i efectúa la secuencia de instrucciones dadas en el algoritmo 5.3.

Una de las propiedades más interesantes de los sistemas de reloj lógico matriciales, puede expresarse de la siguiente forma:

Algoritmo 5.3: Algoritmo de reloj matricial de Fischer en el nodo i

```
1 al recibir M(T') efectúa
      /*Primero, actualiza su tiempo lógico global, revisando las componentes de su
        matriz local*/
2     1 ≤ k ≤ n :  T_i[i,k] := max(T_i[i,k], T'[i,k])
3     1 ≤ k,l ≤ n :  T_i[k,l] := max(T_i[k,l], T'[k,l])
4     Ejecuta R1
5     Entrega M.
```

- Si $min_k(\mathbf{T}_i[k,l])$ es mayor o igual a un cierto valor *val*, entonces el proceso i está seguro que cualquier otro proceso k sabe que el tiempo local de l ha avanzado hasta un valor *val*.
- Si se verifica esta condición, significa que i puede dar por hecho que todos los procesos esperan información proveniente de l, con un tiempo local $>$ *val*. En muchas aplicaciones esto implica que, a partir de cierto momento, los procesos pueden prescindir de la información proveniente de l y, de igual forma, pueden aprovechar esta situación para descartar información obsoleta (i.e. recolectar basura).

5.4. Comentarios finales

El conocimiento de la relación de precedencia causal ayuda a resolver una serie de problemas, entre los que pueden citarse: la depuración de programas distribuidos, la medición del grado de concurrencia de una aplicación o la realización de otros algoritmos distribuidos en los que se requiere ordenar los eventos de una ejecución, como se verá en el siguiente capítulo.

La implantación de un sistema de reloj lógico es un compromiso entre la complejidad en el tamaño de los mensajes intercambiados y la consistencia del reloj. Mientras en un reloj escalar, el tamaño del mensaje es independiente del total de sitios de la red, en los relojes vectoriales y matriciales el tamaño puede convertirse en una carga excesiva para los mensajes que circulan por los enlaces de comunicación, y las estructuras de datos deben modificarse cada vez que se agrega un nuevo sitio a la red. Por otro lado, un reloj escalar solo ofrece una condición de consistencia débil (que para algunas aplicaciones puede ser suficiente), en tanto, los otros ofrecen una condición de consistencia fuerte.

Ejercicios

5.1. Sea T un árbol binario cuya raíz tiene el identificador 1. Los nodos con identificadores 2 y 5 son los hijos de 1. Por su parte, 2 es el padre de 3 y 4, mientras que 5 es el padre de 6 y 7. Se ejecuta un algoritmo PIF desde 1 y se fechan los diferentes eventos del algoritmo con un reloj vectorial. ¿A qué momentos del algoritmo corresponden las siguientes estampillas? (4,6,2,2,6,2,2), (1,1,2,0,0,0,0), (1,0,0,0,6,2,2)

RESPUESTA: A la terminación del algoritmo en el nodo raíz, en el nodo 3 y en el nodo 5, respectivamente.

5.2. ¿Cómo podríamos emular un sistema síncrono, en un ambiente asíncrono?

SUGERENCIA: Necesitamos garantizar que todos los procesos han terminado una "ronda" o pulso, para pasar al siguiente (véase el artículo "Complexity of Network Synchronization", de Baruch Awerbuch, JACM, 32(4), 1985).

5.3. Un protocolo de difusión causal se encarga de entregar el mismo conjunto de mensajes sobre cada posible destinatario, obedeciendo a un orden causal sobre los mensajes. Desarrolla un protocolo de difusión causal, usando un mecanismo de reloj lógico.

RESPUESTA: Cada uno de los participantes mantiene un reloj vectorial.

a) Antes de difundir un mensaje M, un proceso i le añade su estampilla correspondiente, obedeciendo las reglas de actualización (R1 y R2).
b) Al recibir un mensaje M con estampilla V_M, el proceso receptor j difiere la entrega de M hasta que se cumplan las siguientes condiciones

$$V_M[i] = V_j[i] + 1,$$

Para toda k distinta de i, $V_M[k] <= V_j[k]$
c) Luego de la entrega, el reloj local de V_j se actualiza como en el alg. 5.2

5.4. Demuestra que los relojes vectoriales cumplen la propiedad de consistencia fuerte.

SUGERENCIA: Hay que observar que se trata de una demostración en dos sentidos o dos partes. En la primera parte, si e_1 sucedió antes que e_2 entonces la etiqueta del primero será menor que la del segundo (aquí hay que considerar dos casos, cuando ambos suceden en el mismo proceso o cuando ocurren en procesos diferentes). En la segunda parte, debemos demostrar que si tenemos dos etiquetas y la primera es menor que la segunda, entonces el evento que corresponde a la primera sucedió antes que el evento que corresponde a la segunda (intentar por contradicción).

5.5. ¿Qué tipo de consistencia ofrece un reloj de tiempo real?

RESPUESTA: Consistencia débil.

Fotografiar es colocar la cabeza, el ojo y el corazón en un mismo eje.

—Henri Cartier-Bressson

6

Estado global

Resumen El estado global de un sistema distribuido es la unión de los estados individuales de los procesos que lo integran. Tratándose de un sistema asíncrono en el que no se dispone de algún recurso común para almacenamiento de información, la única forma en que un proceso puede conocer el estado de los demás componentes será intercambiando mensajes. Sin embargo, las limitaciones propias del sistema pueden llevar a la construcción de un estado global obsoleto, incompleto o inconsistente (informalmente se entiende que un estado es inconsistente cuando no preserva la relación entre causas y efectos, en el registro de los eventos de los que da cuenta). Al mismo tiempo, la variabilidad de los retardos de la comunicación podría producir, para un mismo instante de la ejecución, dos estados globales diferentes observados desde dos procesos diferentes. En este capítulo presentamos un algoritmo distribuido para construir una "fotografía" en pedazos que, luego de reunirlos, producen un estado global consistente.

6.1. Introducción

Muchos problemas importantes del cómputo distribuido admiten soluciones que contienen una fase en la que se requiere la detección de una propiedad global, la cual puede entenderse como una instancia del problema de *Evaluación de un Predicado Global*, donde el objetivo consiste en evaluar una expresión booleana cuyas variables están referidas a un estado global del sistema [6]. Algunos ejemplos de este tipo de problemas pueden ser: la detección del bloqueo o estancamiento (deadlock), la detección de la terminación, la detección de la pérdida de una ficha, la recolección de basura, la construcción de

puntos de verificación (checkpoints), la restauración de una ejecución y, en general, la depuración y el monitoreo.

Supóngase que existe un proceso p que debe calcular el estado global del sistema en que participa. En este capítulo se examina un procedimiento denominado monitor activo o *fotografía instantánea*, en el cual, p interroga al resto de los integrantes del sistema acerca de su condición local. En la presentación subyace la noción de tiempo y los mecanismos conocidos para su medición, por ello se encontrarán herramientas y conceptos desarrollados previamente. El resto del capítulo se divide en 3 secciones. En la primera presentamos una serie de nociones preliminares que sirven para definir formalmente el problema que queremos abordar. Posteriormente, desarrollamos por refinamientos sucesivos una solución y, por último, reflexionamos sobre los alcances y aplicaciones de este algoritmo.

La notación utilizada en este capítulo se presenta en la tabla 6.1.

e	Un evento
p	Proceso monitor
h	Historia local
σ	Estado local
Σ	Estado global
R	Una corrida
C	Un corte
LC	Reloj lógico
$\rightarrow$	Relación sucedió antes
t	tiempo
χ_{ij}	Estado del canal de comunicación entre los procesos i y j
c_i	Índice del último evento ocurrido en el proceso i
ENT_i	Conjunto de procesos desde los que i puede recibir mensajes
SAL_i	Conjunto de procesos a los que i puede enviar mensajes
`M`, `FOTO`	Mensajes del algoritmo

Tabla 6.1: Notación

6.2. Cortes y causalidad

Sea σ_i^k el estado local del proceso i inmediatamente después de ejecutar el evento e_i^k y σ_i^0 su estado inicial antes de ejecutar algún evento. El *estado global* de una ejecución distribuida será la n-tupla $\Sigma = (\sigma_1 \dots \sigma_n)$ formada por los estados locales de cada uno de los procesos del sistema. Un *corte* de la ejecución es un subconjunto C de su historia global [1] que contiene un prefijo inicial de cada una de las historias locales. Se puede especificar un corte $C = h_1^{c_1} \cup \dots \cup h_n^{c_n}$ mediante la tupla de números naturales $(c_1 \dots c_n)$ correspondiente al índice del último evento ocurrido en cada proceso. El conjunto $(e_1^{c_1} \dots e_n^{c_n})$ de los últimos eventos incluidos en el corte se denomina la *frontera* del corte. Evidentemente, cada corte $(c_1 \dots c_n)$ define un estado global $(\sigma_1^{c_1} \dots \sigma_n^{c_n})$.

Una *corrida* es un orden total R que incluye a todos los eventos de la historia global y es consistente con cada historia local, esto significa que, por cada proceso i, los eventos del mismo aparecen en R en

[1] puede consultarse el capítulo previo "Relojes lógicos" para revisar la definición de la historia global

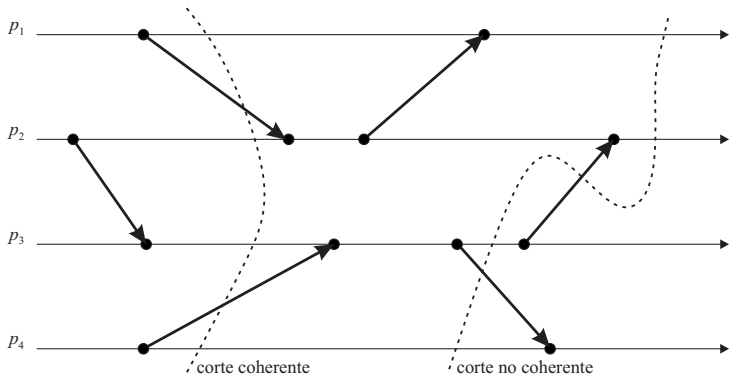


Figura 6.1: Cortes de una ejecución distribuida.

el mismo orden que aparecen en h_i. Obsérvese que una corrida puede o no ser igual a la ejecución que representa y, por otro lado, una misma ejecución puede representarse con más de una corrida.

Sea p un proceso denominado *monitor*, encargado de construir el estado global del sistema en que participa. De primera instancia se puede proponer que el monitor envíe mensajes hacia todos los procesos del sistema, interrogándolos acerca de su estado local. Un proceso i que reciba este mensaje, contestará devolviendo su estado local σ_i. Una vez que todos los procesos del sistema hayan respondido, p puede construir el estado global $(\sigma_1 \dots \sigma_n)$. Esta solución inicial tiene una deficiencia seria, sabiendo que el monitor está sujeto a las mismas incertidumbres que el resto del sistema, es evidente que el estado global que construya puede carecer de un significado real. Visto de otro modo, mientras que cada corte de una ejecución distribuida corresponde con un estado global, solo algunos cortes corresponden con estados que *pudieron* ocurrir. En la figura 6.1 se observan dos cortes realizados sobre una ejecución distribuida, en el primer caso (de izquierda a derecha) el corte representa un estado global que sí ocurrió. Por otro lado, el segundo caso representa un estado global que nunca pudo ocurrir puesto que, en la frontera del corte, el proceso 2 reporta la recepción de un mensaje que el proceso 3 todavía no transmite. Esta clase de cortes pueden llevar a conclusiones equivocadas acerca de la condición del sistema.

La relación de precedencia causal $\rightarrow$ resulta ser una herramienta muy valiosa para distinguir entre los dos tipos de corte recién descritos. Se dice que un corte C es *consistente*, si para todos los eventos e y e'

$$(e \in \mathrm{C}) \wedge (e' \rightarrow e) \Rightarrow e' \in \mathrm{C},$$

y en cualquier otro caso es *inconsistente*. Luego, un *estado global consistente* será aquel que corresponda con un corte consistente. Igualmente, una corrida R se dice *consistente* si para todos los eventos $e' \rightarrow e$ implica que e' aparece antes que e en R. En otras palabras, el orden total impuesto por R sobre los eventos es una extensión del orden parcial definido por la relación de precedencia causal. Se entiende que una corrida $R = e^1 e^2 \dots$ produce una secuencia de estados globales $\Sigma^0 \Sigma^1 \Sigma^2 \dots$, en la que Σ^0 representa el estado global inicial $(\sigma_1^0, \dots, \sigma_n^0)$. Por ello, en lo sucesivo, se usa el término "corrida" para referirse a una secuencia de eventos, así como a la secuencia de estados a que se da lugar.

6.3. Fotografía instantánea

Se sabe que, dependiendo del papel que juegue p en la construcción del estado global, se produce una estrategia de monitor pasivo (como la expuesta en la sección anterior) o de monitor activo, donde p interroga a los demás procesos acerca de sus estados locales en vez de esperar a que estos envíen sus notificaciones. Esta última también se conoce como *protocolo de instantánea* porque se considera que p "congela" o toma una "instantánea" (snapshot) del estado de cada proceso individual, para después reunir una colección de "fotografías" con las que arma una "panorámica" del sistema. Como se observaba previamente, el estado global producido por la instantánea puede no ser consistente y, por tanto, encierra el riesgo de llevar a deducciones equivocadas.

En esta sección se presenta una estrategia para producir instantáneas consistentes, basada en [12]. Se omiten los detalles del procedimiento mediante el que los procesos devuelven sus estados locales a p y se supone que los canales ofrecen un servicio de entrega FIFO.

Sean i y j dos procesos entre los que existe un enlace de comunicación. Se define el estado χ_{ij} de éste canal, como el conjunto de mensajes expedidos por i que aún no ha recibido j. Se denota como ENT_i, al conjunto de procesos que comparten un canal con i, desde los que puede recibir mensajes, en tanto, SAL_i es el conjunto de procesos que comparten un canal con i y a los que puede enviar mensajes. Un canal asociado con un proceso en ENT_i se denomina *entrante*, mientras que uno asociado con un proceso en SAL_i, se denomina *saliente*. Para cada ejecución del protocolo, un proceso i registra su estado local σ_i y los estados de cada uno de sus canales entrantes (χ_{ji}, para toda $j \in ENT_i$).

En la primera versión del protocolo de instantánea, los procesos pueden leer un reloj de tiempo real y acuerdan registrar sus estados locales cuando el reloj marque el valor t_r. Para ello, p escoge y propaga ésta fecha con suficiente antelación, como garantía de que todos los procesos se enterarán de ella antes de que ocurra.

Protocolo de instantánea (versión 1):

1. El proceso p envía el mensaje "tómate la foto al tiempo t_r", a todos los procesos del sistema.
2. Cuando el reloj marca la hora convenida t_r, cada proceso i registra su estado local σ_i, envía un mensaje vacío [2] sobre cada uno de sus canales salientes y comienza a registrar todos los mensajes recibidos por sus canales entrantes. El registro del estado local y la transmisión de mensajes se realizan antes de completar cualquier evento relacionado con la ejecución subyacente.
3. Tan pronto como recibe un mensaje desde j con una estampilla de tiempo mayor o igual a t_r, i declara a χ_{ji} como la colección de mensajes hasta entonces registrados.

Un evento e forma parte del corte consistente C_r asociado al estado global construido, si y solo si $RC(e) < t_r$. Por tanto,

$$(e \in \mathrm{C}_r) \wedge (RC(e') < RC(e)) \Rightarrow (e' \in \mathrm{C}_r).$$

Como el reloj de tiempo real utilizado en esta versión satisface la condición de consistencia débil de reloj, la ecuación implica que C_r es un corte consistente. Puesto que esta condición es el único requisito,

[2] este mensaje sirve para garantizar la terminación del protocolo.

se infiere que también puede usarse un reloj lógico, *LC*, con el mismo fin. Sin embargo, en el protocolo (1) se utilizan dos propiedades de los sistemas síncronos que deben replantearse para adaptarlo:

1. La instrucción "cuando $LC = t$ haz S " no tiene sentido en el nuevo contexto, ya que un reloj escalar no necesariamente se actualiza de manera que alcance un valor particular. Además, aun cuando *LC* pueda marcar el valor t, la instrucción sigue implicando un problema: las reglas de incremento de los relojes lógicos se basan en la ocurrencia de nuevos eventos entonces, para cuando $LC = t$, se habrá ejecutado el evento que causo la actualización de *LC*, en vez del primer evento de S.
   Para resolver el problema se plantean las siguientes reglas. Supóngase que i contiene la instrucción "cuando $LC = t$ haz S", donde S genera solo eventos internos o de transmisión. Antes de ejecutar un evento e, el proceso i realiza la siguiente prueba:

   - Si e es un evento interno o una transmisión y $LC = t - 2$, i ejecuta e y enseguida comienza la ejecución de S.
   - Si e es un evento de recepción que transporta al mensaje m, cuya estampilla es mayor que t y $LC < t - 1$, entonces i pone al mensaje de regreso en el canal (como si apenas fuera a recibirlo), fija *LC* a $t - 1$ y comienza la ejecución de S.

2. En el protocolo (1), el monitor escoge el valor de t_r para que todos los procesos sepan de éste con anticipación. En un sistema asíncrono, en cambio, p no puede calcular un valor semejante. En su lugar, se escoge un valor entero t_l suficientemente grande como para que, según las reglas de actualización, ningún reloj pueda alcanzarlo antes de que todo el sistema esté prevenido.

Al suponer la existencia de t_l, se requiere acotar la velocidad relativa de los procesos y los retardos de mensajes. Con estas consideraciones se propone la versión del protocolo (2):

Protocolo de instantánea (Versión 2):

1. El proceso p envía el mensaje "tómate la foto al tiempo t_l", a todos los procesos del sistema y ajusta su reloj lógico en un valor igual a t_l.
2. Cuando el reloj marca la hora convenida t_l, cada proceso i registra su estado local σ_i, envía un mensaje vacío sobre cada uno de sus canales salientes y comienza a registrar todos los mensajes recibidos por sus canales entrantes. El registro del estado local y la transmisión de mensajes se realizan antes de completar cualquier evento relacionado con la ejecución subyacente.
3. Tan pronto como recibe un mensaje desde j con una estampilla de tiempo mayor o igual a t_l, i declara a χ_{ji} como la colección de mensajes hasta entonces registrados.

Acerca de ésta última versión se puede observar que un proceso no efectúa ninguna acción, relacionada con la construcción del estado global, en el lapso que transcurre entre la recepción del mensaje "tómate la foto..." y la recepción del primer mensaje vacío que lleva su reloj local hasta un valor t_l. Por tanto, puede prescindirse del primer mensaje y usar solo el mensaje vacío dándole un significado diferente, por ejemplo, "tómate la foto ahora", al mismo tiempo se programa a cada proceso para comenzar a grabar su estado local tan pronto como reciba el nuevo mensaje. De esta forma se elimina también la necesidad de usar estampillas en los mensajes y el sistema en conjunto se libera de cualquier referencia de tiempo lógico.

Algoritmo 6.1: Protocolo de instantánea (Versión 3), algoritmo de Chandy-Lamport en el nodo i

```
1  Al recibir FOTO desde j efectúa
2      si (visitado = falso) entonces
3          visitado ← verdadero
4          registra_estado_local()
5          cuenta ← cuenta − 1
6          para todo k ∈ vecinos haz
7              envía FOTO
8              canal[k,i] ← ∅
9              cuenta ← cuenta + 1
10         cierra_registro_canal(j,i)
11     otro
12         cuenta ← cuenta − 1
13         cierra_registro_canal(j,i)
14         si (cuenta = 0) entonces termina
15 Al recibir M ≠FOTO desde j efectúa
16     si (visitado = verdadero) entonces
17         canal[j,i] ← canal[j,i] ∪ {M}
```

Protocolo de instantánea (Versión 3):

1. El proceso p inicia el protocolo enviándose a sí mismo el mensaje "tómate la foto ahora".
2. Si j es el proceso desde el que i recibe el mensaje "tómate la foto ahora", por vez primera, entonces i registra su estado local σ_i y reexpide el mismo mensaje sobre todos sus canales salientes. Ningún evento relacionado con la ejecución subyacente se debe realizar entre estos pasos. El estado del canal χ_{ji} se define como vacío, mientras i comienza el registro de sus demás canales entrantes.
3. Si j es un proceso desde el que i recibe el mensaje "tómate la foto ahora", después de la primera vez, entonces i detiene el registro de mensajes sobre el canal entrante que lo comunica con j y define a χ_{ji} como el conjunto de mensajes recibidos hasta entonces.

Cuando el proceso i ha recibido un mensaje de foto, desde cada uno de sus canales entrantes, su contribución al estado global está terminada y concluye su participación en el protocolo de foto instantánea distribuida.

Por la manera en que se depuró el protocolo, a lo largo de sus diferentes versiones, se garantiza la consistencia del estado global Σ^s que produce. Sin embargo, es posible que la corrida real del sistema jamás pase por Σ^s. A pesar de ello, no se trata de un estado global arbitrario, sino que posee útiles propiedades que lo relacionan con la corrida que lo genera.

Sea Σ^a el estado global del sistema al momento de iniciarse el protocolo de instantánea distribuida, Σ^f el estado al momento en que termina el protocolo y, Σ^s el estado construido por el protocolo. Se argumentará que existe una permutación de los eventos que produce una corrida consistente y que lleva al sistema desde Σ^a hasta Σ^f, pasando por Σ^s.

Sea R' la corrida real seguida por el sistema durante la ejecución del protocolo. Si e_i^* denota el evento que arranca el protocolo en i, se dice que un evento e_i en i es un evento de *pre-registro*, si $e_i \rightarrow e_i^*$ y, en cualquier otro caso, será un evento de *pos-registro*.

Considérese una pareja adyacente de eventos $< e, e' >$ en R', tales que e es un evento de pos-registro y e' es un evento de pre-registro. Si se establece que $\neg(e \rightarrow e')$, entonces pueden intercambiarse resultando una nueva corrida que también es consistente. Si se continúa con este procedimiento de intercambio entre eventos de pos y pre registro, en algún momento se termina con una corrida consistente en donde no existe un evento de pos-registro precediendo a otro de pre-registro. El estado global $\Sigma^{s'}$ asociado con el último evento de pre-registro puede alcanzarse desde Σ^a y desde aquel puede llegarse también hasta Σ^f pero, lo más importante es que $\Sigma^{s'}$ es justamente el estado producido por el protocolo de instantánea.

Supóngase que $e \rightarrow e'$, entonces existen dos casos que deben considerarse:

1. Ambos eventos ocurren en el mismo proceso. Esto significa, por definición, que e' es también un evento de pos-registro.
2. El evento e es una transmisión desde i, mientras que e' es su correspondiente recepción en j. Sin embargo, para entonces i ya habrá enviado un mensaje "tómate la foto" a j y, puesto que los canales tiene un servicio FIFO, e' será un evento de pos-registro.

En consecuencia, un evento de pos-registro no puede preceder causalmente a un evento de pre-registro y, por tanto, cualquier pareja de eventos $< \text{pos} - \text{registro}, \text{pre} - \text{registro} >$ puede intercambiarse. Enseguida, hay que observar que un evento de pos-registro que almacena un estado local, lo debe hacer en un punto e. Además, los estados de los canales que son anotados corresponden con mensajes enviados por eventos de pre-registro y recibidos como evento de pos-registro. Por construcción, estos mensajes son puestos en el canal después de la ejecución de e y, en conclusión, $\Sigma^{s'}$ es Σ^s.

En el algoritmo 6.1 se muestra el protocolo de instantánea (3) considerando el formato convencional utilizado a lo largo del texto. Los mensajes intercambiados por los nodos son los siguientes:

`FOTO`:	Es el marcador o mensaje del algoritmo Chandy-Lamport que dispara la foto en el proceso que la recibe por primera vez.
`M`:	Se refiere a cualquier mensaje perteneciente a la ejecución subyacente a la que se le está "tómando la foto" .

Las variables consideradas son las siguientes:

visitado:	Es un valor booleano, inicialmente falso.
cuenta:	Inicialmente es 0, registra el número de vecinos pendientes de comunicarse.
$canal[k,i]$:	$\forall k \in vecinos$, inicialmente vacío.

El procedimiento $registra\_estado\_local()$ se refiere al registro de las variables locales del algoritmo en ejecución al que se le está tomando la foto. Por otro lado, el procedimiento $cierra\_registro\_canal(j,i)$, se refiere a que se dejen de almacenar en $canal[k,i]$, los mensajes que se han recibido desde k.

6.4. Comentarios finales

La construcción de un estado global es un tema con importantes aplicaciones y extensiones [38] [45]. En este capítulo se ha descrito una técnica para determinar el estado global Σ^s de una ejecución distribuida pero, en vista de los retardos de la comunicación inherentes al sistema, Σ^s solo puede reflejar un instante del pasado. Entonces, para cuando se tiene la posibilidad de evaluar un predicado basado en el estado global construido, las condiciones del sistema pueden haber cambiado. No todo está perdido, existen muchas propiedades que se requieren detectar y que, una vez que se presentan, no se modifican. Tales propiedades (y sus predicados) se llaman *estables*. Entre las propiedades estables de un sistema distribuido se pueden citar: el interbloqueo (deadlock), la terminación, la pérdida de ficha (token), la eliminación de información obsoleta (garbage collection).

Tratándose de propiedades no estables, se contemplan dos clases de predicados cuya evaluación puede resultar útil aun cuando se refiere a una situación del pasado:

1. Se dice que cierto predicado *posiblemente* se ha cumplido, cuando existe una observación de una ejecución, para la que el predicado se evalúa como verdadero durante algún estado global.
2. En tanto, se dice que un predicado *definitivamente* se ha cumplido, si para toda observación de una ejecución existe un estado global en el que se evalúa como verdadero.

El análisis basado en relaciones de causalidad permite realizar inferencias sobre estados globales, basadas en observaciones locales. Sin embargo, para que estas conclusiones tengan sentido, se debe garantizar que toda la comunicación tiene lugar dentro de las fronteras de un sistema de cómputo, o sea que, aparte de los canales de comunicación por los que se intercambian mensajes, no existe otro fenómeno físico a través del cual un proceso pueda modificar el estado de otro. Si no es así se dice que el sistema posee *canales ocultos* [31], en cuyo caso la alternativa es utilizar observaciones hechas a partir de un reloj global de tiempo real, para construir un orden total.

Ejercicios

6.1. Imaginemos un sistema distribuido que, luego de varios días de ejecución, reporta que uno de sus procesos ha quedado fuera de operación. En este escenario, el peor remedio sería reponer todo desde el inicio, ¿cómo podríamos utilizar el algoritmo de Chandy-Lamport para ofrecer una mejor solución?

RESPUESTA: La situación a la que nos referimos se conoce como “rollback” y consiste en “rebobinar” una ejecución distribuida hasta un punto de su pasado reciente, en el que el componente caído se sabe que aún estaba en operación, entonces se le reemplaza por un componente de reserva y se echa a andar el sistema de nueva cuenta. Las fotos que dan cuenta de la historia de la ejecución pueden obtenerse usando el protocolo de Chandy y Lamport.

6.2. ¿Cómo podría usarse un sistema de relojes lógicos para construir una corrida consistente?

SUGERENCIA: Utilizamos un reloj vectorial, que sabemos exhibe la consistencia fuerte, para fechar todos los eventos de la ejecución, y luego construimos un orden sobre los eventos, en el que se respete el orden de las etiquetas.

6.3. ¿Por qué se requiere la propiedad de consistencia débil, para pasar de la versión 1 a la 2, del protocolo de foto instantánea presentado en este capítulo?

RESPUESTA: Sabemos que un corte C_r es consistente si

$$(e_2 \in C_r)\&(e_1 \rightarrow e_2) \Rightarrow e_1 \in C_r$$

Pero, por otro lado, un reloj de consistencia débil significa que

$$e_1 \rightarrow e_2 \Rightarrow RC(e_1) < RC(e_2)$$

lo que transforma la primera expresión en

$$(e_2 \in C_r)\&(RC(e_1) < RC(e_2)) \Rightarrow e_1 \in C_r$$

Para esta última construcción no necesitamos echar mano de la propiedad

$$RC(e_1) < RC(e_2) \Rightarrow e_1 \rightarrow e_2$$

6.4. Imaginemos un algoritmo PI, sabemos que el nodo que lo inicia está imposibilitado para determinar cuando el algoritmo termina globalmente. Explique de qué manera podría usarse un protocolo de foto instantánea para resolver esta limitación, ¿cuál sería el precio a pagar?, ¿de qué otra forma puede atacarse el problema?

SUGERENCIA: El nodo que arranca PI puede estimar cuando el protocolo ha alcanzado las fronteras del grafo y entonces disparar la fotografía. Esta revelará la condición del sistema. Si se equivoca, entonces deberá repetir el procedimiento hasta tener la certeza que PI ha terminado. Cada fotografía tiene un costo de 2m mensajes. Alternativamente, puede sustituirse PI por PIF y por el mismo precio inicial conseguir la detección de terminación global.

Parte III
Fallas

Si dos individuos están siempre de acuerdo en todo, puedo asegurar que uno de los dos piensa por ambos.

—Sigmund Freud

7 Consenso síncrono

Resumen El consenso es un problema fundamental del cómputo distribuido. Se tiene un conjunto de n procesos y se requiere entonces que los participantes elijan un valor que todos terminen compartiendo. Un protocolo que resuelve el problema debe poseer tres propiedades: i) acuerdo, los procesos deben coincidir en el valor final que deciden, ii) validez, el valor decidido debe haber sido propuesto por alguno de los participantes (de otra forma se puede conseguir una solución trivial) y iii) terminación, la decisión debe alcanzarse al cabo de un tiempo finito. Se trata de un problema cuya formulación es muy sencilla y que, sin embargo, puede complicarse cuando existe la posibilidad de fallas en los procesos o los medios con que estos se comunican. Asimismo, los modelos de tiempo y comunicación juegan un papel fundamental en la solución del problema. Este capítulo ofrece una vista en amplitud del problema y luego se concentra en revisar tres situaciones diferentes en la que es posible resolverlo.

7.1. Introducción

El consenso se presenta cada vez que los participantes del sistema deben coincidir en una decisión compartida por todos. Por ejemplo, cuando un conjunto redundante de instrumentos debe emitir un valor único de sus registros, también cuando se tienen replicas de una base de datos y, para mantener la coherencia de las mismas, se necesita coincidir en las operaciones que deben efectuarse sobre cada una de las replicas. El consenso se relaciona con otros problemas como la difusión (confiable, atómica, causal, etc.), o la elección. La importancia de este problema radica en el hecho de que es la base para construir una visión coherente y unificada del estado de un sistema distribuido. Sin embargo, el problema adquiere

diferentes grados de dificultad, dependiendo de las condiciones bajo las que opera el conjunto de procesos que requieren alcanzar el consenso. Esto es, depende de aspectos tales como el modelo de tiempo (síncrono, asíncrono), el modelo de comunicaciones (con intercambio de mensajes, por memoria compartida) pero, sobre todo, depende del tipo de fallas que puede presentarse en los procesos participantes, así como en los medios a través de los cuales se comunican.

En un sistema síncrono, los procesos operan por rondas. A su vez, cada ronda se divide en tres fases: transmisión, recepción y procesamiento. Todos los procesos que participan en el sistema inician simultáneamente cada una de las fases por las que se desarrolla la ronda. Durante la transmisión, cada proceso intercambia mensajes, si acaso tiene, con aquellos vecinos con los que comparte un canal de comunicación. Cada mensaje que se transmite se termina de recibir al cabo de la siguiente fase. Finalmente, cada participante ejecuta su fase de procesamiento a partir de su estado corriente y los mensajes que recién haya recibido. En contraste, en un sistema asíncrono no se asume ninguna cota que limite la duración de las operaciones que desarrolla cualquiera de los procesos participantes.

Cuando los procesos se comunican por un modelo de memoria compartida, se tiene un espacio común de lectura y escritura, sobre el que los procesos emisores operan para depositar información que requieren intercambiar con otros. A su vez, un proceso destinatario, que en esencia puede ser cualquiera, acude por igual a este espacio común para recoger una copia de la información que le concierne. Si tenemos tres procesos i, j y k, y queremos que i comunique un mensaje a los demás, entonces la operación se reduce a conseguir acceso al espacio común para escribir un "recado", esperando que los destinatarios acudan al mismo espacio para leerlo. En contraste, en el modelo de comunicación con intercambio de mensajes, se asume que existen canales entre algunas parejas de procesos. Dependiendo de los detalles del modelo, un canal puede ser unidireccional o bidireccional. Un proceso emisor transmite o "escribe" un mensaje en un extremo de un canal y, en el otro extremo, el proceso de destino lo recibe, al cabo de un tiempo finito. Si retomamos el ejemplo recién presentado, se trata de que i tiene un canal dirigido a j y otro canal diferente, dirigido a k, entonces i transmite la misma información por cada canal, esto es, una vez hacia j y otra hacia k.

Por cuanto concierne a las fallas, nos referimos a una contingencia o eventualidad espontánea, que saca a un proceso de su operación especificada. Se dice que un proceso cae en falla de paro, si interrumpe su operación de manera súbita y permanente. Si el proceso presenta fallas de omisión, significa que presenta lapsos de tiempo durante los cuales deja de transmitir o recibir mensajes. Si presenta un comportamiento bizantino, se entiende que el proceso puede mostrar una operación arbitraria.

El consenso fue descrito por primera vez en el trabajo de Lamport, Shostak y Pease [33]. Posteriormente, Fischer, Lynch y Paterson [21] demostraron la imposibilidad de la existencia de un protocolo determinista que resuelva el consenso en un sistema asíncrono por intercambio de mensajes, cuando existe el riesgo de que al menos uno de los procesos experimente fallas de paro. En esencia, este resultado nos dice que, en vista de la falta de cotas que delimiten la duración de las operaciones, es imposible distinguir entre un proceso caído en paro y un proceso en servicio (pero muy lento). Podemos pensar en este importante resultado como un punto en el mapa que se encuentra en el "territorio" de la imposibilidad. Sin embargo, nos interesa saber también donde comienza este territorio. Dicho de otro modo, cuál es el conjunto mínimo de condiciones que hacen posible la solución del problema.

Por su parte, Herlihy demostró la imposibilidad del consenso en un sistema distribuido con procesos asíncronos y una memoria compartida que soporte solamente operaciones de lectura y escritura [27].

En tanto, si se garantiza que menos de la mitad de los procesos puede fallar, Attiya, Bar-Noy y Dolev [3] establecieron que el sistema de mensajes puede emular confiablemente un sistema de memoria compartida. En estas circunstancias, el resultado de Fischer, Lynch y Patterson equivale al de Herlihy.

Chandra y Toueg [11] definieron una familia de mecanismos parcialmente (o débilmente) síncronos denominados detectores de fallas, a partir de los cuales es posible resolver el consenso. Un detector de fallas es un módulo que acompaña a un proceso mientras ejecuta un algoritmo distribuido, y le sirve para estimar el estado de los otros componentes activos del sistema con quienes intercambia información. Este módulo puede emitir diagnósticos equivocados y, por tanto, considerar que un proceso sigue en funcionamiento cuando en realidad ha caído en falla, o al revés, sospechar que un componente ha caído en falla, cuando no es así (solo es muy lento para comunicarse). A pesar de las imprecisiones de su diagnóstico, la ventaja de un detector de fallas se encuentra en la posibilidad de evitar que un algoritmo distribuido se quede estancado, esperando la comunicación con un proceso que no envía información a tiempo. Visto de otra forma, el detector decide en un plazo de tiempo acotado sobre la condición de un proceso del que se espera información. Esta decisión puede ser errónea y posiblemente pueda revocarse.

En contraste con el enfoque clásico de teoría de la confiabilidad, en computación distribuida se cuenta el número de componentes que pueden causar fallas y no las ocurrencias de un comportamiento que se desvía de su especificación. Se habla de que un sistema es f-tolerante cuando puede continuar satisfaciendo su especificación siempre que no se presenten más de f componentes que puedan generar fallas. La caracterización estadística de un sistema, usando medidas como el tiempo medio entre fallas (mean time to failure) y la probabilidad de fallas (probability of failure) son importantes para los usuarios. Sin embargo, existen ventajas reales al describir un sistema en término del máximo número de componentes deficientes que puede tolerar sobre un intervalo de interés. Afirmar que un sistema es f-tolerante es una medida de las capacidades inherentes a la arquitectura del sistema, en contraste con la tolerancia que se consigue usando componentes con un cierto grado de confiabilidad, y que dependen de la tecnología con que se implementa.

Siempre que las circunstancias del problema garanticen su solución, es posible construir un procedimiento para resolver el consenso. Como puede esperarse, este procedimiento debe atender a los modelos de tiempo, comunicaciones y fallas que se asumen. Cuando se tiene un sistema totalmente síncrono, el consenso es un "camaleón" que se transforma: se sabe que el problema tiene solución cuando se presentan fallas de paro si el número de rondas para resolverlo es mayor que el número de fallas que pueden tolerarse, también se sabe que cuando se presentan fallas de omisión, el problema tiene solución siempre que el número de casos en falla no sea la mayoría. Por último, cuando se presentan fallas bizantinas, el problema tiene solución siempre que el número de casos en falla no exceda la tercera parte del número de participantes.

En este capítulo se presentan tres algoritmos de consenso síncrono para tres tipos de falla: de paro, de omisión y bizantinas. Se busca enfatizar en sus principios de diseño, y ofrecer una comprensión clara sobre las suposiciones adicionales que un protocolo de consenso requiere. La sección 7.2 presenta el modelo de cómputo sobre el que habrán de funcionar las soluciones desarrolladas, también se establece la definición formal del problema. En las secciones 7.3, 7.4, y 7.5 se consideran los casos de fallas de paro, de omisión y bizantinas, respectivamente. Por último, la sección 7.6 ofrece algunas observaciones a manera de conclusión. El resto de este capítulo está basado en las ideas de [44].

La notación utilizada en este capítulo se presenta en la tabla 7.1.

Π	Conjunto de procesos
P	Subconjunto de procesos
(i, j)	Canal de comunicación entre los procesos i y j
$\prec$	Relación de precedencia
r	Número de ronda
f	Número máximo de fallas
t	Número real de fallas
$\mathscr{V}$	Conjunto de valores
v	Un valor
val	Valor mínimo propuesto
b	Número de bits

Tabla 7.1: Notación

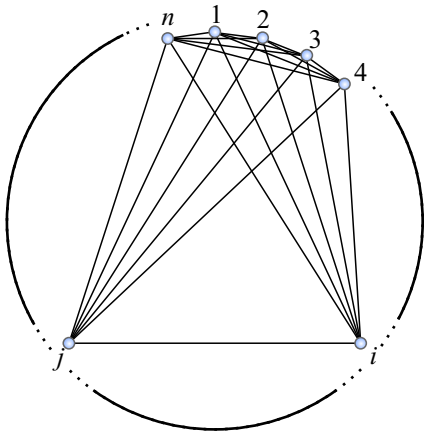


Figura 7.1: La comunicación entre los participantes del consenso síncrono

7.2. El modelo y la formulación del problema

El modelo de sistema sobre el cual se revisan los escenarios del consenso síncrono se describe como conjunto finito de procesos $\Pi = \{1, \ldots, n\}$ que se comunican enviando y recibiendo mensajes a través de canales. Cada pareja de procesos i y j se conecta por un canal denotado como (i, j), véase la figura 7.1.

En un sistema síncrono, cada una de sus realizaciones consiste de una secuencia de rondas que se identifican por enteros consecutivos 1, 2, etc. Para un proceso que participa en la ejecución, el número de ronda se observa como una variable global r que puede leerse y cuya actualización queda a cargo del subsistema de tiempo. Una ronda se compone de tres fases consecutivas:

- Una fase de *transmisión*, durante la cual cada proceso envía mensajes.
- Una fase de *recepción*, durante la cual cada proceso recibe mensajes. La propiedad fundamental del modelo síncrono se encuentra en el hecho que cada mensaje enviado por el proceso i al proceso j, durante la ronda r, se recibe en j durante la misma ronda.
- Una fase de *procesamiento*, durante la cual cada receptor procesa los mensajes recién recibidos y ejecuta un cómputo local.

Por lo que concierne al subsistema de comunicación, se le considera libre de fallas, es decir, no se da la generación espontánea, alteración, pérdida o duplicación de mensajes.

Un proceso presenta una *falta* (fault) durante una ejecución, si su funcionamiento se desvía de lo prescrito por su algoritmo, en otro caso se le dice *correcto*. Debe suponerse que existe un número máximo, f, o cota superior, de procesos con faltas.

Un *modelo de falla* define de qué manera un proceso con falta se desvía de su especificación [26]. Los siguientes tres modelos de falla serán considerados:

- Falla de *paro*. En la que un proceso con falta detiene su ejecución prematuramente, alternativamente se dice que cae en paro o "muere", después de lo cual se le considera fuera de operación. Hay que observar que si un proceso falla a la mitad de la fase de transmisión, entonces podría transmitir solamente un subconjunto de los mensajes que debía enviar.
- Falla de *omisión*. En la que un proceso con falta muere u omite mensajes que se supone debía enviar (omisión de transmisión) u omite mensajes que se supone debía recibir (omisión de recepción). Aquí hay que observar que un proceso puede omitir el envío o la recepción de mensajes durante algunas rondas y posteriormente caer en paro.
- Falla *bizantina*. En la que un proceso con falta puede exhibir un funcionamiento arbitrario (iniciar en un estado arbitrario, enviar mensajes arbitrarios, efectuar una transición de estado arbitraria, etc.)

Puede verse que estos tres modelos son de una severidad creciente: *paro* $\prec$ *omisión*$\prec$ *bizantina*, por cuanto cualquier protocolo que resuelva un problema en un modelo de falla $\mathscr{A}$, lo resolverá también en un modelo de falla menos severa $\mathscr{B}$, i.e. tal que $\mathscr{B} \prec \mathscr{A}$ [26].

En el problema que nos ocupa, cada proceso i propone un valor v y todos los procesos correctos tienen que decidir un valor *val*, a partir del conjunto $\mathscr{V}$ de valores propuestos. Además, la solución del problema debe reunir las siguientes propiedades:

- *Acuerdo*. Significa que no existen dos procesos correctos que decidan valores diferentes.
- *Validez*. Significa que si un proceso decide el valor *val*, entonces *val* fue propuesto por algún proceso.
- *Terminación*. Significa que cada proceso en algún momento llega a tomar una decisión.

Debe notarse que la propiedad de acuerdo solo concierne a los procesos correctos, es decir, permite que un proceso que presenta una falta decida un valor diferente del elegido por los procesos correctos. Esta propiedad puede resultar demasiado débil para algunas aplicaciones donde se requiere escoger un solo valor independientemente si el proceso muestra una falta o no. El *consenso uniforme* es una forma restringida de acuerdo que opera en estos escenarios. En concreto, se define por las dos últimas propiedades recién presentadas, más la siguiente condición de:

- *Acuerdo uniforme*. Significa que no existen dos procesos (correctos o no) que decidan valores diferentes.

Por otro lado un proceso bizantino puede decidir cualquier valor, por lo que es imposible que un protocolo diseñado para resolver consenso sobre un modelo de falla bizantina pueda garantizar acuerdo uniforme o validez. En esta situación parece razonable que la especificación del *consenso bizantino* incluya propiedades de terminación, acuerdo y la siguiente

- *Validez débil*. Significa que si todos los procesos correctos proponen el mismo valor v, entonces un proceso correcto decide v.

Algoritmo 7.1: Algoritmo de consenso bajo fallas de paro ($f < n$), en el nodo i

```
   /*condiciones iniciales*/
1      val ← v
2      previo ←⊥
3      durante la ronda r = 1 hasta f + 1
4          si (previo ≠ val ) entonces
5              para toda j ≠ i haz envía val a j
6          previo ← val
7          si (Recibidos ≠ ∅) entonces
8              val ← min({val}∪Recibidos)
9      regresa val
```

7.3. Consenso síncrono bajo fallas de paro

¿Qué sucede cuando un proceso cae en falla de paro?, visto de otra forma, ¿cuáles son sus efectos?, sucede que el proceso afectado interrumpe sus actividades de manera permanente. Esto puede suceder en un momento arbitrario. Por ejemplo, a la mitad de una operación de transmisión, recepción o procesamiento.

El algoritmo 7.1 es una versión del conocido protocolo de consenso denominado *conjunto de inundación* [4, 36]. Cada proceso i invoca la versión local de este código donde v es el valor que propone inicialmente. La función termina devolviendo el valor acordado por consenso. El símbolo $\perp$ denota un valor por omisión que no puede ser propuesto. En la práctica sucede con frecuencia que el conjunto $\mathscr{V}$ de los valores que se proponen puede ordenarse por alguna relación binaria (que se denota como $<$). En este caso el cálculo se reduce al registro y actualización de una sola variable que contiene el mínimo visto por i hasta la última ronda. Dicho valor será el resultado que decida i al final de su ejecución. El algoritmo se compone de $f+1$ rondas. Cada proceso envía durante la ronda $r \leq f+1$, la información nueva que obtuvo durante la ronda $r-1$, véase la figura 7.2.

Además de su propio identificador i, el número de la ronda r, y el valor inicial v que propone, el nodo mantiene también una colección local de variables, donde registra el desarrollo del algoritmo:

val:	Registra el mínimo valor propuesto del que tiene conocimiento. Inicialmente vale v.
previo:	Guarda el valor del que se enteró en la ronda previa. Inicialmente vale $\perp$.
Recibidos:	El conjunto de los valores recibidos durante la última ronda. Por defecto es $\emptyset$.

Teorema 7.1. *El protocolo descrito por el algoritmo 7.1 resuelve el problema de consenso en un sistema síncrono en el que a lo más $f < n$ procesos pueden caer en paro.*

Demostración. La prueba de que este protocolo satisface las propiedades de terminación y validez es sencilla. La terminación se sigue del número acotado de rondas. La validez se sigue del hecho que el valor decidido fue propuesto por alguno de los participantes.

Para probar que el protocolo satisface la propiedad de acuerdo, se debe demostrar que cualesquiera dos procesos i y j, al terminar la ronda $f+1$, tienen el mismo valor *val* justo antes de decidir su

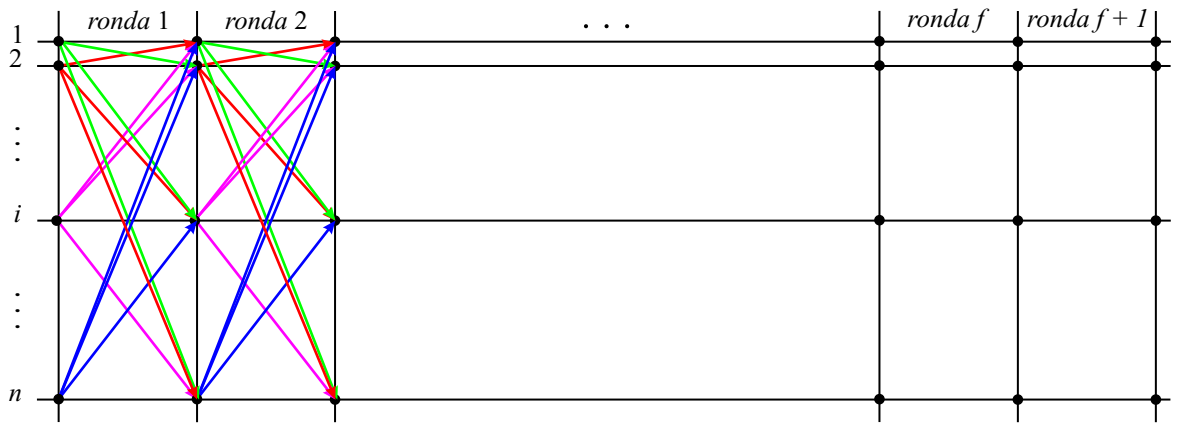


Figura 7.2: Ejecución del algoritmo de consenso síncrono bajo fallas de paro, escenario sin fallas

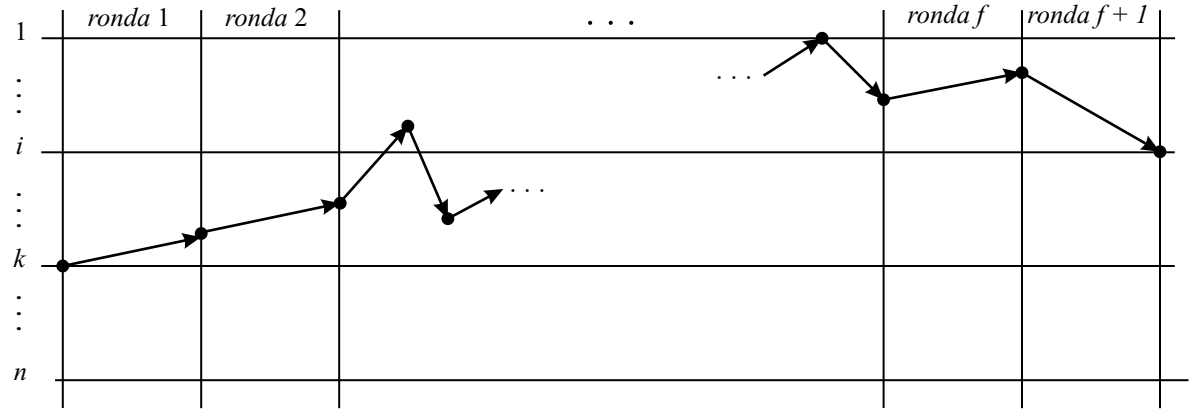


Figura 7.3: Escenario del peor caso para consenso tolerante a fallas de paro ($r = f + 1$)

resultado. Sea *val* el valor conocido por i al final de la última ronda y supongamos que i recibió esta valor por primera vez durante alguna ronda r (por ejemplo $r = 0$, si es el valor propuesto por él mismo). Hay que observar que, sin importar en que momento tomen su decisión, i y j ejecutarán todas las rondas hasta la $f + 1$. Existen dos casos posibles:

- $r < f + 1$. En este caso, i actualiza *val* durante la ronda r y lo transmite a j durante la ronda $r + 1$. Por tanto, j obtiene a *val* durante la ronda $r + 1 \leq f + 1$.
- $r = f + 1$. En este caso, supongamos que i consigue el valor *val* hasta la última ronda $f + 1$. Si esto ocurrió fue porque el proceso desde donde se propuso este valor solo pudo transmitirlo una vez y cayó en paro, solo un proceso recibió este valor, tuvo tiempo de reexpedirlo a otro proceso y también cayó en paro, luego de reexpedirlo a un solo proceso. Esta cadena de procesos que reciben el valor *val*, lo reexpiden una sola vez y caen, incluye a $f + 1$ procesos, a lo largo de $f + 1$ rondas pero, por hipótesis, solo hay f procesos en falla, por tanto, en este conjunto de procesos debe haber alguno correcto, que alcance a transmitir su valor a todos los demás procesos correctos (incluyendo j e i) en la ronda $f + 1$ (véase figura 7.3).

□

Evidentemente la complejidad en tiempo es $f+1$ rondas. Si ahora quisiéramos evaluar la cantidad de información necesaria para alcanzar el consenso, nos interesaría conocer cuántas veces se intercambian los valores propuestos entre los procesos que participan en el protocolo. Sea b el número de bits requeridos para codificar un valor propuesto. Un proceso i transmite el valor mínimo que ha reconocido a lo más una vez hacia cada uno de los otros $n-1$ procesos, consecuentemente la complejidad en bits tiene por cota superior $n(n-1)b \times \min(f+1, \mathscr{V})$, que para el caso de consenso binario (es decir, cuando $\mathscr{V}$ tiene solo dos elementos) se reduce a $2n(n-1)$. Adicionalmente, se puede mejorar si se evita que i envíe un valor al proceso del que recibe dicho valor.

Se ha demostrado que $f+1$ es una cota inferior para el número de rondas (véanse [1, 20, 36, 39]) lo que significa que independientemente del protocolo, siempre es posible requerir hasta de $f+1$ rondas en presencia de hasta f procesos que caen en paro. Sin embargo, podría pensarse que en las ejecuciones donde el número real de fallas (t) es menor que el total de fallas permitidas (f), el número de rondas podría ser menor [17]. Los protocolos recién propuestos no ofrecen esta propiedad de decisión *rápida* o temprana (early), ya que siempre requieren de $f+1$ rondas (aun cuando no hubieran fallas).

7.4. Consenso síncrono bajo fallas de omisión

Aunque el protocolo descrito en el algoritmo 7.1 resuelve el problema de consenso, en presencia de un número arbitrario de fallas de paro, no sirve para resolver el mismo problema en presencia de al menos una falla por omisión. La razón es la siguiente: sea k un proceso con falta que permanentemente omite la recepción de mensajes. Cuando k ejecuta la última ronda $f+1$ del protocolo 7.1, solo conoce el valor que ha propuesto y, consecuentemente, decide éste como resultado, sin importar el valor elegido por los procesos correctos.

Por suerte, las limitaciones del protocolo anterior no son inherentes al problema de consenso uniforme. A continuación se demuestra que la simple inserción de una ronda extra en el protocolo del algoritmo 7.1, permite garantizar la propiedad de acuerdo uniforme aun en presencia de fallas por omisión. Este nuevo protocolo aparece en el algoritmo 7.2.

La nueva idea detrás de este refinamiento es muy sencilla: se trata de evitar que un proceso con falta decida su resultado sin conocer el valor decidido por un proceso correcto. Esto quiere decir que durante la ronda adicional (i.e. $f+2$):

- Cada proceso i primero envía a todos los procesos (incluso a sí mismo) el valor que está por decidir, o sea *val*.
- Enseguida, espera por todos los mensajes que debe recibir durante la ronda $f+2$ y guardándolos en una *Bolsa* (no se trata de un conjunto pues esta estructura puede tener varias copias de un mismo valor).
- Finalmente, i decide basándose en un valor que ha recibido más de f veces.

Esta ronda adicional, evita que un proceso con falla termine su operación decidiendo un valor diferente. Hay que observar que un proceso que incurre en una falla de omisión es un proceso con falta y, consecuentemente, no se le requiere para tomar una decisión, luego la estrategia es correcta. De esta manera, un proceso con falta o bien decide correctamente, o no toma una decisión. También hay que

Algoritmo 7.2: Algoritmo de consenso bajo fallas de omisión ($f < n/2$), en el nodo i

```
   /*condiciones iniciales*/
1      val ← v
2      previo ←⊥
3      durante la ronda r = 1 hasta f + 1
4          si (previo ≠ val ) entonces
5              para toda j ≠ i haz envía val a j
6          previo ← val
7          si (Recibidos ≠ ∅) entonces
8              val ← min({val} ∪ Recibidos)
9      durante la ronda r = f + 2 haz
10         para toda j ≠ i haz envía val a j
11         si (∃ val ∈ Bolsa que aparece n − f veces o más) entonces
12             regresa val
```

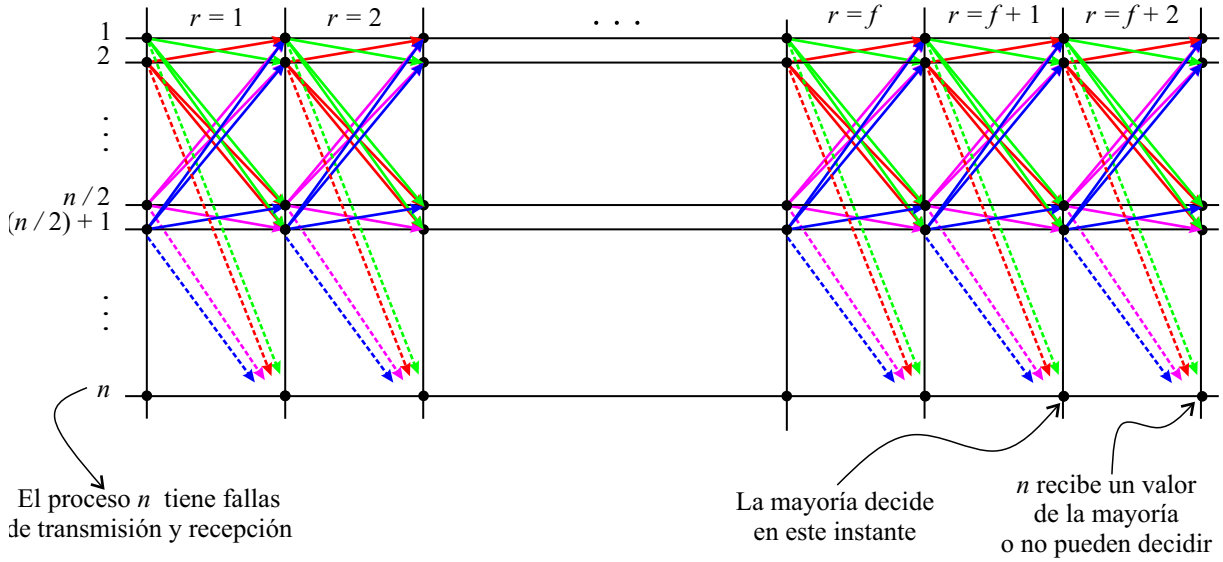


Figura 7.4: Una ejecución del algoritmo de consenso síncrono bajo fallas de omisión

observar que el protocolo muestra otra buena propiedad: un proceso que presenta fallas de omisión por transmisión puede decidir, véase la figura 7.4.

Teorema 7.2. *El protocolo expuesto en el algoritmo 7.2 resuelve el problema de consenso uniforme sobre sistemas síncronos en los que pueden presentarse hasta $f < n/2$ procesos que incurran en fallas de omisión. Para ello se requieren $f + 2$ rondas.*

Demostración. La prueba de la propiedad de validez se deja al lector. Para la propiedad de acuerdo uniforme, primero hay que observar que todo proceso correcto i tiene el mismo valor en su variable *val* al final de la ronda $f + 1$ (esto se sigue del hecho que el protocolo del algoritmo 7.1 satisface la propiedad de acuerdo aun cuando f procesos fallen por paro u omisión). Por tanto, al final de la ronda

$f+1$, al menos $n-f$ procesos tienen el mismo valor en *val*. Consecuentemente, si un proceso recibe al menos $n-f$ copias del mismo valor durante la ronda $f+2$, este valor es necesariamente *val*. Se cumple entonces la propiedad de acuerdo uniforme.

Para probar la propiedad de terminación debe mostrarse que los procesos correctos llegan a una decisión. Primero hay que observar que todos los procesos que no caen en paro (lo cual incluye al menos a los procesos correctos) ejecutan la ronda $f+2$. Por la propiedad de acuerdo, al final de la ronda $f+1$, cada proceso correcto tiene el mismo valor en su variable *val*, se infiere que al menos $n-f$ procesos difunden este valor durante la fase de transmisión de la ronda $f+2$. Por la hipótesis que asume correctos a la mayor parte de los procesos se tiene que $n-f \geq n/2$, de donde se concluye que cada proceso correcto recibe al menos $n-f$ copias de *val* y decide en consecuencia. □

El siguiente resultado que se analiza muestra que, cuando se busca resolver el problema de consenso uniforme bajo el riesgo de fallas por omisión, la restricción sobre el número de procesos con falta que pueden tolerarse no es una limitación del protocolo, sino un requerimiento inherente al problema.

Teorema 7.3. *No existe un protocolo que resuelva el consenso uniforme sobre un sistema síncrono en donde puedan presentarse hasta $f \geq n/2$ procesos que incurran en fallas de omisión*

Demostración. La prueba por contradicción se basa en un argumento clásico de partición. Supóngase que existe un protocolo A que resuelve el consenso binario uniforme (solo los valores 0 y 1 pueden proponerse) en presencia de hasta $f \geq n/2$ procesos con falta.

A continuación se dividen los procesos en dos subconjuntos disjuntos P_0 y P_1, tales que P_0 incluye f procesos y P_1 los restantes $n-f$ procesos. Considérense las tres ejecuciones siguientes:

- Ejecución X_0. Todos los procesos proponen el valor 0, todos los procesos en P_0 son correctos mientras que todos los procesos en P_1 caen en paro desde el inicio. Como por hipótesis el protocolo A es correcto, se concluye que los procesos en P_0 deciden (terminación) el mismo resultado (acuerdo uniforme) que es 0 (validez), esto es, el único propuesto.
- Ejecución X_1. Todos los procesos proponen el valor 1, todos los procesos en P_1 son correctos mientras que todos los procesos en P_0 caen en paro desde el inicio. De manera semejante al caso previo, puesto que por hipótesis el protocolo A es correcto, se concluye que los procesos en P_1 deciden (terminación) el mismo resultado (acuerdo uniforme) que es 1 (validez), es decir, el único propuesto.
- Ejecución X_{01}. Todos los procesos en P_0 proponen 0, mientras todos los procesos en P_1 proponen 1. Todos los procesos en P_0 son correctos y todos los procesos en P_1 tienen las siguientes faltas: incurren en fallas de omisión respecto a los mensajes que deben enviar a los procesos en P_0 y también tiene fallas de omisión respecto a los mensajes que deben recibir de los procesos en P_0. Por otro lado, no se presentan fallas de omisión al interior de P_1.

  Por un lado se tiene que las ejecuciones X_0 y X_{01} son indistinguibles para cualquier proceso $i \in P_0$ entonces, los procesos en P_0 deben decidir 0 en X_{01}.

  Por otro lado, las ejecuciones X_1 y X_{01} son indistinguibles para cualquier proceso $i \in P_1$ entonces, los procesos en P_1 deben decidir 1 en X_{01}.

  En conclusión, para la ejecución X_{01} algunos procesos deciden el valor 0, mientras otros deciden 1. Luego, el protocolo A no garantiza la propiedad de acuerdo uniforme (!).

□

7.5. Consenso síncrono bajo fallas bizantinas

Para completar este capítulo se presenta a continuación un protocolo de consenso que tolera procesos con funcionamiento arbitrario. Se sabe que $f < n/3$ es una condición necesaria para esta clase de protocolos [40].

Por razones didácticas se escogió un algoritmo [8] que requiere una suposición más restrictiva, esta es $f < n/4$, a cambio es muy simple y elegante. Adicionalmente, muestra una propiedad notable, solo utiliza mensajes de tamaño fijo (b bits) ya que cada mensaje transporta un solo valor. Además, el protocolo está basado en el paradigma de emphcoordinador rotatorio que ha probado ser de gran utilidad en el diseño de múltiples algoritmos distribuidos.

Como se estableció previamente, junto a las propiedades de terminación y acuerdo, un protocolo que tolere fallas bizantinas debe garantizar que el valor decidido sea v cuando todos los procesos correctos propongan v (validez débil). Para lograr esta meta, el protocolo que se muestra en el algoritmo 7.3 se basa en el siguiente principio: (1) Si el número de veces que ocurre el valor más propuesto rebasa un umbral, este será el valor que se decida. (2) En otro caso, el coordinador forzará a que un número suficientemente grande de procesos adopte el mismo valor, y con ello se garantice el cumplimiento de la condición anterior.

El protocolo utiliza una secuencia de etapas, cada una compuesta de dos rondas conceptualmente ligadas a las condiciones (1) y (2). Durante cada etapa, cada proceso efectúa un cálculo local para estimar el valor de decisión (almacenado en la variable local *val*, inicialmente igual a v) y el propósito de la secuencia de etapas es garantizar que un cierto valor llegue a presentarse en un número suficientemente grande de procesos como para rebasar el umbral. Concretamente, se tiene que:

- En la primera ronda de la etapa k (esto es, la ronda $r = 2k - 1$) se determina la estimación. Cada proceso intercambia el valor que almacena en *val*, luego determina el valor que observa el mayor número de veces y lo guarda en su variable *más_frecuente*.
- En la segunda ronda de la etapa k (esto es, la ronda $r = 2k$) se adopta una estimación. Para cada proceso i, si el número de ocurrencias del valor más frecuente rebasa el umbral, entonces i lo adopta como nuevo estimado. En otro caso se invoca la intervención del coordinador rotatorio, del modo siguiente: durante la ronda $r = 2k$, el proceso k funge como coordinador y difunde su valor *más_frecuente* para que cada proceso lo adopte como su propio variable (véase la figura 7.5).

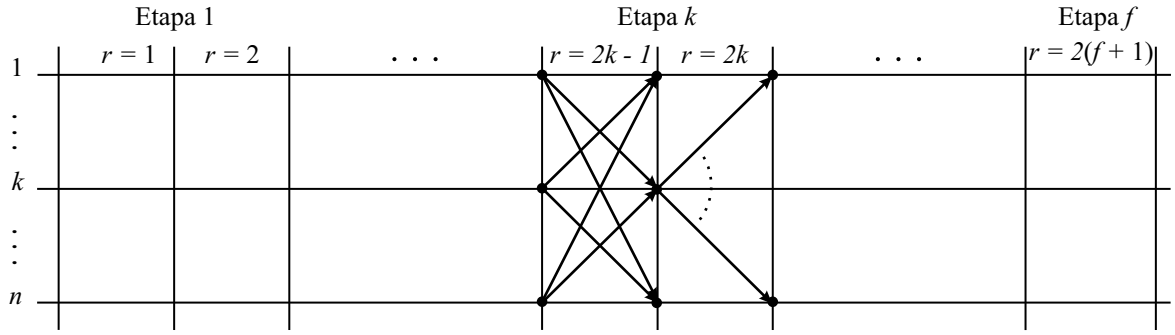


Figura 7.5: Una ejecución del algoritmo de consenso síncrono bajo fallas bizantinas

Algoritmo 7.3: Algoritmo de Berman-Garay para fallas bizantinas ($f < n/4$), en el nodo i

```
   /*condiciones iniciales*/
1      val ← v
2      durante la ronda r = 1,3,...,2f+1 haz
3          para toda j ≠ i haz envía val a j
4          Recibidos ← bolsa /*la bolsa tiene los valores recibidos durante esta ronda*/
5          actualiza más_frecuente
6          actualiza num_repeticiones
7      durante la ronda r = 2,4,...,2(f+1) haz
8          si (i = r/2) entonces
9              para toda j ≠ i haz envía más_frecuente a j
10         si ( valse recibe desde el proceso r/2) entonces val_coordinador ← val
11         otro
12             val_coordinador ← v
13         si (num_repeticiones > n/2+f) entonces val ← más_frecuente
14         otro
15             val← val_coordinador
16     regresa val
```

Dado que existen f procesos con falta, a lo más, a lo largo de $f+1$ etapas necesariamente existe una etapa cuyo coordinador es correcto. Por tanto, este coordinador impondrá el mismo valor estimado en todos los procesos correctos, si acaso no se ha presentado un valor estimado que rebase el umbral de decisión.

Además de su propio identificador i, el número de la ronda r, y el valor inicial v, que propone, el nodo mantiene también una colección local de variables, donde registra el desarrollo del algoritmo:

val:	Registra el valor intercambiado con los demás. Inicialmente vale v.
Recibidos:	La bolsa del proceso donde guarda los valores recibidos durante la última ronda. Se vacía al iniciar cada ronda.
más_frecuente:	El valor que se repite el mayor número de veces en *Recibidos*.
num_repeticiones:	Registra cuántas veces se repite *más_frecuente*, en *Recibidos*.
val_coordinador:	El valor que se recibe del coordinador en turno.

El valor de umbral al que se hace referencia será $n/2+f$. El siguiente resultado demuestra que se requiere este valor para garantizar que la propiedad de acuerdo no puede violarse aun en presencia de f procesos que muestren un comportamiento bizantino. El protocolo presentado mantiene el acuerdo tan pronto como los procesos correctos convergen al mismo valor estimado (propiedad de persistencia).

Lema 7.1. *Suponiendo que $f < n/4$, si todos los procesos correctos llegan al principio de la etapa k con el mismo valor estimado val, entonces no cambiarán este valor en lo que resta del protocolo.*

Demostración. Por la hipótesis, se sigue que la bolsa *Recibidos* de cualquier proceso correcto contiene al menos $n-f$ copias de V al final de la primera ronda de la etapa k y (puesto que $n-f > n/2$) su variable *más_frecuente* contiene este mismo valor. Si $f < n/4$ entonces $n-f > n/2+f$, de aquí se

concluye que durante la segunda ronda de la etapa k, el valor estimado *val* de un proceso correcto se actualiza a *más_frecuente*, es decir, conserva el valor *val*. □

Teorema 7.4. *El protocolo expuesto en el algoritmo 7.3 satisface las propiedades de terminación, acuerdo y validez débil aun en presencia de hasta $f < n/4$ procesos que incurran en fallas bizantinas. Para ello se requieren $2(f+1)$ rondas.*

Demostración. La validez débil es una consecuencia inmediata del lema anterior: si todos los procesos correctos comienzan con el mismo valor v, retendrán este valor hasta el momento de su decisión. La terminación es una consecuencia de la sincronía: un proceso correcto decide al final de la ronda $2(f+1)$.

Por lo que concierne a la propiedad de acuerdo, si transcurren $f+1$ etapas y se presentan f procesos bizantinos a lo más, existe al menos una etapa coordinada por un proceso correcto. Sea k la primera etapa con estas características y $k/2$ el coordinador correspondiente. Si i es un proceso correcto, al final de la etapa k, i tiene un valor v almacenado en *val*:

- Si i ya ha ejecutado la instrucción $val \leftarrow$ *más_frecuente*, se infiere que al menos $n/2+f+1$ procesos tienen a v como valor estimado al inicio de la etapa k. Entonces el coordinador de la etapa k recibe al menos $n/2+1$ copias de v. Por tanto solo ha observado un valor único como el más frecuente y además es el valor de la mayoría, consecuentemente el valor que transmite durante la segunda parte de la ronda k es *más_frecuente* $= v$. Se sigue que sin importar la instrucción de asignación efectuada por cualquier otro proceso correcto j, ya sea ($val \leftarrow$ *más_frecuente*, o bien $val \leftarrow$ *val_coordinador*), al final de etapa k se tendrá en j que $val = v$. Todos los procesos correctos dispondrán del mismo valor estimado al final de la etapa k.
- Si ningún proceso correcto i ha ejecutado la instrucción $val \leftarrow$ *más_frecuente*, entonces todos ejecutarán la instrucción $val \leftarrow$ *val_coordinador*. En consecuencia, todos los procesos correctos dispondrán del mismo valor estimado al final de la etapa k (ya que el coordinador es correcto, se garantiza que envía el mismo valor a todos los procesos).

  En ambos casos, el lema 7.1 garantiza que los procesos correctos no modificarán sus valores estimados en lo que resta de la ejecución. Esta es la propiedad de acuerdo.

□

7.6. Comentarios finales

La tabla 7.2 reúne los resultados expuestos a lo largo de este capítulo. Cada renglón incluye un tipo de consenso, las propiedades que lo definen, el modelo de falla y los requerimientos para su solución.

Problema	Especificación	Modelo	Requerimiento
Consenso	Terminación + Acuerdo + Validez	Paro	$f < n$
Consenso uniforme	Terminación + Acuerdo uniforme + Validez	Omisión	$f < n/2$
Consenso débil	Terminación + Acuerdo + Validez débil	Bizantina	$f < n/3$

Tabla 7.2: Problemas de consenso y modelos de falla

En este capítulo se han presentado protocolos de consenso para modelos de cómputo síncrono. Cada modelo de falla requiere una especificación diferente del problema.

Una observación muy importante de este capítulo es que se asume la existencia de un canal de comunicación entre cada pareja de procesos implicados en el problema. En algunas circunstancias, podría suponerse que este modelo sería equivalente a una capacidad de comunicación mediante difusión (broadcast), sin embargo, este no es el caso. Otra importante consideración que debe tomarse en cuenta es que, el tiempo es fuente de información a partir de la cual pueden inferirse algunas propiedades del sistema. En concreto, tenemos que para los participantes en un sistema síncrono es posible reconocer algunas condiciones de falla, gracias a que existen plazos finitos para la comunicación.

Ejercicios

7.1. Diseña un algoritmo síncrono de consenso bajo fallas de paro, que permita decidir el valor de un conjunto sin relación de orden.

RESPUESTA: En este caso, cada nodo usa un arreglo en el que guarda las propuestas de los otros participantes. El arreglo sería indizado mediante el identificador de cada nodo, i.e. en la i-ésima entrada se guarda la propuesta del nodo i. Al terminar las rondas podemos usar esta estructura para tomar una decisión.

7.2. Dibuja un diagrama de espacio-tiempo, donde se lleve al límite el número de rondas necesarias para alcanzar el consenso síncrono bajo fallas de paro.

SUGERENCIA: Se trata de dibujar una cadena en la que, durante las primeras f rondas, un proceso falla por cada ronda y solamente alcanza a entregar su mensaje una vez y "muere" de inmediato.

7.3. Demuestra que no es posible resolver el consenso síncrono bajo fallas bizantinas, si $f \geq n/3$.

RESPUESTA: Supongamos, por contradicción, que existe un algoritmo para alcanzar consenso en un sistema de 3 procesos, p_0, p_1 y p_2, conectados por un grafo completo (que en este caso es un triángulo). Sean A, B y C, los algoritmos locales que corren en p_0, p_1 y p_2, respectivamente. Ahora vamos a considerar un anillo síncrono con 6 procesos numerados consecutivamente $p_0, \ldots, p_5$ y tales que p_0 y p_3 tienen a A como su algoritmo local, p_1 y p_4 tienen a B como su algoritmo local y, finalmente, p_2 y p_5 tienen a C como su algoritmo local. No podemos suponer que dicho sistema resuelva el consenso puesto que la combinación de A, B y C solo trabaja correctamente sobre un triángulo de procesos pero, en todo caso, el sistema tiene un comportamiento predefinido cuando cada proceso inicia con un valor de entrada y no hay procesos en falla.

Ahora busquemos una asignación de valores iniciales tales que cualesquiera dos procesos conectados podrían "imaginar" que se encuentran en un triángulo, tal que el tercer proceso está en falla. Por nuestra suposición inicial, el consenso podría alcanzarse en esta condición. Sin embargo, debemos llegar a una situación en la que dos procesos correctos, digamos p_0 y p_2, yacen juntos en uno de estos triángulos, al mismo tiempo que cada uno participa en otro triángulo. La decisión a la que llega p_0 en ambas configuraciones en las que participa es la misma porque, desde su perspectiva, es el mismo conjunto de entradas. Lo mismo sucede con p_2. Sin embargo, en el triángulo donde aparecen juntos, esto los lleva a una situación en la que no puede alcanzarse el acuerdo (!).

Incluir fig. 5.5 del libro de Attiya & Welch.

7.4. ¿Cuál es el costo en mensajes de un algoritmo de consenso síncrono uniforme?

SUGERENCIA: El mismo número de mensajes que en el consenso síncrono, más los mensajes que se envían en la ronda $f+2$.

No preguntemos si estamos plenamente de acuerdo, sino tan solo si marchamos por el mismo camino.

—Johann Wolfgang Goethe

8

Detectores de fallas

Resumen En este capítulo se presenta una familia de mecanismos parcialmente (o débilmente) síncronos denominados detectores de fallas (DF), a partir de los cuales es posible resolver el consenso. Un detector de fallas es un "módulo" que acompaña a un proceso mientras ejecuta un algoritmo distribuido, y le sirve para estimar el estado de los otros componentes activos del sistema con quienes intercambia información. Este módulo puede emitir diagnósticos equivocados y, por tanto, considerar que un proceso sigue en funcionamiento cuando en realidad ha caído en falla, o al revés, sospechar que un componente ha caído en falla, cuando no es así (solo es muy lento para comunicarse). A pesar de las imprecisiones de su diagnóstico, la ventaja de un detector de fallas se halla en la posibilidad de evitar que un proceso se quede estancado, esperando la comunicación con otro que no envía información a tiempo. Visto de otra forma, el detector decide en un plazo de tiempo acotado sobre la condición de un proceso del que se espera información. Esta decisión puede ser errónea y posiblemente pueda revocarse. Aquí presentamos las propiedades que caracterizan a los detectores de fallas y demostramos cómo es posible resolver el consenso con ayuda de estos módulos.

8.1. Introducción

El resultado de imposibilidad del consenso asíncrono [21], establece que el modelo de comunicación asíncrona con intercambio de mensajes no ofrece las condiciones mínimas para resolver el consenso binario. La razón fundamental se debe al hecho que, bajo este modelo, es imposible distinguir entre un proceso muy lento y un proceso caído en paro. En este contexto, los detectores de fallas, propuestos por

Chandra y Toueg [11], presentan una alternativa de solución ya que introducen una familia de mecanismos que acotan el tiempo que debe esperarse para recibir respuesta de un proceso. Visto de otra forma, se trata de complementar las características del modelo de comunicación, y con ello volverlo más robusto.

Un hecho notable en la propuesta de los DF es que los autores no describen la manera como estos módulos pueden implementarse. En cambio, se definen en término de un conjunto de propiedades abstractas. Este enfoque permite el desarrollo de aplicaciones cuyo funcionamiento puede demostrarse basándose solo en estas propiedades, sin recurrir a parámetros físicos de bajo nivel, tales como el tiempo exacto con el que debe programarse un temporizador, por ejemplo. Se considera que con este enfoque modular se consigue también un desacoplamiento entre el diseño de las aplicaciones y los mecanismos que subyacen en la detección de fallas. Cada proceso que participa en la solución del consenso se equipa con este módulo al que puede consultar en algún momento crítico de su operación. La familia de los detectores se caracteriza por dos propiedades: una, que sirve para "sospechar" de algún proceso y otra, que sirve para "confiar" en algún proceso. Debemos entender la acción de sospechar como sinónimo de estimar en falla. Del mismo modo, debemos entender la acción de confiar como el hecho de asumir que un proceso no se ha desviado de sus especificaciones de operación. Diversas variantes de estas propiedades se definen por el tiempo en el que pueden elaborarse un predicado sobre el estado de un subconjunto de los procesos que participan en la operación. Aquí cabe preguntarse, ¿qué sucede si un proceso sospecha o confía equivocadamente de otro proceso?

La notación utilizada en este capítulo se presenta en la tabla 8.1.

t	tiempo
Π	Conjunto de procesos
$\mathscr{T}$	Un rango en los números naturales
$\mathbf{F}$	Un patrón de fallas
$\mathbf{F}(t)$	Conjunto de procesos en falla en el tiempo t
$Caídos(\mathbf{F})$	Conjunto de procesos caídos de acuerdo con $\mathbf{F}$
$Correctos(\mathbf{F})$	Conjunto de procesos correctos de acuerdo con $\mathbf{F}$
$\mathscr{D}$	Detector de fallas
r	Número de ronda
v	Un valor
c	Proceso coordinador
$Sospechosos$	Conjunto de procesos con sospecha de falla en paro
$Sospechosos(i,t)$	Procesos de los que sospecha i en el instante t

Tabla 8.1: Notación

8.2. El modelo y las propiedades de los detectores de fallas

En este capítulo se abordan sistemas distribuidos asíncronos. Un sistema se considera formado por un conjunto Π de n procesos, donde cada pareja está conectada por un canal de comunicación confiable. Por razones de exposición, se asume la existencia de un reloj global de tiempo discreto al que, sin embargo, los procesos no tienen acceso. Se considera entonces que dicho "aparato" entrega sus lecturas en el rango $\mathscr{T}$, que corresponde con los números naturales.

Los procesos participantes pueden caer en fallas de *paro*, es decir, detienen prematuramente su operación. Un *patrón de falla* $\mathbf{F}$ es una función de $\mathscr{T}$ a 2^{Π}, donde $\mathbf{F}(t)$ es el conjunto de procesos que han caído en falla hasta el tiempo t. Una vez caído, un proceso no puede recuperarse, esto implica que $\forall t : \mathbf{F}(t) \subseteq \mathbf{F}(t+1)$. Definimos los conjuntos $Caídos(\mathbf{F}) = \bigcup_{t \in \mathscr{T}} \mathbf{F}(t)$, y $Correctos(\mathbf{F}) = \Pi - Caídos(\mathbf{F})$. Si $i \in Caídos(\mathbf{F})$, decimos que *i cae* en $\mathbf{F}$. En tanto que, si $i \in Correctos(\mathbf{F})$, decimos que i es correcto de acuerdo con $\mathbf{F}$. Solo se consideran patrones de falla $\mathbf{F}$, en los que al menos hay un proceso correcto, esto es, $Correctos(\mathbf{F}) \neq \emptyset$. Es importante observar que existe un tiempo t', luego del cual $\Pi - \mathbf{F}(t') = Correctos(\mathbf{F})$.

Cada módulo detector de fallas genera una salida con el conjunto de procesos de los que, en ese momento, sospecha que han caído en paro. La *lista de Sospechosos* es una función de $\Pi \times \mathscr{T}$ a 2^{Π}. $Sospechosos(i,t)$ es la salida del módulo detector del proceso i, en el instante t. Si $j \in Sospechosos(i,t)$ decimos que, de acuerdo con su lista, *i sospecha* de j en el tiempo t. Obsérvese también que los módulos de dos procesos diferentes no necesariamente coinciden en la lista de procesos de los que se sospecha.

Informalmente, un detector de fallas $\mathscr{D}$ provee información (posiblemente incorrecta) acerca del patrón de fallas que ocurre en una ejecución. Formalmente, el detector de fallas $\mathscr{D}$ es una función que mapea cada patrón de fallas a un conjunto de listas de un detector $\mathscr{D}(\mathbf{F})$. Es decir, al conjunto de todas las listas de *Sospechosos* que podrían ocurrir en ejecuciones bajo el patrón $\mathbf{F}$ y el detector $\mathscr{D}$.

Un detector de fallas $\mathscr{D}$ se caracteriza por dos propiedades: *completez* (completeness) y *exactitud* (accuracy).

La completez admite dos variantes: fuerte y débil. Un detector de fallas exhibe la *completez fuerte* si garantiza que en algún momento todos los procesos correctos sospechan permanentemente de todos los procesos en falla. Esto significa que $\mathscr{D}$ satisface la siguiente condición:

$$\forall \mathbf{F}, \forall Sospechosos \in \mathscr{D}(\mathbf{F}), \exists t \in \mathscr{T}, \forall i \in Caídos(\mathbf{F}), \forall j \in Correctos(\mathbf{F}), \forall t' \geq t : i \in Sospechosos(j,t')$$

En tanto, si un detector de fallas exhibe la *completez débil*, se garantiza que en algún momento habrá un proceso correcto que sospeche de todos los procesos en falla, es decir:

$$\forall \mathbf{F}, \forall Sospechosos \in \mathscr{D}(\mathbf{F}), \exists t \in \mathscr{T}, \forall i \in Caídos(\mathbf{F}), \exists j \in Correctos(\mathbf{F}), \forall t' \geq t : i \in Sospechosos(j,t')$$

Sin embargo, por sí sola la completez no aporta todo el poder que se requiere de un detector de fallas, pues bastaría que cada proceso sospechara de todos los demás para satisfacer la completez fuerte, de forma trivial. Se trata de limitar los errores que el mismo detector puede cometer. Por tal razón se introduce la exactitud, la cual admite cuatro variantes:

- *Exactitud fuerte*, si no se sospecha de ningún proceso antes que falle, esto es,

$$\forall \mathbf{F}, \forall Sospechosos \in \mathscr{D}(\mathbf{F}), \forall t \in \mathscr{T}, \forall i, j \in \Pi - \mathbf{F}(t) : i \notin Sospechosos(j,t)$$

- *Exactitud débil*, si existe algún proceso correcto del que nunca se sospecha,

$$\forall \mathbf{F}, \forall Sospechosos \in \mathscr{D}(\mathbf{F}), \exists i \in Correctos(\mathbf{F}), \forall t \in \mathscr{T}, \forall j \in \Pi - \mathbf{F}(t) : i \notin Sospechosos(j,t)$$

	Exactitud			
Completez	Fuerte	Débil	Finalmente fuerte	Finalmente débil
Fuerte	Perfecto	Fuerte	Finalmente perfecto	Finalmente fuerte
	P	S	$\diamond P$	$\diamond S$
Débil	–	Débil	–	Finalmente débil
	Q	W	$\diamond Q$	$\diamond W$

Tabla 8.2: Las ocho clases de detectores de fallas

- *Exactitud finalmente-fuerte* (eventually), si al cabo de cierto tiempo ningún proceso correcto sospecha de ningún proceso correcto

$$\forall \mathbf{F}, \forall Sospechosos \in \mathscr{D}(\mathbf{F}), \exists t \in \mathscr{T}, \forall t' \geq t, \forall i, j \in Correctos(\mathbf{F}) : i \notin Sospechosos(j, t')$$

y, por último,

- *Exactitud finalmente-débil*, si al cabo de cierto tiempo existe un proceso correcto del que ningún proceso correcto sospecha

$$\forall \mathbf{F}, \forall Sospechosos \in \mathscr{D}(\mathbf{F}), \exists t \in \mathscr{T}, \exists i \in Correctos(\mathbf{F}), \forall t' \geq t, \forall j \in Correctos(\mathbf{F}) : i \notin Sospechosos(j, t')$$

8.3. El concepto de equivalencia entre detectores de fallas

Si combinamos las propiedades de completez y exactitud, tendremos que existen ocho tipos diferentes de detectores de falla, como se muestran en la tabla 8.2.

El trabajo original de Chandra y Toueg establece el concepto de "reducibilidad" entre detectores de fallas. Informalmente, se dice que un detector de fallas $\mathscr{D}'$ es *reducible* al detector de fallas $\mathscr{D}$, si existe un algoritmo distribuido que pueda usar $\mathscr{D}$ para emular a $\mathscr{D}'$. También se dice que $\mathscr{D}'$ es *más débil* que $\mathscr{D}$ si, dado este algoritmo de reducción, cualquier cosa que pueda hacerse con $\mathscr{D}'$ puede hacerse , con $\mathscr{D}$. Dos detectores de fallas son *equivalentes* si uno puede reducirse al otro. Basados en estos conceptos, se sabe que los DF de cada columna de 8.2 son equivalentes. Por otra parte, en un ambiente asíncrono no es posible construir ni siquiera un detector débilmente completo y débilmente exacto. Por tanto, solo cabe utilizar un detector con una exactitud finalmente fuerte o finalmente débil y que exhiba cualquier tipo de completez (pues una implica a la otra).

Consideremos el detector finalmente fuerte $\diamond S$, que ofrece completez fuerte y exactitud finalmente débil. Puesto que la completez fuerte implica a la débil, es obvio que el detector $\diamond W$ es más débil que el detector $\diamond S$. Pero es más interesante aun el hecho que el detector $\diamond S$ también es más débil que el detector $\diamond W$. Designemos como $W.Sospechosos$ y $S.Sospechosos$, el conjunto de procesos de los que se sospecha mediante los detectores $\diamond W$ y $\diamond S$, respectivamente. Estamos refiriéndonos a las listas generadas por cada uno de los detectores. El algoritmo 8.1, implementa un detector $\diamond S$ usando para ello un detector $\diamond W$. Esta implementación se basa en dos actividades, cada proceso envía su lista de sospechosos a todos los procesos, de manera regular y permanente. Al recibir este mensaje, un proceso consulta con su propio detector y agrega a su lista de sospechosos aquellos transportados por el mensaje que recién recibe, descontando al proceso que transmite el mensaje.

Algoritmo 8.1: Implementación de $\diamond S$ usando $\diamond W$, algoritmo en el nodo i

```
   /*condiciones iniciales*/
1      S.Sospechosos ← W.Sospechosos
2      envía W.Sospechosos a todos de forma permanente
3  Al recibir m.Sospechosos desde j efectúa
4      S.Sospechosos ← S.Sospechosos ∪ m.Sospechosos − {j}
```

Teorema 8.1. *El conjunto S.Sospechosos satisface la completez fuerte y la exactitud finalmente débil.*

Demostración. Supongamos un proceso i caído en falla. Por la propiedad de $\diamond W$ existe un proceso correcto j que a partir de algún momento sospecha permanentemente de i, esto es, $i \in W.Sospechosos$ de j. Puesto que j envía de manera regular y permanente esta lista a todos los demás procesos con los que interactúa, entonces se volverá parte de $S.Sospechosos$ de todos los procesos correctos. Solo si i enviara su propio mensaje, se removería su identidad de las listas de sospechosos. Pero luego de que caiga en paro su identidad aparecerá permanentemente en la lista de cada proceso correcto. Con ello se prueba que $S.Sospechosos$ satisface la completez fuerte.

Ahora demostraremos que se cumple la exactitud finalmente débil. Por la propiedad de $\diamond W$, existe un proceso correcto i, del que ningún proceso correcto sospecha. Esto implica que en algún momento no será parte de ninguna lista $W.Sospechosos$ y su identidad tampoco será incluida en la lista que transportan los mensajes intercambiado. Además, en algún momento i enviará su propio mensaje, con lo que quedaría removido de cualquier $S.Sospechosos$. □

8.4. Algoritmo de consenso

Si suponemos que el máximo número de procesos que pueden caer en paro no excede la mayoría, entonces podemos resolver el consenso usando un DF de tipo $\diamond S$. Sea f el máximo número de procesos que pueden caer en paro. Considérese un sistema asíncrono tal que $f < \lceil n/2 \rceil$. En estas condiciones, el algoritmo 8.2 resuelve el consenso usando un detector finalmente fuerte. Esto es, un detector que satisface completez fuerte y exactitud finalmente débil. Recordemos que un detector exhibe completez fuerte si garantiza que en algún momento todos los procesos correctos sospechan permanentemente de todos los procesos en falla. Asimismo, exhibe exactitud finalmente débil si al cabo de cierto tiempo existe un proceso correcto del que ningún proceso correcto sospecha.

A lo largo de los años han aparecido diferentes soluciones basadas en un paradigma denominado *coordinador rotatorio*. Aun cuando no se trata del único tipo de solución, nos parece interesante destacarlo porque es un principio que subyace en muchas propuestas. La idea básica consiste en dar por bueno el hecho que no todos los participantes pueden caer en falla (cualquiera que sea su tipo) y que, siempre que los participantes puedan asumir un rol especial, de manera cíclica, entonces puede esperarse que en algún momento este rol recaerá sobre un proceso correcto que aprovechará su condición para impulsar la solución.

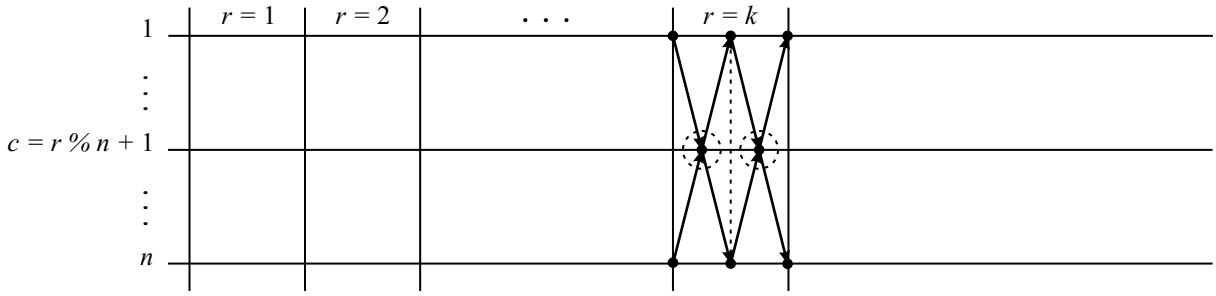


Figura 8.1: Algoritmo de consenso usando $\diamond S$

El algoritmo se desarrolla en rondas asíncronas. Cada proceso sabe que durante la ronda r el proceso $(r \mod n) + 1$ es designado como coordinador. Cada ronda comprende cuatro fases consecutivas. En la fase 1, cada proceso envía su valor propuesto al coordinador correspondiente. En la fase 2, el coordinador reúne los valores propuestos por una mayoría de participantes y a partir de los valores que recibe difunde una nueva (contra) propuesta. Durante la fase 3, cada proceso espera la contrapropuesta del coordinador. Si recibe este nuevo valor, entonces devuelve un acuse de recibo al coordinador, en otro caso se sospecha del mismo y se le informa de esta condición. Durante la fase 4, el coordinador espera recibir los acuses procedentes de una mayoría. Si esto sucede entonces decide su valor propuesto, lo comunica al resto de los procesos y se detiene. Obsérvese cómo el algoritmo asume que existe un canal de comunicación punto a punto entre cada pareja de procesos o bien, el conjunto de procesos está dotado de un sistema de comunicaciones que soporta operaciones de difusión confiable de mensajes, véase la figura 8.1.

El paradigma del coordinador rotatorio garantiza la terminación, en presencia de un DF que tiene una exactitud finalmente débil, porque esto implica que, al cabo de cierto tiempo, existe un proceso correcto, digamos i, del que ningún otro proceso correcto sospecha. Cuando le llega a i el turno de ser coordinador entonces recibirá el número suficiente de acuses de recibo para completar su decisión y comunicarla a los demás. Otro aspecto importante del protocolo es la idea de "amarrar" un valor. Durante la fase 2 el coordinador recibe de cada proceso su valor propuesto junto con una estampilla de tiempo que designa la última ronda en la que este proceso actualizó el valor que está proponiendo. El coordinador escoge el valor asociado con la mayor estampilla que reciba. Con ello se consigue que todas las propuestas subsecuentes coincidirán con el mismo.

Cada proceso que participa en el algoritmo cuenta con un conjunto de variables que dan cuenta de su estado local:

v :	El valor inicialmente propuesto por el proceso.
r :	La ronda corriente, inicialmente es 0.
t :	La estampilla que indica el tiempo durante el cual el proceso ha adoptado el valor que postula, inicialmente es 0.
c :	Identifica al proceso que funge como coordinador de la ronda corriente.
fase :	Designa la fase por la que atraviesa la ronda.
confirma :	Lleva la cuenta del número de procesos que han aceptado el valor propuesto por el coordinador.

Algoritmo 8.2: Algoritmo de Chandra-Toueg para consenso, usando $\diamond S$, en el nodo i

```
   /*condiciones iniciales*/
1      v ← bit de entrada
2      r ← −1 /*número de ronda*/
3      t ← 0 /*estampilla de tiempo*/
4  Al arrancar ronda efectúa
5      fase ← 1
6      r ← r + 1
7      c ← (r mód n) + 1
8      envía (i, r, v, t) a c
9      si (soy c) entonces fase ← fase + 1
10     otro fase ← fase + 2
11 Al recibir (j, r, v, t) efectúa
12     si (fase = 2) entonces
13         espera las primeras n/2 + 1 propuestas
14         elige la propuesta val con la mayor estampilla
15         difunde (c, r, val)
16         confirma ← 1
17         rechaza ← 0
18         fase ← fase + 2
19 Al recibir (c, r, val) o sospechar de c efectúa
20     si (fase = 3) entonces
21         si (m = (c, r, val)) entonces
22             v ← val, t ← r
23             envía CONFIRMA(r) a c
24             envía RECHAZA(r) a c
25 Al recibir CONFIRMA(r) o RECHAZA(r) efectúa
26     si (fase = 4) entonces
27         si (m = CONFIRMA(r)) entonces confirma ← confirma + 1
28         otro rechaza ← rechaza + 1
29     si (confirma ≥ n/2 + 1) entonces difunde DECIDE( val )
30 Al recibir DECIDE( val ) efectúa
31     difunde DECIDE( val )
32     decide el valor val
33     termina
```

rechaza : Lleva la cuenta del número de procesos que NO han aceptado el valor propuesto por el coordinador.

Asimismo, durante la ejecución del algoritmo se intercambian los siguientes mensajes:

(j, r, v, t): El proceso j, durante la ronda r, propone el valor v que adoptó durante la ronda t.

(c, r, val): El coordinador c, de la ronda r, propone el valor val.

$CONFIRMA(r)$: El proceso que envía el mensaje acepta el valor propuesto por el coordinador de la ronda r.

`RECHAZA`(r): El proceso que envía el mensaje descarta el valor propuesto por el coordinador de la ronda r, porque sospecha de él.
`DECIDE`(val): El coordinador ha decidido el valor val.

Lema 8.1. *(Acuerdo) No hay dos procesos correctos que difieran en sus valores decididos.*

Demostración. Sea r el valor de la primera ronda durante la que el coordinador recibe un número mayoritario de mensajes de tipo `CONFIRMA`. Sea val el valor propuesto por el coordinador al final de dicha ronda. Afirmamos que, para toda ronda $r' \geq r$, el coordinador de turno no propone un valor diferente de val. Para demostrarlo usaremos la técnica de inducción sobre $r' - r$. Claramente, la afirmación es trivial para el caso en que $r' - r = 0$. Supongamos ahora que es cierta también para cualquier ronda tal que $r' - r < k$. Ahora, consideremos el caso cuando el coordinador de la ronda $r + k$ envía su propuesta. El valor que debe elegir para difundir será aquel con la mayor estampilla de tiempo, de entre las propuestas que recibe. Se sigue que esta estampilla es, al menos, igual a r, puesto que el coordinador recibe una mayoría de mensajes estampillados y al menos uno de ellos procede de un proceso que actualizó su propuesta durante la ronda r. Supongamos que el valor escogido por el coordinador de la ronda r' procede del proceso j. Ya habíamos aceptado que la estampilla del valor propuesto por j es, al menos, r. También se sigue que dicha estampilla es igual a $r + k - 1$, a lo más. Se tiene entonces que el coordinador de turno debe someter como propuesta, el valor definido durante r. Por lo tanto, el coordinador de la ronda r' propondrá el mismo valor. □

Lema 8.2. *(Terminación) El algoritmo termina en un tiempo finito.*

Demostración. Todo proceso correcto finalmente decide un valor. Sabemos que ninguno de los pasos del algoritmo es bloqueante cuando los procesos que fallan no representan a la mayoría. Por la propiedad de exactitud finalmente débil sabemos que el algoritmo alcanzará una ronda durante la cual, el coordinador (un proceso sin fallas), recibirá un número suficiente de mensajes de tipo `CONFIRMA`. Luego, será capaz de difundir su decisión, que será recibida por todos los procesos correctos. □

Lema 8.3. *El algoritmo cumple el requisito de validez*

Demostración. Es fácil observar que el valor que se decide es propuesto por algún proceso participante. □

8.5. Comentarios finales

Se sabe que $\diamond W$ es el más sencillo de los detectores de falla. Sus propiedades establecen los requisitos mínimos que deben satisfacerse para resolver el consenso. Un detector de este tipo puede cometer un número infinito de errores. Sin embargo, se garantiza que en "algún" momento ciertas condiciones serán satisfechas permanentemente. En la práctica, cuando se trata de resolver un problema que tiene terminación, como el caso del consenso, es suficiente con que estas propiedades se cumplan por un tiempo "suficientemente" largo, como para garantizar el progreso. Por otro lado, en sistemas asíncronos

no se puede cuantificar un tiempo "suficientemente" largo, entonces es preferible definir las propiedades de $\diamond W$, en los términos expuestos en el capítulo.

Supongamos una aplicación en la que se asume un detector de fallas con las propiedades de $\diamond W$, pero el módulo que lo implementa funciona mal y, por ejemplo, hay una caída que no se detecta, mientras que repetidamente se sospecha de los procesos correctos. En estas circunstancias, un protocolo de consenso podría impedir que los procesos alcancen una decisión, pero jamás decidir valores diferentes o inválidos.

Muchas implementaciones de los detectores de fallas se basan en mecanismos temporizados. Por ejemplo, un proceso j puede enviar un mensaje de tipo `ESTOY-VIVO`, hacia el resto de los procesos para indicar que sigue en operación. Un proceso i puede utilizar un temporizador por cada uno de los otros procesos con los que interactúa. Si expira el temporizador que corresponde a j y no recibe noticias de éste, entonces lo agrega en su lista de sospechosos. Si más tarde recibe el mensaje `ESTOY-VIVO` desde j, entonces i reconoce que cometió un error, elimina a j de su lista e incrementa el plazo de su temporizador para prevenir errores futuros.

En un sistema asíncrono, el esquema anterior no implementa un detector con las propiedades de $\diamond W$, porque violaría la propiedad de exactitud correspondiente. Sin embargo, en muchos casos prácticos, al incrementar el plazo de un temporizador, luego de corregir el error, puede garantizar que en algún momento no habrá expiraciones prematuras en, al menos, un proceso correcto. Con ello se conseguirá la propiedad de exactitud de $\diamond W$.

Insistimos en recordar que, en la práctica, es suficiente que está garantía se cumpla por un tiempo suficientemente largo como para garantizar el progreso del algoritmo.

Los detectores de fallas sirven no solamente para resolver el consenso asíncrono, también pueden usarse para resolver otros problemas como la difusión atómica, por ejemplo.

Ejercicios

8.1. ¿Para qué sirve la completez fuerte en el algoritmo 8.2?

RESPUESTA: Esta garantiza que en algún momento, todos los procesos correctos enviarán de manera permanente un mensaje `RECHAZA` a un proceso en falla, cuando este funja como coordinador.

8.2. ¿De qué manera se relacionan los protocolos de consenso y de difusión confiable?

SUGERENCIA: Un protocolo de difusión fiable reúne 3 propiedades: validez, acuerdo e integridad. La primera se refiere a que si un proceso correcto difunde un mensaje `M`, entonces todos los procesos correctos en algún momento entregan este mensaje. La segunda, se refiere a que si un proceso correcto entrega un mensaje `M`, entonces todos los procesos correctos en algún momento también lo hacen. La tercera se refiere a que, para cualquier mensaje `M`, todos los procesos correctos entregan `M` a lo más una vez, y solo si algún proceso difunde a `M`.

8.3. ¿Por qué la exactitud finalmente débil implica a la exactitud finalmente fuerte?

RESPUESTA: Porque si hay un proceso correcto del que nadie sospecha, este puede servir de guía o ejemplo para todos los procesos correctos.

8.4. ¿Podrían haber dos coordinadores simultáneos en un sistema que ejecuta el algoritmo para consenso que utiliza $\diamond S$?

SUGERENCIA: Supongamos que el coordinador de la ronda 1 tarda mucho en completar las fases que le corresponden (pero lo hace) y entonces le llega el turno al coordinador de la ronda 2.

8.5. ¿Cómo se garantiza que el valor que propone un coordinador correcto en la fase 2, será el valor que se decida?

RESPUESTA: i) porque el valor queda amarrado o comprometido con todos los participantes correctos y ii) porque el coordinador usa un algoritmo de difusión fiable durante la fase 4.

La experiencia siempre ha demostrado que jamás suceden bien las cosas cuando dependen de muchos.

—Nicolás Maquiavelo

9

Imposibilidad del consenso asíncrono

Resumen En este capítulo se describen los aspectos más importantes del famoso resultado de imposibilidad del consenso binario asíncrono, también conocido como el teorema de Fischer, Lynch y Paterson (FLP). La idea fundamental de la argumentación consiste en suponer la existencia de un algoritmo capaz de resolver el problema. Al mismo tiempo, se establece que existe un estado inicial desde el cual el sistema puede optar por cualquiera de sus dos posibles valores de salida. Es decir un estado de indecisión. Luego, se demuestra que la ejecución del algoritmo puede progresar y avanzar indefinidamente manteniéndose, sin embargo, en estados de indecisión.

9.1. Introducción

El resultado de Fischer, Lynch y Paterson muestra una limitación fundamental de los sistemas distribuidos asíncronos. En sí, el problema de consenso binario es muy básico, se trata de un conjunto de procesos que deben acordar el valor de un bit. Muchos problemas como la elección, la exclusión mutua y el cálculo de una función global guardan una relación con el consenso porque la solución de cualquiera de ellos puede llevarnos a la solución del consenso. El resultado de imposibilidad implica que ninguno de estos problemas tiene solución bajo las mismas suposiciones de operación. Este capítulo está basado en las ideas de [21] y [25].

Otro aspecto relevante del resultado yace en el hecho que asume formas "suaves" o moderadas de las fallas. Para empezar, considera que solo los procesos, y no los canales, pueden experimentar fallas de

paro. Es decir, el tipo de falla más sencillo. Lo trascendente del resultado consiste en que, si se cumple para modelos de falla tan débiles, entonces también es verdadero para modelos más fuertes.

La notación utilizada en este capítulo se presenta en la tabla 9.1.

n	Número de procesos
x	Un bit
$\mathscr{V}$	Conjunto de valores
v	Un valor
p	identidad de un proceso
(p,v)	Un mensaje con destino p y valor v
Σ	Un estado global
e,f	Eventos
$e(\Sigma)$	Evento e aplicado en Σ
$\alpha(\Sigma)$	Secuencia de eventos α aplicados en Σ
$\Sigma.Val$	Conjunto de valores de decisión alcanzables desde Σ
$\mathscr{G},\mathscr{H}$	Conjuntos de estados globlaes
$\Theta,\Gamma_0,\Gamma_1,$ $\Lambda_0,\Lambda_1,\Omega_0,\Omega_1$	Estados globales

Tabla 9.1: Notación

9.2. Modelo del sistema e hipótesis

Para empezar, formalizaremos la versión del problema. Se tienen n procesos ($n > 2$). Cada proceso p comienza con un valor inicial del conjunto $\{0,1\}$, guardado como un bit en su registro local de entrada x. De manera semejante, cada proceso cuenta con un registro local de salida y, que indica el valor decidido o comprometido. Se dice que un proceso toma una decisión, o alcanza un *estado de decisión*, cuando escribe sobre su registro de salida. Inicialmente, cada registro de salida contiene un valor "nulo", indicando que el proceso aún no ha alcanzado un estado de decisión. En otro caso, el proceso escribe sobre y un valor que puede ser 0 ó 1. Se dice que el registro de salida solo admite una operación de escritura. Esto implica que, una vez que se decide y escribe sobre y, este valor no puede revocarse. Por último, asumimos que cada proceso tiene una capacidad de almacenamiento ilimitada.

Los procesos se comunican intercambiando mensajes entre sí. Un mensaje es una pareja de la forma (p,v), donde p es la identidad del proceso de destino y v es un valor tomado de un conjunto fijo $\mathscr{V}$. El sistema de mensajes mantiene un multiconjunto denominado *buffer de mensajes*, que contiene aquellos mensajes que han sido transmitidos, pero aún no se reciben. El buffer de mensajes soporta dos operaciones abstractas:

- `envía`(p,v). Que inserta (p,v) en el buffer.
- `recibe`(p). Que borra un mensaje (p,v) del buffer y regresa v, en cuyo caso decimos que se entrega (p,v), o bien regresa un marcador especial $\emptyset$ y deja el buffer sin cambios.

Aun cuando el buffer de mensajes actúa de forma no determinista se garantiza que si la operación `recibe`(p) se ejecuta una cantidad infinita de veces, entonces cada mensaje (p,v) guardado en el buffer se entrega en algún momento. En particular, el sistema de mensajes puede devolver el valor especial $\emptyset$

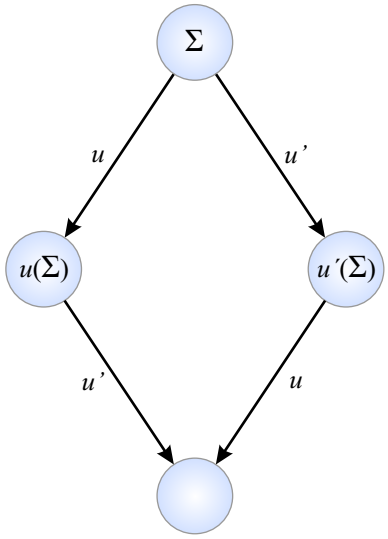


Figura 9.1: Conmutatividad de eventos disjuntos

un número finito de veces, como respuesta a la invocación `recibe`(p), aun cuando el mensaje (p,v) se encuentre en el buffer.

Las variables internas de cada proceso definen su estado interno. El estado global de un sistema se puede entender entonces como el conjunto de los estados internos de los procesos participantes. En el estado global inicial cada proceso tiene un estado interno inicial y el buffer de mensajes se considera vacío.

Un estado global tránsita hasta otro, mediante *pasos*. Cada paso ocurre en dos fases. Sea Σ un estado global. Primero se invoca la operación `recibe`(p) sobre el buffer de mensajes para obtener un mensaje v (que puede ser el valor $\emptyset$ inclusive) luego, dependiendo del proceso p al que va dirigido v, p transita hasta un nuevo estado interno y, envía un conjunto finito de mensajes hacia otros procesos. Como se considera que los procesos son deterministas, cada paso queda determinado por la pareja $e = (p,v)$, que en lo sucesivo llamaremos *evento*. $e(\Sigma)$ denota el nuevo estado global que se alcanza. Asimismo, decimos que e "puede aplicarse o despacharse" en Σ.

Una *agenda* de Σ es una secuencia α, finita o infinita, de eventos que puede aplicarse partiendo de Σ. La secuencia asociada se denomina *corrida*. Si α es finita, denotamos por $\alpha(\Sigma)$ el estado global resultante, el cual se dice es *alcanzable* desde Σ. Un estado global es *accesible* si es alcanzable desde un estado global inicial. En lo sucesivo se consideran que los estados globales son accesibles.

Por otro lado, se deben asentar las suposiciones acerca del ambiente de operación. Básicamente, estas hipótesis de trabajo están contenidas en los siguientes 3 postulados:

- *Independencia inicial*. Cada proceso elige su valor de entrada inicial de forma independiente.
- *Conmutatividad de eventos disjuntos*. Sea Σ un estado global del sistema en el que se encuentran habilitados los eventos e y f. Es decir, pueden despacharse. Si e y f van dirigidos, o deben ocurrir en dos procesos diferentes, entonces pueden conmutar. Esto es, pueden despacharse en orden arbitrario y aun así, luego de atender a ambos, el estado global resultante será el mismo. Este postulado aplica

también para agendas disjuntas, esto es, secuencias de eventos, u y u', aplicables sobre conjuntos de procesos que son ajenos, véase la figura 9.1.

- *Asincronía de eventos*. La atención de un evento de recepción puede retrasarse arbitrariamente. Visto de otra forma, el evento $(p, \emptyset)$ puede aplicarse, dejando al proceso de destino en posibilidad de atender otro paso posterior.

Para los efectos de nuestra argumentación se consideran solamente corridas infinitas. Decimos también que un proceso con falla es aquel que solo toma o aparece durante una cantidad finita de pasos en la corrida. Una corrida será *admisible* si, a lo más, ocurre en ella un proceso con falla. Puesto que se considera que el soporte de las comunicaciones es confiable, entonces se da por hecho que todos los mensajes enviados por los procesos sin falla, en algún momento se entregan a su destino. Por último, decimos que una corrida es de *decisión*, si algún proceso participante alcanza un estado de decisión.

Los requerimientos del protocolo pueden reunirse en las siguientes tres propiedades:

1. *Acuerdo*: Dos procesos no pueden decidir valores diferentes.
2. *No trivialidad*: Tanto el 0, como el 1, son salidas posibles. Esto es, no se admiten protocolos que devuelvan un valor fijo e independiente de las entradas iniciales.
3. *Terminación*: El sistema alcanza el acuerdo al cabo de un tiempo finito.

9.3. Argumentación

La idea básica consiste en demostrar que existe una corrida admisible que puede permanecer indefinidamente sin alcanzar una decisión. En específico, se establece que 1) existe un estado global inicial en el que el sistema puede decidir cualquiera de sus dos valores válidos y, 2) se puede mantener al sistema en estados de "indecisión", de forma permanente.

Para formalizar la noción de "indecisión", se define la *valencia* de un estado global. Sea $\Sigma.Val$ el conjunto de valores de decisión que pueden alcanzarse desde el estado global Σ. Decimos entonces que Σ es bivalente si $|\Sigma.Val| = 2$. En tanto que Σ es monovalente si $|\Sigma.Val| = 1$. En este último caso decimos que Σ es 0-valente si $\Sigma.Val = \{0\}$, y 1-valente si $\Sigma.Val = \{1\}$. Puede inferirse que un estado bivalente, captura una condición del sistema en la que cualquiera de los dos valores posibles podría elegirse por un protocolo que intentara resolver el consenso. En tanto, un estado monovalente es aquel en el que, al protocolo pueden faltarle pasos para finalizar, pero el resultado está ya determinado.

La siguiente duda que podríamos formularnos es, si de verdad existen estados bivalentes, ¿podría ser que todo estado inicial tenga asociado un solo resultado fatal o inevitable?, visto de otro modo, que todo estado inicial implica una única salida.

El siguiente lema establece que, para todo protocolo existe un estado global inicial de indecisión o bivalente.

Lema 9.1. *Todo protocolo de consenso tiene un estado inicial bivalente.*

Demostración. Supongamos, por contradicción, que no existen estados iniciales bivalentes. Entonces, por el requisito de no trivialidad, cualquier estado global inicial es 0-valente o 1-valente. Decimos que dos estados globales son adyacentes si difieren solamente en la variable local x de un proceso. Podemos

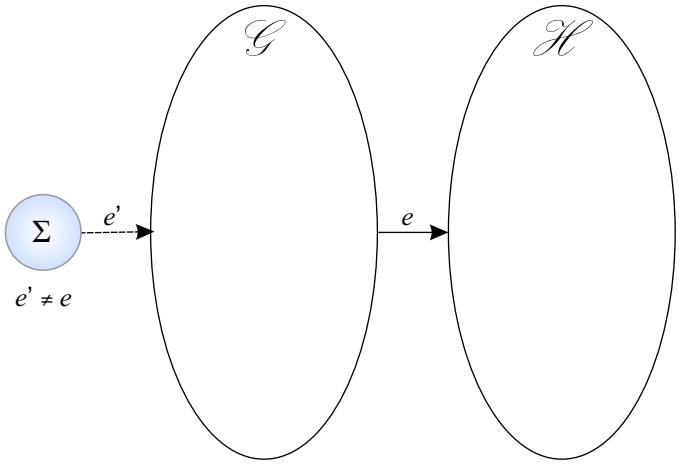


Figura 9.2: Definición de $\mathscr{G}$ y $\mathscr{H}$

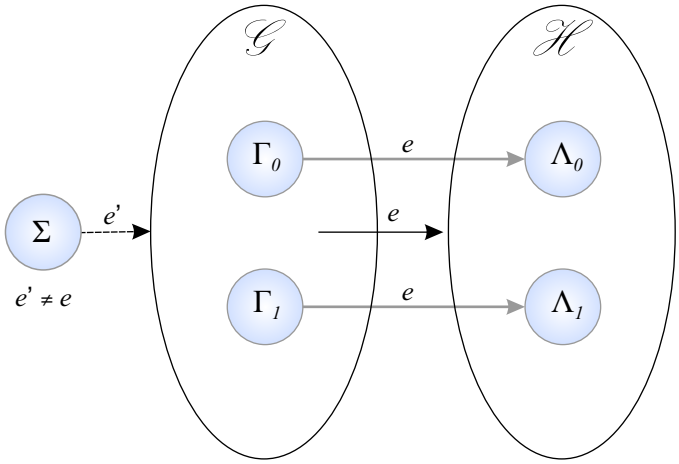


Figura 9.3: Γ_0 y Γ_1 están en $\mathscr{G}$

pensar en dos vectores de n bits idénticos salvo por una entrada. Imaginemos ahora que construimos una numeración con todos los vectores de n bits, ordenados de forma que dos vectores consecutivos representen estados globales adyacentes. Comenzamos por el estado $0,\ldots,0$, que evidentemente es 0-valente, y terminamos por el estado $1,\ldots,1$, que es 1-valente. En algún punto de esta numeración existen dos estados adyacentes tales que, el primero es 0-valente y el segundo es 1-valente. Supongamos ahora que solo difieren en la entrada correspondiente al proceso p. Ahora, vamos a aplicar sobre ambos estados, una corrida admisible de decisión en la que p no ejecuta un solo paso. Puesto que ambos estados difieren solo en el valor inicial de p, el sistema debe alcanzar la misma decisión en ambos casos (!). □

El anterior resultado merece un momento de reflexión para comprender sus alcances. Hay que observar que no se habla de la forma como habrá de decidirse si un estado global es 0-valente o 1-valente. Naturalmente, se puede ver que un vector conteniendo solo 0's es un estado inicial 0-valente, mientras que un vector conteniendo solo 1's es un estado 1-valente. Fuera de estos dos casos extremos, la forma como se evalúa la valencia de un estado puede ser, digamos, por mayoría simple, esto es el valor que

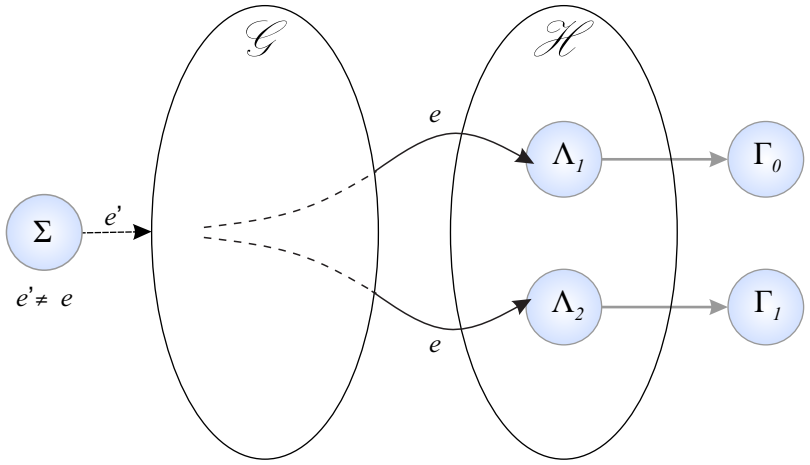


Figura 9.4: Γ_0 y Γ_1 no están en $\mathscr{G}$

aparezca el mayor número de veces determina la valencia del estado. Pero igualmente puede optarse por una función que evalúe el valor mínimo, o el máximo, etc. Entonces, lo que este resultado nos dice es que, independientemente de la forma como se determine su valencia, siempre habrá un estado global inicial bivalente.

El siguiente paso de nuestro planteamiento consiste en demostrar que el sistema puede mantenerse en un estado bivalente.

Lema 9.2. *Sea Σ un estado global bivalente de un protocolo de consenso. Sea e un evento sobre el proceso p aplicable en Σ, esto es que puede procesarse cuando el sistema se encuentra en Σ. Sea $\mathscr{G}$ el conjunto de estados globales alcanzables desde Σ sin aplicar e. Finalmente, sea $\mathscr{H} = e(\mathscr{G})$, el conjunto de estados alcanzables desde $\mathscr{G}$ luego de aplicar e, véase la figura 9.2. Entonces, $\mathscr{H}$ contiene un estado global bivalente.*

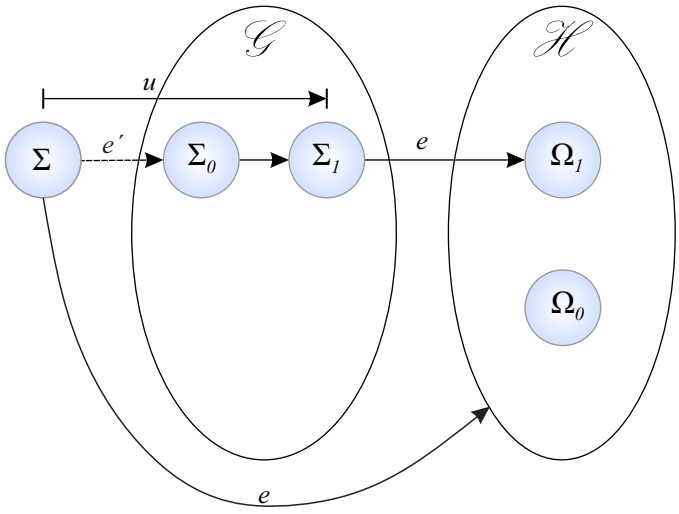


Figura 9.5: Construcción de dos estados vecinos

Demostración. Asumiremos por contradicción, que $\mathscr{H}$ no contiene estados globales bivalentes, entonces

Afirmación 9.3.1 *$\mathscr{H}$ contiene tanto estados 0-valentes, como estados 1-valentes.*

Demostración. Como Σ es un estado bivalente, esto significa que existen Γ_0 (0-valente) y Γ_1 (1-valente), alcanzables desde Σ. Supongamos que Γ_0 y Γ_1 están en $\mathscr{G}$, entonces al aplicar e sobre estos, llegaremos a dos estados monovalentes distintos Λ_0 y Λ_1 que están en $\mathscr{H}$, véase la figura 9.3. En caso contrario, suponiendo que Γ_0 y Γ_1 no están en $\mathscr{G}$, eso significa que deben poder alcanzarse desde $\mathscr{H}$ pero, por hipótesis, $\mathscr{H}$ no contiene estados bivalentes, entonces debe contener dos estados Λ_0 y Λ_1 que lleven a Γ_0 y Γ_1, respectivamente, véase la figura 9.4. En cualquiera de los casos, hemos aceptado que $\mathscr{H}$ contiene estados monovalentes de los dos tipos posibles, es decir, Λ_0 y Λ_1. □

Decimos que dos estados globales son vecinos si uno es alcanzable desde el otro, por la aplicación de un solo evento o paso.

Afirmación 9.3.2 *Existen estados vecinos Σ_0 y Σ_1 en $\mathscr{G}$, tales que $\Omega_0 = e(\Sigma_0)$ es 0-valente y $\Omega_1 = e(\Sigma_1)$ es 1-valente.*

Demostración. Sea u la más pequeña secuencia de eventos aplicados sobre Σ, sin aplicar e, tal que $eu(\Sigma)$ es diferente de $e(\Sigma)$. Tal secuencia existe por la afirmación previa. Supongamos sin pérdida de generalidad, que $e(\Sigma)$ es 0-valente. Por la misma afirmación previa, existe un estado global en $\mathscr{H}$ que es 1-valente. Sea u la secuencia más corta que conduce a un estado 1-valente en $\mathscr{G}$, desde el que podemos llegar al estado 1-valente en $\mathscr{H}$ (al aplicar e). Entonces, los últimos dos estados de la secuencia son los estados vecinos que buscamos, véase la figura 9.5. □

Supongamos ahora que $\Sigma_1 = f(\Sigma_0)$, donde f es un evento aplicable sobre el proceso q. Ahora vamos a desarrollar un análisis de casos.

Caso 1: p y q son dos procesos diferentes. Esta situación implica que e y f cumplen la hipótesis de conmutatividad, pero esto lleva a una contradicción porque f puede aplicarse luego de alcanzar Ω_0, que es 0-valente, y con ello llegar a Ω_1, que es 1-valente (!), véase la figura 9.6.

Caso 2: p y q son el mismo proceso. Supongamos que, partiendo de Σ_0, existe una secuencia finita de pasos s que lleva a un estado de decisión y en la que p no participa. Sea $\Theta = s(\Sigma_0)$ el estado al que se llega al aplicar s sobre Σ_0. Por el postulado de conmutatividad, s puede aplicarse sobre Ω_0 o Ω_1, lo que llevaría a los estados $s(\Omega_0)$ y $s(\Omega_1)$, respectivamente. Ahora, aplicamos e sobre Θ y deberíamos llegar también a $s(\Omega_0)$, mientras que si aplicamos fe sobre Θ deberíamos llegar a $s(\Omega_1)$. Lo cual implica que Θ es bivalente (!), véase la figura 9.7. □

La enseñanza del resultado anterior es que cualquier protocolo que lleve a un sistema desde un estado bivalente hasta un estado monovalente debe tener un paso crítico en el que se toma la decisión. Este paso no puede basarse en el orden de los eventos ejecutados en procesos diferentes porque dichos eventos son conmutables. Lo anterior implicaría que el paso crítico debe ser tomado por un solo proceso, pero esto tampoco funciona porque los demás no pueden distinguir si éste ha caído en falla, o simplemente es muy lento.

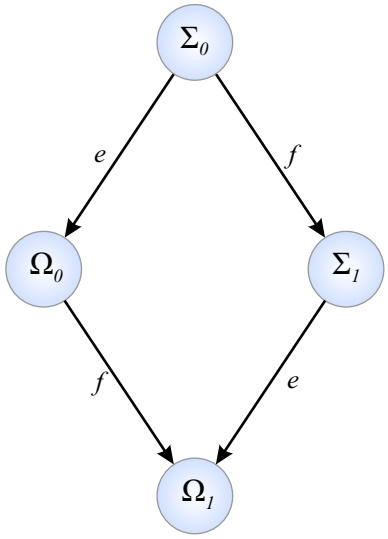


Figura 9.6: caso 1, p y q son dos procesos diferentes

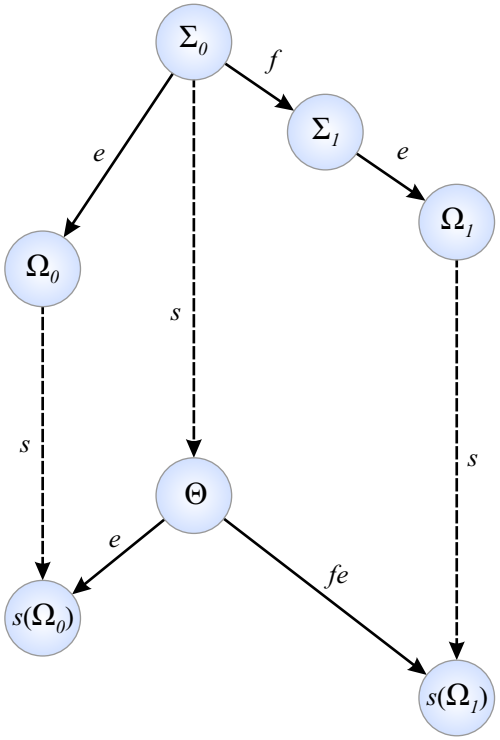


Figura 9.7: caso 2, p y q son el mismo proceso

Teorema 9.1. *No existe un protocolo que satisfaga las propiedades de acuerdo, no trivialidad y terminación, si al menos uno de los procesos que participan puede fallar.*

Demostración. Vamos a construir una corrida admisible que no alcanza una decisión. La corrida comienza en un estado global inicial bivalente Σ_s, que sabemos que existe por el lema 9.1, y consiste en una secuencia de pasos. Aseguramos que cada paso comienza en un estado global bivalente Σ.

Para ello mantenemos una lista (circular) de procesos. En cada etapa, el proceso que se encuentra en la cabeza de la lista recibe un mensaje del buffer o el valor $\emptyset$. Sea e el evento correspondiente a la acción de recibir el mensaje v en el proceso p. Por el lema 9.2, sabemos que existe un estado global bivalente Ω alcanzable desde Σ por una secuencia en la que e es el último en ser atendido. Por la hipótesis de asincronía, entregamos el valor $\emptyset$ a p, luego lo movemos al final de la lista y repetimos este conjunto de pasos por siempre. Esta construcción garantiza que cada proceso se ejecuta un número infinito de veces y que cada mensaje enviado es recibido en algún momento (eventually). Entonces, el protocolo permanece eternamente transitando por estados globales bivalentes. □

9.4. Comentarios finales

Es muy importante cerrar este capítulo con una reflexión acerca de lo que significa la afirmación: "es imposible que exista un protocolo determinista que resuelva el consenso binario asíncrono, si al menos un proceso puede caer en paro". Podríamos, muy inocentemente, programar un autómata determinista que fuera ejecutado en cada proceso participante. Luego, cada uno envía su valor de entrada a todos los demás. Cuando todos hayan recibido este mensaje, se decide en consecuencia. Es muy probable que no ocurran fallas o contingencias y el procedimiento concluyera con éxito. Sin embargo, ¿podríamos afirmar que nuestra propuesta sirve en cualquier escenario? El resultado FLP nos dice que este procedimiento aplicable en todo escenario no existe. Es imposible que exista.

El resultado revisado en este capítulo significa que al sistema subyacente deben agregársele mecanismos de sincronización, a fin de resolver el consenso por métodos deterministas [16, 18], o bien, hacer suposiciones de operación como las que se sirven de base para los llamados detectores de falla no confiables [10, 11].

Debe entenderse que el teorema FLP revela no tanto la dificultad inherente del problema, sino la debilidad del modelo asíncrono. Por suerte, algunas de las suposiciones de funcionamiento que subyacen en el análisis pueden restringirse o eliminarse. Es muy interesante observar cómo un cambio mínimo en las condiciones del problema producen una variante para la que existe solución: una relajación o un requerimiento más laxo o débil en alguna de las condiciones del acuerdo, de su terminación o una restricción sencilla en los tiempos de operación.

En el lado aplicado del problema, no hay que olvidar que existen enfoques alternativos, como los algoritmos no deterministas o los detectores de fallas, con los que puede darse la vuelta a los resultados de imposibilidad y definir algoritmos eficaces.

Se reconoce al consenso como uno de los problemas fundamentales de la computación distribuida, que sigue siendo tema de investigación. Además de su importancia intrínseca, el consenso está rodeado de un elegante conjunto de resultados. Aun se desarrollan nuevos métodos de demostración para conclusiones ya conocidas, muchos de los cuales sirven como herramientas para abordar otros problemas de la misma naturaleza.

Finalmente, debe mencionarse la importancia del problema por cuanto sirve como "estándar" de prueba (benchmark) con el que se evalúan y comparan las capacidades de los diferentes modelos que caracterizan a los sistemas distribuidos.

Ejercicios

9.1. Escribe una secuencia de 8 palabras de 3 bits, de modo que entre dos palabras consecutivas solamente haya un bit de diferencia, ¿en qué parte de la argumentación se utiliza una secuencia como esta?

RESPUESTA: 000, 001, 011, 010, 110, 100, 101, 111, se utiliza en el lema 9.1

9.2. ¿Por qué se requiere en la argumentación que cada proceso tenga una capacidad de almacenamiento ilimitada?

SUGERENCIA: Para "simular" el funcionamiento asíncrono del buffer de recepción.

9.3. ¿Qué diferencia hay entre dos estados vecinos y dos estados adyacentes?

RESPUESTA: Los primeros se refieren a las condiciones de entrada del problema. En tanto, los segundos tienen que ver con la atención de un evento y el tránsito de uno al otro.

9.4. ¿Aplica el resultado FLP para algoritmos no deterministas?

SUGERENCIA: ¿Podríamos resolver el consenso "tirando" volados?

9.5. ¿Qué es un estado accesible y una corrida admisible?

RESPUESTA: Un estado global G es accesible si, partiendo de un estado global inicial G_0, existe una agenda de eventos que puede llevar hasta G. Una corrida es admisible si en ella ocurre a lo más un proceso que cae en paro.

Apéndices

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 [7] 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, el grafo 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 [29, 36].

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 medio de un grafo 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 un grafo de comunicaciones. Al mismo tiempo, es posible ejecutar un algoritmo sobre grafos 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

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 ocurrirá 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 agenda). 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 100 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.

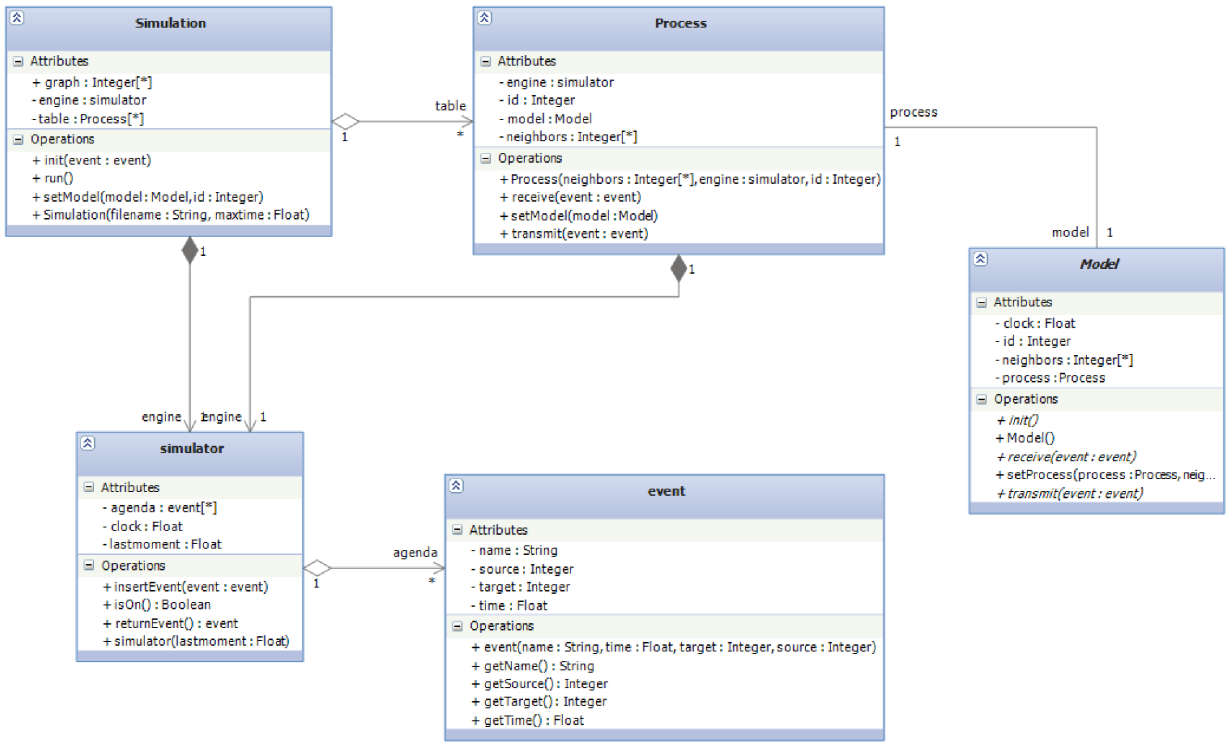


Figura A.1: Diagrama UML de clases del simulador

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: un grafo 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 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).

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.

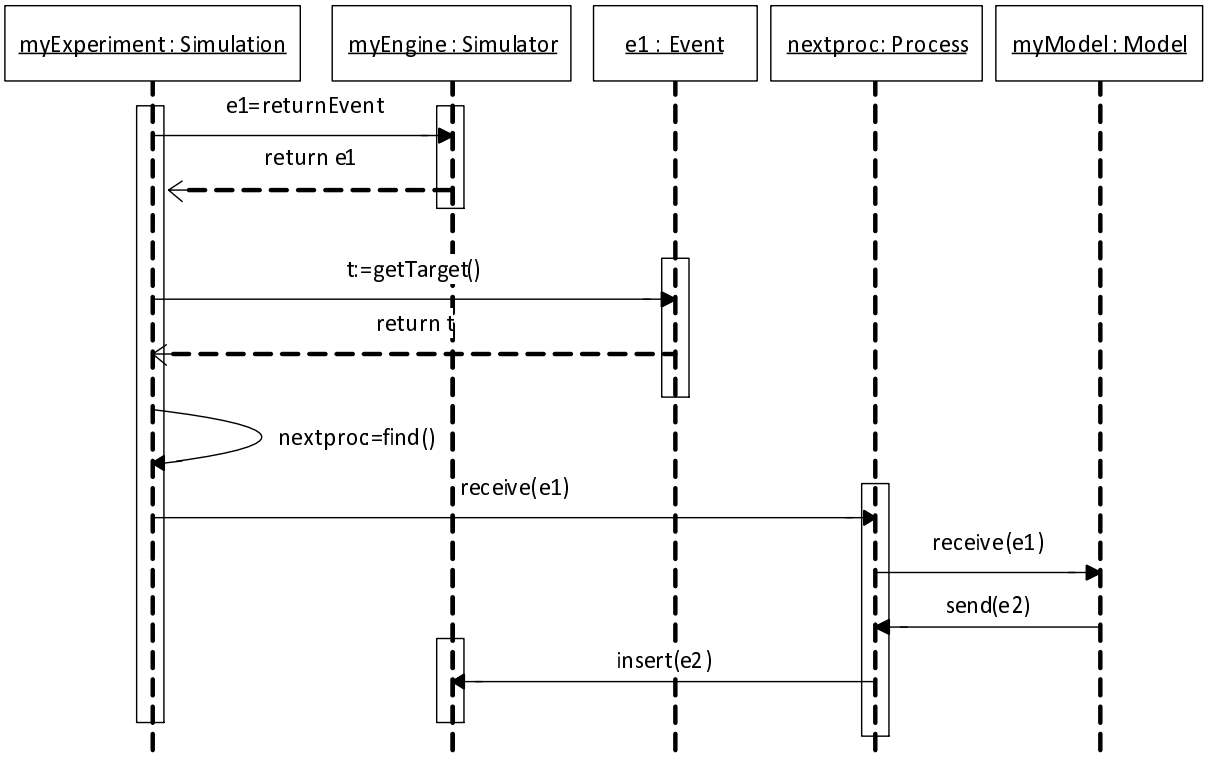


Figura A.2: Diagrama de secuencias

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 cuyo grafo 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 conexo, esto es, que exista un camino entre cualquier par 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: i) cada proceso envíe un mensaje a cada uno de sus vecinos, y ii) estos lo reciben y rexpiden por todas sus líneas de comunicación. Un proceso da por terminada su ejecución, en cuanto termina el envío de mensajes a sus vecinos. (véase el algoritmo A.1)

Algoritmo A.1: Algoritmo PI

```
1 al recibir M desde j efectúa
2     si (visitado = falso) entonces
3         padre ← j
4         visitado ← verdadero
5         envía M a todo k en vecinos −{padre}
6         termina
```

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: Es un valor booleano, inicialmente *falso*. Cambia a *verdadero* si se recibió un mensaje `M`.
padre: Indica el nodo desde el que i recibe el mensaje `M` por primera vez.
vecinos: Representa 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 (véase 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 grafo de comunicaciones a partir del archivo con nombre `sys.argv[1]` y que estará programada para funcionar durante 500 unidades de tiempo simulado (véase figura A.3, líneas `18-21`).

- 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 (véase figura A.3, líneas `47-50`).
- 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, líneas `53-56`).

La siguiente sección del archivo "pi.py", contiene el código del algoritmo que simulará cada nodo (véase figura A.3, líneas `13-31`). 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 la clase base. 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`. Los métodos que hereda son `init()`, `receive()`, y `transmit()`. Los dos primeros son métodos virtuales, lo que quiere decir que están declarados en la clase base, pero son las extensiones quienes se encargan de implementarlos.

La nueva clase que definimos introduce un atributo privado denominado `visited` (de tipo booleano), 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` (véase figura A.3, líneas `18-21`).

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 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`). 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 cambia el valor de la variable `visited` a `True`, luego procede a reexpedir el mensaje m hacia cada uno de sus vecinos. Para ello se utiliza la lista neighbors, que sabemos 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 `transmit()`. La próxima vez que se reciba un paquete de información en el mismo nodo, el algoritmo lo ignorará.

El método `transmit()` recibe como parámetro una instancia de la clase "Event", que contiene el nombre del evento, el tiempo en el que debe ocurrir, el id del nodo de destino y el id del nodo de origen.

El siguiente paso es preparar el archivo de texto que codifique al grafo sobre el que queremos experimentar nuestro algoritmo.

```
1  # ------------------------------------------------------------------------
2  # CODIGO DEL ALGORITMO DISTRIBUIDO
3  # ------------------------------------------------------------------------
4   """ Implementa la simulación del algoritmo de Propagación de Información
5   (Segall) como ejemplo de aplicación """
6
7
8   import sys
9   from event import Event
10  from model import Model
11  from simulation import Simulation
12
13  class Algorithm1(Model):
14      """ La clase Algorithm desciende de la clase Model e implementa los
15      métodos "init()" y "receive()", que en la clase base se definen como
16      abstractos """
17
18      def init(self):
19          """ Aqui se definen e inicializan los atributos del algoritmo """
20          self.visited = False
21          print "inicializo algoritmo"
22
23      def receive(self, event):
24          """ Aquí se definen las acciones concretas que deben ejecutarse
25          cuando se recibe un evento """
26          if self.visited == False:
27              print "recibo mensaje"
28              self.visited = True
29              for t in self.neighbors:
30                  newevent = Event("C", self.clock+1.0, t, self.id)
31                  self.transmit(newevent)
32
33 # ------------------------------------------------------------------------
34 # FUNCION DE INICIO
35 # ------------------------------------------------------------------------
36 # construye una instancia de la clase Simulation recibiendo como parámetros
37 # el nombre del archivo que codifica la lista de adyacencias del grafo y el
38 # tiempo max. de simulación
39
40
41  if len(sys.argv) != 2:
42      print "Please supply a file name"
43      raise SystemExit(1)
44  experiment = Simulation(sys.argv[1], 500)
45
46
47 # asocia un pareja proceso/modelo con cada nodo del grafo
48  for i in range(1,len(experiment.graph)+1):
49      m = Algorithm1()
50      experiment.setModel(m, i)
51
52
53 # inserta un evento semilla en la agenda y arranca
54  seed = Event("C", 0.0, 1, 1)
55  experiment.init(seed)
56  experiment.run()
```

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

Codificando un grafo

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.

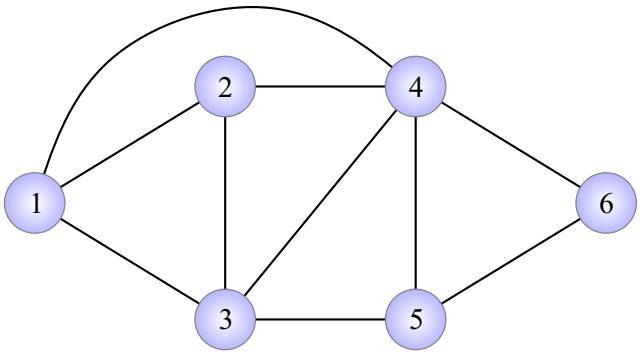


Figura A.4: Red de comunicación

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
```

Esto disparará 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.

A.6. Código del simulador en Python

```
# Este archivo contiene la implementacion de la clase Event (11.11.10)
""" Un objeto de la clase Event encapsula la informacion que se intercambia
entre las entidades activas de un sistema distribuido """

# ------------------------------------------------------------------------------------------
class Event:                    # Descendiente de la clase "object" (default)
    """ Atributos: "name", "time", "target" y "source",
    contiene tambien un constructor y los metodos que devuelven cada
    uno de los atributos individuales """

    def __init__(self, name, time, target, source):
        """ Construye una instancia con los atributos inicializados """
        self.name   = name
        self.time   = time
        self.target = target
        self.source = source

    def getName(self):
        """ Devuelve el nombre del evento """
        return (self.name)

    def getTime(self):
        """ Devuelve el tiempo en el que debe ocurrir el evento """
        return (self.time)

    def getTarget(self):
        """ Devuelve la identidad del proceso al que va dirigido """
        return (self.target)

    def getSource(self):
        """ Devuelve la identidad del proceso que origina el evento """
        return (self.source)
```

Figura A.6: Código de la clase "Event" del simulador de eventos discretos

```
# Este archivo contiene la implementacion de la clase Model (11.11.10)
""" No se pueden generar instancias de esta clase. Se usa para comprometer al
programador a proporcionar, en una clase descendiente, la implementacion de
los metodos o propiedades etiquetados como abstractos. Este es el mecanismo
para programar una aplicacion, i.e. un algoritmo concreto """

# ------------------------------------------------------------------------------
from abc import ABCMeta, abstractmethod


class Model:
    """ Atributos: "clock", "process", "neighbors", "id",
    contiene tambien un constructor y los metodos "setProcess()",
    "transmit()", "init()" y "receive()", estos dos ultimos son
    metodos abstractos que deben implementarse en la aplicacion """

    __metaclass__ = ABCMeta
    def __init__(self):
        """ define el valor inicial de su reloj """
        self.clock = 0.0

    def setTime(self, time):
        """ actualiza el valor del reloj local """
        self.clock = time

    def setProcess(self, process, neighbors, id):
        """ asocia al modelo con su entidad activa (proceso), su lista
        de vecinos y su identificador """
        self.process = process
        self.neighbors = neighbors
        self.id = id

    def transmit(self, event):
        """ invoca el metodo de transmision de su entidad activa (proceso) """
        self.process.transmit(event)

    @abstractmethod
    def init(self):
        """ Que se inicializa? eso se define en la aplicacion """
        pass

    @abstractmethod
    def receive(self, event):
        """ Que se hace con un evento recibido? eso se define la aplicacion """
        pass
```

Figura A.7: Código de la clase “Model” del simulador de eventos discretos

```
# Este archivo contiene la implementacion de la clase Process (11.11.10)
""" Cada objeto de la clase Process representa la entidad activa que reside en
un nodo de la grafica de comunicaciones """

# ------------------------------------------------------------------------------------
class Process:                    # Descendiente de la clase "object" (default)
    """ Atributos: "neighbors", "engine", "id", "model",
    contiene tambien un constructor y los metodos "setModel()",
    "transmit()" y "receive()"  """

    def __init__(self, neighbors, engine, id):
        """ asocia al proceso con su lista de vecinos, su motor de
        simulacion y su identificador """
        self.neighbors = neighbors
        self.engine = engine
        self.id = id

    def setModel(self, model):
        """ asocia al proceso con el modelo que debe ejecutar y viceversa """
        self.model = model
        self.model.setProcess(self, self.neighbors, self.id)
        self.model.init()

    def setTime(self, time):
        """ actualiza el valor del reloj local """
        self.model.setTime(time)

    def transmit(self, event):
        """ invoca al motor para insertar un evento en su agenda """
        self.engine.insertEvent(event)
#        print "transmite mensaje"

    def receive(self, event):
        """ consulta a su modelo para decidir la atencion de un evento """
        self.model.receive(event)
```

Figura A.8: Código de la clase “Process” del simulador de eventos discretos

```
# Este archivo contiene la implementacion de la clase Simulation (11.11.10)
""" Un objeto de la clase Simulation representa un experimento en el que
se ejecuta un algoritmo distribuido sobre una grafica de comunicaciones """

from process import Process
from simulator import Simulator
# ------------------------------------------------------------------------------------
class Simulation:                   # Descendiente de la clase "object" (default)
    """ Atributos: "engine", "graph", "table", contiene tambien un
    constructor y los metodos "setModel()", "init()", "run()" """

    def __init__(self, filename, maxtime):
        """ construye su motor de simulacion, la grafica de comunicaciones y
        la tabla de procesos """
        self.engine = Simulator(maxtime)

        f = open(filename)
        lines = f.readlines()
        f.close()
        self.graph = []
        for line in lines:
            fields = line.split()
            neighbors = []
            for f in fields:
                neighbors.append(int(f))
            self.graph.append(neighbors)

        self.table  = [[]]          # la entrada 0 se deja vacia
        for i,row in enumerate(self.graph):
            newprocess = Process(row, self.engine, i+1)
            self.table.append(newprocess)

    def setModel(self, model, id):
        """ asocia al proceso con el modelo que debe ejecutar y viceversa """
        process = self.table[id]
        process.setModel(model)

    def init(self, event):
        """ inserta un evento semilla en la agenda """
        self.engine.insertEvent(event)

    def run(self):
        """ arranca el motor de simulacion """
        while self.engine.isOn():
            nextevent = self.engine.returnEvent()
            target = nextevent.getTarget()
            nextprocess = self.table[target]
            nextprocess.setTime(nextevent.getTime())
            nextprocess.receive(nextevent)
```

Figura A.9: Código de la clase “Simulation” del simulador de eventos discretos

```
# Este archivo contiene la implementacion de la clase Simulator (11.11.10)
""" Un objeto de la clase Simulator representa el motor de simulacion que
coordina la comunicacion entre los procesos del sistema """

# ----------------------------------------------------------------------------------------
class Simulator:                    # Descendiente de la clase "object" (default)
    """ Atributos: "clock", "agenda", contiene tambien un constructor
    y los metodos "insertEvent()", "returnEvent()", "isOn()" """

    def __init__(self, lastmoment):
        """ define el valor inicial del reloj de simulacion y fija los
        valores extremos de la agenda """
        self.clock = 0.0
        self.agenda = [[-1.0],[lastmoment+0.1]]

    def insertEvent(self, event):
        """ inserta un evento en una lista ordenada por tiempo (agenda),
        con valores extremos fijos (asi evita casos especiales) """
        key=event.getTime()
        newitem = [key, event]
        for i,item in enumerate(self.agenda):
            if key < item[0]:
                self.agenda.insert(i,newitem)
                break

    def returnEvent(self):
        """ devuelve el segundo evento de la agenda (el primero esta fijo) """
        item = self.agenda.pop(1)
        return item[1]

    def isOn(self):
        """ Verdadero si aun hay eventos que procesar """
        return len(self.agenda)>2
```

Figura A.10: Código de la clase “Simulator” del simulador de eventos discretos

B

Fundamentos de la teoría de grafos

En este capítulo se describe algunos de los conceptos básicos de la teoría de grafos.

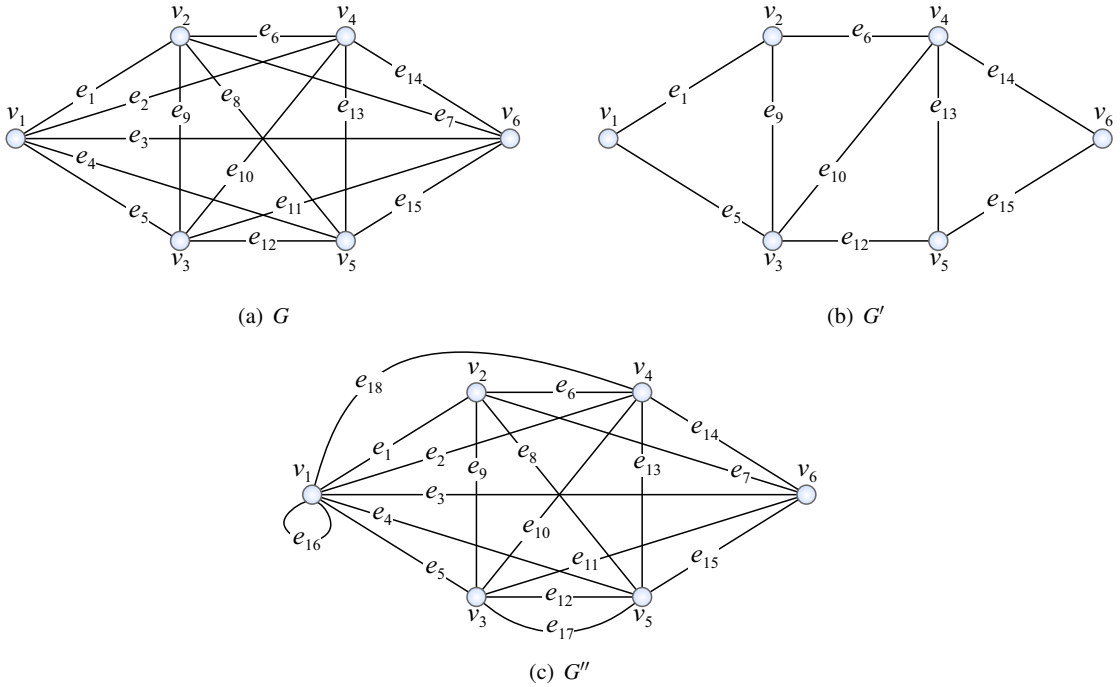
B.1. Vértices y aristas

Definición B.1. Un grafo G se define como $G = (V, E)$ donde V es un conjunto de *vértices* o *nodos* y E es un conjunto de *aristas* o *enlaces*. Cada elemento $e \in E$ une a dos vértices $u, v \in V$. Si $e = (u, v)$ entonces u y v son los *extremos* de e. Si $u, v \in V$ son los extremos de e entonces u y v son *adyacentes*.

$V(G)$ y $E(G)$ denotan el conjunto de vértices y aristas asociados con el grafo G respectivamente. Una arista se representa por una tupla no ordenada de dos vértices (u, v).

La definición B.1 considera un grafo *no dirigido*, es decir, el par (u, v) al igual que (v, u) indican que los vértices u y v son adyacentes en G pero no proporcionan más información. En algunas ocasiones es conveniente asociar una dirección a las aristas de un grafo si se desea establecer que la relación entre los vértices u y v es asimétrica. Por ejemplo, al representar un sistema de almacenamiento distribuido por medio de un grafo, en el cual los vértices son dispositivos de almacenamiento y las aristas representan la posibilidad de migrar datos de un dispositivo a otro. Podría ser que solo se pueda migrar datos del vértice u a v pero no así de v a u. En este caso, es necesario establecer una dirección en cada arista que une a dos vértices, lo cual en este ejemplo denota la migración de datos de un dispositivo a otro.

Definición B.2. Un grafo $G = (V, A)$ es *dirigido* o $digrafo$ si cada arista $a \in A$ une a un par ordenado de vértices $u, v \in V$. En consecuencia, la arista (u, v) es diferente de (v, u).



(a) G (b) G'

(c) G''

Figura B.1: Ejemplos de diagramas de grafos

Definición B.3. El orden de un grafo G es el número de vértices ($|V(G)|$) que lo componen, y el tamaño de G es el número de aristas o arcos ($|E(G)|$) con que cuenta.

Una forma de visualización de los grafos es a través de su representación gráfica en la cual cada vértice se indica por medio de un punto o círculo y cada arista por medio de una línea (véase la figura B.1).

Ejemplo B.1. En la figura B.1(b) $G' = (V', E')$ donde:

$$V'(G') = \{\ v_1, v_2, v_3, v_4, v_5, v_6\ \}$$
$$E'(G') = \{\ e_1 = (v_1, v_2),\ e_5 = (v_1, v_3),\ e_9 = (v_2, v_3),\ e_6 = (v_2, v_4),\ e_{10} = (v_3, v_4),\ e_{12} = (v_3, v_5),$$
$$e_{13} = (v_4, v_5),\ e_{14} = (v_4, v_6),\ e_{15} = (v_5, v_6)\ \}$$

Una arista de la forma $e = (u, v)$ con $u, v \in V : u \neq v$ se denomina arista *simple*. Si más de una arista $e \in E$ une al mismo par de vértices $u, v \in V : u \neq v$ se denominan *aristas múltiples*. Una arista de la forma $e = (u, u)$ se denomina *lazo* o *bucle*. Se dice que G es *simple* si no contiene aristas múltiples ni lazos. Si $V = \emptyset$ entonces G es *vacío* o *trivial*. Si V es un conjunto finito entonces G es *finito*. Un grafo G se dice *completo* si es simple, finito y cada par de vértices distintos $u, v \in V$ son adyacentes.

Ejemplo B.2. En la figura B.1(a), el grafo G es simple y completo, en B.1(b) G' es simple pero no es completo. En B.1(c) G'' no es simple y no es completo. La clasificación de las aristas para G'' es la siguiente:

Tipo	
lazo	e_{16}
múltiples	$e_2, e_{18}, e_{12}, e_{17}$
simples	$e_1, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}, e_{11}e_{13}, e_{14}, e_{15}, e_{18}$

Definición B.4. Para cualquier grafo G y vértice $u \in V$, el conjunto de vecinos $N(u)$ del vértice u se define por:

$$N(u) \stackrel{\text{def}}{=} \{v \in V(G) | v \neq u, \exists e \in E(G) : e = (u,v)\}$$

B.2. Grado de un vértice

Una propiedad importante de un vértice es el número de aristas que inciden sobre este.

Definición B.5. Sea una grafo $G = (V,E)$. El grado de un vértice $v \in V$, denotado como $\delta(v)$, es el número de aristas que son incidentes en v. Los lazos se cuentan dos veces.

Teorema B.1. *Para cualquier grafo G, la suma de los grados de sus vértices es igual a dos veces el número de aristas:*

$$\sum_{v \in V(G)} \delta(v) = 2|E(G)|$$

Corolario B.1. *Para cualquier grafo G, el número de vértices de grado impar es par.*

Definición B.6. Considere un grafo G, se dice que G es *regular* si el grado de todos los vértices en G es el mismo. Si $\delta(v) = k$ para todo $v \in V(G)$ entonces G es $k-$*regular*.

Ejemplo B.3. En la figura B.1(a) el grafo G es 5-regular debido a que para todo vértice $v_i \in V(G)$ $\delta(v_i) = 5$. Por otro lado, los grafos de la figura B.1(b) y (c) (G', G'') no son regulares. Los grados de los vértices para el grafo G'' son los siguientes:

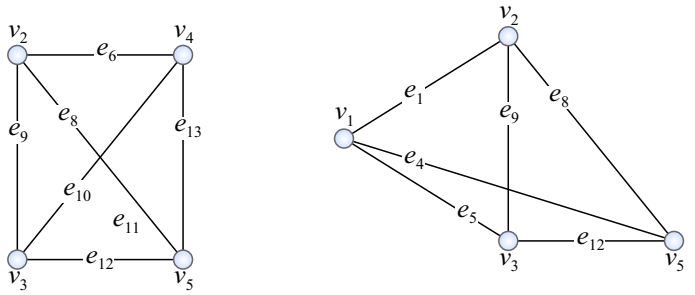
$$\delta(v_1) = 8; \delta(v_2) = 5; \delta(v_3) = 6; \delta(v_4) = 6; \delta(v_5) = 6; \delta(v_6) = 5$$

Si realizamos la suma de los grados de los vértices en G'' se tiene que $\sum_{v_i} \delta(v_i) = 8+5+6+6+6+5 = 36$. Este resultado es $2 \cdot |E(G'')|$.

B.3. Subgrafos

Definición B.7. Un grafo G' es un *subgrafo* de G si $V(G') \subseteq V(G)$ y $E(G') \subseteq E(G')$ tal que para cada arista $e \in E(G')$ los extremos de e estan en $V(G')$.

Cuando se analiza las propiedades de un grafo, es conveniente considerar los subgrafos que se forman por un conjunto específico de vértices o aristas. Estos grafos se denominan grafos inducidos.



(a) Subgrafo inducido por $V' = \{v_2, v_3, v_4, v_5\}$

(b) Subgrafo inducido por $E' = \{e_1, e_4, e_5, e_9, e_8, e_9, e_{12}\}$

Figura B.2: Subgrafos inducidos del grafo G de la figura B.1(a)

Definición B.8. Considere un grafo $G = (V, E)$ y un conjunto de vértices V' tal que $V' \neq \emptyset$. El subgrafo de G con conjunto de vértices V' y conjunto de aristas E' definido por el conjunto de aristas en G tal que sus extremos están en V' se denomina el *subgrafo inducido* por V', $G[V']$. El conjunto de aristas E' esta dado por:

$$E' \stackrel{\text{def}}{=} \{e \in E(G) | e = (u, v) \text{ con } u, v \in V'\}$$

De igual forma, considere un conjunto de aristas E' tal que $E' \neq \emptyset$. El subgrafo de G cuyo conjunto de vértices V' es el conjunto de los extremos de las aristas en E' y conjunto de aristas es E' se denomina el *subgrafo inducido* por E', $G[E']$. El conjunto de vértices V' se define como:

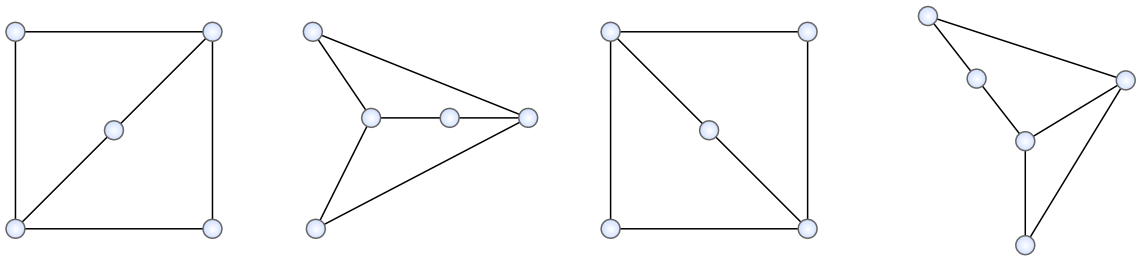
$$V' \stackrel{\text{def}}{=} \{u, v \in V(G) | \exists e \in E' : e = (u, v)\}$$

Ejemplo B.4. En la figura B.2(a) se muestra el subgrafo inducido por $V' = \{v_2, v_3, v_4, v_5\}$. En la figura B.2(b) se ilustra el subgrafo inducido por $E' = \{e_1, e_4, e_5, e_9, e_8, e_9, e_{12}\}$.

B.4. Isomorfismo

Al visualizar los diagramas de dos grafos es posible determinar si estos son isomorfos o no –por ejemplo al reetiquetar los vértices de uno se obtiene el otro (véase la figura B.3). Sin embargo, cuando los diagramas son muy diferentes o el número de vértices y/o aristas es muy grande realizar esta tarea es difícil. Para distinguir si dos grafos son isomorfos la siguiente definición es de gran utilidad.

Definición B.9. Considere dos grafos $G = (V, E)$ y $G^* = (V^*, E^*)$. G y G^* son isomorfos si existe una biyección $\phi : V \to V^*$ tal que si $(u, v) \in E$ entonces $(\phi(u), \phi(v)) \in E^*$.



(a) Dos grafos isomorfos (b) Dos grafos no isomorfos

Figura B.3: Grafos isomorfos y no isomorfos

B.5. Estructuras matriciales

Los grafos pueden representarse de varias formas. Entre las representaciones más comunes se encuentran la matriz de adyacencias y la matriz de incidencias.

Considere un grafo G con un conjunto de vértices $V = \{v_1, v_2, \ldots, v_n\}$ y conjunto de aristas $E = \{e_1, e_2, \ldots e_m\}$. La *matriz de adyacencias* (**A**) de G es una matriz de tamaño $n \times n$ en la cual la entrada $\mathbf{A}[i, j]$ denota el número de aristas que une a los vértices v_i y v_j. Por otro lado, la *matriz de incidencias* (**M**) de G es una matriz de tamaño $n \times m$ en la cual la entrada $\mathbf{M}[i, j]$ denota el número de veces que la arista e_j es incidente al vértice v_i.

Ejemplo B.5. En la figura B.1(b) la matriz de adyacencias y la matriz de incidencias de G' son las siguientes:

$$\mathbf{A} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix} \end{array} \qquad \mathbf{M} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix} \end{array}$$

B.6. Conexidad

Una propiedad importante de los grafos es que tan conectados puedan estar. Algunos grafos están tan ligeramente conectados que el solo hecho de remover un vértice o arista los desconecta. Estos vértices y aristas son en consecuencia muy importantes. Por ejemplo, considere los grafos G y G' de la figura B.1(a) y (b) respectivamente. Suponga que cada vértice representa un hospital y las aristas representan avenidas entre estos. Se notará que G' tiene menos vías de comunicación que G – en cierta forma se podría decir que G' es menos conexo que G. Así también, es claro que en G es posible llegar desde un

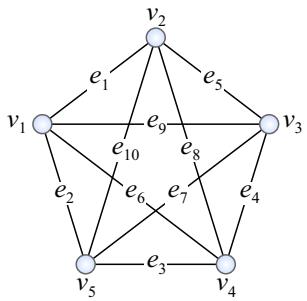


Figura B.4: H

hospital a cualquier otro sin tener que visitar otros hospitales en el camino, en tanto que en G' no es así. Además si las avenidas representadas por las aristas e_{14} y e_{15} se remueven de G' – por ejemplo, si ocurrió algún accidente – el hospital representado por el vértice v_6 queda incomunicado.

Definición B.10. Considere un grafo $G = (V,E)$. Un *camino* $(v_0 - v_k)$ en G es una secuencia de vértices y aristas de la forma $v_0, e_1, v_1, e_2 \ldots, e_k, v_k$ tal que los extremos de e_i son (v_{i-1}, v_i). Un *camino cerrado* en G es un camino con $v_0 = v_k$. Un *camino simple* es un camino en el cual todas las aristas son distintas. Una *trayectoria* es un camino simple en el cual todos los vértices son distintos. Un *ciclo* es un camino cerrado en el cual todos los vértices excepto v_0 y v_k son distintos.

Ejemplo B.6. A continuación se ilustran los conceptos de camino, camino simple y trayectoria. Considere el grafo H de la figura B.4:

$$\begin{aligned} \text{Camino } (v_1 - v_5) &: \{v_1, e_1, v_2, e_5, v_3, e_9, v_1, e_2, v_5\} \\ \text{Trayectoria } (v_1 - v_5) &: \{v_1, e_1, v_2, e_2, v_3, e_3, v_4, e_4, v_5\} \\ \text{Camino simple } (v_1 - v_4) &: \{v_1, e_9, v_3, e_7, v_5, e_{10}, v_2, e_8, v_4\} \end{aligned}$$

Definición B.11. Sea $G = (V,E)$ y considere dos vértices $u, v \in V(G)$. Se dice que u y v están *conectados* si existe una trayectoria $(u - v)$.

Definición B.12. Sea $G = V,E$ un grafo dirigido o no dirigido con $u, v \in V(G)$. La *distancia* entre u y v, $d(u,v)$, es la longitud de la trayectoria más corta $(u - v)$.

Definición B.13. Considere un grafo $G = (V,E)$. Se dice que G es *conexo* si cualquier par de vértices distintos en $V(G)$ están conectados.

Definición B.14. Sea un grafo $G = (V,E)$. Considere una partición de V en conjuntos no vacíos $V_1, V_2, \ldots V_k$ tal que los vértices u y v están conectados si y solo si ambos pertenecen al mismo conjunto V_i. Los subgrafos $G[V_1], G[V_2], \ldots, G[V_k]$ se denominan las *componentes* de G.

Si G esta compuesta por un solo componente entonces G es conexa. Algunos ejemplos de grafos conexos y no conexos se ilustran en la figura B.5.

Definición B.15. Sea $G = (V,E)$ un grafo conexo, el diámetro de G se define como:

$$D(G) \stackrel{\text{def}}{=} max\{d(u,v) | u, v \in V(G)\}$$

(a) (b)

Figura B.5: Ejemplo de grafo conexo (a) y grafo no conexo (b).

B.7. Árboles

Los árboles son una clase especial de grafos. El estudio de estos grafos es importante debido a su amplio uso en diversos campos de la ciencia. Por ejemplo: en los sistemas de cómputo se utilizan en la organización de los datos, en las redes de comunicación para definir la trayectoria más corta entre dos dispositivos de comunicación, entre otras aplicaciones más.

Definición B.16. Sea un grafo $T = (V, E)$. Si T es un grafo simple, conexo y no contiene ciclos entonces se dice que T es un *árbol*.

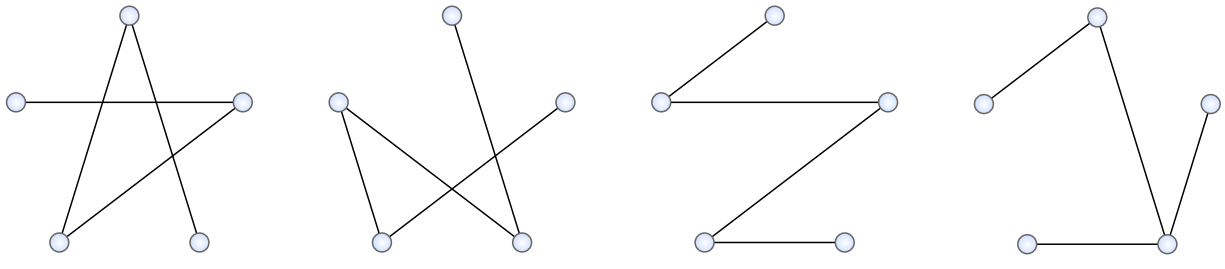
Definición B.17. Considere un grafo $T = (V, E)$. Si T es simple y todos sus componentes son árboles entonces se dice que T es un *bosque*.

En la figura B.6 se ilustra un ejemplo en el cual un grafo esta constituido por varios árboles los cuales en conjunto conforman un bosque.

Teorema B.2. *Para cualquier árbol T, cualquiera dos vértices están conectados por una trayectoria única.*

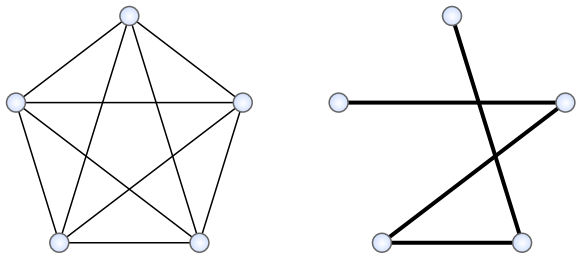
Teorema B.3. *Si T es un árbol con n vértices y m aristas entonces $m = n - 1$*

A continuación se listan algunas de las características y propiedades más importantes de los árboles.



(a) (b) (c) (d)

Figura B.6: Cuatro árboles: (a), (b), (c), y (d), los cuales en conjunto forman un bosque.



(a) (b)

Figura B.7: En (a) se muestra el grafo G y en (b) un árbol generador de G

- Sea $T = (V, E)$ un árbol:

  a. T se dice con raíz, si se designa a un solo vértice $v \in V(T)$ como *raíz*.
  b. Sea $v \in V(T)$ si $\delta(v) = 1$ entonces se dice que v es una *hoja*
  c. Todo vértice que no es raíz u hoja se dice *interior*.
  d. Todo vértice en $V(T)$ excepto la raíz tiene un *padre*. Si el vértice v es el padre del vértice u entonces se dice que u es *descendiente* o *hijo* de v.
  e. Si se agrega una arista más en T entonces se crea un ciclo.

Definición B.18. Un *árbol generador* de un grafo conexo $G = (V, E)$ es un subgrafo T con las siguientes propiedades: T es un árbol y para todo vértice v en V, v está en $V(T)$.

En la figura B.7 se muestra un ejemplo de un grafo G y uno de los varios subgrafos generadores en G.

Para la demostración de los teoremas sugerimos al lector referirse al libro de Bondy&Murty [9].

C

Notación de complejidad

En un contexto distribuido, el tiempo de ejecución o complejidad en tiempo generalmente se determina como el tiempo máximo que ha transcurrido desde el inicio del algoritmo hasta su fin. Así también, se considera otro análisis que permite determinar la eficiencia de un algoritmo, esto es, la complejidad en comunicaciones. Esta última generalmente se determina como el número total de mensajes enviados durante la ejecución del algoritmo distribuido.

C.1. Medidas asintóticas

Una de las principales formas para determinar la eficiencia de un algoritmo distribuido es a través del análisis del tiempo de ejecución o complejidad en comunicaciones. Estas medidas generalmente se obtienen al considerar el *peor caso*. Otra forma es a través del análisis del *caso promedio*, la cual es útil cuando el peor caso no se presenta tan a menudo, sin embargo, esta aproximación es difícil debido a los cálculos necesitados en el análisis probabilístico.

Determinar siempre la eficiencia de un algoritmo no es tarea fácil, es por ello que generalmente solo se realiza una estimación de esta. Una forma conveniente de estimación se denomina *análisis asintótico*. Por ejemplo, considere que la complejidad en tiempo de un algoritmo está dada por $f(n) = 5n^3 + 15n^2 + 25n$, se puede observar que f asintóticamente es a lo más n^3. La *notación asintótica* o *notación "O mayúscula"* permite describir esta relación de la siguiente forma: $f(n) = O(n^3)$. Formalmente,

Definición C.1. Sea f y g dos funciones tales que $f,g : \mathscr{N} \to \mathscr{R}^+$. Se dice que $f(n) = O(g(n))$ si y solo si $\exists\, c \in \mathscr{R}^+$ y $n_0 \in \mathscr{N}\, | f(n) \leq cg(n), \forall n > n_0$. Si $f(n) = O(g(n))$ entonces $g(n)$ es *asintóticamente un límite superior* para $f(n)$.

Definición C.2. El conjunto $O(f(n))$ define un *orden de complejidad*

Proposición C.1. *Propiedades de* $O(f(n))$.

1. $P(n,m) = a_m n^m + \cdots + a_1 n + a_0 \in O(n^m)$.
2. *Si* $f(n) \in O(g(n))$ *y* $g(n) \in O(h(n))$, *entonces* $f(n) \in O(h(n))$
3. *Si* $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = 0 \Rightarrow O(f(n)) \subset O(g(n))$
4. $O(1) \subset O(logn) \subset O(nlogn) \subset O(n^2) \subset \cdots \subset O(n^a) \subset O(2^n) \subset O(n!)$

Por otro lado, para indicar que una función es asintóticamente menor que otra se utiliza la *notación "o minuscula"*. Formalmente,

Definición C.3. Sea f y g dos funciones tal que $f,g : \mathscr{N} \to \mathscr{R}^+$. Se dice que $f(n) = o(g(n))$ si y solo si $\exists\, c \in \mathscr{R}^+$ y $n_0 \in \mathscr{N}\, | f(n) < cg(n), \forall n > n_0$.

Ejemplo C.1. Considere un algoritmo A con tiempo de ejecución $f(n) = \frac{n^2}{2} - \frac{n}{2}$. Determinar $O(f(n))$.

Por la definición C.1 se debe cumplir que: $f(n) \leq g(n) \Rightarrow \frac{n^2}{2} - \frac{n}{2} \leq g(n)$. Sabemos que $f(n)$ asintóticamente es a lo más n^2. Supongamos que $g(n) = n^2$ entonces debemos encontrar c y n_0 tal que $f(n) \leq cg(n)$ $\forall n > n_0$. Si $c = 1$ y $n_0 > 0$ la condición se satisface, en consecuencia el algoritmo A es de orden $O(f(n)) = O(n^2)$.

Las definiciones planteadas en este apartado se basan en el libro de Michael Sipser [49]. Para un análisis más detallado de la eficiencia de un algoritmo sugerimos al lector su consulta. Con respecto a las medidas en complejidad en un ambiente distribuido sugerimos consultar las siguientes referencias [36, 42].

Bibliografía

[1] Aguilera, M., Toueg, S.: A simple bivalency proof that t-resilient consensus requires t+1 rounds. Information Processing Letters (71), 155–178 (1999)

[2] Attiya, H.: Distributed algorithms. Tech. Rep. LN236357, Technion, Haifa, Israel (1994)

[3] Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in message passing systems. In: 9th ACM Simposium on Principles of Distributed Computing, pp. 363–382. ACM, New York (1990)

[4] Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations and Advanced Topics. Prentice Hall (1998)

[5] Awerbuch, B.: Optimal distributed algorithms for for minimum weight spanning tree, leader election and related problems. In: 19th ACM Simposium on the Theory of Computing, pp. 230–240. ACM (1987)

[6] Babaoglu, O., Marzullo, K.: Consistent global states of distributed systems: Fundamental concepts and mechanisms. In: S. Mullender (ed.) Distributed Systems, 2nd edn., chap. 4, pp. 55–96. ACM (1993)

[7] Beazley, M.D.: Python Essential Reference. Addison-Wesley (2009)

[8] Berman, P., Garay, J.: Cloture voting: n/4-resilient distributed consensus in t+1 rounds. Mathematical System Theory **26**(1), 3–19 (1993)

[9] Bondy, J., Murty, U.: Graph Theory with Applications. American Elsevier (1976)

[10] Chandra, T., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. Journal of the ACM **43**(4), 685–722 (1996)

[11] Chandra, T., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM **43**(2), 225–267 (1996)

[12] Chandy, M., Lamport, L.: Distributed snapshots: Determining global states in distributed systems. ACM Trans. on Computer Science **3**(1), 63–75 (1985)

[13] Chang, E., Roberts, R.: An improved algorithm for decentralized extrema finding in circular configurations of processes. Communications of the ACM **22**(5), 281–283 (1979)

[14] Charron-Bost, B.: Concerning the size of logical clocks in distributed systems. Information Processing Letters **39**, 11–16 (1991)

[15] Cheung, T.: Graph traversal techniques and the maximum flow problem in distributed computation. IEEE Trans. in Software Engineering **SE-9**(4), 504–512 (1983)

[16] Dolev, D., Dwork, C., Stockmeyer, L.: On the minimal synchronism needed for distributed consensus. Journal of the ACM **34**(1), 77–97 (1987)

[17] Dolev, D., Reischuck, R., Strong, R.: Early stopping on bizantine agreement. Journal of the ACM **37**(4), 720–741 (1990)

[18] Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. Journal of the ACM **35**(2), 288–323 (1988)

[19] Fidge, C.: Logical time in distributed computing systems. IEEE Computer pp. 28–33 (1991)

[20] Fischer, M., Lynch, N.: A lower bound for the time to assure interactive consistency. Information Processing Letters (71), 183–186 (1982)

[21] Fischer, M., Lynch, N., Paterson, M.: Impossibility of distributed consensus with one faulty process. Journal of the ACM **32**(2), 374–482 (1985)

[22] Fischer, M., Michael, A.: Sacrifying serializability to attain high availability of data in an unreliable network. In: ACM Symposium on Principles of Database Systems, pp. 70–75 (1982)

[23] Gallager, R., Humblet, P., Spira, P.: A distributed algorithm for minimum weight spanning trees. ACM Trans. on Programming Languages and Systems. **5**(1), 66–77 (1983)

[24] Garey, M., Johnson, D.: Computers and Intractability. W. H. Freeman and Co. (1979)

[25] Garg, V.: Elements of Distributed Computing. John Wiley and Sons (2002)

[26] Hadzilacos, V., Toueg, S.: Fault-tolerant broadcasts and related problems. In: S. Mullender (ed.) Distributed Systems, 2 edn., chap. 5, pp. 97–145. ACM Press (1993)

[27] Herlihy, M.: Impossibility and universality results for wait-free synchronization. In: 7th ACM Simposium on Principles of Distributed Computing, pp. 276–290. ACM, New York (1988)

[28] Hirschberg, S., Sinclair, J.: Decentralized extrema-findig in circular configuration of processes. Communications of the ACM **23**, 627–628 (1980)

[29] Holzmann, G.: Design and Validation of Computer Protocols. Prentice Hall (1991)

[30] Kruskal, J.J.: On the shortest spanning subtree of a graph and the traveling salesman problem. Proc. AMS **7**(1), 48–50 (1956)

[31] Lamport, L.: Time, clocks and the ordering of events in a distributed system. Communications of the ACM **21**(7), 558–564 (1978)

[32] Lamport, L., Lynch, N.: Distributed computing: Models and methods. In: J. Van Leeuwen (ed.) Handbook of Theoretical Computer Science, vol. B, chap. 18, pp. 1158–1199. Elsevier and MIT Press (1990)

[33] Lamport, L., Shostak, R., Pease, M.: The bizantine generals problem. ACM Trans. On Programming Languages and Systems **4**(3), 382–401 (1982)

[34] Lavault, C.: Evaluation des Algorithmes Distribués. Hermes, Paris, France (1995)

[35] LeLann, G.: Distributed systems towards a formal approach. Information Processing (77), 155–160 (1977). Elsevier Science

[36] Lynch, N.: Distributed Algorithms. Morgan Kaufman Pub. (1996)

[37] Mattern, F.: Virtual time and global states of distributed systems. In: Q. Cosnard, M. Raynal (eds.) Proceeding of the Parallel and Distributed Algorithms Conference, pp. 215–226 (1988)

[38] Mattern, F.: Efficient algorithms for distributed snapshots and global virtual time approximation. Jornal of Parallel and Distributed Computing **18**, 423–434 (1993)

[39] Moses, Y., Rajsbaum, S.: The unified structure of consensus: A layered approach analysis. In: 17th ACM Simposium on Principles of Distributed Computing, pp. 123–132. ACM, Puerto Vallarta, México (1998)

[40] Pease, L., Shostak, R., Lamport, L.: Reaching agreement in presence of faults. Journal of the ACM **27**(2), 228–234 (1980)

[41] Peleg, D.: Time optimal leader election in general networks. Journal of Parallel and Distributed Computing (8), 96–99 (1990)

[42] Peleg, D.: Distributed Computing A Locality-Sensitive Approach. SIAM (2000)

[43] Prim, R.: Shortest connection networks and some generalizations. Bell Systems Technical Journal (36), 1389–1401 (1957)

[44] Raynal, M.: Consensus in synchronous systems: A concise guided tour. In: Proceedings of the 2002 Pacific Rim Symposium on Dependable Computing (PRDC'02). IEEE, Computer Society (2002)

[45] Raynal, M., Schiper, A., Toueg, S.: The causal ordering abstraction and a simple way to implement it. Information Processing Letters **39**(6), 343–350 (1991)

[46] Raynal, M., Singhal, M.: Logical time: Capturing causality in distributed systems. IEEE Computer **29**(2), 49–56 (1996)

[47] Schneider, B.F.: What good are models and what models are good? In: S. Mullender (ed.) Distributed Systems, 2 edn., chap. 2, pp. 17–26. ACM (1993)

[48] Segall, A.: Distributed network protocols. IEEE Trans. on Information Theory **29**(1), 23–35 (1983)

[49] Sipser, M.: Introduction to the Theory of Computation. International Thomson Publishing (1996)

[50] Tel, G.: Introduction to Distributed Algorithms. Cambridge University Press (1994)

[51] Verjus, J.: Introduction aux systèmes répartis. In: R. Balter et. al. (ed.) Construction des Systèmes d'exploitation Répartis. INRIA, Paris, France (1991)